Contents

Want to learn more about the basics?

- Read the ten-minute guide to get started with using RethinkDB.
- Read the FAQ for programmers new to distributed systems.
- Jump into the cookbook and see dozens of examples of common RethinkDB queries.

Sharding and replication

How does RethinkDB partition data into shards?

RethinkDB uses a range sharding algorithm parameterized on the table's primary key to partition the data. When the user states they want a given table to use a certain number of shards, the system examines the statistics for the table and finds the optimal set of split points to break up the table evenly. All sharding is currently done based on the table's primary key, and cannot be done based on any other attribute (in RethinkDB the primary key and the shard key are effectively the same thing).

For example, if a given table contains a thousand JSON documents whose primary keys are uniformly distributed, alphabetical, upper-case strings and the user states they want two shards, RethinkDB will likely pick split point 'M' to partition the table. Every document with a primary key less than or equal to 'M' will go into the first shard, and every document with a primary key greater than 'M' will go into the second shard. The split point will be picked such that each shard contains close to five hundred keys, and the shards will automatically be distributed across the cluster.

Even if the primary keys contain unevenly distributed data (such as human last names, where some keys are likely to occur much more frequently than others),

the system will still pick a correct split point to ensure that each shard has a roughly similar number of documents (there are many more Smiths in the phone book than Akhmechets).

In advanced situations, the user can specify the split points manually, and the system will partition the table into shards according to the user's request. When the split points for the table change, RethinkDB uses a bit of math to figure out the optimal way to copy as little data as possible. For example, if the split point for a table used to be 'M', and is later changed to 'O', RethinkDB will figure out that it only needs to copy the data between keys 'M' and 'O' from one shard to another.

Internally this approach is more difficult to implement than the more commonly used consistent hashing, but it has significant advantages because it allows for an efficient implementation of range queries.

What governs the location of shards and replicas in the cluster?

RethinkDB's clustering layer is based on three concepts: goals, directory, and blueprints.

The *goals* are what the user (or an automated script) specifies to the clustering system via the web UI or the command line tools. For example, the user might specify that a table is to be broken up into five shards based on specific split points, that each shard is to be replicated five times within the cluster, and that at least two of those replicas are to be located in a European datacenter.

The *directory* is the current physical state of the cluster— how many machines are accessible, what data is currently stored on each machine, etc. The data structures that keep track of the directory are automatically updated when the cluster changes. For example, if a machine dies due to power failure, the directory is updated to represent this change.

The *blueprint* is a data structure automatically generated by RethinkDB to satisfy the user's goals based on the constraints imposed by the physical reality. It contains information about which data each machine in the cluster is to be responsible for. For example, if the user updates the goals to state that the European datacenter is to contain three instead of two replicas, the system will pick an extra machine in the European datacenter to be responsible for an additional replica, update the blueprint with that information, and propagate it to the rest of the cluster.

Each machine in the cluster monitors the part of the blueprint it's responsible for, and constantly tries to match the directory to the blueprint. For example, if the blueprint states that machine M is responsible for a particular shard, but the machine M doesn't have this shard, the machine will automatically react to

copy the data from one of the available replicas, and will take appropriate steps to keep it up to date.

RethinkDB uses a set of heuristics to attempt to satisfy the user's goals in an optimal way. It will try to copy data for new replicas from the closest available machine, evenly distribute replicas of the data across the cluster, try to place burden on machines that aren't dealing with heavy load, etc.

In advanced situations where these heuristics aren't sufficient, the user can include specific hardware requirements into the goals. For example, the user can specify that a given replica is to be located on a specific machine in the cluster, and RethinkDB will generate a blueprint to satisfy these requirements.

How does multi-datacenter support work?

To take advantage of availability zone support, you first have to group machines into datacenters. Once you tell RethinkDB which datacenters machines in the cluster belong to, you can define replication and acknowledgement settings on per-datacenter basis.

For example, you may specify that a table should have at least three replicas in the US_East datacenter, and at least five replicas in US_West. You also might specify that before a write is acknowledged to the client, at least two replicas in each datacenter must acknowledge the write.

Once you set replication and acknowledgement goals on a per-datacenter basis, the system takes care of data replication and enforcing write acknowledgement counts automatically.

RethinkDB uses the same protocol for communication within a datacenter as it does across datacenters. Because the architecture is immediately consistent and does not require quorums on individual document reads and writes, the latency issues commonly associated with cross-datacenter quorums on Dynamo-style systems do not arise in RethinkDB.

Does RethinkDB automatically reshard the database without the user's request?

The short answer is no. The longer answer is that the clustering system is designed with three main principles in mind:

- Common operations such as scaling up and down, rebalancing shards, and increasing/decreasing replication count should easily be performed in a click of a button.
- In cases where it matters, the system should give administrators fine-tuned control, such as pinning specific masters and replicas to specific machines in the cluster.

- Information about the cluster and all operations on the cluster should be programatically accessible.

We felt that performing automatic maintenance operations on the cluster (such as adding shards) is a higher-level component, and that it's crucial to have a really good implementation of the lower-level components done first. As a result, the clustering system is organized into three layers:

- The first layer implements the distributed infrastructure, placing copies of data on specific machines, routing queries, etc.
- The second layer builds on the first and implements various automation mechanisms (e.g. automatically determining how to split shards, where to place copies of the data, automatically picking optimal masters, etc.) This is the layer that compiles goals specified by the user into blueprints.
- The third layers builds on the previous two and provides the user with command line and web-based tools to control the cluster.

Invoking this functionality automatically without the user's request is the next layer in this hierarchy. Currently the user can control the system via the web UI, manually via the command line, or by writing scripts to call the command line tools to perform server automation.

We're exploring best practices to determine whether it's possible to build a really good general purpose automation layer that controls the cluster by automatically enforcing user-specified rules (such as resharding the system when the shard balance drops below a certain threshold).

CAP theorem

Is RethinkDB immediately or eventually consistent?

In RethinkDB data always remains immediately consistent and conflict-free, and a read that follows a write is always guaranteed to see the write. This is accomplished by always assigning every shard to a single authoritative master. All reads and writes to any key in a given shard always get routed to its respective master where they're ordered and evaluated.

RethinkDB supports both up-to-date and out-of-date reads. By default, all read queries are executed up-to-date, which means that every read operation for a given shard is routed to the master for that shard and executed in order with other operations on the shard. In this default mode, the client always sees the latest, consistent, artifact-free view of the data.

The programmer can also mark a read query to be ok with out-of-date data. In this mode, the query isn't necessarily routed to the shards master, but is likely

to be routed to its closest replica. Out-of-date queries are likely to have lower latency and have stronger availability guarantees, but don't necessarily return the latest version of the data to the client.

What CAP theorem tradeoffs are made in RethinkDB?

The essential tradeoff exposed by the CAP theorem is this: in case of network partitioning, does the system maintain availability or data consistency? (Jumping ahead, RethinkDB chooses to maintain data consistency).

Dynamo-based systems such as Cassandra and Riak choose to maintain stronger availability. In these systems if there is a network partition, the clients can write to the same row on both sides of the netsplit. In exchange for the write availability, applications built on top of these systems must deal with various complexities such as clock skew, conflict resolution code, conflict repair operations, performance issues for highly contested keys, and latency issues associated with quorums.

Authoritative systems, such as RethinkDB and MongoDB choose to maintain data consistency. Building applications on top of authoritative-master systems is much simpler because all of the issues associated with data inconsistency do not arise. In exchange, these applications will occasionally experience availability issues.

In RethinkDB, if there is a network partition, the behavior of the system from any given client's perspective depends on which side of the netsplit that client is on. If the client is on the same side of the netsplit as the master for the shard the client is trying to reach, it will continue operating without any problems. If the client is on the opposite side of the netsplit from the master for the shard the client is trying to reach, the client's up-to-date queries and write queries will encounter a failure of availability. For example, if the client is running an up-to-date range query that spans multiple shards, the masters for all shards must be on the same side of the netsplit as the client, or the client will encounter a failure of availability.

If the programmer marks a read query to be ok with out-of-date data, RethinkDB will route the query to the closest available replica instead of routing it to the master. In this case the client will see the data as long as there are replicas of the data on the its side of the netsplit. However, in this case the data has the risk of being out of date. This is usually ok for reports, analytics, cached data, or any scenario in general where having the absolute latest information isn't imperative.

How is cluster configuration propagated?

Updating the state of a the cluster is a surprisingly difficult problem in distributed systems. At any given point different (and potentially) conflicting

configurations can be selected on different sides of a netsplit, different configurations can reach different nodes in the cluster at unpredictable times, etc.

RethinkDB uses semilattices to store and propagate cluster configuration (including goals and blueprints). Various parts of the semilattices are versioned via internal vector clocks. This architecture turns out to have sufficient mathematical properties to address all the issues mentioned above (this result has been known in distributed systems research for quite a while).

Indexing

How does RethinkDB index data?

When the user creates a table, they have the option of specifying the attribute that will serve as the primary key (if the primary key attribute isn't specified, it defaults to 'id'). When the user inserts a document into the table, if the document contains the primary key attribute, its value is used to index the document. Otherwise, a random unique ID is generated for the index automatically.

The primary key of each document is used by RethinkDB to place the document into an appropriate shard, and index it within that shard using a B-Tree data structure. Querying documents by primary key is extremely efficient, because the query can immediately be routed to the right shard and the document can be looked up in the B-Tree.

Does RethinkDB support secondary and compound indexes?

RethinkDB supports both secondary and compound indexes, as well as indexes that compute arbitrary expressions. You can see examples of how to use the secondary index API here.

Availability and failover

What happens when a machine becomes unreachable?

The first thing that happens when a node in the cluster becomes unreachable is that an issue is raised by the RethinkDB cluster. It is immediately and prominently displayed in the WebUI, and is accessible via the command line administration tools.

While the machine remains unreachable, table availability is determined by the following three cases:

- If the machine is acting as a master for any shards, the corresponding tables lose read and write availability (out-of-date reads remain possible as long as there are other replicas of the shards).
- If the machine is acting as a replica for a given table and there aren't enough replicas in the cluster to respect the user's write acknowledgement settings, the table loses write availability (but maintains read availability).
- If the machine isn't acting as a master for any tables, and there are enough replicas to respect the user's write acknowledgement settings, the system continues operating as normal.

There are two possible solutions to this issue. The first option is to simply wait for the machine to become reachable again. If the machine comes back up RethinkDB automatically performs the following actions without any user interaction: replicas on the machine are brought up to date with the latest changes, masters on the machine become active again, the cluster clears the reachability issue from web and command line tools, availability is restored, and the cluster continues operating as normal.

The second option is to declare the machine dead. If the machine is declared dead, it is absolved of all responsibilities, and one of the replicas is automatically elected to act as a new master. After the machine is declared dead, availability is quickly restored and the cluster begins operating normally. (If the dead machine comes back up, it is rejected by the cluster as a zombie).

Currently, RethinkDB does not automatically declare machines dead after a timeout. This can be done by the user, either in the web UI, manually via the command line, or by scripting the command line tools.

What are availability and performance impacts of sharding and replication?

RethinkDB maintains availability if the user increases or decreases the number of replicas in the cluster. In most cases, the replication process should not have a strong performance impact on the real-time system.

RethinkDB may or may not maintain availability if the user modifies the number of shards. In many cases availability will be maintained, but currently it cannot be guaranteed. We're exploring different solutions to remove this limitation.

Query execution

How does RethinkDB execute queries?

When a node in the cluster receives a query from the client, it evaluates the query in the following way.

First, the query is transformed into an execution plan that consists of a stack of internal logical operations. The operation stack fully describes the query in a data structure useful for efficient execution. The bottom-most node of the stack usually deals with data access— it can be a lookup of a single document, a short range scan bounded by an index, or even a full table scan. Nodes closer to the top usually perform transformations on the data – mapping the values, running reductions, grouping, etc. Nodes can be as simple as projections (i.e. returning a subset of the document), or as complex as entire stacks of stacks in case of subqueries.

Each node in the stack has a number of methods defined on it. The three most important methods define how to execute a subset of the query on each machine in the cluster, how to combine the data from multiple machines into a unified resultset, and how to stream data to the nodes further up in small chunks.

As the client attempts to stream data from the server, these stacks are transported to every relevant machine in the cluster, and each machine begins evaluating the topmost node in the stack, in parallel with other machines. On each machine, the topmost node in the stack grabs the first chunk of the data from the node below it, and applies its share of transformations to it. This process proceeds recursively until enough data is collected to send the first chunk to the client. The data from each machine is combined into a single resultset, and forwarded to the client. This process continues as the client requests more data from the server.

The two most important aspects of the execution engine is that every query is completely parallelized across the cluster, and that queries are evaluated lazily. For instance, if the client requests only one document, RethinkDB will try to do just enough work to return this document, and will not process every shard in its entirety. This allows for large, complicated queries to execute in a very efficient way.

The full query execution process is fairly complex and nuanced. For example, some operations cannot be parallelized, some queries cannot be executed lazily (which has implications on runtime and RAM usage), and implementations of some operations could be significantly improved. We will be adding tools to help visualize and understand query execution in a user-friendly way, but at the moment the best way to learn more about it is to ask us or to look at the code.

How does the atomicity model work?

Write atomicity is supported on a per-document basis – updates to a single JSON document are guaranteed to be atomic. RethinkDB is different from other NoSQL systems in that atomic document updates aren't limited to a small subset of possible operations – any combination of operations that can be performed on a single document is guaranteed to update the document atomically. For example, the user may wish to update the value of attribute A to

a sum of the values of attributes B and C, increment the value of attribute D by a fixed number, and append an element to an array in attribute E. All of these operations can be applied to the document atomically in a single update operation.

However, RethinkDB does come with some restrictions regarding which operations can be performed atomically. Operations that cannot be proven deterministic cannot update the document in an atomic way. Currently, values obtained by executing JavaScript code, random values, and values obtained as a result of a subquery (e.g. incrementing the value of an attribute by the value of an attribute in a different document) cannot be performed atomically. If the user runs a query that cannot be executed atomically, by default RethinkDB will throw an error. The user can choose to set the flag on the update operation in the client driver to execute the query in a non-atomic way.

In addition, like most NoSQL systems, RethinkDB does not support updating multiple documents atomically.

How are concurrent queries handled?

To efficiently perform concurrent query execution RethinkDB implements block-level multiversion concurrency control (MVCC). Whenever a write operation occurs while there is an ongoing read, RethinkDB takes a snapshot of the B-Tree for each relevant shard and temporarily maintains different versions of the blocks in order to execute read and write operations concurrently. From the perspective of the applications written on top of RethinkDB, the system is essentially lock-free— you can run an hour-long analytics query on a live system without blocking any real-time reads or writes.

RethinkDB does take exclusive block-level locks in case multiple writes are performed on documents that are close together in the B-Tree. If the contested block is cached in memory, these locks are extremely short-lived. If the blocks need to be loaded from disk, these locks take longer. Typically this does not present performance problems because the top levels of the B-Tree are all cached along with the frequently used blocks, so in most cases writes can be performed essentially lock-free.

Data storage

How is data stored on disk?

The data is organized into B-Trees, and stored on disk using a log-structured storage engine built specifically for RethinkDB and inspired by the architecture of BTRFS. The storage engine has a number of benefits over other available options, including an incremental, fully concurrent garbage compactor, low CPU

overhead and very efficient multicore operation, a number of SSD optimizations, instantaneous recovery after power failure, full data consistency in case of failures, and support for multiversion concurrency control.

The storage engine is used in conjunction with a custom, B-Tree-aware caching engine which allows file sizes many orders of magnitude greater than the amount of available memory. As a result, neither the data nor the index have to fit in RAM — RethinkDB can operate on terabytes of data with only a few megabytes of cache space.

Which file systems are supported?

RethinkDB supports most commonly used file systems. However, like most database systems, RethinkDB uses the DIRECT_IO mechanism to access disk efficiently when it is available. Some filesystems cannot be accessed using the DIRECT_IO mechanism, but they can still be used to run RethinkDB with degraded performance. Some encrypted and journaled filesystems do not allow direct disk access, RethinkDB might fail to start if you try to place data files onto such filesystems.

There is also a known issue with running RethinkDB (and other database systems) on ReiserFS in journaled data mode — it appears to cause a crash instead of reporting an error. See https://github.com/rethinkdb/rethinkdb/issues/20.

How can I perform a backup of my cluster?

RethinkDB ships with simple tools to perform a hot backup of a running cluster. See the backup instructions for more details.

Interested in a less biased overview? Browse through the technical comparison tables between RethinkDB and MongoDB.

Many people have been asking how RethinkDB differs from MongoDB and other NoSQL systems. Our first attempt to address these questions is a high level technical overview comparing RethinkDB with MongoDB. However, the overview is meant to be impartial, and it omits some of the more interesting bits about what makes RethinkDB special (such as our irrational love for Dota). In this post we wanted to describe the product from a more personal perspective.

The best of both worlds

First generation NoSQL products fall into roughly two categories — developer-oriented and operations-oriented.

Developer-oriented products include MongoDB and CouchDB. They typically pay close attention to ease of use, have rich document structure, and offer flexible querying capabilities. However, when compared with their operations-oriented counterparts, they are more difficult to deploy to sharded environments and to scale to significant load.

Operations-oriented systems include Cassandra and Riak. These products are designed for highly available deployments and high scale. Unlike developer-oriented products, in their current form operations-oriented products typically have less powerful querying capabilities, and tend to focus less on ease of use.

With the benefit of hindsight and three years of engineering effort, **RethinkDB aims to embody ideas from each philosophy** to offer the best of both worlds. It is designed to be very easy to use, has a rich data model, and supports extremely flexible querying capabilities. A cluster of RethinkDB nodes can be sharded in only a couple of clicks (really, it's that easy, check out the screencast). Browse through the ReQL API reference to see how RethinkDB attempts to accomplish these goals.

Will Larson said it best on his blog, Irrational Exuberance:

> RethinkDB has captured some of the best ideas from Cassandra and CouchDB. - Reflection on RethinkDB

Raising the bar

We're also working on extending what's possible with NoSQL systems. RethinkDB adds a modern query language, a massively parallel distributed infrastructure, support for distributed joins and subqueries, and has administration tools that are both simple **and** beautiful. Here are some of the reasons our early adopters have been especially excited about RethinkDB.

Modern query language

> @rethinkdb has the best query language of all new databases that I've seen. - @rauchg

- ReQL is a data-driven, abstract, polymorphic query language. It's easy to learn and extremely flexible.
- Each host language driver (currently Python, Ruby, and JavaScript) implements a custom, tightly integrated domain-specific language. It's both pragmatic and fun to use.
- Unlike most NoSQL systems, RethinkDB supports server-side subqueries and distributed join operations. This eliminates complex client-side code and multiple network roundtrips to the database server.

- ReQL is not based on string parsing, so the risk of injection attacks is greatly minimized.

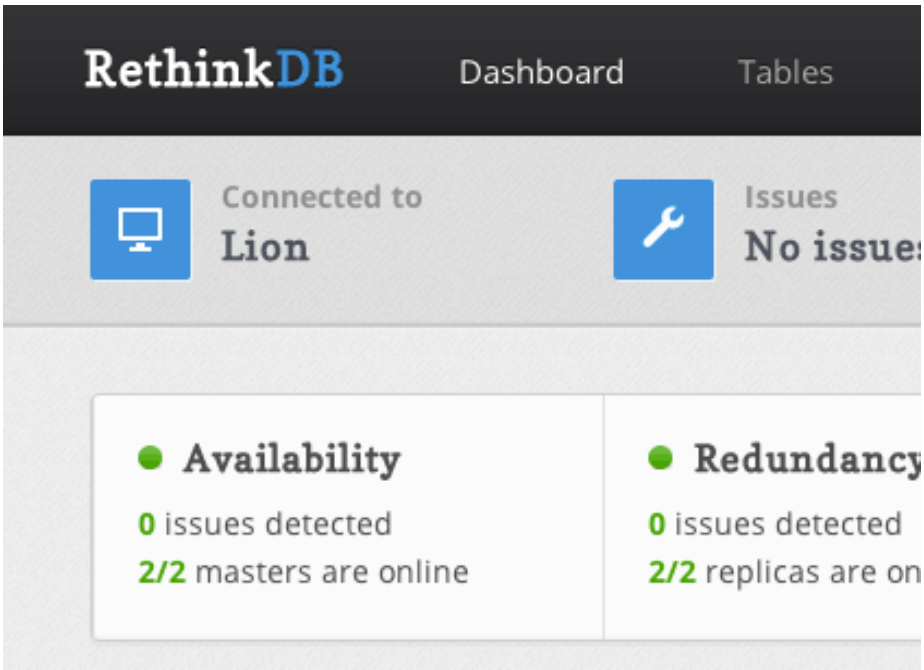
That's not to say that ReQL is complete. There are more features and operations that we're constantly adding to the query language.

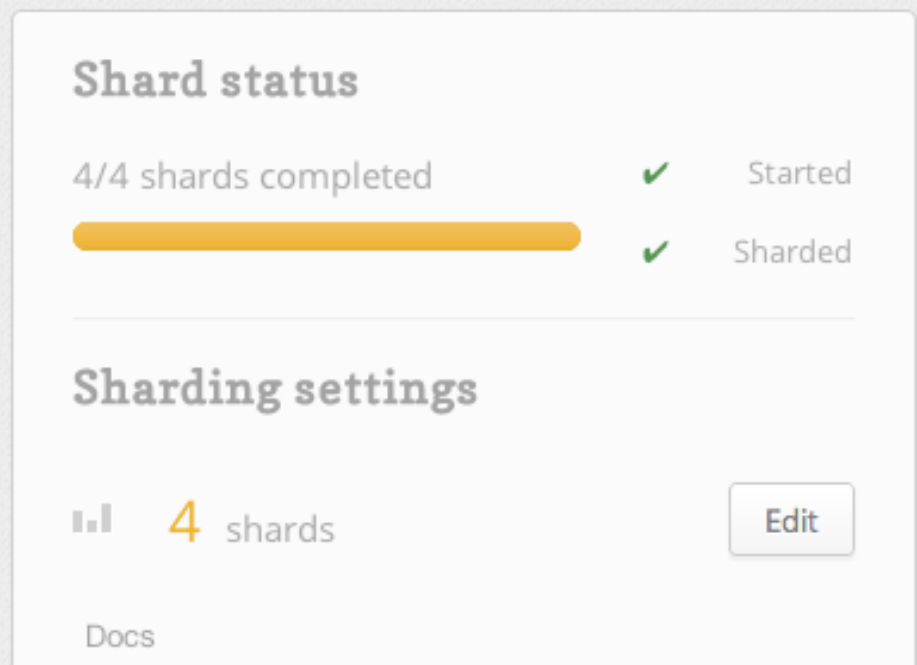
Administration: simple and beautiful

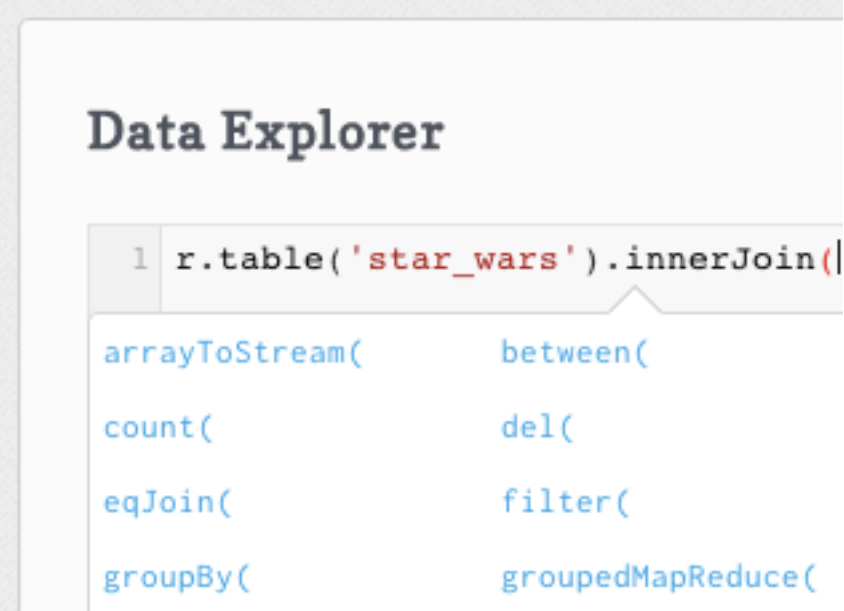
> WOW. @RethinkDB's admin interface is incredible. And installing on OSX is a cinch. Check it out people! - @mjackson

- Sharding, replication, and multi-datacenter setup can be done in a few clicks. Watch the screencast to see how easy it is to setup and operate a cluster.
- All cluster operations are scriptable via CLI tools.
- The Built-in data explorer offers online documentation and query language suggestions.

Many products are easy to use, but we think RethinkDB is **beautiful**. See the screenshots or the screencast video and be the judge yourself.




Shard status
4/4 shards completed
Started
Sharded
Sharding settings
4 shards
Edit
Docs


Data Explorer
1 r.table('star_wars').innerJoin(
arrayToStream(
between(
count(
del(
eqJoin(
filter(
groupBy(
groupedMapReduce(

Massively parallelized distributed infrastructure

> wow joins across shards, this is distributed data dream come true - @kapso

- All queries including joins, aggregation, and subqueries are automatically compiled to distributed programs and executed across the cluster without any effort from the user.
- Data intensive operations are automatically compiled to map/reduce jobs that take advantage of the distributed architecture.
- Cluster protocol is peer-to-peer and doesn't require coordinator nodes. This makes clusters extremely easy to set up and operate.

Of course we're always improving performance and are always working to remove more performance and scalability bottlenecks.

Robust implementation

> RethinkDB looks like a MongoDB done right: MVCC, non blocking writes, durability by default, v8, incremental vacuum, easy sharding, … - @herodiade

- A lock-free architecture and a multi-version concurrency control system allow different workloads to coexist in one cluster.
- A custom, B-Tree aware buffer cache efficiently operates on datasets much larger than the amount of available RAM.
- An asynchronous, event-driven architecture based on highly optimized coroutine code scales across multiple cores and processors, network cards, and storage systems.
- A custom log-structured storage engine with a concurrent, incremental on-disk garbage compactor takes advantage of different types of storage hardware.

Limitations

RethinkDB works really well for many projects, but is not a good choice for every task. Learn more about when RethinkDB isn't a good choice for your application.

What's next

Form your own opinion! Take RethinkDB for a spin and don't forget to give us your feedback so we can work out the quirks faster.

Dive into RethinkDB with the thirty-second quickstart. →

Interested in a more personal perspective? Read our take on what makes RethinkDB different.

This document is our attempt at an unbiased comparison between RethinkDB and MongoDB (for a more partisan view, take a look at the biased comparison written by @coffeemug). We tried to be spartan with our commentary to allow the reader to form their own opinion. Whenever possible, we provide links to the original documentation for further details.

The document is organized by four main categories:

- Development
- Administrative operations
- Scaling
- Database architecture

Development

	RethinkDB	MongoDB
Platforms	Linux, OS X	Linux, Windows, OS X, Solaris
Data model	JSON documents	BSON documents
Data access	Unified chainable dynamic query language	Dynamic rich query language
JavaScript integration	V8 engine	Spidermonkey/V8 engine
Access languages	• Protocol Buffers protocol • 3 official (JavaScript, Python, Ruby) and community supported (.NET / C#, PHP, Go, and many more) libraries	• BSON protocol • 13 official and many community supported libraries

	RethinkDB	MongoDB
Indexing	Multiple types of indexes (primary key, compound, secondary, arbitrarily computed)	Multiple types of indexes (unique, compound, secondary, sparse, geospatial)
Cloud deployment	AWS, dotCloud	MongoDB is available on many cloud platforms

Platforms

MongoDB has binary distributions for:

- Linux 32-bit and 64-bit
- Windows 32-bit and 64-bit
- OS X 64-bit
- Solaris 64-bit

Note: MongoDB 32-bit builds are limited to around 2GB of data.

RethinkDB has binary packages available for:

- Ubuntu 10.04 and higher 32-bit/64-bit
- OS X 64-bit (>= 10.7)
- CentOS 6, 32-bit/64-bit

Data model

MongoDB uses BSON for storing data. The BSON protocol, a custom extension of JSON, supports additional data types (e.g. ObjectId, timestamp, datetime, etc.) that are not part of the JSON specification.

RethinkDB stores JSON documents with a binary on disk serialization. The data types supported by JSON and implicitly RethinkDB are: number (double precision floating-point), string, boolean, array, object, null.

Query language

Accessing data in MongoDB can be done using:

- CRUD operations using BSON objects for inserting, bulk inserting, filtering, and updating documents
- aggregations using map/reduce or the aggregation framework (starting with ver.2.2)

RethinkDB provides a unified chainable query language supporting:

- CRUD operations
- aggregations (including map/reduce and the more advanced grouped_map_reduce)
- JOINs
- full sub-queries

Javascript integration

MongoDB's query language allows JavaScript queries using the $where clause. MongoDB MapReduce functions are defined in JavaScript. MongoDB uses a single-threaded JavaScript engine for executing MapReduce operations.

RethinkDB allows embedding JavaScript expressions anywhere as part of the query language. RethinkDB uses a pool of out-of-process V8 execution engines for isolation.

Access languages

MongoDB has 13 official and many community supported libraries. MongoDB's wire protocol is TCP based and uses BSON.

RethinkDB provides official libraries for Javascript/Node.js, Python, Ruby. Community contributed projects are listed in the GitHub wiki. RethinkDB uses Protocol Buffers over TCP for client-server communications.

Indexing

MongoDB supports unique, compound, secondary, sparse, and geospatial indexes. All MongoDB indexes use a B-tree data structure. Every MongoDB query, including update operations, uses one and only one index.

RethinkDB supports primary key, compound, secondary, and arbitrarily computed indexes stored as B-trees. Every RethinkDB query, including update operations, uses one and only one index.

Cloud deployment

MongoDB can be manually deployed on the majority of cloud platforms (AWS, Joyent, Rackspace, etc.). MongoDB hosting is also available from a wide range of providers either as a dedicated service (MongoHQ, MongoLab, etc.) or as an add-on on Platform-as-a-Service solutions (dotCloud, Heroku, etc.).

RethinkDB can be manually deployed on cloud platforms like AWS or as a custom service on dotCloud using rethinkdb-dotcloud.

Administration

	RethinkDB	MongoDB
CLI tools	Admin CLI	JavaScript interactive she
UI tools	Web-based admin UI	Simple HTTP interface
Failover	1-click replication with customizable per-table acknowledgements	Replica-sets with auto pri
Backup	`rethinkdb-dump`	`mongodump` or snapshottin

CLI Tools

MongoDB provides a JavaScript interactive shell that can be used for inspecting data, testing queries, creating indexes, maintenance scripts, and other administrative functions.

RethinkDB has an administration CLI that can be attached to any node in the cluster and provides fine grained administrative control of the cluster resources. The command-line client offers integrated help and auto-completion.

UI tools

MongoDB has a simple HTTP interface that displays read-only information about a server. 10gen offers a hosted monitoring solution called MMS.

RethinkDB has a web-based admin UI accessible on every node of a cluster that provides high level and guided support for operating the cluster. The admin UI also includes the Data Explorer for experimenting, tuning, and manipulating data.

Failover

The 3 main components of a MongoDB cluster (`mongos`, `mongod`, and the 3 config servers) are highly available. For servers storing data, MongoDB allows setting up replica sets with automatic primary election.

For failover, RethinkDB supports setting up 1-click replication with custom per-table acknowledgements. RethinkDB doesn't yet support primary auto re-elections.

Backup

MongoDB provides different mechanisms for backing up data:

- the `mongodump` utility can perform a live backup of data.
- disk/block level snapshots can be used to backup a MongoDB instance when journaling is enabled. When journaling is disabled, snapshots are possible after flushing all writes to disk and locking the database.

RethinkDB supports hot backup on a live cluster via `dump` and `restore` commands.

Scaling

	RethinkDB
Sharding (supervised/guided/advised/trained)	Guided range-based sharding Automatic range-based sharding
Replication	Sync and async replication
Multi datacenter	Multiple DC support with per-datacenter replication and writ
MapReduce Executing ReQL or Javascript operations	Multiple MapReduce functions Javascript-based MapReduce
Performance	No published results
Concurrency Asynchronous block-level MVCC Read-write locks	Event-based and coroutines Threading

Sharding

MongoDB supports automatic range-based sharding using a shard key. A sharded MongoDB cluster requires 3 config servers and 1 or more `mongos` instances.

RethinkDB supports 1-click sharding from the admin UI. Sharding can be configured also from the CLI which also supports manual assignments of shards to specific machines. Rebalancing the shards can be done through the admin UI.

Replication

MongoDB replication is based on replica sets which use a master-slave log-shipping asynchronous approach. MongoDB replica sets are configured using the interactive shell.

RethinkDB allows setting up replication using the 1-click admin web UI or from the CLI. RethinkDB supports both sync and async replication by specifying the per-table number of write acknowledgements. RethinkDB replication is based on B-Tree diff algorithms and doesn't require log-shipping.

Multi Datacenter Support

MongoDB can be configured to run in multiple datacenters via different mechanisms:

- assigning priorities to members of replica-sets
- support for nearby replication
- tagging (version 2.0+)

RethinkDB supports grouping machines into datacenters with per datacenter replication and write acknowledgement settings through either the admin web UI or CLI. RethinkDB immediate consistency based reads and writes do not require a special protocol for multi DC replication.

MapReduce

MongoDB supports running JavaScript-based MapReduce tasks through the `mapReduce` command or from the interactive shell. MongoDB MapReduce allows pre-filtering and ordering the data for the map phase. It also allows storing the results in a new collection. The various phases of the MongoDB MapReduce implementation make uses of different locks and the JavaScript code is executed in a single thread.

RethinkDB supports multiple MapReduce-like operations: `map`, `reduce`, `groupBy`, `groupedMapReduce` that allow processing data using both ReQL and JavaScript. RethinkDB operations are transparently and fully distributed. None of these operations require any locks. RethinkDB MapReduce functions can be part of chained queries, by preceding, following, or being sub-queries of other operations.

Neither MongoDB nor RethinkDB support incremental MapReduce by default.

Performance

MongoDB doesn't publish any official performance numbers.

RethinkDB's performance has degraded significantly after the addition of the clustering layer, but we hope we'll be able to restore it over the next several releases.

Concurrency

MongoDB uses locks at various levels for ensuring data consistency. In MongoDB, v2.2 writes and MapReduce require write locks at the database level. MongoDB uses threads for handling client connections.

RethinkDB implements block-level multiversion concurrency control . In case multiple writes are performed on documents that are close together in the B-Tree, RethinkDB does take exclusive block-level locks, but reads can still proceed.

Architecture

	RethinkDB
Consistency model	Immediate/strong consistency with support for o
Atomicity	Document level
Durability	Durable
Storage engine with incremental, fully concurrent garbage compactor	Log-structured B-tree serialization Memory mapped files
Query distribution engine	Transparent routing, distributed and parallelized
Caching engine	Custom per-table configurable B-tree aware cachi

Consistency model

MongoDB has a strong consistency model where each document has a master server at a given point in time. Until recently MongoDB client libraries had by default a fire-and-forget behavior.

In RethinkDB data always remains immediately consistent and conflict-free, and a read that follows a write is always guaranteed to see the write. This is accomplished by always assigning every shard to a single authoritative master. All reads and writes to any key in a given shard always get routed to its respective master where they're ordered and evaluated.

Both MongoDB and RethinkDB allow out-of-date reads.

Note: While MongoDB and RethinkDB docs refer to their consistency models as strong and respectively immediate, we think the behavior of the two databases is equivalent.

Atomicity

MongoDB supports atomic document-level updates that add, remove, or set attributes to constant values.

RethinkDB supports advanced atomic document-level updates that can add, remove, or modify attributes with evaluated expressions that can also read current attribute values. There are 2 cases where atomic updates cannot be guaranteed: 1) updates reading data from other documents; 2) updates involving JavaScript evaluations.

Durability

MongoDB supports write-ahead journaling of operations to facilitate fast crash recovery and durability in the storage engine. MongoDB offers a recovery procedure when journaling is not enabled.

RethinkDB comes with strict write durability out of the box inspired by BTRFS inline journal and is identical to traditional database systems in this respect. No write is ever acknowledged until it's safely committed to disk.

Storage engine

MongoDB uses memory mapped files where the OS is in control of flushing writes and paging data in and out.

In RethinkDB, data is organized into B-Trees and stored on disk using a log-structured storage engine built specifically for RethinkDB and inspired by the

architecture of BTRFS. RethinkDB's engine includes an incremental, fully concurrent garbage compactor and offers full data consistency in case of failures.

Query distribution engine

MongoDB clients connect to a cluster through separate `mongos` processes which are responsible for transparently routing the queries within the cluster.

RethinkDB clients can connect to any node in the cluster and queries will be automatically routed internally. Both simple (such as filters, joins) and composed queries (chained operations) will be broken down, routed to the appropriate machines, and executed in parallel. The results will be recombined and streamed back to the client.

Caching engine

MongoDB's storage engine uses memory mapped files which also function as an OS-level LRU caching system. MongoDB can use all free memory on the server for cache space automatically without any configuration of a cache size.

RethinkDB implements a custom B-tree aware caching mechanism. The cache size can be configured on a per-table basis.

Before you start: make sure you've installed RethinkDB — it should only take a minute!

Start the server

First, start the RethinkDB server like this:

```
$ rethinkdb
info: Creating directory 'rethinkdb_data'
info: Listening for intracluster connections on port 29015
info: Listening for client driver connections on port 28015
info: Listening for administrative HTTP connections on port 8080
info: Server ready
```

Point your browser to `localhost:8080` – you'll see an administrative UI where you can control the cluster (which so far consists of one machine), and play with the query language.

Run some queries

Click on the *Data Explorer* tab in the browser. You can manipulate data using JavaScript straight from your browser. By default, RethinkDB creates a database named `test`. Let's create a table:

```
r.db('test').tableCreate('tv_shows')
```

Use the "Run" button or Shift+Enter to run the query. Now, let's insert some JSON documents into the table:

```
r.table('tv_shows').insert([{ name: 'Star Trek TNG', episodes: 178 },
                    { name: 'Battlestar Galactica', episodes: 75 }])
```

We've just inserted two rows into the `tv_shows` table. Let's verify the number of rows inserted:

```
r.table('tv_shows').count()
```

Finally, let's do a slightly more sophisticated query. Let's find all shows with more than 100 episodes.

```
r.table('tv_shows').filter(r.row('episodes').gt(100))
```

As a result, we of course get the best science fiction show in existence.

Next steps

Congrats, you're on your way to database bliss! Now move on to the ten-minute guide and learn how to use the client drivers, get more in-depth information on basic commands, and start writing real applications with RethinkDB.

Before you start:

- Make sure you've installed RethinkDB — it should only take a minute!
- Make also sure you've installed the JavaScript driver.
- Read the thirty-second quickstart.

Import the driver

First, start Node.js:

```
$ node
```

Then, import the RethinkDB driver:

```
r = require('rethinkdb');
```

You can now access RethinkDB commands through the `r` module.

Open a connection

When you first start RethinkDB, the server opens a port for the client drivers (`28015` by default). Let's open a connection:

```
var connection = null;
r.connect( {host: 'localhost', port: 28015}, function(err, conn) {
    if (err) throw err;
    connection = conn;
})
```

The variable `connection` is now initialized and we can run queries.

Create a new table

By default, RethinkDB creates a database `test`. Let's create a table `authors` within this database:

```
r.db('test').tableCreate('authors').run(connection, function(err, result) {
    if (err) throw err;
    console.log(JSON.stringify(result, null, 2));
})
```

The result should be:

```
{ created: 1 }
```

There are a couple of things you should note about this query:

- First, we select the database `test` with the `db` command.
- Then, we add the `tableCreate` command to create the actual table.
- Lastly, we call `run(connection, callback)` in order to send the query to the server.

All ReQL queries follow this general structure. Now that we've created a table, let's insert some data!

Insert data

Let's insert three new documents into the `authors` table:

```
r.table('authors').insert([
    { name: "William Adama", tv_show: "Battlestar Galactica",
      posts: [
     {title: "Decommissioning speech", content: "The Cylon War is long over..."},
     {title: "We are at war", content: "Moments ago, this ship received word..."},
     {title: "The new Earth", content: "The discoveries of the past few days..."}
      ]
    },
    { name: "Laura Roslin", tv_show: "Battlestar Galactica",
      posts: [
      {title: "The oath of office", content: "I, Laura Roslin, ..."},
     {title: "They look like us", content: "The Cylons have the ability..."}
      ]
    },
    { name: "Jean-Luc Picard", tv_show: "Star Trek TNG",
      posts: [
     {title: "Civil rights", content: "There are some words I've known since..."}
      ]
    }
]).run(connection, function(err, result) {
    if (err) throw err;
    console.log(JSON.stringify(result, null, 2));
})
```

We should get back an object that looks like this:

```
{
    "unchanged": 0,
    "skipped": 0,
    "replaced": 0,
    "inserted": 3,
```

```
    "generated_keys": [
        "7644aaf2-9928-4231-aa68-4e65e31bf219",
        "064058b6-cea9-4117-b92d-c911027a725a",
        "543ad9c8-1744-4001-bb5e-450b2565d02c"
    ],
    "errors": 0,
    "deleted": 0
}
```

The server should return an object with zero errors and three inserted documents. We didn't specify any primary keys (by default, each table uses the `id` attribute for primary keys), so RethinkDB generated them for us. The generated keys are returned via the `generated_keys` attribute.

There are a couple of things to note about this query:

- Each connection sets a default database to use during its lifetime (if you don't specify one in `connect`, the default database is set to `test`). This way we can omit the `db('test')` command in our query. We won't specify the database explicitly from now on, but if you want to prepend your queries with the `db` command, it won't hurt.
- The `insert` command accepts a single document or an array of documents if you want to batch inserts. We use an array in this query instead of running three separate `insert` commands for each document.

Retrieve documents

Now that we inserted some data, let's see how we can query the database!

All documents in a table

To retrieve all documents from the table `authors`, we can simply run the query `r.table('authors')`:

```
r.table('authors').run(connection, function(err, cursor) {
    if (err) throw err;
    cursor.toArray(function(err, result) {
        if (err) throw err;
        console.log(JSON.stringify(result, null, 2));
    });
});
```

The result is an array of the three previously inserted documents, along with the generated `id` values.

Since the table might contain a large number of documents, the database returns a cursor object. As you iterate through the cursor, the server will send documents to the client in batches as they are requested. We only have three documents in our example, so we can safely retrieve all the documents at once. The `toArray` function automatically iterates through the cursor and puts the documents into a JavaScript array.

Filter documents based on a condition

Let's try to retrieve the document where the `name` attribute is set to `William Adama`. We can use a condition to filter the documents by chaining a `filter` command to the end of the query:

```
r.table('authors').filter(r.row('name').eq("William Adama")).
    run(connection, function(err, cursor) {
        if (err) throw err;
        cursor.toArray(function(err, result) {
            if (err) throw err;
            console.log(JSON.stringify(result, null, 2));
        });
    });
```

This query returns a cursor with one document — the record for William Adama. The `filter` command evaluates the provided condition for every row in the table, and returns only the relevant rows. Here's the new commands we used to construct the condition above:

- `r.row` refers to the currently visited document.
- `r.row('name')` refers to the value of the field `name` of the visited document.
- The `eq` command returns `true` if two values are equal (in this case, the field `name` and the string `William Adama`).

Let's use `filter` again to retrieve all authors who have more than two posts:

```
r.table('authors').filter(r.row('posts').count().gt(2)).
    run(connection, function(err, cursor) {
        if (err) throw err;
        cursor.toArray(function(err, result) {
            if (err) throw err;
            console.log(JSON.stringify(result, null, 2));
        });
    });
```

In this case, we're using a predicate that returns `true` only if the length of the array in the field `posts` is greater than two. This predicate contains two commands we haven't seen before:

- The `count` command returns the size of the array.
- The `gt` command returns `true` if a value is greater than the specified value (in this case, if the number of posts is greater than two).

Retrieve documents by primary key

We can also efficiently retrieve documents by their primary key using the `get` command. We can use one of the ids generated in the previous example:

```
r.table('authors').get('7644aaf2-9928-4231-aa68-4e65e31bf219').
    run(connection, function(err, result) {
        if (err) throw err;
        console.log(JSON.stringify(result, null, 2));
    });
```

Since primary keys are unique, the `get` command returns a single document. This way we can retrieve the document directly without converting a cursor to an array.

Learn more about how RethinkDB can efficiently retrieve documents with secondary indexes.

Update documents

Let's update all documents in the `authors` table and add a `type` field to note that every author so far is fictional:

```
r.table('authors').update({type: "fictional"}).
    run(connection, function(err, result) {
        if (err) throw err;
        console.log(JSON.stringify(result, null, 2));
    });
```

Since we changed three documents, the result should look like this:

```
{
    "unchanged": 0,
    "skipped": 0,
```

```
    "replaced": 3,
    "inserted": 0,
    "errors": 0,
    "deleted":0
}
```

Note that we first selected every author in the table, and then chained the `update` command to the end of the query. We could also update a subset of documents by filtering the table first. Let's update William Adama's record to note that he has the rank of Admiral:

```
r.table('authors').
    filter(r.row("name").eq("William Adama")).
    update({rank: "Admiral"}).
    run(connection, function(err, result) {
        if (err) throw err;
        console.log(JSON.stringify(result, null, 2));
    });
```

Since we only updated one document, we get back this object:

```
{
    "unchanged": 0,
    "skipped": 0,
    "replaced": 1,
    "inserted": 0,
    "errors": 0,
    "deleted": 0
}
```

The `update` command allows changing existing fields in the document, as well as values inside of arrays. Let's suppose Star Trek archaeologists unearthed a new speech by Jean-Luc Picard that we'd like to add to his posts:

```
r.table('authors').filter(r.row("name").eq("Jean-Luc Picard")).
    update({posts: r.row("posts").append({
        title: "Shakespeare",
        content: "What a piece of work is man..."})
    }).run(connection, function(err, result) {
        if (err) throw err;
        console.log(JSON.stringify(result, null, 2));
    });
```

After processing this query, RethinkDB will add an additional post to Jean-Luc Picard's document.

Browse the API reference for many more array operations available in RethinkDB.

Delete documents

Suppose we'd like to trim down our database and delete every document with less than three posts (sorry Laura and Jean-Luc):

```
r.table('authors').
    filter(r.row('posts').count().lt(3)).
    delete().
    run(connection, function(err, result) {
        if (err) throw err;
        console.log(JSON.stringify(result, null, 2));
    });
```

Since we have two authors with less than two posts, the result is:

```
{
    "unchanged": 0,
    "skipped": 0,
    "replaced": 0,
    "inserted": 0,
    "errors": 0,
    "deleted": 2
}
```

Learn more

Want to keep learning? Dive into the documentation:

- Read the introduction to RQL to learn about the ReQL concepts in more depth.
- Learn how to use map/reduce in RethinkDB.
- Learn how to use table joins in RethinkDB.
- Jump into the cookbook and browse through dozens of examples of common RethinkDB queries.

Before you start:

- Make sure you've installed RethinkDB — it should only take a minute!
- Make also sure you've installed the Python driver.
- Read the thirty-second quickstart.

There is currently no support for Python 3. A Python 3 port is in progress, see Github issue #1050.

Import the driver

First, start a Python shell:

```
$ python
```

Then, import the RethinkDB driver:

```
import rethinkdb as r
```

You can now access RethinkDB commands through the `r` module.

Open a connection

When you first start RethinkDB, the server opens a port for the client drivers (`28015` by default). Let's open a connection:

```
r.connect( "localhost", 28015).repl()
```

The `repl` command is a convenience method that sets a default connection in your shell so you don't have to pass it to the `run` command to run your queries.

Note: the `repl` command is useful to experiment in the shell, but you should pass the connection to the `run` command explicitly in real applications. See an example project for more details.

Create a new table

By default, RethinkDB creates a database `test`. Let's create a table `authors` within this database:

```
r.db("test").table_create("authors").run()
```

The result should be:

```
{ "created": 1 }
```

There are a couple of things you should note about this query:

- First, we select the database `test` with the `db` command.
- Then, we add the `table_create` command to create the actual table.
- Lastly, we call `run()` in order to send the query to the server.

All ReQL queries follow this general structure. Now that we've created a table, let's insert some data!

Insert data

Let's insert three new documents into the `authors` table:

```
r.table("authors").insert([
    { "name": "William Adama", "tv_show": "Battlestar Galactica",
      "posts": [
     {"title": "Decommissioning speech", "content": "The Cylon War is long over..."},
     {"title": "We are at war", "content": "Moments ago, this ship received..."},
     {"title": "The new Earth", "content": "The discoveries of the past few days..."}
      ]
    },
    { "name": "Laura Roslin", "tv_show": "Battlestar Galactica",
      "posts": [
     {"title": "The oath of office", "content": "I, Laura Roslin, ..."},
     {"title": "They look like us", "content": "The Cylons have the ability..."}
      ]
    },
    { "name": "Jean-Luc Picard", "tv_show": "Star Trek TNG",
      "posts": [
     {"title": "Civil rights", "content": "There are some words I've known since..."}
      ]
    }
]).run()
```

We should get back an object that looks like this:

```
{
    "unchanged": 0,
```

```
    "skipped": 0,
    "replaced": 0,
    "inserted": 3,
    "generated_keys": [
        "7644aaf2-9928-4231-aa68-4e65e31bf219",
        "064058b6-cea9-4117-b92d-c911027a725a",
        "543ad9c8-1744-4001-bb5e-450b2565d02c"
    ],
    "errors": 0,
    "deleted": 0
}
```

The server should return an object with zero errors and three inserted documents. We didn't specify any primary keys (by default, each table uses the `id` attribute for primary keys), so RethinkDB generated them for us. The generated keys are returned via the `generated_keys` attribute.

There are a couple of things to note about this query:

- Each connection sets a default database to use during its lifetime (if you don't specify one in `connect`, the default database is set to `test`). This way we can omit the `db('test')` command in our query. We won't specify the database explicitly from now on, but if you want to prepend your queries with the `db` command, it won't hurt.
- The `insert` command accepts a single document or an array of documents if you want to batch inserts. We use an array in this query instead of running three separate `insert` commands for each document.

Retrieve documents

Now that we inserted some data, let's see how we can query the database!

All documents in a table

To retrieve all documents from the table `authors`, we can simply run the query `r.table('authors')`:

```
cursor = r.table("authors").run()
for document in cursor:
    print document
```

The query returns the three previously inserted documents, along with the generated `id` values.

Since the table might contain a large number of documents, the database returns a cursor object. As you iterate through the cursor, the server will send documents to the client in batches as they are requested. The cursor is an iterable Python object so you can go through all of the results with a simple `for` loop.

Filter documents based on a condition

Let's try to retrieve the document where the `name` attribute is set to `William Adama`. We can use a condition to filter the documents by chaining a `filter` command to the end of the query:

```
cursor = r.table("authors").filter(r.row["name"] == "William Adama").run()
for document in cursor:
    print document
```

This query returns a cursor with one document — the record for William Adama. The `filter` command evaluates the provided condition for every row in the table, and returns only the relevant rows. Here's the new commands we used to construct the condition above:

- `r.row` refers to currently visited document.
- `r.row['name']` refers to the value of the field `name` of the visited document.
- The `==` operator is overloaded by the RethinkDB driver to execute on the server. It returns `True` if two values are equal (in this case, the field `name` and the string `William Adama`).

Let's use `filter` again to retrieve all authors who have more than two posts:

```
cursor = r.table("authors").filter(r.row["posts"].count() > 2).run()
for document in cursor:
    print document
```

In this case, we're using a predicate that returns `True` only if the length of the array in the field `posts` is greater than two. This predicate contains two commands we haven't seen before:

- The `count` command returns the size of the array.
- The `>` operator is overloaded by the RethinkDB driver to execute on the server. It returns `True` if a value is greater than a certain value (in this case, if the number of posts is greater than two).

Retrieve documents by primary key

We can also efficiently retrieve documents by their primary key using the `get` command. We can use one of the ids generated in the previous example:

```
r.db('test').table('authors').get('7644aaf2-9928-4231-aa68-4e65e31bf219').run()
```

Since primary keys are unique, the `get` command returns a single document. This way we can retrieve the document directly without iterating through a cursor.

Learn more about how RethinkDB can efficiently retrieve documents with secondary indexes.

Update documents

Let's update all documents in the `authors` table and add a `type` field to note that every author so far is fictional:

```
r.table("authors").update({"type": "fictional"}).run()
```

Since we changed three documents, the result should look like this:

```
{
    "unchanged": 0,
    "skipped": 0,
    "replaced": 3,
    "inserted": 0,
    "errors": 0,
    "deleted":0
}
```

Note that we first selected every author in the table, and then chained the `update` command to the end of the query. We could also update a subset of documents by filtering the table first. Let's update William Adama's record to note that he has the rank of Admiral:

```
r.table("authors").
    filter(r.row['name'] == "William Adama").
    update({"rank": "Admiral"}).run()
```

Since we only updated one document, we get back this object:

```
{
    "unchanged": 0,
    "skipped": 0,
    "replaced": 1,
    "inserted": 0,
    "errors": 0,
    "deleted": 0
}
```

The `update` command allows changing existing fields in the document, as well as values inside of arrays. Let's suppose Star Trek archaeologists unearthed a new speech by Jean-Luc Picard that we'd like to add to his posts:

```
r.table('authors').filter(r.row["name"] == "Jean-Luc Picard").
    update({"posts": r.row["posts"].append({
        "title": "Shakespeare",
        "content": "What a piece of work is man..."})
    }).run()
```

After processing this query, RethinkDB will add an additional post to Jean-Luc Picard's document.

Browse the API reference for many more array operations available in RethinkDB.

Delete documents

Suppose we'd like to trim down our database and delete every document with less than three posts (sorry Laura and Jean-Luc):

```
r.table("authors").
    filter( r.row["posts"].count() < 3 ).
    delete().run()
```

Since we have two authors with less than two posts, the result is:

```
{
    "unchanged": 0,
    "skipped": 0,
    "replaced": 0,
    "inserted": 0,
    "errors": 0,
    "deleted": 2
}
```

Learn more

Want to keep learning? Dive into the documentation:

- Read the introduction to RQL to learn about the ReQL concepts in more depth.
- Learn how to use map/reduce in RethinkDB.
- Learn how to use table joins in RethinkDB.
- Jump into the cookbook and browse through dozens of examples of common RethinkDB queries.

Before you start:

- Make sure you've installed RethinkDB — it should only take a minute!
- Make also sure you've installed the Ruby driver.
- Read the thirty-second quickstart.

Import the driver

First, start a Ruby shell:

```
$ irb
```

Then, import the RethinkDB driver:

```
require 'rethinkdb'
include RethinkDB::Shortcuts
```

You can now access RethinkDB commands through the `r` module.

Open a connection

When you first start RethinkDB, the server opens a port for the client drivers (`28015` by default). Let's open a connection:

```
r.connect(:host=>"localhost", :port=>28015).repl
```

The `repl` command is a convenience method that sets a default connection in your shell so you don't have to pass it to the `run` command to run your queries.

Note: the `repl` command is useful to experiment in the shell, but you should pass the connection to the `run` command explicitly in real applications. See an example project for more details.

Create a new table

By default, RethinkDB creates a database `test`. Let's create a table `authors` within this database:

```
r.db("test").table_create("authors").run
```

The result should be:

```
{ "created"=>1 }
```

There are a couple of things you should note about this query:

- First, we select the database `test` with the `db` command.
- Then, we add the `table_create` command to create the actual table.
- Lastly, we call `run` in order to send the query to the server.

All ReQL queries follow this general structure. Now that we've created a table, let's insert some data!

Insert data

Let's insert three new documents into the `authors` table:

```
r.table("authors").insert([
    { "name"=>"William Adama", "tv_show"=>"Battlestar Galactica",
      "posts"=>[
     {"title"=>"Decommissioning speech", "content"=>"The Cylon War is long over..."},
     {"title"=>"We are at war", "content"=>"Moments ago, this ship received..."},
     {"title"=>"The new Earth", "content"=>"The discoveries of the past few days..."}
      ]
    },
    { "name"=>"Laura Roslin", "tv_show"=>"Battlestar Galactica",
      "posts"=>[
     {"title"=>"The oath of office", "content"=>"I, Laura Roslin, ..."},
     {"title"=>"They look like us", "content"=>"The Cylons have the ability..."}
      ]
    },
    { "name"=>"Jean-Luc Picard", "tv_show"=>"Star Trek TNG",
      "posts"=>[
     {"title"=>"Civil rights", "content"=>"There are some words I've known since..."}
      ]
    }
]).run
```

We should get back an object that looks like this:

```
{
    "unchanged"=>0,
    "skipped"=>0,
    "replaced"=>0,
    "inserted"=>3,
    "generated_keys"=>[
        "71879b1b-e81e-48c4-a42c-f41f83d7133e",
        "f0a93ef3-cec0-4364-bde6-f664088b9e77",
        "f601347b-95a1-4ffe-83bd-ed04d4a3cce5"
    ],
    "errors"=>0,
    "deleted"=>0
}
```

The server should return an object with zero errors and three inserted documents. We didn't specify any primary keys (by default, each table uses the `id` attribute for primary keys), so RethinkDB generated them for us. The generated keys are returned via the `generated_keys` attribute.

There are a couple of things to note about this query:

- Each connection sets a default database to use during its lifetime (if you don't specify one in `connect`, the default database is set to `test`). This way we can omit the `db('test')` command in our query. We won't specify the database explicitly from now on, but if you want to prepend your queries with the `db` command, it won't hurt.
- The `insert` command accepts a single document or an array of documents if you want to batch inserts. We use an array in this query instead of running three separate `insert` commands for each document.

Retrieve documents

Now that we inserted some data, let's see how we can query the database!

All documents in a table

To retrieve all documents from the table `authors`, we can simply run the query `r.table('authors')`:

```
cursor = r.table("authors").run
cursor.each{|document| p document}
```

The query returns the three previously inserted documents, along with the generated `id` values.

Since the table might contain a large number of documents, the database returns a cursor object. As you iterate through the cursor, the server will send documents to the client in batches as they are requested. The cursor is an iterable Ruby object so you can go through all of the results with a simple `for` loop.

Filter documents based on a condition

Let's try to retrieve the document where the `name` attribute is set to `William Adama`. We can use a condition to filter the documents by chaining a `filter` command to the end of the query:

```
cursor = r.table("authors").filter{|author| author["name"].eq("William Adama") }.run
cursor.each{|document| p document}
```

This query returns a cursor with one document — the record for William Adama. The `filter` command evaluates the provided condition for every row in the table, and returns only the relevant rows. Here's the new commands we used to construct the condition above:

- `author` refers to the currently visited document.
- `autor['name']` refers to the value of the field `name` of the visited document.
- The `eq` command returns `true` if two values are equal (in this case, the field `name` and the string `William Adama`).

Let's use `filter` again to retrieve all authors who have more than two posts:

```
cursor = r.table("authors").filter{|author| author["posts"].count > 2}.run
cursor.each{|document| p document}
```

In this case, we're using a predicate that returns `true` only if the length of the array in the field `posts` is greater than two. This predicate contains two commands we haven't seen before:

- The `count` command returns the size of the array.
- The `>` operator is overloaded by the RethinkDB driver to execute on the server. It returns `True` if a value is greater than a certain value (in this case, if the number of posts is greater than two).

Retrieve documents by primary key

We can also efficiently retrieve documents by their primary key using the `get` command. We can use one of the ids generated in the previous example:

```
r.db('test').table('authors').get('7644aaf2-9928-4231-aa68-4e65e31bf219').run
```

Since primary keys are unique, the `get` command returns a single document. This way we can retrieve the document directly without iterating through a cursor.

Learn more about how RethinkDB can efficiently retrieve documents with secondary indexes.

Update documents

Let's update all documents in the `authors` table and add a `type` field to note that every author so far is fictional:

```
r.table("authors").update({"type"=>"fictional"}).run
```

Since we changed three documents, the result should look like this:

```
{
    "unchanged"=>0,
    "skipped"=>0,
    "replaced"=>3,
    "inserted"=>0,
    "errors"=>0,
    "deleted"=>0
}
```

Note that we first selected every author in the table, and then chained the `update` command to the end of the query. We could also update a subset of documents by filtering the table first. Let's update William Adama's record to note that he has the rank of Admiral:

```
r.table("authors").
    filter{|author| author["name"].eq("William Adama")}.
    update({"rank"=>"Admiral"}).run
```

Since we only updated one document, we get back this object:

```
{
    "unchanged"=>0,
    "skipped"=>0,
    "replaced"=>1,
    "inserted"=>0,
    "errors"=>0,
    "deleted"=>0
}
```

The `update` command allows changing existing fields in the document, as well as values inside of arrays. Let's suppose Star Trek archaeologists unearthed a new speech by Jean-Luc Picard that we'd like to add to his posts:

```
r.table('authors').filter{|author| author["name"].eq("Jean-Luc Picard")}.
    update{|author| {"posts"=>author["posts"].append({
        "title"=>"Shakespeare",
        "content"=>"What a piece of work is man..."})
    }}.run
```

After processing this query, RethinkDB will add an additional post to Jean-Luc Picard's document.

Browse the API reference for many more array operations available in RethinkDB.

Delete documents

Suppose we'd like to trim down our database and delete every document with less than three posts (sorry Laura and Jean-Luc):

```
r.table("authors").
    filter{ |author| author["posts"].count < 3 }.
    delete.run
```

Since we have two authors with less than two posts, the result is:

```
{
    "unchanged"=>0,
    "skipped"=>0,
    "replaced"=>0,
    "inserted"=>0,
    "errors"=>0,
    "deleted"=>2
}
```

Learn more

Want to keep learning? Dive into the documentation:

- Read the introduction to RQL to learn about the ReQL concepts in more depth.
- Learn how to use map/reduce in RethinkDB.
- Learn how to use table joins in RethinkDB.
- Jump into the cookbook and browse through dozens of examples of common RethinkDB queries.

Looking for another platform? See the complete list of platforms RethinkDB supports.

With binaries

We provide binaries for both 32-bit and 64-bit Ubuntu Lucid and above (>= 10.04).

To install the server, you have to add the RethinkDB PPA to your list of repositories and install via `apt-get`. To do this, paste the following lines into your terminal:

```
sudo add-apt-repository ppa:rethinkdb/ppa   && \
sudo apt-get update                         && \
sudo apt-get install rethinkdb
```

If you do not have the `add-apt-repository` command, install it first:

- Ubuntu Quantal and above (>= 12.10) — `sudo apt-get install software-properties-common`
- Earlier Ubuntu versions (< 12.10) — `sudo apt-get install python-software-properties`

Compile from source on Ubuntu 13.10

Get the build dependencies

Install the main dependencies:

```
sudo apt-get install git-core g++ nodejs npm libprotobuf-dev libgoogle-perftools-dev \
    libncurses5-dev libboost-all-dev nodejs-legacy
```

Get the source code

Clone the RethinkDB repository:

```
git clone --depth 1 -b v1.11.x https://github.com/rethinkdb/rethinkdb.git
```

Build RethinkDB

Kick off the build process:

```
cd rethinkdb
./configure
make
```

Compile from source on Ubuntu 12.04

Get the build dependencies

Install the main dependencies:

```
sudo apt-get install git-core g++ nodejs npm libprotobuf-dev libgoogle-perftools-dev \
    libncurses5-dev libboost-all-dev
```

Then install a more recent version of `node` with `n`.

```
sudo npm install -g n
sudo apt-get install curl
sudo n stable
```

Get the source code

Clone the RethinkDB repository:

```
git clone --depth 1 -b v1.11.x https://github.com/rethinkdb/rethinkdb.git
```

Build RethinkDB

Kick off the build process:

```
cd rethinkdb
./configure npm=/usr/local/bin/npm
make
```

You will find the `rethinkdb` binary in the `build/release/` subfolder.

Next steps: Now that you've installed RethinkDB, it's time to install client drivers for your language.

Looking for another platform? See the complete list of platforms RethinkDB supports.

Using the installer

Prerequisites: We provide native binaries for OS X Lion and above (>= 10.7).

Download the disk image, run `rethinkdb.pkg`, and follow the installation instructions.

Troubleshooting: If you see the message *"rethinkdb.pkg can't be opened because it is from an unidentified developer"*, right click on the file and choose Open. See GitHub issue #1565 for more information on signed packages.

Using Homebrew

Prerequisites: Make sure you're on OS X Lion or above (>= 10.7) and have Homebrew installed.

Run the following in your terminal:

```
brew update && brew install rethinkdb
```

Compile from source

Get the build dependencies

There are a number of packages required for the build process. On OS X, Xcode is required to build from source.

You will also need to install boost with Homebrew:

```
brew install boost
```

Get the source code

Clone the RethinkDB repository:

```
git clone --depth 1 -b v1.11.x https://github.com/rethinkdb/rethinkdb.git
```

Build RethinkDB

Kick off the build process:

```
cd rethinkdb
./configure --fetch protobuf --fetch v8
make
```

You will find the `rethinkdb` binary in the `build/release/` subfolder.

Next steps: Now that you've installed RethinkDB, it's time to install client drivers for your language.

Looking for another platform? See the complete list of platforms RethinkDB supports.

With binaries

We provide binaries for both 32-bit and 64-bit CentOS 6.

To install the server, add the RethinkDB yum repository to your list of repositories and install:

```
sudo wget http://download.rethinkdb.com/centos/6/`uname -m`/rethinkdb.repo \
          -O /etc/yum.repos.d/rethinkdb.repo
sudo yum install rethinkdb
```

Compile from source with the Epel repository

These instructions have been tested on CentOS 6.5.

Get the build dependencies

CentOS provides neither a v8-devel package nor Node.js, so we need to get them from the Epel repository:

```
sudo rpm -Uvh http://download.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch
```

Install the main dependencies:

```
sudo yum install git-core gcc-c++ ncurses-devel boost-static protobuf-devel nodejs \
    npm gperftools-devel
```

Get the source code

Clone RethinkDB repository:

```
git clone --depth 1 -b v1.11.x https://github.com/rethinkdb/rethinkdb.git
```

Build RethinkDB

Kick off the build process:

```
cd rethinkdb
./configure --dynamic tcmalloc_minimal
make
```

Compile from source without the Epel repository

These instructions have been tested on CentOS 6.5.

Get the build dependencies

Install the main dependencies:

```
sudo yum install git-core gcc-c++ ncurses-devel boost-static svn
```

Get the source code

Clone RethinkDB repository:

```
git clone --depth 1 -b v1.11.x https://github.com/rethinkdb/rethinkdb.git
```

Build RethinkDB

Kick off the build process:

```
cd rethinkdb
./configure --fetch protoc --fetch npm --fetch tcmalloc_minimal
make
```

Next steps: Now that you've installed RethinkDB, it's time to install client drivers for your language.

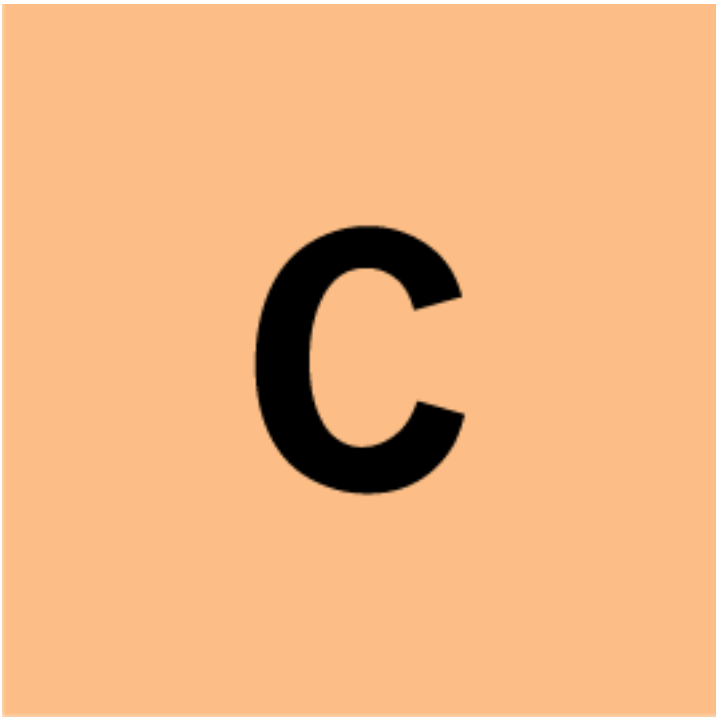
Official drivers »

JavaScript

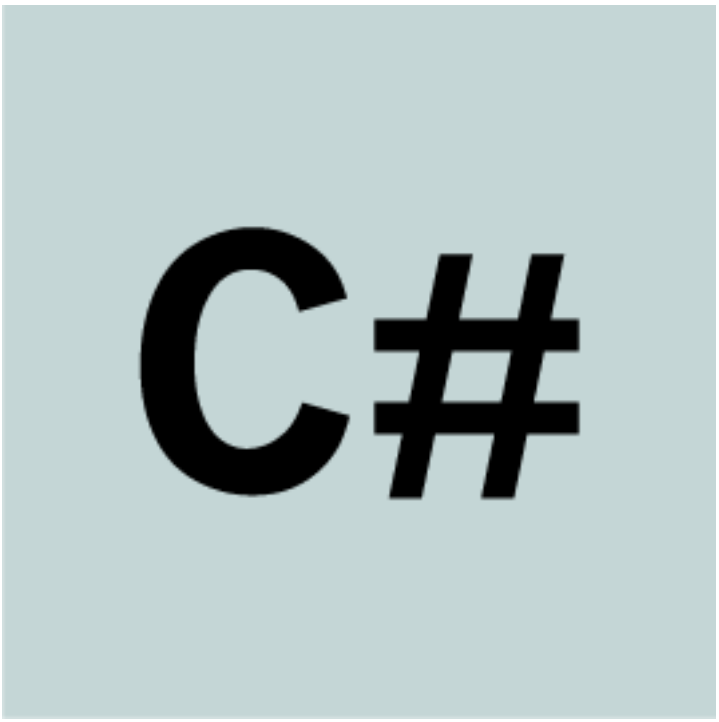
Ruby

Python

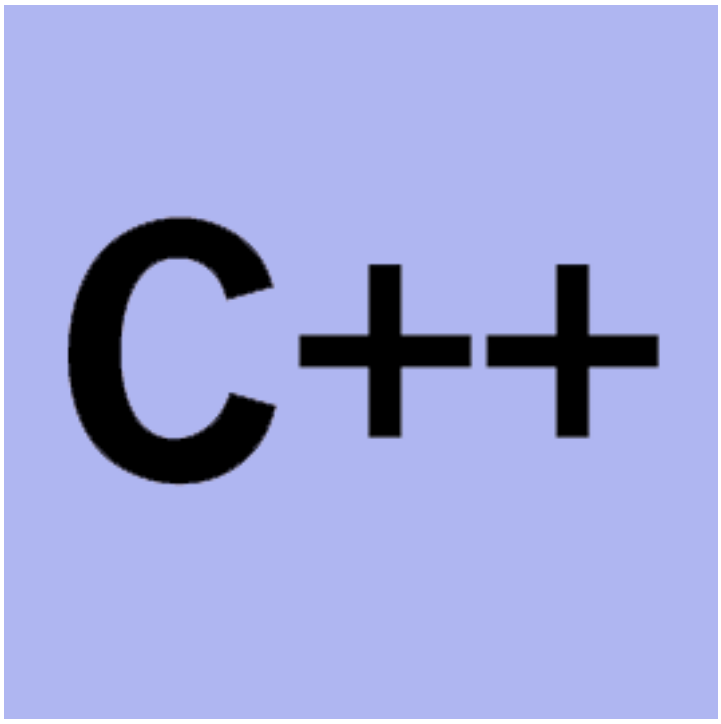
Community-supported drivers »



C



C# / .NET



C++

Clojure

Common Lisp

Elixir

Erlang

Go

Haskell

Java

Perl

PHP

Scala by @kclay

Scala by @esycat

Haven't installed the server yet? Go do that first!

Contribute a driver

Help make RethinkDB available on more platforms— consider contributing a driver for another language, or join one of the existing projects.

Thanks to all our amazing driver contributors!

- The rethinkdb-net team (C# / .NET): https://github.com/mfenniak/rethinkdb-net
- @bitemyapp and @cesarbp (Clojure): https://github.com/bitemyapp/revise
- @atnnn (Haskell): https://github.com/atnnn
- @christopherhesse (Go): https://github.com/christopherhesse/rethinkgo
- @dancannon (Go): https://github.com/dancannon/gorethink
- @danielmewes (PHP): https://github.com/danielmewes/php-rql

- @dkhenry (Java): https://github.com/dkhenry/rethinkjava
- @esycat (Scala): https://github.com/esycat/rql-scala
- @kclay (Scala): https://github.com/kclay/rethink-scala
- @njlg (Perl): https://github.com/njlg/perl-rethinkdb
- @orthecreedence (Common Lisp): https://github.com/orthecreedence/cl-rethinkdb
- @taybin (Erlang): https://github.com/taybin/lethink
- @nuxlli (Elixir): https://github.com/azukiapp/elixir-rethinkdb
- @unbit (C): https://github.com/unbit/librethinkdb
- @jurajmasar (C++): https://github.com/jurajmasar/rethink-db-cpp-driver

Before you install a driver: install RethinkDB first.

Looking for another language? See the complete list of client drivers for RethinkDB.

Installation

Prerequisites: The JavaScript driver requires Node.js >= 0.10.0.

Install the driver with npm:

```
$ npm install rethinkdb
```

Usage

You can use the drivers from Node.js like this:

```
$ node
r = require('rethinkdb')
r.connect({ host: 'localhost', port: 28015 }, function(err, conn) {
  if(err) throw err;
 r.db('test').tableCreate('tv_shows').run(conn, function(err, res) {
    if(err) throw err;
    console.log(res);
   r.table('tv_shows').insert({ name: 'Star Trek TNG' }).run(conn, function(err, res)
    {
      if(err) throw err;
      console.log(res);
    });
  });
});
```

Optional: optimized backend

For faster JavaScript driver performance, read about using an optimized C++ protobuf backend.

Next steps

Move on to the ten-minute guide and learn how to use RethinkDB.

Before you install a driver: install RethinkDB first.

Looking for another language? See the complete list of client drivers for RethinkDB.

Installation

Install the driver with gem:

```
sudo gem install rethinkdb
```

Usage

You can use the drivers from Ruby like this:

```
$ irb
require 'rubygems'
require 'rethinkdb'
include RethinkDB::Shortcuts
r.connect(:host => 'localhost', :port => 28015).repl
r.db('test').table_create('tv_shows').run
r.table('tv_shows').insert({ 'name'=>'Star Trek TNG' }).run
```

Next steps

Move on to the ten-minute guide and learn how to use RethinkDB.

Before you install a driver: install RethinkDB first.

Looking for another language? See the complete list of client drivers for RethinkDB.

Installation

The Python driver for RethinkDB is compatible with Python 2.

Install the driver with pip:

```
$ sudo pip install rethinkdb
```

Usage

You can use the drivers from Python like this:

```
$ python
import rethinkdb as r
r.connect('localhost', 28015).repl()
r.db('test').table_create('tv_shows').run()
r.table('tv_shows').insert({ 'name': 'Star Trek TNG' }).run()
```

Note: If you have `google-app-engine` installed, you may have a name collision between `google-app-engine` and `protobuf`. Renaming/removing the symbolic link `google` in `/usr/lib/python2.7/site-packages` is a temporary solution. You can track progress for a better solution on Github issue #901.

Optional: optimized backend

For faster Python driver performance, read about using an optimized C++ protobuf backend.

Next steps

Move on to the ten-minute guide and learn how to use RethinkDB.

JavaScript

The JavaScript driver can take advantage of the C++ protocol buffer backend for faster performance.

First, install the protobuf library and development files. On Ubuntu, they can be installed by running this command:

```
sudo apt-get install libprotobuf-dev
```

On platforms other than Ubuntu, if the package manager does not have the protobuf development files, they can be downloaded from http://code.google.com/p/protobuf/downloads/list.

Second, the `node-protobuf` npm package must be installed alongside the `rethinkdb` npm package:

```
npm install rethinkdb
npm install node-protobuf
```

The Javascript driver will automatically use the `node-protobuf` package if it is installed. You can verify that it is being used by checking `r.protobuf_implementation`:

```
$ node
r = require('rethinkdb')
r.protobuf_implementation
```

If the output is `'cpp'` then you are running the optimized C++ backend. Well done!

If the output is `'js'` then the `node-protobuf` package was not installed correctly.

Python

For faster performance, the Python driver requires a protobuf library that uses a C++ backend. Out of the box, the Python driver will not use the optimized backend.

To build the optimized backend, first install the protobuf library and development files. On Ubuntu, they can be installed by running this command:

```
sudo apt-get install libprotobuf-dev protobuf-compiler
```

On platforms other than Ubuntu, if the package manager does not have the protobuf development files, they can be downloaded from http://code.google.com/p/protobuf/downloads/list.

The second step for building the C++ implementation is to set the following variable to `cpp`:

```
export PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION=cpp
```

The third step is to install the C++ implementation of the protobuf python library. On most versions of Ubuntu, this command will install the library:

```
sudo apt-get install python-protobuf
```

Some versions of this package (notably, the one installed by pip) do not contain the C++ implementation. The protobuf python library needs to be installed from source. Run following command. It will download and build the protobuf library if the protobuf library is not installed or if it is installed without the C++ implementation.

```
python -c 'from google.protobuf.internal import cpp_message' || \
  protoc --version && pbver=$(protoc --version | \
 awk '{ print $2 }') && wget http://protobuf.googlecode.com/files/protobuf-$pbver.tar.gz &&
  tar xf protobuf-$pbver.tar.gz && cd protobuf-$pbver/python && \
  PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION=cpp python setup.py build
```

Once built, the protobuf library can be installed by using:

```
sudo PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION=cpp python setup.py install
```

The fourth and last step is to reinstall the `rethinkdb` python package. This step will fail if it cannot build the RethinkDB-specific C++ code.

```
sudo pip uninstall rethinkdb
sudo PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION=cpp pip install rethinkdb
```

You can verify that you are running the C++ backend by checking the following:

```
$ python
import rethinkdb as r
r.protobuf_implementation
```

If the output is `'cpp'` then you are running the optimized C++ backend. Bravo!

If the output is `'python'` the driver will still work, but may have performance penalties.

Want to set up a production cluster? See the production cluster setup page to learn how to set up RethinkDB with `init.d` or `systemd`.

Adding a node to a RethinkDB cluster is as easy as starting a new RethinkDB process and pointing it to an existing node in the cluster. Everything else is handled by the system without any additional effort required from the user.

You can set up multiple RethinkDB instances on a single machine, or use multiple physical machines or VMs to run a distributed cluster.

Multiple RethinkDB instances on a single machine

To start the first RethinkDB instance, run this command in your terminal:

```
$ rethinkdb
info: Creating directory /home/user/rethinkdb_data
info: Listening for intracluster connections on port 29015
info: Listening for client driver connections on port 28015
info: Listening for administrative HTTP connections on port 8080
info: Server ready
```

Note the port numbers you can use to access RethinkDB:

- Use the intracluster port (`29015` by default) to connect other nodes in the cluster to this node.
- Point your browser to the HTTP connections port (`8080` by default) to access the web interface.

Now start the second RethinkDB instance on the same machine:

```
$ rethinkdb --port-offset 1 --directory rethinkdb_data2 --join localhost:29015
info: Creating directory /home/user/rethinkdb_data2
info: Listening for intracluster connections on port 29016
info: Attempting connection to 1 peer...
info: Connected to server "Chaosknight" e6bfec5c-861e-4a8c-8eed-604cc124b714
info: Listening for client driver connections on port 28016
info: Listening for administrative HTTP connections on port 8081
info: Server ready
```

You now have a RethinkDB cluster! Try pointing your browser to `localhost:8080` or `localhost:8081` to access the web interface. If you click on the "Servers" tab at the top, you should see both servers in the cluster.

You can also point the client drivers to `localhost:28015` or `localhost:28016` to start running queries (it doesn't matter which node you use — the cluster will automatically route all commands to appropriate nodes).

Note the command line parameters we used to start the second node:

- `--port-offset` — increment all ports by 1 so the two nodes don't try to use the same ports on one machine.
- `--directory` — use a different data directory so the two nodes don't try to access the same files.

- `--join` — tell our new RethinkDB instance to connect to another instance (in this case, `localhost:29015`).

Having trouble accessing the web interface? Try restarting both of your RethinkDB instances with an additional `--bind all` parameter.

Want to connect a third node? You can join it with either of the two existing nodes in the cluster.

A RethinkDB cluster using multiple machines

Starting a cluster on multiple machines or VMs is even easier than starting it on a single machine, because you don't have to worry about port and directory conflicts.

First, start RethinkDB on the first machine:

```
$ rethinkdb --bind all
```

Then start RethinkDB on the second machine:

```
$ rethinkdb --join IP_OF_FIRST_MACHINE:29015 --bind all
```

You now have a RethinkDB cluster!

Note that by default, RethinkDB only opens connections bound to `localhost` in order to prevent unathorized clients on the network from connecting to the server. The `--bind all` option allows connections from anywhere on the network. It works well if the network is protected.

If your network is open to the internet, you might have to take additional precautions. See the security page for more details.

Troubleshooting

Seeing a 'received invalid clustering header' message? RethinkDB uses three ports to operate — the HTTP web UI port, the client drivers port, and the intracluster traffic port. You can connect the browser to the web UI port to administer the cluster right from your browser, and connect the client drivers to the client driver port to run queries from your application. If you're running a cluster, different RethinkDB nodes communicate with each other via the intracluster traffic port.

The message `received invalid clustering header` means there is a port mismatch, and something is connecting to the wrong port. For example, it's common to get this message if you accidently point the browser or connect the client drivers to the intracluster traffic port.

ReQL is the RethinkDB query language. It offers a very powerful and convenient way to manipulate JSON documents. This document is a gentle introduction to ReQL concepts. You don't have to read it to be productive with RethinkDB, but it helps to understand some basics.

Want to write useful queries right away? Check out the ten-minute guide.

ReQL is different from other NoSQL query languages. It's built on three key principles:

1. **ReQL embeds into your programming language.** Queries are constructed by making function calls in the programming language you already know. You don't have to concatenate strings or construct specialized JSON objects to query the database.
2. **All ReQL queries are chainable**. You begin with a table and incrementally chain transformers to the end of the query using the `.` operator.
3. **All queries execute on the server**. While queries are constructed on the client in a familiar programming language, they execute entirely on the database server once you call the `run` command and pass it an active database connection.

Let's look at these concepts in more detail.

Note: the following examples use the Python driver, but most of them also apply to RethinkDB drivers for other languages.

ReQL embeds into your programming language

You start using ReQL in your program similarly to how you'd use other databases:

```
import rethinkdb as r  # import the RethinkDB package
conn = r.connect()     # connect to the server on localhost and default port
```

But this is where the similarity ends. Instead of constructing strings and passing them to the database server, you access ReQL by using methods from the `rethinkdb` package:

```
r.table_create('users').run(conn)   # create a table `users`
r.table('users').run(conn)       # get an iterable cursor to the `users` table
```

Every ReQL query, from filters, to updates, to table joins is done by calling appropriate methods.

This design has the following advantages:

- You can use the same programming environment and tools you're already used to.
- Learning the language is no different from learning any other library.
- There is little to no chance of security issues that arise from string injection attacks.

All ReQL queries are chainable

In ReQL, you can chain commands at the end of other commands using the `.` operator:

```
# Get an iterable cursor to the `users` table (we've seen this above)
r.table('users').run(conn)

# Return only the `last_name` field of the documents
r.table('users').pluck('last_name').run(conn)

# Get all the distinct last names (remove duplicates)
r.table('users').pluck('last_name').distinct().run(conn)

# Count the number of distinct last names
r.table('users').pluck('last_name').distinct().count().run(conn)
```

Almost all ReQL operations are chainable. You can think of the `.` operator similarly to how you'd think of a Unix pipe. You select the data from the table and pipe it into a command that transforms it. You can continue chaining transformers until your query is done. In ReQL, data flows from left to right.

Even if you have a cluster of RethinkDB nodes, you can send your queries to any node and the cluster will create and execute distributed programs that get the data from relevant nodes, perform the necessary computations, and present you with final results without you ever worrying about it.

This design has the following advantages:

- The language is easy to learn, read, and modify.
- It's a natural and convenient way to express queries.
- You can construct queries incrementally by chaining transformations and examining intermediary results.

ReQL is efficient

Server-side execution

While queries are built up on the client, they're only sent to the server once you call the `run` command. All processing happens on the server — the queries don't run on the client, and don't require intermediary network round trips between the client and the server. For example, you can store queries in variables, and send them to the server later:

```
# Create the query to get distinct last names
distinct_lastnames_query = r.table('users').pluck('last_name').distinct()

# Send it to the server and execute
distinct_lastnames_query.run(conn)
```

Read about how this technology is implemented for more details.

Laziness

ReQL queries are executed lazily:

```
# Get up to five user documents that have the `age` field defined
r.table('users').has_fields('age').limit(5).run(conn)
```

For this query RethinkDB will perform enough work to get the five documents, and stop when the query is satisfied. Even if you don't have a limit on the number of queries but use a cursor, RethinkDB will do just enough work to allow you to read the data you request. This allows queries to execute quickly without wasting CPU cycles, network bandwidth, and disk IO.

Like most database systems, ReQL supports primary and secondary indexes to allow efficient data access. You can also create compound indexes and indexes based on arbitrary ReQL expressions to speed up complex queries.

Learn how to use primary and secondary indexes in RethinkDB.

Parallelism

All ReQL queries are automatically parallelized on the RethinkDB server as much as possible. Whenever possible, query execution is split across CPU cores, machines in the cluster, and even multiple datacenters. If you have large, complicated queries that require multiple stages of processing, RethinkDB will automatically break them up into stages, execute each stage in parallel, and combine data to return a complete result.

Query optimization

While RethinkDB doesn't currently have a fully-featured query optimizer, ReQL is designed with one in mind. For example, the server has enough information to reorder the chain for efficiency, or to use alternative implementation plans to improve performance. This feature will be introduced into future versions of RethinkDB.

ReQL queries are functional

So far we've seen only simple queries without conditions. ReQL supports a familiar syntax for building more advanced queries:

```
# Get all users older than 30
r.table('users').filter(lambda user: user['age'] > 30).run(conn)

# If you'd like to avoid writing lambdas, RethinkDB supports an
# alternative syntax:
r.table('users').filter(r.row['age'] > 30).run(conn)
```

This query looks just like any other Python code you would normally write. Note that RethinkDB will execute this query on the server, and it doesn't execute native Python code.

The client drivers do a lot of work to inspect the code and convert it to an efficient ReQL query that will be executed on the server:

- Whenever possible, the client drivers use operator overloading to support expressions such as `user['age'] > 30`.
- The `lambda` expression is executed only once on the client. Internally, the driver passes a special object to the `lambda` function which allows constructing a representation of the query. This representation is then sent to the server over the network and evaluated on the cluster.

Read about how this technology is implemented for more details.

This technology has limitations. While most operations allow you to write familiar code, you can't use native language's operations that have side effects (such as `print`) or control blocks (such as `if` and `for`). Instead, you have to use alternative ReQL commands:

```
# WRONG: Get all users older than 30 using the `if` statement
r.table('users').filter(lambda user:
   print "Testing"    # WRONG: this will only execute once on the client
```

```
    if user['age'] > 30:
        True,
        False).run(conn)

# RIGHT: Get all users older than 30 using the `r.branch` command
r.table('users').filter(lambda user:
    r.branch(user['age'] > 30,
             True,
             False)).run(conn)
```

This design has the following advantages:

- For most queries, you can write familiar, easy to learn code without learning special commands.
- The queries are efficiently transported to the server (via protocol buffers), and evaluated in the cluster.
- RethinkDB has access to the query structure, which allows for optimization techniques similar to those available in SQL. This feature will be added to RethinkDB in the future.

This technology has the following limitation:

- Native language's operations that have side effects or control blocks cannot be used within a `lambda`. Learn more about how this design is implemented for details.

ReQL queries are composable

You can combine multiple ReQL queries to build more complex ones.

Composing simple commands

Let's start with a simple example. RethinkDB supports server-side Javascript evaluation using the embedded V8 engine (sandboxed within outside processes, of course):

```
# Evaluate a Javascript expression on the server and get the result
r.js('1 + 1').run(conn)
```

Because ReQL is composable you can combine the `r.js` command with any other query. For example, let's use it as an alternative to get all users older than 30:

```
# Get all users older than 30 (we've seen this above)
r.table('users').filter(lambda user: user['age'] > 30).run(conn)

# Get all users older than 30 using server-side JavaScript
r.table('users').filter(r.js('(function (user) { return user.age > 30; })')).run(conn)
```

RethinkDB will seamlessly evaluate the `js` command by calling into the V8 engine during the evaluation of the `filter` query. You can combine most queries this way into progressively more complex ones.

Subqueries

Let's say we have another table `authors`, and we'd like to get a list of authors whose last names are also in the `users` table we've seen before. We can do it by combining two queries:

```
# Find all authors whose last names are also in the `users` table
r.table('authors').filter(lambda author:
   r.table('users').pluck('last_name').contains(author.pluck('last_name'))).
    run(conn)
```

Here, we use the `r.table('users').pluck('last_name')` query as the inner query in `filter`, combining the two queries to build a more sophisticated one. Even if you have a cluster of machines and both the `authors` table and the `users` table are sharded, RethinkDB will do the right thing and evaluate relevant parts of the query above on the appropriate shards, combine bits of data as necessary, and return the complete result.

A few things to note about this query:

- We compose the query on the client and call `run` only once. Remember to call `run` only once on the complex query when you're ready for it to be executed.
- You can also perform this query using the inner_join command.

Expressions

Composing queries isn't limited to simple commands and inner queries. You can also use expressions to perform complex operations. For example, suppose we'd like to find all users whose salary and bonus don't exceed $90,000, and increase their salary by 10%:

```
r.table('users').filter(lambda user: user['salary'] + user['bonus'] < 90000)
 .update(lambda user: {'salary': user['salary'] + user['salary'] * 0.1})
```

Rich command-set

In addition to commands described here, ReQL supports a number of sophisticated commands that are composable similarly to the commands described here. See the following documentation for more details:

- Learn how to use map/reduce in RethinkDB.
- Learn how to use table joins in RethinkDB.
- Browse the API reference for more commands.

This design has the following advantages:

- Unlike most NoSQL languages, you can use ReQL to build queries of arbitrary complexity.
- There is no new syntax or new commands for complex queries. Once you understand the composition principle you can write new queries without learning anything else.
- Subqueries can be abstracted in variables, which allows for modular programming in the same way as done by most other modern programming languages.

And just for kicks, ReQL can do math!

Just in case you needed another calculator, ReQL can do that too!

```
# Add two plus two
(r.expr(2) + r.expr(2)).run(conn)

# You only need to specify `r.expr` once for the driver to work
(r.expr(2) + 2).run(conn)

# More algebra
(r.expr(2) + 2 / 2).run(conn)

# Logic
(r.expr(2) > 3).run(conn)

# Branches
r.branch(r.expr(2) > 3,
         1,  # if True, return 1
         2   # otherwise, return 2
  ).run(conn)
```

```
# Compute the Fibonacci sequence
r.table_create('fib').run(conn)
r.table('fib').insert([{'id': 0, 'value': 0}, {'id': 1, 'value': 1}]).run(conn)
r.expr([2, 3, 4, 5, 6, 7, 8, 9, 10, 11]).for_each(lambda x:
  r.table('fib').insert({'id': x,
                 'value': (r.table('fib').order_by('id').nth(x - 1)['value'] +
                       r.table('fib').order_by('id').nth(x - 2)['value'])
                  })).run(conn)
r.table('fib').order_by('id')['value'].run(conn)
```

Read More

Browse the following resources to learn more about ReQL:

- Lambda functions in RethinkDB
- Introduction to map/reduce
- Introduction to Joins
- API Reference

Map/reduce is an operation on a sequence of documents that allows performing distributed data processing at scale. Originally designed by Google and later implemented in systems like Apache Hadoop, a map/reduce query is composed of two parts:

- A **map** operation — each document is mapped to a key/value pair.
- A **reduce** operation — an aggregation operation (such as counting or summation) that operates on values grouped by key.

RethinkDB implements an efficient, distributed map/reduce infrastructure. Many ReQL operations automatically compile to a map/reduce query. However, you can also use map/reduce directly via the grouped_map_reduce command.

Want to learn more about map/reduce? Read the Wikipedia article.

An example

Let's suppose you are running a blog and would like to retrieve the number of posts per category. A map/reduce query to perform this operation would consist of the following steps:

- A **map** step that returns a key/value pair for each post, where the key is the category of the post and the value is `1` (since each post needs to counted once).
- A **reduce** step that sums the values for each category.

Map/reduce in RethinkDB

For our blog, we have a table `posts` that contains blog posts. Here's an example document from the table:

```
{
    "id": "7644aaf2-9928-4231-aa68-4e65e31bf219"
    "title": "The line must be drawn here"
    "content": "This far, no further! ..."
    "category": "Fiction"
}
```

Let's compute the number of posts per category. The `grouped_map_reduce` command requires three arguments: a grouping function, a mapping function, and a reduction function.

- **A grouping function** must return a group of a given document.
- **A map function** must return a value for a given document that will be aggregated.
- **A reduce function** specifies how to reduce all the values.

Note: Hadoop combines the grouping stage and the mapping stage into a single, slightly more complicated mapping function. For ease of use, RethinkDB breaks up the complex mapping function into two simpler operations.

To compute the number of posts per category, we would write:

```
r.table("post").grouped_map_reduce(
   lambda post: post["category"],  # Returns the group (category) for each post
    lambda post: 1,                  # Each post will counted once
    lambda x, y: x + y               # Sum two values
).run()
```

This map/reduce query is equivalent to a simpler ReQL command:

```
r.table("post").grouped_by("category", r.count).run()
```

Note: a more user friendly map/reduce syntax will be available soon — see Github issue #1096.

How map/reduce queries are executed

One important aspect of the `grouped_map_reduce` is that it is distributed and parallelized across shards and CPU cores. This allows map/reduce queries to execute efficiently, but is a source of a common mistake: assuming an **incorrect** reduction order.

Since the reduction step is performed in parallel, the summation operation above may be performed in the following way:

```
(1 + 1) + (1 + 1)
```

In other words, the reduction is **not** always performed from left to right. Here is an example of an incorrect way to write the `grouped_map_reduce` query:

```
r.table("post").grouped_map_reduce(
    lambda post: post["category"],
    lambda post: 1,
    lambda x, y: x + 1 # Will not work! `x` is not an aggregator!
).run()
```

If we have four documents in a single category in a sharded table, here is a possible execution path for the command above:

1. Two of the documents are located on shard 1, the other two documents are on shard 2. RethinkDB runs the reduction on both shards in parallel.
2. The number of documents is computed on shard 1 — the query returns the value `2` for the shard.
3. The number of documents is computed on shard 2 — the query returns the value `2` for the shard.
4. However, the final reduction step (combining the values of two shards) doesn't work. Instead of computing `2 + 2` the query above will compute `2 + 1` instead.

Be careful! Make sure your reduction function doesn't assume the reduction step executes from left to right!

Simplified map/reduce

In cases where you need to do simpler computations, the grouping stage may not be necessary. If you don't need a grouping function you can use the simpler `map` and `reduce` commands directly:

For example, to count the number of posts in a table you can run the following query:

```
r.db("blog").table("posts").map(lambda row: 1).reduce(lambda x, y: x + y).run()
```

An equivalent full map/reduce query would look like this:

```
r.table("post").grouped_map_reduce(
    lambda post: 1,       # We only have one group
    lambda post: 1,       # Each post will counted once
    lambda x, y: x + y    # Sum two values
).run()
```

Both of these queries are equivalent to the following ReQL query:

```
r.db("blog").table("posts").count().run()
```

Read more

- The API documentation for the `grouped_map_reduce` command.
- The Wikipedia article on map/reduce.
- The Hadoop tutorial for map/reduce.

Wondering how to model your data? Read about data modeling in RethinkDB.

Like many traditional database systems, RethinkDB supports `JOIN` commands to combine data from multiple tables. In RethinkDB joins are automatically distributed — a join command is automatically sent to the appropriate nodes across the cluster, the relevant data is combined, and the final result is presented to the user.

Let's see how we can use joins in RethinkDB to query data based on **one to many**, and **many to many** relations.

One to many relations

Using primary keys

Let's suppose we've created two tables: `employees` and `companies`. We'll use these tables to model the notion of people working for organizations (each organization has multiple people working for it, but any given person works at a single organization). Here's an example document in the `employees` table:

```
{
    "id": "543ad9c8-1744-4001-bb5e-450b2565d02c",
    "name": "Jean-Luc Picard",
    "company_id": "064058b6-cea9-4117-b92d-c911027a725a",
    "rank": "captain"
}
```

And here's an example document in the `companies` table:

```
{
    "id": "064058b6-cea9-4117-b92d-c911027a725a",
    "company": "Starfleet",
    "type": "paramilitary"
}
```

We can join the two tables as follows:

```
r.table("employees").eq_join("company_id", r.table("companies")).run()
```

This query joins the `company_id` of the employee table with the primary key of the company table. It returns a sequence of documents where each document contains two fields — the employee information and the company information:

```
{
    "left": {
        "id": "543ad9c8-1744-4001-bb5e-450b2565d02c",
        "name": "Jean-Luc Picard",
        "company_id": "064058b6-cea9-4117-b92d-c911027a725a",
        "rank": "captain"
    },
    "right": {
        "id": "064058b6-cea9-4117-b92d-c911027a725a",
        "company": "Starfleet",
        "type": "paramilitary"
    }
}
```

- The field `left` contains the information from the left table in the query (in this case, the employee)
- The field `right` contains the information from the right table in the query (in this case, the company)

We can chain the `zip` command at the end of the query to merge the two fields into a single document. For example, the following query:

```
r.table("employees").eq_join("company_id", r.table("companies")).zip().run()
```

Returns the following result:

```
{
    "id": "064058b6-cea9-4117-b92d-c911027a725a",
    "name": "Jean-Luc Picard",
    "company_id": "064058b6-cea9-4117-b92d-c911027a725a",
    "rank": "captain",
    "company": "Starfleet",
    "type": "paramilitary"
}
```

Using secondary indexes

Suppose that our data model for the employees stores a company name instead of a company id:

```
{
    "id": "543ad9c8-1744-4001-bb5e-450b2565d02c",
    "name": "Jean-Luc Picard",
    "company_name": "Starfleet",
    "rank": "captain"
}
```

We can create a secondary index on the `company` field of the `companies` table, and perform our query by taking advantage of the secondary index:

```
r.table("companies").index_create("company").run()
```

The query would look like this:

```
r.table("employees").eq_join("company_name",
                      r.table("companies"), index="company").run()
```

Want to learn more about indexes?: Read about using secondary indexes in RethinkDB.

Note: you can also join tables on arbitrary fields without creating an index using the inner_join command. However, arbitrary inner joins are less efficient then equijoins.

Many to many relations

You can also use RethinkDB to query many to many relations. Let's suppose we have a collaborative blogging platform where authors collaborate to create posts (multiple authors can work on any given post, and publish multiple posts).

In order to model this data we'd create three tables — `authors`, `posts` and `authors_posts`, similarly to how we'd do it in a relational system. Here is example data for the `authors` table:

```
{
  "id": "7644aaf2-9928-4231-aa68-4e65e31bf219",
  "name": "William Adama",
  "tv_show": "Battlestar Galactica"
}
{
  "id": "064058b6-cea9-4117-b92d-c911027a725a",
  "name": "Laura Roslin",
  "tv_show": "Battlestar Galactica"
}
```

Here is example data for the `posts` table:

```
{
    "id": "543ad9c8-1744-4001-bb5e-450b2565d02c",
    "title": "Decommissioning speech",
    "content": "The Cylon War is long over..."
}
```

And here is example data for the `authors_posts` table:

```
{
    "author_id": "7644aaf2-9928-4231-aa68-4e65e31bf219",
    "post_id": "543ad9c8-1744-4001-bb5e-450b2565d02c"
}
{
    "author_id": "064058b6-cea9-4117-b92d-c911027a725a",
    "post_id": "543ad9c8-1744-4001-bb5e-450b2565d02c"
}
```

In a many to many relation, we can multiple `eq_join` commands to join the data from all three tables:

```
r.table("authors_posts").eq_join("author_id", r.table("authors")).zip().
  eq_join("post_id", r.table("posts")).zip().run()
```

The result of this query is a stream of documents that includes every post written by every author in our database:

```
{
    "tv_show": "Battlestar Galactica",
    "title": "Decommissioning speech",
    "post_id": "543ad9c8-1744-4001-bb5e-450b2565d02c",
    "name": "William Adama",
    "id": "543ad9c8-1744-4001-bb5e-450b2565d02c",
    "content": "The Cylon War is long over...",
    "author_id": "7644aaf2-9928-4231-aa68-4e65e31bf219"
}
{
    "tv_show": "Battlestar Galactica",
    "title": "Decommissioning speech",
    "post_id": "543ad9c8-1744-4001-bb5e-450b2565d02c",
    "name": "Laura Roslin",
    "id": "543ad9c8-1744-4001-bb5e-450b2565d02c",
    "content": "The Cylon War is long over...",
    "author_id": "064058b6-cea9-4117-b92d-c911027a725a"
}
```

Resolving field name conflicts

If you use the `zip` command after `join`, the document from the right table will be merged into the left one.

ReQL will eventually provide a better function to manipulate the output of a JOIN query.
See Github issue #325 to track progress.

Consider the following query:

```
r.table("employees").eq_join("company_id", r.table("companies"))
```

Suppose its output is as follows:

```
{
    # Employee
    "left": {
        "id": "543ad9c8-1744-4001-bb5e-450b2565d02c",
        "name": "Jean-Luc Picard",
        "company_id": "064058b6-cea9-4117-b92d-c911027a725a",
        "rank": "captain"
```

```
    },
    # Company
    "right": {
        "id": "064058b6-cea9-4117-b92d-c911027a725a",
        "company": "Starfleet",
        "type": "paramilitary"
    }
}
```

The conflicting field is `id`. If you directly use the `zip` command, the `id` field of the result will be the one from the company. There are three ways to resolve potential field conflicts.

Removing the conflicting fields

Suppose that you want to keep the `id` field of the employee, but not the one of the company. You can do it by removing the field `right.id`, then calling the `zip` command.

```
r.table("employees").eq_join("company_id", r.table("companies"))
    .without({"right": {"id": True}}) # Remove the field right.id
    .zip()
    .run()
```

Renaming the fields

If you need to keep both fields, you can rename them with `map` and `without` before using the `zip` command.

```
r.table("employees").eq_join("company_id", r.table("companies"))
    # Copy the field right.id into right.c_id
    .map( r.row.merge({
        "right": {
            "c_id": r.row["right"]["id"]
        }
    }))
    # Remove the field right.id
    .without({"right": {"id": True}})
    .zip()
    .run()
```

Manually merge the left and right fields

You can manually merge the `left` and `right` fields without using the `zip` command. Suppose you want to keep the name of the employee and the name of his company. You can do:

```
r.table("employees").eq_join("company_id", r.table("companies"))
    .map({
        "name": r.row["left"]["name"],
        "company": r.row["right"]["company"]
    }).run()
```

Read more

To learn more, read about data modeling in RethinkDB. For detailed information, take a look at the API documentation for the join commands:

- eq_join
- inner_join
- outer_join
- zip

Secondary indexes are data structures that improve the speed of many read queries at the slight cost of increased storage space and decreased write performance.

RethinkDB supports different types of secondary indexes:

- **Simple indexes** based on the value of a single field.
- **Compound indexes** based on multiple fields.
- **Multi indexes** based on arrays of values.
- Indexes based on **arbitrary expressions**.

Using indexes

Simple indexes

Use simple indexes to efficiently retrieve and order documents by the value of a single field.

Creation

```
# Create a secondary index on the last_name attribute
r.table("users").index_create("last_name").run(conn)

# Wait for the index to be ready to use
r.table("users").index_wait("last_name").run(conn)
```

Querying

```
# Get all users whose last name is "Smith"
r.table("users").get_all("Smith", index="last_name").run(conn)

# Get all users whose last names are "Smith" or "Lewis"
r.table("users").get_all("Smith", "Lewis", index="last_name").run(conn)

# Get all users whose last names are between "Smith" and "Wade"
r.table("users").between("Smith", "Wade", index="last_name").run(conn)

# Efficiently order users by last name using an index
r.table("users").order_by(index="last_name").run(conn)

# For each blog post, return the post and its author using the last_name index
r.table("posts").eq_join("author_last_name", r.table("users"), index="last_name") \
    .zip().run(conn)
```

Want to learn more about joins in RethinkDB? See how to use joins to query *one to many* and *many to many* relations.

Compound indexes

Use compound indexes to efficiently retrieve documents by multiple fields.

Creation

```
# Create a compound secondary index based on the first_name and last_name attributes
r.table("users").index_create("full_name", [r.row["first_name"], r.row["last_name"]]) \
    .run(conn)

# Wait for the index to be ready to use
r.table("users").index_wait("full_name").run(conn)
```

Querying

```
# Get all users whose full name is John Smith.
r.table("users").get_all(["John", "Smith"], index="full_name").run(conn)

# Get all users whose full name is between "John Smith" and "Wade Welles"
r.table("users").between(["John", "Smith"], ["Wade, Welles"], index="full_name") \
    .run(conn)

# Efficiently order users by first name and last name using an index
r.table("users").order_by(index="full_name").run(conn)

# For each blog post, return the post and its author using the full_name index
r.table("posts").eq_join("author_full_name", r.table("users"), index="full_name") \
    .run(conn)
```

Multi indexes

To index a single document multiple times by different values use a multi index. For example, you can index a blog post with an array of tags by each tag.

Creation

Suppose each post has a field `tags` that maps to an array of tags. The schema of the table `posts` would be something like:

```
{
    "title": "...",
    "content": "...",
    "tags": [ <tag1>, <tag2>, ... ]
}
```

```
# Create the multi index based on the field tags
r.table("posts").index_create("tags", multi=True)

# Wait for the index to be ready to use
r.table("posts").index_wait("tags").run(conn)
```

Querying

```
# Get all posts with the tag "travel" (where the field tags contains "travel")
r.table("posts").get_all("travel", index="tags").run(conn)
```

```
# For each tag, return the tag and the posts that have such tag
r.table("tags").eq_join("tag", r.table("posts"), index="tags").run(conn)
```

Indexes on arbitrary ReQL expressions

You can create an index on an arbitrary expressions by passing an anonymous function to `index_create`.

```
# A different way to do a compound index
r.table("users").index_create("full_name2", lambda user:
    r.add(user["last_name"], "_", user["first_name"])).run(conn)
```

The function you give to `index_create` must be deterministic. In practice this means that that you cannot use a function that contains a sub-query or the `r.js` command.

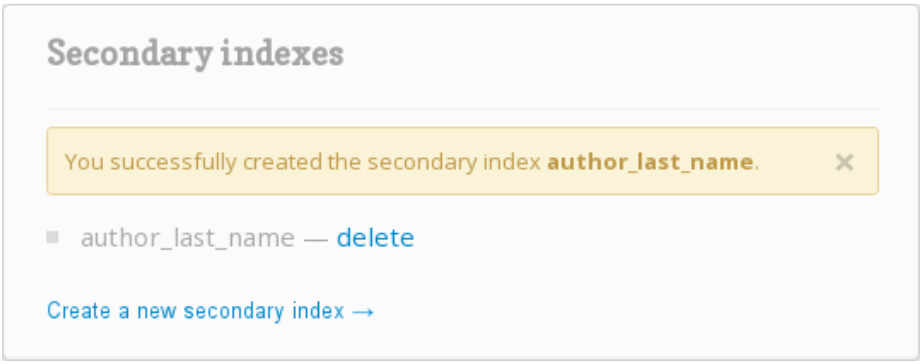
Administrative operations

With ReQL

```
# list indexes on table "users"
r.table("users").index_list().run(conn)

# drop index "last_name" on table "users"
r.table("users").index_drop("last_name").run(conn)

# return the status of all indexes
r.table("users").index_status().run(conn)

# return the status of the index "last_name"
r.table("users").index_status("last_name").run(conn)

# return only when the index "last_name" is ready
r.table("users").index_wait("last_name").run(conn)
```

Manipulating indexes with the web UI

The web UI supports creation and deletion of simple secondary indexes. In the table list, click on the table `users`. You can manipulate indexes through the secondary index panel in the table view.



Limitations

Secondary indexes have the following limitations:

- RethinkDB does not currently have an optimizer. As an example, the following query will not automatically use an index:

  ```
  # This query does not use a secondary index! Use get_all instead.
  r.table("users").filter( {"last_name": "Smith" }).run(conn)
  ```

  You have to explicitly use the `get_all` command to take advantage of secondary indexes.

  ```
  # This query does not use a secondary index! Use get_all instead.
  r.table("users").get_all("Smith", index="last_name").run(conn)
  ```

- You cannot chain multiple `get_all` commands. Use a compound index to efficiently retrieve documents by multiple fields.
- Currently, compound indexes cannot be queried by a prefix.
  See Github issue #955 to track progress.
- RethinkDB does not provide a geospatial index yet.
  See Github issue #1158 to track progress.
- RethinkDB does not support unique secondary indexes even for non-sharded tables.

More

Browse the API reference to learn more about secondary index commands:

- Manipulating indexe: index_create, index_drop and index_list
- Using indexes: get_all, between, eq_join and order_by

RethinkDB has native suport for millisecond-precision times with time zones. Some highlights:

- **Times are integrated with the official drivers**, which will automatically convert to and from your language's native time type.
- **Queries are timezone-aware**, so you can ask questions like "Did this event happen on a Monday in the time zone where it was recorded?"
- **Times work as indexes**, so you can efficiently retrieve events based on when they occured.
- **Time operations are pure ReQL**, which means that even complicated date-time queries can be distributed efficiently across the cluster.

A quick example

Note: Examples below are in Ruby. Head to the API reference to see the commands in other languages.

First, let's create a table and insert some events. We'll insert the first event using a native time object, and the second with the `epoch_time` constructor:

```
> r.table_create('ev').run(conn)
{"created"=>1}
> r.table('events').insert(
    [{'id' => 0, 'timestamp' => Time.now},
     {'id' => 1, 'timestamp' => r.epoch_time(1376436769.923)}]
  ).run(conn)
{"unchanged"=>0, "skipped"=>0, "replaced"=>0, "inserted"=>2, "errors"=>0, "deleted"=>0}
```

Now, let's get those back:

```
> r.table('events').run(conn).to_a
[{"timestamp"=>2013-08-13 16:32:48 -0700, "id"=>0},
 {"timestamp"=>2013-08-13 23:32:49 +0000, "id"=>1}]
```

You'll notice that both times we inserted are returned as native Ruby `Time` objects. They're in different time zones because `Time.now` creates a time object in the local time zone, but `r.epoch_time` creates a UTC time (it doesn't know or care what time zone the client is in). If we had instead inserted `Time.now.utc`, they'd both be in UTC when we retrieved them.

We can now filter based on these times:

```
> r.table('events').filter{|row| row['timestamp'].hours() > 20}.run(conn)
[{"timestamp"=>2013-08-13 23:32:49 +0000, "id"=>1}]
> r.table('events').filter{|row|
    row['timestamp'].in_timezone('-02:00').hours() > 20
  }.run(conn)
[{"timestamp"=>2013-08-13 16:32:48 -0700, "id"=>0},
 {"timestamp"=>2013-08-13 23:32:49 +0000, "id"=>1}]
```

Or create a secondary index on them:

```
> r.table('events').index_create('timestamp').run(conn)
{"created"=>1}
> r.table('events').between(r.epoch_time(1376436769.913),
                            r.epoch_time(1376436769.933),
                            :index => 'timestamp').run(conn)
[{"timestamp"=>2013-08-13 23:32:49 +0000, "id"=>1}]
```

Technical details

Times are stored on the server as seconds since epoch (UTC) with millisecond precision plus a time zone. Currently the only available time zones are minute-precision time offsets from UTC, but we may add support for DST-aware time zones in the future. Time zones are strings as specified by ISO 8601.

Times are considered equal if their seconds since epoch (UTC) are equal, **regardless of what time zone they're in**. This is true for both comparisons and indexed operations.

Most date operations are only defined on years in the range `[1400, 10000]` (but note that times in the year `10000` cannot be printed as ISO 8601 dates).

Leap-seconds aren't well-supported right now: `2012-06-30T23:59:60` and `2012-07-01T00:00:00` parse to the same time.

Inserting times

You can insert times by simply passing a native time object. If the local time object contains a time zone, the inserted time will have that time zone; if it

doesn't, the inserted time will be in UTC (check this if you're using a third-party driver).

```
> r.table('events').insert({'id' => 2, 'timestamp' => Time.now}).run(conn)
{"unchanged"=>0, "skipped"=>0, "replaced"=>0, "inserted"=>1, "errors"=>0, "deleted"=>0}
```

You can also use `r.now` (which the server interprets as the time the query was received in UTC), or construct a time using `r.time`, `r.epoch_time`, or `r.iso8601`.

```
> r.now().to_iso8601().run(conn)
"2013-08-09T18:53:15.012+00:00"
> r.time(2013, r.august, 9, 18, 53, 15.012, '-07:00').to_iso8601().run(conn)
"2013-08-09T18:53:15.012-07:00"
> r.epoch_time(1376074395.012).to_iso8601().run(conn)
"2013-08-09T18:53:15.012+00:00"
> r.iso8601("2013-08-09T18:53:15.012-07:00").to_iso8601().run(conn)
"2013-08-09T18:53:15.012-07:00"
```

Times may be used as the primary key for a table. Two times are considered equal if they have the same number of milliseconds since epoch (UTC), regardless of time zone.

```
> r.table('t').insert({'id' => r.iso8601("2013-08-09T11:58:00.1111-07:00")}).run(conn)
{"unchanged"=>0, "skipped"=>0, "replaced"=>0, "inserted"=>1, "errors"=>0, "deleted"=>0}
> r.table('t').insert({'id' => r.iso8601("2013-08-09T10:58:00.1112-08:00")}).run(conn)
{"unchanged"=>0, "skipped"=>0, "replaced"=>0, "inserted"=>0,
 "first_error"=>"Duplicate primary key.", "errors"=>1, "deleted"=>0}
```

You may also insert a time by inserting a literal pseudotype object. This is useful if, for instance, you exported a row using `:time_format => 'raw'` (see **Retrieving Times** below).

Note: Avoid using keys matching the regular expression `^\$reql_.+\$$` in your objects. RethinkDB considers those to be reserved keywords.

```
> r.expr({'$reql_type$' => 'TIME',
          'epoch_time' => 1376075362.662,
          'timezone' => '+00:00'}).to_iso8601().run(conn)
"2013-08-09T19:09:22.662+00:00"
```

Retrieving times

By default, times are converted into native time objects when they are retrieved from the server. This may be overridden by passing the optarg `time_format` to `run`. The only options right now are `native`, the default, and `raw`. See the API reference if you are uncertain how to pass an optarg in a non-Ruby languge.

Warning: Some languages, like Javascript, don't have an easy way to represent a time in an arbitrary time zone. In this case, time zone information will be discarded when converting to a native time object.

```
> r.now().run(conn)
2013-08-09 19:09:11 UTC
> r.now().in_timezone('-07:00').run(conn)
2013-08-09 12:14:56 -0700
> r.now().run(conn, :time_format => 'raw')
{'timezone'=>'+00:00', "epoch_time"=>1376075362.662, "$reql_type$"=>"TIME"}
> r.now().in_timezone('-07:00').run(conn, :time_format => 'raw')
{"timezone"=>"-07:00", "epoch_time"=>1376075702.485, "$reql_type$"=>"TIME"}
```

You can also transform a time object on the server using either `to_epoch_time` or `to_iso8601`.

```
> r.now().to_epoch_time().run(conn)
1376075986.574
> r.now().to_iso8601().run(conn)
"2013-08-09T19:19:46.574+00:00"
```

Working with times

There are only three useful things you can do with a time: modify it, compare it to another time, or retrieve a portion of it.

Modifying times

You can put a time into a new time zone:

```
> r.expr(Time.now).to_iso8601().run(conn)
"2013-08-09T12:48:59.103-07:00"
> r.expr(Time.now).in_timezone('-06:00').to_iso8601().run(conn)
"2013-08-09T13:49:15.503-06:00"
```

You can also add or subtract a duration (in seconds):

```
> (r.epoch_time(123.456) + 123.456).to_epoch_time().run(conn)
246.912
```

If you subtract two times, you get a duration:

```
> (r.epoch_time(246.912) - r.epoch_time(123.456)).run(conn)
123.456
```

Comparing times

All of the normal comparison operators are defined on times:

```
> (r.epoch_time(1376081287.982) < Time.now).run(conn)
true
```

Times are only compared with millisecond precision:

```
> r.epoch_time(1376081287.9821).eq(r.epoch_time(1376081287.9822)).run(conn)
true
```

There's also the `during` command which is convenient for checking whether a time is in a particular range of times. See more at the API reference.

Retrieving portions of times

If you have a time, you can retrieve a particular portion (like the month, or the hours) relative to the current time zone. (See the full list at the API reference.)

```
> r.expr(Time.now).run(conn)
2013-08-09 13:53:00 -0700
> r.expr(Time.now).month().run(conn)
8
> r.expr(Time.now).hours().run(conn)
13
> r.expr(Time.now).in_timezone('-06:00').hours().run(conn)
14
```

We use the ISO 8601 definition of a week, which starts with Monday, represented as 1.

```
> r.expr(Time.now).day_of_week().run(conn)
5 # Friday
```

We define `r.monday...r.sunday` and `r.january...r.december` for convenience:

```
> r.expr(Time.now).day_of_week().eq(r.friday).run(conn)
true
```

We also let you slice the time into the date and the current time of day (a time and a duration, respectively):

```
> r.now().to_epoch_time().run(conn)
1376351312.744
> r.now().date().to_epoch_time().run(conn)
1376265600
> r.now().time_of_day().run(conn)
85712.744
```

Putting it all together

By combining these operations, you can write surprisingly useful queries in pure ReQL. For example, let's say you have a table of sales your company has made, and you want to figure out how much of the gross comes from people who were working overtime:

```
r.table('sales').filter {|sale|
  # Weekends are overtime.
  sale['time'].day_of_week().eq(r.saturday) |
  sale['time'].day_of_week().eq(r.sunday) |
  # Weekdays outside 9-5 are overtime.
  (sale['time'].hours() < 9) |
  (sale['time'].hours() >= 17)
}.map{|sale| sale['dollars']}.reduce{|a,b| a+b}.run(conn)
```

If your timestamps are stored with time zones, this query will work even if you have sales from different offices in different countries (assuming they all work 9-5 local time).

Since this query is pure ReQL, the entire query will be distributed efficiently over the cluster without any computation being done on the client.

Further, because it's ReQL, the query's individual pieces are easily composable. If you decide you want those numbers on a per-month basis, you can easily switch TO create to a `grouped_map_reduce` query:

```
r.table('sales').filter {|sale|
  # Weekends are overtime.
  sale['time'].day_of_week().eq(r.saturday) |
  sale['time'].day_of_week().eq(r.sunday) |
  # Weekdays outside 9-5 are overtime.
  (sale['time'].hours() < 9) |
  (sale['time'].hours() >= 17)
}.grouped_map_reduce(
  lambda {|sale| sale['time'].month()},
  lambda {|sale| sale['dollars']},
  lambda {|a,b| a+b}
).run(conn)
```

Working with native time objects

Python

RethinkDB accepts Python `datetime` objects:

```
from datetime import datetime
```

The Python driver will throw an error if you pass it a `datetime` without a time zone. (RethinkDB only stores times with time zones.) If you try to run:

```
r.expr(datetime.now()).run(conn)
```

You will get the following error:

```
RqlDriverError: Cannot convert datetime to ReQL time object
without timezone information. You can add timezone information with
the third party module "pytz" or by constructing ReQL compatible
timezone values with r.make_timezone("[+-]HH:MM"). Alternatively,
use one of ReQL's bultin time constructors, r.now, r.time, or r.iso8601.
```

To pass a valid time object to the Python driver, you can:

- Use `r.make_timezone`

  ```
  r.expr(datetime.now(r.make_timezone('-07:00'))).run(conn)
  ```

- Use the `pytz` module

  ```
  from pytz import timezone
  r.expr(datetime.now(timezone('US/Pacific'))).run(conn)
  ```

JavaScript

RethinkDB accepts JavaScript `Date` objects:

```
r.expr(new Date()).run(conn, callback)
```

In JavaScript, `Date` objects store the epoch time, but not the time zone. As a result, the JavaScript driver will not send any time zones to RethinkDB, and will discard the time zones on any time objects it retrieves from RethinkDB (the epoch time will still be correct). If you need to access the time zone of a time stored in RethinKDB, you can retrieve the raw time object like so:

```
r.expr(new Date()).run({connection: conn, timeFormat: "raw"}, callback)
```

Ruby

RethinkDB accepts Ruby `Time` objects. (Note that we only support Ruby 1.9+.)

```
r.expr(Time.now).run(conn)
```

Note: Examples below are in Python. Head to the API reference to see the commands in other languages.

Terminology

SQL and RethinkDB share very similar terminology. Below is a table of terms and concepts in the two systems.

SQL	RethinkDB
database	database
table	table
row	document
column	field
table joins	table joins
primary key	primary key (by default `id`)
index	index

INSERT

This is a list of queries for inserting data into a database.

SQL	ReQL

`INSERT INTO users(user_id,`
`  age,`
`  name)`
`VALUES ("f62255a8259f",`
`  30,`
`  Peter)`
	`r.table("users").insert({`
	`  "user_id": "f62255a8259f",`
	`  "age": 30,`
	`  "name": "Peter"`
	`})`

SELECT

This is a list of queries for selecting data out of a database.

SQL	ReQL
`SELECT * FROM users`	`r.table("users")`
`SELECT user_id, name FROM users`	`r.table("users")` `  .pluck("user_id", "name")`

SQL	ReQL

`SELECT * FROM users WHERE name = "Peter"`	`r.table("users").filter({ "name": "Peter" })`
	
	An alternative is to use the implicit variable `r.row` (the currently visited document):
	
	`r.table("users").filter( r.row["name"] == "Peter" )`
	
	Another alternative is to use an anonymous function:
	
	`r.table("users").filter(lambda doc: doc["name"] == "Peter" )`
	
	If you have a secondary index built on the field `name`, you can run a more efficient query:
	
	`r.table("users") .get_all("Peter", index="name")`

`SELECT user_id, name FROM users WHERE name = "Peter"`	`r.table("users").filter({ "name": "Peter" }).pluck("user_id", "name")`

`SELECT * FROM users WHERE name = "Peter" AND age = 30`	`r.table("users").filter({ "name": "Peter", "age": 30 })`
	
	`r.table("users").filter( (r.row["name"] == "Peter") & (r.row["age"] == 30) )`

`SELECT * FROM users WHERE age > 30`	`r.table("users").filter( r.row["age"] > 30 )`
`SELECT * FROM users WHERE name LIKE "P%"`	`r.table("users").filter( r.row['name'].match("^P")} )`

SQL	ReQL
`SELECT * FROM users WHERE name LIKE "%er"`	`r.table("users").filter( r.row['name'].match("er$")} )`
`SELECT * FROM users ORDER BY name ASC`	`r.table("users").order_by("name")`
`SELECT * FROM users ORDER BY name DESC`	`r.table("users").order_by( r.desc("name") )`
`SELECT user_id FROM users WHERE name = "Peter" ORDER BY name DESC`	`r.table("users").filter({ "name": "Peter" }).order_by( r.desc("name") ).pluck("user_id")`
`SELECT * FROM users LIMIT 5`	`r.table("users").limit(5)`
`SELECT * FROM users LIMIT 5 SKIP 10`	`r.table("users").skip(10).limit(5)`

`SELECT * FROM users WHERE name IN {'Peter', 'John'}`	`r.table("users").filter( r.expr(["Peter", "John"]) .contains(r.row["name"]) )`
	
	If you have a secondary index built on the field `name`, you can run a more efficient query:
	
	`r.table("users") .get_all("Peter", "John", index="name")`

`SELECT * FROM users WHERE name NOT IN {'Peter', 'John'}`	`r.table("users").filter( r.expr(["Peter", "John"]) .contains(r.row["name"]) .not() )`
`SELECT COUNT(*) FROM users`	`r.table("users").count()`
`SELECT COUNT(name) FROM users`	`r.table("users").filter(lambda doc: doc.has_fields("name") ).count()`
`SELECT COUNT(name) FROM users WHERE age > 18`	`r.table("users").filter( (r.row.has_fields("name")) & (r.row["age"] > 18) ).count()`

SQL	ReQL

`SELECT AVG("age")`
`FROM users`
	`(r.table("users")`
	`  .map(lambda user:`
	`  user["age"]`
	`  ).reduce(lambda left, right:`
	`  left+right`
	`  )/r.table("users").count())`

`SELECT MAX("age")`
`  FROM users`
	`r.table("users")`
	`  .map(lambda user:`
	`  user["age"]`
	`  ).reduce(lambda left, right:`
	`  r.branch(`
	`  left>right,`
	`  left,`
	`  right`
	`  )`
	`  )`

`SELECT MIN("age")`
`  FROM users`
	`r.table("users")`
	`  .map(lambda user:`
	`  user["age"]`
	`  ).reduce(lambda left, right:`
	`  r.branch(`
	`  left>right,`
	`  right,`
	`  left`
	`  )`
	`  )`

`SELECT SUM("num_posts")`
`  FROM users`
	`r.table("users")`
	`  .map(lambda user:`
	`  user["num_posts"]`
	`  ).reduce(lambda left, right:`
	`  left+right`
	`  )`

`SELECT DISTINCT(name) FROM users`	`r.table("users").pluck("name").distinct()`

SQL	ReQL

`SELECT *`
`  FROM users`
`  WHERE age BETWEEN 18 AND 65;`
	`r.table("users").filter(`
	`  (r.row["age"] >= 18)`
	`  & (r.row["age"] >= 65)`
	
	If you have a secondary index built on the field `age`, you can run a more efficient query:
	
	`r.table("users")`
	`  .between(18, 65, index="age")`

`SELECT name, 'is_adult' = CASE`
`  WHEN age>18 THEN 'yes'`
`  ELSE 'no'`
`  END`
`FROM users`
	`r.table("users").map({`
	`  "name": r.row["name"],`
	`  "is_adult": r.branch(`
	`  r.row["age"] > 18,`
	`  "yes",`
	`  "no"`
	`  )`
	`})`
	
	`r.table("users").map(lambda user:`
	`  {`
	`  "name": user["name"],`
	`  "is_adult": r.branch(`
	`  user["age"] > 18`
	`  "yes",`
	`  "no"`
	`  )`
	`  }`
	`)`

`SELECT *`
`  FROM posts`
`  WHERE EXISTS`
`  (SELECT * FROM users`
`  WHERE posts.author_id`
`  = users.id)`
	`r.table("posts")`
	`  .filter(lambda post:`
	`  r.table("users")`
	`  .filter( lambda user:`
	`  user.id == post.author_id`
	`  ).count() > 0`
	`  )`

UPDATE

This is a list of commands for updating data in the database.

SQL	ReQL

`UPDATE users`
`    SET age = 18`
`    WHERE age < 18`
	`r.table("users").filter(`
	`    r.row["age"] < 18`
	`).update({`
	`    "age": 18`
	`})`
	
	`r.table("users").filter(lambda doc:`
	`    doc["age"] < 18`
	`).update({`
	`    "age": 18`
	`})`

`UPDATE users`
`    SET age = age+1`
	`r.table("users").update(`
	`    { "age": r.row["age"]+1 }`
	`)`
	
	`r.table("users").update(lambda doc:`
	`    { "age": doc["age"]+1 }`
	`)`

DELETE

This is a list of queries for deleting data from the database.

SQL	ReQL
`DELETE FROM users`	`r.table("users").delete()`

`DELETE FROM users`
`WHERE age < 18`
	`r.table("users")`
	`    .filter( r.row["age"] < 18)`
	`    .delete()`
	
	`r.table("users")`
	`    .filter(lambda doc:`
	`    doc["age"] < 18`
	`    ).delete()`

JOINS

This is a list of queries for performing joins between multiple tables.

SQL	ReQL

`SELECT *`
`FROM posts`
`JOIN users`
`ON posts.user_id = users.id`
	`r.table("posts").inner_join(`
	`  r.table("users"),`
	`  lambda post, user:`
	`  post["user_id"] == user["id"]`
	`).zip()`
	
	Note: `zip()` will merge the user in the post, overwriting fields in case of conflict.
	If you have an index (primary key or secondary index) built on the field of the right table, you can perform a more efficient join with eq_join
	
	`r.table("posts").eq_join(`
	`  "id",`
	`  r.table("users"),`
	`  index="id"`
	`).zip()`

`SELECT posts.id AS post_id,`
`  user.name,`
`  users.id AS user_id`
`FROM posts`
`JOIN users`
`  ON posts.user_id = users.id`
 	
`SELECT posts.id AS post_id,`
`  user.name,`
`  users.id AS user_id`
`FROM posts`
`INNER JOIN users`
`  ON posts.user_id = users.id`
	`r.table("posts").inner_join(`
	`  r.table("users"),`
	`  lambda post, user:`
	`  post["user_id"] == user["id"]`
	`).map({`
	`  "post_id": r.row["left"]["id"],`
	`  "user_id": r.row["right"]["id"],`
	`  "name": r.row["right"]["name"]`
	`})`

SQL	ReQL

`SELECT *`
`  FROM posts`
`  RIGHT JOIN users`
`  ON posts.user_id = users.id`
	`r.table("posts").outer_join(`
	`  r.table("users"),`
	`  lambda post, user:`
	`  post["user_id"] == user["id"]`
	`).zip()`

`SELECT *`
`  FROM posts`
`  RIGHT OUTER JOIN users`
`  ON posts.user_id = users.id`
	Note: You can perform efficient `OUTER JOIN` operations with the concat_map command. The `eq_join` command will eventually be able to behave like an OUTER JOIn, see this github issue.
	
	`r.table("posts").concat_map(lambda post:`
	`  r.table("users")`
	`  .get_all(post["id"],index="id")`
	`  .do( lambda results:`
	`  r.branch(`
	`  results.count() == 0,`
	`  [{"left": post}],`
	`  results.map( lambda user:`
	`  {`
	`  "left": post`
	`  "right": user`
	`  }`
	`  )`
	`  )`
	`  )`
	`).zip()`

SQL	ReQL

`SELECT *`
`  FROM posts`
`  LEFT JOIN users`
`  ON posts.user_id = users.id`
	`r.table("users").outer_join(`
	`  r.table("posts"),`
	`  lambda user, post:`
	`  post.user_id == user.id`
	`).zip()`

`SELECT *`
`  FROM posts`
`  LEFT OUTER JOIN users`
`  ON posts.user_id = users.id`
	`r.table("users").concat_map(lambda user:`
	`  r.table("posts")`
	`  .get_all(user["id"],index="id")`
	`  .do( lambda results:`
	`  r.branch(`
	`  results.count() == 0,`
	`  [{"left": user}],`
	`  results.map( lambda post:`
	`  {`
	`  "left": user`
	`  "right": post`
	`  }`
	`  )`
	`  )`
	`  )`
	`).zip()`

AGGREGATIONS

This is a list of queries for performing data aggregation.

SQL	ReQL

`SELECT category`
`  FROM posts`
`  GROUP BY category`
	`r.table("posts").map(`
	`  r.row["category"]`
	`).distinct()`
	
	`r.table("users").map(lambda user`
	`  user["category"]`
	`).distinct()`

SQL	ReQL

`SELECT category,`
`  SUM('num_comments')`
`  AS total`
`FROM posts`
`GROUP BY category`
	`r.table("posts")`
	`  .group_by('category',`
	`  r.sum('num_comments')`
	`  ).map({`
	`  "category": r.row["group"],`
	`  "total": r.row["reduction"]`
	`})`

`SELECT category,`
`  status,`
`  SUM('num_comments')`
`  AS total`
`FROM posts`
`GROUP BY category, status`
	`r.table("posts")`
	`  .grouped_map_reduce(`
	`  lambda post:`
	`  [post["category"],`
	`  post["status"]],`
	`  lambda post:`
	`  post["num_comments"],`
	`  lambda left, right:`
	`  left + right`
	`  ).map({`
	`  "category": r.row["group"][0],`
	`  "status": r.row["group"][1],`
	`  "total": r.row["reduction"]`
	`})`

`SELECT id,`
`  SUM(num_comments)`
`  AS "total"`
`FROM posts`
`GROUP BY category, num_comments`
`  HAVING num_comments > 7`
	`r.table("posts")`
	`  .filter( r.row["num_comments"]>7`
	`  .grouped_map_reduce(`
	`  lambda post:`
	`  post["category"],`
	`  lambda post:`
	`  post["num_comments"],`
	`  lambda left, right:`
	`  left + right`
	`  ).map({`
	`  "category": r.row["group"],`
	`  "total": r.row["reduction"]`
	`})`

TABLE and DATABASE manipulation

This is a list of queries for creating and dropping tables and databases.

SQL	ReQL
`CREATE DATABASE my_database;`	`r.db_create('my_database')`
`DROP DATABASE my_database;`	`r.db_drop('my_database')`
`CREATE TABLE users (id INT IDENTITY(1,1) PRIMARY KEY, name VARCHAR(50), age INT);`	`r.table_create('users', primary_key="id")` *Note:* RethinkDB is a NoSQL database and does not enforce schemas. *Note*[2]*:* The default primary key is `id`
`TRUNCATE TABLE users;`	`r.table("users").delete()`
`DROP TABLE users;`	`r.table_drop("users")`

Read More

Browse the following resources to learn more about ReQL:

- Lambda functions in RethinkDB
- Introduction to map/reduce
- Introduction to Joins
- API Reference

RethinkDB provides two ways to administer your cluster:

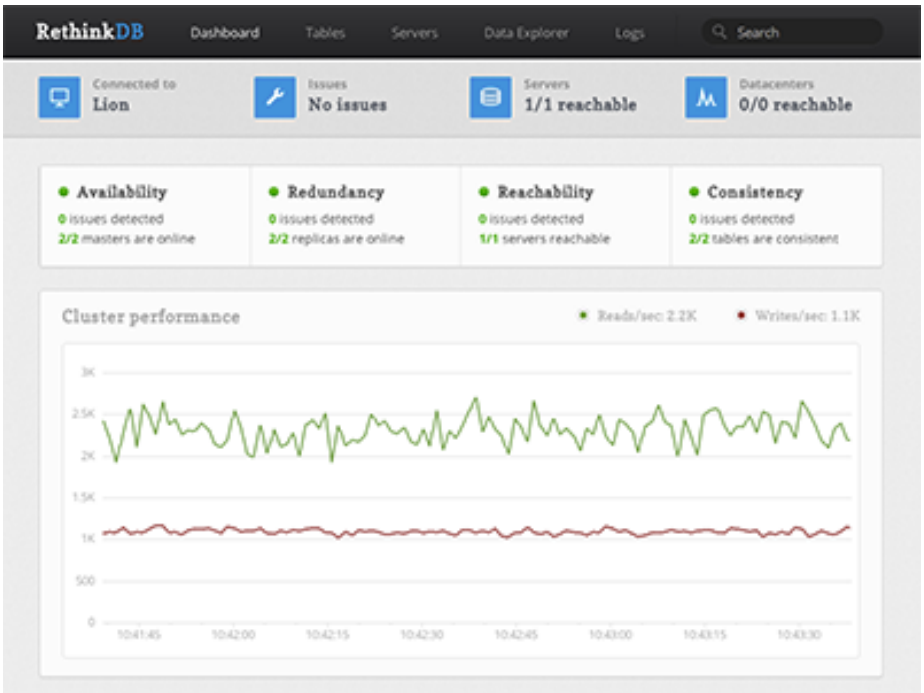
- The **web interface** lets you visualize your cluster, shard, replicate, and run ReQL queries from your browser.
- The **command-line interface** lets you administer your cluster via the command line and is particularly useful for scripting.

Accessing the web interface

The default port for the RethinkDB web interface is `8080`. Point your browser to `http://HOST:8080` to access the web interface.

Having trouble connecting to the web interface? By default, RethinkDB binds itself to `localhost` for security reasons. You can start RethinkDB with the `--bind all` parameter to access it from another machine.

Here's what the web interface looks like:



Using the command-line interface

To use the RethinkDB command-line interface, you can run this command:

```
rethinkdb admin --join <host>:<port>
```

where:

- `host` is the IP address of any machine of your RethinkDB cluster
- `port` is the port for intracluster connections (by default `29015`)

To get help on using the CLI, run the command:

```
rethinkdb admin help
```

You can also get help on a specific command. For example, to get help with the command `rethinkdb admin create`, use the following:

```
rethinkdb admin help create
```

RethinkDB allows you to shard and replicate your cluster on a per-table basis.

Sharding a table is as easy as typing the number of shards you'd like in the web admin and clicking 'Rebalance'. RethinkDB rebalances the table, creates copies in the cluster, and moves necessary data around behind the scenes without any additional work from the user. Similarly, to change the number of replicas you simply set the number in the web UI and hit 'Save'.

Sharding and replication settings can also be controlled from a powerful command-line interface: `rethinkdb admin`.

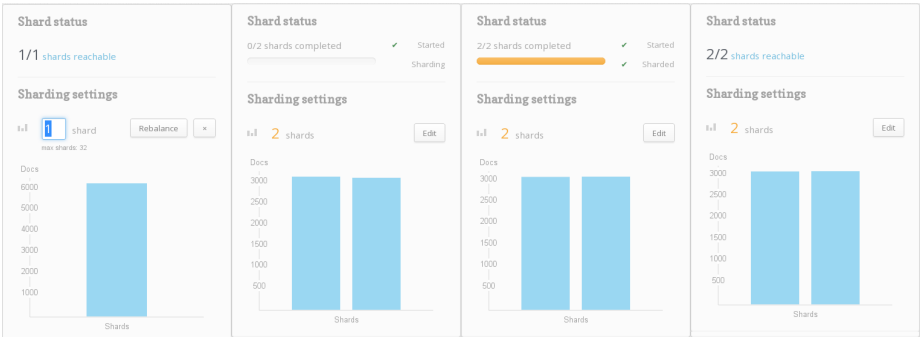
Sharding

Note: Currently, RethinkDB implements range shards, but will eventually be switching to hash shards (follow Github issue #364 to track progress).

Sharding via the web interface

When using the web UI, simply specify the number of shards you want, and based on the data available RethinkDB will determine the best split points to maintain balanced shards. To shard your data:

- Go to the table view (*Tables* → *table name*).
- Click on the *Edit* button under shard settings.
- Set the number of shards you would like.
- Click on the *Rebalance* button.



Sharding via the command-line interface

Connect to your cluster via the command-line interface:

```
rethinkdb admin --join <host>:<port>
```

Shards are managed by specifying a set of split points. A split point is the primary key upon which the table will be sharded. By adding and removing split points, you can add or remove shards to a table.

- Find the UUID of the table you want to shard using `ls`.
- To list existing shards, use `ls <table_name>` or `ls <table_uuid>`.
- Add a new split point, creating a new shard, using `split shard <table_uuid> <split_point>`.
- Remove a split point, removing an existing shard, using `merge shard <table_uuid> <split_point>`.

Replication

There are two parameters that can be set when dealing with replicas in RethinkDB:

- The number of *replicas*: the number of copies of your data.
- The number of *acknowledgements* (also referred to as *acks*): the number of confirmations required before a write is acknowledged.

These two parameters can be specified for each table on a per-datacenter basis or for the whole cluster (which includes machines that are not assigned to any datacenter).

The primary constraints on these parameters are:

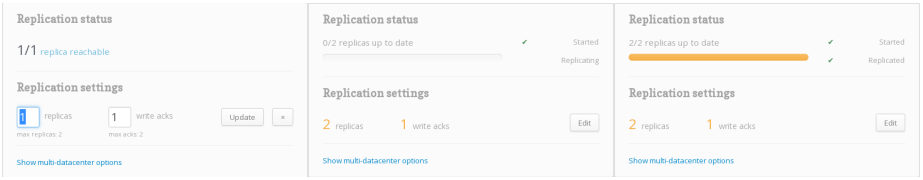
- You cannot require more replicas than you have machines available in your cluster.
- The number of *acks* has to be less than or equal to the number of *replicas*, or no write will ever be acknowledge.

Replication via the web interface

To replicate your data through the web UI:

- Go to the table view (*Tables* > *table name*).
- Click on the *Edit* button under replication settings.
- Set the number of *replicas* and *acks* you would like.

- Click on the *Update* button.



Replication via the command-line interface

Connect to your cluster via the command-line interface:

```
rethinkdb admin --join <host>:<port>
```

You can change the number of *replicas* and *acks* using the following commands:

```
set acks <table> <num_acks> [<datacenter>]
set replicas <table> <num_replicas> [<datacenter>]
```

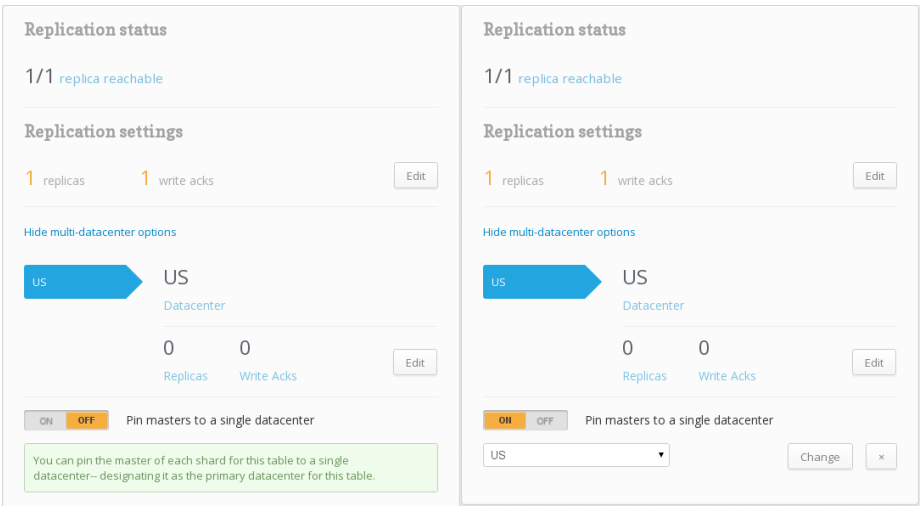
Pinning masters to datacenters

Because RethinkDB is immediately consistent, each shard has to be assigned to a master (also called a primary server). The web interface provides an easy way to pin primaries to a datacenter, but does not let the user pin a primary per shard or per machine basis. If you need this level of control, you will have to use the command-line interface instead.

Choosing a primary using the web interface

By default, the primary for a shard can be put anywhere in the cluster. That is to say, there is no constraint that requires the primary to be in a particular datacenter. In order to set a certain datacenter to contain all the primaries of your table, you will have to:

- Go to the table view (*Tables* > *table name*).
- Click on the *Show multi-datacenter options*.
- Click on the toggle box to *Pin masters to a single datacenter*.
- Choose the datacenter and click the *Update* button.



Choosing a primary using the command-line interface

Connect to your cluster via the command-line interface:

```
rethinkdb admin --join <host>:<port>
```

- Find the UUID of the table you want to shard using `ls`. Once you find the UUID of your table using the `ls` command, you can pin all of the primaries for a table to a particular datacenter with:
- To pin all of the primaries for a table to a particular datacenter, use `set primary <table> <datacenter>`.

The command line interface also provides a more precise way to pin data. You can pin a shard (primary or secondary) to a particular machine. The command to do this is:

```
pin shard <table> <shard> [--master <machine>] [--replicas <machine>...]
```

About automatic failover

Note: RethinkDB does not support fully automatic failover yet, but it is on the roadmap. Follow Github issue #223 for more information.

When a machine fails, it may be because of a network availability issue or something more serious, such as system failure. If the machine will not be able

to reconnect to the cluster, you can **manually declare it dead**, which is a simple one-click operation.

When a machine is declared dead, the system will attempt to recover itself automatically. However, dropping a machine from the cluster can affect table availability and require additional manual intervention in the following cases:

- If the machine was acting as **primary for any shards**, the corresponding tables will lose read and write availability (out-of-date reads remain possible as long as there are other replicas of the shards).
- If the machine was acting as a replica for a given table and there are **not enough replicas in the cluster** to respect the write acknowledgement settings (*acks*), the table will lose write availability, but maintains read availability.

On the other hand, if the machine was acting as a replica for a given table and there are enough replicas to respect the user's write acknowledgement settings (*acks*), the system continues operating as normal and the affected tables will maintain both read and write access. The system will be able to recover itself without additional intervention.

What to do when a machine goes down

In general, when a machine goes down, there are two possible solutions. The first option is to simply wait for the machine to become reachable again. If the machine comes back up, RethinkDB automatically performs the following actions without any user interaction:

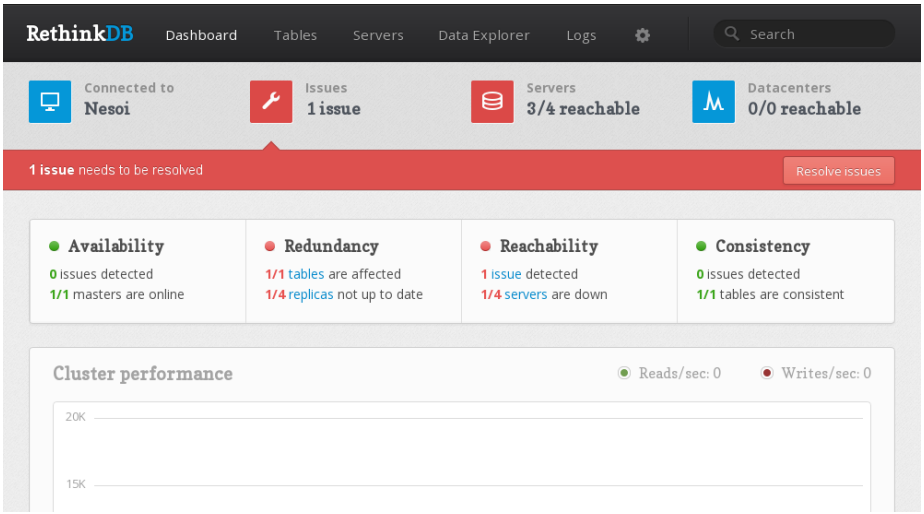
1. Replicas on the machine are brought up to date with the latest changes.
2. Primaries on the machine become active again.
3. The cluster clears the reachability issue from web and command line tools, availability is restored, and the cluster continues operating as normal.

The second option is to declare the machine dead. If a machine is declared dead, it is absolved of all responsibilities, and one of the replicas is automatically elected as a new primary. After the machine is declared dead, availability is quickly restored and the cluster begins operating normally. (If the dead machine comes back up, it is rejected by the cluster as a "zombie" since it might have data conflicts).

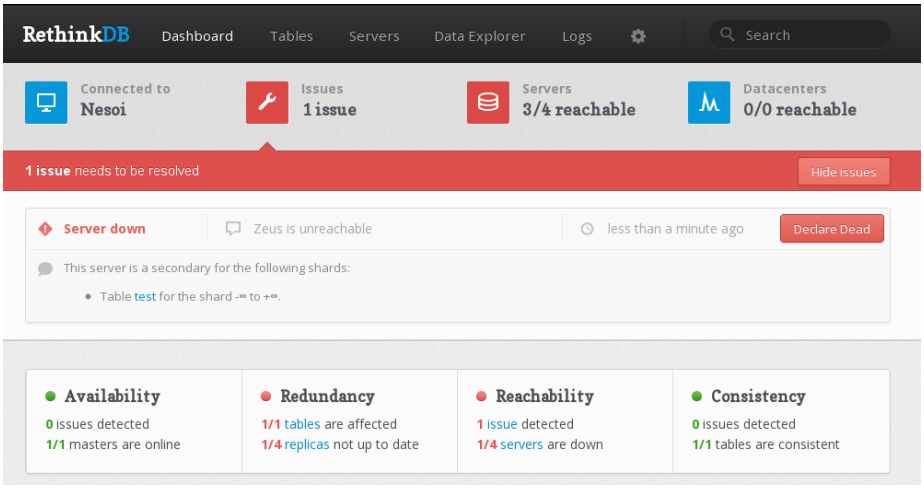
Example failover scenario using the web interface

In this example, we have 4 machines in our cluster and one table requiring 4 replicas and 4 acks.

As soon as one machine dies, the web interface reports an issue. This is what you would see in the webUI:



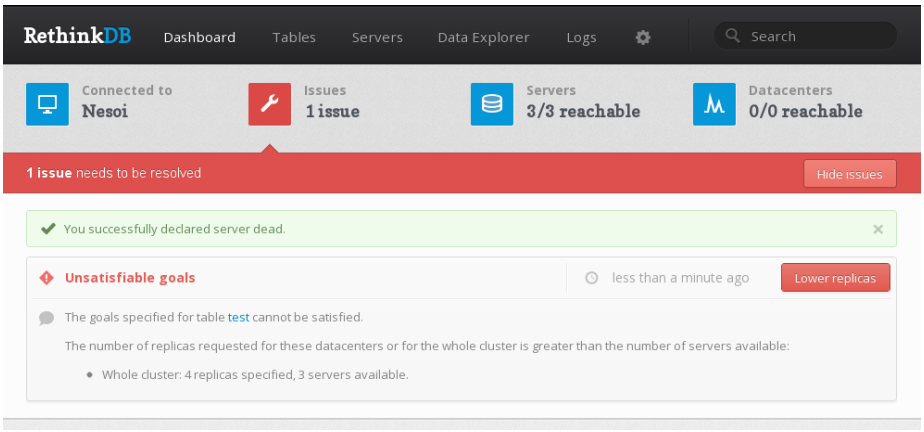
If we click on the *Resolve issues* button, we should see more information about the current issue. In our case, the unreachable machine is a replica (aka secondary) and therefore we have not lost any write availability.



In this case, if we don't want to wait for the machine to come back online, we can declare the machine dead. Declaring a machine dead means that we remove the machine and all its data from the cluster.

Warning: Once we have declared a machine dead, all of its data will be lost. Even if we later restart a RethinkDB instance with the same data directory, we will not be able to reuse the data.

Once the machine is declared dead, we have only three machines left in our cluster. Since we are requiring four replicas and four acks, the system raises a new error: *Unsatisfiable goals*. This error means that our replication requirements are not possible with the current cluster.



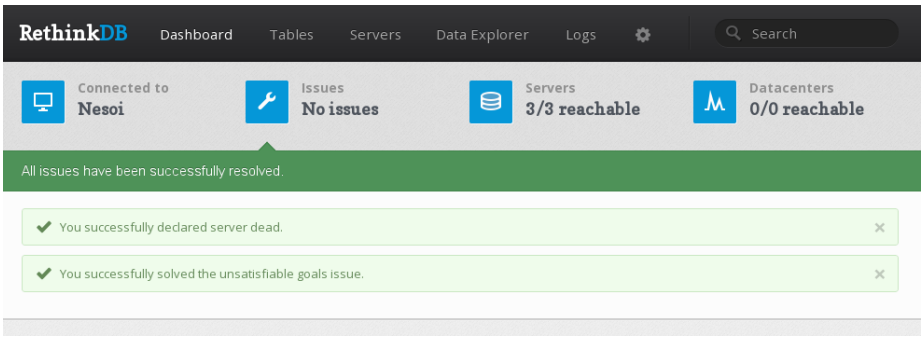
There are two ways for us to solve this issue:

- Connect a new machine to the cluster
- Lower the number of replicas and acks required

If we decide to click on the *Lower replicas* button, the system will lower the replicas just enough to solve the issue (in our case 3 replicas and 3 acks).

Note: There can sometimes be a situation where the system has the option to lower the number of replicas in a specific datacenter or in the whole cluster. In this case, there is no way for the system to know which option is preferred, so it will require the user to reduce the number of replicas manually.

Once the number of replicas has been reduced, there are no remaining issues in our cluster and the warning disappears.



Example failover scenario using the command-line interface

Connect to your cluster via the command-line interface:

```
rethinkdb admin --join <host>:<port>
```

In this example, a machine has already been declared dead so you should see this warning:

```
There is 1 outstanding issue, run 'ls issues' for more information
```

You can resolve this issue with the following:

```
# List the issues first using `ls`:
localhost:29015> ls issues

Machine 9d1b4e33-346d-406b-a1e9-8ce1d47e0f28 is inaccessible.

# If we try to remove the machine with the `rm` command, we will get an error
# stating "unsatisfiable goals":
localhost:29015> rm machine 9d1b4e33-346d-406b-a1e9-8ce1d47e0f28

error: Namespace ecaf9874-5fe2-4627-97ee-69c7cafdd9a8 has unsatisfiable goals

# To solve this problem, we can lower the number of replicas:
localhost:29105> set acks ecaf9874-5fe2-4627-97ee-69c7cafdd9a8 3
localhost:29105> set replicas ecaf9874-5fe2-4627-97ee-69c7cafdd9a8 3

# The list of issues should now be empty:
localhost:29015> ls issues
```

You must migrate your data **before** updating RethinkDB, since file formats are currently incompatible between versions. Migration consists of three simple steps:

- Export your data from the existing version of RethinkDB
- Upgrade RethinkDB to a new version
- Import the data into the new version of RethinkDB

If you have already updated to a new version of RethinkDB, you can find binaries for previous versions on the download archive.

Note: RethinkDB file formats are currently not compatible between versions. A better migration experience is on the roadmap (follow Github issue #1010 to track progress).

Exporting your data

To export your data, use `rethinkdb dump`:

```
rethinkdb dump --connect <host>:<port> [--auth <auth_key>]
```

where:

- `host` is the IP address of any machine of your RethinkDB cluster
- `port` is the port for driver connections (by default 29015)
- `auth_key` is an optional authentication key to connect to the cluster

This command will export all your data to a `tar.gz` file named `rethinkdb_dump_<timestamp>.tar.gz` (this may vary depending on your platform).

Use `rethinkdb dump --help` to see the complete list of options for dumping your data.

Exporting from a version before RethinkDB 1.7? Use the deprecated migration script.

Upgrading RethinkDB

First, upgrade RethinkDB to the latest version. See specific install instructions for your platform.

Then, make sure to move or delete the old RethinkDB data directory (`rethinkdb_data` by default), since the new version will not be able to read the old file.

Importing your data

To import your data, use `rethinkdb restore`:

```
rethinkdb restore <exported_file> --connect <host>:<port> [--auth <auth_key>]
```

where:

- `exported_file` is the data file exported with `rethinkdb dump`: by default named `rethinkdb_dump_<timestamp>.tar.gz` (this may vary depending on your platform)
- `host` is the IP address of any machine of your RethinkDB cluster

- `port` is the port for driver connections (by default 29015)
- `auth_key` is an optional authentication key to connect to the cluster

Use `rethinkdb restore --help` to see the complete list of options for importing your data.

Limitations

The dump/restore command currently comes with some limitations:

- The cluster configuration cannot be exported. The cluster has to be manually reconfigured.
- Secondary indexes cannot be exported. You will have to manually recreate them.

You can find three cookbooks for the three official drivers. The recipes comes from the most frequently asked questions.

- JavaScript
- Python
- Ruby

All of the cookbooks are pretty similar, but each may contain additional recipes specific to its language.

Looking for another language? So far, we've only created cookbooks for the three official RethinkDB drivers. We'll eventually create ones for our community-supported drivers as well, but if you'd like to help us move that process along, contact us at info@rethinkdb.com.

Don't see the recipe you're looking for? Feel free to request a recipe or add a recipe.

Request a new recipe using this form.

Add a new recipe using this form.

Basic commands

Creating a database

You can use the `dbCreate` command as follows:

```
r.dbCreate("blog").run(connection, function(err, result) {
    if (err) throw err;
    console.log(result);
})
```

Another way to create a database is through the web UI. You can reach the web UI at `http://HOST:8080`. Click on the *Tables* tab at the top and then use the *Add Database* button.

Creating a table

You can select the database where you'd like to create the table with the `db` command and use the `tableCreate` command as follows:

```
r.db("blog").tableCreate("posts").run( connection, function(err, result) {
    if (err) throw err;
    console.log(result);
})
```

Note that you can omit the `db` command if you're creating a table in the default database on your connection (set to `test` unless specified in `connect`).

Another way to create a new table is to use the web UI. You can reach the web UI at `http://HOST:8080`. Click on the *Tables* tab at the top of the page and then use the *Add Table* button.

Inserting documents

You can insert documents by calling the `insert` command on the appropriate table:

```
r.table("user").insert({
    name: "Michel",
    age: 26
}).run(connection, function(err, result) {
    if (err) throw err;
    console.log(result);
})
```

You can insert multiple documents at once by passing an array of documents to `insert` as follows:

```
r.table("user").insert([
    {
        name: "Michel",
        age: 26
    },
    {
        name: "Slava",
        age: 30
    }
]).run( connection, function(err, result) {
    if (err) throw err;
    console.log(result);
})
```

Deleting documents

To delete documents, select the documents you'd like to delete and use the `delete` command. For example, let's delete all posts with the author "Michel":

```
r.table("posts").filter(r.row("author").eq("Michel")).delete().
  run(connection, function(err, result) {
    if (err) throw err;
    console.log(result);
})
```

Or, let's try to delete a single user:

```
r.table("posts").get("7644aaf2-9928-4231-aa68-4e65e31bf219").delete().
  run(connection, function(err, result) {
    if (err) throw err;
    console.log(result);
})
```

Here is how we'd delete all documents in a table:

```
r.table("posts").delete().run(connection, function(err, result) {
    if (err) throw err;
    console.log(result);
})
```

Filtering

Filtering based on multiple fields

Suppose you'd like to select all posts where the author's name is "Michel" and the category is "Geek". You can do it as follows:

```
r.table("posts").filter({
    author: "Michel",
    category: "Geek",
}).run(connection, function(err, result) {
    if (err) throw err;
    console.log(result);
})
```

Alternatively, you can build a predicate with the `and` command, and pass it to `filter`:

```
r.table("posts").filter(
     r.row("author").eq("Michel").and(r.row("category").eq("Geek"))
    ).run(connection, function(err, result) {
        if (err) throw err;
        console.log(result);
    })
```

You can also use the prefix notation (passing all arguments to `r.and`), if that's what you prefer:

```
r.table("posts").filter(r.and(r.row("author").eq("Michel"),
                              r.row("category").eq("Geek"))
    ).run(connection, function(err, result) {
        if (err) throw err;
        console.log(result);
    })
```

Similarly, you can use the `r.or` command to filter based on one of many conditions.

Filtering based on the presence of a value in an array

Suppose we have a table `users` with documents of the following form:

```
{
    name: "William Adama"
    emails: ["bill@bsg.com", "william@bsg.com"]
}
```

If we want to retrieve all users that have the email address `user@email.com`, we can write:

```
r.table("user").filter(r.row("emails").contains("user@email.com"))
    .run( connection, function(err, result) {
        if (err) throw err;
        console.log(result);
    })
```

Filtering based on nested fields

In JavaScript you can use the operator `()` to get the value of a field. This operator can be chained to retrieve values from nested fields.

Suppose we have a table `users` with documents of the following form:

```
{
    name: "William Adama"
    contact: {
        phone: "555-5555"
        email: "bill@bsg.com"
    }
}
```

Let's filter based on the nested field `email`:

```
r.table("user").filter(
        r.row("contact")("email").eq("user@email.com")
    ).run(connection, function(err, result) {
        if (err) throw err;
        console.log(result);
    })
```

Efficiently retrieving multiple documents by primary key

If you want to retrieve all the posts with the primary keys `1`, `2`, or `3` you can use the `getAll` command:

```
r.table("posts").getAll(1, 2, 3).run(connection, function(err, result) {
    if (err) throw err;
    console.log(result);
})
```

Efficiently retrieving multiple documents by secondary index

Suppose we have a table `posts` that links posts to authors via an `author_id` field. If we've created a secondary index on `author_id` and want to retrieve all the posts where `author_id` is `1`, `2`, or `3`, we can use the `getAll` command to do it as follows:

```
r.table("posts").getAll(1, 2, 3, {index: 'author_id'}).
  run(connection, function(err, result) {
      if (err) throw err;
      console.log(result);
  })
```

Read about creating secondary indexes in RethinkDB.

Returning specific fields of a document

If you need to retrieve only a few specific fields from your documents, you can use the `pluck` command. For example, here is how you'd return only the fields `name` and `age` from each row in table `users`:

```
r.table("users").pluck("name", "age")
    .run(connection, function(err, result) {
        if (err) throw err;
        console.log(result);
    })
```

This is equivalent to calling `SELECT name, age FROM users` in SQL.

The `pluck` command also supports selecting nested fields in a document. For example, suppose we'd like to select the fields `phone` and `email` from the following document:

```
{
    name: "William Adama"
    contact: {
        phone: "555-5555"
```

```
        email: "bill@bsg.com"
    }
}
```

We can use the following syntax:

```
r.table("users").pluck({contact: {phone: true, email: true}})
    .run(connection, function(err, result) {
        if (err) throw err;
        console.log(result);
    })
```

Filtering based on a date range

Suppose you want to retrieve all the posts whose date field is between January 1st, 2012 (included) and January 1st, 2013 (excluded), you could do:

```
r.table("posts").filter( function(post) {
   return post("date").during( r.time(2012, 1, 1, 'Z'), r.time(2013, 1, 1, 'Z') )
}).run( conn, callback)
```

You can also manually compare dates:

```
r.table("posts").filter( function(post) {
    return post("date").ge( r.time(2012, 1, 1, 'Z') ).
       and(post("date").lt( r.time(2013, 1, 1, 'Z') ))
}).run( conn, callback)
```

Filtering with Regex

If you want to retrieve all users whose last name starts with "Ma", you can use `r.match` this way:

```
// Will return Martin, Martinez, Marshall etc.
r.table("users").filter( function(user) {
    return user("lastName").match("^Ma")
}).run( conn, callback)
```

If you want to retrieve all users whose last name ends with an "s", you can use `r.match` this way:

```
// Will return Williams, Jones, Davis etc.
r.table("users").filter( function(user) {
    return user("lastName").match("s$")
}).run( conn, callback)
```

If you want to retrieve all users whose last name contains "ll", you can use `r.match` this way:

```
// Will return Williams, Miller, Allen etc.
r.table("users").filter( function(user) {
    return user("lastName").match("ll")
}).run( conn, callback)
```

Case insensitive filter

Retrieve all users whose name is "William" (case insensitive).

```
// Will return william, William, WILLIAM, wiLLiam etc.
r.table("users").filter( function(user) {
    return user("name").match("(?i)^william$")
}).run( conn, callback)
```

Manipulating documents

Adding/overwriting a field in a document

To add or overwrite a field, you can use the `update` command. For instance, if you would like to add the field `author` with the value "Michel" for all of the documents in the table `posts`, you would use:

```
r.table("posts").update({ author: "Michel" }).run(connection, function(err, result) {
    if (err) throw err;
    console.log(result);
})
```

Removing a field from a document

The `update` command lets you to overwrite fields, but not delete them. If you want to delete a field, use the `replace` command. The `replace` command replaces your entire document with the new document you pass as an argument. For example, if you want to delete the field `author` of the blog post with the id `1`, you would use:

```
r.table("posts").get("1").replace(r.row.without('author')).
  run( connection, function(err, result) {
    if (err) throw err;
    console.log(result);
})
```

Atomically updating a document based on a condition

All modifications made via the `update` and `replace` commands are always atomic with respect to a single document. For example, let's say we'd like to atomically update a view count for a page if the field `countable` is set to true, and get back the old and new results in a single query. We can perform this operation as follows:

```
r.table("pages").update(function(page) {
   return r.branch(page("countable").eq(true), // if the page is countable
         { views: page("views").add(1) },   // increment the view count
             {}                                    // else do nothing
   )}, {return_vals: true}).run(connection, function(err, result) {
        if (err) throw err;
        console.log(result);
    })
```

Pagination

Limiting the number of returned documents

You can limit the number of documents returned by your queries with the `limit` command. Let's retrieve just the first 10 blog posts:

```
r.table("posts").orderBy("date").
  limit(10).
  run(connection, function(err, result) {
        if (err) throw err;
        console.log(result);
    })
```

Implementing pagination

To paginate results, you can use a combination of the `skip` and `limit` commands. Let's retrieve posts 11-20 from our database:

```
r.table("posts").orderBy("date").
  skip(10).
  limit(10).run(connection, function(err, result) {
      if (err) throw err;
      console.log(result);
  })
```

Transformations

Counting the number of documents in a table

You can count the number of documents with a `count` command:

```
r.table("posts").count().run(connection, function(err, result) {
    if (err) throw err;
    console.log(result);
})
```

Computing the average value of a field

To compute the average of a field, you can use a combination of `map` and `reduce` commands. For example, to compute the average number of comments per post, we would use `map` and `reduce` to add up the total number of comments and then divide that by the total number of posts.

```
r.table("posts").
  map(r.row("num_comments")).
  reduce(function(n, m) { return n.add(m) }).
  div(r.table("posts").count()).
  run(connection, function(err, result) {
        if (err) throw err;
        console.log(result);
  })
```

Using subqueries to return additional fields

Suppose we'd like to to retrieve all the posts in the table `post` and also return an additional field, `comments`, which is an array of all the comments for the relevant post retrieved from the `comments` table. We could do this using a subquery:

```
r.table("posts").map(function(post) {
```

```
    return post.merge({
           comments: r.table("comments").filter(function(comment) {
                 return comment("id_post").eq(post("id"))
             })
        })
    }).run(connection, function(err, result) {
        if (err) throw err;
        console.log(result);
    })
```

Performing a pivot operation

Suppose the table `marks` stores the marks of every students per course:

```
[
    {
        "name": "William Adama",
        "mark": 90,
        "id": 1,
        "course": "English"
    },
    {
        "name": "William Adama",
        "mark": 70,
        "id": 2,
        "course": "Mathematics"
    },
    {
        "name": "Laura Roslin",
        "mark": 80,
        "id": 3,
        "course": "English"
    },
    {
        "name": "Laura Roslin",
        "mark": 80,
        "id": 4,
        "course": "Mathematics"
    }
]
```

You may be interested in retrieving the results in this format

```
[
```

```
    {
        "name": "Laura Roslin",
        "Mathematics": 80,
        "English": 80
    },
    {
        "name": "William Adama",
        "Mathematics": 70,
        "English": 90
    }
]
```

In this case, you can do a pivot operation with the `groupedMapReduce` and `coerceTo` commands.

```
r.db('test').table('marks').groupedMapReduce(function(doc) {
        return doc("name")
    },
    function(doc) {
        return [[doc("course"), doc("mark")]]
    },
    function(left, right) {
        return left.union(right)
    }).map(function(result) {
        return r.expr({
            name: result("group")
        }).merge( result("reduction").coerceTo("OBJECT") )
    })
```

Note: A nicer syntax will eventually be added. See the Github issue 838 to track progress.

Performing an unpivot operation

Doing an unpivot operation to "cancel" a pivot one can be done with the `concatMap`, `map` and `coerceTo` commands:

```
r.table("pivotedMarks").concatMap( function(doc) {
   return doc.without("name").coerceTo("array").map( function(values) {
        return {
            name: doc("name"),
            course: values.nth(0),
            mark: values.nth(1)
        }
```

```
    })
})
```

Note: A nicer syntax will eventually be added. See the Github issue 838 to track progress.

Renaming a field when retrieving documents

Suppose we want to rename the field `id` to `idUser` when retrieving documents from the table `users`. We could do:

```
r.table("users").map(
   r.row.merge({ // Add the field idUser that is equal to the id one
        idUser: r.row("id")
    })
    .without("id") // Remove the field id
)
```

Miscellaneous

Generating monotonically increasing primary key values

Efficiently generating monotonically increasing IDs in a distributed system is a surprisingly difficult problem. If an inserted document is missing a primary key, RethinkDB currently generates a random UUID. We will be supporting additional autogeneration schemes in the future (see https://github.com/rethinkdb/rethinkdb/issues/117), but in the meantime, you can use one of the available open-source libraries for distributed id generation (e.g. twitter snowflake).

Parsing RethinkDB's response to a write query

When you issue a write query (`insert`, `delete`, `update`, or `replace`), RethinkDB returns a summary object that looks like this:

```
{deleted:0, replaced:0, unchanged:0, errors:0, skipped:0, inserted:1}
```

The most important field of this object is `errors`. Generally speaking, if no exceptions are thrown and `errors` is 0 then the write did what it was supposed to. (RethinkDB throws an exception when it isn't even able to access the table; it sets the `errors` field if it can access the table but an error occurs during

the write. This convention exists so that batched writes don't abort halfway through when they encounter an error.)

The following fields are always present in this object:

- `inserted` – Number of new documents added to the database.
- `deleted` – Number of documents deleted from the database.
- `replaced` – Number of documents that were modified.
- `unchanged` – Number of documents that would have been modified, except that the new value was the same as the old value.
- `skipped` – Number of documents that were left unmodified because there was nothing to do. (For example, if you delete a row that has already been deleted, that row will be "skipped". This field is sometimes positive even when operating on a selection, because a concurrent write might get to the value first.)
- `errors` – Number of documents that were left unmodified due to an error.

In addition, the following two fields are set as circumstances dictate:

- `generated_keys` – If you issue an insert query where some or all of the rows lack primary keys, the server will generate primary keys for you and return an array of those keys in this field. (The order of this array will match the order of the rows in your insert query.)
- `first_error` – If `errors` is positive, the text of the first error message encountered will be in this field. This is a very useful debugging aid. (We don't return all of the errors because a single typo can result in millions of errors when operating on a large database.)

Don't see the recipe you're looking for? Feel free to request a recipe or add a recipe.

Request a new recipe using this form.

Add a new recipe using this form.

Basic commands

Creating a database

You can use the `db_create` method as follows:

```
r.db_create("blog").run
```

Another way to create a database is through the web UI. You can reach the web UI at `http://HOST:8080`. Click on the *Tables* tab at the top and then use the *Add Database* button.

Creating a table

You can select the database where you'd like to create the table with the `db` command and use the `table_create` command as follows:

```
r.db("blog").table_create("posts").run
```

Note that you can omit the `db` command if you're creating a table in the default database on your connection (set to `test` unless specified in `connect`).

Another way to create a new table is to use the web UI. You can reach the web UI at `http://HOST:8080`. Click on the *Tables* tab at the top of the page and then use the *Add Table* button.

Inserting documents

You can insert documents by calling the `insert` command on the appropriate table:

```
r.table("user").insert({
    "name" => "Michel",
    "age" => 26
}).run
```

You can insert multiple documents at once by passing an array of documents to `insert` as follows:

```
r.table("user").insert([
    {
        "name" => "Michel",
        "age" => 26
    },
    {
        "name" => "Slava",
        "age" => 30
    }
]).run
```

Deleting documents

To delete documents, select the documents you'd like to delete and use the `delete` command. For example, let's delete all posts with the author "Michel":

```
r.table("posts").filter{|post| post["author"].eq("Michel")}.delete.run
```

Or, let's try to delete a single user:

```
r.table("posts").get("7644aaf2-9928-4231-aa68-4e65e31bf219").delete.run
```

Here is how we'd delete all documents in a table:

```
r.table("posts").delete.run
```

Filtering

Filtering based on multiple fields

Suppose you'd like to select all posts where the author's name is "Michel" and the category is "Geek". You can do it as follows:

```
r.table("posts").filter({
    "author" => "Michel",
    "category" => "Geek",
}).run
```

Alternatively, you can use the overloaded `&` operator to build a predicate and pass it to `filter`:

```
r.table("posts").filter{|post|
    (post["author"].eq("Michel")) & (post["category"].eq("Geek"))
}.run
```

Note: RethinkDB overloads `&` because Ruby doesn't allow overloading the proper *and* operator. Since `&` has high precedence, make sure to wrap the conditions around it in parentheses.

You can also use the `r.and` command, if you prefer not using overloaded `&`:

```
r.table("posts").filter{|post|
    r.and(post["author"].eq("Michel"),
          post["category"].eq("Geek"))}.run
```

Similarly, you can use the overloaded `|` operator or the equivalent `r.or` command to filter based on one of many conditions.

Filtering based on the presence of a value in an array

Suppose we have a table `users` with documents of the following form:

```
{
    "name" => "William Adama"
    "emails" => ["bill@bsg.com", "william@bsg.com"]
}
```

If we want to retrieve all users that have the email address `user@email.com`, we can write:

```
r.table("user").filter{|user| user["emails"].contains("user@email.com")}.run
```

Filtering based on nested fields

In Ruby you can use the operator `[]` to get the value of a field. This operator can be chained to retrieve values from nested fields.

Suppose we have a table `users` with documents of the following form:

```
{
    "name" => "William Adama"
    "contact" => {
        "phone" => "555-5555"
        "email" => "bill@bsg.com"
    }
}
```

Let's filter based on the nested field `email`:

```
r.table("user").filter{|user|
    user["contact"]["email"].eq("user@email.com")
}.run
```

Efficiently retrieving multiple documents by primary key

If you want to retrieve all the posts with the primary keys `1`, `2`, or `3` you can use the `get_all` command:

```
r.table("posts").get_all(1, 2, 3).run
```

Efficiently retrieving multiple documents by secondary index

Suppose we have a table `posts` that links posts to authors via an `author_id` field. If we've created a secondary index on `author_id` and want to retrieve all the posts where `author_id` is `1`, `2`, or `3`, we can use the `get_all` command to do it as follows:

```
r.table("posts").get_all(1, 2, 3, :index=>'author_id').run
```

Read about creating secondary indexes in RethinkDB.

Returning specific fields of a document

If you need to retrieve only a few specific fields from your documents, you can use the `pluck` command. For example, here is how you'd return only the fields `name` and `age` from each row in table `users`:

```
r.table("posts").pluck("name", "age").run
```

This is equivalent to calling `SELECT name, age FROM users` in SQL.

The `pluck` command also supports selecting nested fields in a document. For example, suppose we'd like to select the fields `phone` and `email` from the following document:

```
{
    "name" => "William Adama"
    "contact" => {
        "phone" => "555-5555"
        "email" => "bill@bsg.com"
    }
}
```

We can use the following syntax:

```
r.table("users").pluck({"contact"=>{"phone"=>true, "email"=>true}}).run
```

Filtering based on a date range

Suppose you want to retrieve all the posts whose date field is between January 1st, 2012 (included) and January 1st, 2013 (excluded), you could do:

```
r.table("posts").filter{ |post|
    post.during(r.time(2012, 1, 1, 'Z'), r.time(2013, 1, 1, 'Z'))
}.run(conn)
```

You can also manually compare dates:

```
r.table("posts").filter{ |post|
    (post["date"] >= r.time(2012, 1, 1, 'Z')) &
    (post["date"] < r.time(2013, 1, 1, 'Z'))
}.run(conn)
```

Filtering with regex

If you want to retrieve all users whose last name starts with "Ma", you can use `r.match` this way:

```
# Will return Martin, Martinez, Marshall etc.
r.table("users").filter{ |user|
    user["lastName"].match("^Ma")
}.run(conn)
```

If you want to retrieve all users whose last name ends with an "s", you can use `r.match` this way:

```
# Will return Williams, Jones, Davis etc.
r.table("users").filter{ |user|
    user["lastName"].match("s$")
}.run(conn)
```

If you want to retrieve all users whose last name contains "ll", you can use `r.match` this way:

```
# Will return Williams, Miller, Allen etc.
r.table("users").filter{ |user|
    user["lastName"].match("ll")
}.run(conn)
```

Case insensitive filter

Retrieve all users whose name is "William" (case insensitive).

```
# Will return william, William, WILLIAM, wiLLiam etc.
r.table("users").filter{ |user|
    user["lastName"].match("(?i)^william$")
}.run(conn)
```

Manipulating documents

Adding/overwriting a field in a document

To add or overwrite a field, you can use the `update` command. For instance, if you would like to add the field `author` with the value "Michel" for all of the documents in the table `posts`, you would use:

```
r.table("posts").update({ "author" => "Michel" }).run
```

Removing a field from a document

The `update` command lets you to overwrite fields, but not delete them. If you want to delete a field, use the `replace` command. The `replace` command replaces your entire document with the new document you pass as an argument. For example, if you want to delete the field `author` of the blog post with the id `1`, you would use:

```
r.table("posts").get("1").replace{|doc| doc.without('author')}.run
```

Atomically updating a document based on a condition

All modifications made via the `update` and `replace` commands are always atomic with respect to a single document. For example, let's say we'd like to atomically update a view count for a page if the field `countable` is set to true, and get back the old and new results in a single query. We can perform this operation as follows:

```
r.table("pages").update{|page|
   r.branch(page["countable"].eq(true),      // if the page is countable
         { "views"=>page["views"] + 1 },   // increment the view count
             {}                                  // else do nothing
    )}, {"return_vals"=>true}).run()
```

Pagination

Limiting the number of returned documents

You can limit the number of documents returned by your queries with the `limit` command. Let's retrieve just the first 10 blog posts:

```
r.table("posts").order_by("date")
    .limit(10)
    .run
```

Implementing pagination

To paginate results, you can use a combination of the `skip` and `limit` commands. Let's retrieve posts 11-20 from our database:

```
r.table("posts").order_by("date").
  skip(10).
  limit(10).run
```

Transformations

Counting the number of documents in a table

You can count the number of documents with a `count` command:

```
r.table("posts").count.run
```

Computing the average value of a field

To compute the average of a field, you can use a combination of `map` and `reduce` commands. For example, to compute the average number of comments per post, we would use `map` and `reduce` to add up the total number of comments and then divide that by the total number of posts.

```
r.table("posts").
  map{|post| post["num_comments"]}.
  reduce{|n, m| n + m}.
  div(r.table("posts").count).
  run
```

Using subqueries to return additional fields

Suppose we'd like to to retrieve all the posts in the table `post` and also return an additional field, `comments`, which is an array of all the comments for the relevant post retrieved from the `comments` table. We could do this using a subquery:

```
r.table("posts").map{|post|
    post.merge({"comments" => r.table("comments").filter{|comment|
                comment["id_post"].eq(post["id"])
            }
        })
    }.run
```

Performing a pivot operation

Suppose the table `marks` stores the marks of every students per course:

```
[
    {
        :name => "William Adama",
        :mark => 90,
        :id => 1,
        :course => "English"
    },
    {
        :name => "William Adama",
        :mark => 70,
        :id => 2,
        :course => "Mathematics"
    },
    {
        :name => "Laura Roslin",
        :mark => 80,
        :id => 3,
        :course => "English"
    },
    {
        :name => "Laura Roslin",
        :mark => 80,
        :id => 4,
        :course => "Mathematics"
    }
]
```

You may be interested in retrieving the results in this format

```
[
    {
        :name => "Laura Roslin",
        :Mathematics => 80,
```

```
        :English => 80
    },
    {
        :name => "William Adama",
        :Mathematics => 70,
        :English => 90
    }
]
```

In this case, you can do a pivot operation with the `grouped_map_reduce` and `coerce_to` commands.

```
r.db('test').table('marks').grouped_map_reduce( lambda {|doc|
        doc["name"]
    },
    lambda {|doc|
        [[doc["course"], doc["mark"]]]
    },
    lambda {|left, right|
        left.union(right)
    }).map{ |result|
        r.expr({
            :name => result["group"]
        }).merge( result["reduction"].coerce_to("OBJECT") )
    }
```

Note: A nicer syntax will eventually be added. See the Github issue 838 to track progress.

Performing an unpivot operation

Doing an unpivot operation to "cancel" a pivot one can be done with the `concatMap`, `map` and `coerce_to` commands:

```
r.table("pivoted_marks").concat_map{ |doc|
    doc.without("name").coerce_to("array").map{ |values|
        {
            :name => doc["name"],
            :course => values[0],
            :mark => values[1]
        }
    }
}
```

Note: A nicer syntax will eventually be added. See the Github issue 838 to track progress.

Renaming a field when retrieving documents

Suppose we want to rename the field `id` to `id_user` when retrieving documents from the table `users`. We could do:

```
r.table("users").map{ |user|
    # Add the field id_user that is equal to the id one
    user.merge({
        "id_user" => user["id"]
    })
    .without("id") # Remove the field id
}
```

Miscellaneous

Generating monotonically increasing primary key values

Efficiently generating monotonically increasing IDs in a distributed system is a surprisingly difficult problem. If an inserted document is missing a primary key, RethinkDB currently generates a random UUID. We will be supporting additional autogeneration schemes in the future (see https://github.com/rethinkdb/rethinkdb/issues/117), but in the meantime, you can use one of the available open-source libraries for distributed id generation (e.g. twitter snowflake).

Parsing RethinkDB's response to a write query

When you issue a write query (`insert`, `delete`, `update`, or `replace`), RethinkDB returns a summary object that looks like this:

```
{"deleted"=>0, "replaced"=>0, "unchanged"=>0, "errors"=>0, "skipped"=>0, "inserted"=>1}
```

The most important field of this object is `errors`. Generally speaking, if no exceptions are thrown and `errors` is 0 then the write did what it was supposed to. (RethinkDB throws an exception when it isn't even able to access the table; it sets the `errors` field if it can access the table but an error occurs during the write. This convention exists so that batched writes don't abort halfway through when they encounter an error.)

The following fields are always present in this object:

- `inserted` – Number of new documents added to the database.

- `deleted` – Number of documents deleted from the database.
- `replaced` – Number of documents that were modified.
- `unchanged` – Number of documents that would have been modified, except that the new value was the same as the old value.
- `skipped` – Number of documents that were left unmodified because there was nothing to do. (For example, if you delete a row that has already been deleted, that row will be "skipped". This field is sometimes positive even when operating on a selection, because a concurrent write might get to the value first.)
- `errors` – Number of documents that were left unmodified due to an error.

In addition, the following two fields are set as circumstances dictate:

- `generated_keys` – If you issue an insert query where some or all of the rows lack primary keys, the server will generate primary keys for you and return an array of those keys in this field. (The order of this array will match the order of the rows in your insert query.)
- `first_error` – If `errors` is positive, the text of the first error message encountered will be in this field. This is a very useful debugging aid. (We don't return all of the errors because a single typo can result in millions of errors when operating on a large database.)

Don't see the recipe you're looking for? Feel free to request a recipe or add a recipe.

Request a new recipe using this form.

Add a new recipe using this form.

Basic commands

Creating a database

You can use the `db_create` method as follows:

```
r.db_create("blog").run()
```

Another way to create a database is through the web UI. You can reach the web UI at `http://HOST:8080`. Click on the *Tables* tab at the top and then use the *Add Database* button.

Creating a table

You can select the database where you'd like to create the table with the `db` command and use the `table_create` command as follows:

```
r.db("blog").table_create("posts").run()
```

Note that you can omit the `db` command if you're creating a table in the default database on your connection (set to `test` unless specified in `connect`).

Another way to create a new table is to use the web UI. You can reach the web UI at `http://HOST:8080`. Click on the *Tables* tab at the top of the page and then use the *Add Table* button.

Inserting documents

You can insert documents by calling the `insert` command on the appropriate table:

```
r.table("user").insert({
    "name": "Michel",
    "age": 26
}).run()
```

You can insert multiple documents at once by passing an array of documents to `insert` as follows:

```
r.table("user").insert([
    {
        "name": "Michel",
        "age": 26
    },
    {
        "name": "Slava",
        "age": 30
    }
]).run()
```

Deleting documents

To delete documents, select the documents you'd like to delete and use the `delete` command. For example, let's delete all posts with the author "Michel":

```
r.table("posts").filter(r.row["author"] == "Michel").delete().run()
```

Or, let's try to delete a single user:

```
r.table("posts").get("7644aaf2-9928-4231-aa68-4e65e31bf219").delete().run()
```

Here is how we'd delete all documents in a table:

```
r.table("posts").delete().run()
```

Filtering

Filtering based on multiple fields

Suppose you'd like to select all posts where the author's name is "Michel" and the category is "Geek". You can do it as follows:

```
r.table("posts").filter({
    "author": "Michel",
    "category": "Geek",
}).run()
```

Alternatively, you can use the overloaded `&` operator to build a predicate and pass it to `filter`:

```
r.table("posts").filter(
      (r.row["author"] == "Michel") & (r.row["category"] == "Geek")
    ).run()
```

Note: RethinkDB overloads `&` because Ruby doesn't allow overloading the proper *and* operator. Since `&` has high precedence, make sure to wrap the conditions around it in parentheses.

You can also use the `r.all` command, if you prefer not using overloaded `&`:

```
r.table("posts").filter(r.all(r.row["author"] == "Michel",
                              r.row["category"] == "Geek")).run()
```

Similarly, you can use the overloaded `|` operator or the equivalent `r.any` command to filter based on one of many conditions.

Filtering based on the presence of a value in an array

Suppose we have a table `users` with documents of the following form:

```
{
    name: "William Adama"
    emails: ["bill@bsg.com", "william@bsg.com"]
}
```

If we want to retrieve all users that have the email address `user@email.com`, we can write:

```
r.table("user").filter(r.row["emails"].contains("user@email.com")).run()
```

Filtering based on nested fields

In Python you can use the operator `[]` to get the value of a field. This operator can be chained to retrieve values from nested fields.

Suppose we have a table `users` with documents of the following form:

```
{
    name: "William Adama"
    contact: {
        phone: "555-5555"
        email: "bill@bsg.com"
    }
}
```

Let's filter based on the nested field `email`:

```
r.table("user").filter(
    r.row["contact"]["email"] == "user@email.com"
).run()
```

Efficiently retrieving multiple documents by primary key

If you want to retrieve all the posts with the primary keys `1`, `2`, or `3` you can use the `get_all` command:

```
r.table("posts").get_all(1, 2, 3).run()
```

Efficiently retrieving multiple documents by secondary index

Suppose we have a table `posts` that links posts to authors via an `author_id` field. If we've created a secondary index on `author_id` and want to retrieve all the posts where `author_id` is `1`, `2`, or `3`, we can use the `get_all` command to do it as follows:

```
r.table("posts").get_all(1, 2, 3, index='author_id').run()
```

Read about creating secondary indexes in RethinkDB.

Returning specific fields of a document

If you need to retrieve only a few specific fields from your documents, you can use the `pluck` command. For example, here is how you'd return only the fields `name` and `age` from each row in table `users`:

```
r.table("users").pluck("name", "age").run()
```

This is equivalent to calling `SELECT name, age FROM users` in SQL.

The `pluck` command also supports selecting nested fields in a document. For example, suppose we'd like to select the fields `phone` and `email` from the following document:

```
{
    name: "William Adama"
    contact: {
        phone: "555-5555"
        email: "bill@bsg.com"
    }
}
```

We can use the following syntax:

```
r.table("users").pluck({"contact": {"phone": True, "email": True}}).run()
```

Filtering based on a date range

Suppose you want to retrieve all the posts whose date field is between January 1st, 2012 (included) and January 1st, 2013 (excluded), you could do:

```
r.table("posts").filter( lambda post:
    post.during(r.time(2012, 1, 1, 'Z'), r.time(2013, 1, 1, 'Z'))
).run(conn)
```

You can also manually compare dates:

```
r.table("posts").filter( lambda post:
    (post["date"] >= r.time(2012, 1, 1, 'Z')) &
    (post["date"] < r.time(2013, 1, 1, 'Z'))
).run(conn)
```

Filering with Regex

If you want to retrieve all users whose last name starts with "Ma", you can use `r.match` this way:

```
# Will return Martin, Martinez, Marshall etc.
r.table("users").filter( lambda user:
    user["lastName"].match("^Ma")
).run(conn)
```

If you want to retrieve all users whose last name ends with an "s", you can use `r.match` this way:

```
# Will return Williams, Jones, Davis etc.
r.table("users").filter( lambda user:
    user["lastName"].match("s$")
).run(conn)
```

If you want to retrieve all users whose last name contains "ll", you can use `r.match` this way:

```
# Will return Williams, Miller, Allen etc.
r.table("users").filter( lambda user:
    user["lastName"].match("ll")
).run(conn)
```

Case insensitive filter

Retrieve all users whose name is "William" (case insensitive).

```
# Will return william, William, WILLIAM, wiLLiam etc.
r.table("users").filter( lambda user:
    user["lastName"].match("(?i)^william$")
).run(conn)
```

Manipulating documents

Adding/overwriting a field in a document

To add or overwrite a field, you can use the `update` command. For instance, if you would like to add the field `author` with the value "Michel" for all of the documents in the table `posts`, you would use:

```
r.table("posts").update({ "author": "Michel" }).run()
```

Removing a field from a document

The `update` command lets you to overwrite fields, but not delete them. If you want to delete a field, use the `replace` command. The `replace` command replaces your entire document with the new document you pass as an argument. For example, if you want to delete the field `author` of the blog post with the id `1`, you would use:

```
r.table("posts").get("1").replace(r.row.without('author')).run()
```

Atomically updating a document based on a condition

All modifications made via the `update` and `replace` commands are always atomic with respect to a single document. For example, let's say we'd like to atomically update a view count for a page if the field `countable` is set to true, and get back the old and new results in a single query. We can perform this operation as follows:

```
r.table("pages").update(lambda page:
   r.branch(page["countable"] == True,       // if the page is countable
         { "views": page["views"] + 1 },    // increment the view count
             {}                                   // else do nothing
    )), {"return_vals": True}).run()
```

Pagination

Limiting the number of returned documents

You can limit the number of documents returned by your queries with the `limit` command. Let's retrieve just the first 10 blog posts:

```
r.table("posts").order_by("date").
  limit(10).
  run()
```

Implementing pagination

To paginate results, you can use a combination of the `skip` and `limit` commands. Let's retrieve posts 11-20 from our database:

```
r.table("posts").order_by("date").
  skip(10).
  limit(10).run()
```

Transformations

Counting the number of documents in a table

You can count the number of documents with a `count` command:

```
r.table("posts").count().run()
```

Computing the average value of a field

To compute the average of a field, you can use a combination of `map` and `reduce` commands. For example, to compute the average number of comments per post, we would use `map` and `reduce` to add up the total number of comments and then divide that by the total number of posts.

```
r.table("posts").
  map(r.row["num_comments"]).
  reduce(lambda n, m: n + m).
  div(r.table("posts").count()).
  run()
```

Using subqueries to return additional fields

Suppose we'd like to to retrieve all the posts in the table `post` and also return an additional field, `comments`, which is an array of all the comments for the relevant post retrieved from the `comments` table. We could do this using a subquery:

```
r.table("posts").map(lambda post:
    post.merge({
        "comments": r.table("comments").filter(lambda comment:
            comment["id_post"] == post["id"])
        })
    ).run()
```

Performing a pivot operation

Suppose the table `marks` stores the marks of every students per course:

```
[
    {
        "name": "William Adama",
        "mark": 90,
        "id": 1,
        "course": "English"
    },
    {
        "name": "William Adama",
        "mark": 70,
        "id": 2,
        "course": "Mathematics"
    },
    {
        "name": "Laura Roslin",
        "mark": 80,
        "id": 3,
        "course": "English"
    },
    {
        "name": "Laura Roslin",
        "mark": 80,
        "id": 4,
        "course": "Mathematics"
    }
]
```

You may be interested in retrieving the results in this format

```
[
    {
        "name": "Laura Roslin",
        "Mathematics": 80,
```

```
        "English": 80
    },
    {
        "name": "William Adama",
        "Mathematics": 70,
        "English": 90
    }
]
```

In this case, you can do a pivot operation with the `grouped_map_reduce` and the `coerce_to` commands.

```
r.db('test').table('marks').grouped_map_reduce(lambda doc:
        doc["name"]
    ,
    lambda doc:
        [[doc["course"], doc["mark"]]]
    ,
    lambda left, right:
        left.union(right)
    ).map( lambda result:
        r.expr({
            "name": result["group"]
        }).merge( result["reduction"].coerce_to("OBJECT") )
    )
```

Note: A nicer syntax will eventually be added. See the Github issue 838 to track progress.

Performing an unpivot operation

Doing an unpivot operation to "cancel" a pivot one can be done with the `concatMap`, `map` and `coerceTo` commands:

```
r.table("pivoted_marks").concat_map( lambda doc:
    doc.without("name").coerce_to("array").map( lambda values:
        {
            "name": doc["name"],
            "course": values[0],
            "mark": values[1]
        }
    )
)
```

Note: A nicer syntax will eventually be added. See the Github issue 838 to track progress.

Renaming a field when retrieving documents

Suppose we want to rename the field `id` to `id_user` when retrieving documents from the table `users`. We could do:

```
r.table("users").map(
    # Add the field id_user that is equal to the id one
    r.row.merge({
        "id_user": r.row["id"]
    })
    .without("id") # Remove the field id
)
```

Miscellaneous

Generating monotonically increasing primary key values

Efficiently generating monotonically increasing IDs in a distributed system is a surprisingly difficult problem. If an inserted document is missing a primary key, RethinkDB currently generates a random UUID. We will be supporting additional autogeneration schemes in the future (see https://github.com/rethinkdb/rethinkdb/issues/117), but in the meantime, you can use one of the available open-source libraries for distributed id generation (e.g. twitter snowflake).

Parsing RethinkDB's response to a write query

When you issue a write query (`insert`, `delete`, `update`, or `replace`), RethinkDB returns a summary object that looks like this:

```
{"deleted":0, "replaced":0, "unchanged":0, "errors":0, "skipped":0, "inserted":1}
```

The most important field of this object is `errors`. Generally speaking, if no exceptions are thrown and `errors` is 0 then the write did what it was supposed to. (RethinkDB throws an exception when it isn't even able to access the table; it sets the `errors` field if it can access the table but an error occurs during the write. This convention exists so that batched writes don't abort halfway through when they encounter an error.)

The following fields are always present in this object:

- `inserted` – Number of new documents added to the database.

- `deleted` – Number of documents deleted from the database.
- `replaced` – Number of documents that were modified.
- `unchanged` – Number of documents that would have been modified, except that the new value was the same as the old value.
- `skipped` – Number of documents that were left unmodified because there was nothing to do. (For example, if you delete a row that has already been deleted, that row will be "skipped". This field is sometimes positive even when operating on a selection, because a concurrent write might get to the value first.)
- `errors` – Number of documents that were left unmodified due to an error.

In addition, the following two fields are set as circumstances dictate:

- `generated_keys` – If you issue an insert query where some or all of the rows lack primary keys, the server will generate primary keys for you and return an array of those keys in this field. (The order of this array will match the order of the rows in your insert query.)
- `first_error` – If `errors` is positive, the text of the first error message encountered will be in this field. This is a very useful debugging aid. (We don't return all of the errors because a single typo can result in millions of errors when operating on a large database.)

My insert queries are slow. How can I speed them up?

Have you installed optimized `protobuf` libraries? See this document to learn how to use optimized `protobuf` libraries with the client drivers.

RethinkDB uses a safe default configuration for write acknowledgement. Each write is committed to disk before the server acknowledges it to the client. If you're running a single thread that inserts documents into RethinkDB in a loop, each insert must wait for the server acknowledgement before proceeding to the next one. This can significantly slow down the overall throughput.

This behavior is similar to any other safe database system. Below is a number of steps you can take to speed up insert performance in RethinkDB. Most of these guidelines will also apply to other database systems.

- **Increase concurrency.** Instead of having a single thread inserting data in a loop, create multiple threads with multiple connections. This will

allow parallelization of insert queries without spending most of the time waiting on disk acknowledgement.

- **Batch writes.** Instead of doing single writes in a loop, group writes together. This can result in significant increases in throughput. Instead of doing multiple queries like this:

  ```
  r.db("foo").table("bar").insert(document_1).run()
  r.db("foo").table("bar").insert(document_2).run()
  r.db("foo").table("bar").insert(document_3).run()
  ```

  Combine them into a single query:

  ```
  r.db("foo").table("bar").insert([document_1, document_2, document_3]).run()
  ```

  RethinkDB operates at peak performance when the batch size is around two hundred documents.

- **Consider using soft durability mode.** In soft durability mode RethinkDB will acknowledge the write immediately after receiving it, but before the write has been committed to disk. The server will use main memory to absorb the write, and will flush new data to disk in the background.

  This mode is **not as safe** as the default hard durability mode. If you're writing using soft durability, a few seconds worth of data might be lost in case of power failure.

  Note: while some data may be lost in case of power failure in soft durability mode, the RethinkDB database will not get corrupted.

  You can insert data in soft durability mode as follows:

  ```
  r.db("foo").table("bar").insert(document).run(durability="soft")
  ```

- **Consider using `noreply` mode.** In this mode, the client driver will not wait for the server acknowledgement of the query before moving on to the next query. This mode is even less safe than the soft durability mode, but can result in the highest performance improvement. You can run a command in a `noreply` mode as follows:

  ```
  r.db("foo").table("bar").insert(document).run(noreply=True)
  ```

  You can also combine soft durability and `noreply` for the highest performance:

  ```
  r.db("foo").table("bar").insert(document).run(durability="soft", noreply=True)
  ```

What does 'received invalid clustering header' mean?

RethinkDB uses three ports to operate — the HTTP web UI port, the client drivers port, and the intracluster traffic port. You can connect the browser to the web UI port to administer the cluster right from your browser, and connect the client drivers to the client driver port to run queries from your application. If you're running a cluster, different RethinkDB nodes communicate with each other via the intracluster traffic port.

The message `received invalid clustering header` means there is a port mismatch, and something is connecting to the wrong port. For example, it's common to get this message if you accidently point the browser or connect the client drivers to the intracluster traffic port.

Does the web UI support my browser?

The following browsers are supported and known to work with the web UI:

- Chrome 9 or higher
- Firefox 15 or higher
- Safari 6.02 or higher
- Opera 1.62 or higher

The web UI requires `DataView` and `Uint8Array` JavaScript features to be supported by your browser.

Which versions of Node.js are supported?

The JavaScript driver currently works with Node.js versions 0.10.0 and above. You can check your node version as follows:

```
node --version
```

You can upgrade your version of Node.js via `npm`:

```
sudo npm install -g n
```

If you're trying to run the RethinkDB JavaScript driver on an older version of Node.js, you might get an error similar to this one:

```
/home/user/rethinkdb.js:13727
return buffer.slice(offset, end);
             ^
TypeError: Object #<ArrayBuffer> has no method 'slice'
at bufferSlice (/home/user/rethinkdb.js:13727:17)
at Socket.TcpConnection.rawSocket.once.handshake_callback (/home/user/rethinkdb.js:13552:
```

I get back a connection in my callback with the Node driver

Many people have been reporting that they get back a connection object when they run a query, the object being:

```
{
    _conn: {
        host: 'localhost',
        port: 28015,
        db: undefined,
        authKey: '',
        timeout: 20,
        outstandingCallbacks: {},
        nextToken: 2,
        open: true,
        buffer: <Buffer 04 00 00 00 08 02 10 01>,
        _events: {},
        rawSocket: { ... }
    },
    _token: 1,
    _chunks: [],
    _endFlag: true,
    _contFlag: true,
    _cont: null,
    _cbQueue: []
}
```

This object is not a connection, but a cursor. To retrieve the results, you can call `next`, `each` or `toArray` on this object.

For example you can retrieve all the results and put them in an array with `toArray`:

```
r.table("test").run( conn, function(error, cursor) {
    cursor.toArray( function(error, results) {
        console.log(results) // results is an array of documents
    })
})
```

Looking for another stack? We will try to add as many examples as we can. If you have written a cool app and want us to showcase it, let us know!

US election analysis

Analysis of the 2012 US presidential elections.
Technologies:

- JavaScript
- Data Explorer
- `groupby`, `eqJoin`, map/reduce

Go to the tutorial »

Molly.js

A WebGL molecule viewer (by @psb).
Technologies:

- JavaScript driver with Node.js
- Express
- WebGL

See the code »

Superheroes tutorial

Simple queries on a superheroes dataset.
Technologies:

- Python driver
- Simple creation and insertion
- Filtering based on arrays

Go to the tutorial »

Todo list in Ember.js

Ember.js Todo List in the spirit of TodoMVC.
Technologies:

- Python driver
- Bottle
- Ember.js

See the annotated source »

Asynchronous chat

The classic chat with Node.js.
Technologies:

- JavaScript driver with Node.js
- Express
- Passport

See the annotated source »

Pastie app

A simple Pastie app.
Technologies:

- Ruby driver
- Sinatra
- `filter`, `pluck`, `order_by`

See the annotated source »

Todo list in Backbone

Backbone.js Todo List based on TodoMVC.
Technologies:

- Python driver
- Flask
- Backbone.js

See the annotated source »

Blog example app

A web.py blog application.
Technologies:

- Python driver
- web.py
- Basic updates and filtering

See the annotated source »

db.js

todo.py

¶

A demo web application in the spirit of TodoMVC showing how to use **RethinkDB as a backend for Flask and Backbone.js applications**.
For details about the complete stack, installation, and running the app see the README.

```
import argparse
import json
import os

from flask import Flask, g, jsonify, render_template, req

import rethinkdb as r
from rethinkdb.errors import RqlRuntimeError, RqlDriverE
```

¶

Connection details

¶

We will use these settings later in the code to connect to the RethinkDB server.

```
RDB_HOST =  os.environ.get('RDB_HOST') or 'localhost'
RDB_PORT = os.environ.get('RDB_PORT') or 28015
TODO_DB = 'todoapp'
```

¶

Setting up the app database

¶

The app will use a table `todos` in the database specified by the `TODO_DB` variable. We'll create the database and table here using `db_create` and `table_create` commands.

```
def dbSetup():
   connection = r.connect(host=RDB_HOST, port=RDB_PORT)
    try:
     r.db_create(TODO_DB).run(connection)
     r.db(TODO_DB).table_create('todos').run(connectio
     print 'Database setup completed. Now run the app wit
    except RqlRuntimeError:
     print 'App database already exists. Run the app with
    finally:
        connection.close()


app = Flask(__name__)
app.config.from_object(__name__)
```

¶

Managing connections

todo.py

¶

The pattern we're using for managing database connections is to have **a connection per request**. We're using Flask's `@app.before_request` and `@app.teardown_request` for opening a database connection and closing it respectively.

```
@app.before_request
def before_request():
    try:
     g.rdb_conn = r.connect(host=RDB_HOST, port=RDB_POR
    except RqlDriverError:
     abort(503, "No database connection could be establi

@app.teardown_request
def teardown_request(exception):
    try:
        g.rdb_conn.close()
    except AttributeError:
        pass
```

¶

Listing existing todos

¶

To retrieve all existing tasks, we are using `r.table` command to query the database in response to a GET request from the browser. When `table(table_name)` isn't followed by an additional command, it returns all documents in the table. Running the query returns an iterator that automatically streams data from the server in efficient batches.

```
@app.route("/todos", methods=['GET'])
def get_todos():
   selection = list(r.table('todos').run(g.rdb_conn))
    return json.dumps(selection)
```

¶

Creating a todo

todo.py

¶

We will create a new todo in response to a POST request to `/todos` with a JSON payload using `table.insert`.
The `insert` operation returns a single object specifying the number of successfully created objects and their corresponding IDs:

```
@app.route("/todos", methods=['POST'])
def new_todo():
   inserted = r.table('todos').insert(request.json).run
   return jsonify(id=inserted['generated_keys'][0])
```

```
{
  "inserted": 1,
  "errors": 0,
  "generated_keys": [
   "773666ac-841a-44dc-97b7-b6f3931e9b9f"
  ]
}
```

¶

Retrieving a single todo

¶

Every new task gets assigned a unique ID. The browser can retrieve a specific task by GETing `/todos/<todo_id>`. To query the database for a single document by its ID, we use the `get` command. Using a task's ID will prove more useful when we decide to update it, mark it completed, or delete it.

```
@app.route("/todos/<string:todo_id>", methods=['GET'])
def get_todo(todo_id):
   todo = r.table('todos').get(todo_id).run(g.rdb_conn)
    return json.dumps(todo)
```

¶

Editing/Updating a task

¶

Updating a todo (editing it or marking it completed) is performed on a `PUT` request. To save the updated todo we'll do a `replace`.

```
@app.route("/todos/<string:todo_id>", methods=['PUT'])
def update_todo(todo_id):
   return jsonify(r.table('todos').get(todo_id).replace
```

todo.py

¶

If you'd like the update operation to happen as the result of a `PATCH` request (carrying only the updated fields), you can use the `update` command, which will merge the JSON object stored in the database with the new one.

```
@app.route("/todos/<string:todo_id>", methods=['PATCH']
def patch_todo(todo_id):
   return jsonify(r.table('todos').get(todo_id).update(
```

¶

Deleting a task

¶

To delete a todo item we'll call a `delete` command on a `DELETE /todos/<todo_id>` request.

```
@app.route("/todos/<string:todo_id>", methods=['DELETE'
def delete_todo(todo_id):
   return jsonify(r.table('todos').get(todo_id).delete(

@app.route("/")
def show_todos():
   return render_template('todo.html')


if __name__ == "__main__":
   parser = argparse.ArgumentParser(description='Run the
   parser.add_argument('--setup', dest='run_setup', acti

    args = parser.parse_args()
    if args.run_setup:
        dbSetup()
    else:
        app.run(debug=True)
```

todo.py

¶

Best practices

Managing connections: a connection per request The RethinkDB server doesn't use a thread-per-connnection approach so opening connections per request will not slow down your database.

Fetching multiple rows: batched iterators When fetching multiple rows from a table, RethinkDB returns a batched iterator initially containing a subset of the complete result. Once the end of the current batch is reached, a new batch is automatically retrieved from the server. From a coding point of view this is transparent:

```
for result in r.table('todos').run(g.rdb_conn):
    print result
```

`replace` vs `update` Both `replace` and `update` operations can be used to modify one or multiple rows. Their behavior is different:

- `replace` will completely replace the existing rows with new values
- `update` will merge existing rows with the new values

¶

model.py

¶

The web.py (really basic) blog example using **RethinkDB as the backend for web.py applications**.
For details about the complete stack, installation, and running the app see the README.

```
import os, socket, sys, time

import web

from contextlib import contextmanager

import rethinkdb as r
from rethinkdb.errors import RqlRuntimeError
```

¶

Connection details

¶

We will use these settings later in the code to connect to the RethinkDB server.

```
RDB_CONFIG = {
  'host' : os.getenv('RDB_HOST', 'localhost'),
  'port' : os.getenv('RDB_PORT', 28015),
  'db'   : os.getenv('RDB_DB', 'webpy'),
  'table': os.getenv('RDB_TABLE', 'blogposts')
}
```

¶

The `Connection` object returned by `r.connect` is a context manager that can be used with the `with` statements.

```
def connection():
 return r.connect(host=RDB_CONFIG['host'], port=RDB_CON
```

¶

Listing existing posts

¶

To retrieve all existing tasks, we are using the `r.table` command to query the database in response to a GET request from the browser. We also `order_by` the `posted_at` attribute in a descending manner. Running the query returns an iterator that automatically streams data from the server in efficient batches.

```
def get_posts():
  with connection() as conn:
   return r.table(RDB_CONFIG['table']).order_by(r.desc(
```

model.py

¶

Creating a new post

¶

We create a new blog entry using `insert`.
The `insert` operation returns a single object specifying the number of successfully created objects and their corresponding IDs:

```
{
  "inserted": 1,
  "errors": 0,
  "generated_keys": [
   "b3426201-9992-4a21-ab84-974603657671"
  ]
}
```

```
def new_post(title, text):
  new_post = {'title': title,
    'content': text,
    'posted_at': time.time(),
   'last_modified': time.time()
  }
  with connection() as conn:
   result = r.table(RDB_CONFIG['table']).insert(new_pos
    if result['inserted'] == 1:
    new_post['id'] = result['generated_keys'][0]
    return new_post
    else:
      return None
```

¶

Retrieving a single post

¶

Every new post gets assigned a unique ID. The browser can retrieve a specific task by GETing `/view/<post_id>`. To query the database for a single document by its ID, we use the `get` command.

```
def get_post(id):
  with connection() as conn:
   return r.table(RDB_CONFIG['table']).get(id).run(conn
```

¶

Updating a post

¶

To update the post we'll use the `update` command, which will merge the JSON object stored in the database with the new one.
The `update` operation returns an object specifying how many rows have been updated.

```
def update_post(id, title, text):
  with connection() as conn:
   result = r.table(RDB_CONFIG['table']).get(id) \
    .update({'title': title, 'content': text, 'last_modi
      .run(conn)
   return result.get('modified', 0) == 1
```

model.py

¶

Deleting a post

¶

To delete a post we'll call a `delete` command.
The `delete` operation returns an object specifying how many rows have been deleted.

```
def del_post(id):
  with connection() as conn:
   result = r.table(RDB_CONFIG['table']).get(id).delete
   return result.get('deleted', 0) == 1
```

¶

Database setup

¶

The app will use the table `blogposts` in the database `webpy`. You can override these defaults by defining the `RDB_DB` and `RDB_TABLE` env variables.
We'll create the database and table here using `db_create` and `table_create` commands.

```
def dbSetup():
   connection = r.connect(host=RDB_CONFIG['host'], port=
     try:
      r.db_create(RDB_CONFIG['db']).run(connection)
      r.db(RDB_CONFIG['db']).table_create(RDB_CONFIG['t
      print 'Database setup completed. Now run the app wit
     except RqlRuntimeError:
      print 'App database already exists. Run the app like
     finally:
         connection.close()
```

model.py

¶

Best practices

Managing connections: a connection per request The RethinkDB server doesn't use a thread-per-connnection approach so opening connections per request will not slow down your database.

Fetching multiple rows: batched iterators When fetching multiple rows from a table, RethinkDB returns a batched iterator initially containing a subset of the complete result. Once the end of the current batch is reached, a new batch is automatically retrieved from the server. From a coding point of view this is transparent:

```
for result in r.table('blogposts').run(conn):
    print result
```

`update` vs `replace` Both `update` and `replace` operations can be used to modify one or multiple rows. Their behavior is different:

- `update` will merge existing rows with the new values
- `replace` will completely replace the existing rows with new values

¶

todo.py

¶

A demo web application in the spirit of TodoMVC showing how to use **RethinkDB as a backend for Bottle and Ember.js applications**.
For details about the complete stack, installation, and running the app see the README.

```
import argparse
import json
import os
import socket

import bottle
from bottle import static_file, request

import rethinkdb as r
from rethinkdb.errors import RqlRuntimeError, RqlDriverE
```

¶

Connection details

¶

We will use these settings later in the code to connect to the RethinkDB server.

```
RDB_HOST =  os.getenv('RDB_HOST', 'localhost')
RDB_PORT = os.getenv('RDB_PORT', 28015)
TODO_DB = os.getenv('TODO_DB', 'todoapp')
```

¶

Setting up the app database

¶

The app will use a table `todos` in the database specified by the `TODO_DB` variable (defaults to `todoapp`). We'll create the database and table here using `db_create` and `table_create` commands.

```
def dbSetup():
   connection = r.connect(host=RDB_HOST, port=RDB_PORT)
    try:
     r.db_create(TODO_DB).run(connection)
     r.db(TODO_DB).table_create('todos').run(connectio
     print 'Database setup completed. Now run the app wit
    except RqlRuntimeError:
     print 'App database already exists. Run the app with
    finally:
        connection.close()
```

¶

Managing connections

todo.py

¶

The pattern we're using for managing database connections is to have **a connection per request**. We're using Bottle's `@bottle.hook('before_request')` and `@bottle.hook('after_request')` for opening a database connection and closing it respectively.

```
@bottle.hook('before_request')
def before_request():
    if request.path.startswith('/static/'):
        return
    try:
        bottle.local.rdb_connection = r.connect(RDB_HOST,
    except RqlDriverError:
        bottle.abort(503, "No database connection could be

@bottle.hook('after_request')
def after_request():
    if request.path.startswith('/static/'):
        return
    bottle.local.rdb_connection.close()
```

¶

Listing existing todos

¶

To retrieve all existing tasks, we use the `r.table` command to query the database in response to a GET request from the browser. When `table(table_name)` isn't followed by an additional command, it returns all documents in the table. Running the query returns an iterator that automatically streams data from the server in efficient batches.

¶

```
@bottle.get("/todos")
def get_todos():
    selection = list(r.table('todos').run(bottle.local.r
    return json.dumps({'todos': selection})
```

Creating a todo

todo.py

¶

We will create a new todo in response to a POST request to `/todos` with a JSON payload using `table.insert`.
The `insert` operation returns a single object specifying the number of successfully created objects and their corresponding IDs:

```
@bottle.post("/todos")
def new_todo():
    todo = request.json['todo']
   inserted = r.table('todos').insert(todo).run(bottle.
   todo['id'] = inserted['generated_keys'][0]
   return json.dumps({'todo': todo})
```

```
{
  "inserted": 1,
  "errors": 0,
  "generated_keys": [
   "773666ac-841a-44dc-97b7-b6f3931e9b9f"
  ]
}
```

¶

Retrieving a single todo

¶

Every new task gets assigned a unique ID. The browser can retrieve a specific task by GETing `/todos/<todo_id>`. To query the database for a single document by its ID, we use the `get` command. Using a task's ID will prove more useful when we decide to update it, mark it completed, or delete it.

```
@bottle.get("/todos/<todo_id>")
def get_todo(todo_id):
   todo = r.table('todos').get(todo_id).run(bottle.loca
   return json.dumps({'todo': todo})
```

¶

Editing/Updating a task

¶

Updating a todo (editing it or marking it completed) is performed on a `PUT` request. To save the updated todo we'll do a `replace`.

```
@bottle.put("/todos/<todo_id>")
def update_todo(todo_id):
    todo = {'id': todo_id}
   todo.update(request.json['todo'])
   return json.dumps(r.table('todos').get(todo_id).repl
```

todo.py

¶

If you'd like the update operation to happen as the result of a `PATCH` request (carrying only the updated fields), you can use the `update` command, which will merge the JSON object stored in the database with the new one.

```
@bottle.route("/todos/<todo_id>", method='PATCH')
def patch_todo(todo_id):
   return json.dumps(r.table('todos').get(todo_id).upda
```

¶

Deleting a task

¶

To delete a todo item we'll call a `delete` command on a `DELETE /todos/<todo_id>` request.

```
@bottle.delete("/todos/<todo_id>")
def delete_todo(todo_id):
   return json.dumps(r.table('todos').get(todo_id).dele

@bottle.get("/")
def show_todos():
   return static_file('todo.html', root='templates/', mi

@bottle.route('/static/<filename:path>', method='GET')
@bottle.route('/static/<filename:path>', method='HEAD')
def send_static(filename):
   return static_file(filename, root='static/')

if __name__ == "__main__":
   parser = argparse.ArgumentParser(description='Run the
   parser.add_argument('--setup', dest='run_setup', acti

    args = parser.parse_args()
    if args.run_setup:
        dbSetup()
    else:
     bottle.run(host='localhost', port=5000, debug=True
```

todo.py

¶

Best practices

Managing connections: a connection per request The RethinkDB server doesn't use a thread-per-connnection approach so opening connections per request will not slow down your database.

Fetching multiple rows: batched iterators When fetching multiple rows from a table, RethinkDB returns a batched iterator initially containing a subset of the complete result. Once the end of the current batch is reached, a new batch is automatically retrieved from the server. From a coding point of view this is transparent:

```
for result in r.table('todos').run(connection):
    print result
```

`replace` vs `update` Both `replace` and `update` operations can be used to modify one or multiple rows. Their behavior is different:

- `replace` will completely replace the existing rows with new values
- `update` will merge existing rows with the new values

¶

repasties.rb

¶

A simple Pastie-like app inspired by Nick Plante's toopaste project showing how to use **RethinkDB as a backend for Sinatra applications**.

```
require 'sinatra'
require 'rethinkdb'
```

¶

Connection details

¶

We will use these settings later in the code to connect to the RethinkDB server.

```
RDB_CONFIG = {
  :host => ENV['RDB_HOST'] || 'localhost',
  :port => ENV['RDB_PORT'] || 28015,
  :db  => ENV['RDB_DB']  || 'repasties'
}
```

¶

A friendlly shortcut for accessing ReQL functions

```
r = RethinkDB::RQL.new
```

¶

Setting up the database

repasties.rb

¶

The app will use a table `snippets` in the database defined by the environment variable `RDB_DB` (defaults to `repasties`).
We'll create the database and the table here using `db_create` and `table_create` commands.

```
configure do
  set :db, RDB_CONFIG[:db]
  begin
   connection = r.connect(:host => RDB_CONFIG[:host], :p
  rescue Exception => err
   puts "Cannot connect to RethinkDB database #{RDB_CONFI
     Process.exit(1)
  end

  begin
   r.db_create(RDB_CONFIG[:db]).run(connection)
  rescue RethinkDB::RqlRuntimeError => err
   puts "Database `repasties` already exists."
  end

  begin
   r.db(RDB_CONFIG[:db]).table_create('snippets').run(c
  rescue RethinkDB::RqlRuntimeError => err
   puts "Table `snippets` already exists."
  ensure
    connection.close
  end
end
```

¶

Managing connections

¶

The pattern we're using for managing database connections is to have **a connection per request**.
We're using Sinatra's `before` and `after` for opening a database connection and closing it respectively.

```
before do
  begin
```

repasties.rb

¶

When openning a connection we can also specify the database:

```
  @rdb_connection = r.connect(:host => RDB_CONFIG[:host
 rescue Exception => err
  logger.error "Cannot connect to RethinkDB database #{R
  halt 501, 'This page could look nicer, unfortunately th
 end
end
```

¶

After each request we close the database connection.

```
after do
 begin
  @rdb_connection.close if @rdb_connection
 rescue
  logger.warn "Couldn't close connection"
 end
end

get '/' do
 @snippet = {}
 erb :new
end
```

¶

We create a new snippet in response to a POST request using `table.insert`.

```
post '/' do
 @snippet = {
  :title => params[:snippet_title],
  :body  => params[:snippet_body],
  :lang  => (params[:snippet_lang] || 'text').downcase,
 }
 if @snippet[:body].empty?
   erb :new
 end

 if @snippet[:title].empty?
  @snippet[:title] = @snippet[:body].scan(/\w+/)[0..2]
   erb :new
 end

 @snippet[:created_at] = Time.now.to_i
 @snippet[:formatted_body] = pygmentize(@snippet[:body]

 result = r.table('snippets').insert(@snippet).run(@rdb
```

repasties.rb

¶

The `insert` operation returns a single object specifying the number of successfully created objects and their corresponding IDs

```
{
  "inserted": 1,
  "errors": 0,
  "generated_keys": [
   "fcb17a43-cda2-49f3-98ee-1efc1ac5631d"
  ]
}
```

```
  if result['inserted'] == 1
   redirect "/#{result['generated_keys'][0]}"
  else
    logger.error result
    redirect '/'
  end
end
```

¶

Every new snippet gets assigned automatically a unique ID. The browser can retrieve a specific snippet by GETing `/<snippet_id>`. To query the database for a single document by its ID, we use the `get` command.

```
get '/:id' do
 @snippet = r.table('snippets').get(params[:id]).run(@r

  if @snippet
   @snippet['created_at'] = Time.at(@snippet['created_at
    erb :show
  else
    redirect '/'
  end
end
```

¶

Retrieving the latest `max_results` (default 10) snippets by their language by chaining together `filter`, `pluck`, and `order_by`. All chained operations are executed on the database server and the results are returned as a batched iterator.

```
get '/lang/:lang' do
 @lang = params[:lang].downcase
 max_results = params[:limit] || 10
  results = r.table('snippets').
          filter('lang' => @lang).
          pluck('id', 'title', 'created_at').
          order_by(r.desc('created_at')).
             limit(max_results).
           run(@rdb_connection)

  @snippets = results.to_a
 @snippets.each { |s| s['created_at'] = Time.at(s['creat
  erb :list
end
```

repasties.rb

¶

List of languages for which syntax highlighting is supported.

```
SUPPORTED_LANGUAGES = ['Ruby', 'Python', 'Javascript', '
  'AppleScript', 'Awk', 'C', 'C++', 'Clojure',
  'CoffeeScript', 'Lisp', 'Erlang', 'Fortran', 'Groovy',
  'Haskell', 'Io', 'Java', 'Lua', 'Objective-C',
  'OCaml', 'Perl', 'Prolog', 'Scala', 'Smalltalk'].sort
```

¶

A Sinatra helper to expose the list of languages to views.

```
helpers do
  def languages
    SUPPORTED_LANGUAGES
  end
end
```

repasties.rb

¶
Code is run through Pygments for syntax highlighting. If it's not installed, locally, use a webservice http://pygments.appspot.com/. (code inspired by rocco.rb)

```
unless ENV['PATH'].split(':').any? { |dir| File.executab
 warn "WARNING: Pygments not found. Using webservice."
 PYGMENTIZE=false
else
 PYGMENTIZE=true
end


def pygmentize(code, lang)
 if lang.eql? 'text'
  return code
 end
 lang.downcase!
 if PYGMENTIZE
  highlight_pygmentize(code, lang)
 else
  highlight_webservice(code, lang)
 end
end

def highlight_pygmentize(code, lang)
 code_html = nil
 open("|pygmentize -l #{lang} -f html -O encoding=utf-8,
  pid =
   fork {
    fd.close_read
    fd.write code
    fd.close_write
    exit!
   }
  fd.close_write
  code_html = fd.read
  fd.close_read
  Process.wait(pid)
 end

 code_html
end

require 'net/http'

def highlight_webservice(code, lang)
 url = URI.parse 'http://pygments.appspot.com/'
 options = {'lang' => lang, 'code' => code}
 Net::HTTP.post_form(url, options).body
end
```

repasties.rb

¶

Best practices

Managing connections: a connection per request The RethinkDB server doesn't use a thread-per-connnection approach so opening connections per request will not slow down your database.

Fetching multiple rows: batched iterators When fetching multiple rows from a table, RethinkDB returns a batched iterator initially containing a subset of the complete result. Once the end of the current batch is reached, a new batch is automatically retrieved from the server. From a coding point of view this is transparent:

```
r.table('todos').run(g.rdb_conn).each do |result|
    print result
end
```

¶

Credit

¶

- This sample app was inspired by Nick Plante's toopaste project.
- The snippets of code used for syntax highlighting are from Ryan Tomayko's rocco.rb project.
- Snippets code highlighting is done using Pygments or the Pygments web service
- The Solarized dark Pygments stylesheet was created by Zameer Manji

repasties.rb

¶

License
¶
This demo application is licensed
under the MIT license:
http://opensource.org/licenses/mit-license.php

Add your library: Have you written a cool library related RethinkDB and want us to showcase it? Shoot us an email at info@rethinkdb.com.

Node.js libraries

Drivers and extensions

- rethinkdbdash by @neumino
  An alternative Node.js driver with native promises and a connection pool.
- RQL Promise by @guillaumervls Wraps the RethinkDB driver with when to return promises.
- rethinkdb-co by @hden
  Allows using ECMAScript 6 generators with RethinkDB callbacks.
- rdb-cursor-stream by @guillaumervls Replaces cursors with streams.
- connect-rethinkdb by @guillaumervls A RethinkDB session store for Connect, similar to connect-redis.
- Rethinkdb-pool by @hden
  Connection pool for RethinkDB connections.

ORMs

- reheat by @jmdobry
  JavaScript ORM for RethinkDB with promises.
- thinky by @neumino
  JavaScript ORM for RethinkDB

- JugglingDB-RethinkDB by @fuwaneko
  A RethinkDB adapter for JugglingDB, a multi-database ORM for Node.js.
- Osmos by @mtabini
  A store-agnostic object data mapper for Node.js with support for RethinkDB.

Integrations

- koa-rethinkdb by @hden
  Koa middleware that automatically manages connections via a connection pool.

Python libraries

ORMs

- rwrapper by @dparlevliet
  An ORM designed to emulate the most common usages of Django's database abstraction.
- pyRethinkORM by @JoshAshby
  A Python ORM for RethinkDB.

Integration

- flask-rethinkdb by @linkyndy
  A Flask extension that adds RethinkDB support (also see the pip package).

Ruby libraries

ORMs

- NoBrainer by @nviennot
  A Ruby ORM designed for RethinkDB.

Tools and utilities

Administration

- Chateau by @neumino
  An administrative interface for your data (like phpMyAdmin for RethinkDB).

- Methink by @Calder
  A MySQL to RethinkDB migration script.
- rethink-miner by @baruch
  Stores queries and their results, and displays them from a web interface.
- recli
  CLI to run ReQL queries in JavaScript.
- rethinkdb-cli
  CLI to run ReQL queries in Ruby.

For driver developers

- rethinkdb-driver-development by @neumino
  A tool to retrieve the query objects, protobuf messages and responses.

Deployment tools

- Rethinkdb-vagrant by @RyanAmos
  Lets you install RethinkDB using Vagrant.
- puppet-rethinkdb by @tmont
  A Puppet module for RethinkDB.
- chef-rethinkdb by @AVVSDevelopment
  A RethinkDB cookbook for Chef deployment.
- box-rethinkdb
  Wercker box for RethinkDB, by @mies.
- Dockerfile/rethinkdb by @pilwon
  Trusted Docker build and instruction for deploying a RethinkDB cluster.
- Dockerfiles-examples by @kstaken
  Includes scripts for building an image for Docker with RethinkDB (and other things).
- Docker-cookbooks by @crosbymichael
  A collection of Dockerfiles and configurations to build images for RethinkDB.

There are two ways to model relationships between documents in RethinkDB:

- By using **embedded arrays**.
- By linking documents stored in **multiple tables** (similarly to traditional relational database systems).

Let's explore advantages and disadvantages of each approach. We'll use a simple blog database that stores information about authors and their posts to illustrate each approach.

Using embedded arrays

We can model the relationship between authors and posts by using embedded arrays as follows. Consider this example document in the table `authors`:

```
{
  "id": "7644aaf2-9928-4231-aa68-4e65e31bf219",
  "name": "William Adama", "tv_show": "Battlestar Galactica",
  "posts": [
   {"title": "Decommissioning speech", "content": "The Cylon War is long over..."},
   {"title": "We are at war", "content": "Moments ago, this ship received..."},
   {"title": "The new Earth", "content": "The discoveries of the past few days..."}
  ]
}
```

The `authors` table contains a document for each author. Each document contains information about the relevant author and a field `posts` with an array of posts for that author. In this case the query to retrieve all authors with their posts is really simple:

```
# Retrieve all authors with their posts
r.db("blog").table("authors").run()

# Retrieve a single author with her posts
r.db("blog").table("authors").get(AUTHOR_ID).run()
```

Advantages of using embedded arrays:

- The queries for accessing authors and posts tend to be simpler than when using multiple tables.
- When using this approach, the data is often colocated on disk. If you have a dataset that doesn't fit into RAM, the data can be loaded from disk faster.
- With this approach, any update to the authors document atomically updates both the author data and the posts data.

Disadvantages of using embedded arrays:

- Any operation on an author document requires loading all posts into memory. Any update to the document requires rewriting the full array to disk.
- Because of the previous limitation, it's best to keep the size of the `posts` array to no more than a few hundred documents.

Linking documents in multiple tables

You can use a data modeling technique similar to the one used in relational database systems, and create two tables to store your data. A typical document in the `authors` table would look like this:

```
{
  "id": "7644aaf2-9928-4231-aa68-4e65e31bf219",
  "name": "William Adama",
  "tv_show": "Battlestar Galactica"
}
```

A typical document in the `posts` table would look like this:

```
{
  "id": "064058b6-cea9-4117-b92d-c911027a725a",
  "author_id": "7644aaf2-9928-4231-aa68-4e65e31bf219",
  "title": "Decommissioning speech",
  "content": "The Cylon War is long over..."
}
```

In this example, every post contains an `author_id` field, that links each post to its author. We can retrieve all posts for a given author as follows:

```
# If we have a secondary index on `author_id` in the table `posts`
r.db("blog").table("posts").
 get_all("7644aaf2-9928-4231-aa68-4e65e31bf219", index="author_id").
  run()

# If we didn't build a secondary index on `author_id`
r.db("blog").table("posts").
  filter({"author_id": "7644aaf2-9928-4231-aa68-4e65e31bf219"}).
  run()
```

To get all posts for a given author, we can use the `eq_join` command, similarly to how we'd do a `JOIN` in a relational system. Here is how we could get all posts along with the author information for William Adama:

```
# In order for this query to work, we need to have a secondary index
# on the `author_id` field of the table `posts`.
r.db("blog").table("authors").getAll("7644aaf2-9928-4231-aa68-4e65e31bf219").eq_join(
    'id',
    r.db("blog").table("authors"),
    index='author_id'
).zip().run()
```

Note that the values for `author_id` correspond to the `id` field of the author, which allows us to link the documents.

Advantages of using multiple tables:

- Operations on authors and posts don't require loading the data for every post for a given author into memory.
- There is no limitation on the number of posts, so this approach is more suitable for large amounts of data.

Disadvantages of using multiple tables:

- The queries linking the data between the authors and their posts tend to more complicated.
- With this approach you cannot atomically update both the author data and and the posts data.

Read more

If you aren't sure which schema to use, ask us on Stack Overflow or on IRC. For more detailed information, take a look at the API documentation for the join commands.

- eq_join
- inner_join
- outer_join
- zip

Startup with init.d

On Linux, RethinkDB packages automatically install an init script at `/etc/init.d/rethinkdb` and add default run-level entries.

Quick setup

To get started, copy the the example config file from `/etc/rethinkdb/default.conf.sample` into the instances directory (`/etc/rethinkdb/instances.d/`) and restart the init.d script:

```
sudo cp /etc/rethinkdb/default.conf.sample /etc/rethinkdb/instances.d/instance1.conf
sudo vim /etc/rethinkdb/instances.d/instance1.conf # edit the options
sudo /etc/init.d/rethinkdb restart
```

The basic setup is complete — **you've now got a working server!**

The init.d script looks for filenames ending in `.conf` in `/etc/rethinkdb/instances.d/`, and starts an instance of RethinkDB for each config file found in this directory. The packages do not ship with a default config file, so if you install RethinkDB, it will not automatically be run on system startup until you add a config file to `/etc/rethinkdb/instances.d/`.

Multiple instances

The init.d script supports starting multiple instances on the same machine via multiple `.conf` files in `/etc/rethinkdb/instances.d`. Note that the init.d script produces a feedback line for each registered instance when queried. This is not standard behavior for an init.d script, so if you have a tool that depends upon standard init.d script output, you might need to limit each machine to only one RethinkDB instance in `/etc/rethinkdb/instances.d`.

The `http-port`, `driver-port` and `cluster-port` options are mandatory when defining multiple instances.

Installing from source

If you compiled from source, you can get the init.d script from here on Github. You can get the sample config file on Github from here.

Startup with systemd

Basic setup

Add the file `/usr/lib/tmpfiles.d/rethinkdb.conf` with the content:

```
d /run/rethinkdb 0755 rethinkdb rethinkdb -
```

Then add one more file `/usr/lib/systemd/system/rethinkdb@.service`

```
[Unit]
Description=RethinkDB database server for instance '%i'

[Service]
User=rethinkdb
Group=rethinkdb
ExecStart=/usr/bin/rethinkdb serve --config-file /etc/rethinkdb/instances.d/%i.conf
KillMode=process
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

The `chmod` for the two files should be 644.

Starting RethinkDB instances

First, create the RethinkDB data directory with the following command:

```
rethinkdb create -d /path/to/your/rethinkdb/directory
```

Then, download the default.conf.sample, and move it into `/etc/rethinkdb/instances.d/<name_instance`

Finally, modify the .conf file with your desired settings and then run:

```
sudo systemctl enable rethinkdb@<name_instance>
sudo systemctl start rethinkdb@<name_instance>
```

You've now got a working server!

Multiple instances

Since `systemd` supports multiple instances, starting a new instance of RethinkDB only requires creating another .conf file.

Configuration options

The `.conf` file includes a number of options for exclusive to the init script. The rest of the options are exactly the same as the ones that go on the command line to the RethinkDB server. For more details about these options run `rethinkdb help`.

Supported options

For some of the options below, the default value depends on `<name>`, the name of the config file without the `.conf` extension.

- `runuser` and `rungroup` — specifies which user and group should be used launch the rethinkdb process. **Defaults**: `rethinkdb` and `rethinkdb`.
- `pid-file` — the location of the file with the RethinkDB instance process ID (used by the init script to communicate with the server process).
  Default: `/var/run/rethinkdb/<name>/pid_file`
- `directory` — the data directory where database tables will be stored. This location must be readable and writable by the user or group (or both) specified by `runuser` and `rungroup`. Note, it is best to create the database manually via `rethinkdb create --directory ...` as `runuser` or `rungroup` before enabling auto-start.
  Default: `/var/lib/rethinkdb/<name>/`
- `http-port`, `driver-port`, and `cluster-port` — the web UI port (default `8080`), the client driver port (default `28015`), and intracluster traffic port (default `29015`), respectively.
- `bind` — by default, the server process binds only to loop-back interfaces (`127.*.*.*`) and thus may not be accessible over the network. The `bind` option allows the server process to bind to additional interfaces. You can either specify IPv4 addresses of network interfaces or simply use `all` to bind to all interfaces.
- `join` — rethinkdb allows you to incrementally build a cluster by joining new nodes to others that are already running. The `join` option specifies which rethinkdb node to join via `host:port`. For example, `join=newton:29015` will join the node on host `newton` at intracluster port `29015`. You can also specify multiple `join` options in case some of your nodes are unreachable at the time of startup.

Troubleshooting

Seeing a 'received invalid clustering header' message? RethinkDB uses three ports to operate — the HTTP web UI port, the client drivers port, and the intracluster traffic port. You can connect the browser to the web UI port to administer the cluster right from your browser, and connect the client drivers to the client driver port to run queries from your application. If you're running a cluster, different RethinkDB nodes communicate with each other via the intracluster traffic port.

The message `received invalid clustering header` means there is a port mismatch, and something is connecting to the wrong port. For example, it's common to get this message if you accidently point the browser or connect the client drivers to the intracluster traffic port.

RethinkDB ships with `dump` and `restore` commands that allow easily doing hot backups on a live cluster. The dump and restore commands operate on tar.gz archives of JSON documents (along with additional table metadata). You can run `rethinkdb dump --help` and `rethinkdb restore --help` for more information.

Backup

Back up your data as follows:

```
# Dump the data from a RethinkDB cluster (placed in a file
# rethinkdb_dump_DATE_TIME.tar.gz by default)
$ rethinkdb dump -c HOST:PORT
```

Since the backup process is using client drivers, it automatically takes advantage of the MVCC functionality built into RethinkDB. It will use some cluster resources, but will not lock out any of the clients, so you can safely run it on a live cluster.

Restore

You can reimport the backup into a running cluster as follows:

```
# Reimport an earlier dump
$ rethinkdb restore -c HOST:PORT rethinkdb_dump_DATE_TIME.tar.gz
```

The best way to secure a RethinkDB cluster is to run it on a protected network that doesn't allow access from the outside world. However, this may not always be feasible. For example, cloud deployments often require access from wide area networks.

The following is a list of techniques that help mitigate the risk of attacks for RethinkDB setups that require access from the outside world.

Securing the web interface

First, protect the web interface port so that it cannot be accessed from the outside world. On Unix-based systems, you can use `iptables` to block the port as follows:

```
sudo iptables -A INPUT -i eth0 -p tcp --dport 8080 -j DROP
sudo iptables -I INPUT -i eth0 -s 127.0.0.1 -p tcp --dport 8080 -j ACCEPT
```

Note: You may have to replace `eth0` and `8080` above if you are using another interface or not using the default web interface port.

Now, use one of the following two methods to enable secure access.

Via a socks proxy

Once you block the web interface port in the step above, the easiest way to access it is to use ssh to set up a socks proxy. Run the following command on your local machine (not the one running RethinkDB):

```
ssh -D 3000 USERNAME@HOST
```

Where,

- `HOST` is the ip of any machine on your RethinkDB cluster.
- `3000` can be changed to any port that is available on your local machine.

Then open your browser:

- **If you're using Chrome**, go to *Settings > Advanced settings > Network > Change proxy settings*, and set the *Network proxy* option to manual mode with the following settings:
  - Host: `localhost`
  - Port: `3000`
  - Ignored host: (remove everything)
- **If you are using Firefox**, go to *Edit > Preferences*. Then click on *Advanced > Network > Settings* and create a manual proxy configuration with these settings:
  - Socks host: `localhost`
  - Port: `3000`
  - Check socks v5
  - No proxy for: (remove everything)

You can now visit `localhost:8080` to see the RethinkDB web admin.

Via a reverse proxy

You can use a reverse http proxy to allow access to the web interface from other machines. Most web servers (such as apache or nginx) support this feature. In the following example we'll use apache to set up a reverse proxy.

First, install apache with relevant modules as follows:

```
sudo apt-get install libapache2-mod-proxy-html
sudo a2enmod proxy
sudo a2enmod proxy_http
```

Then create a new virtual host:

```
<VirtualHost *:80>
    ServerName domain.net

    ProxyRequests Off

    <Proxy *>
        Order deny,allow
        Allow from all
        AuthType Basic
        AuthName "Password Required"
        AuthUserFile password.file
        AuthGroupFile group.file
        Require group dbadmin
    </Proxy>

    ProxyErrorOverride On
    ProxyPass   /rethinkdb_admin/   http://localhost:8080/
    ProxyPassReverse   /rethinkdb_admin/   http://localhost:8080/

</VirtualHost>
```

Create the password file in `/etc/apache2/`:

```
htpasswd.exe -c password.file username
```

Almost done. All we have to do now is create a file `group.file` with this the following content:

```
dbadmin: username
```

You can now access the web interface using the following URL: `http://HOST/rethinkdb_admin`.

Securing the driver port

Using the RethinkDB authentication system

RethinkDB allows setting an authentication key using the command line interface. Once you set the authentication key, client drivers will be required to pass the key to the server in order to connect.

Note: the authentication key will be transmitted to the RethinkDB server in plain text. This may be sufficient to thwart basic attacks, but is vulnerable to more sophisticated man-in-the-middle attacks.

First, open the CLI:

```
rethinkdb admin --join HOST:29015
```

Then execute the following command:

```
set auth <authentication_key>
```

You can set the `authentication_key` option to any key of your choice.

You can now connect to the driver port from any network, but must provide the required authentication key. For instance, in JavaScript you would connect as follows:

```
r.connect({host: HOST, port: PORT, authKey: <authentication_key>},
    function(error, connection) { ... })
```

Using SSH tunneling

First, protect the driver port so that it cannot be accessed from the outside world. On unix-based systems, you can use `iptables` to block the port as follows:

```
sudo iptables -A INPUT -i eth0 -p tcp --dport 28015 -j DROP
sudo iptables -I INPUT -i eth0 -s 127.0.0.1 -p tcp --dport 28015 -j ACCEPT
```

Note: You may have to replace `eth0` and `28015` above if you are using another interface or not using the default driver port.

Now create an SSH tunnel on the machine that needs to access the remote RethinkDB driver port:

```
ssh -L <local_port>:localhost:<driver_port> <ip_of_rethinkdb_machine>
```

Where,

- `local_port` is the port you are going to specify in the driver - It can be any available port on your machine.
- `driver_port` is the RethinkDB driver port (28015 by default).
- `ip_rethinkdb_machine` is the IP address of the machine that runs the RethinkDB server.

You can now connect to your RethinkDB instance by connecting to the host `localhost` and port `local_port`:

```
r.connect({host: 'localhost', port: <local_port>},
    function(error, connection) { ... })
```

Securing the intracluster port

To secure the intracluster port, you can use iptables to allow traffic only from the local network:

```
sudo iptables -A INPUT -i eth0 -p tcp --dport 29015 -j DROP
sudo iptables -I INPUT -i eth0 -s 192.168.0.0/24 -p tcp --dport 29015 -j ACCEPT
```

The intracluster port will be accessible from within the local network where you run RethinkDB nodes, but will not be accessible from the outside world.

RethinkDB can be easily deployed on Amazon Web Services. You can use a pre-built AMI (Amazon Machine Image), which takes only a few minutes to set up.

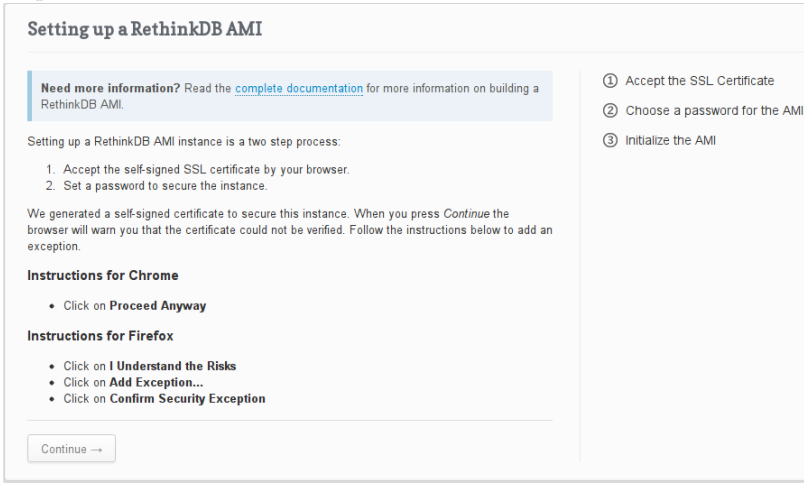
AWS quickstart

Launching an instance

The minimal recommended instance type is M1 Small, however T1 Micro works for simple tests. Follow these instructions to set up an AMI:

1. On the RethinkDB marketplace page, click the **Continue** button. Then select the **1-Click Launch** tab and click on the **Launch with 1-Click** button on the right. Finally, click on the link **Your Sotware** to access RethinkDB.

2. You should see a RethinkDB instance. When the instance is ready, click on the **Access Software** link on the right.
3. You should see a web page to set up the AMI. Click on the **Continue** button and follow the instructions.



Note: RethinkDB uses a self-signed certificate to encrypt your password. You'll have to accept the self-signed certificate in your browser to access the instance.

AMI configuration

The RethinkDB AMI is preconfigured with the following options:

- Ubuntu Server 12.04 LTS
- RethinkDB server
- Official RethinkDB client drivers for Python, JavaScript, and Ruby
- 5 GB of free EBS space for your data

Note: it is possible to attach more specialized EBS volumes and have RethinkDB store your data on them, but this option is not yet available out of the box. If you manually attach an EBS volume, you can SSH into the instance and edit the configuration file to point RethinkDB to the custom volume. See the cluster setup instructions for more details.

Instance administration

SSH access

To connect to your instance over SSH, log in as the user `ubuntu`. Use the private key you chose during the installation process and the public hostname of the

instance. For example:

```
ssh -i rethinkdb.prv -l ubuntu ec2-184-72-203-271.compute-1.amazonaws.com
```

RethinkDB command line administration

You can launch the administration tool from the command line after logging in over ssh:

```
rethinkdb admin --join localhost:29015
```

Security

The default security group opens 4 ports:

- Port 22 is for SSH. The server uses public key authentication.
- Port 80 is for HTTP. It is used during the setup process but otherwise redirects to HTTPS.
- Port 443 is for HTTPS. An Nginx server sits between RethinkDB and the world and provides basic HTTP authentication and secure HTTPS connections for the web UI.
- Port 28015 is for client driver access. The only form of authentication is a key that is sent in plain text over the network.

To secure your instance even further, we recommend that you perform the following steps:

- **Change the authentification key.** Open the RethinkDB command line and execute the command `set auth <your_key>`
- **Restrict access to port 28015** to allow only IP addresses or security groups that should have driver access.

Changing the web UI password

To change the password used to access the web UI, log in over SSH and run the following command:

```
htpasswd /etc/nginx/htpasswd rethinkdb
```

The `htpasswd` tool will prompt for your new password.

Changing the driver API key

To change the API key used by the server to authenticate the drivers, login over SSH and run `rethinkdb admin set auth`.

You can run the following commands to generate a good API key:

```
API_KEY=$(head /dev/urandom | md5sum | cut -f 1 -d ' ')
htpasswd /etc/nginx/htpasswd rethinkdb $API_KEY
echo $API_KEY
```

Cluster administration

To form a two-machine cluster, launch two RethinkDB instances on Amazon. Follow the steps below to ensure that AWS security groups are configured properly:

1. Open the **Security Groups** section of the administration console. If you launched your instance in the US East region, you can find the console here.
2. Select the security group that your instances belong to and open the **Inbound** tab in the bottom half of the page.
3. Note the id of the security group, it should start with `sg-`.
4. Create a new rule to allow instances to connect to one another:
    - Select **Custom TCP rule**.
    - Enter "29015" as the port range
    - As the **Source**, enter the id of the security group (see step 3)
    - Click on **Add Rule**, and **Apply rule changes**

After the rule has been applied, connect to one of the two instances over SSH and change the RethinkDB configuration file to join the two instances (see the cluster setup instructions).

Note: we will automate setup of RethinkDB clusters on AWS in the future.

In this tutorial we'll introduce using RethinkDB in Python by playing with a superhero dataset. We'll show how to insert and retrieve documents, query the database for specific data, and update documents with new information.

Prerequisites

Before following this tutorial you must have RethinkDB installed and running. You'll also need to install the RethinkDB Python library.

Connecting

Let's now confirm the setup is correct and we can connect to the RethinkDB instance. We will assume that RethinkDB is running on the same machine and on the default port (you can change the parameters passed to `connect`):

```
>>> import rethinkdb as r
>>> # connect and make the connection available to subsequent commands
>>> r.connect('localhost', 28015).repl()
>>> print r.db_list().run()

[u'test']
```

The result lists the default database available in RethinkDB.

Tips: The `r.connect(...).repl()` function is useful when using the RethinkDB Python driver from interactive shells making available the connection for all subsequent `run` calls.

Databases and tables

Accessing databases and tables

A single RethinkDB instance can host multiple database, but most of the time you'll work with a single database at a time. RethinkDB comes with a default database so you can quickly experiment with it:

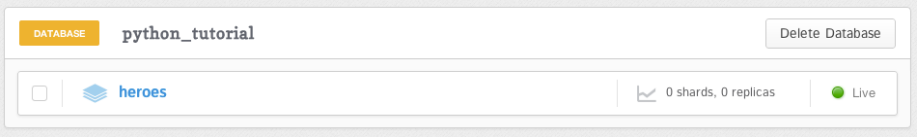
```
>>> test = r.db('test')
```

Creating a Database and a Table

For your application you'll probably want to use a new database and define new tables:

```
// creating a new database
>>> r.db_create('python_tutorial').run()

// creating a new table
>>> r.db('python_tutorial').table_create('heroes').run()
```

You can see the new database and table through the browser-based administrative UI: http://localhost:8080/#tables.



Because we will continue to use the `heroes` table, let's save it as a reference for the next operations:

```
>>> heroes = r.db('python_tutorial').table('heroes')
```

Inserting Documents

RethinkDB stores data in JSON, so passing `dicts` from Python requires no additional conversions:

```
>>> heroes.insert({
    "hero": "Wolverine",
    "name": "James 'Logan' Howlett",
   "magazine_titles": ["Amazing Spider-Man vs. Wolverine", "Avengers", "X-MEN Unlimited", "
    "appearances_count": 98
}).run()

{u'errors': 0,
 u'generated_keys': [u'c6677d9f-1740-4499-bf17-92f10cab30cf'],
 u'inserted': 1}
```

Tips: As you can notice in the result, RethinkDB generates a unique ID for the documents that do not provide one.

Inserting Multiple Documents

You can also insert multiple documents at a time by passing `insert` an array of dicts:

```
>>> heroes.insert([
    {
        "hero": "Magneto",
        "name": "Max Eisenhardt",
        "aka": ["Magnus", "Erik Lehnsherr", "Lehnsherr"],
     "magazine_titles": ["Alpha Flight", "Avengers", "Avengers West Coast"],
        "appearances_count": 42
    },
    {
```

```
        "hero": "Professor Xavier",
        "name": "Charles Francis Xavier",
     "magazine_titles": ["Alpha Flight", "Avengers", "Bishop", "Defenders"],
        "appearances_count": 72
    },
    {
        "hero": "Storm",
        "name": "Ororo Monroe",
     "magazine_titles": ["Amazing Spider-Man vs. Wolverine", "Excalibur", "Fantastic Four"
        "appearances_count": 72
    }

]).run()

{u'errors': 0,
 u'generated_keys': [u'd7d5e949-3f71-4e21-b5b7-42b6e7048ea3',
                     u'747c057e-8810-4479-a6b2-3c28d8057b48',
                     u'372fa6fe-17ec-494b-a926-0d99ba8ced43'],
 u'inserted': 3}
```

Retrieving all documents

Even if we only inserted 4 documents – you can double check that by running `heroes.count().run()` – let's take a quick look at them:

```
>>> heroes.run()

<rethinkdb.net.Cursor object at 0x15d4710>
```

Tips: If the table contains a large number of documents a query will not return all of them at once, which would saturate the network and/or require a lot of memory on the client. Instead the query will return the results in batches and fetch more data as needed.

Retrieving a single document

Let's now retrieve a document by its ID:

```
>>> heroes.get('d7d5e949-3f71-4e21-b5b7-42b6e7048ea3').run()

{u'aka': [u'Magnus', u'Erik Lehnsherr', u'Lehnsherr'],
 u'appearances_count': 42,
 u'hero': u'Magneto',
```

```
 u'name': u'Max Eisenhardt',
 u'id': u'e4bbd5e0-de9c-15ac-672c-d00b9f23a1f5',
 u'magazine_titles': [...]}
```

Querying

RethinkDB supports a wide range of filters, so let's try a couple of different ones. Firstly let's retrieve Professor Xavier by his character name:

```
>>> heroes.filter({'name': 'Charles Francis Xavier'}).run()

<rethinkdb.net.Cursor object at 0x15dc910>
```

Next thing we can do is to **order** the characters based on the number of magazines they've appeared in:

```
>>> heroes.order_by(r.desc('appearances_count')).pluck('hero', 'appearances_count').run()

<rethinkdb.net.Cursor object at 0x15dc450>
```

Tips: Ordering result ascending or descending can be done using `asc` and `desc` respectively. The server-side operation `pluck` allows fetching only the specified attributes of the result documents.

As you see only 1 of the characters has appeared in more than 90 magazines. This is something we could also verify with the query:

```
>>> heroes.filter(row['appearances_count'] >= 90).pluck('hero', 'name', 'appearances_count
```

For the last query example let's retrieve the characters that appeared in a specific magazine:

```
>>> heroes.filter(
      lambda row: row['magazine_titles'].filter(lambda mag: mag == 'Amazing Spider-Man vs. W
    ).pluck('hero').run()
```

Updating multiple documents

We'll finish this tutorial by appending a new magazine to each of the characters and also updating their number of appearances:

```
heroes.update({
    'appearances_count': row['appearances_count'] + 1,
   'magazine_titles': row['magazines'].append('The Fantastic RethinkDB')
})

{u'errors': 0, u'skipped': 0, u'updated': 4}
```

Tips: RethinkDB supports atomic updates at the document level. You can read more about about the RethinkDB atomicity model.

What's next

1. Learn more about RethinkDB queries
2. Learn how to scale an application running on RethinkDB
3. Learn how to experiment and tune queries using the Data Explorer

Before you start:

- Make sure you've installed RethinkDB — it should only take a minute!

Given the timeliness of the 2012 US Presidential Election and the inherent intricacies of the electoral process, we internally used an interesting dataset on poll results to test RethinkDB's query language support in the Javascript, Python, and Ruby client libraries.

We'll use this dataset to walk through RethinkDB's JavaScript query language using the Data Explorer.

If you want to do follow the tutorial with the node driver:

- If you have not done it yet, you may want to read the 10 minute guide.

Import the data

Download the datasets:

```
wget https://raw.github.com/rethinkdb/rethinkdb/next/demos/election/input_polls.json
wget https://raw.github.com/rethinkdb/rethinkdb/next/demos/election/county_stats.json
```

Import them in RethinkDB:

```
rethinkdb import -c localhost:28015 --table test.input_polls --pkey uuid -f input_polls.jso
rethinkdb import -c localhost:28015 --table test.county_stats --pkey uuid -f county_stats.j
```

What does this data set contain?

You have imported two tables:

- `input_polls` contains multiple poll results at state-level
- `county_stats` contains various population stats at county-level

You can take a look at the documents in each set with these queries:

```
r.table('input_polls').limit(1)
```

The result returned should be something similar to:


```
[{
    "uuid":"001b1830-b786-402e-a10b-1c3ea225971d",
    "id":"New Hampshire",
    "Pollster":"Marist Coll.-2",
    "Len":2,
    "GOP":45,
    "EV":4,
    "Dem":45,
    "Day":175.5,
    "Date":"Jun 25"
}]
```


For the table `county_stats`:

```
r.table('county_stats').limit(1)
```

You will get back something with this schema:


```
[{
    "uuid":"0052158f-6f15-4c27-851d-447b76c587ba",
    "state":"17",
    "ctyname":"Champaign County",
    "county":"019",
    "Stname":"Illinois",
    "SUMLEV":"050",
    "Rdeath2011":5.8652045006,
    "Rbirth2011":11.775067919000001,
    "RNETMIG2011":-4.406346528,
    "RNATURALINC2011":5.9098634181000005,
    "RINTERNATIONALMIG2011":3.8009700909,
    "RESIDUAL2011":12,
```

```
    "RESIDUAL2010":1,
    "REGION":2,
    "RDOMESTICMIG2011":-8.207316619,
    "POPESTIMATE2011":201685,
    "POPESTIMATE2010":201370,
    "NPOPCHG_2011":315,
    "NPOPCHG_2010":289,
    "NETMIG2011":-888,
    "NETMIG2010":-59,
    "NATURALINC2011":1191,
    "NATURALINC2010":347,
    "INTERNATIONALMIG2011":766,
    "INTERNATIONALMIG2010":207,
    "GQESTIMATESBASE2010":16129,
    "GQESTIMATES2011":16129,
    "GQESTIMATES2010":16129,
    "ESTIMATESBASE2010":201081,
    "Deaths2011":1182,
    "Deaths2010":270,
    "DOMESTICMIG2011":-1654,
    "DOMESTICMIG2010":-266,
    "DIVISION":3,
    "CENSUS2010POP":201081,
    "Births2011":2373,
    "Births2010":617
}]
```

Data cleanup: chaining, grouped-map-reduce, simple map

We'll first clean up the data in `input_polls`, as we want to calculate the average results of various polls at the state level. We'll also get rid of unnecessary/empty attributes. Finally we'll store the result in a new table:

First let's create a new table that will contain the clean data.

```
r.db("test").tableCreate("polls")
```

Then let's rework the data and save it in `polls`. We are going to group polls per state and compute the average score for each party.

```
r.table("polls").insert(
    r.table("input_polls").groupedMapReduce(
```

```
        // We group per `id`, `id` being a state name
        function(poll) {
            return poll("id")
        },
     // For each poll, we keep the results and add a field polls with the value 1
        function(poll) {
            return {
                Dem: poll("Dem"),
                GOP: poll("GOP"),
                polls: 1,
            }
        },
        // We reduce each group doing the sum for all fields
        function(left, right) {
            return {
                Dem: left("Dem").add(right("Dem")),
                GOP: left("GOP").add(right("GOP")),
                polls: left("polls").add(right("polls")),
            }
        }
    ).map( function(poll) {
        // We previously did the sum
        // Now we divide the fields `Dem` and `GOP` or each group
        // by the number of polls to get the average result
        return {
        Dem: poll("reduction")("Dem").div(poll("reduction")("polls")),
        GOP: poll("reduction")("GOP").div(poll("reduction")("polls")),
            polls: poll("reduction")("polls"),
            id: poll("group")
        }
    })
)
```

If everything went well, you should see that we inserted 51 documents (one per state plus one for Washington DC).

```
{
    "unchanged":0,
    "skipped":0,
    "replaced":0,
    "inserted":51,
    "errors":0,
    "deleted":0
}
```

If you take a look at the Arizona state

```
r.table('polls').get("Arizona")
```

You should get back this document:

```
{
    "Dem": 42.294117647058826,
    "GOP": 48.294117647058826,
    "polls": 17,
    "id": "Arizona"
}
```

Data analysis: projections, JOINs, orderby, group-map-reduce

Based on this data let's try to see if we can figure out **how many voters a party would need to turn to win the states.** For the sake of this post, we'll go with the Democrats.

Let's start with what estimates polls project at the county level by **JOIN**ing the `polls` and `county_stats` tables:

```
r.table('county_stats').eqJoin('Stname', r.table('polls')) // equi join of the two tables
    .zip() // flatten the results
   .pluck('Stname', 'state', 'county', 'ctyname', 'CENSUS2010POP', 'POPESTIMATE2011', 'Dem'
```

Building on this query, next we can find the counties where the Democrats are in minority:

```
r.table('county_stats').eqJoin('Stname', r.table('polls'))
    .zip()
   .pluck('Stname', 'state', 'county', 'ctyname', 'CENSUS2010POP', 'POPESTIMATE2011', 'Dem'
    .filter(function(doc) { return doc('Dem').lt(doc('GOP')) })
```

Or even better where Democrats are within 15% of the Republicans:

```
r.table('county_stats').eqJoin('Stname', r.table('polls'))
    .zip()
   .pluck('Stname', 'state', 'county', 'ctyname', 'CENSUS2010POP', 'POPESTIMATE2011', 'Dem'
   .filter(function(doc) { return doc('Dem').lt(doc('GOP')).and(doc('GOP').sub(doc('Dem'))
```

The last step in answering the initial question of how many voters should the Democrats win to turn the results is just a `groupedMapReduce` away:

```
r.table('county_stats').eqJoin('Stname', r.table('polls'))
    .zip()
   .pluck('Stname', 'state', 'county', 'ctyname', 'CENSUS2010POP', 'POPESTIMATE2011', 'Dem'
   .filter(function(doc) { return doc('Dem').lt(doc('GOP')).and(doc('GOP').sub(doc('Dem'))
    .groupedMapReduce(
        function(doc){ return doc('Stname') },
     function(doc){ return doc('POPESTIMATE2011').mul(doc("GOP").sub(doc("Dem")).div(100)
        function(acc, val) {return acc.add(val)})
    .orderBy('reduction')
```

And the outcome of our quick presidential election data analysis that addresses the question **how many voters the Democrat party would need to turn to win the states** (this assumes 100% turnout of the entire population of a state):

group	reduction
South Dakota	145038.43200000(
North Carolina	153366.368823529
Montana	157493.62000000(
North Dakota	164143.68
Nebraska	497513.07
West Virginia	544240.10666666(
Missouri	558640.414117647
South Carolina	701884.5
Mississippi	774413.12
Arizona	777900.6
Tennessee	992519.71500000(
Indiana	1042707.5200000(
Louisiana	1158958.4533333:
Georgia	2028476.7333333:
Texas	6589834.7899999:

If you followed along, the queries above should have given you a taste of ReQL: **chaining**, **projections**, **order by**, **JOINs**, **grouped-map-reduce**. Of course

this tutorial isn't statistically significant. If you interested in statistically significant results, checkout the election statistics superhero Nate Silver.

Want to learn more about RethinkDB?

- Read the ten-minute guide to get started with RethinkDB.
- Browse the architecture overview for programmers familiar with distributed systems.
- Jump into the cookbook and see dozens of examples of common RethinkDB queries.

RethinkDB overview

What is RethinkDB?

RethinkDB is an open-source, distributed database built to store JSON documents and scale to multiple machines with very little effort. It has a pleasant query language that supports really useful queries like table joins and group by, and is easy to setup and learn.

RethinkDB in under two minutes: see the highlights video.

What are the main differences from other NoSQL databases?

Find out how RethinkDB compares to other NoSQL databases:

- RethinkDB compared to MongoDB — an unbiased technical comparison between RethinkDB and MongoDB.
- RethinkDB vs today's NoSQL — our biased, but more personal take on what makes RethinkDB different.

When is RethinkDB a good choice?

- RethinkDB is a great choice if you need flexible schemas, value ease of use, and are planning to run anywhere from a single node to a sixteen-node cluster.
- If you periodically copy your data into a separate system to do analytics (such as Hadoop), but your analytics are not incredibly computationally intensive, you can significantly simplify things by running your analytical queries in RethinkDB directly. RethinkDB will *not* lock your database.

- Finally, if you are already running a database cluster and are overwhelmed by cluster administration and the complexities of sharding, replication, and failover, you will love RethinkDB. Sharding and replication can be done in a few clicks in the Web UI or on the command line.

When is RethinkDB not a good choice?

- RethinkDB is not a good choice if you need full ACID support or strong schema enforcement — in this case you are better off using a relational database such as MySQL or PostgreSQL.
- If you are doing deep, computationally-intensive analytics you are better off using a system like Hadoop or a column-oriented store like Vertica.
- In some cases RethinkDB trades off write availability in favor of data consistency, so if you absolutely need high write availability and do not mind dealing with conflicts, you may be better off with a Dynamo-style system like Riak.

Practical considerations

What languages can I use to work with RethinkDB?

You can use Ruby, Python, and Javascript/Node.js to write RethinkDB queries. In addition, there are community supported client drivers for more than half a dozen other languages.

If you already know Javascript, all RethinkDB queries can be freely intermixed with Javascript code because the server supports native Javascript execution using the V8 engine.

What are the system requirements?

RethinkDB server is written in C++ and currently runs on 32-bit and 64-bit Linux systems, as well as OS X 10.7 and above. Ruby, Python, Javascript, as well as community supported client drivers can run on any platform where these languages are supported.

RethinkDB doesn't have other strict requirements. It has a custom caching engine and can run on low-memory nodes with large amounts of on-disk data, Amazon EC2 instances, etc. It also has specialized support for high-end hardware and does a great job on high-memory nodes with many cores, solid-state storage, and high-throughput network hardware.

Note: With the default settings (cache of 1GB per table), we recommend using servers with at least 2GB of RAM. This requirement will go away as soon as the new cache memory manager will be available.

Does RethinkDB support SQL?

No, but RethinkDB supports a very powerful, expressive, and easy to learn query language that can do almost anything SQL can do (and many things SQL can't do, such as mixing queries with Javascript expressions and Hadoop-style map/reduce).

How do queries get routed in a RethinkDB cluster?

You can connect your clients to any node in the cluster, and all the queries will automatically be routed to their destination. Advanced queries (such as joins, filters, etc.) will be broken up and routed to the appropriate machines, executed in parallel, the resultset will be recombined, and streamed back to the client. The user never has to worry about sending queries to specific nodes—everything happens automatically behind the scenes.

How does RethinkDB handle write durability?

RethinkDB comes with strict write durability out of the box and is identical to traditional database systems in this respect. By default, no write is ever acknowledged until it's safely committed to disk.

Want to speed up your write queries? Learn how to configure durability options.

How is RethinkDB licensed?

RethinkDB server is licensed under GNU AGPL v3.0. The client drivers are licensed under Apache License v2.0.

We wanted to pick a license that balances the interests of three parties — our end users, our company, and the software development community at large. When picking a license, we decided that these interests can be expressed via three simple goals:

- Allow anyone to download RethinkDB, examine the source code, and use it for free (as in speech and beer) for any purpose.
- Require users that choose to modify RethinkDB to fit their needs to release the patches to the software development community.
- Require users that are unwilling to release the patches to the software development community to purchase a commercial license.
- Given that an enormous amount of software is offered as a service via the network and isn't actually distributed in binary form, the most effective license to fulfill all three goals is GNU AGPL.

We chose to release the client drivers under Apache License v2.0 to remove any ambiguity as to the extent of the server license — you do not have to license any software that uses RethinkDB under AGPL and are completely free to use any licensing mechanism of your choice.

Command syntax

time.year() → number

Description

Return the year of a time object.

Example: Retrieve all the users born in 1986.

```
r.table("users").filter(lambda user:
    user["birthdate"].year() == 1986
).run(conn)
```

Command syntax

sequence[attr] → sequence

singleSelection[attr] → value

object[attr] → value

Description

Get a single field from an object. If called on a sequence, gets that field from every object in the sequence, skipping objects that lack it.

Example: What was Iron Man's first appearance in a comic?

```
r.table('marvel').get('IronMan')['firstAppearance'].run(conn)
```

Command syntax

sequence.union(sequence) → array

Description

Concatenate two sequences.

Example: Construct a stream of all heroes.

```
r.table('marvel').union(r.table('dc')).run(conn)
```

Command syntax

time.day() → number

Description

Return the day of a time object as a number between 1 and 31.

Example: Return the users born on the 24th of any month.

```
r.table("users").filter(
    r.row["birthdate"].day() == 24
)
```

Command syntax

r.db(db_name) → db

Description

Reference a database.

Example: Before we can query a table we have to select the correct database.

```
r.db('heroes').table('marvel').run(conn)
```

Command syntax

array.delete_at(index [,endIndex]) → array

Description

Remove an element from an array at a given index. Returns the modified array.

Example: Hulk decides to leave the avengers.

```
r.expr(["Iron Man", "Hulk", "Spider-Man"]).delete_at(1).run(conn)
```

Example: Hulk and Thor decide to leave the avengers.

```
r.expr(["Iron Man", "Hulk", "Thor", "Spider-Man"]).delete_at(1,3).run(conn)
```

Command syntax

sequence.distinct() → array

Description

Remove duplicate elements from the sequence.

Example: Which unique villains have been vanquished by marvel heroes?

```
r.table('marvel').concat_map(lambda hero: hero['villainList']).distinct().run(conn)
```

Command syntax

array.difference(array) → array

Description

Remove the elements of one array from another array.

Example: Retrieve Iron Man's equipment list without boots.

```
r.table('marvel').get('IronMan')['equipment'].difference(['Boots']).run(conn)
```

Command syntax

r.db_drop(db_name) → object

Description

Drop a database. The database, all its tables, and corresponding data will be deleted.

If successful, the operation returns the object `{"dropped": 1}`. If the specified database doesn't exist a `RqlRuntimeError` is thrown.

Example: Drop a database named 'superheroes'.

```
r.db_drop('superheroes').run(conn)
```

Command syntax

db.table(name[, use_outdated=False]) → table

Description

Select all documents in a table. This command can be chained with other commands to do further processing on the data.

Example: Return all documents in the table 'marvel' of the default database.

```
r.table('marvel').run(conn)
```

Example: Return all documents in the table 'marvel' of the database 'heroes'.

```
r.db('heroes').table('marvel').run(conn)
```

Example: If you are OK with potentially out of date data from this table and want potentially faster reads, pass a flag allowing out of date data.

```
r.db('heroes').table('marvel', True).run(conn)
```

Command syntax

time.minutes() → number

Description

Return the minute in a time object as a number between 0 and 59.

Example: Return all the posts submitted during the first 10 minutes of every hour.

```
r.table("posts").filter(lambda post:
    post["date"].minutes() < 10
).run(conn)
```

Command syntax

sequence.map(mapping_function) → stream

array.map(mapping_function) → array

Description

Transform each element of the sequence by applying the given mapping function.

Example: Construct a sequence of hero power ratings.

```
r.table('marvel').map(lambda hero:
    hero['combatPower'] + hero['compassionPower'] * 2
).run(conn)
```

Command syntax

db.table_create(table_name[, options]) → object

Description

Create a table. A RethinkDB table is a collection of JSON documents.

If successful, the operation returns an object: `{created: 1}`. If a table with the same name already exists, the operation throws `RqlRuntimeError`.

Note: that you can only use alphanumeric characters and underscores for the table name.

When creating a table you can specify the following options:

- `primary_key`: the name of the primary key. The default primary key is id;
- `durability`: if set to `soft`, this enables *soft durability* on this table: writes will be acknowledged by the server immediately and flushed to disk in the background. Default is `hard` (acknowledgement of writes happens after data has been written to disk);
- `cache_size`: set the cache size (in bytes) to be used by the table. The default is 1073741824 (1024MB);
- `datacenter`: the name of the datacenter this table should be assigned to.

Example: Create a table named 'dc_universe' with the default settings.

```
r.db('test').table_create('dc_universe').run(conn)
```

Example: Create a table named 'dc_universe' using the field 'name' as primary key.

```
r.db('test').table_create('dc_universe', primary_key='name').run(conn)
```

Example: Create a table to log the very fast actions of the heroes.

```
r.db('test').table_create('hero_actions', durability='soft').run(conn)
```

Command syntax

sequence.for_each(write_query) → object

Description

Loop over a sequence, evaluating the given write query for each element.

Example: Now that our heroes have defeated their villains, we can safely remove them from the villain table.

```
r.table('marvel').for_each(
   lambda hero: r.table('villains').get(hero['villainDefeated']).delete()
).run(conn)
```

Command syntax

table.delete([durability="hard", return_vals=False]) → object

selection.delete([durability="hard", return_vals=False]) → object

singleSelection.delete([durability="hard", return_vals=False]) → object

Description

Delete one or more documents from a table.

The optional arguments are:

- `durability`: possible values are `hard` and `soft`. This option will override the table or query's durability setting (set in run).
  In soft durability mode RethinkDB will acknowledge the write immediately after receiving it, but before the write has been committed to disk.
- `return_vals`: if set to `True` and in case of a single document deletion, the deleted document will be returned.

Delete returns an object that contains the following attributes:

- `deleted`: the number of documents that were deleted.
- `skipped`: the number of documents that were skipped.
  For example, if you attempt to delete a batch of documents, and another concurrent query deletes some of those documents first, they will be counted as skipped.
- `errors`: the number of errors encountered while performing the delete.
- `first_error`: If errors were encountered, contains the text of the first error.
- `inserted`, `replaced`, and `unchanged`: all 0 for a delete operation..
- `old_val`: if `return_vals` is set to `True`, contains the deleted document.
- `new_val`: if `return_vals` is set to `True`, contains `None`.

Example: Delete a single document from the table `comments`.

```
r.table("comments").get("7eab9e63-73f1-4f33-8ce4-95cbea626f59").delete().run(conn)
```

Example: Delete all documents from the table `comments`.

```
r.table("comments").delete().run(conn)
```

Example: Delete all comments where the field `id_post` is 3.

```
r.table("comments").filter({"id_post": 3}).delete().run(conn)
```

Example: Delete a single document from the table `comments` and return its value.

```
r.table("comments").get("7eab9e63-73f1-4f33-8ce4-95cbea626f59").delete(return_vals=True)
```

The result look like:

```
{
    "deleted": 1,
    "errors": 0,
    "inserted": 0,
    "new_val": None,
    "old_val": {
        "id": "7eab9e63-73f1-4f33-8ce4-95cbea626f59",
        "author": "William",
        "comment": "Great post",
        "id_post": 3
    },
    "replaced": 0,
    "skipped": 0,
    "unchanged": 0
}
```

Example: Delete all documents from the table `comments` without waiting for the operation to be flushed to disk.

```
r.table("comments").delete(durability="soft"}).run(conn)
```

Command syntax

table.get(key) → singleRowSelection

Description

Get a document by primary key.

Example: Find a document with the primary key 'superman'.

```
r.table('marvel').get('superman').run(conn)
```

Command syntax

time.time_of_day() → number

Description

Return the number of seconds elapsed since the beginning of the day stored in the time object.

Example: Retrieve posts that were submitted before noon.

```
r.table("posts").filter(
    r.row["date"].time_of_day() <= 12*60*60
).run(conn)
```

Command syntax

sequence.indexes_of(datum | predicate) → array

Description

Get the indexes of an element in a sequence. If the argument is a predicate, get the indexes of all elements matching it.

Example: Find the position of the letter 'c'.

```
r.expr(['a','b','c']).indexes_of('c').run(conn)
```

Example: Find the popularity ranking of invisible heroes.

```
r.table('marvel').union(r.table('dc')).order_by('popularity').indexes_of(
    r.row['superpowers'].contains('invisibility')
).run(conn)
```

Command syntax

string.match(regexp) → array

Description

Match against a regular expression. Returns a match object containing the matched string, that string's start/end position, and the capture groups. Accepts RE2 syntax (https://code.google.com/p/re2/wiki/Syntax). You can enable case-insensitive matching by prefixing the regular expression with `(?i)`. (See linked RE2 documentation for more flags.)

Example: Get all users whose name starts with A.

```
r.table('users').filter(lambda row:row['name'].match("^A")).run(conn)
```

Example: Parse out a name (returns "mlucy").

```
r.expr('id:0,name:mlucy,foo:bar').match('name:(\w+)')['groups'][0]['str'].run(conn)
```

Example: Fail to parse out a name (returns null).

```
r.expr('id:0,foo:bar').match('name:(\w+)')['groups'][0]['str'].run(conn)
```

Command syntax

sequence.count([filter]) → number

Description

Count the number of elements in the sequence. With a single argument, count the number of elements equal to it. If the argument is a function, it is equivalent to calling filter before count.

Example: Just how many super heroes are there?

```
(r.table('marvel').count() + r.table('dc').count()).run(conn)
```

Example: Just how many super heroes have defeated the Sphinx?

```
r.table('marvel').count(r.row['monstersKilled'].contains('Sphinx')).run(conn)
```

Command syntax

array.set_intersection(array) → array

Description

Intersect two arrays returning values that occur in both of them as a set (an array with distinct values).

Example: Check which pieces of equipment Iron Man has from a fixed list.

```
r.table('marvel').get('IronMan')['equipment'].set_intersection(['newBoots', 'arc_reactor'
```

Command syntax

sequence.eq_join(left_attr, other_table[, index='id']) → stream

array.eq_join(left_attr, other_table[, index='id']) → array

Description

An efficient join that looks up elements in the right table by primary key.

Example: Let our heroes join forces to battle evil!

```
r.table('marvel').eq_join('main_dc_collaborator', r.table('dc')).run(conn)
```

Example: The above query is equivalent to this inner join but runs in O(n log(m)) time rather than the O(n * m) time the inner join takes.

```
r.table('marvel').inner_join(r.table('dc'),
lambda left, right: left['main_dc_collaborator'] == right['hero_name']).run(conn)
```

Example: You can take advantage of a secondary index on the second table by giving an optional index parameter.

```
r.table('marvel').eq_join('main_weapon_origin',
r.table('mythical_weapons'), index='origin').run(conn)
```

Example: You can pass a function instead of an attribute to join on more complicated expressions. Here we join to the DC universe collaborator with whom the hero has the most appearances.

```
r.table('marvel').eq_join(lambda doc:
    doc['dc_collaborators'].order_by('appearances')[0]['name'],
    r.table('dc')).run(conn)
```

Command syntax

sequence.sample(number) → selection

stream.sample(number) → array

array.sample(number) → array

Description

Select a given number of elements from a sequence with uniform random distribution. Selection is done without replacement.

Example: Select 3 random heroes.

```
r.table('marvel').sample(3).run(conn)
```

Command syntax

r → r

Description

The top-level ReQL namespace.

Example: Setup your top-level namespace.

```
import rethinkdb as r
```

Command syntax

singleSelection.merge(object) → object

object.merge(object) → object

sequence.merge(object) → stream

array.merge(object) → array

Description

Merge two objects together to construct a new object with properties from both. Gives preference to attributes from other when there is a conflict.

Example: Equip IronMan for battle.

```
r.table('marvel').get('IronMan').merge(
    r.table('loadouts').get('alienInvasionKit')
).run(conn)
```

Example: Merge can be used recursively to modify object within objects.

```
r.expr({'weapons' : {'spectacular graviton beam' : {'dmg' : 10, 'cooldown' : 20}}}).merge(
    {'weapons' : {'spectacular graviton beam' : {'dmg' : 10}}}
).run(conn)
```

Example: To replace a nested object with another object you can use the literal keyword.

```
r.expr({'weapons' : {'spectacular graviton beam' : {'dmg' : 10, 'cooldown' : 20}}}).merge(
   {'weapons' : r.literal({'repulsor rays' : {'dmg' : 3, 'cooldown' : 0}})}
).run(conn)
```

Example: Literal can be used to remove keys from an object as well.

```
r.expr({'weapons' : {'spectacular graviton beam' : {'dmg' : 10, 'cooldown' : 20}}}).merge(
    {'weapons' : {'spectacular graviton beam' : r.literal()}}
).run(conn)
```

Command syntax

r.iso8601(iso8601Date[, default_timezone=”]) → time

Description

Create a time object based on an iso8601 date-time string (e.g. ‘2013-01-01T01:01:01+00:00’). We support all valid ISO 8601 formats except for week dates. If you pass an ISO 8601 date-time without a time zone, you must specify the time zone with the optarg `default_timezone`. Read more about the ISO 8601 format on the Wikipedia page.

Example: Update the time of John’s birth.

```
r.table("user").get("John").update({"birth": r.iso8601('1986-11-03T08:30:00-07:00')}).run
```

Command syntax

sequence.group_by(selector1[, selector2...], reduction_object) → array

Description

Groups elements by the values of the given attributes and then applies the given reduction. Though similar to `groupedMapReduce`, `groupBy` takes a standardized object for specifying the reduction. Can be used with a number of predefined common reductions.

Example: Using a predefined reduction we can easily find the average strength of members of each weight class.

```
r.table('marvel').group_by('weightClass', r.avg('strength')).run(conn)
```

Example: Groupings can also be specified on nested attributes.

```
r.table('marvel').group_by({'abilities' : {'primary' : True}}, r.avg('strength')).run(conn
```

Example: The nested syntax can quickly become verbose so there's a shortcut.

```
r.table('marvel').group_by({'abilities' : 'primary'}, r.avg('strength')).run(conn)
```

Command syntax

conn.close(noreply_wait=True)

Description

Close an open connection. Closing a connection waits until all outstanding requests have finished and then frees any open resources associated with the connection. If `noreply_wait` is set to `false`, all outstanding requests are canceled immediately.

Closing a connection cancels all outstanding requests and frees the memory associated with any open cursors.

Example: Close an open connection, waiting for noreply writes to finish.

```
conn.close()
```

Example: Close an open connection immediately.

```
conn.close(noreply_wait=False)
```

Command syntax

r.json(json_string) → value

Description

Parse a JSON string on the server.

Example: Send an array to the server'

```
r.json("[1,2,3]").run(conn)
```

Command syntax

value < value → bool

value.lt(value) → bool

Description

Test if the first value is less than other.

Example: Is 2 less than 2?

```
(r.expr(2) < 2).run(conn)
r.expr(2).lt(2).run(conn)
```

Command syntax

r.count

Description

Count the total size of the group.

Example: Just how many heroes do we have at each strength level?

```
r.table('marvel').group_by('strength', r.count).run(conn)
```

Command syntax

time.date() → time

Description

Return a new time object only based on the day, month and year (ie. the same day at 00:00).

Example: Retrieve all the users whose birthday is today

```
r.table("users").filter(lambda user:
    user["birthdate"].date() == r.now().date()
).run(conn)
```

Command syntax

bool & bool → bool

Description

Compute the logical and of two values.

Example: True and false anded is false?

```
(r.expr(True) & False).run(conn)
```

Command syntax

r.now() → time

Description

Return a time object representing the current time in UTC. The command now() is computed once when the server receives the query, so multiple instances of r.now() will always return the same time inside a query.

Example: Add a new user with the time at which he subscribed.

```
r.table("users").insert({
    "name": "John",
    "subscription_date": r.now()
}).run(conn)
```

Command syntax

array.splice_at(index, array) → array

Description

Insert several values in to an array at a given index. Returns the modified array.

Example: Hulk and Thor decide to join the avengers.

```
r.expr(["Iron Man", "Spider-Man"]).splice_at(1, ["Hulk", "Thor"]).run(conn)
```

Command syntax

time.hours() → number

Description

Return the hour in a time object as a number between 0 and 23.

Example: Return all the posts submitted after midnight and before 4am.

```
r.table("posts").filter(lambda post:
    post["date"].hours() < 4
).run(conn)
```

Command syntax

array.change_at(index, value) → array

Description

Change a value in an array at a given index. Returns the modified array.

Example: Bruce Banner hulks out.

```
r.expr(["Iron Man", "Bruce", "Spider-Man"]).change_at(1, "Hulk").run(conn)
```

Command syntax

value >= value → bool

value.ge(value) → bool

Description

Test if the first value is greater than or equal to other.

Example: Is 2 greater than or equal to 2?

```
(r.expr(2) >= 2).run(conn)
r.expr(2).ge(2).run(conn)
```

Command syntax

array.append(value) → array

Description

Append a value to an array.

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')['equipment'].append('newBoots').run(conn)
```

Command syntax

conn.reconnect(noreply_wait=True)

Description

Close and reopen a connection. Closing a connection waits until all outstanding requests have finished. If `noreply_wait` is set to `false`, all outstanding requests are canceled immediately.

Example: Cancel outstanding requests/queries that are no longer needed.

```
conn.reconnect(noreply_wait=False)
```

Command syntax

array.set_difference(array) → array

Description

Remove the elements of one array from another and return them as a set (an array with distinct values).

Example: Check which pieces of equipment Iron Man has, excluding a fixed list.

```
r.table('marvel').get('IronMan')['equipment'].set_difference(['newBoots', 'arc_reactor'])
```

Command syntax

any.info() → object

Description

Get information about a ReQL value.

Example: Get information about a table such as primary key, or cache size.

```
r.table('marvel').info().run(conn)
```

Command syntax

array.set_union(array) → array

Description

Add a several values to an array and return it as a set (an array with distinct values).

Example: Retrieve Iron Man's equipment list with the addition of some new boots and an arc reactor.

```
r.table('marvel').get('IronMan')['equipment'].set_union(['newBoots', 'arc_reactor']).run(
```

Command syntax

r.connect(host="localhost", port=28015, db="test", auth_key="", timeout=20) → connection

r.connect(host) → connection

Description

Create a new connection to the database server. The keyword arguments are:

- `host`: host of the RethinkDB instance. The default value is `localhost`.
- `port`: the driver port, by default `28015`.
- `db`: the database used if not explicitly specified in a query, by default `test`.
- `auth_key`: the authentification key, by default the empty string.
- `timeout`: timeout period for the connection to be opened, by default `20` (seconds).

Create a new connection to the database server.

If the connection cannot be established, a `RqlDriverError` exception will be thrown.

Example: Opens a connection using the default host and port but specifying the default database.

```
conn = r.connect(db='marvel')
```

Command syntax

r.error(message) → error

Description

Throw a runtime error. If called with no arguments inside the second argument to `default`, re-throw the current error.

Example: Iron Man can't possibly have lost a battle:

```
r.table('marvel').get('IronMan').do(
   lambda ironman: r.branch(ironman['victories'] < ironman['battles'],
                            r.error('impossible code path'),
                            ironman)
).run(conn)
```

Command syntax

sequence.grouped_map_reduce(grouping, mapping, reduction, base) → value

Description

Partition the sequence into groups based on the `grouping` function. The elements of each group are then mapped using the `mapping` function and reduced using the `reduction` function.

`grouped_map_reduce` is a generalized form of group by.

Example: It's only fair that heroes be compared against their weight class.

```
r.table('marvel').grouped_map_reduce(
    lambda hero: hero['weightClass'],  # grouping
    lambda hero: hero.pluck('name', 'strength'),  # mapping
   lambda acc, hero: r.branch(acc['strength'] < hero['strength'], hero, acc),
    {'name':'none', 'strength':0}  # base
).run(conn)
```

Command syntax

sequence.is_empty() → bool

Description

Test if a sequence is empty.

Example: Are there any documents in the marvel table?

```
r.table('marvel').is_empty().run(conn)
```

Command syntax

sequence.limit(n) → stream

array.limit(n) → array

Description

End the sequence after the given number of elements.

Example: Only so many can fit in our Pantheon of heroes.

```
r.table('marvel').order_by('belovedness').limit(10).run(conn)
```

Command syntax

db.table_drop(table_name) → object

Description

Drop a table. The table and all its data will be deleted.

If succesful, the operation returns an object: {“dropped”: 1}. If the specified table doesn’t exist a `RqlRuntimeError` is thrown.

Example: Drop a table named ‘dc_universe’.

```
r.db('test').table_drop('dc_universe').run(conn)
```

Command syntax

value != value → bool

value.ne(value) → bool

Description

Test if two values are not equal.

Example: Does 2 not equal 2?

```
(r.expr(2) != 2).run(conn)
r.expr(2).ne(2).run(conn)
```

Command syntax

sequence[index] → object

sequence.nth(index) → object

Description

Get the nth element of a sequence.

Example: Select the second element in the array.

```
r.expr([1,2,3])[1].run(conn)
r.expr([1,2,3]).nth(1).run(conn)
```

Command syntax

r.js(jsString) → value

Description

Create a javascript expression.

Example: Concatenate two strings using Javascript'

```
r.js("'str1' + 'str2'").run(conn)
```

Example: Select all documents where the 'magazines' field is greater than 5 by running Javascript on the server.

```
r.table('marvel').filter(
    r.js('(function (row) { return row.magazines > 5; })')
).run(conn)
```

Example: You may also specify a timeout in seconds (defaults to 5).

```
r.js('while(true) {}', timeout=1.3).run(conn)
```

Command syntax

table.update(json | expr[, durability="hard", return_vals=False, non_atomic=False]) → object

selection.update(json | expr[, durability="hard", return_vals=False, non_atomic=False]) → object

singleSelection.update(json | expr[, durability="hard", return_vals=False, non_atomic=False]) → object

Description

Update JSON documents in a table. Accepts a JSON document, a ReQL expression, or a combination of the two.

The optional arguments are:

- `durability`: possible values are `hard` and `soft`. This option will override the table or query's durability setting (set in run).
  In soft durability mode RethinkDB will acknowledge the write immediately after receiving it, but before the write has been committed to disk.
- `return_vals`: if set to `True` and in case of a single update, the updated document will be returned.
- `non_atomic`: set to `True` if you want to perform non-atomic updates (updates that require fetching data from another document).

Update returns an object that contains the following attributes:

- `replaced`: the number of documents that were updated.
- `unchanged`: the number of documents that would have been modified except the new value was the same as the old value.
- `skipped`: the number of documents that were skipped because the document didn't exist.

- `errors`: the number of errors encountered while performing the update.
- `first_error`: If errors were encountered, contains the text of the first error.
- `deleted` and `inserted`: 0 for an update operation.
- `old_val`: if `return_vals` is set to `True`, contains the old document.
- `new_val`: if `return_vals` is set to `True`, contains the new document.

Example: Update the status of the post with `id` of `1` to `published`.

```
r.table("posts").get(1).update({"status": "published"}).run(conn)
```

Example: Update the status of all posts to `published`.

```
r.table("posts").update({"status": "published"}).run(conn)
```

Example: Update the status of all the post written by William.

```
r.table("posts").filter({"author": "William"}).update({"status": "published"}).run(conn)
```

Example: Increment the field `view` with `id` of `1`. This query will throw an error if the field `views` doesn't exist.

```
r.table("posts").get(1).update({
    "views": r.row["views"]+1
}).run(conn)
```

Example: Increment the field `view` of the post with `id` of `1`. If the field `views` does not exist, it will be set to `0`.

```
r.table("posts").update({
    "views": (r.row["views"]+1).default(0)
}).run(conn)
```

Example: Perform a conditional update.
If the post has more than 100 views, set the `type` of a post to `hot`, else set it to `normal`.

```
r.table("posts").get(1).update(lambda post:
    r.branch(
        post["views"] > 100,
        {"type": "hot"},
        {"type": "normal"}
    )
).run(conn)
```

Example: Update the field `num_comments` with the result of a sub-query. Because this update is not atomic, you must pass the `non_atomic` flag.

```
r.table("posts").get(1).update({
   "num_comments": r.table("comments").filter({"id_post": 1}).count()
}, non_atomic=True ).run(conn)
```

If you forget to specify the `non_atomic` flag, you will get a `RqlRuntimeError`.

```
RqlRuntimeError: Could not prove function deterministic.  Maybe you want to use the non_atomi
```

Example: Update the field `num_comments` with a random value between 0 and 100.
This update cannot be proven deterministic because of `r.js` (and in fact is not), so you must pass the `non_atomic` flag.

```
r.table("posts").get(1).update({
    "num_comments": r.js("Math.floor(Math.random()*100)")
}, non_atomic=True ).run(conn)
```

Example: Update the status of the post with `id` of `1` using soft durability.

```
r.table("posts").get(1).update({status: "published"}, durability="soft").run(conn)
```

Example: Increment the field `views` and return the values of the document before and after the update operation.

```
r.table("posts").get(1).update({
    "views": r.row["views"]+1
}, return_vals=True).run(conn)
```

The result will have two fields `old_val` and `new_val`.

```
{
    "deleted": 1,
    "errors": 0,
    "inserted": 0,
    "new_val": {
        "id": 1,
        "author": "Julius_Caesar",
        "title": "Commentarii de Bello Gallico",
        "content": "Aleas jacta est",
        "views": 207
```

```
    },
    "old_val": {
        "id": 1,
        "author": "Julius_Caesar",
        "title": "Commentarii de Bello Gallico",
        "content": "Aleas jacta est",
        "views": 206
    },
    "replaced": 0,
    "skipped": 0,
    "unchanged": 0
}
```

Accessing ReQL

All ReQL queries begin from the top-level module.

r

r → r

The top-level ReQL namespace.

Example: Set up your top-level namespace.

```
import rethinkdb as r
```

connect

r.connect(host="localhost", port=28015, db="test", auth_key="", timeout=20) → connection

r.connect(host) → connection

Create a new connection to the database server. The keyword arguments are:

- `host`: host of the RethinkDB instance. The default value is `localhost`.
- `port`: the driver port, by default `28015`.
- `db`: the database used if not explicitly specified in a query, by default `test`.
- `auth_key`: the authentification key, by default the empty string.
- `timeout`: timeout period for the connection to be opened, by default `20` (seconds).

If the connection cannot be established, a `RqlDriverError` exception will be thrown.

Example: Opens a connection using the default host and port but specifying the default database.

```
conn = r.connect(db='marvel')
```

Read more about this command →

repl

conn.repl()

Set the default connection to make REPL use easier. Allows calling `.run()` on queries without specifying a connection.

Connection objects are not thread-safe and REPL connections should not be used in multi-threaded environments.

Example: Set the default connection for the REPL, then call `run()` without specifying the connection.

```
r.connect(db='marvel').repl()
r.table('heroes').run()
```

close

conn.close(noreply_wait=True)

Close an open connection. Closing a connection waits until all outstanding requests have finished and then frees any open resources associated with the connection. If `noreply_wait` is set to `false`, all outstanding requests are canceled immediately.

Closing a connection cancels all outstanding requests and frees the memory associated with any open cursors.

Example: Close an open connection, waiting for noreply writes to finish.

```
conn.close()
```

Example: Close an open connection immediately.

```
conn.close(noreply_wait=False)
```

reconnect

conn.reconnect(noreply_wait=True)

Close and reopen a connection. Closing a connection waits until all outstanding requests have finished. If `noreply_wait` is set to `false`, all outstanding requests are canceled immediately.

Example: Cancel outstanding requests/queries that are no longer needed.

```
conn.reconnect(noreply_wait=False)
```

use

conn.use(db_name)

Change the default database on this connection.

Example: Change the default database so that we don't need to specify the database when referencing a table.

```
conn.use('marvel')
r.table('heroes').run(conn) # refers to r.db('marvel').table('heroes')
```

run

query.run(conn, use_outdated=False, time_format='native', profile=False) → cursor

query.run(conn, use_outdated=False, time_format='native', profile=False) → object

Run a query on a connection, returning either a single JSON result or a cursor, depending on the query.

Example: Run a query on the connection `conn` and print out every row in the result.

```
for doc in r.table('marvel').run(conn):
    print doc
```

Read more about this command →

noreply_wait

conn.noreply_wait()

`noreply_wait` ensures that previous queries with the `noreply` flag have been processed by the server. Note that this guarantee only applies to queries run on the given connection.

Example: We have previously run queries with the `noreply` argument set to `True`. Now wait until the server has processed them.

```
conn.noreply_wait()
```

close (cursor)

cursor.close()

Close a cursor. Closing a cursor cancels the corresponding query and frees the memory associated with the open request.

Example: Close a cursor.

```
cursor.close()
```

Manipulating databases

db_create

r.db_create(db_name) → object

Create a database. A RethinkDB database is a collection of tables, similar to relational databases.

If successful, the operation returns an object: `{"created": 1}`. If a database with the same name already exists the operation throws `RqlRuntimeError`.

Note: that you can only use alphanumeric characters and underscores for the database name.

Example: Create a database named 'superheroes'.

```
r.db_create('superheroes').run(conn)
```

db_drop

r.db_drop(db_name) → object

Drop a database. The database, all its tables, and corresponding data will be deleted.

If successful, the operation returns the object `{"dropped": 1}`. If the specified database doesn't exist a `RqlRuntimeError` is thrown.

Example: Drop a database named 'superheroes'.

```
r.db_drop('superheroes').run(conn)
```

db_list

r.db_list() → array

List all database names in the system. The result is a list of strings.

Example: List all databases.

```
r.db_list().run(conn)
```

Manipulating tables

table_create

db.table_create(table_name[, options]) → object

Create a table. A RethinkDB table is a collection of JSON documents.

If successful, the operation returns an object: `{created: 1}`. If a table with the same name already exists, the operation throws `RqlRuntimeError`.

Note: that you can only use alphanumeric characters and underscores for the table name.

When creating a table you can specify the following options:

- `primary_key`: the name of the primary key. The default primary key is id;
- `durability`: if set to `soft`, this enables *soft durability* on this table: writes will be acknowledged by the server immediately and flushed to disk in the background. Default is `hard` (acknowledgement of writes happens after data has been written to disk);

- `cache_size`: set the cache size (in bytes) to be used by the table. The default is 1073741824 (1024MB);
- `datacenter`: the name of the datacenter this table should be assigned to.

Example: Create a table named 'dc_universe' with the default settings.

```
r.db('test').table_create('dc_universe').run(conn)
```

Read more about this command →

table_drop

db.table_drop(table_name) → object

Drop a table. The table and all its data will be deleted.

If succesful, the operation returns an object: {"dropped": 1}. If the specified table doesn't exist a `RqlRuntimeError` is thrown.

Example: Drop a table named 'dc_universe'.

```
r.db('test').table_drop('dc_universe').run(conn)
```

table_list

db.table_list() → array

List all table names in a database. The result is a list of strings.

Example: List all tables of the 'test' database.

```
r.db('test').table_list().run(conn)
```

index_create

table.index_create(index_name[, index_function]) → object

Create a new secondary index on this table.

Example: To efficiently query our heros by code name we have to create a secondary index.

```
r.table('dc').index_create('code_name').run(conn)
```

Read more about this command →

index_drop

table.index_drop(index_name) → object

Delete a previously created secondary index of this table.

Example: Drop a secondary index named 'code_name'.

```
r.table('dc').index_drop('code_name').run(conn)
```

index_list

table.index_list() → array

List all the secondary indexes of this table.

Example: List the available secondary indexes for this table.

```
r.table('marvel').index_list().run(conn)
```

index_status

table.index_status([, index...]) → array

Get the status of the specified indexes on this table, or the status of all indexes on this table if no indexes are specified.

Example: Get the status of all the indexes on `test`:

```
r.table('test').index_status().run(conn)
```

Example: Get the status of the `timestamp` index:

```
r.table('test').index_status('timestamp').run(conn)
```

index_wait

table.index_wait([, index...]) → array

Wait for the specified indexes on this table to be ready, or for all indexes on this table to be ready if no indexes are specified.

Example: Wait for all indexes on the table `test` to be ready:

```
r.table('test').index_wait().run(conn)
```

Example: Wait for the index `timestamp` to be ready:

```
r.table('test').index_wait('timestamp').run(conn)
```

Writing data

insert

table.insert(json | [json] [, durability="hard", return_vals=False, upsert=False]) → object

Insert documents into a table. Accepts a single document or an array of documents.

Example: Insert a document into the table `posts`.

```
r.table("posts").insert({
    "id": 1,
    "title": "Lorem ipsum",
    "content": "Dolor sit amet"
}).run(conn)
```

Read more about this command →

update

table.update(json | exp [, durability="hard", return_vals=False, non_atomic=False]) → object

selection.update(json | exp [, durability="hard", return_vals=False, non_atomic=False]) → object

singleSelection.update(json | exp [, durability="hard", return_vals=False, non_atomic=False]) → object

Update JSON documents in a table. Accepts a JSON document, a ReQL expression, or a combination of the two.

Example: Update the status of the post with `id` of `1` to `published`.

```
r.table("posts").get(1).update({"status": "published"}).run(conn)
```

Read more about this command →

replace

table.replace(json | expr [, durability="hard", return_vals=False, non_atomic=False]) → object

selection.replace(json | expr [, durability="hard", return_vals=False, non_atomic=False]) → object

singleSelection.replace(json | expr [, durability="hard", return_vals=False, non_atomic=False]) → object

Replace documents in a table. Accepts a JSON document or a ReQL expression, and replaces the original document with the new one. The new document must have the same primary key as the original document.

Example: Replace the document with the primary key `1`.

```
r.table("posts").get(1).replace({
    "id": 1,
    "title": "Lorem ipsum",
    "content": "Aleas jacta est",
    "status": "draft"
}).run(conn)
```

Read more about this command →

delete

table.delete([durability="hard", return_vals=False]) → object

selection.delete([durability="hard", return_vals=False]) → object

singleSelection.delete([durability="hard", return_vals=False]) → object

Delete one or more documents from a table.

Example: Delete a single document from the table `comments`.

```
r.table("comments").get("7eab9e63-73f1-4f33-8ce4-95cbea626f59").delete().run(conn)
```

Read more about this command →

sync

table.sync() → object

`sync` ensures that writes on a given table are written to permanent storage. Queries that specify soft durability (`durability='soft'`) do not give such guarantees, so `sync` can be used to ensure the state of these queries. A call to `sync` does not return until all previous writes to the table are persisted.

Example: After having updated multiple heroes with soft durability, we now want to wait until these changes are persisted.

```
r.table('marvel').sync().run(conn)
```

Selecting data

db

r.db(db_name) → db

Reference a database.

Example: Before we can query a table we have to select the correct database.

```
r.db('heroes').table('marvel').run(conn)
```

table

db.table(name[, use_outdated=False]) → table

Select all documents in a table. This command can be chained with other commands to do further processing on the data.

Example: Return all documents in the table 'marvel' of the default database.

```
r.table('marvel').run(conn)
```

Read more about this command →

get

table.get(key) → singleRowSelection

Get a document by primary key.

Example: Find a document with the primary key 'superman'.

```
r.table('marvel').get('superman').run(conn)
```

get_all

table.get_all(key1[, key2...], [, index='id']) → selection

Get all documents where the given value matches the value of the requested index.

Example: Secondary index keys are not guaranteed to be unique so we cannot query via "get" when using a secondary index.

```
r.table('marvel').get_all('man_of_steel', index='code_name').run(conn)
```

Read more about this command →

between

table.between(lower_key, upper_key [, index='id', left_bound='closed', right_bound='open']) → selection

Get all documents between two keys. Accepts three optional arguments: `index`, `left_bound`, and `right_bound`. If `index` is set to the name of a secondary index, `between` will return all documents where that index's value is in the specified range (it uses the primary key by default). `left_bound` or `right_bound` may be set to `open` or `closed` to indicate whether or not to include that endpoint of the range (by default, `left_bound` is closed and `right_bound` is open).

Example: Find all users with primary key >= 10 and < 20 (a normal half-open interval).

```
r.table('marvel').between(10, 20).run(conn)
```

Read more about this command →

filter

sequence.filter(predicate, default=False) → selection

stream.filter(predicate, default=False) → stream

array.filter(predicate, default=False) → array

Get all the documents for which the given predicate is true.

`filter` can be called on a sequence, selection, or a field containing an array of elements. The return type is the same as the type on which the function was called on.

The body of every filter is wrapped in an implicit `.default(False)`, which means that if a non-existence errors is thrown (when you try to access a field that does not exist in a document), RethinkDB will just ignore the document. The `default` value can be changed by passing the named argument `default`. Setting this optional argument to `r.error()` will cause any non-existence errors to return a `RqlRuntimeError`.

Example: Get all the users that are 30 years old.

```
r.table('users').filter({"age": 30}).run(conn)
```

Read more about this command →

Joins

These commands allow the combination of multiple sequences into a single sequence

inner_join

sequence.inner_join(other_sequence, predicate) → stream

array.inner_join(other_sequence, predicate) → array

Returns the inner product of two sequences (e.g. a table, a filter result) filtered by the predicate. The query compares each row of the left sequence with each row of the right sequence to find all pairs of rows which satisfy the predicate. When the predicate is satisfied, each matched pair of rows of both sequences are combined into a result row.

Example: Construct a sequence of documents containing all cross-universe matchups where a marvel hero would lose.

```
r.table('marvel').inner_join(r.table('dc'), lambda marvelRow, dcRow:
    marvelRow['strength'] < dcRow['strength']).run(conn)
```

outer_join

sequence.outer_join(other_sequence, predicate) → stream

array.outer_join(other_sequence, predicate) → array

Computes a left outer join by retaining each row in the left table even if no match was found in the right table.

Example: Construct a sequence of documents containing all cross-universe matchups where a marvel hero would lose, but keep marvel heroes who would never lose a matchup in the sequence.

```
r.table('marvel').outer_join(r.table('dc'),
 lambda marvelRow, dcRow: marvelRow['strength'] < dcRow['strength']).run(conn)
```

eq_join

sequence.eq_join(left_attr, other_table[, index='id']) → stream

array.eq_join(left_attr, other_table[, index='id']) → array

An efficient join that looks up elements in the right table by primary key.

Example: Let our heroes join forces to battle evil!

```
r.table('marvel').eq_join('main_dc_collaborator', r.table('dc')).run(conn)
```

Read more about this command →

zip

stream.zip() → stream

array.zip() → array

Used to 'zip' up the result of a join by merging the 'right' fields into 'left' fields of each member of the sequence.

Example: 'zips up' the sequence by merging the left and right fields produced by a join.

```
r.table('marvel').eq_join('main_dc_collaborator', r.table('dc')).zip().run(conn)
```

Transformations

These commands are used to transform data in a sequence.

map

sequence.map(mapping_function) → stream

array.map(mapping_function) → array

Transform each element of the sequence by applying the given mapping function.

Example: Construct a sequence of hero power ratings.

```
r.table('marvel').map(lambda hero:
    hero['combatPower'] + hero['compassionPower'] * 2
).run(conn)
```

with_fields

sequence.with_fields([selector1, selector2...]) → stream

array.with_fields([selector1, selector2...]) → array

Takes a sequence of objects and a list of fields. If any objects in the sequence don't have all of the specified fields, they're dropped from the sequence. The

remaining objects have the specified fields plucked out. (This is identical to `has_fields` followed by `pluck` on a sequence.)

Example: Get a list of heroes and their nemeses, excluding any heroes that lack one.

```
r.table('marvel').with_fields('id', 'nemesis')
```

Read more about this command →

concat_map

sequence.concat_map(mapping_function) → stream

array.concat_map(mapping_function) → array

Flattens a sequence of arrays returned by the mappingFunction into a single sequence.

Example: Construct a sequence of all monsters defeated by Marvel heroes. Here the field 'defeatedMonsters' is a list that is concatenated to the sequence.

```
r.table('marvel').concat_map(lambda hero: hero['defeatedMonsters']).run(conn)
```

order_by

table.order_by([key1...], index=index_name) -> selection<stream>

selection.order_by(key1, [key2...]) -> selection<array>

sequence.order_by(key1, [key2...]) -> array

Sort the sequence by document values of the given key(s). `orderBy` defaults to ascending ordering. To explicitly specify the ordering, wrap the attribute with either `r.asc` or `r.desc`.

Example: Order our heroes by a series of performance metrics.

```
r.table('marvel').order_by('enemies_vanquished', 'damsels_saved').run(conn)
```

Read more about this command →

skip

sequence.skip(n) → stream

array.skip(n) → array

Skip a number of elements from the head of the sequence.

Example: Here in conjunction with `order_by` we choose to ignore the most successful heroes.

```
r.table('marvel').order_by('successMetric').skip(10).run(conn)
```

limit

sequence.limit(n) → stream

array.limit(n) → array

End the sequence after the given number of elements.

Example: Only so many can fit in our Pantheon of heroes.

```
r.table('marvel').order_by('belovedness').limit(10).run(conn)
```

[]

sequence[start_index[:end_index]] → stream

array[start_index[:end_index]] → array

Trim the sequence to within the bounds provided.

Example: For this fight, we need heroes with a good mix of strength and agility.

```
r.table('marvel').order_by('strength')[5:10].run(conn)
```

[]

sequence[index] &arr; object

sequence.nth(index) → object

Get the nth element of a sequence.

Example: Select the second element in the array.

```
r.expr([1,2,3])[1].run(conn)
r.expr([1,2,3]).nth(1).run(conn)
```

indexes_of

sequence.indexes_of(datum | predicate) → array

Get the indexes of an element in a sequence. If the argument is a predicate, get the indexes of all elements matching it.

Example: Find the position of the letter 'c'.

```
r.expr(['a','b','c']).indexes_of('c').run(conn)
```

Read more about this command →

is_empty

sequence.is_empty() → bool

Test if a sequence is empty.

Example: Are there any documents in the marvel table?

```
r.table('marvel').is_empty().run(conn)
```

union

sequence.union(sequence) → array

Concatenate two sequences.

Example: Construct a stream of all heroes.

```
r.table('marvel').union(r.table('dc')).run(conn)
```

sample

sequence.sample(number) → selection

stream.sample(number) → array

array.sample(number) → array

Select a given number of elements from a sequence with uniform random distribution. Selection is done without replacement.

Example: Select 3 random heroes.

```
r.table('marvel').sample(3).run(conn)
```

Aggregation

These commands are used to compute smaller values from large sequences.

reduce

sequence.reduce(reduction_function[, base]) → value

Produce a single value from a sequence through repeated application of a reduction function.

The reduce function gets invoked repeatedly not only for the input values but also for results of previous reduce invocations. The type and format of the object that is passed in to reduce must be the same with the one returned from reduce.

Example: How many enemies have our heroes defeated?

```
r.table('marvel').map(r.row['monstersKilled']).reduce(
    lambda acc, val: acc + val, 0).run(conn)
```

count

sequence.count([filter]) → number

Count the number of elements in the sequence. With a single argument, count the number of elements equal to it. If the argument is a function, it is equivalent to calling filter before count.

Example: Just how many super heroes are there?

```
(r.table('marvel').count() + r.table('dc').count()).run(conn)
```

Read more about this command →

distinct

sequence.distinct() → array

Remove duplicate elements from the sequence.

Example: Which unique villains have been vanquished by marvel heroes?

```
r.table('marvel').concat_map(lambda hero: hero['villainList']).distinct().run(conn)
```

grouped_map_reduce

sequence.grouped_map_reduce(grouping, mapping, reduction, base) → value

Partition the sequence into groups based on the `grouping` function. The elements of each group are then mapped using the `mapping` function and reduced using the `reduction` function.

`grouped_map_reduce` is a generalized form of group by.

Example: It's only fair that heroes be compared against their weight class.

```
r.table('marvel').grouped_map_reduce(
    lambda hero: hero['weightClass'],  # grouping
    lambda hero: hero.pluck('name', 'strength'),  # mapping
   lambda acc, hero: r.branch(acc['strength'] < hero['strength'], hero, acc),
    {'name':'none', 'strength':0}  # base
).run(conn)
```

group_by

sequence.group_by(selector1[, selector2...], reduction_object) → array

Groups elements by the values of the given attributes and then applies the given reduction. Though similar to `groupedMapReduce`, `groupBy` takes a standardized object for specifying the reduction. Can be used with a number of predefined common reductions.

Example: Using a predefined reduction we can easily find the average strength of members of each weight class.

```
r.table('marvel').group_by('weightClass', r.avg('strength')).run(conn)
```

Read more about this command →

contains

sequence.contains(value1[, value2...]) → bool

Returns whether or not a sequence contains all the specified values, or if functions are provided instead, returns whether or not a sequence contains values matching all the specified functions.

Example: Has Iron Man ever fought Superman?

```
r.table('marvel').get('ironman')['opponents'].contains('superman').run(conn)
```

Read more about this command →

Aggregators

These standard aggregator objects are to be used in conjunction with groupBy.

count

r.count

Count the total size of the group.

Example: Just how many heroes do we have at each strength level?

```
r.table('marvel').group_by('strength', r.count).run(conn)
```

sum

r.sum(attr)

Compute the sum of the given field in the group.

Example: How many enemies have been vanquished by heroes at each strength level?

```
r.table('marvel').group_by('strength', r.sum('enemiesVanquished')).run(conn)
```

avg

r.avg(attr)

Compute the average value of the given attribute for the group.

Example: What's the average agility of heroes at each strength level?

```
r.table('marvel').group_by('strength', r.avg('agility')).run(conn)
```

Document manipulation

row

r.row → value

Returns the currently visited document.

Example: Get all users whose age is greater than 5.

```
r.table('users').filter(r.row['age'] > 5).run(conn)
```

Read more about this command →

pluck

sequence.pluck([selector1, selector2...]) → stream

array.pluck([selector1, selector2...]) → array

object.pluck([selector1, selector2...]) → object

singleSelection.pluck([selector1, selector2...]) → object

Plucks out one or more attributes from either an object or a sequence of objects (projection).

Example: We just need information about IronMan's reactor and not the rest of the document.

```
r.table('marvel').get('IronMan').pluck('reactorState', 'reactorPower').run(conn)
```

Read more about this command →

without

sequence.without([selector1, selector2...]) → stream

array.without([selector1, selector2...]) → array

singleSelection.without([selector1, selector2...]) → object

object.without([selector1, selector2...]) → object

The opposite of pluck; takes an object or a sequence of objects, and returns them with the specified paths removed.

Example: Since we don't need it for this computation we'll save bandwidth and leave out the list of IronMan's romantic conquests.

```
r.table('marvel').get('IronMan').without('personalVictoriesList').run(conn)
```

Read more about this command →

merge

singleSelection.merge(object) → object

object.merge(object) → object

sequence.merge(object) → stream

array.merge(object) → array

Merge two objects together to construct a new object with properties from both. Gives preference to attributes from other when there is a conflict.

Example: Equip IronMan for battle.

```
r.table('marvel').get('IronMan').merge(
    r.table('loadouts').get('alienInvasionKit')
).run(conn)
```

Read more about this command →

append

array.append(value) → array

Append a value to an array.

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')['equipment'].append('newBoots').run(conn)
```

prepend

array.prepend(value) → array

Prepend a value to an array.

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')['equipment'].prepend('newBoots').run(conn)
```

difference

array.difference(array) → array

Remove the elements of one array from another array.

Example: Retrieve Iron Man's equipment list without boots.

```
r.table('marvel').get('IronMan')['equipment'].difference(['Boots']).run(conn)
```

set_insert

array.set_insert(value) → array

Add a value to an array and return it as a set (an array with distinct values).

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')['equipment'].set_insert('newBoots').run(conn)
```

set_union

array.set_union(array) → array

Add a several values to an array and return it as a set (an array with distinct values).

Example: Retrieve Iron Man's equipment list with the addition of some new boots and an arc reactor.

```
r.table('marvel').get('IronMan')['equipment'].set_union(['newBoots', 'arc_reactor']).run(
```

set_intersection

array.set_intersection(array) → array

Intersect two arrays returning values that occur in both of them as a set (an array with distinct values).

Example: Check which pieces of equipment Iron Man has from a fixed list.

```
r.table('marvel').get('IronMan')['equipment'].set_intersection(['newBoots', 'arc_reactor'
```

set_difference

array.set_difference(array) → array

Remove the elements of one array from another and return them as a set (an array with distinct values).

Example: Check which pieces of equipment Iron Man has, excluding a fixed list.

```
r.table('marvel').get('IronMan')['equipment'].set_difference(['newBoots', 'arc_reactor'])
```

[]

sequence[attr] → sequence

singleSelection[attr] → value

object[attr] → value

Get a single field from an object. If called on a sequence, gets that field from every object in the sequence, skipping objects that lack it.

Example: What was Iron Man's first appearance in a comic?

```
r.table('marvel').get('IronMan')['firstAppearance'].run(conn)
```

has_fields

sequence.has_fields([selector1, selector2...]) → stream

array.has_fields([selector1, selector2...]) → array

singleSelection.has_fields([selector1, selector2...]) → boolean

object.has_fields([selector1, selector2...]) → boolean

Test if an object has all of the specified fields. An object has a field if it has the specified key and that key maps to a non-null value. For instance, the object `{'a':1,'b':2,'c': None}` has the fields `a` and `b`.

Example: Which heroes are married?

```
r.table('marvel').has_fields('spouse').run(conn)
```

Read more about this command →

insert_at

array.insert_at(index, value) → array

Insert a value in to an array at a given index. Returns the modified array.

Example: Hulk decides to join the avengers.

```
r.expr(["Iron Man", "Spider-Man"]).insert_at(1, "Hulk").run(conn)
```

splice_at

array.splice_at(index, array) → array

Insert several values in to an array at a given index. Returns the modified array.

Example: Hulk and Thor decide to join the avengers.

```
r.expr(["Iron Man", "Spider-Man"]).splice_at(1, ["Hulk", "Thor"]).run(conn)
```

delete_at

array.delete_at(index [,endIndex]) → array

Remove an element from an array at a given index. Returns the modified array.

Example: Hulk decides to leave the avengers.

```
r.expr(["Iron Man", "Hulk", "Spider-Man"]).delete_at(1).run(conn)
```

Read more about this command →

change_at

array.change_at(index, value) → array

Change a value in an array at a given index. Returns the modified array.

Example: Bruce Banner hulks out.

```
r.expr(["Iron Man", "Bruce", "Spider-Man"]).change_at(1, "Hulk").run(conn)
```

keys

singleSelection.keys() → array

object.keys() → array

Return an array containing all of the object's keys.

Example: Get all the keys of a row.

```
r.table('marvel').get('ironman').keys().run(conn)
```

String manipulation

These commands provide string operators.

match

string.match(regexp) → array

Match against a regular expression. Returns a match object containing the matched string, that string's start/end position, and the capture groups. Accepts RE2 syntax (https://code.google.com/p/re2/wiki/Syntax). You can enable case-insensitive matching by prefixing the regular expression with `(?i)`. (See linked RE2 documentation for more flags.)

Example: Get all users whose name starts with A.

```
r.table('users').filter(lambda row:row['name'].match("^A")).run(conn)
```

Read more about this command →

Math and logic

+

number + number → number

string + string → string

array + array → array

time + number → time

Sum two numbers, concatenate two strings, or concatenate 2 arrays.

Example: It's as easy as 2 + 2 = 4.

```
(r.expr(2) + 2).run(conn)
```

Read more about this command →

-

number - number → number

time - time → number

time - number → time

Subtract two numbers.

Example: It's as easy as 2 - 2 = 0.

```
(r.expr(2) - 2).run(conn)
```

Read more about this command →

*

number * number → number

array * number → array

Multiply two numbers, or make a periodic array.

Example: It's as easy as 2 * 2 = 4.

```
(r.expr(2) * 2).run(conn)
```

Read more about this command →

/

number / number → number

Divide two numbers.

Example: It's as easy as 2 / 2 = 1.

```
(r.expr(2) / 2).run(conn)
```

%

number % number → number

Find the remainder when dividing two numbers.

Example: It's as easy as 2 % 2 = 0.

```
(r.expr(2) % 2).run(conn)
```

&

bool & bool → bool

Compute the logical and of two values.

Example: True and false anded is false?

```
(r.expr(True) & False).run(conn)
```

|

bool | bool → bool

Compute the logical or of two values.

Example: True or false ored is true?

```
(r.expr(True) | False).run(conn)
```

==, eq

value == value → bool

value.eq(value) → bool

Test if two values are equal.

Example: Does 2 equal 2?

```
(r.expr(2) == 2).run(conn)
r.expr(2).eq(2).run(conn)
```

!=, ne

value != value → bool

value.ne(value) → bool

Description

Test if two values are not equal.

Example: Does 2 not equal 2?

```
(r.expr(2) != 2).run(conn)
r.expr(2).ne(2).run(conn)
```

>, gt

value > value → bool

value.gt(value) → bool

Test if the first value is greater than other.

Example: Is 2 greater than 2?

```
(r.expr(2) > 2).run(conn)
r.expr(2).gt(2).run(conn)
```

>=, ge

value >= value → bool

value.ge(value) → bool

Description

Test if the first value is greater than or equal to other.

Example: Is 2 greater than or equal to 2?

```
(r.expr(2) >= 2).run(conn)
r.expr(2).ge(2).run(conn)
```

<, lt

value < value → bool

value.lt(value) → bool

Description

Test if the first value is less than other.

Example: Is 2 less than 2?

```
(r.expr(2) < 2).run(conn)
r.expr(2).lt(2).run(conn)
```

<=, le

value <= value → bool

value.le(value) → bool

Description

Test if the first value is less than or equal to other.

Example: Is 2 less than or equal to 2?

```
(r.expr(2) <= 2).run(conn)
r.expr(2).le(2).run(conn)
```

~

~bool → bool

bool.not_() → bool

Compute the logical inverse (not).

Example: Not true is false.

```
(~r.expr(True)).run(conn)
```

Read more about this command →

Dates and times

now

r.now() → time

Return a time object representing the current time in UTC. The command now() is computed once when the server receives the query, so multiple instances of r.now() will always return the same time inside a query.

Example: Add a new user with the time at which he subscribed.

```
r.table("users").insert({
    "name": "John",
    "subscription_date": r.now()
}).run(conn)
```

time

r.time(year, month, day[, hour, minute, second], timezone) → time

Create a time object for a specific time.

A few restrictions exist on the arguments:

- `year` is an integer between 1400 and 9,999.
- `month` is an integer between 1 and 12.
- `day` is an integer between 1 and 31.
- `hour` is an integer.
- `minutes` is an integer.
- `seconds` is a double. Its value will be rounded to three decimal places (millisecond-precision).
- `timezone` can be `'Z'` (for UTC) or a string with the format `±[hh]:[mm]`.

Example: Update the birthdate of the user "John" to November 3rd, 1986 UTC.

```
r.table("user").get("John").update({"birthdate": r.time(1986, 11, 3, 'Z')}).run(conn)
```

epoch_time

r.epoch_time(epoch_time) → time

Create a time object based on seconds since epoch. The first argument is a double and will be rounded to three decimal places (millisecond-precision).

Example: Update the birthdate of the user "John" to November 3rd, 1986.

```
r.table("user").get("John").update({"birthdate": r.epoch_time(531360000)}).run(conn)
```

iso8601

r.iso8601(iso8601Date[, default_timezone=”]) → time

Create a time object based on an iso8601 date-time string (e.g. ‘2013-01-01T01:01:01+00:00’). We support all valid ISO 8601 formats except for week dates. If you pass an ISO 8601 date-time without a time zone, you must specify the time zone with the optarg `default_timezone`. Read more about the ISO 8601 format on the Wikipedia page.

Example: Update the time of John’s birth.

```
r.table("user").get("John").update({"birth": r.iso8601('1986-11-03T08:30:00-07:00')}).run
```

in_timezone

time.in_timezone(timezone) → time

Return a new time object with a different timezone. While the time stays the same, the results returned by methods such as hours() will change since they take the timezone into account. The timezone argument has to be of the ISO 8601 format.

Example: Hour of the day in San Francisco (UTC/GMT -8, without daylight saving time).

```
r.now().in_timezone('-08:00').hours().run(conn)
```

timezone

time.timezone() → string

Return the timezone of the time object.

Example: Return all the users in the “-07:00” timezone.

```
r.table("users").filter(lambda user:
    user["subscriptionDate"].timezone() == "-07:00"
)
```

during

time.during(start_time, end_time [, left_bound="open/closed", right_bound="open/closed"]) → bool

Return if a time is between two other times (by default, inclusive for the start, exclusive for the end).

Example: Retrieve all the posts that were posted between December 1st, 2013 (inclusive) and December 10th, 2013 (exclusive).

```
r.table("posts").filter(
   r.row['date'].during(r.time(2013, 12, 1, "Z"), r.time(2013, 12, 10, "Z"))
).run(conn)
```

Read more about this command →

date

time.date() → time

Return a new time object only based on the day, month and year (ie. the same day at 00:00).

Example: Retrieve all the users whose birthday is today

```
r.table("users").filter(lambda user:
    user["birthdate"].date() == r.now().date()
).run(conn)
```

time_of_day

time.time_of_day() → number

Return the number of seconds elapsed since the beginning of the day stored in the time object.

Example: Retrieve posts that were submitted before noon.

```
r.table("posts").filter(
    r.row["date"].time_of_day() <= 12*60*60
).run(conn)
```

year

time.year() → number

Return the year of a time object.

Example: Retrieve all the users born in 1986.

```
r.table("users").filter(lambda user:
    user["birthdate"].year() == 1986
).run(conn)
```

month

time.month() → number

Return the month of a time object as a number between 1 and 12. For your convenience, the terms r.january, r.february etc. are defined and map to the appropriate integer.

Example: Retrieve all the users who were born in November.

```
r.table("users").filter(
    r.row["birthdate"].month() == 11
)
```

Read more about this command →

day

time.day() → number

Return the day of a time object as a number between 1 and 31.

Example: Return the users born on the 24th of any month.

```
r.table("users").filter(
    r.row["birthdate"].day() == 24
)
```

day_of_week

time.day_of_week() → number

Return the day of week of a time object as a number between 1 and 7 (following ISO 8601 standard). For your convenience, the terms r.monday, r.tuesday etc. are defined and map to the appropriate integer.

Example: Return today's day of week.

```
r.now().day_of_week().run(conn)
```

Read more about this command →

day_of_year

time.day_of_year() → number

Return the day of the year of a time object as a number between 1 and 366 (following ISO 8601 standard).

Example: Retrieve all the users who were born the first day of a year.

```
r.table("users").filter(
    r.row["birthdate"].day_of_year() == 1
).run(conn)
```

hours

time.hours() → number

Return the hour in a time object as a number between 0 and 23.

Example: Return all the posts submitted after midnight and before 4am.

```
r.table("posts").filter(lambda post:
    post["date"].hours() < 4
).run(conn)
```

minutes

time.minutes() → number

Return the minute in a time object as a number between 0 and 59.

Example: Return all the posts submitted during the first 10 minutes of every hour.

```
r.table("posts").filter(lambda post:
    post["date"].minutes() < 10
).run(conn)
```

seconds

time.seconds() → number

Return the seconds in a time object as a number between 0 and 59.999 (double precision).

Example: Return the post submitted during the first 30 seconds of every minute.

```
r.table("posts").filter(lambda post:
    post["date"].seconds() < 30
).run(conn)
```

to_iso8601

time.to_iso8601() → number

Convert a time object to its iso 8601 format.

Example: Return the current time in an ISO8601 format.

```
r.now().to_iso8601()
```

to_epoch_time

time.to_epoch_time() → number

Convert a time object to its epoch time.

Example: Return the current time in an ISO8601 format.

```
r.now().to_epoch_time()
```

Control structures

do

any.do(arg [, args]*, expr) → any

Evaluate the expr in the context of one or more value bindings.

The type of the result is the type of the value returned from expr.

Example: The object(s) passed to do() can be bound to name(s). The last argument is the expression to evaluate in the context of the bindings.

```
r.do(r.table('marvel').get('IronMan'),
    lambda ironman: ironman['name']).run(conn)
```

branch

r.branch(test, true_branch, false_branch) → any

If the `test` expression returns `False` or `None`, the `false_branch` will be evaluated. Otherwise, the `true_branch` will be evaluated.

The `branch` command is effectively an `if` renamed due to language constraints.

Example: Return heroes and superheroes.

r.table('marvel').map(r.branch(r.row['victories'] > 100, r.row['name'] + 'is a superhero', r.row['name'] + 'is a hero')).run(conn) "'

for_each

sequence.for_each(write_query) → object

Loop over a sequence, evaluating the given write query for each element.

Example: Now that our heroes have defeated their villains, we can safely remove them from the villain table.

```
r.table('marvel').for_each(
   lambda hero: r.table('villains').get(hero['villainDefeated']).delete()
).run(conn)
```

error

r.error(message) → error

Throw a runtime error. If called with no arguments inside the second argument to `default`, re-throw the current error.

Example: Iron Man can't possibly have lost a battle:

```
r.table('marvel').get('IronMan').do(
   lambda ironman: r.branch(ironman['victories'] < ironman['battles'],
                            r.error('impossible code path'),
                            ironman)
).run(conn)
```

default

value.default(default_value) → any

sequence.default(default_value) → any

Handle non-existence errors. Tries to evaluate and return its first argument. If an error related to the absence of a value is thrown in the process, or if its first argument returns `None`, returns its second argument. (Alternatively, the second argument may be a function which will be called with either the text of the non-existence error or `None`.)

Exmple: Suppose we want to retrieve the titles and authors of the table `posts`. In the case where the author field is missing or `None`, we want to retrieve the string `Anonymous`.

```
r.table("posts").map(lambda post:
    {
        "title": post["title"],
        "author": post["author"].default("Anonymous")
    }
).run(conn)
```

Read more about this command →

expr

r.expr(value) → value

Construct a ReQL JSON object from a native object.

Example: Objects wrapped with `expr` can then be manipulated by ReQL API functions.

```
r.expr({'a':'b'}).merge({'b':[1,2,3]}).run(conn)
```

js

r.js(jsString) → value

Create a javascript expression.

Example: Concatenate two strings using Javascript'

```
r.js("'str1' + 'str2'").run(conn)
```

Read more about this command →

coerce_to

sequence.coerce_to(type_name) → array

value.coerce_to(type_name) → string

array.coerce_to(type_name) → object

object.coerce_to(type_name) → array

Converts a value of one type into another.

You can convert: a selection, sequence, or object into an ARRAY, an array of pairs into an OBJECT, and any DATUM into a STRING.

Example: Convert a table to an array.

```
r.table('marvel').coerce_to('array').run(conn)
```

Read more about this command →

type_of

any.type_of() → string

Gets the type of a value.

Example: Get the type of a string.

```
r.expr("foo").type_of().run(conn)
```

info

any.info() → object

Get information about a ReQL value.

Example: Get information about a table such as primary key, or cache size.

```
r.table('marvel').info().run(conn)
```

json

r.json(json_string) → value

Parse a JSON string on the server.

Example: Send an array to the server'

```
r.json("[1,2,3]").run(conn)
```

Command syntax

time.seconds() → number

Description

Return the seconds in a time object as a number between 0 and 59.999 (double precision).

Example: Return the post submitted during the first 30 seconds of every minute.

```
r.table("posts").filter(lambda post:
    post["date"].seconds() < 30
).run(conn)
```

Command syntax

r.avg(attr)

Description

Compute the average value of the given attribute for the group.

Example: What's the average agility of heroes at each strength level?

```
r.table('marvel').group_by('strength', r.avg('agility')).run(conn)
```

Command syntax

cursor.close()

Description

Close a cursor. Closing a cursor cancels the corresponding query and frees the memory associated with the open request.

Example: Close a cursor.

```
cursor.close()
```

Command syntax

query.run(conn, use_outdated=False, time_format='native', profile=False) → cursor

query.run(conn, use_outdated=False, time_format='native', profile=False) → object

Description

Run a query on a connection, returning either a single JSON result or a cursor, depending on the query.

Example: Run a query on the connection `conn` and print out every row in the result.

```
for doc in r.table('marvel').run(conn):
    print doc
```

Example: If you are OK with potentially out of date data from all the tables involved in this query and want potentially faster reads, pass a flag allowing out of date data in an options object. Settings for individual tables will supercede this global setting for all tables in the query.

```
r.table('marvel').run(conn, use_outdated=True)
```

Example: If you just want to send a write and forget about it, you can set `noreply` to true in the options. In this case `run` will return immediately.

```
r.table('marvel').run(conn, noreply=True)
```

Example: If you want to specify whether to wait for a write to be written to disk (overriding the table's default settings), you can set `durability` to `'hard'` or `'soft'` in the options.

```
r.table('marvel')
   .insert({ 'superhero': 'Iron Man', 'superpower': 'Arc Reactor' })
    .run(conn, noreply=True, durability='soft')
```

Example: If you do not want a time object to be converted to a native date object, you can pass a `time_format` flag to prevent it (valid flags are "raw" and "native"). This query returns an object with two fields (`epoch_time` and `$reql_type$`) instead of a native date object.

```
r.now().run(conn, time_format="raw")
```

Command syntax

sequence.concat_map(mapping_function) → stream

array.concat_map(mapping_function) → array

Description

Flattens a sequence of arrays returned by the mappingFunction into a single sequence.

Example: Construct a sequence of all monsters defeated by Marvel heroes. Here the field 'defeatedMonsters' is a list that is concatenated to the sequence.

```
r.table('marvel').concat_map(lambda hero: hero['defeatedMonsters']).run(conn)
```

Command syntax

r.db_list() → array

Description

List all database names in the system. The result is a list of strings.

Example: List all databases.

```
r.db_list().run(conn)
```

Command syntax

number % number → number

Description

Find the remainder when dividing two numbers.

Example: It's as easy as 2 % 2 = 0.

```
(r.expr(2) % 2).run(conn)
```

‘

Command syntax

conn.noreply_wait()

Description

`noreply_wait` ensures that previous queries with the `noreply` flag have been processed by the server. Note that this guarantee only applies to queries run on the given connection.

Example: We have previously run queries with the `noreply` argument set to `True`. Now wait until the server has processed them.

```
conn.noreply_wait()
```

Command syntax

time.day_of_year() → number

Description

Return the day of the year of a time object as a number between 1 and 366 (following ISO 8601 standard).

Example: Retrieve all the users who were born the first day of a year.

```
r.table("users").filter(
    r.row["birthdate"].day_of_year() == 1
).run(conn)
```

Command syntax

number - number → number

time - time → number

time - number → time

Description

Subtract two numbers.

Example: It's as easy as 2 - 2 = 0.

```
(r.expr(2) - 2).run(conn)
```

Example: Create a date one year ago today.

```
r.now() - 365*24*60*60
```

Example: Retrieve how many seconds elapsed between today and date

```
r.now() - date
```

Command syntax

stream.zip() → stream

array.zip() → array

Description

Used to 'zip' up the result of a join by merging the 'right' fields into 'left' fields of each member of the sequence.

Example: 'zips up' the sequence by merging the left and right fields produced by a join.

```
r.table('marvel').eq_join('main_dc_collaborator', r.table('dc')).zip().run(conn)
```

Command syntax

table.insert(json | [json][, durability="hard", return_vals=False, upsert=False]) → object

Description

Insert documents into a table. Accepts a single document or an array of documents.

The optional arguments are:

- `durability`: possible values are `hard` and `soft`. This option will override the table or query's durability setting (set in run).
  In soft durability mode RethinkDB will acknowledge the write immediately after receiving it, but before the write has been committed to disk.
- `return_vals`: if set to `True` and in case of a single insert/upsert, the inserted/updated document will be returned.
- `upsert`: when set to `True`, performs a replace if a document with the same primary key exists.

Insert returns an object that contains the following attributes:

- `inserted`: the number of documents that were succesfully inserted.
- `replaced`: the number of documents that were updated when upsert is used.
- `unchanged`: the number of documents that would have been modified, except that the new value was the same as the old value when doing an upsert.
- `errors`: the number of errors encountered while performing the insert.
- `first_error`: If errors were encountered, contains the text of the first error.
- `deleted` and `skipped`: 0 for an insert operation.
- `generated_keys`: a list of generated primary keys in case the primary keys for some documents were missing (capped to 100000).
- `warnings`: if the field `generated_keys` is truncated, you will get the warning *"Too many generated keys (<X>), array truncated to 100000."*.
- `old_val`: if `return_vals` is set to `True`, contains `None`.
- `new_val`: if `return_vals` is set to `True`, contains the inserted/updated document.

Example: Insert a document into the table `posts`.

```
r.table("posts").insert({
    "id": 1,
    "title": "Lorem ipsum",
    "content": "Dolor sit amet"
}).run(conn)
```

The result will be:

```
{
    "deleted": 0,
    "errors": 0,
    "inserted": 1,
    "replaced": 0,
    "skipped": 0,
    "unchanged": 0
}
```

Example: Insert a document without a defined primary key into the table `posts` where the primary key is `id`.

```
r.table("posts").insert({
    "title": "Lorem ipsum",
    "content": "Dolor sit amet"
}).run(conn)
```

RethinkDB will generate a primary key and return it in `generated_keys`.

```
{
    "deleted": 0,
    "errors": 0,
    "generated_keys": [
        "dd782b64-70a7-43e4-b65e-dd14ae61d947"
    ],
    "inserted": 1,
    "replaced": 0,
    "skipped": 0,
    "unchanged": 0
}
```

Retrieve the document you just inserted with:

```
r.table("posts").get("dd782b64-70a7-43e4-b65e-dd14ae61d947").run(conn)
```

And you will get back:

```
{
    "id": "dd782b64-70a7-43e4-b65e-dd14ae61d947",
    "title": "Lorem ipsum",
    "content": "Dolor sit amet",
}
```

Example: Insert multiple documents into the table `users`.

```
r.table("users").insert([
    {"id": "william", "email": "william@rethinkdb.com"},
    {"id": "lara", "email": "lara@rethinkdb.com"}
]).run(conn)
```

Example: Insert a document into the table `users`, replacing the document if the document already exists.
Note: If the document exists, the `insert` command will behave like replace, not like update

```
r.table("users").insert(
    {"id": "william", "email": "william@rethinkdb.com"},
    upsert=True
).run(conn)
```

Example: Copy the documents from `posts` to `posts_backup`.

```
r.table("posts_backup").insert( r.table("posts") ).run(conn)
```

Example: Get back a copy of the inserted document (with its generated primary key).

```
r.table("posts").insert(
    {"title": "Lorem ipsum", "content": "Dolor sit amet"},
    return_vals=True
).run(conn)
```

The result will be

```
{
    "deleted": 0,
    "errors": 0,
    "generated_keys": [
        "dd782b64-70a7-43e4-b65e-dd14ae61d947"
    ],
    "inserted": 1,
    "replaced": 0,
    "skipped": 0,
    "unchanged": 0,
    "old_val": None,
    "new_val": {
```

```
        "id": "dd782b64-70a7-43e4-b65e-dd14ae61d947",
        "title": "Lorem ipsum",
        "content": "Dolor sit amet"
    }
}
```

Command syntax

any.type_of() → string

Description

Gets the type of a value.

Example: Get the type of a string.

```
r.expr("foo").type_of().run(conn)
```

Command syntax

value == value → bool

value.eq(value) → bool

Test if two values are equal.

Example: Does 2 equal 2?

```
(r.expr(2) == 2).run(conn)
r.expr(2).eq(2).run(conn)
```

Command syntax

sequence.pluck([selector1, selector2...]) → stream

array.pluck([selector1, selector2...]) → array

object.pluck([selector1, selector2...]) → object

singleSelection.pluck([selector1, selector2...]) → object

Description

Plucks out one or more attributes from either an object or a sequence of objects (projection).

Example: We just need information about IronMan's reactor and not the rest of the document.

```
r.table('marvel').get('IronMan').pluck('reactorState', 'reactorPower').run(conn)
```

Example: For the hero beauty contest we only care about certain qualities.

```
r.table('marvel').pluck('beauty', 'muscleTone', 'charm').run(conn)
```

Example: Pluck can also be used on nested objects.

```
r.table('marvel').pluck({'abilities' : {'damage' : True, 'mana_cost' : True}, 'weapons' : Tr
```

Example: The nested syntax can quickly become overly verbose so there's a shorthand for it.

```
r.table('marvel').pluck({'abilities' : ['damage', 'mana_cost']}, 'weapons').run(conn)
```

Command syntax

array.prepend(value) → array

Description

Prepend a value to an array.

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')['equipment'].prepend('newBoots').run(conn)
```

Command syntax

value.default(default_value) → any

sequence.default(default_value) → any

Description

Handle non-existence errors. Tries to evaluate and return its first argument. If an error related to the absence of a value is thrown in the process, or if its first argument returns `None`, returns its second argument. (Alternatively, the second argument may be a function which will be called with either the text of the non-existence error or `None`.)

Exmple: Suppose we want to retrieve the titles and authors of the table `posts`. In the case where the author field is missing or `None`, we want to retrieve the string `Anonymous`.

```
r.table("posts").map(lambda post:
    {
        "title": post["title"],
        "author": post["author"].default("Anonymous")
    }
).run(conn)
```

We can rewrite the previous query with `r.branch` too.

```
r.table("posts").map(lambda post:
    r.branch(
        post.has_fields("author"),
        {
            "title": post["title"],
            "author": post["author"]
        },
        {
            "title": post["title"],
            "author": "Anonymous"
        }
    )
).run(conn)
```

Example: The `default` command can be useful to filter documents too. Suppose we want to retrieve all our users who are not grown-ups or whose age is unknown (i.e the field `age` is missing or equals `None`). We can do it with this query:

```
r.table("users").filter(lambda user:
    (user["age"] < 18).default(True)
).run(conn)
```

One more way to write the previous query is to set the age to be `-1` when the field is missing.

```
r.table("users").filter(lambda user:
    user["age"].default(-1) < 18
).run(conn)
```

One last way to do the same query is to use `has_fields`.

```
r.table("users").filter(lambda user:
    user.has_fields("age").not_() | (user["age"] < 18)
).run(conn)
```

The body of every `filter` is wrapped in an implicit `.default(False)`. You can overwrite the value `False` by passing an option in filter, so the previous query can also be written like this.

```
r.table("users").filter(
    lambda user: (user["age"] < 18).default(True),
    default=True
).run(conn)
```

Command syntax

r.time(year, month, day[, hour, minute, second], timezone) → time

Description

Create a time object for a specific time.

A few restrictions exist on the arguments:

- `year` is an integer between 1400 and 9,999.
- `month` is an integer between 1 and 12.
- `day` is an integer between 1 and 31.
- `hour` is an integer.
- `minutes` is an integer.
- `seconds` is a double. Its value will be rounded to three decimal places (millisecond-precision).
- `timezone` can be `'Z'` (for UTC) or a string with the format `±[hh]:[mm]`.

Example: Update the birthdate of the user "John" to November 3rd, 1986 UTC.

```
r.table("user").get("John").update({"birthdate": r.time(1986, 11, 3, 'Z')}).run(conn)
```

Command syntax

sequence.with_fields([selector1, selector2...]) → stream

array.with_fields([selector1, selector2...]) → array

Description

Takes a sequence of objects and a list of fields. If any objects in the sequence don't have all of the specified fields, they're dropped from the sequence. The remaining objects have the specified fields plucked out. (This is identical to `has_fields` followed by `pluck` on a sequence.)

Example: Get a list of heroes and their nemeses, excluding any heroes that lack one.

```
r.table('marvel').with_fields('id', 'nemesis')
```

Example: Get a list of heroes and their nemeses, excluding any heroes whose nemesis isn't in an evil organization.

```
r.table('marvel').with_fields('id', {'nemesis' : {'evil_organization' : True}})
```

Example: The nested syntax can quickly become overly verbose so there's a shorthand.

```
r.table('marvel').with_fields('id', {'nemesis' : 'evil_organization'})
```

Command syntax

table.index_wait([, index...]) → array

Description

Wait for the specified indexes on this table to be ready, or for all indexes on this table to be ready if no indexes are specified.

The result is an array where for each index, there will be an object like:

```
{
    index: <index_name>,
    ready: True
}
```

Example: Wait for all indexes on the table `test` to be ready:

```
r.table('test').index_wait().run(conn)
```

Example: Wait for the index `timestamp` to be ready:

```
r.table('test').index_wait('timestamp').run(conn)
```

Command syntax

r.epoch_time(epoch_time) → time

Description

Create a time object based on seconds since epoch. The first argument is a double and will be rounded to three decimal places (millisecond-precision).

Example: Update the birthdate of the user "John" to November 3rd, 1986.

```
r.table("user").get("John").update({"birthdate": r.epoch_time(531360000)}).run(conn)
```

Command syntax

sequence.reduce(reduction_function[, base]) → value

Description

Produce a single value from a sequence through repeated application of a reduction function.

The reduce function gets invoked repeatedly not only for the input values but also for results of previous reduce invocations. The type and format of the object that is passed in to reduce must be the same with the one returned from reduce.

Example: How many enemies have our heroes defeated?

```
r.table('marvel').map(r.row['monstersKilled']).reduce(
    lambda acc, val: acc + val, 0).run(conn)
```

Command syntax

sequence.skip(n) → stream

array.skip(n) → array

Description

Skip a number of elements from the head of the sequence.

Example: Here in conjunction with `order_by` we choose to ignore the most successful heroes.

```
r.table('marvel').order_by('successMetric').skip(10).run(conn)
```

Command syntax

db.table_list() → array

Description

List all table names in a database. The result is a list of strings.

Example: List all tables of the 'test' database.

```
r.db('test').table_list().run(conn)
```

Command syntax

conn.use(db_name)

Description

Change the default database on this connection.

Example: Change the default database so that we don't need to specify the database when referencing a table.

```
conn.use('marvel')
r.table('heroes').run(conn) # refers to r.db('marvel').table('heroes')
```

Command syntax

table.between(lower_key, upper_key[, index='id', left_bound='closed', right_bound='open']) → selection

Description

Get all documents between two keys. Accepts three optional arguments: `index`, `left_bound`, and `right_bound`. If `index` is set to the name of a secondary index, `between` will return all documents where that index's value is in the specified range (it uses the primary key by default). `left_bound` or `right_bound` may be set to `open` or `closed` to indicate whether or not to include that endpoint of the range (by default, `left_bound` is closed and `right_bound` is open).

Example: Find all users with primary key >= 10 and < 20 (a normal half-open interval).

```
r.table('marvel').between(10, 20).run(conn)
```

Example: Find all users with primary key >= 10 and <= 20 (an interval closed on both sides).

```
r.table('marvel').between(10, 20, right_bound='closed').run(conn)
```

Example: Find all users with primary key < 20. (You can use `NULL` to mean "unbounded" for either endpoint.)

```
r.table('marvel').between(None, 20, right_bound='closed').run(conn)
```

Example: Between can be used on secondary indexes too. Just pass an optional index argument giving the secondary index to query.

```
r.table('dc').between('dark_knight', 'man_of_steel', index='code_name').run(conn)
```

Command syntax

number / number → number

Description

Divide two numbers.

Example: It's as easy as 2 / 2 = 1.

```
(r.expr(2) / 2).run(conn)
```

Command syntax

sequence.outer_join(other_sequence, predicate) → stream

array.outer_join(other_sequence, predicate) → array

Description

Computes a left outer join by retaining each row in the left table even if no match was found in the right table.

Example: Construct a sequence of documents containing all cross-universe matchups where a marvel hero would lose, but keep marvel heroes who would never lose a matchup in the sequence.

```
r.table('marvel').outer_join(r.table('dc'),
 lambda marvelRow, dcRow: marvelRow['strength'] < dcRow['strength']).run(conn)
```

Command syntax

time.timezone() → string

Description

Return the timezone of the time object.

Example: Return all the users in the "-07:00" timezone.

```
r.table("users").filter(lambda user:
    user["subscriptionDate"].timezone() == "-07:00"
)
```

Command syntax

r.branch(test, true_branch, false_branch) → any

Description

If the `test` expression returns `False` or `None`, the `false_branch` will be evaluated. Otherwise, the `true_branch` will be evaluated.

The `branch` command is effectively an `if` renamed due to language constraints.

Example: Return heroes and superheroes.

```
r.table('marvel').map(
    r.branch(
        r.row['victories'] > 100,
        r.row['name'] + ' is a superhero',
        r.row['name'] + ' is a hero'
    )
).run(conn)
```

If the documents in the table `marvel` are:

```
[{
    "name": "Iron Man",
    "victories": 214
},
{
    "name": "Jubilee",
    "victories": 9
}]
```

The results will be:

```
[
    "Iron Man is a superhero",
    "Jubilee is a hero"
]
```

Command syntax

any.do(arg [, args]*, expr) → any

Description

Evaluate the expr in the context of one or more value bindings.

The type of the result is the type of the value returned from expr.

Example: The object(s) passed to do() can be bound to name(s). The last argument is the expression to evaluate in the context of the bindings.

```
r.do(r.table('marvel').get('IronMan'),
    lambda ironman: ironman['name']).run(conn)
```

Command syntax

table.sync() → object

Description

`sync` ensures that writes on a given table are written to permanent storage. Queries that specify soft durability (`durability='soft'`) do not give such guarantees, so `sync` can be used to ensure the state of these queries. A call to `sync` does not return until all previous writes to the table are persisted.

If successful, the operation returns an object: `{"synced": 1}`.

Example: After having updated multiple heroes with soft durability, we now want to wait until these changes are persisted.

```
r.table('marvel').sync().run(conn)
```

Command syntax

sequence[start_index[:end_index]] → stream

array[start_index[:end_index]] → array

Description

Trim the sequence to within the bounds provided.

Example: For this fight, we need heroes with a good mix of strength and agility.

```
r.table('marvel').order_by('strength')[5:10].run(conn)
```

Command syntax

table.replace(json | expr[, durability="hard", return_vals=False, non_atomic=False]) → object

selection.replace(json | expr[, durability="hard", return_vals=False, non_atomic=False]) → object

singleSelection.replace(json | expr[, durability="hard", return_vals=False, non_atomic=False]) → object

Description

Replace documents in a table. Accepts a JSON document or a ReQL expression, and replaces the original document with the new one. The new document must have the same primary key as the original document.

The optional arguments are:

- `durability`: possible values are `hard` and `soft`. This option will override the table or query's durability setting (set in run).
  In soft durability mode RethinkDB will acknowledge the write immediately after receiving it, but before the write has been committed to disk.
- `return_vals`: if set to `True` and in case of a single replace, the replaced document will be returned.
- `non_atomic`: set to `True` if you want to perform non-atomic replaces (replaces that require fetching data from another document).

Replace returns an object that contains the following attributes:

- `replaced`: the number of documents that were replaced
- `unchanged`: the number of documents that would have been modified, except that the new value was the same as the old value

- `inserted`: the number of new documents added. You can have new documents inserted if you do a point-replace on a key that isn't in the table or you do a replace on a selection and one of the documents you are replacing has been deleted
- `deleted`: the number of deleted documents when doing a replace with `None`
- `errors`: the number of errors encountered while performing the replace.
- `first_error`: If errors were encountered, contains the text of the first error.
- `skipped`: 0 for a replace operation
- `old_val`: if `return_vals` is set to `True`, contains the old document.
- `new_val`: if `return_vals` is set to `True`, contains the new document.

Example: Replace the document with the primary key `1`.

```
r.table("posts").get(1).replace({
    "id": 1,
    "title": "Lorem ipsum",
    "content": "Aleas jacta est",
    "status": "draft"
}).run(conn)
```

Example: Remove the field `status` from all posts.

```
r.table("posts").replace(lambda post:
    post.without("status")
).run(conn)
```

Example: Remove all the fields that are not `id`, `title` or `content`.

```
r.table("posts").replace(lambda post:
    post.pluck("id", "title", "content")
).run(conn)
```

Example: Replace the document with the primary key `1` using soft durability.

```
r.table("posts").get(1).replace({
    "id": 1,
    "title": "Lorem ipsum",
    "content": "Aleas jacta est",
    "status": "draft"
}, durability="soft").run(conn)
```

Example: Replace the document with the primary key `1` and return the values of the document before and after the replace operation.

```
r.table("posts").get(1).replace({
    "id": 1,
    "title": "Lorem ipsum",
    "content": "Aleas jacta est",
    "status": "published"
}, return_vals=True).run(conn)
```

The result will have two fields `old_val` and `new_val`.

```
{
    "deleted": 0,
    "errors":  0,
    "inserted": 0,
    "new_val": {
        "id":1,
        "title": "Lorem ipsum"
        "content": "Aleas jacta est",
        "status": "published",
    },
    "old_val": {
        "id":1,
        "title": "Lorem ipsum"
        "content": "TODO",
        "status": "draft",
        "author": "William",
    },
    "replaced": 1,
    "skipped": 0,
    "unchanged": 0
}
```

Command syntax

table.order_by([key1…], index=index_name) -> selection<stream>

selection.order_by(key1, [key2…]) -> selection<array>

sequence.order_by(key1, [key2…]) -> array

Description

Sort the sequence by document values of the given key(s). `orderBy` defaults to ascending ordering. To explicitly specify the ordering, wrap the attribute with either `r.asc` or `r.desc`.

Example: Order our heroes by a series of performance metrics.

```
r.table('marvel').order_by('enemies_vanquished', 'damsels_saved').run(conn)
```

Example: Indexes can be used to perform more efficient orderings. Notice that the index ordering always has highest precedence. Thus the following example is equivalent to the one above.

```
r.table('marvel').order_by('damsels_saved', index='enemies_vanquished').run(conn)
```

Example: You can also specify a descending order when using an index.

```
r.table('marvel').order_by(index=r.desc('enemies_vanquished')).run(conn)
```

Example: Let's lead with our best vanquishers by specify descending ordering.

```
r.table('marvel').order_by(
    r.desc('enemies_vanquished'),
    r.asc('damsels_saved')
).run(conn)
```

Example: You can use a function for ordering instead of just selecting an attribute.

```
r.table('marvel').order_by(lambda doc: doc['enemiesVanquished'] + doc['damselsSaved']).run
```

Example: Functions can also be used descendingly.

```
r.table('marvel').order_by(r.desc(lambda doc: doc['enemiesVanquished'] + doc['damselsSaved
```

Command syntax

r.db_create(db_name) → object

Description

Create a database. A RethinkDB database is a collection of tables, similar to relational databases.

If successful, the operation returns an object: `{"created": 1}`. If a database with the same name already exists the operation throws `RqlRuntimeError`.

Note: that you can only use alphanumeric characters and underscores for the database name.

Example: Create a database named 'superheroes'.

```
r.db_create('superheroes').run(conn)
```

Command syntax

sequence.coerce_to(type_name) → array

value.coerce_to(type_name) → string

array.coerce_to(type_name) → object

object.coerce_to(type_name) → array

Description

Converts a value of one type into another.

You can convert: a selection, sequence, or object into an ARRAY, an array of pairs into an OBJECT, and any DATUM into a STRING.

Example: Convert a table to an array.

```
r.table('marvel').coerce_to('array').run(conn)
```

Example: Convert an array of pairs into an object.

```
r.expr([['name', 'Ironman'], ['victories', 2000]]).coerce_to('object').run(conn)
```

Example: Convert a number to a string.

```
r.expr(1).coerce_to('string').run(conn)
```

Command syntax

sequence.contains(value1[, value2...]) → bool

Description

Returns whether or not a sequence contains all the specified values, or if functions are provided instead, returns whether or not a sequence contains values matching all the specified functions.

Example: Has Iron Man ever fought Superman?

```
r.table('marvel').get('ironman')['opponents'].contains('superman').run(conn)
```

Example: Has Iron Man ever defeated Superman in battle?

```
r.table('marvel').get('ironman')['battles'].contains(lambda battle:
   (battle['winner'] == 'ironman') & (battle['loser'] == 'superman')
).run(conn)
```

Command syntax

table.index_drop(index_name) → object

Description

Delete a previously created secondary index of this table.

Example: Drop a secondary index named 'code_name'.

```
r.table('dc').index_drop('code_name').run(conn)
```

Command syntax

conn.repl()

Description

Set the default connection to make REPL use easier. Allows calling `.run()` on queries without specifying a connection.

Connection objects are not thread-safe and REPL connections should not be used in multi-threaded environments.

Example: Set the default connection for the REPL, then call `run()` without specifying the connection.

```
r.connect(db='marvel').repl()
r.table('heroes').run()
```

Command syntax

bool & bool → bool

Description

Compute the logical and of two values.

Example: True and false anded is false?

```
(r.expr(True) & False).run(conn)
```

Command syntax

array.insert_at(index, value) → array

Description

Insert a value in to an array at a given index. Returns the modified array.

Example: Hulk decides to join the avengers.

```
r.expr(["Iron Man", "Spider-Man"]).insert_at(1, "Hulk").run(conn)
```

Command syntax

time.during(start_time, end_time[, left_bound="open/closed", right_bound="open/closed"]) rarr; bool

Description

Return if a time is between two other times (by default, inclusive for the start, exclusive for the end).

Example: Retrieve all the posts that were posted between December 1st, 2013 (inclusive) and December 10th, 2013 (exclusive).

```
r.table("posts").filter(
   r.row['date'].during(r.time(2013, 12, 1, "Z"), r.time(2013, 12, 10, "Z"))
).run(conn)
```

Example: Retrieve all the posts that were posted between December 1st, 2013 (exclusive) and December 10th, 2013 (inclusive).

```
r.table("posts").filter(
   r.row['date'].during(r.time(2013, 12, 1, "Z"), r.time(2013, 12, 10, "Z"), left_bound="op
).run(conn)
```

Command syntax

time.to_iso8601() → number

Description

Convert a time object to its iso 8601 format.

Example: Return the current time in an ISO8601 format.

```
r.now().to_iso8601()
```

Command syntax

table.get_all(key1[, key2...], [, index='id']) → selection

Description

Get all documents where the given value matches the value of the requested index.

Example: Secondary index keys are not guaranteed to be unique so we cannot query via "get" when using a secondary index.

```
r.table('marvel').get_all('man_of_steel', index='code_name').run(conn)
```

Example: Without an index argument, we default to the primary index. While `get` will either return the document or `null` when no document with such a primary key value exists, this will return either a one or zero length stream.

```
r.table('dc').get_all('superman').run(conn)
```

Example: You can get multiple documents in a single call to `get_all`.

```
r.table('dc').get_all('superman', 'ant man').run(conn)
```

Command syntax

time.to_epoch_time() → number

Description

Convert a time object to its epoch time.

Example: Return the current time in an ISO8601 format.

```
r.now().to_epoch_time()
```

Command syntax

time.in_timezone(timezone) → time

Description

Return a new time object with a different timezone. While the time stays the same, the results returned by methods such as hours() will change since they take the timezone into account. The timezone argument has to be of the ISO 8601 format.

Example: Hour of the day in San Francisco (UTC/GMT -8, without daylight saving time).

```
r.now().in_timezone('-08:00').hours().run(conn)
```

Command syntax

value <= value → bool

value.le(value) → bool

Description

Test if the first value is less than or equal to other.

Example: Is 2 less than or equal to 2?

```
(r.expr(2) <= 2).run(conn)
r.expr(2).le(2).run(conn)
```

Command syntax

r.expr(value) → value

Description

Construct a ReQL JSON object from a native object.

Example: Objects wrapped with expr can then be manipulated by ReQL API functions.

```
r.expr({'a':'b'}).merge({'b':[1,2,3]}).run(conn)
```

Command syntax

number * number → number

array * number → array

Description

Multiply two numbers, or make a periodic array.

Example: It's as easy as 2 * 2 = 4.

```
(r.expr(2) * 2).run(conn)
```

Example: Arrays can be multiplied by numbers as well.

```
(r.expr(["This", "is", "the", "song", "that", "never", "ends."]) * 100).run(conn)
```

Command syntax

number + number → number

string + string → string

array + array → array

time + number → time

Description

Sum two numbers, concatenate two strings, or concatenate 2 arrays.

Example: It's as easy as 2 + 2 = 4.

```
(r.expr(2) + 2).run(conn)
```

Example: Strings can be concatenated too.

```
(r.expr("foo") + "bar").run(conn)
```

Example: Arrays can be concatenated too.

```
(r.expr(["foo", "bar"]) + ["buzz"]).run(conn)
```

Example: Create a date one year from now.

```
r.now() + 365*24*60*60
```

Command syntax

table.index_status([, index...]) → array

Description

Get the status of the specified indexes on this table, or the status of all indexes on this table if no indexes are specified.

The result is an array where for each index, there will be an object like this one:

```
{
    "index": <index_name>,
    "ready": True
}
```

or this one:

```
{
    "index": <index_name>,
    "ready": False,
    "blocks_processed": <int>,
    "blocks_total": <int>
}
```

Example: Get the status of all the indexes on `test`:

```
r.table('test').index_status().run(conn)
```

Example: Get the status of the `timestamp` index:

```
r.table('test').index_status('timestamp').run(conn)
```

Command syntax

sequence.filter(predicate[, default=False]) → selection

stream.filter(predicate[, default=False]) → stream

array.filter(predicate[, default=False]) → array

Description

Get all the documents for which the given predicate is true.

`filter` can be called on a sequence, selection, or a field containing an array of elements. The return type is the same as the type on which the function was called on.

The body of every filter is wrapped in an implicit `.default(False)`, which means that if a non-existence errors is thrown (when you try to access a field that does not exist in a document), RethinkDB will just ignore the document. The `default` value can be changed by passing the named argument `default`. Setting this optional argument to `r.error()` will cause any non-existence errors to return a `RqlRuntimeError`.

Example: Get all the users that are 30 years old.

```
r.table('users').filter({"age": 30}).run(conn)
```

A more general way to write the previous query is to use `r.row`.

```
r.table('users').filter(r.row["age"] == 30).run(conn)
```

Here the predicate is `r.row["age"] == 30`.

- `r.row` refers to the current document
- `r.row["age"]` refers to the field `age` of the current document
- `r.row["age"] == 30` returns `True` if the field `age` is 30

An even more general way to write the same query is to use a lambda function. Read the documentation about r.row to know more about the differences between `r.row` and lambda functions in ReQL.

```
r.table('users').filter(lambda user:
    user["age"] == 30
).run(conn)
```

Example: Get all the users that are more than 18 years old.

```
r.table("users").filter(r.row["age"] > 18).run(conn)
```

Example: Get all the users that are less than 18 years old and more than 13 years old.

```
r.table("users").filter((r.row["age"] < 18) & (r.row["age"] > 13)).run(conn, callback)
```

Example: Get all the users that are more than 18 years old or have their parental consent.

```
r.table("users").filter((r.row["age"].lt(18)) | (r.row["hasParentalConsent"])).run(conn)
```

Example: Get all the users that are less than 18 years old or whose age is unknown (field `age` missing).

```
r.table("users").filter(r.row["age"] < 18, default=True).run(conn)
```

Example: Get all the users that are more than 18 years old. Throw an error if a document is missing the field `age`.

```
r.table("users").filter(r.row["age"] > 18, default=r.error()).run(conn)
```

Example: Select all users who have given their phone number (all the documents whose field `phone_number` is defined and not `None`).

```
r.table('users').filter(lambda user:
    user.has_fields('phone_number')
).run(conn)
```

Example: Retrieve all the users who subscribed between January 1st, 2012 (included) and January 1st, 2013 (excluded).

```
r.table("users").filter(lambda user:
   user["subscription_date"].during( r.time(2012, 1, 1, 'Z'), r.time(2013, 1, 1, 'Z') )
).run(conn)
```

Example: Retrieve all the users who have a gmail account (whose field `email` ends with `@gmail.com`).

```
r.table("users").filter(lambda user:
    user["email"].match("@gmail.com$")
).run(conn)
```

Example: Filter based on the presence of a value in an array.

Suppose the table `users` has the following schema

```
{
    "name": <type 'str'>
    "places_visited": [<type 'str'>]
}
```

Retrieve all the users whose field `places_visited` contains `France`.

```
r.table("users").filter(lambda user:
    user["places_visited"].contains("France")
).run(conn)
```

Example: Filter based on nested fields.

Suppose we have a table `users` containing documents with the following schema.

```
{
    "id": <type 'str'>
    "name": {
        "first": <type 'str'>,
        "middle": <type 'str'>,
        "last": <type 'str'>
    }
}
```

Retrieve all users named "William Adama" (first name "William", last name "Adama"), with any middle name.

```
r.table("users").filter({
    "name":{
        "first": "William",
        "last": "Adama"
    }
}).run(conn)
```

If you want an exact match for a field that is an object, you will have to use `r.literal`.

Retrieve all users named "William Adama" (first name "William", last name "Adama"), and who do not have a middle name.

```
r.table("users").filter(r.literal({
    "name":{
        "first": "William",
        "last": "Adama"
    }
})).run(conn)
```

The equivalent queries with a lambda function.

```
r.table("users").filter(lambda user:
    (user["name"]["first"] == "William")
        & (user["name"]["last"] == "Adama")
).run(conn)

r.table("users").filter(lambda user:
    user["name"] == {
        "first": "William",
        "last": "Adama"
    }
).run(conn)
```

Command syntax

table.index_list() → array

Description

List all the secondary indexes of this table.

Example: List the available secondary indexes for this table.

```
r.table('marvel').index_list().run(conn)
```

Command syntax

table.index_create(index_name[, index_function]) → object

Description

Create a new secondary index on this table.

Example: To efficiently query our heros by code name we have to create a secondary index.

```
r.table('dc').index_create('code_name').run(conn)
```

Example: You can also create a secondary index based on an arbitrary function on the document.

```
r.table('dc').index_create('power_rating',
lambda hero: hero['combat_power'] + (2 * hero['compassion_power'])
).run(conn)
```

Example: A compound index can be created by returning an array of values to use as the secondary index key.

```
r.table('dc').index_create('parental_planets',
lambda hero: [hero['mothers_home_planet'], hero['fathers_home_planet']]
).run(conn)
```

Example: A multi index can be created by passing an optional multi argument. Multi indexes functions should return arrays and allow you to query based on whether a value is present in the returned array. The example would allow us to get heroes who possess a specific ability (the field 'abilities' is an array).

```
r.table('dc').index_create('abilities', multi=True).run(conn)
```

Command syntax

singleSelection.keys() → array

object.keys() → array

Description

Return an array containing all of the object's keys.

Example: Get all the keys of a row.

```
r.table('marvel').get('ironman').keys().run(conn)
```

Command syntax

r.row → value

Description

Returns the currently visited document.

Example: Get all users whose age is greater than 5.

```
r.table('users').filter(r.row['age'] > 5).run(conn)
```

Example: Accessing the attribute 'child' of an embedded document.

```
r.table('users').filter(r.row['embedded_doc']['child'] > 5).run(conn)
```

Example: Add 1 to every element of an array.

```
r.expr([1, 2, 3]).map(r.row + 1).run(conn)
```

Example: For nested queries functions should be used instead of r.row.

```
r.table('users').filter(
    lambda doc: doc['name'] == r.table('prizes').get('winner')
).run(conn)
```

Command syntax

time.day_of_week() → number

Description

Return the day of week of a time object as a number between 1 and 7 (following ISO 8601 standard). For your convenience, the terms r.monday, r.tuesday etc. are defined and map to the appropriate integer.

Example: Return today's day of week.

```
r.now().day_of_week().run(conn)
```

Example: Retrieve all the users who were born on a Tuesday.

```
r.table("users").filter{ |user|
    user["birthdate"].day_of_week().eq(r.tuesday)
}
```

Command syntax

array.set_insert(value) → array

Description

Add a value to an array and return it as a set (an array with distinct values).

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')['equipment'].set_insert('newBoots').run(conn)
```

Command syntax

sequence.has_fields([selector1, selector2...]) → stream

array.has_fields([selector1, selector2...]) → array

singleSelection.has_fields([selector1, selector2...]) → boolean

object.has_fields([selector1, selector2...]) → boolean

Description

Test if an object has all of the specified fields. An object has a field if it has the specified key and that key maps to a non-null value. For instance, the object `{'a':1,'b':2,'c': None}` has the fields `a` and `b`.

Example: Which heroes are married?

```
r.table('marvel').has_fields('spouse').run(conn)
```

Example: Test if a single object has a field.

```
r.table('marvel').get("IronMan").has_fields('spouse').run(conn)
```

Example: You can also test if nested fields exist to get only spouses with powers of their own.

```
r.table('marvel').has_fields({'spouse' : {'powers' : True}}).run(conn)
```

Example: The nested syntax can quickly get verbose so there's a shorthand.

```
r.table('marvel').has_fields({'spouse' : 'powers'}).run(conn)
```

Command syntax

value > value → bool

value.gt(value) → bool

Test if the first value is greater than other.

Example: Is 2 greater than 2?

```
(r.expr(2) > 2).run(conn)
r.expr(2).gt(2).run(conn)
```

Command syntax

time.month() → number

Description

Return the month of a time object as a number between 1 and 12. For your convenience, the terms r.january, r.february etc. are defined and map to the appropriate integer.

Example: Retrieve all the users who were born in November.

```
r.table("users").filter(
    r.row["birthdate"].month() == 11
)
```

Example: Retrieve all the users who were born in November.

```
r.table("users").filter(
    r.row["birthdate"].month() == r.november
)
```

Command syntax

sequence.inner_join(other_sequence, predicate) → stream

array.inner_join(other_sequence, predicate) → array

Description

Returns the inner product of two sequences (e.g. a table, a filter result) filtered by the predicate. The query compares each row of the left sequence with each row of the right sequence to find all pairs of rows which satisfy the predicate. When the predicate is satisfied, each matched pair of rows of both sequences are combined into a result row.

Example: Construct a sequence of documents containing all cross-universe matchups where a marvel hero would lose.

```
r.table('marvel').inner_join(r.table('dc'), lambda marvelRow, dcRow:
    marvelRow['strength'] < dcRow['strength']).run(conn)
```

Command syntax

~bool → bool

bool.not_() → bool

Description

Compute the logical inverse (not).

Example: Not true is false.

```
(~r.expr(True)).run(conn)
```

Note the parentheses around the query. If you execute

```
~r.expr(True).run(conn)
```

You will get back `-2` because the query executed is `r.expr(True)` which returns `True`, and because `~True` evaluates to `-2` in Python.

Example: The previous query can be rewritten with `not_`

```
r.expr(True).not_().run(conn)
```

Command syntax

sequence.without([selector1, selector2...]) → stream

array.without([selector1, selector2...]) → array

singleSelection.without([selector1, selector2...]) → object

object.without([selector1, selector2...]) → object

Description

The opposite of pluck; takes an object or a sequence of objects, and returns them with the specified paths removed.

Example: Since we don't need it for this computation we'll save bandwidth and leave out the list of IronMan's romantic conquests.

```
r.table('marvel').get('IronMan').without('personalVictoriesList').run(conn)
```

Example: Without their prized weapons, our enemies will quickly be vanquished.

```
r.table('enemies').without('weapons').run(conn)
```

Example: Nested objects can be used to remove the damage subfield from the weapons and abilities fields.

```
r.table('marvel').without({'weapons' : {'damage' : True}, 'abilities' : {'damage' : True}})
```

Example: The nested syntax can quickly become overly verbose so there's a shorthand for it.

```
r.table('marvel').without({'weapons' : 'damage', 'abilities' : 'damage'}).run(conn)
```

Command syntax

r.sum(attr)

Description

Compute the sum of the given field in the group.

Example: How many enemies have been vanquished by heroes at each strength level?

```
r.table('marvel').group_by('strength', r.sum('enemiesVanquished')).run(conn)
```

Command syntax

time.year() → number

Description

Return the year of a time object.

Example: Retrieve all the users born in 1986.

```
r.table("users").filter{ |user|
    user["birthdate"].year().eq(1986)
}.run(conn)
```

Command syntax

sequence[attr] → sequence

singleSelection[attr] → value

object[attr] → value

Description

Get a single field from an object. If called on a sequence, gets that field from every object in the sequence, skipping objects that lack it.

Example: What was Iron Man's first appearance in a comic?

```
r.table('marvel').get('IronMan')[:first_appearance].run(conn)
```

Command syntax

sequence.union(sequence) → array

Description

Concatenate two sequences.

Example: Construct a stream of all heroes.

```
r.table('marvel').union(r.table('dc')).run(conn)
```

Command syntax

time.day() → number

Description

Return the day of a time object as a number between 1 and 31.

Example: Return the users born on the 24th of any month.

```
r.table("users").filter{ |user|
    user["birthdate"].day().eq(24)
}
```

Command syntax

r.db(db_name) → db

Description

Reference a database.

Example: Before we can query a table we have to select the correct database.

```
r.db('heroes').table('marvel').run(conn)
```

Command syntax

array.delete_at(index [,endIndex]) → array

Description

Remove an element from an array at a given index. Returns the modified array.

Example: Hulk decides to leave the avengers.

```
r.expr(["Iron Man", "Hulk", "Spider-Man"]).delete_at(1).run(conn)
```

Example: Hulk and Thor decide to leave the avengers.

```
r.expr(["Iron Man", "Hulk", "Thor", "Spider-Man"]).delete_at(1,3).run(conn)
```

Command syntax

sequence.distinct() → array

Description

Remove duplicate elements from the sequence.

Example: Which unique villains have been vanquished by marvel heroes?

```
r.table('marvel').concat_map{|hero| hero[:villain_list]}.distinct.run(conn)
```

Command syntax

array.difference(array) → array

Description

Remove the elements of one array from another array.

Example: Retrieve Iron Man's equipment list without boots.

```
r.table('marvel').get('IronMan')[:equipment].difference(['Boots']).run(conn)
```

Command syntax

r.db_drop(db_name) → object

Description

Drop a database. The database, all its tables, and corresponding data will be deleted.

If successful, the operation returns the object `{"dropped": 1}`. If the specified database doesn't exist a `RqlRuntimeError` is thrown.

Example: Drop a database named 'superheroes'.

```
r.db_drop('superheroes').run(conn)
```

Command syntax

db.table(name[, opts]) → table

Description

Select all documents in a table. This command can be chained with other commands to do further processing on the data.

Example: Return all documents in the table 'marvel' of the default database.

```
r.table('marvel').run(conn)
```

Example: Return all documents in the table 'marvel' of the database 'heroes'.

```
r.db('heroes').table('marvel').run(conn)
```

Example: If you are OK with potentially out of date data from this table and want potentially faster reads, pass a flag allowing out of date data.

```
r.db('heroes').table('marvel', {:use_outdated => true}).run(conn)
```

Command syntax

time.minutes() → number

Description

Return the minute in a time object as a number between 0 and 59.

Example: Return all the posts submitted during the first 10 minutes of every hour.

```
r.table("posts").filter{ |post|
    post["date"].minutes() < 10
}
```

Command syntax

sequence.map(mapping_function) → stream

array.map(mapping_function) → array

Description

Transform each element of the sequence by applying the given mapping function.

Example: Construct a sequence of hero power ratings.

```
r.table('marvel').map {|hero|
    hero[:combat_power] + hero[:compassion_power] * 2
}.run(conn)
```

Command syntax

db.table_create(table_name[, options]) → object

Description

Create a table. A RethinkDB table is a collection of JSON documents.

If successful, the operation returns an object: `{created: 1}`. If a table with the same name already exists, the operation throws `RqlRuntimeError`.

Note: that you can only use alphanumeric characters and underscores for the table name.

When creating a table you can specify the following options:

- `primary_key`: the name of the primary key. The default primary key is id;
- `durability`: if set to `soft`, this enables *soft durability* on this table: writes will be acknowledged by the server immediately and flushed to disk in the background. Default is `hard` (acknowledgement of writes happens after data has been written to disk);
- `cache_size`: set the cache size (in bytes) to be used by the table. The default is 1073741824 (1024MB);
- `datacenter`: the name of the datacenter this table should be assigned to.

Example: Create a table named 'dc_universe' with the default settings.

```
r.db('test').table_create('dc_universe').run(conn)
```

Command syntax

sequence.for_each(write_query) → object

Description

Loop over a sequence, evaluating the given write query for each element.

Example: Now that our heroes have defeated their villains, we can safely remove them from the villain table.

```
r.table('marvel').for_each {|hero|
    r.table('villains').get(hero[:villain_defeated]).delete
}.run(conn)
```

Command syntax

table.delete[({:durability => “hard”, :return_vals => false})] → object

selection.delete[({:durability => “hard”, :return_vals => false})] → object

singleSelection.delete[({:durability => “hard”, :return_vals => false})] → object

Description

Delete one or more documents from a table.

The optional arguments are:

- `durability`: possible values are `hard` and `soft`. This option will override the table or query's durability setting (set in run).
  In soft durability mode RethinkDB will acknowledge the write immediately after receiving it, but before the write has been committed to disk.
- `return_vals`: if set to `true` and in case of a single document deletion, the deleted document will be returned.

Delete returns an object that contains the following attributes:

- `deleted`: the number of documents that were deleted.
- `skipped`: the number of documents that were skipped.
  For example, if you attempt to delete a batch of documents, and another concurrent query deletes some of those documents first, they will be counted as skipped.
- `errors`: the number of errors encountered while performing the delete.
- `first_error`: If errors were encountered, contains the text of the first error.
- `inserted`, `replaced`, and `unchanged`: all 0 for a delete operation..
- `old_val`: if `return_vals` is set to `true`, contains the deleted document.
- `new_val`: if `return_vals` is set to `true`, contains `nil`.

Example: Delete a single document from the table `comments`.

```
r.table("comments").get("7eab9e63-73f1-4f33-8ce4-95cbea626f59").delete.run(conn)
```

Example: Delete all documents from the table `comments`.

```
r.table("comments").delete.run(conn)
```

Example: Delete all comments where the field `id_post` is `3`.

```
r.table("comments").filter({:id_post => 3}).delete.run(conn)
```

Example: Delete a single document from the table `comments` and return its value.

```
r.table("comments").get("7eab9e63-73f1-4f33-8ce4-95cbea626f59").delete(:return_vals => tr
```

The result look like:

```
{
    :deleted => 1,
    :errors => 0,
    :inserted => 0,
    :new_val => nil,
    :old_val => {
        :id => "7eab9e63-73f1-4f33-8ce4-95cbea626f59",
        :author => "William",
        :comment => "Great post",
        :id_post => 3
    },
    :replaced => 0,
    :skipped => 0,
    :unchanged => 0
}
```

Example: Delete all documents from the table `comments` without waiting for the operation to be flushed to disk.

```
r.table("comments").delete(:durability => "soft").run(conn)
```

Command syntax

table.get(key) → singleRowSelection

Description

Get a document by primary key.

Example: Find a document with the primary key 'superman'.

```
r.table('marvel').get('superman').run(conn)
```

Command syntax

time.time_of_day() → number

Description

Return the number of seconds elapsed since the beginning of the day stored in the time object.

Example: Retrieve posts that were submitted before noon.

```
r.table("posts").filter{ |post|
    post["date"].time_of_day() <= 12*60*60
}.run(conn)
```

Command syntax

sequence.indexes_of(datum | predicate) → array

Description

Get the indexes of an element in a sequence. If the argument is a predicate, get the indexes of all elements matching it.

Example: Find the position of the letter 'c'.

```
r.expr(['a','b','c']).indexes_of('c').run(conn)
```

Example: Find the popularity ranking of invisible heroes.

```
r.table('marvel').union(r.table('dc')).order_by(:popularity).indexes_of{
    |row| row[:superpowers].contains('invisibility')
}.run(conn)
```

Command syntax

string.match(regexp) → array

Description

Match against a regular expression. Returns a match object containing the matched string, that string's start/end position, and the capture groups. Accepts RE2 syntax (https://code.google.com/p/re2/wiki/Syntax). You can enable case-insensitive matching by prefixing the regular expression with `(?i)`. (See linked RE2 documentation for more flags.)

Example: Get all users whose name starts with A.

```
r.table('users').filter{|row| row[:name].match("^A")}.run(conn)
```

Example: Parse out a name (returns "mlucy").

```
r.expr('id:0,name:mlucy,foo:bar').match('name:(\w+)')[:groups][0][:str].run(conn)
```

Example: Fail to parse out a name (returns null).

```
r.expr('id:0,foo:bar').match('name:(\w+)')[:groups][0][:str].run(conn)
```

Command syntax

sequence.count([filter]) → number

Description

Count the number of elements in the sequence. With a single argument, count the number of elements equal to it. If the argument is a function, it is equivalent to calling filter before count.

Example: Just how many super heroes are there?

```
(r.table('marvel').count() + r.table('dc').count()).run(conn)
```

Example: Just how many super heroes have invisibility?

```
r.table('marvel').concat_map{ |row| row[:superpowers] }.count('invisibility').run(conn)
```

Example: Just how many super heroes have defeated the Sphinx?

```
r.table('marvel').count{ |row| row['monstersKilled'].contains('Sphinx') }.run(conn)
```

Command syntax

array.set_intersection(array) → array

Description

Intersect two arrays returning values that occur in both of them as a set (an array with distinct values).

Example: Check which pieces of equipment Iron Man has from a fixed list.

```
r.table('marvel').get('IronMan')[:equipment].set_intersection(['newBoots', 'arc_reactor']
```

Command syntax

sequence.eq_join(left_attr, other_table[, index='id']) → stream

array.eq_join(left_attr, other_table[, index='id']) → array

Description

An efficient join that looks up elements in the right table by primary key.

Example: Let our heroes join forces to battle evil!

```
r.table('marvel').eq_join(:main_dc_collaborator, r.table('dc')).run(conn)
```

Example: The above query is equivalent to this inner join but runs in O(n log(m)) time rather than the O(n * m) time the inner join takes.

```
r.table('marvel').inner_join(r.table('dc')) {|left, right|
    left[:main_dc_collaborator].eq(right[:hero_name])
}.run(conn)
```

Example: You can take advantage of a secondary index on the second table by giving an optional index parameter.

```
r.table('marvel').eq_join('main_weapon_origin',
r.table('mythical_weapons'), :index => 'origin').run(conn)
```

Example: You can pass a function instead of an attribute to join on more complicated expressions. Here we join to the DC universe collaborator with whom the hero has the most appearances.

```
r.table('marvel').eq_join(lambda {|doc| doc[:dc_collaborators].order_by(:appearances)[0][
r.table('dc')).run(conn)
```

Command syntax

sequence.sample(number) → selection

stream.sample(number) → array

array.sample(number) → array

Description

Select a given number of elements from a sequence with uniform random distribution. Selection is done without replacement.

Example: Select 3 random heroes.

```
r.table('marvel').sample(3).run(conn)
```

Command syntax

r → r

Description

The top-level ReQL namespace.

Example: Setup your top-level namespace.

```
require 'rethinkdb'
include RethinkDB::Shortcuts
```

Command syntax

singleSelection.merge(object) → object

object.merge(object) → object

sequence.merge(object) → stream

array.merge(object) → array

Description

Merge two objects together to construct a new object with properties from both. Gives preference to attributes from other when there is a conflict.

Example: Equip IronMan for battle.

```
r.table('marvel').get('IronMan').merge(
    r.table('loadouts').get('alienInvasionKit')
).run(conn)
```

Example: Merge can be used recursively to modify object within objects.

```
r.expr({:weapons => {:spectacular_graviton_beam => {:dmg => 10, :cooldown => 20}}}).merge(
    {:weapons => {:spectacular_graviton_beam => {:dmg => 10}}}
).run(conn)
```

Example: To replace a nested object with another object you can use the literal keyword.

```
r.expr({:weapons => {:spectacular_graviton_beam => {:dmg => 10, :cooldown => 20}}}).merge(
   {:weapons => r.literal({:repulsor_rays => {:dmg => 3, :cooldown => 0}})}
).run(conn)
```

Example: Literal can be used to remove keys from an object as well.

```
r.expr({:weapons => {:spectacular_graviton_beam => {:dmg => 10, :cooldown => 20}}}).merge(
    {:weapons => {:spectacular_graviton_beam => r.literal()}}
).run(conn)
```

Command syntax

r.iso8601(iso8601Date[, {default_timezone:”}]) → time

Description

Create a time object based on an iso8601 date-time string (e.g. ‘2013-01-01T01:01:01+00:00’). We support all valid ISO 8601 formats except for week dates. If you pass an ISO 8601 date-time without a time zone, you must specify the time zone with the optarg `default_timezone`. Read more about the ISO 8601 format on the Wikipedia page.

Example: Update the time of John’s birth.

```
r.table("user").get("John").update(:birth => r.iso8601('1986-11-03T08:30:00-07:00')).run(
```

Command syntax

sequence.group_by(selector1[, selector2...], reduction_object) → array

Description

Groups elements by the values of the given attributes and then applies the given reduction. Though similar to `groupedMapReduce`, `groupBy` takes a standardized object for specifying the reduction. Can be used with a number of predefined common reductions.

Example: Using a predefined reduction we can easily find the average strength of members of each weight class.

```
r.table('marvel').group_by(:weight_class, r.avg(:strength)).run(conn)
```

Example: Groupings can also be specified on nested attributes.

```
r.table('marvel').group_by({:abilities => {:primary => true}}, r.avg(:strength)).run(conn)
```

Example: The nested syntax can quickly become verbose so there's a shortcut.

```
r.table('marvel').group_by({:abilities => :primary}, r.avg(:strength)).run(conn)
```

Command syntax

conn.close(opts={})

Description

Close an open connection. Accepts the following options:

- `noreply_wait`: whether to wait for noreply writes to complete before closing (default `true`). If this is set to `false`, some outstanding noreply writes may be aborted.

Closing a connection waits until all outstanding requests have finished and then frees any open resources associated with the connection. If `noreply_wait` is set to `false`, all outstanding requests are canceled immediately.

Example: Close an open connection, waiting for noreply writes to finish.

```
conn.close
```

Example: Close an open connection immediately.

```
conn.close(:noreply_wait => false)
```

Command syntax

r.json(json_string) → value

Description

Parse a JSON string on the server.

Example: Send an array to the server'

```
r.json("[1,2,3]").run(conn)
```

Command syntax

value < value → bool

value.lt(value) → bool

Description

Test if the first value is less than other.

Example: Is 2 less than 2?

```
(r.expr(2) < 2).run(conn)
r.expr(2).lt(2).run(conn)
```

Command syntax

r.count

Description

Count the total size of the group.

Example: Just how many heroes do we have at each strength level?

```
r.table('marvel').group_by(:strength, r.count).run(conn)
```

Command syntax

time.date() → time

Description

Return a new time object only based on the day, month and year (ie. the same day at 00:00).

Example: Retrieve all the users whose birthday is today

```
r.table("users").filter{ |user|
    user["birthdate"].date().eq(r.now().date())
}.run(conn)
```

Command syntax

bool | bool → bool

Description

Compute the logical or of two values.

Example: True or false ored is true?

```
(r.expr(True) | False).run(conn)
```

Command syntax

r.now() → time

Description

Return a time object representing the current time in UTC. The command now() is computed once when the server receives the query, so multiple instances of r.now() will always return the same time inside a query.

Example: Add a new user with the time at which he subscribed.

```
r.table("users").insert({
    :name => "John",
    :subscription_date => r.now()
}).run(conn)
```

Command syntax

array.splice_at(index, array) → array

Description

Insert several values in to an array at a given index. Returns the modified array.

Example: Hulk and Thor decide to join the avengers.

```
r.expr(["Iron Man", "Spider-Man"]).splice_at(1, ["Hulk", "Thor"]).run(conn)
```

Command syntax

time.hours() → number

Description

Return the hour in a time object as a number between 0 and 23.

Example: Return all the posts submitted after midnight and before 4am.

```
r.table("posts").filter{ |post|
    post["date"].hours() < 4
}
```

Command syntax

array.change_at(index, value) → array

Description

Change a value in an array at a given index. Returns the modified array.

Example: Bruce Banner hulks out.

```
r.expr(["Iron Man", "Bruce", "Spider-Man"]).change_at(1, "Hulk").run(conn)
```

Command syntax

value >= value → bool

value.ge(value) → bool

Description

Test if the first value is greater than or equal to other.

Example: Is 2 greater than or equal to 2?

```
(r.expr(2) >= 2).run(conn)
r.expr(2).ge(2).run(conn)
```

Command syntax

array.append(value) → array

Description

Append a value to an array.

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')[:equipment].append('new_boots').run(conn)
```

Command syntax

conn.reconnect(opts={})

Description

Close and reopen a connection. Accepts the following options:

- `noreply_wait`: whether to wait for noreply writes to complete before closing (default `true`). If this is set to `false`, some outstanding noreply writes may be aborted.

Closing a connection waits until all outstanding requests have finished. If `noreply_wait` is set to `false`, all outstanding requests are canceled immediately.

Example: Cancel outstanding requests/queries that are no longer needed.

```
conn.reconnect(:noreply_wait => false)
```

Command syntax

array.set_difference(array) → array

Description

Remove the elements of one array from another and return them as a set (an array with distinct values).

Example: Check which pieces of equipment Iron Man has, excluding a fixed list.

```
r.table('marvel').get('IronMan')[:equipment].set_difference(['newBoots', 'arc_reactor']).
```

Command syntax

any.info() → object

Description

Get information about a ReQL value.

Example: Get information about a table such as primary key, or cache size.

```
r.table('marvel').info().run(conn)
```

Command syntax

array.set_union(array) → array

Description

Add a several values to an array and return it as a set (an array with distinct values).

Example: Retrieve Iron Man's equipment list with the addition of some new boots and an arc reactor.

```
r.table('marvel').get('IronMan')[:equipment].set_union(['newBoots', 'arc_reactor']).run(c
```

Command syntax

r.connect(opts={}) → connection

Description

Create a new connection to the database server. Accepts the following options:

- `host`: the host to connect to (default `localhost`).
- `port`: the port to connect on (default `28015`).
- `db`: the default database (default `test`).
- `auth_key`: the authentication key (default none).

If the connection cannot be established, a `RqlDriverError` exception will be thrown.

Example: Opens a new connection to the database.

```
conn = r.connect(:host => 'localhost',
                 :port => 28015,
                 :db => 'heroes',
                 :auth_key => 'hunter2')
```

Command syntax

r.error(message) → error

Description

Throw a runtime error. If called with no arguments inside the second argument to `default`, re-throw the current error.

Example: Iron Man can't possibly have lost a battle:

```
r.table('marvel').get('IronMan').do { |ironman|
    r.branch(ironman[:victories] < ironman[:battles],
    r.error('impossible code path'),
    ironman)
}.run(conn)
```

Command syntax

sequence.grouped_map_reduce(grouping, mapping, reduction, base) → value

Description

Partition the sequence into groups based on the `grouping` function. The elements of each group are then mapped using the `mapping` function and reduced using the `reduction` function.

`grouped_map_reduce` is a generalized form of group by.

Example: It's only fair that heroes be compared against their weight class.

```
r.table('marvel').grouped_map_reduce(
    lambda {|hero| hero[:weight_class]},  # grouping
    lambda {|hero| hero.pluck(:name, :strength)},  #  mapping
    {:name => 'none', :strength => 0},  # reduction base
   lambda {|acc, hero| r.branch(acc[:strength] < hero[:strength], hero, acc)}
).run(conn)
```

Command syntax

sequence.is_empty() → bool

Description

Test if a sequence is empty.

Example: Are there any documents in the marvel table?

```
r.table('marvel').is_empty().run(conn)
```

Command syntax

sequence.limit(n) → stream

array.limit(n) → array

Description

End the sequence after the given number of elements.

Example: Only so many can fit in our Pantheon of heroes.

```
r.table('marvel').order_by(:belovedness).limit(10).run(conn)
```

Command syntax

db.table_drop(table_name) → object

Description

Drop a table. The table and all its data will be deleted.

If succesful, the operation returns an object: {“dropped”: 1}. If the specified table doesn’t exist a `RqlRuntimeError` is thrown.

Example: Drop a table named ‘dc_universe’.

```
r.db('test').table_drop('dc_universe').run(conn)
```

Command syntax

value.ne(value) → bool

Description

Test if two values are not equal.

Example: Does 2 not equal 2?

```
r.expr(2).ne(2).run(conn)
```

Command syntax

sequence[index] → object

Description

Get the nth element of a sequence.

Example: Select the second element in the array.

```
r.expr([1,2,3])[1].run(conn)
```

Command syntax

r.js(jsString) → value

Description

Create a javascript expression.

Example: Concatenate two strings using Javascript'

```
r.js("'str1' + 'str2'").run(conn)
```

Example: Select all documents where the 'magazines' field is greater than 5 by running Javascript on the server.

```
r.table('marvel').filter(
r.js('(function (row) { return row.magazines > 5; })')).run(conn)
```

Example: You may also specify a timeout in seconds (defaults to 5).

```
r.js('while(true) {}', :timeout => 1.3).run(conn)
```

Command syntax

table.update(json | expr[, :durability => “hard”, :return_vals => false, :non_atomic => false]) → object

selection.update(json | expr[, :durability => “hard”, :return_vals => false, :non_atomic => false]) → object

singleSelection.update(json | expr[, :durability => “hard”, :return_vals => false, :non_atomic => false]) → object

Description

Update JSON documents in a table. Accepts a JSON document, a ReQL expression, or a combination of the two.

The optional arguments are:

- `durability`: possible values are `hard` and `soft`. This option will override the table or query's durability setting (set in run).
  In soft durability mode RethinkDB will acknowledge the write immediately after receiving it, but before the write has been committed to disk.
- `return_vals`: if set to `true` and in case of a single update, the updated document will be returned.
- `non_atomic`: set to `true` if you want to perform non-atomic updates (updates that require fetching data from another document).

Update returns an object that contains the following attributes:

- `replaced`: the number of documents that were updated.
- `unchanged`: the number of documents that would have been modified except the new value was the same as the old value.
- `skipped`: the number of documents that were skipped because the document didn't exist.
- `errors`: the number of errors encountered while performing the update.

- `first_error`: If errors were encountered, contains the text of the first error.
- `deleted` and `inserted`: 0 for an update operation.
- `old_val`: if `return_vals` is set to `true`, contains the old document.
- `new_val`: if `return_vals` is set to `true`, contains the new document.

Example: Update the status of the post with `id` of `1` to `published`.

```
r.table("posts").get(1).update({:status => "published"}).run(conn)
```

Example: Update the status of all posts to `published`.

```
r.table("posts").update({:status => "published"}).run(conn)
```

Example: Update the status of all the post written by William.

```
r.table("posts").filter({:author => "William"}).update({:status => "published"}).run(conn)
```

Example: Increment the field `view` with `id` of `1`. This query will throw an error if the field `views` doesn't exist.

```
r.table("posts").get(1).update{ |post|
    {:views => post["views"]+1}
}.run(conn)
```

Example: Increment the field `view` of the post with `id` of `1`. If the field `views` does not exist, it will be set to `0`.

```
r.table("posts").update{ |post|
    {:views => (post["views"]+1).default(0)}
}.run(conn)
```

Example: Perform a conditional update.
If the post has more than 100 views, set the `type` of a post to `hot`, else set it to `normal`.

```
r.table("posts").get(1).update{ |post|
    r.branch(
        post["views"] > 100,
        {:type => "hot"},
        {:type => "normal"}
    )
}.run(conn)
```

Example: Update the field `num_comments` with the result of a sub-query. Because this update is not atomic, you must pass the `non_atomic` flag.

```
r.table("posts").get(1).update({
   :num_comments => r.table("comments").filter({:id_post => 1}).count()
}, :non_atomic => true ).run(conn)
```

If you forget to specify the `non_atomic` flag, you will get a `RqlRuntimeError`.

```
RqlRuntimeError: Could not prove function deterministic.  Maybe you want to use the non_atomi
```

Example: Update the field `num_comments` with a random value between 0 and 100.
This update cannot be proven deterministic because of `r.js` (and in fact is not), so you must pass the `non_atomic` flag.

```
r.table("posts").get(1).update({
    :num_comments => r.js("Math.floor(Math.random()*100)")
}, :non_atomic => true).run(conn)
```

Example: Update the status of the post with `id` of `1` using soft durability.

```
r.table("posts").get(1).update({:status => "published"}, :durability => "soft").run(conn)
```

Example: Increment the field `views` and return the values of the document before and after the update operation.

```
r.table("posts").get(1).update(:return_vals => true) { |post|
    :views => post["views"]+1
}.run(conn)
```

The result will have two fields `old_val` and `new_val`.

```
{
    :deleted => 1,
    :errors => 0,
    :inserted => 0,
    :new_val => {
        :id => 1,
        :author => "Julius_Caesar",
        :title => "Commentarii de Bello Gallico",
        :content => "Aleas jacta est",
        :views => 207
```

```
    },
    :old_val => {
        :id => 1,
        :author => "Julius_Caesar",
        :title => "Commentarii de Bello Gallico",
        :content => "Aleas jacta est",
        :views => 206
    },
    :replaced => 0,
    :skipped => 0,
    :unchanged => 0
}
```

Accessing ReQL

All ReQL queries begin from the top-level module.

r

r → r

The top-level ReQL namespace.

Example: Set up your top-level namespace.

```
require 'rethinkdb'
include RethinkDB::Shortcuts
```

connect

r.connect(opts={}) → connection

Create a new connection to the database server. Accepts the following options:

- `host`: the host to connect to (default `localhost`).
- `port`: the port to connect on (default `28015`).
- `db`: the default database (default `test`).
- `auth_key`: the authentication key (default none).

If the connection cannot be established, a `RqlDriverError` exception will be thrown.

Example: Opens a new connection to the database.

```
conn = r.connect(:host => 'localhost',
                 :port => 28015,
                 :db => 'heroes',
                 :auth_key => 'hunter2')
```

Read more about this command →

repl

conn.repl

Set the default connection to make REPL use easier. Allows calling `.run` on queries without specifying a connection.

Connection objects are not thread-safe and REPL connections should not be used in multi-threaded environments.

Example: Set the default connection for the REPL, then call `run` without specifying the connection.

```
r.connect(:db => 'marvel').repl
r.table('heroes').run
```

close

conn.close(opts={})

Close an open connection. Accepts the following options:

- `noreply_wait`: whether to wait for noreply writes to complete before closing (default `true`). If this is set to `false`, some outstanding noreply writes may be aborted.

Closing a connection waits until all outstanding requests have finished and then frees any open resources associated with the connection. If `noreply_wait` is set to `false`, all outstanding requests are canceled immediately.

Example: Close an open connection, waiting for noreply writes to finish.

```
conn.close
```

Example: Close an open connection immediately.

```
conn.close(:noreply_wait => false)
```

reconnect

conn.reconnect(opts={})

Close and reopen a connection. Accepts the following options:

- `noreply_wait`: whether to wait for noreply writes to complete before closing (default `true`). If this is set to `false`, some outstanding noreply writes may be aborted.

Closing a connection waits until all outstanding requests have finished. If `noreply_wait` is set to `false`, all outstanding requests are canceled immediately.

Example: Cancel outstanding requests/queries that are no longer needed.

```
conn.reconnect(:noreply_wait => false)
```

use

conn.use(db_name)

Change the default database on this connection.

Example: Change the default database so that we don't need to specify the database when referencing a table.

```
conn.use('marvel')
r.table('heroes').run(conn) # refers to r.db('marvel').table('heroes')
```

run

query.run(conn[, opts]) → cursor

query.run(conn[, opts]) → object

Run a query on a connection. Accepts the following options:

- `use_outdated`: whether or not outdated reads are OK (default: `false`).
- `time_format`: what format to return times in (default: `'native'`). Set this to `'raw'` if you want times returned as JSON objects for exporting.
- `profile`: whether or not to return a profile of the query's execution (default: `false`).

Returns either a single JSON result or a cursor, depending on the query.

Example: Run a query on the connection `conn` and print out every row in the result.

```
r.table('marvel').run(conn).each{|x| p x}
```

Read more about this command →

noreply_wait

conn.noreply_wait

`noreply_wait` ensures that previous queries with the `noreply` flag have been processed by the server. Note that this guarantee only applies to queries run on the given connection.

Example: We have previously run queries with the `noreply` argument set to `true`. Now wait until the server has processed them.

```
conn.noreply_wait
```

close (cursor)

cursor.close

Close a cursor. Closing a cursor cancels the corresponding query and frees the memory associated with the open request.

Example: Close a cursor.

```
cursor.close
```

Manipulating databases

db_create

r.db_create(db_name) → object

Create a database. A RethinkDB database is a collection of tables, similar to relational databases.

If successful, the operation returns an object: `{"created": 1}`. If a database with the same name already exists the operation throws `RqlRuntimeError`.

Note: that you can only use alphanumeric characters and underscores for the database name.

Example: Create a database named 'superheroes'.

```
r.db_create('superheroes').run(conn)
```

db_drop

r.db_drop(db_name) → object

Drop a database. The database, all its tables, and corresponding data will be deleted.

If successful, the operation returns the object `{"dropped": 1}`. If the specified database doesn't exist a `RqlRuntimeError` is thrown.

Example: Drop a database named 'superheroes'.

```
r.db_drop('superheroes').run(conn)
```

db_list

r.db_list() → array

List all database names in the system. The result is a list of strings.

Example: List all databases.

```
r.db_list.run(conn)
```

Manipulating tables

table_create

db.table_create(table_name[, options]) → object

Create a table. A RethinkDB table is a collection of JSON documents.

If successful, the operation returns an object: `{created: 1}`. If a table with the same name already exists, the operation throws `RqlRuntimeError`.

Note: that you can only use alphanumeric characters and underscores for the table name.

When creating a table you can specify the following options:

- `primary_key`: the name of the primary key. The default primary key is id;
- `durability`: if set to `soft`, this enables *soft durability* on this table: writes will be acknowledged by the server immediately and flushed to disk in the background. Default is `hard` (acknowledgement of writes happens after data has been written to disk);
- `cache_size`: set the cache size (in bytes) to be used by the table. The default is 1073741824 (1024MB);
- `datacenter`: the name of the datacenter this table should be assigned to.

Example: Create a table named 'dc_universe' with the default settings.

```
r.db('test').table_create('dc_universe').run(conn)
```

Read more about this command →

table_drop

db.table_drop(table_name) → object

Drop a table. The table and all its data will be deleted.

If succesful, the operation returns an object: {"dropped": 1}. If the specified table doesn't exist a `RqlRuntimeError` is thrown.

Example: Drop a table named 'dc_universe'.

```
r.db('test').table_drop('dc_universe').run(conn)
```

table_list

db.table_list() → array

List all table names in a database. The result is a list of strings.

Example: List all tables of the 'test' database.

```
r.db('test').table_list().run(conn)
```

index_create

table.index_create(index_name[, index_function]) → object

Create a new secondary index on this table.

Example: To efficiently query our heros by code name we have to create a secondary index.

```
r.table('dc').index_create('code_name').run(conn)
```

Read more about this command →

index_drop

table.index_drop(index_name) → object

Delete a previously created secondary index of this table.

Example: Drop a secondary index named 'code_name'.

```
r.table('dc').index_drop('code_name').run(conn)
```

index_list

table.index_list() → array

List all the secondary indexes of this table.

Example: List the available secondary indexes for this table.

```
r.table('marvel').index_list().run(conn)
```

index_status

table.index_status([, index...]) → array

Get the status of the specified indexes on this table, or the status of all indexes on this table if no indexes are specified.

Example: Get the status of all the indexes on `test`:

```
r.table('test').index_status.run(conn)
```

Example: Get the status of the `timestamp` index:

```
r.table('test').index_status('timestamp').run(conn)
```

index_wait

table.index_wait([, index...]) → array

Wait for the specified indexes on this table to be ready, or for all indexes on this table to be ready if no indexes are specified.

Example: Wait for all indexes on the table `test` to be ready:

```
r.table('test').index_wait.run(conn)
```

Example: Wait for the index `timestamp` to be ready:

```
r.table('test').index_wait('timestamp').run(conn)
```

Writing data

insert

table.insert(json | [json] [, :durability => “hard”, :return_vals => false :upsert => false]) → object

Insert documents into a table. Accepts a single document or an array of documents.

Example: Insert a document into the table `posts`.

```
r.table("posts").insert({
    :id => 1,
    :title => "Lorem ipsum",
    :content => "Dolor sit amet"
}).run(conn)
```

Read more about this command →

update

table.update(json | expr [, :durability => “hard”, :return_vals => false, :non_atomic => false]) → object

selection.update(json | expr [, :durability => “hard”, :return_vals => false, :non_atomic => false]) → object

singleSelection.update(json | expr [, :durability => “hard”, :return_vals => false, :non_atomic => false]) → object

Update JSON documents in a table. Accepts a JSON document, a ReQL expression, or a combination of the two.

Example: Update the status of the post with `id` of `1` to `published`.

```
r.table("posts").get(1).update({status: "published"}).run(conn)
```

Read more about this command →

replace

table.replace(json | expr [, :durability => “hard”, :return_vals => false, :non_atomic => false]) → object

selection.replace(json | expr [, :durability => “hard”, :return_vals => false, :non_atomic => false]) → object

singleSelection.replace(json | expr [, :durability => “hard”, :return_vals => false, :non_atomic => false]) → object

Replace documents in a table. Accepts a JSON document or a ReQL expression, and replaces the original document with the new one. The new document must have the same primary key as the original document.

Example: Replace the document with the primary key `1`.

```
r.table("posts").get(1).replace({
    :id => 1,
    :title => "Lorem ipsum",
    :content => "Aleas jacta est",
    :status => "draft"
}).run(conn)
```

Read more about this command →

delete

table.delete[({:durability => “hard”, :return_vals => false})] → object

selection.delete[({:durability => “hard”, :return_vals => false})] → object

singleSelection.delete[({:durability => “hard”, :return_vals => false})] → object

Delete one or more documents from a table.

Example: Delete a single document from the table `comments`.

```
r.table("comments").get("7eab9e63-73f1-4f33-8ce4-95cbea626f59").delete.run(conn)
```

Read more about this command →

sync

table.sync → object

`sync` ensures that writes on a given table are written to permanent storage. Queries that specify soft durability (`{:durability => soft}`) do not give such guarantees, so `sync` can be used to ensure the state of these queries. A call to `sync` does not return until all previous writes to the table are persisted.

Example: After having updated multiple heroes with soft durability, we now want to wait until these changes are persisted.

```
r.table('marvel').sync.run(conn)
```

Selecting data

db

r.db(db_name) → db

Reference a database.

Example: Before we can query a table we have to select the correct database.

```
r.db('heroes').table('marvel').run(conn)
```

table

db.table(name[, opts]) → table

Select all documents in a table. This command can be chained with other commands to do further processing on the data.

Example: Return all documents in the table 'marvel' of the default database.

```
r.table('marvel').run(conn)
```

Read more about this command →

get

table.get(key) → singleRowSelection

Get a document by primary key.

Example: Find a document with the primary key 'superman'.

```
r.table('marvel').get('superman').run(conn)
```

get_all

table.get_all(key[, key2...], [, :index => 'id']) → selection

Get all documents where the given value matches the value of the requested index.

Example: Secondary index keys are not guaranteed to be unique so we cannot query via "get" when using a secondary index.

```
r.table('marvel').get_all('man_of_steel', :index => 'code_name').run(conn)
```

Read more about this command →

between

table.between(lower_key, upper_key [, :index => 'id', :left_bound => 'closed', :right_bound => 'open']) → selection

Get all documents between two keys. Accepts three optional arguments: `index`, `left_bound`, and `right_bound`. If `index` is set to the name of a secondary index, `between` will return all documents where that index's value is in the specified range (it uses the primary key by default). `left_bound` or `right_bound` may be set to `open` or `closed` to indicate whether or not to include that endpoint of the range (by default, `left_bound` is closed and `right_bound` is open).

Example: Find all users with primary key >= 10 and < 20 (a normal half-open interval).

```
r.table('marvel').between(10, 20).run(conn)
```

Read more about this command →

filter

sequence.filter(predicate[, :default => false]) → selection

stream.filter(predicate[, :default => false]) → stream

array.filter(predicate[, :default => false]) → array

Get all the documents for which the given predicate is true.

`filter` can be called on a sequence, selection, or a field containing an array of elements. The return type is the same as the type on which the function was called on.

The body of every filter is wrapped in an implicit `.default(false)`, which means that if a non-existence errors is thrown (when you try to access a field that does not exist in a document), RethinkDB will just ignore the document. The `default` value can be changed by passing the symbol `default`. Setting this optional argument to `r.error()` will cause any non-existence errors to return a `RqlRuntimeError`.

Example: Get all the users that are 30 years old.

```
r.table('users').filter({:age => 30}).run(conn)
```

Read more about this command →

Joins

These commands allow the combination of multiple sequences into a single sequence

inner_join

sequence.inner_join(other_sequence, predicate) → stream

array.inner_join(other_sequence, predicate) → array

Returns the inner product of two sequences (e.g. a table, a filter result) filtered by the predicate. The query compares each row of the left sequence with each row of the right sequence to find all pairs of rows which satisfy the predicate. When the predicate is satisfied, each matched pair of rows of both sequences are combined into a result row.

Example: Construct a sequence of documents containing all cross-universe matchups where a marvel hero would lose.

```
r.table('marvel').inner_join(r.table('dc')) {|marvel_row, dc_row|
    marvel_row[:strength] < dc_row[:strength]
}.run(conn)
```

outer_join

sequence.outer_join(other_sequence, predicate) → stream

array.outer_join(other_sequence, predicate) → array

Computes a left outer join by retaining each row in the left table even if no match was found in the right table.

Example: Construct a sequence of documents containing all cross-universe matchups where a marvel hero would lose, but keep marvel heroes who would never lose a matchup in the sequence.

```
r.table('marvel').outer_join(r.table('dc')) {|marvel_row, dc_row|
    marvel_row[:strength] < dc_row[:strength]
}.run(conn)
```

eq_join

sequence.eq_join(left_attr, other_table[, index='id']) → stream

array.eq_join(left_attr, other_table[, index='id']) → array

An efficient join that looks up elements in the right table by primary key.

Example: Let our heroes join forces to battle evil!

```
r.table('marvel').eq_join(:main_dc_collaborator, r.table('dc')).run(conn)
```

Read more about this command →

zip

stream.zip() → stream

array.zip() → array

Used to 'zip' up the result of a join by merging the 'right' fields into 'left' fields of each member of the sequence.

Example: 'zips up' the sequence by merging the left and right fields produced by a join.

```
r.table('marvel').eq_join(:main_dc_collaborator, r.table('dc')).zip.run(conn)
```

Transformations

These commands are used to transform data in a sequence.

map

sequence.map(mapping_function) → stream

array.map(mapping_function) → array

Transform each element of the sequence by applying the given mapping function.

Example: Construct a sequence of hero power ratings.

```
r.table('marvel').map {|hero|
    hero[:combat_power] + hero[:compassion_power] * 2
}.run(conn)
```

with_fields

sequence.with_fields([selector1, selector2...]) → stream

array.with_fields([selector1, selector2...]) → array

Takes a sequence of objects and a list of fields. If any objects in the sequence don't have all of the specified fields, they're dropped from the sequence. The remaining objects have the specified fields plucked out. (This is identical to `has_fields` followed by `pluck` on a sequence.)

Example: Get a list of heroes and their nemeses, excluding any heroes that lack one.

```
r.table('marvel').with_fields('id', 'nemesis')
```

Read more about this command →

concat_map

sequence.concat_map(mapping_function) → stream

array.concat_map(mapping_function) → array

Flattens a sequence of arrays returned by the mappingFunction into a single sequence.

Example: Construct a sequence of all monsters defeated by Marvel heroes. Here the field 'defeatedMonsters' is a list that is concatenated to the sequence.

```
r.table('marvel').concat_map {|hero|
    hero[:defeated_monsters]
}.run(conn)
```

order_by

table.order_by([key1…], :index => index_name) -> selection<stream>

selection.order_by(key1, [key2…]) -> selection<array>

sequence.order_by(key1, [key2…]) -> array

Sort the sequence by document values of the given key(s). `orderBy` defaults to ascending ordering. To explicitly specify the ordering, wrap the attribute with either `r.asc` or `r.desc`.

Example: Order our heroes by a series of performance metrics.

```
r.table('marvel').order_by(:enemies_vanquished, :damsels_saved).run(conn)
```

Read more about this command →

skip

sequence.skip(n) → stream

array.skip(n) → array

Skip a number of elements from the head of the sequence.

Example: Here in conjunction with `order_by` we choose to ignore the most successful heroes.

```
r.table('marvel').order_by(:success_metric).skip(10).run(conn)
```

limit

sequence.limit(n) → stream

array.limit(n) → array

End the sequence after the given number of elements.

Example: Only so many can fit in our Pantheon of heroes.

```
r.table('marvel').order_by(:belovedness).limit(10).run(conn)
```

[]

sequence[start_index[..end_index]] → stream

array[start_index[..end_index]] → array

Trim the sequence to within the bounds provided.

Example: For this fight, we need heroes with a good mix of strength and agility.

```
r.table('marvel').order_by(:strength)[5..10].run(conn)
```

[]

sequence[index] → object

Get the nth element of a sequence.

Example: Select the second element in the array.

```
r.expr([1,2,3])[1].run(conn)
```

indexes_of

sequence.indexes_of(datum | predicate) → array

Get the indexes of an element in a sequence. If the argument is a predicate, get the indexes of all elements matching it.

Example: Find the position of the letter 'c'.

```
r.expr(['a','b','c']).indexes_of('c').run(conn)
```

Read more about this command →

is_empty

sequence.is_empty() → bool

Test if a sequence is empty.

Example: Are there any documents in the marvel table?

```
r.table('marvel').is_empty().run(conn)
```

union

sequence.union(sequence) → array

Concatenate two sequences.

Example: Construct a stream of all heroes.

```
r.table('marvel').union(r.table('dc')).run(conn)
```

sample

sequence.sample(number) → selection

stream.sample(number) → array

array.sample(number) → array

Select a given number of elements from a sequence with uniform random distribution. Selection is done without replacement.

Example: Select 3 random heroes.

```
r.table('marvel').sample(3).run(conn)
```

Aggregation

These commands are used to compute smaller values from large sequences.

reduce

sequence.reduce(reduction_function[, base]) → value

Produce a single value from a sequence through repeated application of a reduction function.

The reduce function gets invoked repeatedly not only for the input values but also for results of previous reduce invocations. The type and format of the object that is passed in to reduce must be the same with the one returned from reduce.

Example: How many enemies have our heroes defeated?

```
r.table('marvel').order_by(:strength)[5..10].run(conn)
```

count

sequence.count([filter]) → number

Count the number of elements in the sequence. With a single argument, count the number of elements equal to it. If the argument is a function, it is equivalent to calling filter before count.

Example: Just how many super heroes are there?

```
(r.table('marvel').count() + r.table('dc').count()).run(conn)
```

Read more about this command →

distinct

sequence.distinct() → array

Remove duplicate elements from the sequence.

Example: Which unique villains have been vanquished by marvel heroes?

```
r.table('marvel').concat_map{|hero| hero[:villain_list]}.distinct.run(conn)
```

grouped_map_reduce

sequence.grouped_map_reduce(grouping, mapping, reduction, base) → value

Partition the sequence into groups based on the `grouping` function. The elements of each group are then mapped using the `mapping` function and reduced using the `reduction` function.

`grouped_map_reduce` is a generalized form of group by.

Example: It's only fair that heroes be compared against their weight class.

```
r.table('marvel').grouped_map_reduce(
    lambda {|hero| hero[:weight_class]},  # grouping
    lambda {|hero| hero.pluck(:name, :strength)},  #  mapping
    {:name => 'none', :strength => 0},  # reduction base
   lambda {|acc, hero| r.branch(acc[:strength] < hero[:strength], hero, acc)}
).run(conn)
```

group_by

sequence.group_by(selector1[, selector2...], reduction_object) → array

Groups elements by the values of the given attributes and then applies the given reduction. Though similar to `groupedMapReduce`, `groupBy` takes a standardized object for specifying the reduction. Can be used with a number of predefined common reductions.

Example: Using a predefined reduction we can easily find the average strength of members of each weight class.

```
r.table('marvel').group_by(:weight_class, r.avg(:strength)).run(conn)
```

Read more about this command →

contains

sequence.contains(value1[, value2...]) → bool

Returns whether or not a sequence contains all the specified values, or if functions are provided instead, returns whether or not a sequence contains values matching all the specified functions.

Example: Has Iron Man ever fought Superman?

```
r.table('marvel').get('ironman')[:opponents].contains('superman').run(conn)
```

Read more about this command →

Aggregators

These standard aggregator objects are to be used in conjunction with groupBy.

[count](count-aggregator/)

r.count

Count the total size of the group.

Example: Just how many heroes do we have at each strength level?

```
r.table('marvel').group_by(:strength, r.count).run(conn)
```

sum

r.sum(attr)

Compute the sum of the given field in the group.

Example: How many enemies have been vanquished by heroes at each strength level?

```
r.table('marvel').group_by(:strength, r.sum(:enemies_vanquished)).run(conn)
```

avg

r.avg(attr)

Compute the average value of the given attribute for the group.

Example: What's the average agility of heroes at each strength level?

```
r.table('marvel').group_by(:strength, r.avg(:agility)).run(conn)
```

Document manipulation

pluck

sequence.pluck([selector1, selector2...]) → stream

array.pluck([selector1, selector2...]) → array

object.pluck([selector1, selector2...]) → object

singleSelection.pluck([selector1, selector2...]) → object

Plucks out one or more attributes from either an object or a sequence of objects (projection).

Example: We just need information about IronMan's reactor and not the rest of the document.

```
r.table('marvel').get('IronMan').pluck('reactorState', 'reactorPower').run(conn)
```

Read more about this command →

without

sequence.without([selector1, selector2...]) → stream

array.without([selector1, selector2...]) → array

singleSelection.without([selector1, selector2...]) → object

object.without([selector1, selector2...]) → object

The opposite of pluck; takes an object or a sequence of objects, and returns them with the specified paths removed.

Example: Since we don't need it for this computation we'll save bandwidth and leave out the list of IronMan's romantic conquests.

```
r.table('marvel').get('IronMan').without('personalVictoriesList').run(conn)
```

Read more about this command →

merge

singleSelection.merge(object) → object

object.merge(object) → object

sequence.merge(object) → stream

array.merge(object) → array

Merge two objects together to construct a new object with properties from both. Gives preference to attributes from other when there is a conflict.

Example: Equip IronMan for battle.

```
r.table('marvel').get('IronMan').merge(
    r.table('loadouts').get('alienInvasionKit')).run(conn)
```

Read more about this command →

append

array.append(value) → array

Append a value to an array.

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')[:equipment].append('new_boots').run(conn)
```

prepend

array.prepend(value) → array

Prepend a value to an array.

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')[:equipment].prepend('new_boots').run(conn)
```

difference

array.difference(array) → array

Remove the elements of one array from another array.

Example: Retrieve Iron Man's equipment list without boots.

```
r.table('marvel').get('IronMan')[:equipment].difference(['Boots']).run(conn)
```

set_insert

array.set_insert(value) → array

Add a value to an array and return it as a set (an array with distinct values).

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')[:equipment].set_insert('new_boots').run(conn)
```

set_union

array.set_union(array) → array

Add a several values to an array and return it as a set (an array with distinct values).

Example: Retrieve Iron Man's equipment list with the addition of some new boots and an arc reactor.

```
r.table('marvel').get('IronMan')[:equipment].set_union(['newBoots', 'arc_reactor']).run(c
```

set_intersection

array.set_intersection(array) → array

Intersect two arrays returning values that occur in both of them as a set (an array with distinct values).

Example: Check which pieces of equipment Iron Man has from a fixed list.

```
r.table('marvel').get('IronMan')[:equipment].set_intersection(['newBoots', 'arc_reactor']
```

set_difference

array.set_difference(array) → array

Remove the elements of one array from another and return them as a set (an array with distinct values).

Example: Check which pieces of equipment Iron Man has, excluding a fixed list.

```
r.table('marvel').get('IronMan')[:equipment].set_difference(['newBoots', 'arc_reactor']).
```

[]

sequence[attr] → sequence

singleSelection[attr] → value

object[attr] → value

Get a single field from an object. If called on a sequence, gets that field from every object in the sequence, skipping objects that lack it.

Example: What was Iron Man's first appearance in a comic?

```
r.table('marvel').get('IronMan')[:first_appearance].run(conn)
```

has_fields

sequence.has_fields([selector1, selector2...]) → stream

array.has_fields([selector1, selector2...]) → array

singleSelection.has_fields([selector1, selector2...]) → boolean

object.has_fields([selector1, selector2...]) → boolean

Test if an object has all of the specified fields. An object has a field if it has the specified key and that key maps to a non-null value. For instance, the object `{:a => 1, :b => 2, :c => nil}` has the fields `a` and `b`.

Example: Which heroes are married?

```
r.table('marvel').has_fields(:spouse).run(conn)
```

Read more about this command →

insert_at

array.insert_at(index, value) → array

Insert a value in to an array at a given index. Returns the modified array.

Example: Hulk decides to join the avengers.

```
r.expr(["Iron Man", "Spider-Man"]).insert_at(1, "Hulk").run(conn)
```

splice_at

array.splice_at(index, array) → array

Insert several values in to an array at a given index. Returns the modified array.

Example: Hulk and Thor decide to join the avengers.

```
r.expr(["Iron Man", "Spider-Man"]).splice_at(1, ["Hulk", "Thor"]).run(conn)
```

delete_at

array.delete_at(index [,endIndex]) → array

Remove an element from an array at a given index. Returns the modified array.

Example: Hulk decides to leave the avengers.

```
r.expr(["Iron Man", "Hulk", "Spider-Man"]).delete_at(1).run(conn)
```

Read more about this command →

change_at

array.change_at(index, value) → array

Change a value in an array at a given index. Returns the modified array.

Example: Bruce Banner hulks out.

```
r.expr(["Iron Man", "Bruce", "Spider-Man"]).change_at(1, "Hulk").run(conn)
```

keys

singleSelection.keys() → array

object.keys() → array

Return an array containing all of the object's keys.

Example: Get all the keys of a row.

```
r.table('marvel').get('ironman').keys.run(conn)
```

String manipulation

These commands provide string operators.

match

string.match(regexp) → array

Match against a regular expression. Returns a match object containing the matched string, that string's start/end position, and the capture groups. Accepts RE2 syntax (https://code.google.com/p/re2/wiki/Syntax). You can enable case-insensitive matching by prefixing the regular expression with `(?i)`. (See linked RE2 documentation for more flags.)

Example: Get all users whose name starts with A.

```
r.table('users').filter{|row| row[:name].match("^A")}.run(conn)
```

Read more about this command →

Math and logic

+

number + number → number

string + string → string

array + array → array

time + number → time

Sum two numbers, concatenate two strings, or concatenate 2 arrays.

Example: It's as easy as 2 + 2 = 4.

```
(r.expr(2) + 2).run(conn)
```

Read more about this command →

-

number - number → number

time - time → number

time - number → time

Subtract two numbers.

Example: It's as easy as 2 - 2 = 0.

```
(r.expr(2) - 2).run(conn)
```

Read more about this command →

*

number * number → number

array * number → array

Multiply two numbers, or make a periodic array.

Example: It's as easy as 2 * 2 = 4.

```
(r.expr(2) * 2).run(conn)
```

Read more about this command →

/

number / number → number

Divide two numbers.

Example: It's as easy as 2 / 2 = 1.

```
(r.expr(2) / 2).run(conn)
```

%

number % number → number

Find the remainder when dividing two numbers.

Example: It's as easy as 2 % 2 = 0.

```
(r.expr(2) % 2).run(conn)
```

&

bool & bool → bool

Compute the logical and of two values.

Example: True and false anded is false?

```
(r.expr(True) & False).run(conn)
```

|

bool | bool → bool

Compute the logical or of two values.

Example: True or false ored is true?

```
(r.expr(True) | False).run(conn)
```

eq

value.eq(value) → bool

Test if two values are equal.

Example: Does 2 equal 2?

```
r.expr(2).eq(2).run(conn)
```

ne

value.ne(value) → bool

Test if two values are not equal.

Example: Does 2 not equal 2?

```
r.expr(2).ne(2).run(conn)
```

>, gt

value > value → bool

value.gt(value) → bool

Test if the first value is greater than other.

Example: Is 2 greater than 2?

```
(r.expr(2) > 2).run(conn)
r.expr(2).gt(2).run(conn)
```

>=, ge

value >= value → bool

value.ge(value) → bool

Test if the first value is greater than or equal to other.

Example: Is 2 greater than or equal to 2?

```
(r.expr(2) >= 2).run(conn)
r.expr(2).ge(2).run(conn)
```

<, lt

value < value → bool

value.lt(value) → bool

Test if the first value is less than other.

Example: Is 2 less than 2?

```
(r.expr(2) < 2).run(conn)
r.expr(2).lt(2).run(conn)
```

<=, le

value <= value → bool

value.le(value) → bool

Test if the first value is less than or equal to other.

Example: Is 2 less than or equal to 2?

```
(r.expr(2) <= 2).run(conn)
r.expr(2).le(2).run(conn)
```

not

bool.not() → bool

Compute the logical inverse (not).

Example: Not true is false.

```
r(true).not.run(conn)
```

Dates and times

now

r.now() → time

Return a time object representing the current time in UTC. The command now() is computed once when the server receives the query, so multiple instances of r.now() will always return the same time inside a query.

Example: Add a new user with the time at which he subscribed.

```
r.table("users").insert({
    :name => "John",
    :subscription_date => r.now()
}).run(conn)
```

time

r.time(year, month, day[, hour, minute, second], timezone) → time

Create a time object for a specific time.

A few restrictions exist on the arguments:

- `year` is an integer between 1400 and 9,999.
- `month` is an integer between 1 and 12.
- `day` is an integer between 1 and 31.
- `hour` is an integer.
- `minutes` is an integer.
- `seconds` is a double. Its value will be rounded to three decimal places (millisecond-precision).
- `timezone` can be `'Z'` (for UTC) or a string with the format `±[hh]:[mm]`.

Example: Update the birthdate of the user "John" to November 3rd, 1986 UTC.

```
r.table("user").get("John").update(:birthdate => r.time(1986, 11, 3, 'Z')).run(conn)
```

epoch_time

r.epoch_time(epoch_time) → time

Create a time object based on seconds since epoch. The first argument is a double and will be rounded to three decimal places (millisecond-precision).

Example: Update the birthdate of the user "John" to November 3rd, 1986.

```
r.table("user").get("John").update(:birthdate => r.epoch_time(531360000)).run(conn)
```

iso8601

r.iso8601(iso8601Date[, {default_timezone:"}]) → time

Create a time object based on an iso8601 date-time string (e.g. '2013-01-01T01:01:01+00:00'). We support all valid ISO 8601 formats except for week dates. If you pass an ISO 8601 date-time without a time zone, you must specify the time zone with the optarg `default_timezone`. Read more about the ISO 8601 format on the Wikipedia page.

Example: Update the time of John's birth.

```
r.table("user").get("John").update(:birth => r.iso8601('1986-11-03T08:30:00-07:00')).run(
```

in_timezone

time.in_timezone(timezone) → time

Return a new time object with a different timezone. While the time stays the same, the results returned by methods such as hours() will change since they

take the timezone into account. The timezone argument has to be of the ISO 8601 format.

Example: Hour of the day in San Francisco (UTC/GMT -8, without daylight saving time).

```
r.now().in_timezone('-08:00').hours().run(conn)
```

timezone

time.timezone() → string

Return the timezone of the time object.

Example: Return all the users in the "-07:00" timezone.

```
r.table("users").filter{ |user|
    user["subscriptionDate"].timezone().eq("07:00")
}
```

during

time.during(start_time, end_time [, left_bound="open/closed", right_bound="open/closed"]) → bool

Return if a time is between two other times (by default, inclusive for the start, exclusive for the end).

Example: Retrieve all the posts that were posted between December 1st, 2013 (inclusive) and December 10th, 2013 (exclusive).

```
r.table("posts").filter{ |post|
    post['date'].during(r.time(2013, 12, 1), r.time(2013, 12, 10))
}.run(conn)
```

Read more about this command →

date

time.date() → time

Return a new time object only based on the day, month and year (ie. the same day at 00:00).

Example: Retrieve all the users whose birthday is today

```
r.table("users").filter{ |user|
    user["birthdate"].date().eq(r.now().date())
}.run(conn)
```

time_of_day

time.time_of_day() → number

Return the number of seconds elapsed since the beginning of the day stored in the time object.

Example: Retrieve posts that were submitted before noon.

```
r.table("posts").filter{ |post|
    post["date"].time_of_day() <= 12*60*60
}.run(conn)
```

year

time.year() → number

Return the year of a time object.

Example: Retrieve all the users born in 1986.

```
r.table("users").filter{ |user|
    user["birthdate"].year().eq(1986)
}.run(conn)
```

month

time.month() → number

Return the month of a time object as a number between 1 and 12. For your convenience, the terms r.january, r.february etc. are defined and map to the appropriate integer.

Example: Retrieve all the users who were born in November.

```
r.table("users").filter{ |user|
    user["birthdate"].month().eq(11)
}
```

Read more about this command →

day

time.day() → number

Return the day of a time object as a number between 1 and 31.

Example: Return the users born on the 24th of any month.

```
r.table("users").filter{ |user|
    user["birthdate"].day().eq(24)
}
```

day_of_week

time.day_of_week() → number

Return the day of week of a time object as a number between 1 and 7 (following ISO 8601 standard). For your convenience, the terms r.monday, r.tuesday etc. are defined and map to the appropriate integer.

Example: Return today's day of week.

```
r.now().day_of_week().run(conn)
```

Read more about this command →

day_of_year

time.day_of_year() → number

Return the day of the year of a time object as a number between 1 and 366 (following ISO 8601 standard).

Example: Retrieve all the users who were born the first day of a year.

```
r.table("users").filter{ |user|
    user["birthdate"].day_of_year().eq(1)
}
```

hours

time.hours() → number

Return the hour in a time object as a number between 0 and 23.

Example: Return all the posts submitted after midnight and before 4am.

```
r.table("posts").filter{ |post|
    post["date"].hours() < 4
}
```

minutes

time.minutes() → number

Return the minute in a time object as a number between 0 and 59.

Example: Return all the posts submitted during the first 10 minutes of every hour.

```
r.table("posts").filter{ |post|
    post["date"].minutes() < 10
}
```

seconds

time.seconds() → number

Return the seconds in a time object as a number between 0 and 59.999 (double precision).

Example: Return the post submitted during the first 30 seconds of every minute.

```
r.table("posts").filter{ |post|
    post["date"].seconds() < 30
}
```

to_iso8601

time.to_iso8601() → number

Convert a time object to its iso 8601 format.

Example: Return the current time in an ISO8601 format.

```
r.now().to_iso8601()
```

to_epoch_time

time.to_epoch_time() → number

Convert a time object to its epoch time.

Example: Return the current time in an ISO8601 format.

```
r.now().to_epoch_time()
```

Control structures

do

any.do(arg [, args]*, expr) → any

Evaluate the expr in the context of one or more value bindings.

The type of the result is the type of the value returned from expr.

Example: The object(s) passed to do() can be bound to name(s). The last argument is the expression to evaluate in the context of the bindings.

```
r.do(r.table('marvel').get('IronMan')) { |ironman| ironman[:name] }.run(conn)
```

branch

r.branch(test, true_branch, false_branch) → any

If the `test` expression returns `false` or `nil`, the `false_branch` will be evaluated. Otherwise, the `true_branch` will be evaluated.

The `branch` command is effectively an `if` renamed due to language constraints. The type of the result is determined by the type of the branch that gets executed.

Example: Return heroes and superheroes.

```
r.table('marvel').map{ |hero|
    r.branch(
        hero['victories'] > 100,
        hero['name'].add(' is a superhero'),
        hero['name'].add(' is a hero')
    )
}.run(conn)
```

for_each

sequence.for_each(write_query) → object

Loop over a sequence, evaluating the given write query for each element.

Example: Now that our heroes have defeated their villains, we can safely remove them from the villain table.

```
r.table('marvel').for_each {|hero|
    r.table('villains').get(hero[:villain_defeated]).delete
}.run(conn)
```

error

r.error(message) → error

Throw a runtime error. If called with no arguments inside the second argument to `default`, re-throw the current error.

Example: Iron Man can't possibly have lost a battle:

```
r.table('marvel').get('IronMan').do { |ironman|
    r.branch(ironman[:victories] < ironman[:battles],
    r.error('impossible code path'),
    ironman)
}.run(conn)
```

default

value.default(default_value) → any

sequence.default(default_value) → any

Handle non-existence errors. Tries to evaluate and return its first argument. If an error related to the absence of a value is thrown in the process, or if its first argument returns `nil`, returns its second argument. (Alternatively, the second argument may be a function which will be called with either the text of the non-existence error or `nil`.)

Example: Suppose we want to retrieve the titles and authors of the table `posts`. In the case where the author field is missing or `nil`, we want to retrieve the string `Anonymous`.

```
r.table("posts").map{ |post|
    {
        :title => post["title"],
```

```
        :author => post["author"].default("Anonymous")
    }
}.run(conn)
```

Read more about this command →

expr

r.expr(value) → value

Construct a ReQL JSON object from a native object.

Example: Objects wrapped with `expr` can then be manipulated by ReQL API functions.

```
r.expr({:a => 'b'}).merge({:b => [1,2,3]}).run(conn)
```

Read more about this command →

js

r.js(jsString) → value

Create a javascript expression.

Example: Concatenate two strings using Javascript'

```
r.js("'str1' + 'str2'").run(conn)
```

Read more about this command →

coerce_to

sequence.coerce_to(type_name) → array

value.coerce_to(type_name) → string

array.coerce_to(type_name) → object

object.coerce_to(type_name) → array

Converts a value of one type into another.

You can convert: a selection, sequence, or object into an ARRAY, an array of pairs into an OBJECT, and any DATUM into a STRING.

Example: Convert a table to an array.

```
r.table('marvel').coerce_to('array').run(conn)
```

Read more about this command →

type_of

any.type_of() → string

Gets the type of a value.

Example: Get the type of a string.

```
r.expr("foo").type_of().run(conn)
```

info

any.info() → object

Get information about a ReQL value.

Example: Get information about a table such as primary key, or cache size.

```
r.table('marvel').info().run(conn)
```

json

r.json(json_string) → value

Parse a JSON string on the server.

Example: Send an array to the server'

```
r.json("[1,2,3]").run(conn)
```

Command syntax

time.seconds() → number

Description

Return the seconds in a time object as a number between 0 and 59.999 (double precision).

Example: Return the post submitted during the first 30 seconds of every minute.

```
r.table("posts").filter{ |post|
    post["date"].seconds() < 30
}
```

Command syntax

r.avg(attr)

Description

Compute the average value of the given attribute for the group.

Example: What's the average agility of heroes at each strength level?

```
r.table('marvel').group_by(:strength, r.avg(:agility)).run(conn)
```

Command syntax

cursor.close

Description

Close a cursor. Closing a cursor cancels the corresponding query and frees the memory associated with the open request.

Example: Close a cursor.

```
cursor.close
```

Command syntax

query.run(conn[, opts]) → cursor

query.run(conn[, opts]) → object

Description

Run a query on a connection. Accepts the following options:

- `use_outdated`: whether or not outdated reads are OK (default: `false`).
- `time_format`: what format to return times in (default: `'native'`). Set this to `'raw'` if you want times returned as JSON objects for exporting.

- `profile`: whether or not to return a profile of the query's execution (default: `false`).

Returns either a single JSON result or a cursor, depending on the query.

Example: Run a query on the connection `conn` and print out every row in the result.

```
r.table('marvel').run(conn).each{|x| p x}
```

Example: If you are OK with potentially out of date data from all the tables involved in this query and want potentially faster reads, pass a flag allowing out of date data in an options object. Settings for individual tables will supercede this global setting for all tables in the query.

```
r.table('marvel').run(conn, :use_outdated => true)
```

Example: If you just want to send a write and forget about it, you can set `noreply` to true in the options. In this case `run` will return immediately.

```
r.table('marvel').run(conn, :noreply => true)
```

Example: If you want to specify whether to wait for a write to be written to disk (overriding the table's default settings), you can set `durability` to `'hard'` or `'soft'` in the options.

```
r.table('marvel')
   .insert({ :superhero => 'Iron Man', :superpower => 'Arc Reactor' })
    .run(conn, :noreply => true, :durability => 'soft')
```

Example: If you do not want a time object to be converted to a native date object, you can pass a `time_format` flag to prevent it (valid flags are "raw" and "native"). This query returns an object with two fields (`epoch_time` and `$reql_type$`) instead of a native date object.

```
r.now().run(conn, :time_format=>"raw")
```

Command syntax

sequence.concat_map(mapping_function) → stream

array.concat_map(mapping_function) → array

Description

Flattens a sequence of arrays returned by the mappingFunction into a single sequence.

Example: Construct a sequence of all monsters defeated by Marvel heroes. Here the field 'defeatedMonsters' is a list that is concatenated to the sequence.

```
r.table('marvel').concat_map {|hero|
    hero[:defeated_monsters]
}.run(conn)
```

Command syntax

r.db_list() → array

Description

List all database names in the system. The result is a list of strings.

Example: List all databases.

```
r.db_list.run(conn)
```

Command syntax

number % number → number

Description

Find the remainder when dividing two numbers.

Example: It's as easy as 2 % 2 = 0.

```
(r.expr(2) % 2).run(conn)
```

Command syntax

conn.noreply_wait

Description

`noreply_wait` ensures that previous queries with the `noreply` flag have been processed by the server. Note that this guarantee only applies to queries run on the given connection.

Example: We have previously run queries with the `noreply` argument set to `true`. Now wait until the server has processed them.

```
conn.noreply_wait
```

Command syntax

time.day_of_year() → number

Description

Return the day of the year of a time object as a number between 1 and 366 (following ISO 8601 standard).

Example: Retrieve all the users who were born the first day of a year.

```
r.table("users").filter{ |user|
    user["birthdate"].day_of_year().eq(1)
}
```

Command syntax

number - number → number

time - time → number

time - number → time

Description

Subtract two numbers.

Example: It's as easy as 2 - 2 = 0.

```
(r.expr(2) - 2).run(conn)
```

Example: Create a date one year ago today.

```
r.now() - 365*24*60*60
```

Example: Retrieve how many seconds elapsed between today and date

```
r.now() - date
```

Command syntax

stream.zip() → stream

array.zip() → array

Description

Used to 'zip' up the result of a join by merging the 'right' fields into 'left' fields of each member of the sequence.

Example: 'zips up' the sequence by merging the left and right fields produced by a join.

```
r.table('marvel').eq_join(:main_dc_collaborator, r.table('dc')).zip.run(conn)
```

Command syntax

table.insert(json | [json][, :durability => "hard", :return_vals => false :upsert => false]) → object

Description

Insert documents into a table. Accepts a single document or an array of documents.

The optional arguments are:

- `durability`: possible values are `hard` and `soft`. This option will override the table or query's durability setting (set in run).
  In soft durability mode RethinkDB will acknowledge the write immediately after receiving it, but before the write has been committed to disk.

- `return_vals`: if set to `true` and in case of a single insert/upsert, the inserted/updated document will be returned.
- `upsert`: when set to `true`, performs a replace if a document with the same primary key exists.

Insert returns an object that contains the following attributes:

- `inserted`: the number of documents that were succesfully inserted.
- `replaced`: the number of documents that were updated when upsert is used.
- `unchanged`: the number of documents that would have been modified, except that the new value was the same as the old value when doing an upsert.
- `errors`: the number of errors encountered while performing the insert.
- `first_error`: If errors were encountered, contains the text of the first error.
- `deleted` and `skipped`: 0 for an insert operation.
- `generated_keys`: a list of generated primary keys in case the primary keys for some documents were missing (capped to 100000).
- `warnings`: if the field `generated_keys` is truncated, you will get the warning *"Too many generated keys (<X>), array truncated to 100000."*.
- `old_val`: if `return_vals` is set to `true`, contains `nil`.
- `new_val`: if `return_vals` is set to `true`, contains the inserted/updated document.

Example: Insert a document into the table `posts`.

```
r.table("posts").insert({
    :id => 1,
    :title => "Lorem ipsum",
    :content => "Dolor sit amet"
}).run(conn)
```

The result will be:

```
{
    :deleted => 0,
    :errors => 0,
    :inserted => 1,
    :replaced => 0,
    :skipped => 0,
    :unchanged => 0
}
```

Example: Insert a document without a defined primary key into the table `posts` where the primary key is `id`.

```
r.table("posts").insert({
    :title => "Lorem ipsum",
    :content => "Dolor sit amet"
}).run(conn)
```

RethinkDB will generate a primary key and return it in `generated_keys`.

```
{
    :deleted => 0,
    :errors => 0,
    :generated_keys => [
        "dd782b64-70a7-43e4-b65e-dd14ae61d947"
    ],
    :inserted => 1,
    :replaced => 0,
    :skipped => 0,
    :unchanged => 0
}
```

Retrieve the document you just inserted with:

```
r.table("posts").get("dd782b64-70a7-43e4-b65e-dd14ae61d947").run(conn)
```

And you will get back:

```
{
    :id => "dd782b64-70a7-43e4-b65e-dd14ae61d947",
    :title => "Lorem ipsum",
    :content => "Dolor sit amet"
}
```

Example: Insert multiple documents into the table `users`.

```
r.table("users").insert([
    {:id => "william", :email =>"william@rethinkdb.com"},
    {:id => "lara", :email => "lara@rethinkdb.com"}
]).run(conn)
```

Example: Insert a document into the table `users`, replacing the document if the document already exists.
Note: If the document exists, the `insert` command will behave like replace, not like update

```
r.table("users").insert(
    {:id => "william", :email => "william@rethinkdb.com"},
    :upsert => true
).run(conn)
```

Example: Copy the documents from `posts` to `posts_backup`.

```
r.table("posts_backup").insert( r.table("posts") ).run(conn)
```

Example: Get back a copy of the inserted document (with its generated primary key).

```
r.table("posts").insert(
    {:title => "Lorem ipsum", :content => "Dolor sit amet"},
    :return_vals => true
).run(conn)
```

The result will be

```
{
    :deleted => 0,
    :errors => 0,
    :generated_keys => [
        "dd782b64-70a7-43e4-b65e-dd14ae61d947"
    ],
    :inserted => 1,
    :replaced => 0,
    :skipped => 0,
    :unchanged => 0,
    :old_val => nil,
    :new_val => {
        :id => "dd782b64-70a7-43e4-b65e-dd14ae61d947",
        :title => "Lorem ipsum",
        :content => "Dolor sit amet"
    }
}
```

Command syntax

any.type_of() → string

Description

Gets the type of a value.

Example: Get the type of a string.

```
r.expr("foo").type_of().run(conn)
```

Command syntax

value.eq(value) → bool

Description

Test if two values are equal.

Example: Does 2 equal 2?

```
r.expr(2).eq(2).run(conn)
```

Command syntax

sequence.pluck([selector1, selector2...]) → stream

array.pluck([selector1, selector2...]) → array

object.pluck([selector1, selector2...]) → object

singleSelection.pluck([selector1, selector2...]) → object

Description

Plucks out one or more attributes from either an object or a sequence of objects (projection).

Example: We just need information about IronMan's reactor and not the rest of the document.

```
r.table('marvel').get('IronMan').pluck('reactorState', 'reactorPower').run(conn)
```

Example: For the hero beauty contest we only care about certain qualities.

```
r.table('marvel').pluck('beauty', 'muscleTone', 'charm').run(conn)
```

Example: Pluck can also be used on nested objects.

```
r.table('marvel').pluck({:abilities => {:damage => true, :mana_cost => true}, :weapons => tr
```

Example: The nested syntax can quickly become overly verbose so there's a shorthand for it.

```
r.table('marvel').pluck({:abilities => [:damage, :mana_cost]}, :weapons).run(conn)
```

Command syntax

array.prepend(value) → array

Description

Prepend a value to an array.

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')[:equipment].prepend('new_boots').run(conn)
```

Command syntax

value.default(default_value) → any

sequence.default(default_value) → any

Description

Handle non-existence errors. Tries to evaluate and return its first argument. If an error related to the absence of a value is thrown in the process, or if its first argument returns `nil`, returns its second argument. (Alternatively, the second argument may be a function which will be called with either the text of the non-existence error or `nil`.)

Example: Suppose we want to retrieve the titles and authors of the table `posts`. In the case where the author field is missing or `nil`, we want to retrieve the string `Anonymous`.

```
r.table("posts").map{ |post|
    {
        :title => post[:title],
        :author => post[:author].default("Anonymous")
    }
}.run(conn)
```

We can rewrite the previous query with `r.branch` too.

```
r.table("posts").map{ |post|
    r.branch(
        post.has_fields("author"),
        {
            :title => post[:title],
            :author => post[:author]
        },
        {
            :title => post[:title],
            :author => "Anonymous"
        }
    )
}.run(conn)
```

Example: The `default` command can be useful to filter documents too. Suppose we want to retrieve all our users who are not grown-ups or whose age is unknown (i.e the field `age` is missing or equals `nil`). We can do it with this query:

```
r.table("users").filter{ |user|
    (user[:age] < 18).default(true)
}.run(conn)
```

One more way to write the previous query is to set the age to be `-1` when the field is missing.

```
r.table("users").filter{ |user|
    user[:age].default(-1) < 18
}.run(conn)
```

One last way to do the same query is to use `has_fields`.

```
r.table("users").filter{ |user|
    user.has_fields("age").not() | (user[:age] < 18)
}.run(conn)
```

The body of every `filter` is wrapped in an implicit `.default(false)`. You can overwrite the value `false` by passing an option in filter, so the previous query can also be written like this.

```
r.table('users').filter(:default => true) {|user|
    (user[:age] < 18)
}.run(conn)
```

Command syntax

r.time(year, month, day[, hour, minute, second], timezone) → time

Description

Create a time object for a specific time.

A few restrictions exist on the arguments:

- `year` is an integer between 1400 and 9,999.
- `month` is an integer between 1 and 12.
- `day` is an integer between 1 and 31.
- `hour` is an integer.
- `minutes` is an integer.
- `seconds` is a double. Its value will be rounded to three decimal places (millisecond-precision).
- `timezone` can be `'Z'` (for UTC) or a string with the format `±[hh]:[mm]`.

Example: Update the birthdate of the user "John" to November 3rd, 1986 UTC.

```
r.table("user").get("John").update(:birthdate => r.time(1986, 11, 3, 'Z')).run(conn)
```

Command syntax

sequence.with_fields([selector1, selector2...]) → stream

array.with_fields([selector1, selector2...]) → array

Description

Takes a sequence of objects and a list of fields. If any objects in the sequence don't have all of the specified fields, they're dropped from the sequence. The remaining objects have the specified fields plucked out. (This is identical to `has_fields` followed by `pluck` on a sequence.)

Example: Get a list of heroes and their nemeses, excluding any heroes that lack one.

```
r.table('marvel').with_fields('id', 'nemesis')
```

Example: Get a list of heroes and their nemeses, excluding any heroes whose nemesis isn't in an evil organization.

```
r.table('marvel').with_fields(:id, {:nemesis => {:evil_organization => true}})
```

Example: The nested syntax can quickly become overly verbose so there's a shorthand.

```
r.table('marvel').with_fields(:id, {:nemesis => :evil_organization})
```

Command syntax

table.index_wait([, index...]) → array

Description

Wait for the specified indexes on this table to be ready, or for all indexes on this table to be ready if no indexes are specified.

The result is an array where for each index, there will be an object like:

```
{
    :index => <index_name>,
    :ready => true
}
```

Example: Wait for all indexes on the table `test` to be ready:

```
r.table('test').index_wait.run(conn)
```

Example: Wait for the index `timestamp` to be ready:

```
r.table('test').index_wait('timestamp').run(conn)
```

Command syntax

r.epoch_time(epoch_time) → time

Description

Create a time object based on seconds since epoch. The first argument is a double and will be rounded to three decimal places (millisecond-precision).

Example: Update the birthdate of the user "John" to November 3rd, 1986.

```
r.table("user").get("John").update(:birthdate => r.epoch_time(531360000)).run(conn)
```

Command syntax

sequence.reduce(reduction_function[, base]) → value

Description

Produce a single value from a sequence through repeated application of a reduction function.

The reduce function gets invoked repeatedly not only for the input values but also for results of previous reduce invocations. The type and format of the object that is passed in to reduce must be the same with the one returned from reduce.

Example: How many enemies have our heroes defeated?

```
r.table('marvel').order_by(:strength)[5..10].run(conn)
```

Command syntax

sequence.skip(n) → stream

array.skip(n) → array

Description

Skip a number of elements from the head of the sequence.

Example: Here in conjunction with `order_by` we choose to ignore the most successful heroes.

```
r.table('marvel').order_by(:success_metric).skip(10).run(conn)
```

Command syntax

db.table_list() → array

Description

List all table names in a database. The result is a list of strings.

Example: List all tables of the 'test' database.

```
r.db('test').table_list().run(conn)
```

Command syntax

conn.use(db_name)

Description

Change the default database on this connection.

Example: Change the default database so that we don't need to specify the database when referencing a table.

```
conn.use('marvel')
r.table('heroes').run(conn) # refers to r.db('marvel').table('heroes')
```

Command syntax

table.between(lower_key, upper_key [, :index => 'id', :left_bound => 'closed', :right_bound => 'open']) → selection

Description

Get all documents between two keys. Accepts three optional arguments: `index`, `left_bound`, and `right_bound`. If `index` is set to the name of a secondary index, `between` will return all documents where that index's value is in the specified range (it uses the primary key by default). `left_bound` or `right_bound` may be set to `open` or `closed` to indicate whether or not to include that endpoint of the range (by default, `left_bound` is closed and `right_bound` is open).

Example: Find all users with primary key >= 10 and < 20 (a normal half-open interval).

```
r.table('marvel').between(10, 20).run(conn)
```

Example: Find all users with primary key >= 10 and <= 20 (an interval closed on both sides).

```
r.table('marvel').between(10, 20, :right_bound => 'closed').run(conn)
```

Example: Find all users with primary key < 20. (You can use `NULL` to mean "unbounded" for either endpoint.)

```
r.table('marvel').between(nil, 20, :right_bound => 'closed').run(conn)
```

Example: Between can be used on secondary indexes too. Just pass an optional index argument giving the secondary index to query.

```
r.table('dc').between('dark_knight', 'man_of_steel', :index => 'code_name').run(conn)
```

Command syntax

number / number → number

Description

Divide two numbers.

Example: It's as easy as 2 / 2 = 1.

```
(r.expr(2) / 2).run(conn)
```

Command syntax

sequence.outer_join(other_sequence, predicate) → stream

array.outer_join(other_sequence, predicate) → array

Description

Computes a left outer join by retaining each row in the left table even if no match was found in the right table.

Example: Construct a sequence of documents containing all cross-universe matchups where a marvel hero would lose, but keep marvel heroes who would never lose a matchup in the sequence.

```
r.table('marvel').outer_join(r.table('dc')) {|marvel_row, dc_row|
    marvel_row[:strength] < dc_row[:strength]
}.run(conn)
```

Command syntax

time.timezone() → string

Description

Return the timezone of the time object.

Example: Return all the users in the "-07:00" timezone.

```
r.table("users").filter{ |user|
    user["subscriptionDate"].timezone().eq("07:00")
}
```

Command syntax

r.branch(test, true_branch, false_branch) → any

Description

If the `test` expression returns `false` or `nil`, the `false_branch` will be evaluated. Otherwise, the `true_branch` will be evaluated.

The `branch` command is effectively an `if` renamed due to language constraints. The type of the result is determined by the type of the branch that gets executed.

Example: Return heroes and superheroes.

```
r.table('marvel').map{ |hero|
    r.branch(
        hero['victories'] > 100,
        hero['name'].add(' is a superhero'),
        hero['name'].add(' is a hero')
    )
}.run(conn)
```

If the documents in the table `marvel` are:

```
[{
    :name => "Iron Man",
    :victories => 214
},
{
    :name => "Jubilee",
    :victories => 9
}]
```

The results will be:

```
[
    "Iron Man is a superhero",
    "Jubilee is a hero"
]
```

Command syntax

any.do(arg [, args]*, expr) → any

Description

Evaluate the expr in the context of one or more value bindings.

The type of the result is the type of the value returned from expr.

Example: The object(s) passed to do() can be bound to name(s). The last argument is the expression to evaluate in the context of the bindings.

```
r.do(r.table('marvel').get('IronMan')) { |ironman| ironman[:name] }.run(conn)
```

Command syntax

table.sync() → object

Description

`sync` ensures that writes on a given table are written to permanent storage. Queries that specify soft durability (`{:durability => soft}`) do not give such guarantees, so `sync` can be used to ensure the state of these queries. A call to `sync` does not return until all previous writes to the table are persisted.

If successful, the operation returns an object: `{"synced": 1}`.

Example: After having updated multiple heroes with soft durability, we now want to wait until these changes are persisted.

```
r.table('marvel').sync().run(conn)
```

Command syntax

sequence[start_index[..end_index]] → stream

array[start_index[..end_index]] → array

Description

Trim the sequence to within the bounds provided.

Example: For this fight, we need heroes with a good mix of strength and agility.

```
r.table('marvel').order_by(:strength)[5..10].run(conn)
```

Command syntax

table.replace(json | expr[, :durability => “hard”, :return_vals => false, :non_atomic => false]) → object

selection.replace(json | expr[, :durability => “hard”, :return_vals => false, :non_atomic => false]) → object

singleSelection.replace(json | expr[, :durability => “hard”, :return_vals => false, :non_atomic => false]) → object

Description

Replace documents in a table. Accepts a JSON document or a ReQL expression, and replaces the original document with the new one. The new document must have the same primary key as the original document.

The optional arguments are:

- `durability`: possible values are `hard` and `soft`. This option will override the table or query's durability setting (set in run).
  In soft durability mode RethinkDB will acknowledge the write immediately after receiving it, but before the write has been committed to disk.
- `return_vals`: if set to `true` and in case of a single replace, the replaced document will be returned.
- `non_atomic`: set to `true` if you want to perform non-atomic replaces (replaces that require fetching data from another document).

Replace returns an object that contains the following attributes:

- `replaced`: the number of documents that were replaced
- `unchanged`: the number of documents that would have been modified, except that the new value was the same as the old value
- `inserted`: the number of new documents added. You can have new documents inserted if you do a point-replace on a key that isn't in the table or you do a replace on a selection and one of the documents you are replacing has been deleted
- `deleted`: the number of deleted documents when doing a replace with `nil`
- `errors`: the number of errors encountered while performing the replace.
- `first_error`: If errors were encountered, contains the text of the first error.
- `skipped`: 0 for a replace operation
- `old_val`: if `return_vals` is set to `true`, contains the old document.

- `new_val`: if `return_vals` is set to `true`, contains the new document.

Example: Replace the document with the primary key `1`.

```
r.table("posts").get(1).replace({
    :id => 1,
    :title => "Lorem ipsum",
    :content => "Aleas jacta est",
    :status => "draft"
}).run(conn)
```

Example: Remove the field `status` from all posts.

```
r.table("posts").replace{ |post|
    post.without("status")
}.run(conn)
```

Example: Remove all the fields that are not `id`, `title` or `content`.

```
r.table("posts").replace{ |post|
    post.pluck("id", "title", "content")
}.run(conn)
```

Example: Replace the document with the primary key `1` using soft durability.

```
r.table("posts").get(1).replace({
    :id => 1,
    :title => "Lorem ipsum",
    :content => "Aleas jacta est",
    :status => "draft"
}, :durability => "soft").run(conn)
```

Example: Replace the document with the primary key `1` and return the values of the document before and after the replace operation.

```
r.table("posts").get(1).replace({
    :id => 1,
    :title => "Lorem ipsum",
    :content => "Aleas jacta est",
    :status => "published"
}, :return_vals => true).run(conn)
```

The result will have two fields `old_val` and `new_val`.

```
{
    :deleted => 0,
    :errors => 0,
    :inserted => 0,
    :new_val => {
        :id => 1,
        :title => "Lorem ipsum"
        :content => "Aleas jacta est",
        :status => "published",
    },
    :old_val => {
        :id => 1,
        :title => "Lorem ipsum"
        :content => "TODO",
        :status => "draft",
        :author => "William",
    },
    :replaced => 1,
    :skipped => 0,
    :unchanged => 0
}
```

Command syntax

table.order_by([key1…], :index => index_name) -> selection<stream>

selection.order_by(key1, [key2…]) -> selection<array>

sequence.order_by(key1, [key2…]) -> array

Description

Sort the sequence by document values of the given key(s). `orderBy` defaults to ascending ordering. To explicitly specify the ordering, wrap the attribute with either `r.asc` or `r.desc`.

Example: Order our heroes by a series of performance metrics.

```
r.table('marvel').order_by(:enemies_vanquished, :damsels_saved).run(conn)
```

Example: Indexes can be used to perform more efficient orderings. Notice that the index ordering always has highest precedence. Thus the following example is equivalent to the one above.

```
r.table('marvel').order_by(:damsels_saved, :index => :enemies_vanquished).run(conn)
```

Example: You can also specify a descending order when using an index.

```
r.table('marvel').order_by(:index => r.desc(:enemies_vanquished)).run(conn)
```

Example: Let's lead with our best vanquishers by specify descending ordering.

```
r.table('marvel').order_by(r.desc(:enemies_vanquished),
r.asc(:damsels_saved)      ).run(conn)
```

Example: You can use a function for ordering instead of just selecting an attribute.

```
r.table('marvel').order_by(lambda {|doc| doc[:enemiesVanquished] + doc[:damselsSaved]}).ru
```

Example: Functions can also be used descendingly.

```
r.table('marvel').order_by(r.desc(lambda {|doc| doc[:enemiesVanquished] + doc[:damselsSave
```

Command syntax

r.db_create(db_name) → object

Description

Create a database. A RethinkDB database is a collection of tables, similar to relational databases.

If successful, the operation returns an object: `{"created": 1}`. If a database with the same name already exists the operation throws `RqlRuntimeError`.

Note: that you can only use alphanumeric characters and underscores for the database name.

Example: Create a database named 'superheroes'.

```
r.db_create('superheroes').run(conn)
```

Command syntax

sequence.coerce_to(type_name) → array

value.coerce_to(type_name) → string

array.coerce_to(type_name) → object

object.coerce_to(type_name) → array

Description

Converts a value of one type into another.

You can convert: a selection, sequence, or object into an ARRAY, an array of pairs into an OBJECT, and any DATUM into a STRING.

Example: Convert a table to an array.

```
r.table('marvel').coerce_to('array').run(conn)
```

Example: Convert an array of pairs into an object.

```
r.expr([['name', 'Ironman'], ['victories', 2000]]).coerce_to('object').run(conn)
```

Example: Convert a number to a string.

```
r.expr(1).coerce_to('string').run(conn)
```

Command syntax

sequence.contains(value1[, value2...]) → bool

Description

Returns whether or not a sequence contains all the specified values, or if functions are provided instead, returns whether or not a sequence contains values matching all the specified functions.

Example: Has Iron Man ever fought Superman?

```
r.table('marvel').get('ironman')[:opponents].contains('superman').run(conn)
```

Example: Has Iron Man ever defeated Superman in battle?

```
r.table('marvel').get('ironman')[:battles].contains{|battle|
    battle[:winner].eq('ironman') & battle[:loser].eq('superman')
}.run(conn)
```

Command syntax

table.index_drop(index_name) → object

Description

Delete a previously created secondary index of this table.

Example: Drop a secondary index named 'code_name'.

```
r.table('dc').index_drop('code_name').run(conn)
```

Command syntax

connection.repl

Description

Set the default connection to make REPL use easier. Allows calling `.run` on queries without specifying a connection.

Connection objects are not thread-safe and REPL connections should not be used in multi-threaded environments.

Example: Set the default connection for the REPL, then call `run` without specifying the connection.

```
r.connect(:db => 'marvel').repl
r.table('heroes').run
```

Command syntax

bool & bool → bool

Description

Compute the logical and of two values.

Example: True and false anded is false?

```
(r.expr(True) & False).run(conn)
```

Command syntax

array.insert_at(index, value) → array

Description

Insert a value in to an array at a given index. Returns the modified array.

Example: Hulk decides to join the avengers.

```
r.expr(["Iron Man", "Spider-Man"]).insert_at(1, "Hulk").run(conn)
```

Command syntax

time.during(start_time, end_time[, left_bound="open/closed", right_bound="open/closed"]) → bool

Description

Return if a time is between two other times (by default, inclusive for the start, exclusive for the end).

Example: Retrieve all the posts that were posted between December 1st, 2013 (inclusive) and December 10th, 2013 (exclusive).

```
r.table("posts").filter{ |post|
   post['date'].during(r.time(2013, 12, 1, "Z"), r.time(2013, 12, 10, "Z"))
}.run(conn)
```

Example: Retrieve all the posts that were posted between December 1st, 2013 (exclusive) and December 10th, 2013 (inclusive).

```
r.table("posts").filter{ |post|
   post['date'].during(r.time(2013, 12, 1, "Z"), r.time(2013, 12, 10, "Z"),
         :left_bound => "open",
         :right_bound => "closed")
}.run(conn)
```

Command syntax

time.to_iso8601() → number

Description

Convert a time object to its iso 8601 format.

Example: Return the current time in an ISO8601 format.

```
r.now().to_iso8601()
```

Command syntax

table.get_all(key[, key2...], [, :index => 'id']) → selection

Description

Get all documents where the given value matches the value of the requested index.

Example: Secondary index keys are not guaranteed to be unique so we cannot query via "get" when using a secondary index.

```
r.table('marvel').get_all('man_of_steel', :index => 'code_name').run(conn)
```

Example: Without an index argument, we default to the primary index. While `get` will either return the document or `null` when no document with such a primary key value exists, this will return either a one or zero length stream.

```
r.table('dc').get_all('superman').run(conn)
```

Example: You can get multiple documents in a single call to `get_all`.

```
r.table('dc').get_all('superman', 'ant man').run(conn)
```

Command syntax

time.to_epoch_time() → number

Description

Convert a time object to its epoch time.

Example: Return the current time in an ISO8601 format.

```
r.now().to_epoch_time()
```

Command syntax

time.in_timezone(timezone) → time

Description

Return a new time object with a different timezone. While the time stays the same, the results returned by methods such as hours() will change since they take the timezone into account. The timezone argument has to be of the ISO 8601 format.

Example: Hour of the day in San Francisco (UTC/GMT -8, without daylight saving time).

```
r.now().in_timezone('-08:00').hours().run(conn)
```

Command syntax

value <= value → bool

value.le(value) → bool

Description

Test if the first value is less than or equal to other.

Example: Is 2 less than or equal to 2?

```
(r.expr(2) <= 2).run(conn)
r.expr(2).le(2).run(conn)
```

Command syntax

r.expr(value) → value

Description

Construct a ReQL JSON object from a native object.

Example: Objects wrapped with expr can then be manipulated by ReQL API functions.

```
r.expr({:a => 'b'}).merge({:b => [1,2,3]}).run(conn)
```

Example: In Ruby, you can also do this with just r.

```
r.expr({:a => 'b'}).merge({:b => [1,2,3]}).run(conn)
```

Command syntax

number * number → number

array * number → array

Description

Multiply two numbers, or make a periodic array.

Example: It's as easy as 2 * 2 = 4.

```
(r.expr(2) * 2).run(conn)
```

Example: Arrays can be multiplied by numbers as well.

```
(r.expr(["This", "is", "the", "song", "that", "never", "ends."]) * 100).run(conn)
```

Command syntax

number + number → number

string + string → string

array + array → array

time + number → time

Description

Sum two numbers, concatenate two strings, or concatenate 2 arrays.

Example: It's as easy as 2 + 2 = 4.

```
(r.expr(2) + 2).run(conn)
```

Example: Strings can be concatenated too.

```
(r("foo") + "bar").run(conn)
```

Example: Arrays can be concatenated too.

```
(r(["foo", "bar"]) + ["buzz"]).run(conn)
```

Example: Create a date one year from now.

```
r.now() + 365*24*60*60
```

Command syntax

table.index_status([, index...]) → array

Description

Get the status of the specified indexes on this table, or the status of all indexes on this table if no indexes are specified.

The result is an array where for each index, there will be an object like this one:

```
{
    :index => <index_name>,
    :ready => true
}
```

or this one:

```
{
    :index => <index_name>,
    :ready => false,
    :blocks_processed => <int>,
    :blocks_total => <int>
}
```

Example: Get the status of all the indexes on `test`:

```
r.table('test').index_status.run(conn)
```

Example: Get the status of the `timestamp` index:

```
r.table('test').index_status('timestamp').run(conn)
```

Command syntax

sequence.filter(predicate[, :default => false]) → selection

stream.filter(predicate[, :default => false]) → stream

array.filter(predicate[, :default => false]) → array

Description

Get all the documents for which the given predicate is true.

`filter` can be called on a sequence, selection, or a field containing an array of elements. The return type is the same as the type on which the function was called on.

The body of every filter is wrapped in an implicit `.default(false)`, which means that if a non-existence errors is thrown (when you try to access a field that does not exist in a document), RethinkDB will just ignore the document. The `default` value can be changed by passing the symbol `default`. Setting this optional argument to `r.error()` will cause any non-existence errors to return a `RqlRuntimeError`.

Example: Get all the users that are 30 years old.

```
r.table('users').filter({:age => 30}).run(conn)
```

A more general way to write the previous query is to use Ruby's block.

```
r.table('users').filter{|user|
    user["age"].eq(30)
}.run(conn)
```

Here the predicate is `user["age"].eq(30)`.

- `user` refers to the current document
- `user["age"]` refers to the field `age` of the current document
- `user["age"].eq(30)` returns `true` if the field `age` is 30

Example: Get all the users that are more than 18 years old.

```
r.table("users").filter{|user|
    user["age"] > 18
}.run(conn)
```

Example: Get all the users that are less than 18 years old and more than 13 years old.

```
r.table("users").filter{|user|
    (user["age"] < 18) & (user["age"] > 13)
}.run(conn)
```

Example: Get all the users that are more than 18 years old or have their parental consent.

```
r.table("users").filter{|user| (user["age"].lt(18)) | (user["hasParentalConsent"])}.run(c
```

Example: Get all the users that are less than 18 years old or whose age is unknown (field `age` missing).

```
r.table("users").filter(
    lambda { |user| user["age"] < 18 },
    :default => true
).run(conn)
```

Example: Get all the users that are more than 18 years old. Throw an error if a document is missing the field `age`.

```
r.table("users").filter(
    lambda { |user| user["age"] > 18 },
    :default => r.error()
).run(conn)
```

Example: Select all users who have given their phone number (all the documents whose field `phone_number` is defined and not `None`).

```
r.table('users').filter{|user|
    user.has_fields('phone_number')
}.run(conn)
```

Example: Retrieve all the users who subscribed between January 1st, 2012 (included) and January 1st, 2013 (excluded).

```
r.table("users").filter{|user|
   user["subscription_date"].during( r.time(2012, 1, 1, 'Z'), r.time(2013, 1, 1, 'Z') )
}.run(conn)
```

Example: Retrieve all the users who have a gmail account (whose field `email` ends with `@gmail.com`).

```
r.table("users").filter{|user|
    user["email"].match("@gmail.com$")
}.run(conn)
```

Example: Filter based on the presence of a value in an array.

Suppose the table `users` has the following schema

```
{
    :name => String
    :places_visited => [String]
}
```

Retrieve all the users whose field `places_visited` contains `France`.

```
r.table("users").filter{|user|
    user["places_visited"].contains("France")
}.run(conn)
```

Example: Filter based on nested fields.

Suppose we have a table `users` containing documents with the following schema.

```
{
    :id => String
    :name => {
        :first => String,
        :middle => String,
        :last => String
    }
}
```

Retrieve all users named "William Adama" (first name "William", last name "Adama"), with any middle name.

```
r.table("users").filter({
    :name =>{
        :first => "William",
        :last => "Adama"
    }
}).run(conn)
```

If you want an exact match for a field that is an object, you will have to use `r.literal`.

Retrieve all users named "William Adama" (first name "William", last name "Adama"), and who do not have a middle name.

```
r.table("users").filter(r.literal({
    :name => {
        :first => "William",
        :last=> "Adama"
    }
})).run(conn)
```

The equivalent queries with a lambda function.

```
r.table("users").filter{|user|
    (user["name"]["first"].eq("William")) &
    (user["name"]["last"].eq("Adama"))
}.run(conn)

r.table("users").filter{|user|
    user["name"].eq({
        :first => "William",
        :last => "Adama"
    })
}.run(conn)
```

Command syntax

table.index_list() → array

Description

List all the secondary indexes of this table.

Example: List the available secondary indexes for this table.

```
r.table('marvel').index_list().run(conn)
```

Command syntax

table.index_create(index_name[, index_function]) → object

Description

Create a new secondary index on this table.

Example: To efficiently query our heros by name we can create a secondary index based on the value of that field. We can already quickly query heros by name with the primary index but to do the same based on hero code names we'll have to create a secondary index based on that attribute.

```
r.table('dc').index_create('code_name').run(conn)
```

Example: You can also create a secondary index based on an arbitrary function on the document.

```
r.table('dc').index_create('power_rating') {|hero|
    hero['combat_power'] + (2 * hero['compassion_power'])
}.run(conn)
```

Example: A compound index can be created by returning an array of values to use as the secondary index key.

```
r.table('dc').index_create('parental_planets') {|hero|
    [hero['mothers_home_planet'], hero['fathers_home_planet']]
}.run(conn)
```

Example: A multi index can be created by passing an optional multi argument. Multi indexes functions should return arrays and allow you to query based on whether a value is present in the returned array. The example would allow us to get heroes who possess a specific ability (the field 'abilities' is an array).

```
r.table('dc').index_create('abilities', :multi => true).run(conn)
```

Command syntax

singleSelection.keys() → array

object.keys() → array

Description

Return an array containing all of the object's keys.

Example: Get all the keys of a row.

```
r.table('marvel').get('ironman').keys.run(conn)
```

Command syntax

time.day_of_week() → number

Description

Return the day of week of a time object as a number between 1 and 7 (following ISO 8601 standard). For your convenience, the terms r.monday, r.tuesday etc. are defined and map to the appropriate integer.

Example: Return today's day of week.

```
r.now().day_of_week().run(conn)
```

Example: Retrieve all the users who were born on a Tuesday.

```
r.table("users").filter{ |user|
    user["birthdate"].day_of_week().eq(r.tuesday)
}
```

Command syntax

array.set_insert(value) → array

Description

Add a value to an array and return it as a set (an array with distinct values).

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')[:equipment].set_insert('new_boots').run(conn)
```

Command syntax

sequence.has_fields([selector1, selector2...]) → stream

array.has_fields([selector1, selector2...]) → array

singleSelection.has_fields([selector1, selector2...]) → boolean

object.has_fields([selector1, selector2...]) → boolean

Description

Test if an object has all of the specified fields. An object has a field if it has the specified key and that key maps to a non-null value. For instance, the object `{:a => 1, :b => 2, :c => nil}` has the fields `a` and `b`.

Example: Which heroes are married?

```
r.table('marvel').has_fields(:spouse).run(conn)
```

Example: Test if a single object has a field.

```
r.table('marvel').get("IronMan").has_fields(:spouse).run(conn)
```

Example: You can also test if nested fields exist to get only spouses with powers of their own.

```
r.table('marvel').has_fields({:spouse => {:powers => true}}).run(conn)
```

Example: The nested syntax can quickly get verbose so there's a shorthand.

```
r.table('marvel').has_fields({:spouse => :powers}).run(conn)
```

Command syntax

value > value → bool

value.gt(value) → bool

Description

Test if the first value is greater than other.

Example: Is 2 greater than 2?

```
(r.expr(2) > 2).run(conn)
r.expr(2).gt(2).run(conn)
```

Command syntax

time.month() → number

Description

Return the month of a time object as a number between 1 and 12. For your convenience, the terms r.january, r.february etc. are defined and map to the appropriate integer.

Example: Retrieve all the users who were born in November.

```
r.table("users").filter{ |user|
    user["birthdate"].month().eq(11)
}
```

Example: Retrieve all the users who were born in November.

```
r.table("users").filter{ |user|
    user["birthdate"].month().eq(r.november)
}
```

Command syntax

sequence.inner_join(other_sequence, predicate) → stream

array.inner_join(other_sequence, predicate) → array

Description

Returns the inner product of two sequences (e.g. a table, a filter result) filtered by the predicate. The query compares each row of the left sequence with each row of the right sequence to find all pairs of rows which satisfy the predicate. When the predicate is satisfied, each matched pair of rows of both sequences are combined into a result row.

Example: Construct a sequence of documents containing all cross-universe matchups where a marvel hero would lose.

```
r.table('marvel').inner_join(r.table('dc')) {|marvel_row, dc_row|
    marvel_row[:strength] < dc_row[:strength]
}.run(conn)
```

Command syntax

bool.not() → bool

Description

Compute the logical inverse (not).

Example: Not true is false.

```
r(true).not.run(conn)
```

Command syntax

sequence.without([selector1, selector2...]) → stream

array.without([selector1, selector2...]) → array

singleSelection.without([selector1, selector2...]) → object

object.without([selector1, selector2...]) → object

Description

The opposite of pluck; takes an object or a sequence of objects, and returns them with the specified paths removed.

Example: Since we don't need it for this computation we'll save bandwidth and leave out the list of IronMan's romantic conquests.

```
r.table('marvel').get('IronMan').without('personalVictoriesList').run(conn)
```

Example: Without their prized weapons, our enemies will quickly be vanquished.

```
r.table('enemies').without('weapons').run(conn)
```

Example: Nested objects can be used to remove the damage subfield from the weapons and abilities fields.

```
r.table('marvel').without({:weapons => {:damage => true}, :abilities => {:damage => true}})
```

Example: The nested syntax can quickly become overly verbose so there's a shorthand for it.

```
r.table('marvel').without({:weapons => :damage, :abilities => :damage}).run(conn)
```

Command syntax

r.sum(attr)

Description

Compute the sum of the given field in the group.

Example: How many enemies have been vanquished by heroes at each strength level?

```
r.table('marvel').group_by(:strength, r.sum(:enemies_vanquished)).run(conn)
```

Command syntax

time.year() → number

Description

Return the year of a time object.

Example: Retrieve all the users born in 1986.

```
r.table("users").filter(function(user) {
    return user("birthdate").year().eq(1986)
}).run(conn, callback)
```

Command syntax

sequence(attr) → sequence

singleSelection(attr) → value

object(attr) → value

Description

Get a single field from an object. If called on a sequence, gets that field from every object in the sequence, skipping objects that lack it.

Example: What was Iron Man's first appearance in a comic?

```
r.table('marvel').get('IronMan')('firstAppearance').run(conn, callback)
```

Command syntax

sequence.union(sequence) → array

Description

Concatenate two sequences.

Example: Construct a stream of all heroes.

```
r.table('marvel').union(r.table('dc')).run(conn, callback)
```

Command syntax

time.day() → number

Description

Return the day of a time object as a number between 1 and 31.

Example: Return the users born on the 24th of any month.

```
r.table("users").filter(
    r.row("birthdate").day().eq(24)
).run(conn, callback)
```

Command syntax

r.db(dbName) → db

Description

Reference a database.

Example: Before we can query a table we have to select the correct database.

```
r.db('heroes').table('marvel').run(conn, callback)
```

Command syntax

array.deleteAt(index [,endIndex]) → array

Description

Remove an element from an array at a given index. Returns the modified array.

Example: Hulk decides to leave the avengers.

```
r.expr(["Iron Man", "Hulk", "Spider-Man"]).deleteAt(1).run(conn, callback)
```

Example: Hulk and Thor decide to leave the avengers.

```
r.expr(["Iron Man", "Hulk", "Thor", "Spider-Man"]).deleteAt(1,3).run(conn, callback)
```

Command syntax

sequence.distinct() → array

Description

Remove duplicate elements from the sequence.

Example: Which unique villains have been vanquished by marvel heroes?

```
r.table('marvel').concatMap(function(hero) {return hero('villainList')}).distinct()
    .run(conn, callback)
```

Command syntax

array.difference(array) → array

Description

Remove the elements of one array from another array.

Example: Retrieve Iron Man's equipment list without boots.

```
r.table('marvel').get('IronMan')('equipment').difference(['Boots']).run(conn, callback)
```

Command syntax

connection.addListener(event, listener)

Description

The connection object also supports the event emitter interface so you can listen for changes in connection state.

Example: Monitor connection state with events 'connect', 'close', and 'error'.

```
r.connect({}, function(err, conn) {
    if (err) throw err;

    conn.addListener('error', function(e) {
        processNetworkError(e);
    });

    conn.addListener('close', function() {
        cleanup();
    });

    runQueries(conn);
});
```

Example: As in Node, 'on' is a synonym for 'addListener'.

```
conn.on('close', function() {
    cleanup();
});
conn.close();
```

Command syntax

r.dbDrop(dbName) → object

Description

Drop a database. The database, all its tables, and corresponding data will be deleted.

If successful, the operation returns the object `{dropped: 1}`. If the specified database doesn't exist a `RqlRuntimeError` is thrown.

Example: Drop a database named 'superheroes'.

```
r.dbDrop('superheroes').run(conn, callback)
```

Command syntax

db.table(name[, {useOutdated: false}]) → table

Description

Select all documents in a table. This command can be chained with other commands to do further processing on the data.

Example: Return all documents in the table 'marvel' of the default database.

```
r.table('marvel').run(conn, callback)
```

Example: Return all documents in the table 'marvel' of the database 'heroes'.

```
r.db('heroes').table('marvel').run(conn, callback)
```

Example: If you are OK with potentially out of date data from this table and want potentially faster reads, pass a flag allowing out of date data.

```
r.db('heroes').table('marvel', {useOutdated: true}).run(conn, callback)
```

Command syntax

time.minutes() → number

Description

Return the minute in a time object as a number between 0 and 59.

Example: Return all the posts submitted during the first 10 minutes of every hour.

```
r.table("posts").filter(function(post) {
    return post("date").minutes().lt(10)
})
```

Command syntax

sequence.map(mappingFunction) → stream

array.map(mappingFunction) → array

Description

Transform each element of the sequence by applying the given mapping function.

Example: Construct a sequence of hero power ratings.

```
r.table('marvel').map(function(hero) {
    return hero('combatPower').add(hero('compassionPower').mul(2))
}).run(conn, callback)
```

Command syntax

db.tableCreate(tableName[, options]) → object

Description

Create a table. A RethinkDB table is a collection of JSON documents.

If successful, the operation returns an object: `{created: 1}`. If a table with the same name already exists, the operation throws `RqlRuntimeError`.

Note: that you can only use alphanumeric characters and underscores for the table name.

When creating a table you can specify the following options:

- `primaryKey`: the name of the primary key. The default primary key is id;
- `durability`: if set to `soft`, this enables *soft durability* on this table: writes will be acknowledged by the server immediately and flushed to disk in the background. Default is `hard` (acknowledgement of writes happens after data has been written to disk);
- `cacheSize`: set the cache size (in bytes) to be used by the table. The default is 1073741824 (1024MB);
- `datacenter`: the name of the datacenter this table should be assigned to.

Example: Create a table named 'dc_universe' with the default settings.

```
r.db('test').tableCreate('dc_universe').run(conn, callback)
```

Example: Create a table named 'dc_universe' using the field 'name' as primary key.

```
r.db('test').tableCreate('dc_universe', {primaryKey: 'name'}).run(conn, callback)
```

Example: Create a table to log the very fast actions of the heroes.

```
r.db('test').tableCreate('dc_universe', {hardDurability: false}).run(conn, callback)
```

Command syntax

sequence.forEach(write_query) → object

Description

Loop over a sequence, evaluating the given write query for each element.

Example: Now that our heroes have defeated their villains, we can safely remove them from the villain table.

```
r.table('marvel').forEach(function(hero) {
   return r.table('villains').get(hero('villainDefeated')).delete()
}).run(conn, callback)
```

Command syntax

table.delete([{durability: “hard”, returnVals: false}]) → object

selection.delete([{durability: “hard”, returnVals: false}]) → object

singleSelection.delete([{durability: “hard”, returnVals: false}]) → object

Description

Delete one or more documents from a table.

The optional arguments are:

- `durability`: possible values are `hard` and `soft`. This option will override the table or query's durability setting (set in run).
  In soft durability mode RethinkDB will acknowledge the write immediately after receiving it, but before the write has been committed to disk.
- `returnVals`: if set to `true` and in case of a single document deletion, the deleted document will be returned.

Delete returns an object that contains the following attributes:

- `deleted`: the number of documents that were deleted.
- `skipped`: the number of documents that were skipped.
  For example, if you attempt to delete a batch of documents, and another concurrent query deletes some of those documents first, they will be counted as skipped.
- `errors`: the number of errors encountered while performing the delete.
- `first_error`: If errors were encountered, contains the text of the first error.
- `inserted`, `replaced`, and `unchanged`: all 0 for a delete operation..
- `old_val`: if `returnVals` is set to `true`, contains the deleted document.
- `new_val`: if `returnVals` is set to `true`, contains `null`.

Example: Delete a single document from the table `comments`.

```
r.table("comments").get("7eab9e63-73f1-4f33-8ce4-95cbea626f59").delete().run(conn, callba
```

Example: Delete all documents from the table `comments`.

```
r.table("comments").delete().run(conn, callback)
```

Example: Delete all comments where the field `idPost` is 3.

```
r.table("comments").filter({idPost: 3}).delete().run(conn, callback)
```

Example: Delete a single document from the table `comments` and return its value.

```
r.table("comments").get("7eab9e63-73f1-4f33-8ce4-95cbea626f59").delete({returnVals: true}
```

The result look like:

```
{
    deleted: 1,
    errors: 0,
    inserted: 0,
    new_val: null,
    old_val: {
        id: "7eab9e63-73f1-4f33-8ce4-95cbea626f59",
        author: "William",
        comment: "Great post",
        idPost: 3
    },
    replaced: 0,
    skipped: 0,
    unchanged: 0
}
```

Example: Delete all documents from the table `comments` without waiting for the operation to be flushed to disk.

```
r.table("comments").delete({durability: "soft"}).run(conn, callback)
```

Command syntax

table.get(key) → singleRowSelection

Description

Get a document by primary key.

Example: Find a document with the primary key 'superman'.

```
r.table('marvel').get('superman').run(conn, callback)
```

Command syntax

cursor.hasNext() → bool

array.hasNext() → bool

Description

Check if there are more elements in the cursor.

Example: Are there more elements in the cursor?

```
var hasMore = cursor.hasNext();
```

Example: Retrieve all the elements of a cursor using the `next` and `hasNext` commands and recursion.

```
query.run( conn, function(err, cursor) {
    if (err) throw err;

    var fetchNext = function(err, result) {
        if (err) throw err;
        if (cursor.hasNext()) {
```

```
            processRow(result);
            cursor.next(fetchNext);
        }
     // If you use one connection per query, the connection should be closed.
        // else { conn.close() }
    }

    if (cursor.hasNext()) {
        cursor.next(fetchNext);
    }
   // If you use one connection per query, the connection should be closed.
    // else { conn.close() }
})
```

Command syntax

time.timeOfDay() → number

Description

Return the number of seconds elapsed since the beginning of the day stored in the time object.

Example: Retrieve posts that were submitted before noon.

```
r.table("posts").filter(
    r.row("date").timeOfDay().le(12*60*60)
).run(conn, callback)
```

Command syntax

sequence.indexesOf(datum | predicate) → array

Description

Get the indexes of an element in a sequence. If the argument is a predicate, get the indexes of all elements matching it.

Example: Find the position of the letter 'c'.

```
r.expr(['a','b','c']).indexesOf('c').run(conn, callback)
```

Example: Find the popularity ranking of invisible heroes.

```
r.table('marvel').union(r.table('dc')).orderBy('popularity').indexesOf(
    r.row('superpowers').contains('invisibility')
).run(conn, callback)
```

Command syntax

string.match(regexp) → array

Description

Match against a regular expression. Returns a match object containing the matched string, that string's start/end position, and the capture groups. Accepts RE2 syntax (https://code.google.com/p/re2/wiki/Syntax). You can enable case-insensitive matching by prefixing the regular expression with `(?i)`. (See linked RE2 documentation for more flags.)

Example: Get all users whose name starts with A.

```
r.table('users').filter(function(row){return row('name').match("^A")}).run(conn, callback
```

Example: Parse out a name (returns "mlucy").

```
r.expr('id:0,name:mlucy,foo:bar').match('name:(\w+)')('groups').nth(0)('str').run(conn, c
```

Example: Fail to parse out a name (returns null).

```
r.expr('id:0,foo:bar').match('name:(\w+)')('groups').nth(0)('str').run(conn, callback)
```

Command syntax

sequence.count([filter]) → number

Description

Count the number of elements in the sequence. With a single argument, count the number of elements equal to it. If the argument is a function, it is equivalent to calling filter before count.

Example: Just how many super heroes are there?

```
r.table('marvel').count().add(r.table('dc').count()).run(conn, callback)
```

Example: Just how many super heroes have invisibility?

```
r.table('marvel').concatMap(r.row('superpowers')).count('invisibility').run(conn, callbac
```

Example: Just how many super heroes have defeated the Sphinx?

```
r.table('marvel').count(r.row('monstersKilled').contains('Sphinx')).run(conn, callback)
```

Command syntax

array.setIntersection(array) → array

Description

Intersect two arrays returning values that occur in both of them as a set (an array with distinct values).

Example: Check which pieces of equipment Iron Man has from a fixed list.

```
r.table('marvel').get('IronMan')('equipment').setIntersection(['newBoots', 'arc_reactor']
```

Command syntax

sequence.eqJoin(leftAttr, otherTable[, {index:'id'}]) → stream

array.eqJoin(leftAttr, otherTable[, {index:'id'}]) → array

Description

An efficient join that looks up elements in the right table by primary key.

Example: Let our heroes join forces to battle evil!

```
r.table('marvel').eqJoin('main_dc_collaborator', r.table('dc')).run(conn, callback)
```

Example: The above query is equivalent to this inner join but runs in O(n log(m)) time rather than the O(n * m) time the inner join takes.

```
r.table('marvel').innerJoin(r.table('dc'), function(left, right) {
    return left('main_dc_collaborator').eq(right('hero_name'));
}).run(conn, callback)
```

Example: You can take advantage of a secondary index on the second table by giving an optional index parameter.

```
r.table('marvel').eqJoin('main_weapon_origin',
r.table('mythical_weapons'), {index:'origin'}).run(conn, callback)
```

Example: You can pass a function instead of an attribute to join on more complicated expressions. Here we join to the DC universe collaborator with whom the hero has the most appearances.

```
r.table('marvel').eqJoin(function (doc) { return doc('dcCollaborators').orderBy('appearanc
r.table('dc')).run(conn, callback)
```

Command syntax

sequence.sample(number) → selection

stream.sample(number) → array

array.sample(number) → array

Description

Select a given number of elements from a sequence with uniform random distribution. Selection is done without replacement.

Example: Select 3 random heroes.

```
r.table('marvel').sample(3).run(conn, callback)
```

Command syntax

r → r

Description

The top-level ReQL namespace.

Example: Set up your top-level namespace.

```
var r = require('rethinkdb');
```

Command syntax

singleSelection.merge(object) → object

object.merge(object) → object

sequence.merge(object) → stream

array.merge(object) → array

Description

Merge two objects together to construct a new object with properties from both. Gives preference to attributes from other when there is a conflict.

Example: Equip IronMan for battle.

```
r.table('marvel').get('IronMan').merge(
    r.table('loadouts').get('alienInvasionKit')
).run(conn, callback)
```

Example: Merge can be used recursively to modify object within objects.

```
r.expr({weapons : {spectacular_graviton_beam : {dmg : 10, cooldown : 20}}}).merge(
   {weapons : {spectacular_graviton_beam : {dmg : 10}}}).run(conn, callback)
```

Example: To replace a nested object with another object you can use the literal keyword.

```
r.expr({weapons : {spectacular_graviton_beam : {dmg : 10, cooldown : 20}}}).merge(
  {weapons : r.literal({repulsor_rays : {dmg : 3, cooldown : 0}})}).run(conn, callback)
```

Example: Literal can be used to remove keys from an object as well.

```
r.expr({weapons : {spectacular_graviton_beam : {dmg : 10, cooldown : 20}}}).merge(
  {weapons : {spectacular_graviton_beam : r.literal()}}).run(conn, callback)
```

Command syntax

r.ISO8601(iso8601Date[, {default_timezone:”}]) → time

Description

Create a time object based on an iso8601 date-time string (e.g. ‘2013-01-01T01:01:01+00:00’). We support all valid ISO 8601 formats except for week dates. If you pass an ISO 8601 date-time without a time zone, you must specify the time zone with the optarg `default_timezone`. Read more about the ISO 8601 format on the Wikipedia page.

Example: Update the time of John’s birth.

```
r.table("user").get("John").update({birth: r.ISO8601('1986-11-03T08:30:00-07:00')}).run(c
```

Command syntax

sequence.groupBy(selector1[, selector2...], reductionObject) → array

Description

Groups elements by the values of the given attributes and then applies the given reduction. Though similar to `groupedMapReduce`, `groupBy` takes a standardized object for specifying the reduction. Can be used with a number of predefined common reductions.

Example: Using a predefined reduction we can easily find the average strength of members of each weight class.

```
r.table('marvel').groupBy('weightClass', r.avg('strength')).run(conn, callback)
```

Example: Groupings can also be specified on nested attributes.

```
r.table('marvel').groupBy({'abilities' : {'primary' : true}}, r.avg('strength')).run(conn,
```

Example: The nested syntax can quickly become verbose so there's a shortcut.

```
r.table('marvel').groupBy({'abilities' : 'primary'}, r.avg('strength')).run(conn, callback
```

Command syntax

conn.close([opts,]callback)

Description

Close an open connection. Accepts the following options:

- `noreplyWait`: whether to wait for noreply writes to complete before closing (default `true`). If this is set to `false`, some outstanding noreply writes may be aborted.

Closing a connection waits until all outstanding requests have finished and then frees any open resources associated with the connection. If `noreplyWait` is set to `false`, all outstanding requests are canceled immediately.

Example: Close an open connection, waiting for noreply writes to finish.

```
conn.close(function(err) { if (err) throw err; })
```

Example: Close an open connection immediately.

```
conn.close({noreplyWait: false}, function(err) { if (err) throw err; })
```

Command syntax

r.json(json_string) → value

Description

Parse a JSON string on the server.

Example: Send an array to the server'

```
r.json("[1,2,3]").run(conn, callback)
```

Command syntax

value.lt(value) → bool

Description

Test if the first value is less than other.

Example: Is 2 less than 2?

```
r.expr(2).lt(2).run(conn, callback)
```

Command syntax

r.count

Description

Count the total size of the group.

Example: Just how many heroes do we have at each strength level?

```
r.table('marvel').groupBy('strength', r.count).run(conn, callback)
```

Command syntax

time.date() → time

Description

Return a new time object only based on the day, month and year (ie. the same day at 00:00).

Example: Retrieve all the users whose birthday is today

```
r.table("users").filter(function(user) {
    return user("birthdate").date().eq(r.now().date())
}).run(conn, callback)
```

Command syntax

bool.or(bool) → bool

Description

Compute the logical or of two values.

Example: True or false ored is true?

```
r.expr(true).or(false).run(conn, callback)
```

Command syntax

r.now() → time

Description

Return a time object representing the current time in UTC. The command now() is computed once when the server receives the query, so multiple instances of r.now() will always return the same time inside a query.

Example: Add a new user with the time at which he subscribed.

```
r.table("users").insert({
    name: "John",
    subscription_date: r.now()
}).run(conn, callback)
```

Command syntax

array.spliceAt(index, array) → array

Description

Insert several values in to an array at a given index. Returns the modified array.

Example: Hulk and Thor decide to join the avengers.

```
r.expr(["Iron Man", "Spider-Man"]).spliceAt(1, ["Hulk", "Thor"]).run(conn, callback)
```

Command syntax

time.hours() → number

Description

Return the hour in a time object as a number between 0 and 23.

Example: Return all the posts submitted after midnight and before 4am.

```
r.table("posts").filter(function(post) {
    return post("date").hours().lt(4)
})
```

Command syntax

array.changeAt(index, value) → array

Description

Change a value in an array at a given index. Returns the modified array.

Example: Bruce Banner hulks out.

```
r.expr(["Iron Man", "Bruce", "Spider-Man"]).changeAt(1, "Hulk").run(conn, callback)
```

Command syntax

value.ge(value) → bool

Description

Test if the first value is greater than or equal to other.

Example: Is 2 greater than or equal to 2?

```
r.expr(2).ge(2).run(conn, callback)
```

Command syntax

array.append(value) → array

Description

Append a value to an array.

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')('equipment').append('newBoots').run(conn, callback)
```

Command syntax

conn.reconnect([opts,]callback)

Description

Close and reopen a connection. Accepts the following options:

- `noreplyWait`: whether to wait for noreply writes to complete before closing (default `true`). If this is set to `false`, some outstanding noreply writes may be aborted.

Closing a connection waits until all outstanding requests have finished. If `noreplyWait` is set to `false`, all outstanding requests are canceled immediately.

Example: Cancel outstanding requests/queries that are no longer needed.

```
conn.reconnect({noreplyWait: false}, function(errror, connection) { ... })
```

Command syntax

array.setDifference(array) → array

Description

Remove the elements of one array from another and return them as a set (an array with distinct values).

Example: Check which pieces of equipment Iron Man has, excluding a fixed list.

```
r.table('marvel').get('IronMan')('equipment').setDifference(['newBoots', 'arc_reactor']).
```

Command syntax

cursor.next(callback)

array.next(callback)

Description

Get the next element in the cursor.

Example: Let's grab the next element!

```
cursor.next(function(err, row) {
    if (err) throw err;
    processRow(row);
});
```

Note: The canonical way to retrieve all the results is to use each or toArray. The `next` command should be used only when you may not retrieve all the elements of a cursor or want to delay some operations.

Example: You can retrieve all the elements of a cursor with the `next` command using recursion.

```
query.run( conn, function(err, cursor) {
    if (err) throw err;

    var fetchNext = function(err, result) {
        if (err) throw err;
        if (cursor.hasNext()) {
            processRow(result);
            cursor.next(fetchNext);
        }
     // If you use one connection per query, the connection should be closed.
        // else { conn.close() }
    }

    if (cursor.hasNext()) {
        cursor.next(fetchNext);
    }
   // If you use one connection per query, the connection should be closed.
    // else { conn.close() }
})
```

Example: Another equivalent way to retrieve all the documents is to catch a `RqlDriverError`.

```
query.run( conn, function(err, cursor) {
    if (err) throw err;

    var fetchNext = function(err, result) {
        if (err) {
        if (((err.name === "RqlDriverError") && err.message === "No more rows in the cursor.
                console.log("No more data to process")
           // If you use one connection per query, the connection should be closed here.
                // conn.close()
            }
            else {
                throw err;
            }
        }
        else {
            processRow(result);
```

```
            cursor.next(fetchNext);
        }
    }
    cursor.next(fetchNext);
})
```

Example: With `next`, not all results have to be retrieved from a cursor – to stop retrieving results, break out of the recursive function. For example, this recursive function will stop retrieving results when the `checkRow` function returns true:

```
query.run( conn, function(err, cursor) {
    if (err) throw err;

    var fetchNext = function(err, result) {
        if (err) throw err;

        processRow(result);

        if (checkRow(result)) {
            if (cursor.hasNext()) {
                cursor.next(fetchNext);
            }
        // If you use one connection per query, the connection should be closed.
            // else { conn.close() }
        }
        else {
            cursor.close()
        // If you use one connection per query, the connection should be closed here.
            // else { conn.close() }
        }
    }

    if (cursor.hasNext()) {
        cursor.next(fetchNext);
    }
   // If you use one connection per query, the connection should be closed.
    // else { conn.close() }

})
```

Command syntax

any.info() → object

Description

Get information about a ReQL value.

Example: Get information about a table such as primary key, or cache size.

```
r.table('marvel').info().run(conn, callback)
```

Command syntax

array.setUnion(array) → array

Description

Add a several values to an array and return it as a set (an array with distinct values).

Example: Retrieve Iron Man's equipment list with the addition of some new boots and an arc reactor.

```
r.table('marvel').get('IronMan')('equipment').setUnion(['newBoots', 'arc_reactor']).run(c
```

Command syntax

r.connect(options, callback)

r.connect(host, callback)

Description

Create a new connection to the database server. Accepts the following options:

- `host`: the host to connect to (default `localhost`).
- `port`: the port to connect on (default `28015`).
- `db`: the default database (default `test`).
- `authKey`: the authentication key (default none).

If the connection cannot be established, a `RqlDriverError` will be passed to the callback instead of a connection.

Example: Opens a new connection to the database.

```
r.connect({host:'localhost', port:28015, db:'marvel', authKey:'hunter2'},
          function(err, conn) { ... })
```

Command syntax

r.error(message) → error

Description

Throw a runtime error. If called with no arguments inside the second argument to `default`, re-throw the current error.

Example: Iron Man can't possibly have lost a battle:

```
r.table('marvel').get('IronMan').do(function(ironman) {
    return r.branch(ironman('victories').lt(ironman('battles')),
        r.error('impossible code path'),
        ironman)
}).run(conn, callback)
```

Command syntax

sequence.groupedMapReduce(grouping, mapping, reduction, base) → value

Description

Partition the sequence into groups based on the `grouping` function. The elements of each group are then mapped using the `mapping` function and reduced using the `reduction` function.

`grouped_map_reduce` is a generalized form of group by.

Example: It's only fair that heroes be compared against their weight class.

```
r.table('marvel').groupedMapReduce(
    function(hero) { return hero('weightClass')},  // grouping
   function(hero) { return hero.pluck('name', 'strength')}, // mapping
    function(acc, hero) {  // reduction
      return r.branch(acc('strength').lt(hero('strength')), hero, acc)
    },
    {name:'none', strength:0} // reduction base
).run(conn, callback)
```

Command syntax

sequence.isEmpty() → bool

Description

Test if a sequence is empty.

Example: Are there any documents in the marvel table?

```
r.table('marvel').isEmpty().run(conn, callback)
```

Command syntax

sequence.limit(n) → stream

array.limit(n) → array

Description

End the sequence after the given number of elements.

Example: Only so many can fit in our Pantheon of heroes.

```
r.table('marvel').orderBy('belovedness').limit(10).run(conn, callback)
```

Command syntax

db.tableDrop(tableName) → object

Description

Drop a table. The table and all its data will be deleted.

If succesful, the operation returns an object: {dropped: 1}. If the specified table doesn't exist a `RqlRuntimeError` is thrown.

Example: Drop a table named 'dc_universe'.

```
r.db('test').tableDrop('dc_universe').run(conn, callback)
```

Command syntax

value.ne(value) → bool

Description

Test if two values are not equal.

Example: Does 2 not equal 2?

```
r.expr(2).ne(2).run(conn, callback)
```

Command syntax

sequence.nth(index) → object

Description

Get the nth element of a sequence.

Example: Select the second element in the array.

```
r.expr([1,2,3]).nth(1).run(conn, callback)
```

Command syntax

r.js(jsString) → value

Description

Create a javascript expression.

Example: Concatenate two strings using Javascript'

```
r.js("'str1' + 'str2'").run(conn, callback)
```

Example: Select all documents where the 'magazines' field is greater than 5 by running Javascript on the server.

```
r.table('marvel').filter(
    r.js('(function (row) { return row.magazines > 5; })')
).run(conn, callback)
```

Example: You may also specify a timeout in seconds (defaults to 5).

```
r.js('while(true) {}', {timeout:1.3}).run(conn, callback)
```

Command syntax

table.update(json | expr[, {durability: “hard”, returnVals: false, nonAtomic: false}]) → object

selection.update(json | expr[, {durability: “hard”, returnVals: false, nonAtomic: false}]) → object

singleSelection.update(json | expr[, {durability: “hard”, returnVals: false, nonAtomic: false}]) → object

Description

Update JSON documents in a table. Accepts a JSON document, a ReQL expression, or a combination of the two.

The optional arguments are:

- `durability`: possible values are `hard` and `soft`. This option will override the table or query's durability setting (set in run).
  In soft durability mode RethinkDB will acknowledge the write immediately after receiving it, but before the write has been committed to disk.
- `returnVals`: if set to `true` and in case of a single update, the updated document will be returned.
- `nonAtomic`: set to `true` if you want to perform non-atomic updates (updates that require fetching data from another document).

Update returns an object that contains the following attributes:

- `replaced`: the number of documents that were updated.
- `unchanged`: the number of documents that would have been modified except the new value was the same as the old value.
- `skipped`: the number of documents that were skipped because the document didn't exist.

- `errors`: the number of errors encountered while performing the update.
- `first_error`: If errors were encountered, contains the text of the first error.
- `deleted` and `inserted`: 0 for an update operation.
- `old_val`: if `returnVals` is set to `true`, contains the old document.
- `new_val`: if `returnVals` is set to `true`, contains the new document.

Example: Update the status of the post with `id` of `1` to `published`.

```
r.table("posts").get(1).update({status: "published"}).run(conn, callback)
```

Example: Update the status of all posts to `published`.

```
r.table("posts").update({status: "published"}).run(conn, callback)
```

Example: Update the status of all the post written by William.

```
r.table("posts").filter({author: "William"}).update({status: "published"}).run(conn, call
```

Example: Increment the field `view` with `id` of `1`. This query will throw an error if the field `views` doesn't exist.

```
r.table("posts").get(1).update({
    views: r.row("views").add(1)
}).run(conn, callback)
```

Example: Increment the field `view` of the post with `id` of `1`. If the field `views` does not exist, it will be set to `0`.

```
r.table("posts").update({
    views: r.row("views").add(1).default(0)
}).run(conn, callback)
```

Example: Perform a conditional update.
If the post has more than 100 views, set the `type` of a post to `hot`, else set it to `normal`.

```
r.table("posts").get(1).update(function(post) {
    return r.branch(
        post("views").gt(100),
        {type: "hot"},
        {type: "normal"}
    )
}).run(conn, callback)
```

Example: Update the field `numComments` with the result of a sub-query. Because this update is not atomic, you must pass the `nonAtomic` flag.

```
r.table("posts").get(1).update({
    numComments: r.table("comments").filter({idPost: 1}).count()
},{
    nonAtomic: true
}).run(conn, callback)
```

If you forget to specify the `nonAtomic` flag, you will get a `RqlRuntimeError`.

```
RqlRuntimeError: Could not prove function deterministic.  Maybe you want to use the non_atomi
```

Example: Update the field `numComments` with a random value between 0 and 100.
This update cannot be proven deterministic because of `r.js` (and in fact is not), so you must pass the `nonAtomic` flag.

```
r.table("posts").get(1).update({
    num_comments: r.js("Math.floor(Math.random()*100)")
},{
    nonAtomic: true
}).run(conn, callback)
```

Example: Update the status of the post with `id` of `1` using soft durability.

```
r.table("posts").get(1).update({status: "published"}, {durability: "soft"}).run(conn, call
```

Example: Increment the field `views` and return the values of the document before and after the update operation.

```
r.table("posts").get(1).update({
    views: r.row("views").add(1)
}, {
    returnVals: true
}).run(conn, callback)
```

The result will have two fields `old_val` and `new_val`.

```
{
    deleted: 1,
    errors: 0,
    inserted: 0,
```

```
    new_val: {
        id: 1,
        author: "Julius_Caesar",
        title: "Commentarii de Bello Gallico",
        content: "Aleas jacta est",
        views: 207
    },
    old_val: {
        id: 1,
        author: "Julius_Caesar",
        title: "Commentarii de Bello Gallico",
        content: "Aleas jacta est",
        views: 206
    },
    replaced: 0,
    skipped: 0,
    unchanged: 0
}
```

Accessing ReQL

All ReQL queries begin from the top-level module.

r

r → r

The top-level ReQL namespace.

Example: Set up your top-level namespace.

```
var r = require('rethinkdb');
```

connect

r.connect(options, callback)

r.connect(host, callback)

Create a new connection to the database server. Accepts the following options:

- `host`: the host to connect to (default `localhost`).
- `port`: the port to connect on (default `28015`).
- `db`: the default database (default `test`).

- `authKey`: the authentication key (default none).

If the connection cannot be established, a `RqlDriverError` will be passed to the callback instead of a connection.

Example: Opens a new connection to the database.

```
r.connect({host:'localhost', port:28015, db:'marvel', authKey:'hunter2'},
          function(err, conn) { ... })
```

close

conn.close([opts,]callback)

Close an open connection. Accepts the following options:

- `noreplyWait`: whether to wait for noreply writes to complete before closing (default `true`). If this is set to `false`, some outstanding noreply writes may be aborted.

Closing a connection waits until all outstanding requests have finished and then frees any open resources associated with the connection. If `noreplyWait` is set to `false`, all outstanding requests are canceled immediately.

Example: Close an open connection, waiting for noreply writes to finish.

```
conn.close(function(err) { if (err) throw err; })
```

Example: Close an open connection immediately.

```
conn.close({noreplyWait: false}, function(err) { if (err) throw err; })
```

reconnect

conn.reconnect([opts,]callback)

Close and reopen a connection. Accepts the following options:

- `noreplyWait`: whether to wait for noreply writes to complete before closing (default `true`). If this is set to `false`, some outstanding noreply writes may be aborted.

Closing a connection waits until all outstanding requests have finished. If `noreplyWait` is set to `false`, all outstanding requests are canceled immediately.

Example: Cancel outstanding requests/queries that are no longer needed.

```
conn.reconnect({noreplyWait: false}, function(errror, connection) { ... })
```

use

conn.use(dbName)

Change the default database on this connection.

Example: Change the default database so that we don't need to specify the database when referencing a table.

```
conn.use('marvel')
r.table('heroes').run(conn, ...) // refers to r.db('marvel').table('heroes')
```

run

query.run(conn, callback)

query.run(options[, callback])

Run a query on a connection. Accepts the following options:

- `useOutdated`: whether or not outdated reads are OK (default: `false`).
- `timeFormat`: what format to return times in (default: `'native'`). Set this to `'raw'` if you want times returned as JSON objects for exporting.
- `profile`: whether or not to return a profile of the query's execution (default: `false`).

The callback will get either an error, a single JSON result, or a cursor, depending on the query.

Example: Run a query on the connection `conn` and log each row in the result to the console.

```
r.table('marvel').run(conn, function(err, cursor) { cursor.each(console.log); })
```

Read more about this command →

noreplyWait

conn.noreplyWait(callback)

`noreplyWait` ensures that previous queries with the `noreply` flag have been processed by the server. Note that this guarantee only applies to queries run on the given connection.

Example: We have previously run queries with the `noreply` argument set to `true`. Now wait until the server has processed them.

```
conn.noreplyWait(function(err) { ... })
```

next

cursor.next(callback)

array.next(callback)

Get the next element in the cursor.

Example: Let's grab the next element!

```
cursor.next(function(err, row) {
    if (err) throw err;
    processRow(row);
});
```

Read more about this command →

hasNext

cursor.hasNext() → bool

array.hasNext() → bool

Check if there are more elements in the cursor.

Example: Are there more elements in the cursor?

```
var hasMore = cursor.hasNext();
```

Read more about this command →

each

cursor.each(callback[, onFinishedCallback])

array.each(callback[, onFinishedCallback])

Lazily iterate over the result set one element at a time.

Example: Let's process all the elements!

```
cursor.each(function(err, row) {
    if (err) throw err;
    processRow(row);
});
```

Read more about this command →

toArray

cursor.toArray(callback)

array.toArray(callback)

Retrieve all results and pass them as an array to the given callback.

Example: For small result sets it may be more convenient to process them at once as an array.

```
cursor.toArray(function(err, results) {
    if (err) throw err;
    processResults(results);
});
```

Read more about this command →

close (cursor)

cursor.close()

Close a cursor. Closing a cursor cancels the corresponding query and frees the memory associated with the open request.

Example: Close a cursor.

```
cursor.close()
```

addListener

conn.addListener(event, listener)

The connection object also supports the event emitter interface so you can listen for changes in connection state.

Example: Monitor connection state with events 'connect', 'close', and 'error'.

```
r.connect({}, function(err, conn) {
    if (err) throw err;

    conn.addListener('error', function(e) {
        processNetworkError(e);
    });

    conn.addListener('close', function() {
```

```
        cleanup();
    });

    runQueries(conn);
});
```

Read more about this command →

Manipulating databases

dbCreate

r.dbCreate(dbName) → object

Create a database. A RethinkDB database is a collection of tables, similar to relational databases.

If successful, the operation returns an object: `{created: 1}`. If a database with the same name already exists the operation throws `RqlRuntimeError`.

Note: that you can only use alphanumeric characters and underscores for the database name.

Example: Create a database named 'superheroes'.

```
r.dbCreate('superheroes').run(conn, callback)
```

dbDrop

r.dbDrop(dbName) → object

Drop a database. The database, all its tables, and corresponding data will be deleted.

If successful, the operation returns the object `{dropped: 1}`. If the specified database doesn't exist a `RqlRuntimeError` is thrown.

Example: Drop a database named 'superheroes'.

```
r.dbDrop('superheroes').run(conn, callback)
```

dbList

r.dbList() → array

List all database names in the system. The result is a list of strings.

Example: List all databases.

```
r.dbList().run(conn, callback)
```

Manipulating tables

tableCreate

db.tableCreate(tableName[, options]) → object

Create a table. A RethinkDB table is a collection of JSON documents.

If successful, the operation returns an object: `{created: 1}`. If a table with the same name already exists, the operation throws `RqlRuntimeError`.

Note: that you can only use alphanumeric characters and underscores for the table name.

When creating a table you can specify the following options:

- `primaryKey`: the name of the primary key. The default primary key is id;
- `durability`: if set to `soft`, this enables *soft durability* on this table: writes will be acknowledged by the server immediately and flushed to disk in the background. Default is `hard` (acknowledgement of writes happens after data has been written to disk);
- `cacheSize`: set the cache size (in bytes) to be used by the table. The default is 1073741824 (1024MB);
- `datacenter`: the name of the datacenter this table should be assigned to.

Example: Create a table named 'dc_universe' with the default settings.

```
r.db('test').tableCreate('dc_universe').run(conn, callback)
```

Read more about this command →

tableDrop

db.tableDrop(tableName) → object

Drop a table. The table and all its data will be deleted.

If succesful, the operation returns an object: {dropped: 1}. If the specified table doesn't exist a `RqlRuntimeError` is thrown.

Example: Drop a table named 'dc_universe'.

```
r.db('test').tableDrop('dc_universe').run(conn, callback)
```

tableList

db.tableList() → array

List all table names in a database. The result is a list of strings.

Example: List all tables of the 'test' database.

```
r.db('test').tableList().run(conn, callback)
```

indexCreate

table.indexCreate(indexName[, indexFunction]) → object

Create a new secondary index on this table.

Example: To efficiently query our heros by code name we have to create a secondary index.

```
r.table('dc').indexCreate('code_name').run(conn, callback)
```

Read more about this command →

indexDrop

table.indexDrop(indexName) → object

Delete a previously created secondary index of this table.

Example: Drop a secondary index named 'code_name'.

```
r.table('dc').indexDrop('code_name').run(conn, callback)
```

indexList

table.indexList() → array

List all the secondary indexes of this table.

Example: List the available secondary indexes for this table.

```
r.table('marvel').indexList().run(conn, callback)
```

indexStatus

table.indexStatus([, index...]) → array

Get the status of the specified indexes on this table, or the status of all indexes on this table if no indexes are specified.

Example: Get the status of all the indexes on `test`:

```
r.table('test').indexStatus().run(conn, callback)
```

Example: Get the status of the `timestamp` index:

```
r.table('test').indexStatus('timestamp').run(conn, callback)
```

indexWait

table.indexWait([, index...]) → array

Wait for the specified indexes on this table to be ready, or for all indexes on this table to be ready if no indexes are specified.

Example: Wait for all indexes on the table `test` to be ready:

```
r.table('test').indexWait().run(conn, callback)
```

Example: Wait for the index `timestamp` to be ready:

```
r.table('test').indexWait('timestamp').run(conn, callback)
```

Writing data

insert

table.insert(json | [json] [, {durability: “hard”, returnVals: false, upsert: false}]) → object

Insert JSON documents into a table. Accepts a single JSON document or an array of documents.

Example: Insert a document into the table `posts`.

```
r.table("posts").insert({
    id: 1,
    title: "Lorem ipsum",
    content: "Dolor sit amet"
}).run(conn, callback)
```

Read more about this command →

update

table.update(json | expr [, {durability: “hard”, returnVals: false, nonAtomic: false}]) → object

selection.update(json | expr [, {durability: “hard”, returnVals: false, nonAtomic: false}]) → object

singleSelection.update(json | expr [, {durability: “hard”, returnVals: false, nonAtomic: false}]) → object

Update JSON documents in a table. Accepts a JSON document, a ReQL expression, or a combination of the two. You can pass options like `returnVals` that will return the old and new values of the row you have modified.

Example: Update Superman’s age to 30. If attribute ‘age’ doesn’t exist, adds it to the document.

Example: Update the status of the post with `id` of `1` to `published`.

```
r.table("posts").get(1).update({status: "published"}).run(conn, callback)
```

Read more about this command →

replace

table.replace(json | expr [, {durability: “hard”, returnVals: false, nonAtomic: false}]) → object

selection.replace(json | expr [, {durability: “hard”, returnVals: false, nonAtomic: false}]) → object

singleSelection.replace(json | expr [, {durability: “hard”, returnVals: false, nonAtomic: false}]) → object

Replace documents in a table. Accepts a JSON document or a ReQL expression, and replaces the original document with the new one. The new document must have the same primary key as the original document.

Example: Replace the document with the primary key `1`.

```
r.table("posts").get(1).replace({
    id: 1,
    title: "Lorem ipsum",
    content: "Aleas jacta est",
    status: "draft"
}).run(conn, callback)
```

Read more about this command →

delete

table.delete([{durability: “hard”, returnVals: false}]) → object

selection.delete([{durability: “hard”, returnVals: false}]) → object

singleSelection.delete([{durability: “hard”, returnVals: false}]) → object

Delete one or more documents from a table.

Example: Delete a single document from the table `comments`.

```
r.table("comments").get("7eab9e63-73f1-4f33-8ce4-95cbea626f59").delete().run(conn, callba
```

Read more about this command →

sync

table.sync() → object

`sync` ensures that writes on a given table are written to permanent storage. Queries that specify soft durability (`{durability: 'soft'}`) do not give such

guarantees, so `sync` can be used to ensure the state of these queries. A call to `sync` does not return until all previous writes to the table are persisted.

Example: After having updated multiple heroes with soft durability, we now want to wait until these changes are persisted.

```
r.table('marvel').sync().run(conn, callback)
```

Selecting data

db

r.db(dbName) → db

Reference a database.

Example: Before we can query a table we have to select the correct database.

```
r.db('heroes').table('marvel').run(conn, callback)
```

table

db.table(name[, {useOutdated: false}]) → table

Select all documents in a table. This command can be chained with other commands to do further processing on the data.

Example: Return all documents in the table 'marvel' of the default database.

```
r.table('marvel').run(conn, callback)
```

Read more about this command →

get

table.get(key) → singleRowSelection

Get a document by primary key.

Example: Find a document with the primary key 'superman'.

```
r.table('marvel').get('superman').run(conn, callback)
```

getAll

table.getAll(key[, key2...], [, {index:'id'}]) → selection

Get all documents where the given value matches the value of the requested index.

Example: Secondary index keys are not guaranteed to be unique so we cannot query via "get" when using a secondary index.

```
r.table('marvel').getAll('man_of_steel', {index:'code_name'}).run(conn, callback)
```

Read more about this command →

between

table.between(lowerKey, upperKey [, {index:'id', left_bound:'closed', right_bound:'open'}]) → selection

Get all documents between two keys. Accepts three optional arguments: `index`, `left_bound`, and `right_bound`. If `index` is set to the name of a secondary index, `between` will return all documents where that index's value is in the specified range (it uses the primary key by default). `left_bound` or `right_bound` may be set to `open` or `closed` to indicate whether or not to include that endpoint of the range (by default, `left_bound` is closed and `right_bound` is open).

Example: Find all users with primary key >= 10 and < 20 (a normal half-open interval).

```
r.table('marvel').between(10, 20).run(conn, callback)
```

Read more about this command →

filter

sequence.filter(predicate[, {default: false}]) → selection

stream.filter(predicate[, {default: false}]) → stream

array.filter(predicate[, {default: false}]) → array

Get all the documents for which the given predicate is true.

`filter` can be called on a sequence, selection, or a field containing an array of elements. The return type is the same as the type on which the function was called on.

The body of every filter is wrapped in an implicit `.default(false)`, which means that if a non-existence errors is thrown (when you try to access a field that does not exist in a document), RethinkDB will just ignore the document. The `default` value can be changed by passing an object with a `default` field. Setting this optional argument to `r.error()` will cause any non-existence errors to return a `RqlRuntimeError`.

Example: Get all the users that are 30 years old.

```
r.table('users').filter({age: 30}).run(conn, callback)
```

Read more about this command →

Joins

These commands allow the combination of multiple sequences into a single sequence

innerJoin

sequence.innerJoin(otherSequence, predicate) → stream

array.innerJoin(otherSequence, predicate) → array

Returns the inner product of two sequences (e.g. a table, a filter result) filtered by the predicate. The query compares each row of the left sequence with each row of the right sequence to find all pairs of rows which satisfy the predicate. When the predicate is satisfied, each matched pair of rows of both sequences are combined into a result row.

Example: Construct a sequence of documents containing all cross-universe matchups where a marvel hero would lose.

```
r.table('marvel').innerJoin(r.table('dc'), function(marvelRow, dcRow) {
    return marvelRow('strength').lt(dcRow('strength'))
}).run(conn, callback)
```

outerJoin

sequence.outerJoin(otherSequence, predicate) → stream

array.outerJoin(otherSequence, predicate) → array

Computes a left outer join by retaining each row in the left table even if no match was found in the right table.

Example: Construct a sequence of documents containing all cross-universe matchups where a marvel hero would lose, but keep marvel heroes who would never lose a matchup in the sequence.

```
r.table('marvel').outerJoin(r.table('dc'), function(marvelRow, dcRow) {
    return marvelRow('strength').lt(dcRow('strength'))
}).run(conn, callback)
```

eqJoin

sequence.eqJoin(leftAttr, otherTable[, {index:'id'}]) → stream

array.eqJoin(leftAttr, otherTable[, {index:'id'}]) → array

An efficient join that looks up elements in the right table by primary key.

Example: Let our heroes join forces to battle evil!

```
r.table('marvel').eqJoin('main_dc_collaborator', r.table('dc')).run(conn, callback)
```

Read more about this command →

zip

stream.zip() → stream

array.zip() → array

Used to 'zip' up the result of a join by merging the 'right' fields into 'left' fields of each member of the sequence.

Example: 'zips up' the sequence by merging the left and right fields produced by a join.

```
r.table('marvel').eqJoin('main_dc_collaborator', r.table('dc'))
    .zip().run(conn, callback)
```

Transformations

These commands are used to transform data in a sequence.

map

sequence.map(mappingFunction) → stream

array.map(mappingFunction) → array

Transform each element of the sequence by applying the given mapping function.

Example: Construct a sequence of hero power ratings.

```
r.table('marvel').map(function(hero) {
    return hero('combatPower').add(hero('compassionPower').mul(2))
}).run(conn, callback)
```

withFields

sequence.withFields([selector1, selector2...]) → stream

array.withFields([selector1, selector2...]) → array

Takes a sequence of objects and a list of fields. If any objects in the sequence don't have all of the specified fields, they're dropped from the sequence. The remaining objects have the specified fields plucked out. (This is identical to `has_fields` followed by `pluck` on a sequence.)

Example: Get a list of heroes and their nemeses, excluding any heroes that lack one.

```
r.table('marvel').withFields('id', 'nemesis')
```

Read more about this command →

concatMap

sequence.concatMap(mappingFunction) → stream

array.concatMap(mappingFunction) → array

Flattens a sequence of arrays returned by the mappingFunction into a single sequence.

Example: Construct a sequence of all monsters defeated by Marvel heroes. Here the field 'defeatedMonsters' is a list that is concatenated to the sequence.

```
r.table('marvel').concatMap(function(hero) {
    return hero('defeatedMonsters')
}).run(conn, callback)
```

orderBy

table.orderBy([key1...], {index: index_name}) -> selection<stream>

selection.orderBy(key1, [key2...]) -> selection<array>

sequence.orderBy(key1, [key2...]) -> array

Sort the sequence by document values of the given key(s). `orderBy` defaults to ascending ordering. To explicitly specify the ordering, wrap the attribute with either `r.asc` or `r.desc`.

Example: Order our heroes by a series of performance metrics.

```
r.table('marvel').orderBy('enemiesVanquished', 'damselsSaved').run(conn, callback)
```

Read more about this command →

skip

sequence.skip(n) → stream

array.skip(n) → array

Skip a number of elements from the head of the sequence.

Example: Here in conjunction with `order_by` we choose to ignore the most successful heroes.

```
r.table('marvel').orderBy('successMetric').skip(10).run(conn, callback)
```

limit

sequence.limit(n) → stream

array.limit(n) → array

End the sequence after the given number of elements.

Example: Only so many can fit in our Pantheon of heroes.

```
r.table('marvel').orderBy('belovedness').limit(10).run(conn, callback)
```

slice

sequence.slice(startIndex[, endIndex]) → stream

array.slice(startIndex[, endIndex]) → array

Trim the sequence to within the bounds provided.

Example: For this fight, we need heroes with a good mix of strength and agility.

```
r.table('marvel').orderBy('strength').slice(5, 10).run(conn, callback)
```

nth

sequence.nth(index) → object

Get the nth element of a sequence.

Example: Select the second element in the array.

```
r.expr([1,2,3]).nth(1).run(conn, callback)
```

indexesOf

sequence.indexesOf(datum | predicate) → array

Get the indexes of an element in a sequence. If the argument is a predicate, get the indexes of all elements matching it.

Example: Find the position of the letter 'c'.

```
r.expr(['a','b','c']).indexesOf('c').run(conn, callback)
```

Read more about this command →

isEmpty

sequence.isEmpty() → bool

Test if a sequence is empty.

Example: Are there any documents in the marvel table?

```
r.table('marvel').isEmpty().run(conn, callback)
```

union

sequence.union(sequence) → array

Concatenate two sequences.

Example: Construct a stream of all heroes.

```
r.table('marvel').union(r.table('dc')).run(conn, callback)
```

sample

sequence.sample(number) → selection

stream.sample(number) → array

array.sample(number) → array

Select a given number of elements from a sequence with uniform random distribution. Selection is done without replacement.

Example: Select 3 random heroes.

```
r.table('marvel').sample(3).run(conn, callback)
```

Aggregation

These commands are used to compute smaller values from large sequences.

reduce

sequence.reduce(reductionFunction[, base]) → value

Produce a single value from a sequence through repeated application of a reduction function.

The reduce function gets invoked repeatedly not only for the input values but also for results of previous reduce invocations. The type and format of the object that is passed in to reduce must be the same with the one returned from reduce.

Example: How many enemies have our heroes defeated?

```
r.table('marvel').map(r.row('monstersKilled')).reduce(function(acc, val) {
    return acc.add(val)
}, 0).run(conn, callback)
```

count

sequence.count([filter]) → number

Count the number of elements in the sequence. With a single argument, count the number of elements equal to it. If the argument is a function, it is equivalent to calling filter before count.

Example: Just how many super heroes are there?

```
r.table('marvel').count().add(r.table('dc').count()).run(conn, callback)
```

Read more about this command →

distinct

sequence.distinct() → array

Remove duplicate elements from the sequence.

Example: Which unique villains have been vanquished by marvel heroes?

```
r.table('marvel').concatMap(function(hero) {return hero('villainList')}).distinct()
    .run(conn, callback)
```

groupedMapReduce

sequence.groupedMapReduce(grouping, mapping, reduction, base) → value

Partition the sequence into groups based on the `grouping` function. The elements of each group are then mapped using the `mapping` function and reduced using the `reduction` function.

`grouped_map_reduce` is a generalized form of group by.

Example: It's only fair that heroes be compared against their weight class.

```
r.table('marvel').groupedMapReduce(
    function(hero) { return hero('weightClass')},  // grouping
   function(hero) { return hero.pluck('name', 'strength')}, // mapping
    function(acc, hero) {  // reduction
      return r.branch(acc('strength').lt(hero('strength')), hero, acc)
    },
    {name:'none', strength:0} // reduction base
).run(conn, callback)
```

groupBy

sequence.groupBy(selector1[, selector2...], reductionObject) → array

Groups elements by the values of the given attributes and then applies the given reduction. Though similar to `groupedMapReduce`, `groupBy` takes a standardized object for specifying the reduction. Can be used with a number of predefined common reductions.

Example: Using a predefined reduction we can easily find the average strength of members of each weight class.

```
r.table('marvel').groupBy('weightClass', r.avg('strength')).run(conn, callback)
```

Read more about this command →

contains

sequence.contains(value1[, value2...]) → bool

Returns whether or not a sequence contains all the specified values, or if functions are provided instead, returns whether or not a sequence contains values matching all the specified functions.

Example: Has Iron Man ever fought Superman?

```
r.table('marvel').get('ironman')('opponents').contains('superman').run(conn, callback)
```

Read more about this command →

Aggregators

These standard aggregator objects are to be used in conjunction with groupBy.

count

r.count

Count the total size of the group.

Example: Just how many heroes do we have at each strength level?

```
r.table('marvel').groupBy('strength', r.count).run(conn, callback)
```

sum

r.sum(attr)

Compute the sum of the given field in the group.

Example: How many enemies have been vanquished by heroes at each strength level?

```
r.table('marvel').groupBy('strength', r.sum('enemiesVanquished')).run(conn, callback)
```

avg

r.avg(attr)

Compute the average value of the given attribute for the group.

Example: What's the average agility of heroes at each strength level?

```
r.table('marvel').groupBy('strength', r.avg('agility')).run(conn, callback)
```

Document manipulation

row

r.row → value

Returns the currently visited document.

Example: Get all users whose age is greater than 5.

```
r.table('users').filter(r.row('age').gt(5)).run(conn, callback)
```

Read more about this command →

pluck

sequence.pluck([selector1, selector2...]) → stream

array.pluck([selector1, selector2...]) → array

object.pluck([selector1, selector2...]) → object

singleSelection.pluck([selector1, selector2...]) → object

Plucks out one or more attributes from either an object or a sequence of objects (projection).

Example: We just need information about IronMan's reactor and not the rest of the document.

```
r.table('marvel').get('IronMan').pluck('reactorState', 'reactorPower').run(conn, callback
```

Read more about this command →

without

sequence.without([selector1, selector2...]) → stream

array.without([selector1, selector2...]) → array

singleSelection.without([selector1, selector2...]) → object

object.without([selector1, selector2...]) → object

The opposite of pluck; takes an object or a sequence of objects, and returns them with the specified paths removed.

Example: Since we don't need it for this computation we'll save bandwidth and leave out the list of IronMan's romantic conquests.

```
r.table('marvel').get('IronMan').without('personalVictoriesList').run(conn, callback)
```

Read more about this command →

merge

singleSelection.merge(object) → object

object.merge(object) → object

sequence.merge(object) → stream

array.merge(object) → array

Merge two objects together to construct a new object with properties from both. Gives preference to attributes from other when there is a conflict.

Example: Equip IronMan for battle.

```
r.table('marvel').get('IronMan').merge(
    r.table('loadouts').get('alienInvasionKit')
).run(conn, callback)
```

Read more about this command →

append

array.append(value) → array

Append a value to an array.

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')('equipment').append('newBoots').run(conn, callback)
```

prepend

array.prepend(value) → array

Prepend a value to an array.

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')('equipment').prepend('newBoots').run(conn, callback)
```

difference

array.difference(array) → array

Remove the elements of one array from another array.

Example: Retrieve Iron Man's equipment list without boots.

```
r.table('marvel').get('IronMan')('equipment').difference(['Boots']).run(conn, callback)
```

setInsert

array.setInsert(value) → array

Add a value to an array and return it as a set (an array with distinct values).

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')('equipment').setInsert('newBoots').run(conn, callback)
```

setUnion

array.setUnion(array) → array

Add a several values to an array and return it as a set (an array with distinct values).

Example: Retrieve Iron Man's equipment list with the addition of some new boots and an arc reactor.

```
r.table('marvel').get('IronMan')('equipment').setUnion(['newBoots', 'arc_reactor']).run(c
```

setIntersection

array.setIntersection(array) → array

Intersect two arrays returning values that occur in both of them as a set (an array with distinct values).

Example: Check which pieces of equipment Iron Man has from a fixed list.

```
r.table('marvel').get('IronMan')('equipment').setIntersection(['newBoots', 'arc_reactor']
```

setDifference

array.setDifference(array) → array

Remove the elements of one array from another and return them as a set (an array with distinct values).

Example: Check which pieces of equipment Iron Man has, excluding a fixed list.

```
r.table('marvel').get('IronMan')('equipment').setDifference(['newBoots', 'arc_reactor']).
```

()

sequence(attr) → sequence

singleSelection(attr) → value

object(attr) → value

Get a single field from an object. If called on a sequence, gets that field from every object in the sequence, skipping objects that lack it.

Example: What was Iron Man's first appearance in a comic?

```
r.table('marvel').get('IronMan')('firstAppearance').run(conn, callback)
```

hasFields

sequence.hasFields([selector1, selector2...]) → stream

array.hasFields([selector1, selector2...]) → array

singleSelection.hasFields([selector1, selector2...]) → boolean

object.hasFields([selector1, selector2...]) → boolean

Test if an object has all of the specified fields. An object has a field if it has the specified key and that key maps to a non-null value. For instance, the object `{'a':1,'b':2,'c':null}` has the fields `a` and `b`.

Example: Which heroes are married?

```
r.table('marvel').hasFields('spouse')
```

Read more about this command →

insertAt

array.insertAt(index, value) → array

Insert a value in to an array at a given index. Returns the modified array.

Example: Hulk decides to join the avengers.

```
r.expr(["Iron Man", "Spider-Man"]).insertAt(1, "Hulk").run(conn, callback)
```

spliceAt

array.spliceAt(index, array) → array

Insert several values in to an array at a given index. Returns the modified array.

Example: Hulk and Thor decide to join the avengers.

```
r.expr(["Iron Man", "Spider-Man"]).spliceAt(1, ["Hulk", "Thor"]).run(conn, callback)
```

deleteAt

array.deleteAt(index [,endIndex]) → array

Remove an element from an array at a given index. Returns the modified array.

Example: Hulk decides to leave the avengers.

```
r.expr(["Iron Man", "Hulk", "Spider-Man"]).deleteAt(1).run(conn, callback)
```

Read more about this command →

changeAt

array.changeAt(index, value) → array

Change a value in an array at a given index. Returns the modified array.

Example: Bruce Banner hulks out.

```
r.expr(["Iron Man", "Bruce", "Spider-Man"]).changeAt(1, "Hulk").run(conn, callback)
```

keys

singleSelection.keys() → array

object.keys() → array

Return an array containing all of the object's keys.

Example: Get all the keys of a row.

```
r.table('marvel').get('ironman').keys().run(conn, callback)
```

String manipulation

These commands provide string operators.

match

string.match(regexp) → array

Match against a regular expression. Returns a match object containing the matched string, that string's start/end position, and the capture groups. Accepts RE2 syntax (https://code.google.com/p/re2/wiki/Syntax). You can enable case-insensitive matching by prefixing the regular expression with `(?i)`. (See linked RE2 documentation for more flags.)

Example: Get all users whose name starts with A.

```
r.table('users').filter(function(row){return row('name').match("^A")}).run(conn, callback
```

Read more about this command →

Math and logic

add

number.add(number) → number

string.add(string) → string

array.add(array) → array

time.add(number) → time

Sum two numbers, concatenate two strings, or concatenate 2 arrays.

Example: It's as easy as 2 + 2 = 4.

```
r.expr(2).add(2).run(conn, callback)
```

Read more about this command →

sub

number.sub(number) → number

time.sub(time) → number

time.sub(number) → time

Subtract two numbers.

Example: It's as easy as 2 - 2 = 0.

```
r.expr(2).sub(2).run(conn, callback)
```

Read more about this command →

mul

number.mul(number) → number

array.mul(number) → array

Multiply two numbers, or make a periodic array.

Example: It's as easy as 2 * 2 = 4.

```
r.expr(2).mul(2).run(conn, callback)
```

Read more about this command →

div

number.div(number) → number

Divide two numbers.

Example: It's as easy as 2 / 2 = 1.

```
r.expr(2).div(2).run(conn, callback)
```

mod

number.mod(number) → number

Find the remainder when dividing two numbers.

Example: It's as easy as 2 % 2 = 0.

```
r.expr(2).mod(2).run(conn, callback)
```

and

bool.and(bool) → bool

Compute the logical and of two values.

Example: True and false anded is false?

```
r.expr(true).and(false).run(conn, callback)
```

or

bool.or(bool) → bool

Compute the logical or of two values.

Example: True or false ored is true?

```
r.expr(true).or(false).run(conn, callback)
```

eq

value.eq(value) → bool

Test if two values are equal.

Example: Does 2 equal 2?

```
r.expr(2).eq(2).run(conn, callback)
```

ne

value.ne(value) → bool

Test if two values are not equal.

Example: Does 2 not equal 2?

```
r.expr(2).ne(2).run(conn, callback)
```

gt

value.gt(value) → bool

Test if the first value is greater than other.

Example: Is 2 greater than 2?

```
r.expr(2).gt(2).run(conn, callback)
```

ge

value.ge(value) → bool

Test if the first value is greater than or equal to other.

Example: Is 2 greater than or equal to 2?

```
r.expr(2).ge(2).run(conn, callback)
```

lt

value.lt(value) → bool

Test if the first value is less than other.

Example: Is 2 less than 2?

```
r.expr(2).lt(2).run(conn, callback)
```

le

value.le(value) → bool

Test if the first value is less than or equal to other.

Example: Is 2 less than or equal to 2?

```
r.expr(2).le(2).run(conn, callback)
```

not

bool.not() → bool

Compute the logical inverse (not).

Example: Not true is false.

```
r.expr(true).not().run(conn, callback)
```

Dates and times

now

r.now() → time

Return a time object representing the current time in UTC. The command now() is computed once when the server receives the query, so multiple instances of r.now() will always return the same time inside a query.

Example: Add a new user with the time at which he subscribed.

```
r.table("users").insert({
    name: "John",
    subscription_date: r.now()
}).run(conn, callback)
```

time

r.time(year, month, day[, hour, minute, second], timezone) → time

Create a time object for a specific time.

A few restrictions exist on the arguments:

- `year` is an integer between 1400 and 9,999.
- `month` is an integer between 1 and 12.
- `day` is an integer between 1 and 31.
- `hour` is an integer.
- `minutes` is an integer.
- `seconds` is a double. Its value will be rounded to three decimal places (millisecond-precision).
- `timezone` can be `'Z'` (for UTC) or a string with the format `±[hh]:[mm]`.

Example: Update the birthdate of the user "John" to November 3rd, 1986 UTC.

```
r.table("user").get("John").update({birthdate: r.time(1986, 11, 3, 'Z')})
    .run(conn, callback)
```

epochTime

r.epochTime(epochTime) → time

Create a time object based on seconds since epoch. The first argument is a double and will be rounded to three decimal places (millisecond-precision).

Example: Update the birthdate of the user "John" to November 3rd, 1986.

```
r.table("user").get("John").update({birthdate: r.epochTime(531360000)})
    .run(conn, callback)
```

ISO8601

r.ISO8601(iso8601Date[, {default_timezone:"}]) → time

Create a time object based on an iso8601 date-time string (e.g. '2013-01-01T01:01:01+00:00'). We support all valid ISO 8601 formats except for week dates. If you pass an ISO 8601 date-time without a time zone, you must specify the time zone with the optarg `default_timezone`. Read more about the ISO 8601 format on the Wikipedia page.

Example: Update the time of John's birth.

```
r.table("user").get("John").update({birth: r.ISO8601('1986-11-03T08:30:00-07:00')}).run(c
```

inTimezone

time.inTimezone(timezone) → time

Return a new time object with a different timezone. While the time stays the same, the results returned by methods such as hours() will change since they take the timezone into account. The timezone argument has to be of the ISO 8601 format.

Example: Hour of the day in San Francisco (UTC/GMT -8, without daylight saving time).

```
r.now().inTimezone('-08:00').hours().run(conn, callback)
```

timezone

time.timezone() → string

Return the timezone of the time object.

Example: Return all the users in the "-07:00" timezone.

```
r.table("users").filter( function(user) {
    return user("subscriptionDate").timezone().eq("-07:00")
})
```

during

time.during(startTime, endTime[, options]) → bool

Return if a time is between two other times (by default, inclusive for the start, exclusive for the end).

Example: Retrieve all the posts that were posted between December 1st, 2013 (inclusive) and December 10th, 2013 (exclusive).

```
r.table("posts").filter(
    r.row('date').during(r.time(2013, 12, 1), r.time(2013, 12, 10))
).run(conn, callback)
```

Read more about this command →

date

time.date() → time

Return a new time object only based on the day, month and year (ie. the same day at 00:00).

Example: Retrieve all the users whose birthday is today

```
r.table("users").filter(function(user) {
    return user("birthdate").date().eq(r.now().date())
}).run(conn, callback)
```

timeOfDay

time.timeOfDay() → number

Return the number of seconds elapsed since the beginning of the day stored in the time object.

Example: Retrieve posts that were submitted before noon.

```
r.table("posts").filter(
    r.row("date").timeOfDay().le(12*60*60)
).run(conn, callback)
```

year

time.year() → number

Return the year of a time object.

Example: Retrieve all the users born in 1986.

```
r.table("users").filter(function(user) {
    return user("birthdate").year().eq(1986)
}).run(conn, callback)
```

month

time.month() → number

Return the month of a time object as a number between 1 and 12. For your convenience, the terms r.january, r.february etc. are defined and map to the appropriate integer.

Example: Retrieve all the users who were born in November.

```
r.table("users").filter(
    r.row("birthdate").month().eq(11)
)
```

Read more about this command →

day

time.day() → number

Return the day of a time object as a number between 1 and 31.

Example: Return the users born on the 24th of any month.

```
r.table("users").filter(
    r.row("birthdate").day().eq(24)
).run(conn, callback)
```

dayOfWeek

time.dayOfWeek() → number

Return the day of week of a time object as a number between 1 and 7 (following ISO 8601 standard). For your convenience, the terms r.monday, r.tuesday etc. are defined and map to the appropriate integer.

Example: Return today's day of week.

```
r.now().dayOfWeek().run(conn, callback)
```

Read more about this command →

dayOfYear

time.dayOfYear() → number

Return the day of the year of a time object as a number between 1 and 366 (following ISO 8601 standard).

Example: Retrieve all the users who were born the first day of a year.

```
r.table("users").filter(
    r.row("birthdate").dayOfYear().eq(1)
)
```

hours

time.hours() → number

Return the hour in a time object as a number between 0 and 23.

Example: Return all the posts submitted after midnight and before 4am.

```
r.table("posts").filter(function(post) {
    return post("date").hours().lt(4)
})
```

minutes

time.minutes() → number

Return the minute in a time object as a number between 0 and 59.

Example: Return all the posts submitted during the first 10 minutes of every hour.

```
r.table("posts").filter(function(post) {
    return post("date").minutes().lt(10)
})
```

seconds

time.seconds() → number

Return the seconds in a time object as a number between 0 and 59.999 (double precision).

Example: Return the post submitted during the first 30 seconds of every minute.

```
r.table("posts").filter(function(post) {
    return post("date").seconds().lt(30)
})
```

toISO8601

time.toISO8601() → string

Convert a time object to its iso 8601 format.

Example: Return the current time in an ISO8601 format.

```
r.now().toISO8601()
```

toEpochTime

time.toEpochTime() → number

Convert a time object to its epoch time.

Example: Return the current time in an ISO8601 format.

```
r.now().toEpochTime()
```

Control structures

do

any.do(arg [, args]*, expr) → any

Evaluate the expr in the context of one or more value bindings.

The type of the result is the type of the value returned from expr.

Example: The object(s) passed to do() can be bound to name(s). The last argument is the expression to evaluate in the context of the bindings.

```
r.do(r.table('marvel').get('IronMan'),
    function (ironman) { return ironman('name'); }
).run(conn, callback)
```

branch

r.branch(test, true_branch, false_branch) → any

If the `test` expression returns `false` or `null`, the `false_branch` will be evaluated. Otherwise, the `true_branch` will be evaluated.

The `branch` command is effectively an `if` renamed due to language constraints. The type of the result is determined by the type of the branch that gets executed.

Example: Return heroes and superheroes.

```
r.table('marvel').map(
    r.branch(
        r.row('victories').gt(100),
        r.row('name').add(' is a superhero'),
        r.row('name').add(' is a hero')
    )
).run(conn, callback)
```

forEach

sequence.forEach(write_query) → object

Loop over a sequence, evaluating the given write query for each element.

Example: Now that our heroes have defeated their villains, we can safely remove them from the villain table.

```
r.table('marvel').forEach(function(hero) {
   return r.table('villains').get(hero('villainDefeated')).delete()
}).run(conn, callback)
```

error

r.error(message) → error

Throw a runtime error. If called with no arguments inside the second argument to `default`, re-throw the current error.

Example: Iron Man can't possibly have lost a battle:

```
r.table('marvel').get('IronMan').do(function(ironman) {
    return r.branch(ironman('victories').lt(ironman('battles')),
        r.error('impossible code path'),
        ironman)
}).run(conn, callback)
```

default

value.default(default_value) → any

sequence.default(default_value) → any

Handle non-existence errors. Tries to evaluate and return its first argument. If an error related to the absence of a value is thrown in the process, or if its first argument returns `null`, returns its second argument. (Alternatively, the second argument may be a function which will be called with either the text of the non-existence error or `null`.)

Exmple: Suppose we want to retrieve the titles and authors of the table `posts`. In the case where the author field is missing or `null`, we want to retrieve the string `Anonymous`.

```
r.table("posts").map( function(post) {
    return {
        title: post("title"),
```

```
        author: post("author").default("Anonymous")
    }
}).run(conn, callback)
```

Read more about this command →

expr

r.expr(value) → value

Construct a ReQL JSON object from a native object.

Example: Objects wrapped with `expr` can then be manipulated by ReQL API functions.

```
r.expr({a:'b'}).merge({b:[1,2,3]}).run(conn, callback)
```

Read more about this command →

js

r.js(jsString) → value

Create a javascript expression.

Example: Concatenate two strings using Javascript'

```
r.js("'str1' + 'str2'").run(conn, callback)
```

Read more about this command →

coerceTo

sequence.coerceTo(typeName) → array

value.coerceTo(typeName) → string

array.coerceTo(typeName) → object

object.coerceTo(typeName) → array

Converts a value of one type into another.

You can convert: a selection, sequence, or object into an ARRAY, an array of pairs into an OBJECT, and any DATUM into a STRING.

Example: Convert a table to an array.

```
r.table('marvel').coerceTo('array').run(conn, callback)
```

Read more about this command →

typeOf

any.typeOf() → string

Gets the type of a value.

Example: Get the type of a string.

```
r.expr("foo").typeOf().run(conn, callback)
```

info

any.info() → object

Get information about a ReQL value.

Example: Get information about a table such as primary key, or cache size.

```
r.table('marvel').info().run(conn, callback)
```

json

r.json(json_string) → value

Parse a JSON string on the server.

Example: Send an array to the server'

```
r.json("[1,2,3]").run(conn, callback)
```

Command syntax

time.seconds() → number

Description

Return the seconds in a time object as a number between 0 and 59.999 (double precision).

Example: Return the post submitted during the first 30 seconds of every minute.

```
r.table("posts").filter(function(post) {
    return post("date").seconds().lt(30)
})
```

Command syntax

r.avg(attr)

Description

Compute the average value of the given attribute for the group.

Example: What's the average agility of heroes at each strength level?

```
r.table('marvel').groupBy('strength', r.avg('agility')).run(conn, callback)
```

Command syntax

cursor.close()

Description

Close a cursor. Closing a cursor cancels the corresponding query and frees the memory associated with the open request.

Example: Close a cursor.

```
cursor.close()
```

Command syntax

query.run(conn, callback)

query.run(options[, callback])

Description

Run a query on a connection. Accepts the following options:

- `useOutdated`: whether or not outdated reads are OK (default: `false`).
- `timeFormat`: what format to return times in (default: `'native'`). Set this to `'raw'` if you want times returned as JSON objects for exporting.

- `profile`: whether or not to return a profile of the query's execution (default: `false`).

The callback will get either an error, a single JSON result, or a cursor, depending on the query.

Example: Run a query on the connection `conn` and log each row in the result to the console.

```
r.table('marvel').run(conn, function(err, cursor) { cursor.each(console.log); })
```

Example: If you are OK with potentially out of date data from all the tables involved in this query and want potentially faster reads, pass a flag allowing out of date data in an options object. Settings for individual tables will supercede this global setting for all tables in the query.

```
r.table('marvel').run({connection:conn, useOutdated:true},
    function (err, cursor) { ... }
);
```

Example: If you just want to send a write and forget about it, you can set `noreply` to true in the options. In this case `run` will return immediately.

```
r.table('marvel').run({connection:conn, noreply:true},
    function (err, cursor) { ... }
);
```

Example: If you want to specify whether to wait for a write to be written to disk (overriding the table's default settings), you can set `durability` to `'hard'` or `'soft'` in the options.

```
r.table('marvel')
    .insert({ superhero: 'Iron Man', superpower: 'Arc Reactor' })
    .run({connection:conn, noreply:true, durability: 'soft'},
        function (err, cursor) { ... }
    );
```

Example: If you do not want a time object to be converted to a native date object, you can pass a `time_format` flag to prevent it (valid flags are "raw" and "native"). This query returns an object with two fields (`epoch_time` and `$reql_type$`) instead of a native date object.

```
r.now().run({connection:conn, timeFormat:"raw"},
    function (err, result) { ... }
);
```

Command syntax

sequence.concatMap(mappingFunction) → stream

array.concatMap(mappingFunction) → array

Description

Flattens a sequence of arrays returned by the mappingFunction into a single sequence.

Example: Construct a sequence of all monsters defeated by Marvel heroes. Here the field 'defeatedMonsters' is a list that is concatenated to the sequence.

```
r.table('marvel').concatMap(function(hero) {
    return hero('defeatedMonsters')
}).run(conn, callback)
```

Command syntax

r.dbList() → array

Description

List all database names in the system. The result is a list of strings.

Example: List all databases.

```
r.dbList().run(conn, callback)
```

Command syntax

number.mod(number) → number

Find the remainder when dividing two numbers.

Example: It's as easy as 2 % 2 = 0.

```
r.expr(2).mod(2).run(conn, callback)
```

Command syntax

conn.noreplyWait(callback)

Description

`noreplyWait` ensures that previous queries with the `noreply` flag have been processed by the server. Note that this guarantee only applies to queries run on the given connection.

Example: We have previously run queries with the `noreply` argument set to `true`. Now wait until the server has processed them.

```
conn.noreplyWait(function(err) { ... })
```

Command syntax

time.dayOfYear() → number

Description

Return the day of the year of a time object as a number between 1 and 366 (following ISO 8601 standard).

Example: Retrieve all the users who were born the first day of a year.

```
r.table("users").filter(
    r.row("birthdate").dayOfYear().eq(1)
)
```

Command syntax

number.sub(number) → number

time.sub(time) → number

time.sub(number) → time

Description

Subtract two numbers.

Example: It's as easy as 2 - 2 = 0.

```
r.expr(2).sub(2).run(conn, callback)
```

Example: Create a date one year ago today.

```
r.now().sub(365*24*60*60)
```

Example: Retrieve how many seconds elapsed between today and date

```
r.now().sub(date)
```

Command syntax

stream.zip() → stream

array.zip() → array

Description

Used to 'zip' up the result of a join by merging the 'right' fields into 'left' fields of each member of the sequence.

Example: 'zips up' the sequence by merging the left and right fields produced by a join.

```
r.table('marvel').eqJoin('main_dc_collaborator', r.table('dc'))
    .zip().run(conn, callback)
```

Command syntax

table.insert(json | [json][, {durability: "hard", returnVals: false, upsert: false}]) → object

Description

Insert documents into a table. Accepts a single document or an array of documents.

The optional arguments are:

- `durability`: possible values are `hard` and `soft`. This option will override the table or query's durability setting (set in run).
  In soft durability mode RethinkDB will acknowledge the write immediately after receiving it, but before the write has been committed to disk.
- `returnVals`: if set to `true` and in case of a single insert/upsert, the inserted/updated document will be returned.
- `upsert`: when set to `true`, performs a replace if a document with the same primary key exists.

Insert returns an object that contains the following attributes:

- `inserted`: the number of documents that were succesfully inserted.
- `replaced`: the number of documents that were updated when upsert is used.
- `unchanged`: the number of documents that would have been modified, except that the new value was the same as the old value when doing an upsert.
- `errors`: the number of errors encountered while performing the insert.
- `first_error`: If errors were encountered, contains the text of the first error.
- `deleted` and `skipped`: 0 for an insert operation.
- `generated_keys`: a list of generated primary keys in case the primary keys for some documents were missing (capped to 100000).
- `warnings`: if the field `generated_keys` is truncated, you will get the warning *"Too many generated keys (<X>), array truncated to 100000."*.
- `old_val`: if `returnVals` is set to `true`, contains `null`.
- `new_val`: if `returnVals` is set to `true`, contains the inserted/updated document.

Example: Insert a document into the table `posts`.

```
r.table("posts").insert({
    id: 1,
    title: "Lorem ipsum",
    content: "Dolor sit amet"
}).run(conn, callback)
```

The result will be:

```
{
    deleted: 0,
    errors: 0,
    inserted: 1,
    replaced: 0,
    skipped: 0,
    unchanged: 0
}
```

Example: Insert a document without a defined primary key into the table `posts` where the primary key is `id`.

```
r.table("posts").insert({
    title: "Lorem ipsum",
    content: "Dolor sit amet"
}).run(conn, callback)
```

RethinkDB will generate a primary key and return it in `generated_keys`.

```
{
    deleted: 0,
    errors: 0,
    generated_keys: [
        "dd782b64-70a7-43e4-b65e-dd14ae61d947"
    ],
    inserted: 1,
    replaced: 0,
    skipped: 0,
    unchanged: 0
}
```

Retrieve the document you just inserted with:

```
r.table("posts").get("dd782b64-70a7-43e4-b65e-dd14ae61d947").run(conn, callback)
```

And you will get back:

```
{
    id: "dd782b64-70a7-43e4-b65e-dd14ae61d947",
    title: "Lorem ipsum",
    content: "Dolor sit amet",
}
```

Example: Insert multiple documents into the table `users`.

```
r.table("users").insert([
    {id: "william", email: "william@rethinkdb.com"},
    {id: "lara", email: "lara@rethinkdb.com"}
]).run(conn, callback)
```

Example: Insert a document into the table `users`, replacing the document if the document already exists.
Note: If the document exists, the `insert` command will behave like replace, not like update

```
r.table("users").insert(
    {id: "william", email: "william@rethinkdb.com"},
    {upsert: true}
).run(conn, callback)
```

Example: Copy the documents from `posts` to `postsBackup`.

```
r.table("postsBackup").insert( r.table("posts") ).run(conn, callback)
```

Example: Get back a copy of the inserted document (with its generated primary key).

```
r.table("posts").insert(
    {title: "Lorem ipsum", content: "Dolor sit amet"},
    {returnVals: true}
).run(conn, callback)
```

The result will be

```
{
    deleted: 0,
    errors: 0,
    generated_keys: [
        "dd782b64-70a7-43e4-b65e-dd14ae61d947"
    ],
    inserted: 1,
    replaced: 0,
    skipped: 0,
    unchanged: 0,
    old_val: null,
    new_val: {
```

```
        id: "dd782b64-70a7-43e4-b65e-dd14ae61d947",
        title: "Lorem ipsum",
        content: "Dolor sit amet"
    }
}
```

Command syntax

any.typeOf() → string

Description

Gets the type of a value.

Example: Get the type of a string.

```
r.expr("foo").typeOf().run(conn, callback)
```

Command syntax

value.eq(value) → bool

Description

Test if two values are equal.

Example: Does 2 equal 2?

```
r.expr(2).eq(2).run(conn, callback)
```

Command syntax

sequence.pluck([selector1, selector2...]) → stream

array.pluck([selector1, selector2...]) → array

object.pluck([selector1, selector2...]) → object

singleSelection.pluck([selector1, selector2...]) → object

Description

Plucks out one or more attributes from either an object or a sequence of objects (projection).

Example: We just need information about IronMan's reactor and not the rest of the document.

```
r.table('marvel').get('IronMan').pluck('reactorState', 'reactorPower').run(conn, callback
```

Example: For the hero beauty contest we only care about certain qualities.

```
r.table('marvel').pluck('beauty', 'muscleTone', 'charm').run(conn, callback)
```

Example: Pluck can also be used on nested objects.

```
r.table('marvel').pluck({'abilities' : {'damage' : true, 'mana_cost' : true}, 'weapons' : tr
```

Example: The nested syntax can quickly become overly verbose so there's a shorthand for it.

```
r.table('marvel').pluck({'abilities' : ['damage', 'mana_cost']}, 'weapons').run(conn, call
```

Command syntax

array.prepend(value) → array

Description

Prepend a value to an array.

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')('equipment').prepend('newBoots').run(conn, callback)
```

Command syntax

value.default(default_value) → any

sequence.default(default_value) → any

Description

Handle non-existence errors. Tries to evaluate and return its first argument. If an error related to the absence of a value is thrown in the process, or if its first argument returns `null`, returns its second argument. (Alternatively, the second argument may be a function which will be called with either the text of the non-existence error or `null`.)

Exmple: Suppose we want to retrieve the titles and authors of the table `posts`. In the case where the author field is missing or `null`, we want to retrieve the string `Anonymous`.

```
r.table("posts").map( function(post) {
    return {
        title: post("title"),
        author: post("author").default("Anonymous")
    }
}).run(conn, callback)
```

We can rewrite the previous query with `r.branch` too.

```
r.table("posts").map( function(post) {
    return r.branch(
        post.hasFields("author"),
        {
            title: post("title"),
            author: post("author")
        },
        {
            title: post("title"),
            author: "Anonymous"
        }
    )
}).run(conn, callback)
```

Example: The `default` command can be useful to filter documents too. Suppose we want to retrieve all our users who are not grown-ups or whose age is unknown (i.e the field `age` is missing or equals `null`). We can do it with this query:

```
r.table("users").filter( function(user) {
    return user("age").lt(18).default(true)
}).run(conn, callback)
```

One more way to write the previous query is to set the age to be `-1` when the field is missing.

```
r.table("users").filter( function(user) {
    return user("age").default(-1).lt(18)
}).run(conn, callback)
```

Another way to do the same query is to use `hasFields`.

```
r.table("users").filter( function(user) {
    return user.hasFields("age").not().or(user("age").lt(18))
}).run(conn, callback)
```

The body of every `filter` is wrapped in an implicit `.default(false)`. You can overwrite the value `false` by passing an option in filter, so the previous query can also be written like this.

```
r.table("users").filter( function(user) {
    return user("age").lt(18)
}, {default: true} ).run(conn, callback)
```

Command syntax

r.time(year, month, day[, hour, minute, second], timezone) → time

Description

Create a time object for a specific time.

A few restrictions exist on the arguments:

- `year` is an integer between 1400 and 9,999.
- `month` is an integer between 1 and 12.
- `day` is an integer between 1 and 31.
- `hour` is an integer.
- `minutes` is an integer.
- `seconds` is a double. Its value will be rounded to three decimal places (millisecond-precision).
- `timezone` can be `'Z'` (for UTC) or a string with the format `±[hh]:[mm]`.

Example: Update the birthdate of the user "John" to November 3rd, 1986 UTC.

```
r.table("user").get("John").update({birthdate: r.time(1986, 11, 3, 'Z')}).run(conn, callba
```

Command syntax

sequence.withFields([selector1, selector2…]) → stream

array.withFields([selector1, selector2…]) → array

Description

Takes a sequence of objects and a list of fields. If any objects in the sequence don't have all of the specified fields, they're dropped from the sequence. The remaining objects have the specified fields plucked out. (This is identical to `has_fields` followed by `pluck` on a sequence.)

Example: Get a list of heroes and their nemeses, excluding any heroes that lack one.

```
r.table('marvel').withFields('id', 'nemesis')
```

Example: Get a list of heroes and their nemeses, excluding any heroes whose nemesis isn't in an evil organization.

```
r.table('marvel').withFields('id', {'nemesis' : {'evil_organization' : true}})
```

Example: The nested syntax can quickly become overly verbose so there's a shorthand.

```
r.table('marvel').withFields('id', {'nemesis' : 'evil_organization'})
```

Command syntax

table.indexWait([, index…]) → array

Description

Wait for the specified indexes on this table to be ready, or for all indexes on this table to be ready if no indexes are specified.

The result is an array where for each index, there will be an object like:

```
{
    index: <indexName>,
    ready: true
}
```

Example: Wait for all indexes on the table `test` to be ready:

```
r.table('test').indexWait().run(conn, callback)
```

Example: Wait for the index `timestamp` to be ready:

```
r.table('test').indexWait('timestamp').run(conn, callback)
```

Command syntax

r.epochTime(epochTime) → time

Description

Create a time object based on seconds since epoch. The first argument is a double and will be rounded to three decimal places (millisecond-precision).

Example: Update the birthdate of the user "John" to November 3rd, 1986.

```
r.table("user").get("John").update({birthdate: r.epochTime(531360000)}).run(conn, callbac
```

Command syntax

sequence.reduce(reductionFunction[, base]) → value

Description

Produce a single value from a sequence through repeated application of a reduction function.

The reduce function gets invoked repeatedly not only for the input values but also for results of previous reduce invocations. The type and format of the object that is passed in to reduce must be the same with the one returned from reduce.

Example: How many enemies have our heroes defeated?

```
r.table('marvel').map(r.row('monstersKilled')).reduce(function(acc, val) {
    return acc.add(val)
}, 0).run(conn, callback)
```

Command syntax

sequence.skip(n) → stream

array.skip(n) → array

Description

Skip a number of elements from the head of the sequence.

Example: Here in conjunction with `order_by` we choose to ignore the most successful heroes.

```
r.table('marvel').orderBy('successMetric').skip(10).run(conn, callback)
```

Command syntax

db.tableList() → array

Description

List all table names in a database. The result is a list of strings.

Example: List all tables of the 'test' database.

```
r.db('test').tableList().run(conn, callback)
```

Command syntax

conn.use(dbName)

Description

Change the default database on this connection.

Example: Change the default database so that we don't need to specify the database when referencing a table.

```
conn.use('marvel')
r.table('heroes').run(conn, ...) // refers to r.db('marvel').table('heroes')
```

Command syntax

table.between(lowerKey, upperKey[, {index:'id', left_bound:'closed', right_bound:'open'}])
→ selection

Description

Get all documents between two keys. Accepts three optional arguments: `index`, `left_bound`, and `right_bound`. If `index` is set to the name of a secondary index, `between` will return all documents where that index's value is in the specified range (it uses the primary key by default). `left_bound` or `right_bound` may be set to `open` or `closed` to indicate whether or not to include that endpoint of the range (by default, `left_bound` is closed and `right_bound` is open).

Example: Find all users with primary key >= 10 and < 20 (a normal half-open interval).

```
r.table('marvel').between(10, 20).run(conn, callback)
```

Example: Find all users with primary key >= 10 and <= 20 (an interval closed on both sides).

```
r.table('marvel').between(10, 20, {'right_bound':'closed'}).run(conn, callback)
```

Example: Find all users with primary key < 20. (You can use `NULL` to mean "unbounded" for either endpoint.)

```
r.table('marvel').between(null, 20, {'right_bound':'closed'}).run(conn, callback)
```

Example: Between can be used on secondary indexes too. Just pass an optional index argument giving the secondary index to query.

```
r.table('dc').between('dark_knight', 'man_of_steel', {index:'code_name'}).run(conn, callba
```

Command syntax

number.div(number) → number

Description

Divide two numbers.

Example: It's as easy as 2 / 2 = 1.

```
r.expr(2).div(2).run(conn, callback)
```

Command syntax

sequence.outerJoin(otherSequence, predicate) → stream

array.outerJoin(otherSequence, predicate) → array

Description

Computes a left outer join by retaining each row in the left table even if no match was found in the right table.

Example: Construct a sequence of documents containing all cross-universe matchups where a marvel hero would lose, but keep marvel heroes who would never lose a matchup in the sequence.

```
r.table('marvel').outerJoin(r.table('dc'), function(marvelRow, dcRow) {
    return marvelRow('strength').lt(dcRow('strength'))
}).run(conn, callback)
```

Command syntax

time.timezone() → string

Description

Return the timezone of the time object.

Example: Return all the users in the "-07:00" timezone.

```
r.table("users").filter( function(user) {
    return user("subscriptionDate").timezone().eq("-07:00")
})
```

Command syntax

r.branch(test, true_branch, false_branch) → any

Description

If the `test` expression returns `false` or `null`, the `false_branch` will be evaluated. Otherwise, the `true_branch` will be evaluated.

The `branch` command is effectively an `if` renamed due to language constraints. The type of the result is determined by the type of the branch that gets executed.

Example: Return heroes and superheroes.

```
r.table('marvel').map(
    r.branch(
        r.row('victories').gt(100),
        r.row('name').add(' is a superhero'),
        r.row('name').add(' is a hero')
    )
).run(conn, callback)
```

If the documents in the table `marvel` are:

```
[{
    name: "Iron Man",
    victories: 214
},
{
    name: "Jubilee",
    victories: 9
}]
```

The results will be:

```
[
    "Iron Man is a superhero",
    "Jubilee is a hero"
]
```

Command syntax

any.do(arg [, args]*, expr) → any

Description

Evaluate the expr in the context of one or more value bindings.

The type of the result is the type of the value returned from expr.

Example: The object(s) passed to do() can be bound to name(s). The last argument is the expression to evaluate in the context of the bindings.

```
r.do(r.table('marvel').get('IronMan'),
    function (ironman) { return ironman('name'); }
).run(conn, callback)
```

Command syntax

table.sync() → object

Description

`sync` ensures that writes on a given table are written to permanent storage. Queries that specify soft durability (`{durability: 'soft'}`) do not give such guarantees, so `sync` can be used to ensure the state of these queries. A call to `sync` does not return until all previous writes to the table are persisted.

If successful, the operation returns an object: `{synced: 1}`.

Example: After having updated multiple heroes with soft durability, we now want to wait until these changes are persisted.

```
r.table('marvel').sync().run(conn, callback)
```

Command syntax

sequence.slice(startIndex[, endIndex]) → stream

array.slice(startIndex[, endIndex]) → array

Description

Trim the sequence to within the bounds provided.

Example: For this fight, we need heroes with a good mix of strength and agility.

```
r.table('marvel').orderBy('strength').slice(5, 10).run(conn, callback)
```

Command syntax

table.replace(json | expr[, {durability: "hard", returnVals: false, nonAtomic: false}]) → object

selection.replace(json | expr[, {durability: "hard", returnVals: false, nonAtomic: false}]) → object

singleSelection.replace(json | expr[, {durability: "hard", returnVals: false, nonAtomic: false}]) → object

Description

Replace documents in a table. Accepts a JSON document or a ReQL expression, and replaces the original document with the new one. The new document must have the same primary key as the original document.

The optional arguments are:

- `durability`: possible values are `hard` and `soft`. This option will override the table or query's durability setting (set in run).
  In soft durability mode RethinkDB will acknowledge the write immediately after receiving it, but before the write has been committed to disk.
- `returnVals`: if set to `true` and in case of a single replace, the replaced document will be returned.
- `nonAtomic`: set to `true` if you want to perform non-atomic replaces (replaces that require fetching data from another document).

Replace returns an object that contains the following attributes:

- `replaced`: the number of documents that were replaced
- `unchanged`: the number of documents that would have been modified, except that the new value was the same as the old value
- `inserted`: the number of new documents added. You can have new documents inserted if you do a point-replace on a key that isn't in the table or you do a replace on a selection and one of the documents you are replacing has been deleted
- `deleted`: the number of deleted documents when doing a replace with `null`
- `errors`: the number of errors encountered while performing the replace.
- `first_error`: If errors were encountered, contains the text of the first error.
- `skipped`: 0 for a replace operation
- `old_val`: if `returnVals` is set to `true`, contains the old document.
- `new_val`: if `returnVals` is set to `true`, contains the new document.

Example: Replace the document with the primary key `1`.

```
r.table("posts").get(1).replace({
    id: 1,
    title: "Lorem ipsum",
    content: "Aleas jacta est",
    status: "draft"
}).run(conn, callback)
```

Example: Remove the field `status` from all posts.

```
r.table("posts").replace(function(post) {
    return post.without("status")
}).run(conn, callback)
```

Example: Remove all the fields that are not `id`, `title` or `content`.

```
r.table("posts").replace(function(post) {
    return post.pluck("id", "title", "content")
}).run(conn, callback)
```

Example: Replace the document with the primary key `1` using soft durability.

```
r.table("posts").get(1).replace({
    id: 1,
```

```
    title: "Lorem ipsum",
    content: "Aleas jacta est",
    status: "draft"
}, {
    durability: "soft"
}).run(conn, callback)
```

Example: Replace the document with the primary key `1` and return the values of the document before and after the replace operation.

```
r.table("posts").get(1).replace({
    id: 1,
    title: "Lorem ipsum",
    content: "Aleas jacta est",
    status: "published"
}, {
    returnVals: true
}).run(conn, callback)
```

The result will have two fields `old_val` and `new_val`.

```
{
    deleted: 0,
    errors: 0,
    inserted: 0,
    new_val: {
        id:1,
        title: "Lorem ipsum"
        content: "Aleas jacta est",
        status: "published",
    },
    old_val: {
        id:1,
        title: "Lorem ipsum"
        content: "TODO",
        status: "draft",
        author: "William",
    },
    replaced: 1,
    skipped: 0,
    unchanged: 0
}
```

Command syntax

table.orderBy([key1…], {index: index_name}) -> selection<stream>

selection.orderBy(key1, [key2…]) -> selection<array>

sequence.orderBy(key1, [key2…]) -> array

Description

Sort the sequence by document values of the given key(s). `orderBy` defaults to ascending ordering. To explicitly specify the ordering, wrap the attribute with either `r.asc` or `r.desc`.

Example: Order our heroes by a series of performance metrics.

```
r.table('marvel').orderBy('enemiesVanquished', 'damselsSaved').run(conn, callback)
```

Example: Indexes can be used to perform more efficient orderings. Notice that the index ordering always has highest precedence. Thus the following example is equivalent to the one above.

```
r.table('marvel').orderBy('damselsSaved', {index: 'enemiesVanquished'}).run(conn, callbac
```

Example: You can also specify a descending order when using an index.

```
r.table('marvel').orderBy({index: r.desc('enemiesVanquished')}).run(conn, callback)
```

Example: Let's lead with our best vanquishers by specify descending ordering.

```
r.table('marvel').orderBy(r.desc('enemiesVanquished'), r.asc('damselsSaved'))
.run(conn, callback)
```

Example: You can use a function for ordering instead of just selecting an attribute.

```
r.table('marvel').orderBy(function (doc) { return doc('enemiesVanquished') + doc('damselsS
```

Example: Functions can also be used descendingly.

```
r.table('marvel').orderBy(r.desc(function (doc) { return doc('enemiesVanquished') + doc('d
```

Command syntax

r.dbCreate(dbName) → object

Description

Create a database. A RethinkDB database is a collection of tables, similar to relational databases.

If successful, the operation returns an object: `{created: 1}`. If a database with the same name already exists the operation throws `RqlRuntimeError`.

Note: that you can only use alphanumeric characters and underscores for the database name.

Example: Create a database named 'superheroes'.

```
r.dbCreate('superheroes').run(conn, callback)
```

Command syntax

sequence.coerceTo(typeName) → array

value.coerceTo(typeName) → string

array.coerceTo(typeName) → object

object.coerceTo(typeName) → array

Description

Converts a value of one type into another.

You can convert: a selection, sequence, or object into an ARRAY, an array of pairs into an OBJECT, and any DATUM into a STRING.

Example: Convert a table to an array.

```
r.table('marvel').coerceTo('array').run(conn, callback)
```

Example: Convert an array of pairs into an object.

```
r.expr([['name', 'Ironman'], ['victories', 2000]]).coerceTo('object').run(conn, callback)
```

Example: Convert a number to a string.

```
r.expr(1).coerceTo('string').run(conn, callback)
```

Command syntax

sequence.contains(value1[, value2...]) → bool

Description

Returns whether or not a sequence contains all the specified values, or if functions are provided instead, returns whether or not a sequence contains values matching all the specified functions.

Example: Has Iron Man ever fought Superman?

```
r.table('marvel').get('ironman')('opponents').contains('superman').run(conn, callback)
```

Example: Has Iron Man ever defeated Superman in battle?

```
r.table('marvel').get('ironman')('battles').contains(function (battle) {return battle('wi
```

Command syntax

table.indexDrop(indexName) → object

Description

Delete a previously created secondary index of this table.

Example: Drop a secondary index named 'code_name'.

```
r.table('dc').indexDrop('code_name').run(conn, callback)
```

Command syntax

cursor.toArray(callback)

array.toArray(callback)

Description

Retrieve all results and pass them as an array to the given callback.

Example: For small result sets it may be more convenient to process them at once as an array.

```
cursor.toArray(function(err, results) {
    if (err) throw err;
    processResults(results);
});
```

The equivalent query with the `each` command would be:

```
var results = []
cursor.each(function(err, row) {
    if (err) throw err;
    results.push(row);
}, function(err, results) {
    if (err) throw err;
    processResults(results);
});
```

Command syntax

bool.and(bool) → bool

Description

Compute the logical and of two values.

Example: True and false anded is false?

```
r.expr(true).and(false).run(conn, callback)
```

Command syntax

array.insertAt(index, value) → array

Description

Insert a value in to an array at a given index. Returns the modified array.

Example: Hulk decides to join the avengers.

```
r.expr(["Iron Man", "Spider-Man"]).insertAt(1, "Hulk").run(conn, callback)
```

Command syntax

time.during(startTime, endTime[, options]) → bool

Description

Return if a time is between two other times (by default, inclusive for the start, exclusive for the end).

Example: Retrieve all the posts that were posted between December 1st, 2013 (inclusive) and December 10th, 2013 (exclusive).

```
r.table("posts").filter(
   r.row('date').during(r.time(2013, 12, 1, "Z"), r.time(2013, 12, 10, "Z"))
).run(conn, callback)
```

Example: Retrieve all the posts that were posted between December 1st, 2013 (exclusive) and December 10th, 2013 (inclusive).

```
r.table("posts").filter(
 r.row('date').during(r.time(2013, 12, 1, "Z"), r.time(2013, 12, 10, "Z"), {leftBound: "ope
).run(conn, callback)
```

Command syntax

time.toISO8601() → number

Description

Convert a time object to its iso 8601 format.

Example: Return the current time in an ISO8601 format.

```
r.now().toISO8601()
```

Command syntax

table.getAll(key[, key2...], [, {index:'id'}]) → selection

Description

Get all documents where the given value matches the value of the requested index.

Example: Secondary index keys are not guaranteed to be unique so we cannot query via "get" when using a secondary index.

```
r.table('marvel').getAll('man_of_steel', {index:'code_name'}).run(conn, callback)
```

Example: Without an index argument, we default to the primary index. While `get` will either return the document or `null` when no document with such a primary key value exists, this will return either a one or zero length stream.

```
r.table('dc').getAll('superman').run(conn, callback)
```

Example: You can get multiple documents in a single call to `get_all`.

```
r.table('dc').getAll('superman', 'ant man').run(conn, callback)
```

Command syntax

time.toEpochTime() → number

Description

Convert a time object to its epoch time.

Example: Return the current time in an ISO8601 format.

```
r.now().toEpochTime()
```

Command syntax

time.inTimezone(timezone) → time

Description

Return a new time object with a different timezone. While the time stays the same, the results returned by methods such as hours() will change since they take the timezone into account. The timezone argument has to be of the ISO 8601 format.

Example: Hour of the day in San Francisco (UTC/GMT -8, without daylight saving time).

```
r.now().inTimezone('-08:00').hours().run(conn, callback)
```

Command syntax

value.le(value) → bool

Description

Test if the first value is less than or equal to other.

Example: Is 2 less than or equal to 2?

```
r.expr(2).le(2).run(conn, callback)
```

Command syntax

r.expr(value) → value

Description

Construct a ReQL JSON object from a native object.

Example: Objects wrapped with expr can then be manipulated by ReQL API functions.

```
r.expr({a:'b'}).merge({b:[1,2,3]}).run(conn, callback)
```

Example: In JavaScript, you can also do this with just r.

```
r({a: 'b'}).merge({b: [1,2,3]}).run(conn, callback)
```

Command syntax

number.mul(number) → number

array.mul(number) → array

Description

Multiply two numbers, or make a periodic array.

Example: It's as easy as 2 * 2 = 4.

```
r.expr(2).mul(2).run(conn, callback)
```

Example: Arrays can be multiplied by numbers as well.

```
r.expr(["This", "is", "the", "song", "that", "never", "ends."]).mul(100).run(conn, callback
```

Command syntax

number.add(number) → number

string.add(string) → string

array.add(array) → array

time.add(number) → time

Description

Sum two numbers, concatenate two strings, or concatenate 2 arrays.

Example: It's as easy as 2 + 2 = 4.

```
r.expr(2).add(2).run(conn, callback)
```

Example: Strings can be concatenated too.

```
r.expr("foo").add("bar").run(conn, callback)
```

Example: Arrays can be concatenated too.

```
r.expr(["foo", "bar"]).add(["buzz"]).run(conn, callback)
```

Example: Create a date one year from now.

```
r.now().add(365*24*60*60)
```

Command syntax

table.indexStatus([, index...]) → array

Description

Get the status of the specified indexes on this table, or the status of all indexes on this table if no indexes are specified.

The result is an array where for each index, there will be an object like this one:

```
{
    index: <indexName>,
    ready: true
}
```

or this one:

```
{
    index: <indexName>,
    ready: false,
    blocks_processed: <int>,
    blocks_total: <int>
}
```

Example: Get the status of all the indexes on `test`:

```
r.table('test').indexStatus().run(conn, callback)
```

Example: Get the status of the `timestamp` index:

```
r.table('test').indexStatus('timestamp').run(conn, callback)
```

Command syntax

sequence.filter(predicate[, {default: false}]) → selection

stream.filter(predicate[, {default: false}]) → stream

array.filter(predicate[, {default: false}]) → array

Description

Get all the documents for which the given predicate is true.

`filter` can be called on a sequence, selection, or a field containing an array of elements. The return type is the same as the type on which the function was called on.

The body of every filter is wrapped in an implicit `.default(false)`, which means that if a non-existence errors is thrown (when you try to access a field that does not exist in a document), RethinkDB will just ignore the document. The `default` value can be changed by passing an object with a `default` field. Setting this optional argument to `r.error()` will cause any non-existence errors to return a `RqlRuntimeError`.

Example: Get all the users that are 30 years old.

```
r.table('users').filter({age: 30}).run(conn, callback)
```

A more general way to write the previous query is to use `r.row`.

```
r.table('users').filter(r.row("age").eq(30)).run(conn, callback)
```

Here the predicate is `r.row("age").eq(30)`.

- `r.row` refers to the current document
- `r.row("age")` refers to the field `age` of the current document
- `r.row("age").eq(30)` returns `true` if the field `age` is 30

An even more general way to write the same query is to use an anonymous function. Read the documentation about r.row to know more about the differences between `r.row` and anonymous functions in ReQL.

```
r.table('users').filter(function(user) {
    return user("age").eq(30)
}).run(conn, callback)
```

Example: Get all the users that are more than 18 years old.

```
r.table("users").filter(r.row("age").gt(18)).run(conn, callback)
```

Example: Get all the users that are less than 18 years old and more than 13 years old.

```
r.table("users").filter(r.row("age").lt(18).and(r.row("age").gt(13))).run(conn, callback)
```

Example: Get all the users that are more than 18 years old or have their parental consent.

```
r.table("users").filter(r.row("age").lt(18).or(r.row("hasParentalConsent"))).run(conn, ca
```

Example: Get all the users that are less than 18 years old or whose age is unknown (field `age` missing).

```
r.table("users").filter(r.row("age").lt(18), {default: true}).run(conn, callback)
```

Example: Get all the users that are more than 18 years old. Throw an error if a document is missing the field `age`.

```
r.table("users").filter(r.row("age").gt(18), {default: r.error()}).run(conn, callback)
```

Example: Select all users who have given their phone number (all the documents whose field `phoneNumber` is defined and not `null`).

```
r.table('users').filter(function(user) {
    return user.hasFields('phoneNumber')
}).run(conn, callback)
```

Example: Retrieve all the users who subscribed between January 1st, 2012 (included) and January 1st, 2013 (excluded).

```
r.table("users").filter(function(user) {
   return user("subscriptionDate").during( r.time(2012, 1, 1, 'Z'), r.time(2013, 1, 1, 'Z')
}).run( conn, callback)
```

Example: Retrieve all the users who have a gmail account (whose field `email` ends with `@gmail.com`).

```
r.table("users").filter(function(user) {
    return user("email").match("@gmail.com$")
}).run( conn, callback)
```

Example: Filter based on the presence of a value in an array.

Suppose the table `users` has the following schema

```
{
    name: String
    placesVisited: [String]
}
```

Retrieve all the users whose field `placesVisited` contains `France`.

```
r.table("users").filter(function(user) {
    return user("placesVisited").contains("France")
}).run( conn, callback)
```

Example: Filter based on nested fields.

Suppose we have a table `users` containing documents with the following schema.

```
{
    id: String
    name: {
        first: String,
        middle: String,
        last: String
    }
}
```

Retrieve all users named "William Adama" (first name "William", last name "Adama"), with any middle name.

```
r.table("users").filter({
    name:{
        first: "William",
        last: "Adama"
    }
}).run(conn, callback)
```

If you want an exact match for a field that is an object, you will have to use `r.literal`.

Retrieve all users named "William Adama" (first name "William", last name "Adama"), and who do not have a middle name.

```
r.table("users").filter(r.literal({
    name:{
        first: "William",
        last: "Adama"
    }
})).run(conn, callback)
```

The equivalent queries with an anonymous function.

```
r.table("users").filter(function(user) {
    return user("name")("first").eq("William")
        .and( user("name")("last").eq("Adama") )
}).run(conn, callback)

r.table("users").filter(function(user) {
    return user("name").eq({
        first: "William",
        last: "Adama"
    })
}).run(conn, callback)
```

Command syntax

table.indexList() → array

Description

List all the secondary indexes of this table.

Example: List the available secondary indexes for this table.

```
r.table('marvel').indexList().run(conn, callback)
```

Command syntax

table.indexCreate(indexName[, indexFunction]) → object

Description

Create a new secondary index on this table.

Example: To efficiently query our heros by code name we have to create a secondary index.

```
r.table('dc').indexCreate('code_name').run(conn, callback)
```

Example: A compound index can be created by returning an array of values to use as the secondary index key.

```
r.table('dc').indexCreate('parental_planets', function(hero) {
   return [hero('mothers_home_planet'), hero('fathers_home_planet')];
}).run(conn, callback)
```

Example: A multi index can be created by passing an optional multi argument. Multi indexes functions should return arrays and allow you to query based on whether a value is present in the returned array. The example would allow us to get heroes who possess a specific ability (the field 'abilities' is an array).

```
r.table('dc').indexCreate('abilities', {multi:true}).run(conn, callback)
```

Command syntax

singleSelection.keys() → array

object.keys() → array

Description

Return an array containing all of the object's keys.

Example: Get all the keys of a row.

```
r.table('marvel').get('ironman').keys().run(conn, callback)
```

Command syntax

r.row → value

Description

Returns the currently visited document.

Example: Get all users whose age is greater than 5.

```
r.table('users').filter(r.row('age').gt(5)).run(conn, callback)
```

Example: Accessing the attribute 'child' of an embedded document.

```
r.table('users').filter(r.row('embedded_doc')('child') > 5).run(conn, callback)
```

Example: Add 1 to every element of an array.

```
r.expr([1, 2, 3]).map(r.row.add(1)).run(conn, callback)
```

Example: For nested queries functions should be used instead of r.row.

```
r.table('users').filter(function(doc) {
    return doc('name').eq(r.table('prizes').get('winner'))
}).run(conn, callback)
```

Command syntax

time.dayOfWeek() → number

Description

Return the day of week of a time object as a number between 1 and 7 (following ISO 8601 standard). For your convenience, the terms r.monday, r.tuesday etc. are defined and map to the appropriate integer.

Example: Return today's day of week.

```
r.now().dayOfWeek().run(conn, callback)
```

Example: Retrieve all the users who were born on a Tuesday.

```
r.table("users").filter(
    r.row("birthdate").dayOfWeek().eq(r.tuesday)
)
```

Command syntax

array.setInsert(value) → array

Description

Add a value to an array and return it as a set (an array with distinct values).

Example: Retrieve Iron Man's equipment list with the addition of some new boots.

```
r.table('marvel').get('IronMan')('equipment').setInsert('newBoots').run(conn, callback)
```

Command syntax

sequence.hasFields([selector1, selector2…]) → stream

array.hasFields([selector1, selector2…]) → array

singleSelection.hasFields([selector1, selector2…]) → boolean

object.hasFields([selector1, selector2…]) → boolean

Description

Test if an object has all of the specified fields. An object has a field if it has the specified key and that key maps to a non-null value. For instance, the object `{'a':1,'b':2,'c':null}` has the fields `a` and `b`.

Example: Which heroes are married?

```
r.table('marvel').hasFields('spouse')
```

Example: Test if a single object has a field.

```
r.table('marvel').get("IronMan").hasFields('spouse')
```

Example: You can also test if nested fields exist to get only spouses with powers of their own.

```
r.table('marvel').hasFields({'spouse' : {'powers' : true}})
```

Example: The nested syntax can quickly get verbose so there's a shorthand.

```
r.table('marvel').hasFields({'spouse' : 'powers'})
```

Command syntax

value.gt(value) → bool

Description

Test if the first value is greater than other.

Example: Is 2 greater than 2?

```
r.expr(2).gt(2).run(conn, callback)
```

Command syntax

time.month() → number

Description

Return the month of a time object as a number between 1 and 12. For your convenience, the terms r.january, r.february etc. are defined and map to the appropriate integer.

Example: Retrieve all the users who were born in November.

```
r.table("users").filter(
    r.row("birthdate").month().eq(11)
)
```

Example: Retrieve all the users who were born in November.

```
r.table("users").filter(
    r.row("birthdate").month().eq(r.november)
)
```

Command syntax

sequence.innerJoin(otherSequence, predicate) → stream

array.innerJoin(otherSequence, predicate) → array

Description

Returns the inner product of two sequences (e.g. a table, a filter result) filtered by the predicate. The query compares each row of the left sequence with each row of the right sequence to find all pairs of rows which satisfy the predicate. When the predicate is satisfied, each matched pair of rows of both sequences are combined into a result row.

Example: Construct a sequence of documents containing all cross-universe matchups where a marvel hero would lose.

```
r.table('marvel').innerJoin(r.table('dc'), function(marvelRow, dcRow) {
    return marvelRow('strength').lt(dcRow('strength'))
}).run(conn, callback)
```

Command syntax

bool.not() → bool

Description

Compute the logical inverse (not).

Example: Not true is false.

```
r.expr(true).not().run(conn, callback)
```

Command syntax

sequence.without([selector1, selector2...]) → stream

array.without([selector1, selector2...]) → array

singleSelection.without([selector1, selector2...]) → object

object.without([selector1, selector2...]) → object

Description

The opposite of pluck; takes an object or a sequence of objects, and returns them with the specified paths removed.

Example: Since we don't need it for this computation we'll save bandwidth and leave out the list of IronMan's romantic conquests.

```
r.table('marvel').get('IronMan').without('personalVictoriesList').run(conn, callback)
```

Example: Without their prized weapons, our enemies will quickly be vanquished.

```
r.table('enemies').without('weapons').run(conn, callback)
```

Example: Nested objects can be used to remove the damage subfield from the weapons and abilities fields.

```
r.table('marvel').without({'weapons' : {'damage' : true}, 'abilities' : {'damage' : true}})
```

Example: The nested syntax can quickly become overly verbose so there's a shorthand for it.

```
r.table('marvel').without({'weapons':'damage', 'abilities':'damage'}).run(conn, callback)
```

Command syntax

r.sum(attr)

Description

Compute the sum of the given field in the group.

Example: How many enemies have been vanquished by heroes at each strength level?

```
r.table('marvel').groupBy('strength', r.sum('enemiesVanquished')).run(conn, callback)
```

Command syntax

cursor.each(callback[, onFinishedCallback])

array.each(callback[, onFinishedCallback])

Description

Lazily iterate over the result set one element at a time. The second callback is optional and is called when the iteration stops (when there are no more rows or when the callback returns `false`).

Example: Let's process all the elements!

```
cursor.each(function(err, row) {
    if (err) throw err;
    processRow(row);
});
```

Example: If we need to know when the iteration is complete, `each` also accepts a second `onFinished` callback.

```
cursor.each(function(err, row) {
        if (err) throw err;
        processRow(row);
    }, function() {
        doneProcessing();
    }
);
```

Example: Iteration can be stopped prematurely by returning `false` from the callback. For instance, if you want to stop the iteration as soon as `row` is negative:

```
cursor.each(function(err, row) {
    if (err) throw err;

    if (row < 0) {
        cursor.close()
        return false;
    }
    else {
        processRow(row)
    }
});
```

Note: You need to manually close the cursor if you prematurely stop the iteration.