
RethinkDB Documentation

Release 1.0

support@rethinkdb.com

July 18, 2011

Contents

| | | |
|----------|---|------------|
| 1 | Quick start guide | ii |
| 2 | Getting started | ii |
| 2.1 | Prerequisites | ii |
| 2.2 | Installation | iii |
| | Ubuntu | iii |
| | RHEL 5 / CentOS 5 / SUSE Linux 10 | iii |
| 2.3 | Running the server | iii |
| 2.4 | Language bindings | iv |
| 2.5 | Additional help | iv |
| 3 | Features | v |
| 3.1 | Memcached protocol | v |
| 3.2 | Performance | v |
| | Disk | v |
| | Multicore | vi |
| | Memory | vi |
| 3.3 | Durability | vi |
| | Flush interval | vi |
| | Unsaved data limit | vi |
| | Response mode | vii |
| 3.4 | Data migration and recovery | vii |
| 4 | Advanced features | vii |
| 4.1 | Advanced disk layout | vii |
| | Block device support | vii |
| | Block size | viii |
| | Extent size | viii |
| | Slices | viii |
| 4.2 | Garbage collector | viii |
| 4.3 | Consistency checks | ix |
| 4.4 | Advanced data recovery | ix |
| 5 | Troubleshooting | ix |
| 5.1 | “Too many open files” problem | ix |

RethinkDB is an industrial strength, high performance, durable data store. It supports a wide variety of operations on a set of key/value pairs, and is exposed to the network via a Memcached protocol. The engine is designed to be robust in a wide range of scenarios and operates efficiently on various hardware configurations, workloads, and dataset sizes. It is capable of operating in highly concurrent, high throughput, low latency environments, and in many cases can perform more than a million operations per second at sub-millisecond latency on commodity hardware.

The following is a sample list of scenarios in which RethinkDB can maintain efficient operation:

- Various dataset sizes, including datasets that fit into main memory, large datasets with *active* datasets that fit into main memory, and datasets that require disk access for every operation.
- Workloads with very large, very small, and mixed read/write ratios.
- A wide range of key and value sizes.
- Thousands of concurrent network connections.
- Pipelined as well as blocking operation requests.

In addition to supporting legacy hardware, RethinkDB takes full advantage of modern architectures, including multicore servers, solid-state drives, NUMA architectures, and fast Ethernet. It ships with tools that help easily perform data migration and recovery, run data consistency checks, do performance tuning, and more.

This manual describes the features exposed by RethinkDB in significant detail. For additional questions, please contact RethinkDB support.

1 Quick start guide

Download and install RethinkDB from <http://rethinkdb.com/download/>.

Start it:

```
$ rethinkdb
```

Connect to it via telnet and start typing memcached commands:

```
$ telnet localhost 11211
get foo
END
```

2 Getting started

2.1 Prerequisites

RethinkDB works on modern 64-bit distributions of Linux. We support the following distributions:

- Ubuntu 10.04.1 x86_64
- Ubuntu 10.10 x86_64
- Red Hat Enterprise Linux 5 x86_64
- CentOS 5 x86_64

- SUSE Linux 10

2.2 Installation

Ubuntu

Download the latest package of RethinkDB for Ubuntu from <http://rethinkdb.com/download/>.

Navigate to the directory the package was downloaded to and install RethinkDB and its dependencies:

```
# Get dependencies for RethinkDB
sudo apt-get install libaiol

# Install RethinkDB
dpkg -i rethinkdb-1.0-<VERSION>_amd64.deb
```

RHEL 5 / CentOS 5 / SUSE Linux 10

Download the latest package of RethinkDB for RHEL 5 / CentOS 5 from <http://rethinkdb.com/download/>.

Navigate to the directory the package was downloaded to and install RethinkDB and its dependencies:

```
# Install RethinkDB
rpm -i rethinkdb-1.0-<VERSION>.x86_64.rpm
```

Limitations

RHEL 5, CentOS 5, and SUSE Linux 10 kernels are missing certain system calls; this may affect performance in highly concurrent environments.

In these environments, server-side software will not scale to a large number of concurrent connections. In database environments this normally does not affect real-world performance, but may affect the results of some artificial benchmarks.

2.3 Running the server

Once RethinkDB is installed, start the server:

```
$ rethinkdb
```

This is equivalent to running RethinkDB with the `serve` command:

```
$ rethinkdb serve
```

This command will look for a database file named `rethinkdb_data` in the current directory, create it if it's missing, and start the server on port 11211. Alternatively, specify the database file and the port explicitly:

```
$ rethinkdb -f mydb.file -p 8080
```

To test that the server is operating correctly, we can `telnet` into the appropriate port and type Memcached commands directly. In the following `telnet` session we set a value for a key, get it back, and quit the connection:

```
$ telnet localhost 11211
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
set foo 0 0 3
bar
STORED
get foo
VALUE foo 0 3
bar
END
quit
Connection closed by foreign host.
```

To stop the server, type CTRL + C.

2.4 Language bindings

RethinkDB is binary compatible with the Memcached protocol, and can be used as a drop in replacement for an existing solution without any changes to the application. Client libraries that support the Memcached protocol will also work with RethinkDB. The following page contains a list of client libraries for various languages: <http://code.google.com/p/memcached/wiki/Clients>.

Note that many existing clients have not implemented full support for the Memcached protocol. You may encounter subtle issues with clients that aren't in mainstream use.

For example, if you're using Python with the *pylibmc* library, you can set and get keys in the following way:

```
>>> import pylibmc
>>> conn = pylibmc.Client(["localhost:11211"])
>>> conn.set("some_key", "some_value")
True
>>> conn.get("some_key")
'some_value'
```

2.5 Additional help

To get additional help on specific usage of RethinkDB, use the built-in `help` command. For example, to learn more about the `serve` command:

```
$ rethinkdb -h
$ rethinkdb help serve
```

To get a full list of commands available within RethinkDB:

```
$ rethinkdb help
```

Alternatively, you can get help from the RethinkDB man page that comes with the installation:

```
$ man rethinkdb
```

If you have additional questions, please contact RethinkDB support.

3 Features

3.1 Memcached protocol

RethinkDB implements the Memcached protocol as described on the following page: <http://code.sixapart.com/svn/memcached/trunk/server/doc/protocol.txt>. All specified commands should work as expected, and clients that work with Memcached implementations should continue working with RethinkDB without modification. The following is a list of known discrepancies with the Memcached protocol:

- Currently, only the text protocol is supported.
- Connections over UDP are not supported.
- Delete queues are not supported.
- The `flush_all` command is not supported.
- The `stat` command returns different statistics than specified in the protocol. Some of the statistics that do not make sense in the context of a persistent engine are removed, and new statistics are added.
- Value size limit is increased to 10MB from 1MB specified by Memcached.

3.2 Performance

RethinkDB has a number of features intended to increase performance. Common performance problems encountered with database systems involve disk I/O bottlenecks (number of possible operations per second, throughput, latency, etc.), CPU lock contention, and network bottlenecks. The following features are designed to mitigate performance problems associated with hardware bottlenecks.

Disk

Striping

Modern RAID controllers implement efficient striping across disks by synchronizing rotational disk spindles. Unfortunately, in the case of solid-state drives, no synchronization is possible. Because these drives often have varying latency, the entire array is limited to the speed of the slowest-operating drive at any given time. This significantly increases latency on write operations. RethinkDB implements disk striping that gets around this problem by writing to each disk independently. In order to take advantage of this feature you can partition a RethinkDB database across multiple files (located on one or many disks), and RethinkDB will take care of striping and latency issues automatically:

```
$ rethinkdb -f file1.db -f file2.db
```

If the files `file1.db` and `file2.db` are located on different disks, the I/O performance will double without needing to use a RAID controller and without sacrificing latency.

Note that this feature does not implement mirroring and parity guarantees implemented by advanced RAID controllers. The intention is not to entirely replace RAID, but to support an alternative partitioning method which can be very useful in certain situations.

Active extents

Rotational disks are fundamentally sequential machines—they have a single head that can read from, and write to a single location at a time. Many solid-state storage devices are fundamentally parallel—they have multiple flash memory chips and improve in performance if software distributes writes to multiple disk locations concurrently.

RethinkDB divides disk space into blocks of space called *extents*. Specify the number of concurrent extents by starting the server with the following flag:

```
$ rethinkdb --active-data-extents 4
```

For storage systems based on rotational drives, the value of `active-data-extents` should be set to 1. On write-heavy workloads, many solid-state drives will perform more efficiently if this value is between 2 and 16.

Multicore

RethinkDB has full support for machines with multiple CPUs and for CPUs with multiple cores. By default, the server takes advantage of all available cores on a machine. The number of cores the server should use can be specified explicitly:

```
$ rethinkdb --cores 8
```

This will limit the server to using eight cores. It is OK to over-provision cores (passing a larger number than the machine has), which may or may not affect performance in a real-world scenario.

Memory

The amount of available main memory can drastically affect performance of a database system because main memory is used to cache data and delays the need to go to disk, which is orders of magnitude slower. By default, RethinkDB will use as much memory as necessary (and as the system has available) to operate efficiently. However, this number can be specified explicitly:

```
$ rethinkdb --max-cache-size 8192
```

The cache size is specified in megabytes—the above command limits the cache size to 8GB.

3.3 Durability

Flush interval

For increased performance, RethinkDB delays flushing data to disk in order to batch updates and write them to disk more efficiently. The amount of time between flushes can be controlled explicitly (in milliseconds):

```
$ rethinkdb --flush-timer 1000
```

This tells the server to flush data to disk every second. A longer flush timer allows the server to batch writes more effectively and increase performance. A shorter flush timer flushes the data more often, but ensures that less data can be lost in the event of a power failure.

Unsaved data limit

In environments that operate under extremely high load, the network component is often significantly faster than the disk, which means commands arrive at a faster rate than the storage system can satisfy. In these situations RethinkDB implements throughput throttling—if the disk gets saturated, RethinkDB slows down its responses to commands to give the disk time to catch up.

To maintain high performance, RethinkDB often allows the commands to proceed despite the fact that the disk cannot catch up. This allows the changes to batch in memory and get flushed to disk later. In cases of power failure, this means large amounts of data can be lost. RethinkDB allows controlling precisely how much data is allowed to be cached in RAM without flushing to disk (in megabytes):

```
$ rethinkdb --unsaved-data-limit 1024
```

This allows RethinkDB to cache up to one gigabyte of unsaved data in RAM. In the event of a power failure, no more than one gigabyte of data will be lost. Adjust this limit to set the durability and performance trade-off to an acceptable level.

Response mode

By default, RethinkDB responds to write commands before they get committed to disk. This significantly decreases the latency and allows for increased throughput, but leaves the possibility of data loss in the event of power failure. It is possible to ensure no data loss in the event of a power failure by telling the server not to acknowledge writes until they are safely committed to disk:

```
$ rethinkdb --wait-for-flush y
```

Note that to minimize latency, if `wait-for-flush` is turned on, the flush interval should be set to a low value (or zero) to ensure low latency.

3.4 Data migration and recovery

RethinkDB provides tools for migrating into different solutions by exporting its data to the open Memcached format. The following command extracts the contents of a RethinkDB database:

```
$ rethinkdb extract -f file.db -o memcached.out
```

This command extracts the data from the database file `file.db` into a file named `memcached.out`. The contents of `memcached.out` will be standard Memcached insertion commands which can be piped into a different server that supports the Memcached protocol, or programmatically converted to other formats. For example, if we have a different server that supports a Memcached interface (including RethinkDB) running on a port 8080 we can fill it with the contents of the exported file with the following Unix command:

```
$ cat memcached.out | nc localhost 8080 -q 0
```

The `extract` command works even in cases when the data has been corrupted and server cannot open the database file. In this case, `extract` will try to recover as much data as possible and ignore the corrupted parts of the database file.

4 Advanced features

4.1 Advanced disk layout

RethinkDB allows for tuning of the internal layout of the database file. Depending on the underlying storage system, this may result in a significant boost in performance.

Block device support

RethinkDB can bypass the file system and run directly on the block device. In order for server to use a block device, the device first needs to be formatted:

```
$ rethinkdb create -f /dev/sdb
```

The database can be sharded across multiple devices:

```
$ rethinkdb create -f /dev/sdb -f /dev/sdc
```

If an existing database was previously created on the device, the server will output an error message. The block device can be reformatted by using the `force` argument:

```
$ rethinkdb create -f /dev/sdb -f /dev/sdc --force
```

Once one or more block devices have been formatted, the database server can be started as usual:

```
$ rethinkdb -f /dev/sdb -f /dev/sdc
```

Block size

By default, RethinkDB uses a 4KB block size. In some cases larger block sizes (8KB to 64KB) can yield higher performance. When the database is created, the block size can be specified explicitly as follows (in bytes):

```
$ rethinkdb create --block-size 8192 -f file.db
```

Extent size

Data blocks are grouped into `extents`. Large extents often allow for more efficient disk usage but may lower the performance of the garbage collector. An extent size can be specified explicitly during database creation as follows (in bytes):

```
$ rethinkdb create --extent-size 1048576 -f file.db
```

The above command formats the database with a 1MB extent size. Normally, extents should be able to hold anywhere from 256 to 8192 blocks.

Slices

RethinkDB automatically partitions the database into independent slices, which allows for efficient use of multiple disks and multicore CPUs. The number of slices can be specified explicitly during database creation time as follows:

```
$ rethinkdb create --slices 256 -f file.db
```

4.2 Garbage collector

RethinkDB ships with a concurrent, incremental on-disk garbage collector. Because the server uses a log-structured approach to storage, the database file can fill with unused blocks that need to be garbage collected. The garbage collector kicks in when there are too many unused blocks in a file, and turns off when the number of unused blocks reaches an acceptable level.

The window for garbage collector operation can be specified explicitly on startup as follows:

```
$ rethinkdb --gc-range 0.6-0.8
```

The above argument configures the garbage collector to kick in when 80% of the file contains unused blocks, and to stop collecting when less than 60% of the file contains unused blocks.

An aggressive garbage collection setting will keep a larger proportion of the disk available for live data, but may decrease performance of the system because of higher load on the disk.

4.3 Consistency checks

RethinkDB allows verifying that a given database is consistent and has not been corrupted. The corruption checks can be invoked as follows:

```
$ rethinkdb fsck -f file.db
```

If the database file is corrupted, the command above will report an error explaining the source of corruption.

4.4 Advanced data recovery

The recovery tool described in the data migration and recovery section exposes options to recover data in situations where the tool cannot be run automatically because of substantial metadata corruption. In such cases, block size, extent size, and slice numbers can be specified explicitly to allow the tool to proceed:

```
$ rethinkdb extract -f file.db --force-block-size 4096 \
                             --force-extent-size 1048576 \
                             --force-slice-count 256
```

5 Troubleshooting

5.1 “Too many open files” problem

RethinkDB can consume a large number of open file handles, for example when the number of socket connections is high. If you get a “Too many open files” error, that means that the operating system limit on the number of open file handles has been reached.

On most distributions of Linux you can find out the total limit for open file handles in the system using `sysctl`:

```
$ sysctl fs.file-max
fs.file-max = 764412
```

You can set this by running the following command under a root account or a user account with sufficient privileges:

```
$ sysctl fs.file-max=1592260
fs.file-max = 1592260
```

You can also change the per-process limit temporarily (in the current shell session), by using the `ulimit` command:

```
$ ulimit -n 2048
```

Set the limit to an appropriate number (2048 in the example), that is higher than the number of simultaneous connections to RethinkDB that you plan to have.

It is also possible to set per-user open file handles limits by editing `/etc/security/limits.conf` and setting the soft and hard limit values for `nofile` for the user or group which you use to run the RethinkDB under:

```
rethinkdb soft nofile 2048
rethinkdb hard nofile 8192
```

6 Support

Please report all issues to support@rethinkdb.com. When reporting an issue, please try to include the following pieces of information:

- A description of the environment you're running in (operating system, kernel version, hardware, etc).
- A description of the problem, how it came about, and how it can be reproduced.
- The RethinkDB log file. By default, log messages are written to standard output. In a production environment you may want to point them to a file on disk for easy collection using `--log-file` argument.
- If the problem involves a crash, please include the core dump file associated with the error. Core dumps are usually named `core` and are placed into the directory where the server was run. If you do not see a core dump file, you may need to enable core dumps by running the `ulimit -c unlimited` command.