UNIVERSITÉ LIBRE DE BRUXELLES,
UNIVERSITÉ D'EUROPE

ULB

Advanced Databases

NoSQL databases and

RethinkDB

Academic Year 2015/2016

Mariana Patrício - 000424350
Ward Taya - 000423834

Contents

Introduction and Scope

NoSQL databases are becoming an increasingly important part of the database landscape. If used adequately, they can offer real benefits however, before deciding on implementations, there should be full awareness of the limitations and issues that are associated with these kind of databases. Since relational databases were not designed to deal with the scale and agility challenges that face modern applications NoSQL is, for some kind of applications, an interesting solution. Therefore, we decided to explore an appropriate extend this topic. In this project we will start by doing an overview of the inherent concepts and the motivation of NoSQL databases and their main advantages and disadvantages of it.

As different NoSQL solutions seem to focus on different sets of features we choose to focus on a Document based NoSQL database. For experimental and exemplification purposes we used RethinkDB to explore some more this topic. RethinkDB, as we will see further ahead is a document database that stores JSON documents (also introduced in this paper). We start this part by introducing RethinkDB, its main features, data model and architecture considerations. Throughout the project we will be giving some small examples of how to model in ReQl, the query language used by RethinkDB (also introduced in this topic).

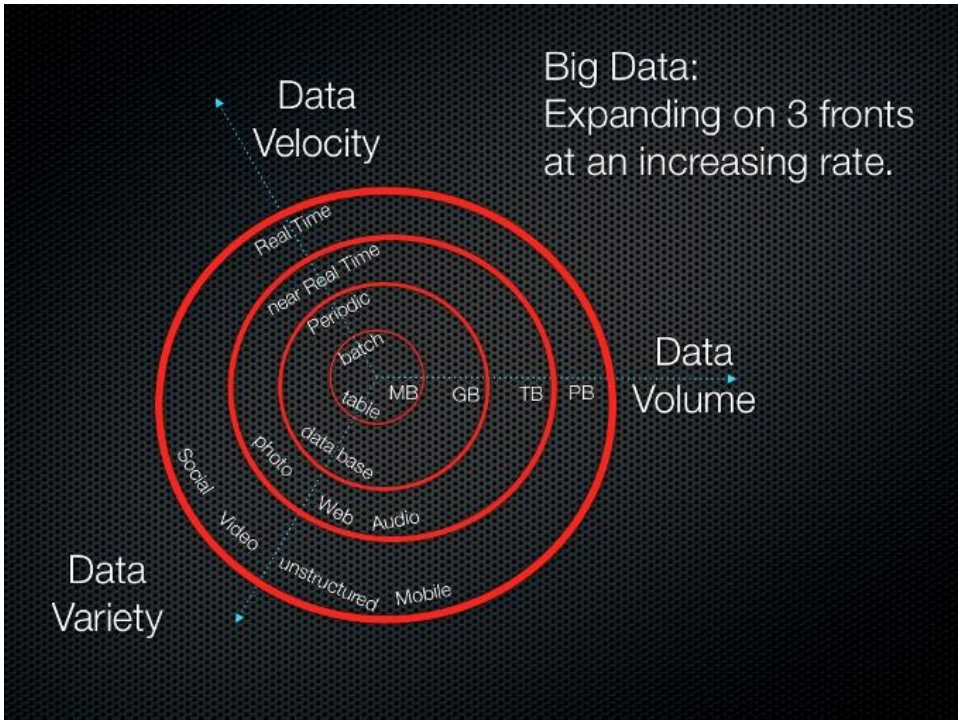
Finally, we are doing an experiment using RethinkDB and MS SQL Server for comparison. The purpose was to test the import of data and to run some different types of queries in both systems in order to better understand the strong points and weaknesses of this NoSQL database when compared to a relational database like MS SQL Server.

Background

Big Data

Big data is a collection of data from traditional and digital sources inside and outside an organization that represents a source for ongoing discovery and analysis.

Data is constantly being generated in many sources, in many types and shapes therefore requiring different methods and techniques to be stored and treated in order to produce content that can be useful for companies and people. Big data is being generated by almost everything at all times. Every digital process and social media exchange produces it. Systems, sensors and mobile devices transmit it. Big data is arriving from multiple sources at an increased volume, velocity and variety. These are the 3 main characteristics of Big Data. The volume of the data determines the value and potential of the data under consideration, and whether it can actually be considered big data or not. Regarding the velocity, the speed at which the data is generated and processed to meet the demands and the inherent challenges. The variety property refers to the type of content, an essential fact that data analysts must know. As stated, Big Data can come in many shapes and from many different sources.



The term Big Data implies a large volume of data – both structured and unstructured – that inundates a business on a day-to-day basis. For better understanding this concepts of structured/unstructured data will be introduced later on this paper. The purpose of storing, treating and analysing Big Data is so that information can be analysed for insights that lead to better decisions and strategic business moves. Therefore the challenge with Big Data is not getting the most data possible, but what to do with it. The insights that can be drawn from it can help an organization achieving cost and time reductions, new product development and optimization as well as reaching smarter decision making.

The users of Big Data can be considered as almost everyone. Big data affects organizations across practically every industry. From Baking to Retail or Health Care it is important to understand customers and boost their satisfaction. For baking for example, it's equally important to minimize risk and fraud while maintaining regulatory compliance. When it comes to health care, patient records, treatment plans and prescription information are generated at high velocity and need to be stored and analysed. Everything needs to be done quickly, accurately – and, in some cases, with enough transparency to satisfy stringent industry regulations. For all of these sectors, knowing the best way to market to customers, the most effective way to handle transactions, and the most strategic way to bring back lapsed business is of high importance.

Big Data is also present in the context of education. By using big data technologies, students track can be kept and evaluated in order to check if they are making adequate progress, for example. In case of the government, when government agencies are able to harness and apply analytics to their big data, they gain significant ground when it comes to managing utilities, running agencies, dealing with traffic congestion or preventing crime.

Manufactures are also users of Big Data. By having the insight produced by it, manufacturers can boost quality and output while minimizing waste – processes that are key in today's highly competitive market. More and more manufacturers are working in an analytics-based culture, which means they can solve problems faster and make more agile business decisions,

Therefore, Big Data is a set of techniques and technologies that require new forms of integration to uncover large hidden values from large datasets that are diverse, complex, and of a massive scale. The concept of Big Data usually implies data sets with sizes beyond the ability of commonly used software tools to capture, treat, manage, and process data within a reasonable time. For that reason, tools designed to search and analyse massive datasets, such as NoSQL database management systems are of high interest when considering all the information that can be generated within a company/organization and need to be treated and stored for future analysis.

Structured and Unstructured Data

Following the introduction to Big Data, we come across another relevant concept in this matter which is unstructured data. In order to better explain it we are presenting a brief explanation of what is structured data first to have a starting point.

- Structured Data

Structured data refers to any data that resides in a fixed field within a record or file. This includes data contained in relational databases and spreadsheets. Structured data has the advantage of being easily entered, stored, queried and analysed. Can be found as text files, displayed in titled columns and rows which can easily be ordered and processed by data mining tools. This type of data is stored, managed and queried in relational database management systems. Examples of those are Microsoft SQL Server, Oracle Database and MySQL.

Contrasting to unstructured data, structured data is data that can be easily organized. Regardless of its simplicity, it is estimated that structured data accounts for only 20% of the data available. Although unstructured data usually exists in a much higher number within the organizations, structured data has a critical role in data analytics. Besides being indispensable to critical business operations it supports a great amount of the operational systems. Relational database management systems (or RDBMSs) are a common choice for the storage of information in new databases used for financial records, manufacturing and logistical information, personnel data, among other operational systems.

- Unstructured Data

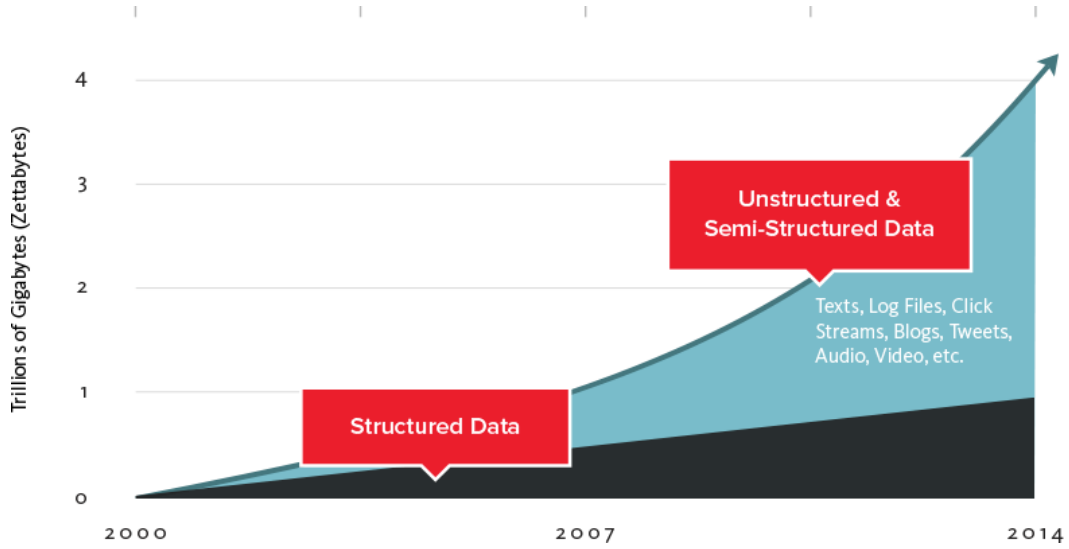
Unstructured Data (or unstructured information) refers to information that either does not have a pre-defined data model or is not organized in a pre-defined manner.

Unstructured data files often include text and multimedia content. Some examples of data that are often unstructured are e-mail messages, word processing documents, videos, photos, audio files, presentations, webpages and many other kinds of business documents. They are considered "unstructured" because the data they contain doesn't fit in a traditional row-column database. In all these instances, the data can provide compelling insights. Using the right tools, unstructured data can add a depth to data analysis that couldn’t be achieved otherwise

As mentioned in the Big Data introduction, the amount of data available is exploding and analysing large data sets is becoming more than a competitive advantage for companies, it is becoming essential to keep up with the market and increase the

competitivity. A great part of this organizations information growth is unstructured data, therefore finding and improving the ways to handle this kind of data is becoming more important in recent years.

Due to usually not including a predefined data model, unstructured data may not match well with relational tables. By its definition it can easily be perceived that ambiguities might happen, which are difficult to identify using conventional software programs. For this kind of data systems NoSQL databases, like RethinkDB, are used to manage it. We will explore this topic further, including RethinkDB data model ahead in this paper.



NoSql

The concept

Relational and NoSQL data models are very different. The relational model used the basic concept of a relation or table. The columns or fields in the table identify the attributes, whilst a tuple or row contains all the data of a single instance of the table. In the relational model, every tuple must have a unique identification or key based on the data. Often, keys are used to join data from two or more relations based on matching identification. The relational model takes data and separates it into many interrelated tables that contain rows and columns. Tables reference each other through foreign keys (which are primary keys in another table) that are stored in columns as well.

NoSQL databases (or Not Only SQL databases) have a very distinct model. Let's take as an example one of the kinds of NoSQL databases: Document databases, which is the type of RethinkDB, as we will see further ahead. It takes the data to store and aggregates it into JSON documents. A JSON document might take all the data stored in a row that would reference 20 tables of a relational database and aggregate it into a single document/object.

Since there is no schema, and no control over the structure of what is added to the JSON document, aggregating this information may lead to duplication (and sometimes inconsistency). Since storage is no longer cost prohibitive, the resulting data model's flexibility, efficiency in distributing the resulting documents, and read and write performance improvement makes it an easy trade-off for web-based applications.
The major difference is therefore that relational technologies have rigid schemas while NoSQL models are schemaless. Relational technology requires strict definition of a schema prior to storing any data into a database. Changing the schema once data is inserted is extremely disruptive and thus frequently avoided, which is a problem when application developers need to constantly and rapidly incorporate new types of data to enrich their apps. In comparison, document databases are schemaless, allowing the apps and users to freely add fields to JSON documents without having to first define changes.

Scalability and Performance

To deal with the increase in concurrent users and the volume of data, applications and their underlying databases need to scale. Scaling up implies a centralized approach that relies on bigger and bigger servers. Scaling out implies a distributed approach that leverages many standard, commodity (physical or virtual) servers.

As more people use an application, more servers are added to the web/application tier, performance is maintained by distributing load across an increased number of servers, and the cost scales linearly with the number of users.

With relational database technology, at some point the capacity of even the biggest server can be exceeded as users and data requirements continue to grow. At that point, the relational database cannot scale further and must be split across two or more servers. This introduces enormous complexities for both application development and database administration. For the cases where this situation is predictable, a NoSQL solution might be an option considering that NoSQL databases were developed from the ground up to be distributed.

NoSQL databases provide an easier, linear, and cost effective approach to database scaling. As the number of concurrent users grows, simply add additional low-cost,

commodity servers to the cluster. A NoSQL database automatically spreads data across servers, without requiring applications to participate. Servers can be added or removed from the data layer without application downtime, with data (and I/O) automatically spread across the servers. Most NoSQL databases also support data replication, storing multiple copies of data across the cluster and even across Data Centers, to ensure high availability and support disaster recovery.

The main advantages of NoSQL

Since it is, by design, made for Big Data, NoSQL is capable of storing, processing, and managing huge amounts of data. This not only includes the structured data collected from a web form or other kind of structured input data, but text messages, word processing documents, videos and other forms of unstructured data as well.

NoSQL databases handle partitioning (sharding) across several servers. So, if the data storage requirements grow too much, inexpensive servers can still be continuously added and connected to the database cluster (horizontal scaling) making them work as a single data service. In contrast, in the relational database world there is a need to buy new, more powerful and thus more expensive hardware to scale up (vertical scaling).

NoSQL databases are designed with scalability in mind, offering a convenient way for companies to transition to new nodes both on premise and in the cloud as well – all while maintaining the high level of performance and availability.

Thus, one of the main advantages of NoSQL databases is their structure that allows it to be easily scalable. Also by being replicated and sharded, it avoids downtime in most of the cases and increases fault tolerance, since it enables a system to continue operating properly in the event of the failure of one of the servers.

Using NoSQL databases allows developers to develop without having to convert in-memory structures to relational structures. Relational databases were not designed to run efficiently on clusters.

The rise of the web as a platform also created a vital factor change in data storage as the need to support large volumes of data by running on clusters. The data storage needs of an ERP application, for example, are lot more different than the data storage needs of a Facebook or another social media platform. Many applications need simple object storage, whereas others require highly complex and interrelated structure storage. NoSQL databases provide support for a range of data structures.

Moreover, NoSQL databases support storing data as it is inserted or captured. Key value stores give the ability to store simple data structures, whereas document NoSQL

databases have the ability to handle a range of flat or nested structures. Being able to handle these formats natively in a range of NoSQL databases lessens the amount of code needed to convert from the source data format to the format that needs storing. Because of the lack of schema of NoSQL databases, they're very capable of managing change — there is no need to rewrite ETL routines if the XML message structure between systems changes.

The main trade-offs

In a distributed system, managing consistency(C), availability (A) and partition tolerance (P) is important. Hence, the CAP Theorem states that it is impossible for a distributed computer system to simultaneously provide all three of these guarantees, therefore we can choose only two of consistency, availability or partition tolerance. Consistency means that all nodes see the same data at the same time. Availability is a guarantee that every request receives a response about whether it succeeded or failed. In order to have Partition tolerance, a system has to continue to operate despite arbitrary message loss or failure of part of the system.

Many NoSQL databases try to provide options where the developer has choices where they can tune the database as their needs. The CAP theorem states that if you get a network partition, you have to trade off availability of data versus consistency of data. Understanding the requirements becomes much more important.

However, if we want to use a NoSQL database for distributed systems and to be scalable, consistency is usually being traded off. When scaling databases one approach is to replicate data in different servers to decrease reading times. So in that case, one user could read data which has not been yet updated, but that eventually will, leaving the database with some consistency issues. The "Consistency" that concerns NoSQL databases is found in the CAP theorem, which signifies the immediate or eventual consistency of data across all nodes that participate in a distributed database. Considering data can be replicated across nodes in order to maintain availability its consistency is maintained differently given the nature and architecture of the system.

Since there are no relations between distinct Table Entities and Rows, thus no foreign keys, referential Integrity is not enforced in NoSQL databases. Therefore, most of the NoSQL systems have to rely on applications to enforce data integrity where SQL uses a declarative approach.

Another drawback is that most of the administration is depends upon scripting like bash, perl e.t.c., in linux environment. This way, NoSQL databases are usual less intuitive and harder to manage than SQL databases.

Different types of NoSQL databases

NoSQL is simply the term that is used to describe a family of databases that are all non-relational. While the technologies, data types, and use cases vary wildly amount them, it is generally agreed that there are four types of NoSQL databases:

- Column

Column-family databases generally serialize all the values of a particular column together on-disk, which makes retrieval of a large amount of a specific attribute fast. This approach lends itself well to aggregate queries and analytics scenarios where might be a need to run range queries over a specific field. Examples of this kind of databases include: Accumulo, Cassandra, Druid, HBase and Vertica.

- Document

A document-oriented NoSQL database stores and aggregates data in "documents" where a document can generally be thought of as a grouping of key-value pairs, instead of in tables of rows and columns as in the relational databases model. This approach results in better data model flexibility, greater efficiently in distributing documents, and superior read and write performance. RethinkDB is one of this kind of databases, so we will focus more on its functioning and structure later on this paper. Some examples are: Apache CouchDB, Clusterpoint, Couchbase, DocumentDB, HyperDex, Lotus Notes, MarkLogic, MongoDB, OrientDB, Qizx and RethinkDB.

- Key-Value

These databases pair keys to values. There are usually no fields to update, instead, the entire value other than the key must be updated if changes are to be made. The simplicity of this scales well but it can limit the complexity of the queries and other advanced features. Examples can be tought as the following: CouchDB, Oracle NoSQL Database, Dynamo, MemcacheDB, Redis, Riak, Aerospike, OrientDB, MUMPS, FoundationDB and HyperDex.

- Graph

These excel at dealing with interconnected data. Graph databases consist of connections, or edges, between nodes. Both nodes and their edges can store additional properties such as key-value pairs. The strength of a graph database is in traversing the connections between the nodes. But they generally require all data to fit on one machine, limiting their scalability. Examples include: Allegro, Neo4J, InfiniteGraph, OrientDB, Virtuoso, Stardog and Sesame.

A brief introduction to JSON

As we have seen before, RethinkDB stores JSON documents, having therefore dynamic schemas.

JSON stands for JavaScript Object Notation. It is a syntax for storing and exchanging data. It is an alternative to XML, easier to use, language independent and easier to understand. Though many programming languages support JSON, it is especially useful for JavaScript-based apps, including websites and browser extensions. JSON has been popularized by web services developed utilizing REST principles.

It was developed to fulfil the need for a real-time server-to-browser communication without using browser plugins such as Flash or Java applets, which were widely used in the early 2000s. In more recent years has been appearing in the market a new breed of databases such as MongoDB, Rethinkdb and Couchbase that store data natively in JSON format.

Regarding the data types, it can represent numbers, Booleans, strings, null, arrays (ordered sequences of values), and objects (string-value mappings). JSON does not natively represent more complex data types like functions, regular expressions, dates, and so on.

A number is very much like a C or Java number, except that the octal and hexadecimal formats are not used. A Boolean, as normally does, represents one of the values true or false. A string is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. A character is represented as a single character string.

An array is an ordered collection of values. An array begins and ends with brackets ([ ]). Values are separated by commas (,). An object is an unordered set of name/value pairs. An object begins and ends with braces ({ }). Each name is followed by colon (:) and the name/value pairs are separated by a comma (,).

To give an example, an object representing the principal attributes of the ULB, in JSON, would look like this:

```
var ulb = {
"name" : "ULB",
"year" : "1834",
"city" : "Brussels",
"type" : "University"
};
```

The previous statement creates an object that can be accessed using the variable ‘ulb’. By enclosing the variable’s value in curly braces, we’re indicating that the value is an object. Inside the object, we can declare any number of properties.

If we storing two landmarks in one variable. To do this, we enclose multiple objects in square brackets, which signifies an array. For instance, to include information about ULB and the Atomium the following code would perform:

```
var bru_landmarks = [{
"name" : "ULB",
"year" : "1834",
"city" : "Brussels",
"type" : "University"
},
{
"name" : "Atomium",
"year" : "1958",
"city" : "Brussels",
"type" : "Museum"
}];
```

The previous array could also be written in XML. The following example also defines the 2 landmark objects mentioned:

```
<landmarks>
    <landmark>
        <name>ULB</name> <year>1834</year>
<city>Brussels</city> <type>University</type>
    </landmark>
    <landmark>
        <name>Atomium</name> <year>1958</year>
<city>Brussels</city> <type>Museum</type>
    </landmark>

</landmarks>
```

RethinkDB – Main Concepts

What is RethinkDB?

RethinkDB is a document-oriented database built to store JSON documents, and scale to multiple machines with very little effort. It has a query language that supports really useful queries like table joins and group by, and is easy to setup and learn.

RethinkDB is the first open-source, scalable JSON database built from the ground up for the realtime web. It inverts the traditional database architecture by exposing an exciting new access model – instead of polling for changes, the developer can tell RethinkDB to continuously push updated query results to applications in realtime. RethinkDB's realtime push architecture dramatically reduces the time and effort necessary to build scalable realtime apps.

In addition to being designed from the ground up for realtime apps, RethinkDB offers a flexible query language, intuitive operations and monitoring APIs, and is easy to setup and learn.

When RethinkDB is a good choice?

RethinkDB is a great choice when the applications could benefit from realtime feeds to the source data.

The query-response database access model works well on the web because it maps directly to HTTP's request-response. However, modern applications require sending data directly to the client in realtime. Use cases where companies benefited from RethinkDB's realtime push architecture include:

- Collaborative web and mobile apps
- Streaming analytics apps
- Multiplayer games
- Realtime marketplaces
- Connected devices

For example, when a user changes the position of a button in a collaborative design app, the server has to notify other users that are simultaneously working on the same project. Web browsers support these use cases via WebSockets and long-lived HTTP connections, but adapting database systems to realtime needs still presents a huge engineering challenge.

RethinkDB is the first open-source, scalable database designed specifically to push data to applications in realtime. It dramatically reduces the time and effort necessary to build scalable realtime apps.

To differ from other available document-based or NoSql querying languages, RethinkDB works differently. It was built from scratch with the aim of having its push architecture as its key of success and usage. RethinkDB's flexibility and ability to scale horizontally is impressive not only for the expectations held for this newcomer in a world of giants leading NoSql (e.g. MongoDB, Cassandra,etc..), but it's impressive in a way it is putting RethinkDB just in the same line of leading technologies.

By the following sections, we will research RethinkDB's features more and later on will demonstrate our insights based on the tests we are performing. We expect by the end of this research to be able to distinguish both strong and weak points of RethinkDB.

"It's often times just one of a few components to an overall data solution, but for its part, RethinkDB means there will be no more busy work getting data refreshed on each client.", Slava Akhmechet, Co-Founder, RethinkDB.

Main features

NoSQL – schema-less approach

RethinkDB is a JSON document database. A JSON document represents a structured object consisting of key/value pairs. The value can be either a primitive data type (integer, string, floating point number) or a nested JSON object (represented in document form). This means, of course, that JSON can describe arbitrarily complex objects.
RethinkDB stores documents in tables. While this might lead one to think that RethinkDB has relational database ingredients, the fact is a table is simply a logical container; the RethinkDB engineers chose to call that container "table" so that developers coming from a relational background would feel comfortable.
A table in RethinkDB places no real restrictions on the structure of its contained documents. In the relational world, all rows within a table necessarily have the same structure (that is, they have the same fields). But RethinkDB is schema-less: No two

documents, not even documents within the same table, need to have the same structure. Of course, it's generally beneficial for all documents in a table to have the same structure, as it simplifies organization and management. But the flexibility is there, if needed. The RethinkDB documentation provides a nice overview of data modelling options in its "yes it's a table but not really" world.

Realtime push architecture

As a technology which has been designed from scratch for this approach, it's considered as its main feature and the very main reason it might be picked over other existing technologies.

Typically, clients are in touch with alterations in database contests by querying the database itself. Thus, if we want to figure what user A has updated in the database, we will need to query it (in different approaches, depends which language is used). Nevertheless, the application is aware of changes only when polling the database.

In RethinkDB, this whole idea has changed. It provides a real-time push architecture. Clients can listen do DB changes, that they're notified once there's any change, any time, without a need to poll the database repeatedly.

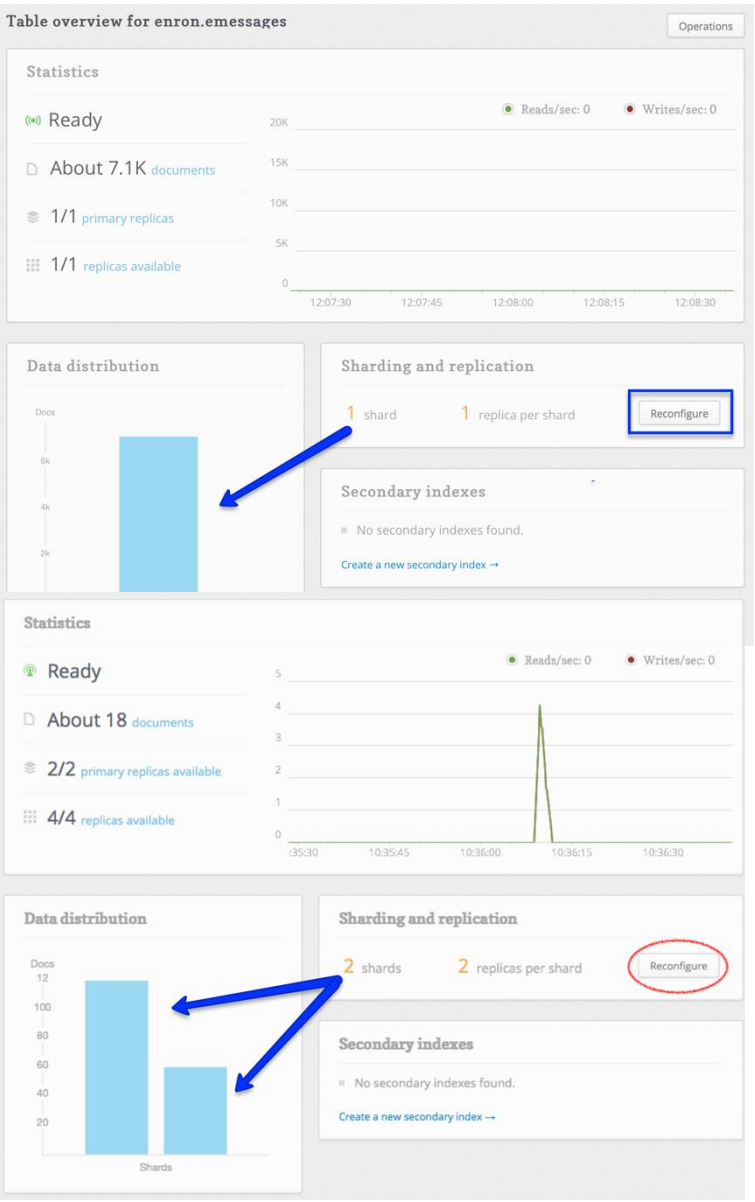
This, named as 'changefeed', option can be applied for a table, a document, or a query. This option can be enabled by using the changes() command, which behaves as an infinite cursor object that provides an endless set of change documents.

Configures can be made to changefeeds to throttle the delivery of information. For example, a changefeed might be configured to wait for N changes before sending a response to the listening application. RethinkDB will merge multiple changes together into a single response. This reduces network traffic. In addition, if many alterations occur between subsequent fetches from the changefeed cursor, RethinkDB will collapse the result so that intermediate changes are discarded, and only the previous and current values are reported.

Change documents also include informative state information, such as whether the item is an initial document (in which case the change is really an addition). Changes are buffered at the server, and if that buffer hits its limit, the server will discard early change documents and insert a special error document into the stream of changes. This error document will indicate how many documents were skipped on account of the buffer overrun.

Sharding and failover in RethinkDB

As it is a distributed database, it distributes data around the cluster by sharding. Thus, it allows us to shard and replicate data on the cluster on a table basis. Such possibility can be configured and enabled through the RethinkDB's management web interface as shown in the pictures at the left side.



By the time writing these words, RethinkDB performs a range sharding, while in future releases it will be sharding based on hash values. As RethinkDB is fully open-source, we tracked the status of this release and it's still under development. (Github issue #364)

(screenshot from Enron's DB which is being tested as part of this project)

RethinkDB expends a great deal of effort ensuring that data change events are quickly dispatched throughout the cluster. And it provides this high-speed event processing mechanism while offering plenty of control over database consistency and durability.

(screenshot from RethinkDB's guideline)

Finally, RethinkDB supports failover, which requires that the cluster have at least three nodes and tables be configured to have more than two shards. If a node becomes unavailable and happens to host the primary replica for a table, then one of the secondary nodes is selected by RethinkDB to become the new primary. No data is lost. Should the lost node come back online, it will resume its position as primary. Note that, even if a majority of nodes for a given replica are lost, data can still be retrieved, though it requires a special recovery operation.

JOINs and GMR

In RethinkDB, the equivalent of a relational JOIN operation can actually be performed. In a relational system, a JOIN of two tables connects specified columns in each table. In RethinkDB, a JOIN of two tables connects specified fields, one in each set of the documents that comprise each table.
In addition, ReQL supports a variant of map-reduce called group-map-reduce (GMR). The map operation will fetch a sequence from the database, a sequence being a set of documents or document fields, and transform it into a different sequence. The reduce operation aggregates the results prior to delivery as the query's response. The additional "group" step can be used to amass the sequences into partitions for which separate results are produced. (For example, we might use the group operation to gather the results of a map-reduce process by gender.)
RethinkDB's GMR system is a low-level API into the database. Higher-level ReQL commands (JOIN, for example) are compiled into GMR queries and executed on the server by the GMR infrastructure. More extensive control of resource allocation for GMR queries is planned in a future release of RethinkDB.

Simplified query language: ReQL

Rethink Query Language, known as ReQL, is a simplified and easy language with a mix from Object Oriented Programming notation, jointly with SQL and NoSQL context. We will focus in this paper on a conversion from SQL to ReQL (detailed in the next section).

From SQL to ReQL

Terminology

SQL and RethinkDB share very similar terminology. Below is a table, retrieved from RethinkDB website, with a comparison of the 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

Following, we will introduce the basic operation that can be done in ReQL. ReQL is the RethinkDB query language. It offers a very powerful and convenient way to manipulate JSON documents. It is different from other NoSQL query languages. It's built on three key principles:

1. ReQL embeds into the programming languages. Queries are constructed by making function calls in the programming languages. There is no need to concatenate strings or construct specialized JSON objects to query the database. Furthermore, ReQL can also be directly used in the Data Explorer functionality of the user interface RethinkDB provides.
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 the run command is called and passed in an active database connection.

Examples

- Create and drop the database:

```
r.dbCreate('advanced_db');
r.dbDrop('advanced_db');
```

- Create and drop a table:

```
r.tableCreate('landmarks', {primaryKey: "landmark_id"});
r.tableCreate('creators');
```

Note that the default key is the id, so if nothing is mentioned an id field will be automatically created and filled every time a tuple is inserted.

```
r.tableDrop("landmarks");
r.tableDrop("creators");
```

- Insert into the table (ex: four documents in the landmark table and one in the creators table):

```
r.table('landmarks').insert([
                { name: "ULB", year: "1834",
                     city: "Brussels", type: "University"
```

```
                },
                { name: "Atomium", year: "1958",
                     city: "Brussels", type: "Museum"
                },
                { name: "Cinquantenaire", year: "1880",
                     city: "Brussels", type: "Park"
                },
                { name: "Church of Our Lady", year: "1280",
                     city: "Bruges", type: "Church"
                }
]);

r.table('creators').insert(
  "landmark_id": "7644aaf2-9928-4231-aa68-4e65e31bf219",

  "name": " founder 1",

  "field": "Sciences"

};
```

- Delete from a table (ex: delete all tuples in the landmark table):

```
r.table("landmarks").delete();
```

- Update (ex: update all “year” attributes that refer to dates prior to 1834 to the value 1500):

```
r.table("landmarks").filter(
    r.row("year").lt(1834)
).update({year: 1500});
```

- Select - specific attributes (ex: select the name and city from the landmark table):

```
r.table("landmarks").pluck("name", "city");
```

- Select – the equivalent of the where clause (ex: get all the tuples where the name is ULB):

```
r.table("landmarks").filter({
    name: "ULB"});
```

- Select – the equivalent of the where … in … clause (ex: get all the tuples where the name is in the following list {“ULB”,”Atomium”}):

```
r.table("landmarks").filter(
  function (doc) {
```

```
    return r.expr(["ULB","Atomium"])
            .contains(doc("name"));
  }
);
```

- Case (ex: retrieve a calculated field that has the value “yes” in case the landmark is in Brussels and no otherwise):

```
r.table("landmarks").map({
    name: r.row("name"),
    is_in_Brussels: r.branch(
        r.row("city").eq(“Brussels”),
        "yes",
        "no"
    )
});
```

- Join (Inner join, outer join and left join):

If we want to make an inner join between two tables (in this case to retrieve all the creators who have a landmark associated) we can do it the following way:

```
r.table("creators").innerJoin(
  r.table("landmarks"),
  function (creator, landmark) {
    return creator("landmark_id").eq(landmark("landmark_id"));
}).zip();
```

We can also make an outter join the same way by replacing the .innerJoin command with the .outterJoin command. If we have an index (primary key or secondary index) built on the field of the right table, we can perform a more efficient join with eqJoin (that will retrieve all the creators with their associated landmarks):

```
r.table("creators").eqJoin(
    "landmark_id",
    r.table("landmarks"),
    {index: "landmark_id"}
).zip();
```

- Order by (Select the values from the table landmark ordered by the “name” field):

```
r.table("landmarks").orderBy("name");
```

- Group by (with count() field) :

```
r.table('landmarks')
 .group('city')
 .count();
```

- Listen for changes:

```
r.db("advanced_db").table("landmarks").changes();
```

With this command the application will be listening for changes in the table to which it is listening. If some update, insertion or deletion is made it will automatically be printed.

This chapter is meant to be just a brief introduction to ReQL. Therefore, if more detailed instructions and further examples are required they can be found in: https://www.rethinkdb.com/docs/sql-to-reql/ javascript/
,
https://rethinkdb.com/docs/guide/javascript/
and
https://www.rethinkdb.com/api/javascript/

RethinkDB Data Model

In this chapter we will introduce the basic components of RethinkDB data model. More detailed information can be found in the website (https://rethinkdb.com/docs/data-modeling/).

ReQL data types

Basic data types

RethinkDB stores six basic single-value data types: numbers, strings, boolean values, the null value, binary objects and times. In addition, it stores two basic composite data types. Objects and arrays are key/value pairs and lists, respectively.
Considering RethinkDB stores JSON documents, some of these this data types were already introduced in the JSON chapter, except for the binary objects and the times, which are not natively supported by JSON documents. For that reason, we won't enter into much detail in the data types that are also supported natively by JSON, giving here just a brief description of them.

- Numbers

Numbers are any real number. RethinkDB uses double precision (64-bit) floating point numbers internally.

- Strings

Strings are any UTF-8 string that does not contain the null code point. Currently RethinkDB does not enforce UTF-8 encoding, but most string operations assume they are operating on UTF-8 strings. Future versions of RethinkDB will enforce UTF-8 encoding and allow null to be included.

- Booleans

Booleans are 'true' and 'false' values.

- Null

It is a value distinct from the number zero, an empty set, or a zero-length string. Natively this might be 'null', 'nil' or 'None', depending on the language. it is often used to explicitly denote the absence of any other value.

- Objects

These are JSON data objects, standard key-value pairs. Any valid JSON object is a valid RethinkDB object, so values can be any of the basic values, arrays, or other objects. Documents in a RethinkDB database are objects. Like JSON, key names must be strings, not integers.

- Arrays

Arrays are lists of zero or more elements. Once more, anything valid in a JSON array is valid in RethinkDB: the elements may be any of the basic values, objects, or other arrays. Arrays in RethinkDB are loaded fully into memory before they're returned to the user, so they're inefficient at large sizes.

- Times

This kind of data type was not mentioned before because JSON does not support date/time data types, which is why there are so many different ways to do it. Since its specification does not mention a format for exchanging dates, there is no date format in JSON, there is only strings and a de-/serializer that decides how to map to date values.

RethinkDB has native support for millisecond-precision times with time zones. Times are integrated with the official drivers, which will automatically convert to and from a language's native time type. Queries are also timezone-aware. Since times work as indexes, events can efficiently be retrieved based on when they occurred. Time operations are pure ReQL, which means that even complicated date-time queries can be distributed efficiently across the cluster.

In RethinkDB, times can easily be inserted by simply passing a native Date object:

```
r.table('events').insert({id: 2, timestamp: new Date()}).run(conn, callback);
```

Other functions can also be used to generated date/time values as r.now (which the server interprets as the time the query was received in UTC), or construct a time using r.time, r.epochTime, or r.ISO8601. 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.

The most useful things that can be done with a date/time value are to modify it, compare it to another time, or retrieve a portion of it. RethinkDB has functions to add or subtract a duration from it (in seconds) and to subtract two times, to get a duration. Furthermore, all of the normal comparison operators are defined on times (ex: lower than, greater than, equal to). There's also the "during" command, which can check whether a time is in a particular range of times.

- Binary Objects

The JSON format natively doesn't support binary data. The binary data has to be escaped so that it can be placed into a string element (i.e. zero or more Unicode chars in double quotes using backslash escapes) in JSON. An obvious method to escape binary data is to use Base64. However, Base64 has a high processing overhead.

Binary objects are similar to BLOBs in SQL databases: files, images and other binary data. A BLOB (Binary Large Object) is a large object data type in the database system. It could store a large chunk of data, document types and even media files like audio or video files. BLOB fields allocate space only whenever the content in the field is utilized.

It's a common task for web applications to accept file uploads from users. Since RethinkDB supports a native binary object type, ReQL users are able to store binary objects directly in the database.

RethinkDB-specific data types

Streams, selections and tables are RethinkDB-specific data types.

- Streams

Streams are lists like arrays. Operations that return streams return a cursor. A cursor is a pointer into the result set. Instead of reading the results all at once like an array, a loop over the results is performed, retrieving the next member of the set with each iteration. This makes it possible to efficiently work with large result sets. Streams are read-only.

- Selections

Selections represent subsets of tables, for example, the return values of filter or get. There are two kinds of selections, Selection<Object> and Selection<Stream>, which

behave like objects or streams respectively. The difference between selections and objects/streams are that selections are writable—their return values can be passed as inputs to ReQL commands that modify the database. Some commands ([orderBy] and [between]) return a data type similar to a selection called a table_slice.

- Tables

Tables are RethinkDB database tables. They behave like selections—they're writable, as documents can be inserted and deleted in/from them.

In the ReQL API documentation it often appears the term Sequence. Sequences are not their own data type—instead, that's a collective word for all the list data types: arrays, streams, selections, and tables.

Geometry data types

In RethinkDB there are geometry data types: point, line and polygon.

ReQL geometry objects are not GeoJSON objects, but they can be converted back and forth between them with the geojson and to Geojson commands. RethinkDB only allows conversion of GeoJSON objects which have ReQL equivalents: Point, LineString, and Polygon.

Coordinates of points on the sphere's surface are addressed by a pair of floating point numbers that denote longitude and latitude. The range of longitude is −180 through 180, which wraps around the whole of the sphere: −180 and 180 denote the same line. The range of latitude is −90 (the South Pole) through 90 (the North Pole). Given two endpoints, a line in ReQL is the shortest path between those endpoints on the surface of the sphere, known as a geodesic. By default, distances are specified in meters, but an optional argument can be passed to distance functions in order to specify the measure to be used. RethinkDB has a set of commands to deal with geospatial data, for creating geospatial objects, modifying them and make calculations based on them.

The geospatial functions are implemented through this set of new geometric object data types:

- Points

It is represented by a single coordinate pair. A point can be created using the point command.

```
r.point(-117.220406,32.719464)
```

- Lines

It is a sequence of two or more coordinate pairs. It can be created the following way:

```
r.line([0,0], [0,5])
```

- Polygons

A multipoint line (at least three coordinate pairs) which does not intersect with itself and whose first and last coordinate pairs are equal. The interior of the polygon is considered filled, that is, part of the polygon. Polygons with "holes" in them, where a hole is another polygon contained by the first, can be created with the [polygonSub][] command. Lines and polygons can be specified using either point objects or sequences of two-number arrays:

```
r.line(r.point(0,0), r.point(0,5), r.point(5,5), r.point(5,0), r.point(0,0))

r.line([0,0], [0,5], [5,5], [5,0], [0,0])
```

Both of those define the same square. If polygon had been specified instead of line they would define a filled square.

Sorting order and grouped data

- Order

Arrays (and strings) sort lexicographically. Objects are coerced to arrays before sorting. To order by a parameter, the orderBy command is used:

```
.orderBy('field')
```

Mixed sequences of data sort in the following order:

Arrays - Booleans – null – numbers – objects - binary objects - geometry objects – times - strings

- Group

The group command partitions a stream into multiple groups based on specified fields or functions. It returns a pseudo type named GROUPED_DATA. ReQL comments called on GROUPED_DATA operate on each group individually. It takes a stream and partitions it into multiple groups based on the fields or functions provided.

For example, if we want to group the key/values by a field value, the function to be used is the 'group':

```
.group('field')
```

Data relationship modelling in RethinkDB

There are two ways to model relationships between documents in RethinkDB:

- By using embedded arrays

We can model the relationship between landmarks and its creators by using embedded arrays as follows. Consider this example document in the table landmarks:

```
{
  "id": "7644aaf2-9928-4231-aa68-4e65e31bf219",
  "city": "Brussels",
  "name": "ULB",
  "type": "University",
  "year": "1834",
  "creators": [
    {"name": "founder 1", "field": "Sciences"},
    {"name": "founder 2", "field": "Literature"}
  ]
}
```

The landmarks table contains a document for each landmark. Each document contains information about the relevant landmark and a field “creators” with an array of creators of that landmark. In this case the query to retrieve all landmarks with their creators is simple:

```
# Retrieve all landmarks with their creators
r.db("advanced_db").table("landmarks")

# Retrieve a single landmark with its creators
r.db("advanced_db").table("landmarks").get(AUTHOR_ID)
```

As for the advantages of using embedded arrays, one of the main ones is that the queries tend to be simple. Furthermore, the data is often collocated on disk, therefore if there is a dataset that doesn’t fit into RAM, data is loaded from disk faster. Finally, any update

to the landmarks document atomically updates both the landmark data and the creator data.

The main disadvantages of using embedded arrays are related to memory issues. Deleting, adding or updating a creator requires loading the entire creator array, modifying it, and writing the entire document back 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.

- By linking documents stored in multiple tables (similar to traditional relational database systems)

We can use a relational data modelling technique and create two tables to store your data. A typical document in the landmarks table would look like this:

```
{
  "landmark_id": "7644aaf2-9928-4231-aa68-4e65e31bf219",
  "city": "Brussels",
  "name": "ULB",
  "type": "University",
  "year": "1834",
}
```

A typical document in the creators table would look like this:

```
{
  "id": "064058b6-cea9-4117-b92d-c911027a725a",
  "landmark_id": "7644aaf2-9928-4231-aa68-4e65e31bf219",
  "name": " founder 1",
  "field": "Sciences"
}
```

Every creator contains a landmark_id field that links each creator to its work. We can retrieve all creators of a given landmark as follows:

```
# If we have a secondary index on `landmark_id` in the table
`creators`
r.db("advanced_db").table("creators").
  get_all("7644aaf2-9928-4231-aa68-4e65e31bf219",
index="landmark_id")
```

```
# If we didn't build a secondary index on `landmark_id`
r.db("advanced_db").table("creators").
  filter({"landmark_id": "7644aaf2-9928-4231-aa68-
4e65e31bf219"})
```

As we have seen previously, 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. There are 3 join commands to perform this function: eqJoin, innerJoin and outterJoin. eqJoin is the more efficient among ReQL join types, and operates much faster. In a relational database, we'd use a JOIN here; in RethinkDB, we use the eq_join command. To get all creators along with the landmark information for ULB we could perform the following (note that In order for this query to work, we need to have a secondary index on the `landmark_id` field of the table `creators`):

```
r.db("advanced_db").table("landmarks").getAll("7644aaf2-9928-
4231-aa68-4e65e31bf219").eq_join(
    'id',
    r.db("advanced_db").table("creators"),
    index='landmark_id'
).zip()
```

In this example we should mention that the values for landmark_id correspond to the id field of the landmark, which allows us to link the documents.

As advantages of using multiple tables we can mention the fact that operations on authors and posts don't require loading the data for every post for a given author into memory. Therefore, there is no limitation on the number of posts, so this approach is more suitable for large amounts of data.

However there are some disadvantages of using multiple tables. First, the queries linking the data between the authors and their posts tend to be more complicated, since we have to execute a join and not only look up within the same table. Furthermore, with this approach we cannot atomically update both the landmark data and the creator data.

RethinkDB Architecture

Sharding and replication

RethinkDB uses a range sharding algorithm parameterized on the table's primary key to partition the data, not being possible to be done based on any other attribute. When the user states they want a given table to use a certain number of shards, the system, by examining the statistics for the table, finds the optimal set of split points to break up the table. The split point will be picked such that each shard contains a number of keys close to the number of documents divided by the number of shards, and the shards will automatically be distributed across the cluster. Even if the primary keys contain unevenly distributed data, the system will still pick a correct split point to ensure that each shard has a roughly similar number of documents. Split points will not automatically be changed after table creation, neither can the user set split points for shards manually. This means that if the primary keys are unevenly distributed, shards may become unbalanced, however, the user can manually rebalance shards when necessary.

The functions of sharding and replication are configured through table configurations, which let the user simply specify the number of shards and replicas per table or for all tables within a database. Users do not need to manually associate servers with tables because RethinkDB uses a set of heuristics to attempt to satisfy table configurations in an optimal way. An internal directory tracking the current state of the cluster is kept: how many servers are accessible, what data is currently stored on each server, etc. The data structures that keep track of the directory are automatically updated when the cluster changes.

Every shard in RethinkDB is assigned to a single authoritative primary replica. All reads and writes to any key in a given shard always get routed to its respective primary, where they're ordered and evaluated. Therefore, data always remains immediately consistent and conflict-free, and a read that follows a committed write is always guaranteed to see the write. However, neither reads nor writes are guaranteed to succeed if the primary replica is unavailable.

The essential trade-off exposed by the CAP theorem is introduced in the following question: in case of network partitioning, does the system maintain availability or data consistency? In RethinkDB data consistency is chosen to be maintained.

Indexing

When the user creates a table, they have the option to specify the attribute that will serve as the primary key. If the primary key attribute is not specified, by default a unique 'ID' attribute is created. Thus, 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 automatically generated.

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 very efficient because the query can immediately be routed to the right shard and the document can be looked up in the B-Tree. RethinkDB supports both secondary and compound indexes, as well as indexes that compute arbitrary expressions.

Query execution

When a node in the cluster receives a query to execute, it starts by evaluating it. 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. The bottom-most node of the stack usually deals with data access, for example, a lookup of a single document, a short range scan using an index or a full table scan. Nodes closer to the top usually perform transformations on the data, such as mapping the values, running reductions or grouping. Each node in the stack has a number of methods defined on it. Some of the most important methods define how to execute the subsets of the queries in the different servers of the clusters and how to combine the data from them into a unified result set. Further ahead it is also of major importance the method that describes how to stream data to the nodes further up in small chunks.

Two important aspects of the execution engine are that every query is completely parallelized across the cluster, and that queries are evaluated in the least strenuous way. For instance, if the client requests only one document, RethinkDB will try to do just enough work to return the referred document, and will not process every shard fully.

Write atomicity is supported on a per-document basis. This means that updates to a single JSON document are guaranteed to be atomic. However, RethinkDB has some restrictions regarding which operations can be performed atomically. For example, values obtained by executing JavaScript code, random values, and values obtained as a result of a subquery cannot be performed atomically. In addition, like most NoSQL systems, RethinkDB does not support updating multiple documents atomically.

To efficiently perform concurrent query execution, RethinkDB implementation allows that whenever a write operation occurs at the same time as read was being performed, a snapshot of the B-Tree is taken for each relevant shard and temporarily maintained different versions of the blocks in order to execute read and write operations concurrently.

Data storage

The data is organized into B-Trees, and stored on disk using the RethinkDB own storage engine. The storage engine has some strong points, including an incremental and fully concurrent garbage compactor, low CPU overhead and very efficient multicore operation, instantaneous recovery after power failure, full data consistency in case of failures, and support for multi-version concurrency control. This storage engine, used jointly with a custom B-Tree-aware caching engine, allows the treatment of file sizes many orders of magnitude greater than the amount of available memory. Thus, makes possible for RethinkDB to operate, for example, on a terabyte of data with about ten gigabytes of free RAM.

Testing RethinkDB Performance

Objectives

As shown in previous sections, RethinkDB is a technology which is built from scratch to serve real-time web applications. Yet, it is also a document-based, JSON, database and hereby we are willing to examine its ability to be used just as a normal database. We believe that many organizations, enterprises, and individuals, will examine all strong and weak points for each database in order to deploy just the perfect one answering all needs, in terms of costs, profitability, performance, availability, scalability, integration, migration, and in our case real-time push architecture as well.

Thus, after we already showed how real-world web application exploiting the ability of real-time architecture, we decided to examine RethinkDB's behaviour as a NoSQL database, compare it with other available technologies, and test its performance comparing to MS SQL Server.

Course of the experiment

1. **Finding the perfect dataset**

Our team decided to deploy an adequately big dataset while considering the fact we will need to migrate the database between the team members as well as attaching it with the submitted report. Moreover, the database structure should offer the option to be deployed in both RethinkDB and MS SQL Server for comparison matters.

In this phase, we agreed on using "*Enron's emails*" for the experiment. Fortunately, we found the dataset online deployed to MySQL. Though, as Enron's emails are more than 0.5 million emails, and based on the reasons described above, we performed the tests on 18k emails jointly with details of all employees, contacts, and references related to these emails. Surely, those are deployed in different tables as stated below.

2. **Loading and migrating data**

Here are the steps performed in this phase:

I. Converting the tables' structure and loading queries from MySQL to fit MS SQL Server.

```
use enron
CREATE TABLE employeelist (
  eid int IDENTITY(1,1) PRIMARY KEY,
  firstName varchar(31) NOT NULL DEFAULT '',
  lastName varchar(31) NOT NULL DEFAULT '',
  Email_id varchar(31) NOT NULL unique DEFAULT '',
  Email2 varchar(31) DEFAULT NULL,
  Email3 varchar(31) DEFAULT NULL,
  EMail4 varchar(31) DEFAULT NULL,
  folder varchar(31) NOT NULL DEFAULT '',
  estatus varchar(50) DEFAULT NULL
  );

  CREATE TABLE emessage (
  mid int PRIMARY KEY DEFAULT '0',
  sender varchar(127) NOT NULL DEFAULT '',
  edate datetime DEFAULT NULL,
  message_id varchar(127) DEFAULT NULL,
  esubject text,
  body text,
  folder varchar(127) NOT NULL DEFAULT '',
);

create index Sendix
on emessage(sender);
```

Part of tables creation after converting to MS SQL Server structure

II. Retrieving 18,000 emails and all related data from other emails.
III. Designing a database with similar structure in MS SQL Server.

The database in both MS SQL Server and RethinkDB contains the following tables:

Employeelist (149 tuples)

eid: Employee-ID

firstName: First name

lastName: Last name

Email_id: Email address (primary). This one can be found in the other tables/dataframes and is useful for matching.

Email2: An additional E-Mail-Address that was replaced by the primary one.

Email3: See above

Email4: See above

folder: The user's folder in the original data dump.

estatus: Last position of the employee. "N/A"s could not be found out.

Emessage (18290 tuples)

mid: Message-ID. Refers to the rows in recipientinfo and referenceinfo.

sender: Email address (updated)

date: Date.

message_id: Internal message-ID from the mailserver.

subject: Email subject

body: Email body. Can be truncated in the R-Version!

folder: Exact folder of the e-mail including subfolders.

Recipientinfo (133903 tuples)

(Note: If an E-Mail is sent to multiple recipients, there is a new row for every recipient!)

rid: Reference-ID

mid: Message-ID from the message-table/-dataframe

rtype: Shows if the receiver got the mail normally ("to"), as a carbon copy ("cc") or a blind carbon copy ("bcc").

rvalue: The recipient's email address.

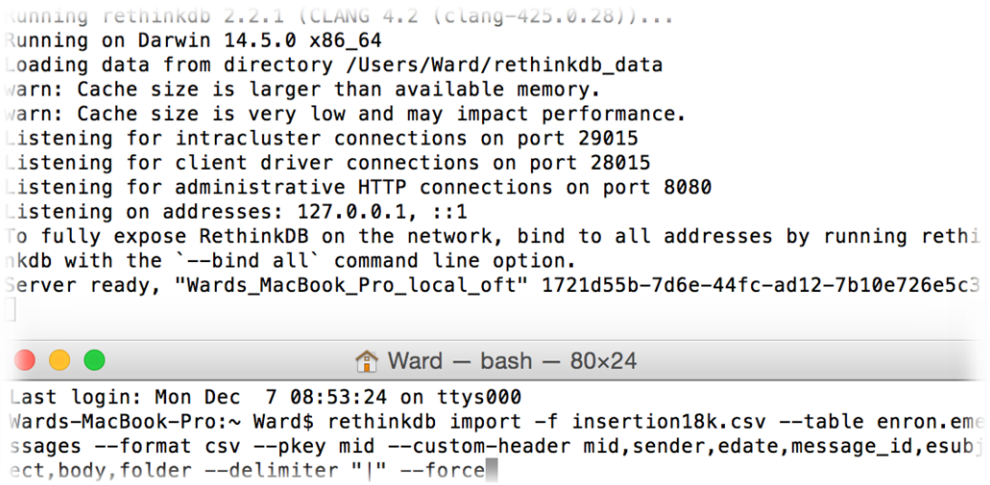
Referenceinfo (4458 tuples)

rfid: referenceinfo-ID

mid: Message-ID

reference: Contains the whole email with shortened headers.

IV. Exporting the data to CSV and using RethinkDB's import function to load it there.



3. Testing the performance

As the full results are shown in the next section, hereby we would like to demonstrate the limitations demonstrated by the proposed tests:

- Having the same schema in both RethinkDB and MS SQL Server is indeed doable as described before. Yet, due to the fact MS SQL Server expects a structured data and a fixed scheme, this may harden real tests showing the ability to use unstructured data in RethinkDB. De facto, the tests are showing the usage of structured data in both RethinkDB and MS SQL Server, rather than comparing SQL vs. NoSQL.
- Showing real-time performance demands a real-time exhibition, which is shown by real-world application in our case. That allows testing it and having the real-time functionality feeling anywhere.
- RethinkDB doesn't offer much metrics and details to show its performance, while MS SQL Server is equipped with built-in tools to analyze and monitor the performance accurately.
- The tests are performed on Mac running RethinkDB natively on Mac OS. Though, MS SQL Server was installed on a virtual machine running on top of Mac OS. In other words, RethinkDB is using the full ability of the testing machine while MS SQL Server is using the partial ability dedicated to the VM it is running on.

Results

RethinkDB vs. MS SQL Server

1) Writing data:

This phase was clear and obvious. For MS SQL Server, we wrote the data using regular data values' insertion commands addressing parsed data and suitable to be imported into MS SQL Server database. Surely, after performing the proper conversion from MySQL.

Yet, the import in RethinkDB was performed using *Homebrew* installed on *Mac OS*, connected to RethinkDB nodes and using cross-connecter based *Python* driver. In other words, the import was not as direct as in MS SQL Server.

Though, when we said the results were obvious, we referred to the actual results not the expectations. Despite the fact RethinkDB imports data using different layers and drivers, it went way faster than MS SQL Server.

The table below is illustrating a clear comparison:

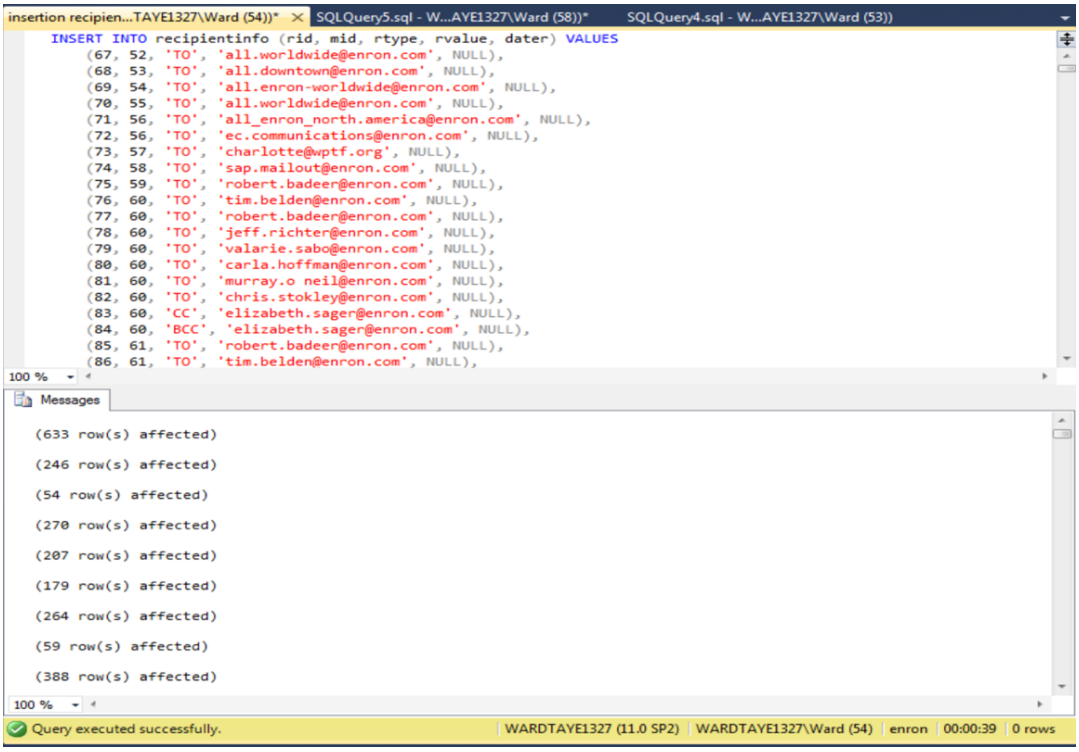
TABLE	Import time (seconds) MS SQL Server	Import time (seconds) RethinkDB	Size (tuples)
Emessages	17*	5.42 **	18290
Employees	0.009	0.002	149
Recipientinfo	39	17	133903
Referenceinfo	2	4	4458

* Due to the size of the table and limitation of MS SQL Sever, the import was performed using 3 different files which combine the full table together.

** Due to limitations of RethinkDB, the data was distributed to 9 different csv files and imported manually. The joint time was calculated using all imports.

```
Wards-MacBook-Pro:rethinkdb_data Ward$ rethinkdb import -f recp.csv --table enro
n.recipientinfo --format csv --pkey eid --custom-header rid,mid,rtype,rvalue,dat
er --delimiter "|" --force
Ignoring header row: [u'rid', u'mid', u'rtype', u'rvalue', u'dater']
[========================================] 100%
133903 rows imported in 1 table
  Done (17 seconds)
Wards-MacBook-Pro:rethinkdb_data Ward$
```

Importing data to recipientinfo in RethinkDB



Importing data to recipientinfo in MS SQL Server

2) Reading data:

For this part of the test we will be performing few queries and showing the results and the performance metrics in both technologies.

Only examples of the actual queries will be provided in this report, yet, the full codes of the queries can be annexed or provided upon request.

Query #1:

Get the recipient of all emails which include the word 'Excel'. For these emails and references show email ID, email subject, and reference id (in descending order).

SQL:

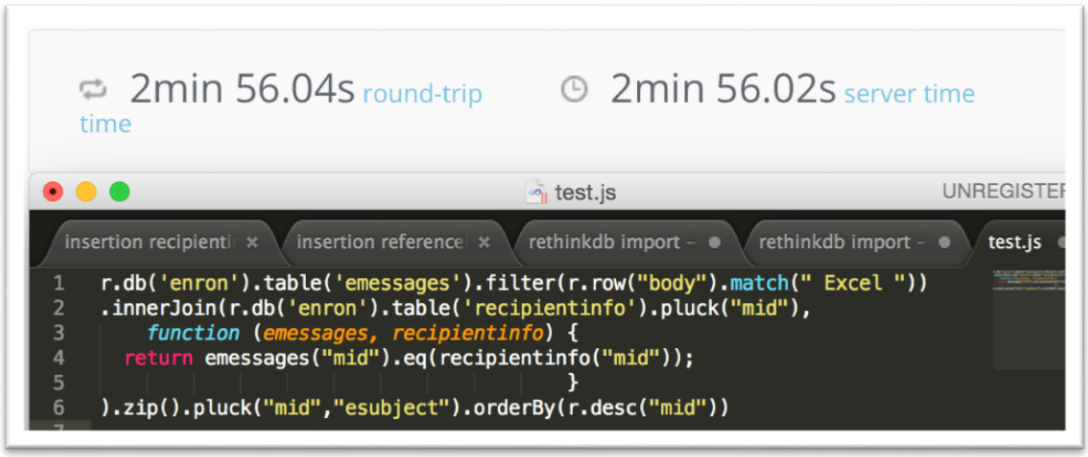
```
select e.mid,e.esubject,r.rid
from emessage e inner join recipientinfo r on e.mid=r.mid
where e.body like '%Excel%'
order by r.rid desc
```

Time: 947ms

ReQL:

r.db('enron').table('emessages').filter(r.row("body").match(" ^Excel^ ")).innerJoin(r.db('enron').table('recipientinfo').pluck("mid"),function (emessages, recipientinfo) { return emessages("mid").eq(recipientinfo("mid"));}).zip().pluck("mid","esubject").orderBy(r.desc("mid"))

Time: 176040 ms



Query1 running time in RethinkDB

Query #2:

As the execution time of the first query in RethinkDB was very high, we decided to examine the exact factors causing such issue. Our main suspicion is the Join function which RethinkDB boasts about while other NoSQL technologies decided to avoid. Thus, the next query will have small modifications over the previous to avoid the join function:

Get all emails which include the word 'Excel'. For these emails show email ID (in descending order) and email subject.

SQL Server runtime: 658ms

RethinkDB runtime: **376ms**

After disposing of the join function in RethinkDB the running time plummeted dramatically.

Query2 metrics

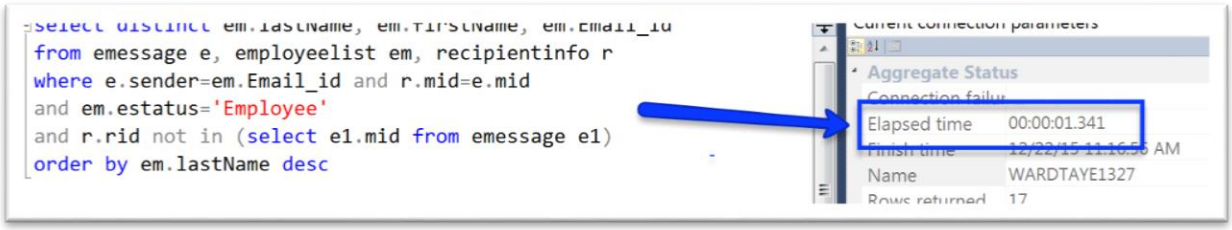
Query #3:

Get the employees names (in descendant order) and their primary emails for those who sent at least one email for non-employees (the first employee is a position, the 2nd means not working in the company at all).

SQL:

```
select distinct em.lastName, em.firstName, em.Email_id
from emessage e, employeelist em, recipientinfo r
where e.sender=em.Email_id and r.mid=e.mid
and em.estatus='Employee'
and r.rid not in
      (select e1.mid from emessage e1)
order by em.lastName desc
```

Time: 1341 ms

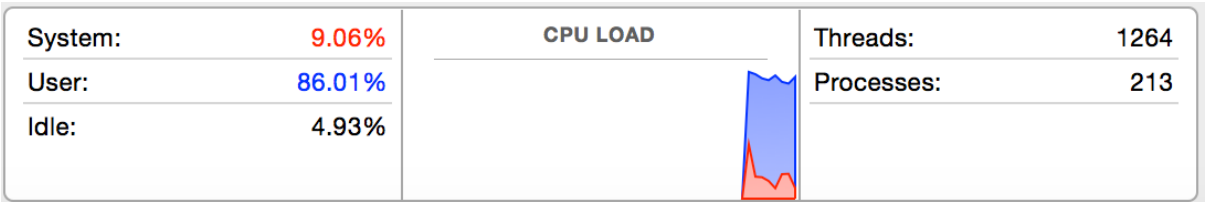


ReQL:

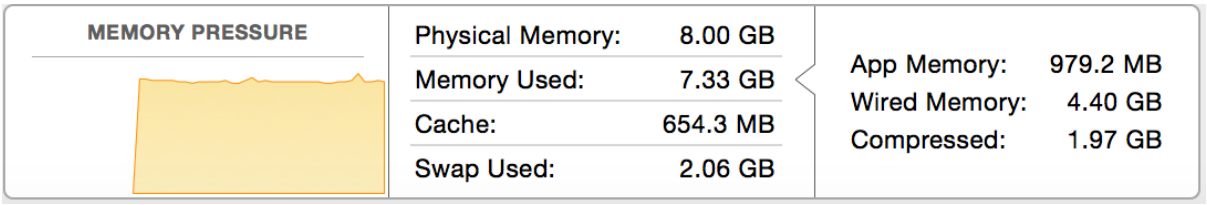
Time: Aborted after 8 minutes

We ran this query 10 times and the results were rather disappointing than surprising. Here are the results faced:

- The server's node (single node in this test) crashed completely several times. Thus, in case of not having another node for failover or even load balancing, the system may go completely down.
- High CPU and memory usage in each time we ran the query. Even though RethinkDB is running natively on Mac OS holding Intel i7 with 8 cores of 2.9Ghz 8MB Cache, 8GB of DDR3 RAM, and 512GB SSD. It jeopardized the machine by overheating and very high CPU usage (reached 93%).
- The query retrieved twice **one tuple as a result** after 8 minutes of runtime in which we had to abort the query to avoid hardware issues. Also, we received once empty result and several times even errors. Please notice, we ran the very same query and received different results!



Only 4.93% CPU Usage in Idleness. More than 95% CPU usage!



±92% Memory in use

Query #4:

Again, here we decided to dig deeper seeking for the feature getting the previous query stuck. Our main suspicious functionality is the subquery usage. Hence, we decided to execute the next query:

Get all recipients which has at least one message.

SQL:

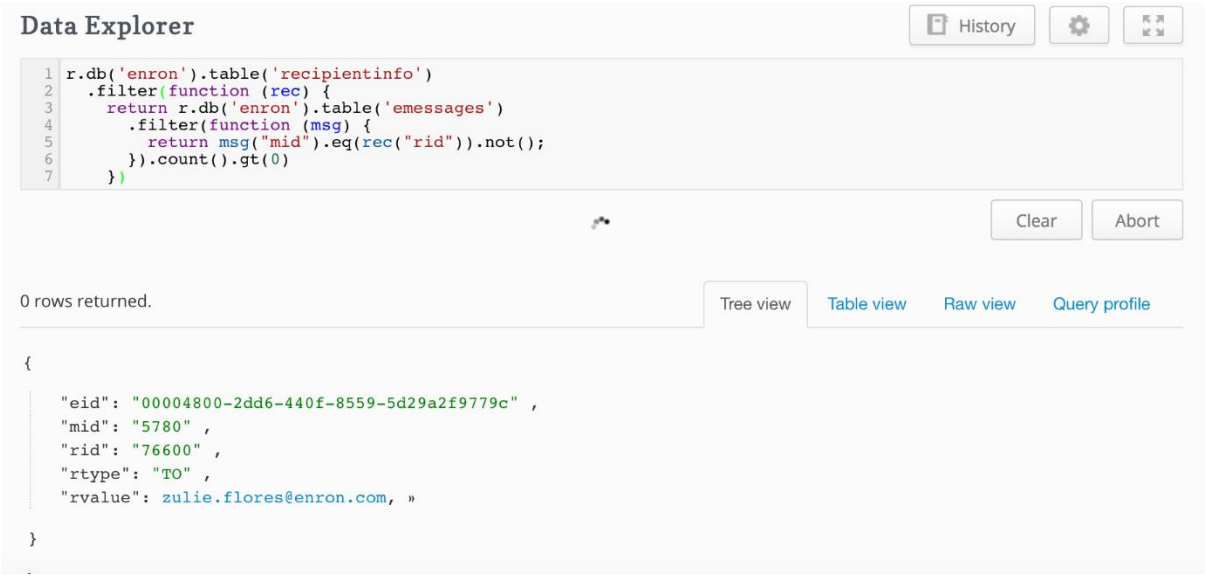
```
select *
from recipientinfo r
where exists (select * from emessage e where r.rid=e.mid)
```

Time: 234ms

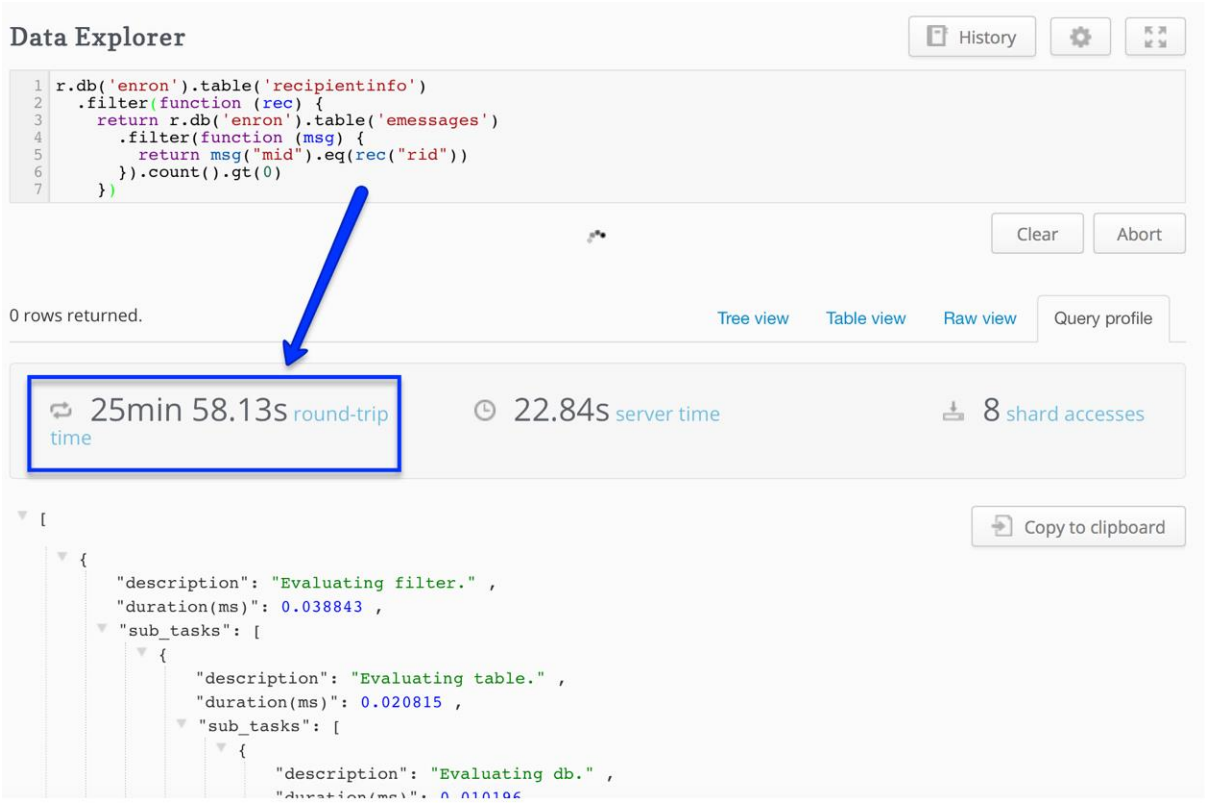
ReQL:

```
r.db('enron').table('recipientinfo')
 .filter(function (rec) {
  return r.db('enron').table('emessages')
   .filter(function (msg) {
    return msg("mid").eq(rec("rid")).not();
   }).count().gt(0)
  })
```

Time: aborted after 25 minutes.



Appears to have non rows returned since the execution is not over. Yet, you may see that there are already results appearing. In the next screenshot you can notice the high runtime!



Our findings

While RethinkDB is a fresh young six years old technology, it has some promising features and some discouraging ones. We started with reviewing the technology, moved toward its syntax and architecture just right before taking it to a competition with MS SQL Server.

RethinkDB excels at an amazing field: real-time web applications. Yet, it lacks much from stability to ACID features, operational work, and surely performance as shown above. Hereby, we will sum up to help deciding whether RethinkDB is the right thing or not:

Take RethinkDB if:

- There is a need for the real-time push architecture: with not a single doubt, there is no DB technology in the market which can give you this option in such efficiency.
- The queries to be performed are simple: we would recommend it for dealing with small number of tables while each one of them is very big.
- The data is unstructured and you are looking for easily scalable technology to allow you adding servers, nodes, and clusters easily.

Avoid RethinkDB if:

- There is no need for the real-time feature: in that case, there are many structured and unstructured data DB technologies in the market which can do the job more efficiently while having bigger community, support, stability and experienced human recourses (for instance MS SQL Server or MongoDB).
- Join tables will be performed very often.
- The database scheme includes big amount of tables.
- There is a need for more metrics for monitoring performance and data.
- You look for a specific integration between the database and other backend components, considering the fact RethinkDB is still a bit limited in that aspect.

Conclusion

NoSQL is a relatively recent technology, but evolving fast due to the recent developments in applications and its data storage and retrieval requirements. Choosing a database management system is a decision that should be made regarding the purpose of its implementation. If ACID support, for example, is needed a NoSQL database might not be the best choice as we have seen. However, for huge amounts of data or real-time features this kind of technology strong points tend to surpass its less desirable features.

RethinkDB is a good example of a Document database, however it is still a recent technology, undergoing some developments and improvements. As we have seen for some types of queries it performs better than a relational database but for some others it performs poorly. Therefore, a choice regarding its usage, or any other NoSQL technology should be made accordingly to its purpose and should be a choice well considered since the different kinds of database systems have different uses and different advantages and disadvantages.

Moreover, there are available tools and features to allow the usage of unstructured data in MS SQL Server, that option might be much encouraging for those seeking a stable DBMS while having the option of storing unstructured data.

For instance, *Filestream* allows us to store and manage unstructured data in SQL Server more easily. Initially, the accounts of FILESTREAM assumed prodigious powers of concentration and cognition, and we mortals all recoiled numbly. The FILESTREAM feature of SQL Server allows taking advantage of the Streaming capabilities of NT File System and ensures transactional consistency between the unstructured data stored in the FILESTREAM Data Container and the structured data stored in the relational tables.

Additionally, there are other available tools and features to allow such usage. Yet, as proposed before, the main question when examining a technology should be a set of priorities to be considered. Thus, we would say if real-time feature is not your main priority, while recalling the facts of existence of server-side technologies to pull data in real-time while usage almost every existing database type, RethinkDB might not be the best option.

Nevertheless, many projects involving the usage of databases include more than one technology to satisfy the best of each aspect's need. In that case, pick RethinkDB to be the technology providing real-time feature while dealing with other data in different databases.

References

Big Data:

http://www.sas.com/en_us/insights/big-data/what-is-big-data.html

http://www.ibm.com/big-data/us/en/

https://www.oracle.com/big-data/index.html

https://en.wikipedia.org/wiki/Big_data

http://searchcloudcomputing.techtarget.com/definition/big-data-Big-Data

http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation

http://www.forbes.com/sites/lisaarthur/2013/08/15/what-is-big-data/

Unstructured data:

http://www.webopedia.com/TERM/U/unstructured_data.html

http://www.webopedia.com/TERM/S/structured_data.html

https://en.wikipedia.org/wiki/Unstructured_data

http://www.robertprimmer.com/blog/structured-vs-unstructured.html

http://www.sherpasoftware.com/blog/structured-and-unstructured-data-what-is-it/

https://www.techopedia.com/definition/13865/unstructured-data

http://www.wired.com/insights/2013/09/whats-the-big-deal-with-unstructured-data/

NoSQL and RethinkDB:

https://en.wikipedia.org/wiki/NoSQL

http://www.couchbase.com/nosql-resources/what-is-no-sql

https://www.thoughtworks.com/insights/blog/nosql-databases-overview

http://www.techrepublic.com/article/nosql-is-a-complete-game-changer-declares-database-expert/

http://www.techrepublic.com/blog/10-things/10-things-you-should-know-about-nosql-databases/

http://databasemanagement.wikia.com/wiki/Relational_Database_Model

http://www.service-architecture.com/articles/database/relational_model_concepts.html

http://www.jamesserra.com/archive/2015/04/types-of-nosql-databases/

http://www.dummies.com/how-to/content/10-advantages-of-nosql-over-rdbms.html

https://www.devbridge.com/articles/benefits-of-nosql/

http://highscalability.com/blog/2010/12/6/what-the-heck-are-you-actually-using-nosql-for.html

http://stackoverflow.com/questions/1443158/binary-data-in-json-string-something-better-than-base64

http://stackoverflow.com/questions/10286204/the-right-json-date-format

http://stackoverflow.com/questions/5414551/what-is-it-exactly-a-blob-in-a-dbms-context

https://www.rethinkdb.com/docs/sql-to-reql/ javascript/

https://rethinkdb.com/docs/guide/javascript/

https://www.rethinkdb.com/api/javascript/

https://rethinkdb.com/docs/data-modeling/

http://blog.carbonfive.com/2014/03/14/rethinkdb-a-qualitative-review/

https://wiki.archlinux.org/index.php/RethinkDB

JSON:

http://www.json.org/

http://www.w3schools.com/json/

https://en.wikipedia.org/wiki/JSON

http://www.copterlabs.com/blog/json-what-it-is-how-it-works-how-to-use-it/

https://developer.mozilla.org/en-US/docs/Glossary/JSON

https://msdn.microsoft.com/en-us/library/bb299886.aspx

Enron dataset:

http://www.ahschulz.de/enron-email-data/