Key-Value stores: a practical overview

Marc Seeger
Computer Science and Media
Ultra-Large-Sites SS09
Stuttgart, Germany

September 21, 2009

Abstract

Key-Value stores provide a high performance alternative to relational database systems when it comes to storing and acessing data. This paper provides a short overview over some of the currently available key-value stores and their interface to the Ruby programming language.

Contents

1 Outline

The rest of this paper is organized as follows:
After an introduction to the general concept of key value stores, I will look at a few different implementations. Each section about an implementation will consist of a short overview of noteworthy features, followed by an overview of the installation procedure and a look at the way you can access the software using the Ruby programming language.
While there are many Key Value stores available, some of them have a limited usability to the average web-programmer. This paper will not strictly focus on big, distributed infrastructures but rather on the simpler key-value stores that could replace a conventional SQL Server such as MySQL in usual web applications. With the tested software solutions, only Cassandra (Chapter 3.4) allows for a truly distributed infrastructure.

2 Key Value stores and the NoSQL movement

SQL, the Structured Query Language was invented in the 1970s at IBM. It is a language designed to handle data that has been structured according to Edgar F. Codd's relational model described in his paper, "A Relational Model of Data for Large Shared Data Banks" and has since then developed to be the standard language for relational databases.
SQL allowed people to construct powerful queries that are able to analyse and sub-sample huge amounts of specially structured data and help operation, accounting or banking departments get indicators for the success of their newly introduced business processes or even just managing the monthly payments to their employees.
Back in the days, huge computers (not only in processing power but also in size) were used to work though seemingly vast amounts of data. This was also the time when using a computer scientist's time to optimize an SQL query was something that actually paid off in the end.
In general, SQL managed to deal with specially structured data and allowed highly dynamic queries according to the needs of the department in question.
While there are still no real competitors for SQL in this specific field, the use-case in everyday web applications is a different one.
You will not find a highly dynamic range of queries full of outer and inner joins, unions and complex calculations over large tables. You will usually find a very object oriented way of thinking. Especially with adoption of such patterns as MVC, the data in the back-end is usually not being modelled for a database, but for logical integrity which also helps people to be able to cope with understanding huge software-infrastructures.
What is being done to put these object-oriented models into relational

databases is a large amount of normalization that leads to complex hierarchies of tables and completely steers against the main idea behind object oriented programming. Servers that adhere to the SQL standard also have to implement a large portion of code that is of no use to simple data storage what so ever and only inflates the memory footprint, security risks and has performance hits as a result.
The fact that SQL allows for arbitrary dynamic queries for complex sets of data is being rendered useless by using an SQL Database only for persistent storage of object oriented data, which is what basically most applications do these days.
This is where Key Value stores come into play. Key value stores allow the application developer to store schema-less data. This data is usually consisting of a string which represents the key and the actual data which is considered to be the value in the "key - value" relationship. The data itself is usually some kind of primitive of the programming language (a string, an integer, an array) or an object that is being marshalled by the programming languages bindings to the key value store. This replaces the need for fixed data model and makes the requirement for properly formatted data less strict.
To make this more clear, here's a short example (using ruby and the simple but limited "pstore" standard library) that should make that concept clear to any programmer.

```
require "pstore"

store = PStore.new("data-file.pstore")
store.transaction do  # begin transaction
  # load some data into the store...
  store[:single_object] = "Lorem ipsum dolor sit amet..."
  store[:obj_heirarchy] = { "Marc Seeger" => ["ruby", "nosql"],
                            "Rainer Wahnsinn"  => ["php", "mysql"] }
end                   # commit changes to data store file
```

In this short example, we have created a key-value store on disk called "data_file.pstore" and added two objects.
The object with the key ":single_object" is basically a simple string, ":obj_hierarchy" on the other hand is a more complex datatype. It is a Hash which contains arrays of strings.
Accessing those objects would be as simple as e.g. assigning the object to a variable, exactly the same way we'd usually interact with hashes in Ruby:

```
my_var = store[:single_object]
```

Syntax for different key-value stores will obviously differ, but they all function in a conceptually similar manner.

They all allow storage of arbitrary data which is being indexed using a single key to allow retrieval. The biggest difference for the "simpler" stores is the way you can (or cannot) authenticate or access different stores (if possible). While the speed advantages in storing and retrieving data might be a reason to consider it over common SQL Databases, another big advantage that emerges when using key-value stores is that the resulting code tends to look clean and simple when compared to embedded SQL strings in your programming language. This is something that people tend to fight with object-relational mapping frameworks such as Hibernate or Active Record. Having an object relational mappers basically seems to emulate a key value store by adding a lot of really complex code between an SQL database and an object-oriented programming language.
A whole community of people come together under the "NoSQL" tag and discuss these advantages and also disadvantages of using alternatives to relational database management systems. One of the more recent meet-ups featured prominent talks from people working for Facebook, Stumbleupon, Linkedin and Last.fm.
People interested in nosql-related technology can look at the videos and slides of that meet-up can at Johan Oskarsson's blog[1] (he's a developer at last.fm).
It has to be noted that the "nosql" movement not only includes the "newly emerging" key-value stores I'm discussing in this paper, but also does include object stores such as e.g. BerkleyDB, O2, GemStone or Statice which all date back to the 70s and 80s and are as "industrial strength" as it gets. One important thing to consider is, that the nosql movement is not against SQL in general. The main idea behind the movement is, that it not okay to automatically assume some kind of SQL server when people talk about data persistence. SQL is much more than just the ability to persistently store data in a specified manner, in fact, it is so much more that it is basically an overkill in most situations. Showing alternatives to the current SQL-dominated persistence landscape is one of the main goals to the nosql community.

3 Contestants

While there are lots of Key Value Stores that focus an scalability, synchronization over several servers and fault tolerance, discussing those individual properties could each fill a paper on their own. I would like to give an entry level introduction on the "simpler" Key-Value stores that focus on simply replacing SQL Servers (read: usually MySQL) for the usual consistency matters. The need for distributing/sharding data because of performance considerations is not as urgent as with larger SQL-Server instances. Key

[1] `http://blog.oskarsson.nu/2009/06/nosql-debrief.html`

Value stores have remarkably high insert/read rates compared to "usual" SQL Servers. While some of the contestants DO support master-slave replication (e.g. redis) or are actually distributed (e.g. Cassandra), the main focus of this paper will not be on those features but rather on the ease of use for simple projects.
I decided to use the following contestants in accordance to their perceived popularity in the, for the most part web-centric, Ruby community:

- CouchDB (Chapter 3.1)
- Tokyo Cabinet (Chapter 3.2)
- Redis (Chapter 3.3)
- Cassandra (Chapter 3.4)

If you have already studied larger systems, you might know the piece of software that is basically the heart of many of the big players in web 2.0 called "memcached". We will NOT look at memcached because it is not designed to be persistent, it just holds everything in memory without saving since its main objective is to function as a caching mechanism. There is a modifications called "memcacheDB", but it doesn not, in my opinion, offer that much differences over the chosen candidates to actually justify adding it to the paper. People looking for a mixture between redis and cassandra might want to take a short look at the projects homepage[2] though.
The key-value stores we want to take a look at can be used as the main DB for the application as they always support persistence as part of their design.
Before discussing each of those servers in detail, here is a short overview:

CouchDB is a key-value store implementation in Erlang that uses HTTP to communicate with clients and Javascript to generate "views"

Tokyo Cabinet Tokyo Cabinet (and its network interface, Tokyo Tyrant) is a key-value store that supports 3 modes of operation: hashtable mode, b-tree mode, and table mode.
It is used by the high-load environment of the Japanese Facebook-equivalent Mixi
Tokyo Cabinet is written in C.

Redis An in-memory key-value store focusing on performance, implemented in C.

Cassandra An extended key-value store originally developed by Facebook. It was developed by some of the key engineers behind Amazon's famous Dynamo database.

[2]MemcacheDB homepage: `http://www.memcachedb.org/`

Here is a quote describing the software, directly taken from the project's homepage[3]:
"Cassandra is a highly scalable, eventually consistent, distributed, structured key-value store. Cassandra brings together the distributed systems technologies from Dynamo and the data model from Googles BigTable. Like Dynamo, Cassandra is eventually consistent. Like BigTable, Cassandra provides a ColumnFamily-based data model richer than typical key/value systems"

3.1 CouchDB

3.1.1 About

From the projects homepage[4]:

> Apache CouchDB is a distributed, fault-tolerant and schema-free document-oriented database accessible via a RESTful HTTP/JSON API. Among other features, it provides robust, incremental replication with bi-directional conflict detection and resolution, and is queryable and indexable using a table-oriented view engine with JavaScript acting as the default view definition language. CouchDB is written in Erlang, but can be easily accessed from any environment that provides means to make HTTP requests. There are a multitude of third-party client libraries that make this even easier for a variety of programming languages and environments.

CouchDB is coded in Erlang. The only bigger C parts are the IBM unicode library and the spidermonkey javascript engine (which also powers e.g. Firefox).
CouchDB uses a simple RESTful HTTP/JSON approach to interfacing with the outside world. Seeing as HTTP is probably one of the most supported protocols, old infrastructure such as loadbalancers, http caches or SSL proxies can still be used in connection with CouchDB.
This allows companies to harvest existing knowledge to easily secure and speed up their HTTP-based CouchDB infrastructure without having to hire an external database specialist.
Another interesting feature of CouchDB is the fact that developers can use "views" to query the software for data. Views are basically map and reduce functions that are implemented in Javascript and sent to CouchDB. Querying these views is simply a matter of making the appropriate HTTP

[3] `http://incubator.apache.org/cassandra/`
[4] `http://couchdb.apache.org`

GET request. CouchDB keeps these views incrementally up to date by simply running new inserts through the map and reduce functions. There are some limitations imposed when creating these views to keep the incremental updates working. Here's a short excerpt from the CouchDB wiki[5]:

> The restriction on map functions is that they must be referentially transparent. That is, given the same input document, they will always emit the same key/value pairs. This allows CouchDB views to be updated incrementally, only reindexing the documents that have changed since the last index update.

This way of querying for data allows web developers to use their HTTP and Javascript skills to interact with CouchDB after just a few minutes of research.

3.1.2 Installation

The installation process is pretty much the usual (taken from the CouchDB readme file[6]):

```
svn co http://svn.apache.org/repos/asf/couchdb/trunk couchdb
cd couchdb
./bootstrap && ./configure
make
sudo make install
```

Some of the dependencies are e.g. Erlang/OTP, GCC, make, OpenSSL and Spidermonkey (the Javascript engine powering Mozilla Firefox).
There are also a lot of prebuilt packages for your favourite Linux package manager available.
According to the CouchDB documentation, it is also possible to compile and run CouchDB on windows using cygwin and the Microsoft C compiler.

3.1.3 Ruby-Interface

There are a lot of different ways of interacting with CouchDB from within Ruby. Seeing as CouchDB supports a simple REST protocol, the "getting started with Ruby" part of the couchDB wiki [7] uses simple HTTP calls and JSON parsing to interact with CouchDB.
While this might work, it probably isn't the most elegant and productive solution, this is why there are a few libraries that encapsulate the REST logic

[5] CouchDB wiki - views:`http://wiki.apache.org/couchdb/Introduction_to_CouchDB_views`

[6] CouchDB Readme: `http://svn.apache.org/viewvc/couchdb/trunk/README?view=markup`

[7] `http://wiki.apache.org/couchdb/Getting_started_with_Ruby`

in something more Ruby-like. Here's a quote from the previously mentioned CouchDB wiki:

> For a simple Ruby wrapper around CouchDB's RESTful API, see CouchRest, which keeps you fairly close the metal, as well as having a few helpful wrappers, like a builtin view pager and companion libraries like CouchModel (for document id and lifecycle management) and Slipcover (for parallel query execution).

We will now take a short look at CouchRest, which is available as a gem on github (*gem install jchris-couchrest -s http://gems.github.com*).
This is the description as given in CouchRest's Readme file:

> CouchRest is based on CouchDB's couch.js test library, which I find to be concise, clear, and well designed. CouchRest lightly wraps CouchDB's HTTP API, managing JSON serialization, and remembering the URI-paths to CouchDB's API endpoints so you do not have to.
>
> CouchRest is designed to make a simple base for application and framework-specific object oriented APIs. CouchRest is Object-Mapper agnostic, the parsed JSON it returns from CouchDB shows up as subclasses of Ruby's Hash. Naked JSON, just as it was mean to be.

As the way CouchREST interacts with CouchDB is nothing out of the ordinary and as I am generally a fan of code examples, I decided to simply use the "quick start" example taken from the CouchRest Readme file to show the basic interaction with the library:

```
# with !, it creates the database if it doesn't already exist
@db = CouchRest.database!("http://127.0.0.1:5984/couchrest-test")
response = @db.save_doc({:key => 'value', 'another key' => 'another value'})
doc = @db.get(response['id'])
puts doc.inspect
```

This is how you save more than one entry to the store:

```
@db.bulk_save([
    {"wild" => "and random"},
    {"mild" => "yet local"},
    {"another" => ["set","of","keys"]}
  ])
```

Another way of using CouchDB that might be interesting for legacy applications is the fact, that Datamapper (the ORM used e.g. with the merb webframework) and Active Record (the ORM that is used with the Rails

webframework) can in theory both be used in connection with CouchDB by using matching adapter libraries. This way, legacy Web Applications can simply be migrated from e.g. MySQL to a CouchDB environment.
Thanks to the way CouchDB can be accessed by simple HTTP GET calls, it would also be possible to query the Database directly with the end-users webbrowser. The returned JSON documents can simply be parsed and properly inserted into the webpage using Javascript on the End-User side. An example of this kind of "stand alone application" can be seen over at "SoFa"[8], the Standalone CouchDB Blog (also used by the O'Reilly CouchDB book)

3.2 Tokyo Cabinet

3.2.1 About

The Tokio* Universe is separated in 3 main software projects that provide different functionalities:

Tokyo Cabinet itself is only the library dealing with all of the data structures (B+tree's, hash tables, ...) used to organize content. It is supposed to provide high performance and scalability while still remaining simple in its concepts. This also allows projects to use Tokyo Cabinet only as a fast way to persistently store their data without any network overhead by simply linking to the Tokyo Cabinet library.

Tokyo Tyrant is the networking interface that provides outside access to the data stored in Tokyo Cabinet. It features a simple client-server model and allows multiple applications to access the same Tokyo Cabinet Database. Besides its own effective binary protocol, it also supports the memcached protocol and HTTP. It's also responsible for asynchronous replication with several different Tokyo Cabinets. A really important feature is the ability to use Lua to script arbitrary funcionality into Tokyo Tyrant. A good overview about the Lua scripting can either be found in the official documentation[9] or the short overview by Ilya Grigorik[10]. This basically allows us to add any atomic operation we'd like to have to the key-value store.

Tokyo Dystopia is an optimized high-speed full-text search engine for Tokyo Cabinets and is e.g. used to search for friends in the Japanese Facebook equivalent mixi.jp

[8] `http://github.com/jchris/sofa`

[9] Official Tokyo Tyrant documentation: `http://tokyocabinet.sourceforge.net/tyrantdoc/#luaext`

[10] Ilya Grigorik's blog post about tokyo cabinet: `http://www.igvita.com/2009/07/13/extending-tokyo-cabinet-db-with-lua/`

3.2.2 Installation

Tokyo Cabinet and Tokyo Tyrant both follow the usual 3 step process of ./configure, make, make install.
If you want to use the Lua scripting capabilities of tokyo tyrant, you would have to use ./configure –enable-lua.

3.2.3 Ruby-Interface

The official homepage[11] distributes the ruby bindings as a tarball which can be compiled and installed.
There are however, a number of gems available via the ruby packaging system. There are also adapters to object relational mappers like Datamapper ("DM") that would allow storage of information in toyko cabinet without changing more than 2 or 3 lines the sourcecode responsible for storing the data.
Here are the results of a simple query over the default gem repositories that come with rubygems:

```
# gem search -r tokyo

*** REMOTE GEMS ***

actsasflinn-ruby-tokyotyrant (0.2.0)
careo-tokyocabinet (1.21)
careo-tokyotyrant (1.3.0.1)
jackowayed-rufus-tokyo (0.1.13.2)
joshbuddy-tokyo_cache_cow (0.0.3)
makoto-dm-tokyo-cabinet-adapter (0.0.2)
nofxx-tokyo_store (0.3.0)
ntalbott-rufus-tokyo (0.1.14)
rubysouth-tokyo_model (0.0.4)
rufus-tokyo (1.0.0)
scottburton11-tokyomapper (0.1.1)
shanna-dm-tokyo-adapter (0.3.2)
shanna-dm-tokyo-cabinet-adapter (0.1.6)
```

3.3 Redis

3.3.1 About

This is taken directly from the readme file that is shipped with the redis sourcecode and describes the program in a way the creators see it. I've

[11] Tokyo Cabinet Homepage: `http://1978th.net/tokyocabinet/`

taken the liberty of highlighting the important things:

> Redis is a database. To be more specific redis is a very simple database implementing a dictionary where keys are associated with values. For example I can set the key "surname_1992" to the string "Smith". The interesting thing about Redis is that values associated to keys are **not limited to simple strings**, they can also be lists and sets, with a number of **server-side atomic operations** associated to this data types.
> Redis takes the **whole dataset in memory**, but the dataset is persistent since from time to time Redis writes a dump of the dataset on disk asynchronously. The dump is loaded every time the server is restarted.
> Redis can be configured to **save the dataset after a given number of seconds elapsed and changes** to the data set. For example you can tell Redis to save after 1000 changes and at least 60 seconds since the same save. You can specify a number of this combinations.
> Because data is written asynchronously, **If a system crash occurs the last few queries can get lost** (that is acceptable in many applications). Redis supports **master-slave replication** from the early days in order to make this a non issue if your application is of the kind where even few lost records are not acceptable.

While this might seem to look pretty fine, it has to be taken with caution. Redis makes heavy use of RAM and only flushes out its data to disk asynchronously. This makes it less than ideal for important data as there is a high probability of data-loss in case of a sever crash (power loss / faulty PSU / bluescreen ...). On the other hand, this makes Redis one of the fastest key-value stores in the field and very usable for the common "not that important" data that one can find in most current web applications, especially in social networks etc. In the rare case of a crash, it will not be that important to save the last status update somebody posted or a single tagging of a single person in a single picture. It is how ever of vital importance to deliver a pleasant usage speed to the user without database backends slowing down due to unneeded transactional integrity etc.
With redis, you can expect over 100.000 operations (insert/read/increment/...) per second on a normal linux box with 50 concurrent clients.
There are some misconceptions on the level of sharding support that redis provides. When talking about sharding and redis, nearly all of the work gets done by the client library. Basically, a cryptographic hash over the key is calculated and a server is chosen according to the results of that, it is a client side setup though.

One of the impressive features is the programming languages support. At the time of writing the official Redis google code page lists the existence of bindings for:

- Ruby
- Python
- PHP
- Erlang
- Tcl
- Perl
- Lua
- Java (a JDBC and a JCA adapter are available)

A quote from the redis page:

> All the client libraries are shipped in the same tar.gz together with Redis but if you want the latest versions check the main repositories. An exception is for both the Java clients that are not still included in the tar.gz since they reached stability recently.

3.3.2 Installation

The installation of redis consists only of untaring the source file and using make to build the server / benchmark suite.

3.3.3 Ruby-Interface

To use redis from within ruby, there is a gem (the ruby term for "library") called *redis* (no surprises here...) which allows easy ruby-like access to the key-value store.
This short example is taken from "Redis and Ruby", a blog post on programmersparadox.com[12]. It will connect to a redis server running on the default redis port (6379) on localhost and set a key:

```
require "rubygems"
require "redis"
r = Redis.new
r.delete("first_key") #clear it out, if it happens to be set
puts "Set the key 'first_key' to 'hello world'"
```

[12] http://www.programmersparadox.com/2009/06/02/redis-and-ruby/

```
r["first_key"] = "hello world"
puts "The value of 'first_key' is:"
puts r["first_key"]
```

One of the nice things is, that redis is able to work with a plaintext protocol using telnet:

```
$ telnet localhost 6379
Trying 127.0.0.1...
Connected to localhost.
Escape character is '\^ ]'.
SET foo 3
bar
+OK
GET foo
$3
bar
```

What all of this means:

Command	Meaning
SET foo 3	set the key "foo" to the next 3 characters
bar	these are the 3 bytes followed by a newline character
+OK	the server tells us that it will insert the data (asynchronously)
GET foo	get the value for the key "foo"
$3	the value will be the next 3 bytes
bar	our result

Some other interesting server-side atomic operations are pretty much self explanatory. Having these operations allows redis to do concurrent access without having to deal with a lot of the most common locking problems:

SET key value set a key to a string value

GET key return the string value of the key

GETSET key value set a key to a string returning the old value of the key

MGET key1 key2 ... keyN multi-get, return the strings values of the keys

SETNX key value set a key to a string value if the key does not exist

INCR key increment the integer value of key

INCRBY key integer increment the integer value of key by integer

DECR key decrement the integer value of key

DECRBY key integer decrement the integer value of key by integer

EXISTS key test if a key exists

DEL key delete a key

TYPE key return the type of the value stored at key

As you can see, it's pretty much the basic things that people who have already dealt with a associative arrays will be familiar with. The only interesting thing is the possibility to increment and decrement a key using an atomic operation. You can see some of the other operations supported by redis over at the projects wiki[13].

3.4 Cassandra

3.4.1 About

Cassandra is a hybrid non-relational database comparable to Google's BigTable (with the difference that is uses a DHT instead of a central server). Cassandra was initially created by Facebook and later transferred to the open-source community. It currently is an Apache Incubator project. Facebook uses Cassandra as email search system where, as of last summer, they had 25TB and over 100m mailboxes.
Here is a quote describing the software, directly taken from the project's homepage[14]:

> "Cassandra is a highly scalable, eventually consistent, distributed, structured key-value store. Cassandra brings together the distributed systems technologies from Dynamo and the data model from Googles BigTable. Like Dynamo, Cassandra is eventually consistent. Like BigTable, Cassandra provides a ColumnFamily-based data model richer than typical key/value systems".

Avinash Lakshman of Facebook summarized it on his first slide of his nosql presentation on Cassandra[15] in one sentence: Cassandra is a structured storage system over a P2P network.
Cassandra's design goals are not the same as for most of the other key value stores since it does not primarily focus on simplicity and an simple data model, but according to Mr Lakshman's presentation rather these:

- High availability

[13] Redis Wiki - Command Reference: `http://code.google.com/p/redis/wiki/CommandReference`

[14] `http://incubator.apache.org/cassandra/`

[15] Avinash Lakshman's slides: `http://www.slideshare.net/Eweaver/cassandra-presentation-at-nosql` or `http://static.last.fm/johan/nosql-20090611/cassandra_nosql.pdf` and video: `http://vimeo.com/5185526`

- Eventual consistency (trade-off strong consistency in favour of high availability)
- Incremental scalability
- Optimistic Replication (Knobs to tune trade-offs between consistency,durability and latency)
- Low total cost of ownership
- Minimal administration

While a lot of interesting technology plays part in the realization of these goals, explaining things like the gossip protocol (used for cluster membership) or the read/write properties would be enough for a paper on its own. If you're interested in these, I recommend you to look at the previously mentioned talk by Avinash Lakshman and the accompanying slides.
As you can see, Cassandra is pretty different when you compare it to the other contestants in this paper. The first thing that differs from the other applications is, that it is implemented in Java. Another huge difference is the fact, that it was designed to be distributed using an "eventually consistent" approach as its data model, something that is also used by Amazon's cloud infrastructure. The details about this can be looked up on the weblog of Werner Vogel. He is Amazon's CTO and runs the weblog `http://www.allthingsdistributed.com` where he discusses the topic in his articles "Eventually Consistent"[16] and "Eventually Consistent - Revisited"[17]. It can be summarized as a way of building large distributed systems that approaches trade-offs between consistency and availability. It also leads to the situation that for Cassandra, speed is not only about the amount of inserts per second, but also about the way more capacity can be added to the system and the connected performance-costs.
The motivation behind Cassandra were basically two problems (taken from Jonathan Ellis's OSCON 09 talk[18]):

- Scaling reads to a relational database is hard
- Scaling writes to a relational database is virtually impossible

Since I did not go into detail about more of the inner workings of Cassandra, I think looking back at Evan Weavers blog post "Up and running with

[16]Werner Vogel - Eventually Consistent: `http://www.allthingsdistributed.com/2007/12/eventually_consistent.html`
[17]Werner Vogel - Eventually Consistent Revisited: `http://www.allthingsdistributed.com/2008/12/eventually_consistent.html`
[18]Jonathan Ellis's OSCON 09 talk: `http://www.slideshare.net/jbellis/cassandra-open-source-bigtable-dynamo`

Cassandra"[19], I should at least mention the things he titled "features that help put Cassandra above the competition" and relate in one way or another to these two central problems (directly quoted from his Evan Weavers blog post):

Flexible schema with Cassandra, like a document store, you do not have to decide what fields you need in your records ahead of time. You can add and remove arbitrary fields on the fly. This is an incredible productivity boost, especially in large deployments.

True scalability Cassandra scales horizontally in the purest sense. To add more capacity to a cluster, turn on another machine. You do not have restart any processes, change your application queries, or manually relocate any data.

Multi-data center awareness you can adjust your node layout to ensure that if one data center burns in a fire, an al

Range queries unlike most key/value stores, you can query for ordered ranges of keys.

List data structures super columns add a 5th dimension to the hybrid model, turning columns into lists. This is very handy for things like per-user indexes.

Distributed writes you can read and write any data to anywhere in the cluster at any time. There is never any single point of failure.

3.4.2 Data model

Since Cassandra is more than just "another Key-Value store", it is necessary to explain the data-structures that Cassandra uses to represent data. Cassandra uses those building blocks to form something like a 4 (or 5) dimensional hash (aka: associative array). It basically means that values can be a collection of other key+value pairs:

- a keyspace
- a column family
- a key
- an (optional) super column
- a column

[19]Evan Weaver - Up and running with Cassandra: `http://blog.evanweaver.com/articles/2009/07/06/up-and-running-with-cassandra/`

The **Keyspace** is the highest hierarchy of data, and there's typically one key space per application (defined in the storage-conf.xml file before startup).
The **Column family** is the next layer below the keyspace. Data in Cassandra can be stored in columns rather than the usual rows. For the advantages over row-oriented RDBMS, I'd suggest the matching Wikipedia article[20]. Column families are addressed by a specific unique Key and allow for often queried columns to be stored together. Each column family is stored as a separate file on disk. This allows for a certain "data structure design" and has also to be done before the start of the application in the storage-conf.xml file. This means that when you select a column family in cassandra, you will receive a group of key+value pairs just like in a multi dimensional hash
The **Key** is identical to other key-value stores. It is automatically indexed and allows for fast data access. It is defined on the fly and does not have to be pre-set in a configuration file. You can also query over a ranges of keys in a column family.
The **Super columns** are basically only groupings of regular columns. They allow you to organize related, sorted column data under a specific unique name. As with keyspaces and column families, they have to be set up in the storage-conf.xml file before start of the application
The **column** is basically where Cassandra stores the raw data

This is the point where I'd like to quote Kirk Heines from the Engine Yard team. In his post on the Engine Yard blog titled "Cassandra and Ruby: A love affair?"[21] he uses a pretty nice description of the Cassandra data model:

> As you can see, using Cassandra is more complicated than using a simple key-value store, even one like Tokyo Cabinet which builds a table model on a row based key-value system. However, just like the first time you tried to learn recursion, once your perspective shifts so that you can grok it, Cassandras structure naturally lends itself to a whole class of otherwise tricky, high labour queries.

3.4.3 Installation / Ruby-Interface

Seeing as we want to check the integration within the ruby programming language, we can simply install the Cassandra server as part of the Cassandra ruby-gem:

```
gem install cassandra
```

That way, we will install the server itself (requires a Java Runtime Environment) and the matching ruby gem.

[20] `http://en.wikipedia.org/wiki/Column-oriented_DBMS`
[21] `http://www.engineyard.com/blog/2009/cassandra-and-ruby-a-love-affair/`

After the installation process, you should be able to use Cassandra by simply requiring the matching gem in ruby:

```
require 'rubygems'
require 'cassandra'
```

The gem utilizes the main way that is usually used to interact with Cassandra: its Thrift interface.
A short quote from the Cassandra wiki[22]:

> In short Thrift allows you easily setup service clients and servers in various programming languages. It generates code from a Thrift file describing the service.

Seeing as Cassandra is very complex compared to the other key value stores and posting pieces of sourcecode would probably not do any good, I'd like to give some more links to excellent blog posts explaining the concept a bit further:

- Even Weavers post about Cassandra: `http://blog.evanweaver.com/articles/2009/07/06/up-and-running-with-cassandra/`
- The Engine Yard blog entry by Kirk Heines: `http://www.engineyard.com/blog/2009/cassandra-and-ruby-a-love-affair/`
- A video directly from the Facbook Engineering Team: `http://www.new.facebook.com/video/video.php?v=540974400803`
- The Cassandra Wiki: `http://wiki.apache.org/cassandra/`

[22] Cassandra wiki - thrift: `http://wiki.apache.org/cassandra/ThriftInterface`

4 Conclusions

The nosql community basically wants to point out that persistence does not necessarily mean SQL. I personally think that key-value stores are a really good approach of bringing this argument across.
As usual in computer science, there are no "silver bullets" when it comes to solving a problem and most solutions will end with a trade-off.
With key-value stores, you'll end up losing the ability to run SQL-like ad-hoc queries. You will however be able to eliminate the pain of having to create complex SQL statements.
While object-relational mappers such as Java's Hibernate or Ruby's Active Record do a pretty decent job of hiding SQL from the developer, you'll still have to deal with the SQL database logic in some form or another (e.g. setting up those tables, thinking about the database layout and modelling your application after it).
Having a simple key-value interface to your persistent data is something that any programmer can learn dealing with within a few minutes as most programming languages already offer similar data-structures (usually called hashes or dictionaries).
Seeing as the key-value store technologies are pretty young and have to compete against, in computer terms "ancient", SQL database competitors, I'd say they're doing a good job in delivering a possibility to create a simply accessible way of storing and retrieving data. Another advantage key-values stores might have compared to SQL databases is their simplicity. The concepts behind their storage mechanisms are usually pretty easy to understand and do not require complex SQL optimization techniques that are pretty close to the technology used in building compilers for programming languages.
While there certainly are probably as many arguments speaking for key-value stores as there are against them, the introduction of some new concepts that seek to solve the problems SQL is facing is considered to be a good thing.
Concerning the contestants, it is hard to compare them as they cater to a different audience. I myself will probably take a close look at Redis when it comes to easy interaction with ruby and ease of use for my own private projects. It is difficult to say when the requirements start outgrowing "simple" software such as Redis and the need for a distributed architecture starts appearing. Seeing as there are so many more alternatives that I simply did not have the time to discuss in this short overview (e.g. MongoDB, Hybertable, HBase, ThruDB, MemcacheDB, Project Voldemort, Scalaris, Dynomite, Ringo, Kai, Groonga, Senna, Lux IO, Tx, repcached, Cagra, kumofs, ROMA, Flare, ...) and there are even Key-Value Store meetings

being held in Japan [23], the future for key value stores seems to look pretty promising and is certainly something that software architects ought to keep an eye on.

[23] http://blog.plathome.com/2009/02/first-key-value-storage-meeting-held.html