# Yes or NoSQL

Jonathan Berkhahn, Casey Link
David Mazary, Andy Street,
Val Komarov

05/08/2011

- Introduction
  - Background
  - Motivation
  - Approach
- Demo Application
  - Virtual Machines
  - Dataset
  - Implementation
  - Query Types
    - Partial Match Query
    - Date Range Query
    - Date and Symbol Query
- Cassandra
  - Tunable Consistency
  - Other Benefits
    - Homogenous Clusters
    - Cross datacenter replication
  - Understanding the Data Model
    - Super Columns
  - Designing a Data Model for a NoSQL Database
    - Stocks Data Model
    - Partial Match Query
- HBase
  - Overview of Data Model
  - Notes on Development
    - API / Library
  - Ease of setting up a cluster
  - Performance
- MySQL
  - Overview of Data Model
  - Notes on Development
  - Ease of setting up a cluster
    - API / Library
  - Performance
- Benchmarking

Conclusion
Amazon EC2 Instructions
Getting started
Setting up EC2
Launching a node
Installing Web Server
Installing Cassandra
Installing HBase

# Introduction

## Background

Although not the first, Google's BigTable paper initiated the current trend of modifying the attributes of data-storage systems to give up some attributes of traditional RDBMS's in order to create massively-scalable cloud data storage systems. Eric Brewer's *Toward Robust Systems* established a Consistency-Availability-Partition tolerance (CAP) theorem stating that data storage systems can have only two of the properties of data consistency, data availability, or partition tolerance.

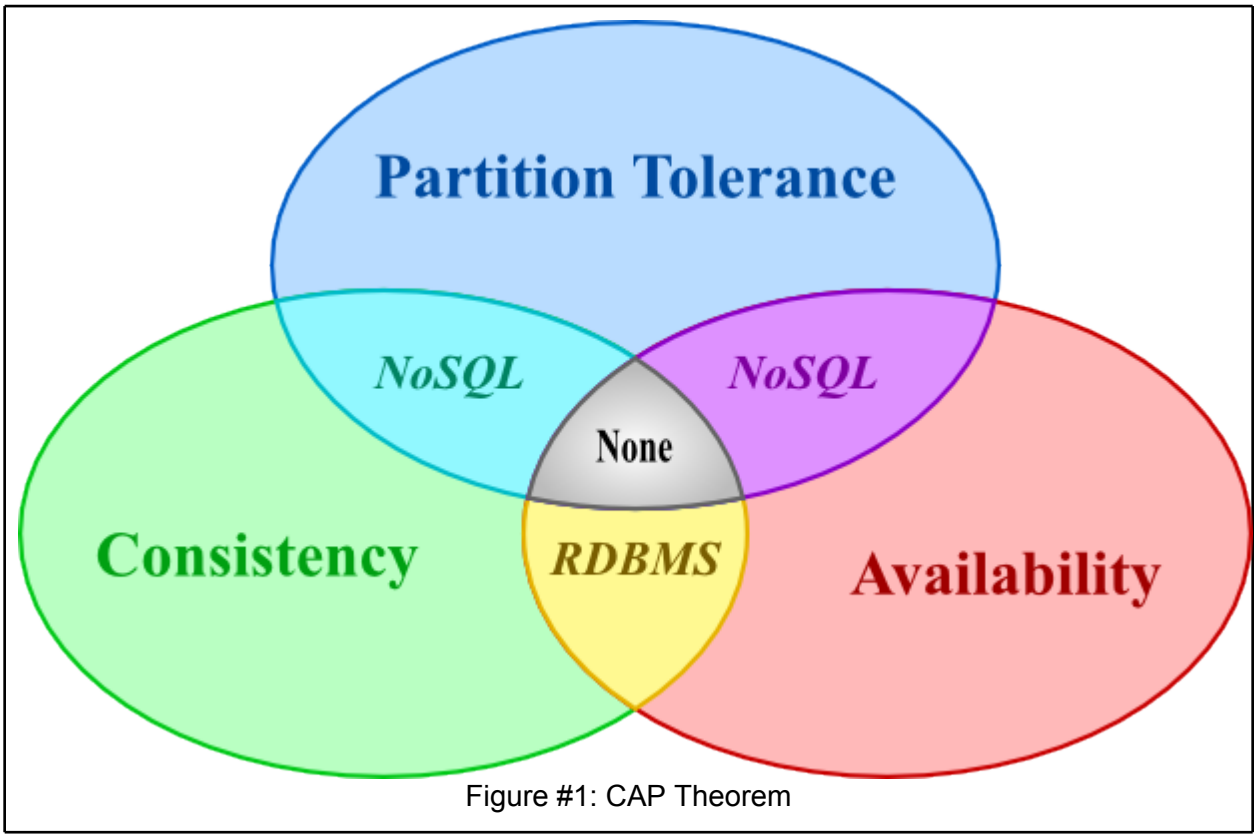


Figure #1: CAP Theorem

Traditional RDBMSs such as Oracle and MySQL were consistent and available, but they were not very scalable as they lacked partition tolerance. Google's BigTable gains Partition Tolerance by running atop the distributed Google File System, Amazon's Dynamo cloud-storage system gains partition-tolerance through the use of distributed hash table assignment of data to storage nodes. These systems developed internally at their respective companies, but open-source implementations of these systems were developed as Apache Cassandra, Apache HBase, Project Voldemort, Riak, HyperTable, and others.

The term NoSQL means *Not-Only SQL*, and includes many other types of storage systems. For example, CouchDB and MongoDB are classified as NoSQL because they are document-based storage systems. Many other types of non-traditional storage systems such as graph-based storage also fall under the NoSQL umbrella. The focus of our investigation was on distributed cloud-based data storage systems, so we did not make use of many such technologies.

## Motivation

Booz Allen Hamilton proposed that we investigate data storage and management platforms in the context of moving applications to a cloud computing model. We were asked to research the various cloud data store solutions, to compare them, and to build a simple application leveraging NoSQL data stores to document the differences and challenges in creating applications leveraging NoSQL data stores.

## Approach

Given these goals, we then built a demo application to study implementation issues, and ran benchmarks to understand the performance characteristics.

# Demo Application

In order to compare and evaluate the NoSQL solutions we created a web application as a demonstration. There were several goals we aimed to achieve with the demo application. Our primary goal was to create a real life example of the NoSQL databases to see them in action. The secondary goals stemmed from this primary goal. They included getting a sense of what it takes to implement these systems in real life, what their APIs were like to work with, the quality of the documentation and community support, and gauging the difficulty of employing there non-relational data models.

Most commercial uses of NoSQL systems involve serving data for a high-traffic, data intensive web application or service. Notable examples of companies using various NoSQL solutions as primary or secondary data stores in production applications are detailed in the table below.

| Company | NoSQL Solution |
| --- | --- |
| Facebook | Cassandra, HBase |
| Google | BigTable |
| Intuit | MongoDB |
| Twitter | Cassandra, HBase |

Given that the application platform most commonly associated with NoSQL data stores is the web, we decided to build our demo application using web technologies.

In order to accurately compare the implementation experiences we built a single web application with multiple database interface layers using a common query interface. This approach allowed us to write the client side HTML and Javascript code as well as the server side code that handled HTTP requests once. The figure below depicts the demo application's architecture.

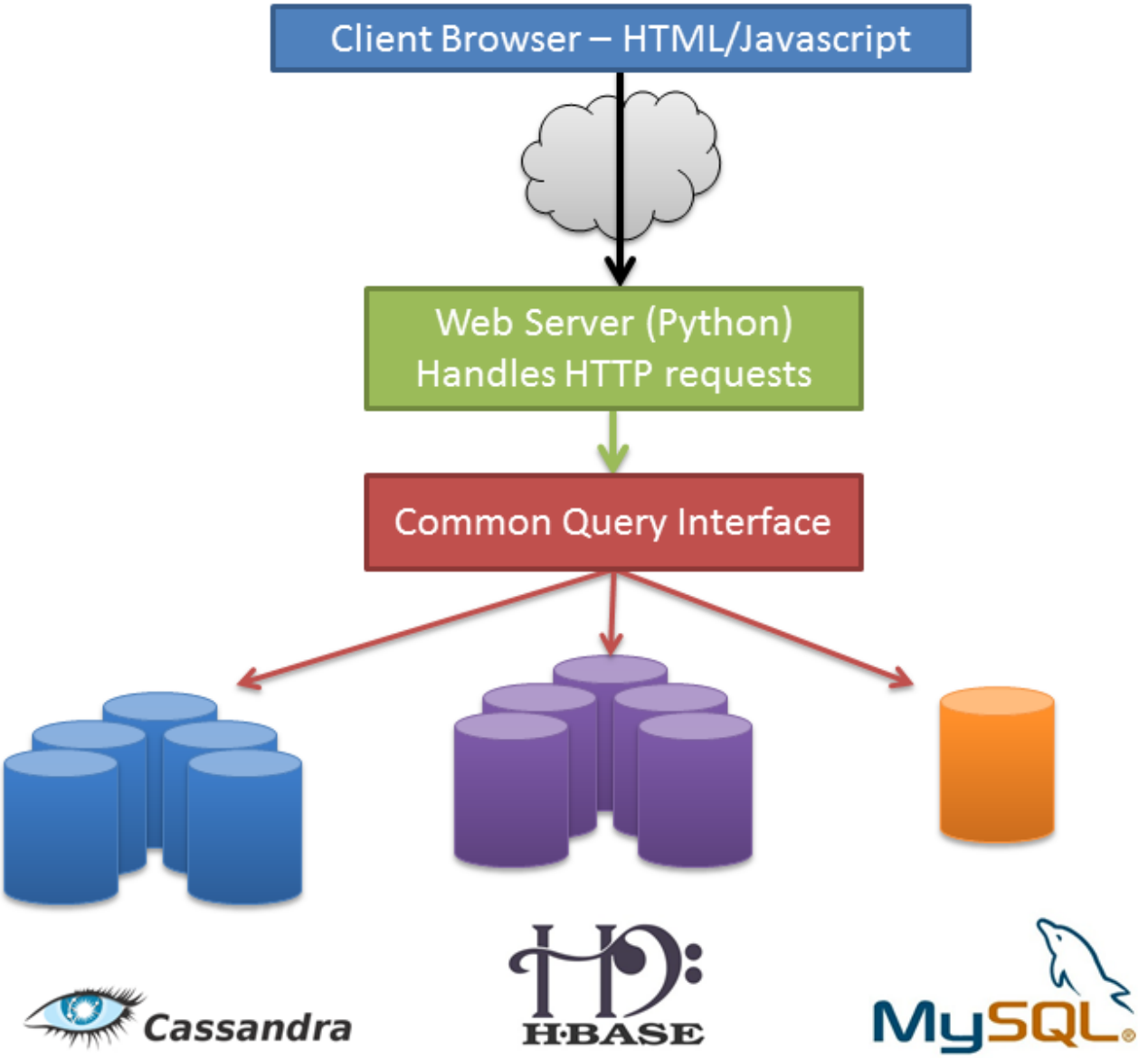


Figure #2: High level overview of the demo application architecture

## Virtual Machines

We chose to use Amazon EC2 as the virtual machine platform for our application. Some alternatives were considered, such as managing virtual machines ourselves with VMWare or acquiring actual hardware. These were rejected due to infeasibility. It simply was not practical to purchase commodity hardware to support the database clusters we planned on running. In the end EC2 was the best solution, because of its convenient web interface, relatively inexpensive rates and decent documentation.

The web server component of the demo application lived on an EC2 Micro instance, which was essentially a virtual machine with 613 MB of RAM and the ability to burst up to two EC2 compute units (virtual CPUs). This system was very cheap to operate 24/7, but could not handle large network or computation loads.

The NoSQL database clusters were setup on EC2 as well, though they were Large or XLarge instances which provided significantly more RAM and CPU power. The database sections later on in this report describe the setup and configuration process. Also, the appendix contains a detailed document with instructions on how to setup HBase and Cassandra clusters using EC2.

# Dataset

Since the demo application focused around the implementation of NoSQL databases, we needed a suitable dataset that would allow us to leverage the capabilities of the systems. Instances of these systems in production use often see datasets that measure into the terabytes or petabytes of data. For example, Twitter users generate over 12 terabytes of data per day, the majority of which is stored in Cassandra, HBase, and other NoSQL databases (Source).

As such we wanted a large dataset to use in our demo application; however, due to the great cost of storing terabytes of data on Amazon EC2/S3 we needed a physically small dataset. Another crucial factor in our choice of a dataset was the price of the data itself. Due to our small budget we only considered freely available open datasets.

We settled on the NASDAQ Exchange Daily 1970-2010 dataset available for free at Infochimps. The data consisted of CSV files containing the following fields:

```
stock symbol, date, open price, close price, daily high, daily low, and trade volume
```

The dataset, while modestly sized at around 800MB uncompressed, included over 8 million unique records. The large number of records proved suitable for storing in NoSQL clusters.

Additionally, unlike other potential datasets we investigated, the structure of the data was simple, which was important when had to create a data model for using Cassandra's and HBase's data model concepts. If the data had been more complicated, for example by exhibiting many relationships internally.

# Implementation

Our demo application consists of three screens. The first was the landing page, a static page with three buttons for each database. Clicking the buttons takes you to the second screen for the corresponding database. The second screen contains an HTML form that allows you to query the dataset according to stock symbol and date range. The third screen displays the open

and close price for every date in the selected range.

Below are screenshots of the demo application. Other than the web service call the Javascript makes to fetch the data, the client side implementation for each database page is identical. As such the form and results page is only depicted once, as there are no visual differences aside from the title text.

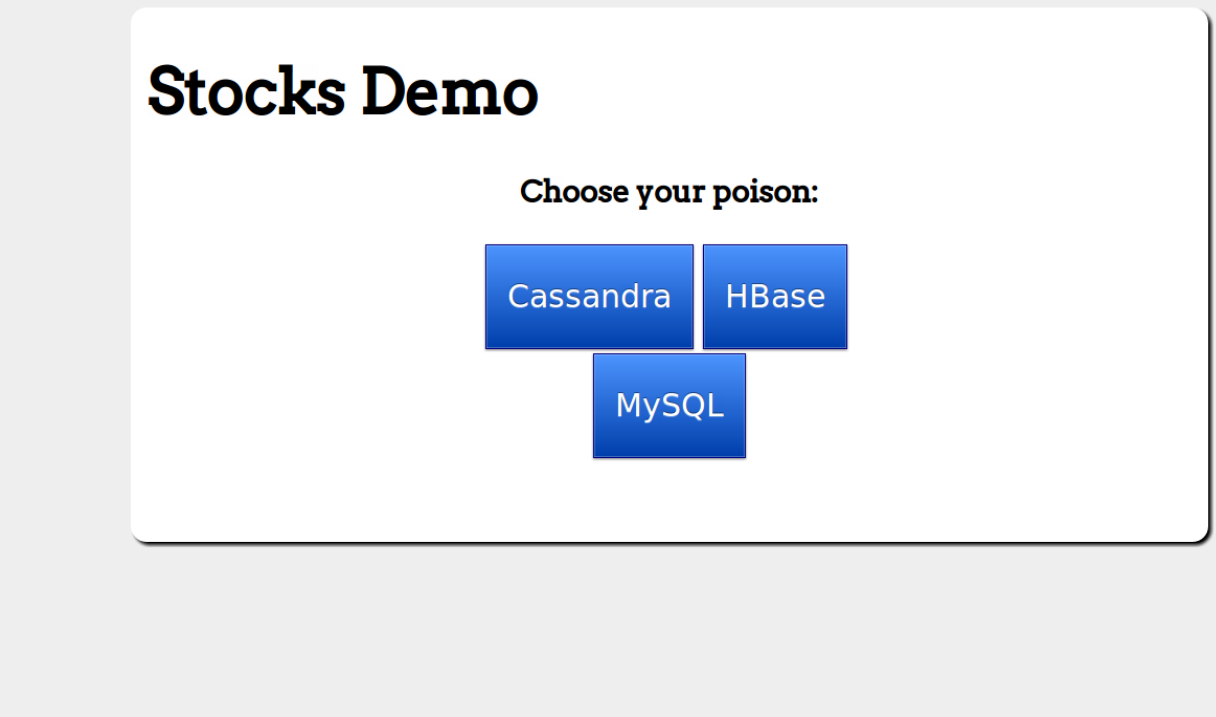


Figure #3: Screenshot of the landing screen.

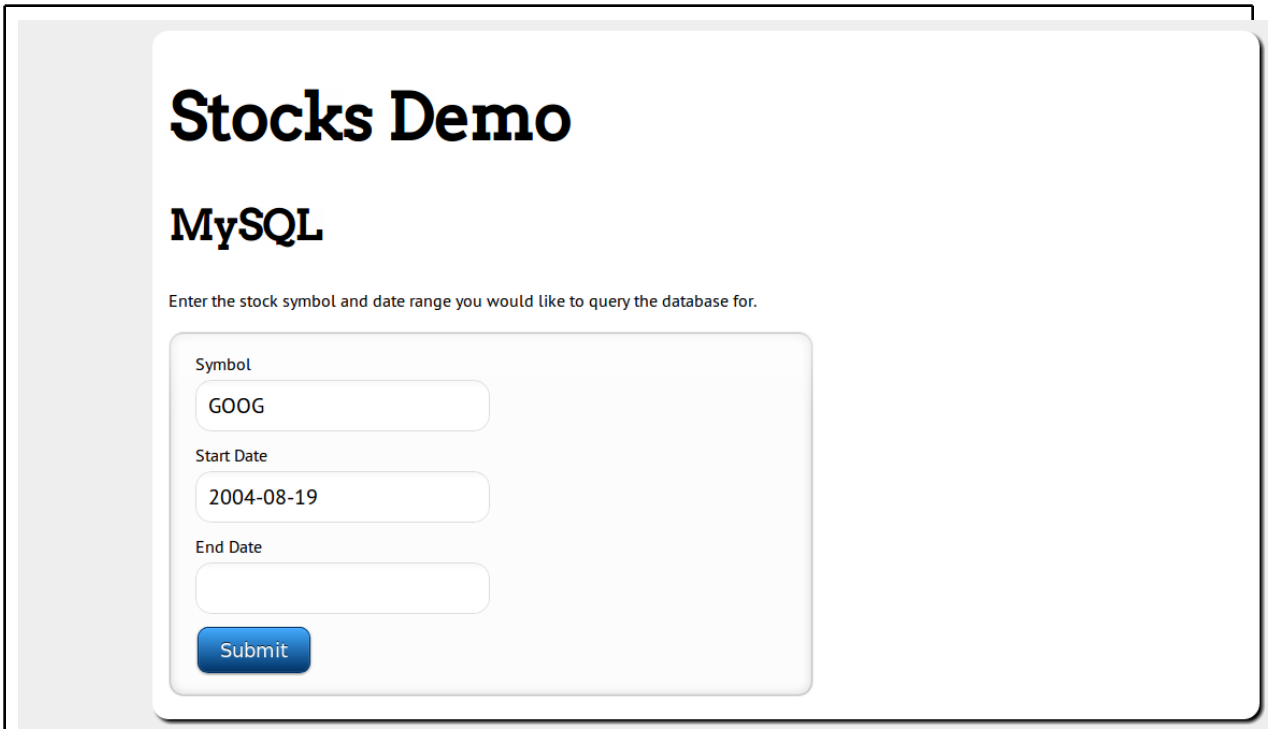


Figure #4: Screenshot of the query input screen.

# Query Types

Having chosen the NASDAQ dataset it was obvious that our demo application should allow us to query the data in different ways. When developing a data model for a NoSQL solution it is important to know in advance how you want to query your data. Unlike a traditional relational data model, where you can simple define relationships among the various elements in the data and then query based off of those relationships, in a NoSQL environment (particularly with Cassandra and HBase) the types of queries you will perform on the data will affect how you want to structure the data. Knowing this, we structured the demo application so that there were three distinct types of queries each database interface layer implemented. The three query types are discussed below.

## Partial Match Query

The partial match query was used in the “Symbol” form field to perform as-you-type lookup of stock symbols. The user in the screenshot depicted below has typed “GO” into the Symbol input field, and the AJAX client has made a request to the URI:
http://www.nosqldemo.com/mysql/symbol/search/?term=GOO
(Note: the ‘mysql’ in the URL above could be changed to ‘cassandra’ or ‘hbase’ in order to change the database the query will be run against)

The results of this query are a JSON list containing every stock symbol that starts with the string “GO”.
`Example: ["GOLD”, "GOLF", "GOOD", "GOODO", "GOODP", "GOOG", "GORX"]`

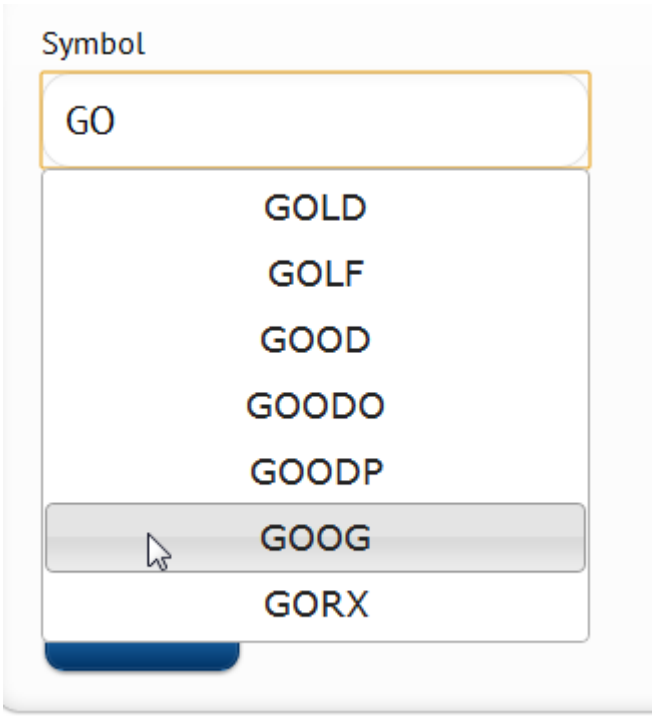


Figure #5: Screenshot of the partial match query

This query is significant because it tests the ability of the underlying database to query a non-numerical range of records. The Cassandra, HBase, and MySQL sections below will elaborate on how this query was implemented for each respective database.

## Date Range Query

After the user has selected a stock symbol he/she is interested in viewing, the date range for the selected symbol must be determined. This is necessary because not every symbol has records for the period our dataset spans (1970 - 2010). Google is a good example, as they were first publicly traded on NASDAQ on August 19 2004. Therefore, it does not make sense to allow the user to select a date outside the range in which the company is listed.

The date range query happens automatically once the user has selected a symbol. An AJAX request is made to the url:
http://www.nosqldemo.com/mysql/symbol/daterange/?term=GOOG
(Note: Again, the ‘mysql’ portion indicates which database is queried)

A JSON map is returned containing the minimum and maximum dates the selected symbol is

available for.
`Example: {"max": "2010-02-08", "min": "2004-08-19"}`

After the date range is received, Javascript is used to constrain the available dates in the date picker widget. As depicted below the available dates for the GOOG symbol range from August 19 2004 to 2010. Naturally, other symbols will have different date ranges.

This query tested the database's ability to traverse a record set and determine the minimum and maximum value.

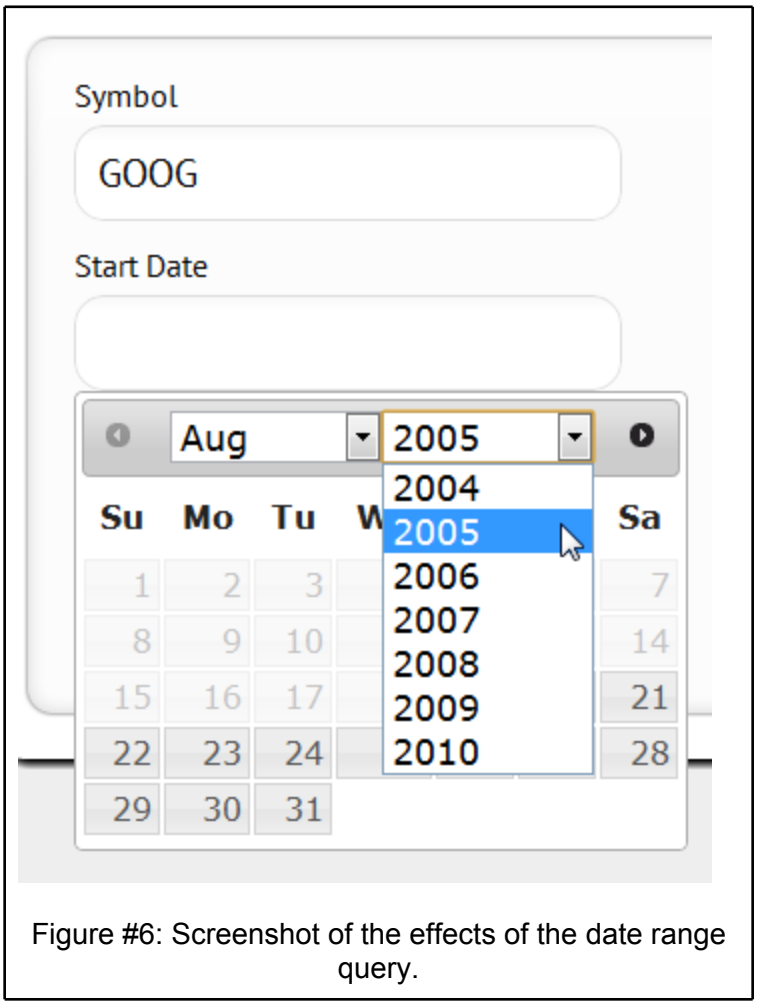


Figure #6: Screenshot of the effects of the date range query.

## Date and Symbol Query

Once the user has selected the stock symbol and a valid date range and clicks the submit button, a database query is made to retrieve the data for said symbol and range. The results are formatted into a table as well as graphed (using Google's free Graph API).

This query requires the database to locate a subset of records for a particular symbol.

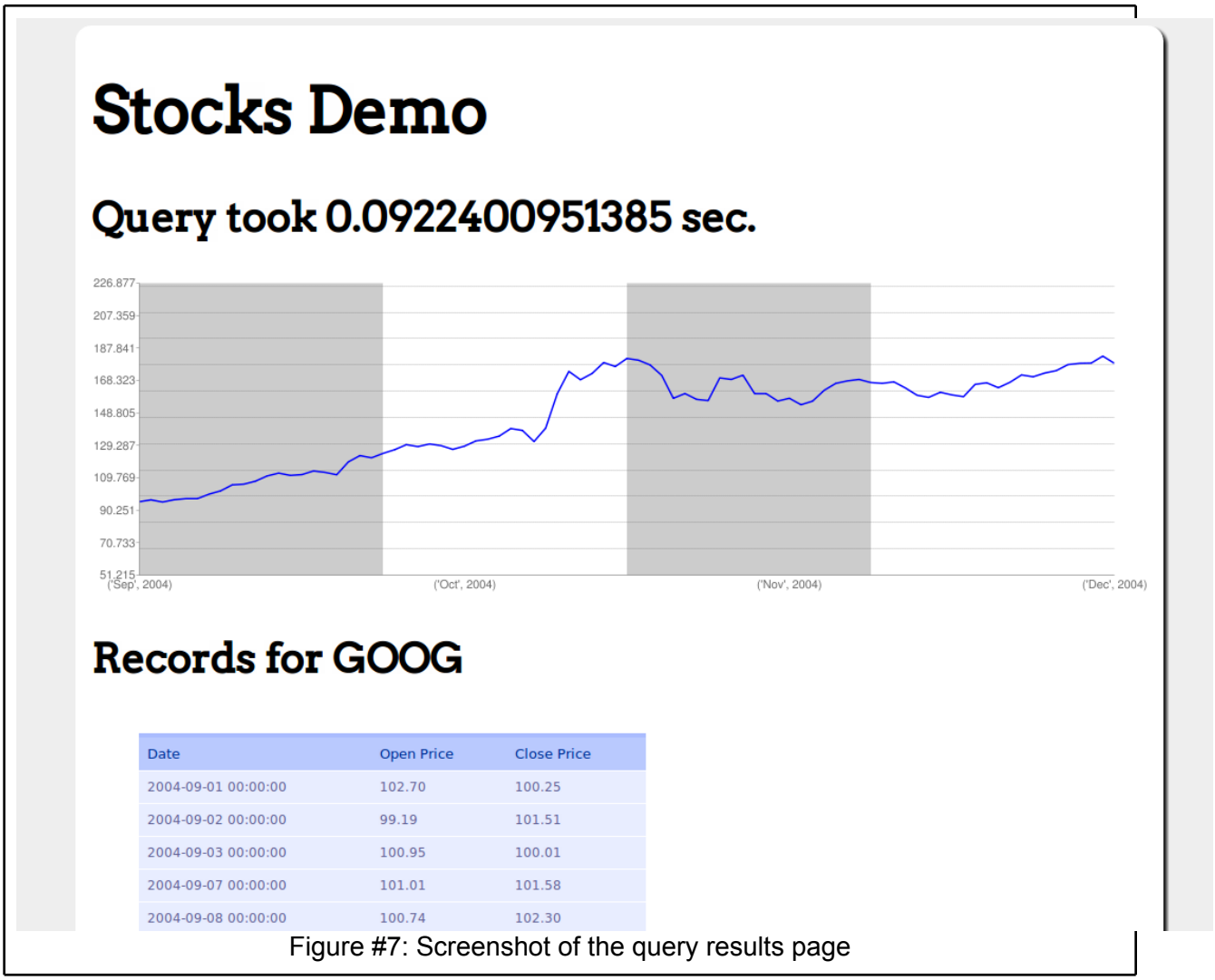

Figure #7: Screenshot of the query results page

# Cassandra

Cassandra is a distributed key-value store written in Java, originally developed by Facebook and released as an open source Apache project. Unlike many NoSQL solutions which take after either Google's BigTable or Amazon's Dynamo, Cassandra's architecture borrows ideas from both: while it partitions, replicates, and locates data in a similar way to Dynamo (Amazon's distributed key-value store), its column family-oriented, log-based data model is distinctly BigTable-esque.

# Tunable Consistency

Generally, Cassandra is located at the intersection of partition-tolerant and available in the CAP theorem, though one of its most touted features is its tunable consistency. Tunable consistency refers to the ability to configure the number of nodes that must agree on a given read or write for it to be considered successful. Possible settings for this are that a write only has to be successfully written to one node responsible for the chunk to be considered successful (it is then silently propagated to the rest in the background, which means there's a window of database inconsistency), by a quorum of nodes responsible for the chunk (e.g. 3 out of 5), or all nodes responsible for the chunk. Reads can be tuned in a similar manner, meaning either one node, a quorum of nodes, or all nodes must agree on a chunk value for it to be returned. This broad range of consistency values allows Cassandra to run the entire gamut of partition-tolerant and available (requiring agreement from one node) to partition-tolerant and consistent (requiring agreement across all nodes).

Obviously, performance will be greatly impacted by which of these consistency settings is used. In the case where consistency isn't tuned to available and partition-tolerant, writes will block until the requisite number of nodes have successfully completed their writes. Necessary consistency will vary based on specific application and consistency requirements. For our performance testing, we used the default consistency of ONE, meaning reads and writes only have to hit one node responsible for the chunk.

# Other Benefits

## Homogenous Clusters

Some of the other major benefits of Cassandra include the homogenous nature of the cluster and the lack of existence of a single point of failure. In a Cassandra cluster, all nodes are considered equal. Queries and writes can hit any node in the cluster, and the operation will automatically propagate to one of the responsible nodes in the cluster. If any node is taken out (or goes down), the chunks of data it was responsible for will silently be distributed across the rest of the nodes in the cluster. This feature also leads to the nice result that there's no single point of failure in a Cassandra cluster (though depending on the consistency settings, a node going down could result in the loss of unpropagated writes).

## Cross data center replication

Facebook's original use case for Cassandra involved locating, replicating, and receiving requests for data across multiple datacenters, and even across the continent. Making sure

this was as performant as possible was one of the main criterion of Cassandra's success, and it's been claimed to be one of Cassandra's most battle tested features. Of course, tunable consistency takes on a large role here: if writes or reads have to query nodes across data before completion, end users will see horrible request latency. Cassandra's big win in this area is silently and efficiently propagating deltas between data centers in the background, asynchronously with actual queries.

# Understanding the Data Model

As mentioned in the overview, Cassandra's data model takes after the row/column family/ column paradigm of BigTable. A brief overview of the major organizational units of the data model are as follow:

- **Keyspaces** are collections of column families. In traditional RDBMS speak, these are roughly equivalent in granularity to a database.
- **Column Families** are collections of rows. Depending on the partitioner used, these rows will be stored 'randomly' throughout the cluster (RandomPartitioner), or in order (OrderPreservingPartitioner).
- **Rows** are collections of columns. A row is retrieved from a column family by querying against a given row key. A given row can have an almost arbitrary number of columns, and these columns do NOT need to match with the columns of other rows in the containing column family (therefore, it's a mistake to think of column families like traditional tables or rows as traditional rows in an RDBMS).
- **Columns**, finally, refer to a key/value/timestamp tuple within a single row. Columns within a given row are stored contiguously on a node, and can be sorted for range queries or indexed for fast, random lookups.

Combining this all together, getting data out of Cassandra can be considered indexing into a multi-layered map (i.e. `map[keyspace][column-family][row-key][column-name]`). As briefly mentioned above, Cassandra allows the organization of rows within a column family and columns within a row to be separately specified in the data model.

The figure below depicts the setup of a single column family in our stock simulation keyspace:

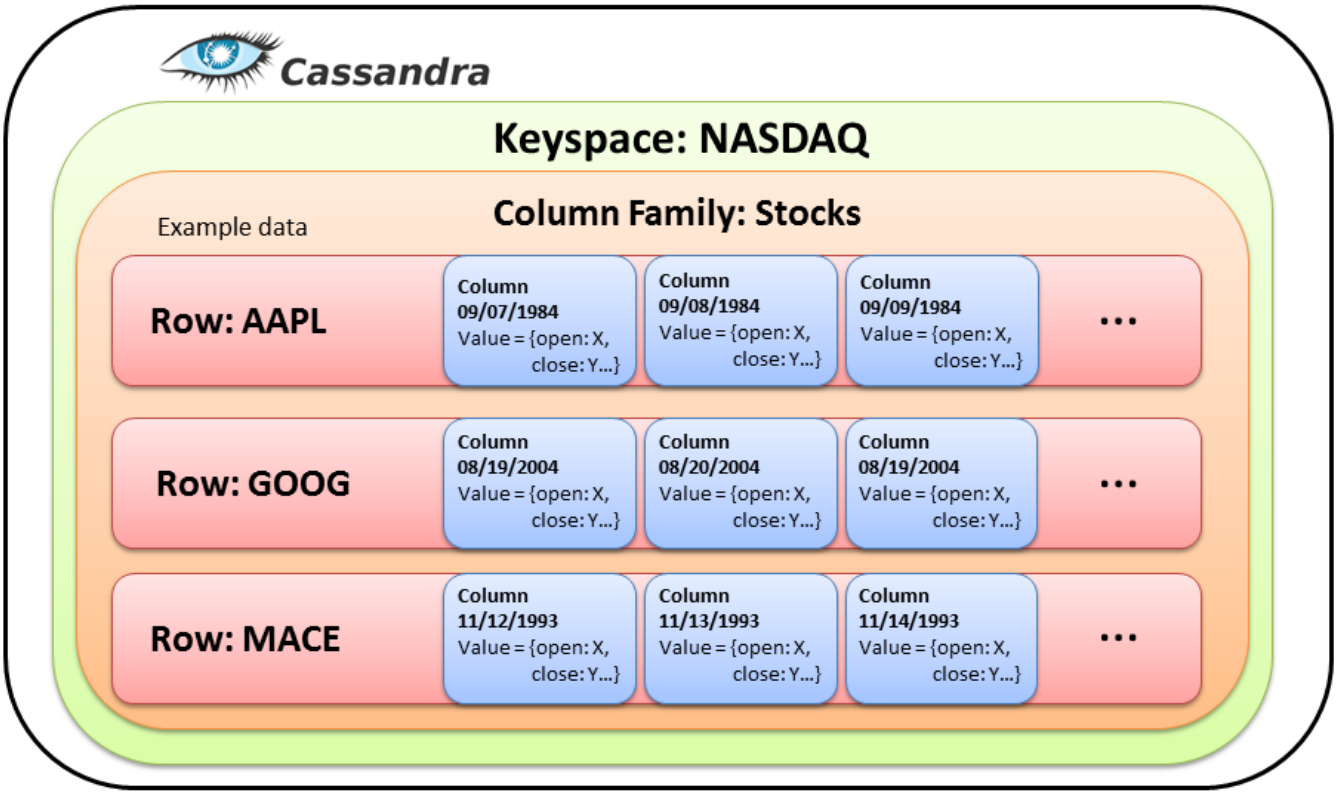


Figure #8: Depicting the Cassandra data model used in the demo application

Notice that the columns within individual rows are not necessarily the same.

## Super Columns

We have one more part of the general Cassandra data model to cover before we delve into the actual data model we used for the stock application. Super columns simply replace the normal 'scalar' value of a column with another map layer. So instead of `map[keyspace][column-family][row-key][column-name]` retrieving a single value, it retrieves yet another map to be indexed into. It is actually easier than it appears at first glance. We've established that a column is a name and a name+value pair:

```
column = ("name", "value")
```

A super columns is exactly the same, but instead of the "value" being a *string* it is another map! The keys in this map are column names, and the values in this map are columns.

```
super_column = ("name",  # this is the super column  name
                { column1 : ("name, "value"), # the first column
                  column2 : ("name, "value")  # the second column
```

```
                    } )
```

So to make a super column, we take a collection of columns (their names plus their name+value pairs) and put them into a map, then put that map in a standard column with a new name.

# Designing a Data Model for a NoSQL Database

Developing data models for NoSQL databases requires a significant shift in thinking from developing for an RDBMS. Storage is cheap, and there is a major shift from valuing data normalization within the database (i.e. minimizing redundancy) to figuring out how to make all queries as fast as possible. Because of this, a column family within a Cassandra cluster can really be thought of as a single view into a set of data. Other views, or column families, will likely duplicate the data stored but organize it in a different way to optimize for a different query. It is common practice to have a column family per type of query one expects to perform.

An obvious downside to this is if additional queries, and thus views/column families, are added after data is already in the database. These column families will have to be back-populated, which can take considerable time and incur load on network and cluster. Cassandra specifically is working on a partial solution to this via 'secondary indices'. A secondary index creates these additional views of data automatically by specifying other columns to index on in the schema definition. For the sake of time and relevance, we won't delve into it in this paper, but currently there **are** limitations to this method that the Cassandra wiki can further enlighten about.

# Stocks Data Model

As mentioned in the Query Types section, we had three queries we were optimizing for. Since two were very similar (existence vs. actual data retrieval), we only needed two column families.

## Partial Match Query

The column family for partial match queries stored only symbol names. We needed to optimize for figuring out all symbols that began with a given string:

- **Row keys**: Each row key is a letter of the alphabet, meaning there were 26 rows
- **Column names**: The column names for a row were just the stock symbols beginning with the the row key letter. Within each row, these column names were stored in alphabetical order to allow for alphabetical range queries.
- **Column values**: The column values were the company name of the stock. While we didn't end up using this data in our autocompleter, it could be a helpful addition.

**Example Use**

Query for ‘GO’: `symbols[‘G’][‘GO’:’GP’] => [(‘GOAL’, ‘Soccer Mutual’), …, (‘GOOG’, ‘Google’), …, (‘GOYA’, ‘Goya Inc.’)]`

### Date Range Query and Symbol/Date Query

The symbol/date column family stored the actual stock data by stock and date:

- **Row keys**: Each symbol has its own row, meaning each symbol is a row key.
- **Column names**: The column names for a row were the dates for which we had data for that stock. Obviously, these column names were not consistent across distinct rows.
- **Column values**: The column value for a given date was a super column containing a map for stock data to values (i.e. {‘price_open’: 1.23, ‘price_close’: 2.34, …}.

**Example Use**
To determine the date range for a given stock, retrieve all the column names for that stock. An example query for GOOG: `stocks[‘GOOG’].column_names`

To get the data for an individual date: `stocks[‘GOOG’][‘1/1/2006’][‘price_open’] => 1.23`

## Notes on Development

### Ease of Setup and Implementation

Cassandra is very easy to setup. The downloadable package contains pre-compiled Jars that and a capable default configuration. If you use the default configuration, setup simply consists of starting the Cassandra service (see the Appendix for details). Additionally, the Cassandra project hosts APT and YUM repositories on their site, which facilitates maintaining the latest stable version.

Setting up a Cassandra cluster is also relatively simple. Cassandra uses a peer-to-peer model that nodes use to locate and communicate with each other. Each Cassandra cluster must have a set of at least one nodes designated as “seeds”. Every potential Cassandra node must know the IP address of at least one seed in the cluster. Once a node has started up it contacts a seed and the cluster negotiates the place of the new node in the ring of nodes.

### API / Library

Cassandra is written in Java, but has bindings for many languages via its Thrift interface.Thrift is a software framework for scalable cross-language services development (Source).

Since we used Python for development of the demo application, we utilized a Python client library for Cassandra called PyCassa. PyCassa wraps the calls to Cassandra's Thrift interface and exposes a simple API.

Overall we were very pleased with the PyCassa. The API was well designed and seamlessly supported Python's builtin data-structures (lists and dicts).

The most significant problems we encountered when implementing the Cassandra interface for the demo application were understanding the data model and coming up with a workable schema (see above), and debugging throughput issues when inserting large numbers (8 million plus) of items. We were expecting to achieve about 10,000 inserts per second, but in practice were achieving barely 100 per second. As it turns out the throughput issues were caused by underpowered EC2 instances. Initially we were performing the insertion process on micro instances, which do not have a dedicated "compute unit", Amazon terms for CPU power. Migrating the insertion process to a medium instance solved the throughput issue.

# HBase

HBase is an open source, non-relational distributed database directly modeled after Google's BigTable. It is developed by the Apache Foundation, and runs on top of HDFS, the Hadoop Distributed Filesystem and an open source analogue to GFS, providing a fault-tolerant way of storing large quantities of data. In terms of the CAP Theorem, it fulfills partition tolerance and consistency while sacrificing availability.

HBase has two main selling points. In guaranteeing consistency, all clients can be assured that once they commit data, all other clients will see the updated data on subsequent reads (i.e. multiple clients can retrieve two different versions of a piece of data because changes haven't propagated across the cluster, as can be the case with Cassandra). Second, since HBase is built on top of Hadoop and HDFS, it has native Hadoop MapReduce support to perform data analysis on its stored data. Indeed, by anecdotal evidence, this seems to be one of the top reasons for deploying an HBase cluster.

Unfortunately, HBase also has two major downsides. First, the name node is a single point of failure of HDFS, and therefore HBase itself. If the master instance goes down, no data will be lost, but the cluster will be unusable until it is brought back online and it completes a replay process. As per [https://issues.apache.org/jira/browse/HDFS-976], Facebook has developed a highly available name node implementation that incorporates a "hot standby" (see http://hadoopblog.blogspot.com/2010/02/hadoop-namenode-high-availability.html), but the solution seems suboptimal and hasn't made its way into the HBase codebase.

Second, in maintaining consistency across all data nodes, writes to the database are much less performant than with a semi-consistent solution such as Cassandra. A write is not considered complete until it has been logged and all versions of the piece of data across the cluster are consistent. By default, HBase locates two copies of any given piece of data on the same rack for faster network transfer, while a third is located on a different rack. While we don't have direct experience with this, trying to store copies of data across multiple datacenters will likely incur a major performance hit (compare this to Cassandra, which touts multiple datacenter support as one of its most battle tested features: http://spyced.blogspot.com/2010/04/cassandra-fact-vs-fiction.html). See our YCSB results section for more information about HBase performance.

# Overview of Data Model

HBase stores data as a table, consisting of rows and columns. All columns in a table belong to a particular column family. Each individual cell, an intersection of row and column coordinates, contains the last 3 versions of that cell by default. Each cell's value is stored as an uninterpreted array of bytes. Inn addition, table row keys are also bytes arrays, allowing almost anything from binary representation of longs to even entire data structures to serve as the keys. All table access starts with a values row key, and then proceeds down the various column families and individual columns.

The schema we used for our application consisted of three tables labeled Stocks, Dates, and Symbols. The Stocks and Dates tables each contain the same data, with the rows sorted in symbol order and date order, respectively. This allows us to perform much quicker lookups depending on what kind of range query were are performing. The third table, Symbols, is used for partial symbol lookup. It consists of 26 rows, one for each letter, and has the values of all the stock symbols starting with that letter stored.

**HBase Data Model**

| | Row | Column Families | |
|---|---|---|---|
| | | Price: | Volume: |
| Price queries by symbol | <symbol><date> | Open, High, Low, Close | Volume |
| Price queries by date | <date><symbol> | Open, High, Low, Close | Volume |

| | Row | Column Families |
|---|---|---|
| Partial Symbol Lookup | | Symbol: |
| | <FirstLetter> | symbol_<symbol> … |

Figure #9: HBase Data Model used in the demo application

# Notes on Development

## API / Library

Pybase, the library we used, is a Python library for HBase based off of Pycassa, the Python library we used for the Cassandra version of our application. We chose it because of the similarity to Pycassa, and because of the lack of having to directly issue Thrift commands. While it suited our purposes well, it is a private project developed by a single person for their own personal use. The extreme lack of documentation proved to be somewhat of a hassle as we developed our project, as many of the difficulties we faced were due to this lack of documentation.

Once we had cut our teeth developing the Cassandra version, development of the HBase version was relatively simple. The library we used was very similar, allowing us to simply rewrite the functions from our Cassandra implementation. However, this was easier said than done. The insertion method changed completely, because HBase stores the data in an entirely different fashion. The other methods were changed to function using the scanner-based data extraction methods of Pybase.

One difficulty in particular we encountered was the use of custom Thrift data structures by Pybase. With our Cassandra model, the data was accessed using normal Python data structures. However, the scanner from Pybase returned Python data structures wrapped in “TCells”, a Thrift data structure. This led to some initial confusion when our methods appeared to be returning nonsense data even when we knew the insertion methods were working correctly.

Another problem we encountered was a problem with Pybase’s scanner argument syntax. Under our initial schema, a symbol that was a subset of another symbol would pick up the entries of both symbols from the database. While it was a relatively simple fix once we realized what the problem was, figuring it out took some time.

### Ease of setting up a cluster

We performed our initial testing of HBase on a single local node running locally. It was relatively simple to start up the server, access the shell, and create the tables manually using Thrift commands. We then accessed the server over local ports. This served well enough for performing testing on our application.

However, when it came time to setup an HBase cluster, the effort to set up a full cluster on EC2 was significantly more complex than that required for Cassandra clusters. The primary difficulty with administering an HBase cluster is the depth of the technology stack involved. HBase sits on top of HDFS (Hadoop Distributed File System) and requires Apache ZooKeeper for locking and name resolution. If data analysis via Hadoop or Apache Hive is also being performed, administrators are looking at maintaining and administrating 4-5 pieces of software (Java, Hadoop, HBase, ZooKeeper, and Hive). This is compared to only 2 for Cassandra (Java and Cassandra).

Additionally, server provisioning is not as simple as the more homogeneous Cassandra. Server roles include name node, secondary name node, and data node, in addition to the introduction of master/slave dichotomies. While servers can take on multiple roles, and much of the role provisioning seemed to be done automatically during our set up, larger deployments may have to take the heterogeneous nature of HBase clusters into account.

## Performance

As expected, HBase had significantly worse performance than Cassandra in our benchmarking tests. This is to be expected because HBase is consistent and partition-resistant, but sacrifices availability. This translates to lower throughput and higher response times. However, it is important to note that in a hypothetical situation with lots of concurrent reads and writes, you are not always guaranteed the most recent data with Cassandra, but you are with HBase.

# MySQL

MySQL has been, and continues to be the leading RDBMS on the market today. Popularized by the LAMP software bundle (it's the M), it has seen vast deployments in the industry and is available for many different operating systems, including Linux, Mac OS, and Windows. For these reasons, it's popularity and ubiquity, MySQL was chosen to be the control group in our performance testing evaluation of various NoSQL technologies.

## Overview of Data Model

The data set chosen for the experimentation is the NASDAQ market scores from 1970 to 2009. The date comes in a series of 26 CSV files, each corresponding to a letter of the alphabet, containing data on stock symbols starting with that letter. The fields specified in the header file are as follows:

**exchange,stock_symbol,date,stock_price_open,stock_price_high,stock_price_low,stock_price_close,stock_volume,stock_price_adj_close**

The exchange field is easy, it will always be "NASDAQ". Stock symbol refers to the unique company identifier for which the record is represented. All other fields represent the stock data as well as the date for the record. Logically, it follows that once the data set has been defined, we identify how it will be accessed in order to properly and efficiently build the data model for the database. There are three queries that are required to be implemented for the webapp to work:

1. Symbol Existence: Given a string representing a symbol, is it valid?
2. Symbol Search: Given a partial string for a symbol, return all symbols that match the partial string (if any)
3. Full record search, bounded by symbol and two dates: Given a complete symbol (existance guaranteed), and two dates (start and end), return all records from the database matching the criteria.

From this, it can be concluded that we will need at least a separate table used specifically for queries 1 and 2. The column layout and data types are as follows:

Symbols:

Id: Primary Key, Integer, Auto Increment
Name: symbol_name, char(10)

This aspect of the data model allows for fast completion of the first two queries. The third query is a little bit trickier. First, matching on symbol_name has to be done, then specifying a range for the dates. Given the structure of the date field, it is perfectly acceptable to leave it's type as string. The date formatted like this: yyyy-mm-dd. So the schema for the table will be unmodified from it's CSV state:

Daily_Stocks_Table:
    exchange: char(50)
    stock_symbol: char(10)
    date: char(12)
    stock_price_open: char(10)
    stock_price_high: char(10)
    stock_price_low: char(10)
    stock_price_close: char(10)
    stock_volume: char(10)
    stock_price_adj_close: char(10)

So, the process so far: define the dataset, identify the types of queries that will be performed, and from this information, create an initial design for the database system. It looks good on paper, but changes will have to be made down the line, especially in the efficiency department.

# Notes on Development

## Ease of setting up a cluster

Unfortunately, due to the CAP theorem discussed earlier, as MySQL follows both Consistency and Availability, but sacrifices Partition Tolerance, it isn't very scalable. So for the experimentation and demo phases, we used one instance of MySQL running on a custom crafted Ubuntu AMI on Amazon's EC2. There are some solutions that can be implemented to allow MySQL to scale to some extent, the most notable being MySQL Cluster, and lesser ones being sharding, memcached overlaying, and mirroring. Despite this, the following section will describe how the testing instance of MySQL server was set up and configured for this project. For reference, these instructions apply to Ubuntu 10.10 Server install.

Starting with a vanilla Ubuntu 10.10 AMI from Amazon EC2, the following command was run:

**sudo apt-get install mysql-server**

It will prompt for a root password, it is highly suggested that a strong password is picked. Installing the base for the MySQL server is that easy. Using the operating system's package manager is the recommended way of going about it.

Next, one must change the bind address that MySQL listens on. The default is the loopback interface, but should be changed to local network interface for remote access:

**sudo nano /etc/mysql/my.cnf**

The line that needs to be changed starts with **bind-address**, it should be the IP of the default LAN port.

Reboot the MySQL instance now:

**sudo restart mysql**

Next, user accounts and permissions have to be set up for proper access, this link does a good job of describing the steps needed:
http://dev.mysql.com/doc/refman/5.1/en/adding-users.html.

For this implementation, the user “stock_user” was added with permissions only to the “Stocks” database for security reasons.

From here, the basics of MySQL have been fully set up and database creation can begin.

## API / Library

Considering the project is being written in python, it was necessary to pick a popular library for accessing the MySQL server instance. The goal was to simulate a typical development environment and server setup as would be used in a live workspace. After some research, MySQLdb was chosen as the Library of choice for the demo application.

According to the wiki page for MySQLdb, it’s named MySQL for Python, to avoid any confusion, the module name is still MySQLdb as far as easy_install is concerned. Here is a quick description: “MySQLdb is an thread-compatible interface to the popular MySQL database server that provides the Python database API.”. So MySQLdb is a module that allows access to MySQL instances via a **low-level** API. That is, there is little wrapping around underlying functions than necessary to conform to an object oriented approach. The goal of this was to make sure there was no additional overhead that could be caused by higher level libraries. In addition, due to the relative simplicity of the demo application and the benchmarking application, MySQLdb’s low level approach worked perfectly.

# Performance

What good is a SQL server if it doesn’t respond fast? This section outlines the various tweaks and design decisions that have been implemented  to speed up the MySQL instance. The alterations slight, the benefits great, here is a step by step process taken to optimize the queries:

1. Looking at the most complicated query first (Symbol and Date Range), 5-6 character string comparisons are being performed when searching for a valid symbol, instead a foreign key relationship can be defined within the Symbols table.
2. The date column can be converted into the MySQL version of date, which is an integral type and therefore would be faster
3. Other columns can be turned into a more accurate representation of their values, instead of just strings, to help the search cost and reduce table size

But...

As it turns out, none of the steps above were necessary, MySQL comes with a very handy statement one can execute, queue CREATE INDEX. It creates a system-wide backed index on the most frequently accessed columns, with the options you specify, allowing one to maintain the wanted types, and keep from complicating the overall schema. The CREATE INDEX command was run on the date and symbol_name columns of the stocks table and yielded performance that was just as good as with having taken the steps above. For a point of reference, a simple query:

```
SELECT * FROM Stocks.Daily WHERE symbol='GOOG';
```

Before optimization took place, it would take ~46 seconds to execute, yielding ~3500 records from a list of ~8,500,000. After the optimization steps were taken, execution time dropped to ~0.02 seconds. A vast improvement.

# Benchmarking

To evaluate the applicative strengths and weaknesses of HBase and Cassandra, the two NoSQL databases we used in our application, we used the Yahoo Cloud-Serving Benchmark (YCSB). This tool runs a scripted proportion of operations using 1 KB records in a key-value storage system, and we used the six workload scripts bundled with the YCSB repository. We ran our benchmarks on a six-node cluster of extra-large Amazon EC2 instances, and set the number of operations in each workload to 20 million. Amazon Web Services uses heterogeneous processor architectures on its back-end, which can be verified by looking at the /proc/cpuinfo of EC2 instances. Therefore, to keep the tests comparable, we ran the benchmarks sequentially upon the same EC2 instances.

Running the benchmarks against HBase version 0.90.2 and Cassandra 0.7.4, we recorded the results for workload throughput and the latency of each operation within each workload.

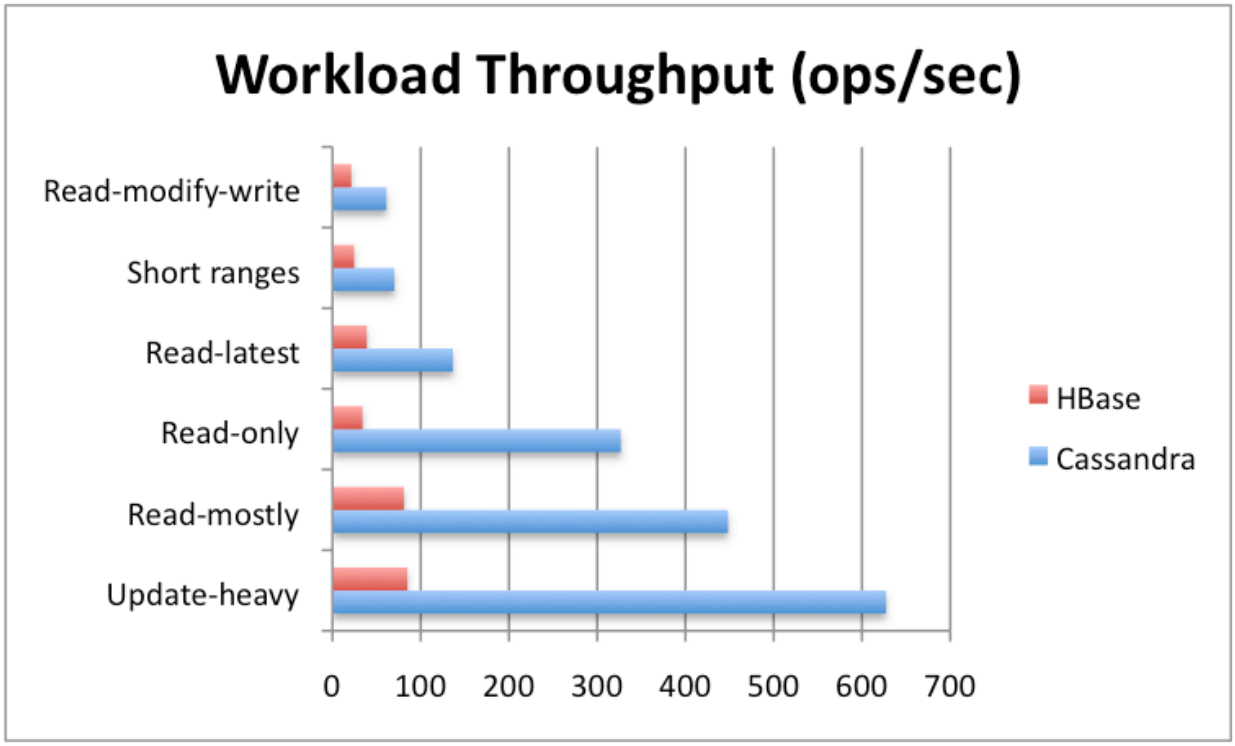


Figure #10: This figure shows throughput for each workload.

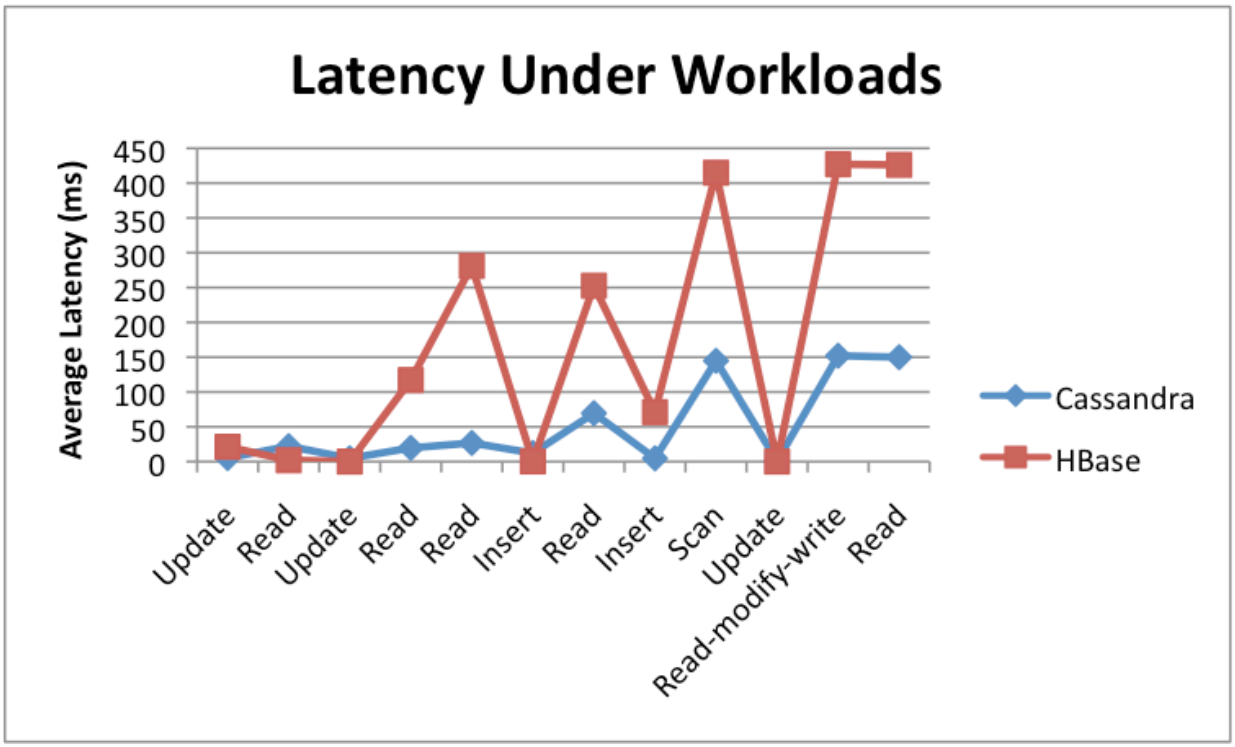


Figure #11: This figure shows latency, broken down by the individual operations from each workload.

While the initial goal was to see if any of these scenarios showed a workload performance

strength for HBase or Cassandra, it became apparent that Cassandra had better throughput and latency. This is because it sacrifices Consistency from the CAP theorem for Availability and Performance, while HBase sacrifices Availability (its data is eventually-consistent) for Consistency and Performance.

We did not benchmark MySQL using these tests because the overall goal of this project was to compare NoSQL cloud-storage solutions, and because the key-value operations in this benchmark can not always directly map to traditional SQL commands, such as the Scan operation used in the Read-Modify-Write workload. In future work, we would like to run these benchmarks against a wider range of NoSQL data stores.

# Conclusion

While we were unable to comprehensively test, benchmark, and evaluate many NoSQL solutions, we did learn several valuable facts when it comes to setting up and implementing them in a practical application. In this section we will share the lessons learned, often with comparison to the SQL-way of doing things, as most readers will be familiar with these systems. That said, it should be noted that NoSQL is a radical shift from the relational database paradigm, and attempting to understand NoSQL concepts in terms of relational terms will only lead to confusion and headache.

The most important takeaway from this project is to know your data. That is, determine how you want to query it, what sorts of questions you want to ask of it, and what sort of results you expect. Unlike SQL databases, where the data model is determined by relationships among elements, data models for NoSQL systems--at least Cassandra and HBase, but probably all key-value type stores--are are determined by what you are going to access. In SQL systems you use indices and JOINs to perform complex queries, whereas in NoSQL systems you have to have your data in a format that supports the types of queries you plan to perform.

The second lesson learned is to understand your NoSQL product's data model. Most RDBM systems support the query language known as SQL and use the same modeling concepts across the board. Of course various the SQL implementations differ slightly in syntax from product to product, but generally speaking they are identical. This is not the case when it comes to NoSQL databases. They employ their own concepts to express their data models. It does not help the matter any that many of the terms they use to describe their concepts are shared between NoSQL and SQL systems alike, though to assume the terms mean the same things across the board is a huge mistake. Likewise, it does no good to attempt to understand one data model in terms of another. This definitely is a disadvantage for NoSQL systems, because, again unlike SQL products, being an expert in one does not carry over well to another. This state of affairs is probably due to the relative infancy of the NoSQL movement. Even though NoSQL systems like Berkley DB have been around for decades, large scale, distributed systems that attempt to provide more structure than a bare one-to-one key-value interface are still developing. Perhaps as they mature the lexicon and concepts will unify; however,

such unification will come at the cost of obscuring the relevant and underlying theoretical and implementation details.

None of these NoSQL solutions are drop in replacements for MySQL, nor are they drop-in replacements for each other. They have important strengths and weaknesses, meaning the individual application and circumstance will be the most important factor in deciding which solution to use. NoSQL is not replacing traditional RDBMS's in the same way that instant messaging and Facebook haven't replaced email. We have covered many of the places where these data management systems shine brightly (and not so brightly), and we hope making a decision on how to organize data in the future has been made more clear.

# Appendix

## Amazon EC2 Instructions

This appendix contains simple instructions for setting up Cassandra and HBase clusters on Amazon EC2.

### Getting started

1. Register for Amazon AWS if you have not yet, and make sure you've signed up for both EC2 and S3.
2. Download the Amazon EC2 API Tools
3. Follow the guide to set up your EC2 environment.

### Setting up EC2

- Create a new security group for Cassandra gateways and nodes
  - `ec2addgrp cassandra-control -d 'Cassandra controller group'`
  - `ec2addgrp cassandra-node -d 'Cassandra node group'`
- Open port 22 for both groups
  - `ec2auth cassandra-control --port-range 22 --protocol tcp`
  - `ec2auth cassandra-node --port-range 22 --protocol tcp`
- Open Gossip, Thrift, and JMX ports between the nodes.
  - `ec2auth cassandra-node --port-range 7000 --protocol tcp -o cassandra-node`
  - `ec2auth cassandra-node --port-range 9160 --protocol tcp -o cassandra-node`
  - `ec2auth cassandra-node --port-range 8080 --protocol tcp -o cassandra-node`
- Open Thrift and JMX ports between the nodes and controllers
  - `ec2auth cassandra-node --port-range 7000 --protocol tcp -o cassandra-control`
  - `ec2auth cassandra-node --port-range 8080 --protocol tcp -o cassandra-control`
- Confirm the security group settings
  - `ec2dgrp cassandra-node`
- Create a keypair.
  - `ec2addkey cassandra-keypair`
  - Save the lines from BEGIN RSA PRIVATE KEY to END RSA PRIVATE KEY somewhere like `~/.ec2/pk-cassandra.pem`
  - `export $CASSANDRA_KEY=~/.ec2/pk-cassandra.pem`

### Launching a node

- Start a node/seed instance.
  - `ec2run --region us-east-1 ami-cef405a7 -t m1.large -g cassandra-node -k cassandra-keypair`
  - Note the instance id. It will look like i-90127833
- Log into the instance
  - Get the instance public address
    - `ec2din instanceid`
  - `ssh -i $CASSANDRA_KEY ubuntu@nodeaddress.compute-1.amazonaws.com`

### Installing Web Server

- Upgrade ubuntu
  - `sudo apt-get update`
  - `sudo apt-get upgrade`
- `Install git`
  - `sudo apt-get install git`
- `Use ssh-keygen to generate a key and add it to your github profile`
- `Checkout the git repo (from the home dir) with:`
  - `git clone git@github.com:Ramblurr/CapstonePython.git`
- `Install dependencies for the project:`
  - `sudo apt-get install python-dev libmysqlclient-dev`
- `Change to the git repo dir and install the python dependencies`
  - `cd CapstonePython`
  - `sudo pip install -r requirements.txt`
- `Launch the webservice`
  - `cd ~/CapstonePython/src`
  - `screen sudo python index.py 80`
  - `Note: sudo is required to bind to port 80`

## Installing Cassandra

- Add the Cassandra APT repository lines
  - `deb http://www.apache.org/dist/cassandra/debian sid main`
    `deb-src http://www.apache.org/dist/cassandra/debian sid main`
- Install the Cassandra package
  - `gpg --keyserver pgp.mit.edu --recv-keys F758CE318D77295D`
  - `gpg --export --armor F758CE318D77295D | sudo apt-key add -`
  - `sudo apt-get update`
  - `sudo apt-get upgrade`
  - `sudo apt-get install cassandra`
- Edit the Cassandra configuration files appropriately.
- Restart Cassandra (it auto-starts immediately following installation)
  - `sudo service cassandra restart`
- Confirm the node status
  - `nodetool  -h localhost ring`

## Installing HBase

- Download the hbase-ec2 scripts.
- Create a new keypair called root:
    - ec2addkey root
    - Save the generated private key and chmod 600
- Edit bin/hbase-ec2-env.sh
    - Fill in the five key variables
    - Remove the '-q' flag from SSH_OPTS
    - Set JAVA_VERSION
    - Set HBASE_VERSION=0.89.20100726
- Edit bin/launch-hbase-zookeeper
    - remove $TOOL_OPTS from the two calls to 'ec2-describe-instances'
- Launch the cluster!
    - ./bin/hbase-ec2 launch-cluster <name> <slaves> <zoos>