Current weather observations are not sufficient spatially or temporally to resolve all weather phenomena, but a large increase in the data intake would cause new problems with storage. NoSQL databases are potential solutions that offer fast queries and the ability to scale to incredibly large sizes. This study examines some of the unique characteristics of weather data and if they can be leveraged in a NoSQL environment. Data models were designed and tested for two NoSQL databases, Cassandra and Hbase, to see if they can meet all of the necessary query functionality and a few basic speed tests were run to gauge baseline query speeds.

Headings:

Database Design

Information Storage & Retrieval Systems – Geography

Information Storage & Retrieval Systems – Science

SQL (Computer Program Language)
AN EXPLORATORY STUDY OF USING NOSQL DATABASES TO STORE WEATHER DATA

by

David P. Miller

A Master’s paper submitted to the faculty of the School of Information and Library Science of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in Information Science.

Chapel Hill, North Carolina
March 2012

Approved by

Arcot Rajasekar
1. Introduction

One of the problems that have faced weather prediction for the last 40 years is a lack of enough data. Official National Weather Service (NWS) reporting stations number only a few thousand in the United States and only collect data every hour, unless certain conditions trigger a more frequent data collection (Surface Weather Observation Stations). Official stations are mostly automated, but cannot detect all kinds of weather, such as tornadoes, shallow fog, and hail (Automated Surface Observing Stations). Occasionally human observers augment the observations to detect these missing conditions, but human observers are not always available for all stations. In addition to official stations, the NWS also gathers data from the 11,000 volunteers in the NWS Cooperative Observer Program, but certain areas of the country still lack sufficient coverage (What is the Coop Program?). Many weather phenomena occur on short time scales and in remote areas that are not fully covered by this station network, and as such NWS stations cannot always give an accurate picture of all of the processes going on. For example, most tornadoes are produced by supercell thunderstorms that are on the order of 10 miles across and can form and die in less than an hour. Furthermore, there are many different regions inside the storm, such as inflow, downdrafts, and the rotating updraft, that would require multiple readings to properly resolve. There is an entire level of weather phenomena that cannot be observed with current data readings. Properly resolving these phenomena could lead to a large increase in the accuracy of weather forecasts.
Recent research that looked at data taken from commercial aircraft revealed that the intensity of the jet stream can vary significantly and frequently (Cardinali 1870-1). Currently the European Centre for Medium-Range Weather Forecasts (ECMWF) takes aircraft data into their model to help resolve upper atmosphere wind and temperature patterns. The data is thinned so that readings are spaced at least 60 kilometers apart, but with this data resolution the detailed structure of the jet stream can be lost. This study looked at a finer granularity of readings and noticed that the velocity readings of aircraft flying though the jet stream showed frequent peaks, possibly due to embedded gravity waves. These features only showed up in high frequency data and the ECMWF have said that underestimation of analyzed jet streaks is a common problem in their model.

Current weather prediction is not the only area that could benefit from an influx of new data. Climate change is one of the most hotly debated topics in politics and weather circles. There is a wealth of historical weather data stored in log books of ships that goes back to the 1600’s and covers many areas that have very little coverage otherwise (Brohan 220). Historically there has been a scarcity of readings from over oceans, and the United Kingdom has hundreds of thousands of log books from the Royal Navy and the Honorable East India Company that cover areas from Greenland around the horn of Africa and into the Indian Ocean. With the addition of all this data we could form a much clearer picture of how our climate has changed from well before the Industrial Revolution up through current times.

A drastic increase in data intake would not put much strain on sensing equipment already in place, it would be a trivial matter to take readings more often from ground stations or aircraft; the primary concerns would be power usage, resolving who has access and ownership of the data, and the storage and processing of all this new data, the last of which is the focus of this study. Traditionally relational databases have been used for most data storage operations
for the past 30 years and excel at storing and retrieving highly structured data like weather data. The drawback of relational databases is that they do not scale well with large scale data, so another solution is needed for a problem of this size. In the last decade NoSQL databases have risen to popularity because they scale incredibly well and can handle the kinds of unstructured data that is commonly found on the web (Han). NoSQL databases serve as the backend for many of the internet’s largest operations, such as Google’s search interface and Amazon’s online store (Decandia, Chang). This research will look into the combination of weather data and NoSQL databases to see if NoSQL databases can provide all of the needed functionality and if the structured nature of weather data can be leveraged for performance gains. Two NoSQL databases will be examined, Cassandra and Hbase. Both will be loaded with the same data and then a series of tests will be run to determine the usability of the query functionality and the performance of each database.

The following sections of this thesis will give a brief overview of important database concepts in both relational and NoSQL database systems, analyze the unique properties of weather data, present a few brief case studies of NoSQL system implementations and compare and contrast them to a weather implementation, outline the experiment that was run, and discuss the results obtained.

2. Database Overview

On a conceptual level all databases are essentially the same, they are made up of tables that contain rows representing individual items that need to be tracked and columns that describe attributes of those items. Relational and NoSQL databases differ though on how these tables are structured. In a relational database a lot of forethought and work is put into dividing an information space up into neat little tables of information. Every attribute should fit nicely
into one table based on its functional relationships and as needed the tables are related to each other so that all of the many-to-one and many-to-many mappings are retained. The classic example of this is a database for a company that contains employees, departments, and projects (Figure 1). Each row in the employee table contains a variety of information for that employee such as name, address and salary which are highly related. Relations are then inserted to point each employee to the departments they work for and the projects they are working on. The exact nature of these mappings is dictated by the structure of the information space; if an employee can only be a part of one department then employee -> department will be a one-to-many relationship. Relational databases tend to have tables that are densely packed since one of the objectives in the design is to eliminate redundant information and missing values. Furthermore, if there are changes to the structure of the database at a later date it can create considerable problems since large parts of the design and query structures may have to be reworked.

![Figure 1](image1.png)

**Figure 1.** A representation of a relational database with arrows representing the relations. Tables are densely packed with few missing values or repeated information.

NoSQL databases do not split information into many small tables but instead keep all of the information in one large, sparsely populated table (Figure 2). Each row still contains information about a single employee, but instead of separating out the department and project

<table>
<thead>
<tr>
<th>Employee</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee ID</td>
<td>Name</td>
<td>Address</td>
<td>Salary</td>
</tr>
<tr>
<td>1</td>
<td>John Smith</td>
<td>11 State St</td>
<td>50,000</td>
</tr>
<tr>
<td>2</td>
<td>Lucy Light</td>
<td>22 High Way</td>
<td>75,000</td>
</tr>
<tr>
<td>3</td>
<td>James Adams</td>
<td>33 Lookout Pl</td>
<td>60,000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Department</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dept ID</td>
<td>Dept Name</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Shipping</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Human Resources</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Projects</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Proj ID</td>
<td>Proj Name</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Lambda</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Alpha</td>
<td></td>
</tr>
</tbody>
</table>
information it is attached in separate columns to each employee. Since there are no relations present then NoSQL databases can store the information based on columns instead of storing based on rows like a traditional relational database. Both Cassandra and Hbase allow columns to be grouped into families to further control what kinds of data are stored together. For example, the database could be storing both numerical readings from an observation, which are small and quick to retrieve, and radar images which can be very large and slow to retrieve. It would take a careful design to separate both kinds of data in a relational database to ensure that queries of the numerical data were not slowed down by the radar images. In a NoSQL system both radar images and numerical data would only need to be tagged with different column families to ensure that they are stored efficiently.

This approach does lead to a lot of redundant information being stored in the database. Furthermore, when new information needs to be inserted into the database structure it only requires the creation of a few new columns and not a large reworking. That is not to say that there is no work put into the design of the data model for a NoSQL system, the design of the tables and creation of row keys has to be tailored to how the information will be queried instead of any inherent structure to the data. This can frequently lead to the same data being stored in multiple tables just to facilitate faster queries.

<table>
<thead>
<tr>
<th>Row key</th>
<th>Employee: Name</th>
<th>Employee: Address</th>
<th>Employee: Salary</th>
<th>Department: Name</th>
<th>Project: Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>JS5032</td>
<td>John Smith</td>
<td>11 State St</td>
<td>50,000</td>
<td>Shipping</td>
<td></td>
</tr>
<tr>
<td>LL75QP</td>
<td>Lucy Light</td>
<td>22 High Way</td>
<td>75,000</td>
<td>Shipping</td>
<td>Lambda</td>
</tr>
<tr>
<td>JA60TY</td>
<td>James Adams</td>
<td>33 Lookout Pl</td>
<td>60,000</td>
<td></td>
<td>Alpha</td>
</tr>
</tbody>
</table>

**Figure 2.** A representation of the same information in a NoSQL database. Tables frequently have missing values and repeated information.
One main concept that is essential to both relational and NoSQL databases is the primary key. The primary key is the piece of information that allows us to uniquely identify each row in the table. There are a couple of important concepts that define all primary keys, uniqueness and permanence. The key needs to be unique so that a query will only find the relevant information and permanent so that if aspects of your data change over time it will not be split and stored in separate places. If we look back at the employee example, what would make a good primary key for an employee? The first natural reaction may be the employee’s name since it is what we use everyday to differentiate people from one another. But looking closer we realize that both concepts behind the primary key are violated when using a name. First, names are not unique. Even in very small companies it is possible for two employees to have the same name so something else would be needed to differentiate them. Second, names are not permanent. It is very common and even expected that when two people get married that one of them takes the other’s name. From a database perspective this would lead to one employee having multiple collections of records that are not connected. Another solution might be to use a person’s Social Security Number as a primary key since it should be unique. Unfortunately, not all Social Security Numbers are unique, they are changed in some cases, and not everyone in the United States has a Social Security Number, so selecting a good primary key takes a lot of forethought and effort.

Historically relational databases have gotten around this problem by creating their own unique identifiers for primary keys that mean nothing outside of the system; examples of this include employee IDs, student IDs and insurance policy numbers. NoSQL databases have a bit more of a challenge since the entire database is one big table and some NoSQL databases only allow searching by the row key. Designing row keys is important not only to separate and store the data but to allow it to be retrieved in the most efficient manner possible.
2.1 Hbase

Hbase is an open source NoSQL database that is based on Google’s BigTable. While Hbase can be run on a single machine, it is designed to have multiple machines each performing specific roles to facilitate faster query speeds (Architecture). There are three main roles, master server, lock server, and region server. A single master server, which may be replicated, oversees the entire operation, ensuring that everything runs smoothly and any machine failures are dealt with quickly. A single lock server, possibly replicated, maintains locks on the region servers so that only authorized access to the data is allowed. Multiple region servers house the actual data, since all of the data is stored in one or more large tables it is split up into chunks and each regions server gets a piece. The data in the table is stored in lexicographic order by the primary key, typically called a row key (Row Key Design). Smart row key design becomes especially important so that similar data is stored together which should improve query performance. One thing needs to be noted, however. Since Hbase stores everything in bytes there can be some unexpected behavior with numerical row keys which will be used for weather data. Negative numbers appear first in lexicographic order, but for simplicity we will avoid this problem by using positive values for row keys. Second, decimal numbers will need to be padded with zeros to ensure that all row keys are a constant sized string.

2.2 Cassandra

Cassandra is another open source NoSQL database that combines aspects of Hbase and Amazon’s Dynamo. The data model is very similar to Hbase’s with a few extra tweaks. Like Hbase the table is stored by columns and columns can be tagged with families to store similar data together. Cassandra takes this one step further with the addition of super columns which
allow the storage of columns inside of columns. This leads to some interesting possibilities with weather data which will be expanded upon later. From a hardware perspective, Cassandra is very different from Hbase. Every machine in a Cassandra system is a built to stand on its own; there are no specific roles for different machines. This allows a Cassandra system to be spread over a much larger geographical distance since the communication time between different clusters of machines is not critical. Data is stored by the hash of the row key; each machine is assigned a range of hashes and any new data that falls in that range is stored on that machine (About Data Partitioning in Cassandra). Individual machines in a cluster are arranged in a ring so that their hash ranges line up sequentially. Cassandra employs a quorum that allows the administrator to determine how many sequential machines will store backups of data. Clients can make queries to any machine in a Cassandra cluster (About Client Requests in Cassandra). When the query is received it will be sent to all of the machines that have copies of the relevant data and when a sufficient number of responses, which can also be set by the administrator, have been received the response is sent back to the client. Since there is no master server in a Cassandra cluster all machines use a gossip protocol to determine the destination for queries and deal with machine failures (About Internode Communications (Gossip)). All machines operate on a heartbeat and every beat each machine sends a list of its hashes to different random machine in their ring. This way all machines have an index of the hashes of all other machines and can route queries effectively. When one machine fails it takes only takes a few beats for enough other machines to realize the failure and redistribute the data load to account for it.

3. Weather Data
As mentioned before, one of the main differences between relational databases and NoSQL databases is that NoSQL databases excel at storing unstructured data. Think for a moment of Amazon’s online store, which is supported by another NoSQL database, Dynamo. Each row in the database could pertain to a different product offered on Amazon.com and store information like the price, quantity and a picture. There would be millions of these products, and while they could be grouped into categories like books and shirts, we would not be able to infer any information about one product from looking at another product. Unstructured data like this can be thought of as a collection of unrelated items, and the main challenge is finding the correct item and returning the relevant information. Weather data is different. The collection of items in weather data, let’s say weather stations, are closely related. We would expect that two weather stations that are separated by a short distance to report similar readings. In the event that the readings differed greatly, there is significance in that as well. It could indicate the presence of a strong weather front moving through or that one of the stations is malfunctioning in some way. It is very difficult to make the same kinds of inferences from a collection of items on Amazon.com.

Furthermore, the way that we search through weather data is different. Ignoring current conditions which will be dealt with separately, it is rare for a person to only search through historical weather data for the conditions for one specific place; it is much more likely to search over a region to get a better picture of what processes were occurring. In addition to searching over a region, most searches will also incorporate some aspect of time to get a better idea of how the weather was changing. Looking deeper into weather data we can divide it into three different categories, time, location and parameters, with parameters acting as a catch-all category for the kinds of data that are searched for, be it numerical readings like temperature or pressure, radar images, or storm reports. The important factor is that weather data needs all
three aspects to be useful, the time of the occurrence, the location of the occurrence, and what actually occurred. A combination of time and location would also make a good candidate for a row key since it neatly divides weather data into a set of instances, but as with any row key design a closer examination is required to anticipate any unique quirks that could arise.

Currently location is encoded in a station identifier which is sent along with the observation. In the United States we typically use the International Civil Aviation Organization (ICAO) airport codes, such as KRDU or KIAD, but there are many different encoding standards worldwide so this can vary depending on the data set being viewed. Another drawback to using a station identifier is that, on its own, it tells us nothing about where the station is located. There would still need to be a separate index that tracked the exact coordinates of each station. In addition, some stations are expected to move, such as the weather instruments that are attached to commercial aircraft. Simply tracking this data by the flight number would lose an essential dimension to the data. Instead of adding an extra level of data that needs to be tracked it would be easier to just track all observations by the Geographic Information System (GIS) coordinates of the station. Each reading could still be tagged with the kind of station that recorded it in case the instrumentation has differing levels of accuracy. GIS also includes an altitude component in addition to latitude and longitude readings which would be very useful from a meteorological standpoint.

Unfortunately, using GIS coordinates for location does not solve all of our problems. While it is useful for satisfying the uniqueness component of a primary key it does not work for the permanence aspect. Ground based stations will move over a long enough period of time, either due to small, consistent movements from regular plate tectonic shifts or large, sudden movements from earthquakes (Deadliest Earthquakes). Meteorologically these movements are irrelevant; all we require is that each reading is accurate for the time and location that it was
taken. From a database perspective, these movements could cause the readings for one station to be stored in different parts of the table and a simple query for a station’s GIS coordinates would not return all of the data. In order to access all of the data for one station we would need to search by all of the coordinates that the station had ever occupied. Fortunately, one of the other basic requirements can be used to bypass this problem. As mentioned before range queries are going to be important for weather data, so in order to get all of the data for a specific station it would only be necessary to do a very small range query around a single point. Queries that encompassed a larger area would naturally pull all readings for the individual stations already. So even though the permanence aspect of the primary key is not satisfied location is still a valuable aspect of uniquely identifying individual observations.

Comparatively, time is a much more straightforward element. The only potential issues could arise with the format that time is stored in. A lot of databases store time formats in something similar to ‘number of seconds since’ an arbitrary date. For Linux systems this is January 1st, 1970. This would cause problems storing data from before 1970; if it was given a negative time then the lexicographic order would sort these times backwards. It is much easier, and makes more sense, to store dates in a simple string format similar to YYYYMMDDhhmmss (Year, Month, Day, hour, minute, second). Problems can arise with data that is taken on different time scales, such as daily or monthly averages. Rather than standardizing the hours and minutes to some arbitrary time, like midnight, these readings should be put in separate tables that only encode time down to the day or month.

The final issue with using location and time as a row key is deciding what order to put them into. If time – location is used then all readings worldwide for a specific time will be stored together, followed by all readings for subsequent times. This setup could cause problems for the database since all new data coming in would be put at the bottom of the existing table and
could end up being stored on the same server. This would overload one particular machine while others were underutilized. Conversely, by storing location – time all data would be first grouped by geographic location and then by time, so new data coming in would be scattered throughout the table and theoretically would be balanced throughout all of the machines. Storing data by location – time would mean that queries that pull a series of data from one station would be faster than queries that pulled data over a wide range of stations at one point in time. Queries that pulled data from a range of stations at multiple times would not be faster either way.

Throughout all of this discussion one specific query has been ignored, returning the current conditions. Since this query would be performed very frequently it makes sense to optimize for it by storing current conditions in their own table. This table would only use the location as a row key so that when new readings came in they would simply overwrite the old readings at that same location and only the most current data would be stored in this table. Time would still need to be tracked as a column to perform cleanup on stations that do move locations. In the event of a station moving, the last data point at the old location would never be overwritten, so having the time stored would allow us to delete these readings after a certain time frame.

4. Use Cases

As previously mentioned, NoSQL database implementations depend largely on the kinds of data being stored and how it will be queried. Before we get into the specific data models that are used, let’s examine a few real world implementations that chose between Hbase and Cassandra and see if any of the reasoning for that choice is applicable to a weather implementation.
Mozilla, the company that makes the Firefox web browser, has a project called Mozilla Labs Test Pilot that collects data from users to help designers create a better user experience (Deinspanjer). They are very open and up front about their data collection tactics and make sure that the user’s privacy is ensured. As the project moved out of the early stages they anticipated increasing their user numbers to tens of millions, collecting over one terabyte of data, and having the majority of their data collection occurring in two short windows of time. This means that their system would need to perform well under peak load and not simply sit idle during downtime. They looked at three NoSQL databases, Hbase, Cassandra, and Riak, and while they ultimately chose Riak mostly for a lower up front cost and less software development to get started, they did make some interesting comments about the utility of both Hbase and Cassandra. One of the main points against both was that a hand made front end would be necessary since both databases do very few security checks like checking payload size and inspecting data for missing or incomplete values. This would also be essential in a weather implementation. One of the major differences between Hbase and Cassandra was that on Hbase some types of administration and upgrades still required a restart of the entire cluster which creates a maintenance window when the service is not available. All of Cassandra’s maintenance only required rolling restarts. This would be especially useful in a weather implementation since there would never be a time when the system is not taking in data, so even a small amount of maintenance would require an alternate way to ensure data is not lost.

Netflix is another company that has moved a number of its services to the cloud and is reliant on NoSQL databases to support them (Izrailevsky). In a blog post last year Yury Izrailevsky, Director of Cloud and System Infrastructure, revealed that Netflix uses both Hbase and Cassandra for different operations and spoke about what each was suited to do. He was impressed with Hbase’s built in data compression support and ability to run real-time queries
while also running MapReduce jobs. As for Cassandra, Netflix was impressed with the ability to split its clusters across the United States to give equal service to different geographical regions. They also made use of the ability to configure the consistency and replication models to the job being performed. Being able to split clusters up geographically could be very useful for a weather implementation. If local readings were stored on a local cluster it would allow writes to process quickly but could slow down reads in some situations. If the majority of people requested a nationwide view of the weather then the system would have to routinely request information from multiple clusters.

In early 2010, Dominic Williams wrote a blog post about how a small company that he was working for was designing a massively multiplayer online game and had just finished the process of converting their back end from Hbase to Cassandra (Williams). He was obviously very positive on Cassandra, citing the flexibility of the quorum, the ease of setup, and low maintenance and tuning. But the most interesting point was how he differentiated the two databases. He says, “Hbase [is] more suitable for data warehousing, and large scale data processing and analysis” which would seem to describe a weather implementation. He goes on to describe Cassandra as “being more suitable for real time transaction processing and the serving of interactive data”. This also describes essential functions to a weather system with real time transaction processing for the new data coming in and the interface serving the interactive data. It is possible that a weather system would be best served by a combination of both systems, and this is a common problem in NoSQL databases. Most NoSQL systems were designed to solve a specific problem that current systems were just not good enough for.

It should be noted that the majority of these use cases were written about 1-2 years ago and since then the features of both databases have expanded, so not all of the conclusions
made may still apply. This is some of the most recent information available but NoSQL
databases are evolving quickly with new features being released a few times a year.

5. Data Models

5.1 Hbase

The data model for Hbase is fairly straightforward. Without all of the enhancements
that Cassandra offers there is not much flexibility to experiment with. Row keys will be a
combination of location – time and columns will contain the various parameters to be tracked.
In order to maximize query speed columns should be grouped into families based on data types,
where numerical readings are separated from larger data types like images. Finer grained
column groupings are acceptable, as long as they do not group dissimilar data types together.

One problem that could not be solved through manipulating the data model was being able to run a query that returned data from multiple stations over a period of time. While Hbase does allow range queries, it only allows them on the row key and only allows one range per query. The previously mentioned query would take multiple ranges on various fields, such as a range on latitude, longitude, and time. Since a typical row key would be stored as latitude, longitude, altitude – time, then stations that had the same latitude would be stored next to one another even if they were on opposite sides of the Earth. Even switching around the order of the parameters would not avoid this problem. What is needed is the ability to create geographic boxes where a small range is taken from a few different parameters.

One other option to rectify this problem was using MapReduce to run these queries. MapReduce is the ability for the Hbase system to split large jobs up into chunks and run them on multiple machines (Overview). The idea was to break the geographic box into a series of lines and then combine those results together. Unfortunately, MapReduce is primarily designed for
extremely large jobs that can take hours to run and use hundreds of machines; the setup
process alone can take over a minute, and is usually fairly resource intensive (How Many
Maps?). Using MapReduce for such a common query would overtax the system and taking over
a minute to return a simple query is simply too long.

5.2 Cassandra

While the data model for Cassandra can be tweaked to resemble Hbase’s exactly, there
were many more options to experiment with to see if a better solution could be devised.
Cassandra gives users a choice on how to partition the data that is read into the table. The
Order Preserving Partitioner (OPP) mimics Hbase where row keys are stored in order and allows
Cassandra to use range queries on the row key. The Random Partitioner (RP) stores row keys by
their MD5 hash which will automatically scatter them across different machines, and makes
range queries on the row key essentially useless. While the OPP seems to give more
functionality, it does run the risk of overloading certain machines and would make the load
balancing something that needs to be watched carefully by the database administrator.
However, due to the structured nature of weather data it may be easier to anticipate server
load since we can anticipate what future row keys will resemble. Most documentation and use
cases that were looked at strongly encourage the use of the RP and since other features made
range scans on row keys redundant, the Random Partitioner should be used.

The second feature that was looked at was the ability to create super columns. This
would give us three dimensions to store data in: the row, the parent column and the child
column. As mentioned earlier, weather data has three important categories, a time, place, and
a parameter. If each category could be mapped to a different dimension it may allow for faster
queries. There are some limits to the data that can be stored in each dimension of a super
column, however. Rows are essentially limitless; Cassandra is designed to expand to infinity in the row dimension. The parent column can hold up to two billion values, but the child column is not recommended to have more than a handful of values (JohnathanEllis). Assuming that time would be mapped into rows since both would keep expanding infinitely, there is not a good match for the remaining two categories. Both location and parameters would easily expand beyond a handful of values so the child column is not useful, and even the parent column could have trouble holding all of the values for location depending on how precise the readings for the GIS coordinates are.

One other concept that has not been explained previously is the timestamp. When a piece of data is stored, the time of insertion is also stored with it. The timestamp is mostly used to resolve write conflicts and for garbage collection, but it is accessible in the data model. One early experiment looked at using the timestamp to store the time dimension, but similar to the child column it is only designed to hold a small number of values and was too important in the underlying features of Cassandra to be manipulated. This idea was quickly discarded.

The final feature that was examined was secondary indexes. After a secondary index is created on a column, the data in that column can be searched through with standard greater than, less than, and equal to operations. This allows us to perform all of the range queries that are necessary to effectively use weather data. The individual latitude, longitude, altitude and time would need to be broken out into separate indexed columns, but this will prevent the need for doing a range scan on the row keys so the random partitioner can be used. This would also allow other columns to be indexed to create more robust query functionality, but this should be approached with caution. Every index created does require extra storage space and slows queries down so only necessary indexes should be created. While it is possible to index every column, what is gained by indexing the temperature column, for example? We could search for
every day at a station where the temperature got above 60 degrees, but how often is this query really going to be run? These questions will be difficult to answer without knowing who the users are and how the system will be used. If the system is simply a data repository that allows users to download large chunks of historical data, then a lot of the analysis can be done in separate programs. However, if the system is designed to do the analysis itself then these types of queries may be necessary.

The final data model is similar to Hbase’s where all rows are stored with row keys using location – time format. Parameters will be stored in columns, with similar data types being grouped together in column families and further refinements as necessary. In addition, latitude, longitude, altitude and the time will also be stored in indexed columns to facilitate querying. The system will use the random partitioner and no super columns. Additional tables will be created as necessary for data that is stored on different time frames, such as daily and monthly averages, and for the current conditions as explained earlier.

6. Experiment

There were two main goals to this research: to explore whether or not weather data could be effectively stored in a NoSQL database and to run some preliminary speed tests to determine if there were major differences between Hbase and Cassandra. The preceding sections of this paper have addressed the first goal and the remaining sections will outline the experiment that was run for the second goal.

6.1 Data

The data used for this experiment was part of the Global Summary of the Day (GSOD) from the National Climatic Data Center (NCDC). This data was acquired from
but is also available from the NCDC directly at http://www7.ncdc.noaa.gov/CDO/cdoselect.cmd?datasetabbv=GSOD&countryabbv=&georegio

The data consisted of a daily average of many of the fields that are reported in a standard NWS observation. For a more detailed look at the specific fields that were used and the data cleaning efforts that were undertaken, see Appendix A.

6.2 System

All experiments were performed on one machine that ran both instances of Hbase and Cassandra. The tests were done on the most current versions of Hbase and Cassandra at the start of the experiment; Hbase version 0.90.4 and Cassandra version 0.80. The machine had an Intel Pentium 4 3.6 GHZ processor, 3 GB of RAM, 450 GB hard drive, and running Ubuntu 10.04. The machine was accessed from an offsite location using a Secure Shell Client. To create tables and insert data a combination of command line interface and Python based client were used. For Hbase all tables and keyspaces were created with the Hbase shell and all data insertion was done through a Jython connection. For Cassandra, all tables and keyspaces were created through the Cassandra CLI and data insertion was done with Pycassa, a Python based connection.

6.3 Tests

There were three different tests performed to get an idea of the relative performance of each database: data insertion, accessing observations from one station over a period of time, and accessing observations from a geographic area over a period of time. For data insertion, a small program was written in Python to take the data from text files and insert it into the database. The first two months of the year 2000 were used which accounted for over 420,000
observations and over 10,000,000 data values. For the one station test, queries were run through the Hbase shell and Cassandra CLI to access data from 10 different stations over increasing periods of time. Each individual query was run twice and averaged. The geographical area test accessed data from a small area around the same stations as the single station test and over the same time periods. Each area was large enough to take in more readings than the single station test. The geographical area test was only run once per area and only on Cassandra since Hbase could not process these complex queries. All query speeds were measured with the Hbase shell and Cassandra CLI, the insertion speeds were measured in wall clock time.

6.4 Results and Discussion

The insertion test results were surprising; Cassandra handed all 420,000 observations in around 22 minutes while Hbase took nearly 14 hours to insert the January 2000 data and after a few tweaks February 2000 was inserted in around 11 hours. While Cassandra is optimized for write operations and it was expected that Cassandra would outperform Hbase, the results were extremely lopsided. I suspect there are two culprits for this result. First, Hbase required that each piece of information be inserted separately. For Cassandra one procedure call could be used to insert all 25 pieces of information for an individual observation, while Hbase required that each individual piece of information be added separately so it took 25 procedure calls to insert one observation. Secondly, there was no log displayed for Cassandra as write operations preceded, while for Hbase every addition created a log that was displayed to the screen. This could have been a feature of the Jython connection that was being used, but it is suspected that the second issue was a bigger contributor to Hbase’s poor performance.

A couple of tweaks were attempted to improve Hbase’s write performance. For the February data the write ahead log was turned off and the size of the write buffer was greatly
increased. This accounts for most of the performance increase between the two months, although February did have about 10,000 less observations than January. Additionally, for the Hbase data there are two distinct phases, preparation and insertion. The preparation of the data involves adding each individual parameter to an array, and then the insertion takes that array and writes it to the database. One alternative that was explored to improve performance was to create a multi-dimensional array of 10,000 observations and then measure the insertion time. This alternative was designed to isolate whether the preparation or the insertion was slowing down the write speed. The multi-dimensional array took around 45 minutes to create and under 15 seconds to insert. Clearly the preparation of the data was reason for the slow write speeds, but unfortunately I did not have the knowledge to correct it. Hbase does not seem to be designed to add multiple pieces of data at once, and I could not find any options to turn off the logging mechanism that was slowing down the preparation of the data.

![Graph: Test # 2: Data from One Station Over a Period of Time]

**Test Number**

In the one station query test Cassandra routinely outperformed Hbase by returning the results roughly twice as fast for every query. Both databases saw performance slow slightly as
more readings were accessed. Both databases performed within reasonable expectations and it is expected these speeds could be improved with in a larger implementation.

Test #3: Cassandra - Data from a Geographic Area Over a Period of Time

The geographic area test saw a significant decrease in performance in Cassandra regardless of how many results were returned. While it is notable that Cassandra could perform these queries, taking over three minutes to return a set of results would not be acceptable in any implementation. Originally the CLI was set to time out after 10 seconds on queries and that limit had to be raised twice before a query would complete.

The degradation in performance is mostly due to how Cassandra handles queries of this type. Each query must contain at least one equality (‘=’) operation in order to execute any greater than/less than (‘>’<’) operations. Cassandra then scans through the table creating a list of all records that match the equality operations and filters those results based on the greater than/less than operations. No part of the geographic area query could satisfy the equality requirement so a dummy variable was inserted into each observation that was set to ‘yes’.
Queries used the dummy variable for equality and attached six greater than/less than operations, one set for latitude, longitude, and time. The altitude dimension was ignored for simplicity and since a data set this small had no conflicts where the altitude was essential. The end result was that Cassandra scanned the entire table to create a list of all the records which was then filtered down based on the area and time constraints. This problem would only be compounded in a larger implementation. The data will need to be broken down into smaller chunks that are stored in separate tables, either geographically by country or temporally by month or day. Another option is to improve how Cassandra handles secondary indexes; they were recently added to Cassandra and will most likely be improved in forthcoming versions.

7. Limitations

There are a couple of major limitations to the speed tests performed in this experiment. First, I did not have access to the kinds of equipment to properly set up a full scale implementation of either Cassandra or Hbase. Both of these databases are designed to run on multiple machines that are each more powerful than the one machine that I did have. Furthermore, while both do offer a stand-alone version of software, some of the essential functionality is emulated or ignored in the stand-alone version so they may not give a complete representation of the capabilities of the system. Second, I came into this project having never used Cassandra or Hbase as well as having a limited programming knowledge base so my ability to tune each one and overcome some of the technical challenges was not sufficient. This was the most apparent on the write performance for Hbase. I am convinced that there is a solution to get Hbase’s speeds comparable to Cassandra’s but I could not find it through all of my searching. Because of this the speed tests should be taken as a general baseline at best.
8. Conclusion

Cassandra was the clear winner when examining both databases. Cassandra could provide all of the necessary query functionality where Hbase could not. Even though the speeds for the geographic area test were unacceptable, it is expected that future versions of Cassandra will improve the secondary index functionality. In addition, write speeds were clearly faster and single station reads were consistently faster. Furthermore, Cassandra is more flexible in the way the system is deployed by allowing clusters to be distributed across a wide geographical area, allowing the administrator to tweak the quorum to suite the data, and giving more options when designing the data model to deal with potential problems.
Appendix A: Data Fields and Cleaning

The data for this experiment consisted of 23 fields:

1-Station Number
2-WBAN number
3-Year
4-Month and Day
5-Temperature
6-Temperature Count
7-Dewpoint
8-Dewpoint Count
9-Sea Level Pressure
10-Sea Level Pressure Count
11-Station Pressure
12-Station Pressure Count
13-Visibility
14-Visibility Count
15-Wind Speed
16-Wind Speed Count
17-Maximum Wind Speed
18-Maximum Wind Gust
19-Maximum Temperature
20-Minimum Temperature
21-Total Precipitation
22-Snow Depth
Indicators for Fog, Rain, Snow, Hail, Thunder, Tornado

For fields 5 – 16 the first value of the pair is the average for the day and the second value is the number of readings that went into the average. Fields 19, 20, and 21 could also have additional flags attached to them to give more information about how the reading was taken, but for this experiment any extra characters were just included with the reading. WBAN numbers were not stored in the database, and fields 3 and 4 were concatenated to create one field that contained the year month and day. In addition, a dummy variable was created to allow Cassandra to perform the geographic area queries.

A separate list of the latitude, longitude, and altitude of each station was also provided and these coordinates were used in the row key for each reading. All longitude and altitude values were already padded with zeros, but latitude needed one extra zero added to account for the conversion of negative numbers. To deal with negative values the following steps were taken. For negative longitude values, the value was added to 360. This created a natural range of values from 0 to 360 that would logically progress from the Prime Meridian eastward around the globe. For negative latitude values, the value was added to 360. This puts northern latitudes at their natural 0 to 90 ranges but converts southern latitudes to a range between 270 and 360. There are no readings between 90 and 270. Both latitude and longitude are stored with their correct negative values in the individual columns. For negative altitudes, the absolute value of the reading was stored. This does leave a potential conflict open where two stations could have the same latitude and longitude and positive/negative altitudes, possibly due to a ground station and low flying aircraft. This data set did not have such a conflict but a real world implementation would need a more elegant solution even though negative altitudes should be relatively small (under 50 meters) and infrequent.

The final list of values read into the database is as follows:
Row key-LatitudeLongitudeAltitude-YearMonthDay

1-Station Number
2-Latitude
3-Longitude
4-Altitude
5-YearMonthDay
6-Temperature
7-Temperature Count
8-Dewpoint
9-Dewpoint Count
10-Sea Level Pressure
11-Sea Level Pressure Count
12-Station Pressure
13-Station Pressure Count
14-Visibility
15-Visibility Count
16-Wind Speed
17-Wind Speed Count
18-Maximum Wind Speed
19-Maximum Wind Gust
20-Maximum Temperature
21-Minimum Temperature
22-Total Precipitation
23-Snow Depth
24-Indicators for Fog, Rain, Snow, Hail, Thunder, Tornado

25-Dummy
Appendix B: Sample Queries

Test #2: Single station

Hbase:

scan 'weather', {STARTROW => '3528136250040-20000101', STOPROW => '3528136250040-20000103'}

scan 'weather', {STARTROW => '3621001330144-20000104', STOPROW => '3621001330144-20000109'}

Cassandra:

get WeatherData where StationNumber = 477610 and YearMoDa > 20000100 and YearMoDa < 20000103;

get WeatherData where StationNumber = 604250 and YearMoDa > 20000103 and YearMoDa < 20000109;

Test #3: Geographic Area

Cassandra:

get WeatherData where Dummy = 'Yes' and Lat > 3500 and Lat < 3600 and Long > 13600 and Long < 13750 and YearMoDa > 20000100 and YearMoDa < 20000103 limit 200;

get WeatherData where Dummy = 'Yes' and Lat > 3600 and Lat < 3800 and Long > 00000 and Long < 00400 and YearMoDa > 20000103 and YearMoDa < 20000109 limit 200;
References


“About Internode Communications (Gossip).” *Apache Cassandra 1.0 Documentation*. Apache Software Foundation. n.d. Online. 27 March 2012.


