Many of us are afraid to follow our passions, to pursue what we want most because it means taking risks and even facing failure. But to pursue your passion with all your heart and soul is success in itself. The greatest failure is to have never really tried.

– Robyn Allan

University of Alberta

DESIGN PATTERNS FOR HBASE CONFIGURATION

by

Dan Han

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

To the people,
for teaching me everything I need to know.

To the Internet,
for offering me everything I want to know.

To my future Mr.Right,
for allowing me to explore the world on my own.

Abstract

Cloud-based infrastructures enable applications to collect and analyze massive amounts of data. NoSQL databases endowed with high availability and excellent scalability through their easy deployment on cloud-computing platforms, become a more attractive data-storage solution for these big-data applications. Unfortunately, to date, there is little methodological support for software development on these platforms. In this work, we focus on applications that collect spatial data over time, since, due to the pervasiveness of mobile application clients, this class of applications is among the most popular applications today. To support the development and maintenance of these applications, this thesis develops a set of general guidelines for the design of HBase storage, taking advantage of the special 3D structure of HBase and a specific three-dimensional "schema" for geospatial applications. These guidelines and schemas have been evaluated with multiple data sets as well as through the migration of an existing geospatial application to the cloud.

Acknowledgements

Behind this dissertation, there are huge amounts of help and support from many people.

My supervisor Eleni Stroulia has been a role model in my entire life. Her passion for technologies, her insight and wisdom, her approach to mentorship, and her sense of humour are what I have been striving to learn and catch up. Paul Sorenson, my co-supervisor, picked me up from a mountain of rubble. I could not have come here without his recognition and encouragement. In addition, I am very thankful for the support of the management team at Cybera Inc, who gave me the generous access to their cloud environment for developing and testing. I also want to place my sense of gratitude to David Chen and Joe Topjian. This work could not have been done without their selfless and patient help.

I am so grateful for meeting so many friends from whom I can learn. It is you who lent me a hand to move forward when I was in trouble. It is you who gave me the confidence to proceed when I was hesitating. It is you who let me know the various cultures and the wonderful life. It is you who made my life here more enjoyable and memorable. With a thankful heart, I find it impossible to name everyone here. But my friends, you will forever be worthy of my sincere acknowledgement. All in all, very nice to meet you in my life!

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the explosive increase of location-aware devices (GPS-enabled smart phones and vehicles, RFIDs, tablets, etc) and the proliferation of sensor-based systems, location-based services that contextualize the user experience, such as advertisement and recommendation, are growing. Benefiting from these location-aware services are millions of users, who continuously register their location updates through their wireless providers. As a result, massive amounts of time-stamped geo-spatial data are being generated through these services, presenting a new challenge for database management systems. These new massive data sets demand a new degree of scalability, while, at the same time, maintaining good load balancing and high up-time. This task becomes even more challenging by the fact that the most typical use of these data sets is their near real-time analysis over time and across space.

Commercial relational database management systems (RDBMSs), such as Teradata [11], Greenplum [4] and Netezza [8] are reported to be able to handle multiple peta-byte data [9]. With their highly optimized performance, they can address the challenges mentioned above. However, although they perform well with respect to data size, they suffer from several limitations [10]. First, their fault-tolerance mechanism cannot deal with frequent failures in complex environments. Second, they are not “elastic”, i.e. they are not flexibly configurable to respond to changing load requirements. Third, the relational data model is not flexible to store unstructured or semi-structured data, which is often the type of data collected by the above mentioned systems. Finally, they are commercial systems, and, as such, they are not available to everyone. Unfortunately, open-source systems, such as MySQL and PostgreSQL, are lagging far behind in terms of scalability. To obtain excellent scalability in MySQL, mature development skills and extensive experience are required.

NoSQL (“Not Only SQL”) databases are non-relational distributed database systems, endowed with good availability, elasticity and scalability through their easy deployment on cloud-computing platforms. Given these properties, they are becoming a more attractive solution for these applications. NoSQL databases can be categorized into four types: key-value stores, column-family stores, document stores and graph databases[7]. HBase belongs to the column-family store type and is the open-source counterpart of Google's Big Table[2] on top of Hadoop[12]. The basic data storage unit in HBase is a cell, which is specified with the *row id*, the *column-family name*, the *column name* and the *version* [3]. Successful utilization of HBase has been reported by many enterprises. For example, Facebook presented how HBase infrastructure helps manage massive amount of data from hundreds of applications[1].

Although the usefulness of NoSQL storage systems has already been demonstrated in practice, a number of questions are still to be answered before they can be easily deployed in the context of software systems. One among them is *how to model the data* in order to optimize the performance of the queries issued. Even though some rough guidance about schema design has been provided by the specific NoSQL database offerings, such as HBase [3], there is still not such a method for guiding developers in systematically designing the structure of the "NoSQL Big Tables" for their particular application. Therefore, *a systematic method for NoSQL data-schema design for geospatial applications* is a timely and important problem in this area.

1.1 The Thesis Contributions

The objective of this work is to develop a set of guidelines for how to design an appropriate data schema for a given geospatial data set in HBase. More specifically, this thesis makes the following broad contributions.

- We have proposed a data model for time-series data in HBase, and evaluated it with several frequently used temporal queries. This data model explores and demonstrates the performance implication and improvement with the appropriate usage of *version* dimension in HBase. This is explained in detail in Chapter 2.

- We have proposed a data model for location data in HBase and evaluated it with several frequently used spatial queries. This data model is based on a hybrid index structure *HGrid*, combining a quad-tree and a regular grid as primary and secondary indices correspondingly. By comparing with two other data models based on quad-tree and regular-grid indices, this data model demonstrates efficient performance for range and k-nearest neighbor queries. Through this study, we also formulate a set of guidelines on how to organize data for geospatial applications in HBase which is discussed in Chapter 3.

- We have proposed a method for transforming data schemas in RDBMSs to HBase. In this method, we focused on the entity-relationship relational data model, and came up with guidelines for developers to follow during migrating the data in RDBMSs to HBase for an application. This method was applied in a practical case study which is described in Chapter 4.
- We have conducted a case study with a real application migrating to a hierarchical cloud, demonstrating (a) the use of existing tools to support the migration and (b) the application of the above guidelines in the design, and verifying the aforementioned data schema transition method. More detailed description can be found in Chapter 4.

1.2 Overview

This thesis consists of three main chapters. Chapter 2 compares a number of alternative data schemas for time-series data sets in HBase and develops a set of guidelines for the organization of these data sets. Chapter 3 presents a new model for geospatial data sets in HBase, and experimentally demonstrates its superiority to two alternatives. These two chapters are expanded versions of earlier publications [5] and [6] correspondingly. Chapter 4 covers a case study that utilizes the two data models proposed in the previous chapters to migrate traditional applications onto a hierarchical cloud, thus extending their capacity to deal efficiently with big geospatial data. This chapter has also been developed as a manuscript to be submitted for publication to a journal. Finally we present our conclusions and plans for future work in Chapter 5.

Bibliography

[1] D. Borthakur, J. Gray, J.S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, et al. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 international conference on Management of data, SIGMOD*, volume 11, pages 1071–1080, 2011.

[2] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[3] Apache Software Foundation. Apache HBase Reference Guide, April 2012.

[4] greenplum. Greenplum, August 2013.

[5] Dan Han and Eleni Stroulia. A three-dimensional data model in hbase for large time-series dataset analysis. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2012 IEEE 6th International Workshop on the*, pages 47–56. IEEE, 2012.

[6] Dan Han and Eleni Stroulia. Hgrid a data model for large geospatial data sets in hbase. In *Cloud 2013*. IEEE, 2013.

[7] R. Hecht and S. Jablonski. Nosql evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 336–341. IEEE, 2011.

[8] ibm. IBM Netezza Data Warehouse Appliances, August 2013.

[9] Sam Madden. From databases to big data. *Internet Computing, IEEE*, 16(3):4–6, 2012.

[10] Adrian Daniel Popescu, Debabrata Dash, Verena Kantere, and Anastasia Ailamaki. Adaptive query execution for data management in the cloud. In *Proceedings of the second international workshop on Cloud data management*, pages 17–24. ACM, 2010.

[11] teradata. Taradata, August 2013.

[12] Tom White. *Hadoop: the definitive guide*. O'Reilly, 2012.

Chapter 2

A Three-Dimensional Data Model for Time-Series Datasets [1]

2.1 Introduction

Cloud Computing, is attracting business owners for the perceived benefits, such as the elasticity of the fluctuating load, the access to large pools of data and computational resources, and the reduced operational costs compared to running in enterprise data centres. Given the advantages of the Cloud, some enterprises have been working on the cloud-based application development and deployment [4]. The majority of applications deployed in the cloud include some of the traditional and emerging cloud-based applications, such as social networking, online shopping, and real time instrumented data processing [8]. Low latency and high availability of service, and excellent system scalability are required for such applications, as the data generated in these applications are growing monotonously over time [8]. Therefore, large-scale ad-hoc analytical processing of the time-series data collected from those cloud-based applications is becoming increasingly valuable to improving the quality and efficiency of existing services, and discovering the knowledge.

Moreover, the success of this movement necessitates a design of scalable database management system which can effectively and efficiently organize and manage the massive amount of data[4]. Because of the open source relational DBMSs with the shortage of cloud features, and a commercial solution which requires expensive cost, RDBMSs are less attractive than the NoSQL database [4]. NoSQL databases, a non-relational distributed database system, usually avoids join operations, typically scales horizontally, does not expose a SQL interface and may be open source [5]. It can be categorized into four types:

[1]A version of this chapter has been published. Dan Han and Eleni Stroulia, 2012 IEEE 6th International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), pages 47-56.

Key-value stores, Column Family stores, Document stores and Graphic databases[10]. In comparison to relational databases, NoSQL databases

- enable the storage of big data, in the order of row key;
- scale horizontally across storage nodes relatively easily; and
- do not provide much data-organization and language support.

This last property is of particular interest to us in the work described in this chapter. Data in Column Family stores, for example, is stored in an "unstructured" manner, based on a primary key and attributes organized in column families. There is no notion of "normalization" and redundancy is allowed for the sake of convenience and efficiency. Given this new and different storage model, the community has not yet formulated any systematic methods for how to actually design the structure of the "BigTables". However, the data organization has a great impact on the performance of the queries implemented on these tables, and therefore, an appropriate data-schema design is a critical part in software developments. Moreover, as various data sets are generated from different applications in which data schemas cannot be shared, researchers and practitioners have to devote lots of time to do experiments with different data organization and management before they have confidence in deploying them into a product line. Therefore, a systematic method for NoSQL database data-schema design is a timely and important problem for researcher and practitioners.

HBase is a particular implementation of Column Family stores in the Hadoop project. The basic data storage unit in HBase is a cell, which is specified with the *row id*, *column-family name*, *column name* and the *version* [9]. This last element of the cell identifier implies that each HBase cell can have multiple versions of a particular data item. This is a particularly interesting property, with important implications for the task of managing time-series data. In relational databases, the values of a data element over time would, most likely, be stored in individual rows, with one of the columns dedicated to the element identifier. Adopting a similar structure in HBase, as would be likely if a developer followed their SQL schema-design knowledge and experience, would ignore this particular HBase property.

In the experiments described in this chapter, we explore a three-dimensional data model for data-organization in HBase, for managing large time-series datasets. This data model exploits the version dimension in HBase. Instead of creating independent rows for each data element in the time series, we associate the *version* element of the HBase cell identifier with each subsequent data-element value in the series. In cases where the time-series data advances indefinitely, we define a *period*, as the maximum number of versions to be "stacked" on the same cell. For example, given an hourly (daily, or monthly) period, all versions of the same data element within an hour will be stacked on the same cell, sharing the same identifier prefix but each one with its unique version identifier; a different row will be created for each distinct hour that values of this data-element are collected.

We have empirically evaluated the performance implications of this data organization with two time-series data sets: the Cosmology dataset [11], produced by a simulation, and the Bixi dataset [1], reporting the availability of shareable bicycles across Montreal. We found using this type of three-dimensional data schemas in HBase, as opposed to the SQL-inspired two-dimensional data schemas, better query-execution performance can be obtained.

This chapter makes two contributions. First, we proposed a three-dimensional data model which uses the HBase cell-identifier "version" as the third dimension along which to store time-series data. This model effectively increases the amount of data that is stored in a single row, and as a result, the data becomes distributed well across the HBase regions in the cluster. Second, through an empirical study, we investigated different ways of storing versioned data and their performance implications. The version dimension makes the data organization like a slice which is composed by rows and columns. This type of data organization is efficient in finding the similarity and dissimilarity between versions. The experiments results suggest that the depth of the version dimension has close relations with the types of queries and the software and hardware configurations.

The rest of the chapter is organized as follows. Section 2.2 reviews the background and related work in this area. Section 2.3 introduces the data models for time-series data, which is instantiated in the dataset domains in Section 2.4. Section 2.5 compares and evaluates the ad-hoc queries performance under different data schemas for the particular datasets. We discuss four extended issues and explicate how to apply the three-dimensional data model into a given application in Section 2.6. We conclude our contributions and future work in Section 2.7.

2.2 Background and Related Research

HBase uses the Hadoop File System (HDFS) as its underlying data storage platform. As we have mentioned in Section 2.1, the basic data storage unit in HBase is a cell, which is identified with the *row id*, *column-family name*, *column name* and the *version* [9]. Each cell can have multiple versions of data. At the physical level, each column family is stored contiguously on disk and the data is physically sorted by *row id*, *column name* and *version*.

It is important to note here that the version dimension is used by HBase for *time-to-live (TTL) calculations* [9]. Column families may be associated with a TTL length, and HBase will automatically delete rows once the expiration time is reached. This applies to all versions of a row - even the current one [9]. The maximum and minimum number of row versions can be configured per column family. Excess versions are removed during major compactions. It is not recommended to set the maximum number of versions to an extremely high level unless those old values are very important to you because this will greatly increase the size of the stored files. This recommendation is relevant when the *version* identifier is used to support concurrency control. However, it can also be used as another dimension along which to store data, in the case of large data sets, when there are seldom concurrent-operation conflicts. HBase distributes data according to *row-key* ranges; as a result, each HBase region server is responsible for handling the requests for a specific range of *row keys*. This storage principle implies that range queries are handled efficiently, because neighboring keys are very likely stored on the same server [6].

The HBase Coprocessor framework, inspired by Googles BigTable coprocessors [7], provides a library and run-time environment for executing user-level code within HBase region servers [13]. It decreases the communication overheads involved with the transfer of data from the region servers to the client, and enables dramatic performance improvement by pushing the computation up to the server, where it can operate on the data directly. As a data-centric programming model [12], it significantly improves the system performance by enabling parallel query processing. To reap the benefits of this framework, an appropriate partitioning of the data is necessary, which implies a well designed data schema. This is the reason why, in our work, we have focused in investigating the impact of different HBase table schemas on the performance of query execution using the Coprocessors framework.

The idea that organizing time-series data into "buckets" corresponding to periods has already been subject of some research. OpenTSDB [3] is a distributed scalable time-series database, written on top of HBase. OpenTSDB offers a data model designed to support data locality and, thus, obtain good query-execution performance. A similar data organization has been applied to Cassandra [2], where time-series data were stored as JSON objects, organized into hourly, daily and monthly buckets. This data organization could give the best query performance when each bucket contained no more than a few tens of data points.

The above studies examine the same problem as we do in this chapter; however the

Table 2.1: Two-dimensional and Three-dimensional Data Models

Data Model	Row	Column	Version
2D	unique Id-timestamp	varying properties	current time
3D	unique Id	varying properties	timestamps

data models they employed consist of two dimensions only. In our three-dimensional data model, the evolution of the data over time is organized and stored in the *version* dimension, instead of the column dimension which is used in OpenTSDB, and the special data model in Cassandra. Benefiting from the third dimension, our data model enables the storage of the data-element details in the table columns, instead of "wrapping" complex data into a JSON object.

2.3 Three-dimensional Data Model

In this chapter, we use the term "data model" to denote an abstraction of the HBase table design, and the term "data schema" as a specific case of the data model for a particular data set. Typically, a relational data schema is described as a two-dimensional table of rows and columns. In this setting, a value can be viewed as a data point in the two-dimensional space. We call this a two-dimensional data model. By analogy, in our three-dimensional data model, each value can be viewed as a point in a three-dimensional space, defined in terms of rows, columns and versions.

Table 2.1 describes the differences between two-dimensional data model and three-dimensional data model in HBase. The data point in two-dimensional data model can be expressed by the row and the column. The version dimension is present but is only used to indicate that the data is up-to-date. So the *sequence id* of a particular value in the time series has to be stored as a part of row key. The data point in the three-dimensional data model can be expressed by the row, column and version intuitively, with the version dimension representing the *sequence id*, which may be monotonically increasing timestamps (for continuously recorded real-time data, for example) or "snapshot identifiers" for ad-hoc sequence data. In this three-dimensional data model, the row key corresponds to a unique identifier for each data element and each column should be used to store some of the data propertie(s).

In general, there are a few basic guidelines for designing a data schema for storing a particular data set on HBase.

- The row key should be as short as possible, because it is stored in every cell in that row [9]; a longer row key will effectively result in much wasted space.

- In order to fully utilize the potential of coprocessors, one has to aim for organizing the data in a way that makes the processing of the most frequent queries "local". And as HBase sorts the row keys in lexicographic order, one should aim at constructing row keys by combining those data-element properties that are usually used to "select" elements of interest. Taking into account the need to keep row key short, one has to balance the trade-off between the row-key length and the number of attributes it combines.

- The various columns should be used to represent the data-element properties whose values change over time. The column name should be kept as short as possible, for the same reason as the row-key should be kept short. It is better to have few column families and columns. In our experiments, we have limited the number of families to one and, in general, no more than a few tens of columns are appropriate.

- Finally, we propose that the version dimension should be used to store the time dimension. It should be designed as a time bucket, but the length of the bucket cannot be too long. It is determined by the size of the unit of the data and the hardware resources where HBase runs on.

2.4 Case Study

2.4.1 The Datasets

The **Cosmology Dataset** [11] is produced by an N-Body simulation of the universe evolution. In the simulation, the universe is represented by a set of particles. There are three varieties of particles: dark matter, gas, and stars. All particles are points in a 3D space and their evolution is simulated over a series of discrete timestamps. Every few timestamps, the simulator generates a snapshot of the state of the simulated universe. Each snapshot records all properties of all particles at the time of the snapshot [11]. We used the "cosmo50" data set, which consists of 321,065,547 particles from 9 snapshots with a total size of around 14 GB in binary format.

The **Bixi Dataset** [1] is a public dataset collected by a bicycle-renting service in the city of Montreal. Users subscribing to the service, can borrow a bike from a station and return it to any other participating station, based on the availability of bikes and empty docks respectively. The data is collected every minute by the sensors equipped in 404 stations

Table 2.2: Examples of the three Data Schemas for the Cosmology Dataset

(a) Data Schema 1

sid-type-pid	**pp:px**	...	**pp:vx**	...	**pp:eps**	**pp:mass**
24-2-33554444	-0.434413	...	-0.349134	...	4.0E-5	5.29952E-10
...	...	...	...	...	...	...
84-2-33554500	-0.142892	...	0.0776743	...	4.0E-5	5.29952E

(b) Data Schema 2

type-pid	**pp:px:v**	...	**pp:vx:v**	...	**pp:eps:v**	**pp:mass:v**
2-33554444	[-0.434413,...]	[...]	[-0.349134,...]	[...]	[4.0E-5,...]	[5.29952E-10,...]
...	[...]	[...]	[...]	[...]	[...]	[...]
2-33554500	[-0.142892,...]	[...]	[0.0776743,...]	[...]	[4.0E-5,...]	[5.29952E,...]

(c) Data Schema 3

type-reversedpid	**pp:px:v**	...	**pp:vx:v**	...	**pp:eps:v**	**pp:mass:v**
2-44445533	[-0.434413,...]	[...]	[-0.349134,...]	[...]	[4.0E-5,...]	[5.2E-10,...]
...	[...]	[...]	[...]	[...]	[...]	[...]
2-00545533	[-0.142892,...]	[...]	[0.0776743,...]	[...]	[4.0E-5,...]	[5.2E,...]

Table 2.3: Three Alternative Data Schemas for the Cosmology Dataset

Data Model	**Row**	**Column**	**Version**
Schema1	sid-type-pid	particle properties	no meaning
Schema2	type-pid	particle properties	snapshot id
Schema3	type-reversedpid	particle properties	snapshot id

around the city and stored in the form of XML. In each XML file, there are station id, name, geographical coordinates, docks' status, and other station-related information. The dataset we used was collected for a period of 70 days, from September 24, 2010 to December 1, 2010. It is a 12 GB dataset that contains 96,842 data-points for all the Montreal stations.

2.4.2 Three Alternative Data Schemas for the Cosmology Dataset

We have experimented with one two-dimensional data schema and two three-dimensional data schemas for the cosmology dataset, depicted in Table 2.3.

The **Data Schema1** is the most straightforward organization for this dataset. It is a simple mapping from the relational database model to this schema, a case of two-dimensional data model. A concrete example for this data schema is shown in Table 2.2(a). The row key is a combination of snapshot id, particle type and the particle index. Each column corresponds to a different attribute of the particles. The composite row key ensures that data within the same snapshot and of the same type is stored together. Therefore queries that focus on one snapshot should perform well in this schema since they will benefit from the data locality. The disadvantage of this data schema is that each row has a small amount of data, i.e., a single particle in a snapshot. As a result, many rows will end up being stored within a single region. Therefore, computations that focus on the data located in the region will cause region hot-spotting and will not sufficiently benefit from the coprocessor-based parallelism. In addition, when a query needs to examine particles across different simulation snapshots or different types of particles, many irrelevant rows will have to be scanned, which, we anticipate, to slow the performance greatly.

Data Schema2 is a three-dimensional data schema. The row key is composed of the particle type and the particle index. The columns hold the values of the particle properties, as the particles are changing over time. Table 2.2(b) demonstrates how the data is organized in this schema with some specific data. Compared with Data Schema1, Data Schema2 makes use of the version dimension to store the snapshot information. This kind of data grouping across snapshots leads to good data locality, for queries examining one particle across snapshots. In addition, it improves the distribution of data across the regions. This data schema still follows the same sequential row key as that in the Data Schema1. When it comes to the computation which only focuses on a range of particles, the region hot-spotting would still occur.

Data Schema3 is an improvement over Data Schema2, in terms of the potential region hot-spotting issues. A case of Data Schema3 is presented in Table 2.2(c). The only difference between Data Schema3 and Data Schema2 is the row key, which is designed as the reversed particle index in order to "disorder" the particles and to distribute particles of the same type across the nodes of the cluster. It can avoid the hot-spotting issues by distributing the sequential particle across the cluster. It can be seen as mimicking hashing of particulars, which is good at querying the scattered data but weak in range query. This data schema gets round the problems existing in the previous data schema, while it loses the data locality strengths in the two data schemas. It is competitive when it comes to dealing with big data and huge computations. However, it cannot show its value in the case of puny computations, which is the inherent shortcoming of hashing partition.

2.4.3 Three Alternative Data Schemas for Bixi dataset

Table 2.4 summarizes three alternative data schemas for the Bixi dataset. This data set is different from the cosmology data set in that, although the number of individual data elements to be tracked over time is relatively small (404 bicycle states as opposed to 321,065,547 particles), there is a longer history of this data over time (100,800 snapshots as opposed to 9). Therefore, in designing the three Bixi data schemas, we focused on experimenting with

Table 2.4: Three Alternative Data Schemas for the Bixi Dataset

Data Model	**Row**	**Column**	**Version**
Schema1	hour-sid	minutes	current time
Schema2	hour-sid	monitoring metrics	minutes [0,59]
Schema3	day-sid	monitoring metrics	minutes [0,1439]

different numbers of values stacked on the version dimension, and the impact of this choice on the performance of a single query.

The **Data Schema1** belongs to the two-dimensional category of data models. In this schema, the row key is constructed as a combination of hourly timestamp and the station id. Each column is the offset of the minute in one hour. Each cell contains the values for all station-specific metrics, as a comma-separated sequence. Accordingly, each row includes metric values generated in one hour. A sample of Data Schema1 is shown in Table 2.5(a).

In **Data Schema2**, similar to Data Schema1, the row key consists of the hourly timestamp and station id. Data Schema 2 is a three-dimensional data schema, and it stores the values recorded for a particular station every minute over the hour in the version dimension. Instead of having all station metrics in a single cell, named groupings of metric, i.e., "metrics1", "metrics2", etc, are stored in separate correspondingly named columns. In Data Schema2, just like in Data Schema1, each row includes the metrics recorded for each station in one hour. See detailed information about Data Schema2 in Table 2.5(b).

Data Schema3, another instance of a three-dimensional data schema, is very similar with Data Schema2. The only difference between these two data schemas is that, in Data Schema3, the version dimension clusters the timestamps into hourly in Data Schema2, while in Data Schema3 it is grouped into daily. Hence in Data Schema3, each row includes metric values for one day. Table 2.5(c) shows a sample of Data Schema3.

2.5 Experimental Results

In this section, we discuss our experiments, including our experimental setup, the sample queries we designed on the two datasets, and the performance results for each query with different data schemas on both datasets.

2.5.1 Environment Setup

Our experiments were performed on a four-node cluster, running on four virtual machines. The four virtual machines run on IBM System X x3500 M2, which has 8-core, 64 GB RAM machine, 8 hard drivers in a RAID 5 configuration, and uses VMWare to host a set

Table 2.5: Examples of Data Schemas for the Bixi Dataset

(a) Data Schema 1

timestamp-sid	**0**	**1**	**...**	**30**	**...**	**59**
2010010101-001	(2,3)	(5,4)	(...)	(10,3)	(...)	(0,3)
...	(...)	(...)	(...)	(...)	(...)	(...)
2010010201-001	(1,4)	(3,6)	(...)	(1,12)	(...)	(3,0)

(b) Data Schema 2

timestamp-sid	**metrics1:[m0-m59]**	**metrics2: [m0-m59]**
2010010101-001	[2,5,...,0]	[3,4,...,0]
...	[...]	[...]
2010010201-001	[1,3,...,0]	[4,6,...,0]

(c) Data Schema 3

timestamp-sid	**metrics1:[m0-m1439]**	**metrics2: [m0-m1439]**
20100101-001	[2,5,...,0]	[3,4,...,0]
...	[...]	[...]
20100102-001	[1,3,...,0]	[4,6,...,0]

of virtual machines. The virtual machines have 2 cores, 8GB of RAM, and a 50GB disk. And they are running 32 bit Ubuntu 10.04. We used Hadoop version 0.20.2, and HBase version 0.93, re-compiled from source to suit our requirements of using the coprocessor framework. Hadoop and HBase are each given 1GB of memory in every running node. HDFS is configured with a replication factor of 2. HBase is managing its own Zookeeper instance running on the same machine as the HMaster. HBase and Hadoop are kept as the default configuration except reconfiguring 5K caching size [2]. For all test cases, we ran the experiment 5 times and took the mean of the last three.

As we have already discussed, our experiment is designed to investigate the differences in performance of read-heavy queries when using different data schemas for the same dataset. The experiment is based on a system which enables users to create a table in HBase, store their data, and process the queries. There are three important components in our system: the *TestClient*, the *HBaseClient* and the *User-Level Coprocessor*. We implemented the user-level coprocessor for both datasets respectively, named as CosmoCoprocessor and BixiCoprocessor. These two implemented coprocessors should first be deployed on the *HBase*

[2] This is the property hbase.client.scanner.caching controls scanner caching. That means how many rows will be fetched from the server in a single round trip in a scan if it is not served from (local, client) memory.

Region Server, before the experiment and instantiated in run time during the experiment.

At run time, the *TestClient* generates the query loads for each dataset according to a pre-defined configuration and sends each individual query to the *HBaseClient*. *HBaseClient* handles the query requests according to the query identifier. The *HBaseClient* has two objects, a callable and callback pair, for each query. The callable object is used to envelope method invocations to the server, using the coprocessor RPC framework. The callback object is invoked when results for the above call become available from the coprocessor [13]. When it receives a query, the *HBaseClient* invokes the caller function which invokes a RPC call to the *region servers*. The RPC calls are received by HBase regions and executed as a batch process. The regions who should handle the RPC calls are determined by the match between the queried range and the range for which each region is responsible. After the coprocessor has completed the task, it returns the results to the client. The callback object in *HBaseClient* performs the aggregation of results from the various region coprocessors. It should be noted that the calls from client side are executed on the corresponding regions in parallel.

2.5.2 Sample Queries

We designed three queries for the Cosmology dataset and one query for Bixi dataset. There are two big challenges in analysing the Cosmology dataset with the existing strategies [11]. First, with the size of simulations growing fast, the data analysis cannot be efficiently performed on shared-memory platforms, with the existing serial data analysis software. Second, the simulation snapshots cannot be loaded into memory efficiently because of the low I/O bandwidth of a single node. As a result, the queries that filter and correlate data from different snapshots have very large memory requirements and become highly I/O constrained. We want to take advantage of HBase platform to explore potential performance improvements to address these challenges.

The three queries we experimented with are inspired by the queries that astronomers might be interested in, as they explore the changes in the constitution of particles over time.

Cosmology Query1: Given a type of particle, a snapshot, a property and an expression for the property value, get all the particles of this type in the snapshot whose property matches the expression. This query invokes a range scan in one snapshot

Cosmology Query2: Given a type of particle and two snapshots, s1 and s2, get all the particles added or destroyed between s1 and s2. This query compares the data across two snapshots.

Cosmology Query3: Given a type of particle, a property, a set of particle ids and a set of snapshots, get the values of the property of the particles with these IDs across the selected snapshots. This third query retrieves the data from multiple snapshots.

We chose to also experiment with the Bixi dataset because of its densely increasing timestamps. We designed a single query for this data set to examine the performance implications of different lengths of data stacked on the version dimension, in the three-dimensional data models.

Bixi Query: For a given list of stations and a time, get their average bike usage for last 1, 2, 4, 8 and 16 days. Its boundary condition is to get such an average for all the 404 stations.

2.5.3 Experimental Analysis

For the Cosmology dataset, we performed experiments for all three queries as described in Section 2.5.2 with three data schemas shown in Section 2.4.2. For the Bixi dataset, one experiment was executed for the query described in Section 2.5.2 with three data schemas presented in Section 2.4.3.

All queries are processed in parallel by user-level coprocessors running server side. The execution times are affected by two parameters. First, range scan, as the basic operation, is the most expensive computation during processing. Consequently, the execution time becomes larger as the number of rows increases. Second, the coprocessor overhead becomes non-negligible when the range scan is not very large. Different row-key design in data schemas determine different range scan, and different data schemas demonstrate the different region server distribution with the same configuration of region size.

Analysis of Cosmology Dataset

Table 2.6(a) shows performance results for Query1, with five scenarios under three different data schemas. These five scenarios try to look up all particles in one snapshot that match a set of conditions. For example, in the first row of the table the conditions, "2;pp;tform;>0.01;84", refers to returning all particles whose *type* is 2 and property *tform* is above 0.01 at snapshot 84. *pp* in the condition, composes the column names along with the properties of particles As the particle index is a part of row key in all three data schemas, the execution time is almost entirely contingent upon the number of particles in the snapshot. Comparing with Data Schema2 and Data Schema3, Data Schema1 provides better performance in all scenarios. As the particles within the same snapshot and of the same type are stored as neighbors, only a few number of regions need to be examined for this query. Since Data Schema2 and Data Schema3 group all snapshots of one particle together, particles with neighboring IDs are scattered across more regions, and consequently, more regions must be involved in this query. As more regions are accessed, more overhead is caused by the additional coprocessor instantiations. As a result, Data Schema1 performs the best for this query.

The experiment for Query2 measures the performance when the queried data is spread in two snapshots. The results are shown in Table 2.6(b). It should be noted that “NA” in the following tables means the result could not be obtained because the queries were timed out under the time-out configuration of 5 minutes. Nine query loads are designed to get all particles destroyed between snapshot S1 and S2. For example, in the first row of the table the conditions, “2;29;24”, represents all star particles existing in snapshot 29 but not in snapshot 24. Here, “2” indicates the particle type is *star*. As the number of particles that need to be compared across the two snapshots, the execution times of the query under Data Schema2 and Data Schema3 increase but not substantially. On the other hand, the last five scenarios fail under Data Schema1. It is also interesting to note that in the first four scenarios, the query performance under Data Schema1 performs much better than that under Data Schema2 and Data Schema3, which is due to the coprocessor overhead that the two latter data schemas suffer. This overhead is however dominated by the cost of the last five bigger queries. So we can conclude that Data Schema2 and Data Schema3 are suitable for computations or queries on large scale datasets.

In table 2.6(c), we show the execution times for nine queries involving 10 to 1450 particles. For example, the first row of the table stands for a query that is to retrieve the values of *eps* for 10 *star* particles whose indexes start from 33554444 over the snapshots defined in the last vector.

Our hypothesis here was that Data Schema3 would perform better because it evenly distributes the data across clusters, which is what the data of Table 2.6(c) reflect. There is only one region involved under Data Schema1, while there are multiple regions involved under Data Schema2 and Data Schema3. This means that under Data Schema3 the data is better distributed, which results in the good computation distribution and load balancing across the nodes. We can also observe that the limit of all three data schemas when serving queries across all snapshots. As this query relates to all the rows, all regions are called for this query along with a coprocessor instantiated. As many coprocessors are running in one HBase Region server in parallel, more resources (including memory, CPU and I/O bandwidth) are required; limited resources lead to the delay of coprocessor which results in HBase HRegion server crash and the time-out exception from Zookeeper. Intuitively, this phenomenon points to the fact that, in addition to a well designed data schema, more performance might be achieved with a bigger number of nodes in the cluster.

Analysis of Bixi Dataset

The Bixi query was evaluated with ten scenarios whose execution times are shown in Table 2.7 and Table 2.8. The distributions of working regions, i.e., regions on which the coprocessor instances run, for five of these scenarios are shown in Table 2.8. The distributions of working regions are expressed in vectors in which each element means the number of working regions in the corresponding HBase region server. From the left to right, the host names of HBase region server in this experiment are HBase2, HBase3,HBase4, and HBase7. The scenarios are designed for getting change trend of 100/200 stations in a period of time.

In Table 2.7, Data Schema3 shows better performance than the other two, and Data Schema2 shows better performance than Data Schema1 for large queries. Both three-dimensional data schemas perform better than the two-dimensional data schema. In addition, Data Schema3, which localizes more values in the version dimension, obtains more benefits from the locality of the data than Data Schema2. In Table 2.8, Data Schema3 has better performance than Data Schema2 in the first three scenarios. This is because the working regions in Data Schema3 are better distributed in the cluster. From the last two scenarios, we can see that the execution time increases rapidly when there are two regions on one HBase region server in Data Schema3, although there is no significant difference between these two data schemas. This phenomenon might be caused by the limited memory resources for coprocessors to execute and for the resulted data to be collected. This indicates, not surprisingly, that cluster configuration is extremely important in terms of performance. In addition to the data schema, better performance might be achieved with an appropriate number of nodes and corresponding data volumes.

2.6 Discussion

In this section, we discuss broader issues related to the three-dimensional data model and comment on the types of applications that can benefit from it, on HBase.

"Qualitative" versus "Quantitative" Suggestions The three-dimensional data model only suggests how to organize the data at a high, "qualitative" level. It does not provide specific suggestions to developers for making decisions on (a) how many columns and col-

Table 2.6: Execution Times for the Cosmology Queries across the three Data Schemas

(a) Query 1

Query1	**Schema1(s)**	**Schema2(s)**	**Schema3(s)**	**Number of Particles**	**Number of Comparisons**
2;pp;tform;>0.01;84	14.383	38.502	34.427	907,021	907,025
2;pp;tform;>0.08;128	17.722	42.751	42.243	604,567	2,743,966
2;pp;tform;>0.05;128	23.208	44.445	42.498	2,237,646	2,743,966
2;pp;tform;>0.08;216	38.496	54.290	54.322	4,257,556	6,396,955
2;pp;tform;>0.08;512	62.361	87.981	87.142	10,278,145	12,417,544

(b) Query 2

Query2	**Schema1(s)**	**Schema2(s)**	**Schema3(s)**	**Number of Particles**	**Number of Comparisons**
2;29;24	0.197	27.434	28.464	4,277	5,568
2;60;24	3.342	32.550	30.481	257,928	67,268
2;84;24	6.446	38.128	34.551	905,734	907,025
2;128;24	14.797	44.611	45.000	2,742,675	2,743,966
2;216;128	NA	52.273	49.012	3,652,989	6,396,955
2;216;24	NA	53.325	55.783	6,395,664	6,396,955
2;512;216	NA	61.991	81.325	6,020,589	12,417,544
2;512;128	NA	66.709	80.449	9,673,578	12,417,544
2;512;24	NA	79.113	76.163	12,416,253	12,417,544

(c) Query 3

Query3	**Schema1(s)**	**Schema2(s)**	**Schema3(s)**	**Number of Particles**
2;pp;eps;[33554444,10]; [24]	44.096	42.515	30.435	10
2;pp;eps;[33554444,10]; [24,512]	50.406	45.986	33.559	20
2;pp;eps;[33554444,10]; [24,60,128,512]	64.306	46.061	33.192	40
2;pp;eps;[33554444,10]; [24,29,60,84,128,512]	97.370	48.634	33.757	60
2;pp;eps;[33554444,10]; [24,36,45,60,84,128,216,512]	177.889	50.636	35.527	80
2;pp;eps;[33554444,50]; [24,29,84,512]	NA	56.561	47.775	200
2;pp;eps;[33554444,50]; [24,29,36,45,60,84,128,216,512]	NA	110.301	59.602	450
2;pp;eps;[33554444,100]; [24,29,36,45,60,84,128,216,512]	NA	429.498	162.808	900
2;pp;eps;[33554444,150]; [24,29,36,45,60,84,128,216,512]	NA	NA	NA	NA

Table 2.7: Execution Time of the Bixi Query

Query1	**Schema1(s)**	**Schema2(s)**	**Schema3(s)**
1day-200stations	1.1	1.4	0.4
2day-200stations	1.9	3.6	0.6
4day-200stations	2.5	4.0	1.2
8day-200stations	12	4.8	4.2
16day-200stations	13.8	7.3	6.2

Table 2.8: Working Region Distributions for Schema2 and Schema3 for Bixi dataset

Query1	**Schema2**		**Schema3**	
	Execution Time(s)	**Working Regions Distribution**	**Execution Time(s)**	**Working Regions Distribution**
1day-100stations	1.258	(0,2,0,0)	0.47	(1,0,0,0)
2day-100stations	1.779	(0,2,0,0)	0.579	(1,0,0,1)
4day-100stations	2.566	(0,2,0,0)	1.161	(1,1,0,1)
8day-100stations	4.280	(0,2,1,0)	4.376	(1,1,0,2)
16day-100stations	5.839	(1,2,2,0)	5.401	(1,2,2,2)

umn families should be for their dataset, (b) how "deep" the version dimension should be kept, or (c) how to design the composite row key. Actually, it is really hard to provide a specific data organization plan as there are so many factors affecting the query performance in HBase, including the dataset characteristics, the data-access patterns and the HBase cluster configuration. However, our experiments and the performance results presented in this chapter can, we hope, be used as a reference in data-schema design.

The ***Apache HBase*** is a relatively new project. The latest version of HBase is 0.94 which was just released in May, 2012. Many functions are not very stable, including the functionalities around versioning. It cannot be avoided that there are some defects during developing an application. Moreover, as HBase is still in the early stages of development, some interfaces are not very convenient to use. But HBase community is striving to meet users' expectations. Ease-of-implementation and robustness concerns aside, this three-dimensional data model in this study can broaden one's views about how to organize the data in HBase or other NoSQL databases.

Dynamic Data versus Static Data The three-dimensional data model is designed to support dynamic data, over time. In the case of datasets that have both static and dynamic data, we suggest that the static data should be stored separately. For example, in Bixi dataset, we can store the static attributes of each station into a separate table[13].

Historical Dataset versus Real-Time Datasets This three-dimensional data model can be used in historical time-series data analysis, or for "write-once read-many" applications, with rare updates. As the "version" dimension in HBase was intended to guarantee concurrency and consistency, this data model cannot be directly used for on-line transaction processing, or for write-intensive applications without any other synchronization mechanism.

Supported versus Non-Supported Datasets In some applications, there are multiple types of data objects; in this chapter, we discussed how this model may be used to organize and store data sets with a single data-element type, i.e., particles and station observations. In other words, complex data sets with multiple object types and relations among them are outside the scope of this data model.

This data model can support in a straightforward manner several types of data sets. First, it is ideal for monitoring metrics: the "version" dimension can be used to store the stacked values of different metrics in time, each metric can be assigned a corresponding column, and the row key will be associated with a period, such as week, month or year. A typical example might be a health-monitoring system collecting metrics at regular intervals. Another example is real-time sensor-based systems, where search is the primary required functionality and updates (almost) never happen.

Complex objects, whose properties change over time, can also be stored with this data model. The row key can be named as the object id, columns represent the object properties, and the version dimension can represents the version index. For example, source-code modules could be stored in this data model, with each file corresponding to an object, and each new committed module version would constitute a new object stacked in the version dimension. The various columns might be associated with meta-data and static-analysis metrics of this file, owners, creators, related bugs, lines of code and so on.

2.7 Conclusion and Future Work

HBase, as a NoSQL database offering, is rapidly becoming the chosen solution for scalable data processing. In this chapter we proposed a three-dimensional data model in HBase for large time-series dataset analysis. This three-dimensional data model provides a new view of data organization and management by using HBase version dimension in a different way.

We have experimented and evaluated the performance impact of this type of data models with two data sets, of different sizes and different time lengths. For each of these data sets, we have compared the performance of several ad-hoc queries, implemented with co-processors, across different data schemas, some of which (do not) use the third HBase dimension. The experiment results show improved performance with the data schemas that use the third dimension of HBase. Our experiments also show that performance is highly impacted by the distribution of the data across cluster nodes, which implies that the design of the row-key is of significant importance. At last, we discussed the application scope for the proposed data model.

There are still several problems to solve. Besides the performance impact, the three-dimensional data model should be evaluated from scalability, elasticity and utilization aspects. Given the feature of the three-dimensional data model, how to extend its applicable scope to on-line transaction processing system is valuable and challenging. In addition to the time-series dataset, many other datasets such as spatial dataset and graphic dataset should also be investigated to suggest the data management design in the future.

Acknowledgement

The authors wish to thank Himanshu Vashishta for many interesting conversations on HBase. This work has been funded by the SAVI Strategic Network, NSERC and AITF.

Bibliography

[1] Bixi Dataset. `https://s3.amazonaws.com/bixidata/bixi_comp.tar.gz`.

[2] Modeling Time Series Data on top of Cassandra. `http://engineering.rockmelt.com/post/17229017779/modeling-time-series-data-on-top-of-cassandra`.

[3] Whats OpenTSDB? `http://opentsdb.net/`.

[4] D. Agrawal, S. Das, and A. El Abbadi. Big data and cloud computing: current state and future opportunities. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 530–533. ACM, 2011.

[5] R. Agrawal, A. Ailamaki, P.A. Bernstein, E.A. Brewer, M.J. Carey, S. Chaudhuri, A.H. Doan, D. Florescu, M.J. Franklin, H. Garcia-Molina, et al. The claremont report on database research. *ACM SIGMOD Record*, 37(3):9–19, 2008.

[6] D. Borthakur, J. Gray, J.S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, et al. Apache hadoop goes realtime

at facebook. In *Proceedings of the 2011 international conference on Management of data, SIGMOD*, volume 11, pages 1071–1080, 2011.

[7] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[8] C. Chen, G. Chen, D. Jiang, B. Ooi, H. Vo, S. Wu, and Q. Xu. Providing scalable database services on the cloud. *Web Information Systems Engineering–WISE 2010*, pages 1–19, 2010.

[9] Apache Software Foundation. Apache HBase Reference Guide, April 2012.

[10] R. Hecht and S. Jablonski. Nosql evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 336–341. IEEE, 2011.

[11] S. Loebman, D. Nunley, Y.C. Kwon, B. Howe, M. Balazinska, and J.P. Gardner. Analyzing massive astrophysical datasets: Can pig/hadoop or a relational dbms help? In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[12] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M.A. Hassaan, R. Kaleem, T.H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, et al. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 12–25. ACM, 2011.

[13] H. Vashishtha and E. Stroulia. Enhancing query support in hbase via an extended coprocessors framework. *Towards a Service-Based Internet*, pages 75–87, 2011.

Chapter 3

HGrid Data Model for Geopspatial Datasets [1]

3.1 Introduction

With the explosive increase of location-aware devices (GPS-enabled smartphones and vehicles, RFIDs, tablets, etc) and the proliferation of sensor-based systems, location-based services that contextualize the user experience are growing. A prominent example of this phenomenon is location-aware advertisement and recommendation, where the user is provided with advice on real-service opportunities close to her. Taking advantage of these location-aware services are millions of users, who continuously register their location updates through their wireless providers. In addition to user-facing services, smart systems embed sensors and activators in our environment for monitoring and management; these systems also generate massive amounts of data updates and rely on analyzing this data over time and across space.

The challenge with these applications is how to guarantee satisfactory performance for real-time analysis, while at the same time, supporting millions of location updates per minute. To address these requirements, database-management systems (DBMS) must scale up while maintaining good load balancing and high up-time [14]. As most typical queries of these applications involve the retrieval of multi-attribute values related with some proximity function to a given geographic location, efficient multi-dimensional geospatial data access is also an important requirement. Relational database-management systems (RDBMSs) support efficient spatial queries with special-purpose index structures, such as K-d tree [1], quad-tree [4] and R-tree [7]. However, RDBMSs are challenged by the scaling requirements of this new breed of applications, requiring complex hardware setup and configura-

[1]A version of this chapter has been published. Dan Han and Eleni Stroulia, 2013 IEEE 6th International Conference on Cloud Computing (CLOUD), pages 910-917.

tion. NoSQL (Not Only SQL) databases, endowed with availability, elasticity and scalability through their easy deployment on cloud-computing platforms, become a more attractive solution for these applications [2]. However, data in NoSQL databases is stored in an unstructured manner, based on a primary key and attributes organized in column families. Even though some rough guidances about schema design have been provided by the specific NoSQL database offerings, such as HBase [5], there is still no systematic method for how to actually design the structure of the "NoSQL Big Tables" for a particular application. The data organization has a great impact on the performance of the queries implemented on these tables, and therefore, *a systematic method for NoSQL data-schema design for geospatial applications* is a timely and important problem in this area.

This is exactly the problem we aim to address with our work on HBase, the open-source implementation of BigTable [3]. To that end, we have developed the *HGrid* data model for organizing geospatial data sets, a hybrid two-tier index structure, tailored to the HBase three-dimensional storage mechanism. The primary index is a quad-tree that divides the data space into rectangular tiles, and encodes each tile according to a Z-ordering traversal [12]. Next, a regular-grid index structure is used to divide each quad-tree tile into a sequence of contiguous rectangular cells. Each cell is assigned a unique identifier, constructed as the concatenation of the cell's *row index* and *column index* in a grid. In this data model, the *row key* of each data point is the concatenation of the *z-value of the quad-tree tile* in which the data point belongs, and the *row index of its regular-grid cell* in the second-tier regular-grid index. The *column name* is constructed by concatenating the *column index of its regular-grid cell* and the *object id*. Finally, the various attributes of each object are stored in *the third dimension*.

We empirically evaluated the performance of this data organization with two synthetic data sets, with uniform and skewed data distribution correspondingly. Compared against a pure quad-tree data model and a pure regular-grid data model, we found that *HGrid* can be flexibly configured for a range of cell sizes, and although it exhibits slightly poorer performance than the regular-grid data model, its index requires less space than the corresponding quad-tree and regular-grid indices, which makes its deployment possible with less resources. It is more scalable and suitable for homogeneously covered and discontinuous spaces.

The rest of the chapter is organized as follows. Section 3.2 reviews the background of this work on geospatial data, multi-dimensional index structures, linearization methods, HBase and introduces related works of geospatial data studies. By comparing with quad-tree against the regular-grid data model, we describe the *HGrid* data model and evaluate it with range query and k-nearest neighbor query in Section 3.3. Section 3.4 reports the experiment result with different data distributions under different data models and summarizes a set of suggestions about HBase schema design and query implementation. We conclude our contributions and future work in Section 3.5.

3.2 Background and Related Work

The data points in geospatial data sets are typically multi-dimensional, including their coordinates (latitude and longitude), a timestamp, and a description (identifier and attributes) for the domain object at the data point [14]. Range queries (identifying the data points within a radius from a given location) and k-nearest neighbor queries (identifying the k data points closest to a given location) are the most common queries on these data sets.

3.2.1 Multi-Dimensional Indices

Spatial data are typically organized using “space-driven” or “data-driven” indices. In data-driven structures, such as R-tree [7], the distribution of the objects to be stored determines the partitioning of space. Since the most common queries in geospatial applications are typically based on locations, in our work we focus on space-driven approaches to data organization. An example of “space-driven” organization is a grid where objects are associated with a grid cell based on their position in the space, and an index of grid-cell identifiers enables rapid access. In this organization, the grid-based spatial index is created first and the data is added incrementally without causing any change to the index structure.

The ***regular grid*** is the simplest grid-based spatial index. It partitions a rectangular domain using rectangular cells of equal size [6]. An associated matrix, i.e., a two-dimensional array, maps each grid cell to the array of data points located within the space covered by the cell. The ***quad-tree*** recursively splits the space into subspaces organized in a search tree. Two methods are commonly used to split the given space [14]: the trie-based approach splits the space at the mid-point of a dimension, resulting in equal-size subspaces. The point-based technique splits the space in subspaces with equal number of data points [1]. Quad-tree is commonly coupled with space-filling curves [10] to linearise the sub-spaces. Z-ordering [12] is an easy-to-compute example of a space-filling curve. We used it in this work because of its simplicity.

3.2.2 HBase Storage Model Overview

HBase uses the Hadoop File System (HDFS) as its underlying data-storage platform. Unlike the two-dimensional tables of traditional RDBMSs, HBase organizes data in a three-dimensional cube. The basic data storage unit in HBase is a cell, which is identified with its *row key*, *column-family name*, *column name* and *version*[5]. Cells with multiple versions of data can be stacked in the third dimension. For example, the third dimension is used to stack the contents of message IDs in the Facebook messaging system [13]. At the physical level, each column family is stored contiguously on disk, and the data is physically sorted by *row key* , *column name* and *version*.

The HBase *Coprocessor* framework, inspired by Google's BigTable *Coprocessor* [3], provides a library and run-time environment for executing user code on the HBase region servers. Coprocessor implementations are executed remotely at the target region(s) hosted by region servers, and their execution results are returned to the client. This design decreases the communication overhead involved in transferring data from the region servers to the client, and enables dramatic performance improvement by pushing the computation to the server, where it can operate on the data directly. To reap the benefits of this framework, an appropriate partitioning of the data is necessary, which implies the need for a well-designed data schema. This is the reason why, in our work, we have focused on investigating the impact of different HBase table schemas on the performance of query execution using the *Coprocessor* framework.

To date, two proposals have been put forward for the organization of geospatial data in HBase. S. Nishimura et. al[14] built a multi-dimensional index layer on top of HBase to perform spatial queries. Ya-Ting Hsu et. al [9] presented a novel key-formulation schema, based on R+-tree for spatial index in HBase. Both studies investigate how to efficiently access the multi-dimensional data with spatial indices, which is part of the problem that we are addressing in this chapter. Their methods demonstrate efficient performance with the spatial indices. However, they only focus on the design of the HBase *row key* with no or little discussion about the *column* and *version* design. To design an appropriate data model for geospatial datasets, which can be easily and directly applied to geospatial applications, in addition to the *row key*, one need also take into account the design of the *column name* and the role of the third dimension. Furthermore, in our work, we implemented the queries with HBase *Coprocessor* to harness the parallelism benefits, while the above studies processed the queries with HBase *Scan*.

3.3 The *HGrid* Data Model for HBase

In this section, we first review the two data models underlying the design of *HGrid*. Next, we describe the *HGrid* data model and the index-construction process. Finally, we explain the implementation of two queries commonly used in location based service under this data model.

3.3.1 Preliminary Data Models

As we have already discussed, the *HGrid* data model for HBase is inspired by two simpler data models: the quad-tree data model and the regular-grid data model.

The Quad-Tree Data Model

The quad-tree data model relies on a trie-based quad-tree index, where Z-ordering [12] is applied to transform the two-dimensional spatial data into a one-dimensional array. In this model, the *row key*, which should be kept as short as possible, is the Z-value in decimal encoding, i.e., “0”,“1”,“2”,“3”. The *column name* is the object ID, and each cell stores one data point in JSON format.

There are three performance concerns about this data model. First, as the quad-tree becomes deeper, the data points end up being organized into more rows. As a result, queries have to scan more grid cells in order to retrieve all data points within a range, or close to a location; at the same time, the more rows are scanned, the more unrelated-to-the-query data is accessed, causing performance deterioration. Moreover, the Z-ordering linearization technique, although appealing because of its simple computation, does not maintain good data locality, and subsequent grid cells are not necessarily close to each other in space. As a result, scanning more contiguous rows is even more likely to inspect irrelevant rows (i.e., rows corresponding to grid cells not sufficiently close to the query location), which increases the amount of accessed data and causes performance to suffer further. The last but not least issue is with the construction of the quad-tree. If the index is built in real time for each query, the construction cost dominates in small queries. If the index is maintained in memory, the granularity of the grid is limited by the amount of memory available, since the memory needed to maintain the index increases as the depth of the tree increases and the size of the grid cells becomes smaller.

The Regular-Grid Data Model

In the regular-grid data model, the *row key* is the row index of the cell in the grid, the *column name* is the column index of the cell, and each storage cell represents one object in JSON format holding all other attributes and values. The third dimension holds a stack of data points located in the same grid cell, and an index is maintained to keep the count of objects in each cell stack in order to support updates.

This index structure maintains data locality: data points close in the *y* dimension are likely to be in the same or neighboring rows and data points close in the *x* dimension are likely to be in the same or neighboring columns. With this data model, the data point can be determined efficiently by both of row and column and the unrelated data can be pruned with the Bloom filter.[2] However, in densely populated spaces, the number of objects in columns in each row increases. Because the time to retrieve a row with n data points more than doubles with n (when n is large) [9], the query performance will decrease. In addition, as there is no mechanism to filter the objects located in one column in this data model, more objects are retrieved in a query, which results in a higher number of false positives. A finer-grained grid would reduce the false positives but it would imply a larger cell-stack in-memory index, which may not be possible due to memory limitations. Given a certain amount of memory, this data model reaches a bottleneck when it comes to a large space with finer-grained grid cells.

Summarizing the relative advantages and disadvantages of the quad-tree and regular-grid data models, we note that the quad-tree data model is not efficient when it comes to large queries, as more irrelevant data must be scanned. The regular-grid data model is preferable in that respect because it has better localization and can provide very good pruning of the unrelated data. Query processing becomes inefficient in the regular-grid data model when it comes to large high-density spaces, as the amount of objects grouped in one row increases rapidly. Both the quad-tree data model and the regular grid data model are constrained by the size of available memory, in terms of how fine-grained the grid cell may become.

Considering the advantages and disadvantages of these two data models, we designed the *HGrid* data model. Using a two-level index structure, the *HGrid* data model avoids the regular-grid drawback by splitting large geographic spaces in tiles using a quad-tree index, and takes advantage of the localization feature of the regular-grid data model in the

[2]The Bloom filter is a space-efficient probabilistic data structure to be used to check whether an element is a member of a set [15]. It is supported in HBase to reduce the disk lookups for unrelated rows or columns.

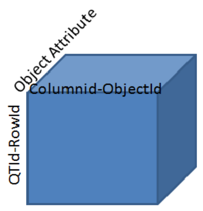
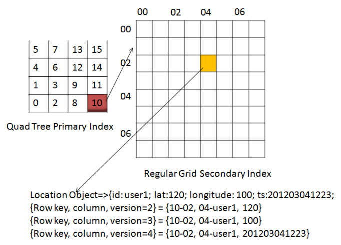


Figure 3.1: *HGrid* Data Model



Location Object=>{id: user1; lat:120; longitude: 100; ts:201203041223;
{Row key, column, version=2} = {10-02, 04-user1, 120}
{Row key, column, version=3} = {10-02, 04-user1, 100}
{Row key, column, version=4} = {10-02, 04-user1, 201203041223}

Figure 3.2: An example of *HGrid* Index

second-level index.

3.3.2 The *HGrid* Data Model Representation

Figure 3.1 diagrammatically depicts the *HGrid* data model. First, the space is divided into equally sized rectangular tiles *T*, encoded with their Z-value. Next, the data points are organized in a regular grid of continuous uniform fine-grained cells. In this model, each data point is uniquely identified in terms of its *row key* and *column name*. The *row key* is the concatenation of the quad-tree Z-value and the regular-grid row index. The *column name* is the concatenation of the regular-grid column index and the object id of the data point. The attributes of the data points are stored in the third dimension.

Figure 3.2 illustrates with an example how the *HGrid* index is constructed. Given a specification of the overall space within which the geospatial data set is to be contained, the minimum boundary rectangle (MBR) of the space, i.e., the smallest rectangle that completely contains the space, is computed. The depth of the quad-tree is determined by the user-specified size of the tile. Each tile is associated with an index that corresponds to its rank according to the Z-ordering linearization. Each quad-tree tile contains all the data points whose coordinates belong in the space covered by the tile. Clearly, many data points may belong in the same tile and share the same tile code; there also may be empty tiles, with no data points at all. For the empty tiles, there are no relevant records stored in HBase.

Next, given the desired regular-grid resolution, the quad-tree tiles are decomposed into equally sized rectangular cells. Each cell is coded with the row and column index of the regular grid; this cell code becomes the secondary index for every data point in the cell.

There are two challenges in the configuration of this index-construction process: deciding (a) the appropriate granularity for tessellating the original space in quad-tree tiles, and (b) the appropriate granularity of the regular grid in the second stage. Based on our experience, we have found that the best resolution depends on the data distribution and the likely queries. The finer the quad-tree tile granularity, the more irrelevant rows will have to be pruned from the return set of *Scan* queries due to poor locality of Z-ordering. Alternatively, if direct *Get* access operations are used, more sub-queries will be required. Therefore, there is a trade-off between the size of tiles in the first stage and the size of cells in the second stage. Much experimentation with different levels of quad-tree and granularity of regular grid is needed in order to optimize the performance for a specific data set.

3.3.3 Query Processing

There are two ways for processing queries in HBase. Using a *Scan* operation, a number of rows corresponding to a range of row keys are retrieved and the response set is computed at the client-side. Using *Coprocessor*, partial response sets are computed in each region and are then aggregated at the client side.

Range queries are commonly used in location-based applications. Given the coordinates of a location and a radius, a vector of data points, located within a distance less than or equal to the radius from the input location, is returned. Relying on the *HGrid* data model and using *Coprocessor*, answering this query involves the following steps.

(1) Given the query input location and the range, the minimum bounding square that completely includes the implied circle around the input location is computed.

(2) Next, the quad-tree tiles that overlap with the computed bounding square are identified. The Z-codes of these tiles provide the primary index of the HBase rows of interest.

(3) Next, the overlap between the bounding square and each intersecting quad-tree tile is computed to identify the regular-grid cells involved. Based on these cells, the secondary index of the rows to be examined and the corresponding column indices become available.

(4) Having now computed the range of rows and columns involved in the query, a sub-query is issued for each selected tile or the parent tile of selected continuous tiles of the quad-tree and processed by user-level *Coprocessor* on the HBase regions; the results of the sub-queries are accumulated at the client-side.

k-Nearest Neighbor (kNN) queries identify a number (k) of data points near to an input location. The process for computing kNN queries on the *HGrid* data model, using a *Scan*-based implementation, is as follows.

(1) First, we apply the density-based range estimation method introduced by Liu et.al [11] to estimate the search range.

(2) Given the search-range estimate, the queried indices of rows and columns are computed as described in steps 2 and 3 of *Range Query* above.

(3) Next, a *Scan* query is issued to retrieve the relevant data points.

(4) If fewer than k data points are returned, the search-range estimation is expanded and the above steps are repeated.

(5) Finally, a sorting step orders the return set in increasing distance from the input location.

3.4 Experimental Results

Our experiments were performed on a four-node cluster, running on four virtual machines on an Openstack Cloud. The virtual machines run 64bit Ubuntu 11.10 and have 2 cores, 4GB of RAM, and a 200 GB disk. We used Hadoop version 1.0.2, and HBase version 0.94. Hadoop and HBase were each given 2GB of Heap size in every running node. HDFS was configured with a replication factor of 2. HBase was managing its own Zookeeper instance running on the same machine as the HMaster.

In the experiment, gzip compression was configured on the table to reduce the data-transmission time. Next, the ROWCOL filter [3] was applied on each table for narrowing the queried range. The scan cache size was set to 5000 and the block cache was set to true, for the query processing. Finally, minor and major compaction were manually done to avoid the large size of store files after the data uploading.

For all test cases, we ran the experiment 5 times and we report the mean of the last three. We implemented the Range Query processing with the *Coprocessor* framework and kNN Query with *Scan*, considering the relatively small queried range in kNN Query.

Table 3.1: Execution Time of Range Query with Various Sizes of Cell of Three Data Models(s)

Radius	Target	QT (km)			RG (km)			HG (km)				
(km)	Objects	$\approx$0.1	$\approx$1	$\approx$10	0.1	1	10	50:0.1	$\approx$10:0.1	$\approx$1:0.1	$\approx$:10:0.01	$\approx$10:0.001
0.01	1	0.112	0.252	7.710	0.131	0.208	6.196	0.208	0.185	0.211	0.188	0.177
0.05	72	0.145	0.249	7.743	0.135	0.221	6.242	0.222	0.231	0.178	0.202	0.241
0.1	315	0.141	0.240	7.731	0.147	0.213	6.257	0.246	0.238	0.175	0.225	0.292
0.5	7,868	0.539	0.692	7.644	0.285	0.478	6.277	0.504	0.509	0.454	0.556	0.906
1	31,411	0.846	0.767	8.252	0.572	0.870	6.232	0.914	0.926	0.803	1.052	1.166
4	502,587	8.787	7.655	9.589	4.544	5.763	7.711	6.224	6.410	7.426	7.243	7.619
8	2,012,583	**NA**	**NA**	**NA**	10.693	16.782	34.468	83.920	42.372	40.542	51.637	51.918
12	4,524,996	**NA**	**NA**	**NA**	27.545	34.576	39.570	*NA*	85.014	93.343	105.635	110.967

[3] This is a type of Bloom Filter. When applied, the hash of the *row key,column family, column family qualifier* is added to the bloom on each key insert. It can help prune the data from both row and columns sides.

Table 3.2: False Positive in Range Query with Various Sizes of Cell of Three Data Models (%)

Radius	Target	QT (km)			RG (km)			HG (km)				
(km)	Objects	≈0.1	≈1	≈10	0.1	1	10	50:0.1	≈10:0.1	≈1:0.1	≈:10:0.01	≈10:0.001
0.01	1	99.91	99.99	99.99	99.00	99.99	99.99	99.00	99.00	99.53	80.00	50.00
0.05	72	95.29	99.70	99.99	82.48	99.28	99.99	82.48	82.48	82.65	35.71	28.00
0.1	315	79.41	98.71	99.98	65.42	96.86	99.97	65.42	65.42	65.03	30.62	23.17
0.5	7,868	86.07	91.93	99.50	36.21	80.19	99.21	34.71	34.71	30.12	22.89	21.67
1	31,411	63.68	67.77	97.99	28.92	65.03	96.86	28.92	28.92	26.25	22.48	21.79
4	502,587	59.72	60.45	67.87	23.40	37.96	49.74	23.39	23.40	23.65	21.66	21.48
8	2,012,583	**NA**	**NA**	**NA**	22.43	30.43	77.65	22.43	22.43	22.19	21.56	21.47
12	4,524,996	**NA**	**NA**	**NA**	22.11	27.64	49.74	*NA*	22.13	22.37	21.54	21.48

3.4.1 The Data Set

For our experiments, we used two synthetic data sets because (a) we needed a "big" data set, with a sufficiently large number of data points, and (b) we needed to control the data distribution and its impact on the performance of the three different data organizations. The synthetic data set was generated based on the Bixi data set [8], which includes minute-by-minute readings from 404 bike stations around the city of Montreal. Each reading consists of the following attributes: timestamp, station id, latitude, longitude, station name, terminal name, number of docks, and number of bikes. With the same format of station object in Bixi data set, the synthetic data set augments the number of stations from 404 to one hundred million, locating them in random coordinates, following a uniform and Zipf distribution [4], using *commons-math3-3.0.jar*. The factor of Zipf distribution is 1.0, which represents moderately skewed data. The simulated space area is 100km*100km. This data set basically represents one hundred million objects at a certain timestamp. The size of data set is about 70GB.

3.4.2 Index Configuration

The granularity of the cells in terms of which the space is tessellated is a very important variable that substantially affects the query-processing performance. This is why, before we compare the three data models against each other, we have explored the "best" cell configuration for each model. In this first round of experiment, we varied the size of the cell to observe how different cell sizes affect the performance of each data model. We set the size of the cell at 0.1km, 1km, and 10km. Consequently, in the regular-grid index, the 10,000km^2 space is divided into $1,000 * 1,000$, $100 * 100$, and $10 * 10$; in the quad-tree index, as the space is split at the mid-point of a dimension each time, the size of the cell is

[4]Skew distribution can follow common distributions, such as Zipf, Gaussian, and Poisson, but many studies consider Zipf distribution to model skewed data.

less or equal to the configured value above.

Table 3.3: Execution Time of Range Query with Three Data Models (s)

	(a) Uniform Data								
Radius (km)	**0.01**	**0.05**	**0.1**	**0.5**	**1**	**4**	**8**	**12**	**16**
QT:≈1	0.251	0.250	0.240	0.692	0.767	7.656	**NA**	**NA**	**NA**
RG:0.1	0.131	0.135	0.147	0.285	0.572	4.544	10.693	27.545	45.323
HG:≈10:0.1	0.185	0.231	0.238	0.509	0.926	6.410	42.372	85.014	141.308
	(b) Skewed Data								
Radius (km)	**0.01**	**0.05**	**0.1**	**0.5**	**1**	**4**	**8**	**12**	**16**
QT:≈1	0.398	0.359	0.375	1.172	1.274	30.240	**NA**	**NA**	**NA**
RG:0.1	0.120	0.132	0.142	0.424	1.140	12.349	**NA**	**NA**	**NA**
HG:≈10:0.1	0.260	0.314	0.317	0.868	2.015	16.843	**NA**	**NA**	**NA**

The Quad-Tree and Regular-Grid Data Models

For our uniform-distribution data set of 100 million data points, configuring the individual cell to cover a 1km*1km space results in 100*100 square cells with an average number of 10,000 data points in each cell. As a row corresponds to a cell in the quad-tree data model, there are approximately 10,000 rows in total, and in each row, there are about 10,000 columns (one for each data point) with a depth of one. In the regular-grid data model, *row key*s correspond to the indices of the grid rows and the *column name*s correspond to the column indices; as a result there are at most 100 rows and 100 columns. In each cell, there may be a stack of about 10,000 data points. Given the same amount of data and the same grid granularity, the quad-tree data model results in a wide, shallow, and long table, while the regular-grid data model results in a narrow, deep, and short table. With a fixed cell size, as the amount of data increases, the quad-tree data model expands in the column dimension (i.e., the table becomes wider), and the regular-grid data model results in a deeper table as more data points get stacked on top of each other in the third dimension.

Table 3.1 reports the response time for range queries issued to the data set organized under the quad-tree and regular-grid data models. The label "≈0.1" in "QT" column refers to an experiment where the configured size of each cell is 0.1km, and the actual cell size is around 0.097km, with the quad-tree depth being 11. The grid is divided into $2^{10} * 2^{10}$ square cells. A range query is issued with the same reference location and a number of different radiuses, ranging from 0.01km to 12km. The *Target Objects* column reports the number of data points returned by that query. The column *FP* (i.e., *False Positive*) represents the percentage of data points returned to the client without actually belonging to the query return set. The higher this percentage, the more undesirable the situation since it implies that many irrelevant rows have been scanned, and have been transferred through the network to the client and have to be inspected and rejected by the client in the post-processing phase.

From Table 3.1 we can see that for the quad-tree data model, as the size of the cell decreases from 10km to 0.1km, the performance improves substantially for the small queries, while for the large queries, the result cannot be returned before the timeout. This is because, for smaller queries, only a small number of false positives rows is included in the data returned to the client. On the other hand, for the larger queries, many irrelevant rows have to be scanned, since the Z-value for cell ordering does not preserve a good locality (i.e., the neighboring relation) among subspaces. Even though the HBase *Coprocessor* framework parallelizes the query processing, at the core of the query processing lies a *Scan* operation; therefore, better pruning of unrelated data and fewer false positives remain the key of performance improvement. The same principle also applies to the regular-grid data model. The "RG" with the size of cell of 0.1km configuration outperforms the other two configurations. The reason is that the finer granularity of grid can enhance of the ability of pruning.

Finer-cell granularity results in improved performance for smaller queries in the quad-tree data model and for all queries in the regular-grid data model. However, there is a limit to how small the size of the cell can become. Smaller cell size implies that a greater number of rows must be scanned to respond to the query. If the number of rows exceeds the scan cache size, a higher number of *Scan* operations between server and client will be required, which will cause the performance to deteriorate. The size of the *scan cache* is constrained by the memory availability on both the client and server side. If a high-memory configuration is available, then increasing the cache size may result in some of the failed cases to work, but the performance trend remains fundamentally the same, because the number of irrelevant rows scanned will continue to increase. For the quad-tree data model, as the index is built and stored in the memory before the query-processing phase, smaller cell size and deeper quad-tree imply increased memory allocation.

From Table 3.1, we can observe that, for the regular-grid data model, best performance can be obtained with the cell size of 0.1km; while for quad-tree data model, the acceptable cell size is approximately 1km, with the quad-tree depth of eight.

The *HGrid* Data Model

In the *HGrid* data model, there are two variables that affect the *HGrid* index: the size of the tile (**T**) in the first tier and the size of the cell (**t**) within a tile. For our uniform-distribution data set of 100 million data points, if the individual cell is set as $1km^2$, the number of tiles can range from 1 (where there is only one tile and 10,000 number of cells), to 10,000 (where there is only one cell in each tile). Correspondingly, the number of rows are varying from 100 to 10,000, and the number of columns are from 1,000,000 to 10,000. Comparing these dimensions to the 10,000 rows and 10,000 columns in the quad-tree data model, and the 100 rows and 100 columns with stacks about 10,000 deep in the regular-grid data model, the *HGrid* table is neither as long as that of the quad-tree data model, nor as deep as that of the regular-grid data model.

Tables 3.1 and 3.2 report the query response times and false positives for various tile sizes, given a fixed cell size in the *HGrid* data model. Smaller-tile organizations exhibit better performance because they support better pruning of irrelevant data. However, smaller tiles also imply a bigger number of sub-queries for every query. The "HG:≈10:0.1" organization, referring to the configuration with a T≈10km quad-tree tile and a t=0.1km regular-grid cell, involves fewer sub-queries and more false positives and outperforms the HG:≈1:0.1 organization with more sub-queries and fewer false positives. This is an evidence of the trade-off between the number of false positives and the number of sub-queries.

The number of rows involved in the query is also an important factor that influences the performance, as evidenced by the fact that the performance of the HG:≈10:0.01 organization is worse than that of the HG:≈10:0.1 organization, as shown in Table 3.1. Thus, we conclude that the *HGrid* data model with tile size of T≈10km and cell size of t=0.1km approximates the best trade-off between the number of false positives and the number of sub-queries.

Table 3.4: Execution Time of kNN Query with Three Data Models (s)

	(a) Uniform Data				
k	**1**	**10**	**100**	**1,000**	**10,000**
QT:≈1	1.766	7.689	7.432	7.759	7.231
RG:0.1	0.307	0.270	0.302	0.596	1.295
HG:≈10:0.1	0.320	0.357	0.373	0.807	2.003
	(b) Skewed Data				
k	**1**	**10**	**100**	**1,000**	**10,000**
QT:≈1	1.737	1.885	1.914	1.900	4.583
RG:0.1	0.147	0.139	0.151	0.480	1.592
HG:≈10:0.1	0.325	0.314	0.358	0.879	3.307

3.4.3 Comparison of the Three Data Models

In this section, we compare the performance of the three data models, with range and kNN queries. We used the appropriate configuration obtained in Section 3.4.2 for each data model: QT:$\approx$1, RG:0.1, and HG:$\approx$10:0.1. In our experiments, we simply applied the same configuration into both uniform data and skewed data.

For uniform data, we randomly select a data point as the query input and a systematic variation of the radius. For skewed data, we selected three data points, each one with 20%, 50%, and 70% probability correspondingly in the Zipf distribution as the query input, and systematically varied the query radius from 0.01km to 4km.

Range Query

We evaluated the range-query performance under three data models with both uniform and Zipf distribution data. Table 3.3 shows the query response time of the three data models for various ranges when the system contains 100 million objects. As the radius increases, the size of irrelevant data vs. the return-set size ratio increases, and the running time also increases because more data points are retrieved. The regular-grid data model outperforms the others, because it supports better data locality and the percentage of irrelevant rows scanned is low. The *HGrid* data model performs much better than the quad-tree data model and slightly worse than the regular-grid data model. The same performance trends persist with both uniform and skewed data. In addition, in Table 3.3, we can also see that for skewed data, the queries with the radius of 8km, 12km, and 16km, cannot get result under these three data models. The reason is that the data points in the result are so large that the execution time exceeds the client socket timeout.

k-Nearest Neighbor Query

We also evaluated the performance for k Nearest Neighbor (kNN) queries using the same data set, under the three data models. Table 3.4 shows the response time (in seconds) for kNN queries, where k takes the values 1, 10, 100, 1,000, and 10,000. As the density-based range estimation method is employed [11], there is only one *Scan* operation in the query processing for uniform data, while for skewed data, more than one *Scan* iterations are invoked to retrieve the data. That is why the performance with skewed data under all data models is worse than that with the uniform data set. For both uniform and skewed data, the regular-grid data model performs best, followed by the *HGrid* data model with the quad-tree data model being the worst. The poor quad-tree locality contributes to the poor performance of the quad-tree data model, and also impacts the performance of *HGrid*, albeit less strongly. For skewed data, with too many false positives, the query with the data points having more than 70% probability cannot get the result below the timeout threshold under all data models when k equals to 10,000. To improve performance, a finer granularity is required to filter irrelevant data scanning.

In summary, the query performance of the *HGrid* data model is better than the quad-tree data model and worse than the regular-grid data model. The *HGrid* data model benefits from the good locality of the second-tier regular-grid index, but suffers from the poor locality of the Z-ordering linearization at the first tier. Better performance can potentially be obtained with alternative linearization techniques. In addition, the experiment result proved once again that a new configuration of these three data models for skewed data should be set.

3.4.4 Best Practices

Based on our experimental results, we have two types of guidelines for the organization of geospatial data in HBase. The first set guides the design of the data schema.

- The *row key* and *column name* should be short, since they are stored with every cell in the row.
- The *row key* should be designed to support pruning of unrelated data easily.
- The amount of data in one row should be kept relatively small. The cost (in time) of retrieving a row has n data increases more than twice with n (when n is large) [9].
- It is better to have one *column family*, only introducing more column families in the case where data access is usually column scoped [5].
- The number of columns should be limited. A number in the hundreds is likely to lead to good performance.
- When the third dimension is used for storing other information rather than time-to-live values, it is preferable to keep it shallow, and be limited to containing up to no more than hundreds of data points, as deep stacks lead to poor insertion performance.
- The Bloom Filter [5] should be configured as it can accelerate the performance by pruning the data from both row and column sides.
- Compression can improve the performance by reducing the amount of data transmission.

The second set of guidelines refers to the implementation of the query-processing mechanism.

- It is more efficient to *Get* one row with n data points than n rows with one data point each [9].
- *Scan* operations are preferable to *Get* operations for retrieving discontinuous keys, even though the *Scan* result is bound to also include data points that are not part of the response data set.
- It is advisable to narrow the range of queried columns with the *Filter* mechanism.
- The number of rows to be scanned for a query should not exceed the scan cache size, which depends on client and server memory. Otherwise, it is better to split the query into several sub-queries.
- When there are too many unrelated rows within the defined scan range, splitting one query into multiple sub-queries with multiple *Scan* operations is more efficient than one query with *Filter* mechanism to retrieve rows one by one.
- The *Scan* operation is preferable for small queries, while *Coprocessor* for large queries.

3.5 Conclusions and Future Work

In this chapter we proposed the *HGrid* data model for HBase, based on a hybrid index structure, combining a quad-tree and a regular grid as primary and secondary indices correspondingly. We comparatively evaluated the performance of the *HGrid* with uniform and skewed data, against the other two data models. Our results demonstrate that the *HGrid* organization scales well and supports efficient performance for range and k-nearest neighbor queries. Benefiting from the hierarchical index, the *HGrid* data model can be flexibly configured and extended. In the first tier, the quad-tree index can be replaced by the hash code of each sub-space or the point-based quad-tree index method is employed. In addition, the granularity in the second stage can be varied from sub-space to sub-space based on the various densities. Therefore, *HGrid* is more scalable and suitable for both homogeneously covered and discontinuous spaces.

In the future, we plan to experiment with alternative space-filling curves for the linearization of the quad-tree first-tier index, such as Hilbert curve [5], and to evaluate the model

[5]It can eliminate discontinuities and improves the overall locality

with real data.

Acknowledgment

The authors wish to thank Cybera Team for generous support. This work has been funded by the SAVI Strategic Network, NSERC, AITF and IBM.

Bibliography

[1] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[2] R. Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[3] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[4] R.A. Finkel and J.L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[5] Apache Software Foundation. Apache HBase Reference Guide, April 2012.

[6] C. Freksa and D.M. Mark. *Spatial Information Theory. Cognitive and Computational Foundations of Geographic Information Science: International Conference COSIT'99 Stade, Germany, August 25-29, 1999 Proceedings*, volume 1661. Springer, 1999.

[7] A. Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[8] Dan Han and Eleni Stroulia. A three-dimensional data model in hbase for large time-series dataset analysis. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2012 IEEE 6th International Workshop on the*, pages 47–56. IEEE, 2012.

[9] Ya-Ting Hsu, Yi-Chin Pan, Ling-Yin Wei, Wen-Chih Peng, and Wang-Chien Lee. Key formulation schemes for spatial index in cloud data managements. In *Mobile Data Management (MDM), 2012 IEEE 13th International Conference on*, pages 21–26. IEEE, 2012.

[10] J. Lawder and P. King. Using space-filling curves for multi-dimensional indexing. *Advances in Databases*, pages 20–35, 2000.

[11] D. Liu, E.P. Lim, and W.K. Ng. Efficient k nearest neighbor queries on remote spatial databases using range estimation. In *Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*, pages 121–130. IEEE, 2002.

[12] G.M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.

[13] Kannan Muthukkaruppan. HBase @ FacebookThe Technology Behind Messages, April 2012.

[14] S. Nishimura, S. Das, D. Agrawal, and A.E. Abbadi. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, volume 1, pages 7–16. IEEE, 2011.

[15] Wikipedia. Bloom Filter, May 2013.

Chapter 4

Migrating an Existing Geospatial Application to a Hierarchical Cloud

4.1 Introduction

Large organizations, including business enterprises and national governments for example, are typically organized as hierarchies or federations of departments. Frequently, the various enterprise departments of government jurisdictions use their own installations of the same software systems to manage their processes. These separate installations may sometimes be dictated by governance rules (for example, health-care delivery may be a provincial and not a federal service), but they are also motivated by several functional requirements. They support location-aware services, a functionality which becomes much simpler and efficient with location-specific deployments. Frequently they are accessible through mobile clients, which impose short network-latency requirements, a fact which also motivates local deployment. However, even as the locality enhances the user-facing services, there is potentially a lot of valuable insight and knowledge to be gained from the analysis of the complete data set. In the age of "Big Data Analytics", allowing this data to remain in geographical silos and missing out on the opportunities that their analysis as a whole may afford is not an option. This is why an architectural solution is needed, to enable (a) the systematic migration of the aggregate data of these systems to the cloud, and (b) their analysis from domain-specific and geospatial perspectives. This is exactly the objective of the work described in this chapter, using as an example a health-care application (named HCA-T), detailed in [28].

Health-care aides (HCAs) are the backbone of the home care system and provide a range of services to people who, for various reasons related to chronic conditions and ageing, are not able to take care of themselves independently. HCAs and the organizations that manage their interactions with clients face a number of challenges in their daily workflows and software systems can address some of them. The HCA-T system, designed after an analysis of the most important challenges in the current workflow, offers two major functionalities. On one hand, it provides a scheduling service that flexibly takes into account a configurable number of preferences, including HCA travel-time minimization, maximization of client-HCA affinity, schedule load balancing, etc. This service addresses "efficiency" concerns, enabling the best coverage of the clients' needs with the available HCAs. On the other hand, a mobile application enables HCAs to access and edit the client's care plan, as well as to provide textual information, images and videos to document the client's state. Typically, HCAs only have access to paper-based client records "in the office" and recording new information collected through the HCA visit takes time. As a result, available information is out of date, and it even takes time to realize that an appointment may have been missed. Real-time access to (and update of) the client record addresses many "quality of care" concerns in the current process. Equally importantly, a path-planning service instructs the HCAs about the location of their client and the possible path there and a corresponding location-tracking functionality on the mobile application may optionally be used to keep the office informed about the location of the HCAs at any point in time (motivated by the HCAs' safety concerns). The HCA-T system is structured in the typical three-layer architecture of most data-driven web applications. MySQL, a relational database system stores persistent data constitutes the base layer. Tomcat, an application server contains most of the application logic in the middle layer. In the middle-layer, there are also many functional components providing instant and location-aware services, such as instant meeting service and road and traffic condition service, which make the request of geographical deployment and time sensitive requirement. Finally, a HTTP server handles requests that come from application clients through the web is the top layer.

HCA-T was evaluated only through simulations; its actual deployment hinges upon changes to the legislation governing the certification and privileges of HCAs. However, if HCA-T was to be deployed in all Canada provinces, each one employing about 20,000 home-care workers every day, we estimate that up to 6GB of data would be collected, i.e., 180GB every month, 1TB per year. There is no dispute that this data size imposes too stringent scalability requirements onto the data-processing infrastructure. Moreover, if one decided to analyze the data as a whole, in order to extract descriptive statistics over the whole country or to compare provincial data against each other over interesting periods of time, the current application architecture, relying on a relational database becomes completely ineffective. Compared to relational database management systems (RDBMSs), NoSQL (Not Only SQL) databases, non-relational distributed database system, endowed with high availability, good elasticity and excellent scalability through their easy deployment on cloud computing platforms, become a more attractive data storage solution for these applications. Given the advantages of the cloud, it is attractive to migrate existing applications to this new platform. On the other hand, the concerns about the cost of migration and the potential risks regarding the impact of the move-to-the-cloud on the system's performance make the migration solution impractical. A new middle-of-the-road solution is necessary, with the actual application remaining essentially the same while enabling cloud-based analytics of the aggregate data.

Cloud computing has become increasingly prevalent, offering virtualized metered resources in a pay-as-you-go way over the Internet [15]. Amazon's Elastic Compute Cloud (Amazon EC2), as a successful commercialized cloud, is widely used. As new extensions of cloud technologies are emerging, the current cloud is called standard cloud computing model, or conventional cloud computing model. A relatively new idea is that of "cloud of clouds" (or multi-cloud) [2], which attempts to solve the issues existing in the conventional cloud (such as the interoperation problem), and offer the benefits of diverse geographical locations, better application resilience, and avoidance vendor lock-in [15]. Yet, no commercial offerings are available now. Another extension to the conventional cloud is called hierarchical (or multi-tier) cloud which is being explored by the Smart Applications on Virtual Infrastructure (SAVI) national research project in Canada [26]. It is designed to include multiple tiers, where the *smart edges* provide limited resources, are geographically near to the user [21] [26], offering fast on-demand deployment to applications and low latency to end-users, and the *core* provides powerful computation and storage resource.

Given the current technological background and the analytics requirements of the class of applications exemplified by HCA-T, we propose a federated architecture for geospatial applications on a hierarchical cloud [26] [21], addressing the above-mentioned requirements and challenges. In this federation-style architecture, the applications are fast and easily deployed on the *smart edges* with a pattern-based deployment service [19], geographically close to their end users. In addition, a new data access and aggregation system (DAAS) periodically and incrementally collects the data generated from these distinct application deployments and supports the implementation of ad-hoc queries in near real time for data analytics and business intelligence.

During this migration, the most important and challenging problem we met and dealt with is to transform the data schemas in RDBMSs to HBase. In RDBMSs, the data schemas are normalized to save space and typically end up being organized in a star shape, with a main table connected to many attached tables. As a result, the main table is often joined with other attached tables to answer the various queries of interest to the client. Even though HBase stores the data into a "BigTable" structure, it is conceptually very different from the relational data model in RDBMSs. The data in a HBase table is partitioned by row key and there is no secondary index support, which makes join operations computationally expensive. Whenever there is a where-condition query over a column attribute, HBase has to scan over the data associated with a given range of row keys and compare the attribute values of the returned rows one by one against the condition to produce the result. In RDBMSs, normalization is advised while in HBase a flat data schema is preferred. In RDBMSs, the data schema is created with the table generation, while in HBase, data schema is not completely defined in the beginning of the table generation, but through the data-insertion procedure. Making this problem even more challenging is that, to date, there is no method for guiding developers in systematically designing an appropriate data schema in HBase for a given application. In this work, we proposed a systematic method about how to transform the entity-relationship data model [4] in a RDBMS to a HBase repository, for spatial and temporal data-intensive applications.

More specifically, the contributions of this study can be summarized as follows.

- A novel federation-style architecture for web-based location-aware legacy applications on hierarchical cloud is proposed. By taking advantage of the multi-tier cloud infrastructure, typical data-driven systems are deployed geographically to minimize wide-area network latency for application end users; at the same time, a data access and aggregation system migrates the data of the different installations via intra-cloud network to provide the integrated data analysis.

- A data access and aggregation system is provided for integration analysis of dispersed data. It is loosely coupled with the original system and can easily be adopted to other systems with some application-specific programming.

- A method is provided to guide the mapping of the relational data schema, originally modelled according to the entity-relationship paradigm in the RDBMS, to HBase. In this method, a set of guidelines is formulated for developers to consider and a five-step transition process is proposed for them to follow during migration of their application data.

The rest of the chapter is organized as follows. Section 4.2 reviews background and related work on data model transformation from RDBMSs to HBase and data integration. Section 4.3 demonstrates the proposed architecture of cloud-enabled three-tier web-based systems on a hierarchical cloud. In Section 4.4, we explicates how the data migration and integration process are designed and implemented in the system. By taking HCA-T application as a case study, we describe how HCA-T2 (the cloud-enhanced evolution version) is built in Section 4.5 and evaluate its performance in Section 4.6. Finally, we discuss some potential issues of the architecture in Section 4.7 and conclude with a statement of our contributions and plans for future work in Section 4.8.

4.2 Background and Related Work

4.2.1 From RDBMSs to HBase

There has already been some research on how to transform a relational data schema in RDBMS to HBase. In [23], the authors propose three guidelines for the transition, which aim for denormalizing the original relations, including grouping the correlated data in a column family, adding the foreign key reference to the relationship and merging the attached data tables to reduce foreign keys. In [25], the authors discusses a case study whose objective was to map the Twitter data schema from MySQL to Cassandra, where the schema de-normalization was carried out in the first step of transition. In [16], the authors report on a relational-data migration exercise to Hive. They denormalized the star schema into universal relation first, then followed data collation method to reduce data storage requirements. The aforementioned work has already demonstrated the importance of schema de normalization, which is also a critical step in our transition process. Besides, inspired by the work presented in [16] where they transformed the data schema from RDBMSs to Hive based on universal relation model, the transition method we presented in this work is based on entity-relationship data model in RDBMSs. Furthermore, in our method, we differentiate the data based on the velocity of increase and the types, and suggest applying different data models according to the data types.

At the same time, there has been some research on how to design data schemas in HBase. S. Nishimura et. al[24] built a multi-dimensional index layer on top of HBase to perform spatial queries. Ya-Ting Hsu et. al [20] presented a novel key-formulation schema, based on R+-tree for spatial index in HBase. In our own previous work[18], we proposed the HGrid data model for large scale geospatial datasets, based on a previous work of a three-dimensional data model for time-series datasets [17]. To our best knowledge, there is still not a systematic way for an application developers to follow during the data schemas migration, which is the main problem we are addressing in this work.

4.2.2 Data Integration

The problem of data integration is very broad, and, in general, there are three ways to address it: federated databases (FDBS), data warehousse, and data-integration systems. Each of these approaches has its pros and cons. As discussed in [1], FDBS is not considered to be a full-blown data-integration system in that it only supports query execution over defined relations imported from existing RDBMSs. In comparison, data warehouses store the data into a new database where analysis services can be implemented, reports generated, etc. It makes the relation redefinition possible, but gives rise to other problems, such as data synchronization between the data warehouse and the underlying data sources. Beyond the functions provided by FDBS and data warehouse, data-integration systems offer a level of data integration by performing on-the-fly at the attribute level. It does not require data replication, but it is much more complicated in nature.

In addition to the substantial amount of work that has been dedicated to these three approaches, recently, two experiences of data warehouses on NoSQL databases have also been reported. Facebook introduced their data warehousing and analytics infrastructure which is built on Hive [29]. Twitter published their unified logging infrastructure for data analytics [22] built on top of Scribe. The two infrastructures rely on different technologies to migrate the data, but they all aggregate the data together and store large scale dataset into HDFS. In this work, we also implemented *DAAS* as a data warehouse and developed it on top of HBase.

4.3 Architecture of Cloud-Enhanced Web Applications

In this section, we describe in detail the "cloud-enhanced architecture" we have developed for web-based location-aware systems in a hierarchical cloud. We first describe how the components are deployed in the cloud infrastructure, then we discuss the run-time data flow among the system components.

4.3.1 The Federated Architecture

Figure 4.1 illustrates how applications are deployed in *Federation Deployment* mode on the SAVI cloud. As shown in the Figure 4.1, instances of the original applications are

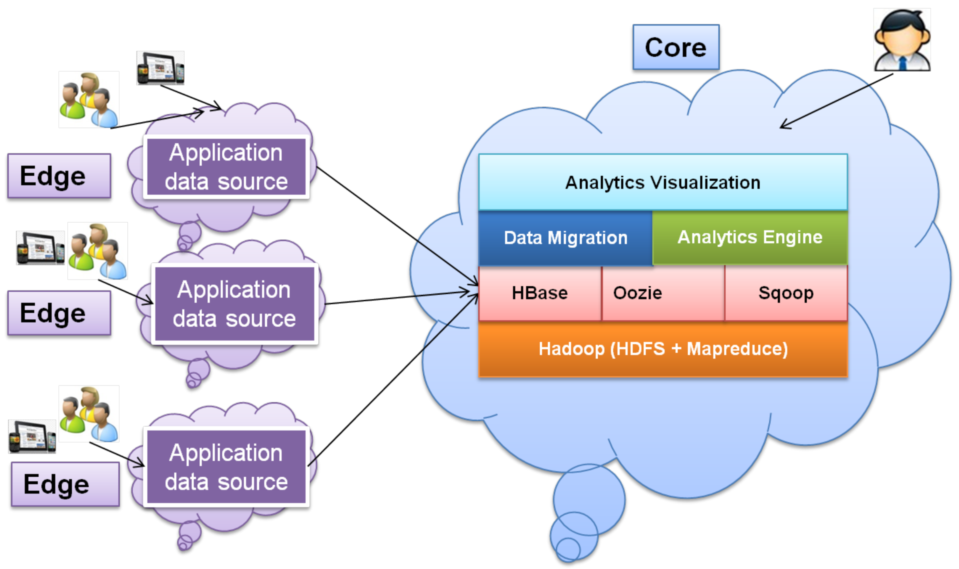


Figure 4.1: Deployment Mode on SAVI cloud

deployed on the *smart edges* to meet the low-latency user requirement, and *DAAS*, as a data warehouse, is deployed on the *core* to incrementally and periodically aggregate the data from all the *smart edge*s for data analysis. This design is based on the characteristics of multiple tiers in a hierarchical cloud infrastructure. The SAVI cloud offers two different tiers: nodes of the *smart edge* tier are geographically distributed and typically have fewer resources, while the *core* node has substantial computation and storage resources and is located at the conceptual (and possibly geographical) center of the cloud. Thus, applications running on the *smart edge* exhibit low latency by taking advantage of being "local" to their clients, while the *core* is an ideal place to centralize and manage the various application installations on the *smart edges* and handle large amounts of data storage and intensive computation. In addition, the intra-network connection between the *smart edges* and the *core* enables the inter-access between tiers, avoiding the issues typically associated with wide-area networks among different clouds. Thus, the performance of data migration from the *smart edges* to the *core* should be better than that between two geographically distributed data centres.

We have also considered two alternative data-storage configurations. Rather than deploying a data repository on each *smart edge*, a single repository might be deployed on the *core*, to be shared by all the application installations running on the *smart edges*. In this design, a single multi-tenant database stores all the data. Even though this approach would make the integrated analysis much easier, thousands of millions of data access workloads between the *smart edges* and the *core* will bring in heavy loads to the intra-network, resulting in load-balancing issues among the applications co-existing in the cloud. We could potentially distinguish the application components either as data-intensive or as computation-intensive, and deploy the former on the *core* and the latter on the *smart edges*. In this design, the data-access workloads would be largely reduced, but the question of how to partition the two breeds of components and how to design the protocols among them are quite challenging. Furthermore, this approach necessitates the re-architecting of the original applications. Given our requirement of minimizing the changes to the original application, we decided to follow the first approach which does not impact the original application.

On the core, *DAAS* is deployed to aggregate the various data sources, and to support efficient query processing and complex analytics. By aggregating data from disparate sources and acting as a data warehouse, it provides a uniform interface for data-driven research and also tools for developers to customize for their own applications. The data-warehouse design, as opposed to a federated-database or a data-integration design, affords the additional functionality of tagging and modifying the collected data, outside its use by the original application, and appropriately sharing (views of) the data with the public.

In this architecture, several open-source projects of the Apache Software Foundation are adopted. HDFS [7], as the distributed file system, provides excellent scalability and fault-tolerance mechanisms. The Hadoop map-reduce framework [7] enables the parallel processing of the data-migration jobs. HBase [6], as a particular NoSQL database offering, is a distributed, scalable, big-data store. It relies (a) on HDFS, for its distributed and replicated storage, and (b) on coprocessors, for efficient parallel query processing. Sqoop [14] is a tool for efficiently transferring bulk data between Hadoop and structured data stores, such as relational databases; in our architecture, Sqoop is used to import the data from RDBMSs to HBase. Oozie [12] is a workflow scheduler to manage Hadoop jobs, which include the jobs out of the box (such as Java map-reduce, Streaming map-reduce, Pig, Hive, Sqoop and Distcp), and system-specific jobs (such as Java programs and shell scripts). Three types of jobs are supported in Oozie: workflow jobs, which are directed acyclical graphs (DAGs) of actions; coordinator jobs, which are designed for recurrent workflow jobs triggered by time (frequency) and data availability; and bundle jobs, aiming for easily managing batches of coordinator jobs. In our architecture, the coordinator job is used to schedule the data-migration workflows, which consist of a Sqoop action and a shell action.

The *DataMigration* component (in Figure 4.1) is responsible for all tasks required in the migration process, including creating HBase tables, discovering data sources, and configuring Oozie jobs and submitting them to Oozie to start the process. The process can be configured either one Oozie coordinator job which serves as the root and coordinates all the child workflow jobs, or many coordinator jobs which can be bundled together and started with a bundle job. In our case study, two coordinator jobs are used to concert data migration process. The *AnalyticsEngine* provides *RESTful* services for users to query the centralized data warehouse in JSON. It requires the query parameters in the form of a JSON object, to be parsed by the *REST* service. It sends the queries to the corresponding coprocessors preloaded on the HBase region servers for processing, and aggregates the returned result in the end. In terms of *AnalyticsVisualization*, we opted for a loosely-coupled approach. It is not our intention to design and implement a universal console covering the different applications, rather, we leave it to the application developers or/and the end users to decide how to visualize the data they want. With the JSON format returned from *AnalyticsEngine*, the console can be flexibly developed to match the application domain and query needs.

With this deployment mode, the changes are transparent for end users of the original application, and a uniform platform is available for administrators and researchers to explore the data.

4.3.2 The Data Flow

Figure 4.2 illustrates how the data flows from the applications on the *smart edges* to the *DAAS* on the *core*. Note that as some data may be sensitive, it is the data owner who determines the access-control policies that are to be applied to their exposed data. The access-control policies should be implemented and enforced by the original application.

Incrementally and periodically, the data is migrated from the legacy databases on the *smart edges* to the HBase cluster through an Oozie *coordinator job*. The coordinator job is initialized with the data-sources information and all workflow configurations. It is then triggered based on the customized frequency or the input events in each interval. The frequency can be set with the unit of minutes, hours, days and months. The input events specify the input conditions that are required in order to execute a coordinator action. The coordinator job includes many workflow jobs, and each of them is associated with a data source on some *smart edge*. In each data-source workflow, many sub-workflows are responsible for the various tables involved. Each workflow responsible for a particular table starts with a Sqoop action, which is to read the data from disparate RDBMS via JDBC and then write them into the HBase. Next, the workflow continues with a shell action, which updates the start index in each iteration to implement the incremental migration. Changes in data sources in the

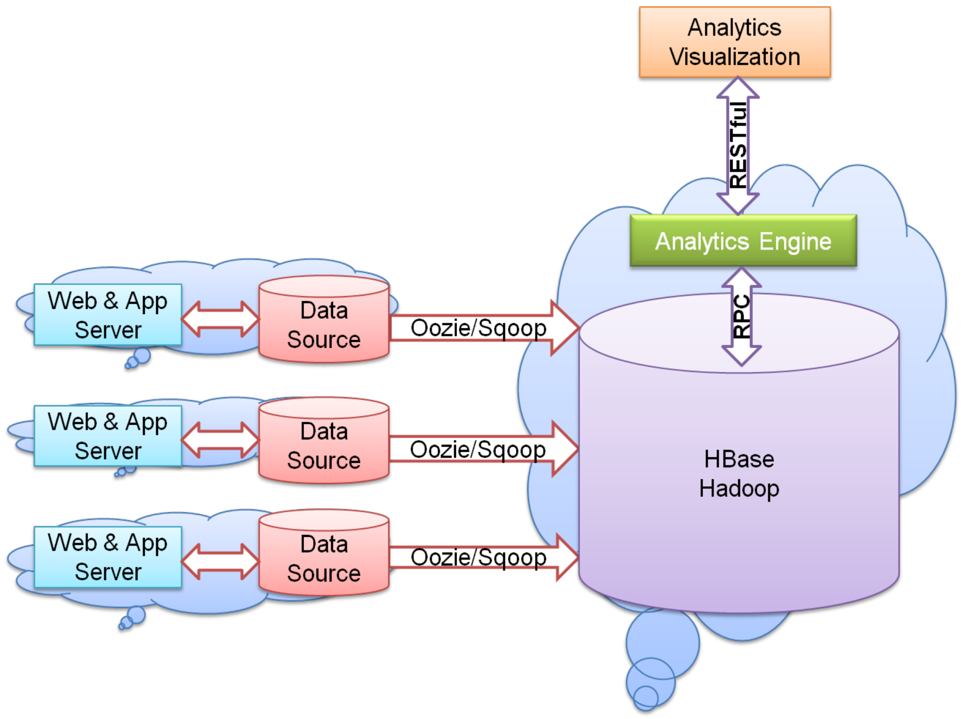


Figure 4.2: Data Flow Architecture

smart edges will be continuously fed into HBase in the *core*.

Finally, the integrated data can be accessed and queried through the *RESTful* service, which is implemented in the *AnalyticsEngine*.

4.4 Data Migration and Integration

The process of data migration from the RDBMS of the original application to HBase gives rise to two conceptual problems:

1. How should the data schema be transformed from the RDBMS to HBase?

2. What exactly is involved in the aggregation-and-migration process?

4.4.1 Data Schemas Transition from RDBMS to HBase

Many models have been proposed for how to organize data in RDBMSs [32], such as object-oriented modelling technology [3], entity-relationship model [4], the universal relation model [30], etc. The most prevalent among them is the entity-relationship model, and more applications are being designed following its guidelines. Hence, we focus on the entity-relationship [4] data model, and propose the following principles and steps for transforming the relational data from RDBMSs into HBase. We have also applied these rules to a case study described in Section 4.5.

1. Classify relations into active and inactive The successful transition relies on a sound understanding on the dataset and how the data is generated in the application. By looking into the data-generation behaviour, one can observe whether the number of a particular entity (or relationship) increases much over time. In some cases, the data is created (inserted into the table) once and seldom updated, for example, the profile information about a person. In other cases, new data instances are regularly updated with instances inserted every minute, hour, day, or week. The former type of entities/relationships are *inactive* while the latter are *active*.

In RDBMSs, it is not advisable to store active and inactive data together. This is avoided by using foreign keys in the active data tables and 'join' operations to reduce data redundancy and save space. However, as HBase does not support 'joins', this is a rather expensive operation and designers tend to obtain good performance by taking advantage of easily available cheap disk space. Even so, the resulting amount of repetitive inactive data requires larger buffers, for these queries with large response sets. Without an efficient filter mechanism, it is very easy to have the query performance deteriorate fast. Furthermore, it is practically unavoidable that the inactive data will eventually change in the future. With a huge number of replications, the update management would become a potential problem. Therefore, we suggest that they be stored into separate tables.

2. De-normalize Relations The ER diagramming notation is a powerful tool for designing the tables of an application in RDBMSs. In an entity-relationship model, both entities and relationships are described by attributes [4]. Relationship tables are designed to normalize the data by using foreign keys. Any queries on these tables should employ 'join' operations to obtain the complete information about the entities involved. To obtain the entity information, regular entity relations are often joined with weak entity relations and entity relations are usually joined with both regular relationship relations and weak relationship relations. In order to avoid 'join' operations, data model de-normalization is used to merge and reduce the number of relations. To de-normalize the data models, the following guidelines may be followed.

a. For inactive data: Regular entity relations should be kept "as is". Weak entity relations should be merged into the main entity. Weak relationship relations should also be merged into the main entity. Regular relationship relations should be merged into the entity, which can be queried more easily.

b. For active data: Regular entity relations should be kept "as is". Entities referenced by weak entity relations, if they are active and do not depend on other entities or relationships, should be merged into these relations; otherwise, they should be kept "as is". Entities connected to a regular relationship relation should be merged to this relationship, if they are also active relations and do not depend on other entities or relationships. Entities linked to a weak relationship relation, if they are also active relations and they do not depend on other entities or relationships, should be merged to this relationship.

3. Apply Appropriate Data models We believe that different data models are suitable for different types of datasets. Therefore, it is important to classify the data into different types and apply different data organizations for each data type. For each data model design, two key points should be considered: (a) how to organize and store the data, in order to be able to obtain more information with one scan operation, and (b) how to prune the query response set, to improve caching efficiency. Clearly, these two qualities need to be balanced against each other, and the trade-off depends on the data types and the query patterns involved. Here, we only consider three kinds of data sets as examples, since they commonly exist in geospatial applications.

a. Time-series Datasets

The key point in using time-series data model proposed in [17] is to group data within a period, to speed up the period queries. In this data model, either the *version*, or the *column* dimension, or both can be used to stack the values within a group. The choice depends on the number of changing attributes of the represented object. If the number of attributes is large, it is recommended to store the attributes as a dimension, leaving the other dimension to store the offset of the period. If the number of attributes is small, usually lower than 5, it is recommended to store all attributes in a JSON object, and to use both *column* and *version* dimensions for storing the period of time and the offset of the period. The question "How long of a time period should be stored in version dimension" highly depends on the application characteristics and queries. Yet, based on the HBase storage characteristics, as the performance is impacted by the number of versions and columns in each row, the number of versions should be limited to the hundreds [18].

b. Spatial Datasets

The term "spatial datasets" refers to data associated with location information, represented as a point with latitude and longitude coordinates. When such datasets are involved, an indexing table should be added in HBase, modelled after the HGrid data model proposed in [18]. The HGrid index structure involves two levels of detail; the first is a coarse-grained tile and can potentially also be replaced with a conceptually meaningful geographical area, and the second is a fine-grained cell which is determined by the spatial query requirements. It should be noted again here that the HGrid data model only focuses on the static location, such as the bicycle station location and the home address. When mobile entities are involved, with changing locations, some modifications should be applied, for example, storing timestamps into the third dimension rather than object attributes.

c. Descriptive Dataset

A common practice in RDBMSs is to maintain an incremental counter as the unique key of entities. This practice is not appropriate for HBase however. The row keys in HBase are used for deciding which data to scan and how to prune it. Artificially constructed row keys cannot play this role; the row keys have to be designed based on actual data attributes, namely these attributes that are likely to be used as part of the queries issued to the repository. As we discussed before, how much unrelated data is included in a query response set substantially impacts the query performance. Therefore, narrowing down the queried data is very important in data modelling. This explains the importance of designing the row keys and column names, and the necessity of row and column filters in HBase. In general, the row key in HBase should be defined and composited by the attributes which are most queried. The answer to the question on "how many attributes should be selected to be a part of row key" highly relies on the query pattern and the effectiveness of the encoding techniques applied on the row key. The more information contained in the row key, the more types of queries can be handled, and the longer the row key will be. In principle, the row key should not be too long, therefore row-key compression techniques and encoding are required.

4. Adjust and Optimize Last but not least, adjustments may have to be made to the HBase schema, based on the anticipated query patterns and HBase storage characteristics. These adjustments include the following.

- An effort should be made to shorten column names, column family names, and row keys as much as possible. Encoding techniques should also be applied for the same purpose, such as string compression, or an additional mapping file.
- The schema should be revisited to examine whether it effectively supports the query

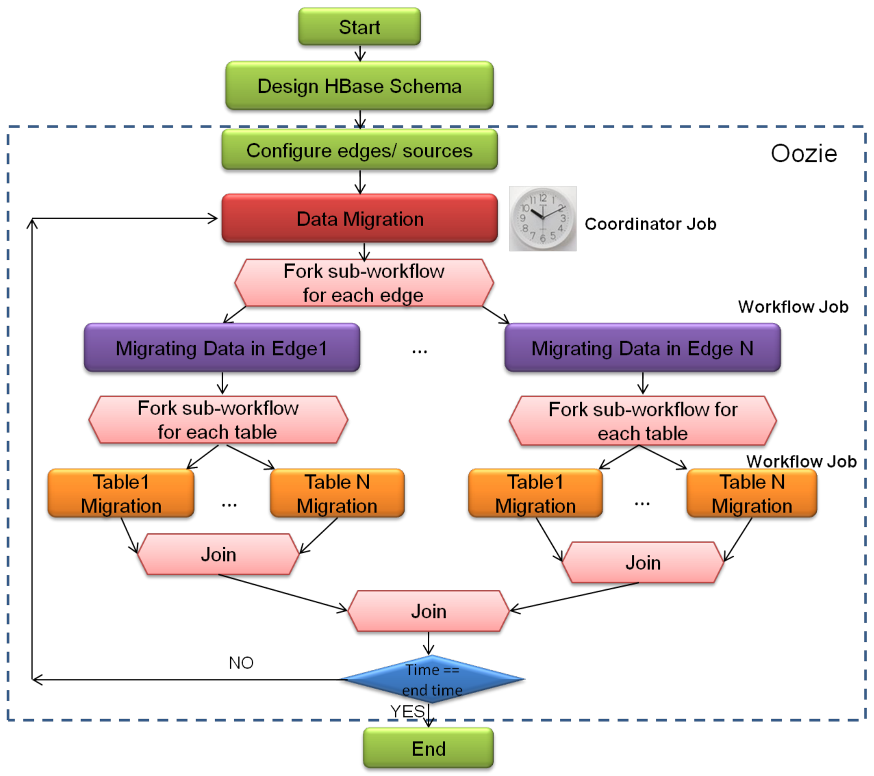


Figure 4.3: Data Migration Work Flow

patterns. In particular, one should examine whether there is any additional information to be added into the tables. For example, if the data is from different sources, the unique indicator to differentiate different sources should be added into the row key as prefix.

4.4.2 The Data-Migration Process

The migration process is implemented through a workflow, as follows. First, the HBase tables are created with a predefined configuration, including the column family name, number of version, the compression technique, the size of block and the filter mechanism. Next, the data sources are identified either with a static configuration file or discovered by a web service. Finally, the data is migrated incrementally and periodically within a conditional loop.

Figure 4.3 illustrates the migration workflow we designed for our case study. The data-aggregation process is implemented with an Oozie coordinator job. The job is triggered periodically based on the configured frequency. This coordinator job includes many workflow jobs, each corresponding to a particular data source on a *smart edge*. In each workflow job, there are many sub-workflow jobs which are responsible for the various tables in the data source. These sub-workflow jobs are also workflow jobs, constructed by a Sqoop action to move the data from the RDBMS to HBase, and a shell action that keeps track of the next starting migration index and the size of the migrated data. The Sqoop jobs read the bulk of data from the relational database via JDBC, and then insert them into the HBase in a form of a customized data model. This migration process is done with map-reduce jobs. As the data moves from the *smart edges* to the *core*, the network might become a bottleneck; this is why the data should be compressed for this step. The shell action updates the where-condition of the "query" in a Sqoop job by string replacement to tell which chunk of data will be migrated in the next iteration. When the time for the next step arrives or the input data is ready, the coordinator job is invoked, and all sub-workflow jobs for each data source start to run.

Unlike with RDBMSs, table-schema creation in HBase cannot be done at the table-creation time; it is implicitly defined through the process of data insertion. As Sqoop only supports simple insertion, with a direct mapping from columns to columns and rows to rows, we extended the Sqoop insertion interface, with the *TimeSeriesPutformat* class to support the time-series data model [17] and the *GeospatialPutformat* class for the HGrid data model [18] [1].

In addition, to support the flexible table schema configuration, we redefined the *hbase-row-key* and *column-family* parameters in Sqoop. The *hbase-row-key* was extended to be a JSON object (shown below) instead of a simple string.

```
{
   ``region":``bc",
   ``combined'':``pid,hid",
   ``timestamp":{
      ``field":``sweek",
      ``format":``yyyy-MM-dd",
      ``version":{
         ``field":"block",
         ``interval'':``1'',
         ``unit":``min''}
   }
}
```

[1]These two classes are implemented by extending the existing Sqoop interface.

The *region* element indicates the data-source name and it should be part of the row key. The *combined* element denotes the values of the columns in the original table that will be part of the row key, in the order they will be concatenated in the composite row key. The *timestamp* element, defined as a nested JSON object, contains multiple keys. The *filed* element indicates the column in the original table, based on which the timestamp will be calculated. The *format* element defines the time-value format of the *field*, which tells the transformation process how to process the timestamp. The *version* element advises the transformation process about which column in the original table should be mapped to the *version* dimension in HBase.

If there is a spatial indexing requirement, the *spatial* JSON object should be nested in the object as follows. The *fileds* element indicates the column name representing latitude and longitude. The *schema* element is required to create the spatial index. The *space* element defines the whole area of interest to the application, as a rectangle defined by its top left point, width, and height. The *indexing* element indicates which indexing method should be applied: quadtree, regular grid or HGrid. The *encoding* element defines how the value of the index should be encoded. The *tile* and *subspace* elements are specific to the two-level index structure, and indicate how the space is split in the first and second levels.

```
``spatial'':{
   ``fields'':``lattd,longtd'',
      ``schema'':{
         ``space'':``-138.95,41.77,60,19'',
         ``indexing'':``2'',
         ``encoding'':``1'',
         ``tile'':``-1'',
         ``subspace'':``0.001''}
   }
```

The column family is also extended to support many column families and renaming through a configured JSON object. Taking the following JSON object as an example, *family* sets the column family name in the HBase table. The *columns* element defines the column names, where *field* indicates which column's values in the relational data schema will become columns in HBase, and the column name in HBase will be prefixed by the strings defined in *prefix*.

```
{
   "family":"d",
    "columns'':  [{
        "field":"wday",
        "prefix":"w"}]
}
```

With the configuration and the extended data model support, the Sqoop job flexibly maps each row and column data in RDBMSs to the corresponding column and row in HBase.

Finally, three other Sqoop parameters should be discussed. The parameter *query* supporting customized SQL queries on RDBMS data sources, provides the flexibility of migrating the necessary data. Second, the number of mapper jobs, defined with the parameter *m*, should also be configured based on the amount of the datasets involved in the migration. It helps to speed up the migration process via parallel map jobs. Another important parameter is compression. The compressed data can reduce the traffic between *core* and *smart edges*, and mitigate the bottleneck incurred by the network because of bulk data transmission.

At run time, data is being generated continuously by the various application installations; this data needs to be migrated incrementally and periodically. Therefore, controlling the incremental process is another problem addressed in this architecture. In principle, there are two alternative solutions: one is to migrate a fixed data chunk each time, with a counter for keeping the next starting index; the other is to migrate newly inserted data by maintaining an index to the last migrated data item. Both approaches have their pros and cons. In the first case, a control mechanism is required to recognize when there is enough new data for a complete data chunk to be moved next. In the second case, the transferred amount of data varies and the frequency of the migration events fluctuates, but new data is migrated in a timely fashion. In our case study, we chose the first approach for its simplicity, by calculating the next migration index with a start index and a configurable chunk size.

4.5 Case Study: Migrating HCA-T to SAVI

The HCA-T (Home-Care Aides - Technology) application, described in Section 4.1, is a data-driven web application that includes a scheduling service to assign home-care aides to visit home-care clients and carry out their care plans. Through a mobile client, it enables HCAs to share audio/images/video notes with the back end. A navigation component advises HCAs about the location of their clients and how to get there. At real time, it also informs them about their own location and that of the other HCAs in their organization. This application is an example of the class of applications that most benefit from a two-tier cloud: different primary-care networks (PCNs) would need access to their information (at their *smart edges*) but only infrequently would they require access to the data from another PCN. Accountants and epidemiologists would need different types of access to the whole data (at the *core*).

Let us now discuss the relational data model of the HCA-T application. *Patient*, *HCA* and *Service* information should be stored into database in the very beginning. With the system running, this data is updated and appended gradually. The service requirements of each patient are input into the system by nurses. Based on that, the schedule and the appointment data are fed by the scheduling system every month. These appointments are followed by HCAs to visit patients. During the appointments, HCAs update the service and upload supplemental information via their mobile devices. The updated information is inserted into *ServiceRecord* and *MediaMessage* tables.

The HCA-T application was developed independently of this work and we simulated its use in order to evaluate the performance of the migrated variant. There are about 100,000 clients (patients) and 22,000 HCAs in a single province (Alberta) in Canada. Each day, typically four visits must be scheduled per client within morning, noon, afternoon and evening time slots, four-hours long each. During each scheduled visit, there are typically around 10 services, required by the client's care plan, and, therefore, up to 10 pieces of information uploaded to the system, including pictures, notes, video and audio. Based on these estimates, around 20 GB record data (excluding the media files themselves) will be generated each month for one province. With ten provinces in Canada in one month, 200GB data will be generated, which will be more than 2 TB data in a year.

In this case study, we deployed the HCA-T application with the cloud-enhanced architecture proposed in Section 4.3 on the SAVI cloud. We deployed the original application on two *smart edges*, the *ON Edge* and *BC Edge*, and the DAAS in the *core*. We will refer to the new cloud-enhanced HCA-T system as *HCA-T2*. In HCA-T2, the users in Ontario and British Columbia province are served by different HCA-T servers based on their locations. The HCA-T Server in each *smart edge* stores and manages the data in MySQL database. And the daily data is replicated and migrated to DAAS at mid-night periodically. With this cloud-enhanced architecture, the application developers do not require any additional programming, and end users are unaware of the deployment changes.

4.5.1 Sample Queries

We considered three types of queries for *HCA-T2*: a window query, a range query, and a time-series statistical query. These queries are inspired by research questions of interest to health-care providers. Administrators want to know the distribution of the patients in each region, which can be found with a query such as: "Which area (East/West/North/South) has most/least patients per region in 2012/2011" (i.e., a window query). Responding to the need of grouping patients with the same conditions together for group exercise, we designed a query for finding the neighbours of a given client, based on his/her home location and a given distance (i.e., a range query). Finally, descriptive-statistics analyses are typically of interest as key performance indicators for managers, executives and administrators. Therefore, we provide the queries like " Get the total number of appointments/services/uploaded images per week/month in British Columbia/Ontario in 2012/2011". With these queries, the researchers and administrators can perform their statistical analysis easily and effectively.

We decided to use the *Coprocessor* framework for the query-processing implementation, because *Coprocessor* results in better performance than *map-reduce* for statistical analyses. With the *Coprocessor* framework, for each query, there should be a *callable* + *callback* pair. The callable object is used to envelope method invocations to the server, using the coprocessor RPC framework. The callback object is invoked when results for the above call become available from the coprocessor. When it receives a query, the called function invokes a RPC call to the HBase region servers. The RPC calls are received by the HBase regions and executed as batch processes. The regions who should handle the RPC calls are determined by the match of the query range against the ranges for which each region is responsible. After the coprocessor has completed the task, it returns the results to the client. The callback object aggregates the results from the various region coprocessors. It should be noted that the calls from client side are executed on the corresponding regions in parallel. However, there is a shortcoming of this implementation that is HBase server should be restarted every time for loading user-level *Coprocessor*.

4.5.2 Transforming HCA-T Data Schema

This section describes how we transformed the table schemas from MySQL to HBase for the HCA-T application following the steps and rules described in section 4.4.1. Specifically, there are five steps to make the transition:

1. restore the ER-diagram;

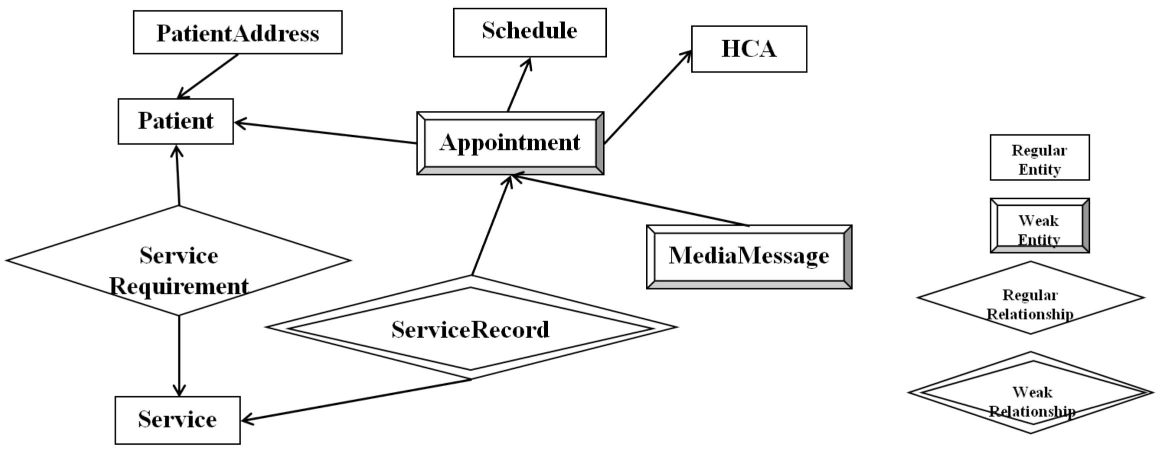


Figure 4.4: HCA-T Entity-Relationship Diagram

2. classify the active and inactive relations;
3. de-normalize relations in RDBMS;
4. apply the existing data models to different datasets; and
5. adjust and optimize the schemas.

In order to restore the entity-relationship model, we must have a complete understanding of how the HCA-T application works to understand the entities and relationships. As per the description of the application in Section 4.5, we drew up the ER-diagram for HCA-T application in Figure 4.4.

Next, we identified and categorized the entities and relationships into active and inactive relations. *HCA*, (which describes the information of HCAs), *Patient* (which describes Patient information), *PatientAddress* (which describes the location information), and *Service* (which describes Service provided in the home-care system), were categorized as inactive entity relations. *ServiceRequirement* (which represents the services required by patients) was categorized as an inactive relationship relation. The active entity relations included *Schedule* (which describes the weekly schedule), *Appointment* (which includes all appointments of *Patient* and *HCAs*), *ServiceRecord* (which stores the service update records generated during the appointment) and *MediaMessage* (which keeps the media data uploaded in a appointment, such as pictures, text, audio, video and notes).

To de-normalize the relations, we followed the rules described in Section 4.4.1. The inactive relations, *Patient*, *HCA*, and *Service*, as the regular entity relation, remain as they are. *PatientAddress*, as the weak entity relation, was merged into *Patient*. *ServiceRequirement* is a regular relationship relation, and was merged into *Patient* based on the queries. The active relation *Appointment*, which is represented by the entities of *HCA*, *Patient* and *Schedule*, is a weak entity. In this entity, *Schedule* is an active data independent of others, hence, we merged *Schedule* into *Appointment*. *ServiceRecord*, which is represented by appointment id, service id, and timestamp, is a weak relationship. *Service* is an inactive data type, and *Appointment* remains as is, because *MediaMessage* depends on it. *MediaMessage*, which is represented by appointment id, and timestamps, is a weak entity. As *ServiceRecord* depends on *Appointment*, so *Appointment* should stay as it is. After this step, the relations become *Patient*, *HCA*, *Service*, *Appointment*, *ServiceRecord* and *MediaMessage*.

The next step is to apply the existing data model based on the data type. The *Appointment*, *ServiceRecord* and *MediaMessage Appointment*increase rapidly over time; they are effectively time-series data. For *Appointment* data, the period of time is week (the coarse-granularity time), and block (the fine-granularity time), and only one attribute exists in an appointment object, so we decided to model the column to store the index of the week day, the version stores the index of block in each day, and the Monday date of each week becomes a part of row key. For the *ServiceRecord* relation, the time period is defined in terms of a block (the coarse-grained time) and the offset of period details to minute (the fine-grained time), along with the index of required services as the attributes for each record. Therefore, we designed the column as the services, and the version as the number of minute in a block. Similarly for *MediaMessage* data, the version as the offset of minutes in a block, while the column represents the media types. The cell stores the attributes of each media object into a JSON object. *Patient*, *HCA*, and *Service* are descriptive and inactive data. *Patient* has the location information; therefore, it should be organized under a spatial index. To apply HGrid data model into the location on *Patient*, the row key is the composition of quad-tree index id and regular grid row index, the column is the composition of column id and patient id. The shortcoming here is that as the patient information is stored into cells, the query about patients based on other attributes, such as first name and last name, can only be processed by scanning all the data in the table. Taking this into account, we created a new indexing table, by keeping the *Patient* relation as it is. For *Patient* and *HCA*, as the descriptive entity relations, we designed the row key as the composite key of the first three letters of first name and the first letter of last name. For *Service*, we just kept the incremental counter as the row key, as it is referenced by *ServiceRecord* as columns.

The last step is the final adjustment and optimization. We added the *smart edges* id into row keys in each table, as it can uniquely represent a data source. As the *smart edge* id can represent a province which represents a conceptually meaningful geographical area, we adjusted the HGrid data model by employing edge id as the first index rather than quadtree index. We shortened and redefined all column names into two letters. Finally, we got our data schemas in HBase as shown in table 4.1.

Table 4.1: HCA-T Transformed Data Schemas in HBase

Table	**Data Schema**				
	Row Key	**CF**	**Columns**	**Version**	**Cell**
Patient	edgeid-encoded(firstname-lastname)	d	encoded attribute names	timestamp	attribute value
HCA	edgeid-encoded(firstname-lastname)	d	encoded attribute names	timestamp	attribute value
Service	edgeid-incremental_counter	d	encoded attribute names	timestamp	attribute value
Appointment	edgeid-encoded(period)-pid-hid	d	week day [0,1..6]	block [0,1,2,3]	JSON object
Patient-Spatial	edgeid-hgrid_row_index	d	HGrid_column_index	attribute index	attribute value
Record	edgeid-encoded(period)-pid-hid	d	service index [s1,s2...sn]	encoded(offset of period)	JSON Object
Media	edgeid-encoded(period)-pid-hid	d	media type id [m1,m2..m5]	encoded(offset of period)	JSON Object

4.5.3 Migrating HCA-T Data

In HCA-T2, we designed two Oozie coordinator jobs for data migration. One is for the inactive data (*HCA*, *Patient* and *Service*), the other is for active data (*Appointment*, *ServiceRecord* and *MediaMessage*). For each *smart edge* where HCA-T2 is installed, there is a workflow corresponding to the data source and including sub-workflows for each table. We configured the job to be executed every day at midnight, and configured the migration chunk size based on the estimated generated data in HCA-T application. For example, the chunk size for appointments is configured as 88,000 rows, as there are 4 work slots in each day for 22,000 HCAs in a province. As a result, there are 5MB data of appointments, 56MB data of service records, and around 50MB data of media messages (excluding the media files themselves) generated in one province to be migrated in an iteration. In terms of number of mappers for each table migration, by considering the cluster resources and the size of migrated data, we configured 1 mapper for *Appointment*, 3 mappers for *ServiceRecord*, and 3 mappers for *MediaMessage*.

4.6 Evaluation

In this section, we report on our empirical evaluation of this work by measuring the migration time and the query execution time of HCA-T2. Because of sensitivity of the data in HCA-T system, in this experiment, we simulated the time-series data with the software spawner [27] and the location data with a Java application. We simulated one year of data for two provinces, resulting in about 400 GB of data. For all test cases, we ran the experiment 5 times and took the mean of the last three.

4.6.1 Environment Setup

Our experiments were performed on an eight-node HBase/Hadoop cluster, running on eight virtual machines in the *core* in SAVI cloud. The virtual machines run 64bit Ubuntu 12.04 and have 2 cores, 4GB of RAM, and a 100 GB disk. In the ON and BC *smart edge*, one MySQL instance in HCA-T application is installed in a single virtual machine (2cores, 4GB of RAM and 200GB disk), respectively.

We used Hadoop version 1.0.2, HBase version 0.94, Sqoop version 1.4.3, and Oozie version 3.3.2. Hadoop and HBase were each given 2GB of Heap size in every running node. HDFS was configured with a replication factor of 3. The maximum number of mapper jobs is 2 in each node, which is consistent with the number of CPU cores. In terms of the HBase client configuration, gzip compression was configured on the table to reduce the data-transmission time. Next, the *ROWCOL* filter was applied on each table for narrowing the queried range. The scan cache size was set to 5000 and the block cache was set to true, for the query processing.

We did some performance tuning for the Hadoop and HBase cluster according to suggestions from Apache website [8] [9]. To make this cluster performance available for comparison and the experiment results understandable, we evaluated the cluster with the standard benchmark first. In terms of the performance of map-reduce, we used *MRBench* [31], with the configuration of 10 number of runs, and for each run there are 1,000,000 input lines, and 10 maps and 8 reduces. The average execution time is 36.14 seconds. In terms of HBase write/read performance, we used the *PerformanceEvaluation* benchmark [2] provided in HBase source code package. With the configuration of 5 clients to write 5GB data in total into HBase randomly, the total execution time is 7m11.588s and the write throughput is 11.87 MB per second, while the read throughput of HBase in this cluster is 786 KB per second.

[2]Each client is inserting 1 million rows with 10 mappers, about 1GB size (1000 bytes per row) in default [10].

4.6.2 Performance Evaluation

With the excellent scalability and fault tolerance mechanism obtained from HBase, migration and query performance become the biggest concerns in *DAAS*. We first studied the impact of the number of involved *smart edges* on the migration execution time. Given the limited resource of Hadoop cluster, we mainly focus on the continuously generated active data. To evaluate the migration performance, in the experiment, we also deployed one more HCA-T system in *ON Edge* to emulate the workload with a third edge. Figure 4.5 demonstrates the effect on performance of the number of *smart edges* involved. In this experiment, the Oozie coordination jobs, in charge of migrating the three tables from one to three *smart edges* to the *core*, are configured to iterate every day at midnight. As shown in Figure 4.5, the execution time increases rapidly when the number of *smart edges* increases to three. From the result, we can also see even though AB edge deployed in Toronto which is near to the core, the migration time is still high. By looking into each table migration in the case of three migrated *smart edges*, we found that the Sqoop job execution time increases substantially. As the Sqoop job consumes much memory and computation when it executes the SQL queries via JDBC, more memory and computing power are required with more *smart edges*. Therefore, a larger Hadoop cluster would be helpful to improve the performance. In addition, we can also observe that the migration for *bc-edge* is a little slower than the other two, which is caused by the different locations: *bc-edge* is physically deployed in Victoria, while *on-edge*, and *ab-edge* is launched in Toronto.

The second set of experiments examine the spatial query performance in HBase. Figure 4.6 shows the performance of the range query (described in Section 4.5) with the various distances. In this experiment, we randomly selected five location points from *BC smart edge*, and got their neighbours with the distances from 0.2 km to 6 km. As shown in Figure 4.6, within 2 seconds, around 40,000 neighbours within the distance of 6 km, can be obtained.

We then evaluated the time serial query performance by comparing against MySQL. The query is to find out how many appointments in two provinces in a given period from 1 week to 3 months. We optimized MySQL with the following settings:

```
{
   key_buffer_size:1G,
   sort_buffer_size:  16M,
   tmp_table_size:1G,
```

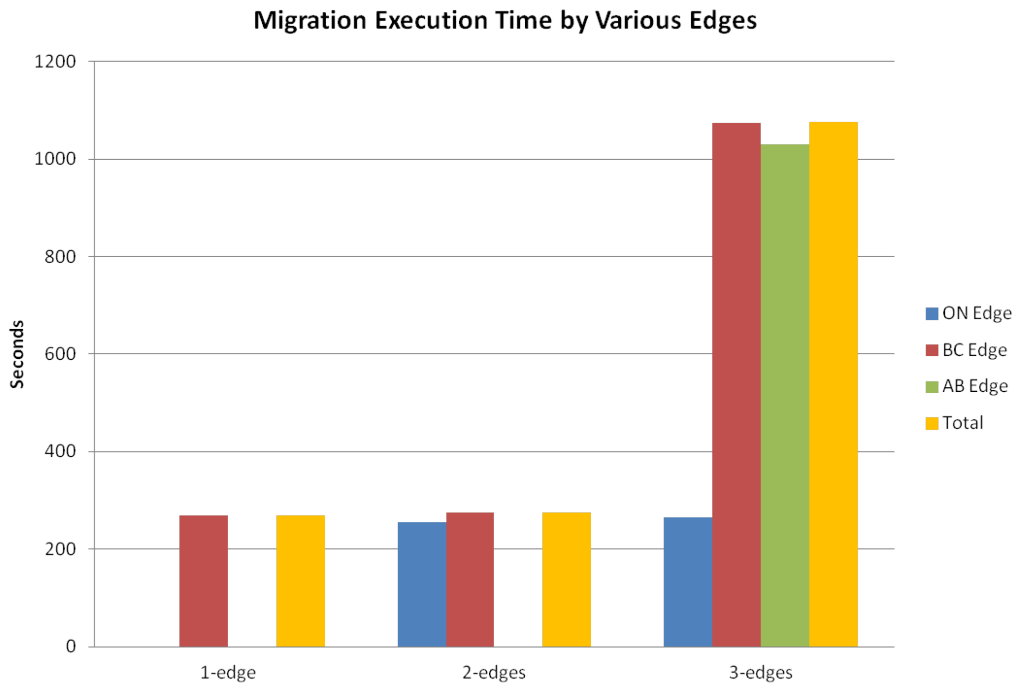


Figure 4.5: Migration Execution Time with Various Edges

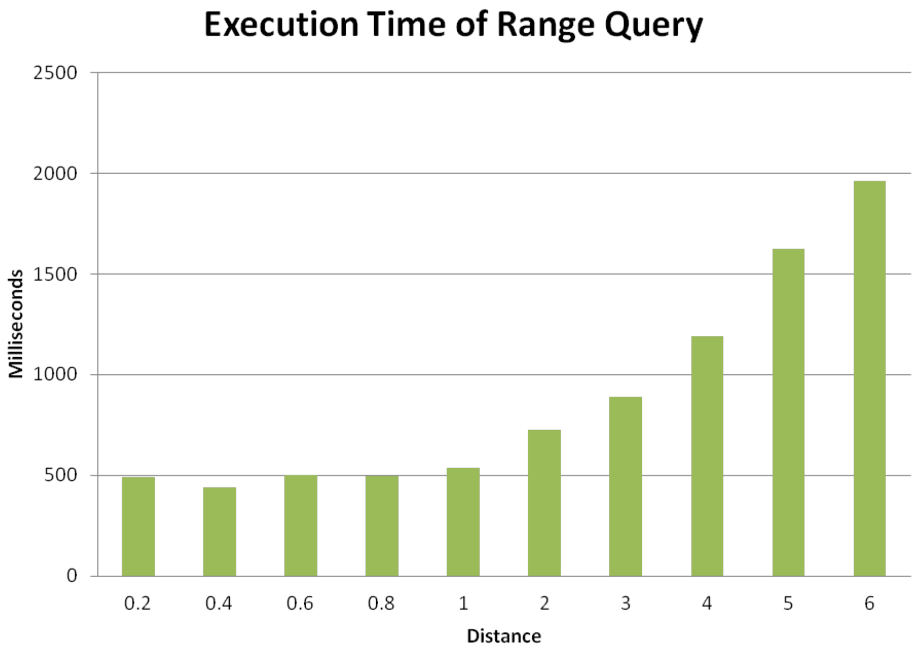


Figure 4.6: Execution Time of Range Queries with Various Distances

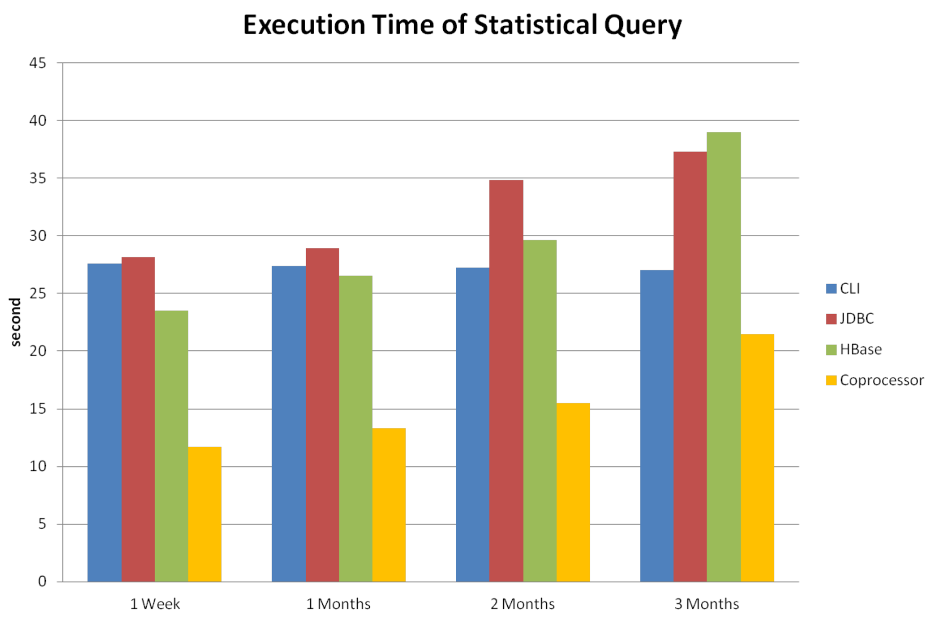


Figure 4.7: Comparison of Statistical Queries on HBase and MySQL

```
    max_heap_table_size:  1G,
    read_buffer_size:  512K,
    read_rnd_buffer_size:  512K,
    myisam_sort_buffer_size:  1G
}
```

Besides MySQL CLI, we also developed the same query with JDBC. It should be noted that the execution times shown in Figure 4.7 with CLI and JDBC are for the query on the data from only one *smart edge*, while in HBase, it is a federated query on the data from both *smart edges*. It is obvious that performance in HBase is better than that in MySQL. By breaking down the query execution time in HBase, we can see that the execution time of *Coprocessor* only accounts for 50% of the total query time show in Figure 4.7. The other half is mostly consumed by network communication and data transmission. Much performance improvement can be obtained by improved network performance in the cloud. Of course, one might argue that with rich experience on MySQL like in Flickr [5], MySQL can also demonstrate improved performance. Although we agree in principle, we argue that such experience is very hard and takes a long time to obtain. What is more, the elasticity and excellent fault-tolerance mechanism offered by HBase makes MySQL less attractive and hang behind already.

4.7 Discussion

DAAS in the cloud-enhanced architecture is designed to migrate data from various data formats and various data sources and support multiple applications. The case study mentioned above only demonstrates one type of data source, namely relational databases. However, with the support of Sqoop and Oozie, it can support many data sources by adding the relevant extensions. For example, besides RDBMSs, Sqoop[14] also supports delimited text or SequenceFiles, which can become another kinds of migrated data sources. Oozie[12] not only supports Sqoop actions, but also other actions such as email actions, shell actions, and custom actions. With customized executors, *DAAS* can be easily extended.

Another important part of the architecture is the method for transforming the RDBMS data schema to HBase. The rules we presented in this chapter have been applied into the prototype implementation in this work. In general, they can be applied to similar geospatial applications, but do not cover all applications of interest. Given the variety of web-based data-driven applications, these rules are unlikely to cover all applications. A more extensive (and more complete) set of such guidelines and a systematic and uniform way are part of our future work.

4.8 Conclusion and Future Work

In this chapter, we presented a cloud-enabled architecture which can support developers deploy web-based applications on a hierarchical cloud, with little change. In addition, this architecture offers a data access and aggregation system for enabling researchers and administrators to perform integrated analytics on the complete data set collected by all application instances, to obtain domain-specific business insights. To enable this migration, we came up with a method for transforming the data, originally modelled according to the entity-relationship paradigm in the RDBMS, to HBase. Taking both the migration performance and cost, and the additional large data processing ability into account, we can safely conclude that the cloud-enhanced architecture we proposed in this work can be easily applied by many typical web-based applications.

Without a doubt, there is still a substantial amount of work to do in the future. First, systematic ways to transform data originally modelled in other styles (such as object-oriented modelling) to HBase are required. Second, the data access and aggregation system in the cloud-enhanced architecture should be extended to be more general for various applications and provide more flexible APIs for application developers. In the future, it should be general enough to embrace a variety of applications. Finally, the *AnalyticsEngine* should be enhanced by more complicated map-reduce jobs and other tools (such as Pig [13], Mahout [11]) to provide more powerful access and analytics ability.

Acknowledgement

The authors would like to thank the support of the cloud team at Cybera Inc, who gave us the generous access to their cloud environment for developing and testing. The author also want to thank the support from SAVI testbed team. This work has been funded by the SAVI Strategic Network, NSERC, AITF and IBM.

Bibliography

[1] Travis A Bennett and Coskun Bayrak. Bridging the data integration gap: from theory to implementation. *ACM SIGSOFT Software Engineering Notes*, 36(3):1–8, 2011.

[2] David Bernstein, Erik Ludvigson, Krishna Sankar, Steve Diamond, and Monique Morrow. Blueprint for the intercloud-protocols and formats for cloud computing interoperability. In *Internet and Web Applications and Services, 2009. ICIW'09. Fourth International Conference on*, pages 328–336. IEEE, 2009.

[3] Michael R Blaha, William J Premerlani, and James E Rumbaugh. Relational database design using an object-oriented methodology. *Communications of the ACM*, 31(4):414–427, 1988.

[4] Peter Pin-Shan Chen. The entity-relationship modeltoward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[5] Kellan Elliott-McCrea. Using, Abusing and Scaling MySQL at Flickr, August 2013.

[6] Apache Software Foundation. Apache HBase Reference Guide, April 2012.

[7] Apache Software Foundation. Apache Hadoop Reference Guide, August 2013.

[8] Apache Software Foundation. Apache Hadoop Wiki Performance Tuning, August 2013.

[9] Apache Software Foundation. Apache HBase Wiki Performance Tuning, August 2013.

[10] Apache Software Foundation. Apache HBase Wiki Testing HBase Performance and Scalability, August 2013.

[11] Apache Software Foundation. Apache Mahout: Scalable machine learning and data mining, August 2013.

[12] Apache Software Foundation. Apache Oozie User Guide, July 2013.

[13] Apache Software Foundation. Apache pig!, August 2013.

[14] Apache Software Foundation. Apache Sqoop User Guide, July 2013.

[15] Nikolay Grozev and Rajkumar Buyya. Inter-cloud architectures and application brokering: taxonomy and survey. *Software: Practice and Experience*, 2012.

[16] Rajeev Gupta, Himanshu Gupta, Ullas Nambiar, and Mukesh Mohania. Enabling active data archival over cloud. In *Services Computing (SCC), 2012 IEEE Ninth International Conference on*, pages 98–105. IEEE, 2012.

[17] Dan Han and Eleni Stroulia. A three-dimensional data model in hbase for large time-series dataset analysis. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2012 IEEE 6th International Workshop on the*, pages 47–56. IEEE, 2012.

[18] Dan Han and Eleni Stroulia. Hgrid a data model for large geospatial data sets in hbase. In *Cloud 2013*. IEEE, 2013.

[19] Bradley Simmons Michael Smit Hongbin Lu, Mark Shtern and Marin Litoiu. Pattern-based deployment service for next generation clouds. In *IEEE 9th World Congress on Services*. IEEE, 2013.

[20] Ya-Ting Hsu, Yi-Chin Pan, Ling-Yin Wei, Wen-Chih Peng, and Wang-Chien Lee. Key formulation schemes for spatial index in cloud data managements. In *Mobile Data Management (MDM), 2012 IEEE 13th International Conference on*, pages 21–26. IEEE, 2012.

[21] Joon-Myung Kang, Hadi Bannazadeh, and Alberto Leon-Garcia. Savi testbed: Control and management of converged virtual ict resources. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 664–667. IEEE, 2013.

[22] George Lee, Jimmy Lin, Chuang Liu, Andrew Lorek, and Dmitriy Ryaboy. The unified logging infrastructure for data analytics at twitter. *Proceedings of the VLDB Endowment*, 5(12):1771–1780, 2012.

[23] Chongxin Li. Transforming relational database into hbase: A case study. In *Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference on*, pages 683–687. IEEE, 2010.

[24] S. Nishimura, S. Das, D. Agrawal, and A.E. Abbadi. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, volume 1, pages 7–16. IEEE, 2011.

[25] Aaron Schram and Kenneth M Anderson. Mysql to nosql: data modeling challenges in supporting scalability. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 191–202. ACM, 2012.

[26] Mike Smit, Joanna Ng, Marin Litoiu, Gabriel Iszali, and Alberto Leon-Garcia. Smart applications on virtual infrastructure. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pages 381–381. IBM Corp., 2011.

[27] Sourceforge. Spawner Data Generator, August 2013.

[28] E Stroulia, I Nikolaidisa, L Liua, S King, L Lessard, et al. Home care and technology: a case study. *Studies in health technology and informatics*, 182:142–152, 2011.

[29] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1013–1020. ACM, 2010.

[30] Moshe Y. Vardi. The universal-relation data model for logical independence. *Software, IEEE*, 5(2):80–85, 1988.

[31] Tom White. *Hadoop: the definitive guide*. O'Reilly, 2012.

[32] Gio Wiederhold. Modeling databases. *Information sciences*, 29(2):115–126, 1983.

Chapter 5

Conclusion and Future Works

"Big Data" is being continuously generated from various sources in high volumes, high velocity, and high variety [4]. A typical cause of the high volume is the frequent data updates collected by sensor-based monitoring equipment and mobile devices over time and/or space [3]. Real-time monitoring systems collect thousands of hundreds of metrics, from hundreds of machines every minute. Retailers log billions of transactions for millions of customers in thousands of stores in a year. Social-network systems handle and record requests from millions of on-line users every day. Scientific simulations and health-care equipments recording medical and biological observations collect numerous high-resolution images and thousands of samples every day. Spatial-data repositories have also been exploding as the number of location-aware applications, which contextualize users' experience based on their frequently updated locations, increases daily. Recommendation and advertisement systems are receiving millions of devices registering their location updates continuously for the service. Bicycle-service system responds to thousands of users' requests according to their current locations and logs each request for business analysis.

Needless to say, such a rapid growth brings about a challenging scalability requirement on the data management system. Compared to relational database management systems (RDBMSs), NoSQL (Not Only SQL) databases, non-relational distributed database system, endowed with high availability, good elasticity and excellent scalability through their easy deployment on cloud computing platforms, become a more attractive data-storage solution for these big-data applications. Given the advantages of the cloud, it is attractive to migrate existing applications to this new platform. However, the little available development support and steep learning curve around this technology seriously impede its adoption. Clearly more development support is needed to advance the adoption of these systems.

In this study, we developed a set of general guidelines for the design of HBase storage, and a specific three-dimensional HBase "schema" for geospatial applications. In addition, we proposed a method of transforming the data schemas in typical RDBMSs to HBase. These guidelines have been evaluated with the migration of an existing geospatial application to the cloud.

5.1 Contributions

More specifically, this thesis makes the following contributions.

- We have proposed a data model for time-series data in HBase, and evaluated it with several frequently used temporal queries. This data model takes advantage of the *version* dimension of HBase to improve the performance of many typical temporal queries, such as the statistical queries on a period of time. The experiment results also provide the evidence that an appropriate schema for a given dataset highly depends on the query pattern.

- We have proposed a data model for spatial data in HBase and evaluated it with several frequently used spatial queries, such as range queries and k-nearest neighbours queries. This data model is based on a hybrid index structure *HGrid*, that combines a quad-tree and a regular grid as primary and secondary indices correspondingly. Comparing *HGrid* against two other data models based on quad-tree and regular-grid indices, we have demonstrated that *HGrid* supports efficient performance with less stringent resource requirements. Through this study, we also formulated a set of guidelines on how to organize data for geospatial applications in HBase.

- We have designed an architecture for extending data-driven web-based applications with an HBase-based analytics component, and a systematic method for migrating existing applications to this architecture.

- We have proposed a method for transforming data schemas in RDBMSs to HBase, and applied it to a practical case study. Inspired by the work in [1] where they transformed the relational data schema to Hive based on a universal relation data model, we focused on the entity-relationship relational data model. In this method, a set of guidelines is formulated for developers to consider and a five-step transition process to follow during the migration of their application data in RDBMSs to HBase.

- We have performed a practical case study with a real application migrating to a hierarchical cloud, demonstrating (a) the use of existing tools to support the migration and (b) the application of the above guidelines in the design, and verifying the aforementioned data schema transition method.

5.2 Summarized Configurations

The query performance in HBase is impacted by a lot of aspects from underlying infrastructure at the bottom up to the configuration of HBase on the top. In this work, by setting up a certain configuration in HBase, Hadoop and its underlying environment, we focused on the performance implication from different data organizations. To make it more clear about how to use the results and reproduce the experiments in this work, more detailed configurations from three levels including application data models, HBase and Hadoop are described in this section.

5.2.1 Application Data Models

To get better query performance, the data feature and the query pattern should be the first aspect to examine. In the real world, the data in applications is usually very complex and not as clear as what we used in experiments like Cosmology dataset and Bixi dataset. Therefore, it is very important to classify the data into different categories.

In terms of time-series data, two questions such as “What is the velocity of increase?” and “How many attributes in an object?” can help to feature the data. With huge number of objects in a timestamp, like Cosmology dataset, it is better to disperse the sequential data across the cluster with the special design of row key, for example, reversed id. With a small number of objects involved in a large number of timestamps, like Bixi dataset, more localized data by taking advantage of the third dimension is a suggested solution. Query patterns should also be considered to answer the question like “Which attributes should be put in the row key, column and version?”.

In terms of spatial data, questions such as “What is the data-set space?”, “What is the query distance in spatial queries?”' and “What is the density of the spatial data?” can help to choose the index structure. The key configuration of the spatial indexing is the granularity of the cell. There is no such an exact value which can be applied to all cases. For each case, the configuration of the cell should be experimented and tuned based on the query requirement and the data-set space. During experiments, the first configuration can be determined by the distance frequently used in queries, then decreased or increased with a large step, until a turning point appears where the performance goes down.

5.2.2 HBase and Hadoop Configuration

Scan Caching Size This is the property *hbase.client.scanner.caching* controls scanner caching. It indicates how many rows will be fetched from the server in a single round trip in a scan if the data is not served in memory. Setting this value to 5000, for example, means there will be 5000 rows transferred at a time to be processed. With Coprocessor framework, a larger value of this property will cost more memory in HBase RegionServer, hence, with a limited memory resource or a lower data processing, a smaller caching size is suggested. Our observation in the previous experiments tells that, when the scanned data in a query is more than the caching size, the execution time increases rapidly. Therefore, in this work, the configuration is calculated based on the size of each row and the memory allocated in coprocessors. Speaking of this point, as the sizes of each row across the different data organizations are different, more performance for each data model might be obtained with a fine tuning on this parameter.

Block Cache This property should be set as true for the frequently accessed row. It can be set via setCacheBlocks method for the Scan instance.

Bloom Filters It can be enabled per column family. There are three types including None, ROW and ROWCOL. If ROW is set, the hash of the row will be added to the bloom on each insert. If ROWCOL is set, the hash which is calculated based on row, column family, and column name will be added to the bloom on each key insert. In this work, we suggest to use ROWCOL filter, which can help prune the data even though more memory are required to store bloom data.

Compression There are many compression techniques available, such as LZO, GZIP, SNAPPY. To use LZO and SNAPPY, additional installations are required. To make it simple, in this work, we used java code-based gzip compression.

There are some more configurations which we did not examine but might have impact on the performance, such as the region size of HBase and the replication factor in Hadoop. For each configuration, lots of experiments are required in order to give more detailed suggestion. Given the limited time and resource, our work just took the first step to explore the problem in this area.

5.2.3 Query Processing

Besides the configurations, the query processing implementation is also a very important factor. Guidelines about how to implement efficient query processing based on HBase characteristics, has been summarized in Chapter 3.

5.3 Future Work

In this thesis, we have made some initial progress on modelling data in HBase for geospatial applications. However, given the end goal of re-architecting various legacy applications and migrating them on the cloud, substantial research is still needed. To this end, we have identified the following three avenues for extending our work.

First, we plan to investigate data beyond the geo-spatial domain. Of particular interest are the data collected by social-networking applications, such as text, images and videos. These new types of data are bound to require different organization models to support the types of queries that administrators and scholars typically issue on them. We propose to follow a similar methodology as the one we have followed in Chapter 2 and Chapter 3 to study these new data types.

Second, we plan to investigate other NoSQL tools, beyond HBase. Considering four general categories of NoSQL databases: key-value stores, column-family stores, document stores and graph databases[2], how to model the data to fit in other types of databases should be investigated as well.

Finally, a more general method for transforming data schemas from RDBMSs to HBase is needed.

Bibliography

[1] Rajeev Gupta, Himanshu Gupta, Ullas Nambiar, and Mukesh Mohania. Enabling active data archival over cloud. In *Services Computing (SCC), 2012 IEEE Ninth International Conference on*, pages 98–105. IEEE, 2012.

[2] R. Hecht and S. Jablonski. Nosql evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 336–341. IEEE, 2011.

[3] Adam Jacobs. The pathologies of big data. *Communications of the ACM*, 52(8):36–44, 2009.

[4] Doug Laney. 3d data management: Controlling data volume. *Velocity, and Variety, Application Delivery Strategies published by META Group Inc*, 2001.