A Three-Dimensional Data Model in HBase for Large Time-Series Dataset Analysis

Dan Han, Eleni Stroulia
Department of Computing Science
University of Alberta
Edmonton, Canada
{*dhan3, stroulia*}*@cs.ualberta.ca*

***Abstract*—In the transition of applications from the traditional enterprise infrastructures to cloud infrastructures, scalable database management system plays an important role in efficiently managing and analysing unprecedented massive amount of data. Compared to RDBMSs, NoSQL databases, are more attractive in addressing this challenge. However, it is not easy to manage data in NoSQL database effectively for non-expert users because of the rare data-organization support. A poor data organization may accidentally abuse the features of NoSQL database and achieve unsatisfactory performance. Therefore, a systematic method for NoSQL database data-schema design is a timely and important problem for researchers and practitioners.**

HBase, as a particular NoSQL database offering, relies (a) on HDFS, for its distributed and replicated storage, and (b) on coprocessors, for efficient parallel query processing. To harness the potential parallelism benefits, an appropriate partitioning of the data across the HBase storage is required. we investigate the effectiveness of the three-dimensional data model, which uses the "version" dimension of HBase to store the values of a data item over time. We have experimented and evaluated the performance impact of this type of data model 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 HBase Coprocessors framework, across different data schemas, some of which (do not) use the third HBase dimension. The experiment results demonstrate improved performance with the data schemas that use the third dimension of HBase.

***Keywords*-Data Model; Data Schema; Time-Series Dataset; HBase; Coprocessor**

I. 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.[1]. 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 [2]. 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 [2]. 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[1]. As 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 [1]. 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 [3]. It can be categorized into four types: Key-value stores, Column Family stores, Document stores and Graphic databases[4]. 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 paper. 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

978-1-4673-3001-5/12/$31.00

them into 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* [5]. 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 paper, 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 [6], produced by a simulation, and the Bixi dataset [7], 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 paper 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 date 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 paper is organized as follows. Section II reviews the background and related work in this area. Section III introduces the data models for time-series data, which is instantiated in the dataset domains in Section IV. Section V 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 VI. We conclude our contributions and future work in Section VII.

II. Background and Related Research

HBase uses the Hadoop File System (HDFS) as its underlying data storage platform. As we have mentioned in Section I, 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* [5]. 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* [5]. 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 [5]. 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 [8].

The HBase Coprocessor framework, inspired by Googles BigTable coprocessors [9], provides a library and run-time environment for executing user-level code within HBase region servers [10]. 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 [11], 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. OpenTDSB [12] is a distributed scalable time-series database, written on top of HBase. OpenTDSB 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 [13], 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 paper; however the 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.

III. Three-dimensional Data Model

In this paper, 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 I 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).

Table I
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

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 [5]; 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.

IV. Case Study

A. The Datasets

The **Cosmology Dataset** [6] 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 [6]. 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** [7] 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

Table II
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 III
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

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 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.

B. 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 III.

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 II(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 II(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 II(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 a mimic way of hashing partition, 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

Table IV
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]

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 shortness of hashing partition.

C. Three Alternative Data Schemas for Bixi dataset

Table IV 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 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 V(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 V(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 V(c) shows a sample of Data Schema3.

Table V
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]

V. 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.

A. 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 ultra-fast hard drivers in a RAID 5 configuration, and uses VMWare to host a set 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 5KB caching size. 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 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 [10]. 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.

B. 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 [6]. 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 little increased 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.

C. Experimental Analysis

For the Cosmology dataset, we performed experiments for all three queries as described in Section V-B with three data schemas shown in Section IV-B. For the Bixi dataset, one experiment was executed for the query described in Section V-B with three data schemas presented in Section IV-C.

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.

1) Analysis of Cosmology Dataset: Table VI(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 VI(b). 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 VI(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 VI(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.

2) Analysis of Bixi Dataset: The Bixi query was evaluated with ten scenarios whose execution times are shown in Table VII and Table VIII. The distributions of working regions, i.e., regions on which the coprocessor instances run, for five of these scenarios are shown in Table VIII. 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 VII, Data Schema3 shows better performance than the other two, and Data Schema2 shows better performance than Data Schema1. 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 VIII, 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.

VI. 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 column 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 paper 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

Table VI
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;24	NA	53.325	55.783	6,395,664	6,396,955
2;512;24	NA	79.113	76.163	12,416,253	12,417,544
2;216;128	NA	52.273	49.012	3,652,989	6,396,955
2;512;128	NA	66.709	80.449	9,673,578	12,417,544
2;512;216	NA	61.991	81.325	6,020,589	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 VII
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 VIII
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)

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[10].

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 paper, 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.

VII. Conclusion and Future Work

HBase, as a NoSQL database offering, is rapidly becoming the chosen solution for scalable data processing. In this paper 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 coprocessors, 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.

References

[1] 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*. ACM, 2011, pp. 530–533.

[2] 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*, pp. 1–19, 2010.

[3] R. Agrawal, A. Ailamaki, P. Bernstein, E. Brewer, M. Carey, S. Chaudhuri, A. Doan, D. Florescu, M. Franklin, H. Garcia-Molina *et al.*, "The claremont report on database research," *ACM SIGMOD Record*, vol. 37, no. 3, pp. 9–19, 2008.

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

[5] "Apache HBase Reference Guide," http://hbase.apache.org/book/book.html.

[6] S. Loebman, D. Nunley, Y. Kwon, B. Howe, M. Balazinska, and J. 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*. IEEE, 2009, pp. 1–10.

[7] "Bixi Dataset," https://s3.amazonaws.com/bixidata/bixi_comp.tar.gz.

[8] D. Borthakur, J. Gray, J. 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*, vol. 11, 2011, pp. 1071–1080.

[9] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

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

[11] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. Hassaan, R. Kaleem, T. 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*. ACM, 2011, pp. 12–25.
[12] "Whats OpenTSDB?" http://opentsdb.net/.
[13] "Modeling Time Series Data on top of Cassandra," http://engineering.rockmelt.com/post/17229017779/modeling-time-series-data-on-top-of-cassandra.