

Chapter 2

Intelligent Web Data Management of NoSQL Data Warehouse

2.1 Introduction

2.1.1 Background

Relational database management systems (RDBMSs) have been widely used for decades to store two dimensional data. However, it will have the I/O bottleneck issues in a world of big data, especially for the read (e.g. query and search) operation. To address this limitation, several systems have already emerged to propose an alternative schema-free database, known as NoSQL [1]. A NoSQL database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases. Motivations for this approach include simplicity of design, horizontal scaling and finer control over availability, especially in using in big data and real-time web applications. The data structure (e.g., tree, graph, and key/value) differs from the RDBMS, and therefore some operations are faster in NoSQL and some in RDBMS. Despite the consistency problem, the benefits of using NoSQL outweigh its disadvantages.

A document-oriented database, or simply called document store for short, is one of NoSQL databases. It is commonly used to store, retrieve, and manage document-oriented information, which encapsulates and encodes data in some standard formats. Encodings in use include XML, JSON, as well as BSON. MongoDB [2] is an example of the leading document-based NoSQL database, which enables new types of applications, better customer experience, and faster time to market and lower costs for organizations of all sizes. Currently, there are some classical solutions and best practices of the MongoDB.

2.1.2 *Challenges and Contributions*

Nowadays, data warehouse plays an important part in most of areas of data management including data analysis, data mining and decision support activities in academic and industrial community. Data administrators create the habit of using high-specialized data structures that data warehouses usually maintain materializing their most essential multidimensional perspectives for business analysis and decision. The data in the data warehouse today is enormous, and besides supporting their information needs that they also enrich their common knowledge by bringing new data from very specialized outsourced repositories. The challenge in the data warehouse is that it would have very little effect on the track of the historical data after reducing the data redundancy and compressing the data.

As we know, dimension tables are used to provide subject-oriented information by providing data elements to filter, aggregate or describe facts in a relational data warehouse. Thus, it is quite important that dimensions remain consistent accordingly in some data warehouse's time frame even when some values of their attributes change over time. When this occurs, dimensions that change slowly over time rather than changing on regular schedule. It is often called as slowly changing dimensions (SCD) [3, 4].

However, most of the publications just discuss the SCD approaches of RDBMS. In the field of the emerging NoSQL databases, few publications have mentioned the technologies about how to formulate the data warehouse of NoSQL. Not only the structure of NoSQL is different from RDBMS, but also NoSQL instance has bigger data than RDBMS in practice. It is just natural that whether distributed computing can integrate with the SCD approach or not is wondered. Currently, MapReduce is a functional programming model for processing large data sets with a parallel, distributed algorithm on a cluster. Therefore, MapReduce framework is used to accelerate the formulation of the data warehouse.

In the rest of this chapter, a SCD approach of document-based NoSQL data warehouse is introduced. MapReduce is taken advantage of to benefit SCD in the scenario of NoSQL databases effectively. The main difficulty to construct document-based NoSQL data warehouses is to achieve the balance among the complexity, efficiency and flexibility. The contributions of this chapter are into several folds. First, this SCD approach can highly compress the historical data in the data warehouse of schema-free document stores without data redundancy. Second, this SCD approach can keep track of the history, providing quicker access to the snapshot at every point of a day or over a period of time. Finally, it is managed to assure that the formulation of the daily cell with an effective lifecycle tag is efficient and transparent to the business of applications.

2.2 Related Works and Emerging Techniques

Although the structure of the NoSQL is different from RDBMS, a relational database is a special case of NoSQL in a broad way. The SCD principles to RDBMS adapt to NoSQL through continuous innovative improvement. Currently, there are at least three categories of strategies to the SCD. The first approach is daily full backup. In this solution, all the daily backups that stand alone formulate the whole data warehouse, which are simple but need very large storage spaces. Each backup is a snapshot of the database at a particular point in time. The second approach is daily incremental backup. An incremental backup includes only those things that have changed since the previous backup and saves those things into a separate and additional location. By definition, the first incremental backup is a full backup since it backs up everything since there is no previous backup to compare to. The next incremental backup backs up only those files that have changed since the previous backup was taken. This incremental backup can result in a much smaller backup. The cost of using incremental backups is one of the management. Since each incremental backup relies on the backup that preceded it, in order to restore the database to an arbitrary point in time all the incremental backups must be available to perform the restore. The pros of this SCD approach cost significantly less disk space used compared to an equivalent set of full backups. Moreover, The cons of this SCD approach are that the baseline full backup and all the incremental backups must be preserved and available in order to restore. Besides, the deleted historical data are not easy to be tracked. Thus, another improvement of this approach is daily incremental backup with the invalid partition. The content of the data warehouse is divided into the last full backup, this daily latest increment and this daily invalid partition. For the track of the deleted data, this is got from the daily invalid partition. The last approach is daily differential backup. Differential backup is a kind of hybrid approach. In reality, it is just incremental backup, but with a fixed starting point. Rather than backing up only changes from the previous day, each differential backup includes all the changes from the baseline of full backup. Compared with the second incremental approach, each day can be restored from only two backups, the initial baseline full plus that day's differential.

Although SCD is not a new problem during the dimensional modeling phase of a data warehouse, the state of art of SCD in the context of document-oriented NoSQL databases is ambiguous. There are many mature SCD approaches to make data warehouse of specific RDBMS. As for the emerging schema-free NoSQL databases, few publications have mentioned the technologies about how to formulate the data warehouse.

2.2.1 *Slowly Changing Dimensions of RDBMS*

While dimension table attributes are relatively static, they are not fixed forever. Dimension attributes change, albeit rather slowly, over time. Dimensional designers must engage business representatives proactively to help determine the appropriate change-handling strategy. While it is assumed that accurate change tracking is unnecessary, business users may be assumed that the data warehouse will allow them to see the impact of each and every dimension change. Special oriented Extract-Transform-Load (ETL) processes are responsible to maintain SCD, acting accordingly to the updating strategy previously defined. Usually, a Change Data Capture (CDC) process detects the update event and records it accordingly to some predefined SCD requisites. Although it is always spoken in terms of process, SCD strategies are applied at the attribute level, for a particular dimension table. For each dimension table, designers define what kind of update attributes will be modified over time. They must prevent all the possible cases of attribute changing, once it is not very recommendable to change dimension table structures when the DWS is already in production. Over a same SCD, distinct updating strategies for a single record is to be applied. Frequently, this involves different processes for different SCD strategies.

Dealing with the issues involves SCD management methodologies referred to as Type 0 through 6 [5]. Type 6 SCDs are also sometimes called Hybrid SCDs.

Type 0: The passive method

The Type 0 method is passive. It manages dimensional changes and no action is performed. Some dimension data can remain the same as it was first time inserted, others may be overwritten. In certain circumstances history is preserved with a Type 0. High order types are employed to guarantee the preservation of history whereas Type 0 provides the least or no control.

Type 1: Overwriting the old value

In this method no history of dimension changes is kept in the database. The old dimension value is simply overwritten by the new one. Thus, it does not track historical data. This type is easy to maintain and is often used for data which changes are caused by processing corrections (e.g. removal special characters, correcting spelling errors).

An example of Type 1 is shown in Fig. 2.1. This example is a teacher table with attributes ID, number, name, and title. ID is the natural key, and number is a surrogate key. If the teacher is promoted to professor, the record would be overwritten. The disadvantage of the Type 1 method is that there is no history in the data warehouse. It has the advantage however that it's easy to maintain.

Type 2: Adding a dimension row

In this methodology, all history of dimension changes is kept by creating multiple records for a given natural key in the dimensional tables. Unlimited history is preserved for each insert. Both the prior and new rows contain as attributes the natural key. The first method to track the data is called 'current indicator'. There

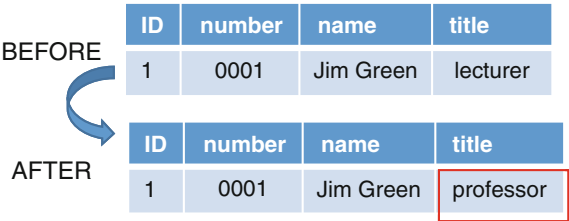


Fig. 2.1 An example of type 1

could be only one record with current indicator set to ‘Y’. Another method to track the data is called ‘effective date’. For ‘effective date’ columns, i.e. start_date and end_date, the end_date for current record usually is set to value 9999-12-31. The 9999-12-31 end_date in row two indicates the current record version. Two examples of these two methods are shown in Figs. 2.2 and 2.3. Introducing changes to the dimensional model in type 2 could be very expensive database operation so it is not recommended to use it in dimensions where a new attribute could be added in the future.

Transactions that reference a particular surrogate key are then permanently bound to the time slices defined by that row of the slowly changing dimension table. An aggregate table summarizing facts by state continues to reflect the historical state. If there are retrospective changes made to the contents of the dimension, or if new attributes are added to the dimension which have different

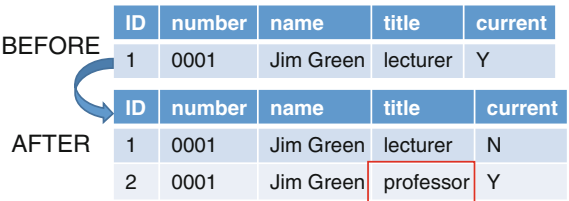


Fig. 2.2 Current indicator method of type 2

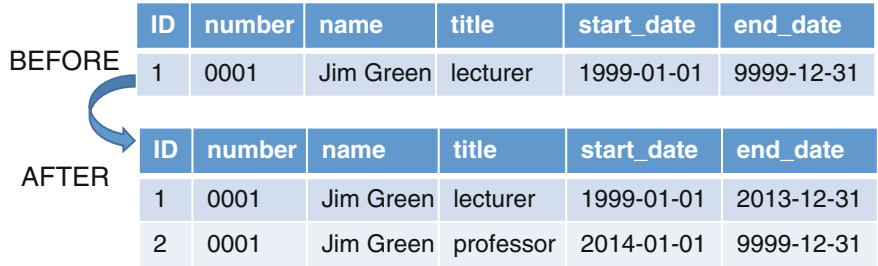


Fig. 2.3 Effective date method of type 2

effective dates from those already defined, then this can result in the existing transactions needing to be updated to reflect the new situation. This can be an expensive database operation, so Type 2 SCDs are not a good choice if the dimensional model is subject to change.

Type 3: Adding a dimension column

This method tracks changes using separate columns and preserves limited history. The Type 3 preserves limited history as it is limited to the number of columns designated for storing historical data. The original table structure in Type 1 and Type 2 is the same but Type 3 adds additional columns. In the following example shown in Fig. 2.4, an additional column has been added to the table to record the title of a teacher—only the previous history is stored. The new value is loaded into ‘current’ column and the old one into ‘previous’ column. This record contains a column for the previous title and current title track the changes. Figure 2.4 shows an example of Type 3.


Type 4: Using historical table

The Type 4 method is usually referred to as using historical table, where one table keeps the current data, and an additional table is used to keep a record of some or all changes. Both the surrogate keys are referenced in the fact table to enhance query performance. For the above example the original table name is teacher and the history table is teacher_history. Figure 2.5 shows an example of Type 4. This method resembles how database audit tables and change data capture techniques function.

Fig. 2.4 An example of type 3

BEFORE

ID	number	name	previous_title	current_title
1	0001	Jim Green	lecturer	lecturer



AFTER

ID	number	name	previous_title	current_title
1	0001	Jim Green	lecturer	lecturer
2	0001	Jim Green	lecturer	professor

Fig. 2.5 An example of type 4

teacher

ID	number	name	title
1	0001	Jim Green	professor

teacher_history

ID	number	name	title	create_date
1	0001	Jim Green	lecturer	2013-12-31

BEFORE

ID	number	name	previous_title	current_title	start_date	end_date	current
1	0001	Jim Green	lecturer	lecturer	1999-01-01	9999-12-31	Y

AFTER

ID	number	name	previous_title	current_title	start_date	end_date	current
1	0001	Jim Green	lecturer	lecturer	1999-01-01	2013-12-31	N
2	0001	Jim Green	lecturer	professor	2014-01-01	9999-12-31	Y

Fig. 2.6 An example of type 6

Type 6: Hybrid methods of types 1, 2, 3 ($1 + 2 + 3 = 6$)
The Type 6 method combines the approaches of types 1, 2 and 3 ($1 + 2 + 3 = 6$). This hybrid method is also called unpredictable changes with single-version overlay. Figure 2.6 shows an example of Type 6.

However, the above classical SCD approaches have some limitations. Type 1 cannot keep track of historical data. Type 2, 3 and 4 have too much redundant data. In our opinion, keeping the track of historical data is the unique SCD question, especially for the NoSQL databases. In our point of view, when keeping history in dimension data is spoken about, all types of SCD can be implemented quite effectively using only SCD type 4. Classifying an SCD as a new Type X is not exactly true, types of SCD are applied to attributes, so in general dimensions with attributes that are type 1 are considered, other attributes that are type 2 or type 3.

2.2.2 Slowly Changing Dimensions of NoSQL

Since the data in the data warehouse are used for data analysis with SCDs, we are only concerned about the daily data rather than the real-time data. That is to say that only the last change of the data each day is valid. Thus, we will merge all the real-time data each day into a set of final daily data. There are two solutions to slowly changing dimensions of NoSQL [6].

Solution 1: document with timestamp

Obviously, we can conclude that the historical collection dimension is a feasible SCD solution to the schema-free document stores. In this solution, the historical data are saved in another separate collection with the timestamp every day. In order to make further comparison, we call this approach “document with timestamp” next. Figure 2.7 shows the structure of document with timestamp. However, this solution has too many disadvantages. The first one is that the data warehouse is

Fig. 2.7 Document with timestamp

ID	A1	...	An	timestamp

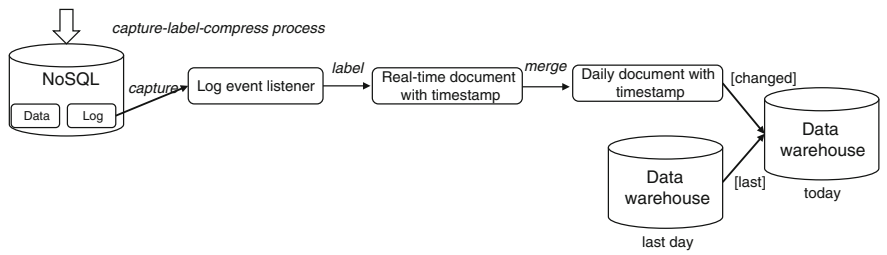


Fig. 2.8 Process of the formulation of the NoSQL data warehouse using document with timestamp

huge enough so that the storage becomes a disaster over a long period. The second one is that it will consume too many excessive storage spaces due to the redundant data. The process of the formulation of the NoSQL data warehouse using this solution is shown in Fig. 2.8. First, we capture the changed document-based data from the NoSQL using the log event listener. Second, we merge the changed data with a timestamp. Third, we compress all the real-time documents with a timestamp on a day with the same surrogate key into one document. At last, all the compressed daily documents with timestamp plus the data warehouse last day formulate the new data warehouse today.

Solution 2: document with a lifecycle tag

Another improved version of the document with timestamp solution is called “document with a lifecycle tag”. Figure 2.9 shows the structure of document with a lifecycle tag. In this solution, we use the start and end timestamp instead of the unique timestamp. Although this solution saves the storage space compared with the first solution, it has the challenge on how to generate the collection dimension in the data warehouse. Similar to the timestamp solution, the process of the formulation of the NoSQL data warehouse using this solution is shown in Fig. 2.10. First, we capture the changed document-based data from the NoSQL using the log event listener. Second, we label the changed data with lifecycle tags. Third, we compress all the real-time documents with lifecycle tags on a day with the same surrogate key into one document. At last, all the compressed daily documents with lifecycle tags plus the data warehouse last day formulate the new data warehouse today.

The above two solutions have the same disadvantage that the daily documents in the data warehouse have excessive storage space due to the data redundancy. For this purpose, we design a new SCD approach adapted to the schema-free document stores using fine-grained element. We call this approach “cell with an effective

Fig. 2.9 Document with a lifecycle tag

ID	A1	...	An	start_date	end_date

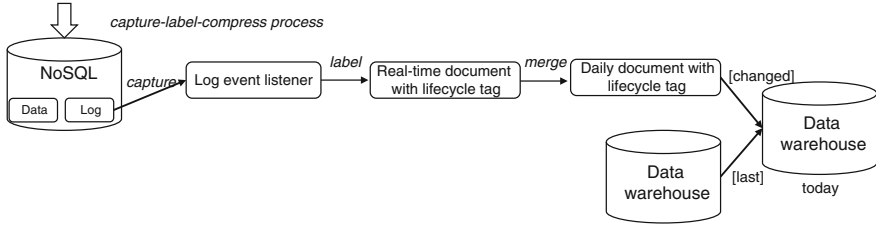


Fig. 2.10 Process of the formulation of the NoSQL data warehouse using document with a lifecycle tag

lifecycle tag using MapReduce”. This idea comes on this premise that the changing of the database is slow enough to have redundant data in the data warehouse.

2.2.3 MapReduce Framework

MapReduce is a programming model for parallel processing large data sets, which was proposed by Google in 2004 [7]. A popular free implementation is Apache Hadoop. The model is inspired by the map and reduce functions commonly used in functional programming, although their purpose in the MapReduce framework is not the same as their original forms. The process is divided into two steps: map and reduce. Developers specify Map functions which take an input pair and produces a set of intermediate key/value pairs. The MapReduce library then groups together all intermediate values associated with the same intermediate keys, and passes them to the Reduce functions which are also specified by the programmers; The Reduce function accepts an intermediate key and a set of values for that key. It merges together these values to form a possibly smaller set of values. The intermediate values are supplied to the user’s reduce function via an iterator. The programming model of MapReduce framework is shown as follows:

- map: $(k1, v1) \rightarrow \text{list}(k2, v2)$. Map Function. Map function takes an input key-value pair $(k1, v1)$ and produces a set of intermediate key-value pairs $(k2, v2)$.
- reduce: $(k2, \text{list}(v2)) \rightarrow \text{list}(k3, v3)$. Reduce Function. Reduce function merges the intermediate key-value pairs $(k2, \text{list}(v2))$ together to produce a smaller set of values $\text{list}(v3)$.

As for the application scenarios of SCD, MapReduce can improve the computational capacity to the creation of the NoSQL data warehouse. Map function in MapReduce corresponds with mapping the document-based data into real-time cells with lifecycle tags, and reduce function in MapReduce works in concert with merge all the real-time cells with lifecycle tags into daily cells with lifecycle tags.

2.3 Requirements

First, we have the following requirements and their corresponding solutions while designing slowly changing dimensions of NoSQL with MapReduce.

- **Keeping track of history:** Since the historical data are used for data analysis and decision making, we want to keeping track of history. In our solution, we use cells with an effective lifecycle tag using MapReduce to track all the changes to the source NoSQL database.
- **Quick access to the historical snapshot at every point of a day or over a period of time:** We want to further analyze the data in the NoSQL data warehouse. This implies needing to support the quick access to the historical data on a specific day or over a period of time.
- **Reduce the storage space and remove the redundant data from the data warehouse:** Since the data in the data warehouse are compressed as the final daily cells with effective lifecycle tags, we expect to reduce the storage space and remove the redundant data of the data warehouse. That is to say that the current cell with timestamp and lifecycle tag solutions need the high compression ratio of the NoSQL data warehouse. In our solution, we split the changed document stores into sets of cells with effective lifecycle tag divided by column.
- **The high efficiency of the creation:** Due to the large amount of data in the source NoSQL database, the creation of the NoSQL data warehouse can benefit from the distributed computing. Therefore, we use MapReduce framework to generate the schema-free document stores in the data warehouse.
- **Data consistency preservation:** We want to preserve the consistency semantics that the data warehouse provides. Since the process of data analysis and decision making need not be done in real time, we can tolerate the delay of the data consistency. We can miss the latest but cannot provide the incorrect historical data.
- **Transparency to the business:** We expect the creation of the data warehouse is independent of the application using NoSQL databases. This will be implemented with the change data capture engine independent of the source database. Thus, we design a log-based change data capture that will impact the performance of the source database at a minimum.

2.4 Architecture

In this section, we discuss the proposed architecture for changing the dimensions of NoSQL with MapReduce.

2.4.1 Deployment Architecture

The typical architecture of slowly changing dimensions of NoSQL with MapReduce is shown in Fig. 2.11. The architecture is composed of four layers in order to reduce the complexity. From the top to the bottom, they are data layer, computing layer, historical storage layer and application layer respectively.

On top is the data layer. Since EDSM of schema-free document stores is designed, MongoDB is adopted as the NoSQL in this layer. All the historical data in the data warehouse come from this source NoSQL. The second layer from the top is the computing layer, which is the core of our proposed middleware. The Capture-Map-Reduce procedure is used to generate the data in the data warehouse. The details are discussed in the next section. The second layer from the bottom is the historical storage layer. The historical data in this layer are stored. At the bottom is the application layer. This layer consists of all kinds of applications using the data warehouse, such as decision making, data analysis and data mining.

2.4.2 Capture-Map-Reduce Procedure

As the name suggests, the tentative plan for the capture-map-reduce procedure consists of the following steps: capture, map and reduce procedure. It is shown in Fig. 2.12. Strictly speaking, the capture-map-reduce procedure refers to a kind of ETL.

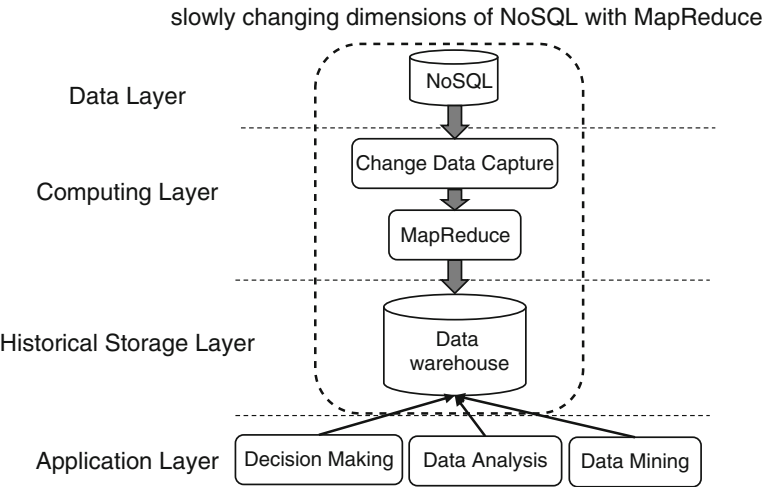


Fig. 2.11 Architecture

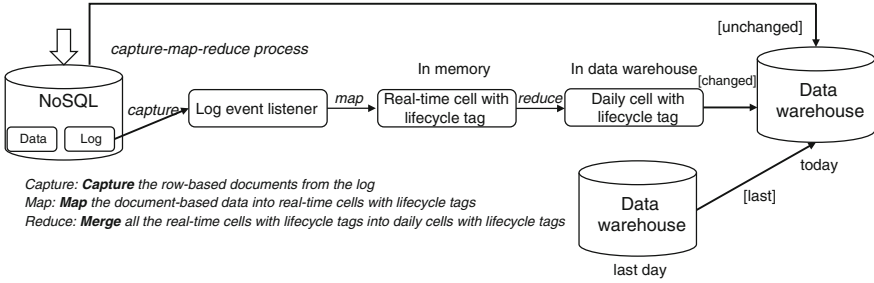


Fig. 2.12 Capture-map-reduce process

2.4.3 Log-Based Capture

Our log-based engine has been designed to support the extraction from different source NoSQL databases. This flexibility is achieved by allowing different captures to be developed and plugged into the capture engine. In this Chapter, schema-free document stores are mainly discussed. This capturer is typically an embedded thread and must follow a few semantic constraints. For example, oplog (operations log) is used to implement MongoDB capture.

In the scenario of change data capture (CDC), only the changed data reading from the operational log are concerned. The document-oriented NoSQL products often provide document-based replication (DBR) events to capture changed data from the operational log. This API is used to connect the listener to get the incremental data. By interpreting the contents of the database operational log one can capture the changes made to the database in a non-intrusive manner.

In the process of DBR, the document events parsed from the oplog contain changes to one or more documents in a transaction in a given collection. A document change may consist of one or two full document copies in its turn. These are generally known as before image (BI) and after image (AI). Each image has a different purpose: the BI, containing data as it was before the document was modified, is used for locating the document to be updated/deleted in the storage engine, while the AI is used for replaying the actual changes. In addition, the one and only usage for such image is to help finding the correct cell to be updated or deleted. In order to describe the CDC approach, the representation of the image is given, denoted as image: (keyname, k_1, k_2, \dots, k_n), where keyname is the unique key of this image, and k_t ($1 \leq t \leq n$) is the name of the cell. An instance of this image is generally denoted as (keyvalue, v_1, v_2, \dots, v_n), where keyvalue is the value of this unique key, v_t ($1 \leq t \leq n$) is the value of the cell. Not both images are needed for every operation. Deletes only need the BI, and inserts just need the AI, while updates generate pairs of images for each row changed. Summing up, BI must contain values that uniquely identifies documents, acting like a primary key,

while AI must contain values that make possible changing the document according to the original execution.

Most databases have binary logs that contain the log of changes as they are applied to the database. However, it is fragile to mine these logs and reverse-engineer the structure, because there is no guarantee that the format will be stable across multiple subsequent versions. In the case of MongoDB though, it is possible to tap into the storage engine API. MongoDB product itself provides a stable that has been used to build many commercial and open-source storage engines. The capture interface is implemented in the form of adapters so that it can be reused to support more NoSQL databases.

2.4.4 MapReduce

In the MapReduce procedure, the changed data (in the form of document-based data) from the log event listener are transformed into the final daily cells with lifecycle tags in the data warehouse. The innovation of our approach is utilizing MapReduce framework. In the first Map procedure, the document-based data are mapped into real-time cells with effective lifecycle tags. For the insert changes, they are mapped into the newborn cells with effective lifecycle tag. For the delete changes, they are mapped into the dead cells with effective lifecycle tag. For the update changes, they are mapped into the dead and newborn cells with effective lifecycle tags in order. The Map procedure is shown in Fig. 2.13, where UUID and

Algorithm 1 Map algorithm *map*

Input:

Collection name *source*;

Output:

cells with effective lifecycle tag *cells*;

```

1: procedure map()
2:   switch (operation) //log event
3:   case insert:
4:     // afterImage ← (keyname, k1, k2, ..., kn) : (keyvalue, v1, v2, ..., vn);
5:     Cell.add((naturalKey, keyname, keyname, tag) : (Cell.UUID(), keyvalue, keyvalue, (currentstamp, null)));
6:     for i = 1 to n do
7:       Cell.add((naturalKey, keyname, ki, tag) : (Cell.UUID(), keyvalue, vi, (currentstamp, null)));
8:     end for
9:   case delete:
10:    beforeImage ← (keyname, k1, k2, ..., kn) : (keyvalue, v1, v2, ..., vn);
11:    for i = 1 to n do
12:      Cell.edit((naturalKey, keyname, ki, tag) : (Cell.lastUUID(), keyvalue, vi, (lastStart(), currentstamp)));
13:    end for
14:   case update:
15:    beforeImage ← (keyname, k1, k2, ..., kn) : (keyvalue, b1, b2, ..., bn) from pair;
16:    afterImage ← (keyname, k1, k2, ..., kn) : (keyvalue, a1, a2, ..., an) from pair;
17:    for i = 1 to n do
18:      Cell.edit((naturalKey, keyname, ki, tag) : (Cell.lastUUID(), keyvalue, bi, (Cell.lastStart(), currentstamp)));
19:      Cell.add((naturalKey, keyname, ki, tag) : (Cell.UUID(), keyvalue, ai, (currentstamp, null)));
20:    end for
21:   end switch
22: endprocedure

```

Fig. 2.13 Map algorithm

Algorithm 2 Reduce algorithm *reduce*

Input:
 real-time cells with effective lifecycle tags *cells*;

Output:
 daily documents *documents*;

```

1: procedure reduce()
2:   classify cells by surrogate key;
3:   for each surrogateKey  $\in$  cells do
4:     list=the cells including this surrogateKey;
5:     merge the cells list into a cell newcell with the
       last value today;
6:   end for
7: endprocedure

```

Fig. 2.14 Reduce algorithm

lastUUID are the functions to obtain an unique key and the last unique key of a cell respectively, and lastStart is the function to get the last start timestamp of the cell. Besides, the add and edit is the function to insert and update the cell.

In the second Reduce procedure, all the real-time cells with effective lifecycle tags are merged into daily cells with effective lifecycle tags. The Reduce procedure is shown in Fig. 2.14. First, all the cells with effective lifecycle tags within a day are collected. Next, they are classified by the surrogate key. Since the daily data rather than the real-time data for further analysis are concerned, the set of the cells with the same surrogate key is merged. The value of this cell depends on the last value of this day.

2.5 Evaluation

Slowly changing dimension (SCD) problem applies to cases where the attribute of a record varies over time. In fact, most of the business data in the NoSQL applications are of this kind. Since only a small part of the cell and the document varies over time, this generates a lot of redundant data. In the SCD process, the probability of the changing of a document and cell is very small (generally less than 10 %). To test the performance of different NoSQL SCD solutions, source schema-free document stores are generated using the script we design. This generator was configured to create results for 180 days each having average 5000 documents. Thus, data for one month gave 900,000 fact documents. Our cell with an effective lifecycle tag solution is compared with the current document with timestamp and lifecycle tag solutions. Our experiments were performed on a six-core server (Xeon(R) CPU

E7-4807 @1.87 GHz, 130 G RAM, 500TB SATA-III disk, Gigabit Ethernet) running MongoDB 2.2.7 and Hadoop MapReduce 2.2. The system was configured with a Windows Server 2012 x64.

The source NoSQL and its data warehouse are initialized as empty. After that, the script is executed to generate the data in the source document stores. At the same time. Our EDSM begin to formulate the data in the data warehouse. The experimental data of document with timestamp and lifecycle tag solutions are observed, and our cell with an effective lifecycle tag solution. In the scenario of SCD, it is assumed that a document remains unchanged with probability 90 %, and a key in a document remains unchanged with probability 80 %. While in the scenario of non-SCD, it is assumed that a document remains unchanged with probability 40 %, and a key in a document remains unchanged with probability 50 %.

2.5.1 Redundancy Rate

First, the redundancy experiment of SCD and non-SCD solution to illustrate the advantage of our solution have been made. Figures 2.15 and 2.16 show the

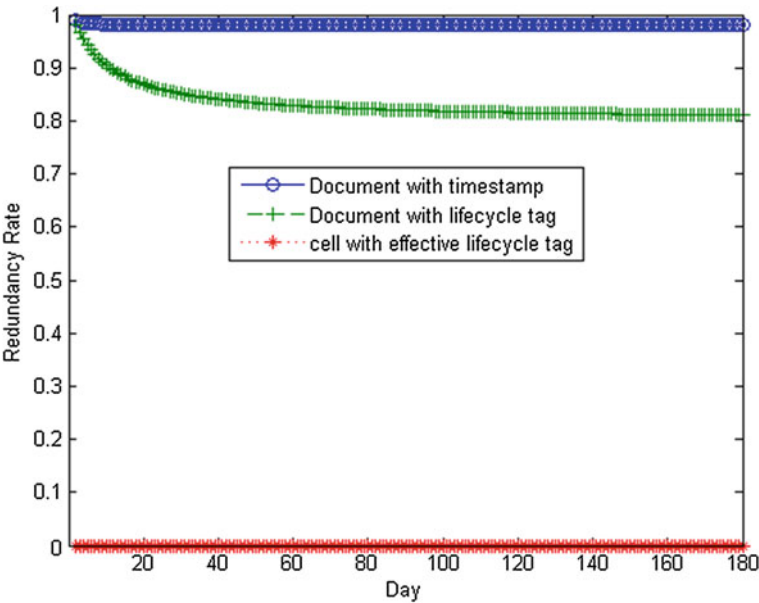


Fig. 2.15 Redundancy rate of SCD

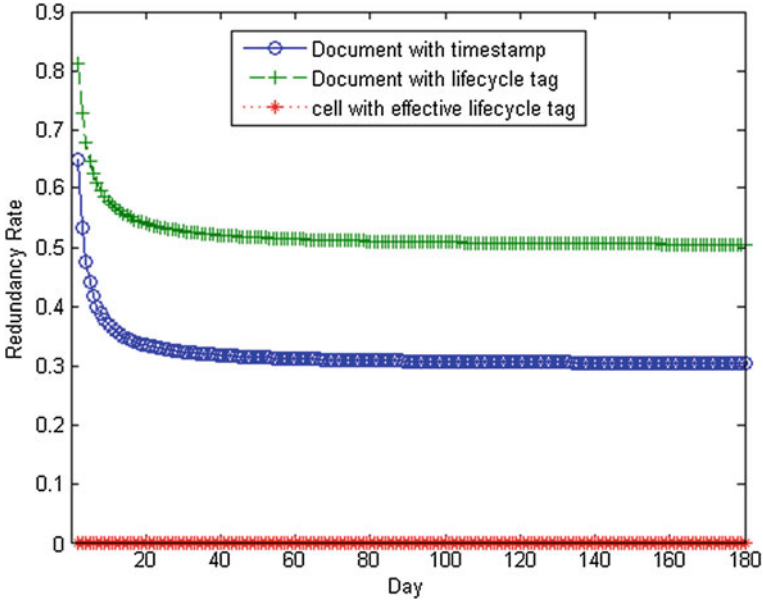


Fig. 2.16 Redundancy rate of non-SCD

redundancy rate every day in the scenario of SCD and non-SCD respectively. As depicted in SCD Fig. 2.15, the first two solutions have the serious bottleneck problem of the redundancy. The redundancy rate of the document with timestamp solution heads toward 100 % in the SCD scenario, since all the documents need to be stored regardless of whether the documents are changed or not. The redundancy rate of the document with lifecycle tag solution is diminished gradually due to the consolidation of the unchanged data at different days. As depicted in non-SCD Fig. 2.16, the redundancy rate of the document with timestamp solution is down with the unchanged probability of the documents. No matter the SCD and non-SCD, our cell with an effective lifecycle tag solution handles the redundancy problem better. Owing to the breakup of the documents into cells, our solution produced no additional redundant data.

2.5.2 Storage Space

Next, the storage space of different solutions is analyzed. It is assumed that the data size increases by average 5000 documents from the source NoSQL every day. The storage space of the corresponding data warehouse every day is observed. Figures 2.17 and 2.18 shows the variation of different three solutions respectively.

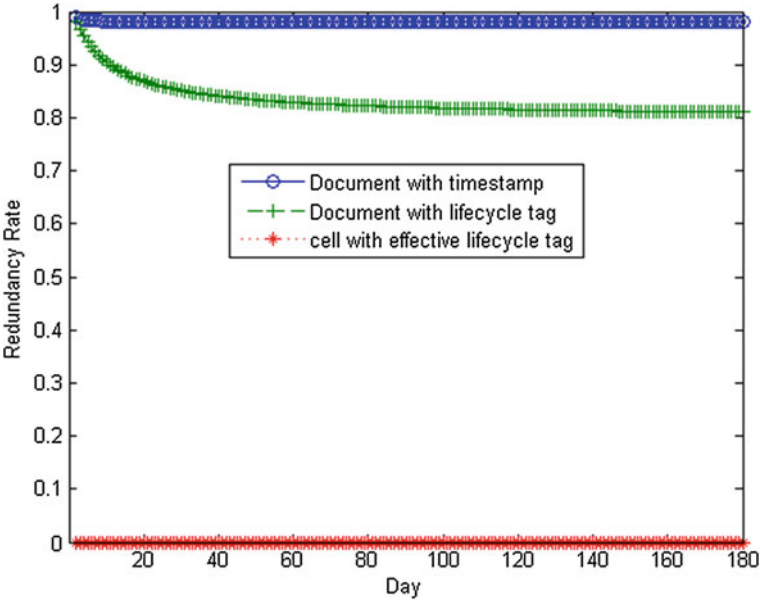


Fig. 2.17 Storage space of SCD

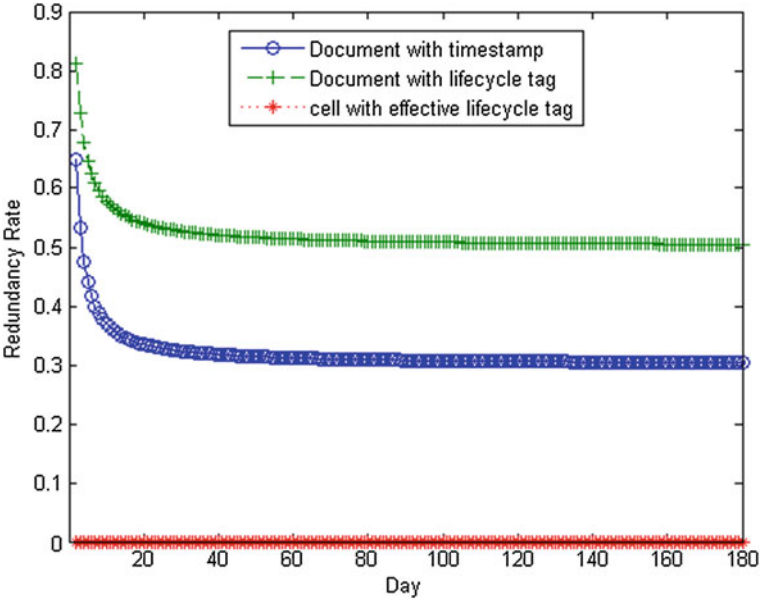


Fig. 2.18 Storage space of non-SCD

Scales of the data size of the corresponding data warehouse increase linearly. The fastest increasing solution is document with timestamp, since all the documents need to be stored regardless of whether the documents are changed or not. As depicted in SCD Fig. 2.17, our cell with effective lifecycle tag solution is superior to the document with lifecycle tag solution. However, our solution takes no remarkable superiority in the non-SCD scenery, which is shown in Fig. 2.18. It is concluded that our cell with an effective lifecycle tag solution is effective in the SCD scenario.

2.5.3 Query Time of Track of History

In the process of the storage space experiment, the query time of the historical data in the data warehouse is measured. In order to make the comparison of different solutions, the historical data on the first day are selected. 8 points to record the time of the same query are selected. Figures 2.19 and 2.20 shows the query time of different three solutions. As depicted in Fig. 2.19, the worst showing on the query performance is the document with timestamp solution due to the large amount of historical data in the data warehouse. The increase of our cell with an effective lifecycle tag solution is lower. Next, the query time over a period of time is

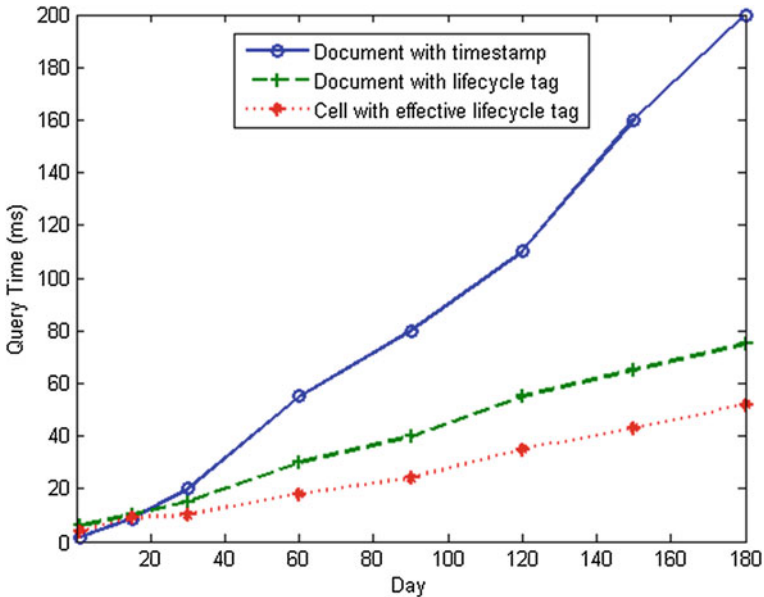


Fig. 2.19 Query time of SCD

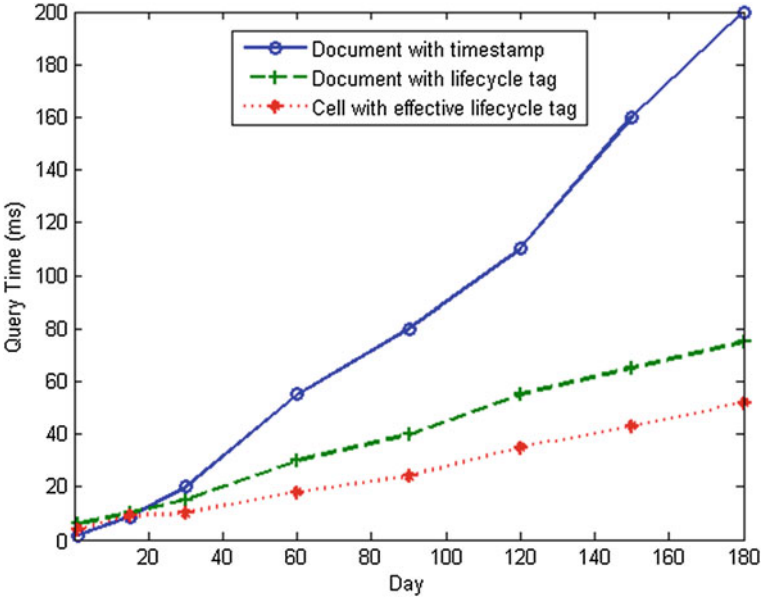


Fig. 2.20 Query time of non-SCD

measured. The historical data from the 30 to 40 days are selected during the 180 days. As depicted in Fig. 2.20, the query time is always the lowest. It is concluded that our solution is feasible in practice.

Generally, data redundancy and data consistency conflict with each other. slowly changing dimensions of NoSQL with MapReduce attempts to assure low data redundancy as well as eventual and weak consistency. The data in the data warehouse might miss the latest value. One day delay is tolerated, since data warehouse is used in the field of data analysis and decision making.

2.5.4 Execution Time of Creation

Finally, the execution time of label-merge procedure (used by the first two solutions) is compared with map-reduce procedure that we use. As depicted in Fig. 2.21, map-reduce solution consumes the shortest time with the deployed Hadoop MapReduce framework.

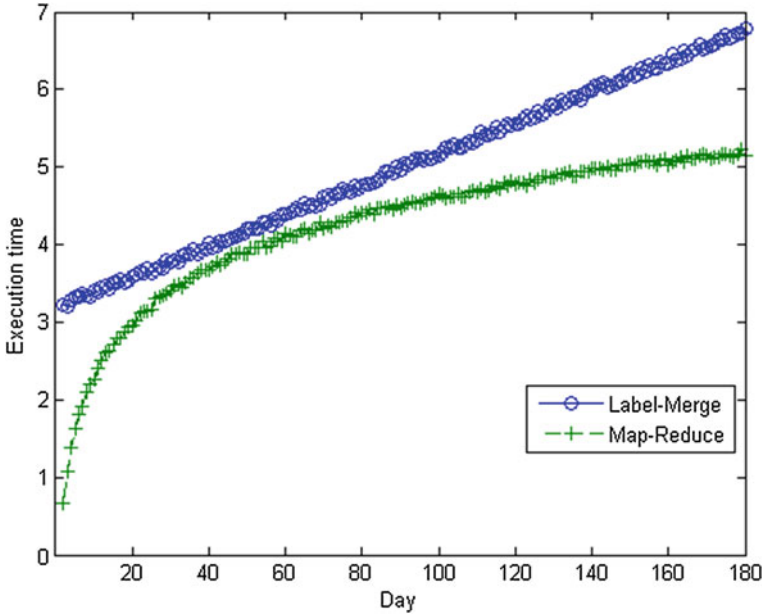


Fig. 2.21 Execution time of label-merge and map-reduce procedures

2.6 Discussion

In this section, we discuss the structure of cell with effective lifecycle tag, and the extreme data storage principles for optimization.

In order to define the slowly changing dimensions of NoSQL with MapReduce, a mathematical representation of the fine-grained model is necessary. In our solution, we map the changes to source NoSQL database into sets of real-time cells with effective lifecycle tags divided by column. Then the real-time cells are merged into daily cells. We assume that the Extract-Transformation-Load (ETL) process extracts all the changed data exists in a source NoSQL database to load into a specific dimension of the data warehouse.

2.6.1 Effective Lifecycle Tag

First, we give the definition of the effective lifecycle tag. The effective lifecycle tag is a 2-tuple of start and end element, which implies the lifecycle of a cell in the NoSQL data warehouse. The effective lifecycle tag is denoted as $(start, end)$. The first element of an effective lifecycle tag is the start timestamp, and the second element of an effective lifecycle tag indicates the end timestamp.

Next, we define some special effective lifecycle tags.

- **Newborn effective lifecycle tag:** For the inserted document of the source NoSQL database, we will split it into several cells in the data warehouse by column. Its value of the effective lifecycle tag is (currentTimestamp, null), where currentTimestamp means the current timestamp.
- **Killed effective lifecycle tag:** For the deleted document of the source NoSQL database, we will split it into several cells in the data warehouse by column. Its value of the effective lifecycle tag is (start, currentTimestamp), where currentTimestamp means the current timestamp.
- **Active effective lifecycle tag:** When the end tag of the effective lifecycle tag is null, it indicates that this cell is alive.
- **Dead effective lifecycle tag:** When the end tag of the effective lifecycle tag is less than the current timestamp, it indicates that this cell becomes the historical data.

An update operation of the source NoSQL database is split into two atomic operations (delete the original data, and insert the new data). Seen from the dimensional point of view, update means the death of the original data and the born of the new data.

2.6.2 *Cell with Effective Lifecycle Tag*

We define the cell with effective lifecycle tag in the data warehouse as a list of attributes that comprise a natural key, a surrogate key, a regular key and an effective lifecycle tag, denoted as (naturalKey, surrogateKey, regularKey, tag), where naturalKey is the unique identification of the dimension of the data warehouse, surrogateKey is the unique identification of the fact of the source NoSQL database, regularKey is the name of the attribute, and tag is the effective lifecycle tag introduced in the last section. This definition is the metamodel of the cell with an effective lifecycle tag. That is to say that the cell in practice is the instance of this model.

2.6.3 *Extreme Data Storage Principles*

From the above characteristics of the cell with an effective lifecycle tag, we arrive at the following conclusions.

- Each changed document in the source NoSQL database generates several cells with effective lifecycle tags.
- Each cell has one and only one effective lifecycle tag.
- Each effective lifecycle tag belongs to a set of cells.

In fact, the data administrators only pay close attention to the daily data of the data warehouse rather than the real-time data. Therefore, we design the extreme data storage middleware to further compress the real-time data into the daily data. In other words, the timestamp of the effective lifecycle tag is exact to the day. We will merge all the real-time cells with effective lifecycle tag into daily cells with a lifecycle tag as the extreme compression using MapReduce framework. This will be discussed in the next section.

Next, we analyze the principles of the historical snapshot. Figure 2.22 shows the method to get the historical 1-day snapshot. The cells with effective lifecycle tags

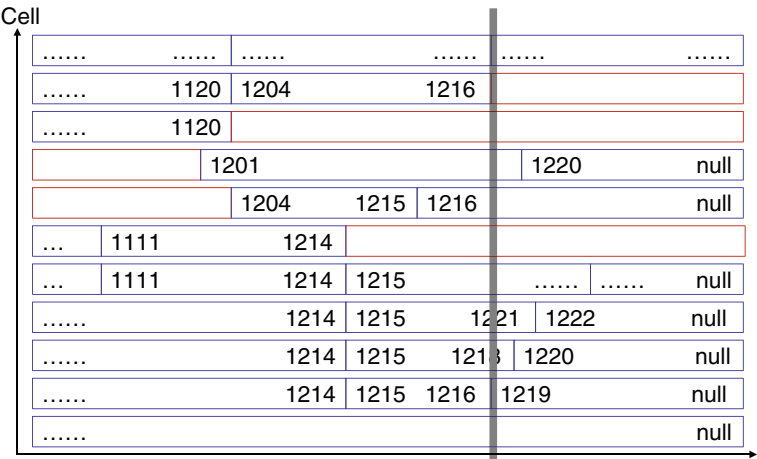


Fig. 2.22 The method to get 1-day historical snapshot

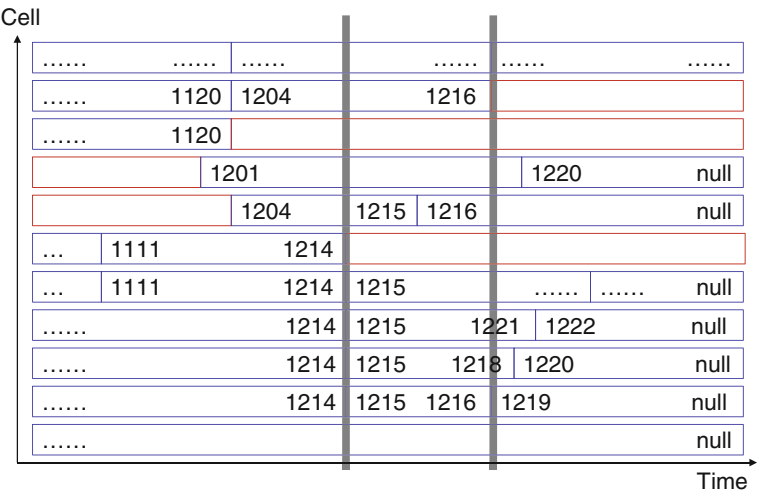


Fig. 2.23 The method to get the historical snapshot over a period of time

that are penetrated by the black line formulate the full snapshot of the historical data in the date 1217. The snapshot of any day is achieved in this way.

Figure 2.23 shows the method to get the historical snapshot over a period of time. The cells with effective lifecycle tag that are penetrated by the two black lines formulate the full snapshot of the historical data between the date 1215 and 1217. The snapshot for some time is achieved in this way.

2.7 Conclusions

As the emerging NoSQL databases, MongoDB is a document database that provides high performance, high availability, and easy scalability. Currently, there are more and more NoSQL applications, such as the social networking site. However, it lacks the approaches to the NoSQL data warehouse. In this chapter, a methodology for slowly changing dimensions of NoSQL with MapReduce is introduced. The innovation of this method is that it reduces the data redundancy in the data warehouse, and optimize the storage structure. Due to the support of the parallel and distributed computing framework, MapReduce is used to improve the performance of the creation of the NoSQL data warehouse.

References

1. Cattell, R. (2011). Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4), 12–27.
2. Chodorow, K. (2013). *MongoDB: The definitive guide*. O'Reilly Media, Inc.
3. Kimball, R., Ross, M., Thornthwaite, W., Mundy, J., & Becker, B. (2008). The data warehouse lifecycle toolkit: Practical techniques for building data warehouse and intelligent business systems.
4. Santos, V., & Belo, O. (2011). Slowly changing dimensions specification a relational algebra approach. In *Proceedings of the International Conference on Advances in Communication and Information Technology* (pp. 50–55).
5. Leonard, A., Mitchell, T., Masson, M., Moss, J., & Ufford, M. (2014). Slowly changing dimensions. In *SQL server integration services design patterns* (pp. 261–273). Apress.
6. Ma, K., & Yang, B. (2015). Introducing extreme data storage middleware of schema-free document stores using MapReduce. *International Journal of Ad Hoc and Ubiquitous Computing*, 274–284.
7. Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1), 107–113.

Intelligent Web Data Management: Software
Architectures and Emerging Technologies

Ma, K.; Abraham, A.; Yang, B.; Sun, R.

2016, XIV, 162 p. 108 illus., 78 illus. in color., Hardcover

ISBN: 978-3-319-30191-4