Partitioning Technology and Fast Content Movements of Big Data

TE-YUAN LIN, CHIOU-SHANN FUH
Computer Science and Information Engineering, National Taiwan University
8F., No.7, Songren Rd., Xinyi Dist., Taipei City 110, Taiwan (R.O.C.)
D03922002@ntu.edu.tw, fuh@csie.ntu.edu.tw

ABSTRACT

Database storage storing abundant data usually accompanies slow performance of query and data manipulation. This thesis presents a model and methodology of faster data manipulation (insert/delete) of mass data rows stored in a big table. In this thesis, it depicts the solution to manipulate large data sets of one table which moves into and out of another logical table with outstanding efficiency compared with traditional transactional way. With this idea, the table structure needs to be redesigned to accommodate and keep data, in other words, the table needs to be "partitioned".
It also covers partitioning strategies which are applied to various scenarios such as the data sliding window scenario, data archiving, and partition consolidation and movement practice.

KEYWORDS

big data, partition table, partitioning, big table, MapReduce

1 INTRODUCTION

1.1 Motivation

The term "Big Data" from software and computer science field is a collection of datasets that grows so large and complex that it becomes difficult to process using traditional relational database management tools or processing applications. In fact, "Big Data" is one of the unstoppable IT trends in the Cloud Computing topic. Nowadays, most relevant applied topics of Big Data are generally derived from MapReduce [1]. However, if we move our focus from the requirement of search/data analysis to real-time data, the shortage of MapReduce is inevitable from its nature of designed architecture. Some disadvantages of MapReduce are, first, good at batched, offline big data processing, but not suitable for real-time transactional jobs. Second, MapReduce uses brute force to compute result instead of indexed data. Third and the most important, for many commercial users, SQL (Structured Query Language)-like data and relational database remains the majority. Once we get back to the traditional usage of RDBMS (Relational Data-Base Management Systems), the common slowness of big data manipulation is unavoidable. A notion came to our minds: Is it possible to insert/delete a huge dataset in no time? Is it possible to leverage technical innovations based on current database platform to minimize maintenance time and maximize efficiency for enterprise level database? How to make these work like a charm automatically? Thus we decide to design a mechanism to fulfill all of the requirements.

1.2 Database Platforms

To better consistently explain the idea thoroughly in this thesis, Microsoft SQL Server (but not limited to this) is chosen as the platform for explain the following samples and the details. The hierarchy of Database Logical Structures is shown in below figure:

ISBN: 978-1 -941968-20-8 ©2015 SDIWC

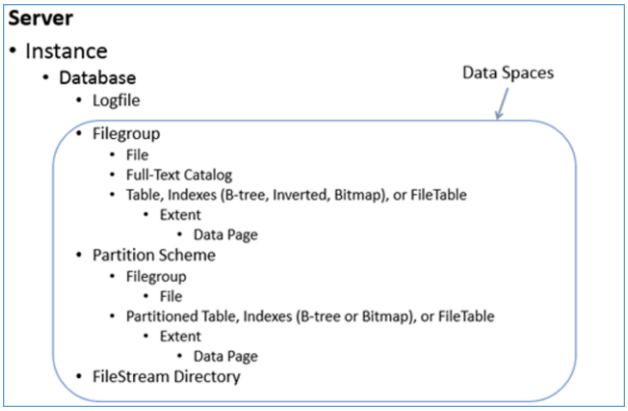

Figure 1: SQL Server Logical Structure.

As described in Figure 1, an instance of the database engine is a copy of the SQL Server executable that runs as an operating system service. Each instance manages several system databases and one or more user databases. Each computer can run multiple instances of the database engine. Applications connect to the instance in order to perform work in a database managed by the instance [2].

SQL Server maps a database over a set of operating-system files which can be categorized in two main types, data and log and with three different file extensions: (.mdf and .ndf for data, and .ldf for log). Data and log information are never mixed in the same file, and individual files are used only by one database. Each database has one set of data spaces, which can be a filegroup, a partition scheme, or a filestream directory [3].

Filegroups are named collections of files and are used to help with data placement and administrative tasks such as backup and restore operations. Partition schemes in the database can map the partitions of a partitioned table or index to filegroups. The filestream directory is used for integrating with unstructured data such as text documents, images, and videos that are often stored outside the database, separate from its structured data. The data of a table is physically stored in one or more files, its logical container is not a file, but a filegroup. A logical container "owns" the logical objects it contains, and a logical object "belongs" to the logical container that holds it. Filegroups own files and tables, therefore tables and files belong to filegroups. Partition schemes own partitioned tables, therefore partitioned tables belong to partition schemes, and if a partition scheme exists, one or more filegroups belong to it. Some examples are:

- A primary filegroup (which always holds the system tables) and a default filegroup must exist. The primary and default filegroups can be the same filegroup or separate filegroups.
- A filegroup may have more than one file.
- A partition scheme can be defined on a single filegroup with one file.
- A partition scheme can be defined on a single filegroup with multiple files.
- A partition scheme can be defined on multiple filegroups, with each filegroup having one or more files.

Figure 2 illustrates the relationships between database's data spaces, filegroups, and partition schemes.

Figure 2: The relationships between data spaces, filegroups, and partition schemes.

1.3 Table and Partition Overview

A table is contained in one or more partitions and each partition contains data rows in either a heap or a clustered index structure. The pages of the heap or clustered index are managed in one or more allocation units, depending on the

column types in the data rows. Figure 3 shows the organization of a table. Table and index pages are contained in one or more partitions. A partition is a user-defined unit of data organization. By default, a table or index has only one partition that contains all the table or index pages. The partition resides in a single filegroup. A table or index with a single partition is equivalent to the organizational structure of tables and indexes in earlier versions of SQL Server [4].

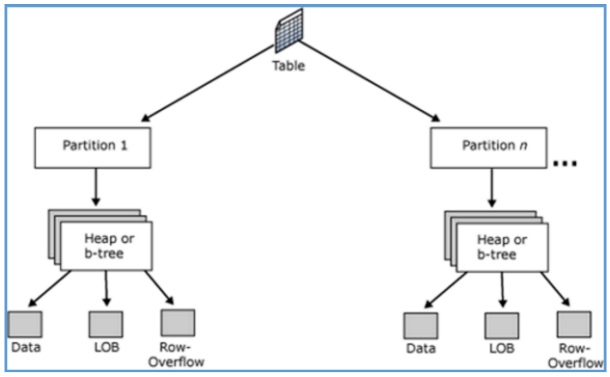

Figure 3: Table and partition organization [4].

When a table or index uses multiple partitions, the data are partitioned, so that groups of rows are mapped into individual partitions, based on a specified column. The partitions can be put on one or more filegroups in the database. The table or index is treated as a single logical entity when queries or updates are performed on the data.

2 METHODOLOGY

2.1 Partition Approach

The two common partitioning techniques applied to the dataset dividend slices are Horizontal Partitioning and Vertical Partitioning. To be simple, no matter what kind of techniques, we can deem them as a "logical view" to a table. For horizontal partitioning views, the table maintains the same structure and columns but is partitioned along the row-level boundaries, for the vertical partitioning views, the vertically partitioned table splits the original table into more than one physical table. Each table has the same number of records and the same primary key, but the other columns are different. These two tables have a 1-to-1 relationship, and this technique might be used to overcome a database engine's limit on the number of columns supported in one table. For example, a financing table might need 1,000 columns that have a 1-to-1 relationship with each other, but the database engine might only support a maximum of 256 columns per table. In that case the logical table could be split (vertically partitioned) into 4 physical tables.

• Horizontal partitioning is a method of dividing rows of data in a table, with different rows belonging to different partitions of that table. We also call it "Row groups".

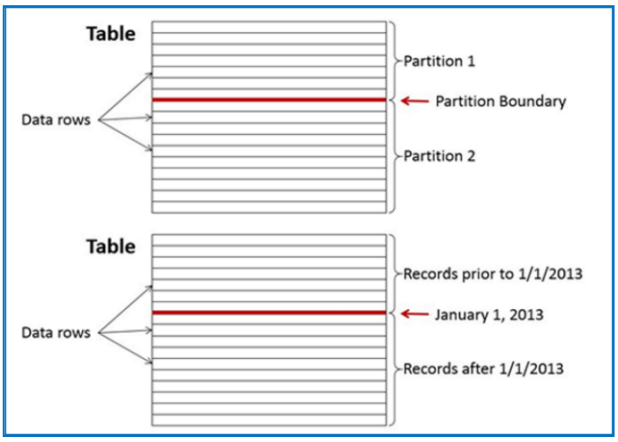

Figure 4: Horizontal partitioning table concept.

• Vertical partitions have subsets of table columns, each subset stores a part of column-wised data. Theoretically, each subset table can be accessed independently, we also call it "Segments".

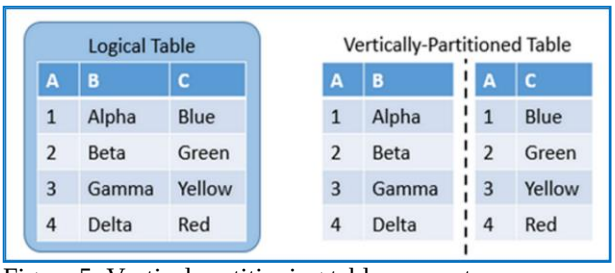

Figure 5: Vertical partitioning table concept.

ISBN: 978-1 -941968-20-8 ©2015 SDIWC

Horizontal and vertical partitions are just like two knives that cut the base table apart from different angles. It is worth pointing out that this "cutting" technique is different from usual SQL statement to filter out rows and columns, on the contrary, the data quantity to the base table remains the same, but had divided into pieces under cover. The smaller pieces can be stored in a very distributed way on the different disks as more as possible, but from developers' point of view, it is nothing but a normal table of no difference.

An example here is the base table (e.g. SalesTable) looks like the following:

OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20130107	111	01	1	6	30.00
20130107	103	04	2	1	17.00
20130107	109	04	2	2	20.00
20130107	103	03	2	1	17.00
20130107	111	05	3	4	20.00
20130108	111	02	1	5	25.00
20130108	102	02	1	1	14.00
20130108	111	03	2	5	25.00
20130108	109	01	1	1	10.00
20130109	111	04	2	4	20.00
20130109	111	04	2	5	25.00
20130109	103	01	1	1	17.00

Figure 6: The sample base table called SalesTable.

After combining with horizontal and vertical techniques, the original bigger table has been divided into 12 smaller tables.

OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20130107	111	01	1	6	30.00
20130107	103	04	2	1	17.00
20130107	109	04	2	2	20.00
20130107	103	03	2	1	17.00
20130107	111	05	3	4	20.00
20130108	111	02	1	5	25.00
OrderDateKey	ProductKey	StoreKey	RegionKey	Quantity	SalesAmount
20130108	102	02	1	1	14.00
20130108	111	03	2	5	25.00
20130108	109	01	1	1	10.00
20130109	111	04	2	4	20.00
20130109	111	04	2	5	25.00
20130109	103	01	1	1	17.00

Figure 7: The concept of how data are stored after partitioned horizontally and vertically.

If we perform a simple query like this:

```
SELECT ProductKey, SUM (SalesAmount)
FROM SalesTable
WHERE OrderDateKey < 20130108
```

The result would look like the following highlighted region with merely 3 small tables:

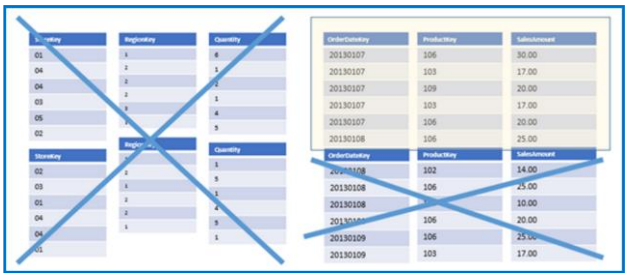

Figure 8: The sample result of vertical partitions filtering.

We can fetch only the needed columns and rows with some filtering condition, and what's even more here is that we do not need to bother any disk I/O (Input/Output) effort on the rest 9 tables (marked with X in the Figure 8), merely the involved disks of the 3 tables required. It is quite different from the original way we perform a select instruction toward the whole table or indexes without partitioning on the same disks. See? It is quite natural to imagine the benefits that the partitioning techniques can bring to us:

- Less storage used and faster storage speed: By means of multiple I/O paths via multiple file groups and disks.
- Smaller, faster backups and restores: By means of backups of changed filegroups only
- Faster inserts and deletes of large data sets By means of partition switching

For the top two points are obviously, but not very clear for the last one? At this point, it is all right, we are going to describe it further in the coming pages soon.

ISBN: 978-1 -941968-20-8 ©2015 SDIWC

Of course, every benefit has its cost that needs to pay, no exception for the partition techniques, they are:

- Partitioned objects require more memory than non-partitioned objects.
- Database administrator must have sufficient knowledge of partitioning.
- Partitioned objects may have some restrictions under certain situations.

Although partitioning requires extra effort when managing very large numbers of records, once the data size scale is big enough, it is surely less total administrative time than trying to manage the records without partitioning.

Here we would like to introduce partitioning methodology of how we boost the performance of data access and manipulation. Whether horizontally or vertically partitioning the dataset is allowed, but operational complexity and the transparency degree also matters. The rule of thumb is: from the database administrator's standpoint, the lower operational complexity the better, on the contrary, from the developer's view, the higher transparency the better. Consider the side impact of vertical partitioning, what happens when the columns are removed from the original base table and placed in its child tables, it is through denormalization, overall data access patterns are more complicated because developers need to know the denormalization relations between each child table very well besides the functional relations of different tables. Combining both horizontal and vertical methods together is even more complicated by multiplying the number of divided slices.

Another consideration is the behavior how we use data. People usually get used to archiving the historical data by "time", "region", or "category". Interestingly these are more similar to horizontally distribution instead of vertical column slices. Thus, here the horizontal partitioning method will be adopted in practice throughout the next implementation.

2.2 Horizontal Partitioning

The three amigos composed of the partitioning definition is by "partition boundary", creating a "partition function" and "partition scheme", followed by assigning to the target table(s). The dependency relationship is shown in Figure 9:

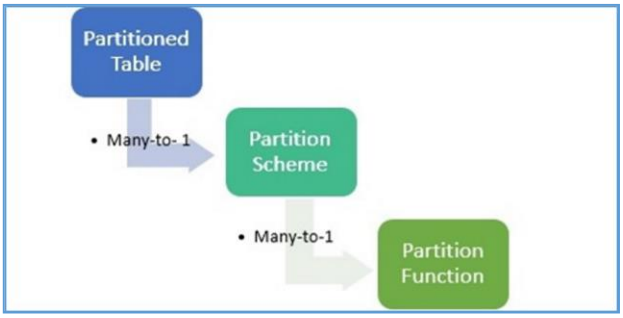

Figure 9: The dependency relationship of partition definitions.

Each component in the relationship must depend on the existence of its predecessor component, and can be used by many of its successors. The partitioned table must be created on a partition scheme, which defines the filegroup storage locations for the partitioned table. This is the biggest difference between non-partitioned tables. Non-partitioned tables also do have a partition, but the number of partition is one, while partitioned table can be separated into thousands of partitions.

More details of some terminologies:

- Partition Boundary: By specifying a set of boundary values from the range of possible values of the input parameter's data type. These boundary values define a number of sub-ranges within the overall range, called partitions.

Each partition:

ISBN: 978-1 -941968-20-8 ©2015 SDIWC

• Partitions are numbered starting with 1.

• Any given possible value within the boundary value range can exist in only one partition.

• Two types (RANGE LEFT/ RANGE RIGHT) of boundary range control which partition a record is stored in when the partitioning value equals the boundary exactly.

Table 1: Example with the boundary value of 1000:

	(Partition #1)	(Partition #2)
RANGE LEFT	<= 1000	> 1000
RANGE RIGHT	< 1000	>= 1000

Say the boundary here is 1,000 and split the overall range as two partitions (#1 and #2), if there is an incoming value of 800 inserted into the partitioned table, which partition number should this value reside in? Quite simply, it is #1 since the incoming value less than the boundary value of 1,000. What if the incoming value is exactly the same as the boundary value of 1,000?

The answer depends on the partition range direction we use. If we use RANGE LEFT, once the incoming value is right equal to the boundary value, then it would be assigned to reside in the "LEFT" partition, which the partition number is #1 here. Meanwhile, if we use RANGE RIGHT, the coming value of 1,000 would fall into the "RIGHT" partition, where the partition number is #2. If we have multiple boundary values, they are like:

- RANGE LEFT and values of (0, 10, 20):

  Partition 1: values from the data type minimum to values <= 0
  Partition 2: values > 0 and values <= 10
  Partition 3: values > 10 and values <= 20
  Partition 4: values > 20 and values <= data type maximum

- RANGE RIGHT and values of (0, 10, 20):

  Partition 1: values from the data type minimum to values < 0
  Partition 2: values >= 0 and values < 10
  Partition 3: values >= 10 and values < 20
  Partition 4: values >= 20 and values <= data type maximum

2.3 Metadata-Only Manipulations

One fact in the world is, moving a large data set could never be faster than the small one. Just like under the same driving mode, traveling 10,000 miles with less gasoline usage than 5 miles only happens in the Arabian Nights.

For instance, we have two tables (Tables A & B), a common manipulation is to move "2008 Data" from Table A to B. The puzzle is that the size of "2008 Data" is up to 10 TB.

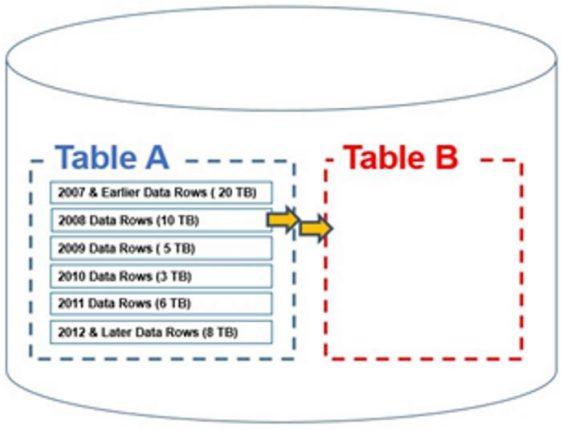

Figure 10: Traditional SQL operation of data manipulation.

By using traditional method, the pseudo SQL statement would be two steps as followed:

```
insert Table B
select * from Table A
where data_period = 2008 Data;
delete Table A
where data_period = 2008 Data
```

In the first step, as it says, select the specified period data set and insert into to the target table (Table B). Due to the data size is up to 10 TB, this operation comes to no surprise that would require several days to most of slow disks, several hours taken even for very high-end storage disk arrays. It merely counted the insertion effort, for any database engine, since it needs to record every action for possible rollback request later, extra disk I/O operation cost charges due to the write-in to log files, and hence, the real duration and the space needed could be doubled.

Unfortunately, it is just the beginning. The second step is another disaster to the whole system because we need to get rid of the data set "2008 Data" from the source table (Table A). The ensuing disk I/O flood will cost another several days, or another several hours at least. After the torture like this, if you dared to touch the disks surface and feel the temperature, believe me, you must regret.

The cost does not yet include the derived network bandwidth, the processor power, and the most of all, the precious waiting time.

Thinking differently is the only way out! The tricky way is: DO NOT REALLY move data sets literally, but move the "metadata" behind them, instead. By changing the link definition of big data set and the partition number where it lives in, it is possible to load or remove data from the partitioned table almost immediately regardless of how big they are.

Let's leverage the technique stated earlier. Partition Table A into six areas so the data within would be divided according to the date boundaries. Table B is a blank table with an empty partition.

Imagine the following concept:

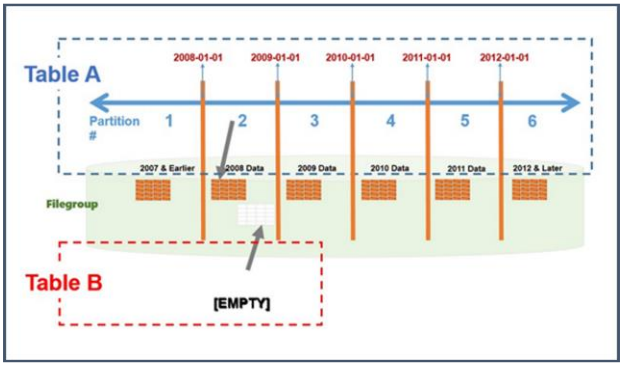

Figure 11: Table and partition metadata movement concept (Before switch out).

After our architecture redesigned with table partitioning techniques, each area of data set has its own link to the logical partition number with filegroup. The default partition of Table B shares the same filegroup with the partition #2 of Table A. By exchanging the link of the partition "2008 Data" metadata definition, which is the concept called "Switch out", the fast performance is no longer a dream. The magic pseudo statement as follows:

```
Alter Table A
SWITCH PARTITION 2
TO B
```

Guess what, this short statement will do the tricks and finish the job in seconds. After the "Switch out" operation, the link definition of data set "2008 Data" has been changed from Table A to B. For users' point of view, it is like the data set "2008 Data" has been deleted from Table A and loaded into Table B.

All of these happens in a twinkle. This is because the "Switch" operation can avoid any real data movement. The new metadata definition looks like the followed figure 12:

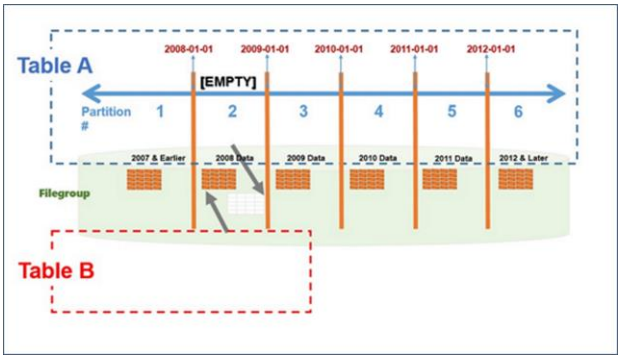

Figure 12: Table and partition metadata movement concept (After switch out).

With the same idea, we can perform "Switch in" operation to pull in the data set we want to the table or specific partition.

2.4 Sliding the Partitions

With the basis of clear interpretation in previous sections, now is a good timing to design a partitioning strategy that leverages partition switching and the performance advantage it brings to manage the big tables in business operations.

The strategy is by sliding the partitions to fade out old data and bring in new data into the partitioned table periodically, we call this method "Sliding window", as it is described by analogy with the network packet terms in TCP/IP (Transmission Control Protocol / Internet Protocol) world. The concept procedures consist of the following ten steps & pseudo codes:

1. Design a partition function based on the proper column of the source table(s).

```
CREATE PARTITION FUNCTION
pf_Annually_Right(datetime)
AS RANGE Right FOR VALUES
(
  '2008-01-01', '2009-01-01',
  '2010-01-01',
  '2011-01-01', '2012-01-01')
```

For the values, though we used the "fixed" dates here, to be more flexible in the real world systems, we strongly suggest to replace the fixed values with corresponding **dynamic values and format** from the real-time system date or some data calculations result with incrementally or diminishingly method.

For example:

```
SUBSTRING(CONVERT(varchar,DATEADD(
dd,+7,getdate()),121),0,12)
```

2. Plan filegroup(s) distribution and design the partition scheme to combine the partition function and the filegroup definition.

```
CREATE PARTITION SCHEME ps_
pf_Annually AS PARTITION
pfDaily_Right ALL TO ([PRIMARY])
```

To be more performance-oriented thinking, plan more than one filegroups and assign to different physical disk arrays will get further parallel disk I/O benefits. Here for demonstration, we put all in the PRIMARY group.

3. Use the current existing source table or design the new source table to accommodate the data pertaining to the partitioning structure.

```
CREATE TABLE [sourceTableName]
(
  columnDate, column2, column3....
)

ON ps_pf_Annually([column_Date])
```

4. Design a staging table (temporary archive table) with identical columns/indexes definitions as the source table.

```
CREATE TABLE [tablename_Staging]
(
  columnDate, column2, column3....
)
```

5. Move the oldest data partition from the source partitioned table to the archive table by “Switch out” command, now the oldest data would reside in the staging archive table, archive it in the background to other tables (will not lock/affect the source table's transaction and performance) or truncate it directly (in seconds, no matter how many rows or how big size it is).

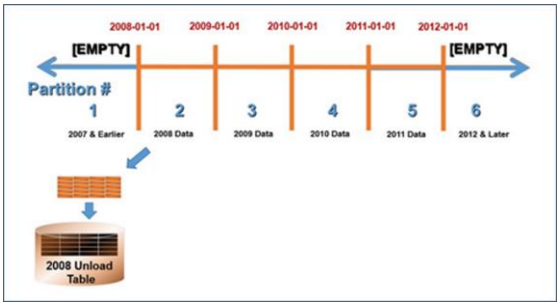

Figure 13: Switch out the old partition to staging table.

```
ALTER TABLE [sourceTableName]
SWITCH PARTITION 2
TO[tablename_Staging];

TRUNCATE TABLE [tablename_Staging]
```

6. Get rid of the boundary value of the oldest partition by “Merge” command.

Figure 14: Before merging the partitions, the oldest boundary value is 2008-01-01.

```
ALTER PARTITION FUNCTION
pf_Annually_Right()

MERGE RANGE (‘2008-01-01’)
```

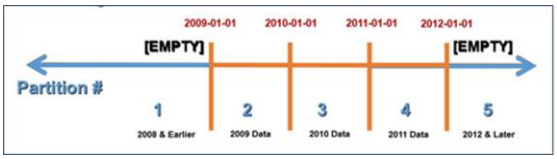

Figure 15: After merging the partitions, the oldest boundary value becomes 2009-01-01.

7. Specify the NEXT USED filegroup for the incoming new partition.

```
ALTER PARTITION SCHEME ps_
pf_Annually
NEXT USED[PRIMARY]
```

8. Create a new boundary value for the new partition at the newest end partition of the table, remember by “Split” command, this will cut the leading end partition into two empty partitions.

```
ALTER PARTITION FUNCTION
pf_Annually_Right()
SPLIT RANGE (‘2012-01-01’)
```

9. (Optional) Get the data from an existing staging table with identical columns/indexes definitions as the target partitioned table by “Switch in” command to the proper partition location.

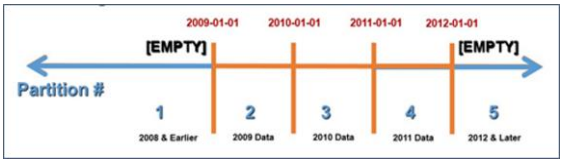

Figure 16: Before extending new partition, the maximum boundary value is 2011-01-01.

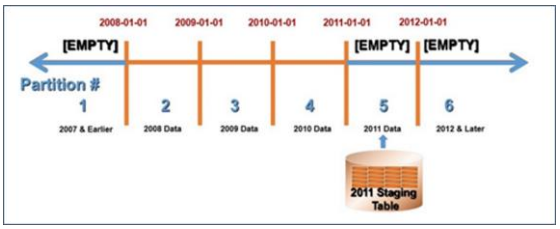

Figure 17: After splitting partitions, the maximum boundary value becomes 2012-01-01, and load the data of the staging table into Partition 5 of the target partitioned table.

Notice: Keep this new partition empty, never put into both the leading ends. Make sure the indexes of the existing staging table are rebuilt/refreshed before switching into the target partitioned table.

10. The staging table now is empty and the source table partition has been refreshed.

In the above steps, we developed a strategy to slide the partitioning structure by cutting out the oldest partition and extend an extra new partition, during the process, not only refreshing the structure, but also deleting the historic data and loading the latest data in seconds. To further automate this process, simply put these steps into the schedule jobs with proper frequency, this idea is totally different from traditional SQL DML (Data Manipulation Language) and extremely efficient in dealing with big data for any enterprise level application.

3 CONCLUSION

By leveraging the "Divide & Conquer" theorem, big data can be partitioned into separate logical units, by changing the metadata relationship, each big dataset unit can "roam around" different tables quickly under moving requirements, and this really raises the performance and the usage flexibility of application possibility in the age of data/information explosion.

Partitioning also has its disadvantages and limitations. The framework has the tight dependency on the database engine. The same scenario with good performance and feasibility in one database platform may not work that well in the others due to the version supportability and internal engine behavior differences. A little bit complex designing requirements before setting out on a journey is also the showstopper widely used in daily data operations. However, we still can reap more benefits from using the idea than its limitations. As the maturity of the framework goes by, the true influence of partitioning is worth expecting.

REFERENCES

[1] J. Dean and S. Ghemawat (December, 2004). MapReduce: Simplified Data Processing on Large Clusters, Paper presented at the meeting of Proceedings of Symposium on Operating Systems Design and Implementation, San Francisco, CA.

[2] Microsoft, "Database Engine Instances," http://technet.microsoft.com/en-us/library/hh231298.aspx, 2013.

[3] Microsoft, "Filestream Overview," http://technet.microsoft.com/en-us/library/bb933993(v=sql.105).aspx, 2013

[4] Microsoft, "Table and Index Organization," http://technet.microsoft.com/en-us/library/ms189051(v=sql.105).aspx, 2013