Performance Evaluation in SQL Server

Simonely Hidalgo Lorenzo
Master in Computer Science
Dr. Nelliud Torres Batista
Electrical & Computer Engineering and Computer Science Department
Polytechnic University of Puerto Rico

Abstract *— A database table with millions of rows could take a long time to retrieve, insert, update, and delete data. The evaluation in this paper consists of create indexes, apply normalization process, and create surrogate key to improve the performance of retrieving data. Explain the differences between multiple types of indexes and which scenarios we can use for each of them. To evaluate the improvements, one table was created in SQL server with 45 million rows. The analysis describes the resources and I/O statistics used by Microsoft SQL Server Management Studio. For non-indexed tables is categorized sequentially searched and indexed table that are compared as B-tree index. Finally, the analysis was performed for normalization and composite key.*

Key Terms *— Microsoft SQL Server Management Studio, optimization, performance, SQL statistics*

INTRODUCTION

In multiple Database books the tables indexes are compared with the index card in a traditional library where we can see a lot shelve with books. Exist different way to find a book in the library for example by author or title where each book has a number assigned in the index card that belong the same number in the shelve books, that is an analogy between library index and database index to do easier how retrieve the data quickly and efficiently. But indexes will affect another transaction as Insert, Update and Delete. Those different scenarios will be discussed and evaluated in this paper.

Indexes are the means to providing an efficient access path between the user and the data, by providing this access path, the user can ask for data from the database and the database will know where to go to retrieve the data [1]. Creating and maintaining an appropriate index file is a major issue in database management systems, by using an appropriate indexing mechanism, the query processing algorithms may not have to search the entire database [2]. When the query optimizer uses an index, it searches the index key columns, finds the storage location of the rows needed by the query and extracts the matching rows from that location [3]. Generally, searching the index is much faster than searching the table because unlike a table, an index frequently contains very few columns per row and the rows are in sorted order [3]. In this paper will be discussed a different type of index, how to create the index in SQL server, the benefits of each index and demonstration of the performance improvement using index. In the normalization section it discusses the rules, and a database was created with denormalized table and normalized table to be used in the performance analysis. Also, the composite key can affect the database performance, creating a surrogate key can bring a benefit but also have a drawback.

COMMON TYPES OF INDEXES

Clustered Indexes

Clustered Index is created with a column or combination of columns that are selected as index and are stored in orders to obtain a fast retrieval of the rows. The columns added to the cluster index should be the most used for the table. Only one cluster index can be created by table since only the physical table is sorted. An index contains keys built from one or more columns in the table or view [3]. These keys are stored in a structure (B-tree) that enables SQL Server to find the row or rows associated with the key values quickly and efficiently [3]. The heap index is the specific order in which the rows are inserted since the table is

created. So the table with cluster index are called cluster table or heap table if has not cluster index.

Non-clustered Indexes

Non cluster index stores the values of the columns selected pointing to the clustered index. Since only one cluster index is allowed per table the data rows are not stored in order. This is nearly identical to how a card catalog works in a library, the order of the books, or the records in the tables, doesn't change, but a shortcut to the data is created based on the other search values [1]. The pointer from an index row in a non-clustered index to a data row is called a row locator [3]. The structure of the row locator depends on whether the data pages are stored in a heap or a clustered table [3]. For a heap, a row locator is a pointer to the row [3]. You can add non key columns to the leaf level of the non-clustered index to by-pass existing index key limits, and execute fully covered, indexed, queries [3].

In table 1, a clustered index was created in column Id so the physical table was rearranged to that specific order. In table 2, the non-clustered index was created for the Item column in ascending order. That lookup table was created using the row locater pointer to the clustered index in table 1.

Table 1
Clustered index

Id	Item	Qty
1	Paper pads	1
2	Pens	3
3	Notebooks	1
4	Books	1
5	Magazines	3

Table 2
Nonclustered index

Item	Row Locator
Books	4
Magazines	5
Notebooks	3
Paper pads	1
Pens	2

Column Store Indexes

The column store index was designed for retrieve large range of rows since traditional index are more efficient for small range. This index is recommended for Data warehousing fac tables. In this index all columns of the table are included and use data compression that reduce the storage capacity. In this index all columns are stored by separately instead of store all column of the same row. The benefit of this type of index is that only the columns and rows required for a query need to be read [1]. This index uses column-based data storage and query processing to achieve gains up to 10 times the query performance in your data warehouse over traditional row-oriented storage [4]. You can also achieve gains up to 10 times the data compression over the uncompressed data size [4]. Beginning with SQL Server 2016 (13.x) SP1, columnstore indexes enable operational analytics: the ability to run performant real-time analytics on a transactional workload [4]. The reason of column store index is used for data warehouse where data do not change frequently is because during the Insert, Update and Delete statements take longer to create. To reduce fragmentation of the column segments and improve performance, the columnstore index might store some data temporarily into a clustered index called a deltastore and a B-tree list of IDs for deleted rows [5]. Figure 1 shows the columnstore index. To return the correct query results, the clustered columnstore index combines query results from both the columnstore and the deltastore [4].

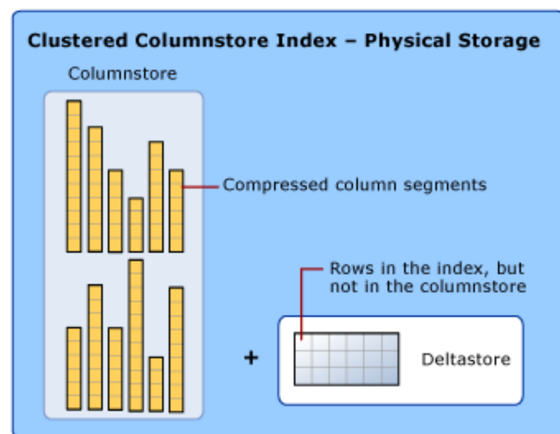

Figure 1
Columnstore Index [4]

If you manage to INSERT a new row, the value will be stored in the deltastore until it reaches the

minimum row group size, then it will be compressed and moved to the Columnstore segment [5].

If you try to DELETE a row, this row will be deleted from the deltastore storage, but it will be marked as deleted on the Columnstore index segment until the index is rebuilt [5].

When performing an UPDATE operation on a row, the row will be deleted from the deltastore storage, and marked as deleted in the Columnstore segment and the new value will be inserted to the deltastore [6].

XML Indexes

XML index are created for the columns with xml data type storing the tags, values, and paths of the column. That help for the query performance but is very costly when the data change and for maintenance. Building the index avoids parsing the whole data at run time and benefits index lookups for efficient query processing [6]. The XML index have two categories: primary and secondary XML index. The first index on the xml type column must be the primary XML index [6]. Using the primary XML index, the following types of secondary indexes are supported: PATH, VALUE, and PROPERTY [6]. Depending on the type of queries, these secondary indexes might help improve query performance [6].

INDEX VARIATIONS

Primary Key

A primary key is a unique value that identify the instance in the table and typically composed of one column but could be composed by multiple columns. When the primary key is assigned in the database table then the primary key index is created by default as cluster index but if a cluster index is already created then will be non-clustered.

Unique Index

A unique index can consist of one or multiples columns. Like primary key this allow only a unique value in the record. At difference of primary key this index accepts null value, only one null value for row is accepted in the unique index.

Filtered Indexes

This index is used in non-clustered index and non-unique. The index limits the amount rows filtering the relevant values. Reducing the number of rows in the index improve the query performance, reduce the storage cost, and reduce the maintenance of the index.

Partitioned Indexes

The table is partitioned horizontally by the column and range selected. The data in this index is stored separated in each partition and can be spread more than one filegroup but the index it still a single logical object. This index is useful for tables with billions of records. During the data retrieve the scan is faster because make the search in specific data subset. It can be used in clustered or non-clustered index. SQL Server supports up to 15,000 partitions by default [7]. Table 3 shows how a table partitioned horizontally by years looks.

Table 3
Partition index

	Col 1	Col 2	Date
Partition 1 year 1			
Partition 2 year 2			
Partition 3 year 3			

CREATING INDEXES

Indexes can be created by using SQL Server Management Studio or Transact-SQL. Figure 2 shows the Transact-SQL command to create index and their syntax options discussed in the type of index and variants sections of this paper.

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
    ON <object> ( column [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
    [ WHERE <filter_predicate> ]
    [ WITH ( <relational_index_option> [ ,...n ] ) ]
    [ ON { partition_scheme_name ( column_name )
         | filegroup_name
         | default
         }
    ]
    [ FILESTREAM_ON { filestream_filegroup_name
    | partition_scheme_name | "NULL" } ]

[ ; ]

<object> ::=
{ database_name.schema_name.table_or_view_name
    | schema_name.table_or_view_name
    | table_or_view_name }

<relational_index_option> ::=
{
    PAD_INDEX = { ON | OFF }
  | FILLFACTOR = fillfactor
  | SORT_IN_TEMPDB = { ON | OFF }
  | IGNORE_DUP_KEY = { ON | OFF }
  | STATISTICS_NORECOMPUTE = { ON | OFF }
  | STATISTICS_INCREMENTAL = { ON | OFF }
  | DROP_EXISTING = { ON | OFF }
  | ONLINE = { ON | OFF }
  | RESUMABLE = { ON | OFF }
  | MAX_DURATION = <time> [MINUTES]
  | ALLOW_ROW_LOCKS = { ON | OFF }
  | ALLOW_PAGE_LOCKS = { ON | OFF }
  | OPTIMIZE_FOR_SEQUENTIAL_KEY = { ON | OFF }
  | MAXDOP = max_degree_of_parallelism
  | DATA_COMPRESSION = { NONE | ROW | PAGE }
     [ ON PARTITIONS ( { <partition_number_expression> | <range> }
     [ , ...n ] ) ]
}

<filter_predicate> ::=
    <conjunct> [ AND <conjunct> ]

<conjunct> ::=
    <disjunct> | <comparison>

<disjunct> ::=
        column_name IN (constant ,...n)

<comparison> ::=
        column_name <comparison_op> constant

<comparison_op> ::=
    { IS | IS NOT | = | <> | != | > | >= | !> | < | <= | !< }
```

Figure 2
Syntax Options to Create Index in SQL Server

PERFORMANCE INDEX EVALUATION AND RESULTS

To demonstrate the index improvements one table was created named PRODUCT with 55 million rows in SQL server, one of the columns created is Id and has integer values. The results were obtained using "Estimate Execution Plan" tool from SQL server found in the Query menu of Microsoft SQL Server Management Studio, as figure 3 shows.

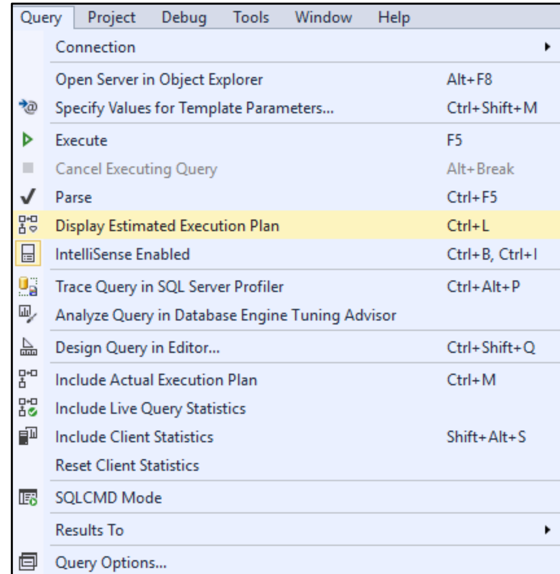

Figure 3
Query Menu of Microsoft SQL Server Management Studio

The results of the SQL statement where the index has not been created yet is showed in the figure 4. In this scenario the search will do a sequential table scan until reach the id desired, also the I/O cost for this execution is 1414.37 and CPU cost is 10.2043. In figure 5 shows the SQL IO statistics where the scan count was 13 and the logical reads was 1,923,853. The total execution time was 9609 ms.

Comparing this result with a clustered index in the figure 6. We can prove the improvement using the same SQL query. The results show the I/O cost of 0.003125 and the CPU Cost 0.0001581. In the figure 7 the elapsed time was 72 ms, the Scan count was 1 and the logical read was 4.

The indexes make a search faster by using a B-Tree structure or a Balanced Tree structure. In the B-Tree structure data is divided into root nodes, non-leaf nodes and leaf nodes. The algorithm used in B-Tree searching is a binary tree search and goes with recursion. The time of complexity is O(log n). The objective is reducing the number of disk access.

Table Scan

Scan rows from a table.

Physical Operation	Table Scan
Logical Operation	Table Scan
Estimated Execution Mode	Row
Storage	RowStore
Estimated I/O Cost	1414.37
Estimated Operator Cost	1424.57 (100%)
Estimated CPU Cost	10.2043
Estimated Subtree Cost	1424.57
Estimated Number of Executions	1
Estimated Number of Rows	1.00004
Estimated Number of Rows to be Read	55659500
Estimated Row Size	15 B
Ordered	False
Node ID	1

Predicate
[Development].[dbo].[PRODUCT].[Id]=(1203165434)
Object
[Development].[dbo].[PRODUCT]
Output List
[Development].[dbo].[PRODUCT].Id

Figure 4
IO cost without index

```
Table 'PRODUCT'. Scan count 13, logical reads 1923853,

 SQL Server Execution Times:
   CPU time = 9609 ms,  elapsed time = 9027 ms.
```

Figure 5
SQL Time Statistics

Clustered Index Seek (Clustered)

Scanning a particular range of rows from a clustered index.

Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Estimated Execution Mode	Row
Storage	RowStore
Estimated Operator Cost	0.0032831 (100%)
Estimated I/O Cost	0.003125
Estimated Subtree Cost	0.0032831
Estimated CPU Cost	0.0001581
Estimated Number of Executions	1
Estimated Number of Rows	1
Estimated Number of Rows to be Read	1
Estimated Row Size	15 B
Ordered	True
Node ID	0

Object
[Development].[dbo].[PRODUCT].[ClusteredIndex-Id]
Output List
[Development].[dbo].[PRODUCT].Id
Seek Predicates
Seek Keys[1]: Prefix: [Development].[dbo].[PRODUCT].Id = Scalar Operator(CONVERT_IMPLICIT(bigint,[@1],0))

Figure 6
IO Cost with Index

```
Table 'PRODUCT'. Scan count 1, logical reads 4,

(1 row affected)

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 72 ms.
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.

 SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
```

Figure 7
SQL Statistics for Clustered Index

Table 2 contains the time statistics results for each SQL statement (SELECT, INSERT, DELETE and, UPDATE). The SELECT statement was faster with index than non-indexed and for INSERT, DELETE and UPDATE took more time. The increase time is because need to do extra work in re-arrangement of the indexes.

Table 2
SQL Time Statistics

SQL Statement	Number of Rows	Execution Time (ms)	
		No Index	Index
SELECT	1	90277	72
INSERT	9763855	39864	62894
DELETE	9763855	47616	83215
UPDATE	9763855	45486	62184

The two values from I/O statistics results that we are using to measure the performance are scan counts and logical reads. Scan count is number of seeks or scans started after reaching the leaf level in any direction to retrieve all the values to construct the final dataset for the output. [8]. The logical reads are number of pages read from the data cache. To obtain those values in the output it needs to be turned on (Set Statistics IO on) before to execute of the SQL Statement. Table 3 contains the result of scan counts and table 4, the logical reads values.

Table 3
SQL I/O Statistics (Scan Counts)

SQL Statement	Number of Rows	Scan Counts	
		Not Index	Indexed
SELECT	1	13	1
INSERT	9,763,855	0	0
DELETE	9,763,855	13	1
UPDATE	9,763,855	13	1

Table 4
SQL I/O Statistics (Logical Reads)

SQL Statement	Number of Rows	Logical Reads	
		Not Index	Indexed
SELECT	8,451,155	1,923,853	4
INSERT	8,451,276	10,109,477	4,865,287
DELETE	8,451,397	11,687,708	9,483,339
UPDATE	8,451,397	11,833,877	4,088,986

In the query performed with index and without index the result shows that with index it faster to identify the record because use less logical reads but when it is necessary to reorganize the index took more time to complete.

NORMALIZATION

The database normalization has multiples benefits. When the normalization rules are applied the benefits are, avoid anomalies in the data, reduce large table into smaller tables avoiding data redundancy, maintain the data integrity reducing multiples entries and updates, the Insert and Update operations will be more quickly. With less data then maximize the storage capacity. The drawback is that with multiples tables then require more joining and complicate queries, also impact the data search performance where will we see it in the results section. Those rules are called first normal form (1NF), second normal form (2NF) and third normal form (3NF).

First normal form (1NF) is now considered to be part of the formal definition of a relation in the basic (flat) relational model; historically, it was defined to disallow multivalued attributes, composite attributes, and their combinations [9]. It states that the domain of an attribute must include only atomic (simple, indivisible) values and that the value of any attribute in a tuple must be a single value from the domain of that attribute [9]. Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a single tuple. In other words, 1NF disallows relations within relations or relations as attribute values within tuples [9]. The only attribute values permitted by 1NF are single atomic (or indivisible) values [9].

Second normal form (2NF) is based on the concept of full functional dependency [9]. Create separate tables for sets of values that apply to multiple records [9]. Relate these tables with a foreign key [9]. A functional dependency $X \rightarrow Y$ is a full functional dependency if removal of any attribute A from X means that the dependency does not hold any more; that is, for any attribute $A \varepsilon X$, $(X - \{A\})$ does not functionally determine Y [9]. A functional dependency $X \rightarrow Y$ is a partial dependency if some attribute $A \varepsilon X$ can be removed from X and the dependency still holds; that is, for some $A \varepsilon X$, $(X - \{A\}) \rightarrow Y$ [9]. A relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on the primary key of R [10x].

Third normal form (3NF) is based on the concept of transitive dependency [9]. A functional dependency $X \rightarrow Y$ in a relation schema R is a transitive dependency if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R, ^10 and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold [9]. Values in a record that are not part of that record's key do not belong in the table. In general, anytime the contents of a group of fields may apply to more than a single record in the table, consider placing those fields in a separate table [10].

Normalization Results

Figure 8 shows one table denormalized, after applying the normalization rules four additional tables were created to decomposing into smaller relational schema with desirable properties as shown in figure 9. I will use both scenarios to evaluate the performance of data collection and space used. These entities belong to a rental company with different branch and each item has its own product number associated to one branch. During the normalization as well as of the of minimizes the risk of update anomalies also reduce the physical storage used, to calculate the space used I executed the system stored procedures “sp_spaceused” provided by SQL server and the results are shown in table 4, where we can see the space reduction, for denormalized table named “INVENTORY_D” the

consume was 5.3 GB and the sum of the four tables created during the normalization was 3.5 GB so the reduction in space was 2.8 GB for a 33%.

INVENTORY_D *

ID
BranchName
LastName
EName
Membership
CProductNumber
CType
CName
CPrice
Country
City
CSaleDate

Figure 8
Denormalized Table

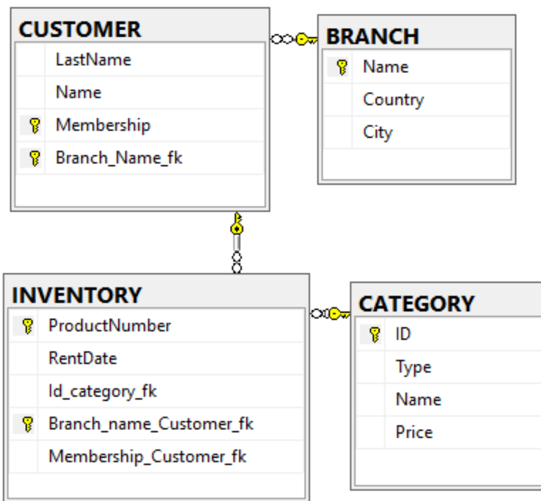

Figure 9
Normalized Table

Table 4
Space Used During Normalization

Table Name	Total Rows	Data Space (kb)	Total Space Used (kb)	Sum of tables used (Gb)
Category	3000791	206376	206384	3.5
Inventory	44535813	3321736	3321744	
Branch	15	8	16	
Customer	391795	20152	20168	
Inventory_D	4453583	5326576	5326592	5.3

To evaluate the resources consumed in normalized and denormalized scenarios I used "Display Estimate Execution Plan" tool provided by SQL Server. The results for an "SELECT" statement query is shown in figure 10, where the table is denormalized and its I/O Cost is 493.204 and CPU cost 48.9896. Versus an I/O cost of 307.571 and CPU cost of 8.1575 for a normalized table shown in the figure 11. The "SELECT" statement was faster for denormalization tables but consume more system resources. The results for the time statistics provided by SQL are found in table 5. The I/O statistic results that measure the query performance for SQL are shown in tables 6 and 7 where we can see the scan count and the logical read that SQL used during the query execution and shown why denormalized tables was faster than normalized. But for the Insert, Update and Delete in the normalized tables are faster than denormalized.

Table Scan
Scan rows from a table.

Physical Operation	Table Scan
Logical Operation	Table Scan
Actual Execution Mode	Row
Estimated Execution Mode	Row
Storage	RowStore
Number of Rows Read	44535813
Actual Number of Rows	8451155
Actual Number of Batches	0
Estimated I/O Cost	493.204
Estimated Operator Cost	542.193 (100%)
Estimated CPU Cost	48.9896
Estimated Subtree Cost	542.193
Number of Executions	1
Estimated Number of Executions	1
Estimated Number of Rows	8782690
Estimated Number of Rows to be Read	44535800
Estimated Row Size	709 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0

Predicate
[Development].[dbo].[INVENTORY_D1].[BranchName]=[@1]
Object
[Development].[dbo].[INVENTORY_D1]
Output List
[Development].[dbo].[INVENTORY_D1].BranchName,
[Development].[dbo].[INVENTORY_D1].LastName,
[Development].[dbo].[INVENTORY_D1].EName,
[Development].[dbo].[INVENTORY_D1].Membership,
[Development].[dbo].[INVENTORY_D1].CProductNumber,

Figure 10
Resources Used in Denormalized Table

Table Scan (Heap)
Scan rows from a table.

Physical Operation	Table Scan
Logical Operation	Table Scan
Actual Execution Mode	Batch
Estimated Execution Mode	Batch
Storage	RowStore
Number of Rows Read	44535813
Actual Number of Rows	8451155
Actual Number of Batches	9395
Estimated I/O Cost	307.571
Estimated Operator Cost	315.728 (88%)
Estimated CPU Cost	8.1575
Estimated Subtree Cost	315.728
Number of Executions	12
Estimated Number of Executions	1
Estimated Number of Rows	851030
Estimated Number of Rows to be Read	44535800
Estimated Row Size	97 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	5

Predicate
[CECS1].[dbo].[INVENTORY].[Branch_name_Customer_fk] as [i].[Branch_name_Customer_fk]='atlanta'
Object
[CECS1].[dbo].[INVENTORY] [i]
Output List
[CECS1].[dbo].[INVENTORY].ProductNumber, [CECS1].[dbo].[INVENTORY].SaleDate, [CECS1].[dbo].[INVENTORY].Id_category_fk, [CECS1].[dbo].[INVENTORY].Branch_name_Customer_fk, [CECS1].[dbo].[INVENTORY].Membership_Customer_fk

Figure 11
Resources Used in Normalized Table.

Table 5
Normalization Time Statistics

SQL Statement	Number of Rows	Execution Time (ms)	
		Denormalized	Normalized
SELECT	8,451,155	104,610	115,720
INSERT	8,451,276	33,520	27,315
DELETE	8,451,397	19,350	14,250

Table 6
Normalization SQL I/O Statistics (Scan Counts)

SQL Statement	Number of Rows	Scan Counts	
		Denormalized	Normalized
SELECT	8,451,155	1	13
INSERT	8,451,276	13	13
DELETE	8,451,397	13	13

Table 7
Normalization SQL I/O Statistics (Logical Reads)

SQL Statement	Number of Rows	Logical Reads	
		Denormalized	Normalized
SELECT	8,451,155	665,822	1,028,578
INSERT	8,451,276	9,268,718	8,940,630
DELETE	8,451,397	8,940,749	9,232,663

COMPOSITIVE KEY

The composite key is a combinations of multiples column that identified the row as primary key and is also known as natural or real key. This type of larger key impact the performance during the Select, Insert, Delete and Update. The surrogate key is not natural key that is auto generated by the system preferably integer value to avoid a composite key. That improve the performance since use a smaller key and help in the index maintenance because the value increment sequentially. This can be added as a new attribute or can be created in other entities used as lookup table. The drawbacks are increase the storage capacity since a new attribute is created, since surrogate key values are just auto-generated values with no business meaning it's hard to tell if someone took production data and loaded it into a test environment [11], Extra column(s)/index for surrogate key will require extra IO when insert/update data [11].

CONCLUSION

We saw the Indexing, normalization and composite key and their behavior in the database performance. Each of one have their benefits and drawbacks. we need to be clear of how the data will be consumed if it for transactional or analytical purpose. For example, the improvement was noticed significantly when the cluster index was created in term of execution time for one simple query executed was 9 seconds faster, but inserting was slower. So probably some of cons doesn't apply to your application. In the data warehouse the normalization makes the data retrieve slow but in a transactional system where need to maintain the integrity of the data, make insert, delete, and update then the normalization is beneficial. Finally,

regarding a surrogate key makes sense that when you perform a search of a primary key that have a long string then will be faster when is performed in a small integer value. Using the SQL tools provided by Microsoft SQL Server Management Studio was useful to evaluate which SQL Statements have better performance according with the applications or business needs.

References

[1] J. Strate and T. Krueger, *Expert Performance Indexing for SQL Server 2012*. New York: Apress, 2012. [Online]. Available: https://www.apress.com/gp/book/9781430237419

[2] B. Thuraisingham, *Database and Applications Security*, Boca Raton, FL: Taylor & Francis Group, 2005.

[3] Microsoft, “Clustered and nonclustered indexes described,” December 14, 2020 [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver15

[4] Microsoft, “Columnstore indexes: overview,” May 26, 2021 [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-overview?view=sql-server-ver15

[5] A. Yaseen, “SQL Server 2014 Columnstore index,” SQLShack, April 29, 2016 [Online]. Available: https://www.sqlshack.com/sql-server-2014-columnstore-index

[6] Microsoft, “XML Indexes (SQL Server),” June 23, 2021 [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/xml/xml-indexes-sql-server?view=sql-server-ver15

[7] Microsoft, “Partitioned tables and indexes,” January 28, 2021 [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/partitions/partitioned-tables-and-indexes?view=sql-server-ver15

[8] Microsoft, “Set statistics IO (Transact SQL),” January 29, 2021 [Online]. Available: https://docs.microsoft.com/en-us/sql/t-sql/statements/set-statistics-io-transact-sql?view=sql-server-ver15

[9] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*. London: Pearson, 2015.

[10] Microsoft, “Description of the database normalization basics,” May 17, 2021 [Online]. Available: https://docs.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description

[11] B. Snaidero, “Surrogate key vs natural key differences and when to use in SQL server,” MSSQLTips, April 16, 2018 [Online]. Available: https://www.mssqltips.com/sqlservertip/5431/surrogate-key-vs-natural-key-differences-and-when-to-use-in-sql-server