In this lab class we will approach the following topics:

1. Schema Tuning

Schema design is a very important aspect in what concerns performance tuning, as one should consider the trade-offs among normalization and denormalization.

In a general sense, the objective of **normalization** is to decide which attributes end up in which tables, thus minimizing update anomalies and maximizing data accessibility. Although normalization is generally regarded as a rule in relational database design, there are still times when database designers may turn to **denormalizing a database in order to enhance performance** and ease of use. A full normalization results in a number of logically separate relations that, in turn, result in physically separate stored files. Join processing over normalized tables requires an additional amount of system resources.

Denormalization can be described as a process for reducing the degree of normalization, with the aim of improving query processing performance. One of the main purposes of denormalization is to reduce the number of joins needed to derive a query answer, by reducing the number of physical tables that must be accessed to retrieve the desired data. Denormalization may **boost query speed**, but also **degrade data integrity due to redundancies**. The denormalization process can easily lead to data duplication that, in turn, leads to update anomaly issues requiring increased database storage requirements.

Common schema optimization (or tuning) techniques revolve around collapsing tables, splitting tables, adding redundant columns, or adding derived columns.

1.1. Denormalization by Collapsing Tables

One of the most common and secure denormalization techniques consists of **collapsing one-to-one relationships** *(see Figure 1)*. This situation occurs when for each row of relation TB1, there is only one related row in relation TB2. While the key attributes of the two relations may or may not be the same, their equal participation in a relationship indicates that they can be treated as a

single unit. There are several advantages of this technique in the form of reduced number of foreign keys on tables, reduced number of indexes (since most indexes are created based on primary/foreign keys), reduced storage space, and reduced amount of time for data modification.

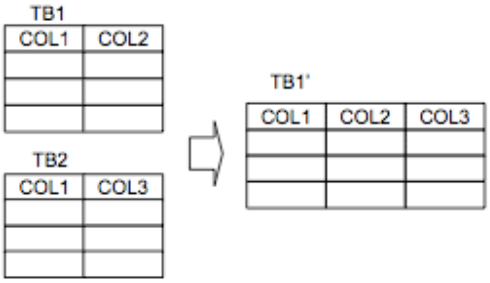

Figure 1: Collapsing one-to-one relationships.

A many-to-many relationship can also be a candidate for table collapsing. The typical many-to-many relationship is represented in the physical database structure by three tables: one table for each of two primary entities and another table for cross-referencing them. These three tables can be merged into two if one of the entities has little data apart from its primary key (i.e., there are not many functional dependencies with the primary key). Such an entity could be merged into the cross-reference table by duplicating the attribute data.

There is, however, a drawback of this second approach. Update anomalies may occur when the merged entity has instances that do not have any corresponding entries in the cross-reference table. Collapsing the tables in both one-to-one and one-to-many eliminates the join, but there may be a significant loss at the abstract level because there is no conceptual separation of the data. In general, collapsing tables involved in many-to-many relationship has a significant number of problems when compared to other denormalization approaches.

1.2. Denormalization by Adding Redundant Columns

Adding redundant columns (i.e., vertical anti-partitioning) can be used when a column from one table is being accessed in conjunction with a column from another table. If this occurs frequently, it may be necessary to combine the data and carry them as redundant.

Reference data, which relates sets of codes in one table and descriptions in another, is a natural example where redundancy can pay off. Instead of obtaining a description of a code via a join, a typical strategy is to duplicate the descriptive attribute in the table where the code is stored. The result is a redundant attribute in that table that is functionally independent of the primary key. Figure 2 provides an illustration.

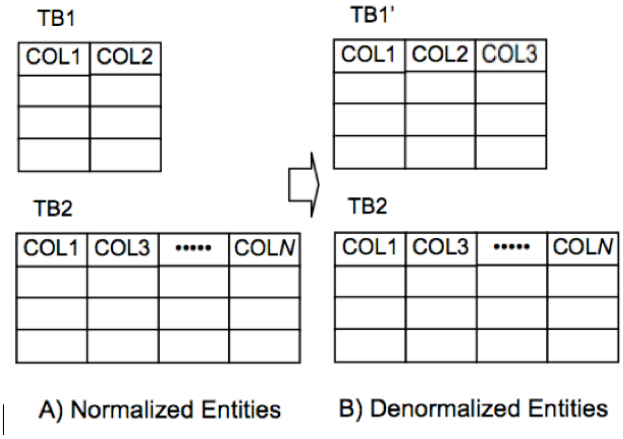

Figure 2: Denormalization by adding redundant columns.

1.3. Denormalization by Adding Derived Attributes

Applications often require the frequent use of data calculated from information stored in the database. On large volumes of data, even simple calculations can consume a substantial processing time.

Storing derived data in the database can substantially improve performance by saving both CPU cycles and data read time, although it violates normalization principles. **An example application is the maintenance of aggregate values.** Storing derived data can help to eliminate joins and reduce time-consuming calculations at runtime. However, maintaining data integrity can be complex if the data items used to calculate a derived data item change unpredictably or frequently, as the new value for derived data must be recalculated each time a component data item changes. Thus, the frequency of access, required response time, and increased maintenance costs must be considered.

1.4. Partitioning Tables

When distinct parts of a table are used by different applications, the table may be split into distinct tables, either vertically or horizontally. Figure 3 illustrates both these approaches.

A **vertical split** involves splitting a table by columns so that a group of columns is placed into one table and the remaining columns are placed into a second table. Vertical splitting can be used when some columns are rarely accessed or when the table has a large number of columns. The result of splitting a table is a reduction of the number of pages/blocks that need to be read because of the shorter row length in each table. With more rows per page, I/O is decreased when large numbers of rows are accessed in physical sequence. A vertically split table should contain one row per primary key in the split tables, as this facilitates data retrieval across tables. **In practice, a view of the joined tables may make this split transparent to the users**.

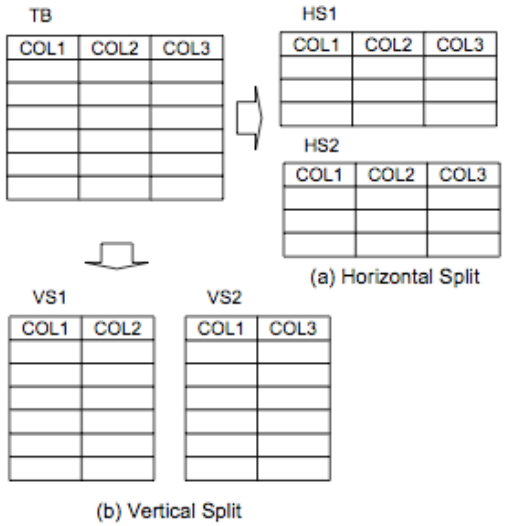

Figure 3: Horizontal and vertical table partitioning.

A **horizontal split** involves splitting a table by rows (i.e., by key ranges). This can be used when a table is large in terms of the number of stored rows. A horizontal split is usually applied when the table split corresponds to a natural separation of the rows such as different geographical sites, or historical versus current data. It can be used when there is a table that stores a large amount of rarely used historical data, and when there are applications that require a quick result obtained from that table. Reducing the size of the table reduces the number of pages analyzed during a table scan, and it can also reduce the number of index pages read in a query. A horizontal table scan can lead to fewer levels of a B+tree structure, thus reducing the number of disk reads that are required.

It is important to note that a database designer, when performing a horizontal split, must be careful to avoid duplicating rows in the new tables so that a "*UNION ALL*" may not generate duplicated results when applied to the two new tables.

In general, **horizontal splitting adds a high degree of complexity to applications**. Consider that it usually requires different table names in queries, according to the values in the tables. This complexity alone usually outweighs the advantages of table splitting in most database applications.

SQL Server has several database partitioning enhancements to improve performance and reduce the administrative overhead involved with managing partitioned data sets. In SQL Server, you can create a partition function, and then a partitioned table or index using this function (see Lab 1). SQL Server will handle the distribution of data ranges to the different partitions according to the range function. This will look like a single table. For performance reasons, it may be better to have the partitioned table span multiple file groups (possibly on different disks or arrays) with each partition allocated to its own file group. It is also possible to create partitioned indexes. By default, indexes created on a partition will use the same partitioning scheme as the partitioned table.

1.5. Materialized views in SQL Server

Using views can help to reduce the drawbacks of schema tuning. Views are virtual tables that are defined as a database object whose definition is based on other items stored in the database. Since no data is stored in the view, the integrity of the data is not an issue. Also, because data is stored only once, the amount of disk space required is minimal.

Most DBMSs process view definitions at run time, so a view does not solve performance issues. However, **SQL Server supports the notion of materialized views**. In SQL Server, these are also called **indexed views** because they have a unique clustered index created on them. There can also exist non-clustered indexes created over the view, so long as it a unique clustered index has been created over it.

Through materialized views, we can have the performance advantages of schema tuning, without the data consistency issues (i.e., the logical schema of the database does not change, thus avoiding consistency issues, but the automatically updated views provide efficient data access). Materialized views are particularly interesting to support derived attributes and redundant columns. Vertical partitioning can also be supported through materialized views, although non-clustered indexes including the attributes used in the query should be a more interesting option.

1.6. Non-Clustered Indexes Over Derived Attributes in SQL Server

In SQL Server, it is possible to create tables involving the usage of **derived attributes** (i.e., attributes whose values are computed from other attributes, using arithmetic calculations, algorithms or procedures). For example, in a relation storing bank accounts, an attribute corresponding to the net cash balance can be derived by adding all deposits and subtracting all disbursements or payments made. The values for the derived attributes are normally computed at query time but, to speed up query processing, it may be interesting to **create non-clustered indexes involving the derived attributes** in the index key.

In SQL Server, it is possible to define indexes on computed attributes, as long as the expressions used to compute the derived attribute are deterministic (i.e., if they always return the same output for the same inputs). This way, the values for the derived attributes become materialized in the database, and we no longer need to compute them at query time.

Remember also that, in SQL Server, it is possible to use the INCLUDE option in the CREATE INDEX statement to add one or more columns (e.g. regular attributes or derived attributes that are deterministic) to the leaf level of a non-clustered index, instead of including additional attributes in the search key for the index. This way, indexes can be kept smaller, but we can nonetheless *cover* the attributes used in the query and execute the query only through accesses to the non-clustered indexes, thus avoiding accesses to the underlying relation.

2. Experiments and Exercises

2.1. Experiments

As mentioned earlier, materialized (i.e. indexed) views can be used to reduce the drawbacks of schema tuning. In this experiment, we will analyze the usage of indexed views in SQL Server. In a previous lab (Lab 1), you have already seen how to use horizontal database partitioning using the mechanisms provided by SQL Server.

It is important to note that the SQL Server query processor treats indexed and non-indexed views differently. If the query optimizer decides to use an indexed view in a query plan, the indexed view is treated the same way as a base table (i.e., the tuples returned by the view are persistently stored in the database, given that an index has been created over the view). As for non-indexed views, only their definition is stored, and not the rows of the view. In this case, the query optimizer incorporates the logic from the view definition (i.e., it replaces the invocation to the view by the query that corresponds to the view definition), this way building an execution plan for the SQL statement that references the non-indexed view.

Consider, for example, the following view, which collapses two tables having a one-to-one relationship:

```
CREATE VIEW EmployeeName WITH SCHEMABINDING AS
SELECT e.BusinessEntityID, p.LastName, p.FirstName
FROM HumanResources.Employee e JOIN Person.Person p
     ON e.BusinessEntityID = p.BusinessEntityID;
```

When *SCHEMABINDING* is specified, the base table or tables cannot be modified in a way that affects the view definition (for example, it is forbidden to execute an "*ALTER TABLE*" statement to remove the attributes of the base tables that are used within the view). The view definition itself must first be modified or dropped and only afterwards, the base tables can be modified.

Execute the statement above to create the view. Then execute each of the following SQL statements, which should produce the same results:

```
-- select using the view
SELECT LastName AS EmployeeLastName, SalesOrderID, OrderDate
FROM Sales.SalesOrderHeader AS soh JOIN EmployeeName as EmpN
     ON soh.SalesPersonID = EmpN.BusinessentityID
WHERE OrderDate > 'January 1, 2013';

-- select using the base tables
SELECT LastName AS EmployeeLastName, SalesOrderID, OrderDate
FROM Sales.SalesOrderHeader AS soh
     JOIN HumanResources.Employee as e
     ON soh.SalesPersonID = e.BusinessEntityID
     JOIN Person.Person as p
     ON e.BusinessEntityID = p.BusinessEntityID
WHERE OrderDate > 'January 1, 2013';
```

Display the execution plan for the two queries above. Looking at the execution plan, you will see that the engine is accessing the base tables in both cases. This is because the engine is replacing the view by its definition in the first query, which becomes equivalent to the second query.

SQL Server will choose a different execution plan if the view is materialized. For this purpose, create the following index to turn the view into an indexed (i.e. materialized) view:

```
CREATE UNIQUE CLUSTERED INDEX IDX_V1
ON EmployeeName (BusinessEntityID, LastName, FirstName);
```

Repeat the previous two queries and check the resulting execution plans. The view is now materialized, and the engine uses it in the first query. Since a materialized view is now available, the engine also uses it to optimize the second query.

Indexed views can also be used to implement the maintenance of aggregate values. Consider the following example:

```
CREATE VIEW Sales.vOrders WITH SCHEMABINDING AS
SELECT SUM(UnitPrice*OrderQty*(1.00-UnitPriceDiscount)) AS Revenue,
       OrderDate, ProductID, COUNT_BIG(*) AS Counting
FROM Sales.SalesOrderDetail AS sod, Sales.SalesOrderHeader AS soh
WHERE sod.SalesOrderID = soh.SalesOrderID
GROUP BY OrderDate, ProductID;

CREATE UNIQUE CLUSTERED INDEX IDX_V2
ON Sales.vOrders (OrderDate, ProductID);
```

The following query will use the indexed view, even though the view is not specified directly in the FROM clause:

```
SELECT SUM(UnitPrice*OrderQty*(1.00-UnitPriceDiscount)) AS Rev,
       OrderDate, ProductID
FROM Sales.SalesOrderDetail AS sod, Sales.SalesOrderHeader AS soh
WHERE sod.SalesOrderID = soh.SalesOrderID
      AND ProductID BETWEEN 700 and 800
      AND OrderDate > 'January 1, 2013'
GROUP BY OrderDate, ProductID
ORDER BY Rev DESC;
```

Run the query above and check its execution plan.

Besides materializing the view through the creation of a clustered index, we can also define a non-clustered index over some of the attributes of the view, including derived attributes such as Revenue from the previous example.

```
CREATE NONCLUSTERED INDEX NEW_IDX_V2
ON Sales.vOrders (Revenue)
INCLUDE (ProductID);
```

This non-clustered index can be particularly useful to optimize queries involving only some of the attributes, leading to index-only plans. An example would be:

```
SELECT ProductID FROM Sales.vOrders WHERE Revenue > 1000;
```

Run the query above and check its execution plan.

2.2. Exercises

Consider the following relational database schema:

```
Clients(ClientId, ClientName)
Activities(ActivityId, ActivityName, ActivityType)
Employees(EmployeeId, EmployeeName)
Projects(ProjectId, ProjectName, ClientId)
      ClientId : FK(Clients)
TimeSheet(ProjectId, ActivityId, EmployeeId, Date, Hours)
      ProjectId : FK(Projects)
      ActivityId : FK(Activities)
      EmployeeId : FK(Employees)
```

2.2.1 – For each of the following cases and, assuming that all the corresponding queries execute very frequently, indicate a possible optimization at the schema level (e.g., addition of new attributes to the existing relations, creation of new relations, partitioning of relations, etc.):

a) Queries that compute the total time spent on each project.
b) Queries that retrieve the names of activities and employees involved in a given project.
c) Queries that retrieve the IDs of activities that take more than 5 hours to be executed.
d) Queries that retrieve the names of the activities of type “simple” (activities may be “simple” or “complex”).

2.2.2 – For each case considered above, explain what is the benefit brought by the schema optimization technique suggested, and indicate a potential problem that may be introduced by the use of the schema optimization technique suggested.

2.2.3 – Explain how the following mechanisms supported by SQL Server can be useful to optimize queries without some of the drawbacks associated with schema tuning:

a) Materialized views.
b) Partition functions and partition schemes.
c) Derived attributes and indexes on derived attributes.
d) Indexes that include additional attributes (regular or derived) beyond the search key.