Data Administration in Information Systems

Lab 3: Query Processing

In this lab class, we will address query processing. For students with a particular interest on the subject of query processing within Microsoft SQL Server, Craig Freedman's SQL Server Blog is a good source of useful information.[1]

1.1 - Basics of Query Processing in SQL Server

At the heart of a relational database there are two major components: the storage engine and the query processor. The **storage engine** writes and reads data from disk. It manages records, controls concurrency, and maintains log files. The **query processor** accepts SQL queries, selects an execution plan, and then executes the chosen plan.

There are two main phases in query processing: query optimization and query execution. **Query optimization** is the process of choosing the fastest execution plan. We will look at this in the next lab class. **Query execution** concerns with executing the plan chosen during query optimization. We will present query execution before query optimization because the set of available execution techniques determines the set of choices available to the optimizer. The basic algorithms used for processing queries are presented below.

1.1.1 - Executing Selections: Full Table Scans

If there are **no indexes on a table**, the data can only be accessed by means of a full table scan. When a table scan is performed, each page from the table is read, starting at the first page and ending at the last page. To read each page (i.e., for each page request), SQL Server performs a logical read, also known as a logical I/O (i.e., the page is requested to the buffer manager and, if the page is not found in the data cache, the operation results in a physical read from disk).

[1] http://blogs.msdn.com/craigfr/

1.1.2 - Executing Selections: Index Scans and Index Seeks

As we saw in the previous lab, SQL Server uses one of two methods to organize the data pages from tables. If a table has a **clustered index**, SQL Server uses a B+Tree file organization, in which the data rows are stored in the leaves of a B+Tree, in order, according to the clustered index key. If a table has no clustered index, then the data rows are not stored in any particular order, and there is no particular order of the data pages.

When executing selections, if the table has a clustered index, then the selection can be executed through:

i. a **clustered index seek** (i.e., the index is traversed from the root node to the first leaf node that satisfies the selection criteria), if the index key is involved in the selection;
ii. a **clustered index scan** (i.e., the leaf pages of the B+Tree are read sequentially, and the requested tuples are selected).

If the table has no clustered index, then the selection is executed through a **table scan**, as we saw earlier. If the table has no clustered index, but if it has a **non-clustered index** that includes all attributes that are involved in the selection (covering index), then the operation can also be executed through an index scan/seek, using only the non-clustered index.

1.1.3 - Executing Joins: Nested-Loop Joins

In nested-loops join, **tables are processed by a series of nested loops**, also referred to as nested iterations. In a two-table join every row selected from the outer table (the table in the outer loop) causes the inner table (the table in the inner loop) to be accessed. This is known as a scan (not to be confused with table scan). The number of times the inner table is accessed is known as its scan count. **The outer table will have a scan count of 1 and the inner table will have a scan count equal to the number of rows selected in the outer table**. If possible, the outer table will use indexes to restrict the access to rows, whereas the inner table will use indexes on the join columns and potentially any other indexes that might be efficient in limiting the rows returned. The index on the join column is the most important index, since, without it, the inner table will be table scanned for each relevant row in the outer table. The optimizer attempts to make the table with the smallest number of qualifying rows the inner table.

1.1.4 - Executing Joins: Merge Joins

Merge joins can be efficient when two large tables of similar size need to be joined and **they are already sorted** (e.g., by virtue of their indexes or a previous sort operation). The scan count for each table is one. The result from a merge join is sorted on the join column. The **equality operator** must be used in the query to join the tables, otherwise a merge join cannot be used.

There are two types of merge join: a **one-to-many (regular merge join)** and a **many-to-many**:

- In the case of a one-to-many, one input will contain unique values in the join column, whereas the other will contain zero, one, or many matching values.
- In the case of a many-to-many merge join, both inputs may contain duplicate values in the join column. A many-to-many merge join requires that a temporary worktable is used. We must keep a copy of each row from input 2 whenever we join two rows. This way, if we later find that we have a duplicate row from input 1, we can play back the saved rows. On the other hand, if we find that the next row from input 1 is not a duplicate, we can discard the saved rows. SQL Server saves these rows in a temporary worktable in the **tempdb** database. The amount of disk space needed depends on the number of duplicates in input 2.

If the join column from one input contains unique values, the query optimizer will not know this unless a unique index is present on that column. If the two join columns from the two input tables both have a clustered index created on them, the query optimizer knows that the rows are physically sorted on the join column, and it won't perform a sort on any of the inputs. Joining these two tables will probably be performed with a merge join, especially if the merge join is a one-to-many.

The presence of an ORDER BY clause on the query, involving the join attributes, will increase the likelihood that a merge join is used. If the two join columns from the two input tables both have a non-clustered index created on them, then the query optimizer knows that the rows are not physically sorted on the join column. In this case the query optimizer will need to sort the inputs. Joining these two tables with a merge join is less likely, unless an ORDER BY clause on the query is used. In this case, the optimizer will decide if a merge join is more efficient.

1.1.5 - Executing Joins: Hash Joins

With hash joins, there is **no need to have indexes** on the tables to be joined. This means that the hash join mechanism can be used to join any two non-indexed inputs. This is very useful, especially when joining intermediate results, where indexes are not available. The **equality operator** must be used in the query to join the tables; otherwise, a hash join cannot be used.

With a hash join, there are two inputs: the build input and the probe input. The build input is used to build a hash index, and the probe input is used to lookup values on that index. The build input is typically the smaller table (i.e., the one with fewest rows, after applying the selection criteria).

Memory is needed for the hash buckets, so hash joins tend to be memory and CPU intensive. They typically perform better than merge joins if one table is large and one is small, and they are better than nested loops joins if both tables are large. However, because a hash index must be built first, hash joins are not efficient when the first row of the join must be retrieved quickly. Hash joins will also produce a non-ordered output, so an extra sorting operation is required if one wishes the output to be ordered.

Hashing mechanisms can be particularly useful to join non-indexed inputs. One example of this are cases involving intermediary results obtained from other join operations, consisting for instance of pointers to the data rows obtained from non-clustered indexes.

Hashing approaches (i.e., the hash aggregate algorithm) are also useful for computing the results of aggregation operators – for example, SUM or MAX with a GROUP BY – or for implementing the DISTINCT operator.

1.2 - Particular aspects of query processing in SQL Server

For better performance in query execution, **SQL Server stores execution plans in a memory cache** (i.e., the procedure cache). Each execution plan has two main components, namely the query plan and the execution context:

- **Query plan** – The bulk of the execution plan is stored in a read-only data structure. For a given SQL statement, there are never more than one or two versions of the query plan in memory; one copy for all serial executions and another for all parallel executions.
- **Execution context** – Each user currently executing the query has a data structure that holds the data specific to their execution, such as parameter values (e.g. constant values used in the SQL statement). The execution context data structures are reused. If a user executes a query and one of the structures is not in use, it is reinitialized with the context for the new user.

When any SQL statement is executed, SQL Server first looks through the procedure cache to see if there is an existing execution plan for that SQL statement. SQL Server reuses any existing plan it finds, saving the overhead of recompiling the SQL statement.

Microsoft SQL Server also uses **intra-query parallelism**, which is the ability to break a single query into multiple subtasks and execute them on multiple processors. By examining the current system workload and configuration, SQL Server determines the optimal number of threads and spreads the parallel query execution across those threads. When a query starts executing, it uses the same number of threads until completion. SQL Server chooses the optimal number of threads each time a parallel query execution plan is retrieved from the procedure cache. As a result, one execution of a query can use a single thread, and another execution of the same query (at a different time) can use two or more threads.

SQL Server can take advantage of **multiple indexes**, selecting small subsets of data based on each index, and then performing an intersection of the two subsets (that is, returning only those rows that meet all the criteria). For example, suppose you want to count orders for certain ranges of customers and order dates, over an hypothetical *orders* table:

```
SELECT count(*) FROM orders
WHERE o_orderdate BETWEEN '9/15/1992' AND '10/15/1992'
AND o_custkey BETWEEN 100 AND 200;
```

SQL Server can exploit indexes on both o_custkey and o_orderdate, and then compute the **index intersection** between the two subsets. This execution plan exploits two indexes, both on the orders table, exploiting the fact that reading the index pages should be faster than reading data pages.

Also when using any index, if all the columns required for a given query are available in the index itself, it is not necessary to fetch the full row. Since the index contains all the columns needed for the query, this originates an **index-only plan**. SQL Server takes index-only plans one step further, through **index intersection**. If no single index can cover a query, but multiple indexes together can, then SQL Server considers merging these indexes.

SQL Server allows users to provide **query hints** such as **FORCE ORDER, LOOP JOIN or KEEP PLAN**, to guide the optimizer into selecting a specific algorithm. A hint instruction called USE PLAN can be used to guide the query optimizer into selecting a particular execution plan, specified in XML. The USE PLAN query hint is specified in an OPTION clause, together with an XML representation of the execution plan.

The following is an example of how the USE PLAN hint could be specified to influence the join type of a simple query that consists of a join between two tables.

```
SELECT count(*) AS Total FROM Sales.SalesOrderHeader h,
                              Sales.SalesOrderDetail d
WHERE h.SalesOrderID = d.SalesOrderID
OPTION (USE PLAN '<ShowPlanXML
               xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan"
               version="0.5" build="9.00.1187.07">
                      <BatchSequence>
                          <Batch>
                              <Statements>
                                  ...
                              </Statements>
                          </Batch>
                      </BatchSequence>
                  </ShowPlanXML>')
```

In the SQL Server console, XML-formatted query plans can be produced through the instruction **SET SHOWPLAN_XML ON**, which causes SQL Server to return a XML representation for the corresponding query plan.

In SQL Server Management Studio, execution plans can be easily analyzed through graphical representations. A list of all icons that can appear in graphical execution plans can be found in the online documentation.[2]

Four specific pieces of information, associated with each operation involved in the query, are of particular interest in evaluating performance:

- **Estimated I/O Cost**. This represents the cost for data input/output activity.
- **Estimated CPU Cost**. This represents the cost for the processor(s).

[2] https://msdn.microsoft.com/en-us/library/ms175913.aspx

- **Estimated Operator Cost**. This represents the cost to the query optimizer of a particular operator (i.e., a step in the complete execution plan), as well as the cost percentage in terms of the total cost for the query.
- **Estimated Subtree Cost**. This represents the cost to the query optimizer of all operations up to a particular step in the query execution plan (i.e. the cost of an operator plus the cost of all previous steps).

These numbers can be affected by many factors, including the server environment in which the database resides, and the current activity load in SQL Server.

2 Exercises and Experiments

We will use SQL Server Management Studio to obtain the query execution plan for several queries over the AdventureWorks database. The query execution plan outlines how SQL Server query optimizer actually runs a specific query.

2.1 - Experiments with SQL Server

a) Open a **New Query** window on the **AdventureWorks** database.

b) Execute the following command to get the query results along with the scan count and number of read operations across the tables involved:

```
SET STATISTICS IO ON;
```

c) Execute the following simple query:

```
SELECT * FROM Sales.Customer;
```

d) Check the query results in the **Results** tab. In particular, note how the **AccountNumber** field seems to be somewhat similar to the **CustomerID** field.

e) Change to the **Messages** tab to check the **scan count**.

f) Select the query above and press the **Display Estimated Execution Plan** button.

g) Note the following features about the execution plan:

- The execution plan uses a **clustered index scan**. You can check that indeed such clustered index exists by right-clicking the **Sales.Customer** table and selecting **Design**. Then on the **Table Designer** menu, select **Indexes/Keys** to see the indexes on the table.

- The execution plan uses two **compute scalar** steps. This is necessary to compute the **AccountNumber** field. To see how the **AccountNumber** field is computed, right-click the **Sales.Customer** table and selecting **Design**. Then select the **AccountNumber** field and, in the **Column Properties** tab, check the **Computed Column Specification** tab.

h) Change the query to:

```
SELECT AccountNumber FROM Sales.Customer
WHERE TerritoryID >= 5;
```

i) In the query execution plan, check that the system has now changed to a **non-clustered index seek** on **TerritoryID**.

j) Check the execution plan for the following query involving an inner join:

```
SELECT *
FROM HumanResources.Employee AS e INNER JOIN Person.Person AS c
     ON e.BusinessEntityID=c.BusinessEntityID
ORDER BY c.LastName;
```

k) Check the execution plan for the following query involving a join and an aggregation:

```
SELECT oh.CustomerID, SUM(LineTotal)
FROM Sales.SalesOrderDetail AS od JOIN Sales.SalesOrderHeader AS oh
     ON od.SalesOrderID=oh.SalesOrderID
GROUP BY oh.CustomerID;
```

l) Finally, check the execution plan for a more complex query that involves multiple joins and an aggregation operation:

```
SELECT c.CustomerID, SUM(LineTotal)
FROM Sales.SalesOrderDetail AS od
     JOIN Sales.SalesOrderHeader AS oh ON od.SalesOrderID=oh.SalesOrderID
     JOIN Sales.Customer AS c ON oh.CustomerID=c.CustomerID
GROUP BY c.CustomerID;
```

m) In this last example, the execution plan performs the following sequence of steps:

1. A **clustered index scan** on **Sales.Customer** that returns the **CustomerID** for all rows.

2. A **non-clustered index scan** on **Sales.SalesOrderHeader** that, for a given **CustomerID**, retrieves one or more **SalesOrderID**.

3. A **merge join** to join customers (**CustomerID**) with their orders (**SalesOrderID**).

4. A **clustered index scan** on **Sales.SalesOrderDetail** to retrieve the order details (**SalesOrderID**, **OrderQty**, **UnitPrice**, and **UnitPriceDiscount**) for all customer orders.

5. Two **compute scalar** steps to calculate the **LineTotal** field from **OrderQty**, **UnitPrice**, and **UnitPriceDiscount**. (Where can you find the actual specification of how this column is computed?)

6. A **hash match (inner join)** to join sales orders (**CustomerID** and **SalesOrderID**) with their corresponding line totals (**SalesOrderID** and **LineTotal**).

7. A **hash match (aggregate)** to group by **CustomerID** and calculate the sum of **LineTotal** within each group.

8. A **select** that is a placeholder that represents the overall query results and cost.

2.2. Exercises

2.2.1. Suppose that table *instructor* has a B+Tree index on *dept_name*, and there are no other indexes. What would be your execution strategy for the following queries that involve negation?

a. $\sigma_{\neg(\text{dept\_name}<\text{"Geography"})}(\text{instructor})$

b. $\sigma_{\neg(\text{dept\_name}=\text{"Geography"})}(\text{instructor})$

c. $\sigma_{\neg(\text{dept\_name}<\text{"Geography"} \vee \text{salary}<5000)}(\text{instructor})$

2.2.2. Assume that only one tuple fits in a block, and assume that the memory holds at most 3 blocks. Show the runs created on each pass of the sort-merge algorithm, when applied to sort the following tuples on the first attribute:

(K, 17), (W, 21), (E, 1), (Y, 13), (P, 3), (L, 8), (X, 4), (Z, 11), (M, 6), (J, 9), (H, 2), (B, 12).

2.2.3. Let relations $r_1(A, B, C)$ and $r_2(C, D, E)$ have the following properties: r_1 has 20 000 tuples, r_2 has 45 000 tuples, 25 tuples of r_1 fit on one block, and 30 tuples of r_2 fit on one block. Assume that only 3 blocks fit in memory. Estimate the number of block accesses required, using each of the following join strategies for $r_1 \bowtie r_2$:

a. Nested-loop join;

b. Block nested-loop join;

c. Merge join;

d. Hash join (use a *fudge factor* of 1).