SQL Coding Best Practices – Focus on Performance

Abstract -

As business requirements demand applications to be more and more flexible and user friendly, the data access and updates become more critical. For applications to be flexible and meet the service level agreements (SLAs) database reads and writes should happen at optimum performance level. Thus it is imperative that the developers and the database administrators (DBAs) must follow the SQL coding best practices to achieve their goal. This document highlights some of the important features on SQL coding best practices which will guide developers and database administrators to optimize the data access layer. These features can be implemented across all RDBMS to achieve best performance results form database.

Introduction

This paper covers important features of SQL coding best practices with primary objective to optimize SQL query performance. This paper highlights some of the proven facts and my personal experiences in SQL coding as a DBA. This white paper presents SQL coding best practices with a one liner action item as a bullet point which is explained in detail with appropriate examples. Also these best practices are supported with real time statistics captured during SQL coding. All the documented features are equally important and there is no order of priority. More over the best practices mentioned in this paper are abstracted in such a way that they can be applied as universal guidelines to access/modify data in any modern and matured RDBMS which supports ASCII SQL Standards.

Avoid Table Scans.

Avoid table scans on large tables. Table Scan(s) means reading the entire table data. Generally there are the queries which do not have any/proper filter condition and the database process has to read through the entire table. Some of the basic rules to avoid table scans is explained below with examples.

i) Do not use SELECT WITH OUT WHERE CLAUSE. Restrict data access by horizontal partition.

ii) Do not use SELECT DISTINCT to populate lists; instead, maintain a separate table of the unique items.

iii) Use proper filter conditions.

Example – 1

Select with No filter condition.

```
SELECT AIRPORT_CODE FROM dbo.CTIY
```

Example – 2

Select with DISTINCT clause on wrong table condition.

```
SELECT DISTINCT AIRPORT_CODE FROM dbo.CITYExample – 3
```

Select with wrong filter column.

```
SELECT DISTINCT AIRPORT_CODE FROM dbo.CITY WHERE COUNTRY = 'INDIA'
```

Example – 4

Select with wrong filter column.

```
SELECT DISTINCT AIRPORT_CODE FROM dbo.CITY WHERE COUNTRY_CODE = '001'
```

* Considering the COUNTRY_CODE column is a non-clustered index.

By using proper filter condition as in Eg–4; the result-set send to application layer is precise and executes faster following the non-clustered scan. This will save considerable amount of data reads

and is faster. As in Eg-3; where the indexes on the underlying table are not in-sync with the filter columns, table scan is a possibility. This situation is covered later portion of this paper.

***Note** – For smaller tables, table scans might benefit in performance. But a clear discretion must be made to identify/judge the table as small and will be remain small for ever.*

Replace "IF EXISTS()" for COUNT (*)

There will be several scenarios where application flow needs to follow different business process if specific data is present or not. In order to, just verify if the specific record is present or not; use "IF EXISTS ()" rather than doing a count (*) operation. This way checking process will be faster as it ends when the first satisfying record is encountered rather than scanning entire table to find the number of records satisfy the condition and then checking for that count greater than ZERO or not. Below example explains how a query can be rewritten to gain performance in a web registration form query. To verify if the user email id is already used for registration.

Example - 5

Original query -

```
IF (SELECT COUNT (USER_ID) FROM dbo.USER_CONTACTS WHERE EMAILID =
'krishna.singitam@symphonysv.com' AND STATUS = 'ACTIVE') > 0
BEGIN
        Message the User to register with a different Email Id.
        ----
END
ELSE
BEGIN
        Continue User registration.
        ----
END
```

Example - 6

Optimized query -

```
IF EXISTS (SELECT TOP 1 USER_ID FROM dbo.USER_CONTACTS WHERE EMAILID =
'krishna.singitam@symphonysv.com' AND STATUS = 'ACTIVE')
BEGIN
        Message the User to register with a different Email Id.
        ----
END
ELSE
BEGIN
        Continue User registration.
        ----
END
```

Implementation –By implementing this change, proved to save 33% of query cost, in a similar scenario as above.

***Note** – In all scenarios it may not save the same cost it all depends on several factors like the table size, table/index statistics and other RDBMS specific features. But this implementation in general for sure will benefit in speeding up of the database operation.*

Keep the transaction(s) as simple and small as possible.

To maintain data consistency, database activates are implemented as single unit-of-work called transactions by the database engine. These transactions will consume database resources as well as lock the database objects for the entire duration of its execution. Thus when a transaction does huge database operation, users/application(s) could face concurrency issues which will reduce the throughput of the database and will drastically impact the scalability of the application(s). So by keeping the transaction smaller, server resources and database objects will not be held for a long duration. As mentioned, data consistency comes at the cost of concurrency which is controlled by active transactions. So all the transactions are required to be closed properly, to make sure no open transaction(s) are left un-committed or un-rollback after its execution. This way the recourses and database object locks are released as and when the database transactions are completed. By following below steps, transaction can be executed faster.

i) Keep the transaction process simpler and its volume smaller.
ii) Close all the transactions explicitly.

Example - 7

Transaction to update the price of all the items by 2% discount.

```
UPDATE      dbo.PRICELIST
SET    OLD_PRICE        =     CURRENT_PRICE
       CURRENT_PRICE    =     CURRENT_PRICE * 0.02
       MODIFY_DATE      =     GETDATE()
```

The above query will update all the records price with this data and could take up lot of transaction log space to complete this task. By breaking this into smaller transactions will speed up the process and will not lock the PRICELIST table for a long time.

Example - 8

Modify above transaction to update the price in batches.

```
IF EXISTS (SELECT TOP 1 FROM dbo.PRICELIST WHERE CURRENT_PRICE !=
OLD_PRICE)
BEGIN
SET ROWCOUNT 10000
UPDATE      dbo.PRICELIST
SET    OLD_PRICE        =     CURRENT_PRICE
       CURRENT_PRICE    =     CURRENT_PRICE * 0.02
       MODIFY_DATE      =     GETDATE ()
END
```

Implementation – This coding approach was implemented in several features to reduce the transaction volume. Same transaction with smaller batch size is used to commit the processed data at regular intervals. By using this coding practice, blocking of other processes when running huge transactions were avoided thus improving the application throughput and performance as well as scalability.

Close and de-allocate all cursor(s) after use.

In situations, to impose a business rule the database process needs to apply the conditions to each and every record/row for a satisfied result set. In such scenarios the SQL developers will extensively use cursor operations to work on each row. However in building such database components (cursors) DBA/SQL developers should take extra care in closing all the cursor objects and releasing the cursor address allocations. Some of the important points in using cursors are:

i) Open cursor for “read only” in case not updating the parent tables.
ii) Use one directional preferably forward only cursor which is the fastest of the cursor types.
iii) Make sure the cursor row size as minimum as possible. (No TEXT, NVARCHAR (MAX) etc)

Example - 9

```
DECLARE BOOK_CUR CURSOR FOR
SELECT ISBN FROM dbo.BOOKS AS WHERE INSTORE = 'TRUE'

OPEN BOOK_CUR
FETCH NEXT FROM BOOK_CUR INTO @l_BOOK
WHILE @@FETCH_STATUS = 0
BEGIN
  ---- logic with join with BOOKS, Authors, Publications and Sales tables.
  ----
  FETCH NEXT FROM PPFF_CUR INTO @l_CURR_PPID
END

CLOSE BOOK_CUR
DEALLOCATE BOOK_CUR
```

SET NOCOUNT ON in Stored Procedures

Stored procedures are the database objects consisting of logical SQL statements which are executed as a single unit when executed. When a client application invokes a Stored Procedures all the SQL statements executed will spool the records affected on its successful completion back to the client. This will use some amount of network band width. In a typical Client/Server or Web based environment this intermittent information is irrelevant to the application. Over a high scaling application with multiple concurrent users all such information spooled to the client over the network could narrow the network band width. This will impact the network throughput between DB server and the client and so the performance. To avoid such unwanted information to be sent to client, database code should suppress the affected rows for each SQL statements within a stored procedure. This can be done by using the set command as "SET NOCOUNT ON" at the beginning of the stored procedure, thus reducing the load on the network.

Example – 10

Objectiove – To supress the rows affected messages from SP.

```
CREATE PROCEDURE dbo.[P_COPY_USER_INFO]
-----
Input/Output Parameters.
----
AS
BEGIN
        SET NOCOUNT ON
        -----
        Input/Output Parameters.
        ----
END
GO
```

Note *– In some cases rows affected by a query needs to be captured to process further or pass the number of rows affected information as message back to client. Developer need to see how useful this option is to a specific situation and tailor the code to best suit the requirements.*

Move ad-hoc queries to Stored Procedures or Functions.

In some situations, applications do access/update data in the database using SQL queries. As long as these are basic CRUD operations with simple validations will not cause problems. But as these become more and more complex transactions, it is always best to move the SQL code as a database object in the database. More over the applications which allow ad-hoc queries from the client on the data server will not have any control on the volume of data it is going to work leading to long running transactions/queries. Below are some of the major advantages along with the performance gain by implementing such queries as stored procedures.

i) Gain in performance as a critical path of execution is stored in memory call procedure cache.
ii) Stored procedures run as a data server process, so in most cases it will be better equipped machine than a client for faster execution of the query.
iii) Higher security can be achieved by granting access to stored procedure, rather than underlying tables.
iv) Applications can share the database components in a central database. Code reusability.
v) Easy to maintain.

Avoid "RECOMPILE" option in stored procedure in code.

Stored Procedure is precompiled SQL code which is executed by database engine as a unit of work when invoked by the application. The major benefit of stored procedure is gain in performance as mentioned above. This is achieved as the critical execution path of the stored procedure is stored in the database memory called procedure cache. When a SP is complied first time its optimum execution plan is stored in database server memory.** When it is invoked subsequently database engine reads procedure cache from the memory and executes the stored procedure faster. In case the stored procedure was written to be force recompile every time it executes, a new plan is generated before execution. Thus takes more time to execute the stored procedure. So avoid explicitly recompiling the stored procedures in code. In case, if stored procedures are required to be recompiled, such activity should be taken as part of database maintenance activity.

Note *– i) All the SP's are required to be recompiled when the system statistics are updated or when indexes are recreated for regenerating the new execution paths Also there are some smart RDBMS available which recompile the stored procedures as and when the database engine detects such need. DBA need to make some settings to make it an automatic maintenance process.*
ii) In case a stored procedure is run always after maintenance activity can be coded with recompile option.
***Need to verify RDBMS specific database settings for setting the value for procedure cache.*

Avoid repetitive calls to functions.

There will be scenarios where a function will be called several times with in the SQL code. Developer should try to avoid such repetitive calls which could lead to unnecessary network traffic. These can be stored a temporary table or variables and can be used to process the data. Also to use built in functions which are much faster than the user-defined counterparts. Below example explains while inserting/updating of several millions rows with UTCDATE format. In Such case million calls to a user-defined function to get the UTCDATE (Universal Time Coordinator) can be avoided by replacing with built-in function.
Example - 11
Original Query -

```
SELECT dbo.F_MISC_UTCDATE_GET(GETDATE())
```

Optimized Query -

```
SELECT GETUTCDATE()
```

Implementation – This was implemented in several database objects like stored procedures/functions which inserts/updates/deletes several records in the database. In order to avoid the unwanted calls to the user-defined function as in original query, DBMS provided built-in function is used which is faster. The savings will be magnified as the number of calls avoided to the user-defined function.

Use intermediate materialization or derived tables.

In case of a table/result-set which is required at several places to join as a filter in the same SQL code, those records can be stored in memory as a temporary table. In other words, instead of calling the same sub-query several times to get the same result-set from the base table(s), the data can be obtained once and moved to memory as a temporary table/result-set. This can be used in a join/filter as a regular table. The principle works, as the table scan on a small table is much faster and efficient when it is cached to memory, thus reading the data from memory instead from disk, thus improving the performance. This gain in performance increases, when the result-set/table reads are several times with in the same query.

Example – 12
Original Query -

```
SELECT COUNTRYCODE FROM dbo.COUNTRY
```

Optimized Query -

```
SELECT TOP 50000 COUNTRYCODE FROM dbo.COUNTRY
```

Implementation – This is implemented in several Stored Procedures objects, to pre fetch the data in a faster way. The gain in performance was about 5-10% of total query cost over a wide range complex SQL code.

Avoid aggregate functions as far as possible.

Avoid aggregate functions like AVG(), SUM(), MAX(), MIN() if they are not required. By using the aggregate functions the underlying indexes are not fully used for best performance. In case the calculation involves complex business logic for finding the aggregate values, using a calculated filed in a table is recommended. Also an option of using DB/CLR stored procedures is recommended. Materialized views (Indexed views) will boost the performance as well. Of-course the maintenance is a small overhead to some extent. Considering a strict SLA application developers require to opt for better performance with a little extra code maintenance.

Note *– In case of adding a new feature/object in the database, SQL Developer should consider the pros and cons from performance as well as maintenance point of view.*

Working with smaller subset of data by appropriate filter criteria.

One of major factor that determines the SQL query time is the volume of data it requires to process during its execution. In general, during early stages of coding the developer may not have the clear picture of the volume of the data he/she will be handling. This could result in bad queries which will work with big volumes of data than it actually supposed to. So a developer should understand the business requirement and translate the same into a SQL query with the perspective of precise and optimum data access/updates. By using proper joins and applying the best possible filter conditions, the query will process smaller volume of data. This will bring out the best performance throughput from the query. Also developer should, avoid unnecessary joins, and avoid huge

calculations in the query which will help in boosting in the performance. Below example supports the gain in performance of a stored procedure by cutting down the execution time from 18min to 40sec, by rearranging the avoiding a join condition.

Avoid cast operators like UPPER (), CONVERT (), CAST () etc…

In a filter condition while comparing the data in a SQL query avoid using the conversion/casting/formatting the column field. Instead store the value to be compared in a local variable after converting into required data type or compatible format. And also make sure the underlying column remains in the same data type as per the table definition. By doing this, SQL query will use the available indexes do a index scan/seek, rather than a table scan in the case when the column of the table gets reformatted/changed to different data type. By using a proactive database design approach the data can be set to be case insensitive and making sure to define the data types in the required and convenient format. During design phase, DBA should understand the requirements and suggest the best possible approach to avoid the unwanted conversion/casting/formatting functions in the SQL query.

*Example – 13

Avoid data formatting function.

Original Query -

```
SELECT COUNTRYCODE FROM dbo.COUNTRY
WHERE UPPER(COUNTRYCODE) = UPPER('India')
```

Optimized Query -

```
SELECT COUNTRYCODE FROM dbo.COUNTRY
WHERE COUNTRYCODE = UPPER('India')
```

***Note** – Use UI validation to store the codes in UPPER case always or Set the server setting as case-insensitive.*

**Example – 14

Avoid cast function –

Original Query -

```
SELECT ORDERDATE, SUM(QUANTITY), SUM(PRICE) FROM dbo.ORDERS
WHERE PURCHASEDATE
BETWEEN '2006/01/01 00:00:00.000' AND '2007/01/01 00:00:00.000'
GROUP BY ORDERDATE
```

Optimized Query -

```
SELECT ORDERDATE, SUM(QUANTITY), SUM(PRICE) FROM dbo.ORDERS
WHERE PURCHASEDATE BETWEEN '20060101' AND '20070101'
GROUP BY ORDERDATE
```

***Note** - This is design approach to be decided at the initial stage to decide how the date-time data type column is used and what if the time portion is not significant. In such case the date can be stored in the database as an integer (YYYYMMDD) format, thus improves the performance of the query. This is a widely used approach in all kinds of Data Warehousing applications.*

Avoid negation operators.

It is know fact that a query written with negation operator takes more time than a similar regular operator. Try to avoid the negative operator in SQL usage and modify the query to benefit from the

better coding practices. Below example illustrates the typical usage of avoiding the negative operator.

Example - 15

Original query -

```
IF NOT EXISTS (SELECT TOP (1) ISNB FROM dbo.PUBLICATION WHERE COUNTRY =
'INDIA' AND YEAR BETWEEN '01/01/2006' AND '01/01/2007')
BEGIN
        ----
        ----
END
```

Optimized query -

```
IF EXISTS (SELECT TOP (1) ISNB FROM dbo.PUBLICATION WHERE COUNTRY =
'INDIA' AND YEAR BETWEEN '01/01/2006' AND '01/01/2007')
BEGIN
END
ELSE
BEGIN
        ----
        ----
END
```

Implementation – This is implemented in several Stored Procedures objects, to check if exists do nothing and if not process the data as required. The gain in performance was about 5-10% of total query cost over a wide range complex SQL code.

Avoid long alias names.

When sending the results from the server to client the data is sent over network in packets as per the network band width. When the names of the columns/results sets are aliased with long names means more data is required to be shipped over the network to the client.

Avoid "ORDER BY" clause if data sorting is not required.

Sorting of data is required generally in presentation layer as an user may expect/feel good to see the results in a specified order. So the resultant data is required to be sorted in the query using the 'ORDER BY' SQL clause. Such data sorting operations will take temporary space and CPU time and thus reducing the query performance. In order to overcome, the developer should understand if the data sorting is actually required or not. In case the result-set data is just for the internal processing and not for presentation, 'ORDER BY' can be avoided. By avoiding 'ORDER BY' clause where it is not required, the query will not spend time and resources for sorting of data and will save the server resources and speedup the query. Below is a typical example which explains the method and its benefits.

Example – 16

Original Query – 'ORDER BY' clause is removed where it is not required.

```
INSERT #TEMP_TAB
SELECT US.USRSTATID AS STATUS,
        US.BASESTATID AS BASESTATID,
        US.SEQ,
        US.NAME,
        US.DESCRIPTION
FROM    ARP2.USRSTAT US
```

```
WHERE          US.SUBCODE = @SUBCODE AND US.ISDELETED = 0
ORDER BY US.SEQ
```

Modified Query –

```
SELECT US.USRSTATID AS STATUS,
        US.BASESTATID AS BASESTATID,
        US.SEQ,
        US.NAME,
        US.DESCRIPTION
FROM   ARP2.USRSTAT US
WHERE US.SUBCODE = @SUBCODE AND US.ISDELETED = 0
```

Implementation - Above example is taken from the function called- F_STATUS_SEL.
The function is using a table variable to insert data in a SEQ order before returning from the function. Both the sort (ORDER BY clause) and insert operations are taking more resources wrt to cost of the query. By removing the ORDER BY clause, we can completely avoid the insert operation, thus saving 69% of cost from the function.

Avoid Cartesian joins and Outer joins and joins more than 6-8 tables.

SQL queries joining multiple tables should be joined with proper join statements/conditions. If the joins do not fall under inner/left-outer/right-outer joins, it will be a Cartesian join. These joins will be very expensive as they expand all the tables participating in the join. As an example if two tables each of 1000 records wrongly joined in a Cartesian join will bring the result set if 1 million records. Thus one wrong join condition makes a simple query an expensive one which will drastically hit on the performance of the query. So during SQL coding special attention to be paid while defining the table joins conditions. Also avoid the 'OUTER JOINS' as much as possible. The LEFT and RIGHT 'OUTER JOINS" will dilute the filter conditions and are very expensive as well. Another important fact is, for most of the transactional applications number of joining tables in a single query should be with a 6-8 tables. If the requirement forces the joins to be more tables the DBA/Developer need to opt for temporary tables or indexed views for better performance. If these situation(s) can be detected in early stages, database design approach should consider such scenarios and design the tables to avoid the need to join more tables. This could be approached by de-normalization of the tables or indexed views etc…

Summary

Above mentioned are some of the important performance guidelines which can be overlooked by developers during SQL coding. By understanding the concepts of these guidelines SQL developer/DBA can help them selves to implement them in any RDBMS as per the coding syntax and features available. In some of the cases implementation details with gain in performance and examples could not be documented due IP security policy and document size restrictions. The current document highlights the points which are generic and can be applied to any RDBMS. There are several other performance related guidelines/(may be tips) which can be implemented but are specific to one/some(not generic) of the RDBMS like Oracle/MSSQL/Sybase/UDB etc, which is out of the scope of the current discussion. Also all the guidelines mentioned above are equally important and are not documented in order of preference in implementation or gain in performance. SQL developer should first understand the business requirements and then try to implement these to gain best throughput from SQL query.

Conclusions

Most of database performance issues can be avoided by following performance guidelines during SQL coding. Although all the above mentioned guidelines may not be implemented in all real situations as the requirements may demand to deviate from these guidelines. In such situations developers/DBA should notify the Business and Database Architect about the possible performance issues. DBA/Database Architect should capture all such deviations. These deviations in performance guidelines should be considered as influencing factors for the capacity planning of the data server. By educating the SQL developer with the best practices and making them implement these in the coding will bring best quality product. A well designed database with well optimized SQL code will always enhance the life of the product.

Author –

Krishna Rao Singitam, Database Architect & DBA.
Symphony Services – Lawson GOC.

References –

All the reference examples and implementation details are from my personal experience as a DBA and Database modeler.

Quantifying results (Query Cost) are captures from tools used to during optimization of various queries during performance tuning on live projects.

http://www.devshed.com/c/a/MySQL/SQL-Performance-and-Tuning-Considerations/

http://www.sql-server-performance.com/articles/dev/sql_best_practices_p3.aspx

www.sqlskills.com