QUERY OPTIMIZATION IN MICROSOFT SQL SERVER

Blerta Haxhijaha, Jaumin Ajdari, Bujar Raufi, Xhemal Zenuni, Florie Ismaili

Faculty of Computer Science and Engineering, South East European University
e-mails: {bh27486; j.ajdari; b.raufi; xh.zenuni; f.ismaili}@seeu.edu.mk
Macedonia

Abstract: The banking software solutions depend heavily on databases, not only for storing data but also for everyday operations, which require processing of that data. Database inserts, updates, deletes; as well as calculation and retrieval operations are integral parts, and optimizing their execution is essential for an efficient business operation. This paper examines several techniques for improving database query performance, by looking at some common operations and examining ways to enhance their execution. MS SQL Server is used as an experimental approach to examine the proposed techniques.

Key words: Query, Optimization, SQL Server, Performance, Banking.

1. INTRODUCTION

Banking software solutions rely greatly on databases. Typical banking institutions provide numerous services to customers, ranging from transactional and saving account operations, national and international money transfers, and lending facilities. They record large amounts of data daily, for all customers and their personal and financial features, as well as for all their accounts, transactions per account and credit liabilities [1]. All the data related to these entities and services is kept in databases, and is subject to daily updates and inserts. Hence, a database that is fine-tuned to perform these operations in an optimal manner is crucial for a smooth and reliable execution of banking business processes.

According to Rys [2], large banking applications traditionally rely on relational database management systems (RDBMS) in order to support their core data store and services. According to [3], [4] and [5], the basic steps of query processing are: 1. Scanning, parsing, and decomposition of the SQL query, 2. Query optimization and 3. Query code generation and execution. In order to ensure good performance, these RDBMSs use the query optimizer engine, which is responsible for generating one or more execution plans for a given query and then choose the most efficient plan to use. Different query execution plans can exist for a given query, and the difference in cost between the best and worst plans may be several orders of magnitude [6]. In case of complex query executions, many disk accesses are required, and the transfer of data from disk is slow relative to the speed of main memory and CPU. Hence, it is worthwhile to allow for a considerable amount of processing by the query optimizer engine to choose a plan that minimizes disk accesses [3], [4], [5] and [6]. Relational query languages such as SQL, provide a high-level declarative interface to access the data stored in relational databases [7].

According to [8], around 80% of database performance issues can be reduced by tuning SQL query statements. Although most of the query optimizer's decisions are not in control

of the database designer or application developer, performance can be influenced with certain database design elements and techniques. This paper will investigate several common scenarios in the banking domain that involve database query processing, and will suggest alternatives that increase the execution performance of these queries.

The rest of this paper is organized as follows: section 2 presents a summary of some related work in this area. Section 3 investigates three different scenarios from the banking domain which involve database operations, discussing ways to optimize their performance. Finally, section 4 provides a summary of the reviewed techniques and gives some conclusions.

2. RELATED WORK

Numerous research papers have investigated techniques and approaches for achieving better performance in SQL Server, and methods for writing faster-running SQL queries. Most of the papers however, provide a collection of general techniques and do not investigate common database queries from a specific business domain, as in this research done. A short summary of several research papers that discuss query optimization techniques in Microsoft SQL Server is provided in the remaining part of this section.

Oktavia and Sujarwo [9] have measured the performance of four different sub-query evaluation methods in SQL Server 2012: EXISTS, IN, relational operator (=) and relational operator (=) combined with the TOP clause. They have found that, in the absence of indexes, the best performance is obtained with the IN and EXISTS options. However, after applying the indexing strategies suggested by the SQL Server Database Engine Tuning Advisor, the usage of the relational operator (=) combined with the TOP clause, gave the best performance results.

Lungu et al. [10] maintain that the usage of primary, foreign and unique key constraints contributes to the creation of better execution plans, as opposed to those created when triggers are used. They also recommend avoiding the usage of local variables in scripts that contain multiple queries, especially when these variables are passed between the queries, as this degrades performance.

Corlatan et al. [11] recommend the usage of stored procedures over ad-hoc queries. If this is not possible, they recommend using the *sys.sp_executesql* system stored procedure for running ad-hoc queries, in order to facilitate the reuse of execution plans. Moreover, in client server applications, they recommend the placement of the SET NOCOUNT ON command at the beginning of stored procedures, which leads to performance improvements, especially over slower networks. The reason behind this is that the command prevents the number of affected rows in stored procedures from being returned to the client, hence reducing the network traffic which comes as a result of packet exchanges from the server to the client.

Kumari [12] and Habimana [13] warn against the arbitrary and unnecessary usage of the expensive HAVING clause, which should be used only in cases which call for its original purpose, i.e. to filter out rows from a result set which don't fulfill the HAVING condition. Further, she suggests the creation of indexes on integer columns over those on character ones, whenever possible. This way, the number of index pages that store the index keys is reduced, since integer values are smaller in size than character values. Kumari also suggests that in cases of frequent joins between tables, index creation on the joined columns should be considered, as this can significantly improve the performance of the queries using the tables in question.

Blakely et al. [14] show that it is possible to make use of the integration of the .NET Common Language Runtime (CLR) into SQL Server in order to gain performance increases in certain scenarios. For example, the authors maintain that CLR scalar user-defined functions (which correspond to .NET static methods), outperform the equivalent Transact-SQL user defined functions (UDFs), especially for computationally complex functions. Moreover, since CLR UDFs are parallelizable, they allow for better performance in multiprocessor systems when compared to the T-SQL UDFs, which cannot execute in parallel.

Chaudhuri and Narasayya [15] have proposed techniques that aim to automate the management of statistics in databases. Statistics are relevant in the query optimization arena because the query optimizer component relies on them for generating query execution plans. The techniques presented by the authors help in avoiding the creation of non-useful statistics, as well as in identifying non-essential statistics that are preferred to be dropped in order to reduce overhead costs of statistics update.

The same authors in [16] have proposed techniques which aim to provide progress estimation for SQL query executions, maintaining that feedback on how much of a query's execution has been completed is very useful to the user/DBA, especially in cases of long-running queries. These last two studies were part of the AutoAdmin research project at Microsoft, which aims at making database systems more self-tuning and self-managing. A summary of studies and advances made at the AutoAdmin research project is available at [17].

3. BANKING SCENARIOS INVOLVING DATABASE OPERATIONS

In this section, three scenarios from the banking domain which involve database operations will be reviewed, followed by discussions related to the performance of the queries involved.

3.1. Safe usage of the nolock hint

Consider a bank client who has a saving account opened in a bank. The client also owns an e-banking service; namely, the client can log in to his e-banking account to view all of his accounts as well as account transactions.

Remember that a saving account is an account which earns the client positive interest. The accrued positive interest is paid to the client at the end of each month, but the bank, for accounting purposes, typically calculates the interest amount which it owes to the client incrementally on a daily basis. This daily accrual of interest in banks typically happens via some automatic daily process, which queries all the active saving accounts in the database and increments the accrued interest balance for them.

Back to our scenario - during this database operation of querying and updating the daily accrued interest of saving accounts, the client may log in to his e-banking account and request to see a report for his past saving account withdrawals. (ex. a report showing all the transactions from one week ago of the money withdrawals from his saving account). Since both operations - the interest calculation on the saving account, and the retrieval of past withdrawals from the saving account – will try to access the same tables in the database, blocking might take place, as described below:

- The interest calculation runs within a transaction block, and requests an exclusive lock on the table containing the saving accounts and the column for the accrued interest which needs to be updated;

- The retrieval operation for the report, in case it started after the operation above, would have to wait until the update transaction is complete and the exclusive lock released, in order to just read those past withdrawal transactions, which have finished one week ago and are not related at all to the interest amount.

So, even-though the two operations are not related to each other, the second operation would have to wait until the first one is finished.

To demonstrate this with T-SQL, suppose that we have a table called *SavingAccounts*, which among other attributes (columns), contains also a column named *AccruedInterest*. In order to simulate the duration of updating the *AccruedInterest* column for all active *SavingAccounts*, the WAITFOR command will be used, whereas the exclusive lock that would be placed on the *SavingAccount* table during the update process, will be simulated by placing the *tablockx* hint on that table:

```
BEGIN TRAN
BEGIN TRY

--simulate the setting of the x-lock during the update process
select top 1 *
from SavingAccounts with (tablockx)
--simulate the delay to complete the update
WAITFOR DELAY '00:00:04'

COMMIT TRAN
END TRY

BEGIN CATCH
     WHILE @@TRANCOUNT > 0
          ROLLBACK TRAN
END CATCH
```

Right after the transaction block above starts executing, the following statement is ran on another session, trying to retrieve the withdrawal transactions from a saving account, that have completed the previous week:

```
SELECT wd.*
FROM SavingAccounts AS sa JOIN Withdrawals AS wd
ON wd.AccountId = sa.Accountd
WHERE sa.AccountId=220693 AND Date
BETWEEN DATEADD(d,-14, getdate()) AND DATEADD(d,-7,getdate())
```

This retrieval operation will be waiting until the previous statement is complete, even though it is not interested in the accrued interest at all. In order to overcome this unnecessary waiting, the *nolock* hint is introduced on the *SavingAccounts* table during the retrieval operation, as follows:

```
SELECT ws.*
FROM SavingAccounts AS sa (nolock) JOIN Withdrawals AS ws
ON ws.AccountId = ws.AccountId
WHERE sa.AccountId=220693 AND Date
BETWEEN DATEADD(d,-14, getdate()) AND DATEADD(d,-7,getdate())
```

With the *nolock* hint in place, the same query completes considerably faster, as can be seen from table 1, which summarizes the execution times in seconds for the 3 queries presented above.

This is a typical scenario where introducing the *nolock* hint would be safe and advantageous. The *nolock* hint in Microsoft SQL Server is equivalent to the READ UNCOMMITED transaction isolation level in SELECT statements, and as such, rightfully raises many concerns about the dangers of using it. The dangers of the READ UNCOMMITTED isolation level are that it allows the reading of "dirty" data, which has not committed yet and can be potentially rolled back. Because of this, the *nolock* hint should be indeed used sparingly and with much care, because it might lead to inconsistent and even false data.

Table 1: Query execution times

Query	Duration (seconds)
UPDATE query	4.019
SELECT query	4.141
SELECT query with nolock	0.128

In the above scenario however, when a report is involved that displays data related to transactions that have already committed and cannot be further updated, using the *nolock* hint is completely safe and will produce an improvement in performance.

3.2. Avoiding unnecessary casts that prevent index usage

Another scenario that will be investigated is related to the character data types, such as char and varchar, and the implications their misuse in equality comparisons might have on the decision of the query optimizer whether to use an index scan or an index seek.

Consider the case of a banking database schema which defines a table named *ResidentClients*, for storing the necessary information related to bank clients who are residents of the country the bank is located in. One field of this table would typically be *NationalIdentification Number*, that would hold, as the name suggests, the national identification number (also known as Tax Code in certain countries) of the client. Depending on the country regulations, this ID would either consist of alphanumerical characters, or would only contain numerical characters. The database schema defines this column's data type as varchar, in order to be able to support banks in countries with different regulations. The schema in question also defines a nonclustered index on the *NationalIdentification Number* column.

Now, let us consider a bank using the database schema described above, in a country whose regulations impose that only digits are to be used in national identification numbers. This means that no records in the *ResidentClients* table would contain alpha characters.

Hence, both of the following SELECT statements would be valid in this case:

```
SELECT * FROM ResidentClients
WHERE NationalIdentityNumber=123456789

SELECT * FROM ResidentClients
WHERE NationalIdentityNumber='123456789'
```

However, the execution plan for the first SELECT statement differs from the one belonging to the second SELECT statement, as can be seen in the two figures below:

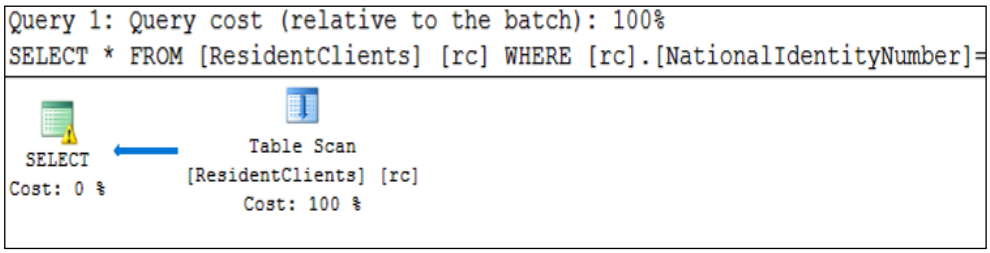

Fig. 1. Execution plan when casting is involved

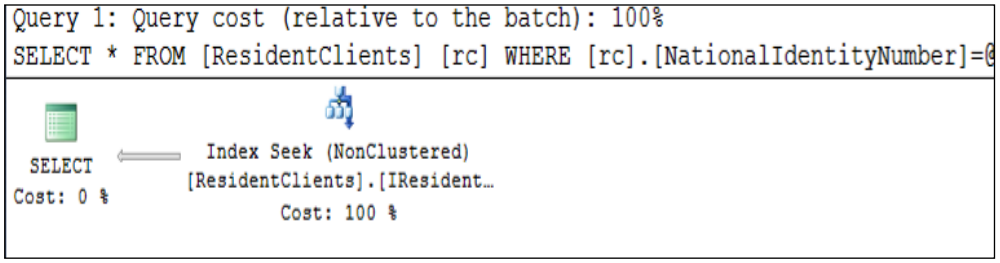

Fig. 2. Execution plan when casting is not involved

It can be observed that the index on the *National IdentificationNumber* column is being used only in the second case, because in the first one, the casting to varchar that SQL Server implicitly makes, prevents the index usage and forces a table scan, which can be a much more expensive operation.

Consequently, care must be taken in order to avoid unnecessary cast operations which are detrimental to performance by preventing the appropriate indexes from being used.

3.3. Comparing performance when temporary result sets are used

Generating different types of reports for national banks, financial regulator institutions and relevant stakeholders is typically mandatory for all banks. Thus, reports generation is another process in the banking industry, which is dependent on databases, and optimizing the data retrieval queries for reporting needs is essential.

Queries used for reporting purposes tend to be long and somewhat complicated, usually involving data preparation steps (i.e. preparing temporary result sets with data from several tables that fulfill certain conditions), and then using these result sets in conjunction with other data in order to provide the final result sets. In preparing these temporary result sets, different SQL options can be used: table variables, temporary tables, common table expressions (CTEs) etc.

In this section, we are going to make a comparison between the performances of table variables against CTEs in storing temporary result sets, which are used for reporting purposes. A query sample using a table variable is presented below:

```
DECLARE @Condition_fulfilling_clients TABLE (ClientId INT)

INSERT INTO @Condition_fulfilling_clients (ClientId)
SELECT rc.ClientId
FROM SavingAccounts AS sa JOIN ResidentClients AS rc ON sa.Own-
erId=rc.ClientId
```

```
SELECT *
FROM ResidentClients AS rc JOIN @Condition_fulfilling_clients AS
cfc ON rc.ClientId=cfc.ClientId
WHERE rc.CityOfBirth = 'Skopje'
```

The equivalent of this query, but using a CTE instead of the table variable is presented next:

```
WITH Condition_fulfilling_clients
AS (
      SELECT rc.ClientId
      FROM SavingAccounts AS sa
      JOIN ResidentClients AS rc ON sa.OwnerId=rc.ClientId
      )

SELECT *
FROM ResidentClients AS rc JOIN Condition_fulfilling_clients AS
cfc ON rc.ClientId=cfc.ClientId
WHERE rc.CityOfBirth='Skopje'
```

Both queries are executed together in a batch in SQL Server 2012, and the corresponding execution plan is given in Fig.3.

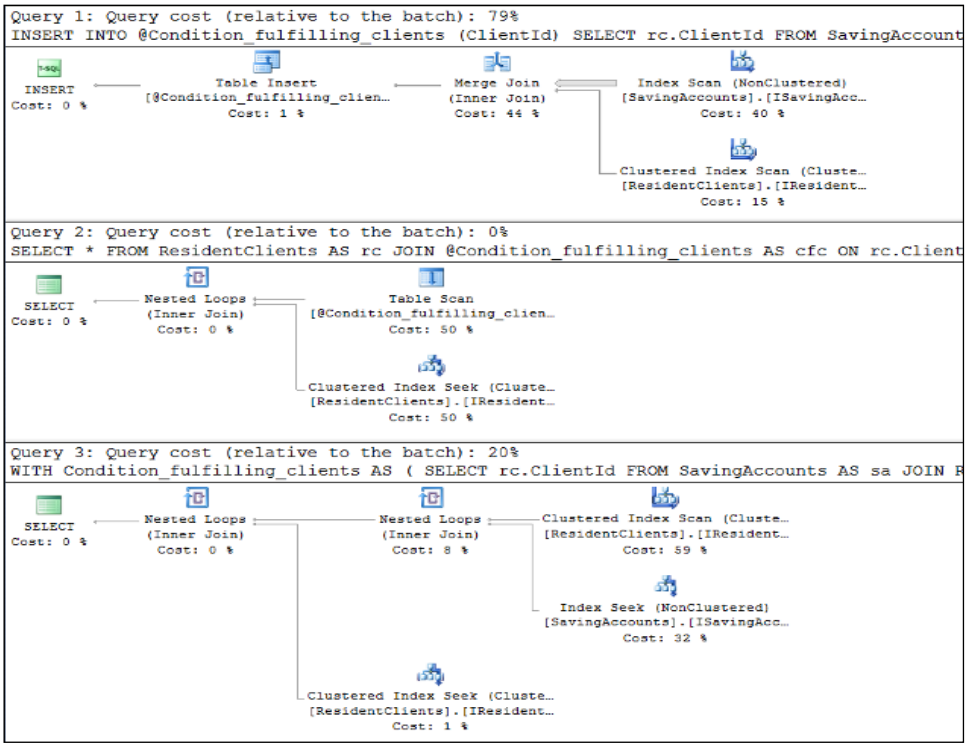

Fig. 3. Relative query costs when using table variable and CTE

The execution plan shows that using a CTE in our case costs around 4 times less than using a table variable to store the same results, and this marks a significant performance increase. It must be noted however, that these ratios might differ significantly when different tables and/or filtering conditions are used, because execution plan costs vary depending on index availability/ usability in order to complete the task in hand.

So, the point to be made here is that when fast retrieval of data is critical, and temporary result sets are involved, it is worthwhile to spend some time in examining the query costs of the alternative techniques available in SQL Server for completing the required operations.

4. CONCLUSIONS

Software solutions in the banking domain rely heavily on databases for performing daily business operations, and render the query optimal performance critical and tuning-worthy. When attempting to optimize query performance, it is important to consider the context and particularities of the application domain of the database operations. This way, the provided performance improvement solutions respond to both the requirements and limitations of the specific domain in question.

This paper reviewed three scenarios from the banking domain, along with the database operations involved in each. It started with providing a case where the *nolock* hint usage was both safe and successful in removing unnecessary wait times in retrieval operations. The usage of the *nolock* hint however, must remain rare because of the dangers associated with reading uncommitted data, and should be restricted to cases when only the retrieval of already committed data that is not subject to further change is required.

The second scenario stressed the importance of avoiding unnecessary casts which prevent index usage and degrade performance.

The final scenario described the importance of investigating the cost of different techniques available in Microsoft SQL Server for completing a task. It demonstrated an example case where the usage of a common table expression (CTE) was faster than the usage of a table variable, and stressed the value in performing such cost comparisons when preparing mission critical retrieval operations that require fast completion.

REFERENCES

[1] B. Ubiparipović, E. Đurković. Application of Business Intelligence in the Banking Industry. *Management Information Systems*, Vol. 6, No. 4, 2011, pp. 023 – 030.

[2] M. Rys. Scalable SQL. *Magazine Communications of the ACM*, Vol.54, No.6, 2011, pp. 48-53.

[3] R. Elmasri, Sh. B. Navathe. *Fundamentals of database systems*. 7th edition, London, UK: Pearson Education Limited, 2016.

[4] A. Silberschatz, H. F. Korth, S. Sudarshan, A. B. Tucker. *Database Systems Concepts*, 6th Edition, New York, NY: McGraw-Hill Education, 2010.

[5] G. R. Bamnote, S. S. Agrawal. Introduction to Query Processing and Optimization. *International Journal of Advanced Research in Computer Science and Software Engineering*, Volume 3, Issue 7, July 2013, pp. 53 – 56.

[6] M. L. Rupley, Jr. *Introduction to Query Processing and Optimization*. 2008, http://cs.iusb.edu/technical_reports/TR-20080105-1.pdf. (from web, January 2018)

[7] S. Chaudhuri. An Overview of Query Optimization in Relational Systems.in *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Seattle, Washington, USA, 1998, p.34-43.

[8] V. K. Myalapalli, P. R. Savarapu. High Performance SQL. *11th IEEE India Conference on Emerging Trends and Innovation in Technology*, 2014, pp. 1-6.

[9] T. Oktavia, and S. Sujarwo. Evaluation of sub query performance in SQL server. *International Conference on Advances Science and Contemporary Engineering*, EPJ Web of Conferences, vol. 68, 2014.

[10] I. Lungu, N. Mercioiu, and V. Vladucu. Optimizing Queries in SQL Server 2008. *Scientific Bulletin – Economic Sciences*, Vol. 9, No. 15, 2010, pp. 103-108.

[11] C. G. Corlatan, M.M. Lazar, V. Luca, O. T. Petricica,. Query Optimization Techniques in Microsoft SQLServer. *Database System Journal*, 2014, Vol. 5, No. 2, pp. 33-48.

[12] N. Kumari. SQL Server Query Optimization techniques - Tips for Writing Efficient and Faster Queries. *International Journal of Scientific and Research Publications*, Vol. 2, No. 6, 2012, pp. 1 - 4.

[13] Jean Habimana. Query Optimization Techniques – Tips for Writing Efficient and Faster SQL Queries. *International Journal of Scientific and Technology Research*, vol. 4, issue 10, October 2015, pp. 22 – 26.

[14] J. A. Blakeley, V. Rao, I. Kunen, A. Prout, M. Henaire, Ch. Kleinerman. .NET Database Programmability and Extensibility in Microsoft SQL server. *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008, pp. 1087–1098.

[15] S. Chaudhuri, V. Narasayya. Automating Statistics Management for Query Optimizers. *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, issue 1, 2001, pp. 7-20.

[16] S. Chaudhuri, V. Narasayya,R. Ramamurthy. Estimating Progress of Long Running SQL Queries. *SIGMOD 2004*, pp. 803 – 814.

[17] N. Bruno, S. Chaudhuri, A.Ch. König, V. Narasayya, R. Ramamurthy, M. Syamala. AutoAdmin Project at Microsoft Research: Lessons Learned. *IEEE Data Engineering Bulletin*, vol. 34, 2011, pp. 12 – 19.

Information about the authors:

Blerta Haxhijaha is currently studying for her postgraduate in Software and Application Development at South East European University. She has been involved in banking software implementation projects in banks in several countries in Europe. Her current research interest is in databases and data processing.

Jaumin Ajdari – Assoc. Prof. at Faculty of Contemporary Sciences and Technologies, South East European University. His current research interest is in parallel processing, data processing, databases and data analytics.

Bujar Raufi – Assoc. Prof. at Faculty of Contemporary Sciences and Technologies, South East European University. His current research interest is in adaptive web, semantic web, computer graphics and data analytics.

Xhemal Zenuniu – Assist. Prof. at Faculty of Contemporary Sciences and Technologies, South East European University. His current research interest is in semantic web, contemporary distributed systems, intelligent agent and data analytics.

Florije Ismaili – Assoc. Prof. at Faculty of Contemporary Sciences and Technologies, South East European University. His current research interest is in web services, cloud computing and information retrieval.

Manuscript received on 13 January 2018