Journal of Computer Science 6 (5): 585-590, 2010
ISSN 1549-3636
© 2010 Science Publications

Increasing Database Performance through Optimizing Structure Query Language Join Statement

[1]Ossama K. Muslih and [2]Imad Hasan Saleh
[1]Department of Software Engineering, Faculty of Science and Information Technology, Alzaytoonah University of Jordan, Jordan
[2]Department of Computer Science, Faculty of Information Technology
Amman Arab University of Jordan, Jordan

Abstract: Problem statement: A join statement is a select statement with more than table in the FROM clause. A join predicate is a predicate in the WHERE clause that combines the columns of two of the tables in the join. Any database gives you the ability to join various tables together through different types of joins, resulting large number of rows to process. Query language can be used to join these tables and as it is well known query language should be declarative, so we can write alternative formulas to perform join statements. Different formulas provide variation in performance. **Approach:** This research presented a transparent middle layer between application interface front end and database back end. **Results:** The responsibilities of this layer were catching the SQL commands sent by application before reaching the database then examining these commands to see if they join more than one table, after that rewriting the SQL command taking into consideration the order of executing join predicates and none join predicates. This research focused on rewriting the SQL commands without application modification. **Conclusion:** Rewriting stage is the most complex stage because the system will restructure the SQL command with new syntax taking two things in its consideration, the first one was rewriting the command with better performance syntax after getting the help from recommendation dictionary, the second one was resulting the same data (output) as previous old command.

Key words: Join predicate, driving table, inner table, nested loop join, sort-merge join and materialized views

INTRODUCTION

Most of relational database management systems RDBMSs provide facilities to see the execution plan for any SQL statement. And from these execution plans we can see the problem of the command and why it takes more time that it should take. A poor command can be discovered by monitoring some facts such as, the command which makes a lot of disk access or makes full table scan to read a small number of data. Another fact is the execution plan shows that the SQL command do not use indexes related to selected table (Bryan and Hinze, 2003).

The output of a query optimizer for a declarative query statement is called a Query Execution Plan (QEP). The structure of a QEP determines the order of operations for query execution. The QEP is typically represented using a tree structure where each node represents a physical database operator (e.g., nested loop join and table scan). Multiple plans may exist for the same query and it is a query optimizer's top priority to choose an optimal plan. To supplement the QEP, most query optimizers produce performance related information such as cost information, predicates, selectivity estimates for each predicate and statistics for all objects referenced in the query statement. Figure 1 shows an example of nested loop join statement (Chris *et al.*, 2003).

In nested loop joins one of the tables defined as outer table (driving table), the other PDF created with table called inner table and for each row in the outer table all matching rows in the inner table are retrieved. Another way to join two tables is a sort-merge joins. Figure 2 shows an example of sort-merge joins (Ramez and Navathe, 2004).

In sort-merge joins, the two row sources are sorted on the values of the columns used in the join predicate. If a row source has already been sorted in a previous

Corresponding Author: Ossama K. Muslih, Department of Software Engineering, Faculty of Science and Information Technology, Alzaytoonah University of Jordan, Jordan

operation, the sort-merge operation skips the sort on that row source. Sorting could make this join technique expensive, especially if sorting cannot be performed in memory. The merge operation combines the two sorted row sources to retrieve every pair of rows that contain matching values for the columns used in the join predicate (Kephart and Chess, 2003).

Nested loop joins used when we have small number of rows that have a good driving condition between the two tables. But a sort-merge joins used for large amounts of data or the join condition between two tables is not an equijoin. From these two examples we find that, we can rewrite the SQL statement (change the original statement from old syntax to new syntax) and use optimizer hints before sending these statements to optimizer. If we rewrite the statement in proper way, we can make the database optimizer take a good decision and the best execution plan for executing the statement.

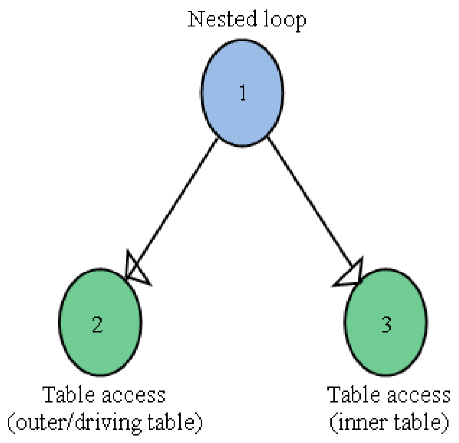


Fig. 1: Nested loops join

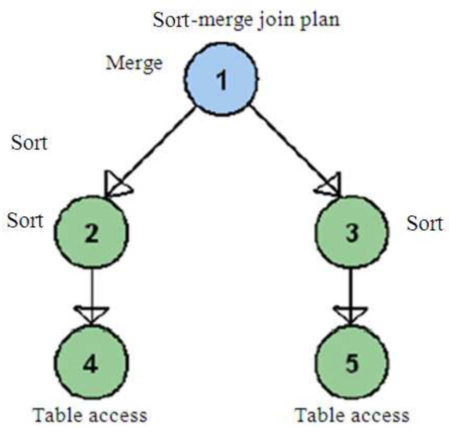


Fig. 2: Sort-merge joins

The problem description: A join statement is a select statement with more than one table in the FROM clause. A join predicate is a predicate in the WHERE clause that combines the columns of two of the tables in the join. A non join predicate is a predicate in the WHERE clause that references only one table. As in the following example if we want to join three tables (CLIENTS) table which contains information about our clients and (CONFERENCES) table which contains information about conferences done by clients and (CONFERNCE_DETAILS) table which contains the details of the conference. The WHERE clause (WHERE CLN.CLN_ID = CNF.CNF_CLN_ID_CALLER) and (AND CNF_CNF_ID = CNFD.CNFD_CNF_ID) are join predicate but (AND CLN_CLN_ID = 2323) is non join predicate.

```
SELECT * FROM
   CLIENTS CLN, CONFERENCES CNF,
   CONFERNCE_DETAILS CNFD
WHERE CLN.CLN_ID =
      CNF.CNF_CLN_ID_CALLER
AND CNF.CNF_ID =
      CNFD.CNFD_CNF_ID
AND CLN.CLN_ID = 2323
```

Determining the sequence of joining more than one table is very important decision, joining two tables like making a nested loop, so each record in the first table (outer or driving table) will be matched with each record in the second table (inner or drive table). The important thing in joining is placing the non join predicate in the first of order of join predicate. By making a non join predicate table the driving table of a join operation, the RDBMS effectively reduces join operation. For example, for a nested loop join, the main loop is reduced. The non join predicate results in less rows (or no rows at all), so the inner loop is executed less (or not at all) (Priya, 2003).

Let us take an example with numbers to see how much important to select the driving table first. For previous SELECT statement let us consider these facts for three tables as summarized in Table 1.

Table 1: Tables summary

	(CLN)	(CNF)	(CNFD)
Number of Rows	100,000	1,000,000	10,000,000
Rows for client ID (2323)	1	10,000	100,000

If the database starts by joining CNF table with CNFD table the join results by (1,000,000*10,000,000) 1E+13 loops, in spite of that not all data in CNF and CNFD tables belongs to CLIENT number 2323 as in the SELECT statement, so the database joining the two tables for all clients then the database will join the resulting data (1E+13) row with CLN table after applying the non join predicate (AND CLN_CLN_ID = 2323) which will result for one record, the total loops will be (1E+13*1) for joining the three tables.

But if the database starts by joining CLN table with CNF table after applying the non join predicate (AND CLN_CLN_ID = 2323), the join results by (1*10,000) loops, then joining the third table CNFD, the number of loops will be (10,000*100,000) 1E+9 loops which is of course less than the previous method.

The proposed solution: The proposed solution will follow these steps, as in the Fig. 3 taking into consideration that the repository is already built and will be configured:

1. First we have to connect to target database and consolidate the repository with target database to get any changes or modifications, then the system waits for any SQL command from application layer
2. The system analyzes SQL command to see if the command is using sequential search. Also the system may find that the SQL command does not need tuning in this step
3. The system fetches the recommendation dictionary; Basically this dictionary contains rules for writing the SQL command in tuned syntax. If there is any rule satisfies the SQL command the system will send the command to rewrite stage else the system will leave the command as it is

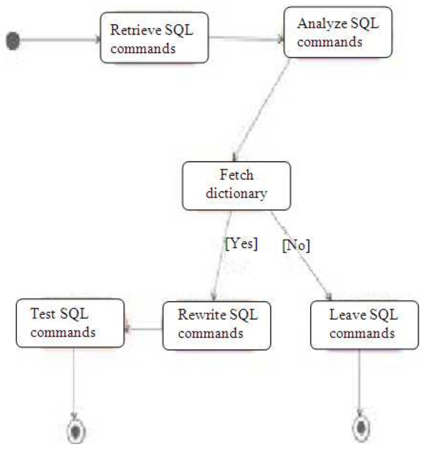


Fig. 3: Proposed system design

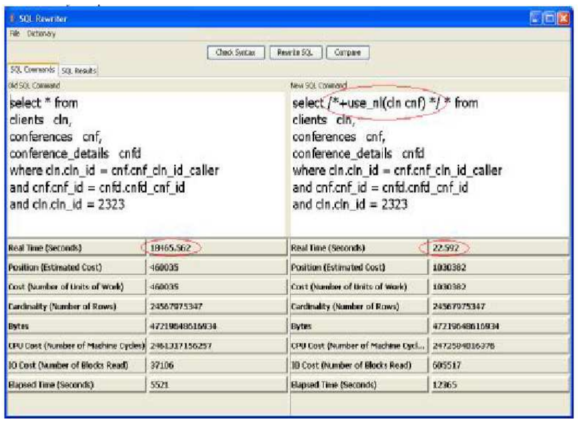
4. The system will leave the SQL command as it is if there is no rule or recommendation satisfies this command
5. Rewriting stage is the most complex stage because the system is going to restructure the SQL command with new syntax taking two things in its consideration, the first one is rewriting the command with better performance syntax after getting the help from recommendation dictionary, the second one is resulting the same data (output) as previous old command (Chang *et al*., 2000)
6. The system will test the new SQL command and compare it with old one to see difference in time, IO, network roundtrips, execution time and many

MATERIALS AND METHODS

The optimizer of the database will not start by none predicate join unless the programmer writes the none join predicate first in the WHERE clause or writing hints in the SQL statement (John, 2002).

To make database optimizer takes the decision of starting by non join predicate table, the programmer must write hints for optimizer. Hints can be written with any SQL command after the first word of the statement, optimizer hints must by start with (/*+) and end by (*/). As an example of using hints to make the database optimizer starts by joining CLN table and CNF table using nested loop operation as following:

```
SELECT /*+USE_NL (CLN CNF)*/ *
FROM
    CLIENTS CLN, CONFERENCES CNF,
    CONFERNCE_DETAILS CNFD
WHERE CLN.CLN_ID =
                CNF.CNF_CLN_ID_CALLER
AND CNF.CNF_ID =
                CNFD.CNFD_CNF_ID
AND CLN.CLN_ID = 2323
```



```
select * from
clients   cln,
conferences   cnf,
conference_details   cnfd
where cln.cln_id = cnf.cnf_cln_id_caller
and cnf.cnf_id = cnfd.cnfd_cnf_id
and cln.cln_id = 2323
```

```
select /*+use_nl(cln cnf) */ * from
clients   cln,
conferences   cnf,
conference_details   cnfd
where cln.cln_id = cnf.cnf_cln_id_caller
and cnf.cnf_id = cnfd.cnfd_cnf_id
and cln.cln_id = 2323
```

Real Time (Seconds)	18465.562	Real Time (Seconds)	22.592
Position (Estimated Cost)	460035	Position (Estimated Cost)	1030382
Cost (Number of Units of Work)	460035	Cost (Number of Units of Work)	1030382
Cardinality (Number of Rows)	24567975347	Cardinality (Number of Rows)	24567975347
Bytes	4721984861 6934	Bytes	472196488610934
CPU Cost (Number of Machine Cycles)	2461317156257	CPU Cost (Number of Machine Cycl...	2472584016378
IO Cost (Number of Blocks Read)	37106	IO Cost (Number of Blocks Read)	605517
Elapsed Time (Seconds)	5521	Elapsed Time (Seconds)	12365

Fig. 4: Join rewriting

The proposed system will try to find any SQL statement with join predicate and examine if the statement has non join predicate, the system will rewrite the statement and send a hint to database optimizer to start with non join predicate table as shown in Fig. 4.

The system will try to do this process, so the system will tokenize the statement and searches for WHERE clause, then searches if there are joins experimental scenarios were considered to evaluate and compare performance by running join statements without any attention if the none join predicate will be used at the first or not. Then watch the system how it will convert the join statement to use none join predicate as a driving table (Jiao and Hurson, 2002).

First scenario: Before rewrite: In this scenario we sent join SQL statements to the system without any attention of none join predicate, the following SQL statements are just an example of this type of statements:

```
SELECT * FROM
            COUNTRIES CON, CLIENTS CLN,
            CONFERENCES CNF
WHERE CON.CON_ID =
     CLN.CLN_CON_ID
AND CLN.CLN_ID =
     CNF.CNF_CLN_ID_CALLER
AND CON.CON_NAME = ‘JORDAN’;
```

```
SELECT * FROM
     CLIENTS CLN, CONFERENCES CNF,
     CONFERENCE_DETAILS CNFD
WHERE CLN.CLN_ID =
     CNF.CNF_CLN_ID_CALLER
AND CNF.CNF_ID =
     CNFD.CNFD_CNF_ID
AND CLN.CLN_ID = 2323;
```

Second scenario: After rewrite: In this scenario we watched the transformed SQL commands generated by the system that makes none predicate join as a driving table:

```
SELECT /*+USE_NL(CON CLN) */ * FROM
     COUNTRIES CON, CLIENTS CLN,
     CONFERENCES CNF
WHERE CON.CON_ID =

     CLN.CLN_CON_ID
AND CLN.CLN_ID =
     CNF.CNF_CLN_ID_CALLER
```

predicates and non join predicates. If exists the system will add a hint to SQL statement to start with non join predicate. To see the achievement done by the system when converting the command, we have to execute the SQL command before rewriting process then collect all metrics and statistics related to this command, after that we have to execute the SQL command after the rewriting process and collect the same metrics and statistics, then compare all of these outputs. The two

```
AND CON.CON_NAME = 'JORDAN';
```

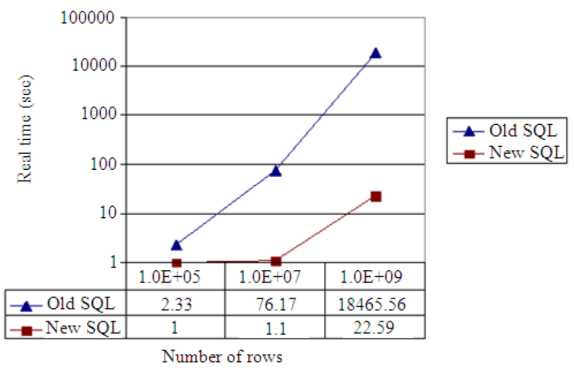
```
SELECT /*+USE_NL(CLN CNF) */ * FROM
     CLIENTS CLN, CONFERENCES CNF,
     CONFERENCE_DETAILS CNFD
WHERE CLN.CLN_ID =
     CNF.CNF_CLN_ID_CALLER
AND CNF.CNF_ID =
     CNFD.CNFD_CNF_ID
AND CLN.CLN_ID = 2323;
```

RESULTS AND DICUSSION

The above SQL statements are generated from the system as a result of the SQL statement sent in the first scenario in the order. The benchmark depends on two things:

- The number of rows that the table holds
- The number of rows that will be filtered by none join predicate

So the two scenarios were repeated with different number of rows for three tables and different number of rows filtered by none join predicate. Table 2 shows the total number of rows and filtered rows in the three tables when the run executed and the output metrics generated for each run. Table 3 shows the total number of rows and filtered rows in the three tables when the run executed and the output statistics generated for each run.



	1.0E+05	1.0E+07	1.0E+09
Old SQL	2.33	76.17	18465.56
New SQL	1	1.1	22.59

Fig. 5: Logarithmic chart for real time results comparison

Table 2: Metrics for join

	Number of rows for three joined tables					
Total rows	1E+12		1E+15		1E+18	
Filtered row	1*100*1000		1*1000*10000		1*10000*100000	
Metrics	Old	New	Old	New	Old	New
Real time (sec)	2.33	1.00	76.17	1.10	18465.56	22.59
Position (estimated cost)	5984	385	41835	3736	460035	1030382
Cost (units of work)	5984	385	41835	3736	460035	1030382
Cardinality (number of rows)	93966	93966	794461	794461	24E+9	24E+9
Bytes	18E+7	18E+7	15E+8	15E+8	47E+12	47E+12
CPU cost (machine cycles)	69E+7	56E+6	44E+8	48E+7	24E+11	24E+11
IO cost (blocks read)	5866	375	41068	3653	37106	37106
Elapsed time (sec)	72	5	503	45	5521	12365

Table 3: Statistics for join

	Number of rows for three joined tables					
Total rows	1E+12		1E+15		1E+18	
Filtered rows	1*100*1000		1*1000*10000		1*10000*100000	
Statistics	Old	New	Old	New	Old	New
Recursive calls (number of SQL)	0	0	7	7	576	576
DB block gets (number of IO)	0	0	0	0	0	0
Consistent gets (number of buffer)	154116	1656	15E+6	83277	20E+8	826345
Physical reads (number of blocks)	0	0	0	0	1665661	1665661
Redo size (number of blocks)	0	0	0	0	0	0
Bytes sent	1975	1975	10E+7	10E+7	93E+7	93E+7
Bytes received	374	274	733711	733711	7333711	7333711
Net. roundtrips (count)	1	1	66668	66668	666668	666668
Sorts (memory) **(Count)**	0	0	2	2	4	4
Sorts (disk) (count)	0	0	0	0	0	0

SQL command real time execution is the most important measurement used to evaluate the performance between old command and new command. Figure 5 illustrates a logarithmic chart comparison.

CONCLUSION

Most of the researches focus in deep in the area of SQL command performance enhancements. Some of them focused on enhancing database optimizer capabilities, others focused on rewriting SQL command by using materialized views. But all of them focused in the area after the database captures the SQL command. This study did the opposite side; it focused in the area before the SQL command reaches the database. This study tried to enhance the performance of the database by sending a well done, error free and a professional SQL commands.

The real time results are plotted as logarithmic chart has been analyzed the comparisons detailed as follows:

- The real time coefficient for old SQL commands increased dramatically according to number of rows in the table, because the old SQL will process the multiplication of all rows in the three joined tables
- The real time coefficient for new SQL commands increased slightly with increasing number of rows, because the new SQL will process just the filtered rows from three joined tables

There is no semantic reasoning engine built in to SQL optimizers, next steps in research will include the completion and expand the logical design of our method, followed by its implementation.

REFERENCES

Bryan, G. and A. Hinze, 2003. Open Issues in Semantic Query Optimization in Relational DBMS. Department of Computer Science, University of Aikato, New Zealand. http://www.cs.waikato.ac.nz/~hinze/isdb/publications/genet_hinze_TR102004.pdf

Chang, S.P., M.H. Kim and Y. Lee, 2000. Rewriting LAP queries using materialized views and dimension hierarchies in data warehouses. Supervised by IITA.

Chris, M.G., M.M. Dalkilic, D.P. Groth and E.L. Robertson, 2002. Improving query evaluation with approximate functional dependency based decomposition. Lecture Notes Comput. Sci., 2405: 26-41. DOI: 10.1007/3-540-45495-0_3

Jiao, Y. and A.R. Hurson, 2002. Mobile agents in mobile data access systems. Lecture Notes Comput. Sci., 2915: 144-162. DOI: 10.1007/3-540-36124-3_9

John, P.M., 2002. Aggregate navigation using materialized views and query rewrite. Counterpoint Technologies, Inc. http://www.nyoug.org/Presentations/SIG/DataWarehousing/aggrtnav.pdf

Kephart, J.O. and D.M. Chess, 2003. The vision of autonomic computing. IEEE Comput., 36: 41-52.

Priya, V., 2003. Oracle Database SQL Tuning Workshop. Oracle Corporation.

Ramez, E. and S.B. Navathe, 2004. Fundamentals of Database Systems. 4th Edn., Pearson Addison Wesley, ISBN: 9780321122261.