SQL OPTIMIZATION IN A PARALLEL PROCESSING DATABASE SYSTEM

Nayem Rahman
Intel Corporation, Email: nayem.rahman@intel.com

Abstract—**A database management system (DBMS) with a parallel processing database system is different from conventional database systems. Accordingly, writing SQL for a parallel processing DBMS requires special attention to maintain parallel efficiency in DBMS resources usage such as CPU and I/O. This paper discusses the techniques in SQL writing, tuning, utilization of index, data distribution techniques in a parallel processing DBMS architecture. The resource savings statistics based on several experiments show significant reduction of computing resources usage and improvement of parallel efficiency (PE) can be achieved by using different optimization techniques.**

***Index Terms*—Database System; DBMS; Computing Resources; CPU; I/O; Data Warehouse; Parallel Processing**

I. INTRODUCTION

Data warehouse systems resources are destined for use by reporting, analytical and Business Intelligence (BI) tools, enabling business people to make all sorts of decisions based on data warehouse information. It is critical that enough computing resources be available for use by the analytical community to retrieve and process information into intuitive presentations (i.e., reads). This means that the data warehouse batch processing should use the minimum resources possible.

There are two aspects that we need take into consideration in maintaining parallel processing capability in a data warehouse. First, in the load process we need to make sure that the database system resources such as CPU time and IO utilization are minimal. This can be done by improving the parallel efficiency of SQL. Second, we need to make sure the SQL used by analytical tools and ad-hoc queries are parallel efficient. The purpose of this paper is to show how data warehouse load SQL and analytical reporting SQL could be made efficient by focusing on and taking advantage of parallel processing architecture of database system.

II. LITERATURE RESEARCH

Data warehousing and data management are considered as one of the six physical capability clusters of IT-infrastructure services [19]. A lot of research work has been done on different areas of data warehousing during the last one decade. Several research work on data warehousing focused on data warehouse design [7, 1, 3, 14, 4, 5, 6], ETL tools [9, 15, 17], data maintenance [11], view materialization [2, 8, 10, 12, 13, 21, 22], and implementation [20, 24]. In this paper, we present techniques to cause load processes to consume fewer resources. We focus on writing efficient SQL that conforms to parallel processing architecture. We address the problem of DBMS resource consumption by ensuring SQL efficiency, defining indexes. In order to reserve more resources for queries we propose writing SQL in such a way that it takes advantages of parallel processing architecture of a data warehouse system to save computing resources in batch processes thus releasing them for analytical queries.

III. OPTIMIZATION IN A PARALLEL PROCESSING DATABASE SYSTEM

In a parallel processing DBMS architecture a large number of individual access module processors (AMP) are used. Several AMPs reside in a node. The database engine dispatches optimized SQL plan to AMPs in a multimode system. These AMPs work in parallel – "shared nothing" architecture [18]. Each AMP holds individual data storage. Data in a table reside in all or most of the AMPs. So individual AMP is in full control of a portion of a database and maintains its portion of table data on disks. During a table-load the database engine sends data to different AMPs for storage based on index column(s) distinct values. When it comes to data retrieval the database engine instructs the AMPs to return data to the user based on data-request by a particular SQL. Thus data load and retrieval can be done in parallel. Given the data of any table need to be stored in different AMPs index column(s) plays a critical role. The index must be created such a way that it facilitates parallelism in data storage, data retrieval and join operations between tables.

A. Row Redistribution in a Parallel Processing DBMS

A parallel processing database system database joins 2 tables (or spool files) at a time and puts the result into a spool file. Then it joins that to another table or spool file, and so on until all the tables are joined. In each of these joins the rows to be joined on each table must reside on the same AMP. If the 2 tables have the same primary index (PI) then all the rows that will join together already reside on the same AMP. If the 2 tables have different PI's (primary index) the DBMS needs to do one of two things: either duplicate (one of the) table(s) on all AMPs or redistribute one of the tables (using a PI that is the same as the other table) so that the rows being joined now reside on the same AMP. So the reason for redistribution is always that the 2 tables being joined do not have the same PI. Sometimes we cannot do anything about this; it is just the way it works. Other times, we can build a derived table, narrowing the selection of rows to a smaller number, and try to make DBMS duplicate the table on all AMPS. If it does not disturb other processes; the best way to eliminate redistribution is to build the tables being joined with the same PI (this is not always possible).

B. Duplicating on All AMPs & Product Join

If table-joins have different PI's or they contain skewed data, DBMS copies data to all AMPs for parallel processing. We cannot avoid it in many situations. To ensure that the duplication takes less resources temporary tables could be used with reduced row and column numbers. Product Join happens when a 'JOIN' occurs between a large and a small table. To improve performance two things can be done:

narrow down the rows and columns of that small table; if the smaller table contains static data with few records in that case column values could be placed in memory variables. That way the JOIN with the smaller table could be entirely eliminated.

C. Primary Index (PI) Choice Criteria

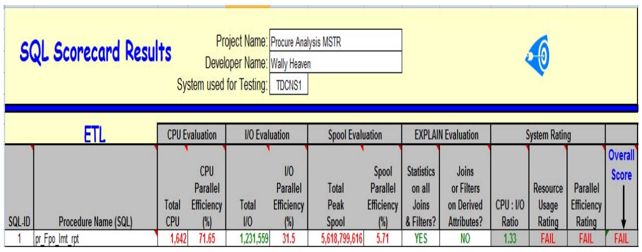
The primary index selection should be based on data access, distribution and volatility. In regards to access demographics, columns that would appear with a value in a WHERE clause need to be considered. Need to choose the column most frequently used for access to maximize the number of one-AMP operations. To ensure data distribution demographics the more unique the index values, the better the distribution. To avoid data volatility, the data values should not often change for an index column. Any changes to PI values may result in heavy I/O overhead. Join activity against the PI definition. For large tables, the number of distinct Primary Index values should be much greater than the number of AMPs.

D. Sync up Source & Target Table PI's

Common PI's between source and target tables help bulk inserts. The DBMS optimizer performs PI-based MERGE JOINs. In a large join operation, a merge join requires less I/O CPU time than nested join. A merge join usually reads each block of the INNER table only once, unless a large number of hash collisions occur. In a real world scenario we noticed that due to missing common PI's the SQL of a stored procedure became 90% skewed. It pulled records from two large tables with several join columns. Run time was 5 hours and 6 minutes to load 9 million rows. After PI synchronization the run-time dropped to 1 minute 11 seconds.

E. Global Temporary Tables vs. Derived Tables

The solution to some of the resource intensive queries includes conversion of a derived table (DT) to a global temporary table (GTT). This is because the GTT can have statistics collected whereas the DT cannot. The GTT approach makes the optimizer plans more aggressive and rely more heavily on collected stats as opposed to sampled statistics. As in all of life, there is trade-offs: relying on collected stats would produce better running queries than the random samples. With data skew, the random samples were often wrong and caused wrong choices to be made. We can achieve better performance plans for tables (GTT) with collected statistics. We cannot collect statistics on derived or volatile tables so these do not perform as well. Figure 1 shows performance results of an SQL that used derived tables. The result shows that per evaluation criteria the SQL failed in terms of computing resources usage such as CPU, IO and spool space usage. Their parallel efficiencies are very poor.

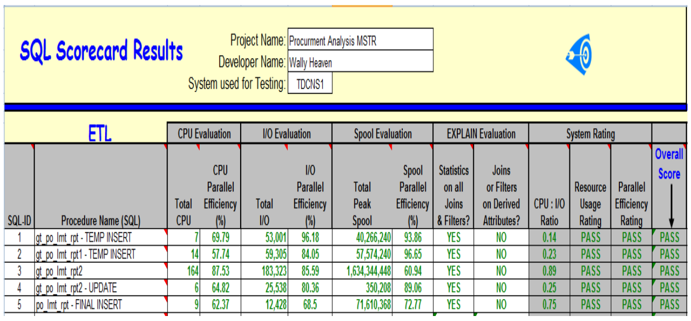

SQL Scorecard Results

Project Name: Procure Analysis MSTR
Developer Name: Wally Heaven
System used for Testing: TDCNS1

ETL		CPU Evaluation		I/O Evaluation		Spool Evaluation		EXPLAIN Evaluation		System Rating			
SQL-ID	Procedure Name (SQL)	Total CPU	CPU Parallel Efficiency (%)	Total I/O	I/O Parallel Efficiency (%)	Total Peak Spool	Spool Parallel Efficiency (%)	Statistics on all Joins & Filters?	Joins or Filters on Derived Attributes?	CPU : I/O Ratio	Resource Usage Rating	Parallel Efficiency Rating	Overall Score
1	pr_Fpo_lmt_rpt	1,642	71.65	1,231,559	31.5	5,618,799,616	5.71	YES	NO	1.33	FAIL	FAIL	FAIL

Fig. 1 Resource Usage with an SQL that uses derived tables.

Figure 2 shows that each SQL passed in terms of performance evaluation criterion. Computing resources consumption such CPU, IO and spool usage is much lower compared to the resources used shown in Figure 3. Each SQL also shows that they higher parallel efficiency.


SQL Scorecard Results

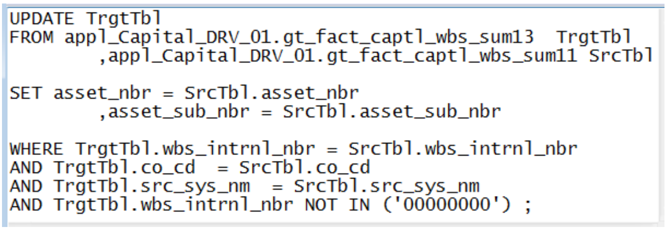
Project Name: Procurment Analysis MSTR
Developer Name: Wally Heaven
System used for Testing: TDCNS1

ETL		CPU Evaluation		I/O Evaluation		Spool Evaluation		EXPLAIN Evaluation		System Rating			
SQL-ID	Procedure Name (SQL)	Total CPU	CPU Parallel Efficiency (%)	Total I/O	I/O Parallel Efficiency (%)	Total Peak Spool	Spool Parallel Efficiency (%)	Statistics on all Joins & Filters?	Joins or Filters on Derived Attributes?	CPU : I/O Ratio	Resource Usage Rating	Parallel Efficiency Rating	Overall Score
1	gt_po_lmt_rpt - TEMP INSERT	7	69.79	53,001	96.18	40,266,240	93.86	YES	NO	0.14	PASS	PASS	PASS
2	gt_po_lmt_rpt1 - TEMP INSERT	14	57.74	59,305	84.05	57,574,240	96.65	YES	NO	0.23	PASS	PASS	PASS
3	gt_po_lmt_rpt2	164	87.53	183,323	85.59	1,634,344,448	60.94	YES	NO	0.89	PASS	PASS	PASS
4	gt_po_lmt_rpt2 - UPDATE	6	64.82	25,538	80.36	350,208	89.06	YES	NO	0.25	PASS	PASS	PASS
5	po_lmt_rpt - FINAL INSERT	9	62.37	12,428	68.5	71,610,368	72.77	YES	NO	0.75	PASS	PASS	PASS

Fig. 2 Resource Usage with SQL's that use GTT.

F. Avoid UPDATE Between Large Tables

Update is good when source table has fewer rows.


```
UPDATE TrgtTbl
FROM appl_Capital_DRV_01.gt_fact_captl_wbs_sum13  TrgtTbl
       ,appl_Capital_DRV_01.gt_fact_captl_wbs_sum11 SrcTbl

SET asset_nbr = SrcTbl.asset_nbr
       ,asset_sub_nbr = SrcTbl.asset_sub_nbr

WHERE TrgtTbl.wbs_intrnl_nbr = SrcTbl.wbs_intrnl_nbr
AND TrgtTbl.co_cd   = SrcTbl.co_cd
AND TrgtTbl.src_sys_nm   = SrcTbl.src_sys_nm
AND TrgtTbl.wbs_intrnl_nbr NOT IN ('00000000') ;
```

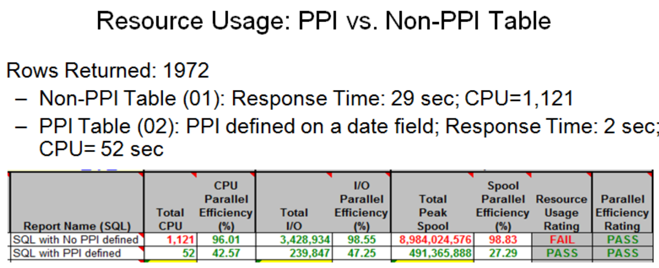
Fig. 3 Typical Update to narrow down the rows.

In one experiment we noticed that the UPDATE operation by joining large source table caused CPU consumption of 1,013 seconds. Using a subset of data from large tables with global temporary tables will help in computing resource saving. In a simulation we noticed that by using global temporary table for a sub-set of data in source table the UPDATE operation took only 600 CPU seconds. This indicates a 50% reduction in computing resources consumption.

G. Partitioned Primary Index (PPI)

It is a good idea to use Partitioned Primary Indexes (PPI) whenever possible. The score-card result shows better performance when rows are retrieved based on PPI defined on a date column (for instance). Queries which specify a restrictive condition on the partitioning column avoid full table scans as distinct PPI-based column values are stored in buckets. Larger tables are good candidates for partitioning. The greatest potential gain derived from partitioning a table is the ability to read a small subset of the table instead of the entire table.

In a PPI-defined table a query with filters on PPI column will directly pull data based on particular bucket(s) instead of scanning the whole table. In Figure 4 we show a comparison of performance metrics by performing SQL scorecard analysis of both PPI and non-PPI tables. The experiment results show that the SQL with PPI defined uses only 4.64% of the resources to pull rows from a PPI table compared to a non-PPI table. This significant amount of resource savings (95.36%) was possible due to ability to read a small subset of data using partitioning in the table instead of scanning the entire table.



Resource Usage: PPI vs. Non-PPI Table

Rows Returned: 1972

- Non-PPI Table (01): Response Time: 29 sec; CPU=1,121
- PPI Table (02): PPI defined on a date field; Response Time: 2 sec; CPU= 52 sec

Report Name (SQL)	Total CPU	CPU Parallel Efficiency (%)	Total I/O	I/O Parallel Efficiency (%)	Total Peak Spool	Spool Parallel Efficiency (%)	Resource Usage Rating	Parallel Efficiency Rating
SQL with No PPI defined	1,121	96.01	3,428,934	98.55	8,984,024,576	98.83	FAIL	PASS
SQL with PPI defined	52	42.57	239,847	47.25	491,365,888	27.29	PASS	PASS

Fig. 4 Resource Usage: PPI vs. No PPI tables.

Figure 4 shows a comparison of query response time and computational resource savings between PPI and No-PPI queries. The first query was run to pull 1972 rows, with no PPI defined. The response time was 29 seconds and CPU consumption was 1,121 seconds in row one. The first row shows that failed scorecard process. It was using too much resources and spool space for the query it pulled. The joins were exposed to too much data in source tables as opposed to rows being pulled. The same query was run against the same table but with PPI defined on a date field. For the second run the response time was two seconds and resource consumption was 52 seconds for row two. In this case, database optimizer pulled the rows based on PPI defined on the date field.

The database technology is available on the market for the last three decades. The commercial database companies have come up with several indexing techniques. The PPI technique is considered as one of the efficient indexing used in parallel processing database systems.

IV. Parallel Efficiency and DBMS Resource Usage

A parallel processing DBMS system plays two important roles towards maintaining an efficient data warehouse system. First, it helps running load, reporting, and ad-hoc queries faster than queries of traditional data warehouse system. Data warehouses hold mostly historical data of a business enterprise. Most of the tables in a data warehouse hold a large volume of data. Refreshing these large tables, running queries against large volume of data require significant processing capability of a data warehouse system. If a data warehouse system cannot load millions of rows within an SLA that increases data latency between operational database and the data warehouse [16]. This impacts tactical decision making capability of management. On the other hand, if a data warehouse cannot return query results within a reasonable time – (less than five minutes) then reporting capability of business tools is impacted. Slowness of a database system is not acceptable to the analytical community. A parallel efficient database system allows pulling rows much faster.

In a parallel processing database system SQL needs to be written in such a way that it takes advantage of parallel processing architecture of DBMS system. In a real world application the author of this paper observed that in many cases the application developers failed to realize they need to take into consideration the parallel processing architecture in writing SQL queries. This hurts the database system because in most cases the user query goes to a few machines as opposed to all machines available to process a query in parallel. When that happens system becomes skewed. SQL sent to those few machines struggle to handle a query while other available machines are sitting idle and end up using abnormally high amount of resources. We did some experiments between conventional queries and queries written in compliance with parallel processing database system. We have taken several measures such as use of indexes for data accessibility and data distribution to different machines while loading; use of global temporary tables to stage required with an amount of rows to make join operations efficient; synch up source and target table indices based on join columns and avoid expensive row redistribution on the fly while loading or retrieving rows; use of partitioned primary index allow queries to retrieve rows from the data warehouse directly from a specified location by avoiding all-rows scan; and use of global temporary tables (to enable collection of statistics) and avoid using derived tables (consisting of SQL on the fly) to prepare for join operations.

TABLE I: Computing Resource Consumption: ASIS vs. Optimized

ETL	CPU Evaluation		I/O Evaluation		Spool Evaluation			
Report Name (SQL)	Total CPU	CPU Parallel Efficiency (%)	Total I/O	I/O Parallel Efficiency (%)	Total Peak Spool	Spool Parallel Efficiency (%)	Resource Usage Rating	Parallel Efficiency Rating
pr_Ffact_captl_wbs_sum - ASIS	20	77.24	60,711	83.1	305,816,332,288	70.75	FAIL	FAIL
pr_Ffact_captl_wbs_sum - Optimized	13	72.66	44,647	81.88	79,175,168	90.9	PASS	PASS

Table-1 shows the variation of computing resource consumption by an as-is version vs. Optimized version. In as-is version the stored procedure SQL was using derived tables in SQL which caused missing statistics. As a result JOIN operations were suffering from performance issue. Collection of statistics is very important to improve parallel efficiency. It provides database parser engine to come up with a plan that is cost efficient for the requested SQL. Collects statistics provides information as to how many rows it is going to access during join operation. To overcome missing statistics issue in derived tables in SQL we removed the derived table from the SQL. We used global temporary tables instead which allowed us to collect statistics on join and filter columns. So, the as-is version of SQL that used derived tables in SQL caused spool space increase (virtual space) due to missing statistic in derived tables while performing join operations. In table 1, we can see the first row shows huge spool usage (in red) and a spool parallel efficiency of 70.75% only. This has caused overall SQL scorecard failure. On the other hand, the second row in the table shows a significant amount of spool space reduction. This has been achieved by removing derived table from the SQL and using temporary tables (with collect stats) instead. The second row in the table also shows significant improvement of spool parallel efficiency (90.9%).

TABLE III: Computing Resource Consumption: ASIS vs. Optimized

ETL	CPU Evaluation		I/O Evaluation		Spool Evaluation			
Report Name (SQL)	Total CPU	CPU Parallel Efficiency (%)	Total I/O	I/O Parallel Efficiency (%)	Total Peak Spool	Spool Parallel Efficiency (%)	Resource Usage Rating	Parallel Efficiency Rating
pr_Ffact_fin_gl_ap_exp_dtl - ASIS	1,285	32.35	9,218,673	60.77	73,774,083,497	54.1	FAIL	FAIL
pr_Ffact_fin_gl_ap_exp_dtl - Optimized	9	65.99	37,144	85.6	152,563,712	96.97	PASS	PASS
pr_Ffact_fin_gl_ap_exp_dtl1 - Optimized	5	63.38	15,737	90.07	16,808,960	93.95	PASS	PASS
pr_Ffact_fin_gl_ap_exp_dtl2 - Optimized	7	73.8	28,909	96.83	480,419,328	99.91	PASS	PASS
pr_Ffact_fin_gl_ap_exp_dtl3 - Optimized	8	67.71	31,672	91.14	26,244,608	86.96	PASS	PASS
pr_Ffact_fin_gl_ap_exp_dtl4 - Optimized	35	77.01	61,477	95.44	455,132,160	96.6	PASS	PASS
pr_Ffact_fin_gl_ap_exp_dtl5 - Optimized	18	20.9	91,946	32.71	28,469,760	1.67	PASS	PASS
pr_Ffact_fin_gl_ap_exp_dtl6 - Optimized	6	61.75	46,059	95.38	13,360,640	83.9	PASS	PASS
pr_Ffact_fin_gl_ap_exp_dtl7 - Optimized	23	84.32	102,193	98.84	16,215,040	95.35	PASS	PASS
pr_Ffact_fin_gl_ap_exp_dtl8 - Optimized	102	88.97	328,822	83.2	3,496,506,880	99.22	PASS	PASS

In Table-2, the first row shows the scorecard results of an as-is SQL version. It shows that the scorecard failed in terms of CPU consumption, IO generation and IO parallel efficiency; and spool usage and spool parallel efficiency. In this SQL the columns were not based on index columns, each source table had a large number of rows but not all data needed from user standpoint and also not all source columns needed for this from user standpoint. So, before making joins to eight large source tables we decided to use global temporary tables to stage required rows and columns from each of the source tables. In building the global temporary tables we made sure that join columns are reflected in the index to increase PE during table. Rows two to eight in Table II show a very small amount of computing resource consumption for staging data in each of the temporary tables. They also show the scorecard passed for each in terms of CPU, IO and spool space usage and their respective parallel efficiency. We also noticed that resource consumption has dropped and parallel efficiency has improved significantly. The last row in Table II shows the final SQL consisting joins to all temporary tables loaded under rows two to eight. As each temporary table holds a very small amount of rows and required fields only the join performance and overall load performance of SQL looks very impressive (the last 9 rows in Table II). The table shows a savings of 1072 (1285 - 213) CPU seconds. Thus, we can see that SQL written in compliance with parallel processing architecture of database system helps in improving data load and retrieval performance.

V. Conclusion

In this paper we assert that writing SQL for a parallel processing database system is not the same as it is for a traditional database system. The purpose of this paper is to point out the areas that an SQL developer needs to look at to comply with parallel processing architecture of underlying database system. This helps in achieving two important goals in a data warehousing system. One is to make sure minimal amount of computing resources is used. This is needed to keep data warehousing environment stable and healthy. The other goal is to allow the databases engine retrieve query results within a few seconds. Response time is the key to analytical community [23]. We have proposed a few techniques to improve parallel efficiency such as avoid using derived tables so statistics of join and filter column values do not get lost. We have also come up with techniques for writing SQL in small blocks with the help of global temporary tables. The global temporary tables are useful for run-time use and they do not need journaling and hence, helps in loading analytical table efficiently. In our experimental results we showed that writing SQL in compliance with parallel processing architecture helps in improving SQL performance and allows for saving CPU, IO and spool space usage.

VI. ACKNOWLEDGEMENTS

The author wishes to thank the anonymous referees for the time they spent on the detailed comments that were helpful in improving this paper. The author also thanks Joan Schnitzer for an excellent editing job.

References

[1] N. Rahman, J. Marz, and S. Akhter, “An ETL Metadata Model for Data Warehousing,” Journal of Computing and Information Technology (CIT). Volume 20, No 2, 2012, Pages, 95-111, 2012.

[2] R. Chirkova, Li, Li, and J. Li, “Answering Queries using Materialized Views with Minimum Size”. The VLDB Journal, 15(3), 191-210, 2006.

[3] D. Dey, Z. Zhang and P. De, P., “Optimal Synchronization Policies for Data Warehouse”, Information Journal on Computing, 18(2), 229-242, 2006/

[4] J. Evermann, “An Exploratory Study of Database Integration Processes”. IEEE Transactions On Knowledge and Data Engineering, 20(1), 2008

[5] J. García-García and C. Ordonez, “Extended aggregations for databases with referential integrity issues”. Data & Knowledge Engineering, 69, 73–95, 2010.

[6] T. Georgieva, “Discovering Branching and Fractional Dependencies in Databases”. Data & Knowledge Engineering, 66, 311-325, 2008.

[7] J.H. Hanson and M.J. Willshire, “Modeling a faster data warehouse”. International Database Engineering and Applications Symposium (IDEAS 1997).

[8] D.H. Hwang and H. Kang, “XML View Materialization with Deferred Incremental Refresh: the Case of a Restricted Class of Views”. Journal of Information Science and Engineering, 21, 1083-1119, 2005.

[9] A. Karakasidis, P. Vassiliadis and E. Pitoura, “ETL Queues for Active Data Warehousing”. In Proceedings of the 2nd International Workshop on Information Quality in Information Systems, IQIS 2005, Baltimore, MD, USA, 2005.

[10] T. Kim, J. Kim and H. Lcc, “A Metadata-Oriented Methodology for Building Data warehouse: A Medical Center Case”. Informs & Korms, Seoul, Korea, 2000.

[11] W. Labio, J. Yang, Y. Cui, H. Garcia-Molina and J. Widom, “Performance issues in Incremental Warehouse Maintenance”. In Proceedings of the VLDB, Cairo, Egypt, 2000.

[12] K.Y. Lee, J.H. Son and M.H. Kim, “Efficient Incremental View Maintenance in Data Warehouses”. In Proceedings of the Tenth International Conference on Information and Knowledge Management (CIKM'01), Atlanta, Georgia, USA, 2001.

[13] M. Mohania and Y. Kambayashi, “Making Aggregate Views Self-Maintainable”. Journal of Data and Knowledge Engineering, 32(1), 87 – 109, 2000.

[14] P. Ram and L. Do, “Extracting Delta for Incremental Data Warehouse Maintenance”. In Proceedings of the 16th International Conference on Data Engineering, San Diego, CA, 2000.

[15] H. Schwarz, R. Wagner and B. Mitschang, “Improving the Processing of Decision Support Queries: The Case for a DSS Optimizer”. In proceedings of the International Database Engineering & Applications Symposium. (IDEAS '01), 2001.

[16] M.A. Sharaf and P.K. Chrysanthis, “Optimizing I/O-Intensive Transactions in Highly Interactive Applications”. In Proceedings of the 35th SIGMOD international conference on Management of data, Providence, Rhode Island, USA, 785-798, 2009.

[17] A. Simitsis, P. Vassiliadis and T. Sellis, “Optimizing ETL Processes in Data Warehouses”. In Proceedings of the 21st International Conference on Data Engineering (ICDE'05).and Knowledge Management (CIKM'01), Atlanta, Georgia, USA., 2005.

[18] T. Koffing, "Teradata Architecture", Retrieved on 01/06/2012 from: http:// www.coffingdw.com/sql/tdsqlutp/teradata_architecture.htm

[19] W. Weill, M. Subramani and M. Broadbent, “Building IT Infrastructure for Strategic Agility”. MIT Sloan Management Review, 2002.

[20] J. Widom, “Research Problems in Data Warehousing”. In proceedings of the 4th Int'l Conference on Information and Knowledge Management (CIKM), 1995.

[21] Y. Zhuge, H. García-Molina, J. Hammer and J. Widom, “View Maintenance in a Warehousing Environment”. In Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD’95), San Jose, CA USA, 1995.

[22] Y. Zhuge, J.L. Wiener and H. Garcia-Molina, “Multiple View Consistency for Data Warehousing”. In Proceedings of the Thirteenth International Conference on Data Engineering, Birmingham U.K.and Knowledge Management (CIKM'01), Atlanta, Georgia, USA, 1997.

[23] N. Rahman, “Refreshing Data Warehouses with Near Real-Time Updates,” Journal of Computer Information Systems, 2007, Volume 47, Part 3, Pages 71-80, 2007.

[24] N. Rahman, “Saving DBMS Resources While Running Batch Cycles in Data Warehouses,” International Journal of Technology Diffusion (IJTD), 2010, Volume 1, Issue 2, Pages 42-55, 2010.