A BIT-SELECTOR TECHNIQUE FOR PERFORMANCE OPTIMIZATION OF DECISION-SUPPORT QUERIES

Ricardo Jorge Santos[1] and Jorge Bernardino[1,2]
[1] *CISUC – Centre of Informatics and Systems of the University of Coimbra, Coimbra, Portugal*
[2] *ISEC – IPC – Superior Institute of Engineering – Polytechnic Institute of Coimbra, Coimbra, Portugal*

Keywords: OLAP Query Performance Optimization, Bitmap and Bitwise Operations in OLAP.

Abstract: Performance optimization of decision support queries has always been a major issue in data warehousing. A large amount of wide-ranging techniques have been used in research to overcome this problem. Bit-based techniques such as bitmap indexes and bitmap join indexes have been used and are generally accepted as standard common practice for optimizing data warehouses. These techniques are very promising due to their relatively low overhead and fast bitwise operations. In this paper, we propose a new technique which performs optimized row selection for decision support queries, introducing a bit-based attribute into the fact table. This attribute's value for each row is set according to its relevance for processing each decision support query by using bitwise operations. Simply inserting a new column in the fact table's structure and using bitwise operations for performing row selection makes it a simple and practical technique, which is easy to implement in any Database Management System. The experimental results, using benchmark TPC-H, demonstrates that it is an efficient optimization method which significantly improves query performance.

1 INTRODUCTION

Over the last decades, data warehouses (DW) have become excellent decision-support means for almost every business area. Decision making information is mainly obtained through usage of tools performing On-Line Analytical Processing (OLAP) against DW databases. Since these databases usually store the whole business' history, they frequently have a huge number of rows, and grow to gigabytes or terabytes of storage size, making query performance one of the most important issues in data warehousing.

In the past, much research has been done proposing a wide range of techniques which can be used to achieve performance optimization of OLAP databases, such as, among others: *Partitioning* (Agrawal, 2004; Bellatreche, 2005; Bernardino, 2001), *Materialized Views and Aggregates* (Agrawal, 2000; Gupta, 1999), *Indexing* (Chaudhuri, 1997; Chee-Yong, 1999; Gupta, 1997), *Data Sampling* (Furtado, 2002), *Redefining database schemas* (Bizarro, 2002; Vassiliadis, 1999), and *Hardware optimization*, such as memory and CPU upgrading, distributing data through several physical drives, etc. to improve data distribution and/or access seeking efficient table balancing.

Sampling has an implicit statistical error margin and almost never supplies an exact answer to the queries according to the whole original data. Using materialized views is often considered as a good technique, but it has a big disadvantage. Since they consist on aggregating the data to a certain level, they have poor generic usage and each materialized view is usually built for speeding up a limited class of queries instead of the whole set of usual decision queries. Furthermore, they use a large amount of storage space and they also increase database maintenance efforts. Hardware improvements for optimization issues are not part of the scope of this paper. Although much work has been done with these techniques separately, few have focused on their combination, except for aggregation and indexing (Bellatreche, 2002, 2004; Santos, 2007).

The author in (Pedersen, 2004) refers that standard decision making OLAP queries which are executed periodically at regular intervals are, by far, the most usual form of obtaining decision making information. This means that this information is usually based on the same regular SQL instructions. This makes it relevant and important to optimize the performance of a set of predefined decision support queries, which would be executed repeatedly at any time, by a significant number of OLAP users.

Therefore, our goal is to optimize performance for a workload of given representative decision support queries, without defeating the readability and simplicity of its schema. The performance of *ad-hoc* queries is not treated. The presented technique aims to optimize the access to all fact table rows which are relevant for processing each decision support query, thus optimizing their execution time. As we shall demonstrate throughout the paper, this technique is very easy and simple to implement in any DBMS. Basically, it takes advantage of using an extra bit-based attribute which should be included in the fact table, for marking which rows are relevant for processing each decision support query.

The remainder of this paper is organized as follows. Section 2 presents related work on performance optimization research and specific bit-based methods. Section 3 explains our bit-selector technique and how to implement and use it. Section 4 presents an experimental evaluation using the TPC-H benchmark. Finally, some conclusions and future work are given in Section 5.

2 RELATED WORK

Data warehousing typically uses the relational data schema for modelling data in a warehouse. The data is modelled either using the star schema or the snowflake schema. In this context, OLAP queries require extensive join operations between fact and dimension tables (Kimball, 2002). Many techniques have been proposed to improve query performance, such as those which we have referred in the first section of this paper, among others.

Work in (Bellatreche, 2005) proposes a genetic algorithm for schema fragmentation, focused on how to fragment fact tables based on dimension table's partitioning schemas. Fragmenting the DW as a way of speeding up multi-way joins and reducing query execution cost is a possible optimization method, as shown in that paper. In (Bellatreche, 2004; Santos, 2007) the authors obtain tuning parameters for better use of partitioning, join indexes and views to optimize the total cost in a systematic system usage form. The method in (Bizarro, 2002) shows how to tune database schemas towards performance-orientation, illustrating their proposal with the same benchmark used in this paper to demonstrate our technique. We shall compare our results with theirs in this paper's experimental evaluation.

As we mentioned before, optimization research based on bitmaps has been proposed and regularly used in practice almost since the beginning of data warehousing, mainly in indexing (Gupta, 1997; O'Neil, 1995; Wu, 1998). The authors in (Hu, 2003) present bitmap techniques for optimizing queries together with association rule algorithms. They show how to use a new type of bitmap join index to efficiently execute complex decision support queries with multiple outer join operations involved and push the outer join operations from the data flow level to the bitmap level, achieving significant performance gain. They also discuss a bitmap based association rule algorithm. In (Agrawal, 2004) the authors propose novel techniques for designing a scalable solution to a physical design issue such as incorporating adequately partitioning with database design. Both horizontal and vertical partitioning is considered. The technique uses bitmaps for referencing the relevant columns of a given table for each query executed for a given workload. These bitmaps are then used to generate which column-groups of the table are interesting to consider for its horizontal and/or vertical partitioning.

Our technique essentially consists on adding a new integer attribute to be used as a bitmap for each row referring if it is relevant or not for each query. To know which rows are needed for processing each query we only need to test this new attribute's value – the *bit-selector* – recurring to a simple bitwise modulus operation (by comparing the remainder of an integer division, identifying the executing query). We aim for minimizing data access costs for processing the query workload, thus improving its performance by reducing execution time.

3 BIT-SELECTOR TECHNIQUE

Bitmap indexes are very common techniques used for upgrading performance, for they accelerate data searching and reduce data accesses. It is well known that the list of rows associated with a given index key value can be represented by a bitmap or bit vector. In this case, each row in a table is associated with a bit in a long string, an *N*-bit string if there are *N* rows in the table, with the bit set to 1 in the bitmap if the associated row is contained in the represented list; otherwise, the bit is set to 0. Our technique uses the same principle, but relating if the row is relevant for executing a given query.

3.1 Defining the Bit-Selector

Consider a table `T` with rows $\texttt{TR}_1$, $\texttt{TR}_2$, $\texttt{TR}_3$, and $\texttt{TR}_4$. Suppose a given workload with queries $\texttt{Q}_1$, $\texttt{Q}_2$ and $\texttt{Q}_3$. If all rows were necessary for processing query

Q_1, only the second and third rows were needed for query Q_2, and only the first three rows were necessary for processing query Q_3, we could represent this according to Table 1. For each row, we use 1 to define it as relevant for each query in column, and 0 if it is not.

Table 1: A bitmap example for Row-Query Bit-selecting.

	Q_3	Q_2	Q_1	Binary Value	Decimal Value
TR_1	1	0	1	101	5
TR_2	1	1	1	111	7
TR_3	1	1	1	111	7
TR_4	0	0	1	001	1

This way, the decimal value for each row may be obtained by transforming the binary value for the query workload. Observing Formula 1, we present the general conversion formula for obtaining the decimal value for bit-selection of each table row TR_i, given a workload of *N* queries { $Q_1, Q_2, …, Q_N$ }:

$$TR_i \text{ Bit-Selector Decimal Value} = QS_1 \times 2^0 + QS_2 \times 2^1 + … + QS_N \times 2^{(N-1)} \quad (1)$$

(Bit-Selector decimal value formula)

where QS_N represents the value 1 if row TR_i is relevant for Q_N, and 0 otherwise. This can be formulated simplified and generalized to Formula (2).

$$TR_i \text{ Bit Selector Decimal Value} = \Sigma\ (QS_J \times 2^{(J-1)}) \quad (2)$$

(Bit-Selector decimal value generic formula)

3.2 Using the Bit-Selector

Since the Bit-Selector is bit based, to know if a row TR_i is needed for processing a given query Q_N, we need to test if the Nth bit of its binary value is equal to 1. To do this, we only need to perform a modulus (MOD) operation (equal to the remainder of an integer division) on its bit-selector decimal value, using a power of base 2 and exponent equal to N. The generic formula for this is shown in Formula 3.

$$\text{Row } TR_i \text{ is interesting for } Q_N \text{ if (Bit-Selector Value MOD } 2^N) >= 2^{(N-1)} \quad (3)$$

(Rule for defining if a given row is relevant for a given query using the Bit-Selector technique)

Mainly, data access issues in data warehousing address fact tables, since they usually have a huge number of rows, when comparing to dimension tables. To use the bit-selector in the DW, we propose adding it as a column in its fact tables. This implies that query instructions executed against fact tables need to take this under consideration if they are to take advantage in using the bit-selector technique.

Using the decision support benchmark TPC-H (TPC-H) and DBMS Oracle 10g (Oracle), we shall now demonstrate examples on how to adapt queries for using our technique, for the whole set of 22 queries which belong to this benchmark.

To use our technique, note that the only modification in the schema is adding an integer column `L_BitSelector` in fact table `LineItem`. We shall now demonstrate how to update the *Bit-Selector* value for using our technique, and how to rewrite query instructions to take advantage of it. Since we cannot present an explanation for each of the queries due to space constraints, we shall use queries Q_1 and Q_{21} as examples. We also make considerations over each of the rewritten queries, comparing them to their respective original, in what concerns involved data operations and probable impact in query processing time.

Consider TPC-H query 1 (Q_1), which uses only the fact table `LineItem`, presented next:

```
SELECT
   L_ReturnFlag, L_LineStatus,
   SUM(L_Quantity) AS Sum_Qty,
   SUM(L_ExtendedPrice) AS Sum_Base_Price,
   SUM(L_ExtendedPrice*(1-L_Discount)) AS
      Sum_Disc_Price,
   SUM(L_ExtendedPrice*(1-L_Discount)*
      (1+L_Tax)) AS Sum_Charge,
   AVG(L_Quantity) AS Avg_Qty,
   AVG(L_ExtendedPrice) AS Avg_Price,
   AVG(L_Discount) AS Avg_Discount,
   COUNT(*) AS Count_Order
FROM
   LineItem
WHERE
   L_ShipDate<=TO_DATE('1998-12-01',
                       'YYYY-MM-DD')-90
GROUP BY
   L_ReturnFlag, L_LineStatus
ORDER BY
   L_ReturnFlag, L_LineStatus
```

To practice our technique, we account all fact table rows which are relevant for Q_1. This is done by using the fixed conditions in Q_1's WHERE clause, which defines the row filters. If this is the first time we are setting the `L_BitSelector` column for query Q_1, by applying Formula 2, the SQL statement for marking which rows of `LineItem` are relevant for processing this query is similar to:

```
UPDATE LineItem
   SET L_BitSelector = L_BitSelector + 1
WHERE
   L_ShipDate<=TO_DATE('1998-12-01',
                       'YYYY-MM-DD')-90
```

To rewrite query Q_1 to take advantage of the *Bit-Selector* attribute, the only modification in Q_1 would be in the WHERE clause, using Formula 3. The

WHERE clause of the rewritten query Q_1 would be:

```
WHERE MOD(L_BitSelector,2)>=1
```

This is a very slight modification to the original instruction, and should imply a small increase in its execution time, for instead of just executing a comparison of preset values (original Q_1 WHERE clause), in the modified instruction there is the need to execute a MOD operation for each row, and then compare values.

Consider TPC-H query 21 (Q_{21}), which performs a join with dimension table `Orders`. This table is only mentioned in the WHERE clause, in which it is used as a filter for selecting which rows in `LineItem` are needed in the query. Since our technique selects only the table's relevant rows, the join with table `Orders` becomes unnecessary, therefore discarding a heavy table join, leaving `Orders` out of the modified query. For the same reason, we can also exclude table `Nation` by selecting as relevant all `LineItem` rows (in conjunction with the selection criteria mentioned before due to the `Orders` row filtering in the WHERE clause) with `L_SuppKey = S_SuppKey` only for Saudi Arabia suppliers (`N_Name='SAUDI ARABIA'`). There are also conditional filters based on the fact table itself, with EXISTS and NOT EXISTS conditions, which should also be coped with to perform the selection of the relevant `LineItem` rows pretended for Q_{21}.

The original TPC-H query Q_{21} is similar to:

```
SELECT * FROM (
   SELECT
      S_Name, COUNT(*) AS NumWait
   FROM
      Supplier, LineItem L1, Orders, Nation
   WHERE
      S_SuppKey = L1.L_SuppKey AND
      O_OrderKey = L1.L_OrderKey AND
      O_OrderStatus = 'F' AND
      L1.L_ReceiptDate>L1.L_CommitDate AND
      EXISTS (
         SELECT *
         FROM LineItem L2
         WHERE L2.L_OrderKey=L1.L_OrderKey AND
            L2.L_SuppKey<>L1.L_SuppKey) AND
      NOT EXISTS (
         SELECT *
         FROM LineItem L3
         WHERE L3.L_OrderKey=L1.L_OrderKey AND
            L3.L_SuppKey<>L1.L_SuppKey AND
            L3.L_ReceiptDate>L3.L_CommitDate)
         AND
      S_NationKey = N_NationKey AND
      N_Name = 'SAUDI ARABIA'
   GROUP BY
      S_Name
   ORDER BY
      NumWait DESC, S_Name)
WHERE RowNum <= 100
```

After updating the value of `L_BitSelector` for the first time to optimize query Q_{21} according to our technique, the new instruction for Q_{21} would be:

```
SELECT * FROM (
   SELECT
      S_Name, COUNT(*) AS NumWait
   FROM
      Supplier, LineItem
   WHERE
      S_SuppKey = L_SuppKey AND
      MOD(L_Queries,2^21) >= 2^20
   GROUP BY
      S_Name
   ORDER BY
      NumWait DESC, S_Name)
WHERE RowNum <= 100
```

As it can be seen, the complexity of the original instruction Q_{21} has mostly decreased. Sub-querying and selection within the fact table itself has been discarded. The joins of `LineItem` with `Orders`, and `Supplier` with `Nation`, have been ruled out. Several condition testing such as value comparisons have also been discarded. The gain of processing time in this case should be very significant.

In conclusion, we may state that most decision support queries modified to comply with the proposed technique become simpler than the original instructions. They also significantly reduce the number of conditions to be tested and calculations to be performed on each row, reducing query processing costs. As seen in TPC-H query 21 (Q_{21}), the technique can also lead to avoid the need to execute heavy table joins involving fact tables.

3.3 Practical Update Procedures for the Bit-Selector

According to Formula 2, the generic instruction for updating the Bit-Selector column in any fact table for a given Query *N* would be similar to:

```
UPDATE FactTable
   SET
      L_BitSelector = L_BitSelector+2^(N-1)
   WHERE
     {List of Conditions in the Q_N WHERE clause}
```

For new incoming fact rows, the update may be performed both for new or previously considered queries. On the other hand, if a predefined query, which has already modified the *Bit-Selector* values, changes in a way that it needs to access a different set of rows than the ones that were marked as relevant, this change implies that the *Bit-Selector* must also be updated. To do this, we need to unmark the rows marked earlier as significant, and then mark again those which are now significant. Using Q_1 as an example, suppose we had already marked the significant rows, executing an instruction similar to:

Table 2: Time execution of the TPC-H query workload (Standard vs. Bit-Selector).

Database Size	Standard Exec. Time (seconds)	Bit-Sel. Exec. Time (sec)	Difference	% Exec. Time	Times Faster/Slower
1 Gbytes	675	418	-257	62%	1.61 times faster
2 Gbytes	1 831	882	-949	48%	2.08 times faster
4 Gbytes	4 266	1 634	-2 632	38%	2.61 times faster
8 Gbytes	10 332	3 384	-6 948	33%	3.05 times faster

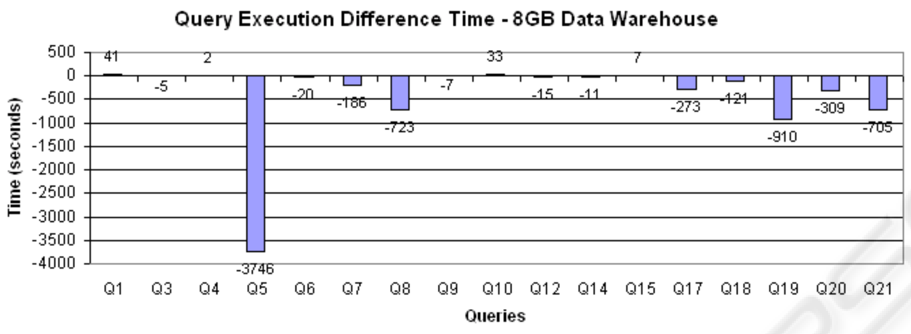

Figure 1: Query execution difference time – 8 Gbytes data warehouse.

```
UPDATE LineItem
   SET
      L_BitSelector = L_BitSelector + 1
   WHERE
      L_ShipDate<=TO_DATE('1998-12-01',
                          'YYYY-MM-DD')-90
```

As we discussed in the previous section, to determine which `LineItem` rows should be used for Q_1, we only need to test if `MOD(L_BitSelector,2)>=1` in its WHERE clause. Now assume that, instead of wanting the rows where `L_ShipDate<=TO_DATE('1998-12-01','YYYY-MM-DD')-90`, we want rows where `L_ShipDate<=TO_DATE('1998-12-01','YYYY-MM-DD')-180`. The algorithm for updating `L_BitSelector` to do this should be:

```
FOR EACH Row IN LineItem
   IF (MOD(L_BitSelector,2)>=1) AND
      (L_ShipDate>TO_DATE('1998-12-01',
                          'YYYY-MM-DD')-180)
      SET L_BitSelector = L_BitSelector - 1
   ELSE
      IF (MOD(L_BitSelector,2)=0) AND
         (L_ShipDate<=TO_DATE('1998-12-01',
                             'YYYY-MM-DD')-180)
         SET L_BitSelector = L_BitSelector + 1
   END IF
NEXT
```

The first half of this algorithm voids all rows previously defined as relevant for Q_1 and which are now to be discarded, by diminishing the decimal value responsible for its corresponding significant bit. The second half of the algorithm defines which fact table rows that were not and are now relevant for Q_1, in the same manner, by using the generic Formula 2. The rows which were already considered as relevant for the original Q_1 and remain relevant for the altered Q_1 do not need to be updated and are not, saving update time and resource consumption.

4 EVALUATION

To test our technique, we used TPC-H benchmark using DBMS Oracle 10g on a Pentium IV 2.8GHz machine, with 1 Gbyte SDRAM and 7200 rpm 160 Gbytes hard disk, with Windows XP Professional. We performed all experiments on 4 different scale sizes of the database: 1, 2, 4 and 8 Gbytes. Note that the sequence represents each next size as the double of the precedent. This will allow us to state conclusions regarding scalability of the results.

Table 2 presents the execution time for the set of TPC-H queries that need data from the fact table, for each database size. These are the queries to which our technique can be applied. For the fairness of experiments, all databases where index optimized the "standard" way, defining each table's primary key and building all relevant bitmap join indexes. Figure 1 shows the differences between standard and our technique's execution times, for each modified query, in the largest sized database.

It can be seen in Figure 1 that our technique brings advantages for most queries in the workload. As expected, queries Q_1 and Q_{15} present a small increase of execution time in all scenarios, for instead of just executing a comparison of fixed values (in the original Q_1 WHERE clause), modified instructions include executing a MOD operation for each row and then compare values. As also expected, queries Q_5, Q_8, Q_{19} and Q_{21} present the highest gains, because our technique discards the need for doing heavy join operations. Figure 2 shows overall workload execution time for each sized database.

Table 3: TPC-H original queries execution time with original fact table size vs. modified bit-selector fact table.

Database Size	Execution Time in the Original Schema	Execution Time in the Altered Schema	% Exec. Time Increase
1 Gbytes	675 seconds	706 seconds	4,6 %
2 Gbytes	1 831 seconds	1 921 seconds	4,9 %
4 Gbytes	4 266 seconds	4 484 seconds	5,1 %
8 GBytes	10 332 seconds	10 911 seconds	5,6 %

Authors in (Bizarro, 2002) use TPC-H as motivation for modifying the database schema in a performance-oriented manner. In their experimental evaluation, a workload of 10 TPC-H queries $\{Q_1, Q_3, Q_4, Q_5, Q_6, Q_7, Q_8, Q_9, Q_{10}, Q_{21}\}$ is executed against a 1 Gbyte database and execution time is analyzed. Their results show that the workload executes 2.19 times faster using the new schema, than with the original one. Consulting Table 3 in this paper, we can calculate that our Bit-Selector technique executes this same query workload 1.69 times faster. However, results presented in (Bizarro, 2002) are mainly due to one query only (Q_5). If Q_5 was excluded from the workload, their proposal would execute 1.84 times faster, while our proposal would execute 1.67 times faster. Furthermore, their experiments only consider 10 TPC-H queries, while we consider all of them. Therefore, we can state that our proposal seems better for optimizing a wide range of queries, compared with (Bizarro, 2002).

Analyzing Figure 2, we can state that the results indicate a very significant performance optimization, speeding up an increasing percentage of standard query execution time while the database size grows.

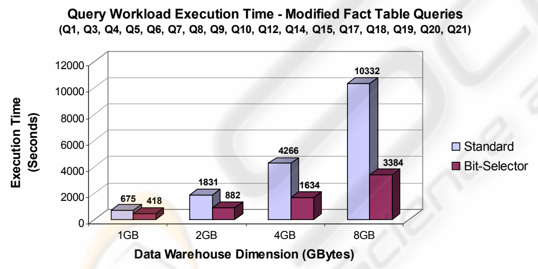

Figure 2: Query workload execution time for the modified fact table queries.

Figure 3 shows execution results for the TPC-H queries which were not modified because they do not access the fact table's data. As can be seen, these queries approximately maintained their execution times using the Bit-Selector technique. Since the schema modifications in our technique only changes the fact tables and queries which execute against it, other queries do not suffer any impact.

In what concerns database size, the modified schema in (Bizarro, 2002) increases 612 Mbytes (66%) of its original size. The size increase implied in our proposal (with the Bit-Selector as a 4 byte integer) is very low (3%), compared to the prior.

Finally, we address queries which access the fact table, but do not use the Bit-Selector column, i.e, the Bit-Selector column has not been updated for optimizing these queries. This can be measured by executing the exact original query instructions, using the fact table modified with the inclusion of the Bit-Selector column. Therefore, we executed workload $\{Q_1, Q_3, Q_4, Q_5, Q_6, Q_7, Q_8, Q_9, Q_{10}, Q_{12}, Q_{14}, Q_{15}, Q_{17}, Q_{18}, Q_{19}, Q_{20}, Q_{21}\}$ using original TPC-H query instructions against the new Bit-Selector fact table for each database. The results, presented in Table 3, show that execution time increased around 5%. This is the average increase for *ad-hoc* queries which access the fact table and are not to include in the set of queries used for the Bit-Selector, for the database used in our experiments. This was expected, because the altered fact table is bigger, due to the inclusion of the Bit-Selector, implying that the DBMS accesses a slightly bigger amount of data blocks.

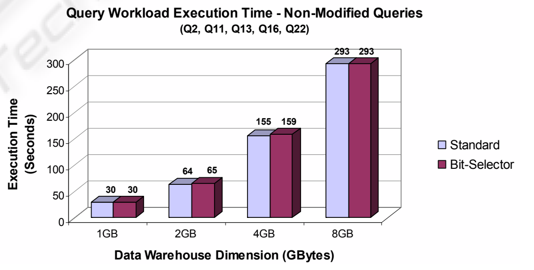

Figure 3: Query workload execution time for the modified fact table queries.

Many proposals in past research work in optimization imply data structure changes, increase of database size and query complexity, loss of schema legibility, among other negative aspects. As we stated earlier, the only modification to do within the database schema is the inclusion of a new integer type column (the Bit-Selector) in its fact tables, which will imply database size growth multiplying the Bit-Selector's size by the number of rows in the fact tables. Compared with most research work, our technique seems to be one of the best in what concerns database size overhead and schema modifications, while providing a very significant gain of query execution time.

5 CONCLUSIONS

This paper presents an efficient, simple and easy to implement alternative technique for optimizing OLAP query performance. It significantly reduces execution time of repeatable queries which need to access at least one fact table. Using our technique, the TPC-H workload executed 1.61, 2.08, 2.61 and 3.05 times faster than when "traditionally" index optimized, for the 1, 2, 4 and 8 GByte sized databases, respectively. Queries which do not access a fact table maintain their average response time.

We have also referred that *ad-hoc* query processing time increases because of the inclusion of an extra attribute in the fact table, which implies a size growth. However, both measured size and time increases are very small and should be considered as acceptable, when compared with the needed storage size in other optimization data structures such as partitions, pre-built aggregates and materialized views. We can state that the results show a very significant performance gain, speeding up an increasing percentage of standard query execution time while the database size grows.

Although query instructions need to be modified to take advantage of the proposed technique, the resulting rewritten instructions are often simpler than the original ones. The technique also makes it possible, for certain queries, to discard heavy time and resource consuming operations such as fact table joins. We also illustrated how to update the bit-selector attribute to optimize the performance for new queries or modify the row selecting of previously defined queries.

As future work, we intend to implement this method in real-world data warehouses and measure its impact on real world system's performance.

REFERENCES

S. Agrawal, S. Chaudhuri and V. R. Narasayya, *"Automated Selection of Materialized Views and Indexes in SQL Databases"*, 26th Int. Conference on Very Large Data Bases (VLDB), 2000.

S. Agrawal, V. Narasayya and B. Yang, *"Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design"*, ACM SIGMOD Conference, 2004.

L. Bellatreche and K. Boukhalfa, *"An Evolutionary Approach to Schema Partitioning Selection in a Data Warehouse Environment"*, Intern. Conf. on Data Warehousing and Knowledge Discovery (DAWAK), 2005.

L. Bellatreche, M. Schneider, H. Lorinquer and M. Mohania, *"Bringing Together Partitioning, Materialized Views and Indexes to Optimize Performance of Relational Data Warehouses"*, DAWAK, 2004.

L. Bellatreche, M. Schneider, M. Mohania and B. Bhargava, *"PartJoin: An Efficient Storage and Query Execution Design Strategy for Data Warehousing"*, DAWAK, 2002.

J. Bernardino, P. Furtado and H. Madeira, *"Approximate Query Answering Using Data Warehouse Stripping"*, DAWAK, 2001.

P. Bizarro and H. Madeira, *"Adding a Performance-Oriented Perspective to Data Warehouse Design"*, DAWAK, 2002.

S. Chaudhuri and V. Narasa11a, *"An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server"*, 23rd VLDB, 1997.

C. Chee-Yong, *"Indexing Techniques in Decision Support Systems"*, PhD Thesis, Univ. of Wisconsin, 1999.

P. Furtado and J. P. Costa, *"Time-Interval Sampling for Improved Estimations in Data Warehouses"*, DAWAK, 2002.

H. Gupta et al., *"Index Selection for OLAP"*, Int. Conference on Data Engineering (ICDE), 1997.

H. Gupta and I. S. Mumick, *"Selection of Views to Materialize under a Maintenance Cost Constraint"*, 8th Int. Conf. on Database Theory (ICDT), 1999.

X. Hu, T. Y. Lin and E. Louie, *"Bitmap Techniques for Optimizing Decision Support Queries and Association Rule Algorithms"*, Int. Database Eng. and Applications Symposium (IDEAS), 2003.

P. O'Neil and G. Graefe, *"Multi-Table Joins Through Bitmapped Join Indices"*, SIGMOD Record, Vol. 24, No. 3, September 1995.

T. B. Pedersen, *"How is BI Used in Industry?"*, DAWAK, 2004.

R. J. Santos and J. Bernardino, *"PIN: A Partitioning & Indexing Optimization Method for OLAP"*, Int. Conf. on Enterprise Information Systems (ICEIS), 2007.

TPC-H Decision Support Benchmark, Transaction Processing Council, www.tpc.org.

P. Vassiliadis and T. Sellis, *"A Survey of Logical Models for OLAP Databases"*, ACM SIGMOD Int. Conference on Management of Data (ICMD), 1999.

M. C. Wu and A. P. Buchmann, *"Encoded Bitmap Indexing for Data Warehouses"*, 14th ICDE, 1998.

R. Kimball and M. Ross, *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling, 2nd Edition*, Wiley & Sons, 2002.