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

Ricardo Jorge Santos[1], 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*
lionsoftware.ricardo@gmail.com, jorge@isec.pt

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 have become excellent decision-support resources for almost every business area. Decision making information is mainly obtained through usage of tools performing On-Line Analytical Processing (OLAP) against data warehouse databases. Because 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:

1) *Partitioning* (Agrawal, 2004; Bellatreche, 2000, 2005; Bernardino, 2001], which reduces the data which is necessary to scan for each OLAP query;
2) *Materialized Views and Aggregates* (Agrawal, 2000; Baralis, 1997; Gupta, 1999), which store summarized data and pre-calculated attributes, also aiming to reduce the data necessary to scan and reducing time consumption for calculating aggregate functions;
3) *Indexing* (Chaudhuri, 1997; Chee-Yong, 1999; Gupta, 1997), which speeds up accessing and filtering data;
4) *Data Sampling* (Furtado, 2002), giving approximate answers to queries based on representative samples of subsets of data instead of having to scan the entire data;
5) *Redefining database schemas* (Bizarro, 2002; Vassiliadis, 1999), to improve data distribution and/or access by seeking efficient table balancing;
6) *Hardware optimization*, such as memory and CPU upgrading, distributing data through several physical drives, etc.

Sampling is a technique which has an implicit statistical error margin attached to it 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 limited 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 may take up immense space within the data warehouse 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 implies that this kind of 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 the database 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 the queries' execution time. As we shall demonstrate throughout the paper, this technique is very easy and simple to implement in any Database Management System. 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 using bit-based methods in data warehouse optimization. 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 technology typically uses the relational data schema for modeling the data in a warehouse. The data can be modeled either using the star schema or the snowflake schema. In this context, OLAP queries require extensive join operations between fact tables and dimension tables (Kimball, 2002). Several optimization techniques have been proposed to improve query performance, such as materialized views (Agrawal, 2000; Baralis, 1997; Bellatreche, 2000; Gupta, 1999), advanced indexing techniques using bitmap indexes, join and projection indexes (Agrawal, 2000; Chaudhuri, 1997; Chee-Yong, 1999; Gupta; 1997; O'Neil, 1995), and data partitioning (Agrawal, 2004; Bellatreche, 2000, 2002, 2005), among others.

The authors in (Agrawal, 2000) present how to automatically choose a suitable set of materialized views and consequent indexes from the workload experienced by the system. This solution has been integrated within the Microsoft SQL Server 2000 tuning wizard. In (Gupta, 1999) a maintenance-cost based selection is presented for selecting which materialized views should be built. In (Chaudhuri, 1997; Chee-Yong, 1999; Gupta, 1997) authors illustrate features on which types of indexing should be performed based on system workload, attribute cardinality and other data characteristics. The work in (Bellatreche, 2005) presents a genetic algorithm for schema fragmentation selection, focused on how to fragment fact tables based on the dimension table's partitioning schemas. Fragmenting the data warehouse as a way of speeding up multi-way joins and reducing query execution cost is another possible optimization method, as shown in that paper. In (Bellatreche, 2004; Santos, 2007) the authors obtain tuning parameters for better use of data partitioning, join indexes and materialized views to optimize the total cost in a systematic system usage form.

The technique proposed in (Bizarro, 2002) presents how to tune database schemas towards performance-orientation, illustrating a demonstration of their proposal with the same benchmark used in this paper to demonstrate our optimization 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 decision

support queries together with association rule algorithms. They show how to use a new type of predefined 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 numeric integer attribute to be used as a bitmap for each row referring if it is relevant or not for each query. This way, to know which rows are necessary for processing each query we only need to test the value of this new attribute – the bit-selector – recurring to a simple bitwise modulus operation (by comparing the remainder of an integer division, identifying the query which is being executed). We aim for minimizing data access costs for processing the given query workload, thus improving its performance by reducing execution time.

3 THE BIT-SELECTOR TECHNIQUE

Bitmap indexes are one of the most common used techniques for upgrading performance, for they can 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 TR_1, TR_2, TR_3, TR_4, TR_5 and TR_6. Suppose a given workload with queries Q_1, Q_2 and 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
TR_5	0	0	1	001	1
TR_6	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 into its respective decimal value. 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 }:

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

Formula 1. Bit-Selector decimal value formula

Where QS_N represents the bit value 1 if row TR_i is relevant for Q_N, and 0 otherwise. This can be mathematically simplified and generalized to the final formula shown in Formula 2.

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

Formula 2. Bit-Selector decimal value generic formula (final).

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 N^{th} 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.

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

Formula 3. Rule for defining if a given row is relevant for a given query using the Bit-Selector technique

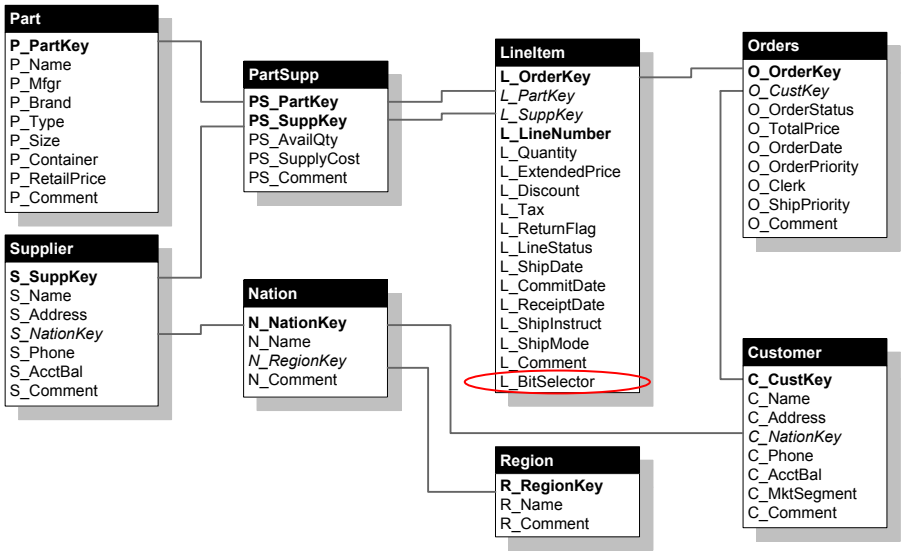

Figure 1. TPC-H benchmark database schema with the inclusion of the Bit-Selector attribute within the LineItem fact table

Mainly, data access problems 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 data warehouse, 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.

Recurring to the decision support benchmark TPC-H (TPC-H) and DBMS Oracle 10g (Oracle), we shall now demonstrate some examples on how to adapt decision support queries for using our technique, for the whole set of 22 queries which belong to this benchmark. Figure 1 shows the modified TPC-H database schema. To use our technique, note that the only modification necessary to perform in the schema is adding an integer column `L_BitSelector` in fact table `LineItem`.

We shall now demonstrate how to update the *Bit-Selector* attribute's value for using our technique, concerning the set of 22 TPC-H queries, and how to rewrite query instructions in order to take advantage of it. Since we cannot present an explanation for each of the queries due to space constraints in this paper, we shall use queries Q_1, Q_6 and Q_{21} as examples. We also make considerations over each of the rewritten queries, comparing them to their respective original, in what concerns then 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 put into practice our technique, we need to account all fact table rows which are relevant for Q_1. This can be done by using the fixed conditions existing 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 the generic formula presented in Formula 2, the SQL update statement for determining 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 the generic Formula 3 presented in the prior section. The WHERE clause of the rewritten query Q_1 would then become:

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

This is a very slight modification to the original instruction, and should imply a small increase its execution time, for instead of just executing a comparison of preset values (in the 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. On the other hand, consider TPC-H query 6 (Q_6):

```
SELECT
   SUM(L_ExtendedPrice*L_Discount) AS Revenue
FROM
   LineItem
WHERE
   L_ShipDate>=TO_DATE('1994-01-01',
                       'YYYY-MM-DD') AND
   L_ShipDate<TO_DATE('1995-01-01',
                      'YYYY-MM-DD'),12) AND
   L_Discount BETWEEN .06-0.01 AND
                      .06+0.01 AND
   L_Quantity < 24
```

Since all rows involved in the processing and returning results of Q_6 come only from the fact table `LineItem`, applying our method makes it possible to obtain the query's results just by verifying the value of `L_BitSelector`, dismissing all other comparisons which needed to be made in the original instruction. Therefore, to update the value of `L_BitSelector` for the first time, in order to optimize query Q_6, the instruction is similar to:

```
UPDATE LineItem
   SET L_BitSelector = L_BitSelector +2^5
WHERE
   L_ShipDate>=TO_DATE('1994-01-01',
                       'YYYY-MM-DD') AND
   L_ShipDate<TO_DATE('1995-01-01',
                      'YYYY-MM-DD'),12) AND
   L_Discount BETWEEN .06-0.01 AND
                      .06+0.01 AND
   L_Quantity < 24
```

The following new rewritten instruction for executing Q_6 is:

```
SELECT
   SUM(L_ExtendedPrice*L_Discount) AS Revenue
FROM
   LineItem
WHERE
   MOD(L_BitSelector,2^6) >= 2^5
```

Differing from Q_1, the modifications in Q_6 due to our technique should now save query processing time, for it reduces several fixed value comparisons.

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 the fact table `LineItem` are needed in the query's response. Since our technique selects the relevant rows in `LineItem`, the join with table `Orders` becomes unnecessary, therefore discarding the need for a heavy table join, leaving table `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 the suppliers from Saudi Arabia (`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 of Q_{21} has mostly decreased. Sub-

querying and selection within the fact table itself has been discarded. The joins of table `LineItem` with table `Orders`, and table `Supplier` with table `Nation`, have been ruled out. Several condition testing such as value comparisons have also been discarded. The gain of query 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

Since the TPC-H benchmark is composed with a set of 22 decision support queries, the maximum value for the bit-selector column is a 22 bit value in which all digits are 1 (1111111111111111111111), equal to the decimal value 4194303. If we were to add a new decision support query (Query 23) to the set of 22 queries already defined, the only thing needed to accommodate this for using the bit-selector technique is to update the bit-selector attribute, adding a 23 bit value ($2^{(23-1)}$) to the rows which would be significant for this query, according to the generic formula shown in Formula 2. This could be accomplished by the following generic instruction:

```
UPDATE LineItem
   SET
      L_BitSelector = L_BitSelector+2^(23-1)
   WHERE
     {List of Conditions in the Q23 WHERE clause}
```

Therefore, it is obvious to state that 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 QN WHERE clause}
```

If the update is to be made for new incoming fact rows in the data warehouse, this update may be performed both for new or previously considered queries.

On the other hand, if a previously defined query, which has already modified the *Bit-Selector* attribute 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 also needs to be updated. In order to do this, it is needed do unmark the rows which were marked earlier as significant, and then mark again those which are now significant. Using TPC-H query Q_1 as an example, suppose we had already updated `L_BitSelector` for this query, marking the rows which are significant. This was done by executing an instruction similar to:

```
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 rows in `LineItem` should be used for Q1, we only need to test if `MOD(L_BitSelector,2)>=1` in the WHERE clause of Q_1. Now assume that, instead of wanting the rows in which `L_ShipDate<=TO_DATE('1998-12-01','YYYY-MM-DD')-90`, we wanted the rows in which `L_ShipDate<=TO_DATE('1998-12-01','YYYY-MM-DD')-180`. The algorithm for updating `L_BitSelector` in order 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 the update algorithm would void 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 would define which fact table rows that were not and are now relevant for Q_1, in the same manner, by using the generic formula presented in 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.

3.4 Remarks and Considerations on the Bit-Selector Technique

The technique is simple and practical to implement. Most results should be promising due to the relatively low overhead and fast bitwise operations in the used bit-based techniques.

For *ad-hoc* decision queries which are to be executed only once, our technique should not be applied, because the resources and time needed to update the fact table would not result in a significant gain. If the query needs to access almost every row in the fact table, the technique is also not very efficient, for its nature is to simplify and optimize selecting and accessing only the relevant rows for processing the query. The more rows are needed for this, the fewer the gain given by the technique. Our technique is best for: queries which need the same set of fact table rows, repeatedly; and also if a small number of rows in the fact table (at most, 50% of the total number of rows in the table) is needed for query processing. Nevertheless, according to (Pedersen, 2004), these features represent a large class of decision queries to be executed in any business data warehouse.

On the other hand, research work on data warehouse optimization oftenly proposes methods and techniques that need to alter or reconstruct data structures, such as indexes and partitions, if not the database schema itself. This needs to be done with the database off-line from users, because the DBMS processes involved require exclusive access to those data structures. Such procedures imply a decrease in availability. With our technique, there is no need to set the data warehouse off-line, because to optimize each new query, we only need to execute an update for the bit-selector attribute. Therefore, it promotes continuous data warehouse usage, increasing its availability, in contrast with most other optimization methods and techniques with higher levels of complexity. It also presents a much lower overhead in data storage size when compared with techniques such as partitioning or creating materialized views. Another advantage of this modular approach is that it can be incorporated in every relational DBMS without any modification.

4 EXPERIMENTAL EVALUATION

To test the proposed technique, we implemented the TPC-H benchmark using DBMS Oracle 10g on Pentium IV 2.8 GHz machine, with 1 Gbyte of SDRAM and 7200 rpm 160 Gbytes hard disk with IDE U-DMA 133, with Windows XP Professional. We performed all experiments on four 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 3 presents the execution time for the set of queries in the TPC-H benchmark that need data from the fact table, for each predefined database size. These are the queries to which our bit-selector technique can be applied. For the fairness of the experiments, all databases where index optimized the "standard" way, defining each table's primary key and building all relevant bitmap join indexes. Figures 2, 3, 4 and 5 show the differences between standard and our technique's execution times, for each modified query, in each tested database.

Table 3. Time execution of the TPC-H query workload (Standard vs. Bit-Selector)

TPC-H Database Size	Standard Execution Time (seconds)	Bit-Selector Execution Time (seconds)	Execution Time Difference	% Execution 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

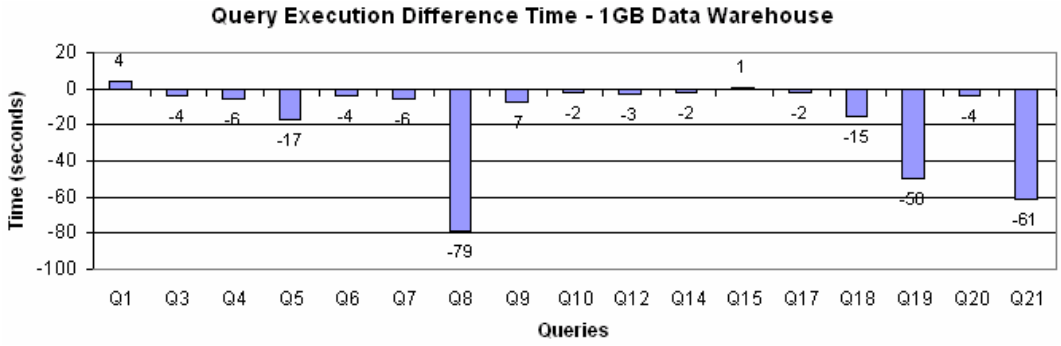

Figure 2. Query execution difference time – 1 Gbyte data warehouse

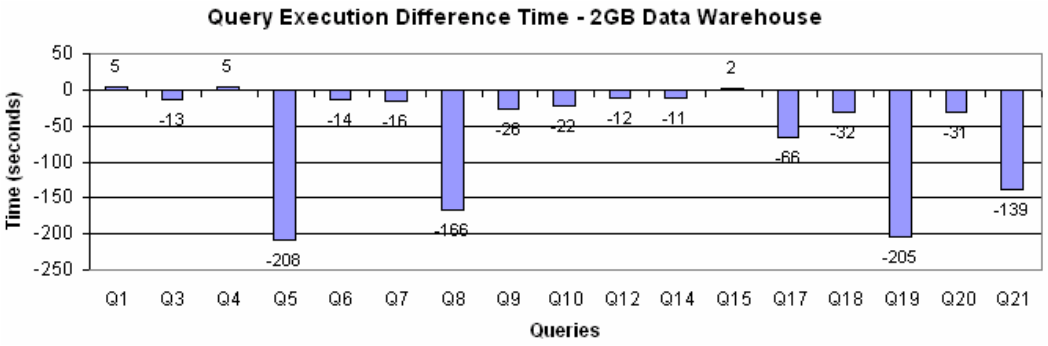

Figure 3. Query execution difference time – 2 Gbytes data warehouse

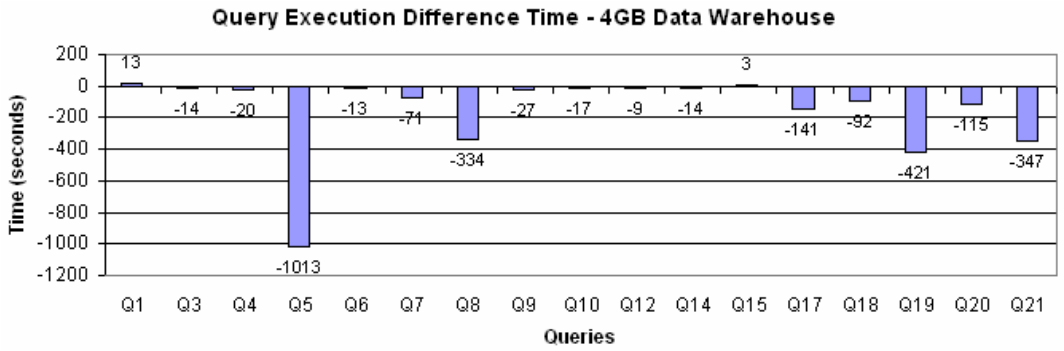

Figure 4. Query execution difference time – 4 Gbytes data warehouse

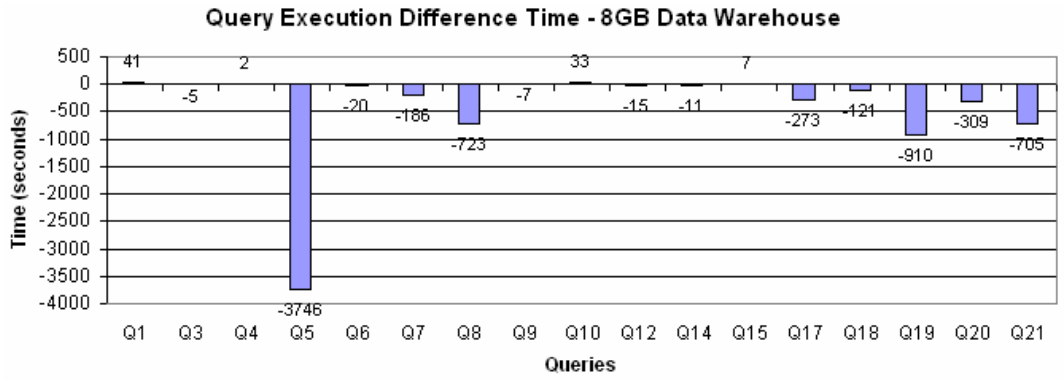

Figure 5. Query execution difference time – 8 Gbytes data warehouse

It can be seen in the individual query results that the proposed technique brings advantages for most queries in the workload, and the overall performance is efficiently improved. 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 (for the original Q_1 WHERE clause), the 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 with our technique the need for performing heavy join operations for these queries has been dismissed. Figure 6 shows the overall query workload execution time for each of the database sizes used in this evaluation.

As we mentioned earlier, authors in (Bizarro, 2002) propose to modify the database schema in a performance-oriented perspective. They use TPC-H benchmark database as an example and tune it looking for highest performance, taking under account the attributes which are most queried, calculations, frequently accessed dimensions, types of attributes, table joins, etc. 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 its execution time is analyzed. They state that for the fairness of experiments, all queries were fully optimized. Their results show that the workload executes 2.19 times faster using their new proposed 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 than without using our technique. 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, experiments in (Bizarro, 2002) only consider 10 TPC-H benchmark queries, while we consider all of them. Therefore, we can state that our proposal seems more continuous and consistent for optimizing a wide range of queries, when compared with the gains in (Bizarro, 2002).

Analyzing Figure 6, 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 7 shows the results for the execution of the TPC-H benchmark queries which were not modified because they do not access the fact table's data. As can be seen by observing this figure, the non-modified queries approximately maintained their execution times when using the Bit-Selector technique. Since the modifications of the schema for our technique only modifies the fact tables and queries which execute against it, other queries do not suffer any impact. Table 4 presents the impact in database size for the implementation of our Bit-Selector technique.

The modified database schema proposed in (Bizarro, 2002) presents an increase of 612 Mbytes (66%) of its original size. From this point of view, as seen in Table 4, the increase of size using our proposal (with the Bit-Selector column defined as a 4 byte integer) is very low (3%), when compared with the prior.

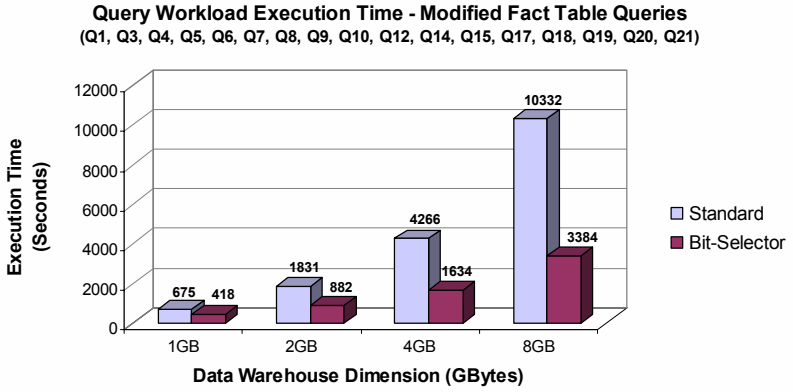

Figure 6. Query workload execution time for the modified fact table queries

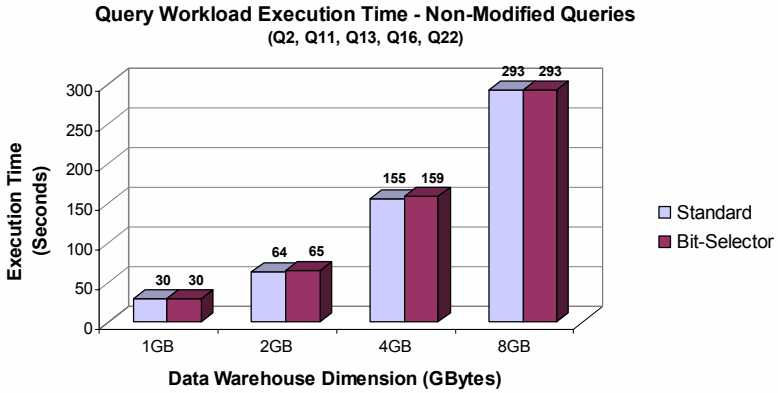

Figure 7. Query workload execution time for the modified fact table queries

Table 4. TPC-H original fact table size vs. modified bit-selector fact table size

Database Size	LineItem Original Size	Number of Rows in LineItem	LineItem Size with Bit-Selector	% Size Increase
1 Gbytes	801 Mbytes	6 001 215	825 Mbytes	3 %
2 Gbytes	1 602 Mbytes	11 997 996	1 650 Mbytes	3 %
4 Gbytes	3 204 Mbytes	23 996 604	3 300 Mbytes	3 %
8 GBytes	6 408 Mbytes	47 989 007	6 600 Mbytes	3 %

Table 5. TPC-H original queries execution time with original fact table size vs. modified bit-selector fact table

Database Size	Workload Exec. Time in the Original Schema	Workload Exec. Time in the Altered Schema	% Execution 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 %

Finally, we address the implications of our technique regarding decision support queries which access the fact table's data, but do not take advantage of 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 which need fact table data, using the fact table already modified with the inclusion of the Bit-Selector column. Therefore, we performed the execution of the query 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 the original TPC-H benchmark query instructions against the new fact table with Bit-Selector for each database. The results can be seen in Table 5.

It can be seen that query workload execution time increased around 5%. This is the average increase in execution time for *ad-hoc* decision support queries which access the fact table and are not to be included in the set of queries used for the Bit-Selector, for the database used in our experiments. This was somewhat expected, because the altered fact table is bigger, due to the inclusion of the Bit-Selector attribute, which means that the DBMS needs to access a slightly bigger amount of data blocks in order to access the same amount of factual data as in the original schema.

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

5 CONCLUSIONS AND FUTURE WORK

This paper presents an efficient, simple and easy to implement alternative technique for optimizing the performance of data warehouse OLAP queries, which significantly reduces the execution time of repeatable queries which need to access at least one fact table. Using our technique, the TPC-H query 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 size and time increases measured are almost insignificant and should be considered as acceptable, when compared with the storage size needed for other kinds of optimization data structures such as partitions, pre-built aggregates and materialized views. We can state that the results indicate a very significant performance optimization of the query workload, 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, without having to perform traditional off-line data warehouse update reoptimization procedures. This brings advantages due to enabling continuous usage fashion.

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 International 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.

E. Baralis, S. Paraboschi and E. Teniente, *"Materialized View Selection in a Multidimensional Database"*, 23rd Int. Conf. Very Large Data Bases (VLDB), 1997.

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, K. Karlapalem, M. Schneider and M. Mohania, *"What Can Partitioning Do for your Data Warehouses and Data Marts"*, Int. Database Engineering and Applications Symposium (IDEAS), 2000.

L. Bellatreche, M. Schneider, H. Lorinquer and M. Mohania, *"Bringing Together Partitioning, Materialized Views and Indexes to Optimize Performance of Relational Data Warehouses"*, Int. Conference on Data W. and Knowledge Discovery (DAWAK), 2004.

L. Bellatreche, M. Schneider, M. Mohania and B. Bhargava, *"PartJoin: An Efficient Storage and Query Execution Design Strategy for Data Warehousing"*, Int. Conference on Data Warehousing and Knowledge Discovery (DAWAK), 2002.

J. Bernardino, P. Furtado and H. Madeira, *"Approximate Query Answering Using Data Warehouse Stripping"*, Int. Conference on Data Warehousing and Knowledge Discovery (DAWAK), 2001.

P. Bizarro and H. Madeira, *"Adding a Performance-Oriented Perspective to Data Warehouse Design"*, Int. Conference on Data Warehousing and Knowledge Discovery (DAWAK), 2002.

S. Chaudhuri and V. Narasa11a, *"An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server"*, 23rd International Conference on Very Large Data Bases (VLDB), 1997.

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

P. Furtado and J. P. Costa, *"Time-Interval Sampling for Improved Estimations in Data Warehouses"*, Int. Conference on Data Warehousing and Knowledge Discovery (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. Conference 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 Engineering 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.

Oracle 10g DBMS, Oracle Corporation, 2005.

T. B. Pedersen, *"How is BI Used in Industry?"*, Int. Conference on Data Warehousing and Knowledge Discovery (DAWAK), 2004.

R. J. Santos and J. Bernardino, *"PIN: A Partitioning & Indexing Optimization Method for OLAP"*, Int. Conference 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 Int. Conference on Data Engineering (ICDE), 1998.

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