Queries on Dates: Fast yet not Blind

Jaroslaw Szlichta[1], Parke Godfrey[1], Jarek Gryz[1], Wenbin Ma[2], Przemyslaw Pawluk[1], Calisto Zuzarte[2]

[1] York University, Computer Science & Engineering, Toronto, Canada

{jszlicht, godfrey, jarek, pawluk}@cse.yorku.ca

[2] IBM Laboratory, Toronto Canada

{wenbinm, calisto}@ca.ibm.com

ABSTRACT

Data warehouses are repositories of electronically stored data which are designed to support reporting and analysis. The analysis of historical data often involves aggregation over time. Thus, time is critical in the design of a data warehouse. We describe novel techniques for storing date information and optimization of queries that reference the date dimension. We show how to embed intelligence into the date key and how to exploit monotonic dependencies. We present the value of these techniques for the improvement of performance when combined with partitioning and indexes. We evaluate these techniques on our prototype implemented in IBM® DB2® V9.7 over the current draft version of the TPC-DS benchmark.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems - *Query processing*

General Terms

Performance, Design, Experimentation

Keywords

Data Warehouse, Business Intelligence, Data Modeling, Time Dimension, Monotonic Dependencies.

1. INTRODUCTION

The time dimension is a significant aspect of data warehouses. Data warehouses are often designed to assist in analysis of business data over a historical period. Tables often include attributes which refer to time. This makes it possible for business analysts to detect the existence of temporal patterns [1]. For example, keeping the date of a sale enables analysis over different quarters or months of the year, and allows for the comparison of corresponding quarters or months over various years.

The amount of historical data grows quickly. For example, consider an organization in the telecommunications industry tracking phone calls over different cities and countries. Integration of data is a complex extract-load-transform (ETL) process that uses multiple sources. In order to make the storage of data efficient for analysis, the data is often aggregated in a warehouse. It is important to balance the degree of aggregation so it supports various types of drill down and roll up queries. OLAP [11] functions such as SUM, COUNT, MAX, MIN and AVG support the process of granulating data [2]. As a warehouse continually grows, it is important to design scalability from the very beginning so query performance is not sacrificed.

We introduce extensions to modeling of the time dimension in the data warehouse, and optimization of queries using these extensions. Section 2 describes our motivation and background. In Section 3, we show how the standard representation of date can be improved with embedding intelligence in the key, and we report the results of our experimental evaluation using monotonic dependencies. Concluding remarks and future work are described in Section 4.

Techniques described in this work have been implemented as a prototype in IBM® DB2® V9.7 [14] and tested on TPC-DS benchmark which models decision support systems [13].

2. BACKGROUND AND MOTIVATION

Time is often a central component of a data warehouse. Our observations with customer queries at IBM have shown that almost all queries involve time attributes. Therefore, being able to optimize queries with respect to data warehouse's time dimensions could offer large returns. We observe it to be quite common that the selectivity (filter factor) for the time predicates in queries to be the greatest filtering of all the predicates. This means the date dimension table is often joined first with the fact table in the query plan. We show that this *blind* join can be replaced with a pair of *fast* probes.

Large fact tables [10] are often partitioned to speed up evaluation and for easier roll-in and roll-out of data. The date surrogate key which is a date sequence number enables physical partitioning of the fact table on the date key (foreign key from the date dimension). Partitioning the fact table on the date key is well-kept throughout the changes as date is normally unchanged [3]. We have observed cases when the optimizer cannot exploit partition elimination in order to reduce I/O. The entire fact table may need to be scanned rather than just the relevant partitions for the query. We need to ensure that the optimizer can take advantage of the partitioning. We describe this problem and offer a solution in Section 3.

2.1 Multidimensional model

The common design of a data wareshouse is a multidimensional model based on a star or snowflake structure of fact and dimension tables [5]. A date dimension refers to a table that is granulated by day, whereas a time dimension represents the time of day. Such a dimension is practically universal as it appears in any data warehouse that is a historical repository of data [3]. It is often recommended to separate the time of the day from the date dimension in order to keep the date dimension small. As the time description is not usually required in a data warehouse, it is a good practice to keep the time attribute out of the date table in the fact table. Time of day might be represented as the number of milliseconds, seconds or minutes since midnight. For optimization reasons, the granularity of the date in data dimension table is sometimes even further aggregated to the level of week or month. It is also a common practice to keep the data from previous years aggregated with higher granularity.

The time dimension is traditionally used for tracking changes over measures. A model which allows a conceptual representation of time-varying levels, attributes and hierarchies is described in [4] and [7]. In that model, the time dimension can be used additionally to track changes in the other dimensions. In our work, we focus on the time dimension as the entry point to the fact table.

2.2 Description of the date

A measure is taken at the intersection of the fact table and dimensions such as date, item or location. All measurement rows in the fact table are at the same granularity [6]. Fact measures are usually numeric and additive. Dimension tables are a fundamental part of a data warehouse that contain the description of the data [9]. The dimension tables are necessary to understand the data in the fact tables. The fact tables can have a very large number of records, but are compact in terms of the number of columns. It is typical that the fact tables take about 90% of the entire space of the data wareshouse. Conversely, the dimension tables are shallow in their number of records, but have many columns for the descriptive attributes. Date dimension tables, apart from storing the date as a column, can also keep descriptions of the date which can be filtering fields and labels for business reporting. Columns in the date table can include, for example, day of week, the day number in a calendar month, the month number in year, month name, and fiscal periods. We can similarly include a holiday indicator, weekday indicator, the name of a holiday (Easter, Thanksgiving, Valentine's Day), the name of special events (Back to School, Super Bowl), and so forth.

2.3 Designing the date dimension

Designing date dimensions is a challenge. There are two main methods to represent the date in a data warehouse. The first approach is to keep the date dimension in the fact table, as shown in Fig. 1. If the date attribute is explicitly in a fact table, we can make a direct SQL query involving date that will not necessitate a join. Filtering based on date can avoid a join with a date dimension table (which could be quite expensive), so that the query is evaluated solely over the fact table.

The second technique is to create a separate date dimension table. There are good reasons why the second method is used more often. SQL date functions do not assist in filtering based on weekdays, weekends, holidays, major events and fiscal periods. Since there are no such built-in functions, it is better to store these as data in a dimensional table. Also, most database systems do not support index calculations using functions (e.g., MONTH, DAY) [3]. Lastly, the SQL data type DATE may be eight bytes, whereas INTEGER is four bytes. (Note that DB2 uses a compact four byte DATE: two for year, one for month and one for the day.) Therefore, using INTEGER as a surrogate key for date can save space: for every row, four bytes. If we have a fact table with a billion rows, every additional byte per row is another gigabyte of storage. And this storage is not necessarily inexpensive. To achieve high performance, many providers of data warehouse systems require expensive, proprietary hardware [12]. Because of the cost of the disk, which must be efficient and reliable, database schema designers must pay attention to saving space. More importantly, most queries scan rows from the fact table. By making the fact table more compact, this reduces the I/O expense of evaluating these queries. (Even though there is no space savings in DB2 as there might be in other database systems, designers often still use INTEGER surrogate keys to be consistent with other dimension surrogate keys.) A star schema with an additional dimension table with date is shown in Fig.2.

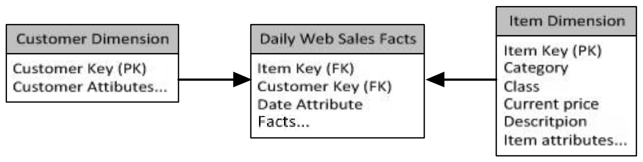

Figure 1. Date in fact table.

Designers sometimes replace a date dimension table by representing time via buckets in the fact table [3]. This solution is not commonly used though, because it is not flexible. With a predefined number of buckets which represent month one, month two, and so on, at some point the table has to be altered in order to add a bucket for a new month, or to shift all the buckets. This may not be the best choice as the month first on the list will be lost. A second disadvantage of this approach is that it is not possible to keep the description of date as specified in Section 2.2 so there is no way to get information regarding what the date refers to.

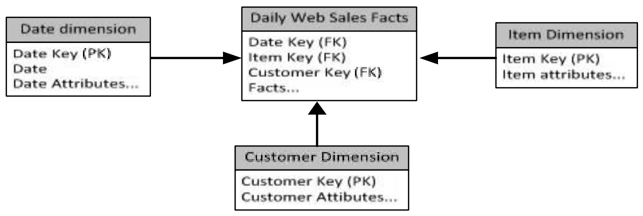

Figure 2. Date as additional table.

Surrogate keys are unique identifiers usually generated as sequential integers [8]. A join between the fact table and a dimension table is based on surrogate keys if we use surrogate keys for dimension tables. It is considered good practice to use surrogate keys instead of operational keys (derived from external names such as production codes) which may have built-in dependencies. Avoiding using an operational key as the primary key of the table is a good idea because our expectations might be invalidated over the time. In business organizations, operational codes such as product codes are reassigned after some period of time. Surrogate keys offer the data warehouse a mechanism to distinguish between two different instances of the same product with different operational codes. Developing the data wareshouse with operational keys might be faster, but implementing surrogate keys can bring benefits in the long run. The main advantage of surrogate keys is that they enable keeping track of changes independently from the key. Even if the operational key changes in the next ETL cycle, it can still stay in the data warehouse with the same surrogate key. Also, extracting data from multiple sources may be easier using surrogate keys as they allow integrating data from multiple source systems even if they do not have well-matched source keys. Using INTEGER (four bytes) for a surrogate key is adequate for a dimension table as it provides over two billion possible values (2^{32-1}).

3. QUERY REWRITES

As discussed above, there are often compelling reasons in a data warehouse to have an explicit dimension table for date, and, furthermore, one that uses surrogate keys. This is standard practice. There are disadvantages to this, however. A prevalent class of queries accesses a fact table, and has predicates on date that mention natural date values. Their evaluation will blindly join the fact table to the dimension table. This can be quite expensive. Often, a fact table will be partitioned over date. Data warehouse systems use partitioning to accommodate very large tables, making it easier to administer (back up data, re-organize data, roll-in new data, and roll-out old data) and to improve query performance. In this case, however, all partitions of the fact table will have to be scanned, because the natural date values in the query cannot be used to establish a range to scan just the relevant partitions, which are partitioned on the surrogate keys.

We seek to optimize such queries by avoiding the join to the date table, and selecting just the relevant partitions of the fact table. We introduce two query-rewrite techniques that achieve this.

In Section 3.1, we consider how to embed information of the natural date value into the date surrogate key. This preserves most of the benefits of using surrogate keys, and it lets us rewrite the query for optimization. There are certain disadvantages to using embedded surrogate keys, however, so this does not offer a general solution. In Section 3.2, we introduce a universal solution that rewrites the query within the optimizer considering monotonic dependency. The monotonic dependency is a dependency between two attributes in which one attribute is a monotonic function of the other. We note that in the implementation of almost all data warehouses that use surrogate keys for date, the order of surrogate keys is precisely the same as the order of the natural date values. That is to say, the surrogate keys and natural date values are monotone with respect to each other. We can use this monotonic dependency to rewrite queries.

- **Definition 1:** A real function f is *monotonic* on or over an interval I if it is either increasing on I($f(x_1) \leq f(x_2)$ whenever $x_1 < x_2$) or decreasing on I($f(x_1) \geq f(x_2)$ whenever $x_1 < x_2$).
- **Definition 2:** Also, function f is *strictly monotonic* if it is either strictly increasing or strictly decreasing.

In Section 3.3, we present an experimental evaluation of the monotonic-dependency rewrite over queries from the TPC-DS benchmark.

3.1 Embedding the intelligence

Our first consideration is to embed intelligence into the surrogate key. While this would be problematic for surrogate keys generally, this can work for time because the date dimension table is static. The representation of the date does not change over time, so keys might be assigned in a meaningful way. By embedding intelligence into the surrogate key, a query can directly filter over the fact table avoiding an expensive join (assuming all information for the query which is necessary is embedded in the key). A typical query with a join between a fact table and a dimension table taken from TPC-DS benchmark has a form like query Q_{12} shown below. The description of the schema of TPC-DS database can be found in [13].

A critical factor in making the technique efficient and useful is to find a good function which generates the key. A function which can be used is one converting the date into a number which consists of the year, month and day. The date 22th February 1999 would be converted into the key of the date dimension table: 1999022. This is shown in the query Q_{12}'. A more sophisticated function can be established for generating the key. Assume the surrogate key is a 4-byte integer. The date takes eight digits so there are still free digits. These can be used for indicators for weekday, holiday, current day, or other fields which appear often. We can also use a binary representation to make the key with embedded intelligence more compressed.

Q_{12}
```
select i_item_desc, i_category
  ,i_class , i_current_price
  ,sum(ws_ext_sales_price) as itemrevenue
  ,sum(ws_ext_sales_price)*100
     /sum(sum(ws_ext_sales_price)) over
     (partition by i_class) as revenueratio
from web_sales, item, date_dim
where ws_item_sk = i_item_sk
   and i_category in
       ('Sports', 'Books','Home')
   and ws_sold_date_sk = d_date_sk
   and d_date between
      cast('1999-02-22' as date)
      and (cast('1999-02-22' as date)
         + 30 days)
group by i_item_id, i_item_desc ,i_category
   ,i_class, i_current_price
order by i_category, i_class, i_item_id
   ,i_item_desc, revenueratio
fetch first 100 rows only
```

With embedding intelligence in the key, it is possible to keep historical information as well as some years into the future in the dimension table, because the function used is known in advance. One hundred fifty years of daily records is around 54,750 rows, which is a small table compared to fact tables which are counted in terabytes. An ETL process also takes benefit from filling the date dimension table in advance. Whenever the records are inserted to the fact table, a join between the fact table and the date dimension table is not needed to find the surrogate key.

Q_{12}''
```
select i_item_desc, i_category ...
from web_sales, item, date_dim
where ...and d_date_sk
      between 19990222 and 19990324
group by ...order by ...fetch...
```

3.2 Monotonic dependencies

In this section, our technique using monotonic dependencies is described. This technique was used in our prototype which has been implemented in DB2 V9.7. The following rewrites were considered as part of developing the prototype within the optimizer.

If in the table there is monotonic dependency between the surrogate key and another column, a small number of lookups can be used as part of the query. Two probes can be made into the dimension table in order to calculate the range of the surrogate keys to be used as a filter over the fact table. Query Q_{12}, with the observation of the monotonic dependency between the date surrogate key and the natural date values, can be rewritten to the form shown below (query Q_{12}'). The two probes are selected from date table: mindate and maxdate surrogate key. These two probes provide us with minimum and maximum surrogate values of the key which are used to set the filter in the WHERE clause. In our first attempt, we have rewritten the query to use subqueries for these probes in the WHERE clause. Based on our experiments and analyzing the access paths, we discovered that the optimizer does not treat it as a predicate on the same field, which meant it was not giving the optimal access plan. Our next try was to put the subqueries for the probes into the FROM clause. This is successful. The optimizer then gives the same access plans as it does for queries with constants in a BETWEEN range predicate.

Q_{12}'

```
select ...
from tpcds.web_sales, tpcds.item
   ,(select t1.mindate, t2.maxdate from
         (select min(d_date_sk) as mindate
          from tpcds.date_dim
          where d_date >= cast('1999-02-22'
             as date) ) as t1,
         (select max(d_date_sk) as maxdate
          from tpcds.date_dim
          where d_date <= (cast('1999-02-
          22' as date) +  30 days) ) as t2
    ) as dt
where ws_item_sk = i_item_sk and i_category
   in ('Sports', 'Books', 'Home')
   and  ws_sold_date_sk between
      dt.mindate AND dt.maxdate
group by ... order by ...fetch...
```

The following conditions are required in order to implement an efficient automatic rewrite:

1. **Relationship integrity predicate (condition for optimization).** The primary key from table A matches a surrogate foreign key from table B (B.fk = A.pk) and relationship between table A (tpcds.date_dim in Q_{12}) and B (tpcds.web_sales in Q_{12}) is one to many.
2. **Partitioning or index key (condition for optimization).** The foreign key B.fk is a range partitioning key, or there exists an index on it which can be used by the index manager for the subquery range predicate as a start and stop key.
3. **Local predicate (condition for optimization).** Table A has a simple local predicate in the form of 'A.col <relational operator> literal' where the relational operator is one of $\geq$, $\leq$, $>$, $<$, $=$ and A.col is not A.pk. In our implementation, we consider queries with a binary relationship predicate. This can be extended to queries with more complex expressions such as the IN operator. This kind of query also exists in the TPC-DS benchmark.
4. **Monotonic dependency (semantically required condition).** There exists a monotonic dependency declaration between the primary key of table A, column A.pk, and A.col in the local predicate in the query. At present in our prototype in DB2, we have a way to express this increasing or decreasing monotonic dependency internally. We are working to have a mechanism to declare such dependencies as formal constraints. Note that maintenance of such a dependency declaration is not expensive. When a new row is inserted in table A, only the rows with immediately preceding and succeeding values of A.pk by order need to be checked to ensure that the new A.col value maintains the monotone condition. Given there will be an index on the primary key of A, this check is inexpensive. The dependency between attributes d_date_sk and d_date is strict monotonic.
5. **No select on the A dimension table (condition for optimization).** Table A is not involved in the select. The fact that A has no column output from the current select is optional. If there is no column output from A, the join is eliminated. Otherwise, we can still do the rewrite and take advantage of partitioning.

In the proposed query rewrite, the function MIN and MAX is used because there are no guarantees that the fill rate in the date dimension table is 100%. There is currently no method to inform the query rewrite engine about this. With the possibility of passing this information, we would be able to simplify the query by removing MIN and MAX and replacing the relational operators by an offset equation so fewer IOs would be used for the probes.

Note that the rewritten query makes appropriate use of the partitioning. All the fact tables in the TPC-DS benchmark include the surrogate key from the date dimension, but they do not include the actual date. So the original query cannot take advantage of the partitioning with the surrogate key from the date dimension because the filter is based on the natural date value. The situation is different with the proposed query rewrite algorithm. The filter uses the surrogate key from the date dimension so it perfectly matches the partitioning key which is kept in the fact table. Partitioning for large fact tables is highly effective because it allows old data to be removed gracefully and new data to be loaded and indexed without disturbing the rest of the fact table. Also selecting the data based on partitioning key is more efficient. We have checked the access plan for our rewritten query, and the partition key based on the date surrogate key is used.

The second query from the TPC-DS benchmark we considered is Q_7 which joins the fact table with the date dimension and uses a description of the date as a filter. In the TPC-DS schema, there is an additional column which keeps the year (d_year). In the query Q_7, the filter is set to the year 2000. We chose this query because it selects over many more days (one year) whereas Q_{12} was only over 30 days. This query involves a select on a column in the date dimension table. We wanted to check how the proposed query rewrite behaves with a higher selection of data from the database.

The rewrite can be done with the same technique as for Q_{12}, but this time with an equality operator (filtering for year 2000). The rewrite cannot be used if instead of a predicate on d_date there is a predicate on d_month. Here the monotonic dependency is only local and there is no monotonic dependency between d_date_sk and d_month defined. On the other hand if there is predicate on both columns: d_year and d_month then the suggested strategy to optimize the query can still be used. In Q_7' if we would be able to pass information that the fill rate is 100%, and assuming there is check constraint d_year = YEAR (d_date), the rewritten query could have just used a filter on a d_date. At present, we have not addressed this. Therefore, we can use MIN and MAX in the same way we did for query Q_{12}.

Q_7

```
select i_item_id,
cast(avg(cast(ss_quantity as decfloat))
      as decimal(10,6)) agg1,
   cast( avg(ss_list_price) as
      decimal(10,6)) agg2,
   cast(avg(ss_coupon_amt) as
      decimal(10,6)) agg3,
   cast(avg(ss_sales_price) as
      decimal(10,6)) agg4
from store_sales, customer_demographics,
   date_dim, item, promotion
where ss_sold_date_sk = d_date_sk
   and ss_item_sk = i_item_sk
   and ss_cdemo_sk = cd_demo_sk
   and ss_promo_sk = p_promo_sk
   and cd_gender = 'M'
   and cd_marital_status = 'S'
   and cd_education_status = 'College'
   and (p_channel_email = 'N' or
   p_channel_event = 'N') and d_year = 2000
group by i_item_id order by i_item_id
fetch first 100 rows only
```

The rewritten query then with the filter on year is shown as below.

Q_7'

```
select ...
from store_sales, ..., promotion
   ,(select t1.mindate, t2.maxdate from
         (select min(d_date_sk) as mindate
            from date_dim
            where d_year = 2000 ) as t1,
         (select max(d_date_sk) as maxdate
          from date_dim
          where d_year = 2000 ) as t2) as dt
where ss_item_sk = i_item_sk
   and ss_cdemo_sk = cd_demo_sk
   and ss_promo_sk = p_promo_sk
   and cd_gender = 'M'
   and cd_marital_status = 'S'
   and cd_education_status = 'College'
   and (p_channel_email = 'N' or
      p_channel_event = 'N')
   and ss_sold_date_sk between
      dt.mindate and dt.maxdate
group by...order by...fetch...
```

3.3 EXPERIMENTAL EVALUATION

The query transformations described above have been implemented in DB2 V9.7 The experiments were performed on a performance testing machine with operating system SUSE Linux Enterprise Server SP1 with 4 processors (Intel(R) Xeon(R) CPU 5160 @ 3.00GHz), 22GB memory using TPC-DS benchmark with size 1GB. The performance test results are shown in bar chart Fig 3.

In our experiments, we measured the performance of the 13 queries from the benchmark that use dates in predicates and that match our rewrite rules. As expected a significant reduction of the execution time is achieved. The optimization is achieved primarily by avoiding the join between the fact table and the date table. Also, as we have mentioned, the cardinality reduction due to the selection on the date table is greater than due to the selections on other tables, so the first join is done between the fact table and the dimension table. Eliminating this first join from the access plan brings significant benefits. An index on the date foreign key in the fact table is enough for efficient evaluation. Note that more substantial performance improvements could be achieved if the date foreign key in the fact table is also a partitioning key.

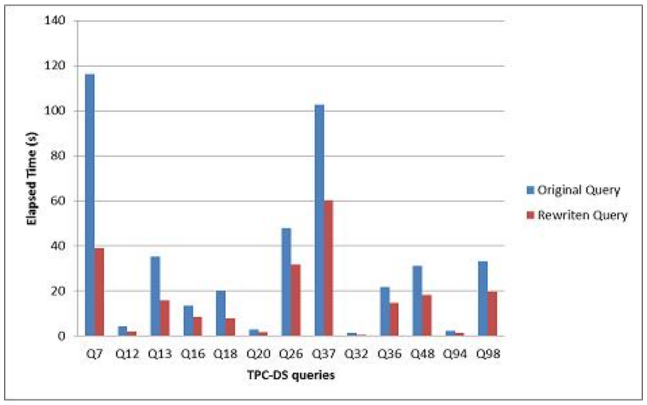

Figure 3. Visualized performance test results.

Fig. 3. shows the execution times for the 13 queries from the benchmark executed with and without our rewrite. The results demonstrate significant performance improvement, on average a reduction of 48% in elapsed time. For the queries discussed in Section 3.2, the reduction was from 116.34 seconds to 38.98 seconds (66%) for Q_7 and for the query with smaller filter, Q_{12}, from 4.26 seconds to 2.11 seconds (50%).

4. CONCLUSIONS AND FUTURE WORK

Queries that involve predicates over time and date are exceedingly common for most data warehouses. For many design reasons (Section 2.3), date is often represented explicitly as a dimension table of its own, with the primary key of the date table done as a surrogate key. While this design can have compelling advantages, the surrogate key can cause problems for queries.

A majority of queries are over a fact table. A query often uses natural date values in predicates. However, date in the fact table is recorded by surrogate key. This necessitates a potentially quite expensive join between the fact table and the date dimension table when the query is evaluated. There is an additional problem when a fact table has been partitioned by date, as it is common practice in data warehouse systems in order to accommodate very large tables (e.g. in distributed systems). Since the date range (surrogate values) over the fact table cannot be determined from the query (natural values) all the partitions of the fact table must be scanned.

We have sought to optimize such queries involving date in data warehouses by removing this join, and by choosing just the relevant partitions of the fact table when the table is distributed. We explored two solutions (Section 3). The first is to encode date information into the so-called surrogate key. In some situations, this is acceptable, and it preserves most of the advantages of using a surrogate key. Such an embedded surrogate key, however, makes it possible to execute many queries without a join to the date table. However, embedded surrogate keys are not always acceptable. Our second solution is more general. We introduce the notion of monotonic dependency, and show that surrogate data keys will be monotone with respect to the natural data values they represent in most any data warehouse design. By making this monotone dependency known to the query optimizer, queries with date predicates can often be rewritten to avoid a join with the date table, and to select just the relevant partitions of the fact table.

We built a prototype of this in IBM DB2 V9.7, (as a branch of the code base). We ran the benchmark experiments over TPC-DS to demonstrate the efficiency of our approach using the monotonic dependency. Of TPC-DS's 99 queries, 13 matched our rewrite. Every one of the 13 benefited, with an average performance gain of 48% (Section 3.3). The other queries in TPC-DS were not affected as they were not rewritten. (There is an additional optimization cost because of the additional rewrite rules, but it is marginal.)

There are some things that we plan do to extend this work:

- Many more than 13 of the 99 queries in TPC-DS involve date predicates. We know that we can extend our rewrite rules to cover many more of the cases seen in TPC-DS queries (for instance, to cover the case of sub-queries in an IN predicate). This is a matter of further implementation.
- We are working to improve the integration of our monotonic-dependency methods into the cost-based optimizer to achieve better cardinality estimation. This could potentially improve the performance gain we already observe for queries with date predicates a good deal more.
- In our prototype, the monotonic dependency is presented to the DB2 optimizer in an internal fashion. There is a way to declare such dependencies explicitly. We are working to state them as integrity constraints (devising a new class of constraints to support that cover monotonic dependencies).
- These techniques can be extended, we feel, to cover more dimensions in data warehouses. Monotonic dependencies exist elsewhere too, specifically with geo-spatial information. Thus, these techniques should extend well for other common data types.

We have demonstrated that dramatic gains in query performance can be had in data warehouse queries by recognizing monotonic dependencies over time data. Our techniques look promising to generalize many more types of queries that involve time and spatial predicates.

5. ACKNOWLEDGMENT

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

TPC-DS is the trademark of The Transaction Processing Performance Council.

6. REFERENCES

[1] Pedersen, T., Jensen, Ch., and Dyreson, C. 2001. A foundation for capturing and querying complex multidimensional data. Information Systems 26 (2001) 383-423.

[2] Riedewald, M., Agrawal, D., and Abbadi, A.E. 2002. Efficient Integration and Aggregation of Historical Information. Proceedings of the 2002 ACM SIGMOD international conference on Management of data.

[3] Kimball, R., Ross, M., 2002. The Data Wareshouse Toolkit Second Edition. The Complete Guide to Dimensional modeling. John Wiley & Sun.

[4] Malinowski, E., Zimányi, E. 2006. A conceptual Solution for Representing Time in Data Warehouse Dimensions. Conferences in Research and Practice in Information Technology Series, Vol. 166.

[5] Inmon, W. 2002. Building the Data Warehouse. John Wiley & Sons.

[6] Jarke, M., Lenzerini, M., Y., Vassiluiou & Vassiliadis, P. 2003. Fundamentals of Data Warehouse. Springer.

[7] Gregersen, H. & Jensen, C. 1998. Conceptual modeling of time-varying information. Technical report, Time Center, TR-35.

[8] Lightstone, S., Teorey, T., Nadeau, T. 2007. Physical database design. Morgan Kaufmann.

[9] Agrawal, R., Gupta A., Sarawagi, S. 1995. Modeling multidimensional databases. IBM Technical Report 1995. Appeared also in Proceedings of the 13th International Conference on Data Engineering 1997, pp 232-243.

[10] Gyssens, M., Lakshmanan, L.V.S. 1997. A foundation for multidimensional databases, Proceedings of the 23rd Conference on Very Large Databases, pp 106-115.

[11] Shoshani, A. 1997. OLAP and statistical databases: similarities and differences, Proceedings of the 16th ACM Symposium on Principles of Database Systems, pp 185-196.

[12] http://www.ibm.com/software/data/db2.

[13] http://www.tpc.org.

[14] http://publib.boulder.ibm.com/infocenter/db2luw/v9.