XPS a Database Server for Data Warehousing

Andreas Weininger
IBM Data Management Solutions
Andreas.Weininger@informix.com

ABSTRACT
A database server used for implementing a data warehouse must support other features than a database server used for OLTP. Therefore, in this paper we will look specifically at features necessary for efficiently processing queries on a database with a star schema model, a database schema which is used very often in data warehousing. We will especially analyze the features provided for this by the IBM Extended Parallel Server (XPS). There are special star join methods like the Push-Down Hash Semi Join, new access methods like Generalized Key (GK) indices, and specific index usages like multi-index scans for supporting the efficient processing of star schema queries.

1. INTRODUCTION
We will describe in this paper some of the data warehousing features of IBM Extended Parallel Server. The emphasis will be on features for the efficient processing of queries on a star schema. Therefore, we first introduce a star schema. After this we will look in detail into the individual features: First, we discuss a join index called Generalized Key index. Then we look at two specific index usage methods, Skip Scans and multi-index scans. Finally we will describe an efficient join method for star schemata, the Push-Down Hash Semi-Join.

2. STAR SCHEMA
A star schema is a very common database schema for modeling a data warehouse [2]. Much of the following discussion will assume a star schema. Therefore, we will define a schema in this section which will be used in the following discussions. A star schema contains a table f which is called fact table and n dimension tables $d_1, \ldots, d_n$. Figure 1 shows an example of a star schema.

The fact table is usually be far the largest tables. It contains columns $fk_1, \ldots, fk_n$ which are foreign keys referencing the corresponding dimension tables, and in addition the columns $c_1, \ldots, c_m$. The dimension tables d_i have a primary key

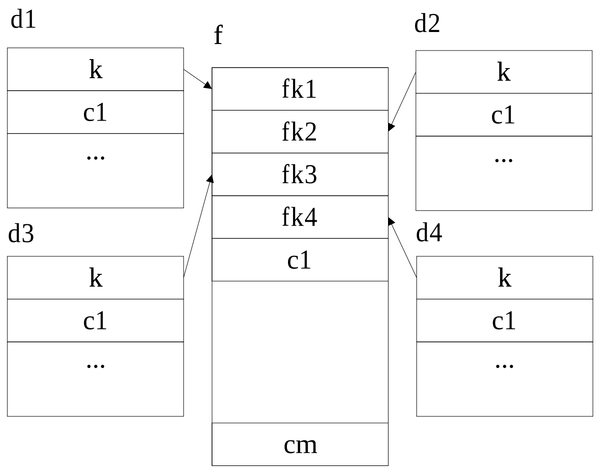

Figure 1: Example of a star schema

k and additional columns $c_1, \ldots, c_{m_i}$. For instance, a fact table may contain sales, and the dimension can be something like products, store, date/time, etc. Figure 2 shows a typical query for this star schema.

In general the restrictions on the dimension tables d_i can be any predicates on the columns $d_i.c_1, \ldots, d_i.c_{m_i}$. Typical are range expressions besides equality expressions. An overview of other techniques for processing star joins can be found in [3].

3. GENERALIZED KEY INDICES
The idea of Generalized Key (GK) indices is to build an index on the result of a query. One can think of a regular index i on the columns $c_1, \ldots, c_n$ of a table t as the result of the following query:

```
SELECT $c_1, \ldots, c_n$, rowid
FROM $t$
ORDER BY $c_1, \ldots, c_n$
```

This index could be created as a GK index in the following way:

```
SELECT $f.c_1, f.c_2$
FROM   $f, d_1, d_2, d_3, d_4$
WHERE  $f.fk_1 = d_1.k \quad \wedge$
       $f.fk_2 = d_2.k \quad \wedge$
       $f.fk_3 = d_3.k \quad \wedge$
       $f.fk_4 = d_4.k \quad \wedge$
       $d_1.c_1 = \gamma_1 \quad \wedge$
       $d_2.c_1 = \gamma_2 \quad \wedge$
       $d_3.c_1 = \gamma_3 \quad \wedge$
       $d_4.c_1 = \gamma_4$
```

Figure 2: Query Q_1: Example of a query in a star schema. $\gamma_1, \ldots, \gamma_4$ are constants.

```
CREATE GK INDEX $i$ ON $t$(
  SELECT $c_1, \ldots, c_n$
  FROM $t$)
```

There is no order by clause and no rowid specified in this index creation. The reason for this that this information is implied implicitly by the creation of the index. The values used for doing lookups in the index are coming from the projection list of the select statement.

Writing an index creation in this way wouldn't provide any advantage over regular indices, but GK indices allow not only this simple select statement but much more general ones. It is possible to specify several tables in the from clause, a where clause, and complex expressions in the projection list. It is not possible to specify a group by clause, because in this case the index would be no longer point to individual rows.

How does a GK index help in the processing of queries in a star schema? Let's look at the query from figure 2: This query is joining the four dimensions d_1, d_2, d_3, and d_4 with the fact table f. There is a restriction on column c_1 of each dimension table. The GK index in figure 3 could speed up the processing of a query of this kind, by doing the join between the fact table and the dimension tables during index creation and not during query execution.

```
CREATE GK INDEX $gki$ ON $f$ (
   SELECT $d_1.c_1, d_2.c_1, d_3.c_1, d_4.c_1$
   FROM   $f, d_1, d_2, d_3, d_4$
   WHERE  $f.fk_1 = d_1.k \quad \wedge$
          $f.fk_2 = d_2.k \quad \wedge$
          $f.fk_3 = d_3.k \quad \wedge$
          $f.fk_4 = d_4.k$)
```

Figure 3: GK index for the effient processing of query Q_1 in figure 2.

When this index is built the query in the index definition is executed and produces tuples in the following format: $(d_1.c_1, d_2.c_1, d_3.c_1, d_4.c_1$, rowid in $f)$. The rowids are for table f since this index is built on this table. Then these tuples are used for building the index like a regular multi-column b-tree index.

How is this index used for processing query Q_1? The values $\gamma_1, \ldots \gamma_4$ are used to do a look up in *gki* to get the rows in f which correspond to the where clause in query Q_1. Then the server fetches these rows and does a projection on columns $f.c_1, f.c_2$. Therefore, this query is executed by a simple index lookup although it is joining five tables.

If query Q_1 would do a count(*) in the projection list instead of getting $f.c_1, f.c_2$, then this query could be executed by a key-only index access. The GK index can even be used, if additional columns from the dimension tables are needed. In this case the GK index is used to reduce the size of the fact table in the join.

Like a regular concatenated index, a GK index can be applied if a prefix of the columns in the index is specified in the query. The decison whether to use a GK index is done by the optimizer in XPS based on the costs of the different query plans.

The GK index is not restricted to star schemata. The only condition which a join must fulfil to be usable in a GK index is the join must be an equi-join and it must transitively join *on key* to the table on which the index is built. An equi-join on two tables t_1 and t_2 with join condition $t_1.c_1 = t_2.c_2$ joins t_1 *on key* to t_2 if c_1 is the primery key of t_1 or if there is a unique index on $t_1.c_1$.

Therefore, a GK index can also be used for a snow flake schema like the one in figure 4. A snow flake schema is another schema which is popular in data warehousing.

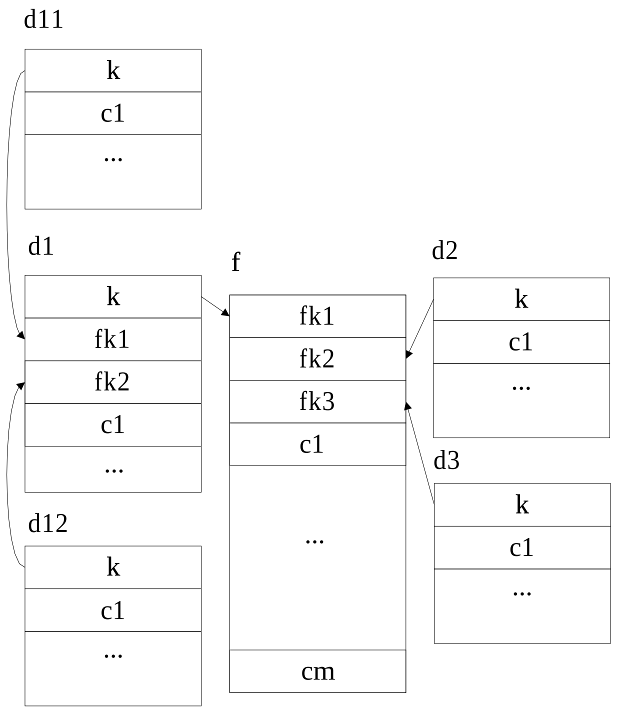

Figure 4: Example of a snow flake schema

When the GK index in figure 5 has been created, then the query Q_2 shown in figure 6 can be processed without accessing any table besides the fact table f. Note that the projection list of index gki_2 doesn't contain any references to columns from table d_1.

```
CREATE GK INDEX $gki_2$ ON $f$ (
    SELECT $d_{11}.c_1, d_{12}.c_1, d_2.c_1, d_3.c_1$
    FROM   $f, d_{11}, d_{12}, d_2, d_3$
    WHERE  $f.fk_1 = d_1.k \quad \wedge$
           $d_1.fk_1 = d_{11}.k \quad \wedge$
           $d_1.fk_2 = d_{12}.k \quad \wedge$
           $f.fk_2 = d_2.k \quad \wedge$
           $f.fk_3 = d_3.k)$
```

Figure 5: GK index for snow flake schema in figure 4

```
SELECT $f.c_1, f.c_2$
FROM   $f, d_1, d_2, d_3, d_{11}, d_{12}$
WHERE  $f.fk_1 = d_1.k \quad \wedge$
       $d_1.fk_1 = d_{11}.k \quad \wedge$
       $d_1.fk_2 = d_{12}.k \quad \wedge$
       $f.fk_2 = d_2.k \quad \wedge$
       $f.fk_3 = d_3.k \quad \wedge$
       $d_{11}.c_1 = \gamma_1 \quad \wedge$
       $d_{12}.c_1 = \gamma_2 \quad \wedge$
       $d_2.c_1 = \gamma_3 \quad \wedge$
       $d_3.c_1 = \gamma_4$
```

Figure 6: Query Q_2: Example of query for the snow flake schema in figure 4

GK indices can not only be used to avoid joins but also to build indices on expressions. This is called a virtual column index. Consider the following query:

```
SELECT ...
FROM $t$
WHERE $0 \leq c_1 + c_2 \wedge c_1 + c_2 \leq 10$
```

Normally it isn't possible to use an index on c_1 and/or c_2 for this query, but it is possible to build a GK index on the sum $c_1 + c_2$, which can be used to process the above where clause:

```
CREATE GK INDEX $gki_3$ ON $t$(
    SELECT $c_1 + c_2$ FROM $t$
)
```

Another interesting usage of GK indices are selective indices. In this way it is possible to build an index on only part of the data (usually the part which is used most often). This saves disk space and index build time. The optimizer is able to decide that such an index will be used, if the query is only accessing data in the range which is covered the index.

This is an example of a selective index:

```
CREATE GK INDEX $gki_3$ ON $t$(
    SELECT ...
    FROM $t$
    WHERE $t.date \geq 1995$
)
```

It is possible to have a GK index which is a join index, a virtual column index, and a selective index at the same time. The combination of a materialized view [1] and regular indices on the materialized view can have a effect similar to GK indices. The main advantage of GK indices over this combination of materialized view and index is that it avoids the space required for storing materialized view. For instance, storing a view corresponding to the GK index *gki* in figure 3 would require much more space than storing the fact table itself. Therefore, for example there is no reasonable way to do something like a selective index with the combination of materialized view and regular indices.

4. INDEX SCANS IMPROVEMENTS

In this section we will first describe a technique called Skip Scan which improves the performance of index scans in a data warehousing environment. After this we will describe a method for using several indices to process a single complicated where clause. The techniques presented in this section do not require that the data is stored in a specific schema like a star schema or a snow flake schema, but will work for any index scans on a table.

4.1 Skip Scan

In an OLTP system an index scan usually returns just a few rows or even a single row. In data warehousing there are often predicates which return a significant part of the data. A traditional index scan is not efficient for that since one or more random disk IOs have usually to be executed for each row fetched (depending on how much of the index tree is cached in the buffer pool).

A Skip Scan solves this problem since it first sorts the rowids which the index yields. Therefore, the rows of the table can be retrieved sequentially, only pages which contain rows have to be read (in comparison to a full table scan), and it is guaranteed that each page is read only once. Since reading rows sequentially is much faster than reading rows randomly, a Skip Scan provides much better performance for large result sets. The sorting of the rowids can be done efficiently by constructing a bitmap where bit i is set if and only if rowid i is in the result.

A disadvantage of a Skip Scan compared to a regular index scan is that the rows returned are no longer in index order. Therefore, the resulting rows must be sorted again if the index order is required. This is one reason, why it usually doesn't make sense to use Skip Scans in an OLTP environment.

4.2 Multi-Index Scan

Where clauses of data warehousing queries may be pretty complicated even if no joins are done. Look at the following query Q_3:

SELECT ...
FROM t
WHERE $(c_1 = 1 \land c_2 = \text{"abc"}) \lor \neg c_3 = 99$

If we assume that there are indices on all of the columns c_1, c_2, c_3, the optimzer would have to decide whether to do a full table scan or which one of the three indices to use. If the optimizer decides for doing an index scan it would usually pick the index with the highest selectivity. But the optimzer would have to decide for one single index to use.

A multi-index scan is a technique which is able to make use of all available indices. The idea is to get the set of rowids corresponding to each base expression (i.e. $c_1 = 1$, $c_2 = \text{"abc"}$, and $c_3 = 99$ in the example above) from the index. Then the set operation corresponding to the operation in the where clause is applied to these rowid sets. For instance a Boolean AND corresponds to intersecting the sets. An efficient representation for a set of rowids is a bitmap where bit i is set if and only if rowid i is in the set. Set operations like union and intersection can be implemented by bitwise Boolean operations. The final bitmap can be used for a Skip Scan as described in section 4.1.

The figure 7 shows how a multi-index scan can be done for the Q_3 shown above.

If there are just AND expressions in the where clause, a multi-column index could also be used for processing the query. But there are several advantages of a multi-index scan compared to a multi-column index:

- A multi-column index can only be used for AND expressions. Multi-index scans can be used for arbitrary complex expressions.
- A multi-column index can only be applied if no column in the prefix of the index is missing.
- If there are n columns in a table, n single column indices are sufficient to allow any possible multi-index scan. But $n!$ multi-column indices are necessary to cover just all possible AND expressions.

5. STAR JOIN

Normally it is not possible to handle all joins in a star schema with GK indices. The reason for this is that a large number of GK indices is necessary to handle all these possible joins. GK indices – like any multi-column indices – allow only one range condition, all the other conditions must be equality conditions.

Therefore, XPS supports a specific kind of star join, called *Push-Down Hash Semi-Join*. This join is executed as a hash join as the name indicates. Normally the best way to do hash joins in a star schema is with a right deep tree. Figure 8 shows a right deep query tree for query Q_1 in figure 2.

Let's assume that all the hash tables fit in memory. We also assume that each of dimension tables $d_1, \ldots, d_4$ has 10,000 rows and that the fact table f contains 100,000,000 rows. In addition let's assume that each of the conditions $d_i.c_1 = \gamma_i$

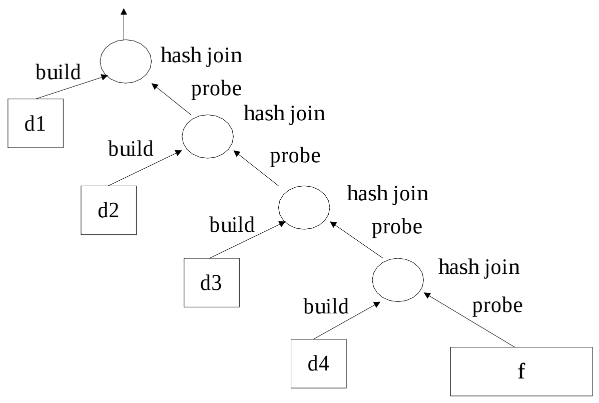

Figure 8: Right deep tree for query Q_1

selects 10% of the rows in the dimension table. Let's also assume that all the conditions are independent. This means that the first probing will be with 100,000,000, the second one with 10,000,000, the next one with 1,000,000 rows and the last one with 100,000 rows.

This is what the Push-Down Hash Semi-Join tries to improve: This join pushes the keys used for building the hash tables down to a multi-index scan on the fact table. This multi-index scan reduces the number rows used for probing against the first and all the following hash tables to 100,000 for the example above.

Figure 9 shows the Push-down hash semi-join plan for query Q_1.

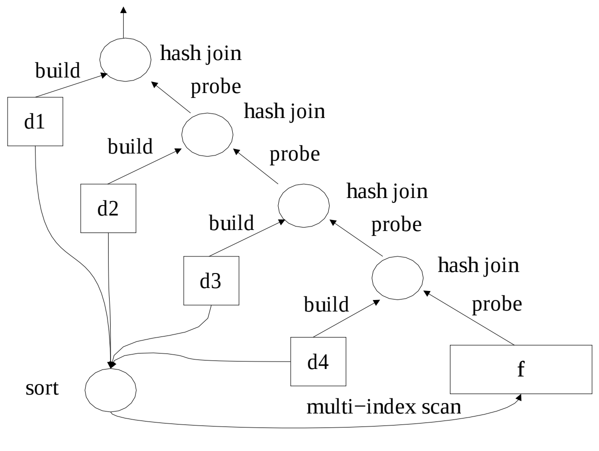

Figure 9: Push-down hash semi-join for query Q_1

Pushing down the keys makes only sense if the condition for a particular dimension table is sufficiently selective. It is also necessary for the Push-Down Hash Semi Join that the keys in the dimension tables are unique. For being able to do the multi-index scans on the fact table f, it is necessary to have the corresponding indices on this table.

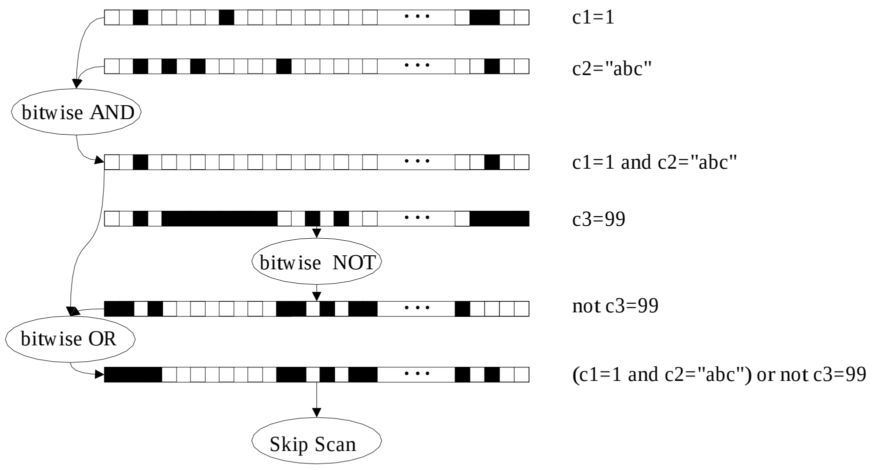

Figure 7: Example: Multi-index scan for query Q_3

In comparison to the GK index join, there is no requirement to have only one range condition on one dimension table. The restrictions on the dimension tables can have any format, as long as they are sufficiently selective.

6. CONCLUSIONS

In this paper we tried to show why it necessary to have specific processing methods to be able to execute queries in a data warehouse with a star schema efficiently. Each of the methods which were described in this paper (GK indices, Skip Scan, multi-index scan, Push-Down Hash Semi Join) are important in particular situations. Therefore, a combination of all these methods is implemented in XPS.

7. REFERENCES

[1] Ashish Gupta and Inderpal Singh Mumick, editors. *Materialized Views: Techniques, Implementations, and Applications.* The MIT Press, Cambridge, Massachusetts, 1999.

[2] Ralph Kimball. *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses.* John Wiley & Sons, February 1996.

[3] Patrick E. O'Neil and Goetz Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, 1995.