Optimizing Star Join Queries for Data Warehousing in Microsoft SQL Server

César A. Galindo-Legaria, Torsten Grabs, Sreenivas Gukal, Steve Herbert, Aleksandras Surna, Shirley Wang, Wei Yu, Peter Zabback, Shin Zhang

Microsoft Corp., One Microsoft Way, Redmond, WA 98052

{cesarg, torsteng, sgukal, stevhe, asurna, shwang, weiyu, pzabback, xinzh}@microsoft.com

Abstract— **As mainstream data warehouses are growing into the multi-terabyte range, adequate performance for decision support queries remains challenging for database query processors. Proper choice of query plan is essential in data warehouses where fact tables often store billions of rows. This paper discusses query optimization and execution strategies that Microsoft SQL Server employs for decision support queries in dimensionally modeled relational data warehouses. Our approach is based on pattern matching to detect typical star query patterns. When matching the pattern, the optimizer generates additional query plan alternatives specifically optimized for data warehouse performance. For high selectivity queries, the plans use nested loops joins and seeks. Medium selectivity queries in turn rely on right-deep hash joins with bitmap filters. Bitmap filters perform semi-join reductions to efficiently prune out non-qualifying rows early. Final plan choice is left for cost-based optimization which also compares the data warehouse specific plans against conventional query plans. We conducted an extensive experimental investigation using both synthetic workloads and several customer workloads. As our results show, the new plan shapes and execution strategies yield significant performance improvements across the targeted workloads as compared to earlier versions of Microsoft SQL Server.**

I. Introduction

Many companies today rely on data warehouses to support their decisions with business insight derived from operational data. As a consequence, both the number of data warehouses deployed across IT departments and the size of individual data warehouses have increased significantly. Main-stream data warehouses today hold several terabytes of data with table sizes passing the one billion row threshold.

So far, specialized data warehouse systems or proprietary implementations have oftentimes covered data warehouses of these dimensions. However, as large warehouses become more widely deployed, customers expect their database system vendors to cover requirements for large data warehousing with their products. The major challenges in that area have already been discussed in previous work on Online Analytical Processing (OLAP) in the database research community [13]. To name just a few, commercial database technology faces the following major requirements: (1) low query response times are essential even though queries need to process huge amounts of data, (2) small maintenance window for Extract, Transform, Load (ETL) are crucial to keep warehouse content up-to-date, and (3) solutions to store the vast amount of warehouse data in a cost-efficient way.

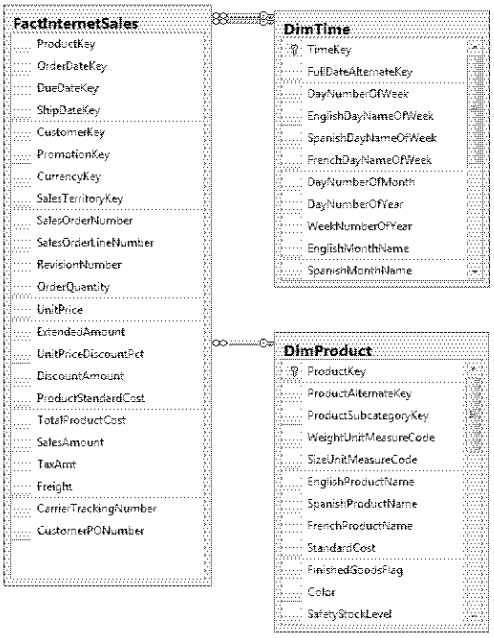


Fig. 1 Star schema example with a fact table and two dimension tables

In this paper, we focus on the first aspect mentioned above, namely how to achieve competitive query performance in the presence of multi-terabyte tables in a relational data warehouse. The paper discusses the approach chosen by Microsoft SQL Server which we have been extending over the last couple of releases of Microsoft SQL Server – with major additions now in the recent SQL Server 2008 CTP-3 release. Our approach is based on the observation that many data warehouses today are designed dimensionally, e.g., following R. Kimball's approach [6]. Dimensional modeling relies on the distinction of so-called dimension tables with fairly static data in contrast to so-called fact tables that store transactional data. Usually, fact tables are several orders of magnitude larger than dimension tables. Dimensional modeling leads to the well-known so-called star schema and snowflake schema design for data warehousing [6]. The AdventureWorksDW database that is part of the sample databases of Microsoft SQL Server is an example of a snowflake schema design. Figure 1 shows a part of the database that we will use as our running example throughout the paper.

Queries over such a schema follow a very typical pattern: they process rows from the fact table which are joined to

dimension table rows. Filter predicates on the dimension tables reduce the result set as needed to cover the business question. Most of these queries group by dimension columns and finally aggregate over columns of the fact table. We denote these queries as *star join queries*. The following query is an example of a star join query against the AdventureWorksDW schema introduced above:

```
select ProductAlternateKey, calendaryear,
       sum(salesamount)
from FactInternetSales fact
     join dimtime
       on fact.orderdatekey = timekey
     join dimproduct
       on dimproduct.productkey =
          fact.productkey
where calendaryear between 2003 and 2004
      and ProductAlternateKey like 'BK%'
group by ProductAlternateKey,calendaryear
```

A further important observation is that these queries typically fall into different classes regarding their selectivity against the main fact table accessed by the query. Selectivity against the fact table is defined by the fraction of rows of the fact table that are needed to process the overall query result. I.e., the fewer rows qualify for a specific query, the higher the selectivity of this query.

Our approach for Microsoft SQL Server relies on these observations regarding database schema, query shape and query selectivity in order to build pattern recognition mechanisms into the query processor. This is then used to augment the conventional query optimization process with functionality specifically targeted at data warehousing scenarios. In a nutshell, our approach comprises the following steps: (1) the query processor automatically identifies star and snowflake schemas and reliably detects fact tables and dimensions, (2) the query processor specifically matches star join queries, reasons about their selectivity classes and devises different query processing strategies for low, medium, and high selectivity queries, and (3) further performance improvements at runtime dynamically adjust to changing situations while processing the query.

In technical terms, the major contributions of this paper are (1) index intersection plans for high selectivity queries and (2) bitmap filter support for multiple dimensions, targeted at medium selectivity queries. Index intersection plans implement cross products over the dimension tables and then use the dimension keys to seek into the fact table. This plan shape is highly effective for very selective star join queries. But, it is usually not considered in regular query optimization since cross products lead to an explosion of the number of rows in intermediate query results. The bitmap filter technology that is scheduled to be released in SQL Server 2008 allows the query processor to perform multiple semi-join reductions simultaneously while processing the fact table. Our implementation of bitmap filters is highly optimized as it relies on the work already performed by processing the build side of hash joins prevalent in medium selectivity star join queries. A major advantage of bitmap filter query plans therefore is that they have little overhead as compared to other strategies for semi-join reduction. In addition, they do not require any specific physical design, i.e., they are immediately available without additional storage overhead for indexes or additional maintenance time during ETL.

An extensive experimental analysis has studied the performance characteristics of these new technologies that are available with pre-release SQL Server 2008 CTPs. Our study has covered both artificial workloads as well as real life workloads from Microsoft internal and external data warehousing customers. Our results show that the pattern recognition mechanisms reliably identify dimensionally modeled schemas and star join queries. For typical workloads, our experiments have shown that the new technology yields an average improvement of 15%-25% in expected query response times.

II. Data Warehousing Terminology

A. Logical Design

In contrast to transactional line-of-business applications where a normalized database schema is essential, logical database schema design for relational data warehouses puts less emphasis on normalization. Instead, many relational data warehouse designs today follow a so-called *dimensional modeling* approach that has been made popular by R. Kimball [6]. This leads to the well-known data warehousing schema types such as star and snowflake schemas. Following dimensional modeling for a relational data warehouse distinguishes so-called dimension tables from fact tables. *Dimension tables* hold master data representing entities such as products, customers, stores, or countries. The so-called *fact table* in turn stores transactional data such as sales or orders.

Dimension tables and fact tables are linked by primary key-foreign key relationships. A dimension table is small, i.e., it typically holds thousands up to several millions of rows. A fact table in turn is large. It usually holds hundreds of millions to billions of rows. Therefore, logical design needs to pay close attention to storage requirements of the fact table. Composite keys based on the business key of the dimension oftentimes cover several columns.

This is problematic for the fact table since the multi-column composite key would need to be repeated for each fact table row. Hence, a common practice in dimensional modeling is to use small so-called surrogate keys to implement the relationships between a fact table and its dimensions. A surrogate key is an integer-typed identity column that serves as an artificial primary key of the dimension table. The fact table refers to the surrogate key as opposed to the business key of the dimension. This significantly reduces storage requirements for large fact tables. Figure 1 illustrates a dimensionally modeled data warehouse schema using dimension and fact tables with surrogate keys.

The so-called *snowflake schema* design spreads one or several dimensions over multiple levels, e.g., customer, country, and region for a customer dimension. The levels are represented by separate tables which gives the schema the form of a snowflake. The *star schema* design in turn does not

spread any of its dimensions over several tables. A star schema has the form of a star where the dimension tables are grouped around the fact table in the center of the star.

B. Star Queries

With dimensionally modeled star schemas or snowflake schemas, decision support queries follow a typical pattern: the query selects several measures of interest from the fact table, joins the fact rows with one or several dimensions along the surrogate keys, places filter predicates on the business columns of the dimension tables, groups by one or several business columns, and finally aggregates the measures retrieved from the fact table. The SQL query in the introductory section is an example of a star join query.

C. Physical Design

Decision support queries usually vary a lot over time. Hence, workloads for data warehouses tend to have a high ratio of ad-hoc queries. This makes physical design for decision support queries and a dimensionally modeled data warehouse schema challenging. With Microsoft SQL Server, data warehouse customers oftentimes start with a "blue-print" physical design that they tune and refine over time as workloads evolve.

1) *Physical Design of the Fact Table:* The blue-print physical design aims to anticipate the typical star query shape and builds the following indexes over the fact table: The clustered index of the fact table uses several dimension surrogate foreign key columns as index keys. Usually, the most frequently used columns occur first in the list of index keys, and the most selective ones are the leading key columns. This aims to provide a good access path for most of the queries in the workload. In addition, a single-column non-clustered index is created for each dimension surrogate foreign key column in the fact table. This provides a highly efficient access path for queries that are very selective in one of their dimensions.

2) *Physical Design of Dimension Tables:* For dimension tables, the "blue-print" physical design suggests the following indexes: A non-clustered primary key constraint index is created on the surrogate key column of the dimension. The clustered index is built over the columns of the business key of the dimension entity. This facilitates well-performing lookup and update for dimensions at ETL time in the maintenance window of the data warehouse which is oftentimes a highly time-critical process. For large dimension tables, the blue-print design considers additional non-clustered indexes over columns that are frequently used in highly selective predicates.

The blue-print design outlined above serves as a good starting point for physical designs in relational data warehouses built with Microsoft SQL Server.

III. Overview: Selectivity Ranges

1) *Query selectivity classes:* The biggest cost in executing a decision support query in a dimensionally modeled relational data warehouse is processing of the fact table. This is because even for highly selective queries an order of magnitude more rows are being retrieved from the fact table than from any dimension. Therefore, proper choice of access path into the fact table is essential for good query performance.

We categorize star join queries into three different classes which helps us in identifying proper plan choices for those queries. The query classes are illustrated by Figure 2. As the figure shows, the first class covers highly selective queries. These queries process up to 10% of the rows in the fact table. We denote this class as *high selectivity queries*. The second class in turn is less selective than the first one. Queries in the second class process more than 10% and up to 75% of the fact table rows. Queries in this class are called *medium selectivity queries*.

Queries in the third class finally require processing of more than 75% of the rows stored in the fact table. We call these queries *low selectivity queries*.

2) *Plan choice based on selectivity.* Our classification of queries into different selectivity ranges against the fact table helps us identify alternatives for well-performing query plans. The following paragraphs discuss the query classes and their plan choices in more detail.

Since *high selectivity star queries* retrieve not more than 10% of the fact table rows, those queries can usually afford random access into the fact table. Hence, query plans for that class heavily rely on nested loop joins in combination with index seeks and bookmark lookups into the fact table. Section VI discusses specialized optimization strategies for highly selective star join queries in more detail.

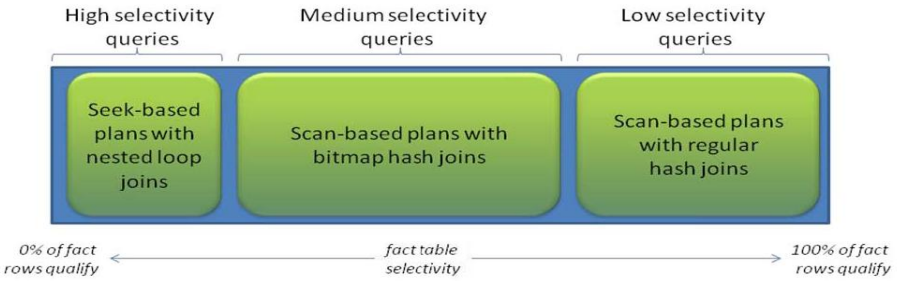


Fig. 2 Selectivity ranges for star queries

Since *medium selectivity star queries* process a significant fraction of the rows in the fact table, hash joins with fact table scans or fact table range scans are usually the preferred choice of access path into the fact table. Microsoft SQL Server further improves performance of these hash joins with bitmap filters. Section VI discusses bitmap filters and further optimization strategies for medium selectivity star join queries in more detail.

IV. Overview of the Optimization Process for Star Join Queries

Figure 3 gives an overview of the optimization process in Microsoft SQL Server. As the figure shows, standard heuristics for join query optimization generate an initial set of query plan alternatives. Special extensions are then invoked to generate additional plan alternatives. In the case of data warehousing, the extension detects star and snowflake schemas, detects star query patterns and performs an analysis of the selectivity against the fact table. If the schema and the query shape match the patterns, the extension adds further query plans to the plan space that is then covered by cost-based optimization to choose the plan for execution.

V. Detecting Star Join Queries

Many physical designs for data warehouses follow the star schema, but do not completely specify the relationships between the fact and dimension-tables. E.g., often the foreign key constraint of the dimension surrogate key in the fact table is not explicitly specified. This is done to avoid the overhead of constraint enforcement during updates.

Without the foreign key constraints explicitly specified, we have to depend on heuristics to detect star schema query patterns. One of the transformation rules in the Microsoft SQL Server optimizer combines multiple joins into a single *n*-ary join operator. The following heuristics are then applied to detect star join query patterns:

1) The largest of the tables participating in the *n*-ary join is considered the fact table. There are additional restrictions on the minimum size of the fact table. I.e., if even the largest table is not beyond a specific size, the *n*-ary join is not considered a star join.
2) All join conditions of the binary joins in a star join query have to be single column equality predicates. The joins have to be inner joins. While this might sound restrictive, it covers the vast majority of joins between the fact table and dimension tables on the surrogate key in typical star schemas. If a join has a more complex join condition which does not fit the pattern described above, it is excluded from the star join. E.g., a 5-way join can lead to a 3-way star join (with two additional joins later on), if two of the joins have more complex join predicates.

It is important to note that those are heuristics rules. There are rare circumstances conceivable which lead the heuristics to pick a dimension table as fact table. This influences the plan choice, but has no impact on the correctness of the selected plan.

The binary joins participating in a star join are then ordered by decreasing selectivity. Join selectivity in this context is defined as ratio of input cardinality of the fact table and result cardinality of the join. I.e., the join selectivity indicates how much a specific dimension reduces the cardinality of a fact table. As a general rule, we want to consider joins with higher selectivity first.

VI. Execution Strategies

In this section we review a number of execution strategies to compute star joins. Our starting point is an execution plan that results from using standard optimization heuristics for join queries, such as avoiding "bushy" tress and cross products. From this starting point, we show a number of opportunities for improved execution plans.

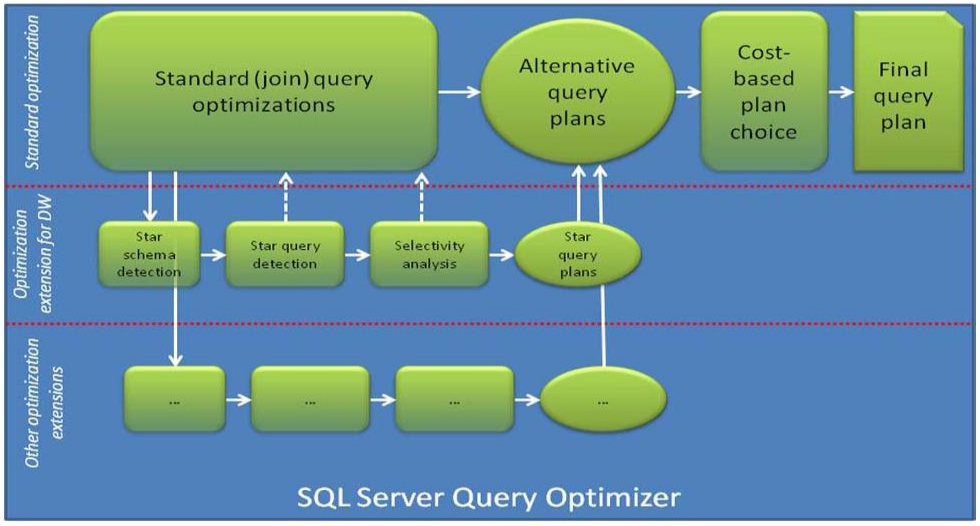


Fig. 3 Overview of the optimization process and data warehouse specific extensions

A. Query Processing Background

We start by reviewing very briefly some query processing basics relevant for star join queries. Thorough descriptions of query execution algorithms can be found in standard database texts, or survey papers, e.g. [1].

Hash join. A *build* input is read in its entirety to build an in-memory hash table, with the join column as its key. Then the *probe* input is read, looking up matches for each row in the hash table.

Hash join spill. If the build input is too large to fit in memory, then the joins is *spilled* to disk. This requires temporary disk storage of both build and probe inputs.

Parallel join execution. On multi-processor machines, parallel execution can be used to reduce the query response time. This is common in data warehouse scenarios, which tend to execute few concurrent long-running queries, on large data volumes. In Microsoft SQL Server, we use the *exchange* operator to implement parallelism [2]. A parallel hash join has exchange operators feeding its inputs. These exchange operators partition rows based on the join key, and flow the rows for each partition to different threads, each executing an instance of the hash join. Data output from multiple threads is then gathered together or further repartitioned for later processing.

Parallel table scan. The contents of a table can be directly handed off to different threads by the Storage Engine. This primitive operation does not partition data based on a key, but simply attempts to distribute load evenly across the multiple threads and minimize coordination work.

Optimizing join orders is combinatorically hard, even for very simple cost models [4], [5]. Real implementations have to rely on heuristics to reduce the size of the search space. We introduce next a couple of concepts from join reordering. More information can be found in [3], [4].

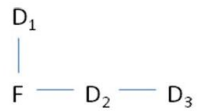
Query graph. A query graph represents the tables and join conditions in a query. Each table corresponds to a node in the graph, and each predicate between columns of two tables corresponds to an edge between the nodes of those tables. The names "star join" and "snowflake join" are descriptive of the shape of the query graphs for those queries.

No cross-product heuristic. A common heuristics used in join optimization is to avoid cross products, because they can explode the size of intermediate results and the cost of processing.

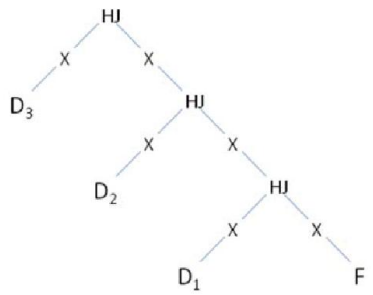
Linear join heuristic. Another common heuristic to avoid considering "bushy" trees, that is, plans where both inputs of a join are produced themselves by joins. A plan that is not "bushy" is called "linear." Executing "bushy" trees is somewhat more involved, and this tree topology does not provide benefits for navigational, index-lookup join scenarios. Like the previous heuristic, it can cut significantly the optimizer search space.

B. Baseline execution plan

Throughout this section we will discuss various execution plans for the following query graph.



The fact table F is directly joining with dimensions D_1 and D_2; D_2 in turn is joined with D_3. Each of the dimension tables may have some filtering conditions, which are not shown in the query graph. Restricted by the common join optimization heuristics, the baseline plan for this query is the following. Assume that the join with D_1 is more selective on F than the join with (D_2, D_3), so it is executed first.



The build side of hash joins (HJ) is drawn as the left child, and the probe side is on the right. Assume the system has enough main memory to hold the three dimension tables at the same time. All hash tables will be built first, and the contents of F will be pipelined up the tree. Each row of F goes through two steps for each join: First the row is assigned to the right thread by the exchange operator (X), then a probe is made to the hash table to find the matching rows from the dimension.

The bulk of the cost for this execution plan is in (1) reading the fact table, and (2) executing each join. In terms of CPU cost, partitioning and probing rows from the fact table is the dominant cost.

If main memory is not large enough to hold all the dimension tables, then at least one hash join needs to spill. The cost of the plan goes up as rows from both inputs need to be written and then read back from disk.

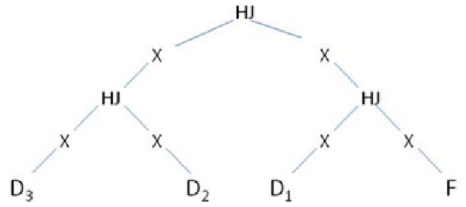
We now go through multiple optimization techniques over this baseline execution plan. The following two high-level observations hold for the collection of techniques presented next:

1) There is no one best technique for all combinations of selectivity and table sizes. Each can introduce significant performance *improvement* when used in the right situation, but can also perform worse than the baseline when used in the wrong situation. Cost based decisions are required and used in SQL Server.
2) The techniques are not mutually exclusive and a combination of them can be used on the execution plan of a single star or snowflake join query.

C. Bushy Trees

Snowflake joins can have multiple trees of dimension tables all rooted at the fact table. The heuristic of linear execution plans implies joining rows from the fact table even for dimensions that are further away in the query graph. In

our baseline execution plan, F is joined with both D_2 and then D_3. We can improve by computing dimension joins first, as shown in the following execution plan. Again, assume that the join with D_1 is more selective on F, so we want to execute it first.

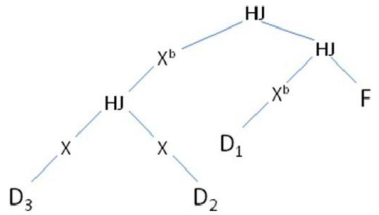


In comparison with the baseline execution plan, we have eliminated one join with F. In general, bushy trees allow joining F once for each tree of dimensions in the query graph. The impact of this optimization depends on how much F is filtered through the various joins. In this example, if joining with D_1 makes the size of F comparable to a dimension table, then there is little gain from the bushy tree. On the other hand, if the query has low selectivity, this optimization can reduce CPU cost by one third.

The argument can be generalized for other types of join queries, but it is very clear on snowflake DW scenarios due to the query graph topology and the relative sizes of tables.

D. Broadcast Exchange

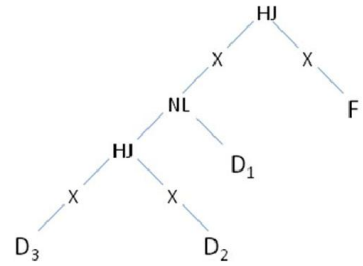
In our baseline execution plan, we mentioned that cost per join is dominated by the number of rows to process from the fact table and is made up of two components: (1) distributing rows to the appropriate thread; (2) probing and finding matches. Instead of partitioning both sides of the join input, we can instead *broadcast* the contents of the dimension table to each parallel thread, and avoid partitioning the rows of the fact table. This is valid due to join distributivity over union, i.e., T JOIN (R_1 UNION R_2) = (T JOIN R_1) UNION (T JOIN R_2). The execution plan looks as follows, where X^b denotes broadcast exchange.



Compared with the previous strategy, use of broadcast exchange can cut the CPU time by up to 50%, for queries that are not highly selective on F, as long as the hash tables fit in memory. The issue here is the increase on the size of hash tables, which will be multiplied by the degree of parallelism. If this increase in memory leads to hash join spilling, the net result will be degradation in performance. For this reason, we utilize broadcast exchange only when the build side is relatively small.

E. Cross Products

Cross products of dimension tables can be used to reduce the number of joins with the fact table. The following execution plan trades two joins with F for a cross product of dimensions and a multi-column join. The nested loops (NL) operation computes the cross product of D_1 with (D_2, D_3).

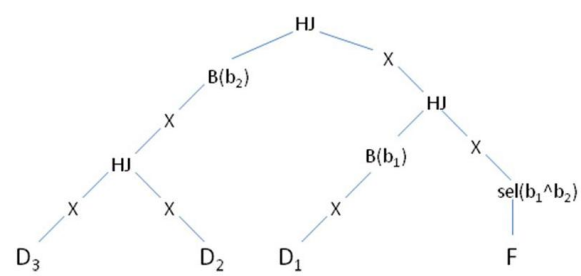


A delicate issue for this plan is the potential size explosion from the cross product. The resulting hash table can be very large and lead to spilling, resulting in net performance degradation even though a join is eliminated.

We currently do not generate this strategy in SQL Server, due to its risk. However, we do consider cross product in combination with multi-column indices, as is explained later in this paper.

F. Approximate Semi-Join Reduction for Medium-Selectivity Queries Using Bitmap Filters

When the fact table is restricted through predicates in various dimensions, the sequence of binary joins will filter the fact table gradually. We can leverage the fact that hash join reads the build side entirely, before starting to process the probe side. As we read the build input, we assemble a lossy bitmap representation of join key values seen. The bitmap is sized based on the number of values in the key domain, bits are set based on a hash function. Such bitmap is then applied to filter rows of F early and avoid join processing. The plan can look as follows, where operation B(b) sets up a bitmap of key values seen.



This strategy can achieve full filtering of rows from F before going through data partitioning and join execution logic. It also makes join order less important, because filtering is done up front.

Two operations were added to the execution plan, bitmap computation and membership test. The additional cost of bitmap computation cost is negligible, considering the relative table sizes. The cost of membership test is relatively cheap,

compared with joining, but it needs to be applied on all rows of F.

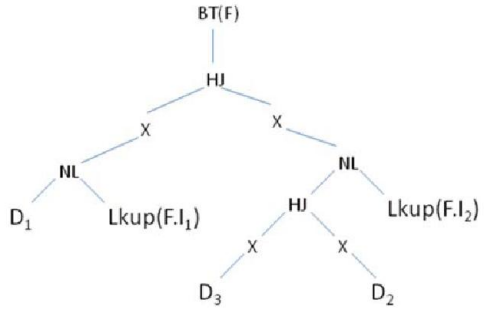
Since bitmap filters are hash based they are not necessarily precise. Multiple column values can be mapped to the same bit in the bitmap. This can generate false positives during membership test; i.e., a fact table row might pass the bitmap filter even though this row does not qualify in the join. For very small dimensions with only a few rows this is not an issue. In this case the bitmap filters are virtually exact and very rarely generate falls positives. For dimensions with several thousand or even millions of rows Microsoft SQL Server determines the size of the bitmap filter as a function of the distinct value estimate of the dimension to control the effectiveness of the filter. In all our experiments we have rarely seen the rate of false positives of a bitmap filter as a problem.

Bitmap filters are dynamically introduced during query execution and they are discarded after the execution is complete. Therefore they do not incur any additional disk space and maintenance overhead. Dynamic creation of bitmap filters is very efficient and in case of hash joins the bitmap filter can even be derived from the hash table. Note however that dynamic bitmap filters are not restricted to hash joins. They can be used in combination with other stop-and-go operators – operators which require processing of one input in entirety before the other input can be processed. A hash join fulfills this requirement, but so does a merge join if the input needs to be sorted.

The initial order of bitmap filter evaluation (i.e., membership test) is based on the selectivity of the join producing the bitmap: the most selective join will generate the most selective bitmap filter which will be evaluated first. Join- and bitmap filter-selectivities are only estimates. The actual selectivities depend on the data distribution of the selected rows. Therefore, Microsoft SQL Server monitors the actual selectivity of a bitmap filter during runtime and reorders bitmap filters dynamically to guarantee that the most selective filter is evaluated first.

G. Index Intersection

In our baseline query plan, we discussed two main cost components: (1) reading the fact table and (2) executing each join. Earlier strategies have focused on reducing component (2). We now consider two execution strategies targeting reduction of component (1).



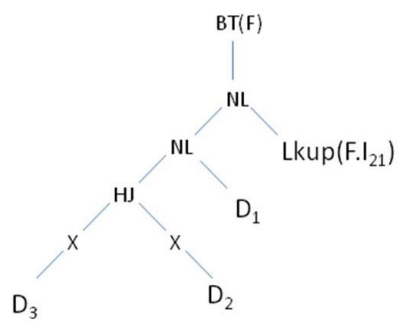
When the fact table has indices supporting joins with each dimension, we consider finding the qualifying rows on each of those indices and intersecting the resulting record ids (RIDs). We typically require accessing the fact table after RID intersection, to retrieve additional columns for the qualifying rows. We use BT(F) in the plan to denote the operation of accessing the base table F through a RID, to get required columns. We use Lkup(F.I) to denote index lookup on table F. Hash join is used in this example to implement index intersection. The plan shows parallel execution, but for selective enough queries a serial execution may be picked to avoid parallelism overhead.

The main cost of this query comes from the final RID-based fact table access to retrieve columns. Compared to hash-based plans, index-based plans have the advantage, in general, of getting only the necessary rows from disk, instead of the whole table. However, they have the disadvantage of requiring random reads.

When the combined selectivity of dimension joins is high enough, these index-based plans are by far the most efficient. But as selectivity goes down, the cost of random I/O can quickly kill the performance.

H. Multi-Column Index Lookup

In addition to index intersection, Microsoft SQL Server also considers computing the cross product of dimension tables to lookup into a multi-column index. The plan looks as follows:



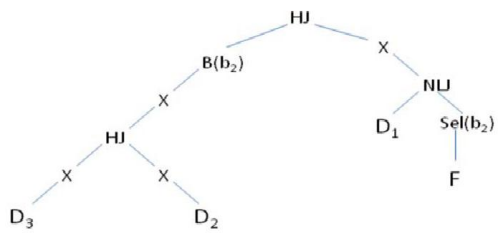
This strategy has the risk of intermediate size explosion through the cross-product operation, so it used only for small dimension tables. Unlike the cross-product strategy presented earlier, there is no need to take up temporary memory to materialize the intermediate result. As cross-product rows are computed, they are directly used to lookup into the index of F.

I. Combining Nested Loops Join with Right-Deep Hash Joins

In many data warehousing schemas the fact table is clustered on the surrogate key of one of the dimension tables. E.g., the fact table representing details of sales transactions might be clustered on date and/or time.

Date is typically one of the dimensions and many queries will restrict the result to a certain date range. In such a scenario it might be more efficient to read all qualifying rows from the date dimension and request only qualifying rows from the fact table. Since the fact table is clustered on date,

this can be done very efficiently by sequentially scanning the qualifying range of rows and without processing all rows of the fact table. In this particular context, such a nested loop join is usually much more efficient than a hash join which has to process all fact table rows. Therefore, SQL Server considers this additional plan alternative if the fact table is clustered on one of the dimensions used in the query. Assuming that D_1 is the dimension the fact table is clustered on, this leads to the following execution plan:



VII. EXPERIMENTAL EVALUATION

A. Synthetic Selectivity Range Analysis

The objective of these experiments is (1) to investigate the performance characteristics of star join queries in the different selectivity ranges and (2) to relate actual query plans observed to the query plans introduced in this paper.

Experimental setup. We conducted this series of experiments with a variant of the AdventureWorksDW database that ships as a sample database with Microsoft SQL Server. Our variant of AdventureWorksDW holds about 250'000 rows in its main fact table as opposed to the original version with only 50'000 rows. AdventureWorksDW is well-suited to investigate star join queries because it is dimensionally modeled using a snowflake schema.

Our experiments use the star join query introduced in Section I. as a template. Different instantiations of the template vary the predicates in its WHERE clause to investigate the following selectivity ranges:

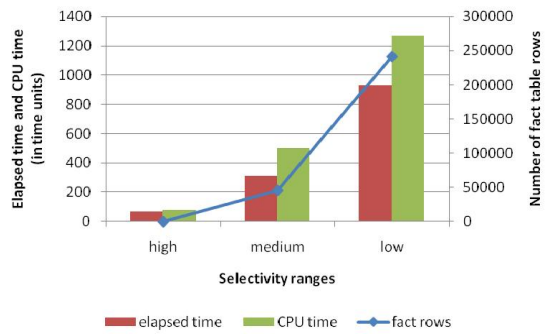
- *High selectivity*: less than 1% of the fact table rows qualify for the query result.
- *Medium selectivity*: about 25% of the fact table rows qualify for the query result.
- *Low selectivity*: all fact table rows contribute to the query result.

Each of these data-points fall into one of the three selectivity ranges defined in Section III.

Our experiments track CPU time, elapsed time, number of rows processed from the fact table, and the query plans used to process the queries.

Outcome and discussion. The figure below plots the results for this series of experiments. As the figure shows, both elapsed time and CPU resource consumption of a query increase sub-linearly with the number of fact table rows in the query result: as we increase the number of fact table rows by a factor of 250 from about 200 (with high selectivity) to 50'000 (for medium selectivity), elapsed time and CPU time only increase by a factor of 8. A similar observation holds for the step from medium to low selectivity: as we increase the number of fact table rows by a factor of 5 from 50'000 to about 250'000, both elapsed time and CPU consumption only increase by a factor of 3.

The default choice of query plan for the different selectivity classes confirms our findings: The high selectivity class queries use a variant of the index intersection plan shapes discussed in this paper. The medium selectivity class in turn relies on semi-join reduction plans using multiple bitmap filters. The low selectivity class finally uses standard right-deep hash joins to implement the query.



B. Real-world Workload Study

As part of the development effort for the functionality discussed in this paper, we have been tracking performance for several "real-world" customer workloads and industry benchmarks. The following paragraphs discuss our most important findings from this effort.

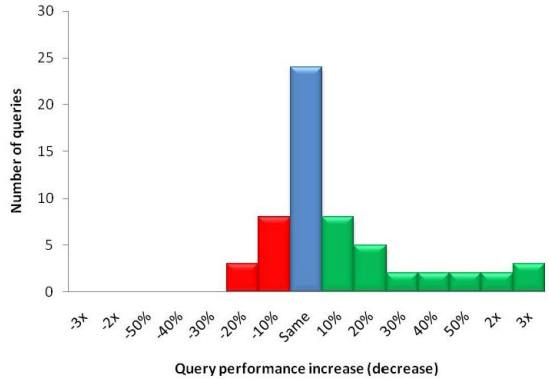
Our experiments cover standard performance characteristics like overall time required to complete the workload. However, the main metric tracked for all workloads is how many queries experience what improvement or what degradation of wall clock response time as compared to a baseline where the specific star join optimizations discussed in this paper are disabled. In an ideal world, all queries would perform better with the new functionality. However, reality is not always ideal and real-world query processing is subject to statistical adversities, e.g., during cardinality estimation, or when unknown correlations are present in the data.

All the subsequent charts therefore plot the number of queries in performance improvement or degradation buckets ranging from three times degradation of query response time (-3x) up to three times improvement of response time (3x).

Retail customer. This series of experiments is based on a data warehousing customer in the online retail business. The customer is characterized by a dimensionally modeled snowflake schema and canonical star join queries. The amount of raw data we used to populate a snapshot of the warehouse for our experiments was about 100GB.

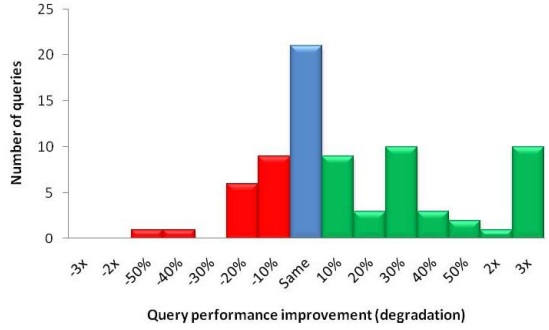
The figure below charts the number of queries in the customer workload against the different query performance improvement and degradation buckets. Overall, this workload fares very well with the new star join query optimizations. 11 queries face a small performance degradation of less than 20% in terms of query response time, while 24 queries achieve significant performance gains with speedup up to a factor of 3.

Star join optimizations yield an improvement of about 17% in expected response time for queries in this workload.

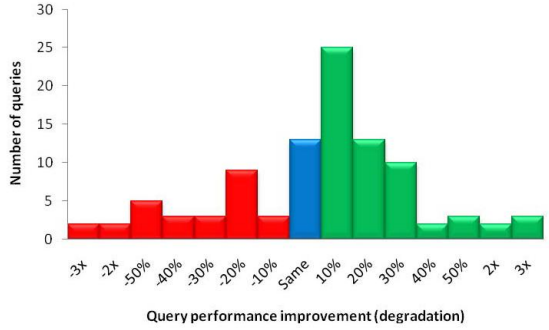


Query degradation is mainly due to the use of heuristics. For example, detection of fact table as well as dimension tables is based on heuristics. Hence, the optimizer may falsely detect star join pattern. Another reason is limitations in the optimizer model, e.g., exhaustive enumeration of join orders is prohibitive.

Decision support benchmark. This series of experiments investigates performance for a snapshot of the decision support benchmark TPC-DS currently under development by the TPC [15]. Note that the benchmark is still under development and we just used a snapshot as a performance tracking workload. Our experiments covered the 100GB scale factor of the benchmark. The figure below reports on the outcome of these experiments. As the figure shows, queries are much more likely to benefit from the new functionality than to face an increase in response time: out of about 80 queries that we tracked, 38 run faster with star join optimizations while only 17 queries run slower. The biggest performance improvements also dramatically outweigh the biggest degradation: while only 2 queries experience degradation by 40% or 50%, respectively, 10 queries achieve performance improvements of a factor of 3 or more. Overall, our star join optimizations improve expected query response times in this workload by 23%.



Microsoft sales organization data warehouse. This workload tracks performance for a data warehouse that is used internally for decision support in Microsoft's sales organization. We took a sample snapshot of the database with a size of about 750GB which includes indexes. Queries in this workload are challenging for query processing – many of them have more than 10 joins. The figure below graphs the results for this workload. A first observation is that a few queries face significant degradations with increases in response time by up to a factor of 3. We have performed a more detailed analysis of these cases and identified issues beyond the scope of this paper such as cross-table data correlations as the root cause. We currently see this as an important area for further research and development. However, the overall experience also for this workload is a positive one: out of 99 queries, 58 queries ran faster and only 27 queries performed worse. The wall-clock time needed to run the complete workload improved by 18%.



VIII. Related Work

Data warehousing, decision support, and online analytical processing (OLAP) have been an active area of research since more than a decade [14]. Performance and efficient execution strategies for star join queries are essential in this context which is reflected in the amount of previous work, e.g. [11], [12]. Related execution strategies are for instance the RID intersection approach discussed in [13]. Semi-join reductions which we leverage for the bitmap filters discussed in this paper have been investigated intensively in earlier work on local hash join implementation and distributed query processing, e.g., [10], [16].

Commercial database system vendors have also made significant investments into data warehousing support. IBM/Informix RedBrick for instance considers index union and semi-join reduction plans with bitmap filters similar to the strategies described in this paper [9]. Oracle 10g R2 in turn has a dedicated optimization mode for star join queries. In contrast to our approach, it requires the presence of bitmap indexes (in contrast to bitmap filters) to leverage its full performance advantages [8]. The major contribution of our approach is its seamless integration into the regular query optimization and query execution pipeline. This allows us to consider query plans derived from normal plan generation and specific data warehousing optimizations and compare them in a cost-based fashion to pick the most promising alternative.

IX. Conclusions

Today's relational data warehouses are usually modeled dimensionally using fact tables and dimension tables – which

leads to the well-known schema types of a snowflake or a star schema. Decision support queries follow a typical pattern that joins the fact table with one or several dimension tables. These queries are therefore denoted as star join queries.

Relational database query processors can leverage the common schema shape and query structure to offer additional optimizations specifically targeted at data warehousing scenarios. Based on a pattern matching approach along these lines, this paper contributes novel execution strategies for star join queries such as index intersection, dimension cross-product with fact table lookup, and semi-join reduction using bitmap filter technology. Based on estimated selectivity of the query against the fact table, we augment the plan search space with plan alternatives most promising for this selectivity class while ultimate plan choice is still performed in a cost-based fashion. We complement the new optimization strategies with sophisticated execution-time improvements, such as autonomic dynamic re-ordering of bitmap filters at query runtime.

An extensive performance study investigated both synthetic workloads as well as real-world customer workloads. Our results show that indeed different plans are appropriate in different selectivity classes and that the star join performance optimizations yield the expected performance improvements.

The optimization and execution techniques discussed in this paper have been added to Microsoft SQL Server over the course of several releases with multiple bitmap filter semi-join reductions being available in SQL Server 2008[1].

[1] The functionality discussed in this paper is planned for SQL Server 2008. Pre-release versions of SQL Server 2008 including all the techniques discussed here are available for public download at http://www.microsoft.com/sql.

REFERENCES

[1] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing. Survey*. 25(2): 73-170 (1993).

[2] G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System," in *Proc. SIGMOD 1990*, p.102.

[3] K. Ono, and G. Lohman, "Measuring the Complexity of Join Enumeration in Query Optimization," in *Proc. VLDB 1990*, p. 314.

[4] S. Cluet, and G. Moerkotte, "On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products," in *Proc. ICDT 1995*, p. 54.

[5] T. Ibaraki, and T. Kameda, "On the Optimal Nesting Order for Computing N-Relational Joins," *ACM Transactions on Database Systems*, vol. 9(3), pp. 482-502, 1984.

[6] R. Kimball, and M. Ross, *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*, 2nd ed., 2002.

[7] RedBrick White Paper, available: ftp://ftp.software.ibm.com/-software/data/informix/pubs/whitepapers/redbrick_wp040904.pdf.

[8] Oracle Data Warehousing Guide 10g R2, available: http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14223.pdf.

[9] A. Weininger, "Efficient execution of joins in a star schema," in *Proc. SIGMOD 2002*, p. 542.

[10] P. A. Bernstein, and D.-M. W. Chiu, "Using Semi-Joins to Solve Relational Queries," *Journal of the ACM*, vol. 28(1), pp. 25-40, 1981.

[11] R. Pieringer, K. Elhardt, F. Ramsak, et al., "Combining Hierarchy Encoding and Pre-Grouping: Intelligent Grouping in Star Join Processing," in *Proc. ICDE 2003*, p. 329.

[12] N. Karayannidis, A. Tsois, T. K. Sellis, et al., "Processing Star Queries on Hierarchically-Clustered Fact Tables," in *Proc. VLDB 2002*, p. 730.

[13] V. Raman, L. Qiao, W. Han, et al., "Lazy, adaptive rid-list intersection and its application to index anding," in *Proc. SIGMOD 2007*, p. 773.

[14] S. Chaudhuri, and U. Dayal, "An Overview of Data Warehousing and OLAP Technology," *SIGMOD Record*, vol. 26(1), pp. 65-74, 1997.

[15] The Transaction Performance Council (TPC) website [Online]. Available: http://www.tpc.org.

[16] Goetz Graefe, Ross Bunker, and Shaun Cooper, "Hash Joins and Hash Teams in Microsoft SQL Server," in Proc. VLDB 1998, p. 86.