Processing Star Queries on Hierarchically-Clustered Fact Tables

Nikos Karayannidis[1], Aris Tsois[1], Timos Sellis[1], Roland Pieringer[2], Volker Markl[4], Frank Ramsak[3],Robert Fenk[3], Klaus Elhardt[2], Rudolf Bayer[5]

[1] Institute of Communication and Computer Systems and Department of Electrical and Computer Engineering, National Technical University of Athens, Zographou 15773 Athens, Hellas {nikos, atsois, timos}@dblab.ece.ntua.gr

[2] TransAction Software GmbH Gustav-Heinemann-Ring 109,D-81739 München, Germany {pieringer, elhardt}@transaction.de

[3] Bayerisches Forschungszentrum für Wissensbasierte Systeme, Orleansstraße 34, D- 81667 München, Germany, {robert.fenk, frank.ramsak}@forwiss.de

[4] IBM Almaden Research Center, K55/B1, 650 Harry Road, San Jose, CA 95120-6099, marklv@us.ibm.com

[5] Institut für Informatik, TU-München, Orleansstraße 34, D-81667 München, Germany, bayer@in.tum.de

Abstract

Star queries are the most prevalent kind of queries in data warehousing, OLAP and business intelligence applications. Thus, there is an imperative need for efficiently processing star queries. To this end, a new class of fact table organizations has emerged that exploits path-based surrogate keys in order to hierarchically cluster the fact table data of a star schema [DRSN98, MRB99, KS01]. In the context of these new organizations, star query processing changes radically. In this paper, we present a complete abstract processing plan that captures all the necessary steps in evaluating such queries over hierarchically clustered fact tables. Furthermore, we present optimizations for surrogate key processing and a novel early grouping transformation for grouping on the dimension hierarchies. Our algorithms have been already implemented in a commercial relational database management system (RDBMS) and the experimental evaluation, as well as customer feedback, indicates speedups of orders of magnitude for typical star queries in real world applications.

Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

1. Introduction

Data warehousing (DW) has evolved into a major trend in database technology through the last decade. Furthermore, the multidimensional paradigm seems to be the undisputed winner as a design choice for such databases. Regardless of the underlying physical layer, relational technology or proprietary multidimensional structures, the conceptual model adopted is a data warehouse consisting of facts (or measures) organized into a set of dimensions, which in turn are organized into levels of different aggregation (i.e., detail) that comprise one or more hierarchies. In particular, for relational databases, the multidimensional data warehouse consists of one or more *star schemata* [CD97a].

The information stored in a star schema is in the form of detailed facts organized by dimension values and can produce all kinds of invaluable insights with appropriate querying. The most prevalent kind of query submitted to such a system is the *star query*. Star queries impose restrictions on the dimension values that are used for selecting specific facts; these facts are further grouped and aggregated according to the user demands. The major bottleneck in evaluating such queries has been the join of the central (and usually very large) fact table with the surrounding dimension tables (also known as a *star join*). To cope with this problem various indexing schemes have been developed [NG95, NQ97, Sar97, CI98, WB98, Wu99, WOS01]. Also precomputation of aggregation results has been studied extensively - mainly as a view maintenance problem - and is used as a means of accelerating query performance in data warehouses [GM95, Rou98, SDJL96].

However, the need for doing On-Line Analytical Processing (OLAP) on the data makes processing of *ad hoc* star queries, i.e. queries that are not known in advance, also a necessity. For this kind of queries the usage of precomputed aggregation results is extremely limited or even impossible in some cases. Even when elaborate indexes are used, due to the arbitrary ordering of the fact table tuples, there might be as many I/Os as are the tuples resulting from the fact table. The only alternative one can have for such queries is a good physical clustering of the data, and it is exactly for this reason that a new class of primary organizations for the fact table has emerged [DRSN98, MRB99, KS01]. These organizations exploit a special kind of key that is based on the hierarchy paths of the dimensions, in order to achieve hierarchical clustering of the facts. This physical clustering results in a reduced I/O cost for the majority of star queries, which are based on the dimension hierarchies. Moreover, [MRB99] and [KS01] exploit a multidimensional index for storing the tuples. A typical star join is transformed then into a multidimensional range query, which is very efficiently computed using the underlying multidimensional data structures.

In this paper, we study the processing of ad hoc star queries over hierarchically clustered fact tables. We show that the processing entailed is significantly different from the ones in previous approaches. In particular, we present a complete abstract processing plan that covers all the necessary steps for answering such queries. This plan directly exploits the benefits of hierarchically clustered fact tables and opens the road for new optimization challenges. To this end we propose optimizations for the processing of surrogate keys, when evaluating dimension restrictions, and a novel early grouping transformation that drastically reduces the number of fact table tuples participating in a sequence of joins with dimension tables. Our proposals have already been implemented in a commercial RDBMS [TBHC] and have been deployed to customers. Finally, we present preliminary measurements that have been confirmed in real-life applications and show significant performance gains for typical star queries.

The rest of the paper is organized as follows. In section 2 we give an overview of related work. Section 3 describes the general concept of processing star queries using hierarchical clustering, while section 4 contains optimizations for the proposed processing plan. Measurement results are presented in section 5. Section 6 recapitulates our work and summarizes the benefits of our proposal compared to the conventional processing plans of star queries.

2. Related Work

One of the most important parts of a star query is the processing of the star join. Star join processing has been studied extensively and specific solutions have been also implemented in commercial products. See [CD97b] for an overview.

The standard query processing algorithm to execute a star join over n dimensions first evaluates the predicates on the dimension tables, either on normalized (snowflake) or de-normalized (star) schema, resulting in a set R_i of n_i tuples of dimension D_i. It then builds a cartesian product of the dimension result tuples (R_1 x R_2 x … x R_n). The cardinality of the cartesian product is $n_1 \cdot n_2 \cdot \ldots \cdot n_n$ for the n restricted dimensions. With these cartesian product tuples, we perform a direct index access on the composite index built on the fact table. For non-sparse fact tables and queries that restrict most dimensions of the composite index in the order of the index attributes, the access to the fact tuples is quite fast. The next processing step then joins the resulting fact tuples with the dimension tables in order to allow grouping and aggregating.

However, for large sparse fact tables and high dimensionality, such a query processing plan does not work efficiently enough. The cardinality of the cartesian product resulting from the dimension predicates grows very fast, whereas the number of affected tuples in the fact table may be relatively small. This is the point where a call is made for specialized indexing or clustering methods.

Bitmap indices are often used to speed up the access to the fact table. The bitmaps corresponding to the different dimension values are ANDed or ORed depending on the selection condition. The resulting bitmap is used to extract tuples from the fact table [NG95, NQ97]. When the query selectivity is high, only a few bits in the result bitmap are set. If there is no particular order among the fact table tuples, we can expect each bit to access a tuple in different page. Thus there will be as many I/Os as there are bits set.

Multidimensional clustering has been discussed in the field of multidimensional access methods (e.g., [GG97] and [Sam90]). [ZSL98] addresses the issue of hierarchical clustering for the one-dimensional case. The importance of good physical clustering in OLAP has been shown in [KR98], where packed R-trees are exploited for storing the results of the data cube operator ([GBLP96]). In [DRSN98], the benefits of hierarchical clustering for star queries was observed as a side effect of using a chunked file organization for enabling caching with chunk as the caching unit.

Among others, in [MRB99] the UB-tree multidimensional index [Bay97] is used as a primary organization of the fact table. Surrogate keys based on the dimension hierarchies are exploited and hierarchical clustering of the fact table is achieved. Consequently star joins are transformed to multidimensional range queries. The combination of the two mechanisms results in a greatly reduced I/O cost for star joins.

In [KS01] a physical organization based on a hierarchical chunking of the fact table is presented. Fact data are clustered physically according to the dimensional hierarchies. To achieve this clustering, special path-based dimension keys are exploited. In particular, these keys guide the clustering (called *chunking*) process. Star joins are transformed to range queries in the multidimensional and multi-level data space of a cube. The adopted multidimensional structure is a variant of the Grid File [NHS84].

Several aspects of processing and optimizing star join queries on hierarchically clustered fact tables are also presented in [TT01]. The paper considers a star schema with UB-Tree organized fact tables and dimension tables stored sorted on a composite surrogate key. For a particular class of star join queries, the authors investigate the usage of sort-merge joins and a set of other heuristic optimizations.

Based on the significance of the above organizations, it is clear that there is a need for a general query processing framework that addresses all issues involved in star query processing over hierarchically clustered fact tables. We proceed next to present such a framework.

3. Processing Star Queries

3.1 Preliminary Concepts

OLAP data are divided into two main categories. The *measures* (or *facts*) are mainly numeric values, which correspond to measurements of some value related to an event at specific points in time (e.g., amount of money appearing in a line of an invoice at a particular day, or balance of an account at the end of each day, etc.) and are expected to change rapidly. The *dimension data* (or simply *dimensions*) are used to characterize the measures and are considered to be almost static in (or slowly changing with) time. The dimension values characterize a specific measure value in the same way that coordinate values characterize a specific point in a multidimensional space. Examples of dimensions for a retailing business can be *DATE*, *PRODUCT*, *CUSTOMER*, *LOCATION* etc.

Each dimension represents a distinctive property of a measure. In a relational OLAP (ROLAP) implementation a dimension is stored into one or more *dimension tables* $\{D_1, D_2, D_{3\ldots}\}$ each having a set of attributes. In the simplest case, a dimension is represented by only one table with only one attribute, say h_1. Based on the values of h_1 one may add additional attributes ($h_2, h_3, \ldots$) to the dimension table in order to form a classification hierarchy. In this case the h_1 attribute is classified by the h_2 attribute, which is further classified by the h_3 attribute, etc. We call the attributes h_1, h_2, h_3, … *hierarchical attributes* because they participate in the definition of the hierarchy. For example *day*, *month* and *year* can be a hierarchical classification in the *DATE* dimension. In general, a single dimension may contain many different hierarchical classifications that stem from a common grain level (i.e., the most detailed level). For the purposes of this paper we will assume a single hierarchy for each dimension.

A dimension table may also contain one or more *feature attributes f*. A feature attribute is a descriptive attribute and is semantically different from a hierarchical attribute in that it cannot participate in the dimension hierarchy. Feature attributes contain additional information about a number of hierarchical attributes and are always functionally dependent on one (or more) hierarchical attribute. For example, *population* could be a feature attribute dependent on the *region* attribute of dimension *LOCATION*.

3.2 Database Schema

As mentioned earlier, the dimensions are used to characterize measures, which in turn are stored in *fact tables*. A fact table may contain one or more *measure attributes* and is always linked (by foreign key attributes) to some dimension tables. This logical organization consisting of a central table (the fact table) and surrounding tables (the dimension tables) that link to it through 1:N relationships is known as the *star schema* [CD97a]. In a typical scenario, the hierarchical attribute representing the most detailed level will be the primary key of the respective dimension. Each such attribute will have a corresponding foreign key in the fact table.

In order to create a fact table that is clustered according to the dimension hierarchies we first need to apply a *hierarchical encoding* (HE) on each dimension table. We achieve this by assigning to each dimension table D containing the hierarchical attributes h_m, h_{m-1}, …, h_1 (h_m being the most aggregated level and h_1 the most detailed one) a *surrogate key* (*sk*) attribute that has a unique value for each tuple. This is something very common in data warehousing practice, since surrogate keys provide a level of independence from the keys of the tables in the source systems [Kim96]. In our case, surrogate keys are defined over h_m, h_{m-1}, ..., h_1 and are essentially the means to achieve hierarchical clustering of the fact table data. We will refer to these keys as *hierarchical surrogate keys (hsk)* or simply *h-surrogates*.

The main idea is that an h-surrogate value for a specific dimension table tuple is constructed as a combination of encoded values of the hierarchical attributes of the tuple. For example, if h_1, h_2, h_3 are the hierarchical attributes of a dimension table from the most detailed level to the most aggregated one, then the h-surrogates for this dimension table will be represented by the values $oc_1(h_3)/oc_2(h_2)/oc_3(h_1)$, where the functions oc_i (i = 1,2,3) define a numbering scheme for each hierarchy level and assign some *order-code* to each hierarchical attribute value. Obviously the h-surrogate attribute of a dimension table is a key for this table since it determines all hierarchical attributes, which in turn functionally determine all feature attributes. The h-surrogate should be a system assigned and maintained attribute, and typically should be made transparent to the user.

The actual implementation of the hierarchical surrogate keys depends heavily on the underlying physical organization of the fact table. Proposals for physical organizations [MRB99, KS01] exploit such path-based surrogate keys in order to achieve hierarchical clustering of the stored data of a fact table.

In this paper we adopt a de-normalized approach for the design of a dimension, i.e., we represent each dimension with only one table. The hierarchical attributes (h_1, h_2, …,h_m), the feature attributes (f_1, f_2, …, f_k) as well as the hierarchical surrogate key *hsk* are stored in a unique dimension table D. De-normalization of the dimension tables is a common data warehousing practice. It is based on the rationale that the major overhead in storage space comes from the fact table and therefore, normalizing the dimension tables will not exhibit any significant space savings. On the other hand, de-normalization enhances performance significantly, since it avoids the consequent joins between the tables of the same dimension. Although the alternative of normalized schemata (also known as *snowflake schemata* [CD97a]), is also another option, in this paper we will not address it due to lack of space and for the sake of simplicity of the presented abstract processing plan (see section 3.4). However, our ideas are fully applicable to normalized schemata as well, with the only difference that extra joins between the several dimension tables (corresponding to separate hierarchy levels) must be included in the plan.

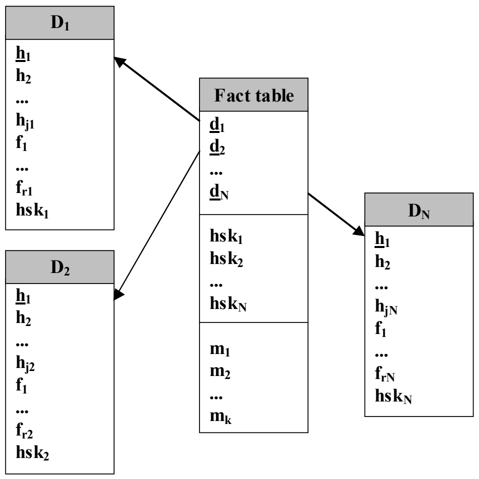

Figure 1: Star schema with flat dimension tables

The star schema of Figure 1 is a typical star schema where the dimension tables have been hierarchically encoded. This schema consists of N dimensions stored in the dimension tables D_1, …, D_N. Each dimension is logically structured in a hierarchy. The hierarchy elements for dimension D_i are h_1, h_2, … h_{ji}. Each dimension table D_i may also include a set of feature attributes $f_1, f_2, \ldots, f_{ri}$ that characterize one or more hierarchical attributes. In Figure 1 we depict h_1, i.e. the most detailed level in each hierarchy as the primary key of each dimension table. In Figure 1 we can also see the h-surrogate attribute (hsk_i), which is an alternate key for each table D_i (i = 1,…N).

The fact table contains the measure attributes (m_1, m_2, …m_k), the reference to the h-surrogate of each dimension (hsk_1, hsk_2, ..., hsk_N) and a reference to the most detailed hierarchical attribute of each dimension (d_1, d_2, ...,d_N). Hence, d_1 is a reference to h_1 of D_1, d_2 is a reference to h_1 of D_2 and so on. All measure values refer to the most detailed level of the hierarchy of each dimension. For an example of a star schema, the reader is referred to Figure 4 of section 3.5.

In the fact table of Figure 1, we have two alternative composite keys: (a) (d_1, d_2, ...,d_N) that links to the corresponding lowest hierarchical attribute of each dimension and (b) (hsk_1, hsk_2, ..., hsk_N) that links to the h-surrogate attribute. Note that the former is not necessary in order to achieve hierarchical clustering of the data and thus could be omitted in order to reduce storage overhead.

In this paper, we use a special physically organized schema. The fact table is stored hierarchically clustered in a multidimensional index, i.e., the index attributes of this clustering index are the h-surrogates.

3.3 Star Queries

OLAP queries typically include restrictions on multiple dimension tables that trigger restrictions on the (usually very large) fact table. This is known as a *star join*. In this paper, we use the term *star query* to refer to flat SQL queries, defined over a single star schema, that include a star join. Star queries represent the majority of OLAP queries. In particular, we are interested in *ad hoc* OLAP queries. With the term "ad hoc" we refer to queries that are not known in advance and therefore the administrator cannot optimize the DBMS specifically for these.

In Figure 2, we depict an SQL query template for ad hoc star queries. The template defines the most complex query structure supported and uses abstract terms that act as placeholders. Note that queries conforming to this template have a structure that is a subset of the above template and instantiate all abstract terms.

Our template will be applied on a schema similar to the one in Figure 1, which is a typical star schema. Looking at the part containing the join constraints between the fact table and the dimension tables (**JC**), we see that it includes a star join. Apart from the star join, there is a `GROUP BY` and `HAVING` clause (**HP**). In general any attribute (hierarchical, feature, or measure) can appear in a `GROUP BY` clause (**GA$_h$**, **GA$_f$**, **GA$_m$**). However, most queries impose a grouping on a number of hierarchical and/or feature attributes. Finally, there is an `ORDER BY` clause for controlling the order of the presented results (**OL**).

LOCPRED$_i$(D_i) is a *local predicate* on a dimension table D_i (**LP**). The characterization "local" is because this predicate includes restrictions only on D_i and not on other dimension tables or the fact table. This predicate is very important for the h-surrogate processing phase explained later, and is used to produce the necessary h-surrogate specification for accessing the fact table.

Note that the vast majority of OLAP queries contain an equality restriction on a number of hierarchical attributes and more commonly on hierarchical attributes that form a complete path in the hierarchy. E.g., the query "show me sales for area A in region B for each month of 1999" contains two whole-path restrictions, one for a dimension *LOCATION* and one for a *DATE*: (a) *LOCATION.region* = 'A' AND *LOCATION.area* = 'B' and (b) *DATE.year* = 1999. This is reasonable since the core of analysis is conducted along the hierarchies. We call this kind of restrictions *hierarchical prefix path* (HPP) restrictions. Note also that even if we impose a restriction on an intermediate level hierarchical attribute, we can still have an HPP restriction, as long as hierarchical attributes functionally determine higher level ones.

```
SELECT    SGA, Aggr
FROM      ft, D
WHERE     JC AND LP AND MP
GROUP BY  GA_h, GA_f, GA_m
HAVING    HP
ORDER BY  OL
```

SGA:	Selection attribute(s) of dimension table(s) ($D_i.h_j \in \mathbf{GA_h}$ or $D_i.f_j \in \mathbf{GA_f}$) and/or, measure attribute(s) of the fact table (ft.$m_i \in \mathbf{GA_m}$).
Aggr:	Aggregation function(s) (MIN, MAX, COUNT, SUM) on measure attribute(s) of the fact table (ft.m_i) and/or, on attribute(s) of the dimension table(s) ($D_i.h_j$ or $D_i.f_j$, $D_i \in \mathbf{D}$).
ft:	The fact table.
D:	Dimension table(s) involved in the query (D_1, D_2, …, D_N).
JC:	Natural join conditions; joining the fact table ft with the involved dimension tables D_i ($D_i \in \mathbf{D}$) on key-foreign key (ft.$d_i = D_i.h_1$)
LP:	A conjunction of local predicates on some of the involved dimension tables: $\mathbf{LP}$=LOCPRED$_1(D_1) \wedge$LOCPRED$_2(D_2) \wedge \ldots \wedge$LOCPRED$_k(D_k)$; $D_1, D_2, \ldots, D_k \in \mathbf{D}$
MP:	Restriction predicate on measure attribute(s) of the fact table.
GA$_h$:	Grouping hierarchical attribute(s) of dimension table(s) ($D_i.h_k$, $D_i \in \mathbf{D}$).
GA$_f$:	Grouping feature attribute(s) of dimension table(s) ($D_i.f_k$, $D_i \in \mathbf{D}$).
GA$_m$:	Grouping measure attribute(s) of fact table (ft.m_i)
HP:	Restriction predicate on grouping attributes ($\mathbf{GA_h} \cup \mathbf{GA_f} \cup \mathbf{GA_m}$) and/or on aggregation functions.
OL:	An ordered list of attributes. ($\mathbf{OL} \subseteq \mathbf{GA_h} \cup \mathbf{GA_f} \cup \mathbf{GA_m}$)

Figure 2: The ad hoc star query template

Finally, **MP** is a predicate that contains any constraints on measures of the fact table. Those constraints do not reference any dimension tables. An example would be to ask for sales figures that exceed a certain value threshold.

3.4 Abstract Processing Plan

In this section we will describe the major processing steps entailed when we want to answer star queries over a hierarchically clustered fact table.

Step 1 – Identifying relevant fact table data: The processing begins with the evaluation of the restrictions on the individual dimension tables, i.e., the evaluation of the local predicates (section 3.3). This step performed on a hierarchically encoded dimension table will result in a set of h-surrogates that will be used in order to access the corresponding fact table data. Due to the hierarchical nature of the h-surrogate this set can be represented by a number of h-surrogate intervals called the *h-surrogate specification*. Using the notation of [KS01] an interval can for example have the form $v_3/v_2/*$, where v_3, v_2 are specific values of the h_3 and h_2 hierarchical attributes of the dimension in question. The symbol '*' means all values of the h_1 attribute in the dimension tuples that have $h_3 = v_3$ and $h_2 = v_2$. In the case of a *DATE* dimension, the h-surrogate specification could be 1999/January/* to allow for any day in this month. We will show in the next section that this step can be performed very efficiently. We will use the term *range* to denote the h-surrogate specification arising from the evaluation of the restriction on a single dimension.

Once the h-surrogate specifications are determined for all dimensions, the evaluation of the star join follows. In hierarchically clustered fact tables this translates to one or more simple range queries on the underlying multidimensional structure that is used to store the fact table data. Moreover, since data are physically clustered according to the hierarchies and the ranges originate from hierarchical restrictions, this will result in a very efficient evaluation of the range selection ([MRB99]).

Step 2 – Computing necessary joins: The tuples resulting from the fact table contain the h-surrogates, the measures and the dimension table primary keys. At this stage, there might be a need for joining these tuples with a number of dimension tables in order to retrieve certain hierarchical and/or feature attributes that the user wants to have in the final result and might also be needed for the grouping operation. We call these joins *residual joins*.

Step 3 – Performing grouping and ordering: Finally, the resulting tuples may be grouped and aggregated and the groups further filtered and ordered for delivering the result to the user.

The abstract processing plan comprising of the above phases is illustrated in Figure 3 and can be used to answer the single block queries described in section 3.3. This plan is abstract in the sense that it does not determine specific algorithms for each processing step: it just defines the processing that needs to be done. That is why it is expressed in terms of *abstract operators* (or *logical operators*), which in turn can be mapped to a number of alternative *physical operators* that correspond to specific implementations.

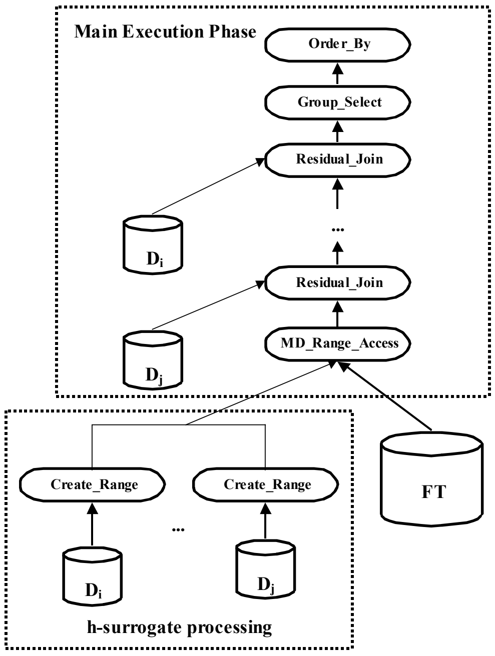

Figure 3: The abstract processing plan

The plan can be logically divided in two main processing phases: the *hierarchical surrogate key processing (HSKP) phase* which corresponds to Step 1 mentioned earlier, and the *main execution phase (MEP)* corresponding to the other two steps. Next we describe the operators appearing in the abstract processing plan of Figure 3.

Create_Range is responsible for evaluating the local predicate (**LP** in Figure 2) on each dimension table. This evaluation will result in an h-surrogate specification (set of ranges) for each dimension. All these together define one (or more, disjoint) hype-rectangle(s) in the multidimensional space of the fact table. In the next section we will present some implementation hints that allow for the efficient processing for this operation.

MD_Range_Access receives as input the h-surrogate specifications from the *Create_Range* operators and performs a set of range queries on the underlying multidimensional structure that holds the fact table data. Apart from the selection of data points that fall into the desired ranges, this operator can perform further filtering based on predicates on the measure values (**MP**) and projection (without duplicate elimination) of fact table attributes.

Residual_Join is a join on a key-foreign key equality condition among a dimension table and the tuples originating from the *MD_Range_Access* operator. This way, each incoming fact table record is joined with at most one dimension table record. The join is performed in order to enrich the fact table records with the required dimension table attributes. These attributes might be required in the `SELECT`, `GROUP BY`, `HAVING` and `ORDER BY` clauses.

Group_Select performs grouping and aggregation on the resulting tuples and evaluates any restrictions appearing in the `HAVING` clause. Finally, *Order_By* simply sorts the tuples in the required output order.

Note that not all operators in the abstract plan may be needed for the execution of a particular query. The plan represents the most complex abstract plan that might be required to answer a supported query. For example, if the result records are not required in a specific order then the final *Order_By* operator will not be applied. Also, many queries will not restrict all available dimensions nor will require feature or hierarchical attributes from all dimension tables. This means that only a restricted number of *Create_Range* and *Residual_Join* operators may be used. In the simplest possible query (`SELECT * FROM ft`) only the *MD_Range_Access* operator is needed.

3.5 Example

In this section we first describe an example schema of a simplified data warehouse. Then we present an abstract processing plan for an example query on this data warehouse.

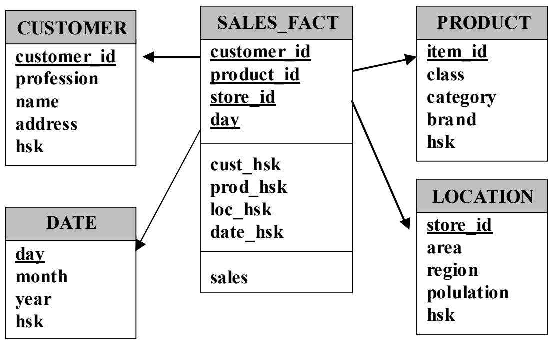

Figure 4: The schema of the data warehouse

The data warehouse stores sales transactions recorded per item, store, customer and date. It contains one fact table *SALES_FACT*, which is defined over the dimensions: *PRODUCT*, *CUSTOMER*, *DATE* and *LOCATION* with the obvious meanings. The single measure of *SALES_FACT* is *sales* representing the sales value for an item bought by a customer at a store at a specific day. The schema of the fact table is shown in Figure 4 and the dimension hierarchies are depicted in Figure 5.

The dimension *DATE* is organized in three levels: Day-Month-Year. Hence, it has three hierarchical attributes (*day, month, year*).

The dimension *CUSTOMER* is organized in only two levels: Customer-Profession. For each customer the dimension table contains an ID, a name, an address and a profession. The dimension has two hierarchical attributes (*customer_id, profession*) and two feature attributes (*name, address*). The *LOCATION* dimension is organized into three levels: Store-Area-Region. Stores are grouped into geographical areas and the areas are grouped into regions. For each area, the population is stored as feature attribute. The dimension has three hierarchical attributes (*store_id, area, region*) and one feature attribute (*population*) that is assigned to the Area level.

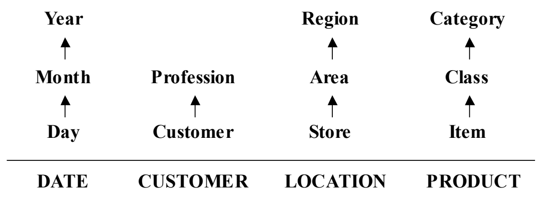

Figure 5: The dimension hierarchies of the example

Finally, the *PRODUCT* dimension is organized into three levels: Item-Class-Category. Items are grouped into product classes and those classes are grouped into categories. For example, one category could be "air condition". Also, the attribute *brand* characterizing each item is a feature attribute.

Let us now define an example query on the above schema: We want to see the sum of sales by area and month for areas with population more than 1 million, for the months of the year 1999 and for products that belong to the category "air condition". The corresponding SQL expression of this query is given next, while the abstract processing plan for this query is shown in Figure 6.

```
SELECT L.area, D.month, SUM(F.sales)
FROM SALES_FACT F, LOCATION L, DATE D,
PRODUCT P
WHERE F.day = D.day AND F.store_id =
L.store_id AND F.product_id = P.item_id AND
D.year = 1999 AND L.population>1000000 AND
P.category = "air condition"
GROUP BY L.area, D.month
```

Having described the framework for query processing of OLAP queries, we move next to discuss various optimization issues that arise.

4. Optimization Issues

In this section we will focus on particular parts of the abstract processing plan presented previously. In particular, our interest will be centered on the hierarchical surrogate key processing phase and the grouping processing step. We propose some processing hints for the former and a transformation of the abstract plan for the latter that can lead to better processing plans.

4.1 Optimization of the h-surrogate processing phase

Hierarchical surrogate keys play a dominating role in the processing of star queries over hierarchically clustered multidimensional data with hierarchies. We have already presented an abstract processing plan for our target queries. In this plan, we have seen that the very first operation that needs to be executed is the access to dimension tables and the extraction of appropriate h-surrogate ranges (*Create_Range* in Figure 3).

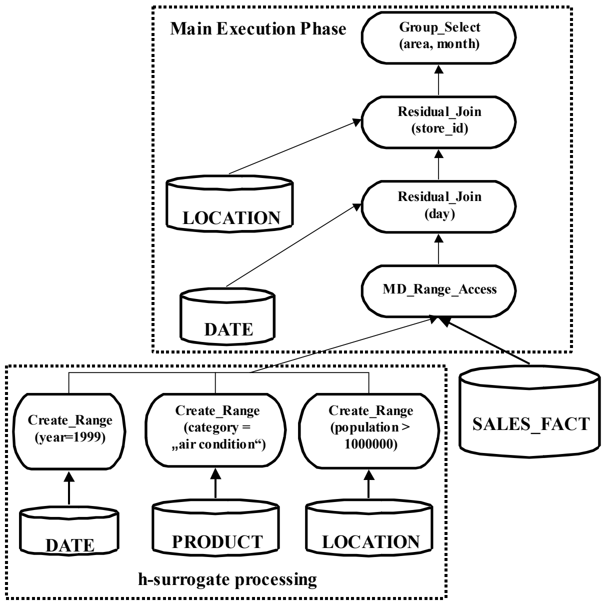

Figure 6: The abstract processing plan for the example query

For the vast majority of dimension restrictions, the *Create_Range* operator can be implemented very efficiently. If we consider hierarchical prefix path (HPP) restrictions (see section 3.3), then the first matching tuple on each dimension suffices in order to retrieve the appropriate h-surrogate value that will generate the ranges. For example, if we have the restriction *PRODUCT.category* = "air condition" AND *PRODUCT.class* = "A", then essentially what we want is all the leaves of the subtree with root "air condition"/"A"/ defined in the tree instantiating the hierarchy of dimension *PRODUCT*. Therefore, if we retrieve the h-surrogate value corresponding to the first tuple that qualifies and truncate the part from the right that corresponds to level Item, then this will be the same for all matching tuples. Next we can use this truncated h-surrogate value in order to create a range.

Moreover, if we have stored more information on the correlations between the attributes of a dimension, apart from the definition of the hierarchy, then we can benefit from the above processing scheme, even for non hierarchical prefix path restrictions. Suppose we have a hierarchy $h_m, h_{m-1},..., h_1$ on a dimension and we have a restriction of the form: h_k = c_1 AND h_p = c_2 AND … h_i = c_i, where $h_k, h_p,...,h_i$ do not form a prefix of $(h_m,...,h_1)$ and h_i is the most detailed of the referenced attributes. If we know that h_i functionally determines h_j, for all $j > i$, then we can still apply the above strategy. For example, for the restriction *DATE.month* = "AUG99", we know that the *month* attribute determines the *year* attribute and thus only the first tuple that has this value for month suffices for our processing needs. Similar observations hold for the restrictions on the feature attributes as well.

A very simple but also quite drastic optimization strategy for the processing of the *Create_Range* operator, would be the use of a composite (B-tree) index, for each dimension table D_i, defined over the attributes $h_m, h_{m-1},\ldots,h_1, hsk_i$. This index's purpose would be twofold: (a) it could be used to speedup the retrieval of D_i tuples, when a hierarchical prefix path restriction appears in a local predicate for D_i and (b) it could also be used as a table that stores the mapping between hierarchical prefix paths and h-surrogate values. The former use is the classic exploitation of an index, while the latter gives us the opportunity to use this index solely to evaluate all predicates that contain restrictions on hierarchical attributes only (and not on feature attributes), without accessing D_i, regardless of the existence, or not, of a hierarchical prefix match. Even if we do not have a match with the search-key of the index and we have to fully scan the index, this will be obviously more efficient than scanning the dimension table D_i. Naturally, smaller tuples of the index will deliver us the required h-surrogate values with much less I/O cost than if we had to read the D_i tuples.

Another issue worth mentioning is that in some cases, local predicates on dimensions can result in a number of distinct h-surrogate values not forming a set of intervals. This inevitably will result in a large number of range queries. However, very often this evaluation produces a set of h-surrogate values that belong to the same "family" in the hierarchy and thus can be merged into a single interval, reducing this way the total number of intervals created. For example, a local predicate on the LOCATION dimension with a restriction *population* > *1000000*, could result to two areas that can be expressed by two intervals. The restriction, however, may qualify a large number of hierarchy paths with a corresponding number of distinct h-surrogates. A clever h-surrogate processing phase can detect such cases and reduce the number of intervals by merging h-surrogates of the same area. This would generate two intervals instead of a large number.

4.2 Grouping on Surrogates

According to the abstract plan, the fact table tuples retrieved by the *MD_Range_Access* operator are joined with a number of dimension tables. Then, the *Group_Select* operator groups the joined tuples and computes the required aggregates. Although the join operations are on key - foreign key equalities they can be quite expensive to perform. This is due to the large number of involved tuples from both the fact and dimension tables. The large number of tuples leads also to an expensive grouping and aggregation operation. Our experience from experimental tests and from real world OLAP applications indicates that a significant part of the query processing time is spent in the joining and grouping steps.

In order to optimize these steps one can perform various transformations like the ones described in [YL95, CS94, GHQ95, LJ01] and [LMS94]. Among these, the transformation that seems to improve the plan in most of the cases is the Eager-GroupBy transformation [YL95]. This transformation pushes the grouping operator bellow one or more join operators. This way the join is performed on a much smaller number of tuples while the grouping operation is performed on smaller size tuples.

The interesting observation in our case is that we can exploit the existence of h-surrogates in the fact table tuples and perform a new kind of transformation initially described in [Elh01]. This, so called *pre-grouping* transformation, allows the grouping of fact table tuples before all join operations leading to a significant reduction of both the join and grouping effort. Furthermore, in particular cases the transformation can remove completely one or more join operations.

Let us now illustrate our transformation with an example query on the schema of section 3.5. Assume we want to have a report with the professions of all customers and the average sales value for each such profession. In our schema each customer has only one profession. The original plan would join the fact table with the *CUSTOMER* dimension table and group the result with respect to the *profession* attribute. For each group the query would report the profession and the average sales value of the group.

Using the pre-grouping transformation we can modify the plan and perform the grouping before the join. In order to do that we use the h-surrogate attribute of the fact table that corresponds to the *CUSTOMER* dimension (*cust_hsk*). In our example the structure of *cust_hsk* is: *profession/customer_id*. Using only the *profession* part of *cust_hsk* we can group the fact table tuples before joining them. The advantages of this transformation are obvious for both the grouping and join operations. Figure 7 illustrates the original and the transformed plan of our example query.

One might think that the transformed plan in Figure 7(b) need not contain at all the join with the *CUSTOMER* table. However, the join is maintained because the h-surrogates are implemented as an encoding of the path they represent. Recall that h-surrogates do not store the path value using the actual values of the hierarchical attributes. However, if for some reason the query of our example did not require in the output the actual values of the attribute *profession* then the transformed plan would skip completely the join with the *CUSTOMER* dimension table. This demonstrates that pre-grouping not only speeds-up the evaluation of the grouping and join operations but can also remove join operations from the plan.

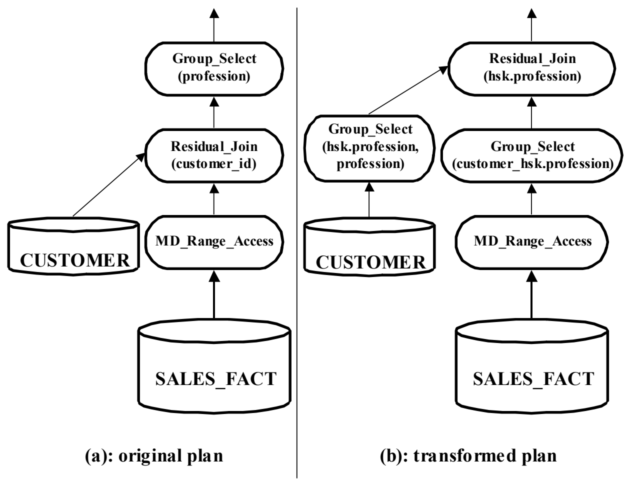
Group_Select (profession)
Residual_Join (customer_id)
CUSTOMER
MD_Range_Access
SALES_FACT
(a): original plan
Residual_Join (hsk.profession)
Group_Select (hsk.profession, profession)
Group_Select (customer_hsk.profession)
CUSTOMER
MD_Range_Access
SALES_FACT
(b): transformed plan

Figure 7: The pre-grouping transformation

We argue that the pre-grouping transformation cannot be implemented using only the Eager-GroupBy or other similar transformations. As described in detail in [Tso02], pre-grouping can be considered to be a complex transformation that combines a novel transformation, called *Surrogate-Join*, with the Eager-GroupBy and other basic algebraic transformations. The *Surrogate-Join* transformation is applicable to joins on key-foreign key equality when a limited number of attributes are projected after the join. The transformation, when applicable, modifies the join condition making it an equality predicate on a different pair of attributes while adding two grouping operations, one for each input source of the join.

In order to perform this transformation it is required that attributes in one of the input sources of the join must be (known to be) functionally related to attributes in the other source. The usage of functional dependencies and the modification of the join condition make *Surrogate-Join* essentially different from any previously defined transformation for grouping operations. The star schema and abstract plan that we adopt are particularly suitable for the application of the pre-grouping transformation since h-surrogates are functionally related to the hierarchical attributes of the dimension tables and all residual join operations are performed on key-foreign key equality conditions. In the next section we use the pre-grouping transformation as part of the general optimization algorithm applied on the Main Execution Phase.

4.3 Optimization of the Main Execution Phase

In this section we give an overview of a heuristic algorithm used to optimize the Main Execution Phase (MEP) of an abstract plan. The algorithm uses only the syntactic properties of the query and its main contribution is the application of the pre-grouping transformation. Although a cost-based optimizer could achieve further optimizations the experimental measurements (see section 5) report significant speed-ups when using the proposed heuristic algorithm.

The algorithm assumes the existence of an original abstract plan where a *Residual_Join* operation exists for each dimension appearing in the `FROM` part of the query. It then follows 4 steps:

- In the first step the redundant *Residual_Join* operators are identified and removed.
- In the second step the pre-grouping transformation is applied and a new *Group_Select* operator may be added. All affected operators are modified as needed.
- In the third step some *Residual_Join* operators may be pulled up above the original *Group_Select* operator in order to perform grouping as soon as possible.
- Finally, step 4 may eliminate the original *Group_Select* operator if this operator is redundant.

Details of the algorithm are in the Appendix.

5. Performance Evaluation

The technology introduced in this paper is fully implemented in the commercial relational DBMS TransBase HyperCube [TBHC]. This section presents preliminary measurement results that evaluate the performance of the proposed techniques.

The measurements are performed on a two processor PC Pentium III, 750 MHz, with 256 MB RAM and 30 GB IDE hard disk.

The DW schema consists of a fact table with three dimensions *CUSTOMER*, *PRODUCT* and *DATE* and 3 measures: *quantity*, *value* and *unit_price*. The data used come from a large electronic retailer in Hellas. The *CUSTOMER* dimension contains 1,4 million records, *PRODUCT* consists of 27.000 products and the *DATE* dimension covers 7 years on day granularity. 15.543.380 records are stored in the fact table, amounting to 1,5 GB.

The query workload consisted of 220 ad hoc star queries from a real-world application. We classified the queries into three groups according to their selectivity on the fact table (i.e., number of tuples retrieved from the fact table):

- [0.0-0.1]: 0% to 0.1% of fact table, i.e., 0 to about 15K records
- [0.1-1.0]: 0.1% to 1% of fact table, i.e., 15K to 160K records
- [1.0-5.0]: 1.0% to 5.0% of fact table, i.e., 160K to 780K records

The goal of the performance evaluation was to measure three alternative execution plans:

(a) the conventional star join plan (STAR),
(b) the abstract execution plan as described in section 3 (called AEP) and
(c) the enhanced version taking the pre-grouping optimizations of section 4 into account (called OPT).

FT Sel. %	[0.0-0.1]			[0.1-1.0]			[1.0-5.0]		
	STAR	AEP	OPT	STAR	AEP	OPT	STAR	AEP	OPT
MIN	0	0	0	65	2	2	274	11	6
MAX	30	6	3	290	9	6	1219	47	27
MEDIAN	1	1	1	182	8	5	477	23	13
STD-DEV	4.9	1.2	0.5	75.6	3.1	1.6	346.0	14.1	7.9

Table 1: Response time (in sec) for the three techniques for the three query classes

STAR uses secondary indexes that are created on the dimension keys of the fact table. The restrictions on the dimension tables are evaluated and the resulting dimension keys are used for index intersection on the fact table. The resulting records are joined with the dimension tables, in order to perform grouping and get the final result. This is the typical processing of star queries in commercial DBMSs (e.g., *star transformation* in Oracle [Ora01]). This processing has two major steps: the index intersection and the tuple materialization. While the index intersection has largely been optimized (e.g., with bitmap indexes [NQ97]) the materialization of results is still the bottleneck of non-clustering indexes. Consequently, we neglect the index intersection time for STAR and just measure the time for fact record materialization, residual joins and grouping. For AEP and OPT the complete processing including index access is measured, therefore favoring STAR.

Table 1 shows the response time analysis (in seconds) for the three alternative processing plans. As the three classes contain queries with different result set size and thus different response times we use the maximum, minimum, median time and the standard deviation to analyze the performance.

Our results show that the standard STAR processing is outperformed by our approaches. However, for small queries, i.e., the class [0.0-0.1], the speedup is below an order of magnitude. In general, for small result sets, the advantage of clustering over non-clustering is not that large. The picture changes drastically, when we consider larger queries (classes [0.1-1.0] and [1.0-5.0]), which are more typical for OLAP applications. The hierarchical clustering of AEP leads to an average speedup compared to STAR of 24 and with the additional optimization of pre-grouping an additional factor of about two is gained.

Note also that STAR has a very high deviation in the response times for queries within one class. This is mainly for two reasons: (a) STAR performance deteriorates very fast as the fact table selectivity is increased and (b) since the fact table is not stored clustered the number of performed I/Os may differ significantly from one query to another. On the other hand, the deviation for AEP and OPT remains low, showing a much more stable behavior.

6. Summary and Conclusions

In this paper, we have focused on the processing of the most common type of query in data warehouse and OLAP environments: the star query. For realistic database sizes a star query may take from a couple of minutes to a few hours to execute, depending on the complexity of the query and the number of tuples retrieved from the fact table. The need for fast answers to ad hoc star queries is a real-world problem for all contemporary business intelligence applications.

One of the most promising techniques for efficiently evaluating such queries is the use of fact table organizations that store data clustered according to the dimension hierarchies. A special hierarchical encoding is imposed on the data and star joins are transformed to multidimensional range queries on the underlying multidimensional structures. The conventional star query evaluation plan changes radically and new processing steps are required. To this end, we have introduced an abstract execution plan (AEP) where we describe all the necessary processing steps for the evaluation of star queries over hierarchically clustered fact tables. Furthermore, we have identified ways to minimize the processing effort entailed in the evaluation of dimension restrictions and the extraction of h-surrogate ranges. In addition, we have presented a novel early grouping transformation that dramatically reduces the overhead of both residual joining and grouping.

Our experimental evaluation has showed an average speed-up of more than 20 compared to a conventional plan and more than 40 for the optimized AEP plan, for queries that retrieved more than 0.1% of the fact table tuples. In summary, the major contribution of this paper is that it sets a processing framework for evaluating star queries over hierarchically clustered fact table organizations. The significance of our proposal is amplified by the fact that it has already been incorporated in a commercial RDBMS and our experimental evaluation results have been confirmed in real-life applications. It is important to point out that this processing framework may also apply to clustering and query processing of other hierarchically structured data, such as XML documents. We plan to further investigate this possibility in the future. In addition, our current work includes extensive experimental evaluation of the optimization techniques as well the development of further optimizations, such as alternative methods for reducing the number of h-surrogate ranges and consequently the number of range queries evaluated over the fact table.

7. Acknowledgements

This work has been partially funded by the European Union's Information Society Technologies Programme (IST) under project EDITH (IST-1999-20722).

8. Bibliography

[Bay97] R. Bayer. The universal B-Tree for multi-dimensional Indexing: General Concepts. WWCA '97. Tsukuba, Japan, LNCS, Springer Verlag, March, 1997.

[CD97a] S. Chaudhuri, U. Dayal: An Overview of Data Warehousing and OLAP Technology. SIGMOD Record 26(1): 65-74 (1997)

[CD97b] S. Chaudhuri, U. Dayal: Data Warehousing and OLAP for Decision Support (Tutorial). SIGMOD Conference 1997: 507-508

[CI98] C. Y. Chan, Y. E. Ioannidis: Bitmap Index Design and Evaluation. SIGMOD Conference 1998: 355-366

[CS94] S. Chaudhuri, K. Shim: Including Group-By in Query Optimization. VLDB 1994: 354-366

[DRNS98] P. Deshpande, K. Ramasamy, A. Shukla, J. F. Naughton: Caching Multidimensional Queries Using Chunks. SIGMOD Conference 1998: 259-270

[Elh01] K. Elhardt: MHC Processing with TransBase Operator Trees. Technical Report, TransAction Software GmbH 2001.

[GBLP96] J. Gray, A. Bosworth, A. Layman, H. Pirahesh: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. ICDE 1996: 152-159

[GG97] V. Gaede and O. Günther. Multidimensional Access Methods. ACM Computing Surveys 30(2), 1997.

[GHQ95] A. Gupta, V. Harinarayan, D. Quass: Aggregate-Query Processing in Data Warehousing Environments. VLDB Conference 1995: 358-369

[GLS01] C. A. Galindo-Legaria, M. Joshi: Orthogonal Optimization of Subqueries and Aggregation. SIGMOD Conference 2001

[GM95] A. Gupta, I. S. Mumick: Maintenance of Materialized Views: Problems, Techniques, and Applications. Data Engineering Bulletin 18(2): 3-18 (1995)

[Kim96] R. Kimball. The Data Warehouse Toolkit. John Wiley & Sons, New York. 1996.

[KR98] Y. Kotidis, N. Roussopoulos: An Alternative Storage Organization for ROLAP Aggregate Views Based on Cubetrees. SIGMOD Conference 1998: 249-258

[KS01] N. Karayannidis, and T. Sellis, "SISYPHUS: A Chunk-Based Storage Manager for OLAP Cubes", Proceedings of the 3rd International Workshop on Design and Management of Data Warehouses (DMDW'2001), Interlaken, Switzerland, June 2001

[LMS94] A. Y. Levy, I. S. Mumick, Y. Sagiv: Query Optimization by Predicate Move-Around. VLDB Conference 1994: 96-107

[MRB99] V. Markl, F. Ramsak, R. Bayern: Improving OLAP Performance by Multidimensional Hierarchical Clustering. Proc. of the Intl. Database Engineering and Applications Symposium, pp. 165-177, 1999.

[NG95] P. E. O'Neil, G. Graefe: Multi-Table Joins Through Bitmapped Join Indices. SIGMOD Record 24(3): 8-11 (1995)

[NHS84] J. Nievergelt, H. Hinterberger, K. C. Sevcik: The Grid File: An Adaptable, Symmetric Multikey File Structure. TODS 9(1): 38-71 (1984)

[NQ97] P. E. O'Neil, D. Quass: Improved Query Performance with Variant Indexes. SIGMOD Conference 1997: 38-49

[Ora01] Oracle 8i Documentation, 2001.

[Rou98] N. Roussopoulos: Materialized Views and Data Warehouses. SIGMOD Record 27(1): 21-26 (1998)

[Sam90] H. Samet. The Design and Analysis of Spatial Data Structures. Addison Wesley, 1990

[Sar97] S. Sarawagi: Indexing OLAP Data. Data Engineering Bulletin 20(1): 36-43 (1997)

[SDJL96] D. Srivastava, S. Dar, H. V. Jagadish, A. Y. Levy: Answering Queries with Aggregation Using Views. VLDB Conference 1996: 318-329

[TBHC] The TransBase HyperCube relational database system, available at: http://www.transaction.de/

[Tso02] A. Tsois: Decomposing pre-grouping: the Surrogate-Join transformation. Technical Report, KDBS Lab, NTUA, TR-2002-01.

[TT01] D. Theodoratos, A. Tsois: Heuristic Optimization of OLAP Queries in Multidimensionally Hierarchically Clustered Databases. DOLAP 2001.

[WB98] M.C. Wu, A. P. Buchmann: Encoded Bitmap Indexing for Data Warehouses. ICDE 1998: 220-230

[WOS01] K.Wu, E. J. Otoo, A. Shoshani: A Performance Comparison of bitmap indexes. CIKM 2001: 559-561

[Wu99] Ming-Chuan Wu: Query Optimization for Selections Using Bitmaps. SIGMOD Conference 1999: 227-238

[YL94] W. P. Yan, P-Å. Larson: Performing Group-By before Join. ICDE 1994: 89-100

[YL95] W. P. Yan, P.-Å. Larson: Eager Aggregation and Lazy Aggregation. VLDB Conference 1995

[ZSL98] C. Zou, B. Salzberg, and R. Ladin: Back to the Future: Dynamic Hierarchical Clustering. Proc. Of ICDE, 1998, pp. 578-587.

APPENDIX: Heuristic Algorithm

In order to present the details of the algorithm, we will use the following definitions:

Hlevel: The *Hlevel* of a hierarchy attribute h_k in a dimension table is defined to be k. The Hlevel of a feature attribute f of a dimension is k, if f is known to be functionally dependent on h_k, 1 otherwise.
Grouping Order: Let $g_1, ..., g_k$ be the set of grouping attributes of the `GROUP BY` clause which belong to dimension D_i. For dimension D_i, the *grouping order* $GO(D_i)$ is defined to be the minimum $Hlevel(g_i)$ for $1 \le i \le k$.
Aggregation Order: Let $a_1, ..., a_k$ be the set of aggregation attributes in the `SELECT` (**Aggr**) or `HAVING` (**HP**) clause that belong to dimension D_i. The *aggregation order* $AO(D_i)$ for D_i is defined to be the minimum $Hlevel(a_i)$ for $1 \le i \le k$.
Dimension Order: The dimension order $DO(D_i)$ for D_i is defined to be the minimum among $AO(D_i)$ and $GO(D_i)$. If $AO(D_i)$ and $GO(D_i)$ are not defined then $DO(D_i)=\infty$.

Given the above definitions we proceed to give the details of the heuristic optimization algorithm. The algorithm assumes that each hierarchical attribute h_k functionally determines all higher level hierarchical attributes of the dimension: h_{k+1}, h_{k+2}, …

Algorithm HO

1) Eliminate redundant joins:
If the attributes of a dimension table D_i are used neither in the `SELECT` (**SGA**, **Aggr**) part of the query nor in the `HAVING` (**HP**) or `ORDER BY` (**OL**) part, and no feature attributes of D_i appear in the list of grouping attributes (**GA$_f$**) then the *Residual_Join* operator for D_i is removed from the plan.

2) Introduce the new *Group_Select* operator:
Compute the total order of the query: TO=max($DO(D_i)$) where D_i in **D**. If TO>1, add a new *Group_Select* operator (*nGS*) just above the *MD_Range_Access* operator. The details of adding this new operator are as follows:

2a) The new operator *nGS* will group on the prefix part of h-surrogates (hsk_i) truncated bellow the $DO(D_i)$ level. For each aggregation function of the original *Group_Select* operator (*oGS*) that operates on a measure attribute X_k=Aggr$_k$(ft.m_k), the *nGS* operator contains the aggregation nX_k=Aggr$_k$(ft.m_k). Furthermore, if $AO(D_i)$ is defined for some dimension, then an additional aggregation is added to the *nGS*; the role of this aggregation is to count the number of tuples in each group: CNT=COUNT(*). Finally, *nGS* contains an aggregation Rd_i=MIN(d_i) for each dimension D_i that participates in a *Residual_Join* operator.

2b) For each aggregation attribute Rd_i of *nGS* we modify the join condition of the *Residual_Join* operator for the corresponding dimension D_i. The condition becomes: Rd_i=$D_i.h_1$.

2c) Modify the list of grouping attributes of the original *Group_Select* operator in the following way: each grouping attribute $D_i.h_k$ in **GA$_h$** is replaced by the h-surrogate attribute hsk_i truncated bellow level k.

2d) Modify the aggregation terms of *oGS* so that each term *oldTerm* is replaced by a term *newTerm* according to the following table:

oldTerm	***newTerm***
X_k=Aggr$_k$(ft.m_k)	X_k=Aggr$_k$(nX_k)
COUNT(ft.m_k)	SUM(nX_k)
SUM(D$_i$.a_k)	SUM(CNT*D$_i$.a_k)
COUNT(*)	SUM(CNT)
COUNT(D$_i$.h_k)	SUM(CNT)
COUNT(D$_i$.f_k)	SUM(CNT⊗D$_i$.f_k)

In the above table the expression function CNT⊗$D_i.f_k$ returns 0 when f_k is NULL and CNT otherwise. Also, $D_i.a_k$ is any attribute of D_i.

3) Pull up *Residual_Join* operators:
If a dimension D_i has $GO(D_i)>AO(D_i)$ or there is a feature attribute of D_i in the list of grouping attributes (**GA$_f$**) or an attribute of D_i is used in the `HAVING` predicate (**HP**), then the *Residual_Join* operator for this dimension remains bellow the original *Group_Select* (*oGS*) operator. All other *Residual_Join* operators are pulled up, after *oGS*.

4) Eliminate the original *Group_Select* operator:
If *nGS* has been added to the plan and there is no *Residual_Join* operator after *nGS* and before the original *Group_Select* operator (*oGS*), then *oGS* is not needed and is removed. The selection predicate (**HP**) of *oGS* is moved into *nGS* and the terms nX_k=Aggr$_k$(ft.m_k) are renamed to X_k=Aggr$_k$(ft.m_k).