Combining Hierarchy Encoding and Pre-Grouping: Intelligent Grouping in Star Join Processing

Roland Pieringer¹, Klaus Elhardt¹, Frank Ramsak², Volker Markl³, Robert Fenk², Rudolf Bayer⁴, Nikos Karayannidis⁵, Aris Tsois⁵, Timos Sellis⁵

¹Transaction Software GmbH, Thomas-Dehler-Str. 18, D-81737 München, Germany
{pieringer,elhardt}@transaction.de

²Bayerisches Forschungszentrum für Wissensbasierte Systeme, Boltzmannstr. 3, D-85748 Garching, Germany
{robert.fenk,frank.ramsak}@forwiss.de

³IBM Almaden Research Center, K55/B1, 650 Harry Road, San Jose, CA 95120-6099, marklv@us.ibm.com

⁵Institute of Communication and Computer Systems and Department of Electrical and Computer Engineering, National Technical University of Athens, Zographou 15773 Athens, Hellas
{nikos,atsois,timos}@dabl.ece.ntua.gr

¹Institut für Informatik, TU-München, Boltzmannstr. 3, D-85748 Garching, Germany
bayer@in.tum.de

Abstract

Efficient star query processing is crucial for a performant data warehouse (DW) implementation and much work is available on physical optimization (e.g., indexing and schema design) and logical optimization (e.g., pre-aggregated materialized views with query rewriting). One important step in the query processing phase is, however, still a bottleneck: the residual join of results from the fact table with the dimension tables in combination with grouping and aggregation. This phase typically consumes between 50% and 80% of the overall processing time. In typical DW scenarios pre-grouping methods only have a limited effect as the grouping is usually specified on the hierarchy levels of the dimension tables and not on the fact table itself. In this paper, we suggest a combination of hierarchical clustering and pre-grouping as we have implemented in the relational DBMS Transbase. Exploiting hierarchy semantics for the pre-grouping of fact table result tuples is several times faster than conventional query processing. The reason for this is that hierarchical pre-grouping reduces the number of join operations significantly. With this method even queries covering a large part of the fact table can be executed within a time span acceptable for interactive query processing.

1 Introduction

Optimizing star query processing has been an important topic in recent years. Star query processing can be roughly divided into three major steps: (1) evaluation of dimension and measure predicates, (2) fetching result tuples from the fact table and (3) residual joins, grouping and aggregation, sorting.

Lots of work is available on the optimization of the first two steps, especially in the field of index structures for efficient fact table access ([1]). Bitmap indexes and multidimensional indexes (e.g., the UB-Tree) are popular approaches. Also physical clustering [12], [6] has been investigated with great effort (e.g., hierarchical clustering).

Our goal is to speed up the third phase, i.e., residual join and grouping of the star query processing. The third execution phase becomes the major part, consuming between 50% and 80% of the total execution time after the first and second phases have been optimized. Minimizing the effort for this phase can speed up the overall query execution time significantly.

While new join techniques for these so-called star joins have been proposed (e.g., hash-based, bitmap join indexes, etc.) the potentially large number of fact table tuples that have to be joined with the dimension tables is the main problem at this point. Pre-grouping techniques have been proposed to reduce the number of join tuples
by introducing a grouping and aggregation phase before the first residual joins. These techniques rely on the ability to exploit functional dependencies among attributes of a table. In DW applications, grouping and aggregation is usually specified on the hierarchies of the dimensions, limiting the effect of these pre-grouping methods.

In this paper, we show how the combination of hierarchical clustering of the fact table and pre-grouping significantly reduces the cost for the third execution phase. Hierarchical clustering is based on the idea that the hierarchy semantics of one dimension are encoded into the dimension key used in the fact table. These hierarchical surrogates (h-surrogates) are a compact representation of the hierarchy path of a dimension member making it possible to use hierarchy semantics not only in the dimension tables but also directly on the fact table without requiring costly residual joins. While the h-surrogates were originally designed to improve the clustering and indexing of the fact table, they can also be used efficiently in pre-grouping. With the hierarchy semantics encapsulated in the h-surrogates the pre-grouping algorithms can exploit more functional dependencies and thus achieve a much higher reduction of the number of join tuples.

A simple example illustrates the effects of the hierarchical pre-grouping method: Assume we have a DW with a time dimension (besides other dimensions) categorized by year – month – day and a well populated fact table. Let a query restricting the result to year 2001 (besides other restrictions) qualify 100.000 fact table records. If the result has to be grouped w.r.t. month, we have to join 100.000 records with the time dimension table. When applying pre-grouping, the number of join operations is reduced only marginally (depending on the grouping on other dimensions). When applying hierarchical pre-grouping, the number of join operations is reduced by a factor of 30, because all days of one month are aggregated to the month by using the hierarchical encoding information in the fact table. This reduction may lead to an overall speedup by a factor from 2 up to 5.

Our measurements with a five dimensional real world DW show an average reduction of the join cardinality by more than a factor of 100. This leads to an overall speedup in time of a factor of 4 to 7 for the third processing phase.

The rest of the paper is organized as follows: In Section 2 we explain some basic preliminaries like the DW schema and the query template used in this paper. Section 3 describes the three grouping methods: no pre-grouping, early grouping of [15] and [3], and hierarchical pre-grouping. The integration of pre-grouping into a DBMS is discussed in Section 4. Section 5 contains some advanced topics, especially advanced aggregation handling. In Section 6 we present some measurements on a real world scenario to compare hierarchical pre-grouping with standard pre-grouping. Section 7 shows the related work and Section 8 concludes the paper.

2 Preliminaries

2.1 Schema

For our discussions we use a conventional star schema [2] with a fact table consisting of dimension (qualitative) and measure (quantitative) attributes [8]. For the dimensions typically one or more hierarchical classifications based on the dimension attributes (often referred to as features) exist. The primary key of the dimension represents the most detailed level of the dimension hierarchies.

As mentioned above, we add hierarchical surrogates to the fact table. For a detailed discussion of h-surrogates please refer to [12], [6], or [7]. The basic concept is to use an encoding for the hierarchy paths of the hierarchy levels \(h, h_1, \ldots, h_1\) (\(h\) being the most aggregated level and \(h_1\) the most detailed one). We call the encoding a hierarchy surrogate key (hsk). It is unique for each dimension tuple. In this paper, we use the notation \(h_1/\ldots/h_1\) for h-surrogates. For example, an h-surrogate for a geographic hierarchy is country/region-city/store, e.g., Germany/South/ Munich/BMW. Please consider that in the implementation a space efficient encoding is used instead of the “plain text” attributes.

As running example throughout this paper we use the schema depicted in Figure 1. This 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 measures of SALES_FACT are price, quantity, and sales representing the values for an item bought by a customer at a store at a specific day. The schema of the fact and dimension tables is shown in Figure 1. The dimension hierarchies are depicted in Figure 2. The meanings of the hierarchies are obvious.

Figure 1. Sample schema

The CUSTOMER dimension has two hierarchical attributes (person_id, profession) and two feature attributes (name, address). The dimension LOCATION has four hierarchical attributes (store_id, city, region, country) and

<table>
<thead>
<tr>
<th>CUSTOMER</th>
<th>SALES_FACT</th>
<th>PRODUCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>person_id, profession</td>
<td>person_id, product_id, store_id, day, month, year</td>
<td>item_id, group, category, brand</td>
</tr>
<tr>
<td>name, address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>store_id, city, region, country, polulation</td>
<td>location</td>
<td></td>
</tr>
<tr>
<td>Ask</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DATE</th>
<th>LOCATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>day, month, year</td>
<td>store_id, city, region, country</td>
</tr>
<tr>
<td>Ask</td>
<td>Ask</td>
</tr>
</tbody>
</table>
one feature attribute (*population*) that is assigned to the city level.

![Figure 2. The dimension hierarchies of the example](image)

Finally, the *PRODUCT* dimension is organized into three levels: item – group – category. The attribute *brand* characterizing each item is a feature attribute. Note that all dimension tables contain the attribute *hsk*, i.e., the h-surrogate for the dimension hierarchy. These h-surrogates are referenced by *hsk_cust*, *hsk_prod*, *hsk_loc*, and *hsk_date* in the fact table.

### 2.2 Specifying functional dependencies

Each hierarchical attribute is directly functionally dependent on the lower level attribute, i.e., \( h_i \rightarrow h_{i+1} \), and \( h_i \) is the child level of \( h_{i+1} \) in the hierarchy tree. Consequently, each hierarchy attribute is functionally dependent on the leaf hierarchy attribute \( h_1 \) (most detailed level).

The knowledge of functional dependencies is crucial for a good performance for grouping on dimension feature attributes that are not components of h-surrogates. Thus meta data must be available and stored in the data dictionary.

An alternative to exploiting information via meta data is to use a snowflake schema with correct and complete foreign key constraints.

The following tables are an example for such a snowflake schema:

| Location_leaf (store_id, city_id, region_id, country_id) |
| Location_city(city_id, city_name, population) |

and the foreign key reference:

| Location_leaf.city_id REFERENCES Location_city.city_id |

Dimension tables for the remaining hierarchy levels can be normalized correspondingly. In such a schema, the optimizer knows that *city_name* and *population* are functionally dependent on the hierarchy level *city_id* and can use this knowledge, in order to optimize grouping on *city_name* and *population*.

A snowflake schema usually provides more information about hierarchical (and functional) dependencies and therefore enables more efficient query processing. In a well designed snowflake schema, feature attributes can be assigned to hierarchy levels.

Most DBMS recommend to use a star schema instead of a snowflake schema, in order to reduce join complexity. However, the combination of h-surrogates and correctly designed snowflake schema will result in better performance than a comparable star schema, if the DBMS is able to make use of this knowledge.

### 2.3 Query template

In **Figure 3** we depict a SQL query template for ad hoc star-queries. The notation \{X\} represents a set of X objects. The template defines typical star queries and uses abstract terms that act as placeholders. Note that the queries that conform to this template generally have a structure that is a subset of the template and they instantiate all abstract terms.

```
SELECT \{D.h\}, \{D.f\}, \{FT.m\},
{AGGR(FT.m) AS AM},
{AGGR(D.h) AS AH}, AGGR(D.f) AS AF}
FROM FT, D1,..., Dn
WHERE FT.d1 = D1.h1 AND
FT.d2 = D2.h1 AND
...
FT.dn = Dn.h1 AND
LOCPRED(D1) AND
LOCPRED(D2) AND
...
LOCPRED(Dn) AND
MPRED({FT.m})
GROUP BY \{D.h\}, \{D.f\}, \{FT.m\}
HAVING HPRED({AM}, {AH}, {AF})
ORDER BY <ordering fields>
```

**Figure 3. Star query template**

Our template is 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, we see that it includes a star-join. Apart from the star-join, there is a GROUP BY and a HAVING clause. In general, any attribute (hierarchical, feature, or measure) can appear in a GROUP BY clause. However, most queries group the result by a number of hierarchical and/or feature attributes. Finally, there is an ORDER BY to control the order of the presented results.

\( \{D.h\} \) (\( \{D.f\} \)) stands for hierarchical (feature) attributes of any dimension. The same hierarchy (feature) attributes must occur also in the GROUP BY clause.
AGGR(x) stands for a standard SQL aggregation function (MIN, MAX, SUM, COUNT, AVG). Note that not only aggregates over single dimension or fact table attributes are allowed, but also the combination of those. Sometimes even more complex expressions are necessary. For illustration purpose we first restrict the queries to aggregation types as specified in the template, but we will extend the description later to complex aggregation problems.

LOCPRED(D_i) is a local predicate on a dimension table D_i. 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 triggers the filtering on each dimension table.

Finally, MPRED({FT.m}) is a predicate that contains any constraints on measures of the fact table, e.g., to ask for sales that exceed a certain value threshold.

3 Pre-Grouping on h-surrogates

In this section, we first introduce an abstract execution plan and discuss the basic principles of early grouping and hierarchical pre-grouping.

3.1 Standard method: join before grouping

Figure 4: Standard abstract execution plan

Figure 4 shows the abstract execution plan (AEP) for star query processing as it is described in [7]. We use the operators Fact Table Access and Predicate Evaluation to abstract from optimizations based on multidimensional clustering indexes on the fact table (see Figure 4).

The tuples that are qualified by the restrictions of the query (Predicate Evaluation operator) are fetched from the fact table and are joined with all necessary dimension tables (Residual Join operator). We do not distinguish between two-way or multi-way joins. This is a physical property of the operator tree of the DBMS. After the residual join the tuples are grouped (Group-Select operator), filtered again (Having operator) and sorted (OrderBy operator). We call this method NOPREGROUP.

3.2 Early grouping

The early grouping technique as proposed in [15] and [3] uses knowledge about functional dependencies of the join tables. Consider a query restricting a table A and grouping on attribute g of table B: B.g. The tables A and B are equi-joined by the attributes a of A and b of B: A.a = B.b, where b is primary key of B. We can group on A.a before joining with B, because A.a functionally determines B.g via the join condition and the key properties (see Section 4.1).

The resulting groups of the early grouping step are a superset of the final groups, because B.g can have the same value for distinct B.b. An additional final grouping step is necessary.

We will refer to the early grouping method by the term EARLYGROUP.

The abstract execution plan is the same as for hierarchical pre-grouping and is shown in Figure 5. The dimension tables D_{e1}, ..., D_{ei} are joined before post-grouping (e = early), D_{d1}, ..., D_{dn} are joined after post-grouping (d = delayed). The second residual join (delayed residual join) is an optimization to delay the residual join for the dimension tables D_{d1}, ..., D_{dn}. This delay is possible, if the early grouping is already exact (see also Section 4.2), i.e., if the h-surrogate prefixes represent the same groups as the grouping attributes of the dimension. For example, for GROUP BY month, the h-surrogate prefixes year/month (e.g., t_1=(2000, June), t_2=(2001, June)) would result in finer groups (g_1 = (2000, June), g_2 = (2001, June)) than the final groups (g = (June)) and thus would require early residual joins, in order to merge groups.
3.3 Hierarchical pre-grouping

Hierarchical pre-grouping (PREGROUP) is an extension of early grouping. We use h-surrogate prefixes instead of the hierarchical grouping attributes \(D.h_k\) as specified in the GROUP BY clause. We assume that the DBMS has information about hierarchical relationships of the dimension attributes, e.g., via meta data in the data dictionary.

Instead of applying pre-grouping on the user defined join attribute which has the finest granularity of the hierarchy, we group on the hierarchy level \(h_k\) as specified in the GROUP BY clause. Note that \(h_k\) is not yet available in this pre-grouping step. We use the h-surrogate corresponding to \(D.h_k\). The h-surrogate prefix \(h_t/h_{t-1}/…/h_1\) reduces the number of resulting groups dramatically (see example in Section 4.1). Thus pre-grouping takes place on the fact table result tuples.

Usually final grouping is necessary to merge the groups of the pre-grouping phase (see Section 4.2). The groups of the pre-grouping operation are joined with the dimension tables, in order to get the values for the (user defined) grouping attributes.

Figure 5 shows the optimized AEP with the pre-grouping optimization.

4 Integration of pre-grouping into the DBMS

4.1 Definitions and formal description of pre-grouping

Grouping is a standard SQL operator (GROUP BY) partitioning a set of tuples into disjoint tuple sets \(S_k\), then aggregating over each set and finally constructing one result tuple for each set.

Definition 1 (Grouping Attribute, Grouping Value):

The grouping attributes are the attributes that are specified in the GROUP BY operator. All tuples with the same grouping values, i.e., the values of the grouping attributes, contribute to one grouping result tuple. □

Definition 2 (Aggregation Attribute):

The aggregation attributes are the attributes that occur in the SELECT or HAVING clause and are not grouping attributes. □

The following two definitions are from [15] with modified notation.

Definition 3 (Row Equivalence):

Consider a table \(R(…, K, …)\), where \(K = \{k_1, k_2, …, k_n\}\) is a set of attributes. Two rows \(t, t' \in R\) are equivalent, w.r.t. \(K\), if: \(\wedge_{i=1, \ldots, n} (t.k_i = t'.k_i)\), which we also write as \(t.k_i = t'.k_i\). □

Definition 4 (Functional Dependency, Functional Determination):

Consider a table \(R(K, a, …)\), where \(K = \{k_1, k_2, …, k_n\}\) is a set of attributes and \(a\) is a single attribute. \(K\) functionally determines \(a\), denoted by \(K \rightarrow a\), if the following condition holds:

\[\forall t, t' \in R, \{(t.K = t'.K) \Rightarrow (t.a = t'.a)\}.\]

We also say, \(a\) is functionally dependent on \(K\). □

The key attributes (and candidate key attributes) always functionally determine all remaining attributes of \(R\). There also can be a chain of functional dependencies: \(a_i \rightarrow a_j \rightarrow \ldots \rightarrow a_k\). In this case, \(a_k\) is also functionally dependent on \(a_i\): \(a_i \rightarrow a_k\). For two different values of \(a_i\) there can be the same value for \(a_k\), if \(a_j \rightarrow a_k\).

h-surrogates \(h_{sk}\) are built from the combination of the corresponding hierarchy levels, \(h_{sk} = h_t/h_{t-1}/…/h_1\), where \(h_t\) is the most aggregated (top level) and \(h_1\) the most detailed level (leaf level) of the hierarchy. Recall that due to
the hierarchical relationship of the hierarchy levels \( h_1, h_2, \ldots, h_t \), the following hierarchical (functional) dependency holds: \( h_1 \rightarrow h_2 \rightarrow \ldots \rightarrow h_t \).

In SQL-92 we are not able to express functional relationships \( h_1 \rightarrow h_2 \rightarrow \ldots \rightarrow h_t \) within one table. Thus, additional meta data is necessary for this information. For example, in a time hierarchy we usually have the functional dependency chain: \( \text{Day} \rightarrow \text{Month} \rightarrow \text{Year} \) (e.g., \( "20020630" \rightarrow "June2002" \rightarrow "2002" \)).

For pre-grouping we need the definition of h-surrogate prefixes.

**Definition 5 (h-surrogate prefix, hsk prefix):**

An h-surrogate prefix (hsk prefix) consists of a subset of the h-surrogate components: \( hsk(k) = h_i/h_{i+1}/\ldots/h_k \), \( 1 \leq k \leq t \), where \( h_t \) is the most aggregated level. An h-surrogate prefix specifies a sub-tree of the hierarchy.

**Definition 6 (Hlevel):**

The \( Hlevel \) of a hierarchy attribute \( h_i \) 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. \( h_i \) is the most aggregated hierarchical attribute that \( f \) is functionally dependent on.

**Definition 7 (Grouping Order, GO):**

Let \( g_1, \ldots, g_k \) be the set of grouping attributes of the \textit{GROUP BY} clause that 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 \( 1 \leq i \leq k \).

**Definition 8 (Aggregation Order, AO):**

Let \( a_1, \ldots, a_k \) be the set of aggregation attributes in the \textit{SELECT} or \textit{HAVING} 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 \( 1 \leq i \leq k \).

**Definition 9 (Dimension Order, DO):**

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 \( DO(D_i) = \infty \).

4.2 Exact dimension grouping

Grouping is a property of the dimension, i.e., it depends on the grouping attributes of a dimension \( D_i \).

We define an exactness criterium for grouping. Informally, grouping w.r.t. a dimension \( D_i \) is exact, if a hierarchy prefix from top level \( h_t \) to level \( h_k \) occurs in the grouping clause and no other dimension attributes of \( D_i \) are grouping attributes. For example, for \textit{GROUP BY month}, and an h-surrogate prefix \textit{year/month}, grouping is not exact, because the h-surrogate prefix differs from the grouping attributes of the dimension (\( hsk_1 = 2000/June, hsk_2 = 2001/June \)).

**Definition 10 (exact grouping):**

Let \( G = \{g_1, g_2, \ldots, g_k \} \) be a set of grouping attributes of the \textit{GROUP BY} clause for dimension \( D_i \). We say that \( G \) is exact for \( D_i \) (exact grouping), if all dimension attributes of \( D_i \) occurring in \( G \) form a hierarchical prefix \( h_0, h_1, \ldots, h_t \), where \( h_t \) is the most aggregated level of the hierarchy.

For grouping attributes of dimension \( D_i \) that fulfill the exact grouping criterium, i.e., \( \text{exactGrouping}(D_i) = \text{TRUE} \), we can delay the residual join with \( D_i \) after the post-grouping phase. If none of the hierarchy attributes occur in the \textit{SELECT} or \textit{HAVING} clause, we even can omit the residual join.

If we have exact grouping for all dimensions of the grouping attributes, the post-grouping step can be omitted.

For example, grouping on \textit{country and region} is an exact grouping, to only group on \textit{region} is not an exact grouping. The same holds for grouping on \textit{population}.

4.3 Algorithm for operator tree generation

The integration of hierarchical surrogates into a DBMS requires extensions of

- the compiler (DDL),
- query processor (computation and maintenance of h-surrogates) and
- optimizer.

In this section we present an algorithm that builds an optimized AEP (OAEP) with pre-grouping.

The OAEP (Algorithm 1) contains the operators \textit{Fact Table Access}, \textit{ResidualJoin}, \textit{Having}, and \textit{OrderBy} (as in the standard AEP) and \textit{Pre Group} and \textit{Post Group} as new operators (see Figure 5).

**Algorithm 1 (generating the plan):**

\[
\text{OAEP} = \text{Fact Table Access (LOC<PRED,(D_i), \ldots, LOC<PRED,(D_k), MPRED(F.m_k), \ldots, MPRED(F.m_k))}
\]

\[
\text{GD} = \text{groupingDimensions()}
\]

\[
\text{NormalAgg} = \{ F.m_k \mid F.m_k \in \text{AGG_SELECT} \}
\]

\[
\text{SpecialAgg} = \{ D_i.h_k \mid D_i.h_k \in \text{AGG_SELECT} \} \cup \{ D_i.f_k \mid D_i.f_k \in \text{AGG_SELECT} \}
\]

\[
\text{OAEP += PreGroup (\{(GD_i, GO(GD_i))\}, \{F.m_k\}, normalAgg)}
\]

\[
\text{FOR ALL \{GD_i \in GD \} \sim \text{exactGroup(GD_i)} \land \text{extended(GD_i)}
\]

\[
\text{OAEP += ResJoin(GD_i)}
\]

\[
\text{hsk} = \text{HSK(\{GD_i \in GD \} \sim \text{exactGroup(GD_i)} \land \sim \text{extended(GD_i)} \}}
\]

\[
\text{dattr = DimAtt(\{GD_i \in GD \} \sim \text{exactGroup(GD_i)}}
\]
The generation of the execution plan is done by the concatenation of operators (built by the corresponding function calls). The first operator is Fact Table Access specified by the local predicates of the dimension tables and the predicates on the fact table.

The function groupingDimensions specifies the dimensions that are needed for further processing, i.e., the dimensions necessary for the SELECT, GROUP BY and HAVING clauses. The dimensions are stored in $GD = \{GD_1, ..., GD_j\}$, if $k$ dimensions are needed.

NormalAgg contains the fact table measure attributes, SpecialAgg contains the dimension attributes that are aggregated due to the SELECT and HAVING clause specification. The aggregation attributes are qualified by AGG_SELECT. The measure attributes are aggregated in a conventional way, whereas the dimension attributes are handled differently (see Section 5.1). The PreGroup operator contains all dimensions of $GD$. For each dimension, grouping is done on h-surrogate prefixes with grouping order GO depending on the grouping attributes (see Section 4.1). The measure attributes of the grouping clause are also used as grouping attributes.

The ResidualJoin operators are appended after the pre-grouping phase. For each dimension of $GD$ that is not an exact dimension w.r.t. the exactGroup property of section 4.2 a residual join is appended. In addition to these dimensions we add the residual joins for dimensions that are extended dimensions due to the grouping extension. An extended dimension is a dimension that contains aggregation attributes but does not have grouping attributes (see Section 5.1). Such a dimension is considered also for pre-grouping.

The PostGroup operator uses h-surrogates, (joined) dimension attributes and measure attributes of the fact table (if occurring in the GROUP BY clause) as grouping attributes. The h-surrogate attributes ($hsk$) are used for the dimensions which are “exact” and not extended. For not exact groups the real attribute values (dattr) are used as grouping attributes in order to get the final granularity of the groups. The functions HSK resp. DimAtt return h-surrogate resp. dimension attributes. Aggregation is applied on all attributes to aggregate. The special aggregate semantic is used for dimension attributes as described in Section 4.4.

All dimensions that are not yet joined and are needed for further processing are joined via ResidualJoin operators. These dimensions are “exact” and not extended and occur in the select clause.

Finally the Having and OrderBy operators are appended.

4.4 Aggregation

For pre- and post-grouping, the aggregation functions MIN, MAX and SUM are implemented straightforward. COUNT (and AVG) has to be modified for the post-grouping step:

The aggregation function values for COUNT of the pre-grouping phase have to be added for all merged groups (AVG = SUM/COUNT and therefore is affected by the new COUNT computation). For example, consider the following SQL statement that aggregates a measure FACT.a and groups w.r.t. hierarchy level $h_k$ of dimension $D$.

```sql
SELECT COUNT(FACT.a) FROM FACT, D
WHERE ...
GROUP BY D.hk
```

This statement can be transformed to the following SQL statement, in order to show the aggregation computation:

```sql
SELECT SUM(cnt) FROM
(SELECT COUNT(FACT.a) AS cnt,
 D.hk AS H FROM FACT, D
 WHERE ...
 GROUP BY FACT.hsk(k))
 GROUP BY H
```

The pre-grouping is done on the h-surrogate prefix for dimension $D$ on level $h_k$ specified informally by $hsk(k)$. After pre-grouping, we sum the counters and group w.r.t. the grouping attribute $h_k$.

More sophisticated aggregation semantics is necessary for aggregation on dimension attributes (Section 5.1) and for complex expressions in aggregation functions (Section 5.2).

4.5 When pre-grouping cannot be applied

Some aggregation functions cannot be handled in the current Transbase implementation by pre-grouping. Such functions are expressions containing a combination of fact table and dimension table attributes like $\text{MIN}(F.m_j \times D.h_k)$. For fact table processing it is not clear whether $\text{MIN}(F.m_j)$ or $\text{MAX}(F.m_j)$ has to be used for the aggregate, because $D.h_k$ can have opposite signs compared to $\text{MIN}(F.m_j)$. Then $\text{MIN}(F.m_j) \times \text{MIN}(D.h_k) > \text{MAX}(F.m_j) \times \text{MIN}(D.h_k)$.

---

1 We require that reference constraints exist for the foreign key relationship of the dimension attributes in the fact table to the most detailed level of the corresponding dimension tables. Otherwise the residual join would further restrict the result set, if groups exist that contain grouping values not existing in the corresponding dimension table.
For the general case, such expressions can be handled only with huge effort and are not easy to implement (consider complex expressions where all cases have to be stored for each group).

4.6 Cost estimations

Pre-grouping is not superior in all cases. Thus we need rules to decide in which cases an optimizer should generate a plan with pre-grouping and in which not.

Consider a query where the result is grouped w.r.t. several dimensions. The result is a number of groups that is almost the same as the fact table result tuples. In this case, two grouping operations, one for pre-grouping and the second for post-grouping are applied, while the tuples for the residual join are not reduced considerably. The additional grouping operation can be more expensive than the cardinality reduction of the pre-grouping (see also the measurement results in Section 6).

Since statistics for multidimensional data are limited, we only consider the data distribution for a single dimension. A simple heuristic is the following:

Compute the upper bound of the number of groups by the combination of the grouping attributes of the query. If this number of groups is smaller than the number of estimated fact table result tuples, then apply pre-grouping. Generally speaking, the optimizer has to decide whether the reduction of join attributes is more beneficial than an additional grouping operation.

5 Advanced topics

In this section we only draw a short outline for some topics that are important for the integration of pre-grouping to address real DW scenarios, but the details are beyond the scope of this paper.

5.1 Aggregation on dimension attributes

In most cases, star queries aggregate fact table measure attributes. But the aggregation operation has to be modified, if a dimension attribute $a_d$ is involved in the aggregation. Pre-Grouping is done on the h-surrogate prefix w.r.t. the grouping order GO of the dimension (see Section 4.1). Each resulting group from the pre-group operation represents a number of tuples with the same value of $a_d$. The value of $a_d$, however, is still unknown. The aggregation operations MIN and MAX are implemented straightforward and evaluated after the residual join. SUM has to be modified: We additionally compute COUNT(*) in the pre-group step and multiply it with the value of $a_d$ after the residual join.

AVG is computed by SUM/COUNT and is also affected by the special SUM computation.

For example consider a query with SUM,LOCATION.population) in the SELECT clause. For each tuple $t_k$ representing group $S_k$ of the pre-grouping phase we store in $t_k'.cnt$ the number of original fact table result tuples contributing to this group (COUNT(*)). In the residual join phase with LOCATION, we compute the aggregation value: $a = t_k'.cnt \times LOCATION.population$.

5.2 Expressions in aggregations

A more difficult problem are complex aggregations, i.e., expressions in the aggregation functions, especially, if fact and dimension attributes occur within such an expression. An example is a measure with different currencies and exchange rates. In this case, we have a currency dimension table Currency (CID, exchrate) and the following SQL statement:

```
SELECT SUM(F.turnover * C.exchrate)
FROM Fact F, Currency C
WHERE F.dcurrency = C.cid AND ...
```

At pre-grouping time only $F.turnover$ is known. Thus special preparation is necessary: We have to split the expression into an expression that can be computed at pre-grouping and an expression that is computed later (similar to Section 5.1). In the example above, we have to delay the computation of $C.exchrate$ after the residual join.

Note that various expressions can be split into expressions that can be handled as described, e.g., SUM($F.m_1*D.h_j + F.m_2*D.h_j$) is split into SUM($F.m_1*D.h_j$) + SUM($F.m_2*D.h_j$). Special information has to be stored within the groups, i.e., the atomic expressions and the arithmetic operations.

Additionally, we must care about the NULL semantic. If one of the attributes involved in the expression is NULL, it does not contribute to SUM. (and thus to AVG). Therefore, we have to check if any attribute is NULL: for measure attributes at pre-grouping, for dimension attributes after the residual join.

Complex expressions can be handled for the aggregation functions SUM and AVG. Complex MIN and MAX expressions cause a huge effort to implement pre-grouping (see Section 5.1).

5.3 Postfiltering dimensions

The discussion so far was based on the assumption that the tuples retrieved from the fact table (Predicate Evaluation and Fact Table Access) are exact w.r.t. the predicates of the query.

However, in some cases, the optimizer could decide to retrieve a super set and reduce the tuples via a post-
filtering operation. For example, a superset can be the consequence, if an index on a dimension key is not used or does not exist (non key dimension). Usually a residual join is necessary to evaluate the final fact table result tuples (the residual join acts as post-filtering operation).

It is possible to do pre-grouping in such a case. Instead of performing the residual join on the fact table result set we first pre-group the tuples w.r.t. h-surrogate prefixes. For the h-surrogate prefix of a “post-filter” dimension $D_i$ we extend the dimension order $DO$ for dimension $D_i$ as $DO(D_i) = \min(GO(D_i), AO(D_i), RO(D_i))$, where $RO(D_i)$ is the restriction order of $D_i$ and corresponds to the minimum hierarchy level of the restricted dimension.

The tuples of the fact table super set are pre-grouped w.r.t. h-surrogate prefixes of all participating dimensions. The subsequent residual join filters the groups according to the restrictions on the “post-filter” dimensions.

Note that this case is a very important scenario in real DW. Often the schema cannot be designed optimally for every query. With this extension of pre-grouping we can speed up such queries significantly.

The order of residual joins influences the performance. When joining with a (non-key) dimension that reduces the number of groups (post-filtering), the subsequent joins profit from a reduced join cardinality.

6 Performance measurements

A full featured implementation of the technology introduced in this paper is available in the commercial relational DBMS Transbase Hypercube [14]. This section presents measurement results that evaluate the performance of the proposed techniques.

The measurements are performed on a two processor PC Pentium II, 400 MHz, with 768 MB RAM and a SCSI hard disk. All data is stored on one disk. The queries are executed with cold cache, i.e., cache effects (operating system and DBMS) are eliminated.

The DW schema consists of a fact table with five dimensions Warehouse, Product, Date, Transaction and Sales Payment and 49 measures, such as sales, total etc. The tuples are very large (average size of the tuples is 349 byte) and the space overhead of h-surrogates is 12 byte, i.e., about 3% of the complete tuple. The dimensions are organized w.r.t. the hierarchies and cardinalities as listed in Table 1. The dimensions Transaction and Sales Payment have no hierarchy.

The used data comes from a large electronic retailer in Greece. The fact table has 8.579.458 records, i.e., 2.79 GB raw data.

The query workload consists of 880 ad hoc star queries from a real-world application. We classify the queries into three groups according to their selectivity on the fact table (i.e., number of tuples retrieved from the fact table):

- $C_1 = [0.0-0.25]$: 0% to 0.25% of fact table, i.e., 0 to about 21K records (502 queries)
- $C_2 = [0.25-1.0]$: 0.1% to 1% of fact table, i.e., 21K to 85K records (234 queries)
- $C_3 = [1.0-10.0]$: 1.0% to 10.0% of fact table, i.e., 85K to 858K records (144 queries)

The classification All is used in the measurement results and is the union of $C_1$, $C_2$ and $C_3$: $All = C_1 \cup C_2 \cup C_3$.

The queries vary in the following parameters:

- Dimension Predicates: different hierarchy levels
- Grouping Attributes: different grouping attributes (Table 2) and different number of grouping dimensions (from 0 to 3, see Table 3)

In Table 2 we show the occurrences of the hierarchy levels of the dimensions in the queries. Table 3 lists the number of grouping dimensions in the queries. Note that in most queries we have two or three grouping dimensions, but there are also some queries without a grouping clause.

<table>
<thead>
<tr>
<th>Table 1. Dimensions and hierarchies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimension</td>
</tr>
<tr>
<td>Warehouse</td>
</tr>
<tr>
<td>Product</td>
</tr>
<tr>
<td>Date</td>
</tr>
<tr>
<td>Transaction</td>
</tr>
<tr>
<td>Sales Payment</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2. Grouping attributes in queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warehouse</td>
</tr>
<tr>
<td>Country</td>
</tr>
<tr>
<td>Geodept</td>
</tr>
<tr>
<td>County</td>
</tr>
<tr>
<td>City</td>
</tr>
<tr>
<td>Area</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3. Number of dimensions in group by</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Dimensions:</td>
</tr>
<tr>
<td>1 Dimensions:</td>
</tr>
<tr>
<td>2 Dimensions:</td>
</tr>
<tr>
<td>3 Dimensions:</td>
</tr>
</tbody>
</table>
The goal of the performance evaluation was to measure three alternative execution plans:
(a) the abstract execution plan as described in Section 3.1 (NOPREGROUP),
(b) the early grouping without hierarchical surrogates as explained in Section 3.2 (EARLYGROUP) and
(c) the hierarchical pre-grouping as described in Section 3.3 (PREGROUP).
Since the time for the grouping and residual join phases covers a large part of the complete query execution time (more than 50%), an optimization of this phase reduces query execution time significantly. As join strategy, we use an optimized nested loop join that stores each joined tuple within a hash table. All tuples of the fact table the same values of a dimension are stored once in the hash table and are looked up then for each further occurrence. Table 4 shows the average time that this third phase consumes compared to the complete query execution in the case for NOPREGROUP, EARLYGROUP, and PREGROUP for the Transbase implementation.

Table 4. Average time of 3rd query processing phase

<table>
<thead>
<tr>
<th></th>
<th>All</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOPREGROUP</td>
<td>58%</td>
<td>46%</td>
<td>72%</td>
<td>79%</td>
</tr>
<tr>
<td>EARLYGROUP</td>
<td>47%</td>
<td>39%</td>
<td>57%</td>
<td>61%</td>
</tr>
<tr>
<td>PREGROUP</td>
<td>30%</td>
<td>22%</td>
<td>38%</td>
<td>43%</td>
</tr>
</tbody>
</table>

In Table 5, we show the improvement of pre-grouping w.r.t. the grouping cardinality. NOPREGROUP/EARLYGROUP contains the reduction of numbers of groups of EARLYGROUP compared to NOPREGROUP. A value of 1.0 means that there is no reduction, a value of 2 means that the resulting number of groups for EARLYGROUP is 50% of NOPREGROUP etc.

The improvement of EARLYGROUP is low compared to NOPREGROUP. 50% of all queries have an improvement between 1,5 and 14,1, where the median is 2.6. For the PREGROUP case, the improvement is 245,8 to 4.708,0 with a median of 1.139,5.

This explains the advantage of hierarchical grouping and thus the speedup of the query execution times (Table 6).

Table 5. Comparison of grouping cardinality

<table>
<thead>
<tr>
<th></th>
<th>NOPREGROUP/EARLYGROUP</th>
<th>NOPREGROUP/PREGROUP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All</td>
<td>C1</td>
</tr>
<tr>
<td>MIN</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1. Quartile</td>
<td>1.5</td>
<td>1.4</td>
</tr>
<tr>
<td>MEDIAN</td>
<td>2.6</td>
<td>2.1</td>
</tr>
<tr>
<td>3. Quartile</td>
<td>14.1</td>
<td>7.2</td>
</tr>
<tr>
<td>MAX</td>
<td>530.931,0</td>
<td>1.210,6</td>
</tr>
</tbody>
</table>

Table 6. Speedup of EARLYGROUP and PREGROUP compared to NOPREGROUP

<table>
<thead>
<tr>
<th></th>
<th>EARLYGROUP/ NOPREGROUP</th>
<th>PREGROUP/ NOPREGROUP</th>
<th>PREGROUP/ EARLYGROUP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All</td>
<td>C1</td>
<td>C2</td>
</tr>
<tr>
<td>MIN</td>
<td>0.3</td>
<td>0.3</td>
<td>0.6</td>
</tr>
<tr>
<td>1. Quartile</td>
<td>0.9</td>
<td>0.9</td>
<td>1.0</td>
</tr>
<tr>
<td>MEDIAN</td>
<td>1.2</td>
<td>1.1</td>
<td>1.8</td>
</tr>
<tr>
<td>3. Quartile</td>
<td>3.4</td>
<td>2.2</td>
<td>4.5</td>
</tr>
<tr>
<td>MAX</td>
<td>15.2</td>
<td>8.8</td>
<td>15.2</td>
</tr>
</tbody>
</table>
Table 6 shows the comparison of the user queries against the warehouse described above. The table presents the speedup, i.e., \( \text{EARLYGROUP/NOPREGROUP} \) means the speedup of \( \text{EARLYGROUP} \) compared to \( \text{NOPREGROUP} \).

The column \( \text{All} \) contains all queries, the column \( C_i \) contains the results for queries of class \( C_i \) etc.

As one can see, the speedup of \( \text{EARLYGROUP} \) compared to the standard execution plan (\( \text{NOPREGROUP} \)) is between 1.1 and 1.7 for the median. The speedup of \( \text{PREGROUP} \) compared to \( \text{EARLYGROUP} \) is again between 2.3 and 3.9. This leads to a speedup of \( \text{PREGROUP} \) compared to \( \text{NOPREGROUP} \) from 3.6 to 6.6 for the median. The speedup depends on the query classes. Generally speaking, the speedup is the higher the more tuples belong to the fact table result set, i.e., for query classes \( C_3 \), because the number of join operations can be reduced significantly.

Note that the speedup of \( \text{PREGROUP} \) compared to \( \text{NOPREGROUP} \) of 50\% of all queries lies between 3.0 and 6.5 (the range from first to third quartile). The speedup of \( \text{PREGROUP} \) compared to \( \text{EARLYGROUP} \) of 50\% of all queries is between 1.0 and 5.8.

The maximum speedup of \( \text{PREGROUP/EARLYGROUP} \) of 35.1 comes from the fact that in this query \( \text{EARLYGROUP} \) (14.8 seconds) is slower than \( \text{NOPREGROUP} \) (10.7 seconds) and \( \text{PREGROUP} \) (0.4 seconds) is much faster than both. Thus, the speedup of \( \text{PREGROUP} \) compared to \( \text{EARLYGROUP} \) is larger than compared to \( \text{NOPREGROUP} \).

Considering the overall query execution times with optimized fact table access and pre-grouping optimization, the queries do not take longer than 70 seconds (276 seconds for \( \text{NOPREGROUP} \)). In the average for query class \( C_3 \) they take about 16 seconds (56 seconds for \( \text{NOPREGROUP} \)). Thus, even queries covering a large fraction of the fact table are executed within acceptable time frames on a machine with comparably low performance characteristics.

7 Related work

The publications contributing most to the pre-grouping methods described in this paper are [3] and [15] as mentioned in the introduction. [3] describes three principles of pre-grouping, i.e., invariant grouping, simple coalescing and generalized coalescing. [15] and [16] describe an early grouping and aggregation method very similar to the coalescing methods of [3]. As explained in Section 3, these methods do not consider hierarchical pre-grouping.

In the last years two additional publications, [9] and [10], discuss pre-grouping. [9] compares different grouping methods and introduces a mathematical model to estimate group sizes and [10] extends pre-grouping by practical implementation issues such as partial pre-grouping. [7] we present the basic abstract execution plan for star queries on a schema with hierarchical encoding and a primary clustering multidimensional index on the fact table. Also the concept of pre-grouping is introduced shortly. In this paper we extend the discussion of pre-grouping by implementation issues and an exact performance evaluation.

8 Conclusion and future work

In this paper we present a combination of two technologies: early grouping first introduced in [3] and [15] and hierarchical encoding, described in [12] and [6]. Our approach addresses the expensive join and grouping phase which usually takes 50 to 80 percent of the execution time of a star query assuming that the fact table access is already optimized. By reducing the join phases, the join and grouping phase can be sped up in average by a factor of five. This leads to a speedup of the whole star queries by a factor of two to three. With these techniques it is possible to significantly reduce pre-aggregated materialized views (and the consequent space and maintenance problems) in order to perform queries with large fact table result sizes. In our measurements we show that even when accessing 10 percent of the fact table, we have acceptable query execution times for interactive query processing.

An additional issue of hierarchical pre-grouping is that the amount of memory necessary for a hash based grouping is reduced significantly (dependent on the reduction of the group cardinalities). A smaller amount of required main memory often avoids writing buckets to secondary storage and thus guarantees good performance. See [10] for a method to cope with main memory restriction.

We also discuss extensions of the basic implementation, such as complex aggregation expressions, complex schemata, post-filtering on dimensions etc. These implementation extensions are important for practical usage of pre-grouping.

There are still some open issues. Optimization w.r.t. join strategies, i.e., multi-way joins, hash joins have to be implemented. A reasonable cost model is very important to decide for the correct grouping strategy. Also extended \text{GROUP BY} \text{operators}, such as the cube operator have to be considered to fulfill current OLAP requirements.

Acknowledgement

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


