Maik Thiele, Jens Albrecht, Wolfgang Lehner

**Optimistic Coarse-Grained Cache Semantics for Data Marts**

Erstveröffentlichung in / First published in:

*18th International Conference on Scientific and Statistical Database Management (SSDBM’06)*. Vienna, 03.-05.07.2006. IEEE, S. 311-321. ISBN 0-7695-2590-3

DOI: [https://doi.org/10.1109/SSDBM.2006.38](https://doi.org/10.1109/SSDBM.2006.38)

Diese Version ist verfügbar / This version is available on:

[https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-788403](https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-788403)
Optimistic Coarse-Grained Cache Semantics for Data Marts

Maik Thiele  
Dresden University of Technology  
01069, Dresden  
amaik.thiele@inf.tu-dresden.de

Jens Albrecht  
GfK Marketing Services  
90319, Nuremberg  
jens.albrecht@gfk.com

Wolfgang Lehner  
Dresden University of Technology  
01069, Dresden  
lehner@inf.tu-dresden.de

Abstract

Data marts and caching are two closely related concepts in the domain of multi-dimensional data. Both store pre-computed data to provide fast response times for complex OLAP queries, and for both it must be guaranteed that every query can be completely processed. However, they differ extremely in their update behaviour which we utilise to build a specific data mart extended by cache semantics. In this paper, we introduce a novel cache exploitation concept for data marts – coarse-grained caching – in which the containedness check for a multi-dimensional query is done through the comparison of the expected and the actual cardinalities. Therefore, we subdivide the multi-dimensional data into coarse partitions, the so called cubletets, which allow to specify the completeness criteria for incoming queries. We show that during query processing, the completeness check is done with no additional costs.

1. Introduction

Data marts storing pre-aggregated data, prepared for further roll-ups, play an essential role in data warehouse environments and lead to significant performance gains in the query evaluation. Since the number of all possible group-bys increases exponentially with the number of dimensions, it is usually impossible to precompute the whole cube. Furthermore, not every group-by is semantical reasonable and therefore, a computation is not necessary. For this reason, data marts contain only a subset of all possible group-bys derived from the knowledge about the well-known workload. Anyway, it must be revisable for every query on the data mart whether these could be answered completely or not what leads to an analogy between data mart and cache.

Data Mart vs. Cache The notion of data marts as precomputed summarized tables which provide fast access to pre-aggregated data is very similar to the cache idea, which says that frequently used tuples are held under the assumption of locality. The most important issue in such a system is: can the query be computed from the cache/data mart effectively, and is this lookup fast enough, so that the lookup costs do not exceed the time required to bypass the cache? Other aspects of cache systems such as invalidation or replacement lose importance in the domain of data marts, where the time period between updates can be as long as a few days or even weeks (Figure 1). Furthermore, data mart updates are restricted to append operations, i.e. data gets never be replaced and is valid for the whole data mart life time. This is different to cache systems where the initial load is relative small and is getting replaced over time. Nevertheless it
must be guaranteed that every query on the data mart can be processed completely.

**Contribution of This Paper** In this paper we introduce a novel concept for aggregate data processing called coarse-grained caching. The idea is to decompose all aggregates into coarse partitions of the multi-dimensional space, the cubelets. A cubelet is a container for all aggregates at any aggregation level for a certain partition of the data. In detail we make following contributions:

1. Based on the distinction between classification and feature attributes we propose a novel partitioning scheme for multi-dimensional data sets, dividing the data in so called cubelets.

2. The partitioning into cubelets enables us to define a simple completeness criterion. We provide an optimistic algorithm, which computes the query from the cache and does the completeness check in a single step with no additional cost. Only if the completeness check fails, the missing parts of the data are identified, computed and stored in the respective cubelets. A second computation from the cache will definitely give the correct result. The optimistic approach is extremely efficient if the cache can be filled in advance for a specific workload. As a prerequisite, we assume that also null aggregates are materialized in the data mart.

3. We evaluate the scalability of our cache enabled data mart using real and synthetic datasets.

**Organization of This Paper** The paper is organized as follows: Section 2 introduces the notion of cubelets and formally defines completeness in the domain of multi-dimensional data. Section 3 illustrates the query processing infrastructure. Experimental study is reported in Section 4. Finally, we present our experimental results in Section 5 and conclude in Section 6.

## 2 Partitioning the Multi-Dimensional Data Space

The coarse-grained cache concept relies on special characteristics of the multi-dimensional data model, which are described in the following section.

### 2.1 The Cubelet Partitioning Schema

The generally known multi-dimensional data cube is stretched by orthogonal dimensions, which can be further divided into classification and feature attributes. Classification attributes $CA_i$ ($i = 0, ..., n$) define a hierarchy of dimensional elements and are ordered according to their functional dependencies. Instances of classification attributes are denoted as classification nodes $N$, e.g. ‘DVD’ as instance of the classification attribute ‘product family’ (see Figure 2). The highest classification attribute $CA_0 = TOP$ with the single instance ‘ALL’ is member of each dimensional structure. Aside the product dimension, we extend our ongoing example by a second shop dimension with the classification hierarchy (Region, Country, State, City, ShopId). Feature attributes $FA_i$ ($i = 0, ..., n$) of a dimension represent all descriptive features or properties of the dimensional elements (Figure 2). In contrast to classification attributes, a feature attribute does not functionally determine any other feature attribute. Some feature attributes may be valid for all dimensional elements, whereas other feature attributes exist only on certain nodes of the classification hierarchy [8]. For example, properties like ‘color’ or ‘brand’ are valid for all dimensional elements in the product dimension, whereas properties like ‘screen size’ or ‘resolution’ may be only valid for the classification node ‘TV’. To simplify our descriptions, we restrict our analysis to feature attributes which occur in each dimensional element within a dimension, e.g. ‘color’ or ‘brand’.

The granularity and range of data within the multi-dimensional structure is defined by the partitioning scheme $P$ denoting the granularity, and a partitioning descriptor $PD$ specifying the range, i.e. the selection criteria:

**Partitioning Schema:** A partitioning schema $P$ is an $n$-tuple $(CA_1, ..., CA_n)$, where each element is a classification attribute, prefixed with the dimension identifier, e.g. “Product.Family” or “Shops.Country”.

**Partitioning Descriptor:** A partitioning descriptor $PD$ is an $n$-tuple $(N_1, ..., N_n)$, where each $N_i$ is a node of the classification level described by the elements of $P$. 

![Figure 2. Classification and feature attributes of the product dimension](image)
If the partitioning schema $P$ solely consists of classification attributes at the lowest granularity, e.g. $P = (P.ProductId, S.ShopId)$, it is referred to as raw partitioning schema $P_{R}$. In the two-dimensional context stretched by the product and shops dimension, ("DVD", "Germany") is a valid partitioning descriptor for the partitioning schema (Product.Group, Shops.Country).

### 2.2 Cubelets

With regard to our cache concept, the multi-dimensional data as well as the queries are represented as cubelets. A cubelet $C$ is a triple $(P, PD, FA)$, where $P$ describes the cubelet granularity, and where $PD$ denotes the cubelet context, which can be further segmented by a set of feature attributes $FA$. Thus, the cubelet which can be seen as a data container decomposes the multi-dimensional space into low-dimensional coarse partitions, which reduces the complexity making it easier to manage the multi-dimensional space. Although a cubelet has a very simple structure, it can contain data at any dimensionality specified by a nested set of feature attributes. Cubelets which share the same partitioning schema $P$ can be combined to a cubelet set $S$. The partitioning schema of a cubelet set $S$ is denoted as set partitioning schema $P_S$.

$$S = \{C_1, ..., C_n \mid \forall i P_i = P_S\}$$

As the aggregates in the data mart all have the same defined granularity according to the classification attributes, we denote this partitioning schema as global partitioning schema $P_G$. Based on that, we distinguish between two types of cubelets: cubelets sharing the global partitioning schema $P_G$ and cubelets having an arbitrary partitioning schema $P$. For our ongoing example, we assume $P_G = (Product.Group, Shops.Country)$. The raw data from which the precomputed aggregates and thus the cubelets are derived is partitioned by the raw partitioning schema $P_R$. The appropriate cubelet sharing $P_R$ is defined as follows:

Raw Cubelets: A raw cubelet $C_R = (P_R, PD, FA)$ shares the raw partitioning schema and is part of the raw cubelet set $S_{raw}$ (Figure 3a).

The data mart partitioned by partitioning schema $P_G$ consists of total cubelets and data cubelets. Total cubelets represent the general availability of the data at the global partitioning granularity, whereas data cubelets represent the data mart content itself, i.e. the precomputed aggregates.

Total Cubelets: A total cubelet $C_T = (P_G, PD, \emptyset)$ shares the global partitioning schema $P_G$. It consists of one cell $A = \{0|1\}$, which denotes the availability of the data cubelet with the same partitioning descriptor, i.e. $PD_{C_T} = PD_{C_Q}$ (Figure 3b). Each total cubelet is part of the total cubelet set $S_{total}$.

Data Cubelets: A data cubelet $C_D = (P_G, PD, FA)$ shares the global partitioning schema $P_G$. It spans a set of cells holding either a numerical value, a null value denoting that no data is available in the cell context, or the value n.a. (not available) indicating that the cell state is unknown, i.e. is not computed. For each data cubelet $C_D$, there exists one total cubelet $C_T$, whereas $PD_{C_T} = PD_{C_Q}$ (Figure 3c). Each data cubelet is part of the data cubelet set $S_{Data}$.

Queries on cubelets are represented as cubelets as well. Cardinality cubelets are derived from a set of total cubelets and define the expected cardinality, i.e. the completeness condition. Query cubelets holding the query result are checked against this condition to verify the completeness.

Cardinality Cubelets: A cardinality cubelet $C_C = (P_C, PD_C, \emptyset)$ consists of one cell which denotes the number of total cubelets addressed by partitioning schema $P_C$ and partitioning descriptor $PD$, whereas $P_C \geq P_G$ (Figure 3c).

Query Cubelets: A query cubelet $C_Q = (P_Q, PD_Q, FA_Q)$ is derived from a set of data cubelets $C_D$, holding the query result and the cardinality of each cell (Figure 3d), whereas $P_Q \geq P_G$. For each query cubelet $C_Q$, there exists one cardinality cubelet $C_C$ with $P_Q = P_C$ and $PD_Q = PD_C$.

The granularity of the data mart content specified by the global partitioning schema $P_G$ defines the lower limit for all queries on the data mart. This means that no query cubelet can be answered if at least one classification attribute of $P_Q$ has a lower granularity than the corresponding classification attribute in the global partitioning schema. After having provided an idea of how to partition the multi-dimensional data mart and queries into different types of cubelets, the next section introduces basic cubelet operations that bear special significance for the completeness requirements.

### 2.3 Operators on Cubelets

In this section, basic cubelet operations are defined to sketch the use of cubelets in broad terms and to facilitate a subsequent description of the query processing.

- The roll-up operator corresponds to an aggregation process where at least one attribute of the new partitioning schema is "coarser" than the attributes of the partitioning schema of the source cubelet set. The op-
erator can only be applied to cubelet sets. The result of a roll-up is always a single cubelet.

\[ C' := \{ P, PD \} S, \text{ where } P \geq P_S \]

- The **equalize operator** replaces the partitioning schema of a cubelet or rather a cubelet set with the global partitioning schema \( P_G \). The equalize operator applied to a cubelet coarser than the global partitioning schema results in a cubelet set.

\[ S' := \downarrow S \text{ or } S' := \downarrow C, \text{ iff } P_C > P_G \]

- The **collapse operator** removes all feature attributes from a cubelet \( C \), thus decrementing the dimensionality of the cubelet to the dimensionality of the partitioning schema.

\[ C' := \leftarrow C := (P, PD, \emptyset) \]

- The **compare operator** checks two cubelets, a total cubelet \( C_T \) and a query cubelet \( C_Q \), for different cardinality values in the same cell context. The result of the compare operator is a set of cubelets denoting the feature attribute context with differing cardinalities:

\[ S_{\Delta} := \Delta(C_T, C_Q) \]

By applying the roll-up operator to a set of cubelets, the cubelet cells aggregated according to a specified aggregation function. To simplify further descriptions, we restrict our analysis to the \( SUM() \) and \( COUNT() \) aggregation functions, e.g. \( C' := \downarrow SUM(P, PD) S \). Since \( COUNT() \) is independent of the aggregation type of an attribute ("flow," "stock" or a "value-per-unit"), it can always be applied in combination with other aggregation functions [9], e.g. \( C' := \downarrow \{ SUM, COUNT\}(P, PD) S \). Other functions like \( AVG() \), \( MIN() \) and \( MAX() \) work as well but are beyond the scope of this paper.

Aside from the operators on cubelets we also need operators for cubelet sets, more precisely **join** and **minus** operators similar to the natural join and minus in the relational algebra. Both operators can solely be applied to cubelet sets sharing the global partitioning schema. For two given cubelet sets \( S \) and \( S' \), where \( P_G = P_S = P_{S'} \), the operators are defined as following:

- The result of the **join operator** is the set of all combinations of cubelets in \( S \) and \( S' \) that are equal in their partitioning description.

\[ S'' := \uplus (S, S'), C \subseteq S \land C' \subseteq S' \]

\[ S'' \supseteq C'' := (P_G, PD_C, FA_C \cap FA_C'), \text{ if } PD_C = PD_C' \]
The result of the minus operator is a set of those cubelets that hold the value not available (n.a.) in a feature value combination of \( S' \) but exist in the corresponding context of \( S \).

\[
S'' := -(S, S'), \quad \text{where} \quad PD_C = PD_{C'} \land FA_C = FA_{C'}
\]

### 2.4 Cubelet Completeness Specification

In order to verify the completeness of a data cubelet, the cardinality metric is introduced. In general, the cardinality is the number of data cubelet cells in a set of cubelets addressed by a partitioning schema \( P \), a partitioning descriptor \( PD \) as well as a set of feature attributes \( FA \). Thus, the cardinality is an implicit result of the roll-up operator specified above. Depending on the cubelets the roll-up is applied to, we can distinguish between two types of cardinalities.

**Expected Cardinality:** The expected cardinality stored in a cardinality cubelet \( C_C \) consisting of one cell is the number of total cubelets addressed by \( C_C \), \( C_C := \lceil \text{COUNT}(P_C, PD_C)S_{\text{Data}} \rceil \).

**Actual Cardinality:** The actual cardinality of a query cubelet cell is the number of cells from the underlying data cubelets which either hold a numerical or a null value, \( C_Q := \lceil \text{COUNT}(P_Q, PD_Q)S_{\text{Data}} \rceil \). This means that the actual cardinality is incremented by

\[
\begin{cases} 
1 & \text{if the cell is not n.a.} \\
0 & \text{if the cell is n.a.}
\end{cases}
\]

The cardinality cubelet is derived from low-dimensional total cubelets, whereas the query cubelet is derived from high-dimensional data cubelets. But since the roll-up operates solely on classification attributes, which are the same for data and total cubelets, the comparison of both cardinalities leads to the completeness definition.

**Completeness:** A query cubelet \( C_Q \) addressing a set of data cubelets is answered completely if the actual cardinality for each cell of \( C_Q \) is the same as the expected cardinality in \( C_C \), whereas \( C_C =\leftarrow C_Q \).

To illustrate the previous definitions, the next section presents a detailed example.

### 2.5 Example

The schema in Figure 3 (p. 4) illustrates our further descriptions. Consider a global partitioning schema \( P := (S.COUNTRY, P.GROUP) \) for the total cubelet set \( S_{\text{Total}} \) and the data cubelet set \( S_{\text{Data}} \). The data cubelet set consists of three cubelets \( (GER, TV), (GB, TV) \) and \( (FR, TV) \), which are further segmented by a set of feature attributes \( FA := \{ \text{color}, \text{brand} \} \) (Figure 3e). The numerical values in the data cubelet cells denote the sales of the product group television in the appropriate countries Germany, Great Britain and France. These sales values are subdivided into the colors "black" and "silver," and the brands "Sony" and "Aiwa." For each data cubelet, there exists one total cubelet without the additional feature attributes (Figure 3b). The total cubelet set is derived from the raw cubelet set by replacing the raw partitioning schema with the global partitioning schema \( P_G \) and by removing the feature attributes \( S_{\text{Total}} := \lceil \lceil S_{\text{Raw}} \rceil \rceil \).

The data cubelet in context \((GER, TV)\) holds a null value for cell \{black, Aiwa\}. This means that not a single black Aiwa TV set was sold in Germany. In contrast, the cell \{black, Aiwa\} in cubelet \((FR, TV)\) is "not available" (n.a.), which means that this cell has not been computed and therefore, its value is unknown.

A query cubelet \( C_Q \) with the partitioning schema \((S.REGION, P.GROUP)\) and partition descriptor \((EU4, TV)\) is specified (Figure 3e). The value 'EU4' is the fusion of the four different countries Germany, Great Britain, France and Spain. The corresponding cardinality cubelet is defined as \( C_C := \leftarrow C_Q \) (Figure 3d). The expected cardinality is the number of total cubelets addressed by the cardinality cubelet. For our example, the expected cardinality is 3 since no total cubelet exists for Spain. That means, no sales data is available in the raw data for that country. This is an important observation: The expected cardinality is not just the Cartesian product of the requested classification attribute values, which would be 4 for our example. It denotes the general availability of data at the granularity of the global partitioning schema.

The actual cardinalities, together with the summarized sales values, are derived from the data cubelet set \( S_{\text{Data}} \).

\[
C_Q := \{ \text{SUM, COUNT} \}(P.REGION, EU4)S_{\text{Data}}
\]

To compare both cubelets, we apply the compare operator \( S_{\Delta} := \Delta(C_T, C_Q) \), which leads to the cubelet \{black, Aiwa\}, which has a cardinality of only 2. To identify the missing cubelet cell, the coarse cubelet set \( S_{\Delta} \) is broken down to the global partitioning schema and joined with the total cubelet set \( S_{\text{Total}} \).

\[
S_{\text{Ref}} := \approx (\lceil S_{\Delta}, S_{\text{Total}} \rceil)
\]

The resulting cubelet set denotes the cubelets which should be available. A minus with the data cubelet set leads to the missing cubelet which must be computed to answer the query completely.

### 2.6 The Semantics Of Null Cells

It is essential for our cardinality comparison to distinguish between null and n.a. cells, denoting the difference
of an aggregate value which is computed but null and an aggregate combination which is not existent within the data mart, i.e. the value is unknown. This semantics implies that resulting null values must be stored explicitly and cannot be omitted. The knowledge of the non-existence of a cell is a necessary and valuable piece of information, required to increment the actual cardinality of a derived query cubelet. Otherwise – with a very high probability – the actual cardinality would be always smaller than the expected cardinality since null values occur quite often in sparse multidimensional data. If the null value for the example from the last section would not be stored, the actual cardinality for the corresponding cell would by 2 instead of 3.

3 Designing the Data Mart Cache

In the previous section we formally introduced our cubelet partitioning schema. In this section we want to utilize the cubelet idea and illustrate the cache infrastructure as well as the query processing workflow.

3.1 Query Processing Infrastructure

A cache as well as a data mart are collections of duplicated data, where the original values would be expensive to compute relative to reading the cache or the data mart. In contrast to a cache a data mart is barely limited according to the available storage space. This means that a data mart can be pre-filled with the typical workload so it can answer the majority of all queries. Nevertheless it must be guaranteed that a query is answered completely, i.e. is computed on raw data in case that data was missing in the cache enabled data mart. This can be achieved by applying the cubelet idea as illustrated below.

Global Partitioning Schema The most important step during the setup of the cache enabled data mart is the specification of the global partitioning schema. As mentioned in section 2.1 the global partitioning schema consists of a set of attributes associated to a classification hierarchy, in other words each attribute which belongs to a classification hierarchy is part of the global partitioning schema. The granularity of the global partitioning schema influences the query workload which can be answered by the data mart as well as the query processing time. A high granularity restricts the workload to a few “coarse” queries which can be answered very efficiently, whereas a low granularity is more flexible with respect to the workload but requires more processing resources. Since the typical workload is well-known in the most applications the global partitioning schema can be specified according to that points. As in the previous section we consider a global partitioning schema \( P_G = \langle Product.Group, Shops.Country \rangle \).

 Aggregate and Publish Table The design of the query processing infrastructure is based on the notion of dividing the multi-dimensional space into coarse uniform partitions along the classification hierarchy, the cubelets. According to the separation into data and total cubelets, the aggregates are stored into the so-called aggregate table and the availability state is stored in the publish table (see Figure 4 for an example). The feature attributes for the aggregates are stored separately in the feature set table that is linked from the aggregate table by the foreign-key featureid. Each feature set consists of a set of tuples \( FA_i = v_i \), whereas each tuple represents a feature value pair, e.g. \( \text{color} = \text{'silver'} \). These data structures reduce the storage space of the aggregate table since the most cubelets share the same feature sets. For example the feature set \( \text{color} = \text{'silver'} \) occurs in almost every cubelet independent of the values for product group and country.

The availability of the raw data is stored in the publish table which is initial filled with following query.

![Figure 4. Example for the cache infrastructure](image)

As an ongoing example, we consider a query \( Q \) which asks for the aggregate sales of the product groups ‘DVD’ and ‘TV’ in two regions ‘EU2’ and ‘EU5’ denoting federations of two and five European countries; additional divided by three different feature sets (see feature table in Figure 4):

```sql
SELECT s.region, a.group, SUM(sales) FROM AGG a, Shop s WHERE a.country = s.country AND s.region IN (EU2, EU5) AND a.featureid IN (2, 3, 5) GROUP BY s.region, a.group, a.featureid
```
An overview of the query processing can be seen in Figure 5. For each query $Q$ which should be answered using the cache, the selection and group-by attributes need to be analyzed to determine the expected cardinality for each cubelet, i.e. the number of aggregates, required to answer the query. From query $Q$, a new query $Q'$ is derived by replacing the aggregate table in the FROM clause with the publish table and replacing the occurrence of all feature attribute predicates in the SELECT and GROUP – BY clause. Furthermore the aggregation function is replaced with a $\text{COUNT}(*)$. This query is executed and the results, the expected cardinality for each cubelet, are temporarily stored in a table expcard_temp (step 1 in Figure 5).

SELECT s.region, p.group, COUNT(*)
FROM PUBLISH p, SHOP s
WHERE p.country = s.country
AND s.region IN (EU2,EU5)
GROUP BY s.region, p.group

For our example, the expected cardinalities are 2 and 5 respectively. Next, a $\text{COUNT}(*)$ is added to the SELECT clause of the original query $Q$. This query is executed to obtain the query result and the cardinalities for each derived aggregate (step 2). The additional $\text{COUNT}(*)$ does not lead to any measurable extra costs for the query processing (see section 4.1). Again the query result is temporarily stored in a table called query_temp.

In the third step, the cardinalities of the result aggregates are compared to the expected cardinalities by joining both tables over the classification attributes, e.g. region and group. Each join tuple which differs in the cardinalities denotes that an aggregate is missing in the appropriate cubelet (this corresponds to the compare operator in section 2.3).

SELECT q.*, e.card
FROM query_temp q, expcard_temp e
WHERE q.productgroup= e.productgroup
AND q.region = e.region
AND q.card <> e.card

If the result of that join is empty, the query is completely answered and the process stops. Otherwise, the missing aggregates need to be identified and computed. For the ongoing example, the result tuple ('EU5', 'TV', 5) is incomplete since the cardinality is 4 instead of 5. This example was chosen to demonstrate the whole query processing workflow. Since the data mart content is oriented on the typical workload, the cardinality check will be positive for the majority of the queries and therefore the query processing will be finished after the third step with a high probability.

The incomplete coarse aggregate ('EU5', 'TV', 5) must be decomposed into the cubelet granularity, group and country, to identify the one missing base aggregate. This is done by joining that incomplete aggregate with the product and shop dimensions to determine the global partitioning schema of the aggregate table, e.g. group and country. Furthermore, we need a join to the publish table to avoid those tuple which are not available anyway. For the example, we get 5 aggregate tuples which we denote as expected aggregates, i.e. all the aggregates which must be available to answer the query.

Then, we determine all base aggregate tuples, which are actual available, i.e. the existing aggregates (step 4). Therefore, we join the incomplete aggregate ('EU5','TV',5) with the appropriate total cubelets in the publish table and the dimension tables to resolve the global partitioning schema, i.e. group and country. For the example, we get 4 aggregate tuples.

To obtain only the missing base aggregates, we perform a $\text{MINUS}$ operation between both sets, the expected aggregates and the existing aggregates.
The resulting tuples are the missing base aggregates; in our scenario that is exactly one. These missing aggregates must be computed on the raw data (step 5) and merged into the data mart cache, more precisely in the aggregate table (step 6). Finally, the whole cache exploitation process with query \( Q \) must be repeated, since in the meantime the cache might have been updated. But different to the first run the second run will be finished guaranteed after step.

4 Performance Analysis

In this section, we show the results of experiments to validate our scalability and compression expectations. Both real and synthetic datasets were used in the experiments. The synthetic dataset satisfies Zipf distribution (skew = 1.5) and consists of 1.000.000 aggregates. The real dataset records market research data containing 2.853.234 tuples.

4.1 Scalability

The first set of experiments studies the scalability of the coarse-grained cache concept. Therefore, we generated a set of queries which could be completely answered just by using the data mart. Figure 6a illustrates the scalability of our cache concept as the number of aggregates increases from 100 thousand to 2.5 million with a query requesting 200 tuples. Figure 6b shows similar characteristics, as the runtime increases with the number of requested tuples. Both figures show that the runtime goes up linearly as the aggregate table size as well as the number of requested aggregates increases, i.e. coarse-grained caching is scalable with respect to the number of aggregates and the number of requested tuples.

4.2 Completeness Check

A second set of experiments studies the cost of the completeness check consisting of the computation of the expected cardinality, the additional \( \text{COUNT}(*) \) to determine the actual cardinality and the comparison of both values. The expected cardinality is computed using the publish table holding the availability of data at the granularity of the global partitioning schema. Compared to the aggregate table the dimensionality of the publish table is very low which is reflected in the size of both tables. To determine the completeness for the 2.8 Mio aggregates of the real data set only 1.019 records need to be stored in the publish table. Furthermore, the size of the publish table is fixed whereas the aggregate table grows over time. The size of the publish table strongly depends on the global partitioning schema which is specified by a set of classification attributes at a specified granularity. The higher the number of classification attributes and the lower the granularity, the more records need to be stored. Nevertheless the number of classification attributes is much lower than the number of feature attributes so the processing time of the publish table requests can be neglected in contrast to the overall runtime.

Furthermore, we measured the impact of the \( \text{COUNT}(*) \) which must be added to each query performed on the data mart. Table 1 illustrates the average runtimes for five different queries evaluated 50 times on the real dataset. The result was that the additional \( \text{COUNT}(*) \) led to an insignificant runtime overhead of 1.7 % against queries without the \( \text{COUNT}(*) \).

<table>
<thead>
<tr>
<th>Query</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>with count(*)</td>
<td>350</td>
<td>462</td>
<td>308</td>
<td>584</td>
<td>648</td>
</tr>
<tr>
<td>without</td>
<td>321</td>
<td>455</td>
<td>299</td>
<td>518</td>
<td>640</td>
</tr>
</tbody>
</table>

Table 1. Runtime Comparison of Queries with and without \( \text{COUNT}(*) \)

To verify the completeness of a query the expected and actual cardinality need to be compared. Therefore, the table with the expected cardinality for each coarse aggregate is joined with the query result including the actual cardinalities for each tuple. Through the low dimensionality of the global partitioning schema, the amount of tuples, i.e. the amount of cardinality cubelets derived from the publish table, is very low too. So, this join benefits significantly from the hash join operator, which builds an in-memory hash table of the smaller of the two relations. That can be seen in Figure 6b where the cardinality comparison does not influence the overall query processing for increasing requests.

4.3 Evaluation

Since the computation and comparison of the expected and actual cardinalities does not increase query runtimes significantly, our completeness check is almost for free. That means, our coarse-grained cache concept achieves its best performance if all requested tuples are already materialized in the data mart. This is the case for the majority of all queries, since the data mart content is oriented on the typical workload. Because of our optimistic approach we get the query result with the additional completeness verification at almost the same costs as the query processing itself.

In a last experiment, we measured the influence of missing aggregates on scalability. With the increase of missing aggregates, which must be computed on raw data, the overall runtime goes up linearly (see Figure 6c). But once more,
the benefit for the majority of all queries which can be answered in one run strongly exceeds the runtime overhead generated by a few queries.

5 Related Work

The general problem of answering queries using view has been studied extensively in [6] [10]. Answering queries with aggregations using views has been studied in [11]. A fundamental problem is the query rewriting which, can be proved, is NP-complete [7] [10]. Finding rewritings for aggregate queries introduces additional sources for complexity compared to conjunctive queries without aggregation [1] [2]. Materialized views have the drawback of providing preaggregated data at fixed levels, implying that only a certain class of aggregate queries profits from the preaggregation. Within a cubelet we are able to hold aggregates at different levels. Furthermore, we avoid the rewrite mechanism since the containedness check is done on instance level by counting and comparing cardinalities.

DCache [3] uses cache groups and introduces the notion of cache key constraints and referential cache constraints to ensure value and domain completeness. Once these constraints are specified by the DBAs, DCache can asynchronously populate the cache tables on demand. However, DCache is inapplicable for caching aggregates and the use of cache constraints in the multi-dimensional model can easily lead to huge amounts of data which has to be loaded into the cache database.

Caching scheme specific to OLAP applications is proposed in [5]. They decompose the multi-dimensional space into chunks. For incoming queries, the required chunks are computed and split into two subsets: cached chunks and not cached chunks. To answer the query, the system will compute the missing chunks from the raw data. This approach was further extended in [4] considering also chunks at different aggregation levels. The chunk based caching is quite similar to our approach. Both split the multi-dimensional data into uniform semantic data regions which is very natural in the OLAP domain. But instead of storing the cache data in chunked multi-dimensional arrays, we use the notion of partitions to define the completeness criteria for all resulting aggregates of a query.

To reuse the results of former queries, summarizability of aggregates as the units of caching is a necessary prerequisite which was not mentioned in detail in this paper. For definitions on derivability see the work of [9].

6 Summary and Outlook

We have introduced a novel cache exploitation framework for multi-dimensional data which works very well in the data mart domain with its specific update characteristics. A further contribution of this paper is the formal introduction of cubelets that allowed us to define query completeness through the comparison of expected and actual cardinalities. We evaluated the scalability of our framework using real and synthetic datasets. Since the coarse-grained caching framework maintains the relational nature of data it can be easily implemented into existing data mart environments.

As mentioned in section 2.6 it is essential for the cardinality comparison to store null aggregates explicitly. Depending on the density of the raw data, this results to a high amount of null aggregates that need to be stored. First experiments show that null aggregates make up 90% of all aggregates. Due to the special characteristics of null aggregates which do not contribute anything to the result of a query but are necessary to determine the exact cardinalities, they can be stored in an alternative and compact manner. It can be observed that there are dependencies between aggregates of related feature attribute sets. Thus, it is obvious that when an aggregate with the feature ‘color’ and the feature value ‘blue’ has a null value for a specific measure the finer ag-
aggregate 'color=blue, size=large' also must be null for the appropriate measure. So, instead of storing all aggregates assigned with null values, specific aggregates can be identified from which all other null aggregates can be derived whenever it is necessary to determine the exact cardinalities. A first promising prototype achieved a compression ratio of more than 90%. In future work, we plan to integrate these lossless null reduction algorithm in our coarse-grained cache framework.

We are currently working on the finalization of the implementation of our coarse-grained cache for a real-world market research project in cooperation with GfK Marketing Services, Nuremberg.

References