Reihe Informatik
1 / 1998

Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing

Guido Moerkotte

Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing

Guido Moerkotte

Lehrstuhl für praktische Informatik III
Universität Mannheim
Seminargebäude A5
68131 Mannheim
Germany
moer@pi3.informatik.uni-mannheim.de

Abstract

Small Materialized Aggregates (SMAs for short) are considered a highly flexible and versatile alternative for materialized data cubes. The basic idea is to compute many aggregate values for small to medium-sized buckets of tuples. These aggregates are then used to speed up query processing. We present the general idea and present an application of SMAs to the TPC-D benchmark. We show that application of SMAs to TPC-D Query 1 results in a speed up of two orders of magnitude. Then, we elaborate on the problem of query processing in the presence of SMAs. Last, we briefly discuss some further tuning possibilities for SMAs.

1 Introduction

Among the predominant demands put on data warehouse management systems (DWMSs) is performance, i.e., the highly efficient evaluation of complex analytical queries. A very successful means to speed up query processing is the exploitation of index structures. Several index structures have been applied to data warehouse management systems (for an overview see [2, 17]). Among them are traditional index structures [1, 3, 6], bitmaps [15], and R-tree-like structures [9].

Since most of the queries against data warehouses incorporate grouping and aggregation, it seems to be a good idea to materialize according views. The most popular of this approaches is the materialized data cube, where for a set of dimensions, for all their possible grouping combinations, the aggregates of interest are materialized. Then, query processing against a data cube boils down to a very efficient lookup. Since the complete data cube is very space consuming [5, 18], strategies have been developed for materializing only those parts of a data cube that pay off most in query processing [10]. Another approach–based on [14]–is to hierarchically organize the aggregates [12]. But still the

storage comsumption can be very high, even for a simple grouping possibility, if the number of dimensions and/or their cardinality grows. On the user side, the data cube operator has been proposed to allow for easier query formulation [8]. But since we deal with performance here, we will throughout the rest of the paper use the term *data cube* to refer to a *materialized data cube* used to speed up query processing.

Besides high storage consumption, the biggest disadvantage of the data cube is its inflexibility. Each data cube implies a fixed number of queries that can be answered with it. As soon as for example an additional selection condition occurs in the query, the data cube might not be applicable any more. Furthermore, for queries not foreseen by the data cube designer, the data cube is useless. This argument applies also to alternative structures like the one presented in [12]. This inflexibility—together with the extrordinary space consumption—maybe the reason why, to the knowledge of the author, data cubes have never been applied to the standard data warehouse benchmark TPC-D [19]. (cf. Section 2.4 for space requirements of a data cube applied to TPC-D data) Our goal was to design an index structure that allows for efficient support of complex queries against high volumes of data as exemplified by the TPC-D benchmark.

The main problem encountered is that some queries refuse the application of a (traditional) index structure (like B-Trees [1, 3] and Extendible Hashing [6]) due to efficiency reasons. A typical situation is, when e.g. more than one tenth of a relation qualifies for a selection predicate. Then the only effect of using an index is to turn sequential I/O into random I/O (in the presence of a non-clustered index). Even worse, some queries are designed such that the use of an index structure is prohibitively expensive. An example of such a query is Query 1 (cf. Fig. 3) of the TPC-D Benchmark [19]. Its low selectivity—95%-97% of all tuples qualify—forbids the use of an index, and a sequential scan is the only possibility to "efficiently" evaluate this query. Taking a look at the TPC-D benchmark results[1] it becomes clear that Query 1 is among the two or three[2] most time consuming TPC-D queries.

Small materialized aggregates (SMAs) are designed such that they are useful even for queries where traditional indexes fall short. They differ from traditional indexes in three important aspects:

- They exhibit a very simple sequential organization.
- They directly reflect (and exploit) the physical organization of the indexed table.
- A single SMA is rarely useful, but in most situations a set of SMAs is required to answer a query efficiently.

SMAs share the first property with the lately introduced *projection indexes* [16]. In fact, SMAs can be seen as a generalization of projection indexes. In a projection index on a certain attribute, for all tuples in the relation to index, the attribute value is stored sequentially in a file. SMAs generalize this approach in that an aggregate value is stored for a set of tuples instead of mere projection values.

The above differences result in several advantages:

[1] see http://www.tpc.org

[2] Depending on the platform.

- SMAs can be used where traditional index structures fail.
- SMAs are very space efficient.
- SMAs are easy to implement.
- SMAs are cheap to maintain.
- SMAs are amenable to bulkloading.

The latter point is especially important for applications like data warehousing. Although there is this overwhelming set of advantages, there also exists a slight disadvantage: query processing—especially the generation of query execution plans—becomes a little more complex. Hence, we devote one section to this problem.

The rest of the paper is organized as follows. Section 2 presents the basic version of SMAs. This section also illustrates the usage of SMAs for processing Query 1 of the TPC-D benchmark and presents benchmark results for Query 1. Section 3 introduces query processing techniques exploiting SMAs. Section 4 briefly discusses further tuning measures and improvements of SMAs. Section 5 concludes the paper.

2 The Idea of SMAs

2.1 Definition of simple SMAs

We assume the relations for which SMAs are computed to be physically organized into a sequence of buckets. Examples of buckets are single pages or consecutive sequences of pages. A bucket *must* reflect the physical organization of the relation since the order of the entries in the SMA will directly correspond to the physical order of the buckets on disc. Hence, buckets can only be sets of consecutive tuples on disk. In this respect, SMAs are similar to projection indexes [16].

The main idea of SMAs is to compute and materialize a single value (or a set of values) for each bucket of tuples. These values will be aggregates. For all buckets, the resulting values are materialized in a separate SMA-file. The SMA-file is sequentially organized: the value for the first bucket is the first value in the SMA-file, the second value is the second value in the SMA-file and so on. Contrary to traditional index structures, a SMA-file does not contain any other additional information.

The above situation is illustrated in Figure 1. It contains three buckets with three tuples each. Every tuple contains one attribute L_SHIPDATE, whose value is specified in the figure. Two SMA-files materialize the minimum and maximum value found for L_SHIPDATE in each bucket. Further, there is one SMA-file materializing the number of tuples in each bucket.

SMAs can be specified by a simple SQL query and a specification of the bucket. However, there is one major point to obey:

- The **select** clause may contain only a single entry.

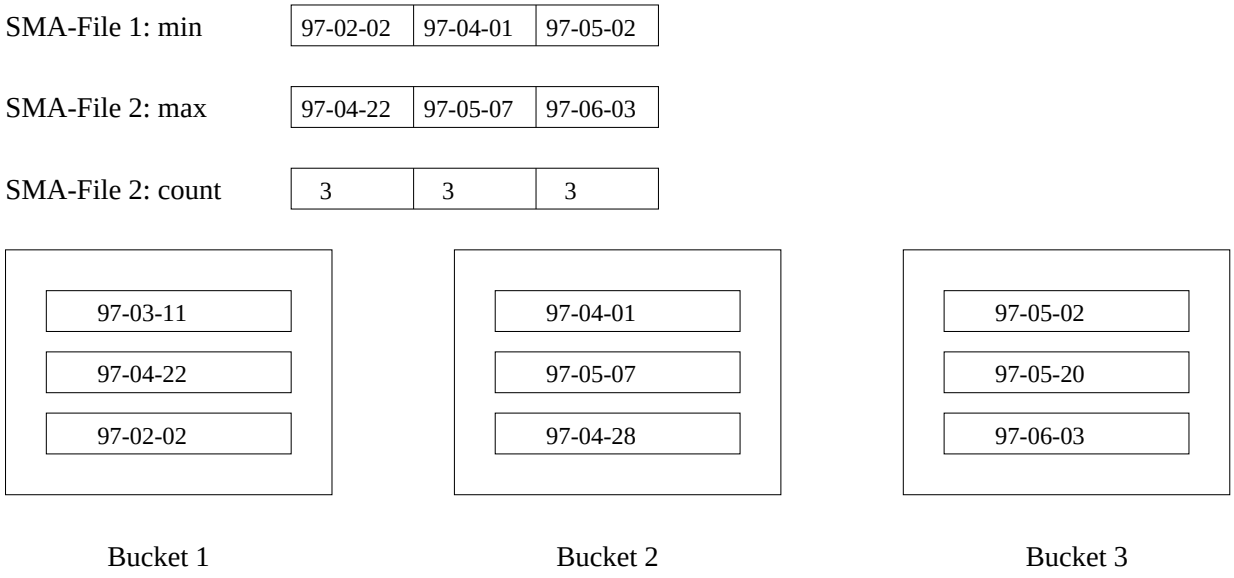

Figure 1: Buckets and SMA-Files

Another restriction we apply for the moment is that we forbid joins. Hence, we allow only for a single entry within the **from** clause. This restriction will be relaxed in Section 4. Further, we do not allow an order specification. The use of grouping is defered until the next subsection, the specification of bucket sizes until Section 4.

The following is a typical definition of a SMA called *min*:

define sma min
select min(L_SHIPDATE)
from L_LINEITEM

For every page, the minimum of all shipdates of tuples on that page is materialized. The consequence of this SMA definition is that a single SMA-file is created which is filled with the minimum values of shipdates found among the tuples in a bucket. Besides *min*, we allow for the aggregate functions *max*, *sum*, and *count* in the **select** clause of a SMA definition.

Some of the advantages of SMAs become clear already. They are very space efficient. Assume that a bucket corresponds to a 4K-page and a single date field can be stored in 32 bits, then the size of a single SMA-file is only 1/1000th of the size of the original data. Hence, many SMA-files can easily be supported. Further, due to the direct correspondance between SMA-file entries and buckets (via the order), SMA-files are easy to update. The algorithms behind are simple and very efficient. At most one additional page access is needed for an updated tuple. Last not least, bulkloading a SMA-file requires only simple algorithms and is very efficient. For every bucket the aggregate can easily be computed and storing this aggregate is cheap: only one page access is needed for 1000 pages of

tuples. Since nothing else has to be done (unlike in conventional index structures where pointer updates, splitting and the like occur) bulkloading and updating are both very simple and efficient operations.

2.2 Use and motivation of simple SMAs

In general, SMAs are used for two purposes. Given a query, SMAs are used

- to evaluate the selection predicate and
- to compute the aggregate values specified in the **select** clause of the query.

Whereas the usefulness of SMAs for the computation of aggregate values is quite obvious, the question arises how and when SMAs can be used to evaluate selection predicates. For the case where a bucket contains exactly a single tuple, a SMA degenerates to a projection index. Hence, we refer the reader to [16] for this case.

Although we defer the general answer to this question to Section 3, we give an important use of SMAs for selection predicate evaluation which is based on the exploitation of clustering. We base our discussion on implicit clustering since (1) we think this is the predominant application area of SMAs and (2) this case motivates SMAs quite nicely. Nevertheless, the following discussion applies to other clustering strategies as well.

Implicit clustering—sometimes called clustering by time of creation (TOC)–is often found in the data warehouse context where dates (times) of all kinds are of high importance. Examples of important dates are dates of order, shipment, arrival of items at customers, sending the bill, and payment [11, 13]. The TPC-D benchmark takes this fact into account by exhibiting four of these dates. In almost every of its queries at least one of these dates is referenced.

A time-of-creation clustering strategy is now (often implicitly) applied if new orders are stored in the data warehouse by appending them to the old orders. Note that this will be the case in most data warehouses. This kind of implicit clustering (which in practice is often imperfect but still exploitable) results in an implicit clustering on order dates. Since old orders will be processed earlier than new orders, a similar argument applies to shipdates and all the other dates mentioned before. Note however, that this does not result in a strict clustering or even ordering on orderdate or shipdate. Instead, this clustering is only approximated by reality: due to not available parts, a shipdate can be deferred, some shops might be late in providing their order information and so on. But the bottom line is that most likely there is some clustering effect of this kind, especially since data warehouses often contain data comprising several years. Figure 2 visualizes the effect of implicit clustering. For every order tuple, it contains one point. The x-value of a tuple is the date of its introduction into the data warehouse and the y-value is its order date. Since order tuples are typically introduced into the data warehouse after their arrival (order) date, all points lie to the right of the diagonal. Since all data points are clustered around the diagonal or at least some line close to it, we call this diagonal data distribution. In practice, there will be an average time needed before the data is entered into the database and the real intervals needed will exhibit a normal distribution around

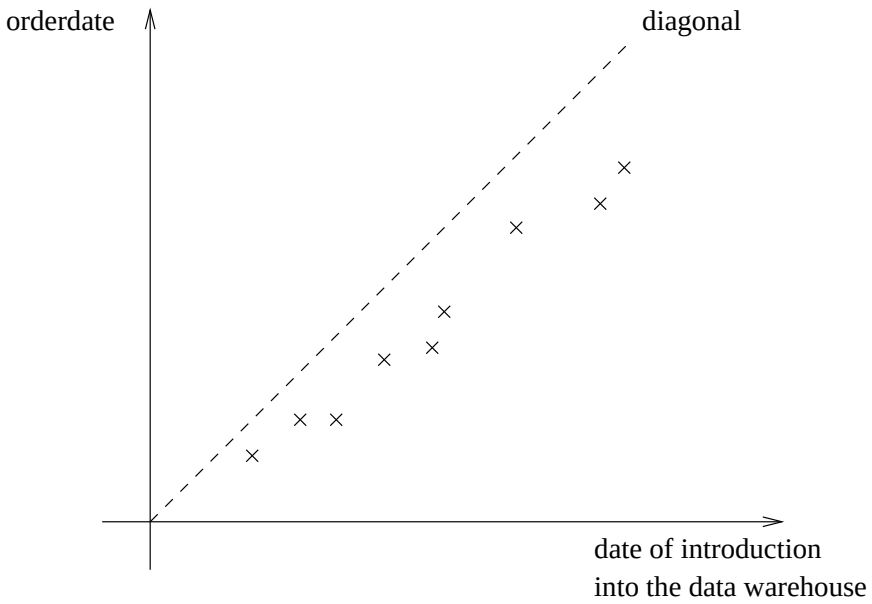

Figure 2: Diagonal Data Distribution

this average time. Consequently, the clustering effect becomes manifest. (Note that the TPC-D benchmark is not very realistic in this respect: it applies uniform distribution within an interval.) The same applies to shipdate. Assuming a certain average time needed to prepare the shipment, the actual times needed will be normally distributed and, again, the clustering effect becomes manifest. Of course, this clustering effect can also occur for non-date values, imprinted by seasonal effects, promotions and the like.

For (implicitly) clustered data, SMAs can be used very effectively to select those buckets in which qualifying tuples can be found. Consider the query

select count(*)
from L_LINEITEM
where L_SHIPDATE < 97-04-30

and assume that the attribute values shown in Fig. 1 are L_SHIPDATE values. Then, by inspecting the *max* SMA-file, it is easy to see that all the tuples in Bucket 1 qualify. By inspecting the *min* SMA-file, we see that none of the tuples in Bucket 3 qualify. Bucket 2 is called ambivalent since it does not qualify due to its *max* value and it does not disqualify due to its *min* value.

To answer the query we inspect the *count* SMA-file to retrieve the total count of qualifying tuples of Bucket 1 and add the number of qualifying tuples of Bucket 2. The latter we only get by inspecting the bucket itself. This example nicely illustrates the exploitation of diagonal data distribution: only the original tuples contained in ambivalent buckets have to be investigated. For clustered data, these ambivalent buckets are rare.

```
SELECT L_RETURNFLAG, L_LINESTATUS,
       SUM(L_QUANTITY) AS SUM_QTY,
       SUM(L_EXTENDEDPRICE) AS SUM_BASE_PRICE,
       SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS SUM_DISC_PRICE,
       SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)*(1+L_TAX)) AS SUM_CHARGE,
       AVG(L_QUANTITY) AS AVG_QTY,
       AVG(L_EXTENDEDPRICE) AS AVG_PRICE,
       AVG(L_DISCOUNT) AS AVG_DISC,
       COUNT(*) AS COUNT_ORDER
FROM LINEITEM
WHERE L_SHIPDDATE <= DATE '1998-12-01' - INTERVAL '[delta]' DAY
GROUP BY  L_RETURNFLAG, L_LINESTATUS
ORDER BY  L_RETURNFLAG, L_LINESTATUS
```

Figure 3: Query 1 of the TPC-D Benchmark

2.3 Grouping SMAs

Let us consider Query 1 of the TPC-D benchmark (cf. Fig. 3). This query involves a grouping. In order to be useful, a SMA has to reflect the grouping of the query or a finer grouping ([10]). This is done by specifying a **group by** clause in the specification of the SMA.

For every possible group, there will be a single SMA-file containing the aggregated values for this group. For example, Query 1 results in four groups. Hence, there will be four SMA-files, one for each combination of L_RETURNFLAG and L_LINESTATUS values. In order to process Query 1 soleley on the basis of SMAs, eight SMA definitions are necessary. They are given in Figure 4 where we took the liberty to abbreviate some of the attribute names. The SMAs *min* and *max* do not need a grouping and are of the kind discussed before. All the other attributes need a grouping with the attributes L_RETURNFLAG and L_LINESTATUS. Hence, each of these SMAs will be materialized in four SMA-files, one for each possible group. As a total there will be 26 SMA-files which seems to be quite high a number, but the next subsection will reveal that the time to build them as well as their storage costs are quite low.

Query 1 can now be answered by these aggregates in the following way. First, the SMAs *min* and *max* are used to classify the pages of the relation LINEITEM into qualifying, disqualifying and ambivalent pages. Second, for each qualifying page, for every group, the according values are extracted from the remaining SMAs and summed up in a per-group-wise fashion. For every ambivalent page, the page is visited and the needed aggregates are computed from the tuples contained in the page. These two steps are performed "in sync". That is, all the SMAs are scanned sequentially and at the same time: for every page in the LINEITEM file, the corresponding SMA values in all SMA-files are considered and the according action of the second step is taken. Note that this results in a sequential scan of the ambivalent pages of the LINEITEM file. In a last step, the average aggregates are computed from the sum aggregates by dividing by the count aggregate.

```
define sma  max                    define sma  dis
select      max(L_SHIPDATE)        select      sum(L_DISCOUNT)
from        L_LINEITEM             from        L_LINEITEM
                                   group by    L_RETFLAG, L_LINESTAT

define sma  min                    define sma  ext
select      min(L_SHIPDATE)        select      sum(L_EXTENDEDPRICE)
from        L_LINEITEM             from        L_LINEITEM
                                   group by    L_RETFLAG, L_LINESTAT

define sma  count                  define sma  extdis
select      count(*)               select      sum(EXTPRICE * (1-DIS))
from        L_LINEITEM             from        L_LINEITEM
group by    L_RETFLAG, L_LINESTAT  group by    L_RETFLAG, L_LINESTAT

define sma  qty                    define sma  extdistax
select      sum(L_QUANTITY)        select      sum(EXTPRICE * (1-DIS)
from        L_LINEITEM                         * (1+TAX))
group by    L_RETFLAG, L_LINESTAT  from        L_LINEITEM
                                   group by    L_RETFLAG, L_LINESTAT
```

Figure 4: The SMAs needed for Query 1 of TPC-D

Of course, these aggregates are specifically tailored for Query 1. In this respect the situation is not different from computing a data cube for it. However, the data cube's definition must include all possible selection attributes within its grouping clause. Hence, if order dates, shipment dates, and receipt dates are of interest, they must be present in the group specification—resulting in higher storage requirements. If one is forgotten, the data cube is of no use anymore. Not so for SMAs, new SMAs can be easily added for new attributes of interest. Hence, they are much more flexible than data cubes. Further, they are more versatile. If another query with restrictions on any of the attributes aggregated in some SMA occures, the SMA can be used to more efficiently answer the query.

2.4 Performance

In this section we briefly report on some experiments highlighting the crucial questions concerning the performance of SMA-based query processing: space requirements, creation time and query processing time. Before we give the actual performance figures let us recall some basic properties of SMA-files that justify why a brief performance evaluation is sufficient. First note that SMA-file sizes are linear in the number of buckets. Further, exactly one bucket summary has to be computed for every bucket. Its computation is independent of other buckets. Hence, there is no problem in scaling SMAs to very large data warehouses. Since creation and query processing times are also linear in the number of buckets, it suffices to give the performance for a single sufficiently large database. We

do so by discussing the performance for TPC-D Query 1 at a database size of 1 GB, the smallest allowed size of the TPC-D benchmark. The reason is a lack of disk space at our institution.

In order to process this query, the eight SMA files given in the last section are needed. The creation times and space requirements are summarized in the following table[3]:

sma file	count	max	min	qty	dis	ext	extdis	extdistax
creation time	117s	116s	103s	104s	100s	101s	95s	99s
size	736p	184p	184p	1468p	1468p	1468p	1468p	1468p

For counts and dates, 4 bytes are needed. For all other aggregate values we used 8 bytes. The total space requirement of *all* SMA-files amounts to 8444 4 K-pages or 33.776 MB. In our system, the LINEITEM relation consumes 733.33 MB. Hence, the accumulated size of all SMAs is only about 4 % of the total space. This shows that though several SMA files are needed in order to answer a single query, they are still very space efficient. The creation time for every SMA (not only a single SMA-file) is less than 2 minutes. In comparison, a B^+ tree on shipdate (though of no use for Query 1) consumes about 230 MB. Its creation time is far beyond the 15 minutes needed to create all SMAs.

Next, we compare the space requirements of SMAs to the space requirements of a materialized data cube. For Query 1, 6 aggregates of 8 bytes are necessary. Hence, every entry in the data cube is 48 byte wide. For the two flags, 4 possibilities exist. Every date attribute of LINEITEM (L_SHIPDATE, L_COMMITDATE, L_RECEIPTDATE) has a range of seven years or 2556 days. Hence, for the data cube we get a total storage requirement [5, 18] of about

- 479.25 $KB = 2556^1 * 4 * 48$ B if only one date is used as a dimension,
- 1196.25 $MB = 2556^2 * 4 * 48$ B if two dates are added as dimensions, and
- 2985.95 $GB = 2556^3 * 4 * 48$ B if all three dates are added as dimensions.

Adding SMAs for the two missing dates would require an additional 17.34 MB amounting to a total of 51.12 MB of storage. Comparing 51.12 MB to 2985.95 GB, the low storage overhead of SMAs compared to materialized data cubes becomes manifest.

For query response time, two aspects are of interest. First, the optimal case, that is when the relation is sorted on the restricted attribute. If LINEITEM is sorted on shipdate, the query processing time is

Query 1	without SMAs (cold & warm)	with SMAs (cold)	with SMAs (warm)
	128s	4.9s	1.9s

Processing Query 1 with SMAs becomes two orders of magnitude faster!

This is the optimal case. The question remains what happens if the number of ambivalent buckets grows. This question is answered by Figure 5. The x-axis shows the percentage of buckets that have to be investigated and the runtime of

[3]Measured on a Sparc Ultra I, 167 Mhz, with two Barracuda 4GB disks, running Solaris 5.5 using our AODB data warehouse management system configured at 8MB intertransaction buffer, 1MB intratransaction buffer.

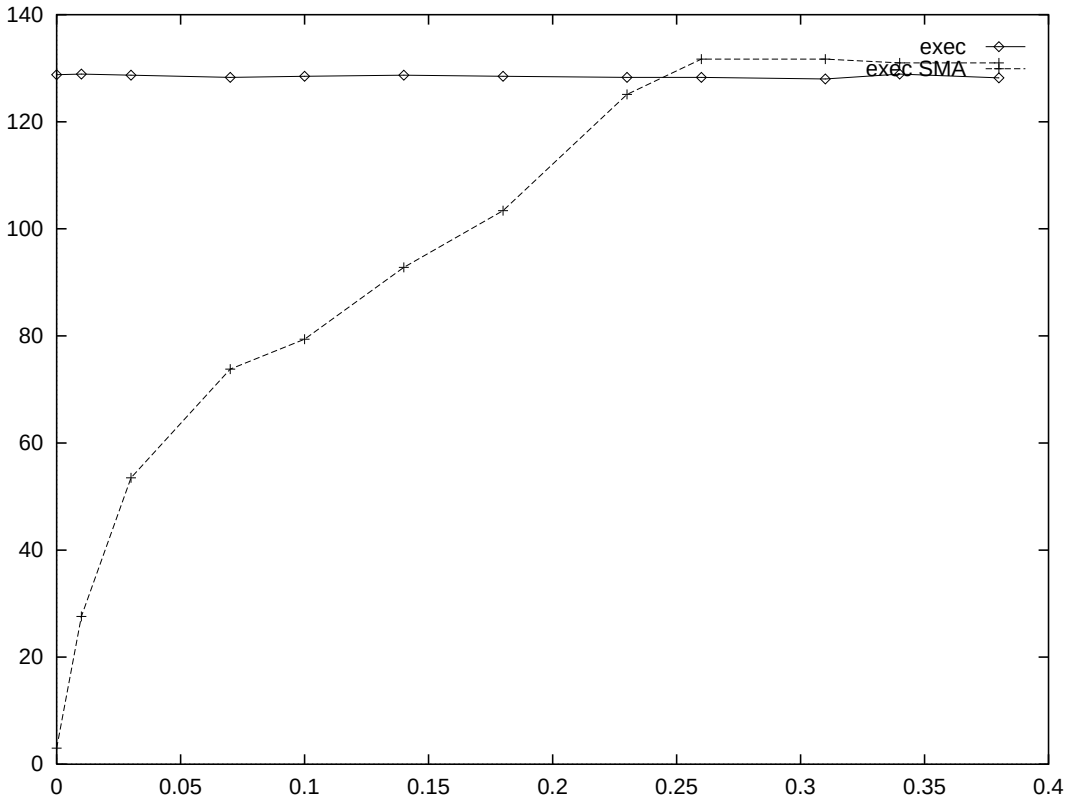

Figure 5: Runtime dependent on percentage of buckets to be processed

1. Query 1 without SMAs
2. Query 1 with SMAs (warm)

The breakeven point is at about 25% of the total number of buckets. That is, if more than 25% of all buckets are ambivalent and hence have to be accessed, then SMAs don't pay anymore. However, even if SMAs are erraneously applied—e.g. due to a bad decision of the query optimizer—the overhead remains small with less than 2% of the total run time.

3 Query Processing

This section discusses the problem of how to exploit SMAs for query processing. Since they differ considerably from traditional index structures, the generated plans will look different. As a side effect of discussing plan generation, the versatility of SMAs will become clear: SMAs can be exploited in many ways. This is contrary to data cubes.

We attack the problem of query plans in three steps. First, we partition all the buckets of a relation R into three sets: qualifying buckets, disqualifying buckets, and ambivalent buckets. This partitioning will then be summarized into a procedure *grade*. The

procedure *grade* will then be used within two new algebraic operators *SMA_Scan* and *SMA_GAggr* which the query processor can apply for evaluating queries involving selections and grouping with aggregates.

3.1 Partitioning the Buckets of a Relation R

Let us start by considering atomic selection predicates. They can be of different form:

- $A = c$
- $A \leq c$ $(A < c)$
- $A \geq c$ $(A > c)$
- $A \leq B$ $(A < B)$

where A and B are attributes of a single relation R and c is a constant. The first goal is to divide the buckets of R into qualifying, disqualifying and ambivalent buckets. This will be done for a single atomic selection predicate, and a single SMA. This information can then be used in order to evaluate more complex selection predicates involving *and* and *or* operations. These boolean connectives can also be used if more than a single SMA can be exploited.

Let BU denote all the buckets of relation R, BU_i the i-th buckets of relation R. Given a SMA $max(A)$, $max_i(A)$ denotes the maximum of all values of attribute A found in bucket BU_i, analogously, $min_i(A)$ denotes the minimum value. Given an atomic predicate, a single SMA, and a bucket i, we can assess bucket BU_i as follows:

- For $A = c$
  - if $c < min_i(A)$ then $BU_i \in BU_d$
  - if $c > max_i(A)$ then $BU_i \in BU_d$
  - else $BU_i \in BU_a$
- For $A \leq c$
  - if $max_i(A) \leq c$ then $BU_i \in BU_q$
  - if $min_i(A) > c$ then $BU_i \in BU_d$
  - else $BU_i \in BU_a$
- For $A \geq c$
  - if $min_i(A) \geq c$ then $BU_i \in BU_q$
  - if $max_i(A) < c$ then $BU_i \in BU_d$
  - else $BU_i \in BU_a$
- $A \leq B$

- if $max_i(A) \leq min_i(B)$ then $BU_i \in BU_q$
- if $min_i(A) > max_i(B)$ then $BU_i \in BU_d$
- else $BU_i \in BU_a$

where BU_q denotes the qualifying buckets, BU_d denotes the disqualifying buckets and BU_a denotes the ambivalent buckets. The *else* case is also applied if the max/min aggregates are not defined. The correctness of the above rules should be obvious.

Having two partitionings BU_q^1, BU_d^1, BU_a^1 and BU_q^2, BU_d^2, BU_a^2 for a some predicate/SMA combination, we can compute the partitioning if the two combinations are conjunctively or disjunctively connected:

and

$$\begin{aligned} BU_q &= BU_q^1 \cap BU_q^2 \\ BU_d &= BU_d^1 \cup BU_d^2 \\ BU_a &= BU \setminus (BU_q \cup BU_d) \end{aligned}$$

or

$$\begin{aligned} BU_q &= BU_q^1 \cup BU_q^2 \\ BU_d &= BU_d^1 \cap BU_d^2 \\ BU_a &= BU \setminus (BU_q \cup BU_d) \end{aligned}$$

SMAs with *min* and *max* aggregates can also be exploited for the evaluation of selection predicates if their definitions contain a **group by** clause. Consider SMA definitions of the following form:

define sma name
select max(A)
from R
group by $B_1, \ldots, B_n$

The rules to derive a partitioning of BU are similar to those stated above except that we have to consider the maximum value of A for all groups. The case for *min* is analogous.

But not only *min* and *max* aggregates are useful for selection predicates. If A is the only grouping attribute in a *count* SMA, like in

define sma name
select count(*)
from R
group by A

then we can use this information to evaluate selection predicates on A. Let $count_{A,i}[x]$ denote the number of tuples in bucket i exhibiting a value x for attribute A. Then we can partition BU by the following rules. For every possible value of x, we compute a partitioning BU^x as follows:

- For $A = c$
  - if $x = c$ and $count_{A,i}[x] > 0$ then $BU_i \in BU_q^x$
  - else $BU_i \in BU_d^x$
- For $A \leq c$
  - if $x \leq c$ and $count_{A,i}[x] > 0$ then $BU_i \in BU_q^x$
  - else $BU_i \in BU_d^x$
- For $A \geq c$
  - if $x \geq c$ and $count_{A,i}[x] > 0$ then $BU_i \in BU_q^x$
  - else $BU_i \in BU_d^x$

We then integrate the partitions BU^x into a single partitioning of BU by applying the following rules:

$$\begin{aligned} BU_q &= \bigcap_x BU_q^x \\ BU_d &= \bigcap_x BU_d^x \\ BU_a &= BU \setminus (BU_q \cup BU_d) \end{aligned}$$

Summarizing, whenever we have a selection predicate involving an attribute A of a relation R and a SMA-definition in which A occurs, we can compute a partitioning of the buckets of relation R into qualifying, disqualifying and ambivalent buckets. Let us integrate this procedure into a function *grade* that for a given bucket and predicate returns *qualifies*, *disqualifies* or *ambivalent*. This function will be used within the next two algegebraic operators implementing a *SMA-Scan* and a *SMA-GAggr* exploiting SMAs.

3.2 SMA-Scan

The *SMA_Scan* operator is an operator of the physical algebra and implements the iterator concept [7]. The three parameters of the iterator are the relation R to be scanned, the predicate to be evaluated on its tuples and a set of SMAs useful for partitioning the buckets of R.

The iterator is implemented as a class and provides an *init* procedure that initializes the internal data structures and computes the number of the first qualifying or ambivalent bucket. Additionally, the bucket is fetched from disk. This is summarized in a subroutine *getBucket*. Successive calls to the function *next* then return pointers to qualifying tuples. A tuple qualifies if it is in a qualifying bucket or if the predicate applied to the tuples yields true. The pseudo code of *SMA_Scan* is given in Figure 6.

```
class SMA_Scan {
  SMA_Scan(R,pred,smas);

  init() {
    currBucketNo = -1; getBucket();
  }

  Tuple* next() {
    while(buckets left) {
     if(there is an unseen tuple in bucket) {
        get this tuple;
        if(currGrade == qualifies)
          return tuple;
        else if (pred(tuple))
          return tuple;
     else
       getBucket();
    }
  }

  getBucket() {
    do {
      advance currBucketNo; advance all smas;
      currGrade = grade(currBucketNo, pred);
    }
    while(currGrade != qualifies and currGrade != ambivalent)
    read bucket currBucketNo;
  }
};
```

Figure 6: The SMA-Scan Iterator

3.3 SMA-GAggr

The *SMA-GAggr* operator computes the *GAggr* operator of Dayal [4] in the presence of SMAs. The *GAggr* operator performs a grouping together with the computation of aggregates. The *SMA-GAggr* uses some SMAs—called selection SMAs—for selecting qualifying buckets and tuples. Hence, it encompasses the *SMA-Scan* operator. However, more aggregates—the aggregate SMAs—are used to compute the queried aggregates. For qualifying buckets, the aggregate values are readily available within the aggregate SMAs. Ambivalent buckets must be inspected explicitly and the tuples must be grouped in order to compute the aggregate values. As for the *SMA_Scan* operator, the *SMA-GAggr* operator scans the relation and all SMAs in parallel.

The computation of the aggregates is performed in three phases in a rather standard manner. For every group, a tuple wide enough to hold all the result aggregates is allocated. If the result aggregates do not contain a *count(*)* and if averages are demanded by the query, we add it. The aggregate values are initialized by 0 for *sum*, *count*, and *avg* aggregates. For the latter, we first compute the sum and devide by the count in the last phase. For *min* and *max* aggregates, the minimum and maximum value are used for initialization. In the second phase, for every bucket these values are then advanced in the obvious way. For example, for the *sum* aggregate the aggregate value of some qualifying bucket is added. For ambivalent buckets, the according value is added for each tuple contained in it. In the last phase, we devide the sums which should be averages by the computed count.

The *SMA-GAggr* is a pipeline breaker. Within its *init* function, the result is computed. The *next* function then merely returns one result after another. The pseudocode of *SMA-GAggr* can be found in Figure 7.

4 Tuning Possibilities

There are several tuning possibilities to further enhance the performance of SMAs. The first obvious tuning measure is the bucket size. Here, the following trade off must be investigated. If the bucket size is small, then the SMA-files will become very large and more I/O for SMAs is the consequence. If the bucket sizes are large, then—due to imperfect clustering—many ambivalent buckets occur and for these the original relation must be accessed. Note that bucket sizes below a page size do not make sense.

This trade off can be mitigated by using hierarchical SMAs. Every SMA-file is again partitioned into buckets and for each bucket a second level SMA is computed. The advantage is that even for imperfectly clustered relations, the second level SMA is useful for rather high and rather low selectivities. If a second level bucket qualifies or disqualifies, the first level SMA-file need not to have to be accessed, which saves some I/O. If the second level bucket is ambivalent, then the first level SMA-file can be exploited to inspect the situation at a finer grain. Since second level SMA-files will be very small we do not think that higher levels are useful. Also we think it is preferable to switch to hierarchical SMAs instead of increasing the bucket size.

```
class SMA_GAggr {
   SMA_GAggr(R, pred, aggregateSpec, groupSpec, selectionSMAs, aggregateSMAs);

   init(const) { /* computes the result */
     forall(bucket in buckets) {
        switch(grade(bucket, pred)) {
          case qualifies: advance the result aggregates using the
                          aggregate SMAs;
          case disqualifies: do nothing
          case ambivalent: advance aggregates by inspecting the
                           tuples within the bucket;
        }
     }
     perform post processing for average aggregates;
   }

   Tuple* next() {return next unseen group;}
}
```

Figure 7: The SMA_Group Iterator

A last possibility to further enhance performance by SMAs is to generalize SMAs to encompass semi-joins. To see this, consider queries containing the following pattern:

select R.*
from R, S
where R.A θ S.B

where θ is a comparison operator. If we can associate a min/max value of the $S.B$ values with each bucket of R, SMAs can be used to decrease the input to the semi-join.

5 Conclusion

We introduced SMAs as an alternative to data cubes. Unlike data cubes, SMAs are more versatile to exploit in several kinds of queries. Performancewise, SMAs accellerate query execution by two orders of magnitude. Further, they are proofed to be very space efficient compared to data cubes, when the number of dimensions grows.

Some enhancements to SMAs were briefly discussed. Among them hierarchical SMAs and SMAs encompassing semi-joins. We plan further investigations on these and possibly other variations of SMAs.

References

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(4):290–306, 1972.

[2] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *Sigmod Record*, 26(1):65–74, 1997.

[3] D. Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, 1979.

[4] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *VLDB*, pages 197–208, 1987.

[5] P. Deshpande, J. Naughton, K. Ramasamy, A. Shukla, K. Tufte, and Y. Zhao. Cubing algorithms, storage estimation, and storage and processing alternatives for OLAP. *IEEE Data Engineering Bulletin*, 20(1):3–11, 1997.

[6] R. Fagin, J. Nievergelt, N. Pippenger, and H. Strong. Extendible hashing – a fast access method for dynamic files. *ACM Trans. on Database Systems*, 4(3):315–344, 1979.

[7] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), June 1993.

[8] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A relational aggregation operator generlizing group-by, cross-tab, and sub-totals. In *Proc. IEEE Conference on Data Engineering*, pages 152–169, 1996.

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 47–57, 1984.

[10] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 205–216, 1996.

[11] W. H. Inmon. *Building the Data Warehouse (2nd ed.)*. John Wiley & Sons, 1996.

[12] T. Johnson and D. Shasha. Some approaches to index design for cube forests. *IEEE Data Engineering Bulletin*, 20(1):27–35, 1997.

[13] R. Kimball. *The Data Warehouse Toolkit*. John Wiley & Sons, 1996.

[14] V. Y. Lum. Multi-attribute retrieval with combined indexes. *Communications of the ACM*, 13:660–665, 1970.

[15] P. O'Neil. Model 204 architecture and performance. In *2nd Int. Workshop on High Performance Transaction Systems*, pages 40–59, Pacific Grove, CA, 1987. LNCS 359, Springer.

[16] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, page to appear, 1997.

[17] S. Sarawagi. Indexing OLAP data. *IEEE Data Engineering Bulletin*, 20(1):36–43, 1997.

[18] A. Shukla, P. Deshpande, J. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1996.

[19] Transaction Processing Council (TPC). TPC Benchmark D. http://www.tpc.org, 1995.