COMBI-Operator – Database Support for Data Mining Applications

Alexander Hinneburg and Dirk Habich
Martin-Luther-University of Halle
hinneburg, habich@informatik.uni-halle.de

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

Abstract

Database support for data mining has become an important research topic. Especially for large high-dimensional data volumes, comprehensive support from the database side is necessary. In this paper we identify the data intensive subproblem of aggregating high-dimensional data in all possible low-dimensional projections (for instance estimating low-dimensional histograms), which occurs in several established data mining techniques. Second, we show that existing OLAP SQL-extensions are insufficient for high-dimensional data and propose a new SQL-operator, which seamlessly fits into the set of existing OLAP GROUP BY operators. Third, we propose efficient implementations for the operator, which take the limited resources of main memory into account. We demonstrate on a number of real and synthetic data sets that for the identified subproblem our new implementations yield a large speedup (up to factor 10) over existing methods built in commercially available database systems.

1 Motivation

Due the flood of data stored in nowadays databases there is a strong need to find scalable methods to analyze large data volumes efficiently. In the last decade a number of useful data mining techniques were developed, including algorithms for building decision trees and finding clusterings using high-dimensional data. One of the main problem addressed is dealing with large data volumes which do not fit into the main memory.

From a database perspective we can ask how a database system can efficiently support data mining algorithms in a general way. Some approaches has been published in the recent years [24, 26] focusing mainly on classifiers. However, most important in our context is to identify general data intensive tasks, which are shared by different data mining algorithms.

In this paper we identify the data intensive subproblem of aggregating data in all possibles low-dimensional projections of high-dimensional data. This task occurs in several data mining algorithms, for example, the *HD-Eye*-system [12,13] – an advanced visual clustering system – needs to derive histograms in all two-dimensional projections of a high dimensional data space. Similar situations occur in mining correlations with images [4], also during the build phase of a decision tree, of baysian classifiers and in cases of association rule mining.

In our motivation we want to use the simple example of determining all low-dimensional histograms of an high-dimensional data space. The number of low-dimensional histograms with dimensionality k is $\binom{n}{k}$ for the dimensionality n of the high-dimensional data space. This can be a quite large number for instance for $n = 256$ and $k = 2$ we have to derive about $\binom{256}{2} = 32640$ two-dimensional histograms. Figure 1 demonstrates the explosion of grouping combinations for a given set of grouping columns. As the data volume may be very large a transfer possibly over the network to the data mining client has to be avoided. As a consequence we propose to perform the data intensive tasks directly within the database and transfer only aggregated data to the mining client.

For our histogram example the naive approach to use a database system would be to issue for each histogram a single SQL-statement to the database. However, this causes the database to read the same data multiple times from the secondary storage. Advanced database systems also offer the GROUPING SET operator, which allows to query the database for all his-

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

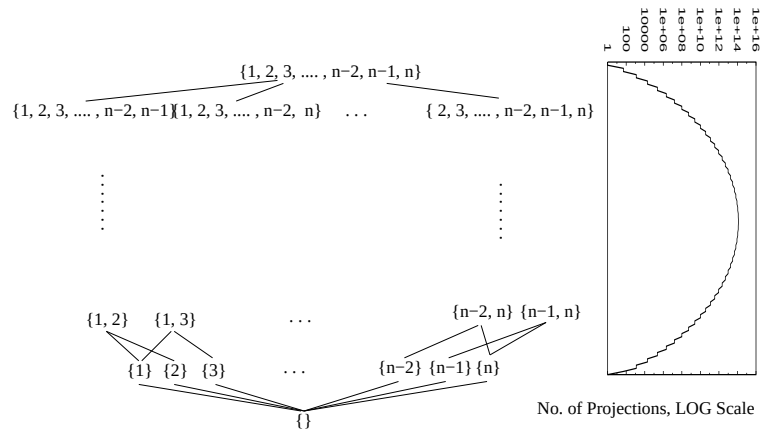


Figure 1: Explosion of the number of grouping combinations for n grouping attributes

tograms using a single SQL statement. The disadvantage here is that the histograms in all projections have to be enumerated in the query statement. A projection can be seen as a combination of attributes. The use of the GROUPING SET operator results into very large query statements, exceeding the parser capabilities of most database systems. The second disadvantage is that the used implementations for the GROUPING SET operator yields only a minimal performance speed up over the naive approach.

Our idea is to use a new SQL operator – GROUPING COMBINATION – to avoid the enumeration of attribute combinations used for the aggregation in the low-dimensional projections of the data space. As a result the query size will be reasonable small. We also propose algorithms to evaluate queries using the new operator, which determine during a single scan of the data as many histograms as possible fitting into the given amount of main memory. We will show that our new algorithms yield a tremendous speedup over current implementations.

In the reminder of the paper we discuss related work in section 2. In section 3 we describe the new operator and how it fits into the setting of other OLAP Group By operators. In section 4 we propose algorithms to evaluate the operator, which can deal with very large data volumes and in section 5 we demonstrate the performance of the algorithms. We conclude the paper with section 6.

2 Related Work

The problem of how to support data mining applications by a database system has been studied in three different contexts. The first context consists in supporting the tasks around data mining. One recent approach [18] supports data management tasks including transformations or handling of different data sources. The proposed OLE DB-DM architecture basically provides an environment to access data from different sources in a unified way, which is specific for data mining. Another somewhat complimentary approach by Chaudhuri et al. [6] is to support database queries, which use results coming out of a data mining algorithm.

In the second context researchers proposed data mining specific query languages, which assume the data mining algorithms (basically association rule mining) to be integrated within the database engine. For instance the M-SQL language [14] provides a special MINE operator to support association rule mining. The MINE RULE operator proposed in [17] is a similar example for finding generalized association rules. The Query flocks [25] provides a generate-and-test model to mine association rules.

Approaches in the third context provide database support for particular data mining algorithms by extending the database query language or making use of known multi-purpose extensions in order to reformulate the algorithms (whole or partially) in terms of the extended language. This is an very interesting way of tightly coupling the database with mining algorithms, which has been done for association rule mining [3, 23], clustering [19] and decision trees [24]. These examples for tight coupling make use of sophisticated SQL queries and several mechanisms for general database extensions like user defined table functions and table operators [15]. Another very useful possibility to extend the database is to provide user-defined aggregates [26]. The main disadvantage here is that the mining algorithm has to be written as user-defined functions or user-defined aggregates, which involves significant code rewrites. Also the capabilities of the database to optimize the user-defined code are very limited.

Complementary to user-defined functions is the approach of extending SQL by application specific operators. One of the advantages of extending SQL by special operators is the ability of the system to provide special optimization techniques. [10] demonstrates the motivation of the OLAP grouping extension by intro-

ducing CUBE and ROLLUP operators. Implementational perspectives are covered in [1, 21, 27] (see sec. 4.1 for details).

3 Operator Design

In this section, we outline the design of our proposed COMBI operator. In a first step, we show the usage of our operator and explain its syntax and semantics.

3.1 Overview of SQL Grouping Operators

Using a relational database in a transaction-oriented environment does not require any sophisticated aggregation functionality. However, discovering the database system as the central integration and analysis platform in the context of predefined statistical analyses or an Online Analytical Processing environment, simple grouping functionality could not satisfy the demand arising from those applications.

Basically two different directions were addressed from a language design and implementational point of view. On the one hand, sequence processing was enhanced in the SELECT clause by introducing the OVER clause with column-wise ordering, partitioning and dynamic windowing. On the other hand, complex grouping was enabled by introducing CUBE and ROLLUP operators [10].

The CUBE operator generates all possible grouping combinations for a given set of grouping columns resulting in 2^n different grouping combinations for n grouping columns. The ROLLUP operator additionally considers functional dependencies between grouping columns (in case of a classification hierarchy) and yields $n+1$ grouping combinations for a list of n grouping columns. The GROUPING SETS operator – recently added to standard SQL[1] – finally provides a basis to explicitly specify all grouping combinations. Obviously all combinations are related to each other. For example, a data cube of height 1 with two grouping attributes $A1$ and $A2$ may be expressed as follows:

```
CUBE(A1,A2) = ROLLUP(A1), ROLLUP(A2)
            = GROUPING SETS((),(A1)),
              GROUPING SETS((),(A2)),
            = GROUPING SETS((A1,A2),(A1),(A2),())
```

3.2 Computing Projections by Exploiting Grouping Extensions

Aggregation of k-dimensional projections of n-dimensional data sets with $k \ll n$ is an important operations in the context of several data mining applications [2,4,9,12,20]. A common task of the mentioned methods is to derive multi-dimensional histograms in low-dimensional subspaces. For subspaces of dimensionality k all attribute combinations of size $\binom{n}{k}$ attributes have to be considered. With the number of attributes reasonably low, we may use the GROUPING SETS clause to specify the necessary combinations within a single SQL query. For our running example we assume that we have four already discretized attributes (this means the continuous attributes are binned into discrete buckets) A_1, A_2, A_3, and A_4 and we may issue the following statement to compute bin counts for the histograms in all six two-dimensional subspaces:

```
SELECT A1, A2, A3, A4, COUNT(*) AS CNT
  FROM ...
GROUP BY GROUPING SETS (
            (A1, A2), (A1, A3), (A1, A4),
            (A2, A3), (A2, A4), (A3, A4))
```

In case of high-dimensional data, the specification of all possible combinations leads to very large query strings. In general the query size grows in this case like $\binom{n}{k}$, which exceeds the parser capabilities of today's database systems in case of typical data like images, music or documents with hundreds of attributes.

Another alternative would be to issue a CUBE operator over the given set of attributes and eliminate the unncessary grouping combinations in a HAVING clause. For the ongoing example with four attributes, we would yield the following expression:

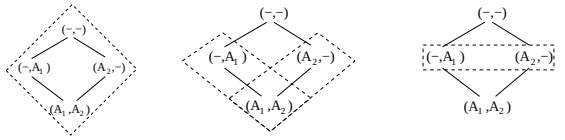
```
SELECT A1, A2, A3, A4, COUNT(*) AS CNT
  FROM ...
GROUP BY CUBE(A1, A2, A3, A4)
  HAVING NOT(
   -- the 1-combination ...
   (GROUPING(A1) = 1 AND GROUPING(A2) = 1 AND
    GROUPING(A3) = 1 AND GROUPING(A4) = 1)
   -- all 3-combinations ...
OR (GROUPING(A1) = 1 AND GROUPING(A2) = 1
                     AND GROUPING(A3) = 1)
OR   ...
   -- the 4-combination ...
OR (GROUPING(A1) = 0 AND GROUPING(A2) = 0 AND
    GROUPING(A3) = 0 AND GROUPING(A4) = 0))
```

Although this would return the required two-dimensional histograms, the queries size – in particular the HAVING clause – grows in this case with the size of the subset lattice minus the size of the k-level with the required combinations. In this setting the query length grows exponentially in the number of used attributes, which also excludes an efficient handling of the query string by the database SQL parser.

3.3 The COMBI Operator

Relying only on existing grouping functionality does not yield a viable solution to position a database system as a solid platform for efficient statistical analyses. Moreover, considering the set of grouping extensions, we strongly believe that an operator returning all combinations of the same cardinality within an aggregation lattice is a seamless extension to the

[1] see ISO/IEC 9075-1:1999/Amd 1:2001 at http://www.iso.ch



CUBE(A_1,A_2) RU(A_1),RU(A_2) COMBI((A_1,A_2),1)

Figure 2: Relationship of CUBE, ROLLUP, and GROUPING COMBINATIONS

family of complex grouping operators. Figure 2 illustrates the effect of the COMBI operator compared to CUBE/ROLLUP.

First of all, a CUBE operator returns all possible combinations, which is (a) not feasible for a large number of attributes and (b) implies that most of the resulting combinations are eliminated afterwards. Hierarchical data cubes in general are composed by multiple ROLLUP operators, each for a single dimension, so that the ROLLUP operator returns the vertical direction of an aggregation lattice. In the same vein, the GROUPING COMBINATIONS operator (aka. COMBI operator for short) returns the horizontal direction of an aggregation lattice.

To summarize, the GROUPING COMBINATIONS operator returns all $\binom{n}{k}$ grouping combinations for a given set of n grouping columns of cardinality k. The syntax integrates seamlessly into the SQL grouping extensions of CUBE, ROLLUP, and GROUPING SETS:

```
GROUP BY GROUPING
         COMBINATIONS((A1,A2, ... ,An), k)
```

This expression may be seen as a shortcut for the GROUPING SETS expression with $\binom{n}{k}$ combinations each with a cardinality of k, i.e.:

```
GROUP BY GROUPING SETS((A1,A2,...,Ak),
                       (A1,A2,...,Ak-1,Ak+1),
                       ...
                      ))
```

The first parameter of the GROUPING COMBINATIONS expression consists of a single grouping combination. The second parameter k denotes the cardinality of the combinations. Obviously the value of k is supposed to range from 0 to n with special cases discussed in the following subsection.

3.4 Additional operator syntax and semantics

In the context of the COMBI operator, it is worth mentioning that the semantics of the GROUPING function applies as well to the GROUPING COMBINATIONS operator. The GROUPING function denotes for every single value of a tuple of the resulting table whether the corresponding entry is a system generated NULL value (denoting a super aggregate value) or a user-defined value. For example consider the following query executed on a sample data set with a GROUPING function defined for A_1:

```
SELECT A1, A2, A3, A4
       GROUPING(A1) AS GRP1,
       GROUPING(A2) AS GRP2,
       GROUPING(A3) AS GRP3,
       COUNT(*) AS CNT
  FROM ...
GROUP BY GROUPING
         COMBINATIONS((A1,A2,A3,A4), 2)
```

A1	A2	A3	A4	GRP1	GRP2	GRP3	CNT
a1_1	a2_1	NULL	NULL	0	0	1	17
a1_2	a2_1	NULL	NULL	0	0	1	11
...							
NULL	a2_1	a3_1	NULL	1	0	0	23

As the resulting table may have many NULL values, we propose an alternative way of formating the result to code the position of the groups in additional rows declared by a new GROUPINGPOS function which avoids the NULL values. An example for the syntax and the results are shown below:

```
SELECT C1,C2
       GROUPINGPOS(C1) AS GRPPOS1,
       GROUPINGPOS(C2) AS GRPPOS2,
       COUNT(*) AS CNT
  FROM ...
GROUP BY GROUPING
         COMBINATIONS((A1,A2,A3,A4), (C1,C2))
```

C1	C2	GRPPOS1	GRPPOS2	CNT
a1_1	a2_1	1	2	17
a1_2	a2_1	1	2	11
...				
a2_1	a3_1	2	3	23

For the second syntax style the number of result columns is only of the order the cardinality of the combination size k instead of the number of attributes n. Also the huge amount of NULL values does not occur. To avoid ambiguous parameter settings the size of the combinations is coded by the number of result columns (in the example C_1 and C_2), which are enclosed by the second bracket block.

Additionally, special cases have to be considered regarding the cardinality of the grouping combinations k and the number of attributes n.

- For $k = n$, the GROUPING COMBINATIONS operator returns a single set with all grouping columns given as the first parameter, e.g.:

  ```
  GROUP BY GROUPING
           COMBINATIONS((A1,A2, ..., An), n)
  =
  GROUP BY GROUPING SETS((A1,A2,...,An))
  =
  GROUP BY A1,A2,...,An
  ```

- For $k = 1$, the operator corresponds semantically to a grouping set with n single sets, each consisting of exactly a single grouping column. More exactly:

```
GROUP BY GROUPING
        COMBINATIONS((A1,A2, ...,An), 1)
=
GROUP BY GROUPING SETS((A1),(A2),...,(An))
```

- For $k = 0$, the proposed GROUPING COMBINATIONS operator produces a single grouping combination encompassing all tuples of the underlying table, e.g.

  ```
  GROUP BY GROUPING
          COMBINATIONS((A1,A2, ...,An), 0)
  ```

  is equal to a query without any explicit GROUP BY clause but an aggregation function in the SELECT clause.

To put it into a nutshell from a language point of view, the aggregation within subspaces of high-dimensional data is an important task within a huge variety of data mining applications. Unfortunately, the currently existing SQL language constructs are not well suited to fulfill those requirements. The proposed GROUPING COMBINATIONS operator extends the set of GROUP BY operators designed especially for the OLAP context and provides a uniform and flexible meaning to support such subspace aggregations.

4 COMBI Operator Implementation

Within this subsection, we focus on the implementational perspective of the operator design and demonstrate different methods to compute the required grouping combinations. In a first step, we review existing algorithms used to support the computation of general grouping conditions. Thereafter we illustrate the main memory and the sortorder based computation of the COMBI operator.

4.1 Implementational Perspectives of OLAP Operators

The introduction of the CUBE operator triggered a huge variety of implementations. Since all optimization strategies apply to the other members of the grouping operators as well, we simply refer to the CUBE operator attracting the main attention within the research community. The naive implementation of a CUBE operator basically consists of the multiple execution of the query, each run computing a different grouping combination. Obviously this approach comes close to the method of holding the information at the client and calling the database to compute a single grouping combination with a simple SQL statement.

The main idea of rewrite optimization strategies is based on subset stacking. Using this strategy, a grouping combination can be computed from the result of a preceding group by operation, if the current group by operator is restricted to a subset of grouping columns. The problem is to pick the optimal source for a given group by combination according to smallest parent (minimal data transfer) and sharing sort orders (avoiding additional sort operators).

A greedy based algorithm considering only the local decisions between two levels of an aggregation lattice is given in [1]. An optimal method in the number of sort operations is given in [21]. The same idea is used to efficiently compute iceberg cubes [8], i.e. data cubes with a HAVING clause. The grouping combinations are computed in a bottom up manner so that grouping combinations may be eliminated as early as possible if the predicate can not be satisfied for higher dimensional data cubes [5], [11].

The way of supporting complex grouping operators inside the database consists in providing a special physical plan operator or special index structures to boost the computation process. Representatives of the second alternatives may be seen in cube trees [22] and cube forests [16]. For our implementation, the first alternative to implement a completely new operator is intensively discussed in [7] and [27] within the context of the CUBE operator. The most similar approach to our implementation of the COMBI operator is discussed in [27]. In a first step, the data stream is split into chunks of the size of the main memory. In a second step, the result of the CUBE operator is computed step-by-step for each chunk. In opposite to the COMBI operator, the primary goal of this approach is to find the optimal sort order to benefit from subset stacking, which is not possible for the COMBI operator.

4.2 Main Memory Based Implementation

The main requirement regarding the implementations of the COMBI operator is to compute as many grouping sets as possible during a single scan of the data. The intermediate results of the different grouping sets reside in main memory during the scan. In principle there are two possibilities to organize the results in the main memory. First, if we can compute an upper bound of the number of groups of a specific grouping set, we could store the intermediate results of this grouping set in an array with constant access time. For the histogram example we represent all possible multidimensional grid cells in an array. However, we have to pay for the constant access time with the wasted memory resulting from unused grid cells (grid cells with no data point in).

The second possibility is to store only the needed intermediate results of the different grouping sets in a hash map container. Alternatively for the hash map also a search tree structure could be used. In terms of the histograms this means that only those grid cells are stored, which contain at least one single data point. The efficient use of memory is payed by an approximated constant or logarithmic access time.

The main memory based algorithm is the most I/O efficient algorithm among our implementation for the

Algorithm 1 Main memory based algorithm for the COMBI operator.

Require: Relation $R(A_1, \ldots, A_n), k$ with $0 \leq k \leq n$, ContainerType=hash map or array

```
1: if ContainerType=hash map then
2:    C:= initialize the data structures for all grouping combinations (A_1,...,A_n) of size k with container hash map
3: else
4:    C:= initialize the data structures for all grouping combinations (A_1,...,A_n) of size k with container array
5: end if
6: for all tuple t ∈ R do
7:    for all Grouping Combinations c ∈ C do
8:       add the projection c.P(t) to c.container
9:    end for
10: end for
11: Return C
```

COMBI operator, because only one single table scan of the data table is needed. However, the number of grouping combinations may be to large to fit into the main memory at the same time. To handle the problem that not all grouping combinations fit into the main memory, we suggest to consider only a subset of them during a single scan of the data table and to apply multiple scans. This strategy however (similar to the approach of [27]) implies that the set of grouping combinations is split into partitions resulting in a minimal number of scans. For example, consider for grouping columnns A,B,C, and D with $|A| = 5$, $|B| = 4$, $|C| = 3$, $|D| = 2$ and a space constraint of 20 units. All six 2-combinations could be computed within 4 scans if the partioning scheme applies to {AB},{AC},{AD,CD},{BC,BD}. The size of the available main memory can be given as a parameter of the algorithm. For simplicity, we rely on the number p, $1 \leq p \leq \binom{n}{k}$ of grouping combinations, which are determined during a scan of the data table.

The partitioned algorithm is an efficient extension of the main memory algorithm to deal with a limited amount of memory. Good choices for the parameter p depending on the data distribution can be found with sampling from the data table or using statistics of the data table.

4.3 Sort Based Implementation

While the above algorithm implements a partitioning scheme on a grouping combination basis, the following approach exploits the existence of an ordering scheme of the underlying data and reduces the overhead needed for a complete re-scan. For example, for three grouping columns A, B, and C with a space constraint of one single 2-combination, the above approach may decide to compute the AB-combination in a first scan, followed by AC and BC combinations

Algorithm 2 Partitioned algorithm for the COMBI operator.

Require: Relation $R(A_1, \ldots, A_n), k$ with $0 \leq k \leq n$, ContainerType=hash map or array, p with $1 \leq p \leq \binom{n}{k}$

```
1: for i = 0; i < ⌈(n k)/p⌉; i++ do
2:    if ContainerType=hash map then
3:       C:= initialize the data structures for p grouping combinations of size k with container hash map
4:    else
5:       C:= initialize the data structures for p grouping combinations of size k with container array
6:    end if
7:    for all tuple t ∈ R do
8:       for all Grouping Combinations c ∈ C do
9:          add the projection c.P(t) to c.container
10:      end for
11:   end for
12:   write out C
13: end for
```

in the following scans. The total costs accumulate to three full database scans. In the opposite, the order-based implementation utilizes the ordering scheme of the underlying data insofar as all combinations are simultaneously computed as much as possible.

The following example with grouping columns A, B, C and a cardinality of $|A| = 4$, $|B| = 3$, and $|C| = 2$ evaluating the SQL expression GROUPING COMBINATIONS$((A, B, C), 2)$ yields the following trace. For each record of the underlying data, the trace shows the corresponding action ($S()$ implies storing the value of the combination in main memory and increasing the number of required storage units, $W()$ denotes writing the value to the output and releasing the allocated main memory storage space).

```
     A   B   C      Action    Storage
=====================================
 1: A1  B1  C1  ==> S(A1B1)
                    S(A1C1)
                    S(B1C1)       3
 2: A2  B1  C1  ==> S(A2B1)
                    S(A2C1)       5
 3: A3  B1  C1  ==> S(A3B1)
                    S(A3C1)       7
 4: A4  B1  C1  ==> S(A4B1)
                    S(A4C1)
                    W(B1C1)       8
 5: A1  B2  C1  ==> S(A1B2)
                    S(B2C1)      10
 6: A2  B2  C1  ==> S(A2B2)      11
 7: A3  B2  C1  ==> S(A3B2)      12
 8: A4  B2  C1  ==> S(A4B2)
                    W(B2C1)      12
 9: A1  B3  C1  ==> S(A1B3)
                    S(B3C1)      14
```

```
10: A2  B3  C1  ==> S(A2B3)      15
11: A3  B3  C1  ==> S(A3B3)      16
12: A4  B3  C1  ==> S(A4B3)
                    W(B3C1)
                    W(A1C1)
                    W(A2C1)
                    W(A3C1)
                    W(A4C1)      12
13: A1  B1  C2  ==> S(A1C2)
                    S(B1C2)
                    W(A1B1)      13
14: A2  B1  C2  ==> S(A2C2)
                    W(A2B1)      13
15: A3  B1  C2  ==> S(A3C2)
                    W(A3B1)      13
16: A4  B1  C2  ==> S(A4C2)
                    W(B1C2)
                    W(A4B1)      12
17: A1  B2  C2  ==> S(B2C2)
                    W(A1B2)      12
18: A2  B2  C2  ==> W(A2B2)      11
19: A3  B2  C2  ==> W(A3B2)      10
20: A4  B2  C2  ==> W(A4B2)
                    W(B2C2)       8
21: A1  B3  C2  ==> S(B3C2)
                    W(A1B3)
                    W(A1C2)       9
22: A2  B3  C2  ==> ...
23: A3  B3  C2  ==>
24: A4  B3  C2  ==>
```

From this sample trace, we may deduce the following observations as a basis for the proposed generic algorithm. First of all, the optimal ordering of n grouping columns requires with regards to the cardinality of the grouping columns that the condition $|A_i| \geq |A_i + k|$ for $k > 0$ and $1 \leq i \leq n$ holds. Every other ordering scheme may apply as well but reduces the savings gained by applying the algorithm (in fact, the approach outlined in previous subsection may be seen as an extreme variant without any ordering of the underlying data stream).

Secondly, the order-based processing implies a partitioning scheme of the grouping columns. More formally, if $A_1, ..., A_n$ reflect the list of all grouping columns, $A_1, ..., A_k$ are called *dependant grouping columns* with A_1 the most and A_k the least dependant column. $A_{k+1}, ..., A_n$ are called *invariant grouping columns*. In the above example, for lines 1 to 12, A and B are dependant columns while C is an invariant column, i.e. all grouping combinations with an invariant grouping column can be given to the output as soon as there is a change in the least dependant column B, e.g. *W(B_1C_1)* at line 4.

The basic algorithm of the sort based implementation is given in 3. In the first phase, all required grouping combinations $g(t)$ are considered for each single tuple t of the raw data. If main memory is already allocated for this combination, the histogram value will be updated ($S(g())$). Otherwise, a new slot holding the information is allocated. In the second phase, all combinations held within main memory are checked as a candidate of an early result production ($W(g())$ operation). Figure 3 shows the three different 2D-projections of the three grouping columns illustrating the sequence of the generation of the single grouping combinations.

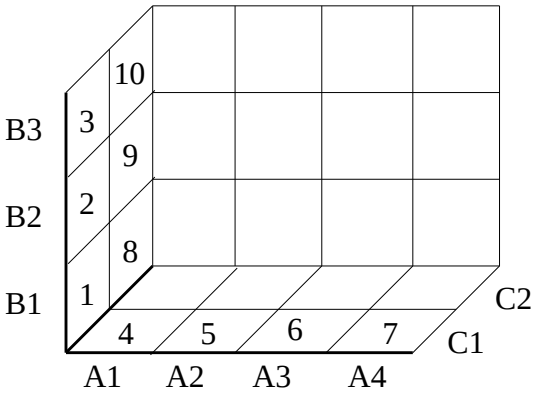


Figure 3: sequence of producing partial results

Algorithm 3 Sort based algorithm

Require: Relation R
{G: set of grouping combinations}
{M: main memory slots}
1: **for all** tuple $t \in R$ **do**
2: {Phase 1: compute all grouping combinations}
3: **for all** grouping combinations $g \in G$ **do**
4: **if** ($g(t)$ already allocated in main memory) **then**
5: update $S(g(t))$
6: **else**
7: allocate new $S(g(t))$
8: **end if**
9: **end for**
10: {Phase 2: check for partial results}
11: **for all** grouping combinations $g \in M$ **do**
12: **if** ($g()$ is ready for early output) **then**
13: write to output $W(g())$
14: **end if**
15: **end for**
16: **end for**

The benefit of the order-based implementation compared to previous algorithms consists in the advantage of a lower overall storage capacity. Compared to the main-memory implementation with a storage overhead of $(4*3)+(4*2)+(3*2) = 26$ for computing all three combinations simultaneously, the order-based implementation exhibits a storage capacity of only 16 units, because combinations of invariant grouping columns may be given to the output during the scan. Compared to the partioned implementation, the main advantage of the order-based approach is that with a restricted storage capacity, the potentially succeeding scans do

not have to start at the beginning but at the position where the storage overflow occurred in the preceding run. For example, with only 5 units main memory for the above sample scenario, 4 scans are required to starting at line 1,3, 6, and 11. The net cost (with 24 as the cost for scanning the full data set) compute to 79 lines = 3.3 scans. With a main memory capacity of 9 units, only a single rescan starting at line 5 yielding overall net cost of 1.8 scans. Table 1 compares the total costs for each the implementation with regard to the sample scenario.

The basic algorithm is depicted in algorithm 4. The extension regarding the simple sort based implementation consists of a mechanism controlling the necessity of a rescan using the boolean variable *reScanInitiated*. Moreover, the variable *reScanLine* records the current position for the start of the potentially next re-scan. From a database point of view, this intermediate scan requires the existence of an index based organisation of the raw data accoding to the sort order.

Algorithm 4 Partial ReScan algorithm

```
Require: Relation R
    {G: set of grouping combinations
     M: main memory slots}
    {First initialize parameters}
 1: reScanLine := currentLine := 1
 2: repeat
 3:    reScanInitiated := false
 4:    for all tuple t ∈ R starting at currentLine do
 5:       {Phase 1: compute all grouping combinations}
 6:       currentLine++
 7:       for all grouping combinations g ∈ G do
 8:          if (g(t) already allocated in main memory) then
 9:             update S(g(t))
10:          else if (enough memory available AND reScanInitiated == false) then
11:             allocate new S(g(t))
12:          else if (reScanInitiated == false) then
13:             reScanInitiated := true
14:             reScanLine := currentLine
15:          end if
16:       end for
17:       {Phase 2: check for partial results}
18:       for all grouping combinations g ∈ M do
19:          if (g() is ready for early output) then
20:             write to output W(g())
21:          end if
22:       end for
23:    end for
24:    currentLine := reScanLine
25: until reScanInitiated ≠ true
26: end
```

To summarize, the benefits of the sort-based implementation may be seen from two perspectives. On the one hand, the implementation requires less main memory compared to the hash/array based implementation and produces results as early as possible so that succeeding operators may work in a pipelined manner. On the other hand, tight memory restrictions (especially for high dimensional grouping combinations), imply that necessary rescans do not have to restart from the beginning of the raw data set.

5 Performance Analysis

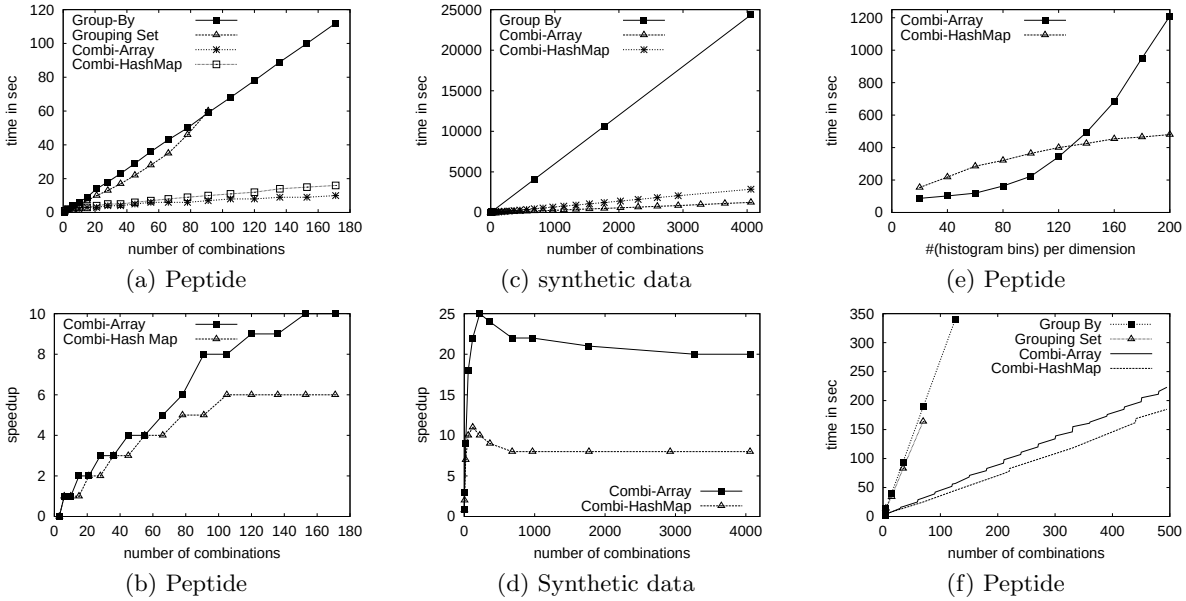
For our experiments we implemented a prototype of the Combi operator on top of the DB2 database system (V8.1) in C++. The prototype accesses the database via ODBC driver. As test data we used synthetic data consisting of a mixture of Gaussian and uniform distributions and a real data set, which comes from a simulation of a biological molecule. The synthetic data set has 100 attributes with 300000 data points (tuples) and the biology data set has 19 attributes with 100000 data points. In our current prototype we implemented the main memory based and the partition based algorithm with array and hash map containers. We conducted the experiments on a Linux Pentium with 512 MB RAM.

We used the computation of histograms in projections of high-dimensional spaces as example application, which occurs in many data mining algorithms as a data intensive subtask. In our first experiments we showed how the Combi operator algorithms scale up with an increasing number of grouping combinations each corresponding in our case to a histogram in a projection. We compared our algorithms firstly with the naive Group By approach, which determines each grouping combination using a separate SQL statement and secondly with the Grouping Sets operator.

Figure 4 shows the details of the comparison. In figure 4(a,b) we used the smaller data set and determined histograms in two-dimensional projections. The two-dimensional histograms are parameterized to have at most $20 \times 20 = 400$ bins, which are almost populated with data points. The Grouping Set operator showed a slightly better performance than the naive approach using Group By. But it converges with increasing number of grouping combinations towards the naive approach and due to internal restrictions of the database system it is not applicable for more than 100 grouping combinations. The speedup is determined with the minimal execution time of Group By and Grouping Sets. As the histograms are small and almost all bins are populated, the array based implementation shows a better performance than the hash map based implementation. As the histograms are small only a single scan of the data is needed. The largest performance gain over the existing database algorithm shows the array based implementation, which is ten times faster than Grouping Sets or Group By.

	MainMemory	Partitioned	SortBased	SortBased
Storage Requirment	4*3+4*2+3*2 = 26	max(4*3, 4*2, 3*2)	5	9
No of Scans	1	3	3.3	1.8

Table 1: Comparison of different COMBI-implementations



(a) Peptide (c) synthetic data (e) Peptide

(b) Peptide (d) Synthetic data (f) Peptide

Figure 4: Experimental comparision of the array and hash map based implementations for the COMBI operator with GROUPING SETS and GROUP BY.

In the second experiment, which is shown in the parts (c) and (d) of figure 4, we used the large synthetic data set (100 attributes, 300.000 tuples) and determined the histograms for a fraction of the set of three-dimensional projections of the 100 dimensional space. As mentioned above, the GROUPING SET operator is not applicable in this scenario, because of internal restrictions of the database system. As the size of the histograms is $40 \times 40 \times 40 = 16000$ grid cells, we used the partition-based algorithm, which allows the computation of 600 histograms during a single scan. This bound explains the bumps at the left side of the speedup curves, where the speedups are maximal. Also in this case the array-based implementation shows the best performance with an speedup of 22 over the otherwise only possible naive approach based on a number of GROUP BY statements.

The previous experiments may lead to the conclusion that the array-based variant outperforms the hash map based implementation in every case. However, this is only true, when not much memory of the array is wasted. In our example application this may happen, when the dimensionality of the projected histograms grows a little bit and/or the number of bins per dimension gets larger. In figure 4(e) we demonstrate that in the later case of larger cardinality of bins per dimension beyond a certain point the hash map-based implementation performs much better. The dimensionality of the projected histograms is fixed to three dimensions in this experiment. The other case is elaborated in the experiment shown in figure 4(f), where the projected histograms have a dimensionality of four with 40 bins per dimension. Due to the larger dimensionality there are several empty bins in the histograms which contain no data points and waste memory in the array based implementation. With the specific setting the array based approach can store and compute 30 histograms during a scan, while the hash map based algorithm can handle 220 histograms per scan. As a consequence the array based implementation have to perform much more scans – visible as small steps in the plots – as the hash map based one. However, both algorithms outperform also in this case the GROUPING SETS operator and the GROUP BY approach by magnitudes.

6 Conclusion

In this paper we identified aggregation in subspaces formed by combinations of attributes as an important task common to many data mining algorithms. In order to get a tight coupling of database and mining algorithm we exploited several existing formulations of the problem in SQL. The main drawbacks of the existing operators are (1) very large query size and/or (2) suboptimal performance. We proposed a

new operator fitting seamlessly into the set of OLAP GROUP BY extensions. We introduce the new operator at the language level with proper syntax and semantics and we provided several algorithms for the implementation. Experimentally we evaluated the pros and cons of different implementations and showed that our algorithms outperform existing operator implementations by magnitudes. We believe that our new operator could make a database system a good basis for many business intelligence applications especially in the more sophisticated data mining application area.

References

[1] Sameet Agarwal, Rakesh Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. On the computation of multidimensional aggregates. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 506–521. Morgan Kaufmann, 1996.

[2] Charu C. Aggarwal and Philip S. Yu. Finding generalized projected clusters in high dimensional spaces. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 70–81. ACM, 2000.

[3] Rakesh Agrawal and Kyuseok Shim. Developing tightly-coupled data mining applications on a relational database system. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*, pages 287–290. AAAI Press, 1996.

[4] Amihood Amir, Reuven Kashi, and Nathan S. Netanyahu. Analyzing quantitative databases: Image is everything. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 89–98. Morgan Kaufmann, 2001.

[5] Kevin S. Beyer and Raghu Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 359–370. ACM Press, 1999.

[6] Surajit Chaudhuri, Vivek R. Narasayya, and Sunita Sarawagi. Efficient evaluation of queries with mining predicates. In *Proceedings of the 18th International Conference on Data Engineering*. IEEE Computer Society, 2002.

[7] Prasad Deshpande, Jeffrey F. Naughton, Karthikeyan Ramasamy, Amit Shukla, Kristin Tufte, and Yihong Zhao. Cubing algorithms, storage estimation, and storage and processing alternatives for olap. *Data Engineering Bulletin*, 20(1):3–11, 1997.

[8] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing iceberg queries efficiently. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 299–310. Morgan Kaufmann, 1998.

[9] Johannes Gehrke, Raghu Ramakrishnan, and Venkatesh Ganti. Rainforest - a framework for fast decision tree construction of large datasets. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 416–427, 1998.

[10] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 152–159. IEEE Computer Society, 1996.

[11] Jiawei Han, Jian Pei, Guozhu Dong, and Ke Wang. Efficient computation of iceberg cubes with complex measures. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 1–12. ACM Press, 2001.

[12] Alexander Hinneburg and Daniel A. Keim. Optimal grid-clustering: Towards breaking the curse of dimensionality in high-dimensional clustering. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 506–517. Morgan Kaufmann, 1999.

[13] Alexander Hinneburg, Daniel A. Keim, and Markus Wawryniuk. Hdeye: Visual mining of high-dimensional data (demo). In *SIGMOD 2002, Proceedings ACM SIGMOD International Conference on Management of Data, June 3-6, 2002, USA*. ACM Press, 2002.

[14] Tomasz Imielinski, Aashu Virmani, and Amin Abdulghani. DataMine: application programming interface and query language for database mining. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)*, pages 256–262. AAAI Press, 1996.

[15] Michael Jaedicke and Bernhard Mitschang. User-defined table operators: Enhancing extensibility for ORDBMS. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, 1999, Edinburgh, Scotland, UK*, pages 494–505. Morgan Kaufmann, 1999.

[16] Theodore Johnson and Dennis Shasha. Some approaches to index design for cube forest. *Data Engineering Bulletin*, 20(1):27–35, 1997.

[17] Rosa Meo, Giuseppe Psaila, and Stefano Ceri. A new SQL-like operator for mining association rules. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, 1996, Mumbai (Bombay), India*, pages 122–133. Morgan Kaufmann, 1996.

[18] Amir Netz, Surajit Chaudhuri, Usama M. Fayyad, and Jeff Bernhardt. Integrating data mining with SQL databases: OLE DB for data mining. In *Proceedings of the 17th International Conference on Data Engineering*, pages 379–387. IEEE Computer Society, 2001.

[19] Carlos Ordonez and Paul Cereghini. SQLEM: fast clustering in SQL using the EM algorithm. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 559–570. ACM Press, 2000.

[20] Cecilia M. Procopiuc, Michael Jones, Pankaj K. Agarwal, and T. M. Murali. A monte carlo algorithm for fast projective clustering. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 418–427. ACM Press, 2002.

[21] Kenneth A. Ross and Divesh Srivastava. Fast computation of sparse datacubes. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 116–125. Morgan Kaufmann, 1997.

[22] Nick Roussopoulos, Yannis Kotidis, and Mema Roussopoulos. Cubetree: Organization of and bulk updates on the data cube. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 89–99. ACM Press, 1997.

[23] Sunita Sarawagi, Shiby Thomas, and Rakesh Agrawal. Integrating association rule mining with relational database systems: alternatives and implications. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 343–354. ACM Press, 1998.

[24] Kai-Uwe Sattler and Oliver Dunemann. SQL database primitives for decision tree classifiers. In *Proceedings of the tenth international Conference on Information and Knowledge Management*, pages 379–386, 2001.

[25] Shalom Tsur, Jeffrey D. Ullman, Serge Abiteboul, Chris Clifton, Rajeev Motwani, Svetlozar Nestorov, and Arnon Rosenthal. Query flocks: A generalization of association-rule mining. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, 1998, Seattle, Washington, USA*, pages 1–12. ACM Press, 1998.

[26] Haixun Wang and Carlo Zaniolo. Using SQL to build new aggregates and extenders for object- relational systems. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 166–175. Morgan Kaufmann, 2000.

[27] Yihong Zhao, Prasad Deshpande, and Jeffrey F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 159–170. ACM Press, 1997.