Freie Universität Bozen
Libera Università di Bolzano
Free University of Bozen · Bolzano

Fakultät für Informatik | Facoltà di Scienze e Tecnologie informatiche | Faculty of Computer Science

Parallel Computation of Ad-hoc Multi-Dimensional Aggregates

Andreas Heinisch

supervised by

Prof. Johann Gamper

Abstract

The θ-constrained multidimensional aggregation operator (θ-MDA) is an effective and flexible operator for the computation of ad-hoc complex aggregation queries. θ-MDA separates the specification of groupings (base table B) for which aggregates are reported from the specification of the associated aggregation groups over which the aggregates are computed (detail table R). In order to evaluate θ-MDA queries several algorithms have been proposed, among them *TCMDA* and *TCMDA*$^+$. For the evaluation, the base table B has to be updated for every entry of detail table R. This leads to an immense number of incremental updates on the base table B for range aggregates. *TCMDA*$^+$ reduces the number of incremental updates by reducing range aggregates to point aggregates followed by a post-processing step. However, both algorithms do not take advantage of the current evolution of modern computer architectures, where the clocking frequency stays constant and the number of computing cores increases.

In this thesis, we parallelize the *TCMDA* and the *TCMDA*$^+$ algorithm in order to take advantage of the evolution of multi-core architectures. In order to distribute the workload, we propose a horizontal partitioning of the base table B among the available processors. The incremental updates on these partitions can be computed by the available processors in isolation, where the horizontal partitioning strategy distributes the workload effectively. In addition, the reduction of the detail table R using point aggregates in *TCMDA*$^+$ is pushed to the database management system (DBMS). Both algorithms were implemented using the C programming language on top of an Oracle database. The dataset used in order to evaluate the performance of the algorithms was generated using the *dbgen* tool of the TPC-H benchmark framework. The empirical evaluation shows the effectiveness of the optimizations. The parallel algorithms using the reduction of the detail table R to SQL outperform the current state of the art algorithms in order of magnitudes in all tested data settings, where the main part of the computation time resides at the DBMS to construct the separate intermediate result tables using a *GROUP BY* statement.

To Sonja, Lara and Lea
Haec ornamenta sunt mea.

Acknowledgements

I would like to thank my supervisor, Prof. Johann Gamper, for the encouragement, patience and inputs during the thesis. He were always at disposal for discussions and responded to my thoughts and questions instantly. Without his valuable suggestions, corrections and tremendous knowledge in the field of multi-dimensional aggregation, this thesis would not have been possible.

The completion of this work would have been more difficult without the countless distractions in the rare leisure time by the fellow students, who became good friends. I would like to thank Christian Krüger, Philipp Hofer, Faizur Rahman Robin and Manfred Gerstgrasser for the friendship and various point of views concerning the parallelization. Philipp, we might find, once again, some frostburn gauntlets and Manfred, some day you will become a better LOL player than I am.

I would like to thank Christian Ammendola for the explanations regarding the separate intermediate result tables. Moreover, I am indebted to Anton Dignös for the help during the implementation of the algorithms and the assistance to setup the development environment.

Finally, I would like to thank my wife Sonja. All the things would not have been possible without the sacrifices you have made. Words cannot describe how thankful I am for this and it probably would need another thesis to express my heartfelt love for you. Your deep love, patience and encouragement helped me during the completion of the thesis, especially in these moments where I was frustrated. Thank you for raising smart toddlers, when I spend more time with the computer than with you. Last but not least, I would like to express my love to my daughters Lara and Lea for the tolerance whenever I occupied the desk in their playroom.

Andreas

Contents

List of Tables

List of Figures

1 Introduction

1.1 Motivation

The analysis of large amounts of data is an important task in different areas, such as business intelligence [19], health care [15] and scientific fields [10]. This requires enhanced techniques for a flexible formulation and efficient evaluation of complex multi-dimensional aggregation queries. On account of this, the SQL:2003 standard [25] introduced window functions in order to overcome the limitations of analytical processing in SQL. However, window functions still lack a high-performance and broad support for complex analytical queries such as cumulative aggregation over more than one dimension. To overcome the limitations of SQL window queries, the θ-constrained multidimensional aggregation operator (θ-MDA) [10] was introduced which supports a flexible formulation and efficient evaluation of complex multi-dimensional aggregation queries based on θ-conditions. For the evaluation of θ-MDA queries several approaches have been proposed, among them the *θ-Constrained Multi-Dimensional Aggregation (TCMDA)* [10] and the *θ-Constrained Multi-Dimensional Aggregation Plus (TCMDA$^+$)* [13].

TCMDA separates the specification of groupings (base table B) for which aggregates are reported from the specification of the associated aggregation groups over which the aggregates are computed (detail table R). This basically results in a nested loop over the base table B and the detail table R, where for every entry of detail table R the base table B has to be updated. For typical θ-MDA with hundreds of millions of detail tuples this leads to an immense number of incremental updates on the base table B for range aggregates, where the operators $\leq, \geq$ and $\neq$ are used in the θ-conditions. *TCMDA$^+$* reduces these incremental updates by reducing range aggregates to point aggregates (which use only $=$ in the θ-conditions) followed by a post-processing step.

Although, compared to the performance of SQL, the θ-MDA operator features an acceptable performance, there is still space for improvements. Both algorithms are designed to run on a single processor machine using exactly one executing thread and do not take advantage of the current evolution of modern computer architectures, where the clocking frequency stays constant and the number of computing cores increases [55]. In addition, parts of the computation in the *TCMDA$^+$* algorithm can be transformed to SQL. In this thesis we investigate these optimizations which increase the performance and allow the execution of θ-MDA queries with almost no limitations on the size of the input relations.

1.2 Complex Aggregation Queries: Running Example

An example for complex aggregation queries are moving and cumulative aggregates over multiple dimensions. Before the introduction of *window functions* in the SQL:2003 [25] standard, expressing such queries either resulted in unacceptable running times or were difficult to express. The key concept of window functions is to sort the input relation and to compute the aggregates during the scan of the sorted relation. For each row in the result a moving window determines a contiguous range of rows over which the aggregate value for that specific row is computed. However, for cumulative aggregations over multiple dimensions such an ordering does not exist, and the tuples that contribute to an aggregation result are formed by non-contiguous rows [10].

Consider the *Orders* relation of the TPC-H[1], which stores information about sales orders. Each row in the relation represents an order with the following information: the order key (*OrdKey*), the clerk key (*ClerkKey*), the total price (*TotPrice*), the order priority (*OrdPrior*), the order date (*OrdDate*) and others. Table 1 illustrates a simplified instance of the orders relation with eight orders shipped at three consecutive days.

Orders

	OrdKey	ClerkKey	TotPrice	OrdPrior	OrdDate
r_1	O1	C1	220	3	2013-04-18
r_2	O2	C2	440	2	2013-04-18
r_3	O3	C1	100	1	2013-04-18
r_4	O4	C3	240	1	2013-04-18
r_5	O5	C1	260	3	2013-04-19
r_6	O6	C3	640	2	2013-04-19
r_7	O7	C2	450	2	2013-04-19
r_8	O8	C2	300	1	2013-04-20

Table 1: Instance of Orders Relation

Consider the following query to analyze the number of orders in order to measure the development of the orders with respect to the order date and priority of the shipment:

Query 1: Compute the total number of orders per date and priority, the cumulative number of orders per date, the negated number of orders per priority, and the cumulative number of orders per date and priority.

The result of the query is shown in Table 2. The first two columns, i.e., *OrdDate* and *OrdPrior*, represent the different combinations of order dates and priorities. These columns form the base table for which the various aggregate functions are computed. To enhance reading, the base table and the aggregation results are separated by a vertical line.

The first aggregate *CntDP*, which counts all the orders per day and priority, can be expressed using the *GROUP BY* clause in *SQL*, because the aggregation groups are defined by identical values on the grouping attributes. The second aggregate, *CumCntD*, which reports the cumulative number of all orders per date, can be expressed with *SQL window functions* and the *UNBOUNDED PRECEDING* ordering clause. The third aggregate, *NegCntP*, which computes the number of orders per priority except those of a specific priority, can be constructed by using the *GROUP BY* clause and a self-join. The last aggregate, *CumCntDP*, which summarizes the number of orders per day and priority, is a two-dimensional cumulative aggregate where the window functions of SQL do not provide an adequate support. SQL window functions rely on the principle that the input relation can be ordered such that all tuples that contribute to an aggregation result are formed by contiguous rows. However, for the aggregation *CumCntDP* such an ordering does not exist.

Table 3 shows selected aggregate values of the result in Table 2 together with the corresponding aggregation groups, i.e., the tuples over which the aggregation

[1]TPC-H benchmark framework: `http://www.tpc.org/tpch/`

X

	OrdDate	OrdPrior	CntDP	CumCntD	NegCntP	CumCntDP
x_1	2013-04-18	1	2	4	5	2
x_2	2013-04-18	2	1	4	5	3
x_3	2013-04-18	3	1	4	6	4
x_4	2013-04-19	2	2	7	5	5
x_5	2013-04-19	3	1	7	6	7
x_6	2013-04-20	1	1	8	5	3

Table 2: Result of Query 1

values are computed. In the case of *CntDP* and *CumCntD* the aggregate is computed over contiguous groups of the result and hence can be computed using window functions. However, for the two-dimensional aggregate *CumCntDP* such a contiguous set of tuples does not exist and it is impossible to order the *Orders* relation such that the aggregation groups consists of adjacent tuples without introducing gaps in other aggregation groups of *CumCntDP*. In order to express such multi-dimensional aggregation queries in SQL a complex formulation with joins is required which for large inputs leads to unacceptable running times.

Aggregate value	Aggregation group
x_1.CntDP	{r_3, r_4}
x_3.CntDP	{r_1}
x_1.CumCntD	{r_1, r_2, r_3, r_4}
x_4.CumCntD	{r_1, r_2, r_3, r_4, r_5, r_6, r_7}
x_3.CumCntDP	{r_1, r_2, r_3, r_4}
x_6.CumCntDP	{r_3, r_4, r_8}

Table 3: Illustration of Aggregation Groups

1.3 θ-Constrained Multi-Dimensional Aggregation

The θ-constrained *multi-dimensional aggregation* (θ-MDA) operator introduced in [10] allows a succinct, systematic, and intuitive formulation of complex OLAP queries, where the grouping can be done along more than one dimension and the aggregation groups are identified by a general θ-condition. The operator requires four arguments which are illustrated in Figure 1. The parameters of the operator are a base table B which includes aggregation groups *for* which the aggregates are computed, a detail table R containing all tuples *over* which the aggregates are computed, a list of aggregate functions $\overrightarrow{l}$ and a list of grouping conditions $\overrightarrow{\theta}$. The θ-MDA operator computes the result table X for each tuple in the base table B by reporting the aggregation results according to the aggregation functions in $\overrightarrow{l}$ where the grouping conditions $\overrightarrow{\theta}$ determine the aggregation group from the detail table R. Figure 1 illustrates, where the shaded areas represent different aggregation groups over which the corresponding aggregate values are reported.

The algorithm for the computation of θ-MDA works in the following four steps. First, algebraic aggregates are replaced by their distributive sub-aggregates (e.g., *avg* is replaced by *sum* and *count*). Secondly, the algorithm constructs the result table X from the base table B and initializes the aggregation results. Next, for every tuple in the detail table R the aggregates in the result table X are updated. Finally, the algorithm applies the super-functions to the values of the sub-aggregates (e.g., in order to get *avg* the *sum* is divided by the *count*). Compared to SQL versions this evaluation strategy presented in [10] performs

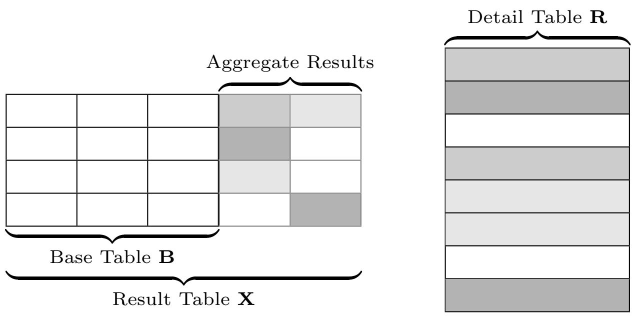


Figure 1: θ-MDA Operator Overview

efficiently. Practically, the algorithm evaluates a nested loop with $\mathcal{O}(|r| \cdot |b|)$ complexity. Since for typical OLAP applications the detail table R is very large and exceeds in orders of magnitudes the size of the base table B, the runtime of the algorithm highly depends on the size of the detail table R. In addition, the algorithm is sensitive with respect to the type of the constraints used in the theta conditions. Constraint operators such as $\leq$, $\geq$ or $\neq$ (used in range queries) exhibit unacceptable running times with respect to theta conditions using only equality constraints (used in point queries). This can be explained because range aggregates require a larger amount of updates in the base table B for every tuple in the detail table R than point aggregates.

In order to overcome the larger amount of incremental updates for range aggregates a new evaluation strategy for θ-MDA queries was introduced in [13]. The approach differs from the θ-MDA algorithm introduced in [10] by two main aspects.

The first aspect takes advantage of the efficient computation of point aggregates in θ-MDA queries. First, each grouping condition θ_i is transformed to $\tilde{\theta}_i$ such that each $\tilde{\theta}_i$ contains an equality constraint $r.A = b.A$ for each attribute $A \in R$ that is used in the corresponding θ_i. Next, the approach computes an intermediate result table, $\tilde{X}$, using point aggregates where the base table is a projection of R to the set of all attributes in R that are used in θ_i. This requires significantly less updates in $\tilde{X}$ than range queries would cause. In the end, the final result table, X, is derived from $\tilde{X}$ using the aggregates g_i in combination with the original conditions in θ_i. The functions g_i to compute the aggregates of the intermediate values are called super aggregates [33]. Even though each tuple of $\tilde{X}$ may affect multiple tuples in X, the overall runtime is significantly reduced since $\tilde{X}$ is typically much smaller than R.

However, even if the conditions in $\tilde{\theta}_i$ contain only equality constraints, a single tuple in R might still affect several tuples in $\tilde{X}$. This is the case if a $\tilde{\theta}_i$ involve only a subset of the grouping attributes in other $\tilde{\theta}_i$s, i.e., $R_i \neq Rj$ with $i \neq j$, which produces duplicate groupings and hence *redundant* updates in $\tilde{X}$. This leads to the second aspect of the new evaluation strategy in [13]. In order to avoid duplicate groupings in $\tilde{X}$, *separate* intermediate result tables, $\tilde{X}^i$, are used for each $\tilde{\theta}_i(f_i)$ where every intermediate result table $\tilde{X}_i$ requires exactly one update for each $r \in R$. However, the creation of separate intermediate result tables $\tilde{X}_i$ introduces an overhead in the evaluation of θ-MDA queries for conditions in θ_i containing only equality constraints in comparison to the evaluation strategy in [10]. Since equality constraint conditions in θ_i are highly selective, the set of affected tuples in X can be efficiently identified by using one or more indexes corresponding to the theta conditions θ_i in the result table X.

1.4 Contribution

Regardless of the optimizations of the θ-MDA evaluation strategies presented in [10, 13] there is still space for improvements. Both of the evaluation strategies do not take into consideration the current evolution of modern computer architectures where machines are composed by processors with several cores. Nowadays, even desktop computers consist of processors with more than one core where the number of cores per processor constantly grows. In addition to the multicore processors, programmable graphics processing units (GPUs) have emerged where the GPU is a massively parallel processor including more than hundred processors which support thousands of active threads. These GPUs deliver high computational power in order to solve highly parallel problems.

Moreover, parts of the θ-MDA evaluation strategy given in [13] can be transformed to SQL (e.g., the computation of the separate intermediate result tables using point aggregates). These statements are optimized by the DBMS and executed efficiently by utilizing the database facilities and optimization strategies. After the execution of these SQL statements, the returned result is further processed in order to elaborate the result of the requested θ-MDA query.

In this thesis we focus on the limitations of the presented approaches for computing θ-MDA queries which does not exploit the potential of current computer architectures and the DBMS facilities. The main contribution can be summarized as follows:

- We develop a partitioning approach for the algorithms in [10, 13] in order to distribute the workload among the processors and to allow the parallel execution of the algorithms. Since the algorithm presented in [13] performs an additional step, i.e., the creation of the intermediate result tables, for the computation of point aggregates, we additionally developed a partitioning strategy in order to parallelize the approach in [10].

- For the approach illustrated in [13], we define a strategy in order reduce the computation of the separate intermediate result tables to SQL. The created SQL statements can be optimized and executed efficiently by the DBMS and the returned result can be further processed to complete the result of the θ-MDA query.

- We define the algorithms of the introduced partitioning and parallelization strategies on top of the evaluating strategies given in [10, 13]. In addition, we include the reduction of the computation of the separate intermediate result tables using SQL in the parallelized algorithm based on the sequential evaluation strategy illustrated in [13].

- We implement the newly introduced algorithms as well as the algorithms depicted in [10, 13] as C programs on top of an Oracle database and present the results of an in-depth empirical evaluation using the *TPC-H*[2] benchmark framework. The results show that the defined algorithms executed in parallel using the reduction of the separate intermediate result tables using SQL outperform the existing algorithms in [10, 13] in order of magnitudes in all tested data settings.

1.5 Organization of the Thesis

In Section 2 related work regarding to the topic of the thesis is presented.

Section 3 provides additional details about the θ-MDA operator and its evaluation strategies together with a running example which is used throughout the entire thesis.

Section 4 presents the parallel evaluation approach for θ-MDA queries on top of the strategies in [10, 13]. Furthermore, the section introduces the transformation of the computation of the separate intermediate result tables using point aggregates presented in [13].

Section 5 introduces the reduction of θ-MDA queries to a *GROUP BY* statement followed by a post-processing step.

Section 6 illustrates the evaluation of the parallelized algorithms including the utilization of the database facilities.

In Section 6.4 details about the implementation of the newly introduced θ-MDA algorithms is presented with information about the parameters and the result format of the developed applications.

In Section 7, we present the results of the experiments showing that the runtimes of θ-MDA queries highly depend on the transformed *GROUP BY* statements, rather than on the number of processors, if the number of distinct groups in the separate intermediate result tables are small. As a result, the constraints operators in the theta conditions θ_i play a secondary role in the post-processing step in the previously described scenario. If the number of distinct groups in the separate intermediate result tables are large, the execution time of the post-processing step increases where the constraints operators in the theta conditions θ_i become dominant.

In conclusion, in Section 8 final remarks together with future work about the operator is presented.

2 Related Work

In this section we give an overview of the investigations about multi-dimensional data aggregation techniques and about the computation of large amount of data in parallel.

2.1 Multi-Dimensional Data Aggregation

There exists a variety of work about multi-dimensional data aggregation to enhance flexibility and/or to provide a better performance. The *CUBE* operator [34] is part of the SQL and allows the N-dimensional generalization of the SQL aggregate functions and the *GROUP BY* operator using equality constraint conditions over several attributes. For aggregation queries over a part of the data cube, grouping sets [17, 18] can be used to define group attribute combinations over which aggregates are computed. The *UNPIVOT* operator introduced in [32] provides techniques to rotate columns of a relation into rows. This allows alternative definitions of the groups, where the aggregations are applied on the column values wanted in the final output. In [48] alternatives for coupling data mining with database systems to accommodate complex data analysis is proposed. All the introduced operators provide techniques to enhance the formulation of aggregation queries and/or to improve the efficiency of their computation using specialized algorithms. However, research efforts rarely consider the optimizations and implementations of more complex OLAP expressions, such as cumulative aggregates.

A broader support for complex OLAP expressions was provided by SQL with the introduction of the *WINDOW* construct in the SQL:2003 [25] standard. The key concept of window functions is to sort the input relation and to compute the aggregates during the scan of the sorted relation. For each row in the result a moving window determines a contiguous range of rows over which the aggregate value for that specific row is computed. This allows the formulation of moving and cumulative aggregates as well as rank aggregates. In addition, a wide variety of new aggregate functions was introduced in the SQL/OLAP amendment [14] improving the capabilities of SQL with respect to complex OLAP. Nonetheless, the efficient evaluation of general complex OLAP queries constitutes a problem. Moreover, window functions do not support the computation of cumulative aggregates over multiple dimensions, because windows functions rely on a sorted input relation, where tuples belonging to the same group are formed by non-contiguous rows [10]. For input relations grouped over more than one dimension there may exist no ordering so that the tuples of all groups are arranged contiguously.

Another approach to improve the performance of aggregation queries is based on the pre-computation of the aggregates together with incremental updates. In [36] a query optimization technique using the pre-computing materialized views is presented. The approach targets issues, where the materialization of all views (cells) of the corresponding data cube is too expensive. The strategy uses a lattice framework to express the dependencies among the views to determine the set of views to be materialized. To avoid the re-computation of the materialized views as soon as the source relations change, [43] introduces an approach which allows to incrementally update pre-computed aggregates by considering only the changes in the source relations. An improvement of this approach is presented in [41] further reduces the computational effort of the strategy. Although the pre-computation of data cubes works well for point aggregates, the performance of range aggregates suffers, since the cells of the data cubes has to be accessed repeatedly [13]. To overcome this problem, [38] presented an approach for computing range queries in data cubes. The approach is based on the pre-computation of the prefix sums of the data cube, which can be used to answer ad-hoc aggregation queries. Subsequent work has studied techniques to lower the comparable high update costs of the prefix sum cube [21, 29, 42].

To efficiently answer ad-hoc OLAP queries with a single scan of the detail table, the *multi-dimensional join* (MDJ) [16] and later the *generalized multi-dimensional join* (GMDJ) [9] have been introduced. The operator has been used in complex OLAP settings to transform general sub-query expressions into expressions that use GMDJ instead of joins, outer joins or set difference [13]. In [53] the GDMJ in combination with MapReduce to compute aggregation queries over RDF data is used.

The *θ constrained multi-dimensional aggregation (θ-MDA)* operator [10] extends the MDJ and presents a detailed cost model together with algebraic transformation rules. θ-MDA outperforms SQL for complex multi-dimensional aggregation queries, such as range aggregates over multiple dimensions. This approach has been improved further by reducing the range aggregates to point aggregates in [13]. The optimizations presented in this thesis are based on the θ-MDA operator and pushes the main computational effort of the optimizations presented in [13] to the database, which significantly reduces the runtime of θ-MDA queries.

2.2 Parallel Processing of Large Data

The trend towards massively parallel computers increased the investigations to exploit multiple processors for complex OLAP. The work presented in [30] focuses on algorithms for the construction of data cubes on distributed-memory parallel computers to provide efficient query processing for OLAP applications using precomputed aggregates. In [31] the previous work is extended by using parallel processing for OLAP and data mining on a parallel and scalable infrastructure. For the parallel evaluation of aggregates,[51] introduces various strategies to split the aggregate computation between the sites and the coordinator to optimize performance. However, the previously presented investigations focus on the distributed infrastructure or consider only the case of simple SQL aggregates. For the case of complex OLAP queries, [9] presents a framework to decrease the communication costs in a distributed data warehouse setting. The work presented in this thesis focuses on the computation of complex OLAP queries on parallel computers, where the communication costs are assumed to be very cheap.

There exist considerable research activities around the MapReduce [2] paradigm. MapReduce is a programming model and an associated implementation for processing large datasets, where the user specifies the computation in terms of a map and a reduce function [23]. The advantage of the MapReduce paradigm is that the underlying system automatically parallelizes the computation, handles machine failures and schedules the inter-machine communication to make efficient use of the network and disks [24]. The paradigm has been adapted for machine learning on multicore [20] as well as for simplified relational data processing on large clusters [57]. A popular open-source implementation of the MapReduce paradigm is *Apache Hadoop* framework [56]. In addition, there exist several implementations of MapReduce on various hardware platforms. For instance, in [47] the current state of the art implementation of the MapReduce paradigm for shared-memory systems (*Phoenix*) is introduced whereas [37] presents the *Mars* framework for the computation of MapReduce on graphics processors (GPUs). Mars hides the programming complexity of the GPU and has been recently integrated into Hadoop [26].

The work in [44] conducts benchmarks, where the observed performance of parallel SQL database management systems (DBMS) was strikingly better compared to the Hadoop system once the process to load the data into the system was finished. They conclude that MapReduce is more like an Extract-Transform-Load system, because it loads the data without the indexing and reorganization costs in an ad-hoc manner. They argue that the performance advantage of the DBMSs results from a number of technologies developed over the past, such as B-tree indexes, novel storage mechanisms and sophisticated parallel algorithms for querying large amounts of relational data. The advantages of the Hadoop MapReduce compared to the parallel DBMSs are the ease of use and the fault tolerance, but MapReduce suffers in terms of computation time. The same group of authors argue in [54] that the MapReduce paradigm complements the DBMS technology and provide insights how the systems should complement each other. The work has been extended in [8], where an architectural hybrid of MapReduce and DBMS technologies for analytical workloads is provided.

There exist application programming interfaces with respect to the MapReduce paradigm for the parallelization of algorithms, e.g., *OpenMP* [22], and *MPI* [52]. OpenMP is an API that supports multi-platform shared memory multiprocessing programming in the C programming language which uses a scalable model that gives programmers a simple and flexible interface to develop parallel application. MPI is a standardized and portable message-passing system designed to write portable message-passing programs in the C programming language for both shared or distributed memory architectures. In addition to the pure OpenMP and pure MPI programming models, there exist several investigations of the combination of both techniques (hybrid programming model) [45, 39]. The work in [45] investigates cases, where the hybrid programming model is the superior solution due to reduced communication needs and memory consumption, or improved load balance. They state that the machine topology has significant impact on performance of the parallelization strategies. The investigations in [39] came to the same conclusions, i.e., the best programming paradigm depends on the given problem, the hardware components of the cluster and the network facilities.

The algorithms presented in this thesis were parallelized using OpenMP, because they are designed to run on symmetric multiprocessing (SMP) systems. Since the main part of the computation resides at the DBMS, the computation can be parallelized by specialized parallel SQL database managements systems like *DBMS-X* or *Vertica* presented in [44]. However, we consider the implementation using the MapReduce paradigm in future investigations of the introduced *PTCMDA*$^+$*-SQL* algorithm.

3 θ-Constrained Multi-Dimensional Aggregation

In this section, we summarize the θ-MDA operator and various evaluation algorithms that have been proposed in the past. First, the θ-MDA operator is formally defined and illustrated using our running example. Next, the evaluation algorithm in order to compute θ-MDA queries is described. Moreover, the optimization strategies of the θ-MDA operator are defined, i.e., the reduction of range to point aggregates and the usage of separate intermediate result tables.

3.1 Definition

θ-MDA takes as input four parameters which are the base table, the detail table, a list of aggregate functions and a list of grouping conditions. In the following the θ-MDA operator is formally defined assuming multi-set semantics, i.e., the generalized algebraic operators ($\pi, \sigma, \cup$, etc.) are consistent with SQL and operate on and return multi-sets. Additionally, $\mathbf{B}$ and $\mathbf{R}$ are used to represent the database schemas $(B_1, ..., B_k)$ and $(R_1, ..., R_p)$, respectively, and $x.\mathbf{B}$ refers to $(x.B_1, ..., x.B_k)$. Finally, $E \rightarrow C$ renames E to C and $attr(E)$ denotes the set of attributes used in E.

Definition 1. (***θ-MDA Operator [10]***) Let $B(\mathbf{B})$ and $R(\mathbf{R})$ tables, θ_i be conditions with $attr(\theta_i) \subseteq \mathbf{B} \cup \mathbf{R}$, and $l_i = (f_{i_1} \rightarrow C_{i_1}, ..., f_{i_{k_i}} \rightarrow C_{i_{k_i}})$ with $1 \leq i \leq m$, be a list of aggregate functions over attributes $A_{i_1}, A_{i_2}, \ldots, A_{i_{k_i}}$ in $\mathbf{R}$. The θ-MDA operator is defined as

$$X = \mathcal{G}^{\theta}(B, R, (l_1, ..., l_m), (\theta_1, ..., \theta_m)),$$

where $\mathbf{X} = (\mathbf{B}, C_1, ..., C_{1_{k_1}}, ..., C_{m_1}, ..., C_{m_{k_m}})$ is the schema of the result table and each tuple $b \in B$ produces a result tuple $x \in X$ with

$$\begin{aligned} x.\mathbf{B} &= b.\mathbf{B}, \\ x.C_{i_j} &= f_{i_j}(\{r.A_{i_j} | r \in R \wedge \theta_i(b, r)\}), \forall C_{i_j} \in \mathbf{X}. \end{aligned}$$

The base table is called B, the detail table R and the final result table of the θ-MDA operator is denoted by X. For a base tuple, $b \in B$, the conditions θ_i determine the sets of detail tuples, $r \in R$, over which the aggregates f_{i_j} are evaluated. The aggregation results are the values of attributes C_{i_j} in the result relation.

Example 1. In the following example, we present the formulation of *Query 1* from Section 1.2 using the θ-MDA operator. The query can be expressed as $\mathcal{G}^{\theta}(B, Orders \rightarrow r, (l_1, l_2, l_3, l_4), (\theta_1, \theta_2, \theta_3, \theta_4))$ where

$$\begin{aligned} &B : \pi[OrdDate, OrdPrior]Orders \\ &l_1 : (count(OrdDate) \rightarrow CntDP) \\ &\theta_1 : r.OrdDate = b.OrdDate \wedge r.OrdPrior = b.OrdPrior \\ &l_2 : (count(OrdDate) \rightarrow CumCntD) \\ &\theta_2 : r.OrdDate \leq b.OrdDate \\ &l_3 : (count(OrdDate) \rightarrow NegCntP) \\ &\theta_3 : r.OrdPrior \neq b.OrdPrior \\ &l_4 : (count(OrdDate) \rightarrow CumCntDP) \\ &\theta_4 : r.OrdDate \leq b.OrdDate \wedge r.OrdPrior \leq b.OrdPrior \end{aligned}$$

The step-wise computation of the result of *Query 1* is illustrated in Figure 2. At the beginning of the computation, all aggregate attributes of the result table X are set to their corresponding initial values of the aggregate functions, i.e., 0 for *sum* and *count*, NULL for *max* and *min*. After the initialization of the aggregate attributes, each tuple of the Orders table is processed, where all aggregate attributes of the affected tuples are updated in the result table X. In our example, we begin with tuple r_1 and update all affected aggregate attributes of the result table by one. After the first tuple has been elaborated, the next tuple r_2 of the Orders table is loaded and processed as tuple r_1. Finally, after all tuples of the Orders table have been handled, the result table X is complete.

X

	OrdDate	OrdPrior	CntDP	CumCntD	NegCntP	CumCntDP
x_1	2013-04-18	1	0	0	0	0
x_2	2013-04-18	2	0	0	0	0
x_3	2013-04-18	3	0	0	0	0
x_4	2013-04-19	2	0	0	0	0
x_5	2013-04-19	3	0	0	0	0
x_6	2013-04-20	1	0	0	0	0

Orders

	...	OrdPrior	OrdDate
r_1	...	3	2013-04-18
r_2	...	2	2013-04-18
r_3	...	1	2013-04-18
r_4	...	1	2013-04-18
r_5	...	3	2013-04-19
r_6	...	2	2013-04-19
r_7	...	2	2013-04-19
r_8	...	1	2013-04-20

X

	OrdDate	OrdPrior	CntDP	CumCntD	NegCntP	CumCntDP
x_1	2013-04-18	1	0	**1**	**1**	0
x_2	2013-04-18	2	0	**1**	**1**	0
x_3	2013-04-18	3	**1**	**1**	0	**1**
x_4	2013-04-19	2	0	**1**	**1**	0
x_5	2013-04-19	3	0	**1**	0	**1**
x_6	2013-04-20	1	0	**1**	**1**	0

Orders

	...	OrdPrior	OrdDate
~~r_1~~	...	~~3~~	~~2013-04-18~~
r_2	...	2	2013-04-18
r_3	...	1	2013-04-18
r_4	...	1	2013-04-18
r_5	...	3	2013-04-19
r_6	...	2	2013-04-19
r_7	...	2	2013-04-19
r_8	...	1	2013-04-20

X

	OrdDate	OrdPrior	CntDP	CumCntD	NegCntP	CumCntDP
x_1	2013-04-18	1	0	**2**	**2**	0
x_2	2013-04-18	2	**1**	**2**	1	**1**
x_3	2013-04-18	3	1	**2**	**1**	**2**
x_4	2013-04-19	2	0	**2**	1	**1**
x_5	2013-04-19	3	0	**2**	**1**	**2**
x_6	2013-04-20	1	0	**2**	**2**	0

Orders

	...	OrdPrior	OrdDate
~~r_1~~	...	~~3~~	~~2013-04-18~~
~~r_2~~	...	~~2~~	~~2013-04-18~~
r_3	...	1	2013-04-18
r_4	...	1	2013-04-18
r_5	...	3	2013-04-19
r_6	...	2	2013-04-19
r_7	...	2	2013-04-19
r_8	...	1	2013-04-20

X

	OrdDate	OrdPrior	CntDP	CumCntD	NegCntP	CumCntDP
x_1	2013-04-18	1	**1**	**3**	2	**1**
x_2	2013-04-18	2	1	**3**	**2**	**2**
x_3	2013-04-18	3	1	**3**	**2**	**3**
x_4	2013-04-19	2	0	**3**	**2**	**2**
x_5	2013-04-19	3	0	**3**	**2**	**3**
x_6	2013-04-20	1	0	**3**	2	**1**

Orders

	...	OrdPrior	OrdDate
~~r_1~~	...	~~3~~	~~2013-04-18~~
~~r_2~~	...	~~2~~	~~2013-04-18~~
~~r_3~~	...	~~1~~	~~2013-04-18~~
r_4	...	1	2013-04-18
r_5	...	3	2013-04-19
r_6	...	2	2013-04-19
r_7	...	2	2013-04-19
r_8	...	1	2013-04-20

X

	OrdDate	OrdPrior	CntDP	CumCntD	NegCntP	CumCntDP
x_1	2013-04-18	1	2	4	5	2
x_2	2013-04-18	2	1	4	5	3
x_3	2013-04-18	3	1	4	6	4
x_4	2013-04-19	2	2	7	5	5
x_5	2013-04-19	3	1	7	6	7
x_6	2013-04-20	1	1	8	5	3

Orders

	...	OrdPrior	OrdDate
~~r_1~~	...	~~3~~	~~2013-04-18~~
~~r_2~~	...	~~2~~	~~2013-04-18~~
~~r_3~~	...	~~1~~	~~2013-04-18~~
~~r_4~~	...	~~1~~	~~2013-04-18~~
~~r_5~~	...	~~3~~	~~2013-04-19~~
~~r_6~~	...	~~2~~	~~2013-04-19~~
~~r_7~~	...	~~2~~	~~2013-04-19~~
~~r_8~~	...	~~1~~	~~2013-04-20~~

Figure 2: Step-wise Processing of Query 1

3.2 Evaluation Strategies

Two algorithms following the above approach for the evaluation of θ-MDA queries are given in [10].

The first algorithm, *BasicTCMDA*, works in four steps. In the first step, the algebraic aggregates are replaced by their distributive sub-aggregates, e.g., *avg* is replaced by *sum* and *count*. In the second step, the algorithm constructs the result table X from B and initializes the aggregation results. Step 3 scans the detail table and computes the aggregates. Finally, the result is computed by applying the super-aggregates to the values of the sub-aggregates, e.g., to get *avg* the *sum* is divided by the *count*. *BasicTCMDA* becomes expensive if the result table X grows, since for each tuple $r \in R$ all rows in the result table X are considered.

The second algorithm, *IndexedTCMDA* (see Algorithm 1), avoids the problem that for every tuple in the detail table all tuples of the result table have to be scanned. The algorithm creates an index on the result table X. Therefore, for a tuple $r \in R$ and a condition θ the set of affected tuples in X can be efficiently identified. The first and the last step, where the aggregates transformation occurs are the same as in *BasicTCMDA*. In the second step, in addition to constructing the result table, one or more indexes on the result table with respect to the theta conditions are created. In step 3 the relevant result tuples are identified by using the created indexes which reduces the update operations significantly for highly selective constraint operators such as $=$. In the case of less selective constraint operators, such as $\leq$ and $\geq$, the performance gain of *IndexedTCMDA* in comparison with *BasicTCMDA* is smaller. For constraint operators where indexes cannot be consulted such as $\neq$ the performance achievement of *IndexedTCMDA* is negligible compared to the *BasicTCMDA* algorithm.

3.3 Reducing Range to Point Aggregates

In order to overcome the larger amount of incremental updates for range aggregates an improved evaluation strategy for θ-MDA queries for less selective constraint operators, such as $\leq$, $\geq$ and $\neq$, was provided in [13]. In the following the optimizations of the θ-MDA operator are formally defined.

Definition 2. (***Reduction to Point Aggregates [13]***) Let $B(\mathbf{B})$ and $R(\mathbf{R})$ tables, θ_i be conditions with $attr(\theta_i) \subseteq \mathbf{B} \cup \mathbf{R}$, and $l_i = (f_{i_1} \to C_{i_1}, ..., f_{i_{k_i}} \to C_{i_{k_i}})$ with $1 \leq i \leq m$, be a list of aggregate functions over attributes $A_{i_1}, ..., A_{i_{k_i}}$ in $\mathbf{R}$, $G = (g_{i_1}, ..., g_{i_{k_i}})$ be the corresponding super aggregates [33]. Furthermore, let $\mathbf{R}_i = \mathbf{R} \cap attr(\theta_i)$ denote the attributes in $\mathbf{R}$ that occur in θ_i. Then $\mathcal{G}^\theta(B, R, (l_1, ..., l_m), (\theta_1, ..., \theta_m))$ can be computed as follows:

1. construct $\tilde{\Theta} = (\tilde{\theta}_1, ..., \tilde{\theta}_m)$:

$$\tilde{\theta}_i(r, b) = \bigwedge_{A \in \mathbf{R}_i} r.A = b.A \text{ with } r \in R, b \in B;$$

2. compute an intermediate result table:

$$\tilde{X} = \mathcal{G}^\theta(\pi_{\mathbf{R_1} \cup ... \cup \mathbf{R_m}}(R), R, l_i, \tilde{\Theta});$$

Algorithm 1: *IndexedTCMDA*$(B, R, (l_1, ..., l_m), (\theta_1, ..., \theta_m))$

```
// Step 1: Replace algebraic aggregates by distributive sub-aggregates
Let $l'_i = l_i, 1 \leq i \leq m$;
foreach algebraic aggregate $f_{i_j} \in \{l'_i, ..., l'_m\}$ do
    Replace $f_{i_j}$ with its distributive sub-aggregates $f^1_{i_j}, ..., f^{p_{i_j}}_{i_j}$;

// Step 2: Construct result table X
// Note: $v_j$'s are the initial values of the aggregate functions (0 for sum and count,
   NULL for max and min)
Let $\mathbf{N} = (v'_{1_1}, ..., v'_{m_1}, ..., v'_{m_{k_m}})$;
Let $X = B \times N$;
Build indexes for $X$;
// Step 3: Compute the aggregates
foreach tuple $r \in R$ do
    foreach $\theta_i \in \{\theta_1, ..., \theta_m\}$ do
        Fetch the rows $X_i = \{x \in X | \theta_i(x, r)\}$ using the created index;
        foreach $x \in X_i$ do
            Update the aggregates $f_{i_1}, ..., f_{i_{k_i}}$ in $x$;

// Step 4: Apply the super-functions
if $l_1, ..., l_m$ contains algebraic aggregates then
    foreach row $x \in X$ do
        foreach algebraic aggregate $f_{i_j}$ in $l_1, ..., l_m$ do
            Let $g_{i_j}$ be the super function of $f_{i_j}$;
            In $x$, replace $f^1_{i_j}, ..., f^{p_{i_j}}_{i_j}$ by a single column $f_{i_j}$ and set
            $x.f_{i_j} = g_{i_j}(x.f^1_{i_j}, ..., x.f^{p_{i_j}}_{i_j})$;

return $X$;
```

3. compute the result table:

$$X = \{b \circ v | b \in B \wedge v = (g_{i_1}(\tilde{X}_{[b,\tilde{\theta}_1]}), ..., g_{i_{k_i}}(\tilde{X}_{[b,\tilde{\theta}_m]}))\},$$

where $\tilde{X}_{[b,\tilde{\theta}_i]} = \pi_{\mathbf{R}_i, C_{i_j}}\{\tilde{x} \in \tilde{X} | \theta_i(\tilde{x}, b)\}$.

Example 2. Following, we present the evaluation of *Query 1* from Section 1.2 using reduction to point aggregates. First, the conditions are transformed to

$$\tilde{\theta}_1(r, b) \equiv (r.OrdDate = b.OrdDate \wedge r.OrdPrior = b.OrdPrior),$$
$$\tilde{\theta}_2(r, b) \equiv (r.OrdDate = b.OrdDate),$$
$$\tilde{\theta}_3(r, b) \equiv (r.OrdPrior = b.OrdPrior),$$
$$\tilde{\theta}_4(r, b) \equiv (r.OrdDate = b.OrdDate \wedge r.OrdPrior = b.OrdPrior).$$

Afterwards, using these conditions the initial result in the intermediate result table is computed as illustrated in Figure 3. First, all aggregate attributes of the intermediate result table $\tilde{X}$ are set to their corresponding initial values of the aggregate functions, i.e., 0 for *sum* and *count*, NULL for *max* and *min*. After the initialization of the aggregate attributes, each tuple of the Orders table is processed, where all aggregate attributes of the affected tuples are updated in the result table $\tilde{X}$ using the previously created equality conditions $\tilde{\theta}_i$. Using these equality conditions $\tilde{\theta}_i$ significantly reduces the incremental updates in the intermediate result table $\tilde{X}$. After all tuples of the Orders table have been processed, the intermediate result table $\tilde{X}$ is complete. Finally, the final result table

X is derived from the intermediate result table $\tilde{X}$ using the super-aggregates $G = (g_{i_1}, ..., g_{i_{k_i}})$ in combination with the original conditions $(\theta_1, ..., \theta_m)$.

$B : \pi[OrdDate, OrdPrior]Orders$
$l_1 : (count(OrdDate) \rightarrow CntDP)$
$\theta_1 : r.OrdDate = b.OrdDate \wedge r.OrdPrior = b.OrdPrior$
$l_2 : (count(OrdDate) \rightarrow CumCntD)$
$\theta_2 : r.OrdDate \leq b.OrdDate$
$l_3 : (count(OrdDate) \rightarrow NegCntP)$
$\theta_3 : r.OrdPrior \neq b.OrdPrior$
$l_4 : (count(OrdDate) \rightarrow CumCntDP)$
$\theta_4 : r.OrdDate \leq b.OrdDate \wedge r.OrdPrior \leq b.OrdPrior$

Orders

	...	OrdPrior	OrdDate
r_1	...	3	2013-04-18
r_2	...	2	2013-04-18
r_3	...	1	2013-04-18
r_4	...	1	2013-04-18
r_5	...	3	2013-04-19
r_6	...	2	2013-04-19
r_7	...	2	2013-04-19
r_8	...	1	2013-04-20

Step 1: $(\tilde{\theta}_1, ..., \tilde{\theta}_4)$

Step 2:

$\tilde{X}$

	OrdDate	OrdPrior	CntDP	CumCntD	NegCntP	CumCntDP
$\tilde{x}_1$	2013-04-18	1	2	4	3	2
$\tilde{x}_2$	2013-04-18	2	1	4	3	1
$\tilde{x}_3$	2013-04-18	3	1	4	2	1
$\tilde{x}_4$	2013-04-19	2	2	3	3	2
$\tilde{x}_5$	2013-04-19	3	1	3	2	1
$\tilde{x}_6$	2013-04-20	1	1	1	3	1

$(\theta_1, ..., \theta_4)$

Step 3:

X

	OrdDate	OrdPrior	CntDP	CumCntD	NegCntP	CumCntDP
x_1	2013-04-18	1	2	4	5	2
x_2	2013-04-18	2	1	4	5	3
x_3	2013-04-18	3	1	4	6	4
x_4	2013-04-19	2	2	7	5	5
x_5	2013-04-19	3	1	7	6	7
x_6	2013-04-20	1	1	8	5	3

Figure 3: Processing Query 1 Using Reduction to Point Aggregates

Even if the conditions in $\tilde{\theta}_i$ contain only equality constraints, a single tuple in R might still affect several tuples in $\tilde{X}$, for instance, if equality conditions in $\tilde{\theta}_i$ constrain only a subset of all grouping attributes in R, i.e., $R_i \subset R1 \cup ... \cup R_m$. In our running example this concerns the conditions $\tilde{\theta}_2$ and $\tilde{\theta}_3$, which produce redundant updates in $\tilde{X}$. In order to tackle these redundant updates separate intermediate result tables were introduced in [13] where every intermediate result table $\tilde{X}_i$ requires exactly on update for each $r \in R$. Next, the strategy of using separate intermediate result tables is formally defined.

Definition 3. (***Separate Intermediate Result Tables [13]***) Let $B(\mathbf{B})$ and $R(\mathbf{R})$ tables, θ_i be conditions with $attr(\theta_i) \subseteq \mathbf{B} \cup \mathbf{R}$, and $l_i = (f_{i_1} \rightarrow C_{i_1}, ..., f_{i_{k_i}} \rightarrow C_{i_{k_i}})$ with $1 \leq i \leq m$, be a list of aggregate functions over attributes $A_{i_1}, A_{i_2}, ..., A_{i_{k_i}}$ in $\mathbf{R}$, $G = (g_{i_1}, ..., g_{i_{k_i}})$ be the corresponding super aggregates [33]. Furthermore, let $\mathbf{R}_i = \mathbf{R} \cap attr(\theta_i)$ denote the attributes in $\mathbf{R}$ that occur in θ_i. Then $\mathcal{G}^\theta(B, R, (l_1, ..., l_m), (\theta_1, ..., \theta_m))$ can be computed as follows:

1. construct $\tilde{\Theta} = (\tilde{\theta}_1, ..., \tilde{\theta}_m)$:

$$\tilde{\theta}_i(r, b) = \bigwedge_{A \in \mathbf{R}_i} r.A = b.A \text{ with } r \in R, b \in B;$$

2. compute m intermediate result tables:

$$\tilde{X}^i = \mathcal{G}^\theta(\pi_{\mathbf{R_i}}(R), R, l_i, \tilde{\theta}_i) \text{ for } i = 1, ..., m;$$

3. compute the result table:

$$X = \{b \circ f | b \in B \wedge f = (g_{i_1}(\tilde{X}^1_{[b,\tilde{\theta}_1]}), ..., g_{i_{k_i}}(\tilde{X}^i_{[b,\tilde{\theta}_m]}))\},$$

where $\tilde{X}^i_{[b,\tilde{\theta}_i]} = \pi_{\mathbf{R}_i, C_{i_j}}\{\tilde{x} \in \tilde{X}^i | \theta_i(\tilde{x}, b)\}$.

Example 3. In the next example, the evaluation of *Query 1* is performed using reduction to point aggregates together with separate intermediate result tables. Again, the conditions in θ are transformed to contain only equality constraints as in Example 2. Next, the intermediate result tables are created using the corresponding groupings from the conditions $\tilde{\theta}_i$. Figure 4 shows the four intermediate result tables, $\tilde{X}^1, \tilde{X}^2, \tilde{X}^3$ and $\tilde{X}^4$, where $\tilde{X}^1$ and $\tilde{X}^4$ contain two grouping attributes, OrdDate and OrdPrior, whereas $\tilde{X}^2$ and $\tilde{X}^3$ have one grouping attribute, i.e., OrdDate and OrdPrior, respectively. Processing a tuple from the detail table R now requires exactly one update in each of the intermediate result tables. After the creation of the separate intermediate result tables, the final result table X is built using the entries of the previously created intermediate result tables by applying the original conditions θ_i.

$B : \pi[OrdDate, OrdPrior]Orders$
$l_1 : (count(OrdDate) \rightarrow CntDP)$
$\theta_1 : r.OrdDate = b.OrdDate \wedge r.OrdPrior = b.OrdPrior$
$l_2 : (count(OrdDate) \rightarrow CumCntD)$
$\theta_2 : r.OrdDate \leq b.OrdDate$
$l_3 : (count(OrdDate) \rightarrow NegCntP)$
$\theta_3 : r.OrdPrior \neq b.OrdPrior$
$l_4 : (count(OrdDate) \rightarrow CumCntDP)$
$\theta_4 : r.OrdDate \leq b.OrdDate \wedge r.OrdPrior \leq b.OrdPrior$

Orders

	...	OrdPrior	OrdDate
r_1	...	3	2013-04-18
r_2	...	2	2013-04-18
r_3	...	1	2013-04-18
r_4	...	1	2013-04-18
r_5	...	3	2013-04-19
r_6	...	2	2013-04-19
r_7	...	2	2013-04-19
r_8	...	1	2013-04-20

Step 1: $(\tilde{\theta}_1, ..., \tilde{\theta}_4)$

Step 2:

$\tilde{X}^1$

	OrdDate	OrdPrior	CntDP
$\tilde{x}^1_1$	2013-04-18	1	2
$\tilde{x}^1_2$	2013-04-18	2	1
$\tilde{x}^1_3$	2013-04-18	3	1
$\tilde{x}^1_4$	2013-04-19	2	2
$\tilde{x}^1_5$	2013-04-19	3	1
$\tilde{x}^1_6$	2013-04-20	1	1

$\tilde{X}^4$

	OrdDate	OrdPrior	CumCntDP
$\tilde{x}^4_1$	2013-04-18	1	2
$\tilde{x}^4_2$	2013-04-18	2	1
$\tilde{x}^4_3$	2013-04-18	3	1
$\tilde{x}^4_4$	2013-04-19	2	2
$\tilde{x}^4_5$	2013-04-19	3	1
$\tilde{x}^4_6$	2013-04-20	1	1

$\tilde{X}^2$

	OrdDate	CumCntD
$\tilde{x}^2_1$	2013-04-18	4
$\tilde{x}^2_2$	2013-04-19	3
$\tilde{x}^2_3$	2013-04-20	1

$\tilde{X}^3$

	OrdPrior	NegCntP
$\tilde{x}^3_1$	1	3
$\tilde{x}^3_2$	2	3
$\tilde{x}^3_3$	3	2

$(\theta_1, ..., \theta_4)$

Step 3:

X

	OrdDate	OrdPrior	CntDP	CumCntD	NegCntP	CumCntDP
x_1	2013-04-18	1	2	4	5	2
x_2	2013-04-18	2	1	4	5	3
x_3	2013-04-18	3	1	4	6	4
x_4	2013-04-19	2	2	7	5	5
x_5	2013-04-19	3	1	7	6	7
x_6	2013-04-20	1	1	8	5	3

Figure 4: Processing Query 1 with Separate Intermediate Result Tables

In the following, the *TCMDA*$^+$ (see page 16) algorithm combining reduction to point aggregates and separate intermediate result tables is presented as given in [13]. First, the algorithm creates the separate intermediate result tables for each condition $\tilde{\theta}_i$. This might lead to intermediate result tables with identical grouping attributes such as $\tilde{X}^1$ and $\tilde{X}^4$ in our running example. The algorithm merges these intermediate result tables with identical groupings $\mathbf{R}_i$ to a single intermediate result table containing a column for each aggregate function. Next, indexes are created over the intermediate result tables together with the equality constraint conditions $\tilde{\theta}_i$. Afterwards, the detail table is scanned and for each tuple in the detail table $r \in R$ the according intermediate result tables is updated. If a tuple $\tilde{x}_i$ exists in the according intermediate result table $\tilde{X}^i$, then the entry is incrementally updated. Otherwise, a new entry in the intermediate result table $\tilde{X}^i$ is created and the aggregate value is initialized to the functions evaluated over r. Last, the final result table X is built where the aggregate attributes are initialized to their according neutral values v_i by combining the partial aggregates from the intermediate result tables $\tilde{x}_i$ using the super-aggregates.

Algorithm 2: *TCMDA*$^+(B, R, (l_1, ..., l_m), (\theta_1, ..., \theta_m))$

```
// Step 1: Initialize intermediate result tables
Let $\mathbf{R}_i \leftarrow \mathbf{R} \cap attr(\theta_i)$ for $i = 1, ..., m$;
Let $(\mathbf{R}_{j_1}, F_{j_1}), ..., (\mathbf{R}_{j_k}, F_{j_k})$, $k \leq m$, be a partitioning of the $f_i$ according to $\mathbf{R}_i$;
foreach partition $(\mathbf{R}_j, F_j)$ do
    $\tilde{X}^j \leftarrow$ empty table with schema $(\mathbf{R}_j, C_{j_1}, \ldots, C_{j_{k_j}})$;
    Create an index on $\tilde{X}^j$ over the attributes $\mathbf{R}_j$;
    $\tilde{\theta}_j(r, b) = \bigwedge_{A \in \mathbf{R}_j} r.A = b.A$;

// Step 2: Scan detail table $R(\mathbf{R})$ and update intermediate result tables
foreach tuple $r \in R(\mathbf{R})$ do
    foreach partition $(\mathbf{R}_j, F_j)$ do
        if $\exists \tilde{x} \in \tilde{X}^j$ such that $\tilde{\theta}_j(r, \tilde{x})$ then
            $\tilde{x}.C_{j_i} \leftarrow g_{j_i}(\tilde{x}.C_{j_i}, f_{j_i}(\{r\}))$ for $i = 1, ..., k_j$;
        else
            $\tilde{X}^j \leftarrow \tilde{X}^j \cup \{r.\mathbf{R}_j \circ (f_{j_1}(\{r\}), ..., f_{j_{k_j}}(\{r\}))\}$;

// Step 3: Build final result table $X$
$X = B(\mathbf{B}) \times \{(v_1, ..., v_m)\}$;
foreach $x \in X$ do
    for $i = 1$ to $m$ do
        $\tilde{X}_{[x,\theta_i]} \leftarrow \{\tilde{x} \in \tilde{X}^j \mid \mathbf{R}_i = \mathbf{R}_j \wedge \theta_i(\tilde{x}, x)\}$;
        $x.C_i \leftarrow g_i(\tilde{X}_{[x,\theta_i]})$;

return $X$;
```

4 Parallel Computation of θ-MDA Queries

In the previous section none of the presented algorithms take advantage of the current evolution of modern computer architectures. The evaluation strategies are executed by a single thread regardless of the number of supported threads of the current machine where the algorithm is executed. This leads to a situation where a single core of the computing machine performs the entire computation of the θ-MDA query whereas the remaining cores are idle and a huge potential computing power is wasted.

In this section we introduce the parallel computation of the *IndexedTCMDA* and the *TCMDA*$^+$ algorithm. We give a formal definition and illustrate both strategies on a schematic overview together with the evaluation of both evaluation strategies on the running example.

4.1 Parallel *BasicTCMDA/IndexedTCMDA*

In the following section we introduce the steps in order to parallelize the evaluation algorithms *BasicTCMDA* and *IndexedTCMDA*.

4.1.1 Partitioning Strategies

The computation of *BasicTCMDA/IndexedTCMDA* can be parallelized using several partitioning strategies.

Partitioning of the Detail Table. The evaluation of *BasicTCMDA* and *IndexedTCMDA* can be parallelized using partitioning of the detail table R in P partitions, i.e., $R = R_1 \cup \dots \cup R_P$, where P is the number of threads or processor units. This partitioning strategy requires that every processor receives the entire base table B, because every tuple in the detail table $r \in R$ may contribute to every tuple in the base table $b \in B$. As a consequence, the main memory requirements of the algorithm increase. In addition, the database facilities must be physically separated either on different disks or on different servers. Otherwise, reading from the database in parallel has a negative influence on the reading performance on a single machine but can be moderated by a database cache controller [49]. Since the partitioning of the detail table R introduces unknown circumstances outside of the actual algorithm, we do not take this strategy into consideration.

Partitioning of the Aggregate Functions. The evaluation of the aggregate functions f_{i_j} to be computed could be partitioned among the processors where each processor evaluates a *single* aggregate function f_{i_j} in the corresponding base table B_i. This requires that the workload to evaluate the specified function is more or less similar. If the number of the aggregate functions f_{i_j} to be computed is smaller than the number of processors P, the workload is not efficiently distributed because not all processors are used.

Partitioning of the Base Table. The evaluation of *BasicTCMDA* and *IndexedTCMDA* can be parallelized using partitioning of the base table B in P partitions where P is the number of threads or processor units. The number of processors P can either be specified by the user or is a physical boundary of the current machine where the θ-MDA evaluation algorithms are executed. The partitioning of B in P parts can be applied since $B = B_1 \cup \cdots \cup B_P$ where every processor can evaluate each partition in isolation [10]. This strategy does not suffer from the drawbacks of the partitioning strategies introduced before, because the workload of the computation can be distributed evenly among the available processors.

4.2 Partitioning of the Base Table

In the following we give a formal definition for parallelizing the θ-MDA operator by partitioning the base table B in P partitions.

Proposition 1. (***Parallelized θ-MDA Operator (IndexedTCMDA)***) Let $B(\mathbf{B})$ and $R(\mathbf{R})$ tables, θ_i be conditions with $attr(\theta_i) \subseteq \mathbf{B} \cup \mathbf{R}$, and $l_i = (f_{i_1} \to C_{i_1}, ..., f_{i_{k_i}} \to C_{i_{k_i}})$ with $1 \leq i \leq m$, be a list of aggregate functions over attributes $A_{i_1}, A_{i_2}, ..., A_{i_{k_i}}$ in $\mathbf{R}$. Further, let $B = B_1 \cup \cdots \cup B_P$ with $B_1 \cap \cdots \cap B_P = \emptyset$, where P denotes the number of processors with $1 \leq p \leq P$. Then $X = \mathcal{G}^\theta(B, R, (l_1, ..., l_m), (\theta_1, ..., \theta_m))$ can be computed with the following parallel evaluation strategy:

1. compute P θ-MDA queries with different base tables:

$$X_p = \mathcal{G}^\theta(B_p, R, (l_1, ..., l_m), (\theta_1, ..., \theta_m)) \text{ with } 1 \leq p \leq P,$$

   where $\mathbf{X} = (\mathbf{B}, C_1, ..., C_{1_{k_1}}, ..., C_{m_1}, ..., C_{m_{k_m}})$ is the schema of the result table and each tuple $b \in B_p$ produces a result tuple $x \in X_p$ with

$$\begin{aligned} &x.\mathbf{B} = b.\mathbf{B}, \\ &x.C_{i_j} = f_{i_j}(\{r.A_{i_j} | r \in R \wedge \theta_i(b, r)\}), \forall C_{i_j} \in \mathbf{X}; \end{aligned}$$

2. construct the final result table X:

$$X = X_1 \cup \cdots \cup X_P.$$

Figure 5 illustrates the detail table R together with the partitioning of the base table B and the final result table X. Initially, the algorithm divides the base table B into P partitions and constructs the partitions of the result table X_p from the partitions of the base table B_p. Next, for every tuple in the detail table $r \in R$ the aggregates in the partitions of the final result table X_p are updated by a single processor in isolation. Lastly, the final result table X is constructed by combining the partitions of the final result tables X_p, i.e., $X = X_1 \cup \cdots \cup X_P$.

Theorem 1. *The evaluation strategy proposed in Proposition 1 correctly computes θ-MDA.*

Proof 1. First of all, the base table B is partitioned into P disjoint partitions, i.e., $B = B_1 \cup \cdots \cup B_P$ with $B_1 \cap \cdots \cap B_P = \emptyset$ where P denotes the number of processors with $1 \leq p \leq P$. Next, for every partition of the base table B_p the θ-MDA operator $X_p = \mathcal{G}^\theta(B_p, R, (l_1, ..., l_m), (\theta_1, ..., \theta_m))$ is applied which computes the result tables X_p for the corresponding partition of the base table B_p. Finally, the partitions of the result tables X_p are combined in order to

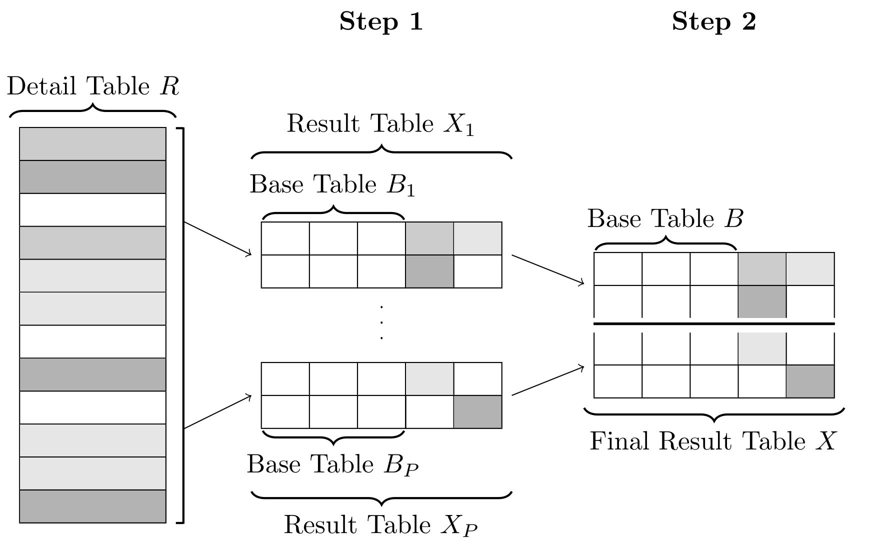
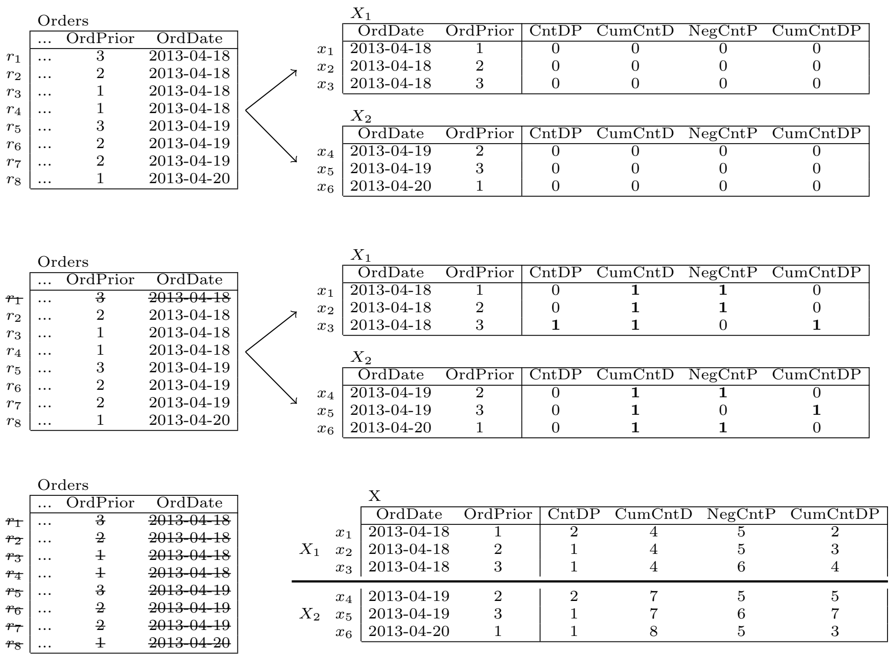


Figure 5: IndexedTCMDA: Parallel Computation of the θ-MDA Operator

obtain the final result table, i.e., $X = X_1 \cup \cdots \cup X_P$. Since for every partition of the detail table B_p the entire detail table R is considered and the union of the result tables $X = X_1 \cup \cdots \cup X_P$ does not add spurious tuples in the final result table X because of the disjoint partitioning, the evaluation strategy proposed in Proposition 1 correctly computes θ-MDA.

Note that each partition X_p can be computed independently from other partitions and hence allows the distribution of the workload each partition requires. We further assume that each partition of the base table B_p requires almost identical processing time, i.e., the workload of a single processor in order to create the result partition X_p from the base table partition B_p can be compared. Overall, this leads to a nearly perfect distribution of the workload required in order to complete each result partition X_p among the computing threads or processors. The number of tuples in each partition will differ by at most one since the number of tuples in the final result table X divided by the number of processors P determines the number of tuples of each partition $|X_p| = \lfloor |X|/P \rfloor$. Afterwards, the remaining tuples of the final result table $|X| - \lfloor |X|/P \rfloor P$ can be equally distributed among the partitions X_p. Finally, each partition contains $|X_p| = \lfloor |X|/P \rfloor + sgn(\lfloor (|X| - \lfloor |X|/P \rfloor P)/p \rfloor)$ number of tuples.

Example 4. In the following example, we illustrate the steps of the strategy in order to evaluate *Query 1* given in Proposition 1 in parallel. The partitioning process and the step by step evaluation of *Query 1* is shown in Figure 6. For the sake of simplicity we assume two processing units $P = 2$. At the beginning of the computation the base table B and the final result table X are split into two partitions $B = B_1 \cup B_2$ and $X = X_1 \cup X_2$, respectively. In this example, the two final result tables contain the same number of tuples to be processed, i.e., $|X_1| = |X_2| = 3$, where $|X_1|$ and $|X_2|$ denote the number of tuples in X_1 and X_2, respectively. After the partitioning, all aggregate functions of each partition are initialized with their corresponding initial values. Next, each tuple of the Orders table is processed, where all the aggregate attributes of the affected tuples are updated in the partitions of the result table X_p. In conclusion, the final result table X is obtained by combining the partitions of the final result tables X_p, i.e., $X = X_1 \cup \cdots \cup X_P$.



Orders

	...	OrdPrior	OrdDate
r_1	...	3	2013-04-18
r_2	...	2	2013-04-18
r_3	...	1	2013-04-18
r_4	...	1	2013-04-18
r_5	...	3	2013-04-19
r_6	...	2	2013-04-19
r_7	...	2	2013-04-19
r_8	...	1	2013-04-20

X_1

	OrdDate	OrdPrior	CntDP	CumCntD	NegCntP	CumCntDP
x_1	2013-04-18	1	0	0	0	0
x_2	2013-04-18	2	0	0	0	0
x_3	2013-04-18	3	0	0	0	0

X_2

	OrdDate	OrdPrior	CntDP	CumCntD	NegCntP	CumCntDP
x_4	2013-04-19	2	0	0	0	0
x_5	2013-04-19	3	0	0	0	0
x_6	2013-04-20	1	0	0	0	0

Orders

	...	OrdPrior	OrdDate
~~r_1~~	...	~~3~~	~~2013-04-18~~
r_2	...	2	2013-04-18
r_3	...	1	2013-04-18
r_4	...	1	2013-04-18
r_5	...	3	2013-04-19
r_6	...	2	2013-04-19
r_7	...	2	2013-04-19
r_8	...	1	2013-04-20

X_1

	OrdDate	OrdPrior	CntDP	CumCntD	NegCntP	CumCntDP
x_1	2013-04-18	1	0	**1**	**1**	0
x_2	2013-04-18	2	0	**1**	**1**	0
x_3	2013-04-18	3	**1**	**1**	0	**1**

X_2

	OrdDate	OrdPrior	CntDP	CumCntD	NegCntP	CumCntDP
x_4	2013-04-19	2	0	**1**	**1**	0
x_5	2013-04-19	3	0	**1**	0	**1**
x_6	2013-04-20	1	0	**1**	**1**	0

Orders

	...	OrdPrior	OrdDate
~~r_1~~	...	~~3~~	~~2013-04-18~~
~~r_2~~	...	~~2~~	~~2013-04-18~~
~~r_3~~	...	~~1~~	~~2013-04-18~~
~~r_4~~	...	~~1~~	~~2013-04-18~~
~~r_5~~	...	~~3~~	~~2013-04-19~~
~~r_6~~	...	~~2~~	~~2013-04-19~~
~~r_7~~	...	~~2~~	~~2013-04-19~~
~~r_8~~	...	~~1~~	~~2013-04-20~~

X

		OrdDate	OrdPrior	CntDP	CumCntD	NegCntP	CumCntDP
	x_1	2013-04-18	1	2	4	5	2
X_1	x_2	2013-04-18	2	1	4	5	3
	x_3	2013-04-18	3	1	4	6	4
	x_4	2013-04-19	2	2	7	5	5
X_2	x_5	2013-04-19	3	1	7	6	7
	x_6	2013-04-20	1	1	8	5	3

Figure 6: IndexedTCMDA: Parallel Computation of Query 1

In Example 1, the first tuple of the Orders table r_1 requires 13 updates in the final result table X. In this example, the number of updates are divided almost equally among the partitions, where the first partition X_1 requires 6 and the second partition X_2 needs 7 updates. Hence, the workload of the processors computing the partitions is balanced. Nevertheless, the parallel execution of the θ-MDA query introduces also costs. First, the partitioning process adds a small overhead, i.e., the time needed to partition the final result X into the partitions $X_1, \ldots, X_P$. The overhead in order to combine the partitions $X = X_1 \cup \cdots \cup X_P$ is negligible since each partition can be written in succession to disc. Additionally, in the case of *IndexedTCMDA* for each partition X_p a separate index has to be created and maintained. On the other hand, the different indexes on the partitions X_p are smaller and hence can be scanned more efficiently compared to the single index constructed on the final result table X.

4.3 Parallel *TCMDA*$^+$

In this section we illustrate the strategies to parallelize the optimized evaluation algorithm *TCMDA*$^+$.

4.3.1 Partitioning Strategies

The computation of $TCMDA^+$ can be parallelized using several partitioning strategies.

Vertical Partitioning of the Intermediate Result Tables. The construction of the separate intermediate result tables could be divided, where each processor computes its single intermediate result table. This partitioning strategy requires that the workload to complete the separate intermediate result tables $\tilde{X}^i$ is balanced, i.e., the number of tuples and the number of aggregate functions f_{i_j} to be computed in the separate intermediate tables $\tilde{X}^i$ are comparable. Since the size of an intermediate result table $|\tilde{X}^i|$ increases the maintenance cost of the corresponding hash index, the individual construction time of the separate intermediate result tables $\tilde{X}^i$ differs strongly. In addition, if the number of processors P exceeds the number of intermediate result tables $\tilde{X}^i$, the workload is not effectively distributed since the supernumerous processors are running idle. As a consequence, the composition of the final result table X has to be postponed until all processors have completed their separate intermediate result table $\tilde{X}^i$. Despite the increased number of tuples in the intermediate result tables $|\tilde{X}^i|$, the corresponding hash index can be accessed in constant time in order to identify a tuple which has to be updated.

Partitioning of the Aggregate Functions. The evaluation of the aggregate functions f_{i_j} to be computed could be partitioned among the processors, where each processor evaluates a *single* aggregate function f_{i_j} in the according intermediate result table $\tilde{X}^i$. As in the first partition strategy, this requires that the workload in order to evaluate the specified function is similar. This partitioning strategy encounters the same drawback as the vertical partitioning of the intermediate result tables presented before. If the number of the aggregate functions f_{i_j} to be computed fall below the number of processors P, the workload is not efficiently distributed because the supernumerary processors are not in operation.

Horizontal Partitioning of the Separate Intermediate Result Tables. In order to distribute the workload among the processing units and in order to balance the computational effort we propose a partitioning of the separate intermediate result tables $\tilde{X}^i$ in P partitions. Again, P is the number of processors and the partitioning of the separate intermediate result tables can be applied since $\tilde{X}^i = \tilde{X}^i_1 \cup \cdots \cup \tilde{X}^i_P$. This partitioning strategy does not encounter the drawbacks of the partitioning strategies presented before. The workload of the computation can be distributed efficiently among the processors, because the number of tuples in the separate intermediate result tables never fall below the number of processors P.

4.3.2 Horizontal Partitioning of the Intermediate Result Tables

Proposition 2 introduces the parallelized θ-MDA Operator with separate intermediate result tables using reduction to point aggregates.

Proposition 2. (***Parallelized θ-MDA Operator (TCMDA$^+$)***) Let $B(\mathbf{B})$ and $R(\mathbf{R})$ tables, θ_i be conditions with $attr(\theta_i) \subseteq \mathbf{B} \cup \mathbf{R}$, and $l_i = (f_{i_1} \to C_{i_1}, ..., f_{i_{k_i}} \to C_{i_{k_i}})$ with $1 \leq i \leq m$, be a list of aggregate functions over attributes $A_{i_1}, A_{i_2}, ..., A_{i_{k_i}}$ in $\mathbf{R}$, $G = (g_{i_1}, ..., g_{i_{k_i}})$ be the corresponding super aggregates [33]. Let $B = B_1 \cup \cdots \cup B_P$ with $B_1 \cap \cdots \cap B_P = \emptyset$ and $R = R_1 \cup \cdots \cup R_P$ with $R_1 \cap \cdots \cap R_P = \emptyset$ where P denotes the number of processors. Furthermore, let $\mathbf{R}_i = \mathbf{R} \cap attr(\theta_i)$ denote the attributes in $\mathbf{R}$ that occur in θ_i. Then the parallelized $\mathcal{G}^\theta(B, R, (l_1, ..., l_m), (\theta_1, ..., \theta_m))$ with separate intermediate result tables using reduction to point aggregates can be computed as follows:

1. construct $\tilde{\Theta} = (\tilde{\theta}_1, ..., \tilde{\theta}_m)$:
$$\tilde{\theta}_i(r, b) = \bigwedge_{A \in \mathbf{R}_i} r.A = b.A \text{ with } r \in R, b \in B;$$
2. compute $P \cdot m$ intermediate result tables:
$$\tilde{X}_p^i = \mathcal{G}^\theta(\pi_{\mathbf{R_i}}(R), R_p, l_i, \tilde{\theta}_i) \text{ for } i = 1, ..., m \text{ and } p = 1, ..., P;$$
3. compute P result tables:
$$X_p = \{b \circ f | b \in B_p \wedge f = (g_{i_1}(\tilde{X}^1_{p[b,\tilde{\theta}_1]}), ..., g_{i_{k_i}}(\tilde{X}^i_{p[b,\tilde{\theta}_m]}))\},$$
where $\tilde{X}^i_{p[b,\tilde{\theta}_i]} = \pi_{\mathbf{R}_i, C_{i_j}}\{\tilde{x} \in \tilde{X}^i_p | \theta_i(\tilde{x}, b)\}$;
4. compute the final result table X
$$X = X_1 \cup \cdots \cup X_P.$$

Figure 7 depicts the partitioning of the detail table R together with the partitioning of the separate intermediate result tables $\tilde{X}_p^i$. First of all, the detail table R is divided into P partitions, i.e., $R_1 \cup \cdots \cup R_P$ with $R_1 \cap \cdots \cap R_P = \emptyset$. Next, from a *single* partition of the detail table R_p the separate intermediate result tables $\tilde{X}_p^i$ are constructed. Note the difference to the partitioning strategy described in Section 4.1 where the tuples of the detail table $r \in R$ contribute to *every* partition of the final result table X_p. The partitioning of the detail table R can be applied since the groups of the partitioned separate intermediate result tables $\tilde{X}_p^i$ are produced by a projection $\pi_{\mathbf{R}_i}$ where $\pi_{\mathbf{R}_{i_1}} \cup \cdots \cup \pi_{\mathbf{R}_{i_P}}$. In the next stage, for every tuple in the partitions of the detail table $r \in R_p$ the aggregates in the partitions of the separate intermediate result tables $\tilde{X}_p^i$ are updated by a single processor in isolation using the conditions containing only equality constraints $\tilde{\theta}$. After the completion of the partitions of the separate intermediate result tables $\tilde{X}_p^i$, the final result table X is constructed where *every* partition of the separate intermediate result tables $\tilde{X}_p^i$ contribute to *every* partition of the final result table X_p because of the initial partitioning of the detail table R. Finally, the final result table is constructed by combining the partitions of the result tables $X = X_1 \cup \cdots \cup X_P$.

Theorem 2. *The evaluation strategy proposed in Proposition 2 correctly computes θ-MDA.*

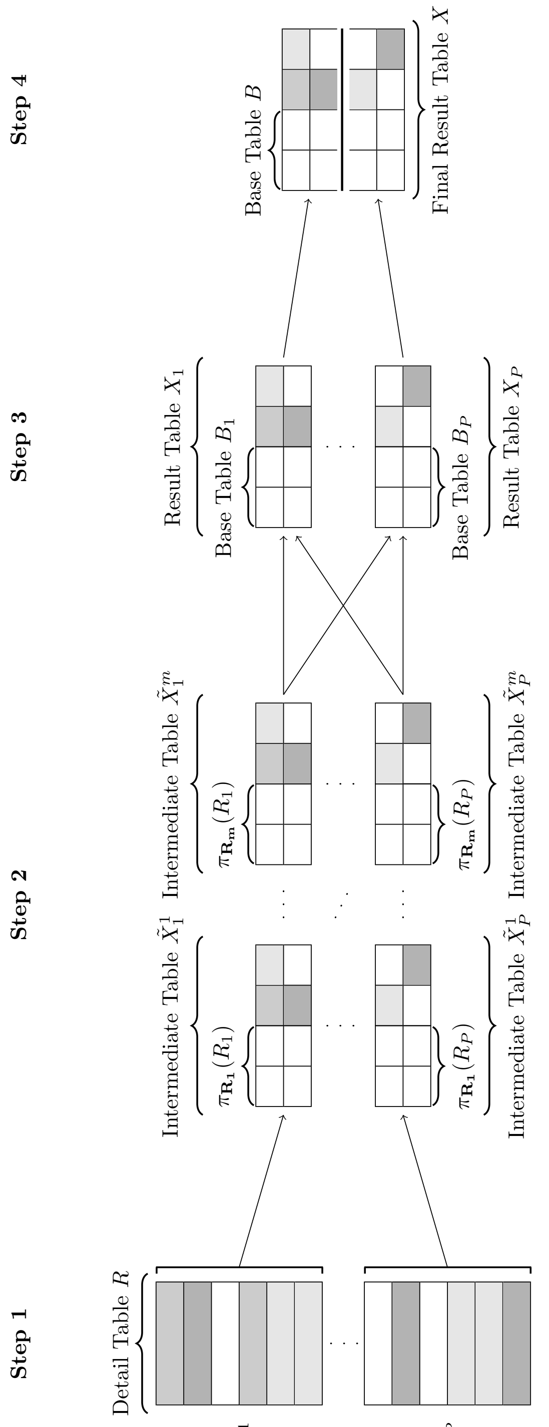

Step 1
Detail Table R
R_1
R_P
Step 2
Intermediate Table $\tilde{X}_1^1$
$\pi_{\mathbf{R_1}}(R_1)$
$\pi_{\mathbf{R_1}}(R_P)$
Intermediate Table $\tilde{X}_P^1$
Intermediate Table $\tilde{X}_1^m$
$\pi_{\mathbf{R_m}}(R_1)$
$\pi_{\mathbf{R_m}}(R_P)$
Intermediate Table $\tilde{X}_P^m$
Step 3
Result Table X_1
Base Table B_1
Base Table B_P
Result Table X_P
Step 4
Base Table B
Final Result Table X

Figure 7: TCMDA$^+$: Parallel Computation of the θ-MDA Operator

Proof 2. The first step in Proposition 2 in order to create the equality constraint conditions $\tilde{\theta}_i$ is the same as in the previous definition of *TCMDA*$^+$. Next, the detail table R is partitioned into P partitions, i.e., $R = R_1 \cup \cdots \cup R_P$ with $R_1 \cap \cdots \cap R_P = \emptyset$ where P denotes the number of processors with $1 \leq p \leq P$. Then, the θ-MDA operator is applied to every partition of the detail table R_p using the equality constraint conditions, i.e., $\tilde{X}^i_p = \mathcal{G}^\theta(\pi_{\mathbf{R_i}}(R), R_p, l_i, \tilde{\theta}_i)$ for $i = 1, ..., m$ and $p = 1, ..., P$, in order to obtain the partitions of the separate intermediate result tables $\tilde{X}^i_p$. As a result, the equality constraint conditions $\tilde{\theta}_i$ are applied on the entire detail table R. In the next stage, the base table B is partitioned into P disjoint partitions, i.e., $B = B_1 \cup \cdots \cup B_P$ with $B_1 \cap \cdots \cap B_P = \emptyset$ with $1 \leq p \leq P$. Now, the result tables X_p are constructed using the original constraint conditions θ_i together with the partitions of the detail table B_p and the partitions of the separate intermediate result tables $\tilde{X}^i_p$. Finally, the final result table is constructed by combining the previously created result tables $X = X_1 \cup \cdots \cup X_P$. On account of the union does not add auxiliary tuples in the final result table X because of the disjoint partitioning and seeing that the entire detail table R and the entire partitions of the separate intermediate tables $\tilde{X}^i_p$ are considered, the evaluation strategy proposed in Proposition 2 correctly computes θ-MDA.

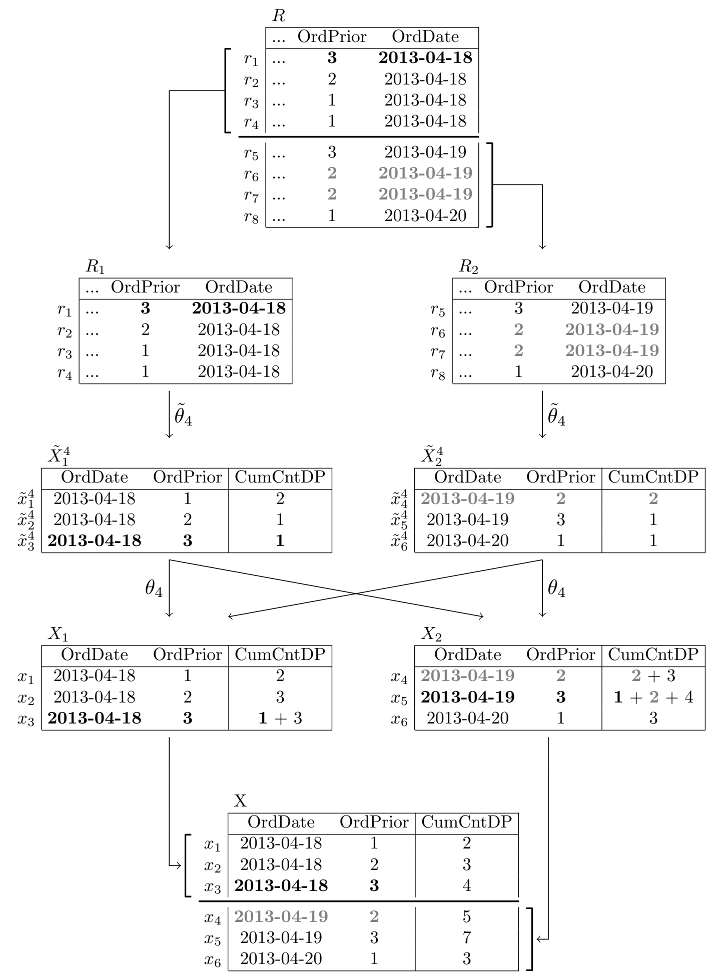
In contrast to the partitioning strategy described in Section 4.1 where the tuples of the detail table $r \in R$ contribute to *every* partition of the final result table X_p, the tuples of the detail table $r \in R$ in this strategy contribute to a *single* intermediate result table $\tilde{X}^i$. As a consequence, the detail table R can be divided into P partitions, $R = R_1 \cup \cdots \cup R_P$ with $R_1 \cap \cdots \cap R_P = \emptyset$, where each processor computes the separate intermediate result table $\tilde{X}^i_p$ from the corresponding partition of the detail table R_p. This does not imply that the separate intermediate result tables $\tilde{X}^i_p$ are disjoint, i.e., $\tilde{X}^1_1 \cap \cdots \cap \tilde{X}^m_P \neq \emptyset$.

In practice, the number of distinct grouping values included in the constraint conditions $\mathbf{R}_i = \mathbf{R}_i \cap attr(\tilde{\theta}_i)$ are small and overlap with the different partitions of the detail table R_p, i.e., $\pi_{\mathbf{R}_1} \cap \cdots \cap \pi_{\mathbf{R}_P} \neq \emptyset$, since the tuples of the detail table R are distributed randomly. As a result, the number of tuples in the separate intermediate result tables $\tilde{X}^i_p$ are equal containing the evaluation of the different aggregate functions.

Example 5. In the next example, we present the different steps in order to evaluate *Query 1* given in Proposition 1 in parallel using reduction to equality constraints together with separate intermediate result tables. For the sake of convenience we assume two processors $P = 2$ and illustrate the computation on the separate intermediate table $\tilde{X}^4$ from Example 3. Figure 8 shows the processing steps without the initial partitioning of the detail table R and the final result table X. In order to enhance reading and to indicate their contribution to the final result, the tuples r_1, r_6 and r_7 from the partitions of the detail tables R_1 and R_2 are written in bold and in different colors.

At the beginning, the detail table is split into two partitions $R = R_1 \cup R_2$ with $R_1 \cap R_2 = \emptyset$ where each partition contains the same number of tuples, i.e., $|R_1| = |R_2|$. After the partitioning of the detail table R, two separate intermediate result table partitions $\tilde{X}^4_1$ and $\tilde{X}^4_2$ are created in order to compute the projection $\pi_{\mathbf{R}}$ where $\mathbf{R}_i = \mathbf{R} \cap attr(\theta_i)$ denotes the attributes $\mathbf{R}$ that occur in θ_i. For instance, the tuples r_6 and r_7 from detail table R_2 are projected to the single tuple $\tilde{x}_4^4$ in the partition of the separate intermediate result table $\tilde{X}_4^4$. After the completion of the partitions of the separate intermediate result tables $\tilde{X}_1^4$ and $\tilde{X}_2^4$ the partitions of the final result tables X_1 and X_2 are computed using the original conditions θ_i. In the example, tuple $\tilde{X}_3^4$ from the first partition of the separate intermediate result table $\tilde{X}_1^4$ contributes to tuples of both partitions of the final result tables $x_3 \in X_1$ and $x_5 \in X_2$. In the end, the final result table X is obtained by combining the partitions of the final result tables X_p, i.e., $X = X_1 \cup \cdots \cup X_P$.

As in Example 4, the required number of updates are divided almost equally among the partitions of the separate intermediate result tables X_p^i and the final result tables X_p. As a consequence, the workload of the processors generating the different partitions is balanced in spite of the maintenance costs of the partitions and the indexes of the separate intermediate result tables X_p^i and the final result tables X_p.



R

	...	OrdPrior	OrdDate
r_1	...	**3**	**2013-04-18**
r_2	...	2	2013-04-18
r_3	...	1	2013-04-18
r_4	...	1	2013-04-18
r_5	...	3	2013-04-19
r_6	...	2	2013-04-19
r_7	...	2	2013-04-19
r_8	...	1	2013-04-20

R_1

	...	OrdPrior	OrdDate
r_1	...	**3**	**2013-04-18**
r_2	...	2	2013-04-18
r_3	...	1	2013-04-18
r_4	...	1	2013-04-18

R_2

	...	OrdPrior	OrdDate
r_5	...	3	2013-04-19
r_6	...	2	2013-04-19
r_7	...	2	2013-04-19
r_8	...	1	2013-04-20

$\tilde{X}_1^4$

	OrdDate	OrdPrior	CumCntDP
$\tilde{x}_1^4$	2013-04-18	1	2
$\tilde{x}_2^4$	2013-04-18	2	1
$\tilde{x}_3^4$	**2013-04-18**	**3**	**1**

$\tilde{X}_2^4$

	OrdDate	OrdPrior	CumCntDP
$\tilde{x}_4^4$	2013-04-19	2	2
$\tilde{x}_5^4$	2013-04-19	3	1
$\tilde{x}_6^4$	2013-04-20	1	1

X_1

	OrdDate	OrdPrior	CumCntDP
x_1	2013-04-18	1	2
x_2	2013-04-18	2	3
x_3	**2013-04-18**	**3**	**1** + 3

X_2

	OrdDate	OrdPrior	CumCntDP
x_4	2013-04-19	2	2 + 3
x_5	**2013-04-19**	**3**	**1** + 2 + 4
x_6	2013-04-20	1	3

X

	OrdDate	OrdPrior	CumCntDP
x_1	2013-04-18	1	2
x_2	2013-04-18	2	3
x_3	**2013-04-18**	**3**	4
x_4	2013-04-19	2	5
x_5	2013-04-19	3	7
x_6	2013-04-20	1	3

Figure 8: TCMDA$^+$: Parallel Computation of Query 1

5 Reducing the Intermediate Tables to SQL

The optimization approach of using separate intermediate result tables in order to reduce redundant updates presented in [13] computes the entire intermediate result tables in main memory using an index which is created on the fly. This strategy requires that for each tuple $r \in R$ the created index has to be consulted in order to check whether the tuple is present or not. In addition, if the currently scanned tuple is not present in the index, the currently updated intermediate result table and the index have to be updated.

Since the conditions $\tilde{\theta}_i$ contain only equality constraints the most efficient index in order to identify already present tuples in the intermediate result tables is a hash index [50] where the index needs to be reorganized or recreated in order to avoid large overflow chains which decrease the performance of the hash index [50]. The computation of the separate intermediate result tables is based on a projection $\pi_{\mathbf{R_i}}(R)$, where $\mathbf{R}_i = \mathbf{R} \cap attr(\theta_i)$ denotes the attributes in $\mathbf{R}$ that occur in θ_i using the generated equality constraints $\tilde{\theta}_i$. This can be transformed to a SQL *GROUP BY* statement and handed over to the DBMS. These SQL *GROUP BY* statements can be optimized and executed efficiently by highly optimized database facilities. This avoids the creation and maintenance of the separate intermediate result tables together with the hash index in the actual algorithm. Basically, the algorithm retrieves the intermediate result tables $\tilde{X}^i$ generated by the DBMS and continues with the generation of the final result table X using the original conditions θ_i.

In the next sections we present the parallel computation of the *TCMDA*$^+$ with separate intermediate result tables using reduction to SQL. First of all, a formal definition together with a schematic overview in order to enhance understanding is given. Next, the correctness of the newly introduced evaluation strategy is shown. Finally, the presented approach is evaluated on the running example.

In the following we introduce the reduction of the computation of the separate intermediate result tables using equality constraint conditions to SQL.

Proposition 3. (*Reducing Intermediate Result Tables to SQL (TCMDA*$^+$*)*) Let $B(\mathbf{B})$ and $R(\mathbf{R})$ tables, θ_i be conditions with $attr(\theta_i) \subseteq \mathbf{B} \cup \mathbf{R}$, and $l_i = (f_{i_1} \rightarrow C_{i_1}, ..., f_{i_{k_i}} \rightarrow C_{i_{k_i}})$ with $1 \leq i \leq m$, be a list of aggregate functions over attributes $A_{i_1}, A_{i_2}, ..., A_{i_{k_i}}$ in $\mathbf{R}$, $G = (g_{i_1}, ..., g_{i_{k_i}})$ be the corresponding super aggregates [33]. Let $B = B_1 \cup \cdots \cup B_P$ with $B_1 \cup \cdots \cup B_P = \emptyset$ where P denotes the number of processors. Furthermore, let $\mathbf{R}_i = \mathbf{R} \cap attr(\theta_i)$ denote the attributes in $\mathbf{R}$ that occur in θ_i. Then the parallelized $\mathcal{G}^{\theta}(B, R, (l_1, ..., l_m), (\theta_1, ..., \theta_m))$ with separate intermediate result tables using reduction to SQL can be computed as follows:

1. compute m intermediate result tables using SQL:

```
$\tilde{X}^i$ = SELECT $\mathbf{R}_i$,
        $f_{i_1}(R.A_{i_1})$ AS $C_{i_1}$,
        ...,
        $f_{i_{k_i}}(R.A_{i_{k_i}})$ AS $C_{i_{k_i}}$
     FROM $R$
     GROUP BY $\mathbf{R}_i$
```

   with $1 \leq i \leq m$;

2. compute P result tables:

$$X_p = \{b \circ f | b \in B_p \wedge f = (g_{i_1}(\tilde{X}^1_{[b,\theta_1]}), ..., g_{i_{k_i}}(\tilde{X}^i_{[b,\theta_m]}))\},$$

where $\tilde{X}^i_{[b,\theta_i]} = \{\tilde{x} \in \tilde{X}^i | \theta_i(\tilde{x}, b)\}$ and $1 \leq p \leq P$;

3. construct the final result table X:

$$X = X_1 \cup \cdots \cup X_P.$$

Theorem 3. *The evaluation strategy proposed in Proposition 3 correctly computes θ-MDA.*

Proof 3. The computation of the separate intermediate tables $\tilde{X}^i$ is based on the θ-MDA operator using equality constraint conditions $\tilde{\theta}$ and a projection of the detail table $\pi_{\mathbf{R}_i}$ where $\mathbf{R}_i = \mathbf{R} \cap attr(\theta_i)$. The projection of the detail table $\pi_{\mathbf{R}_i}$ is a relational operator and can be transformed to SQL where the entire detail table R is considered. On account of Proposition 2 applies the θ-MDA operator using equality constraint conditions $\tilde{\theta}$ on the projection of the detail table $\pi_{\mathbf{R}_i}$, the evaluation of the θ-MDA operator can be computed by a single *GROUP BY* [10]. Hence, the reduction of the computation of the separate intermediate result tables $\tilde{X}^i$ in Proposition 3 is correct.

After the computation of the separate intermediate result tables $\tilde{X}^i$, the base table B is partitioned into P disjoint partitions, i.e., $B = B_1 \cup \cdots \cup B_P$ with $B_1 \cap \cdots \cap B_P = \emptyset$ with $1 \leq p \leq P$. Next, the result tables X_p are constructed using the original constraint conditions θ_i together with the partitions of the detail table B_p and the separate intermediate result tables $\tilde{X}^i$. In the end, the final result table is constructed by combining the previously created result tables $X = X_1 \cup \cdots \cup X_P$. On account of the union does not add auxiliary tuples in the final result table X because of the disjoint partitioning and seeing that the entire detail table R and the entire partitions of the separate intermediate tables $\tilde{X}^i$ are considered, the evaluation strategy proposed in Proposition 3 correctly computes θ-MDA.

Figure 9 illustrates the evaluation strategy using the reduction of the intermediate result tables to SQL. At the beginning, the separate intermediate result tables $\tilde{X}^i$ are constructed using SQL statements on the entire detail table R. Note that the generated SQL statements are executed by the DBMS and therefore the evaluation strategy does not exert any influence on their execution, for instance, whether the DBMS executes the SQL statement on the same machine using a single thread or distributes the query over a database cluster. Since reading from the database in parallel has no or a negative influence on the reading performance on a single machine [49], the algorithm executes each SQL statement by a single processor. In addition, the computation of the separate intermediate result tables $\tilde{X}^i$ using SQL in the DBMS avoids the partitioning of the separate intermediate result tables $\tilde{X}^i_p = \tilde{X}^i_1 \cup \cdots \cup \tilde{X}^i_P$ due to the computation of the SQL statement is done over the entire detail table R.

In contrast to the evaluation strategy in Proposition 2 where the intermediate result tables $\tilde{X}^i$ together with the hash index are maintained in main memory, the separate intermediate result tables $\tilde{X}^i$ in Proposition 3 are reduced to SQL and its computation is outsourced to the DBMS. The generated

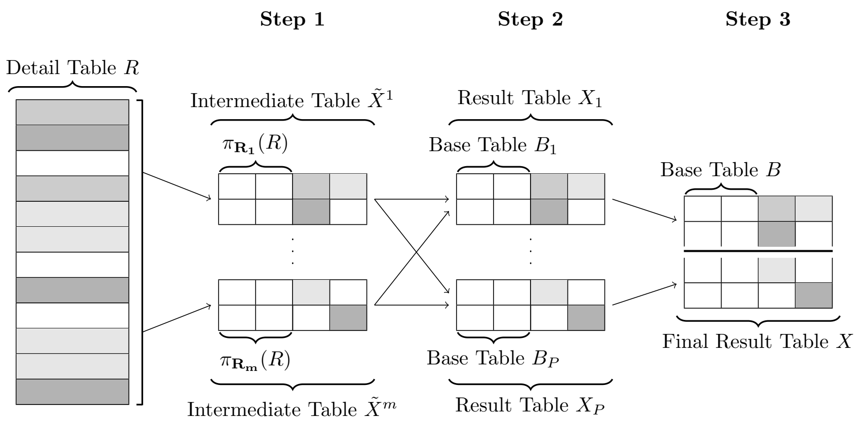


Figure 9: Reducing Intermediate Result Tables to SQL

SQL statement to compute the separate intermediate result tables $\tilde{X}^i$ can be optimized and executed efficiently by highly optimized database facilities. This eliminates the maintenance and update costs of the indexes on the separate intermediate result tables $\tilde{X}^i$. In addition, the theta conditions $\tilde{\theta}_i$ for the evaluation of the equality constraints on the separate intermediate result tables are not needed. The algorithm continues as in Proposition 2 with the computation of the partitions of the final result tables X_p from the base table partition B_p. Every partition X_p can be computed independently from other partitions and hence the workload required for each partition X_p is distributed among the processors. As in Proposition 2 we assume that each partition of the base table B_p requires almost identical processing time, i.e., nearly the same number of updates. In the end, the result tables X_p are combined to the final result table $X = X_1 \cup \dots \cup X_P$.

Example 6. In this example we present the evaluation of *Query 1* using reduction of group tables to SQL illustrated in Proposition 3. For the sake of convenience we assume two processors $P = 2$ and illustrate the computation on the separate intermediate table $\tilde{X}^4$ from Example 3. At the beginning the separate intermediate result tables $\tilde{X}^i$ are created using the following SQL statements.

$\tilde{X}^1 \equiv$
```
SELECT r.OrdDate, r.OrdPrior, COUNT(r.OrdDate, r.OrdPrior) CntDP
FROM Orders r GROUP BY r.OrdDate, r.OrdPrior
```

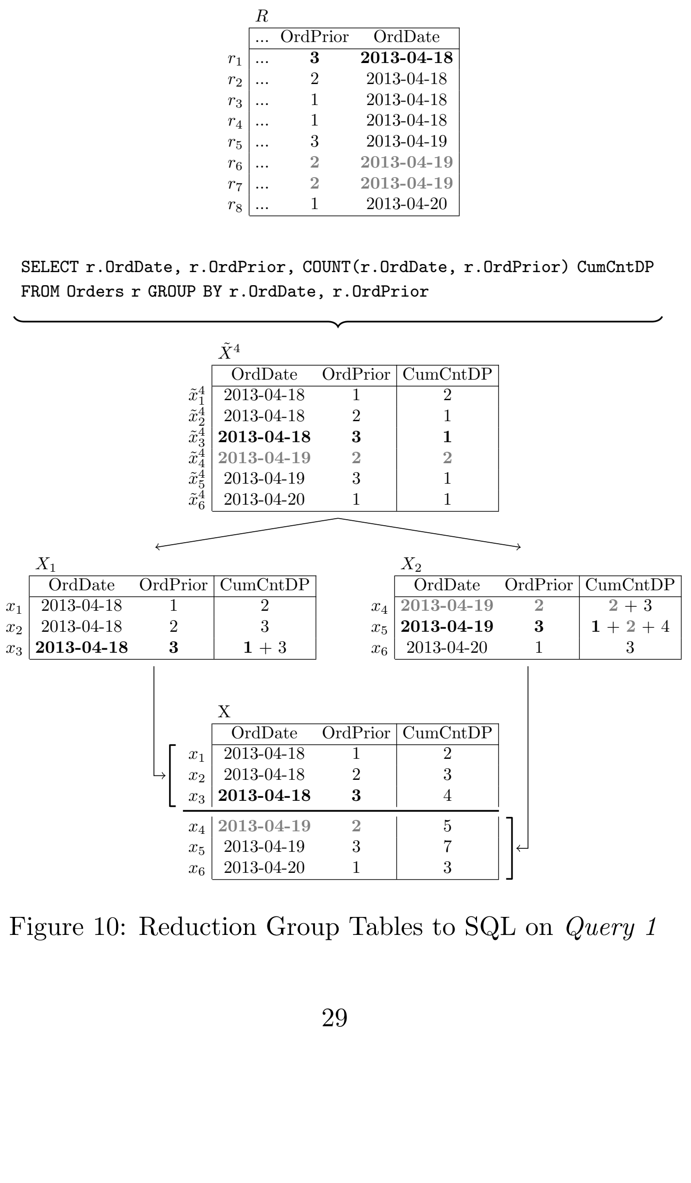
$\tilde{X}^2 \equiv$
```
SELECT r.OrdDate, COUNT(r.OrdDate) CumCntD
FROM Orders r GROUP BY r.OrdDate
```

$\tilde{X}^3 \equiv$
```
SELECT r.OrdPrior, COUNT(r.OrdPrior) NegCntP
FROM Orders r GROUP BY r.OrdPrior
```

$\tilde{X}^4 \equiv$
```
SELECT r.OrdDate, r.OrdPrior, COUNT(r.OrdDate, r.OrdPrior) CumCntDP
FROM Orders r GROUP BY r.OrdDate, r.OrdPrior
```

After the creation of the separate intermediate result tables $\tilde{X}^1, \dots, \tilde{X}^4$ by the DBMS, the partitions of the final result tables $X = X_1 \cup X_2$ are constructed and initialized. After the partitioning, each tuple of the separate intermediate result tables $\tilde{X}^1, \ldots, \tilde{X}^4$ is processed, where all the aggregate attributes of the affected tuples are updated in the partitions of the result table X_1 and X_2. Figure 10 illustrates the processing steps without the partitioning of the final result table X. To enhance reading and to indicate their contribution to the final result, the tuples r_1, r_6 and r_7 of the detail tables R are written in bold and in different colors. In the example, the tuples r_6 and r_7 from detail table R are projected to the single tuple $\tilde{x}_4^4$ in the separate intermediate result table $\tilde{X}^4$ and tuple $\tilde{X}_3^4$ contributes to tuples of both partitions of the final result tables $x_3 \in X_1$ and $x_5 \in X_2$. At the end, the final result table X is obtained by combining the partitions of the final result tables X_p, i.e., $X = X_1 \cup \cdots \cup X_2$.

The computation of the separate intermediate result tables $\tilde{X}^i$ from the detail table R is outsourced to the DBMS, where it can processed and transformed efficiently. This is the most time consuming operation in the algorithm, since the detail table R is typically large. The remaining number of updates required are divided almost equally among the partitions of the result tables X_p and as a result the workload of the processors generating the different partitions is balanced in spite of the maintenance costs of the partitions and the indexes of the result tables X_p.



R

	...	OrdPrior	OrdDate
r_1	...	**3**	**2013-04-18**
r_2	...	2	2013-04-18
r_3	...	1	2013-04-18
r_4	...	1	2013-04-18
r_5	...	3	2013-04-19
r_6	...	**2**	**2013-04-19**
r_7	...	**2**	**2013-04-19**
r_8	...	1	2013-04-20

```
SELECT r.OrdDate, r.OrdPrior, COUNT(r.OrdDate, r.OrdPrior) CumCntDP
FROM Orders r GROUP BY r.OrdDate, r.OrdPrior
```

$\tilde{X}^4$

	OrdDate	OrdPrior	CumCntDP
$\tilde{x}_1^4$	2013-04-18	1	2
$\tilde{x}_2^4$	2013-04-18	2	1
$\tilde{x}_3^4$	**2013-04-18**	**3**	**1**
$\tilde{x}_4^4$	**2013-04-19**	**2**	**2**
$\tilde{x}_5^4$	2013-04-19	3	1
$\tilde{x}_6^4$	2013-04-20	1	1

X_1

	OrdDate	OrdPrior	CumCntDP
x_1	2013-04-18	1	2
x_2	2013-04-18	2	3
x_3	**2013-04-18**	**3**	**1** + 3

X_2

	OrdDate	OrdPrior	CumCntDP
x_4	**2013-04-19**	**2**	**2** + 3
x_5	**2013-04-19**	**3**	**1** + **2** + 4
x_6	2013-04-20	1	3

X

	OrdDate	OrdPrior	CumCntDP
x_1	2013-04-18	1	2
x_2	2013-04-18	2	3
x_3	**2013-04-18**	**3**	4
x_4	**2013-04-19**	**2**	5
x_5	2013-04-19	3	7
x_6	2013-04-20	1	3

Figure 10: Reduction Group Tables to SQL on *Query 1*

6 Algorithms and Implementation

In this section we define the algorithms in order to compute θ-MDA queries in parallel based on the previously presented approaches in Section 4.1 for *IndexedTCMDA* and in Section 4.3 for *TCMDA*$^+$. Further, we present the algorithm in order to compute θ-MDA queries using the reduction of the separate intermediate result tables to SQL illustrated in Section 5. Basically, the introduced algorithms are modifications of the algorithms presented in Section 3 and their main structure is preserved. In Section 6.1 we provide additional details about the parallel *IndexedTCMDA* algorithm. In Section 6.2 we illustrate the parallel *TCMDA*$^+$ algorithm and in Section 6.3 we depict further details about the computation of the of the separate intermediate result tables using SQL. Finally, in 6.4 we present some details about the implementation of the θ-MDA algorithms.

6.1 Parallel *IndexedTCMDA* Algorithm

In this section we present the algorithm in order to compute θ-MDA queries in parallel based on the previously presented approach in Section 4.1. The introduced algorithm is called *IndexedPTCMDA* which stands for *Indexed Parallel θ-Constrained Multi-Dimensional Aggregation.* The algorithm requests an additional parameter P which indicates the number of partitions of the base table B. If the number of processors is equal one, i.e., $P = 1$, the algorithms *IndexedTCMDA* and *IndexedPTCMDA* are the same, since a single partition of the final result table $X = X_1$ is constructed. Note that *IndexedPTCMDA* does not natively distribute the workload of the partitions among the processors, i.e., the implementation is responsible to distribute the computation of the partitions of the final result table X_p, where every partition of the final result table X_p can be evaluated separately by a processor in isolation.

The first step of the algorithm is the same as in the original *IndexedTCMDA* algorithm defined on page 13 to compute the algebraic aggregates correctly. Step 2 to 4 of the proposed algorithm differ from the basic algorithm *IndexedTCMDA* in order to follow the proposed parallel execution strategy. These modifications are explained in more detail in the proceeding.

First, the algorithm splits the base table B into disjoint partitions $B = B_1 \cup \cdots \cup B_P$ with $B_1 \cap \cdots \cap B_P = \emptyset$, constructs the partitions of the final result table X_p from the partitions of the detail tables B_p and initializes the aggregation results. In addition, for each of the partitions of the final result table X_p an index is created in order to improve the retrieval time of the tuples during the computation of the aggregation results. Step 3 scans the detail table R and computes the aggregates for every partition of the final result table X_p using the created index over the conditions θ from step 2. Step 4 is a minor modification of the same step of the original algorithm *IndexedTCMDA*. In this step the aggregation results are computed by applying the super-aggregates to the values of the sub-aggregates. Instead of applying the super-aggregates to the entire final result table X as in the *IndexedTCMDA* algorithm, the *IndexedPTCMDA* algorithm executes the super-aggregates to every partition of the final result table X_p. In conclusion in order to complete the computation of the final result table X, the previously created partitions X_p can be combined according to step 2 of Proposition 1, i.e., $X = X_1 \cup \cdots \cup X_P$.

Algorithm 3: *IndexedPTCMDA*$(B, R, P, (l_1, ..., l_m), (\theta_1, ..., \theta_m))$

```
// Step 1: Replace algebraic aggregates by distributive sub-aggregates
(Same as step 1 in IndexedTCMDA)
// Step 2: Construct partitions of base table $B$ and result table $X$
// Note: $v_j$'s are the initial values of the aggregate functions (0 for sum and count,
   NULL for max and min)
Let $\mathbf{N} = (v'_{1_1}, ..., v'_{m_1}, ..., v'_{m_{k_m}})$;
Partition $B$ such that $B = B_1 \cup \cdots \cup B_P$ and $B_1 \cap \cdots \cap B_P = \emptyset$;
for $p = 1$ to $P$ do
    Let $X_p = B_p \times N$;
    Build indexes for $X_p$;
// Step 3: Compute the aggregates
foreach tuple $r \in R$ do
    for $p = 1$ to $P$ do
        foreach $\theta_i \in \{\theta_1, ..., \theta_m\}$ do
            Fetch the rows $X_{p_i} = \{x \in X_p | \theta_i(x, r)\}$ using the created index;
            foreach $x \in X_{p_i}$ do
                Update the aggregates $f_{i_1}, ..., f_{i_{k_i}}$ in $x$;
// Step 4: Apply the super-functions
(Same as step 4 in IndexedTCMDA applied on every partition of result table X)
// Step 5: Combine previously created partitions of result table X
$X = X_1 \cup \cdots \cup X_P$;
return $X$;
```

6.2 Parallel *TCMDA*$^+$ Algorithm

In the following section we present the algorithm to compute θ-MDA queries in parallel based on the previously presented approach in Section 4.3. The presented algorithm is called *PTCMDA*$^+$ which stands for *Parallel θ-Constrained Multi-Dimensional Aggregation Plus.* The algorithm requests an additional parameter P which indicates the number of partitions of the detail table R as well as the number of partitions of the base table B. As discussed in Section 6.1, *TCMDA*$^+$ and *PTCMDA*$^+$ execute the same steps, if the number of partitions is equal one, i.e., $P = 1$, and therefore, the entire workload of the computation is performed by a single processor. Since the algorithm provides only the partitioning strategy of the workload, the implementation is responsible to distribute the evaluation of the partitions of the detail table R_p and the final result table X_p among the available processors.

All the steps of the algorithm are basically the same as in the *TCMDA*$^+$ algorithm defined on page 16. The main difference between the *TCMDA*$^+$ and the *PTCMDA*$^+$ algorithm is that in every step of *PTCMDA*$^+$ algorithm the according operations are applied on every partition of the separate intermediate result tables $\tilde{X}_p^i$ or on every partition of the result tables X_p, respectively.

In the first step, the algorithm creates the schema $\mathbf{R}_i$ of the separate intermediate result tables $\tilde{X}^i$ for each condition $\tilde{\theta}_i$. This might lead to intermediate result tables $\tilde{X}^i$ with identical grouping attributes. These are merged by the algorithm to a single intermediate result table containing a column for each aggregate function. Next, P empty separate intermediate result tables $\tilde{X}_p^i$ together with the indexes on the empty separate intermediate result tables are constructed using the previously created schemas in order to allow the partitioning of the detail table R in the second step. The first step is completed after the creation of the equality constraint conditions $\tilde{\theta}_i$ for the partitions of the separate intermediate result tables $\tilde{X}_p^i$.

In the second step, the detail table R is partitioned into P partitions $R = R_1 \cup \cdots \cup R_P$ with $R_1 \cap \cdots \cap R_P = \emptyset$ in order to allow the division of the workload among the processors. Next, the partitions of the detail table R_p are scanned and for each tuple in the partitions of the detail table $r \in R_p$ the according partition of the separate intermediate result table $\tilde{X}_p$ is updated. If a tuple $\tilde{x}_i$ exists in the according partition of the intermediate result table $\tilde{X}_p^i$, then the entry is incrementally updated. Otherwise, a new entry in the partition of the intermediate result table $\tilde{X}_p^i$ is created and the aggregate value is initialized to the functions evaluated over the tuple of the according partition of the detail table $r \in R_p$.

In the third step, the algorithm splits the base table B into disjoint partitions $B = B_1 \cup \cdots \cup B_P$ with $B_1 \cap \cdots \cap B_P = \emptyset$, constructs the partitions of the final result table X_p from the partitions of the detail table B_p, creates an index for each of the partitions of the final result table X_p and initializes the aggregation results. The index is created in order to improve the retrieval time of the tuples during the computation of the aggregation results. Further, the partitions of the final result table X_p are built, where the aggregate attributes are initialized to their according neutral values v_i by combining the partial aggregates from the partitions of the intermediate result tables $\tilde{X}_p^i$ using the super-aggregates.

The final step in order to complete the computation of the final result table X consists of the combination of the previously created partitions X_p according to step 4 of Proposition 2, i.e., $X = X_1 \cup \cdots \cup X_P$.

6.3 Parallel *TCMDA*$^+$ with SQL

In the next section we present the *PTCMDA*$^+$ algorithm using the reduction of the intermediate result tables to SQL based on the approach illustrated in Section 5. The introduced algorithm is called *PTCMDA*$^+$*-SQL* which stands for *Parallel θ-Constrained Multi-Dimensional Aggregation Plus - SQL*. The algorithm computes the intermediate result tables $\tilde{X}^i$ using SQL and the DBMS. As a consequence, the partitioning of the detail table R as in the *PTCMDA*$^+$ algorithm is not needed and the additional parameter P indicates only the number of partitions of the base table B. If the number of partitions is equal one, i.e., $P = 1$, the third step of the algorithm is the same as in the *PTCMDA*$^+$ algorithm. The developed algorithm *PTCMDA*$^+$*-SQL* combines the advantages of constructing the separate intermediate result tables $\tilde{X}^i$ using SQL and the partitioning of the workload of the final result table X_p in order to compute the partitions in parallel. Given that the DBMS is located outside the algorithm, the DBMS has to be prepared for the execution of parallel SQL statements, for instance, *PARALLEL* hint on tables in Oracle databases [7]. In addition, the implementation is responsible of the distribution of the workload in the third step of the algorithm in order to compute the partitions of the final result tables X_p among available processors, since the algorithm provides only the partitioning strategy.

The algorithm *PTCMDA*$^+$*-SQL* begins with the initialization of the schemas of the separate intermediate result tables like the *PTCMDA*$^+$ algorithm. Next, the intermediate result tables $\tilde{X}^i$ with identical grouping attributes are merged to a single intermediate result table containing a column for each aggregate function. Then the separate intermediate result tables $\tilde{X}^i$ are computed using SQL.

Algorithm 4: $PTCMDA^{+}(B, R, P, (l_1, ..., l_m), (\theta_1, ..., \theta_m))$

```
// Step 1: Initialize intermediate result tables
Let $\mathbf{R}_i \leftarrow \mathbf{R} \cap attr(\theta_i)$ for $i = 1, ..., m$;
Let $(\mathbf{R}_{j_1}, F_{j_1}), ..., (\mathbf{R}_{j_k}, F_{j_k})$, $k \leq m$, be a partitioning of the $f_i$ according to $\mathbf{R}_i$;
foreach partition $(\mathbf{R}_j, F_j)$ do
    for $p = 1$ to $P$ do
        $\tilde{X}_p^j \leftarrow$ empty table with schema $(\mathbf{R}_j, C_{j_1}, \ldots, C_{j_{k_j}})$;
        Create an index on $\tilde{X}_p^j$ over the attributes $\mathbf{R}_j$;
    $\tilde{\theta}_j(r, b) = \bigwedge_{A \in \mathbf{R}_j} r.A = b.A$;

// Step 2: Scan partition of detail table $R(\mathbf{R})$ and update intermediate result tables
Partition $R$ such that $R = R_1 \cup \cdots \cup R_P$;
foreach tuple $r \in R_p(\mathbf{R})$ do
    foreach partition $(\mathbf{R}_j, F_j)$ do
        if $\exists \tilde{x} \in \tilde{X}_p^j$ such that $\tilde{\theta}_j(r, \tilde{x})$ then
            $\tilde{x}.C_{j_i} \leftarrow g_{j_i}(\tilde{x}.C_{j_i}, f_{j_i}(\{r\}))$ for $i = 1, ..., k_j$;
        else
            $\tilde{X}_p^j \leftarrow \tilde{X}_p^j \cup \{r.\mathbf{R}_j \circ (f_{j_1}(\{r\}), ..., f_{j_{k_j}}(\{r\}))\}$;

// Step 3: Build final result table $X$
Partition $B$ such that $B = B_1 \cup \cdots \cup B_P$ and $B_1 \cap \cdots \cap B_P = \emptyset$;
for $p = 1$ to $P$ do
    $X_p = B_p(\mathbf{B}) \times \{(v_1, ..., v_m)\}$;
    Build indexes for $X_p$;
foreach $x \in X_p$ do
    for $i = 1$ to $m$ do
        for $p = 1$ to $P$ do
            $\tilde{X}_{[x,\theta_i]} \leftarrow \{\tilde{x} \in \tilde{X}_p^j \mid \mathbf{R}_i = \mathbf{R}_j \wedge \theta_i(\tilde{x}, x)\}$;
            $x.C_i \leftarrow g_i(\tilde{X}_{[x,\theta_i]})$;

// Step 4: Combine previously created partitions of result table X
$X = X_1 \cup \cdots \cup X_P$;
return $X$;
```

After the separate intermediate result tables $\tilde{X}^i$ have been built, the base table B is divided into disjoint partitions $B = B_1 \cup \cdots \cup B_P$ with $B_1 \cap \cdots \cap B_P = \emptyset$. Next, the partitions of the final result table X_p are created from the partitions of the detail table B_p together with an index on the partitions of the final result table X_p in order to improve the retrieval time of the tuples during the computation of the aggregation results. In the next stage, the partitions of the final result table X_p are computed where the aggregate attributes are initialized to their according neutral values v_i by combining the partial aggregates from the partitions of the intermediate result tables $\tilde{X}^i$ using the super-aggregates. Finally, the evaluation of the θ-MDA query using the *PTCMDA*$^{+}$*-SQL* algorithm is completed by combining the created partitions of the final result table X_p according to step 3 of Proposition 3, i.e., $X = X_1 \cup \cdots \cup X_P$.

Algorithm 5: *PTCMDA*$^+$-*SQL*$(B, R, P, (l_1, ..., l_m), (\theta_1, ..., \theta_m))$

```
// Step 1:  Initialize intermediate result tables
Let $\mathbf{R}_i \leftarrow \mathbf{R} \cap attr(\theta_i)$ for $i = 1, ..., m$;
Let $(\mathbf{R}_{j_1}, F_{j_1}), ..., (\mathbf{R}_{j_k}, F_{j_k})$, $k \leq m$, be a partitioning of the $f_i$ according to $\mathbf{R}_i$;
// Step 2:  Create intermediate result tables
foreach partition $(\mathbf{R}_j, F_j)$ do
    $\tilde{X}^j \leftarrow$ SELECT $\mathbf{R}_j, f_{j_1}(\mathbf{R}.A_{j_1})\ C_{j_1}, \ldots, f_{j_{k_i}}(\mathbf{R}.A_{j_{k_j}})\ C_{j_{k_j}}$
            FROM $R$ GROUP BY $\mathbf{R}_j$;
// Step 3:  Build final result table $X$
Partition $B$ such that $B = B_1 \cup \cdots \cup B_P$ and $B_1 \cap \cdots \cap B_P = \emptyset$;
for $p = 1$ to $P$ do
    $X_p = B_p(\mathbf{B}) \times \{(\upsilon_1, ..., \upsilon_m)\}$;
    Build indexes for $X_p$;
foreach $x \in X_p$ do
    for $i = 1$ to $m$ do
        $\tilde{X}_{[x,\theta_i]} \leftarrow \{\tilde{x} \in \tilde{X}^j \mid \mathbf{R}_i = \mathbf{R}_j \wedge \theta_i(\tilde{x}, x)\}$;
        $x.C_i \leftarrow g_i(\tilde{X}_{[x,\theta_i]})$;
// Step 4:  Combine previously created partitions of result table X
$X = X_1 \cup \cdots \cup X_P$;
return $X$;
```

6.4 Implementation

We developed four programs for the evaluation of the algorithms presented before, where the first program evaluates the current algorithm *IndexedTCMDA* given in [10]. The second program computes the *TCMDA*$^+$ algorithm from Section 2 illustrated in [13]. This allows the comparison of the performance results of the original algorithms with the performance achievements of the newly introduced algorithms. The remaining two programs execute the presented strategies to evaluate the algorithms in parallel including the reduction of the separate intermediate result tables to SQL. All the programs were implemented using the *C* programming language [3] on top of an Oracle[4] database using the *Oracle Call Interface (OCI)* [5] and are designed as command line tools.

6.4.1 Parameters

The program expects several parameters to compute the θ-MDA. First, the program requires a parameter s which specifies the semantics of the algorithm, i.e., *TCMDA* for the *IndexedTCMDA* or *TCMDAP* for the *TCMDA*$^+$ algorithm. The program requests the detail table R and the base table B encoded as SQL queries. The program option *-r* indicates the SQL statement for the detail table R whereas the program option *-b* delivers the SQL statement for the base table B. The program expects the conditions θ_i and the list of aggregate functions f_{i_j} which are handed over using the *-c* and *-f* program option, respectively. These options have to be repeated for each group of requested aggregate functions.

In order to execute the algorithms in parallel, the command line tool expects the parameter *-t*. For the reduction of the separate intermediate result tables to SQL, the program requires the *-d* parameter option. In the following, we show how *Query 1* can be encoded using the parameters to invoke the application:

[3]`http://en.wikipedia.org/wiki/C_(programming_language)`
[4]`http://www.oracle.com/us/products/database/index.html`
[5]`http://www.oracle.com/technetwork/database/features/oci/index.html`

```
-s"tcmda" or -s"tcmdap"
-r"SELECT OrdDate, OrdPrior FROM Orders"
-b"SELECT OrdDate, OrdPrior FROM OrdersBase"

-c"r.OrdDate=b.OrdDate&r.OrdPrior=b.OrdPrior"
-f""COUNT(OrdPrior) AS CntDP"
-c"r.OrdDate<=b.OrdDate"
-f""COUNT(OrdDate) AS CumCntDP"
-c"r.OrdPrior<>b.OrdPrior"
-f""COUNT(OrdPrior) AS NegCntP"
-c"r.OrdDate<=b.OrdDate&r.OrdPrior<=b.OrdPrior"
-f""COUNT(OrdDate) AS CumCntDP"
```

After the program has been started with the previously introduced parameters, it loads the base table B into main memory, reads every tuple of the detail table $r \in R$ and builds the final result table X using either the strategy provided by the *IndexedTCMDA* or the *TCMDA*$^+$ algorithm, respectively. The output of the program is provided on the command line using the comma separated value format. The output of *Query 1* on the command line looks like the following:

```
OrdDate, OrdPrior, CntDP, CumCntDP, NegCntP, CumCntDP
2013-04-18, 1, 2, 4, 5, 2
2013-04-18, 2, 1, 4, 5, 3
2013-04-18, 3, 1, 4, 6, 4
2013-04-19, 2, 2, 7, 5, 5
2013-04-19, 3, 1, 7, 6, 7
2013-04-20, 1, 1, 8, 5, 3
```

6.4.2 Execution in Parallel

The implementation is responsible to distribute the computation of the partitions of the *IndexedPTCMDA*, *PTCMDA*$^+$ or the *PTCMDA*$^+$*-SQL* algorithm. At the moment, there exist several strategies to parallelize an algorithm in the C programming language. In the following we introduce the different possibilities and our thought process behind the decisions in favor of an implementation approach.

POSIX Threads In shared memory multiprocessor architectures (SMPs), threads can be used to implement parallelism [6]. POSIX threads, usually Pthreads, is a POSIX standard for threads, where the standard defines an API in order to create and manipulate threads [5]. Implementations are available for the most Unix-like POSIX-conformant operating systems [5].

Pthreads provide a low-level parallelism, i.e., low-level locking and fine-grained synchronization techniques. As a result, the developer has to ensure threading synchronization for the data structures as well as for the procedures of the algorithms. In addition, the developer must preset the number of threads at the compile time. This leads to non-scalable applications, if the program runs on a platform where more processors are available. Overall, Pthreads allow fine-grained control in order to parallelize an algorithm. Since our strategies are based on high-level partitioning techniques, we decided to focus on the following implementation techniques.

MPI Message Passing Interface (MPI) is a standardized and portable message-passing system designed in order to write portable message-passing programs in the C programming language [52]. MPI was designed to provide high performance applications on both massively parallel machines and on workstation clusters [3]. The advantage of MPI is that it operates on both shared or distributed memory architectures, but typically the performance is limited by the communication network between the nodes [3]. Our partitioning strategies to parallelize the introduced algorithms are designed to run on symmetric multiprocessing (SMP) systems. Since the communication overhead of applications parallelized using MPI on SMP systems is larger compared to the other implementation techniques, we decided to implement our algorithms using the next parallelization approach.

OpenMP Open Multi-Processing (OpenMP) is an API that supports multi-platform shared memory multiprocessing programming in the C programming language [22]. OpenMP uses a scalable model that gives programmers a simple and flexible interface to develop parallel applications ranging from the standard desktop computer to the supercomputer [4].

We decided to parallelize our algorithms using the OpenMP API because of the advantages over the previously introduced programming techniques. First, OpenMP provides several features to write scalable solutions. For instance, OpenMP allocates the same number of threads as number of cores available on the machine where the algorithm runs. It provides an automatic handling of the data layout and unifies the code for both the serial and the parallel applications [4].

To write scalable applications the hybrid model of parallel programming is used with a combination of both OpenMP and MPI. The algorithms implemented in the hybrid model use MPI between the distributed cluster nodes whereas OpenMP is used on the individual nodes. As already discussed, OpenMP offers better performance on symmetric multiprocessing (SMP) systems compared to Pthreads and MPI, we developed the parallelization strategies using the OpenMP API.

6.4.3 Indexes

The hash index to retrieve the tuples identified by the θ-conditions in the base table B and in the separate intermediate result tables $\tilde{X}$ was implemented using the *hcreate_r* hash table functions [6]. These functions require a null-terminated string as a search key. However, hashing variable-length strings is inefficient, if every character is processed separately [27]. This is the case in the *hseach_r* searching function. As a result, the performance of the hash index could be improved by converting the variable-length strings to numeric values using a cyclic redundancy check (CRC) to avoid hash collisions [46]. This is the main reason why the hash index within the application is inferior to the hash index in the Oracle database.

[6]`http://linux.die.net/man/3/hcreate_r`

7 Experiments

In this section we present the experiments to evaluate the performance of the introduced algorithms *IndexedPTCMDA*, *PTCMDA*$^+$ and *PTCMDA*$^+$*-SQL* in comparison with the performance of the algorithms presented in [10, 13], i.e., *IndexedTCMDA* and *TCMDA*$^+$. The five algorithms were tested using a large amount of data, i.e., up to $100M$ tuples contained in the detail table R and up to $375k$ tuples included in the base table B. Section 7.1 gives further details about the setup of the experiments including information about the generated data. In conclusion, the results and details about the experiments are presented with final remarks about the acquired insights.

7.1 Setup and Data

The machine to run the experiments contains $16GB$ of main memory and has two AMD Opteron processors including two cores with one thread per core. Both processors operate at a clock rate of $2.6GHz$. The machine used for running the experiments scaling the number of processors P contains $24GB$ of main memory and has two Intel Xeon processors including six cores with two threads per core. The processors of the second machine operate at a clock frequency of $2.67GHz$. The operating system installed on both machines is Ubuntu 10.04 and the database is Oracle 11g.

Since the performance of the proposed θ-MDA algorithms depend on different variables, the experiments scale one variable at a time while keeping the other variables fixed. In the following the variables influencing the performance of the proposed algorithms considered in our experiments are introduced:

- $|R|$: The number of tuples in the detail table R.
- $|B|$: The number of distinct tuples in the base table B.
- $|\theta|$: The number of conditions within one single θ-condition.
- $|\Theta|$: The number of θ-conditions in the θ-MDA query.
- $\left|\tilde{X}\right|$: The number of distinct tuples in the intermediate result table $\tilde{X}$.
- P: The number of processors computing the θ-MDA query.
- $\oplus$: The constraint operators used in the θ-conditions, i.e., $=, \leq, \neq$

As dataset, we used the *Orders* relation of the TPC-H benchmark framework [7]. In the running example the *Orders* relation contains several attributes such as *o_orderdate*, *o_clerk* and *o_orderpriority*.

For the experiments to compare the *IndexedTCMDA* algorithm with the *IndexedPTCMDA* algorithm, we generated an *Orders* table including $10M$ tuples using the command line tool *dbgen* of the TPC-H benchmark framework. For the remaining algorithms, i.e., *TCMDA*$^+$, *PTCMDA*$^+$ and *PTCMDA*$^+$*-SQL*, the *Orders* table contains about $100M$ tuples. Out of the generated *Orders* tables, we built several materialized views changing either the number of tuples in the detail tables or varying the number of distinct values for specific attributes

[7]TPC-H benchmark framework:`http://www.tpc.org/tpch/`

in the base tables. Further, we declared various queries to test the different influence factors of the algorithms separately. The main structure of the query used for the experiments is the following:

$$\begin{aligned}
&R : Orders\_XM \rightarrow r\\
&B : Orders\_Yk \rightarrow b\\
&l_1 : count(o\_orderdate)\\
&\theta_1 : \mathbf{r}.o\_orderdate \oplus \mathbf{b}.o\_orderdate\\
&\theta_2 : \mathbf{r}.o\_orderdate \oplus \mathbf{b}.o\_orderdate \wedge\\
&\quad \wedge\ \mathbf{r}.o\_orderpriority \oplus \mathbf{b}.o\_orderpriority
\end{aligned}$$

The query was executed by replacing *Orders_XM* for the detail table R and *Orders_Yk* for the base table B with the according materialized views specified in the settings of the experiments. For the *IndexedTCMDA* versus the *IndexedPTCMDA* algorithm, the θ-condition contains solely the *o_orderdate* attribute, i.e., θ_1, whereas the θ-condition of the *TCMDA*$^+$ variants involves the combination of the *o_orderdate* and the *o_orderpriority* attributes, i.e., θ_2.

To evaluate the performance of the introduced parallel algorithms, we created different graphs, where the y-axis shows the elapsed runtime in seconds needed by the algorithms to compute the result in form of the final result table, unless stated otherwise. For the performance evaluation of the parallelized algorithms, we follow the approach presented in [28], where execution times are normalized by introducing the *relative efficiency* and the *relative speed-up*. The relative efficiency is defined as the fraction of the execution time on one processor T_1 divided by the time T_P on P processors times the number of processors, i.e. $E_{relative} = \frac{T_1}{T_P P}$. The related quantity, the relative speed-up is computed using the relative efficiency $E_{relative}$ times the number of processors P, i.e., $S_{relative} = PE_{relative}$.

7.1.1 Scaling $|R|$

The experiments with scaling the number of tuples in the detail table $|R|$ are designed to evaluate the effect of the size of the detail table R on the performance of the θ-MDA algorithms.

IndexedTCMDA* vs. *IndexedPTCMDA For the experiments of the *IndexedTCMDA* algorithm versus the *IndexedPTCMDA* algorithm, we created five materialized views out of the main *Orders* table, i.e., *Orders_2M*, *Orders_4M*, ..., *Orders_10M* containing *2M, 4M, ..., 10M* tuples. These materialized views represent the different detail tables. For the base table B, we created the materialized view *Orders_1k*, which contains $1k$ distinct values for the *o_orderdate* attribute. The distinct values of the *o_orderdate* attribute were randomly selected from the main *Orders* table, where the *o_orderdate* attribute has a cardinality of 2406.

TCMDA*$^+$ vs. *PTCMDA*$^+$/*TCMDA*$^+$ vs. *TCMDA*$^+$*-SQL For the experiments of the *TCMDA*$^+$ versus the *PTCMDA*$^+$ and of the *TCMDA*$^+$ algorithm versus the *TCMDA*$^+$*-SQL* algorithm, we created five materialized views out of the main *Orders* table, i.e., *Orders_20M*, *Orders_40M*, ..., *Orders_100M* table containing *20M, 40M, ..., 100M* number of tuples. These materialized views represent the different detail tables. For the base table B, we created the materialized view *Orders_10k*, which contains $10k$ distinct values for the combination of the *o_orderdate* and the *o_orderpriority* attribute, i.e., (*o_orderdate*, *o_orderpriority*). The distinct values of the attribute combination (*o_orderdate*, *o_orderpriority*) were randomly selected from the main *Orders* table, where the cardinality of *o_orderdate* attribute is 2406 and the cardinality of the *o_orderpriority* attribute is 5. As a consequence, the size of the separate intermediate result table $|\tilde{X}|$ accumulates to $2406 * 5 = 12030$ distinct tuples. Since the *o_orderdate* attribute contains less than $10k$ distinct values, we used the previously described combination to retain the data distribution of the TPC-H benchmark framework.

7.1.2 Scaling $|B|$

The experiments with scaling the number of distinct tuples in the base table $|B|$ are designed to evaluate the effect of the size of the base table B on the performance of the θ-MDA algorithms.

For the experiments, we created five materialized views out of the main *Orders* table, i.e., *Orders_1k*, *Orders_2k*, ..., *Orders_5k* containing *1k, 2k, ..., 5k* distinct tuples. These materialized views represent the different base tables. Since the main *Orders* table does not contain up to $5k$ distinct values for the *o_orderdate* attribute, we generated random dates within the minimum and maximum range of the *o_orderdate* attribute.

7.1.3 Scaling $|\theta|$

The experiments with scaling the number of θ-conditions $|\theta|$ are designed to evaluate the effect of the number of the constraint conditions within one single θ-condition on the performance of the θ-MDA algorithms. For the experiments, we modified the number of constraint conditions within on single θ-condition in the main structure of the query to the following:

$$\begin{aligned}
\theta_1 &: \bigwedge_{i=1}^{|\theta|} \mathbf{r}.o\_orderdate \oplus \mathbf{b}.o\_orderdate \\
\theta_2 &: \bigwedge_{i=1}^{|\theta|} (\mathbf{r}.o\_orderdate \oplus \mathbf{b}.o\_orderdate \wedge \\
&\wedge\ \mathbf{r}.o\_orderpriority \oplus \mathbf{b}.o\_orderpriority)
\end{aligned}$$

7.1.4 Scaling $|\Theta|$

In this section we present the experiments with scaling $|\Theta|$. These experiments are designed to test the performance of the θ-MDA algorithms using several θ-conditions within a θ-MDA query. For the experiments, we changed the number of θ-conditions within a θ-MDA query in the main structure of the query to the following:

$$\begin{aligned}
l_1 &: \mathit{count}(o\_orderdate) \\
\theta_1 &: \mathbf{r}.o\_orderdate \oplus \mathbf{b}.o\_orderdate \\
&\dots \\
l_{|\Theta|} &: \mathit{count}(o\_orderdate) \\
\theta_{|\Theta|} &: \mathbf{r}.o\_orderdate \oplus \mathbf{b}.o\_orderdate
\end{aligned}$$

As in the experiment with scaling $|\theta|$, the *o_orderpriority* attribute was added for the variants of the *TCMDA*$^+$ algorithms to each of the above θ-conditions.

7.1.5 Scaling P

In this section we present the experiments with scaling the number of processors P. These experiments are designed to test the efficiency of the partitioning strategies to parallelize the θ-MDA algorithms.

For all the experiments, we used the main *Orders* table with $100M$ tuples as the detail table R and the materialized view *Orders_10k* including $10k$ distinct tuples as the base table B. The query was executed using the created materialized views and a variable number of processors P, where P differs from 1 to 24.

7.1.6 Scaling $\left|\tilde{X}\right|$

In this section we present the experiments with scaling the number of distinct tuples of the separate intermediate result tables $|\tilde{X}|$. These experiments are designed to test the efficiency of the θ-MDA algorithms using large separate intermediate result tables $\tilde{X}$, since the size of the separate intermediate result tables $|\tilde{X}|$ strongly influences the performance of the *TCMDA*$^+$ algorithms [13].

The experiments utilize the same materialized views as the experiments with scaling the number of processors P, namely the main *Orders* table with $100M$ tuples as detail table R and the *Orders_10k* table including $10k$ distinct tuples as the base table B. In order to have large intermediate result tables $\tilde{X}$, we used the combination of the *o_clerk* attribute and the *o_orderpriority* attribute. The *o_clerk* attribute has a cardinality of $75k$ whereas the *o_orderpriority* attribute exhibits a cardinality of 5. The query used to evaluate the algorithms is expressed as

$$\begin{aligned}
&R : \sigma_{o\_orderpriority=X}(Orders\_100M) \to r \\
&B : Orders\_10k \to b \\
&l_1 : \mathit{count}(o\_clerk) \\
&\theta_1 : \mathbf{r}.o\_clerk \oplus \mathbf{b}.o\_clerk \wedge \\
&\quad \wedge\ \mathbf{r}.o\_orderpriority \oplus \mathbf{b}.o\_orderpriority
\end{aligned}$$

The above query was executed using the created materialized views by replacing the $\sigma_{o\_orderpriority=X}$ condition with the appropriate values of the *o_orderpriority* attribute, i.e., *1-URGENT*, *2-HIGH*, *3-MEDIUM*, *4-NOT SPECIFIED* and *5-LOW*. This ensures that the size of the separate intermediate result tables $|\tilde{X}|$ scale from $75k, 150k, \dots, 375k$ distinct tuples.

7.2 *IndexedTCMDA* vs. *IndexedPTCMDA*

In this section we present the results of the experiments in order to evaluate the performance improvements of the *IndexedPTCMDA* algorithm compared with its non-parallel counterpart, the *IndexedTCMDA* algorithm. In all the tested settings, the parallel algorithm *IndexedPTCMDA* operates using four processors, i.e., $P = 4$. Since the *IndexedTCMDA* does not scale with large detail tables and large base tables, we decided to evaluate these algorithms using smaller data sets as for the experiments of the *TCMDA*$^+$, the *PTCMDA*$^+$ and *PTCMDA*$^+$*-SQL* algorithms.

7.2.1 Results Scaling $|R|$

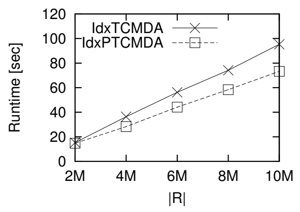
Figure 11 shows the result of the experiment described in Section 7.1.1. On the x-axis of the graphs the size of the detail table $|R|$ is displayed. The experiments provide the following insights.

If the θ-MDA query requests a point aggregate, i.e., the constraint operator $\oplus$ is $=$, the algorithms deliver the best performance. This can be explained due to the fact, that for each tuple in the detail table $r \in R$ and for each θ-condition only one update in the base table B has to be done. In addition, the according tuples in the base table B are identified using the hash index. The parallelization of the *IndexedTCMDA* algorithm using the constraint operator $\oplus$ is $=$, does not significantly decrease the runtime of the algorithm. Given that the partitioning strategy divides the base table B into P partitions, only the incremental updates in the base table B and the retrieval of the tuple using the hash index is divided among the processors. Considering that these operations are not time consuming for the point aggregate ($\oplus$ is $=$), the achieved speed-up is marginal. In fact, the average speed-up of the *IndexedPTCMDA* algorithm is 1.25 with an efficiency of 30%.

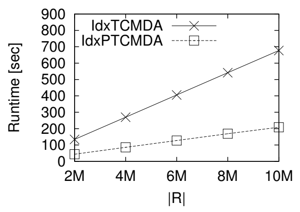
For the θ-MDA queries applying the range aggregate using the $\leq$ constraint operator, the average speed-up increases to 3.175 with an efficiency of 79.375%. For the range aggregate using the range aggregate ($\oplus$ is $\neq$), the average speed-up increases to 3.345 and an efficiency of 83.625. As described above, only the incremental updates and the retrieval of the according tuples in the base table B are distributed among the processors. For the range aggregate ($\oplus$ is $\leq$), these incremental updates and the retrievals of the according tuples in the base table B are expensive. This workload is divided among the processors leading to the results presented above. For the range aggregate ($\oplus$ is $\neq$), no index is appropriate in order to retrieve the according tuples in the base table B and therefore the entire base table B has to be traversed. As a result, the speed-up and efficiency increases further for the $\neq$ constraint operator, because the processors are working longer at full capacity. This decreases the overhead of factors such as data communications, synchronizations and software overhead [40].

All the experiments have in common that a considerable high amount of computation time resides in the DBMS, i.e., 65% for $=$, 20% for $\leq$ and 5% for the $\neq$ constraint operator. This reduces the calculated speed-up and efficiency of the parallelized algorithms, because the processors do not read in parallel. Since the *IndexedTCMDA* algorithm reads each tuple individually whereas the *IndexedPTCMDA* reads large fractions of the detail table R before processing the tuples, the two graphs are moving apart as the size of the detail table R

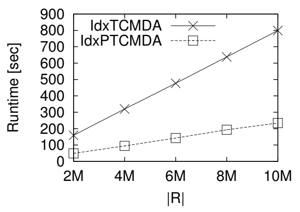
increases.



(a) $\oplus$ is $=$



(b) $\oplus$ is $\leq$



(c) $\oplus$ is $\neq$

Figure 11: IndexedTCMDA vs. IndexedPTCMDA: Results Scaling $|R|$ ($|B| = 1k, |\Theta| = |\theta| = 1, P = 4$)

7.2.2 Results Scaling $|B|$

Figure 12 shows the result of the experiment described in Section 7.1.2. On the x-axis of the graphs the size of the base table $|B|$ is displayed.

As in the previous experiments with scaling $|R|$, both algorithms exhibit the best performance, if the constraint operator $\oplus$ is $=$. In this case, the runtime of the algorithms is not affected by the increasing size of the base table $|B|$ as for the same reasons explained in Section 7.2.1. For the range aggregate operators $\leq$ and $\neq$, the runtime for the *IndexedTCMDA* increases dramatically whereas the evaluation time of the *IndexedPTCMDA* increases only slightly. This can be explained due to the fact, that the partitioning of the base table B creates smaller base tables for every processor. For instance, with four processors $P = 4$ and a base table B containing about $1k$ distinct values, each processor has to compute its own base table with $1k/4 = 250$ tuples. As a consequence, the index of the processor's base table is smaller and can be accessed faster for the $\leq$ and $\neq$ constraint operators. In addition, the number of incremental updates for each processor decreases. The division of the workload among the processors has the same effect as executing the θ-MDA query using a smaller base table using a single processor. This effect can be observed in the graphs, where the runtime of th *IndexedTCMDA* algorithm with a base table containing $1k$ tuples is the same as the runtime of the *IndexedPTCMDA* algorithm using four processors and a base table with $4k$ tuples.

For the experiments with scaling the base table B, we achieve super linear speed-up [11]. The effect can be explained, because each processor has to search a smaller index and performs less incremental updates for its base table. Moreover, the *IndexedTCMDA* algorithm reads each tuple individually whereas the *IndexedPTCMDA* reads large fractions of the detail table R before processing the tuples.

7.2.3 Results Scaling $|\theta|$

In Figure 13 the result of the experiment described in Section 7.1.3 is presented. On the x-axis of the graphs the number of conditions within one single θ-condition is shown.

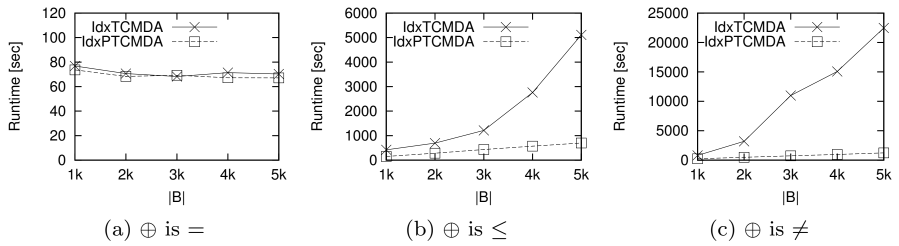


Figure 12: IndexedTCMDA vs. IndexedPTCMDA: Results Scaling $|B|$ ($|R| = 10M, |\Theta| = |\theta| = 1, P = 4$)

The runtime of both algorithms increases as the number of the conditions within one single θ-condition growths. This increase can be explained due to the fact, that the evaluation of the single θ-conditions is more expensive, since the θ-conditions contain more constraint operators. For the runtimes of the different constraint operators $\oplus$, the algorithms feature the same behavior as for the experiments with scaling the detail table R or the base table B, specifically the impact of the size of the index on the base table B and the number of incremental updates in the base table B.

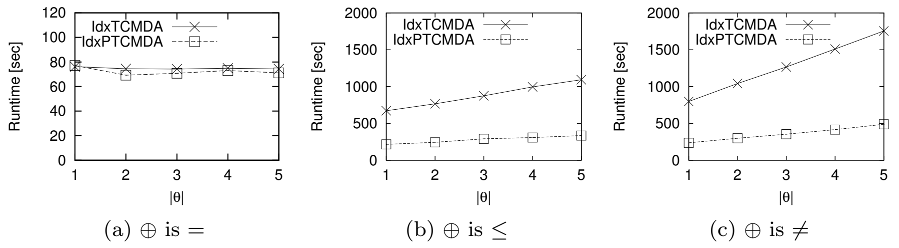


Figure 13: IndexedTCMDA vs. IndexedPTCMDA: Results Scaling $|\theta|$ ($|R| = 10M, |B| = 1k, |\Theta| = 1, P = 4$)

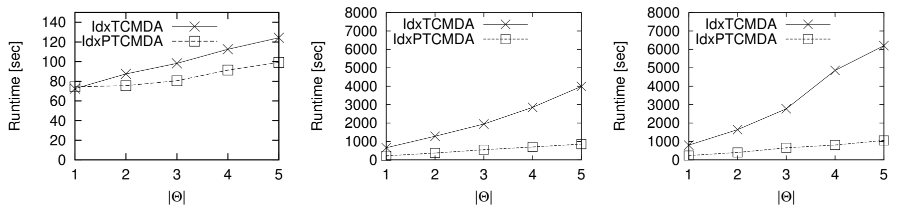
7.2.4 Results Scaling $|\Theta|$

In Figure 14 the result of the experiment described in Section 7.1.4 is shown. On the x-axis of the graphs the number of θ-conditions $|\theta|$ in Θ is reported.

The runtime of both algorithms increases as the number of θ-conditions $|\theta|$ in Θ growths. This increase can be explained due to a larger number of θ-conditions in Θ results in more incremental aggregates to be computed in the base table B. Again, the algorithms exhibit the same behavior for the runtimes as in the experiments with scaling $|\theta|$, i.e., smaller base tables and as consequence less incremental updates for every tuple in the detail table R.

7.2.5 Results Scaling P

In Figure 15 the result of the experiment described in Section 7.1.5 is presented. On the x-axis of the graphs the number of processors P is shown. In the first



(a) $\oplus$ is $=$ (b) $\oplus$ is $\leq$ (c) $\oplus$ is $\neq$

Figure 14: IndexedTCMDA vs. IndexedPTCMDA: Results Scaling $|\Theta|$ ($|R| = 10M, |B| = 1k, |\theta| = 1, P = 4$)

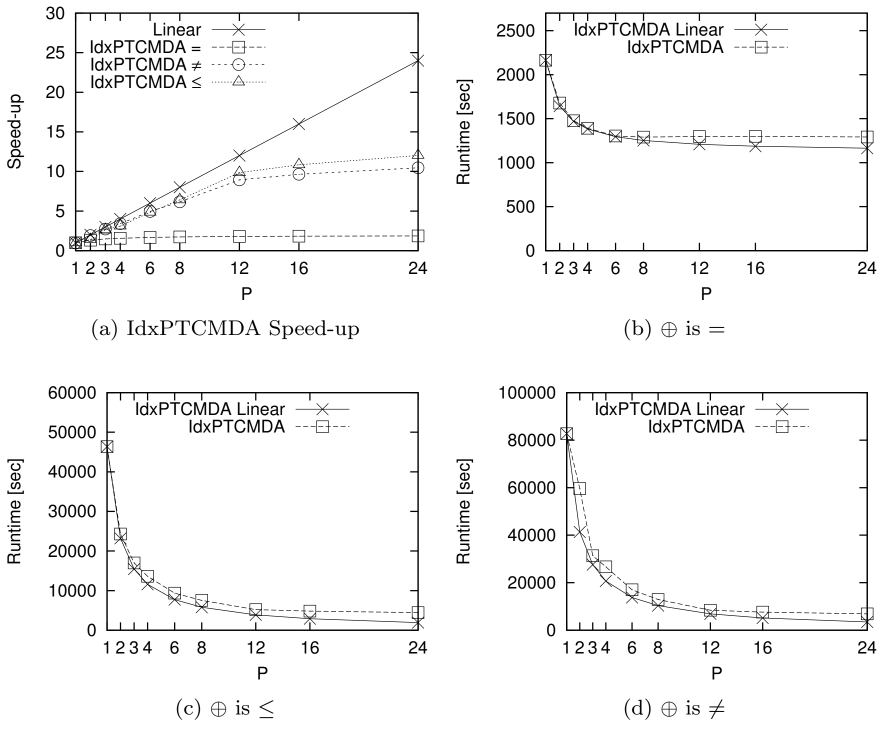
graph 15a the y-axis reflects the achieved speed-up. For the remaining graphs, the elapsed time in seconds to compute the θ-MDA query is reported.

In the first experiment 15a the theoretical speed-up of the entire computation of the θ-MDA together with the achieved speed-up for the different constraint operators $\oplus$ is shown. The achieved speed-up for the $=$ constraint operator is very poor, since only the incremental updates are distributed among the processors. These incremental updates for the $=$ constraint operator are very cheap, because at most one tuple identified by the hash index has to be updated in the base table B. For the other constraint operators, namely $\leq$ and $\neq$, the achieved speed-up is close to the linear speed-up until using 12 processors. Afterwards, the achieved speed-up decreases and the curve becomes flatter.

The other three experiments show the theoretical linear runtimes together with the achieved runtimes of the *IndexedPTCMDA* algorithm. To calculate the theoretical linear runtimes, we executed the algorithm using a single thread, subtracted the database access time and divided the runtime by the number of processors. The subtraction of the database access times ensures that the calculated runtimes are correct due to the fact, that database access was not parallelized. As the graphs show, the achieved runtimes are close to the theoretical linear runtimes in all the tested settings. In addition, the more processors are added to the computation of the algorithm, the more the theoretical linear runtime drifts apart from the achieved runtime. This can be explained, because the maintenance, synchronization and data sharing costs increase as the number of processors growths. Finally, the runtimes of the algorithm lean towards the database times, i.e., reading the tuples from the detail table R and the base table B. As a result, even with an infinite number of processors P, the computation of the θ-MDA query is limited to the database access time, if the reading operation from the database is not parallelized.

7.3 *TCMDA*$^+$ vs. *PTCMDA*$^+$

In this section we present the results of the experiments in order to evaluate the performance of the *TCMDA*$^+$ algorithm compared with the parallelized *PTCMDA*$^+$ algorithm. Again, in all the tested settings the parallel algorithm *PTCMDA*$^+$ operates using four processors, i.e., $P = 4$. For the following experiments, we implicitly limit the size of the separate intermediate result tables $|\tilde{X}|$. The size of the separate intermediate result tables $|\tilde{X}|$ is specified indirectly by the cardinality of the according attribute in the θ-MDA query. In our examples,



(a) IdxPTCMDA Speed-up

(b) $\oplus$ is $=$

(c) $\oplus$ is $\leq$

(d) $\oplus$ is $\neq$

Figure 15: IndexedTCMDA vs. IndexedPTCMDA: Results Scaling P $(|R| = 100M, |B| = 10k, |\Theta| = |\theta| = 1)$

the cardinality is $|o\_orderdate| \times |\ o\_orderpriority\ | = 12030$.

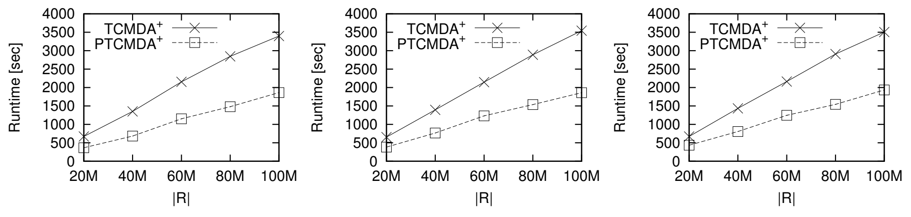
7.3.1 Results Scaling $|R|$

Figure 16 shows the result of the experiment described in Section 7.1.1. On the x-axis of the graphs the size of the detail table $|R|$ is displayed. The experiments allow the following insights.

The first insight is, that the runtime of the algorithms are independent from the constraint operator $\oplus$. This can be explained due to the fact, that both algorithms construct the separate intermediate result tables $\tilde{X}$ using the equality constraint conditions before the evaluation of the original θ-conditions. After the construction the former contains about 12030 distinct values. These tuples are used as the new detail table to construct the final result table X.

For the parallelized algorithm, the average speed-up is 1.88 with an average efficiency of 47%. These measurements can be explained due to the parallelization is used during the construction of the intermediate result tables $\tilde{X}$ using the equality constraint operator. As the previously described experiments have shown, the obtained parallelism is marginal in these circumstances. In the next stage of the algorithm, where the final result table X is constructed using the separate intermediate result table $\tilde{X}$ with the original θ-conditions, the parallelization shows the same characteristics as in the *IndexedPTCMDA* algorithm presented in Section 7.2. Basically, the same steps are executed in the last stage of the *TCMDA*$^+$ algorithm as in the *IndexedTCMDA* algorithm with the difference, that the *TCMDA*$^+$ operators on the compressed detail table R, i.e., the separate intermediate result table $\tilde{X}$.

The last insight is, that in all experiments the two graphs are moving apart as the size of the detail table R increases. This behavior can be explained due to the fact, that the *TCMDA*$^+$ algorithm reads each tuple individually whereas the *PTCMDA*$^+$ reads large fractions of the detail table R before processing the tuples. This effect is intensified, because a considerable high amount of computation time resides in the DBMS, i.e., about 40%.



(a) $\oplus$ is $=$ (b) $\oplus$ is $\leq$ (c) $\oplus$ is $\neq$

Figure 16: TCMDA$^+$ vs. PTCMDA$^+$: Results Scaling $|R|$ ($|B| = 10k, |\tilde{X}| = 12030, |\Theta| = 1, |\theta| = 2, P = 4$)

7.3.2 Results Scaling $|B|$

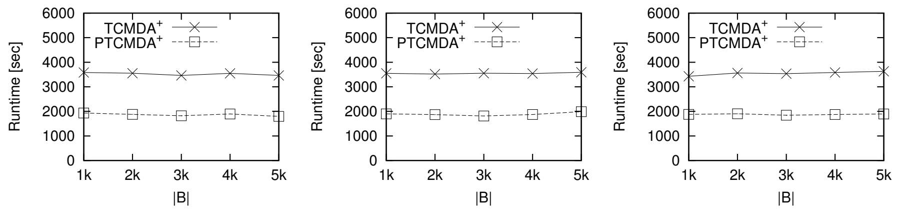
Figure 17 shows the result of the experiment described in Section 7.1.2. On the x-axis of the graphs the size of the detail table $|B|$ is displayed.

From Figure 17 it is possible to recognize, that both algorithms are affected neither by the size of the base table B nor by the type of the constraint operator $\oplus$. As described above, the construction of the separate intermediate result tables $\tilde{X}$ requires always the same number of iterations, i.e., $|R| \cdot |\Theta| \cdot 1 = 100M \cdot 1 \cdot 1 = 100M$, where the resulting intermediate result table $\tilde{X}$ contains in the worst case 12030 distinct tuples, i.e., the cardinality of the *o_orderdate* times the *o_orderpriority* attribute. This construction is independent from the size of the base table B. In these tests, the size of the base table B is to small with respect to the size of the detail table R in order to have an effect on the runtime of the algorithms.

For the parallelized algorithm *PTCMDA*$^+$, the Figure 17 shows speed-up which remains constant on 1.9 with an efficiency of 47.5%. This can be explained due to the same circumstances described previously and Section 7.3.1 with scaling the number of tuples in the detail table R, namely that *TCMDA*$^+$ algorithm is not affected by the size of the base table B and that the parallelism is marginal using the equality constraint condition.

7.3.3 Results Scaling $|\theta|$

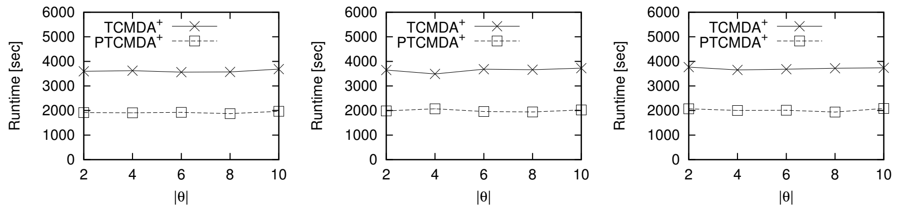
In Figure 18 the result of the experiment described in Section 7.1.3 is shown. On the x-axis of the graphs the number of conditions within one single θ-condition is reported.



(a) $\oplus$ is $=$ (b) $\oplus$ is $\leq$ (c) $\oplus$ is $\neq$

Figure 17: TCMDA$^+$ vs. PTCMDA$^+$: Results Scaling $|B|$ $(|R| = 100M, |\tilde{X}| = 12030, |\Theta| = 1, |\theta| = 2, P = 4)$

From Figure 18 it is possible to observe, that the runtimes of the algorithms are not affected by the number of conditions within one single θ-condition. The reason for this situation is the same as described above with scaling the number of tuples in the detail table B, where the size of the base table B is to small with respect to the size of the detail table R to affect the runtime of the algorithms. As a consequence, the computation costs of additional conditions within one θ-condition in the base table B are negligible.



(a) $\oplus$ is $=$ (b) $\oplus$ is $\leq$ (c) $\oplus$ is $\neq$

Figure 18: TCMDA$^+$ vs. PTCMDA$^+$: Results Scaling $|\theta|$ $(|R| = 100M, |\tilde{X}| = 12030, |B| = 10k, |\Theta| = 1, P = 4)$

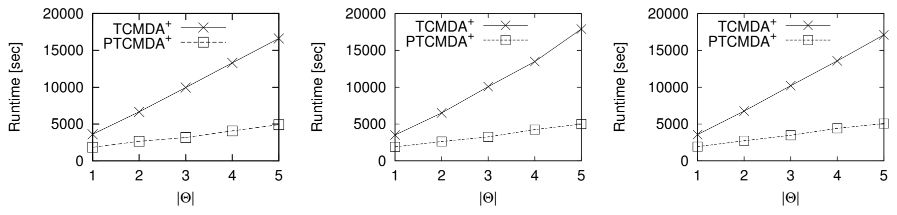
7.3.4 Results Scaling $|\Theta|$

In Figure 19 the result of the experiment described in Section 7.1.4 is shown. On the x-axis of the graphs the number of θ-conditions $|\theta|$ in Θ is reported.

The runtime of both algorithms increases as the number of θ-conditions $|\theta|$ in Θ growths. This increase can be explained due to the fact, that a larger number of θ-conditions in Θ causes additional separate intermediate result tables $\tilde{X}$ to be computed. This is because the *TCMDA*$^+$ and the *PTCMDA*$^+$ algorithm construct for every θ-condition in Θ a completely new separate intermediate result tables $\tilde{X}$ from scratch. In addition, a higher number of θ-conditions in Θ results in more incremental aggregates to be computed in the base table B.

The constructions of the separate intermediate result tables $\tilde{X}$ and these incremental updates are partially absorbed by the *PTMCDA*$^+$ algorithm. The partitioning strategies of the detail table R and the base table B result in smaller construction costs of the separate intermediate result tables $\tilde{X}$ and in smaller base tables. As a result, less updates for every tuple in the separate intermediate result tables $\tilde{X}$ and less incremental updates in the final result table are needed. These circumstances cause an average speed-up of 2.84 with an average efficiency of 70%.

Another observation on the graphs in Figure 19 is, that the impact of the number of θ-conditions in Θ on the *TCMDA*$^+$ algorithm is larger than the impact on the parallelized algorithm *PTCMDA*$^+$. In fact, the graph representing the runtime of the *PTCMDA*$^+$ is much steeper than the line of non-parallelized algorithm *TCMDA*$^+$. Furthermore, as the number of θ-conditions in Θ increases, the runtimes of the *TCMDA*$^+$ and the *PTCMDA*$^+$ algorithms drift apart. This phenomenon can be explained, because the utilization of the processors increases, since the computation done in parallel growths.



(a) $\oplus$ is $=$ (b) $\oplus$ is $\leq$ (c) $\oplus$ is $\neq$

Figure 19: TCMDA$^+$ vs. PTCMDA$^+$: Results Scaling $|\Theta|$
$(|R| = 100M, |\tilde{X}| = 12030, |B| = 10k, |\theta| = 1, P = 4)$

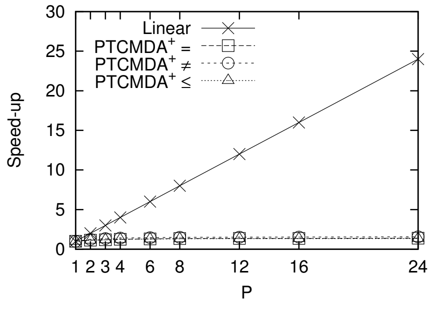
7.3.5 Results Scaling P

In Figure 20 the result of the experiment described in Section 7.1.5 is presented. On the x-axis of the graphs the number of processors P is shown. In the first graph 15a the y-axis reflects the achieved speed-up. For the remaining graphs, the elapsed time in seconds to compute the θ-MDA query is reported.

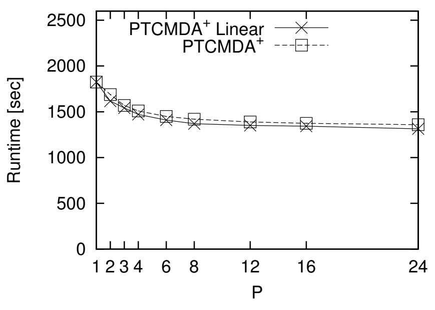
The first experiment 15a shows the achieved speed-up of the parallelized *TCMDA*$^+$ algorithm. For every constraint operator $\oplus$ the achieved speed-up is poor. The average speed-up is about 1.35 with an average efficiency of 33.75%. This behavior can be explained due to almost the entire runtime of the algorithm elapses in the database. This includes operations, such as reading and constructing the tuples from the detail table R. As the experiments comparing the different constraint operators show, the achieved runtimes closely tend to the theoretical linear runtimes of the *PTCMDA*$^+$ algorithm. Since the reading from the database was not parallelized, the achieved and theoretical linear runtimes of the experiments lean towards the runtime of the database access. In fact about 80% of the runtime of the *PTCMDA*$^+$ resides in the database.

7.4 *TCMDA*$^+$ vs. *TCMDA*$^+$*-SQL* vs. *PTCMDA*$^+$*-SQL*

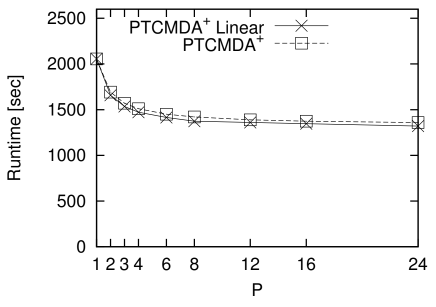
In this section we present the results of the experiments to evaluate the performance of the *TCMDA*$^+$ algorithm compared with the *TCMDA*$^+$*-SQL* and the *PTCMDA*$^+$*-SQL* algorithm. The *TCMDA*$^+$*-SQL* in contrast to the *TCMDA*$^+$ algorithm reduces the construction of the separate intermediate table $\tilde{X}$ to SQL. Since the attribute combination used in the experiments have a cardinality of



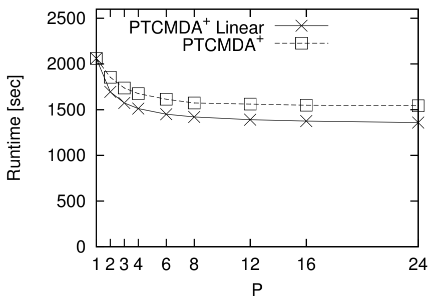
(a) PTCMDA^{+} Speed-up



(b) $\oplus$ is $=$



(c) $\oplus$ is $\leq$



(d) $\oplus$ is $\neq$

Figure 20: TCMDA^{+} vs. PTCMDA^{+}: Results Scaling P ($|R| = 100M, |\tilde{X}| = 12030, |B| = 10k, |\Theta| = |\theta| = 1$)

$|o\_orderdate| \times |\ o\_orderpriority\ | = 12030$, the completion time of the final result table is negligible. As a consequence, the main part of the runtime of the *TCMDA^{+}-SQL* algorithm resides in the database and the parallelization of the final step is not apparent in the runtimes of the *TCMDA^{+}-SQL* compared to the *PTCMDA^{+}-SQL* algorithm.

7.4.1 Results Scaling $|R|$

In this section we introduce the result of the experiment described in Section 7.1.1. This experiment shows the main characteristics of the *TCMDA^{+}-SQL* algorithm, where the runtime remains almost constant. Hence, we forego additional experiments using different parameters except with scaling the size of the separate intermediate result table $\tilde{X}$. On the x-axis of the graphs the size of the detail table $|R|$ is displayed. The results of the experiment are shown in Figure 21 and allows the following insights.

The first insight is, that the *TCMDA^{+}-SQL* algorithm clearly outperforms the *TCMDA^{+}* algorithm up to two orders of magnitude. As the size of the detail table R increases, the more the runtime of the *TCMDA^{+}* algorithm is affected. This phenomenon can be explained by fact, that the hash table management for the construction of the separate intermediate result table $\tilde{X}$ in the application is less efficient as in the DBMS as described in Section 6.4.3.

Another insight is, that almost the entire runtime of the *TCMDA^{+}-SQL* algorithm is elapsed in the database during the construction of the separate intermediate result table $\tilde{X}$. In fact, in the worst case the construction of the final result table takes about $5s$ for the $\leq$ constraint operator for all the experiments scaling the different parameters.

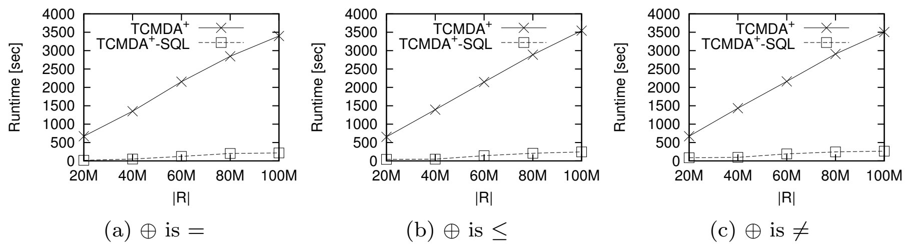


Figure 21: TCMDA$^+$ vs. TCMDA$^+$-SQL: Results Scaling $|R|$ $(|B| = 10k, |\tilde{X}| = 12030, |\Theta| = 1, |\theta| = 2)$

7.4.2 Results Scaling $|\tilde{X}|$

In this section, we present the experiments as described in Section 7.1.6. On the x-axis of the graphs the number of distinct tuples contained in the separate intermediate result table $\tilde{X}$ is shown.

As discussed before, the runtime of *TCMDA$^+$-SQL* algorithm is mainly affected by the size of the separate intermediate result table $\tilde{X}$. Hence, using additional processors does not decrease the runtime significantly. However, as the size of the separate intermediate result table $\tilde{X}$ increases, the computation time of the final result tables growths. As a result, the computation done in parallel leads to smaller base tables, which reduces the number of incremental updates for every tuple in the separate intermediate result table $\tilde{X}$. This reduces the computation time significantly, because the main part of the runtime shifts from the database access towards the computation of the final result table. Figure 22 shows the performance gain, where the average speed-up is about 2.72 with an average efficiency of 68%, where the former increases as the size of the separate intermediate result table $\tilde{X}$ growths.

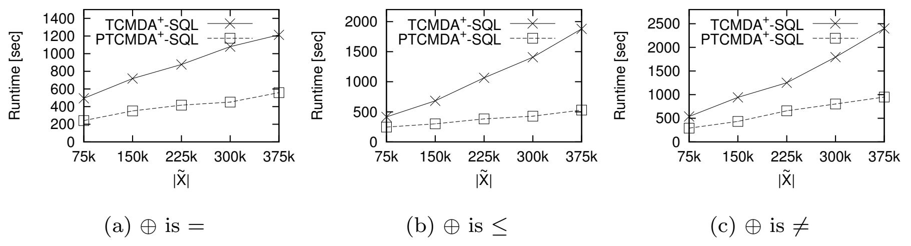


Figure 22: PTCMDA$^+$ vs. PTCMDA$^+$-SQL: Results Scaling $\left|\tilde{X}\right|$ $(|R| = 100M, |B| = 10k, |\Theta| = 1, |\theta| = 2, P = 4)$

7.5 Discussion and Summary

In this section, we present a general overview of the different experimental results together with a discussion of the performance outcomes. In addition, we take a closer look at the achieved speed-up with the according efficiency.

As discussed above, the *IndexedTCMDA* and the parallel *IndexedTCMDA* algorithm do not scale using large base and detail tables whereas the *TCMDA*$^+$ and the parallel *TCMDA*$^+$ can be executed using very large detail tables. Compared to the *IndexedTCMDA* algorithm, the *TCMDA*$^+$ algorithm executes an additional step for the equality constraint condition. As a consequence, the *IndexedPTCMDA* algorithm reaches almost the same runtime for small detail tables as the *PTCMDA*$^+$ algorithm.

The *TCMDA*$^+$*-SQL* algorithm outperforms all the former algorithms up to two orders of magnitude in every tested setting, since the main part of incurred workload resides in the database, if the number of distinct tuples in the separate intermediate result tables are small. This is also the reason, why the algorithm is almost not affected by any parameter involved in the computation. In the case, where the distinct tuples in the separate intermediate result tables increases, the parallelized version of the algorithm *PTCMDA*$^+$*-SQL* partially absorbs the increased workload to compute the final result table, since the computational effort shifts from the database to the processing step of the algorithm.

For the experiments comparing the *IndexedTCMDA* with the *IndexedPTCMDA*, we achieved super linear speed-up [11], since we only evaluated the original with the newly implemented algorithm. Hence, we unknowingly introduced performance improvements, such as reading several tuples from the detail table at once. These improvements could also be implemented in the original algorithm. As a consequence, the super linear speed-up most probably will disappear.

As the experiments show, we did not reach the theoretical speed-up using four processors, because of the reasons stated in *Amdahl's law [12, 1].* The law specifies the maximum expected speed-up of an algorithm when only parts can be parallelized, since every algorithm has a sequential component. In our case, this component is the database access on a single machine. Therefore, even with an infinite number of processors, the execution times of the algorithms will not decrease beyond the database computation times. However, Amdahl's law describes a limit on the speed-up given a fixed data size. Here, *Gustafson's law [35]* provides a counterpoint. The law states, that computations involving arbitrarly large data sets can be efficiently parallelized, since the sequential portion of the algorithm remains fixed or grows very slowly with the input size. Hence, additional processors can solve queries without any limit.

This is the case in our parallelized algorithms, where the theoretical size of the detail table using the reduction of the separate intermediate result tables to SQL is unlimited. In the experiments we are not interested in solving a fixed problem in the shortest possible time, but rather in solving the largest possible problem in a reasonable amount of time. Hence, various refinements could be introduced in the applications of the operator, e.g., an enhanced fully coupled oceanographic fact table [10] with large amount of data gathered at hundreds of survey stations.

8 Conclusions and Future Work

The analysis of large amounts of data is an important task in different applications areas. This requires enhanced techniques for the flexible formulation and efficient evaluation of complex multi-dimensional aggregation queries. On account of this, the θ-constrained multidimensional aggregation operator (θ-MDA) was introduced in [10]. θ-MDA separates the specification of groupings (base table B) for which aggregates are reported from the specification of the associated aggregation groups over which the aggregates are computed (detail table R). For the evaluation of such queries several approaches have been introduced, namely the *TCMDA [10]* and the *TCMDA*$^+$ *[13]* algorithm.

For the evaluation, the base table B has to be updated for every entry of detail table R. This leads to an immense number of incremental updates on the base table B for range aggregates. The *TCMDA*$^+$ algorithm enhances this approach by compressing the associated detail table R using equality constraint conditions which reduces the number of incremental updates in the base table B. However, both algorithms do not take advantage of the current evolution of modern computer architectures, where the clocking frequency stays constant and the number of computing cores increases [55].

In this thesis, we develop a partitioning strategy to distribute the workload of the incremental updates in the base table B among the available processors. Further, we define a reduction of the computation of the separate intermediate result tables to SQL. These SQL statements can be optimized and executed efficiently by the DBMS and the result can be further processed to complete the θ-MDA query. We implement the parallelized algorithms using the C programming language on top of the Oracle database. In the experiments we compare the performance of the parallel algorithms with their sequential counterparts and the impact of using the reduction to SQL to compress the detail table R.

The results show that the developed *PTCMDA*$^+$*-SQL* algorithm executed in parallel outperforms the state of the art approaches by up to two orders of magnitude. This allows the usage of large base tables together with indefinitely large detail tables, where the parallelization absorbs any influences of the different parameters involved in the computation of the θ-MDA query. On account of the optimizations, the algorithm utilizes the available resources of the computing machine and almost reduces the evaluation of θ-MDA conditions to the computation time of the groups using the DBMS, providing that the number of distinct groups in the detail table R are small.

Future work can be driven into various directions. First, the intermediate result tables with θ-conditions including the same groups but different constraint conditions could be shared. Second, we will identify real world applications with very large intermediate result tables for which additional optimization strategies are required, e.g., implementations relocated to massively parallel processors, i.e., GPUs. Another optimization concerns the implementation of the operator, namely leveraging MapReduce techniques for distributed query processing or the hybrid parallel programming model using MPI combined with OpenMP on clustered multi-core machines. Other future research could be focus on the integration of the θ-MDA operator as an algebraic operator into the kernel of PostgreSQL [8] together with optimizations for the query optimizer.

[8]`http://www.postgresql.org`

References

[1] Amdahl's law. `http://en.wikipedia.org/wiki/Amdahl's_law`. Accessed: 2013-07-01.

[2] Mapreduce programming model. `http://en.wikipedia.org/wiki/MapReduce`. Accessed: 2013-07-08.

[3] The message passing interface (mpi) standard. `http://www.mcs.anl.gov/research/projects/mpi/`. Accessed: 2013-06-17.

[4] Open multi-processing (openmp). `http://en.wikipedia.org/wiki/OpenMP`. Accessed: 2013-06-17.

[5] Posix threads. `http://en.wikipedia.org/wiki/POSIX_Threads`. Accessed: 2013-06-17.

[6] Posix threads programming. `https://computing.llnl.gov/tutorials/pthreads`. Accessed: 2013-06-17.

[7] Using parallel execution in oracle. `http://docs.oracle.com/cd/E11882_01/server.112/e25523/parallel.htm#VLDBG010`. Accessed: 2013-06-10.

[8] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[9] Michael O Akinde and Michael H Bohlen. Efficient computation of subqueries in complex olap. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 163–174. IEEE, 2003.

[10] Michael O. Akinde, Michael H. Böhlen, Damianos Chatziantoniou, and Johann Gamper. theta-constrained multi-dimensional aggregation. *Inf. Syst.*, 36(2):341–358, 2011.

[11] Selim G. Akl. Superlinear performance in real-time parallel computation. *Journal of Supercomputing*, 29:89–111, 2001.

[12] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[13] Christian Ammendola, Michael H. Böhlen, and Johann Gamper. Efficient evaluation of ad-hoc range aggregates. In *15th International Conference on Data Warehousing and Knowledge Discovery (DaWaK-13)*, Prague, Czech Republic, August 26 - 29 2013. 14 pages.

[14] Coming Attractions. Sql standardization: the next steps. *SIGMOD Record*, 29(1):63, 2000.

[15] Ranjit Bose. Knowledge management-enabled health care management systems: capabilities, infrastructure, and decision-support. *Expert Systems with Applications*, 24(1):59–71, 2003.

[16] Damianos Chatziantoniou, Theodore Johnson, Michael Akinde, and Samuel Kim. The md-join: An operator for complex olap. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 524–533. IEEE, 2001.

[17] Damianos Chatziantoniou and Kenneth A Ross. Querying multiple features of groups in relational databases. In *VLDB*, volume 96, pages 295–306, 1996.

[18] Damianos Chatziantoniou and Kenneth A Ross. Groupwise processing of relational queries. In *VLDB*, volume 97, pages 476–485. Citeseer, 1997.

[19] Qiming Chen, Umeshwar Dayal, and Meichun Hsu. Olap-based data mining for business intelligence applications in telecommunications and e-commerce. In Subhash Bhalla, editor, *DNIS*, volume 1966 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2000.

[20] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19:281, 2007.

[21] Seok-Ju Chun, Chin-Wan Chung, Ju-Hong Lee, and Seok-Lyong Lee. Dynamic update cube for range-sum queries. In *VLDB*, pages 521–530, 2001.

[22] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[25] Andrew Eisenberg, Jim Melton, Krishna G. Kulkarni, Jan-Eike Michels, and Fred Zemke. Sql: 2003 has been published. *SIGMOD Record*, 33(1):119–126, 2004.

[26] Wenbin Fang, Bingsheng He, Qiong Luo, and Naga K Govindaraju. Mars: Accelerating mapreduce with graphics processors. *Parallel and Distributed Systems, IEEE Transactions on*, 22(4):608–620, 2011.

[27] Martin Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *In Proc 7th CPM*, pages 130–140. Springer, 1996.

[28] Ian Foster. *Designing and building parallel programs, Chapter 3: Developing Models*, volume 95. Addison-Wesley Reading, 1995.

[29] Steven Geffner, Divyakant Agrawal, Amr El Abbadi, and Terry Smith. Relative prefix sums: An efficient approach for querying dynamic olap data cubes. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 328–335. IEEE, 1999.

[30] Sanjay Goil and Alok Choudhary. High performance olap and data mining on parallel computers. *Data Mining and Knowledge Discovery*, 1(4):391–417, 1997.

[31] Sanjay Goil and Alok Choudhary. A parallel scalable infrastructure for olap and data mining. In *Database Engineering and Applications, 1999. IDEAS'99. International Symposium Proceedings*, pages 178–186. IEEE, 1999.

[32] Goetz Graefe, Usama M Fayyad, Surajit Chaudhuri, et al. On the efficient gathering of sufficient statistics for classification from large sql databases. In *KDD*, pages 204–208, 1998.

[33] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and subtotals. *Data Min. Knowl. Discov.*, 1(1):29–53, January 1997.

[34] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and subtotals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[35] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[36] Venky Harinarayan, Anand Rajaraman, and Jeffrey D Ullman. Implementing data cubes efficiently. In *ACM SIGMOD Record*, volume 25, pages 205–216. ACM, 1996.

[37] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[38] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. *Range queries in OLAP data cubes*, volume 26. 1997.

[39] Gabriele Jost, Haoqiang Jin, Dieter an Mey, and Ferhat F Hatay. Comparing the openmp, mpi, and hybrid programming paradigms on an smp cluster. In *Proceedings of EWOMP*, volume 3, page 2003, 2003.

[40] Vipin Kumar. *Introduction to Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[41] Ki Yong Lee and Myoung Ho Kim. Efficient incremental maintenance of data cubes. In *Proceedings of the 32nd international conference on Very large data bases*, pages 823–833. VLDB Endowment, 2006.

[42] Weifa Liang, Hui Wang, and Maria E Orlowska. Range queries in dynamic olap data cubes. *Data & Knowledge Engineering*, 34(1):21–38, 2000.

[43] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of data cubes and summary tables in a warehouse. In *ACM SIGMOD Record*, volume 26, pages 100–111. ACM, 1997.

[44] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178, New York, NY, USA, 2009. ACM.

[45] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. IEEE, 2009.

[46] M.V. Ramakrishna and Justin Zobel. Performance in practice of string hashing functions. In *Proc. Int. Conf. on Database Systems for Advanced Applications*, pages 215–223, 1997.

[47] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24. IEEE, 2007.

[48] Sunita Sarawagi, Shiby Thomas, and Rakesh Agrawal. *Integrating association rule mining with relational database systems: Alternatives and implications*, volume 27. ACM, 1998.

[49] Dennis Shasha. Database tuning: Principles, experiments, and guidance, chapter 2: Tuning the guts. In *SBBD*, pages 15–62, 2003.

[50] Dennis Shasha. Database tuning: Principles, experiments, and guidance, chapter 3: Index tuning. In *SBBD*, pages 63–93, 2003.

[51] Ambuj Shatdal and Jeffrey F Naughton. Adaptive parallel aggregation algorithms. In *ACM SIGMOD Record*, volume 24, pages 104–114. ACM, 1995.

[52] Marc Snir, Steve W Otto, David W Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: the complete reference*. MIT press, 1995.

[53] Radhika Sridhar, Padmashree Ravindra, and Kemafor Anyanwu. Rapid: Enabling scalable ad-hoc analytics on the semantic web. In *The Semantic Web-ISWC 2009*, pages 715–730. Springer, 2009.

[54] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: Friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[55] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.

[56] Tom White. *Hadoop: the definitive guide*. O'Reilly, 2012.

[57] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040. ACM, 2007.