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

Late Initialization for θ-Constrained Multi-Dimensional Aggregation

Author
Christian Ammendola

Supervisor
Prof. Johann Gamper

March 2011

Abstract

In today's *information society* the analysis of large amounts of data is an important task in different areas, such as in *business intelligence* and scientific fields, requiring, among others, techniques for the flexible formulation and efficient evaluation of complex multi-dimensional aggregation queries. *Online analytical processing (OLAP)* is a mature field concerned with the analysis of multi-dimensional data. Since the definition of SQL:99 [10] and the introduction of *window functions* in SQL:2003 [12] different OLAP techniques are integrated also in the SQL standard. However, SQL still lacks of a broad support for complex analytical queries such as cumulative aggregation over more than one dimension. *θ-constrained multi-dimensional aggregation (θ–MDA)* [2], an operator supporting a flexible formulation and efficient evaluation of complex multi-dimensional aggregation queries based on θ-conditions, overcomes the limitations of SQL. The evaluation approach for θ–MDA handles the computation of the result groups and the computation of the aggregates separately. In cases in which both the result groups and the aggregates are derived from the same input relation this results into two scans of that relation. Due to the fact, that in typical θ–MDA scenarios the input relation is very large, for example, several hundred millions of tuples, loading and scanning it twice is very costly. In this thesis we propose an approach that requires only one scan of the input relation for θ–MDA queries in which both the result groups and the aggregates are computed from the same input relation. The main idea of our approach, to which we refer to as *θ–MDA with late initialization*, is to compute the result groups on demand while the aggregates are computed. We investigate differences in the on-the-fly computation of result groups which arise due to the use of different constraint operators, such as $=$, $\leq$, and $\neq$, in the θ-conditions and design a solution that handles these differences. We propose an algorithm for our approach and evaluate its performance with respect to the evaluation approaches presented in [2] by implementing them as Java programs on top of an Oracle database. The data set used in the evaluation experiments is generated by using the `dbgen` tool of the TPC-H benchmark framework. θ–MDA with late initialization performs better in all tested settings and the single load and scan of the input relation reduces the evaluation runtime by up to 65 percent for an input relation with 10 million tuples (on average the improvement was about 54 percent). Further, in contrast to the evaluation strategies in [2], our approach results not to be sensitive to the type of constraint operators used in the θ-conditions, for scenarios with input tables having up to 10 million tuples and up to 500 result groups. Indeed, the θ–MDA strategies presented in [2] feature a runtime being many times faster for queries with only equality constraint operators as compared to queries with constraint operators different than $=$. θ–MDA with late initialization aligns the execution times of the latter to queries with only equality constraint operators. The evaluation strategy for θ-MDA presented in this thesis represents a step forward with respect to the goal of efficient evaluation of complex multi-dimensional aggregation queries. The proposed solution was designed to work particularly well for cases in which the number of result groups is rather small. Future work could be to investigate limitations of our approach with respect to a growing number of result groups and to search for strategies to handle these limitations.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Motivation

Multi-dimensional aggregation queries are an important class of queries in data analysis finding application in several areas such as business intelligence [19], data stream analysis [8] and various scientific disciplines [2]. Especially *online analytical processing (OLAP)* is a mature approach concerned with the analysis of multi-dimensional data which has its roots in 1962 [13]. The first time the term OLAP was used, however, is in 1993 when Ted Codd published a white paper [6] defining 12 rules for the development of an OLAP product. Today OLAP is extensively used in business intelligence and features a broad tool support [21].

Since SQL/OLAP [11], an extension of SQL:99 [10], OLAP techniques for simple multi-dimensional data aggregation are supported directly by SQL, however, encountering limitations in the possibilities of formulating complex multi-dimensional aggregation queries. Although, with the introduction of window functions in SQL:2003 [12] some of these limitations have been faced, there are still aggregation queries which cannot be easily expressed in SQL resulting in complex formulations which cannot be handled effectively by the optimizer of database management systems. One class of such queries are cumulative aggregation over more than one dimension.

In order to face the limitations of SQL for complex multi-dimensional aggregation queries, in [2] an operator, referred to as *θ-constrained multi-dimensional aggregation (θ–MDA)*, is presented. This operator allows an easy formulation and efficient computation of aggregation queries over multi-dimensional data based on an approach which separates the definition of the result groups and of the aggregates in a clean way. Although, compared to SQL, θ–MDA features good performance in the execution of multi-dimensional aggregation queries, in some settings there is space for improvement. Indeed, queries in which the result groups are a selection-projection (SP) of the source relation result in loading and scanning that input relation twice. Due to the fact, that the source relation in typical θ–MDA scenarios is very large avoiding to scan it repeatedly would result is an improved performance.

1.2 Complex Aggregation Queries: An Example

An example of complex aggregation queries are moving and cumulative aggregations over one or more dimensions. As aforementioned not all such queries can be easily expressed in SQL and be handled efficiently by database management systems. With the introduction of *window functions* in SQL:2003 [12] a broader support for analytical queries was provided, such as moving and cumulative aggregation. However, there are still limitations. Indeed, window functions rely on a sorted input relation for the computation of the aggregates, thus, queries, for which no appropriate ordering of the input relation exists, are not supported. In order to illustrate this phenomenon, we continue with an example in the following.

Figure 1(a) presents a relation which will be used in all examples throughout the thesis. The table contains tuples representing services done for patients in a hospital. Each row represents a service performed for some patient such as a blood or urine test. The relation consists of four columns: the first column, *ServiceDate (SD)*, represents the date when the service was performed; the second column, *ServiceCode (SC)*, represents the code of the performed service which identifies it uniquely; the third column, *Urgency (U)*, specifies how urgent the need for the service has been (the higher the value the less urgent the need for the service); finally, the fourth column, *PatiengAge (PA)*, specifies the age of the patient who got the service.

LaboratoryServices (LS)

	SD	SC	U	PA
r_1	2010-11-12	U1	1	24
r_2	2010-11-12	H7	2	47
r_3	2010-11-12	U3	2	33
r_4	2010-11-12	U1	3	28
r_5	2010-11-13	H7	1	56
r_6	2010-11-13	H7	2	16
r_7	2010-11-13	F5	3	35
r_8	2010-11-13	U1	3	21
r_9	2010-11-14	U1	2	52

(a) *LaboratoryServices* Relation Instance.

x

	SD	U	$C_{SD,U}$	NC_{SD}	CC_{SD}	$CC_{SD,U}$
x_1	2010-11-12	1	1	5	4	1
x_2	2010-11-12	2	2	5	4	3
x_3	2010-11-12	3	1	5	4	4
x_4	2010-11-13	1	1	5	8	2
x_5	2010-11-13	2	1	5	8	5
x_6	2010-11-13	3	2	5	8	8
x_7	2010-11-14	2	1	8	9	6

(b) Result of *Query 1*.

Figure 1: Running Example

In the following we present *Query 1*, a complex multi-dimensional aggregation query which cannot be expressed directly based on SQL window functions. The input relation used in the query is *LS* of Figure 1(a).

> *Query 1*: Compute the number of services per day and urgency, the negated number of services per day, the cumulative number of services per day, and the cumulative number of services per day and urgency.

The result of *Query 1* is shown in Figure 1(b). The first aggregate $C_{SD,U}$, which counts all the services (i.e. tuples of the input relation) provided per day and urgency, can be expressed using the `GROUP BY` clause of SQL. For the computation of the second aggregate NC_{SD}, which counts the total number of provided services besides those of a specific day, two SQL queries using `GROUP BY` have to be joined. Next, the computation of the aggregate CC_{SD} that counts all services provided until a specific day, can be accomplished by the use of window functions, which allow to express cumulative aggregates. Window functions require the input relation to be sorted so that tuples belonging to the same group are arranged contiguously. In the case of CC_{SD} such an ordering of the tuples of LS exists, namely by sorting the input relation by SD resulting in the order r_1, r_2, ..., r_9. Finally, $CC_{SD,U}$, which computes a cumulative aggregate over two dimensions, cannot be computed using window functions. This is the case, because no proper ordering over the attributes SD and U can be found. For the tuples of LS two possible order exist: first, the order over SD, U results in the sequence r_1, r_2, ..., r_9; second, the order over U, SD results in the sequence r_1, r_5, r_2, r_3, r_6, r_9, r_4, r_7, r_8. However, none of these orders is appropriate for computing the aggregates with window functions. For instance, $CC_{SD,U}$ of tuple x_5 would require LS to be ordered by U, SD because the tuples r_1, r_2, r_3, r_5, r_6 contribute to the computation of that aggregate and, thus, have to be arranged in a contiguous sequence. However, tuple x_3 would require the ordering by SD, U, because the tuples that contribute to the computation of that aggregate are r_1, r_2, r_3. In order to express the aggregation query for $CC_{SD,U}$ in SQL a complex formulation is required which among others includes joins. As a consequence the input relation might be loaded into main memory several times. For large input relations this would result in unbearable execution times.

1.3 θ-Constrained Multi-Dimensional Aggregation and its Limitations

The θ–MDA operator presented in [2] evaluates analytical queries by identifying the input tuples which contribute to the computation of a specific aggregate based on θ-conditions. In figure 2 an overview of the parts involved in the evaluation of θ–MDA queries is provided. The parameters of the operator are an input table **r** containing all tuples over which the aggregates are computed, a base table **b** that stores all result groups for which the aggregates are computed, a list of aggregates, and θ-conditions associated to these aggregates. The θ-conditions define which tuples of **r** contribute to which aggregates.

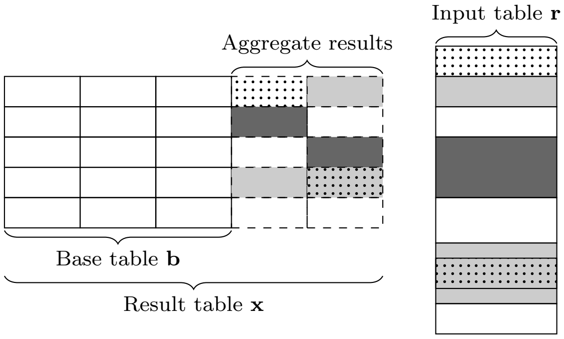


Figure 2: θ–MDA Operator Overview

The evaluation of aggregation queries with θ–MDA involves two main steps:

1. Construction of the base table (which can be any set of result group tuples).

2. Computation of the aggregates.

The strategy applied by θ–MDA for computing the aggregates is to iterate through the tuples of $\mathbf{r}$, and, for each of them to update the aggregates in the result relation $\mathbf{x}$ which are affected by that tuple.

The evaluation strategy for θ–MDA presented in [2] performs efficiently compared to SQL versions, however, in specific situations it shows limitations. Indeed, in cases in which the base table $\mathbf{b}$ is a SP of the input table $\mathbf{r}$, the evaluation process of θ–MDA has to load and iterate through all elements of $\mathbf{r}$ twice, one time for computing $\mathbf{b}$ (in step 1) and one time for computing $\mathbf{x}$ (in step 2). Because in typical θ–MDA scenarios the input relation is very large, loading and iterating through the tuples of $\mathbf{r}$ *twice* can be very expensive. Further, the runtime of the computation strategies for θ–MDA presented in [2] are sensitive with respect to the type of constraint operators used in the θ-conditions. Indeed, θ–MDA queries with constraint operators such as $\leq$, $\geq$, or $\neq$ feature execution times being many times longer than for queries with only equality constraint operators.

1.4 Contribution

In this thesis we focus on the limitation of the evaluation strategy for θ–MDA presented in [2] that loads and traverses the input table $\mathbf{r}$ twice, in the case in which the base table $\mathbf{b}$ is a projection of the input table $\mathbf{r}$, contributing as follows:

- We develop an evaluation approach for θ–MDA, to which we refer to as θ–MDA with late initialization. This new approach requires to load and traverse the input relation $\mathbf{r}$ only once for cases in which the base table is a SP of the input table. The approach supports the aggregate functions *sum*, *count*, *min*, *max*, and *avg* and θ-conditions using constraint operators $=$, $\leq$, $\geq$, $<$, $>$, and $\neq$.

- We define an algorithm for the developed approach with a time complexity being lower than the time complexity of the θ–MDA evaluation algorithms presented [2]. The time complexity of our algorithm is $O(|\mathbf{r}|)$ independently from the type of constraint operators used in the θ-conditions compared to the time complexity of the algorithms proposed in [2] which ranges from $O(|\mathbf{r}|)$ in the best case to $O(|\mathbf{r}| \cdot |\mathbf{b}|)$ in the worst case, depending on the constraint operators used[1].

- We evaluate the developed approach of θ–MDA with late initialization by implementing our algorithm as well as the algorithms presented in [2] as Java programs on top of an Oracle database and then testing them based on the TPC-H benchmark framework[2]. From the experiments our approach results to be more efficient in all tested settings and with a runtime not being sensitive to the type of constraint operators used.

[1] Note that $|\mathbf{r}|$ and $|\mathbf{b}|$ represent the number of tuples of the input relation and base relation respectively.

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

1.5 Organization of the Thesis

The remaining of the thesis is structured as follows. In Section 2 we present work related to the thesis topic. In Section 3 we provide more details about the θ–MDA operator and the strategy it uses for evaluating complex aggregation queries. Section 4 presents the evaluation approach we developed for θ–MDA, namely θ–MDA with *late initialization*, which requires only one scan of the input relation for the cases in which the base table is a projection of the input table. Section 5 illustrates the algorithm we propose for our θ–MDA evaluation strategy and in Section 6 we present the results of the experiments we run for evaluating the performance of the algorithm. Finally, in Section 7 concluding remarks and proposals for future work are presented.

2 Related Work

In literature there exists a vast variety of work about multi-dimensional data aggregation techniques for providing more control in the formulation of queries and a better performance. In [7] the *CUBE* operator is presented which is part of the SQL standard. This operator allows to express aggregation queries with equality constraints over several attributes in a concise way. For aggregation queries in which not the whole data cube is required, grouping sets [10] can be used. In the SQL standard an operator for grouping sets is provided which allows to compute aggregates over specified group attribute combinations and resulting in performance improvements with respect to *CUBE* in cases in which not the whole data cube is required. In [18] the *UNPIVOT* operator is introduced, which provides facilities to transform the data of a relation by switching columns to rows. All the mentioned operators provide more flexibility in expressing aggregation queries and improve their evaluation efficiency because a lower number of queries is required for the computation of the aggregates and because specialized algorithms can be adopted by the query execution engine. However, these operators provide a limited support for the expression of complex aggregation queries, so, for example, cumulative aggregates are not supported.

A broader support for complex multi-dimensional aggregation queries is provided by SQL since the introduction of *window functions* in SQL:2003 [12]. Indeed, the computation of cumulative aggregates is supported by the definition of a moving window over a sorted input relation. However, also window functions show limits in the computation of complex aggregates. For instance, the computation of cumulative aggregates over multiple dimensions is not possible. This is the case, because the computation strategy applied by window functions resides on a sorted input relation in which tuples belonging to the same group are arranged contiguously. However, for input relations grouped over more than one dimension there may exist no ordering so that the tuples of all groups are arranged contiguously.

Much effort has also been put into research focusing on the optimization of the evaluation efficiency of aggregation queries based on pre-computation of aggregates and incremental updates of pre-computed aggregates. In [9] a strategy for pre-computing data cubes based on materialized views is presented. This approach targets situations in which the materialization of all views, i.e. cells of the data cubes, is too expensive and provides a strategy for selecting a subset of views to be materialized based on which the other cells can be computed. The strategy resides on the use of a lattice that represents the dependencies among different views based on which an adequate set of the views to be materialized is computed. In order to avoid the re-computation of views as soon as the source relations change in [16] an approach is presented which allows to incrementally update pre-computed aggregates by considering only the changes of the source relations. The two main steps for the incremental view update are the *propagate* step, in which for each materialized view the aggregates over only the changes in the source relation are computed, referred to as *delta cuboids*, and a *refresh* step, in which the views are updated based on the computed *delta cuboids*. An improvement of this approach is presented in [14] which reduces the computational effort of the strategy by minimizing the number of *delta cuboids* to be computed. Another aggregate maintenance approach is presented in [22, 23] which targets temporal aggregation. They reason, that maintaining a materialized view for temporal aggregates might be too expensive, because with the insertion of one new tuple with a long time interval in the source relation, a big portion of the materialized view might have to be updated. They propose an approach for the maintenance of temporal aggregates in a tree data structure, more precisely an *SB-tree*. This data structure features characteristics of *B-trees*, ensuring good lookup performance, and of *segment-trees*, providing efficient updates even for tuples with long time intervals. Further, the approach they propose supports also cumulative temporal aggregates. In [25, 24] the *multiversion SB-tree* data structure is proposed, an extension of the *SB-tree* which supports range-temporal aggregation queries, i.e. temporal aggregation queries with a predicate over the key (group attribute value) range. Although, the performance optimization of aggregation queries based on (partially) pre-computed aggregates can provide good results there exist settings in which more flexibility is required.

Further research in the area of multi-dimensional aggregation queries has been done around the *multi-dimensional join (MDJ)* operator introduced in [1]. This operator and its generalized version, *generalized multi-dimensional join (GMDJ)* [3], provide a tool for an easy formulation of complex aggregation queries over multi-dimensional data and an efficient evaluation of such queries. The evaluation approach adopted by *MDJ* separates the computation of the result groups and of the aggregates making them more independent and, thus, providing more flexibility in their definition. Further, the evaluation approach enables an efficient computation of ad-hoc queries which might lack of adequate indexes or pre-computed views, by reducing the number of required scans of the source relations. The computation of the aggregates is done based on general θ-conditions which are used to identify the tuples of the source relation which affect the aggregates. In [2] the *θ-constrained multi-dimensional aggregation (θ–MDA)* operator, based on *GMDJ*, is presented together with a formally defined cost model, an exhaustive definition of transformation rules for θ–MDA to SQL, and performance experiments showing the better performance of θ–MDA with respect to its formulation in SQL. Further, in [2] recent advances of the support for multi-dimensional aggregation queries in commercial database management systems is considered. In [4] the use of the *GMDJ* for the efficient computation of sub-queries in complex OLAP settings is presented. An algorithm that transforms general sub-query expressions into expressions which use *GMDJ* instead of joins, outer joins, or set difference is provided. In [5] and [17] strategies for an efficient evaluation of aggregation queries based on *GMDJ* in a distributed computational environment are presented. In the former, the *GMDJ* is used for the computation of aggregation queries over data spread among different data warehouses. A central coordinator component is responsible for forwarding a *GMDJ* query to the different data warehouses and for merging the results computed by those data warehouses once they are returned. In the latter, efficient computation of analytical aggregation queries over RDF [15] data is presented. The approach uses *GMDJ* in connection with *Map Reduce*[3] to compute aggregation queries over RDF data. The work presented in this thesis is also based on the *MDJ* operator, more precisely on θ–MDA, and proposes an algorithm for θ–MDA which improves the evaluation performance of complex multi-dimensional aggregation queries.

[3] *Map Reduce* is a strategy for distributed computing over large amounts of data [20]

3 θ-Constrained Multi-Dimensional Aggregation

θ-constrained multi-dimensional aggregation is an operator for the evaluation of aggregation queries over multi-dimensional data [2]. As presented before, the operator computes aggregates for the result groups of a base table using an input table storing the detailed data and θ-conditions for the association of tuples of the input table to the result groups.

θ–MDA gets as input four parameters which are the base table, the input table, a list of aggregates which have to be computed and θ-conditions used for computed the aggregates. Before continuing with a formal definition of the operator we introduce some notions which will be used in the rest of the thesis. We will use algebraic operators, such π, σ, or $\cup$. We will use single curly brackets $\{...\}$ for sets and double curly brackets $\{\{...\}\}$ for bags. Further, $\mathbf{B}$ and $\mathbf{R}$ are used to denote database schemas $(B_1, ..., B_t)$ and $(R_1, ..., R_p)$ respectively and $x.\mathbf{B}$ is used to refer to $(x.B_1, ..., x.B_t)$. $E \rightarrow C$ denotes the renaming of E to C, $attr(E)$ denotes the set of attributes used in E and f_i denotes an aggregate function. Based on these notions the θ–MDA operator is defined as follows.

Definition 1 (θ**–MDA Operator [2]**)**.** Let $\mathbf{b}(\mathbf{B})$ and $\mathbf{r}(\mathbf{R})$ be tables, θ_i, $1 \leq i \leq m$, be conditions with $attr(\theta_i) \subseteq attr(\mathbf{B}) \cup attr(\mathbf{R})$, let $\Theta = (\theta_1, ...\theta_m)$, let $l_i = (f_{i_1}(A_{i_1}) \rightarrow C_{i_1}, ..., f_{i_{k_i}}(A_{i_{k_i}}) \rightarrow C_{i_{k_i}})$, $1 \leq i \leq m$, be a list of aggregate functions over attributes $A_{i_1}, A_{i_2}, ..., A_{i_{k_i}}$ in $attr(\mathbf{R})$, and let $L = (l_1, ..., l_m)$. The θ–MDA operator is defined as

$$\mathbf{x} = \mathcal{G}^{\theta}(\mathbf{b}, \mathbf{r}, L, \Theta)$$

where $\mathbf{X} = (\mathbf{B}, C_{1_1}, ..., C_{1_k}, ..., C_{m_1}, ..., C_{m_{k_m}})$ is the schema of the result table and each tuple $b \in \mathbf{b}$ produces a result tuple $x \in \mathbf{x}$ with

- $x.\mathbf{B} = b.\mathbf{B}$.
- $x.C_{i_j} = f_{i_j}(\{r.A_{i_j} | r \in R \wedge \theta_i(b, r)\})$, for each $C_{i_j} \in \mathbf{X}$.

In Definition 1 the base table is denoted by $\mathbf{b}$, the input table by $\mathbf{r}$ and the result table by $\mathbf{x}$. The aggregates of the result table are $x.C_{i_j}$.

Example 1. In this example we present the formal definition of *Query 1* based on the θ–MDA operator introduced in Definition 1. The operator for the query is

$$\mathcal{G}^{\theta}(\mathbf{b}, LaboratoryServices \rightarrow LS, (l_1, l_2, l_3, l_4), (\theta_1, \theta_2, \theta_3, \theta_4))$$

and the parameters of the operator are defined as follows:

$$\begin{aligned}
&\mathbf{b} : \pi_{SD,U} LaboratoryServices\\
&l_1 : (count(SD) \rightarrow C_{SD,U})\\
&\theta_1 : \mathbf{r}.SD = \mathbf{b}.SD \wedge \mathbf{r} = \mathbf{b}.U\\
&l_2 : (count(SD) \rightarrow NC_{SD})\\
&\theta_2 : \mathbf{r}.SD \neq \mathbf{b}.SD\\
&l_3 : (count(SD) \rightarrow CC_{SD})\\
&\theta_3 : \mathbf{r}.SD \leq \mathbf{b}.SD\\
&l_4 : (count(SD) \rightarrow CC_{SD,U})\\
&\theta_4 : \mathbf{r}.SD \leq \mathbf{b}.SD \wedge \mathbf{r} \leq \mathbf{b}.U
\end{aligned}$$

The evaluation steps of *Query 1* based on θ–MDA are presented in Table 1. At the beginning all aggregates of $\mathbf{x}$ are 0. At each step of the computation one tuple of the input relation is processed and all aggregates in $\mathbf{x}$ affected by that tuple are updated. In our example we start with processing tuple r_1 and all affected aggregates are incremented by 1. Then the tuple r_1 is discarded and the next tuple of the input relation is loaded, namely r_2. After having processed all tuples of the input relation in this way, the result table is computed.

LS

	SD	U	...
r_1	12	1	...
r_2	12	2	...
r_3	12	2	...
r_4	12	3	...
r_5	13	1	...
r_6	13	2	...
r_7	13	3	...
r_8	13	3	...
r_9	14	2	...

$\mathbf{x}$

	SD	U	$C_{SD,U}$	NC_{SD}	CC_{SD}	$CC_{SD,U}$
x_1	12	1	0	0	0	0
x_2	12	2	0	0	0	0
x_3	12	3	0	0	0	0
x_4	13	1	0	0	0	0
x_5	13	2	0	0	0	0
x_6	13	3	0	0	0	0
x_7	14	2	0	0	0	0

LS

	SD	U	...
~~r_1~~	~~12~~	~~1~~	...
r_2	12	2	...
r_3	12	2	...
r_4	12	3	...
r_5	13	1	...
r_6	13	2	...
r_7	13	3	...
r_8	13	3	...
r_9	14	2	...

$\mathbf{x}$

	SD	U	$C_{SD,U}$	NC_{SD}	CC_{SD}	$CC_{SD,U}$
x_1	12	1	**1**	0	**1**	**1**
x_2	12	2	0	0	**1**	0
x_3	12	3	0	0	**1**	0
x_4	13	1	0	**1**	0	0
x_5	13	2	0	**1**	0	0
x_6	13	3	0	**1**	0	0
x_7	14	2	0	**1**	0	0

LS

	SD	U	...
~~r_1~~	~~12~~	~~1~~	...
~~r_2~~	~~12~~	~~2~~	...
r_3	12	2	...
r_4	12	3	...
r_5	13	1	...
r_6	13	2	...
r_7	13	3	...
r_8	13	3	...
r_9	14	2	...

$\mathbf{x}$

	SD	U	$C_{SD,U}$	NC_{SD}	CC_{SD}	$CC_{SD,U}$
x_1	12	1	1	0	**2**	1
x_2	12	2	**1**	0	**2**	**1**
x_3	12	3	0	0	**2**	0
x_4	13	1	0	**2**	0	0
x_5	13	2	0	**2**	0	0
x_6	13	3	0	**2**	0	0
x_7	14	2	0	**2**	0	0

LS

	SD	U	...
~~r_1~~	~~12~~	~~1~~	...
~~r_2~~	~~12~~	~~2~~	...
~~r_3~~	~~12~~	~~2~~	...
~~r_4~~	~~12~~	~~3~~	...
~~r_5~~	~~13~~	~~1~~	...
~~r_6~~	~~13~~	~~2~~	...
~~r_7~~	~~13~~	~~3~~	...
~~r_8~~	~~13~~	~~3~~	...
~~r_9~~	~~14~~	~~2~~	...

$\mathbf{x}$

	SD	U	$C_{SD,U}$	NC_{SD}	CC_{SD}	$CC_{SD,U}$
x_1	12	1	1	5	4	1
x_2	12	2	2	5	4	3
x_3	12	3	1	5	4	4
x_4	13	1	1	5	8	2
x_5	13	2	1	5	8	5
x_6	13	3	2	5	8	8
x_7	14	2	1	8	9	6

Table 1: *Query 1* Computation Steps.

In [2] two algorithms for the computation of θ–MDA queries based on the approach presented above are proposed.

The first algorithm, `BasicTCMDA` (see page 11), consists of four steps and applies a brute force approach for computing the result relation $\mathbf{x}$. In step 1 of the algorithm, algebraic aggregates are replaced by distributive sub-aggregates. For example, the aggregate *avg* is replace by the aggregates *sum* and *count*. This step is required in order to have a correct computation of the aggregates. Next, in step 2 the result relation with all aggregates set to their initial values is constructed. For the aggregates *count* and *sum* the initial value is 0 and for the aggregates *min* and *max* it is *NULL*. Then, in step 3 for each tuple of the input table $\mathbf{r}$ and each tuple of $\mathbf{x}$ all θ-conditions are checked, and for matching ones the aggregate is updated. Finally, in step 4 the sub-aggregates transformed in step 1 are transformed back into algebraic ones.

The second algorithm, `IndexedTCMDA` (see page 12), is similar to the algorithm `BasicTCMDA`, but instead of iterating through all tuples of $\mathbf{x}$ for each input tuple of $\mathbf{r}$ and θ-condition, it uses an index to find only those tuples which are affected by the processed tuple of $\mathbf{r}$. Step 1 and step 4 of `IndexedTCMDA`, in which the aggregates transformation occurs, are exactly the same as in `BasicTCMDA`. In step 2 the only difference to step 2 of `BasicTCMDA` is that an index for the result table $\mathbf{x}$ is created. And in step 3 for each processed tuple of the input table $\mathbf{r}$ and each θ-condition, all tuples of $\mathbf{x}$ are retrieved for which the θ-condition holds. For retrieving these tuples the index is used. In this way the algorithm performes less iterations. For instance, if only one tuple of $\mathbf{x}$ is affected than the algorithm iterates over only this tuple instead of all tuples of $\mathbf{x}$. Based on the use of the index the `IndexedTCMDA` algorithm is faster than `BasicTCMDA` for restrictive θ-conditions, i.e. if the θ-conditions match only few tuples of $\mathbf{x}$ for the processed tuples of $\mathbf{r}$. For instance, queries with only equality constraint operators are very restrictive, instead, queries with constraint operators such as $\leq$, $\geq$, or $\neq$ are less restrictive.

Algorithm 1: BasicTCMDA($\mathbf{b}, \mathbf{r}, (l_1, ..., l_m), (\theta_1, ..., \theta_m)$)

```
// Step 1: Replace algebraic aggregates by distributive
   sub-aggregates
Let $l'_i = l_i, 1 \leq i \leq m$;
foreach algebraic aggregate $f_{i_j} \in \{l'_1, ..., l'_m\}$ do
    Replace $f_{i_j}$ with its distributive sub-aggregates $f^1_{i_j}, ..., f^{p_{i_j}}_{i_j}$
end

// Step 2: Construct result table x
// Note: $v_j$'s are the initial aggregate values (0 for count
   and sum, NULL for min and max
Let $\mathbf{N} = (v_{1_1}, ..., v_{m_1}, ..., v_{m_{k_m}})$;
Let $\mathbf{x} = \mathbf{b} \times N$;

// Step 3: Compute the aggregates
foreach tuple $r \in \mathbf{r}$ do
    foreach row $x \in \mathbf{x}$ do
        foreach $\theta_i \in \{\theta_1, ..., \theta_m\}$ do
            if $\theta_i(x, r)$ is true then
                Update the aggregates $f_{i_1}, ..., f_{i_{k_i}}$ in x;
            end
        end
    end
end

// Step 4: Apply the super-functions
if $l_1, ..., l_m$ contains algebraic aggregates then
    foreach row $x \in \mathbf{x}$ 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
            $x.f_{i_j} = g_i(x.f^1_{i_j}, ..., x.f^{p_{i_j}}_{i_j})$;
        end
    end
end

return x;
```

Algorithm 2: IndexedTCMDA($\mathbf{b}, \mathbf{r}, (l_1, ..., l_m), (\theta_1, ..., \theta_m)$)

```
// Step 1: Replace algebraic aggregates by distributive
   sub-aggregates
(Same as step 1 in BasicTCMDA).

// Step 2: Construct result table and indexes
// Note: $v_j$'s are the initial aggregate values (0 for count
   and sum, NULL for min and max
Let $\mathbf{N} = (v_{1_1}, ..., v_{m_1}, ..., v_{m_{k_m}})$;
Let $\mathbf{x} = \mathbf{b} \times N$;
Build index for $\mathbf{x}$;

// Step 3: Compute the aggregates
foreach tuple $r \in \mathbf{r}$ do
    foreach $\theta_i \in \{\theta_1, ..., \theta_m\}$ do
        Fetch the rows $\mathbf{x}_i = \{x \in \mathbf{x} | \theta_i(x, r)\}$ using the index;
        foreach $x \in \mathbf{x}_i$ do
            Update the aggregates $f_{i_1}, ..., f_{i_{k_i}}$ in $x$;
        end
    end
end

// Step 4: Apply the super functions
Same as in BasicTCMDA.

return $\mathbf{x}$;
```

4 Late Initialization for θ–MDA

In the previous sections we learned about the θ–MDA operator and that it performs better than SQL implementations for complex aggregation queries over multi-dimensional data. Among others, one parameter of the operator is the base relation which contains all the groups for which the aggregate values have to be computed. In the current version of the evaluation algorithms for θ–MDA the base table has to be constructed completely before the computation of the result table begins. For cases in which the base table is a projection of the input relation $\mathbf{r}$ this results in two scans of $\mathbf{r}$, one scan for computing $\mathbf{b}$ and one scan for computing $\mathbf{x}$. In this section we propose an approach for θ–MDA that requires only one scan of the input relation $\mathbf{r}$ in cases in which the base table is a projection of $\mathbf{r}$. This approach is based on the current θ–MDA strategy presented in the previous section and we refer to it as *θ–MDA with late initialization*.

The rest of this section is structured as follows. In Section 4.1 we define the concept of late initialization for θ–MDA. Section 4.2 presents different cases which have to be considered by the late initialization approach. In Section 4.3 our evaluation strategy for θ–MDA with late initialization is presented. Finally, in Section 4.4 an optimization of that strategy is introduced.

4.1 Definition

For the computation of θ–MDA queries based on the evaluation strategies presented in [2], in cases in which the base table is a projection of the input table $\mathbf{r}$, two scans of $\mathbf{r}$ are required, one for computing $\mathbf{b}$ and one for computing $\mathbf{x}$. In this section we introduce the main idea of an approach that, for cases in which the base table is a projection of the input relation, allows to compute the result of θ–MDA by scanning $\mathbf{r}$ only once. This goal can be achieved, by computing the elements of the base table on demand while the aggregates of the result relation are computed. We refer to this computation on-the-fly of the base table elements as late initialization.

In Figure 3 the idea of late initialization for θ–MDA is presented graphically by comparing the main evaluation workflow of the θ–MDA approach defined in Section 3 (see Figure 3(a)) with the computational workflow of θ–MDA with late initialization (see Figure 3(b)). In the first case, the base table $\mathbf{b}$ has to be computed before continuing with the computation of the result relation $\mathbf{x}$, requiring two scans of the input relation $\mathbf{r}$. In the second case, instead, the tuples of $\mathbf{b}$ are computed on demand while computing the aggregates of $\mathbf{x}$, requiring only one scan of $\mathbf{r}$. In the following a formal definition of late initialization for θ–MDA is provided.

Definition 2 (**θ–MDA with Late Initialization**). Let $\mathbf{r}(\mathbf{R})$ be a table, θ_i, $1 \leq i \leq m$, be conditions with $attr(\theta_i) \subseteq attr(\mathbf{R})$, let $\Theta = (\theta_1, ...\theta_m)$, let $l_i = (f_{i_1}(A_{i_1}) \to C_{i_1}, ..., f_{i_{k_i}}(A_{i_{k_i}}) \to C_{i_{k_i}})$, $1 \leq i \leq m$, be a list of aggregate functions over attributes $A_{i_1}, A_{i_2}, ..., A_{i_{k_i}}$ in $attr(\mathbf{R})$, and let $L = (l_1, ..., l_m)$. Let $\mathbf{b}(\mathbf{B}) = \pi_{B_1,...,B_t}(\sigma(\mathbf{r}))$ be a SP of $\mathbf{r}$ with $B_1, ..., B_t$ in $attr(\mathbf{R})$. Late initialization for θ–MDA is defined as the strategy that computes the tuples of $\mathbf{b}$ on demand while the tuples of $\mathbf{r}$ are processed for the computation of the aggregates of $\mathbf{x}$, i.e. that requires only one scan of $\mathbf{r}$ in order to compute $\mathbf{x}$, by proceeding as follows:

1. Start with an empty result relation $\mathbf{x} \leftarrow \emptyset$.
2. Let $\mathbf{r}' \subset \mathbf{r}$ be the set of argument tuples that have already been processed and $r \in \mathbf{r} \setminus$ **r'** the current tuple. Then the result table $\mathbf{x}$ is updated as follows:

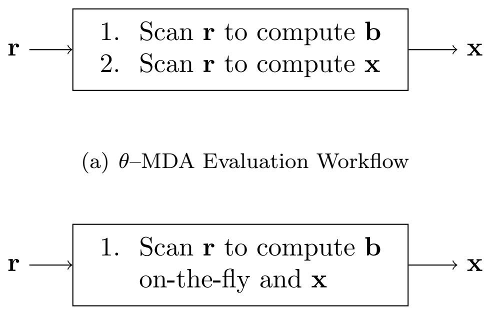
   (a) If $!\exists x \in \mathbf{x}(x.\mathbf{B} = r.\mathbf{B})$ then

   $x \leftarrow r.\mathbf{B} \times (C'_{1_1}, ..., C'_{m_{k_m}})$

   with $x.C'_{i_j} \leftarrow f_{i_j}(\{\{r'.A_{i_j} \mid r' \in \mathbf{r}' \wedge \theta_i(x, r')\}\});$

   $\mathbf{x} \leftarrow \mathbf{x} \cup \{x\};$

   (b) $\forall x \in \mathbf{x}, \forall \theta_i \in \{\theta_1, ..., \theta_m\}$ update aggregates $f_{i_1}, ..., f_{i_{k_i}}$ if $\theta_i(x, r)$ is `true`.



(a) θ–MDA Evaluation Workflow

(b) θ–MDA with Late Initialization Evaluation Workflow

Figure 3: Query Evaluation Workflow

From Definition 2 it is possible to recognize that the computational process of θ–MDA with late initialization starts with an empty result table $\mathbf{x}$. Then, it starts to iterate through all the tuples r of $\mathbf{r}$ and extends $\mathbf{x}$ if the result group to which r belongs is not already given in $\mathbf{x}$. Further, the aggregates of a newly added tuple of $\mathbf{x}$, as well as, of the already existing tuples of $\mathbf{x}$ are computed or updated respectively. Based on this approach of late initialization the result groups of $\mathbf{b}$ are created as needed while the aggregates in $\mathbf{x}$ are computed, requiring only one scan of $\mathbf{r}$.

Example 2. In this example we illustrate the single computational steps of the late initialization strategy of Definition 2. Table 2 presents the steps for computing the aggregate $C_{SD,U}$ of *Query 1* based on that strategy. We can see that the computation starts with an empty result relation $\mathbf{x}$ and at each step one tuple of $\mathbf{r}$ is processed. The first tuple of $\mathbf{r}$ to be processed is r_1. Due to the fact, that at the point of processing r_1 the table $\mathbf{x}$ does not contain any result group with SD = 2010-11-12 and $U = 1$, a new tuple, x_1, is added to $\mathbf{x}$. The aggregate of x_1 is initialized to 0 because r_1 is the first tuple to be processed and, therefore, there are no previously processed tuples of $\mathbf{r}$ which could affect

x_1. Next, all aggregates of tuples of **x** which are affected by r_1 are incremented by 1. In our case only $C_{SD,U}$ of x_1 has to be incremented. Next, we proceed with processing tuple r_2. Also in this case there exists no entry in **x** which represents the result group to which r_2 belongs, therefore, a new tuple, x_2, is added to **x**. Due to the fact, that no previously processed tuples of **r** affect the aggregate of x_2, $C_{SD,U}$ is set to 0. Then, all the aggregates affected by r_2 are incremented by 1, which in our case is only $C_{SD,U}$ of x_2. We continue with processing tuple r_3. In this case there already exists a tuple in **x** representing the result group to which r_3 belongs, thus, no new tuple has to be added to **x**. Therefore, we proceed by updating all aggregates of tuples affected by r_3, which is only x_2. We increment $C_{SD,U}$ by 1. For the remaining tuples of **r** we proceed in the same way as for r_1 to r_3. After having processed all tuples of **r**, the result relation **x** for aggregate $C_{SD,U}$ is entirely computed.

	x	SD	U	$C_{SD,U}$
r_1(2010-11-12, U1, 1, 24)	x_1	**2010-11-12**	**1**	**1**
r_2(2010-11-12, H7, 2, 47)	x_1	2010-11-12	1	1
	x_2	**2010-11-12**	**2**	**1**
r_3(2010-11-12, U3, 2, 33)	x_1	2010-11-12	1	1
	x_2	2010-11-12	2	**2**
...	...			
r_9(2010-11-14, U1, 2, 52)	x_1	2010-11-12	1	1
	x_2	2010-11-12	2	2
	x_3	2010-11-12	3	1
	x_4	2010-11-13	1	1
	x_5	2010-11-13	2	1
	x_6	2010-11-13	3	2
	x_7	**2010-11-14**	**2**	**1**

Table 2: Example of θ–MDA with Late Initialization

In the above example, newly added tuples of **x** are not affected by previously processed tuples of **r**. We choose this example in order to focus on the main idea of the late initialization approach for θ–MDA. However, the aggregates of newly added tuples of **x** can be affected by one or more tuples of **r** processed in previous steps. In such cases, the computation of the aggregates of that newly added tuples can be challenging. Indeed, for the computation of that aggregates one or more of the previously processed tuples of **r** would be required. However, those tuples are not available anymore, because after being processed they are discarded. Due to the fact, that the aggregates of the already existing tuples of **x** store the information about previously processed tuples of **r**, the required information can be extracted from the aggregates of **x**. Theorem 1 states this fact formally.

Theorem 1 (Aggregate Computation for new Tuples of $\mathbf{x}$). *Let* $\boldsymbol{r}(\boldsymbol{R})$ *be a table,* $\boldsymbol{b}(\boldsymbol{B}) = \pi_{B_1,...,B_t}(\sigma(\boldsymbol{r}))$ *a SP of* $\boldsymbol{r}$*, and* r *a tuple of* $\boldsymbol{r}$*. If we compute* $\boldsymbol{x}$ *using the* θ*–MDA strategy of Definition 2 and* r *is the currently processed tuple of* $\boldsymbol{r}$*, and if there exists no tuple* x *in* $\boldsymbol{x}$ *with* $x.\boldsymbol{B} = r.\boldsymbol{B}$*, then a new tuple* $x = x.\boldsymbol{B} \times (C'_{i_j}, ..., C'_{m_{k_m}})$ *has to be added to* $\boldsymbol{x}$*. The aggregates*

$$x.C'_{i_j} = F(\boldsymbol{x}, f_{i_j}, \theta_i)$$

of the new tuple x *can be computed from the aggregates of the already existing tuples of* $\boldsymbol{x}$*.*

The theorem states that the aggregates, $x.C'_{i_j}$, of newly added tuples of $\mathbf{x}$ can be computed by a function $F(\mathbf{x}, f_{i_j}, \theta_i)$. This function extracts the required information about previously processed tuples $\mathbf{r}' \subset \mathbf{r}$, which affect the aggregate $x.C'_{i_j}$, from the aggregates of the already existing tuples of $\mathbf{x}$, thus, not requiring to access the tuples in $\mathbf{r}'$ directly. However, the strategy needed for extracting the needed information from the aggregates in $\mathbf{x}$ depends from the type of constraint operators used in θ_i, thus, requiring different functions $F(\mathbf{x}, f_{i_j}, \theta_i)$.

4.2 Case Analysis

In this section we study the different cases arising when computing the aggregates of newly added tuples of $\mathbf{x}$, with the help of $F(\mathbf{x}, f_{i_j}, \theta_i)$, due to the use of different constraint operators in the θ-conditions. Before continuing with the analysis we introduce the notion of *θ-result-groups* in Definition 3, which is going to be used in the rest of this thesis.

Definition 3 (θ-Result-Group). Let $\mathbf{r}(\mathbf{R})$ be a table, θ_i, $1 \leq i \leq m$, be conditions with $attr(\theta_i) \subseteq attr(\mathbf{R})$, let $\Theta = (\theta_1, ...\theta_m)$, let $l_i = (f_{i_1}(A_{i_1}) \to C_{i_1}, ..., f_{i_{k_i}}(A_{i_{k_i}}) \to C_{i_{k_i}})$, $1 \leq i \leq m$, be a list of aggregate functions over attributes $A_{i_1}, A_{i_2}, ..., A_{i_{k_i}}$ in $attr(\mathbf{R})$, let $L = (l_1, ..., l_m)$. Let $\mathbf{b}(\mathbf{B}) = \pi_{B_1,...,B_t}(\sigma(\mathbf{r}))$ be a SP of $\mathbf{r}$ with $B_1, ..., B_t$ in $attr(\mathbf{R})$, and let $\mathbf{x} = \mathcal{G}^\theta(\mathbf{b}, \mathbf{r}, L, \Theta)$. Then

$$\mathbf{g}_i^\theta = \pi_{attr(\theta_i)}(\mathbf{x})$$

is the set of all the result groups in $\mathbf{x}$ for the condition θ_i, i.e. the set of θ-result-groups for θ_i.

From Definition 3 we can deduce the that one θ-result-group $g_i^\theta \in \mathbf{g}_i^\theta$ may occur in more than one tuple of $\mathbf{x}$, if $attr(\theta_i) \subset attr(\mathbf{x})$.

4.2.1 θ-Conditions with the Constraint Operator =

In this section we present the application of late initialization to θ–MDA for θ-conditions with only equality constraint operators. We will do this based on an example and show that for θ-conditions with only equality constraint operators late initialization can be applied without encountering difficulties. i.e. a simple strategy for computing $F(\mathbf{x}, f_{i_j}, \theta_i)$ exists.

Indeed, in a scenario with conditions using only equality constraints, each tuple of the input relation affects exactly one θ-result-group. Due to the fact, that more than one tuple in $\mathbf{x}$ can store the same θ-result-group, the situation can arise in which for a newly inserted tuple of $\mathbf{x}$, there already exist entries in $\mathbf{x}$ storing the θ-result-group for a specific condition θ_i. In such cases, the aggregates of that tuples of $\mathbf{x}$ have to be considered for the computation of the aggregates of the new tuple of $\mathbf{x}$. Because for conditions with only equality constraint operators there exists only one θ-result-group per condition, only one tuple of $\mathbf{x}$ storing the θ-result-group has to be considered, because it stores all the information about the already processed tuples of $\mathbf{r}$ belonging to the same θ-result-group. In the following we present this situation by an example.

Example 3. In this example we illustrate the steps that have to be done in order to compute $\mathbf{x}$ with late initialization for a query with θ-conditions using only the constraint operator $=$. The definition of the input parameters of the θ–MDA operator for this example are the following:

$$\begin{aligned}
&\mathbf{r} : LS\\
&\mathbf{b} : \pi_{SD,U}(LS)\\
&l_1 : (count(SD) \rightarrow C_{SD,U})\\
&\theta_1 : \mathbf{r}.SD = \mathbf{b}.SD\\
&l_2 : (count(SD) \rightarrow C_{SD,U})\\
&\theta_2 : \mathbf{r}.SD = \mathbf{b}.SD \wedge \mathbf{r}.U = \mathbf{b}.U
\end{aligned}$$

The input relation used in this example is the relation LS of our running example (Table 1(a) on page 2). The aggregates C_{SD} and $C_{SD,U}$ of a specific tuple of $\mathbf{x}$ count the number of tuples of $\mathbf{r}$ belonging to the θ-result-groups represented by that tuple.

In Table 3 we illustrate the single steps required to compute $\mathbf{x}$ with late initialization. It is possible to recognize, that we start with an empty result relation $\mathbf{x}$. Subsequently, at each step in this example, one tuple of the input relation is processed and the result relation updated accordingly. In the following paragraphs the steps for computing $\mathbf{x}$ are described in more detail.

The first tuple to be processed in Table 3 is r_1. At that point, there exists no tuple in $\mathbf{x}$ which has the values 2010-11-12 and 1 for the attributes SD and U respectively. Thus, a new tuple x_1 is inserted into $\mathbf{x}$ and the aggregates C_{SD} and $C_{SD,U}$ of this tuple are computed. Because r_1 is the first tuple of $\mathbf{r}$ to be processed, no other tuple of $\mathbf{r}$ has to be considered for the computation of the aggregates of x_1. Thus, both aggregates C_{SD} and $C_{SD,U}$ are set to 0. Next, all tuples of $\mathbf{x}$ affected by r_1 are updated, resulting in the aggregates of x_1 to be $C_{SD} = 1$ and $C_{SD,U} = 1$.

The second tuple to be processed is r_2 and also for this tuple a new entry, x_2, has to be added to $\mathbf{x}$. The computation of the aggregates for x_2, however, differs from the previous case. In fact, for the computation of C_{SD}, for which only the attribute SD has to be considered, there already exists tuple in $\mathbf{x}$ storing the θ-result-group for θ_1 to which the tuple r_2 belongs, namely x_1. Thus, x_1 has to be taken into account for the computation of C_{SD} of x_2. Therefore, C_{SD} is initialized to 1, and $C_{SD,U}$ to 0. Then, all aggregates of $\mathbf{x}$ affected by r_2 are updated, which are both aggregates of x_2 and the aggregate C_{SD} of x_1.

Continuing this process until each tuple of the input relation has been processed, produces a result relation $\mathbf{x}$ with correctly computed aggregates. This shows, that the computation of the aggregates of newly added tuples of $\mathbf{x}$ can be easily accomplished for θ-conditions which contain only equality constraint operators. The presented approach works also if any of the aggregate functions *sum*, *min*, *max*, and *avg*[4] is used.

x

		SD	U	C_{SD}	$C_{SD,U}$
r_1(2010-11-12, U1, 1, 24)	x_1	**2010-11-12**	**1**	**1**	**1**
r_2(2010-11-12, H7, 2, 47)	x_1	2010-11-12	1	**2**	1
	x_2	**2010-11-12**	**2**	**2**	**1**
r_3(2010-11-12, U3, 2, 33)	x_1	2010-11-12	1	**3**	1
	x_2	2010-11-12	2	**3**	**2**
...		...			
r_9(2010-11-14, U1, 2, 52)	x_1	2010-11-12	1	4	1
	x_2	2010-11-12	2	4	2
	x_3	2010-11-12	3	4	1
	x_4	2010-11-13	1	4	1
	x_5	2010-11-13	2	4	1
	x_6	2010-11-13	3	4	2
	x_7	**2010-11-14**	**2**	**1**	**1**

Table 3: Example of Late Initialization for Equality

4.2.2 θ-Conditions with the Constraint Operators $\leq$, $\geq$, $<$, $>$

In this section we continue with the presentation of how to apply the late initialization strategy for θ–MDA to queries with conditions using the constraint operators $\leq$, $\geq$, $<$, and $>$. We will do this by an example of θ–MDA in which only the constraint operator $\leq$ is used. We will see that the computation of the aggregates of newly added tuples is not as straightforward as for θ–MDA queries with conditions using only equality. θ–MDA queries with θ's using $\geq$, $<$, and $>$ feature the same characteristics as for $\leq$ and, thus, are not presented in more detail here.

In the previous section we learned, that for θ-conditions with only equality constraints, each tuple of **r** affects exactly one θ-result-group of **x**. In the case of conditions with one of the constraint operators $\leq$, $\geq$, $<$, and $>$ this is different. Indeed, in such cases each tuple of **r** can affect more than one θ-result-group of **x**. This means, that the information about already processed tuples of **r** is spread among several tuples of **x** and, thus, has to be extracted from them. However, as we will see later in this section, the problem in doing this is that the aggregates of the different θ-result-groups which have to be considered for computing the aggregates of a new tuple of **x** might store redundant information about already processed tuples of **r**. This information must be removed for a correct computation of the new aggregates. In the following we continue with the presentation of the example.

[4] In the case of *avg*, the aggregates has to be split into *count* and *sum* in order to be computed correctly as mentioned in Section 3.

Example 4. In this example we illustrate the computation of $\mathbf{x}$ with late initialization for a query using only the constraint operator $\leq$. The definition of the θ–MDA operator parameters is the following:

$$
\begin{aligned}
&\mathbf{r} : LS \\
&\mathbf{b} : \pi_{SD,U} LS \\
&l_1 : (count(SD) \rightarrow CC_{SD}) \\
&\theta_1 : \mathbf{r}.SD \leq \mathbf{b}.SD \\
&l_2 : (count(SD) \rightarrow CC_{SD,U}) \\
&\theta_2 : \mathbf{r}.SD \leq \mathbf{b}.SD \wedge \mathbf{r}.U \leq \mathbf{b}.U
\end{aligned}
$$

Also in this example the used input relation is LS of our running example. The aggregates CC_{SD} and $CC_{SD,U}$ represent cumulative counts, i.e. they count all the tuples of $\mathbf{r}$ with values for the attributes SD and U being smaller-equal than the values of SD or SD and U of the considered tuple of $\mathbf{x}$ respectively.

In Table 4 we illustrate the situation in which $\mathbf{x}$ reflects the result relation after a correct processing of the tuples r_1 to r_5 of the input table, and r_6 is the next tuple of $\mathbf{r}$ to be processed. Due to the fact, that $\mathbf{x}$ does not contain any tuple with $SD = $ 2010-11-13 and $U = 2$, a new tuple, x_5, has to be added to $\mathbf{x}$. The next step to do is to compute the aggregates of x_5. Considering $CC_{SD,U}$, first we have to retrieve the aggregates $CC_{SD,U}$ of θ-result-groups which store information about previously processed tuples of $\mathbf{r}$ which affect the aggregate of the new tuple x_5. This information is store in the tuples x_1, x_2, and x_4, because they count tuples of $\mathbf{r}$ with values for SD and U being smaller than 2010-11-13 and 2 respectively. Due to the fact, that the aggregate values of x_1, x_2, and x_4 are already cumulative, they cannot be used directly for the computation of the aggregate $CC_{SD,U}$ of x_5. Indeed, if we would compute $CC_{SD,U}$ for x_5 by adding up the values of the aggregate $CC_{SD,U}$ of x_1, x_2, and x_4, namely 1, 3, and 2, and updating the result by incrementing it by 1, we would result in $CC_{SD,U} = 7$ for x_5 which is not correct[5]. This is the case, because for instance the tuples of $\mathbf{r}$ with $SD = $ 2010-11-12 and $U = 1$ are counted by the aggregate $CC_{SD,U}$ of all three tuples, x_1, x_2, and x_4. This shows, that the values of x_1, x_2, and x_3 have to be decomposed in some way in order to produce the correct result for $CC_{SD,U}$ which is 5. Finding a general approach for extracting the required information of previously processed tuples of $\mathbf{r}$ from the aggregates in $\mathbf{x}$, however, is not trivial, and depending on the number of constraints and attributes used in a condition the complexity of the computation increases.

[5] Compare with the result relation in table 1(b) on page 2.

$\mathbf{x}$

		SD	U	CC_{SD}	$CC_{SD,U}$
After processing	x_1	2010-11-12	1	4	1
r_1 to r_5	x_2	2010-11-12	2	4	3
	x_3	2010-11-12	3	4	4
	x_4	2010-11-13	1	5	2
...			...		
r_6(2010-11-13, H7, 2, 16)	x_1	2010-11-12	1	4	1
	x_2	2010-11-12	2	4	3
	x_3	2010-11-12	3	4	4
	x_4	2010-11-13	1	5	2
	x_5	**2010-11-13**	**2**	**?**	**?**

Table 4: Example of Late Initialization for Less-Equal

θ–MDA queries with constraint operators $\geq$, $<$, or $>$ feature the same characteristics as queries with the constraint operator $\leq$, regarding the approach required for the computation of the result relation $\mathbf{x}$. Therefore, we will not provide separate examples for those constraint operators.

4.2.3 θ-Conditions with the Constraint Operator $\neq$

In this section we present the application of late initialization to θ–MDA queries with conditions using only the constraint operator $\neq$. We will do this by an example and show that also in this case the computation of aggregates of newly added tuples of $\mathbf{x}$ is not as easy as for θ–MDA with only equality constraints.

In the previous section we learned, that for θ-conditions with constraint operators $\leq$, $\geq$, $<$, and $>$, each tuple of $\mathbf{r}$ can affect more than one θ-result-group. For θ–MDA queries with conditions using only inequality constraints, it is similar. In fact, for each tuple of $\mathbf{r}$ all θ-result-groups besides one are affected. Therefore, also in this case, the information about previously processed tuples is spread among several θ-result-groups in $\mathbf{x}$, and has to be extracted in order to be used for the computation of the aggregates of new tuples of $\mathbf{x}$. An example of θ–MDA with late initialization and inequality constraints follows.

Example 5. In this example we illustrate the strategy for computing $\mathbf{x}$ with late initialization for a query using only the constraint operator $\neq$. The definition of the θ–MDA operator parameters is the following:

$$\begin{aligned}
&\mathbf{r} : LS\\
&\mathbf{b} : \pi_{SD,U} LS\\
&l_1 : (count(SD) \rightarrow NC_{SD})\\
&\theta_1 : \mathbf{r}.SD \neq \mathbf{b}.SD\\
&l_2 : (count(SD) \rightarrow NC_{SD,U})\\
&\theta_2 : \mathbf{r}.SD \neq \mathbf{b}.SD \wedge \mathbf{r}.U \neq \mathbf{b}.U
\end{aligned}$$

It is possible to recognize that also in this example the input relation used is the LS of our running example. The aggregates NC_{SD} and $NC_{SD,U}$ of a specific tuple x of $\mathbf{x}$ count all tuples of the input relation with values different than the values of SD or SD and U of x respectively.

Table 5 illustrates the step of processing tuple r_6 after having previously processed tuples r_1 to r_5. Table $\mathbf{x}$ does not contain any tuple with SD = 2010-11-13 and $U = 2$ and, therefore, a new entry has to be added for r_6. A new tuple, x_5, is added to $\mathbf{x}$ and its aggregates have to be compute. Due to the fact, that the previously processed tuples of $\mathbf{r}$ do not affect the aggregate $ND_{SD,U}$, the value of that aggregate is the total number of previously processed tuples of $\mathbf{r}$. Therefore, this information has to be extracted from the aggregates of x_1 to x_4. However, due to the fact, that the aggregates of x_1 to x_4 contain the number of tuples processed with values different that the θ-result-groups represented by the tuple, the aggregates of x_1 to x_4 have to be decomposed in some way, so that we can compute the total number of processed tuples. Another possibility for computing $NC_{SD,U}$ of x_5 could be to use a counter for each θ-condition with only inequality constraint operators, which tracks the number of tuples processed until each point. However, as soon as a θ-condition contains not only inequality constraint operators the approach based on counters might not work anymore. Therefore, also in the case of inequality θ-conditions, the computation of aggregates for newly added tuples of $\mathbf{x}$ is not trivial.

$\mathbf{x}$

		SD	U	NC_{SD}	$NC_{SD,U}$
After processing r_1 to r_5	x_1	2010-11-12	1	1	4
	x_2	2010-11-12	2	1	3
	x_3	2010-11-12	3	1	4
	x_4	2010-11-13	1	4	4
...				...	
r_6(2010-11-13, H7, 2, 16)	x_1	2010-11-12	1	2	5
	x_2	2010-11-12	2	2	4
	x_3	2010-11-12	3	2	5
	x_4	2010-11-13	1	4	5
	x_5	**2010-11-13**	**2**	**?**	**?**

Table 5: Example of Late Initialization for Not-Equal

4.3 Reducing θ to Equality Constraints

In this section we present a comprehensive approach for the computation of θ–MDA queries based on the idea of late initialization. The main idea of the designed solution is to compute an intermediate result relation $\mathbf{x}^{eq}$ by reducing all θ constraint operators of a θ–MDA query to the constraint operator = and, then, to compute the final result relation $\mathbf{x}$ for the initial constraint operators based on the relation $\mathbf{x}^{eq}$. In this way, $\mathbf{x}^{eq}$ can be easily computed using late initialization as demonstrated in Section 4.2.1 without requiring different and complex computational strategies for calculating the aggregates of newly inserted tuples of $\mathbf{x}$, i.e. without the need of different functions $F(\mathbf{x}, f_{i_j}, \theta_i)$. Regarding the computation of $\mathbf{x}$ based on $\mathbf{x}^{eq}$ details will be provided in the following. In Definition 4 we formally define the idea of θ–MDA with *Late Initialization* which is based on the computation of the intermediate result relation $\mathbf{x}^{eq}$.

Definition 4 (**Late Initialization by Reduction to Equality**). Let $\mathbf{r}(\mathbf{R})$ be a table, θ_i, $1 \leq i \leq m$, be conditions with $attr(\theta_i) \subseteq attr(\mathbf{R})$, let $\Theta = (\theta_1, ...\theta_m)$, let $l_i = (f_{i_1}(A_{i_1}) \rightarrow C_{i_1}, ..., f_{i_{k_i}}(A_{i_{k_i}}) \rightarrow C_{i_{k_i}})$, $1 \leq i \leq m$, be a list of aggregate functions over attributes $A_{i_1}, A_{i_2}, ..., A_{i_{k_i}}$ in $attr(\mathbf{R})$, and let $L = (l_1, ..., l_m)$. Let $\mathbf{b}(\mathbf{B}) = \pi_{B_1,...,B_t}(\sigma(\mathbf{r}))$ be a SP of $\mathbf{r}$ with $B_1, ..., B_t$ in $attr(\mathbf{R})$. Let θ_i^{eq} be θ_i with all constraint operators replaced by $=$, and $\Theta^{eq} = (\theta_1^{eq}, ...\theta_m^{eq})$ be a tuple of the θ-conditions with constraint operators reduced to equality. Then θ–MDA with late initialization is computed as follows:

1. Compute the intermediate result relation

$$\mathbf{x}^{eq} = \mathcal{G}^{\theta}(\mathbf{b}, \mathbf{r}, L, \Theta^{eq})$$

   for the θ-conditions reduced to equality which can be computed by θ–MDA using the late initialization evaluation strategy.

2. Compute the result relation $\mathbf{x}(\mathbf{X})$ where $\mathbf{X} = (\mathbf{B}, D_{1_1}, ..., D_{m_1}, ..., D_{m_{k_m}})$ is the schema of the result table and for each $b \in \mathbf{b}$ there is a tuple in $\mathbf{x}$ with

$$\begin{aligned} & x.\mathbf{B} = b.\mathbf{B} \\ & \mathbf{x}_{i_j}^{eq} = \pi_{attr(\theta_i) \cup attr(C_{i_j})}\{x^{eq} | x^{eq} \in \mathbf{x}^{eq} \wedge \theta_i(x, x^{eq})\} \\ & \mathbf{c}_{i_j} = \{\{x_{i_j}^{eq}.C_{i_j} | x_{i_j}^{eq} \in \mathbf{x}_{i_j}^{eq}\}\} \\ & x.D_{i_j} = \begin{cases} sum(\mathbf{c}_{i_j}) & \text{if } f_{i_j} \text{ is } sum \text{ or } count \\ f_{i_j}(\mathbf{c}_{i_j}) & \text{if } f_{i_j} \text{ is } min \text{ or } max \end{cases} \end{aligned}$$

The evaluation strategy for θ–MDA with late initialization of Definition 4 consists of two main steps. In the first step, the intermediate result relation x^{eq} for the θ-conditions with constraint operators reduced to equality is computed, based on the late initialization strategy presented in Definition 2 on page 14. This is possible because, as we learned previously, that strategy can be easily implemented for θ-conditions containing only equality constraint operators. In the second step, the result relation $\mathbf{x}$ is computed based on $\mathbf{x}^{eq}$. This can be done, because $\mathbf{x}^{eq}$ contains all the information about the tuples of $\mathbf{r}$ and this information can be easily extracted from $\mathbf{x}^{eq}$. More details about the extraction follow.

In Figure 4 the approach of θ–MDA with *Late Initialization* based on the computation of the intermediate result relation $\mathbf{x}^{eq}$ is presented by an example. In the example we use the input relation *LS* of our running example and the aggregates $C_{SD,U}$, NC_{SD}, and $CC_{SD,U}$ of *Query 1.* It is possible to recognize that at the beginning of the computational workflow in Figure 4 (1) we find the input parameters for θ–MDA. Then, based on these parameters in (2) the intermediate result relation $\mathbf{x}^{eq}$ is computed using the θ-conditions with constraint operators reduced to $=$. For instance, θ_1 remains unchanged, θ_2 is reduced to $\mathbf{r}.SD = \mathbf{b}.SD$ and θ_3 to $\mathbf{r}.SD = \mathbf{b}.SD \wedge \mathbf{r}.U = \mathbf{b}.U$. Finally, in (3) the result relation $\mathbf{x}$ is computed based on $\mathbf{x}^{eq}$. More details about this are provided in the following.

(1)

$\mathbf{r} : LS$

	SD	U	...
r_1	2010-11-12	1	...
r_2	2010-11-12	2	...
r_3	2010-11-12	2	...
r_4	2010-11-12	3	...
	...	...	...

$\mathbf{b} : \pi_{SD,U}(LS)$
$l_1 : (count(SD)) \rightarrow C_{SD,U}$
$\theta_1 : \mathbf{r}.SD = \mathbf{b}.SD \wedge \mathbf{r}.U = \mathbf{b}.U$
$l_2 : (count(SD)) \rightarrow NC_{SD}$
$\theta_2 : \mathbf{r}.SD \neq \mathbf{b}.SD$
$l_3 : (count(SD)) \rightarrow CC_{SD,U}$
$\theta_3 : \mathbf{r}.SD \leq \mathbf{b}.SD \wedge \mathbf{r}.U \leq \mathbf{b}.U$

$\downarrow$

(2)

$\mathbf{x}^{eq}$

	SD	U	$C^{eq}_{SD,U}$	NC^{eq}_{SD}	$CC^{eq}_{SD,U}$
x^{eq}_1	2010-11-12	1	1	4	1
x^{eq}_2	2010-11-12	2	2	4	2
x^{eq}_3	2010-11-12	3	1	4	1
x^{eq}_4	2010-11-13	1	1	4	1
x^{eq}_5	2010-11-13	2	1	4	1
x^{eq}_6	2010-11-13	3	2	4	2
x^{eq}_7	2010-11-14	2	1	1	1

$\downarrow$

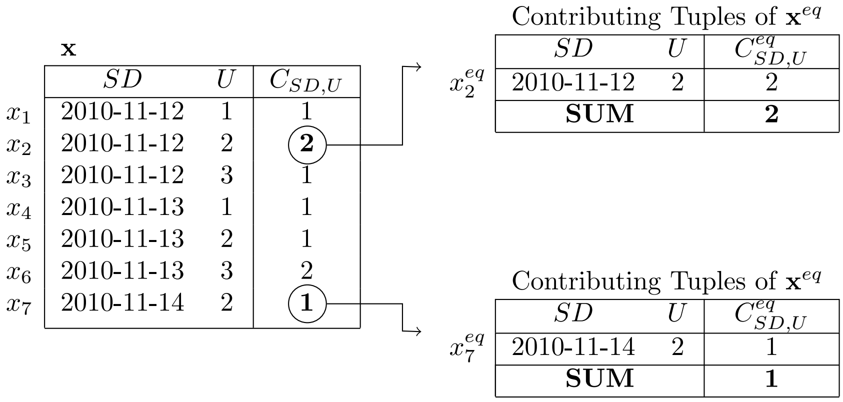
(3)

$\mathbf{x}$

	SD	U	$C_{SD,U}$	NC_{SD}	$CC_{SD,U}$
x_1	2010-11-12	1	1	5	1
x_2	2010-11-12	2	2	5	3
x_3	2010-11-12	3	1	5	4
x_4	2010-11-13	1	1	5	2
x_5	2010-11-13	2	1	5	5
x_6	2010-11-13	3	2	5	8
x_7	2010-11-14	2	1	8	6

Figure 4: θ–MDA with Late Initialization - General Solution (Example)

In the following we illustrate how the aggregate $C_{SD,U}$ of $\mathbf{x}$ can be computed based on $\mathbf{x}^{eq}$. In Figure 5 the result relation $\mathbf{x}$ with only the aggregate $C_{SD,U}$ is depicted together with further illustration about how some of the aggregate values are computed. For instance, the aggregate of tuple x_2 is computed by summing the aggregate values $C^{eq}_{SD,U}$ of all tuples of $\mathbf{x}^{eq}$ with $SD =$ 2010-11-12 and $U = 2$. In this specific case the only tuple of $\mathbf{x}^{eq}$ which contributes to the computation of the aggregate $C_{SD,U}$ of the tuple x_2 is x^{eq}_2. Due to the fact, that in θ_1 only equality constraint operators are used, each tuple of $\mathbf{x}$ is affected by exactly one tuple of $\mathbf{x}^{eq}$. In fact, the aggregate values of the tuples in $\mathbf{x}^{eq}$ are already the correct aggregate values of $\mathbf{x}$.



x

	SD	U	$C_{SD,U}$
x_1	2010-11-12	1	1
x_2	2010-11-12	2	**2**
x_3	2010-11-12	3	1
x_4	2010-11-13	1	1
x_5	2010-11-13	2	1
x_6	2010-11-13	3	2
x_7	2010-11-14	2	**1**

Contributing Tuples of $\mathbf{x}^{eq}$

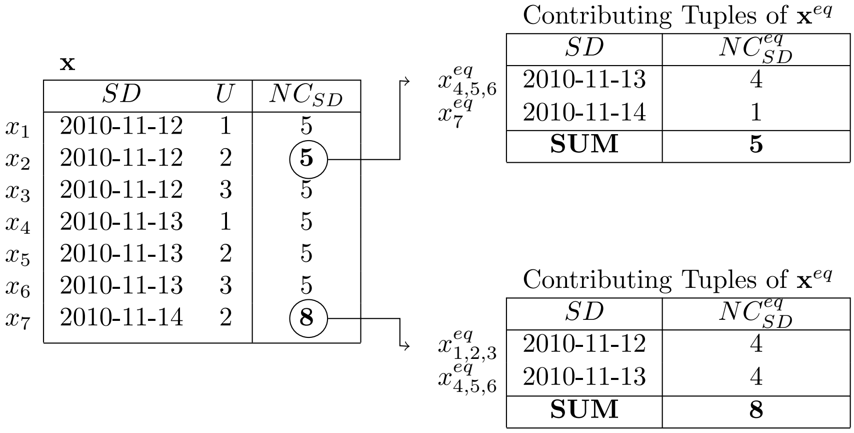
	SD	U	$C^{eq}_{SD,U}$
x^{eq}_2	2010-11-12	2	2
	SUM		**2**

Contributing Tuples of $\mathbf{x}^{eq}$

	SD	U	$C^{eq}_{SD,U}$
x^{eq}_7	2010-11-14	2	1
	SUM		**1**

Figure 5: Transformation of $\mathbf{x}^{eq}$ to $\mathbf{x}$ for Aggregate $C_{SD,U}$

In Figure 6 the result relation $\mathbf{x}$ for only the aggregate NC_{SD} is depicted and also some additional information is provided about how some of the aggregate values of $\mathbf{x}$ are computed. For example, the aggregate of tuple x_2 is computed by summing the aggregate values NC^{eq}_{SD} of all tuples of $\mathbf{x}^{eq}$ with $SD \neq$ 2010-11-12, which are the tuples x^{eq}_4 to x^{eq}_7. However, in this case, the tuples x^{eq}_4 to x^{eq}_6 represent the same θ-result group, and, therefore, they are counted only one time. In Figure 6 we used $x^{eq}_{4,5,6}$ to identify the tuple that represents the θ-result group of the tuples x^{eq}_4 to x^{eq}_6. Due to the fact, that in θ_2 only inequality constraint operators are used, each tuple is affected by all tuples of $\mathbf{x}^{eq}$ besides the one with the specified value for SD.



x

	SD	U	NC_{SD}
x_1	2010-11-12	1	5
x_2	2010-11-12	2	**5**
x_3	2010-11-12	3	5
x_4	2010-11-13	1	5
x_5	2010-11-13	2	5
x_6	2010-11-13	3	5
x_7	2010-11-14	2	**8**

Contributing Tuples of $\mathbf{x}^{eq}$

	SD	NC^{eq}_{SD}
$x^{eq}_{4,5,6}$	2010-11-13	4
x^{eq}_7	2010-11-14	1
	SUM	**5**

Contributing Tuples of $\mathbf{x}^{eq}$

	SD	NC^{eq}_{SD}
$x^{eq}_{1,2,3}$	2010-11-12	4
$x^{eq}_{4,5,6}$	2010-11-13	4
	SUM	**8**

Figure 6: Transformation of $\mathbf{x}^{eq}$ to $\mathbf{x}$ for Aggregate NC_{SD}

Finally, in Figure 7 the result relation $\mathbf{x}$ for only the aggregate $CC_{SD,U}$ is presented together with details about how some of the aggregate values $CC_{SD,U}$ are computed. For example, the aggregate of tuple x_7 is computed by summing the aggregate values $CC^{eq}_{SD,U}$ of all tuples of $\mathbf{x}^{eq}$ with $SD \leq$ 2010-11-12 and $U \leq 2$, which are the tuples x^{eq}_1, x^{eq}_2, x^{eq}_4, x^{eq}_5, and x^{eq}_7. The aggregates of the other tuples of $\mathbf{x}$ are computed in the same way as for tuple x^{eq}_7.

$\mathbf{x}$

	SD	U	$CC_{SD,U}$
x_1	2010-11-12	1	1
x_2	2010-11-12	2	**3**
x_3	2010-11-12	3	4
x_4	2010-11-13	1	2
x_5	2010-11-13	2	5
x_6	2010-11-13	3	8
x_7	2010-11-14	2	**6**

Contributing Tuples of $\mathbf{x}^{eq}$

	SD	U	$CC^{eq}_{SD,U}$
x^{eq}_1	2010-11-12	1	1
x^{eq}_2	2010-11-12	2	2
	SUM		**3**

Contributing Tuples of $\mathbf{x}^{eq}$

	SD	U	$CC^{eq}_{SD,U}$
x^{eq}_1	2010-11-12	1	1
x^{eq}_2	2010-11-12	2	2
x^{eq}_4	2010-11-13	1	1
x^{eq}_5	2010-11-13	2	1
x^{eq}_7	2010-11-14	2	1
	SUM		**6**

Figure 7: Transformation of $\mathbf{x}^{eq}$ to $\mathbf{x}$ for Aggregate $CC_{SD,U}$

The approach of θ–MDA with late initialization presented in this section works in the same way also if other types of aggregate functions are used. The only difference is, that instead of summing the aggregate values of the matching tuples of $\mathbf{x}^{eq}$, for instance, the minimum or the maximum value has to be computed. Further, the θ–MDA approach with late initialization works also for θ-conditions containing combinations of different types of constraint operators, for example, $\leq$ and $\neq$ could be used together in the same θ-condition. Therefore, independently from the constraint operators and the aggregate functions used, the approach presented in this section allows to compute the result relation $\mathbf{x}$ for θ–MDA queries using late initialization. This means, that only one scan of the input relation in necessary for θ–MDA.

4.4 Optimization with Multiple Group Tables

In this section we describe an optimized version of the general solution for evaluating θ–MDA queries with late initialization presented in the previous section. The idea for this modification is to minimize the number of updates of the intermediate result relation for each processed tuple of $\mathbf{r}$ and each θ-condition. Indeed, in the θ–MDA evaluation approach with late initialization presented previously it could be that more than one tuple of x^{eq} has to be updated for a tuple r of $\mathbf{r}$ and a specific θ_i. For instance, in the example depicted in Figure 4 (on page 23) the tuples x_1^{eq} to x_3^{eq} have to be updated when the θ-condition θ_2 is processed for input tuple r_4. This multiple updates can be avoided by using a separate intermediate result relation x_i^{eq} for each condition θ_i, which allows to keep the number of updates performed on the intermediate result relation equal to 1 for each tuple of $\mathbf{r}$ and θ-condition. In the following definition we formalize this concept of the optimized version of θ–MDA with *Late Initialization.*

Definition 5 (**Late Initialization with Separate Tables** x^{eq})**.** Let $\mathbf{r}(\mathbf{R})$ be a table, θ_i, $1 \leq i \leq m$, be conditions with $attr(\theta_i) \subseteq attr(\mathbf{R})$, let $\Theta = (\theta_1, ...\theta_m)$, let $l_i = (f_{i_1}(A_{i_1}) \to C_{i_1}, ..., f_{i_{k_i}}(A_{i_{k_i}}) \to C_{i_{k_i}})$, $1 \leq i \leq m$, be a list of aggregate functions over attributes $A_{i_1}, A_{i_2}, ..., A_{i_{k_i}}$ in $attr(\mathbf{R})$, and let $L = (l_1, ..., l_m)$. Let $\mathbf{b}(\mathbf{B}) = \pi_{B_1,...,B_t}(\sigma(\mathbf{r}))$ be a SP of $\mathbf{r}$ with $B_1, ..., B_t$ in $attr(\mathbf{R})$. Let θ_i^{eq} be θ_i with all constraint operators replaced by $=$, and $\Theta^{eq} = (\theta_1^{eq}, ...\theta_m^{eq})$ be a tuple of the θ-conditions with constraint operators reduced to equality, let $\mathbf{b}_i = \pi_{attr(\theta_i)}(\mathbf{b})$ be the base table for the attributes of θ_i. Then θ–MDA with late initialization is computed as follows:

1. Compute the intermediate result relations

$$(\mathbf{x}_1^{eq}, ..., \mathbf{x}_m^{eq}) = \mathcal{G}_{opt}^{\theta}(\mathbf{b}_i, \mathbf{r}, l_i, \theta_i^{eq})$$

   for the θ-conditions reduced to equality which can be computed by θ–MDA with late initialization.

2. Compute the result relation $\mathbf{x}(\mathbf{X})$ where $\mathbf{X} = (\mathbf{B}, D_{1_1}, ..., D_{m_1}, ..., D_{m_{k_m}})$ is the schema of the result table and for each $b \in \mathbf{b}$ there is a tuple in $\mathbf{x}$ with

$$\begin{array}{l} x.\mathbf{B} = b.\mathbf{B} \\ \mathbf{c}_{i_j} = \{\{x_i^{eq}.C_{i_j} | x_i^{eq} \in \mathbf{x}_i^{eq} \wedge \theta_i(x, x_i^{eq})\}\} \\ x.D_{i_j} = \begin{cases} sum(\mathbf{c}_{i_j}) & \text{if } f_{i_j} \text{ is } sum \text{ or } count \\ f_{i_j}(\mathbf{c}_{i_j}) & \text{if } f_{i_j} \text{ is } min \text{ or } max \end{cases} \end{array}$$

In Definition 5 the approach for computing the result relation $\mathbf{x}$ differs only from the one of the general solution because several intermediate result relations are computed. The computation starts with calculating the intermediate result relations $\mathbf{x}_i^{eq}$ and then based on the different $\mathbf{x}_i^{eq}$'s the final result relation is computed.

In Figure 8 the late initialization strategy of Definition 5 is presented by an example. The input relation used in the example is LS of our running example and the aggregates $C_{SD,U}$, NC_{SD}, and $CC_{SD,U}$ of *Query 1*. It has to be noted, that due to space reasons for the values of the attribute SD we wrote only the day instead of the full date, for example, 12 stays for 2010-11-12. It is possible to recognize that at the beginning of the computational workflow in Figure 8 (1) we find the input parameters for θ–MDA. Then, based on these parameters in (2) the intermediate result relations $\mathbf{x}_i^{eq}$ are computed using the θ-conditions with constraint operators reduced to $=$. For instance, θ_1 remains unchanged, θ_2 is reduced to $\mathbf{r}.SD = \mathbf{b}.SD$ and θ_3 to $\mathbf{r}.SD = \mathbf{b}.SD \wedge \mathbf{r}.U = \mathbf{b}.U$. Finally, in (3) the result relation $\mathbf{x}$ is constructed based on the intermediate result relations $\mathbf{x}_i^{eq}$.

(1)

$\mathbf{r} : LS$

	SD	U	...
r_1	12	1	...
r_2	12	2	...
r_3	12	2	...
r_4	12	3	...
	...	...	...

$\mathbf{b} : \pi_{SD,U}(LS)$

$l_1 : (count(SD)) \rightarrow C_{SD,U}$

$\theta_1 : \mathbf{r}.SD = \mathbf{b}.SD \wedge \mathbf{r}.U = \mathbf{b}.U$

$l_2 : (count(SD)) \rightarrow NC_{SD}$

$\theta_2 : \mathbf{r}.SD \neq \mathbf{b}.SD$

$l_3 : (count(SD)) \rightarrow CC_{SD,U}$

$\theta_3 : \mathbf{r}.SD \leq \mathbf{b}.SD \wedge \mathbf{r}.U \leq \mathbf{b}.U$

$\downarrow$

(2)

$\mathbf{x}_1^{eq}$

	SD	U	$C_{SD,U}^{eq}$
$x_{1_1}^{eq}$	12	1	1
$x_{1_2}^{eq}$	12	2	2
$x_{1_3}^{eq}$	12	3	1
$x_{1_4}^{eq}$	13	1	1
$x_{1_5}^{eq}$	13	2	1
$x_{1_6}^{eq}$	13	3	2
$x_{1_7}^{eq}$	14	2	1

$\mathbf{x}_2^{eq}$

	SD	$NC_{SD,U}^{eq}$
$x_{2_1}^{eq}$	12	4
$x_{2_2}^{eq}$	13	4
$x_{2_3}^{eq}$	14	1

$\mathbf{x}_3^{eq}$

	SD	U	$CC_{SD,U}^{eq}$
$x_{3_1}^{eq}$	12	1	1
$x_{3_2}^{eq}$	12	2	2
$x_{3_3}^{eq}$	12	3	1
$x_{3_4}^{eq}$	13	1	1
$x_{3_5}^{eq}$	13	2	1
$x_{3_6}^{eq}$	13	3	2
$x_{3_7}^{eq}$	14	2	1

$\downarrow$

(3)

$\mathbf{x}$

	SD	U	$C_{SD,U}$	NC_{SD}	$CC_{SD,U}$
x_1	12	1	1	5	1
x_2	12	2	2	5	3
x_3	12	3	1	5	4
x_4	13	1	1	5	2
x_5	13	2	1	5	5
x_6	13	3	2	5	8
x_7	14	2	1	8	6

Figure 8: θ–MDA with Late Initialization - Optimized Solution (Example)

5 Evaluation of θ–MDA

In this section we define an algorithm for the computation of θ–MDA queries based on the approach of θ–MDA with late initialization presented in the previous section, i.e. based on optimized solution presented in Section 4.4. The algorithm is a modification of the `BasicTCMDA` and `IndexedTCMDA` algorithms presented in Section 3.

The remaining of this section is structured as follows. In Section 5.1 we provide details about the developed algorithm. In Section 5.2 we analyse the complexity of the that algorithm and compare it with the complexity of `BasicTCMDA` and `IndexedTCMDA`. Finally, in Section 5.3 the implementation of the algorithms is presented.

5.1 Algorithm

In this section we present an algorithm for computing θ–MDA queries with late initialization based on the approach described in Section 4.4. The algorithm is named `IndexedTCMDA-LI`[6] and is defined on the next page. Step 1 and step 5 of the algorithm are equal as in the `BasicTCMDA` algorithm defined on page 11. These steps transform algebraic aggregates into distributive ones before the computation of the result relation begins, and transform them back, once the result relation is computed. This is necessary for a correct computation of the aggregates. Step 2 to 4, instead, differ from those of the other two algorithms and implement the late initialization strategy described in Section 4.4. These steps are explained in more detail in the following.

Step 2 of the `IndexedTCMDA-LI` algorithm prepares the data structures necessary for storing the result groups and their aggregates. According to Definition 5 different data structures are required for the computation of the result relation, namely m intermediate result relations $\mathbf{x}_i^{eq}$ and the final result relation $\mathbf{x}$. Also in `IndexedTCMDA-LI` the same data structures are used for the computation of θ–MDA with late initialization. Further, for each of these data structures an index is created in order to improve the retrieval time of single tuples during the process of aggregate computation in step 3. Finally, in step 2 also the equality θ-conditions, θ_i^{eq}, are created which are used in order to compute the intermediate result relations $\mathbf{x}_i^{eq}$ in step 3 of the algorithm.

[6] Note that `LI` in the algorithm name stays for late initialization.

Algorithm 3: IndexedTCMDA-LI($\pi_{A_1,...,A_s}(\sigma(\mathbf{r})), \mathbf{r}, (l_1, ..., l_m), (\theta_1, ..., \theta_m)$)

```
// Step 1: Replace algebraic aggregates
(Same as step 1 in BasicTCMDA)

// Step 2: Initialize data structures and parameters
// Initialize empty (intermediate) result relation(s)
Let $\mathbf{B} = (A_1, ..., A_s)$, $\mathbf{x} = \mathbf{B} \times (C_{1_1}, ..., C_{m_{k_m}})$
Initialize an index for $\mathbf{x}$ over the attributes $attr(\mathbf{B})$;
foreach $\theta_i \in \{\theta_1, ..., \theta_m\}$ do
    Let $\mathbf{A}_i^\theta = (A_{i_1}^\theta, ...A_{i_{o_i}}^\theta)$ where $\{A_{i_1}^\theta, ..., A_{i_{o_i}}^\theta\} = attr(\theta_i) \subseteq \{A_1, ..., A_s\}$;
    Let $\mathbf{x}_i^{eq} = \mathbf{A}_i^\theta \times (C_{i_1}, ..., C_{i_{k_i}})$;
    Initialize an index for $\mathbf{x}_i^{eq}$ over the attributes $attr(\mathbf{A}_i^\theta)$;
end
// Initialize the equality $\theta$-conditions
for $i = 1$ to $m$ do
    $\theta_i^{eq} = \theta_i$ with all constraint operators converted to equality;
end

// Step 3: Compute the aggregates
// Note: $\upsilon_j$'s are the initial aggregate values (0 for count
   and sum, NULL for min and max).
foreach tuple $r \in R$ do
    if $!\exists x \in \mathbf{x}(x.\mathbf{B} = r.\mathbf{B})$ then
        $\mathbf{x} = \mathbf{x} \cup \{r.\mathbf{B} \times (\upsilon_{1_1}, ..., \upsilon_{m_{k_m}})\}$;
    end
    foreach $\theta_i^{eq} \in \{\theta_1^{eq}, ..., \theta_m^{eq}\}$ do
        Fetch tuple $x_i^{eq}(x_i^{eq} \in \mathbf{x}_i^{eq} \wedge \theta_i^{eq}(x_i^{eq}, r) = true)$ using index;
        if $x_i^{eq} = NULL$ then
            $x_i^{eq} = (r.\mathbf{A}_i, \upsilon_{i_1}, ..., \upsilon_{i_{k_i}})$;
            $\mathbf{x}_i^{eq} = \mathbf{x}_i^{eq} \cup \{x_i^{eq}\}$;
        end
        Update aggregates of $x_i^{eq}$ based on $r$;
    end
end

// Step 4: Build the result table x
foreach $x \in \mathbf{x}$ do
    foreach $\theta_i \in \{\theta_1, ..., \theta_m\}$ do
        $\mathbf{x}_{match}^{eq} = \{x_i^{eq} \in \mathbf{x}_i^{eq} | \theta_i(x, x_i^{eq})\}$;
        Compute aggregates of $x$ based on $\mathbf{x}_{match}^{eq}$;
    end
end

// Step 5: Apply the super-functions
(Same as in step 4 in BasicTCMDA)

return $\mathbf{x}$;
```

In step 3 of algorithm `IndexedTCMDA-LI` the aggregates of the intermediate result relations $\mathbf{x}_i^{eq}$ are computed. For each tuple $r \in \mathbf{r}$ the data structures $\mathbf{x}$, $\mathbf{x}_1^{eq}$, ..., $\mathbf{x}_m^{eq}$ are extended by a new tuple, if the result group to which r belongs is not present in these data structures. The aggregates of the newly added tuples are initialized to their initial values v_j, which is 0 for the aggregates *count* and *sum* and *NULL* for the aggregates *min* and *max*. Further, in this step, for each tuple $r \in \mathbf{r}$ and for each condition θ_i^{eq} the aggregates in the data structures $\mathbf{x}_1^{eq}$, ..., $\mathbf{x}_m^{eq}$ are updated. It has to be noted, that for each r and each θ_i^{eq} exactly one tuple of $\mathbf{x}_i^{eq}$ is updated. At the end of step 3, the computation of the entries of the intermediate result relations $\mathbf{x}_i^{eq}$ is completed. In Table 6 we illustrate step 3 of the `IndexedTCMDA-LI` algorithm for the computation of the aggregates $C_{SD,U}$, NC_{SD}, and $CC_{SD,U}$ of *Query 1.* It is possible to recognize, that we start with three empty data structures $\mathbf{x}_1^{eq}$ to $\mathbf{x}_3^{eq}$, one for each θ-condition of the example. Due to space reasons, for the attribute SD we use an abbreviated version of the value by writing only the day instead of the whole date, for example, 12 instead of 2010-11-12. At each step, in Table 6, a tuple of $\mathbf{r}$ is processed and the data structures are extended and/or updated accordingly. It is possible to recognize, that at the end of the computation the data structures $\mathbf{x}_i^{eq}$ contain the same entries as the ones of Figure 8 step (2) (on page 27).

$\mathbf{x}_1^{eq}$

SD	U	$C^{eq}_{SD,U}$

$\mathbf{x}_2^{eq}$

SD	NC^{eq}_{SD}

$\mathbf{x}_3^{eq}$

SD	U	$CC^{eq}_{SD,U}$

r_1

SD	U	$C^{eq}_{SD,U}$
12	1	1

SD	NC^{eq}_{SD}
12	1

SD	U	$CC^{eq}_{SD,U}$
12	1	1

r_2

SD	U	$C^{eq}_{SD,U}$
12	1	1
12	2	1

SD	NC^{eq}_{SD}
12	2

SD	U	$CC^{eq}_{SD,U}$
12	1	1
12	2	1

... ...

r_9

SD	U	$C^{eq}_{SD,U}$
12	1	1
12	2	2
12	3	1
13	1	1
13	2	1
13	3	2
14	2	1

SD	NC^{eq}_{SD}
12	4
13	4
14	1

SD	U	$CC^{eq}_{SD,U}$
12	1	1
12	2	2
12	3	1
13	1	1
13	2	1
13	3	2
14	2	1

Table 6: Step 3 of Algorithm `IndexedTCMDA-LI`

Next, in step 4 of the algorithm `IndexedTCMDA-LI` the result relation $\mathbf{x}$ is computed based on $\mathbf{x}_1^{eq}$, ... $\mathbf{x}_m^{eq}$. This is done, by retrieving all aggregates of the intermediate result relations x_i^{eq} for any tuple of $\mathbf{x}$ and each θ-condition and summing them for *count* and *sum* aggregates or by calculating the minimum or maximum value for the aggregates *min* and *max* respectively. At the end of this step, the result relation $\mathbf{x}$ is completely computed and ready for the last step in which distributive sub-aggregates are transformed back to algebraic aggregates.

5.2 Complexity Analysis

In this section we analyse and compare the time complexity of the algorithms `BasicTCMDA`, `IndexedTCMDA`, and `IndexedTCMDA-LI`. The setting for which we analyse the complexity is the one with $\mathbf{b} = \pi_{B_1,...,B_t}(\sigma(\mathbf{r}))$ and $\{B_1, ..., B_t\} \subseteq attr(\mathbf{r})$. Before proceeding with more details about time complexity for the θ–MDA algorithms, we introduce the following notions:

$\lvert\mathbf{r}\rvert$	The size of the input table.
$\lvert\mathbf{b}\rvert$	The size of the base table.
c	Some constant number greater equal 1.
$\lvert\Theta\rvert$	The number of θ-conditions.
C_θ	The cost for checking a θ-condition.
$C_{grp_{check}}$	The cost for checking whether a result group exists.
$C_{grp_{add}}$	The cost for adding a new result group.
C_{idx}	The cost for consulting an index.
C_{agg}	The cost for updating an aggregate.

Based on these notions, in the following we present the time complexity of the three θ–MDA algorithms taking into account the influence of the constraint operators used in a query. It has to be noticed, that the size of the input relation $|\mathbf{r}|$ in typical θ–MDA scenarios is much larger than the size of the base table $|\mathbf{b}|$. In fact, the former can be several thousands, ten thousands, or even more times larger then the latter. Moreover, the number of θ-conditions in a query, $|\Theta|$, is usually quite small with values less than ten. The constant c reflects the number of tuples which have to be updated on average for each tuple of $\mathbf{r}$ and θ-condition and is used in connection with equality and inequality constraint operators. We expect c to have small values on average. The value depends from the θ-conditions of a query and from the data of the input relation. For instance, in a scenario with only one θ-condition with only equality constraint operators c is 1, whereas, in a scenario with two θ-conditions, both with only equality constraint operators, the first over two attributes and the second over one of these attributes, it might be that for the second θ-condition more than one tuple of the result relation has to be updated for a processed tuple of $\mathbf{r}$, thus, resulting in $c > 1$. Such a situation is illustrated in the example of Table 3 on page 18. The remaining notions presented in the list above represent constants and reflect the cost of, for instance, checking a θ-condition or updating an aggregate. Based on the introduced notion, we can state that the parameter $|\mathbf{r}|$ is the one influencing the performance of the evaluation of θ–MDA queries most.

5.2.1 Complexity for Constraint Operator =

In Table 7 the time complexity or the θ–MDA algorithms, for queries with only equality constraint operators, is presented.

The time complexity formula of `BasicTCMDA` consists of two main parts, one part expressing the complexity of computing $\mathbf{b}$ and one part expressing the complexity of computing $\mathbf{x}$. From the formula it is possible to recognize that the cost of computing $\mathbf{b}$ consists of the cost of checking whether a result group already exists in $\mathbf{b}$, which is done for each tuple of $\mathbf{r}$, and the cost of adding a result group to $\mathbf{b}$, which occurs $|\mathbf{b}|$ times. For the computation of $\mathbf{x}$, instead, for each tuple of $\mathbf{r}$ and each θ-condition all tuples of the result relation are traversed and the θ-condition checked. The cost of this computation is expressed by the part of the formula $|\mathbf{r}| \cdot |\Theta| \cdot |\mathbf{b}| \cdot C_\theta$. Further, the computation of $\mathbf{x}$ implies the update of the aggregates, which is performed only for those tuples of the result relation for which θ-conditions match. In the case of θ-conditions with only equality constraint operators on average about c tuples match, thus, resulting in the complexity formula $|\mathbf{r}| \cdot |\Theta| \cdot c \cdot C_{agg}$. The overall complexity of `BasicTCMDA` in O notation is $O(|\mathbf{r}| \cdot |\mathbf{b}|)$.

The complexity formula of `IndexedTCMDA` also consists of two parts, one concerning the computation of the base table $\mathbf{b}$ and one concerning the computation of the result relation $\mathbf{x}$. The part of the formula concerned with the computation of $\mathbf{b}$ is the same as for `BasicTCMDA`. The formula expressing the cost of computing $\mathbf{x}$, instead, is different from that of `BasicTCMDA`. Indeed, `IndexedTCMDA` uses an index for the retrieval of the tuples of the result relation which have to be updated for each processed tuple of $\mathbf{r}$ and each θ-condition instead of traversing all tuples of $\mathbf{x}$. The complexity formula for this fragment of computation is $|\mathbf{r}| \cdot |\Theta| \cdot C_{idx}$. Regarding the update of the aggregates, `IndexedTCMDA` has the same complexity as `BasicTCMDA`, namely $|\mathbf{r}| \cdot |\Theta| \cdot c \cdot C_{agg}$. The overall cost of computing θ–MDA queries with only equality constraint operators for `IndexedTCMDA` is lower than for `BasicTCMDA`, namely $O(|\mathbf{r}|)$.

Algorithm	*Complexity Formula*		*O Notation*
`BasicTCMDA`	(Compute $\mathbf{b}$)	$\lvert\mathbf{r}\rvert \cdot C_{grp_{check}}+$ $\lvert\mathbf{b}\rvert \cdot C_{grp_{add}}+$	$O(\lvert\mathbf{r}\rvert \cdot \lvert\mathbf{b}\rvert)$
	(Compute $\mathbf{x}$)	$\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot \lvert\mathbf{b}\rvert \cdot C_\theta+$ $\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot c \cdot C_{agg}$	
`IndexedTCMDA`	(Compute $\mathbf{b}$)	$\lvert\mathbf{r}\rvert \cdot C_{grp_{check}}+$ $\lvert\mathbf{b}\rvert \cdot C_{grp_{add}}+$	$O(\lvert\mathbf{r}\rvert)$
	(Compute $\mathbf{x}$)	$\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot C_{idx}+$ $\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot c \cdot C_{agg}$	
`IndexedTCMDA-LI`	(Compute $\mathbf{x}_i^{eq}$)	$\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot C_{idx}+$ $\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot 1 \cdot C_{agg}$ $\lvert\mathbf{b}\rvert \cdot C_{grp_{add}}$	$O(\lvert\mathbf{r}\rvert)$
	(Compute $\mathbf{x}$)	$\lvert\mathbf{b}\rvert \cdot \lvert\Theta\rvert \cdot C_{idx}+$ $\lvert\mathbf{b}\rvert \cdot \lvert\Theta\rvert \cdot 1 \cdot C_{agg}$	

Table 7: Time Complexity for only Constraint Operators =

Finally, in Table 7 the time complexity of `IndexedTCMDA-LI` is presented. Also in this case the formula consists of two parts, one part reflecting the cost of computing the intermediate result relations $\mathbf{x}_i^{eq}$ and one part reflecting the cost of computing the result relation $\mathbf{x}$ from the relations $\mathbf{x}_i^{eq}$. The computation of the intermediate result relations $\mathbf{x}_i^{eq}$ in `IndexedTCMDA-LI` is the costly part of the computation and comprises the iteration through all tuples of the input relation $\mathbf{r}$, and for each of them the addition of a result group if it is missing in the result relation and the update of the aggregates. In order to retrieve the aggregates which have to be updated `IndexedTCMDA-LI` uses an index. The cost of this computational effort can be expressed by $|\mathbf{r}| \cdot |\Theta| \cdot C_{idx} + |\mathbf{r}| \cdot |\Theta| \cdot 1 \cdot C_{idx} + |\mathbf{b}| \cdot C_{grp_{add}}$. From the formula it is possible to recognize, that for each tuple of $\mathbf{r}$ and each θ-condition, exactly 1 aggregate is updated and not c. This is the case, because `IndexedTCMDA-LI` uses a separate intermediate result relations (see Section 4.4 for more details about this). The computation of $\mathbf{x}$ based on the result relations $\mathbf{x}_i^{eq}$ has a lower impact on the complexity of the algorithm and can be expressed by formula $|\mathbf{b}| \cdot |\Theta| \cdot C_{idx} + |\mathbf{b}| \cdot |\Theta| \cdot 1 \cdot C_{agg}$. The overall complexity of `IndexedTCMDA-LI` for θ-conditions with only equality constraint operators in O notation is $O(|\mathbf{r}|)$, being lower than the complexity of `BasicTCMDA` and equal to the complexity of `IndexedTCMDA`.

5.2.2 Complexity for Constraint Operator $\leq$, $\geq$, $<$, $>$

In Table 8 the time complexity of the θ–MDA algorithms for queries with constraint operators $\leq$, $\geq$, $<$, and $>$ is presented.

The complexity formula of `BasicTCMDA` for θ–MDA queries with constraint operators $\leq$, $\geq$, $<$, and $>$ differs from the formula for queries with only equality constraint operators in the part expressing the cost of computing $\mathbf{x}$. Indeed, when the constraint operators $\leq$, $\geq$, $<$, and $>$ are used, for the computation of $\mathbf{x}$ on average $\frac{|\mathbf{b}|}{2}$ aggregates have to be updated for any tuple of $\mathbf{r}$ and θ-condition. The resulting time complexity formula for this part of the computation is $|\mathbf{r}| \cdot |\Theta| \cdot |\mathbf{b}| \cdot C_\theta + |\mathbf{r}| \cdot |\Theta| \cdot \frac{|\mathbf{b}|}{2} \cdot C_{agg}$. Although, more aggregate updates are performed for θ–MDA queries with constraint operators $\leq$, $\geq$, $<$, and $<$ with respect to queries with only equality constraints, the overall complexity of `BasicTCMDA` in O notation is the same as for queries with only equality constraint operators, namely $O(|\mathbf{r}| \cdot |\mathbf{b}|)$.

Also for `IndexedTCMDA` the part of the formula related to the complexity of computing $\mathbf{x}$ differs for θ–MDA queries with constraints $\leq$, $\geq$, $<$, and $>$ compared to the formula for queries with only $=$. Same as in `BasicTCMDA` the average number of aggregate updates per $r \in \mathbf{r}$ and θ-condition is $\frac{|\mathbf{b}|}{2}$. However, in the case of `IndexedTCMDA` the complexity in O notation is higher if the constraint operators $\leq$, $\geq$, $<$, and $>$, namely $O(|\mathbf{r}| \cdot |\mathbf{b}|)$, as compared to the complexity for only equality constraint operators which is $O(|\mathbf{r}|)$.

Similarly as for `BasicTCMDA` and `IndexedTCMDA` also for `IndexedTCMDA-LI` the complexity formula of θ–MDA queries with constraint operators $\leq$, $\geq$, $<$, and $>$ is different from the complexity formula if only $=$ is used. Indeed, during the computation of $\mathbf{x}$ based on the result relations $\mathbf{x}_i^{eq}$ for each result group and θ-condition on average $\frac{|\mathbf{b}|}{2}$ aggregate updates are necessary. The complexity formula which reflects this phenomenon is $|\mathbf{b}| \cdot |\Theta| \cdot C_{idx} + |\mathbf{b}| \cdot |\Theta| \cdot \frac{|\mathbf{b}|}{2} \cdot C_{agg}$. However, the impact of this part of the formula is not very big with respect to the overall complexity of the algorithm, which is $O(|\mathbf{r}|)$ in O notation.

Algorithm	*Complexity Formula*		*O Notation*
`BasicTCMDA`	(Compute **b**)	$\lvert\mathbf{r}\rvert \cdot C_{grp_{check}}+$ $\lvert\mathbf{b}\rvert \cdot C_{grp_{add}}+$	$O(\lvert\mathbf{r}\rvert \cdot \lvert\mathbf{b}\rvert)$
	(Compute **x**)	$\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot \lvert\mathbf{b}\rvert \cdot C_{\theta}+$ $\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot \frac{\lvert\mathbf{b}\rvert}{2} \cdot C_{agg}$	
`IndexedTCMDA`	(Compute **b**)	$\lvert\mathbf{r}\rvert \cdot C_{grp_{check}}+$ $\lvert\mathbf{b}\rvert \cdot C_{grp_{add}}+$	$O(\lvert\mathbf{r}\rvert \cdot \lvert\mathbf{b}\rvert)$
	(Compute **x**)	$\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot C_{idx}+$ $\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot \frac{\lvert\mathbf{b}\rvert}{2} \cdot C_{agg}$	
`IndexedTCMDA-LI`	(Compute $\mathbf{x}_i^{eq}$)	$\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot C_{idx}+$ $\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot 1 \cdot C_{agg}$ $\lvert\mathbf{b}\rvert \cdot C_{grp_{add}}$	$O(\lvert\mathbf{r}\rvert)$
	(Compute **x**)	$\lvert\mathbf{b}\rvert \cdot \lvert\Theta\rvert \cdot C_{idx}+$ $\lvert\mathbf{b}\rvert \cdot \lvert\Theta\rvert \cdot \frac{\lvert\mathbf{b}\rvert}{2} \cdot C_{agg}$	

Table 8: Time Complexity for Constraint Operators $\leq$, $\geq$, $<$, $>$

5.2.3 Complexity for Constraint Operator $\neq$

In Table 8 the time complexity of the θ–MDA algorithms for queries with only inequality constraint operators is presented.

The complexity formula of `BasicTCMDA` differes from its complexity formulas presented in the previous sections in the part reflecting the cost of computing $\mathbf{x}$. Indeed, for θ–MDA queries with only inequality constraint operators, for each tuple of $\mathbf{r}$ and θ-condition on average $|\mathbf{b}| - c$ aggregates have to be updated. The formula reflecting this part of the computation is $|\mathbf{r}| \cdot |\Theta| \cdot |\mathbf{b}| \cdot C_{\theta} + |\mathbf{r}| \cdot |\Theta| \cdot (|\mathbf{b}| - c) \cdot C_{agg}$. The overall time complexity expressed in O notation, however, is not affected by this change in the formula an is $O(|\mathbf{r}| \cdot |\mathbf{b}|)$.

Similarly as for `BasicTCMDA`, also the complexity formula of `IndexedTCMDA` changes with respect to its formulas presented in the previous sections. The number of aggregate updates performed by `IndexedTCMDA` during the computation of $\mathbf{x}$ is the same as for `BasicTCMDA`. The overall time complexity of the algorithm for `IndexedTCMDA` for θ–MDA queries with only inequality constraint operators is the same as if the constraint operators $\leq$, $\geq$, $<$, and $<$ are used, namely $O(|\mathbf{r}| \cdot |\mathbf{b}|)$.

Finally, the time complexity formula of `IndexedTCMDA-LI` also changes with respect to the formulas for θ–MDA queries with constraint operators $=$, $\leq$, $\geq$, $<$, and $>$. Indeed, during the computation of $\mathbf{x}$ based on the result relations $\mathbf{x}_i^{eq}$ for each result group and θ-condition on average $|\mathbf{b}| - 1$ aggregate updates are necessary, resulting in the complexity formula $|\mathbf{b}| \cdot |\Theta| \cdot C_{idx} + |\mathbf{b}| \cdot |\Theta| \cdot (|\mathbf{b}| - 1) \cdot C_{agg}$. The overall time complexity in O notation remains $O(|\mathbf{r}|)$

Algorithm	*Complexity Formula*		*O Notation*
BasicTCMDA	(Compute $\mathbf{b}$)	$\lvert\mathbf{r}\rvert \cdot C_{grp_{check}} +$ $\lvert\mathbf{b}\rvert \cdot C_{grp_{add}} +$	$O(\lvert\mathbf{r}\rvert \cdot \lvert\mathbf{b}\rvert)$
	(Compute $\mathbf{x}$)	$\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot \lvert\mathbf{b}\rvert \cdot C_\theta +$ $\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot (\lvert\mathbf{b}\rvert - c) \cdot C_{agg}$	
IndexedTCMDA	(Compute $\mathbf{b}$)	$\lvert\mathbf{r}\rvert \cdot C_{grp_{check}} +$ $\lvert\mathbf{b}\rvert \cdot C_{grp_{add}} +$	$O(\lvert\mathbf{r}\rvert \cdot \mathbf{b})$
	(Compute $\mathbf{x}$)	$\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot C_{idx} +$ $\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot (\lvert\mathbf{b}\rvert - c) \cdot C_{agg}$	
IndexedTCMDA-LI	(Compute $\mathbf{x}_i^{eq}$)	$\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot C_{idx} +$ $\lvert\mathbf{r}\rvert \cdot \lvert\Theta\rvert \cdot 1 \cdot C_{agg}$ $\lvert\mathbf{b}\rvert \cdot C_{grp_{add}}$	$O(\lvert\mathbf{r}\rvert)$
	(Compute $\mathbf{x}$)	$\lvert\mathbf{b}\rvert \cdot \lvert\Theta\rvert \cdot C_{idx} +$ $\lvert\mathbf{b}\rvert \cdot \lvert\Theta\rvert \cdot (\lvert\mathbf{b}\rvert - 1) \cdot C_{agg}$	

Table 9: Time Complexity for only Constraint Operators $\neq$

5.3 Implementation

In this Section we present some details about the implementation of the θ–MDA algorithms. We developed three programs, one for each algorithm, using the Java[7] programming language and the Oracle[8] database. The programs are designed as command line tools.

Figure 9 provides an overview of the program workflow of the `BasicTCMDA` algorithm implementation. The parameters $\mathbf{r}$, $\mathbf{b}$, L, and Θ of the program are those of θ–MDA with late initialization as defined in Definition 4 on page 22. The program expects parameter $\mathbf{r}$ to be encoded as an SQL query, $\mathbf{b}$ as a comma-separated list of attribute names of $\mathbf{r}$, L as a comma-separated list of aggregates over attributes of $\mathbf{r}$ and Θ as a comma-separated list of constraints. In the following we show how *Query 1* (see its formal definition on page 9) can be encoded into the parameters of the program:

```
r  :  SELECT SD, U FROM LS
b  :  SD, U
L  :  count(*), count(*), count(*), count(*)
Θ  :  r.SD = b.SD & r = b.U,
      rD <> b.SD,
      rD ≤ b.SD,
      rD ≤ b.SD & r ≤ b.U
```

Once the command line tool is started with parameters $\mathbf{r}$, $\mathbf{b}$, L, and Θ, it first loads the input relation $\mathbf{r}$ from the database management system into main memory and computes the base table $\mathbf{b}$ (step 1 in Figure 9) and then (in step

[7] http://www.java.com

[8] http://www.oracle.com/us/products/database/index.html

2) it loads the input table **r** again in order to compute the result table **x** using the strategy of the `BasicTCMDA` algorithm described on page 11.

The output of the program is provided on the command line using a comma-separated value format. For *Query 1* the output looks like as follows:

```
SD,U,count(*),count(*),count(*),count(*)
12,1,1,5,4,1
12,2,2,5,4,3
12,3,1,5,4,4
13,1,1,5,8,2
13,2,1,5,8,5
13,3,2,5,8,8
14,2,1,8,9,6
```

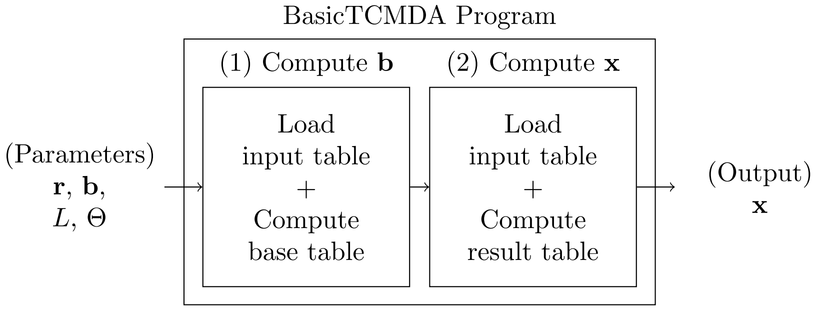

BasicTCMDA Program
(1) Compute **b**
(2) Compute **x**
Load input table + Compute base table
Load input table + Compute result table
(Parameters) **r**, **b**, L, Θ
(Output) **x**

Figure 9: `BasicTCMDA` Program Workflow

The main program workflow of the `IndexedTCMDA` algorithm implementation is depicted in Figure 10. The input parameters, step 1, and the output of the program are the same as for the implementation of `BasicTCMDA` presented above. Differences can be found in step 2 which implements the computation strategy of the `IndexedTCMDA` algorithm presented on page 12. The main difference in the strategy of this algorithm, compared to the `BasicTCMDA` algorithm, is that it uses an index in order to retrieve the tuples of the result relation to be modified (see Section 3 for more details).

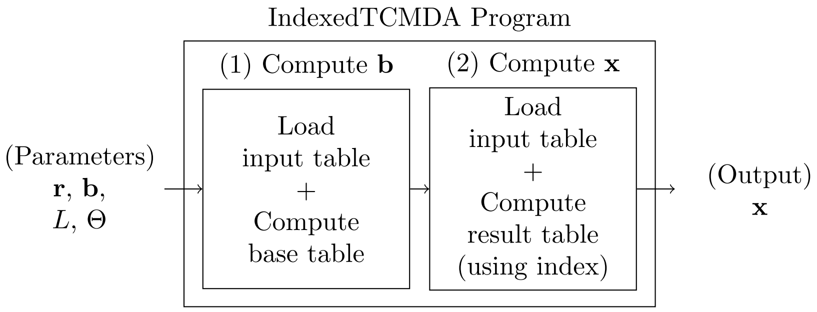


Figure 10: `IndexedTCMDA` Program Workflow

In our implementation of `IndexedTCMDA` we create a separate index for each $\theta_i \in \Theta$. Depending on the constraint operators used in θ_i different index strategies are used:

- For θ_i's containing only $=$ and $\neq$ constraint operators a hash index structure is used.
- For θ_i's containing not only the constraint operators $=$ and $\neq$, but also one or more of the constraint operators $\leq$, $\geq$, $<$, and $>$ a hierarchical index structure is used.

In the case of θ_i's containing only $=$ and $\neq$ constraint operators, in which a hash index is used, the key k_j of the index for result tuple $x_j \in X$ is generated using the attribute values of the result tuple x_j of all attributes involved in the θ_i constraint. The value p_j of the index associated to a key k_j is a pointer to result tuple x_j. Figure 11 depicts the index structure for $\theta_1 : \mathbf{r}.SD = \mathbf{b}.SD$ & $\mathbf{r}.U = \mathbf{b}.U$ of Example 1. The keys of the index consist of the values for the attributes SD and U, whereas, the values p_1 to p_7 represent the pointers to the result relation tuples x_1 to x_7 respectively.

Key	Value
(12,1)	p_1
(12,2)	p_2
(12,3)	p_3
(13,1)	p_4
(13,2)	p_5
(13,3)	p_6
(14,2)	p_7

Figure 11: Hash Index Structure for θ_1 of in Example 1

In the case of θ_i's containing not only $=$ and $\neq$ but also at least one of the constraint operators $\leq$, $\geq$, $<$, $>$, in which a hierarchical index is used, the construction of that index structure consists of several steps:

1. The constraints within one θ_i are sorted according to the selectivity[9] of their operator, i.e. $=$ is the most selective operator, followed by $\leq$, $\geq$, $<$, $>$, and, finally, $\neq$ is the less selective one.
2. One hash index is created for the $=$ constraints, one hash index for the $\neq$ constraints, and one tree index for each other constraint having an operator different than $=$ and $\neq$.
3. The created indexes are organized hierarchically, starting with the hash index for $=$ constraints as the root of the hierarchy, pointing to the index structures for $\leq, \geq, <, >$ constraints and these pointing to the hash index for $\neq$ constraints.

Figure 12 shows the hierarchical index for $\theta_3 : \mathbf{r}.SD \leq \mathbf{b}.SD$ & $\mathbf{r} \leq \mathbf{b}.U$ of Example 1. It is possible to recognize that for the first constraint $\mathbf{r}.SD \leq \mathbf{b}.SD$ the index on level 1 of the index hierarchy is created. For each element of this level 1 index a separate index is created storing the values of the second constraint $\mathbf{r}.U \leq \mathbf{b}.U$.

[9] With selectivity we mean the number of matches using a specific constraint operator, e.g. in general with $=$ the number of matching tuples is smaller than with $\leq$.

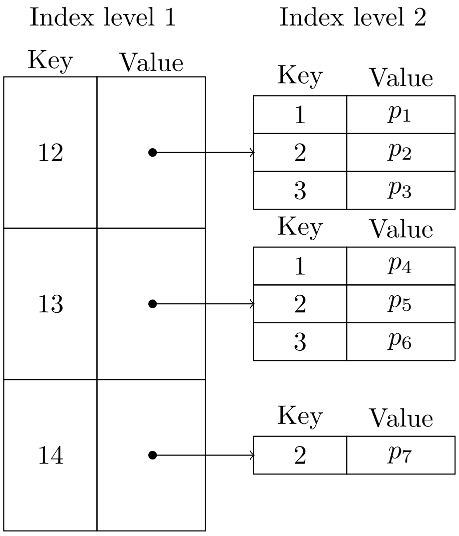


Figure 12: Hierarchical Index Structure for θ_4 of Example 1

Finally, Figure 13 depicts the program workflow of the `IndexedTCMDA-LI` algorithm. In this case in step 1 loads the input table and scans it in order to compute the intermediate result relations $\mathbf{x}_i^{eq}$. In step 2 the program computes the result relation $\mathbf{x}$ based on the intermediate result relation so that in this step there is no need to load and scan the input table again. Also in `IndexedTCMDA-LI` an index is used in order to retrieve the tuples of $\mathbf{x}_i^{eq}$ and $\mathbf{x}$ to be modified. The index strategy used is the same as the one for `IndexedTCMDA` presented before.

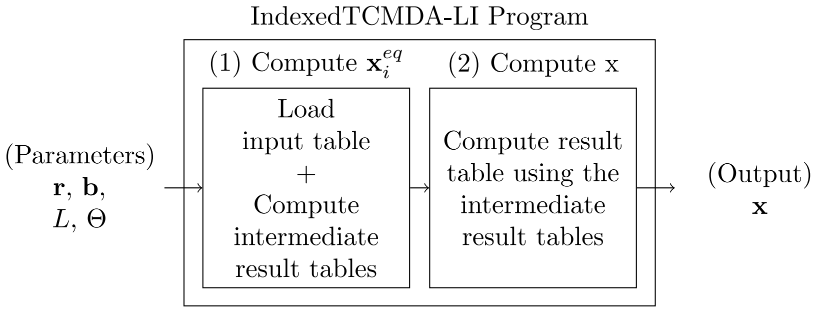

IndexedTCMDA-LI Program
(1) Compute $\mathbf{x}_i^{eq}$
(2) Compute x
Load input table + Compute intermediate result tables
Compute result table using the intermediate result tables
(Parameters) $\mathbf{r}$, $\mathbf{b}$, L, Θ
(Output) $\mathbf{x}$

Figure 13: `IndexedTCMDA-LI` Program Workflow

6 Experiments

In this section we present the experiments that we run in order to evaluate the performance of the `IndexedTCMDA-LI` algorithm and to compare it with the performance of `BasicTCMDA` and `IndexedTCMDA`. For this purpose we tested these three algorithms in different settings on a large amount of data, based on their implementation presented in Section 5.3.

In the rest of this section we present details about the experiments and their results. Section 6.1 provides information about the set-up of the experiments, and, in Section 6.2 the results of the experiments are presented.

6.1 Setup and Data

The machine used for running the experiments has two AMD Opteron processors, one with 1.8GHz and the other with 2.6GHz, and 16GB of main memory. The operating system installed on the machine is Ubuntu 10.04 and the database is Oracle 11g. For the tests we executed the programs starting the Java Virtual Machine with 3GB of main memory assigned to it.

Due to the fact, that the performance of the θ–MDA algorithms is affected by different variables, we tested it by scaling one variable at a time while keeping the others fixed. The factors influencing the performance of the algorithms which are considered in our experiments are:

- $|\mathbf{r}|$: The number of tuples in the input relation $\mathbf{r}$.
- $|\mathbf{b}|$: The number of tuples in the base relation $\mathbf{b}$.
- $|\theta|$: The number of conditions within one single θ-condition.
- $|\Theta|$: The number of θ-conditions in the query.
- $\oplus$: The constraint operators used in the θ constraints (e.g. $=$, $\leq$).

It has to be noted, that in the parameters considered in the experiments an additional parameter to those used in the time complexity formulas in Section 5.2 is used. Indeed, the parameter $|\theta|$ used here, is not considered in this form in the complexity analysis. This is the case, because in that analysis we considered the time for checking a θ-constraint as constant, independently from the number of constraints used in the condition. However, in this section we present experiments in which we scale the number of constraints used in θ-conditions in order to measure its effect on the performance of the algorithms.

For the execution of the tests we used the *Orders* table of the TPC-H benchmark framework[10] which has different attributes such as *o_orderdate*, *o_comment*, *o_orderstatus*, and *o_orderpriority*. Using the command line tool `dbgen` of the TPC-H framework we generated an *Orders* table with 10M tuples. Based on this table we created different materialized views varying in the number of tuples and in the number of distinct values for specific attributes (in order to have base tables with different size), and defined different queries so that the influence of the factors $|\mathbf{r}|$, $|\mathbf{b}|$, $|\theta|$, $|\Theta|$ and $\oplus$ could be tested separately.

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

6.2 Results

In this section we present the results of the experiments in order to evaluate the performance of the `IndexedTCMDA-LI` algorithm and compare it with the performance of `BasicTCMDA` and `IndexedTCMDA`.

6.2.1 Scaling $|\mathbf{r}|$

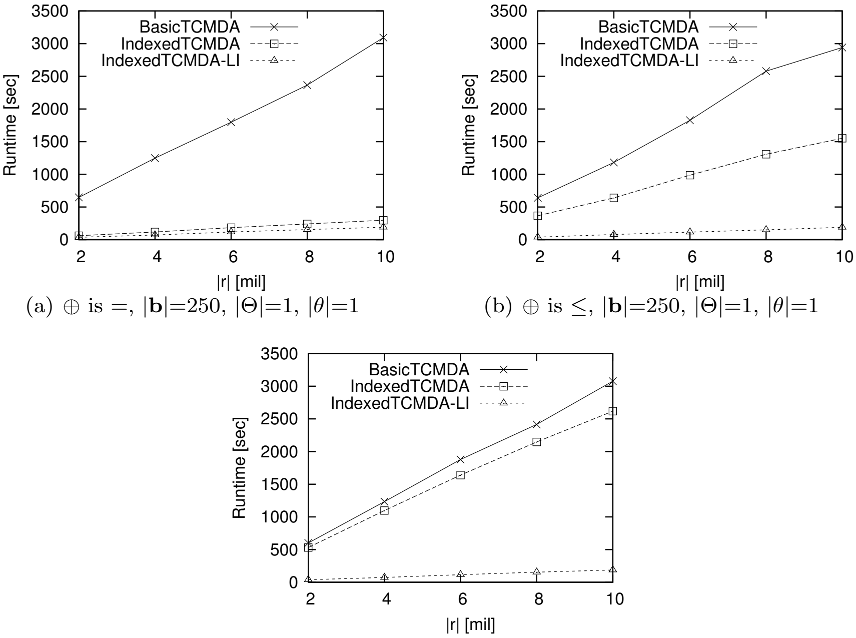
The experiments with scaling $|\mathbf{r}|$ are designed in order to test the effect of the size of the input relation $\mathbf{r}$ on the performance of the θ–MDA algorithms. For achieving this goal we created five materialized views *Orders_2M*, *Orders_4M*, ..., *Orders_10M* of the table *Orders* where *2M*, *4M*, ..., *10M* indicate that the view contains 2, 4, ..., 10 million of tuples respectively. Each of these materialized views has 250 distinct values for the attribute *o_orderdate*. The query used in these experiments is the following:

$$
\textbf{Query r:}\quad
\begin{array}{l}
\mathbf{r} : Orders\_XM \\
\mathbf{b} : \pi_{o\_orderdate}(\mathbf{r}) \\
l_1 : count(*) \\
\theta_1 : \mathbf{r}\_orderdate \oplus \mathbf{b}.o\_orderdate
\end{array}
$$

The query was executed using the different materialized views, i.e. by replacing $Orders\_XM$ with $Oders\_2M$, $Orders\_4M$, ..., and $Orders\_10M$, and different constraint operators $\oplus$, namely $=$, $\leq$, and $\neq$. Figure 14 and 15 show the results of these experiments. On the x-axis of the graphs the size of the input relation $\mathbf{r}$ is reported, the y-axis reflects the time in seconds needed by the algorithm for the computation of the result relation. From the results presented in Figure 14 and 15 we gain different insights.

One insight is, that for queries with conditions containing only $=$ constraint operators the algorithm `IndexTCMDA-LI` is faster than both the other algorithms. The slowest of the three algorithms is `BasicTCMDA` (see Figure 14(a)). For instance, `BasicTCMDA` is about 10 times slower than `IndexedTCMDA` and about 16 times slower than `IndexedTCMDA-LI` for the experiment with 10 million tuples for $\mathbf{r}$. This results can be explained due to the fact, that the `BasicTCMDA` algorithm for each tuple in $\mathbf{r}$ and each θ-conditions traverses all tuples of the result relation in order to find the tuples to be updated, which in this concrete example means about $|\mathbf{r}| \cdot |\Theta| \cdot |\mathbf{b}| = 10\text{M} \cdot 1 \cdot 250 = 2500\text{M}$ iterations. The `IndexedTCMDA` algorithm, instead, uses the index to find the tuples of the result relation that have to be modified, and then traverses only these found tuples. For our test setting with $\oplus$ being $=$ for each tuple of $\mathbf{r}$ and each θ-condition only one tuple of the result relation is traversed, i.e. about $|\mathbf{r}| \cdot 1 \cdot 1 = 10\text{M}$ iterations have to be performed. In the case of `IndexedTCMDA-LI`, independently from the constraint operators used, the algorithm needs always about $|\mathbf{r}| \cdot |\Theta| \cdot 1$ iterations for the construction of the $\mathbf{x}_i^{eq}$'s resulting in about 10M iterations. The difference in the runtime between `IndexedTCMDA` and `IndexedTCMDA-LI` can be explained by the fact, that additionally to the iterations performed by the two algorithm for computing the result relation and the intermediate result relations respectively, the former does $|\mathbf{r}|$ iterations for computing the base table $\mathbf{b}$ and, the latter $|\mathbf{b}| \cdot |\Theta| \cdot 1$ for computing $\mathbf{x}$ from the $\mathbf{x}_i^{eq}$'s. This means that for the experiments with 10 million tuples `IndexedTCMDA` requires 10M+10M=20M iterations in total, whereas `IndexedTCMDA-LI` only 10M+(250·1·1)≈10M iterations.

Another insight is, that in the case of queries containing constraint operators different than $=$, such as $\leq$ and $\neq$, `IndexedTCMDA-LI` outperformes the runtime of both, `BasicTCMDA` and `IndexedTCMDA` (see Figure 14(b) and 14(c)). The better performance of `IndexedTCMDA-LI` compared to `BasicTCMDA` is due to the fact, that the latter makes about $|\mathbf{r}| \cdot |\Theta| \cdot |\mathbf{b}|$ iterations compared to the $|\mathbf{r}| \cdot |\Theta| \cdot 1$ of `IndexedTCMDA-LI`. Compared to `IndexedTCMDA`, the algorithm `IndexedTCMDA-LI` is faster because of the following reasons. `IndexedTCMDA` performs about $|\mathbf{r}| \cdot |\Theta| \cdot M$ iterations for the computation of the result relation, where M is the average number of matching tuples of the result relation for the processed tuples of the $\mathbf{r}$. In the case of the constraint operator $\leq$, in our experiments M is approximately $|\mathbf{b}|/2$ and for the constraint operator $\neq$ it is $|\mathbf{b}| - c$[11]. Compared to $M = 1$ for algorithm `IndexedTCMDA-LI`, the algorithm `IndexedTCMDA` results in much more iterations.



(a) $\oplus$ is $=$, $|\mathbf{b}|$=250, $|\Theta|$=1, $|\theta|$=1

(b) $\oplus$ is $\leq$, $|\mathbf{b}|$=250, $|\Theta|$=1, $|\theta|$=1

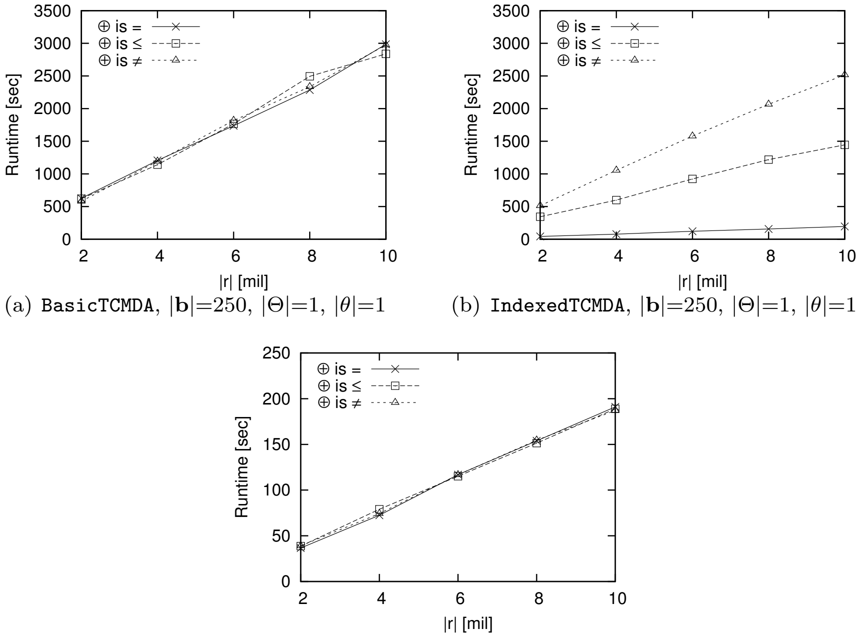
(c) $\oplus$ is $\neq$, $|\mathbf{b}|$=250, $|\Theta|$=1, $|\theta|$=1

Figure 14: Experiment Results with Scaling $|\mathbf{r}|$

Figure 15 depicts the comparison of the runtime of the algorithms for the experiments with scaling $|\mathbf{r}|$, focusing on the effect of the different constraint operators used. It is possible to recognize, that the runtime of the `BasicTCMDA` and `IndexdTCMDA-LI` algorithms is not significantly affected by the type of constraint operator used in the query (see Figure 15(a) and 15(b)). Instead, the execution time of the `IndexedTCMDA` algorithm highly depends from the type of constraint operators used (see Figure 15(c)). These phenomenon can be explained as follows. In the case of `BasicTCMDA` and `IndexedTCMDA-LI` the number of iterations done by the algorithms is always the same independently from which constraint

[11] See Section 5.2 for more details about these formulas.

operator is used. More precisely, the former algorithm performs always about $|\mathbf{r}| \cdot |\Theta| \cdot |\mathbf{b}|$ iterations, whereas, the latter algorithm always about $|\mathbf{r}| \cdot |\Theta| \cdot 1$. In the case of `IndexedTCMDA`, instead, the number of iterations performed is about $|\mathbf{r}| \cdot |\Theta| \cdot M$, where M is defined as in the previous paragraph. Due to the fact, that M depends from the type of used constraint operators, the runtime of `IndexedTCMDA` also depends from the used constraint operators. In our experiments M is 1 for the constraint operator $=$, approximately $|\mathbf{b}|/2$ for $\leq$, and $|\mathbf{b}| - 1$ for $\neq$.



(a) `BasicTCMDA`, $|\mathbf{b}|$=250, $|\Theta|$=1, $|\theta|$=1 (b) `IndexedTCMDA`, $|\mathbf{b}|$=250, $|\Theta|$=1, $|\theta|$=1

(c) `IndexedTCMDA-LI`, $|\mathbf{b}|$=250, $|\Theta|$=1, $|\theta|$=1

Figure 15: Comparison of Constraint Operators with Scaling $|\mathbf{r}|$

The experiments with scaling $|\mathbf{r}|$ showed that `IndexTCMDA-LI` outperforms `BasicTCMDA` and `IndexedTCMDA` in all settings of the tests.

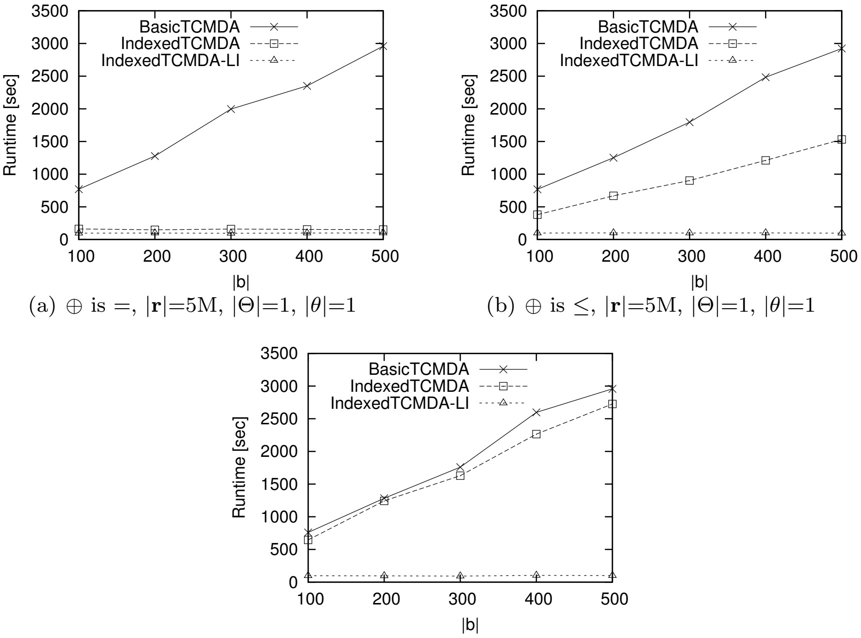
6.2.2 Scaling $|\mathbf{b}|$

The experiments with scaling $|\mathbf{b}|$ presented in this section, are designed in order to test the effect of the size of the base table on the performance of the θ–MDA algorithms. Also for this experiments we created five materialized views, namely *Orders_100*, *Orders_200*, ..., *Orders_500* of the table *Orders* where *100*, *200*, ..., *500* identify the number of distinct values for the attribute *o_orderdate* in the respective view. Each of these materialized views has 5 million tuples. The query used in these experiments is the following:

Query b: $\mathbf{r} : Orders\_X$
$\mathbf{b} : \pi_{o\_orderdate}(\mathbf{r})$
$l_1 : count(*)$
$\theta_1 : \mathbf{r}\_orderdate \oplus \mathbf{b}.o\_orderdate$

The query was executed using the different materialized views, i.e. by replacing *Orders_X* with *Orders_100*, *Orders_200*, ..., *Orders_500*, and different constraint operators $\oplus$, namely $=$, $\leq$, and $\neq$. Figure 16 and 17 show the results of these experiments. On the x-axis of the graphs the size of the base table **b** is reported, and the y-axis depicts the runtime of the algorithms in seconds.

From Figure 16 it is possible to recognize that in the experiment results with scaling $|\mathbf{b}|$ the `IndexedTCMDA-LI` algorithm has a lower runtime than `BasicTCMDA` and `IndexedTCMDA-LI` in all tested settings. This phenomenon could be identified also in the experiment with scaling $|\mathbf{r}|$ in Section 6.2.1 and the explanation for it is the same.



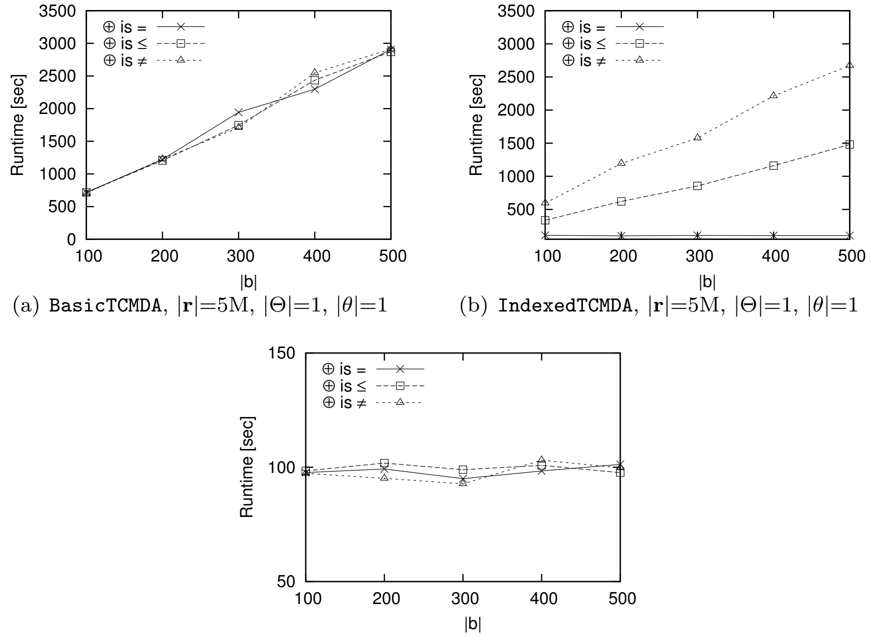
(a) $\oplus$ is $=$, $|\mathbf{r}|$=5M, $|\Theta|$=1, $|\theta|$=1

(b) $\oplus$ is $\leq$, $|\mathbf{r}|$=5M, $|\Theta|$=1, $|\theta|$=1

(c) $\oplus$ is $\neq$, $|\mathbf{r}|$=5M, $|\Theta|$=1, $|\theta|$=1

Figure 16: Experiment Results with Scaling $|\mathbf{b}|$

Also the comparison of the runtime of the θ–MDA algorithms in combination with different constraint operators depicted in Figure 17 features the same behaviour as for scaling $|\mathbf{r}|$. More precisely, the runtime of `BasicTCMDA` and `IndexedTCMDA-LI` is not affected significantly by using different constraint operators in our experiments, whereas, the runtime of `IndexedTCMDA` varies depending on the constraint operator used. The explanation for this phenomenon is the same as for scaling $|\mathbf{r}|$ (see previous section).



(a) BasicTCMDA, $|\mathbf{r}|$=5M, $|\Theta|$=1, $|\theta|$=1 (b) IndexedTCMDA, $|\mathbf{r}|$=5M, $|\Theta|$=1, $|\theta|$=1

(c) IndexedTCMDA, $|\mathbf{r}|$=5M, $|\Theta|$=1, $|\theta|$=1

Figure 17: Comparison of Constraint Operators with Scaling $|\mathbf{b}|$

Finally, it can be noticed that, whereas the runtime of the IndexedTCMDA-LI algorithm increases with increasing $|\mathbf{r}|$ (see Figure 15(c) on page 43), in the case of scaling $|\mathbf{b}|$, in our tests, the runtime remains constant for all tested settings of $|\mathbf{b}|$ (see Figure 17(c)). This should be the case, because the size of the $\mathbf{b}$ is not large enough with respect to the size of $\mathbf{r}$, in order to affect the runtime of the algorithm.

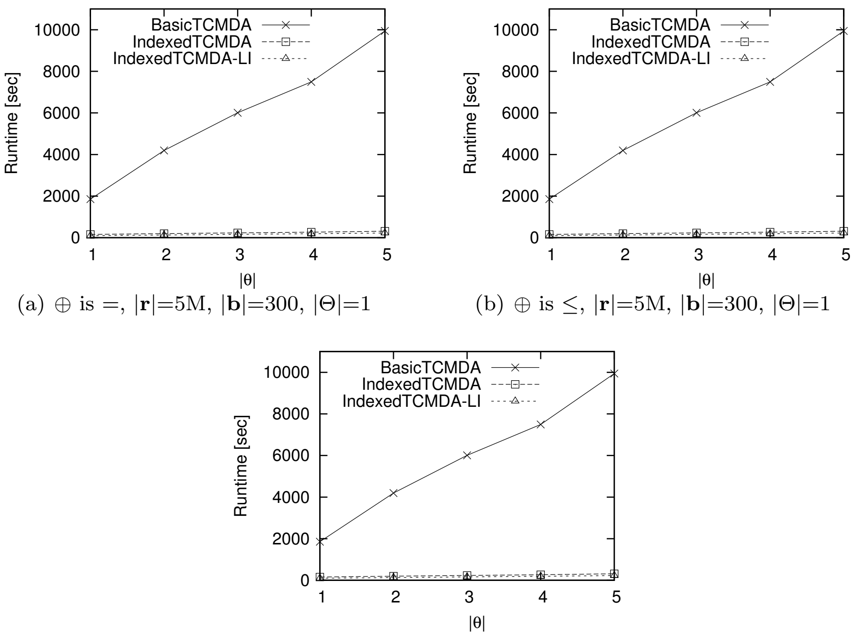
6.2.3 Scaling $|\theta|$

The experiments with scaling $|\theta|$ presented in this section, are designed in order to test how θ-conditions of different size (i.e. having a different number of constraints) affect the performance of the θ–MDA algorithms. For this experiments we used five materialized views *Order_1*, *Order_2*, ... *Order_5* of the table *Orders* where *1*, *2*, ..., *5* identify the number of *o_orderdate* attributes materialized in the view. The different *o_orderdate* attributes have been materialized by joining the *Orders* table with itself two or more times. In each of the materialized views the number of distinct tuples over the *o_orderdate* attributes (independently whether the view has one or more *o_orderdate* attributes) is 300. For example, *Orders_2* consists of two date attributes $o\_orderdate_1$ and $o\_orderdate_2$ and the number of tuples $(o\_orderdate_1, o\_orderdate_2)$ in the view with distinct values is 300 resulting is a base table of size 300. Further, the number of tuples it the views is 5 million. The query used in these experiments is the following:

$$\begin{array}{ll} \textbf{Query } \theta\textbf{:} & \mathbf{r} : Orders\_X \\ & \mathbf{b} : \pi_{o\_orderdate_1,\dots,o\_orderdate_{|\theta|}}(\mathbf{r}) \\ & l_1 : count(*) \\ & \theta_1 : \mathbf{r}\_orderdate_1 \oplus \mathbf{b}.o\_orderdate_1 \wedge .. \\ & \quad .. \wedge \mathbf{r}\_orderdate_{|\theta|} \oplus \mathbf{b}.o\_orderdate_{|\theta|} \end{array}$$

The query was executed using the different materialized views, i.e. by replacing *Orders_X* with *Orders_1*, *Orders_2*, ..., *Orders_5*, different constraint operators $\oplus$, namely $=$, $\leq$, and $\neq$, and the values 1, 2, ..., 5 for $|\theta|$. Figure 18 and 19 depict the results of these experiments. On the x-axis of the graphs the number of constraints of the θ-condition is reported and the y-axis depicts the runtime of the algorithms in seconds.

Also in these experiments the `IndexedTCMDA-LI` algorithm performs better than both the algorithm `BasicTCMDA` and `IndexedTCMDA` (see Figure 18). Again, the explanation for this results is the same as the one for the experiments with scaling $|\mathbf{r}|$ (see Section 6.2.1).



(a) $\oplus$ is $=$, $|\mathbf{r}|$=5M, $|\mathbf{b}|$=300, $|\Theta|$=1

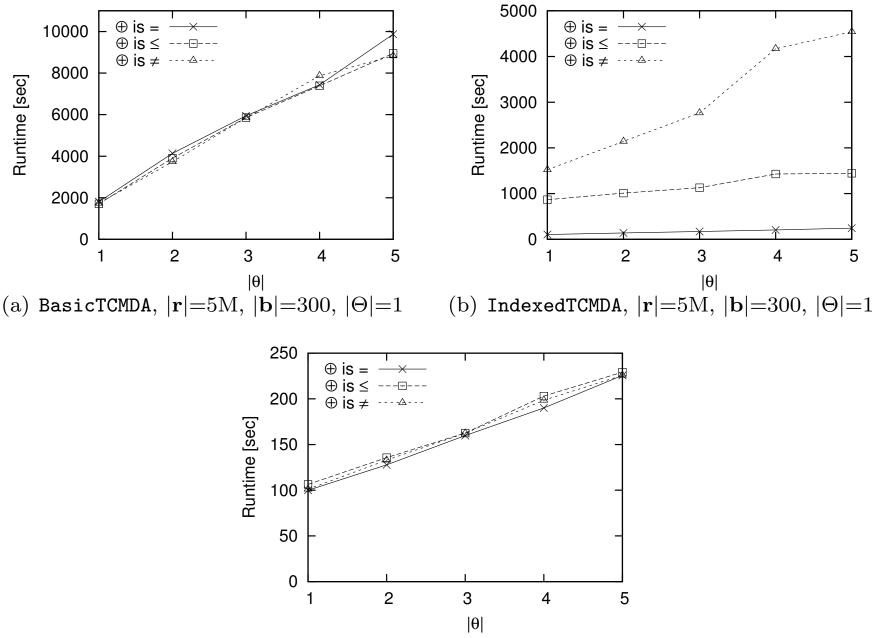
(b) $\oplus$ is $\leq$, $|\mathbf{r}|$=5M, $|\mathbf{b}|$=300, $|\Theta|$=1

(c) $\oplus$ is $\neq$, $|\mathbf{r}|$=5M, $|\mathbf{b}|$=300, $|\Theta|$=1

Figure 18: Experiment Results with Scaling $|\theta|$

In these experiments with scaling $|\theta|$ `BasicTCMDA` reaches execution times that are higher than the execution times in the experiments for scaling $|\mathbf{r}|$ and scaling $|\mathbf{b}|$. For instance, the `BasicTCMDA` algorithm in the experiments with scaling $|\mathbf{r}|$ with 10 million tuples in $\mathbf{r}$ (and $|\mathbf{b}|$=250 and $\oplus$ is $\neq$) has a runtime of about 51 minutes and with scaling $|\mathbf{b}|$ with 500 tuples in $\mathbf{b}$ (and $|\mathbf{r}|$=5M and $\oplus$ is $\neq$) the runtime is of about 49 minutes. In these experiments with scaling $|\theta|$, instead, the runtime of `BasicTCMDA` for $|\theta|$=2 is about 63 minutes being already more than 25 percent larger than the maximum runtime of the previously mentioned settings; for $|\theta|$=5 the runtime reaches 148 minutes. This is the case, because for `BasicTCMDA` the check of the θ-condition is implemented by an iteration through all constraints of the conditions. This means that the number of iterations performed by the algorithm increases with an increasing number of constraints $|\theta|$. In fact, the implementation of `BasicTCMDA` performs about $|\mathbf{r}| \cdot |\Theta| \cdot |\theta| \cdot |\mathbf{b}|$ iterations.

For the `IndexedTCMDA` and `IndexedTCMDA-LI` algorithms, instead, the impact of $|\theta|$ is not that big as for `BasicTCMDA`. Indeed, the line representing the runtime of the `BasicTCMDA` algorithm is much steeper than the line for the `IndexedTCMDA` and `IndexedTCMDA-LI` algorithms (Figure 18). For instance the runtime for `BasicTCMDA` algorithm for $|\theta|$=5 and constraint operator $\neq$ is 5.1 times larger than for $|\theta|$=1. For the `IndexedTCMDA` and the `IndexedTCMDA-LI` algorithms, instead, the runtime for the same settings is 2 and 1.2 times larger, respectively. This is due to the help of the index used by `IndexedTCMDA` and `IndexedTCMDA-LI` for the retrieval of the tuples that have to be modified which restricts the number of iterations necessary for checking θ-constraints.



(a) `BasicTCMDA`, $|\mathbf{r}|$=5M, $|\mathbf{b}|$=300, $|\Theta|$=1 (b) `IndexedTCMDA`, $|\mathbf{r}|$=5M, $|\mathbf{b}|$=300, $|\Theta|$=1

(c) `IndexedTCMDA-LI`, $|\mathbf{r}|$=5M, $|\mathbf{b}|$=300, $|\Theta|$=1

Figure 19: Comparison of Constraint Operators with Scaling $|\theta|$

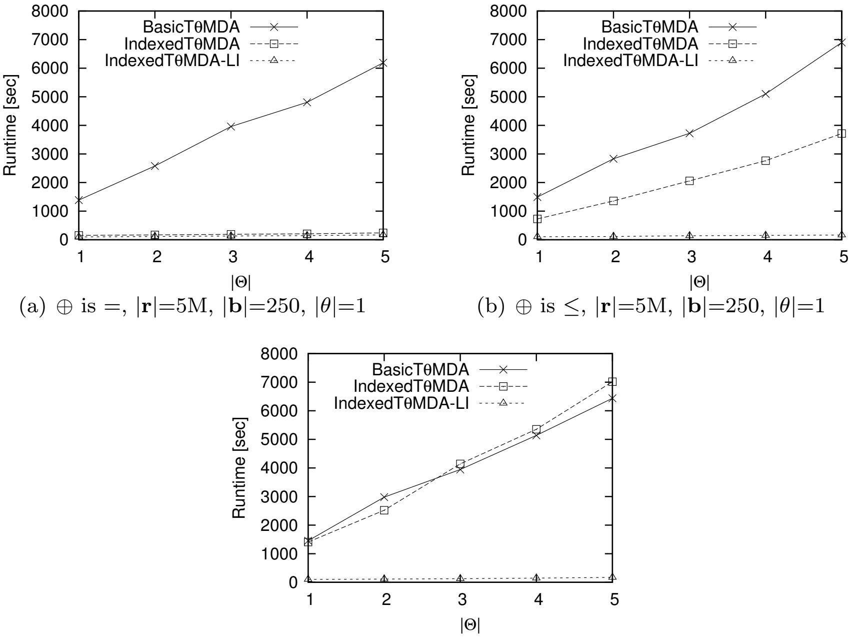
The comparison of the runtime of the algorithm for the different constraint operators among each other is depicted in Figure 19 and features the same behaviour as for the experiments with scaling $|\mathbf{r}|$ and scaling $|\mathbf{b}|$, namely, the runtime of the `BasicTCMDA` and `IndexedTCMDA-LI` algorithms is not affected significantly by using different constraint operators, whereas, the runtime of `IndexedTCMDA` varies depending on the constraint operator used. The reason for this phenomenon is the same as for scaling $|\mathbf{r}|$ and $|\mathbf{b}|$.

6.2.4 Scaling $|\Theta|$

In this section we present the experiments with scaling $|\Theta|$. They are designed in order to test how the number of θ-conditions used in a query affects the performance of the θ–MDA algorithms. For these experiments we used one materialized view with 5 million tuples and 250 distinct values for the *o_orderdate* attribute. The query used in these experiments is the following:

$$
\begin{array}{ll}
\textbf{Query } \Theta\textbf{:} & \mathbf{r} : Orders \\
& \mathbf{b} : \pi_{o\_orderdate}(\mathbf{r}) \\
& l_1 : count(*) \\
& \theta_1 : \mathbf{r}\_orderdate \oplus \mathbf{b}.o\_orderdate \\
& ... \\
& l_{|\Theta|} : count(*) \\
& \theta_{|\Theta|} : \mathbf{r}\_orderdate \oplus \mathbf{b}.o\_orderdate
\end{array}
$$

The query was executed using the abovementioned view, the constraint operators $=$, $\leq$, and $\neq$, and one to five θ-conditions. Figure 20 and 21 provide an overview of the results of theses experiments. On the x-axis of the graphs the number of used θ-conditions is reported and the y-axis depicts the runtime of the algorithms in seconds.



(a) $\oplus$ is $=$, $|\mathbf{r}|$=5M, $|\mathbf{b}|$=250, $|\theta|$=1

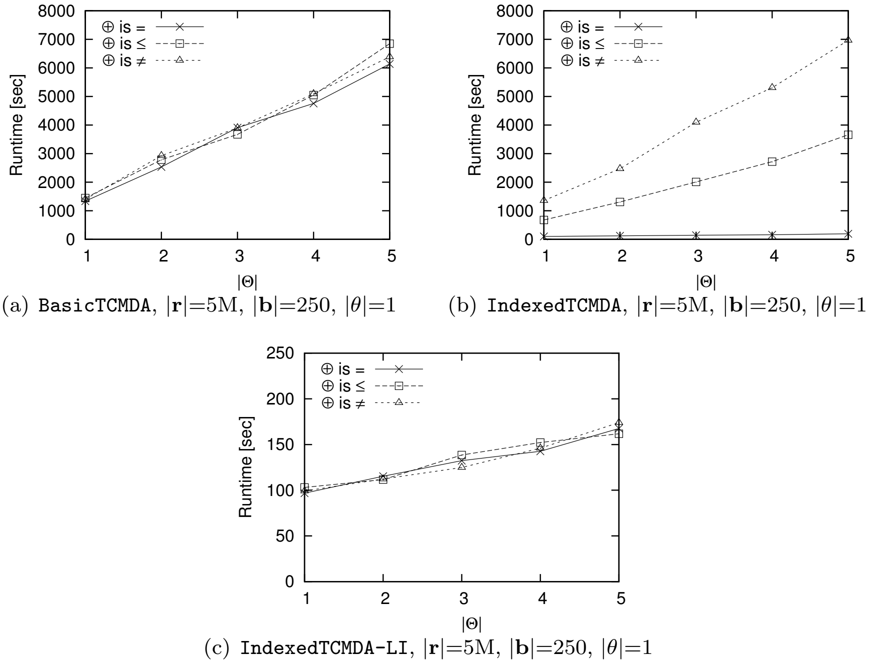
(b) $\oplus$ is $\leq$, $|\mathbf{r}|$=5M, $|\mathbf{b}|$=250, $|\theta|$=1

(c) $\oplus$ is $\neq$, $|\mathbf{r}|$=5M, $|\mathbf{b}|$=250, $|\theta|$=1

Figure 20: Experiment Results with Scaling $|\Theta|$

Again the `IndexedTCMDA-LI` algorithm performes better than `BasicTCMDA` and `IndexedTCMDA` (see Figure 20). A deeper analysis of the runtime behaviour of the three algorithms shows, that the number of θ-conditions has a big impact on the execution time of the `BasicTCMDA` and the `IndexedTCMDA` algorithms, whereas, the `IndexedTCMDA-LI` algorithm scales better for an increasing $|\Theta|$. More precisely, the execution time of `BasicTCMDA` for $|\Theta|$=5 is about 4.6 times larger than for $|\Theta|$=1, of `IndexedTCMDA` it is about 5.1 times larger, and of `IndexedTCMDA-LI` only about 1.8 times larger. This can be explained by the fact, that `IndexedTCMDA-LI` for each tuple of $\mathbf{r}$ and each θ-condition iterates through and updates only one tuple of $\mathbf{x}^{eq}$. Instead, `BasicTCMDA` and `IndexedTCMDA`, for each tuple of $\mathbf{r}$ and each θ-condition, iterate through and update several tuples of $\mathbf{x}$.

Figure 21 depicts the comparison of the runtime for the different constraint operators among each other. It is possible to recognize that the behaviour is similar as for the other experiments, namely, the runtime of the `BasicTCMDA` and `IndexedTCMDA-LI` algorithm is not affected significantly by using different constraint operators, whereas, the runtime of `IndexedTCMDA` varies depending on the constraint operator used. The explanation of this phenomenon is the same as for the other experiments.



(a) `BasicTCMDA`, $|\mathbf{r}|$=5M, $|\mathbf{b}|$=250, $|\theta|$=1 (b) `IndexedTCMDA`, $|\mathbf{r}|$=5M, $|\mathbf{b}|$=250, $|\theta|$=1

(c) `IndexedTCMDA-LI`, $|\mathbf{r}|$=5M, $|\mathbf{b}|$=250, $|\theta|$=1

Figure 21: Comparison of Conatraint Operators with Scaling $|\Theta|$

With the experiments for scaling $|\Theta|$ we conclude the experiments section and continue with some concluding remarks and proposals for future work in Section 7.

7 Conclusions and Future Work

Today complex multi-dimensional aggregation queries find application in many areas such as business intelligence or in scientific disciplines. The θ–MDA operator presented in [2] provides a tool for the flexible formulation and efficient evaluation of complex aggregation queries over multi-dimensional data. The evaluation strategy applied by the operator structures the computational process into one part concerned with constructing the base table, and one part concerned with computing the aggregates. In cases in which the base table is a projection of the input table, that approach results in loading the input relation in both parts of the computational process. In typical θ–MDA settings the input relation is very large, thus, loading it twice impacts the evaluation performance.

In this thesis we propose an evaluation approach for θ–MDA which needs to load the input relation only once for cases in which the base table is a projection of the input table. The main idea of the approach is to compute the result groups of the base table on demand, during the computation of the aggregates. We refer to this computation on-the-fly of the result groups as late initialization of the base table. In the thesis we provide an algorithm for the evaluation of θ–MDA with late initialization, namely `IndexedTCMDA-LI`, and present performance experiments to evaluate it. In the experiments we compare the performance of `IndexedTCMDA-LI` with the performance of the algorithms `BasicTCMDA` and `IndexedTCMDA` for θ–MDA proposed by [2] which do not use the late initialization strategy. The results show that `IndexedTCMDA-LI` performs better than the other two algorithms in all tested settings which has two main reasons. One reason is the single load of the input relation in `IndexedTCMDA-LI` which yields a performance improvement of up to about 65 percent (on average about 54 percent). Another reason is the reduction of all constraint operators to equality in `IndexedTCMDA-LI` that has a huge impact of the performance of the algorithm for aggregation queries using constraint operators different than $=$. Indeed, for such queries in our experiments `IndexedTCMDA-LI` reaches execution times being up to forty times shorter as for `IndexedTCMDA` and `BasicTCMDA`.

Future work could focus on different performance issues of the late initialization strategy presented in this thesis. The approach we developed performs particularly well in our experiments, in which we used base table with up to 500 result groups. However, for cases with large base tables, for instance, storing several hundred thousand result groups, the performance of θ–MDA with late initialization could decline significantly due to the process of computing the result relation $\mathbf{x}$ based on the intermediate result relations $\mathbf{x}_i^{eq}$. Thus, future research could investigate limitations of our approach with respect to a growing number of result groups and search for strategies to handle these limitations. Another future research could be to implement the θ–MDA algorithm with late initialization directly into a database management system such as PostgreSQL[12] and than to compare its efficiency with the performance of SQL formulations of the same queries.

[12] http://www.postgresql.org/

References

[1] Michael Akinde, Damianos Chatziantoniou, Theodore Johnson, and Samuel Kim. The MD-join: An operator for complex OLAP. In Proceedings of the 17th International Conference on Data Engineering, pages 524–533, Washington, DC, USA, 2001. IEEE Computer Society.

[2] Michael Akinde, Michael H. Böhlen, Damianos Chatziantoniou, and Johann Gamper. θ-constrained multi-dimensional aggregation. Information Systems, 36:341–358, April 2011. ISSN 0306-4379.

[3] Michael O. Akinde and Michael H. Böhlen. Generalized MD-joins: Evaluation and reduction to SQL. In Proceedings of the VLDB 2001 International Workshop on Databases in Telecommunications II, DBTel '01, pages 52–67, London, UK, 2001. Springer-Verlag. ISBN 3-540-42623-X.

[4] Michael O. Akinde and Michael H. Böhlen. Efficient computation of subqueries in complex OLAP. In Proceedings of the 19th International Conference on Data Engineering, pages 163–, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[5] Michael O. Akinde, Michael H. Böhlen, Theodore Johnson, Laks V. S. Lakshmanan, and Divesh Srivastava. Efficient OLAP query processing in distributed data warehouses. Information Systems, 28:111–135, March 2003. ISSN 0306-4379.

[6] E. F. Codd, S. B. Codd, and C. T. Salley. Providing OLAP (On-Line Analytical Processing) to User-Analysis: An IT Mandate, 1993.

[7] Jim Gray, Adam Bosworth, Andrew Layman, Don Reichart, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In Proceedings of the Twelfth International Conference on Data Engineering, pages 152–159, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7240-4.

[8] Jiawei Han, Yixin Chen, Guozhu Dong, Jian Pei, Benjamin W. Wah, Jianyong Wang, and Y. Dora Cai. Stream cube: An architecture for multi-dimensional analysis of data streams. Distributed Parallel Databases, 18: 173–197, September 2005. ISSN 0926-8782.

[9] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In Proceedings of the 1996 ACM SIGMOD international conference on Management of data, pages 205–216, New York, NY, USA, 1996. ACM. ISBN 0-89791-794-4.

[10] SQL Standards Commitee-American National Standards Institute. ISO/IEC 9075:1999 information technology–database languages–SQL, 1999.

[11] SQL Standards Commitee-American National Standards Institute. Information technology–database languages–sql–AMENDMENT 1: On-line analytical processing (SQL/OLAP), 2001.

[12] SQL Standards Commitee-American National Standards Institute. ISO/IEC 9075:2003 information technology–database languages–SQL, 2003.

[13] Kenneth E. Iverson. A programming language. John Wiley & Sons, Inc., New York, NY, USA, 1962. ISBN 0-471430-14-5.

[14] 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, VLDB '06, pages 823–833. VLDB Endowment, 2006.

[15] Eric Miller and Frank Manola. RDF primer. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[16] Barinderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of data cubes and summary tables in a warehouse. In Proceedings of the 1997 ACM SIGMOD international conference on Management of data, pages 100–111, New York, NY, USA, 1997. ACM. ISBN 0-89791-911-4.

[17] Radhika Sridhar, Padmashree Ravindra, and Kemafor Anyanwu. RAPID: Enabling scalable ad-hoc analytics on the semantic web. In Proceedings of the 8th International Semantic Web Conference, Proceedings of ISWC 2009, pages 715–730, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-04929-3.

[18] Goetz Graefe Usama, Usama Fayyad, and Surajit Chaudhuri. On the efficient gathering of sufficient statistics for classification from large SQL databases. In Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining, pages 204–208. AAAI Press, 1998.

[19] Wikipedia. Business intelligence — Wikipedia, the free encyclopedia, 2011. URL `http://en.wikipedia.org/wiki/Business_intelligence`. [Online; accessed 16-February-2011].

[20] Wikipedia. Map reduce — Wikipedia, the free encyclopedia, 2011. URL `http://en.wikipedia.org/wiki/MapReduce`. [Online; accessed 23-February-2011].

[21] Wikipedia. Online analytical processing — Wikipedia, the free encyclopedia, 2011. URL `http://en.wikipedia.org/wiki/Online_analytical_processing`. [Online; accessed 16-February-2001].

[22] Jun Yang and Jennifer Widom. Incremental computation and maintenance of temporal aggregates. In Proceedings of the 17th International Conference on Data Engineering, pages 51–60, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1001-9.

[23] Jun Yang and Jennifer Widom. Incremental computation and maintenance of temporal aggregates. The VLDB Journal, 12:262–283, October 2003. ISSN 1066-8888.

[24] Donghui Zhang, Alexander Markowetz, Vassilis J. Tsotras, Dimitrios Gunopulos, and Bernhard Seeger. On computing temporal aggregates with range predicates. ACM Trans. Database Syst., 33:12:1–12:39, June 2008. ISSN 0362-5915.

[25] Donhui Zhang, Alexander Markowetz, Vassilis Tsotras, Dimitrios Gunopulos, and Bernhard Seeger. Efficient computation of temporal aggregates with range predicates. In Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, PODS '01, pages 237–245, New York, NY, USA, 2001. ACM. ISBN 1-58113-361-8.