Department of Informatics, University of Zürich

BSc Thesis

Optimization of Mixed Queries in MonetDB System

Alphonse Mariyagnanaseelan

Matrikelnummer: 15-712-698

Email: alphonse.mariyagnanaseelan@uzh.ch

September 12, 2018

supervised by Prof. Dr. Michael Böhlen and Oksana Dolmatova

University of Zurich[UZH]

Department of Informatics

Acknowledgements

I would first like to express my sincere gratitude to my supervisor Oksana Dolmatova. She was all ears whenever I ran into trouble, had new ideas or when I had questions. I am very thankful for how she steered me in the right direction and for the feedback she provided. I would also like to thank Prof. Dr. Michael Böhlen for granting me the great opportunity to write my Bachelor thesis on a very interesting topic at the Database Technology Group of the University of Zurich.

I also want to express my profound gratitude to my parents, sister, friends and particularly my girlfriend Arlin Jebarsan for their support and encouragement throughout my studies and during the process of writing this thesis. This achievement would not have been possible without them.

Abstract

The current approach to analyze scientific data stored in a DBMS is to export the data to a statistics software like MATLAB or R, perform the analysis, and then import the data back into the DBMS, in case the result has to be stored or further manipulated using relational operations like selections or joins. Those steps are necessary due to the fact that DBMSs currently only support simple aggregation operations and vector operations with attributes of a relation, but more sophisticated linear algebra operations are not available yet. We implement the matrix addition operation in MonetDB, a column-store DBMS. Further, we extend the query optimizer of MonetDB to optimize mixed queries, that consists of both matrix related operations and relational operations. We focus on the optimization of matrix addition in combination with selections, where we investigate the properties of such an optimization, we define equivalence rules for the rewriting of such mixed queries, and we discuss the trade-off in our optimization. The experimental evaluation shows that our optimization can reduce the query time by up to 95%, however the mentioned trade-off has a measurable and counterproductive effect in some cases. Since the optimization is done by the query optimizer, the user can simply declare the matrix operation, similar to join operations, and perform other relational operations on the result.

Zusammenfassung

Die derzeitige Herangehensweise für die Auswertung wissenschaftlicher Daten, die in einer Datenbank gespeichert sind, ist, dass die Daten in eine Statistik Software wie MATLAB oder R exportiert, die Auswertung durchgeführt und dann die Daten wieder zurück in die Datenbank importiert wird, falls die resultierten Daten in der Datenbank gespeichert oder mit relationalen Operationen, wie Selektionen oder Joins, weiter verarbeitet werden sollen. Diese Schritte sind notwendig, da Datenbankmanagementsysteme zurzeit nur einfache Aggregationsoperationen und Vektoroperationen mit Attributen einer Relation ermöglichen, hingegen komplexere Operationen der linearen Algebra noch nicht zur Verfügung stehen. Wir implementieren die Matrix Addition Operation in MonetDB, eine Spaltenorientierte Datenbank. Ferner erweitern wir den Query Optimierer von MonetDB, sodass dieser gemischte Queries, welche aus Matrix Operationen sowie relationalen Operationen bestehen, optimiert. Wir fokussieren uns auf die Optimierung der Matrix Addition Operation in Kombination mit Selektionen, wobei wir die Eigenschaften solcher Optimierungen untersuchen, Equivalenzregeln für die Umformulierung von gemischten Queries definieren und den Trade-off in unserer Optimierung diskutieren. Die experimentelle Evaluierung zeigt, dass unsere Optimierung die Abfragezeit um bis zu 95% verringern kann, jedoch hat der erwähnte Trade-off einen messbaren und kontraproduktiven Effekt in bestimmten Fällen. Da die Optimierung durch den Query Optimierer durchgeführt wird, kann der Benützer, ähnlich wie bei Join Operationen, die Matrix Operation deklarieren und andere relationale Operationen auf dem Resultat ausführen.

Contents

1. Introduction

Since the beginning of the Information Age, scientists as well as corporations produce large amounts of data. The amount of data that is produced, however, has grown dramatically and keeps growing. Database management systems (DBMS) provide an optimal solution to store structured data while enforcing constraints on the data, performing consistency checks, and enables the usage of indices on data that is often accessed. They allow us to query the data, combine different sets of data and filter them using an abstract and declarative language. Even simple mathematical operations like *summation*, *average* and *count*, but also vector operations, like the addition of two columns of a relation, can be performed.

Scientific data, however, often has to be processed in more sophisticated mathematical operations such as linear algebra operations. Since such operations currently cannot be performed in DBMSs, the usual approach is to export the data to some statistics software like MATLAB, R or to process the data using programming languages like Python. In case the data has to be filtered or combined with another set of data, the processed data has to be imported back into the DBMS, which implies time and memory costs, and is also more prone to errors. R nowadays provides database integrations, which allows us to directly connect to the database, so that we can process data without importing and exporting them. But there is still a transformation of data structure and the optimizer of the DBMS stay unused for the analytical operations.

If we integrated such linear algebra operations into DBMSs, not only do we profit from the saved time and memory costs, but we can also benefit from staying on the same data structure and using already existing facilities of DBMSs. Namely, we can use and extend the query optimizer to further improve the query time of queries that contain both matrix operations as well as relational operations. Selections, that would normally be performed on the result relation of a linear algebra operation, could be used to shrink the input to the linear algebra operation.

Column-store DBMSs are known to have performance advantages when it comes to aggregations, analytical calculations, operations that involve only few columns. Since they physically store tables by columns, whereas traditional DBMSs store them by rows, they have some interesting advantages, e.g. compression of repeating value, or performing vector operations. These properties are certainly useful when it, for example, comes to storing sparse matrices or when performing vector-based matrix operations.

The goals of this thesis are to implement the matrix addition operation in MonetDB, investigate on optimizations of queries, that involve both relational and matrix related operations, in particular selections and the matrix addition operation, and extend the query optimizer of MonetDB to perform such optimization for the matrix addition operation.

2. Problem Definition

The goal of this thesis is to implement matrix addition in a column-store database management system without introducing a new datatype, and then extend the query optimizer of it to improve the performance of mixed queries, that contain matrix operations as well as relational operations, in particular the matrix addition operation and selections.

Attributes in a relation can hold many different data types, not limited to numeric types only, and the records of a relation do not have an ordering. For matrices however the order of columns as well as the order of rows is significant. Moreover, matrices can only hold numeric values. Thus matrix addition, as shown in Figure 2.1, and generally matrix operations are not defined on relations.

```
1  SELECT *
2  FROM (r ON a, b ORDER BY d) ADD (s ON e, f ORDER BY g, h)
3  WHERE g > 1;
```

Listing 2.1: SQL syntax of matrix addition

We create new SQL construct to refer to matrices in the context of matrix operations. The matrix operation is placed in the `FROM` clause of an SQL query, where the input relation is defined, implying that the result of a matrix operation on relations is again a relation. The concrete SQL syntax for a matrix addition is shown in Listing 2.1. The additional parameters provide the necessary constraints to use the relations as matrices during one relational operation. When the matrix operation is evaluated, we get a new relation as result, that can be further modified in other relational operations as well as matrix operations. As implied, the input relations r and s can be replaced by subqueries, which also allows us to perform nested matrix operations.

Figure 2.2 depicts the relational version of the matrix addition shown in Figure 2.1, and is

$$R = \begin{pmatrix} 5 & 3 \\ 1 & 3 \\ 5 & 2 \\ 2 & 3 \\ 6 & 1 \end{pmatrix}, \quad S = \begin{pmatrix} 2 & 4 \\ 3 & 3 \\ 4 & 3 \\ 1 & 2 \\ 1 & 4 \end{pmatrix}, \quad R + S = \begin{pmatrix} 7 & 7 \\ 4 & 6 \\ 9 & 5 \\ 3 & 5 \\ 7 & 5 \end{pmatrix}$$

Figure 2.1.: Matrix R and S, addition of R and S

r

a	b	c	d
6	1	8	m
5	3	6	i
1	3	6	j
5	2	5	k
2	3	3	l

$+$

s

e	f	g	h
1	2	4	3
2	4	1	2
3	3	1	9
4	3	2	8
1	4	4	9

$=$

$result$

a	b	c	d	g	h
7	7	6	i	1	2
4	6	6	j	1	9
9	5	5	k	2	8
3	5	3	l	4	3
7	5	8	m	4	9

Figure 2.2.: Addition of relations r and s

produced by Query 2.1. The relations r and s contain additional attributes that do not belong to the matrix, namely c, d, g and h. Those attributes provide additional information, that can be interpreted as an attachment to the rows of the matrix. They also provide the information of how the matrices are ordered, which is what enables us to use these relations in matrix operations in the first place.

In the Query 2.1 we also perform a selection on the result relation of the matrix addition. We refer to this type of queries as *mixed queries*, since they consist of both relational and matrix related operations. Note that in our example query the selection is done after the matrix addition. However, in case the selection filters out many tuples, that is, the selection has a low selectivity, we should consider performing it before executing the matrix operation.

The matrix addition is an operation of linear complexity dependent on the number of tuples n and the number of columns m, i.e. $\mathcal{O}(n \cdot m)$. To perform the matrix addition operation on relations, we also have to sort the tuples, which leads to a complexity of $\mathcal{O}(n \cdot m + n \log n)$. If m is very small, the asymptotic complexity of the matrix addition operation degenerates to the complexity of the sorting. Selection has a complexity of $\mathcal{O}(n)$ in the general case. Since the sorting has to be performed before the selection in either case, it might not make sense to perform the selection before the addition anymore. We want to find threshold values to determine when not to use the optimization.

In this thesis, we provide an implementation of the matrix operation, including functions that can be reused for the implementation of other matrix operations. Moreover, we examine the conditions of the optimization of the matrix addition operation, give a concrete implementation of the optimization in a DBMS and evaluate the effectiveness of the implementation.

3. Approach

In this chapter we look at MonetDB, the DBMS we used to implement the matrix addition and its optimizations. We look at how MonetDB stores information on the disk, what data structures it uses to represent the data, how a query is processed, and which intermediary steps are taken to create the execution plan.

We then look at how the matrix addition operation can be defined on relation. Based on that, we describe the extensions we have made to MonetDB for the implementation of the matrix addition.

Then we consider the optimization of the matrix addition operation. We describe the constraints, formulate equivalence rules and discuss the trade-offs. Last, we look at the responsibilities of the query optimizer, and where in the architecture of MonetDB this optimization is actually done.

3.1. MonetDB

MonetDB is a pioneer in column-store database management systems, which is being developed since 1993 in the Centrum Wiskunde & Informatica (CWI) in the Netherlands. Besides being used in health care and telecommunications, it also finds usage as scientific databases [1]. Further, it is also used as a base for data management research.

Architecture

As a column-store DBMS, MonetDB stores tables as a collection of attributes, instead of storing them by tuples. Internally columns are stored in Binary Association Tables (BAT), which work like arrays. The index of a BAT is called object-identifier (OID) and is used to identify records. Thus a row of a table has the same OID across the BATs that refer to the columns of the table. Figure 3.1 illustrates how the relation r is stored internally in MonetDB.

MonetDB has a layered architecture, where the top layer is the SQL front-end which acts as the access point for users. The user's SQL query, which is declarative and operates on relations, is step by step transformed, first into a symbol tree, then into a relation tree, which is in MonetDB the internal representation of the query tree. Further, the relation tree is transformed into a statement tree, which is finally translated to the MonetDB Assembly Language (MAL), an imperative language that operates on BATs. During that transformation the optimizer performs

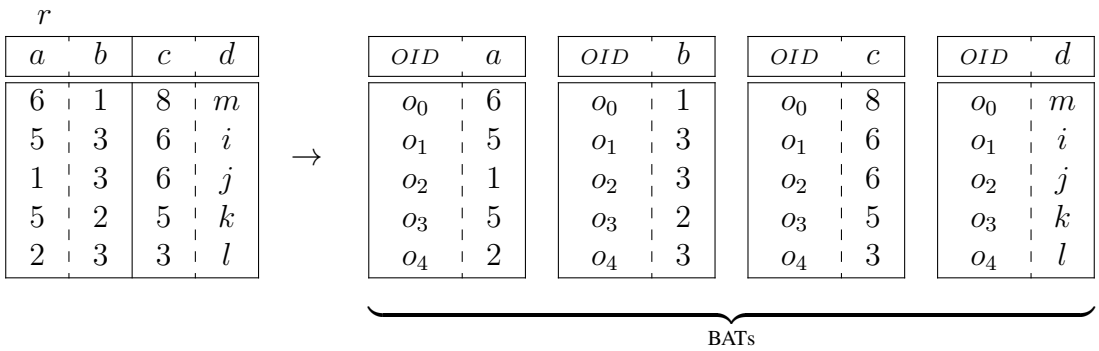

Figure 3.1.: BATs of relation r

strategic optimizations, by reordering, merging and removing relations. This is done on the relation tree by applying optimization rules and heuristics [1].

In the middle layer, the MAL-plan that has been produced based on the input query is optimized. This optimization is called the tactical optimization and it is done in a fashion that is common in compilers of programming languages [1]. Part of the tactical optimization is also the lazy evaluation, which is to only perform calculations if the result is needed. BATs that are not needed for the creation of the result relation are not evaluated, even if they have been defined in the above layers. Thus, projections for example do not have to be optimized during the strategic optimization, which, I believe, reduces the complexity of the strategic optimizer's architecture.

On the lowest layer MonetDB processes attributes of relations as BATs. MonetDB includes a library which consists of BAT algebra operators, which correspond to MAL instructions in the MAL-plan. MAL instructions take BATs as input and perform simple BAT algebra operations on them. Complex operations are broken down into operations as small as possible, which allows them to be reused and which keeps the complexity of the code low.

Arithmetic Operations

Database management systems, and MonetDB in particular, already supports basic arithmetic operations, e.g. scalar multiplication of a column, addition of two columns as shown in Listing 3.1, etc. However, this method only allows us to perform operations on one single relation. To perform a matrix operation where the input matrices must be generated from different relations, we have perform joins first. An approach to matrix addition using such joins and vector addition is only possible in special cases, for example if the relations have a one-to-one relationship.

```
SELECT a + b FROM r;
```

Listing 3.1: Addition of two columns in SQL

We implement the syntax as shown in 2.1, which provides a layer of abstraction. Matrix operations can now be defined in the `FROM` clause of the query, where the input relation is defined. This enables us to directly work with the result of the matrix operation as a relation, and perform relational operations on it. Our implementation of matrix addition internally uses the already existing BAT algebra operation for addition.

3.1.1. Symbol Tree

The DBMS client takes an SQL query and passes it to the server, where it is parsed. Yacc (Yet Another Compiler-Compiler) is a parser generator that is used in MonetDB to generate the SQL parser. A context-free grammar with actions (snippet of C code) attached to its rules (keywords and variables) is given to Yacc as input. The C snippets are executed as soon as the corresponding rule is fulfilled, which is used to create and extend nodes of the parse tree, that we also call *symbol tree*.

Each symbol tree node has a type that is marked with a token, which refers to a keyword. A symbol tree node's child can be a number, a character, a string, another symbol or a list of symbols. Due to the latter the symbol tree is highly flexible, which allows us to parse sophisticated syntaxes. However, the symbol tree only contains the structure of the query, the correctness of referred relations and attribute names. Due to lack of a rigorous structure in the symbol tree, we can't do much beyond parsing with it.

3.1.2. Relation Tree

As a next step, the symbol tree is transformed into the *relation tree*. Each relation tree node represents one relation, which is either a raw table that exists in the database, or the result of an operation on some input tables. While traversing the symbol tree, relation tree nodes are created according to the tokens. Relation and attribute names from the symbol tree are used to create references to tables, which are then placed as nodes in the relation tree as well.

A node of the relation tree consists of one or two child nodes, a declaration of the operator type, and a list of expressions. The expressions are interpreted differently, depending on the type of the relation tree node. For example, in a projection node the expressions may refer to column names or vector operations between two columns or between a column and a scalar, whereas in a selection node the expressions may refer to comparisons between two columns or between a column and some value.

The strategic query optimizer acts on the relation tree level. After the relation tree is fully built, the strategic optimizer can rearrange and modify its nodes and the attached expressions, as long as the optimized relation tree would evaluate to the same relation as the unoptimized tree. Relation trees with different structures that evaluate to the same result relation are called equivalent. There are rules on how such restructuring can be done in the general case, which are called *equivalence rules*.

```
SELECT *
FROM (r ON a, b ORDER BY d) ADD (s ON e, f, ORDER BY g, h)
WHERE a = 3 AND ((d = h AND (c > 5 OR b < 9)) OR c = g) AND c < 5;
```

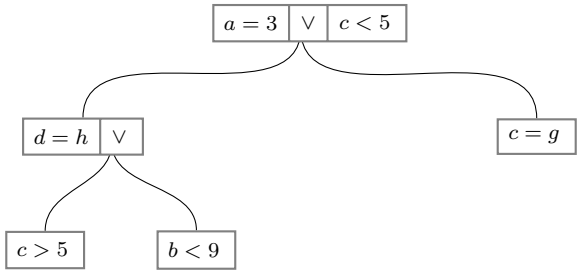

Figure 3.2.: Expression subtree of the selection node

Expression Subtree

An expression is either an attribute of a relation, in which case the name of the relation and the attributes are stored explicitly in the node. An expression can be an atom, which is just a single value, or it can be the name of an attribute. Further, it can be an operator on one or two expressions, which are stored as left and right child nodes. One such operator is the *comparison operator*, which has two child expressions and a type, that defines how the child nodes are compared. The type of a comparison operator can be for example *equality*, *inequality* or *less-than*. A type special type of comparison expression is the logical OR operator, which has as child nodes two lists of expressions.

In the case of selections, the predicates that formulate the conditions for the selection are stored as expressions. In a selection node, the associated list of expressions is interpreted as a conjunction of the expressions. The two lists of expressions that the OR expression node has as children, are again interpreted as conjunctions, which then forms a tree. Figure 3.2 illustrates the expression subtree for the corresponding predicates in the selection.

3.1.3. Statement Tree

The statement tree is another intermediary layer between the relation tree and the MAL-plan. It is a unique feature of MonetDB, which uses the advantages of it being a column-store DBMS. The nodes of the statement tree are called statements, which refer to one or more attributes. If a statement refers to a collection of attributes, it is called a *statement list*. Statement lists are mainly used to produce result statements, that refer to a relation on the relation tree level, and consist of multiple attributes. This also allows us to express operations on attributes, as compared to the relation tree, where we are restricted to operations on relations.

The statement tree provides a convenient way to compose relations by simply choosing the attributes and putting the corresponding statements into a statement list. This comes in handy when we define matrix operations, in particular the matrix addition. In matrix operations,

attributes, that are considered as part of the matrix, are used in matrix related operations, whereas the other attributes are not. By redistributing the attributes into new statement lists, we can form temporary relations, which then can be processed differently. Afterwards the statement lists can be concatenated back together to form a combined result relation.

3.2. Addition

Since matrix operations are only defined on matrices, to perform matrix operations on relations, we have to impose the constraints of matrices on relations during the operation. The constraints for matrices are that the order of columns and rows are significant, and a matrix only consists of numbers.

$$r \ \oplus_{O_1,A_1,O_2,A_2} \ s \tag{3.1}$$

The matrix addition operation expects two matrices as input. Since we want to directly use relations as input, we have to provide some additional parameters to the matrix operation. The parameters provide the matrix operation with information of how each relation must be ordered during the matrix operation, and which attributes are actually part of the matrices and are considered for the addition. The relational algebra operation for the matrix addition is shown in Equation 3.1.

The *application part* of a relation describes which attributes are used as columns in the matrix operation. The application part must consist of one or more attributes of numeric datatype. The order of the attributes is significant. In the relational algebra operation for the matrix operation A_1 refers to the application part of the left relation, and A_2 refers to the application part of the right relation.

r

a	b	c	d
6	1	8	m
5	3	6	i
1	3	6	j
5	2	5	k
2	3	3	l

application part | descriptive part | order

s

e	f	g	h
1	2	4	3
2	4	1	2
3	3	1	9
4	3	2	8
1	4	4	9

application part | descriptive part | order

Figure 3.3.: Relations r and s

Since the order of rows in a matrix is significant, but isn't in a relation, we have to determine some ordering for the relation to use it in a matrix operation. This is done with the order

specification, which must consist of one or more attributes, and it mustn't contain attributes, which were specified in the application part. The order specification for the left relation is provided to the matrix addition operation with the parameter O_1, and the order specification for the right relation is provided through O_2. During the matrix addition operation, first both input relations are ordered according to the specifications, then their application part attributes are used to perform the matrix addition.

Other attributes of the relation, which are not part of the matrix and thus not included in the application part, are called the descriptive part of a relation. The order specification is also considered to be part of the descriptive part. When performing the matrix addition operation with relations, we preserve the descriptive part in the result relation, so that the additional information is not lost.

The schema of the result relation for the matrix addition consists of the application part of the left input relation, and of both input relation's descriptive parts. More precisely, the result relation for a matrix addition consists of the attributes $A_1 \cup R - A_1 \cup S - A_2$, where R is the schema of the left relation and S is the schema of the right relation.

```
(input_relation ON application_part ORDER BY order_specification)
```

Listing 3.2: SQL syntax to define matrix

In SQL, we use the syntax shown in Listing 3.2 when an operation is only defined over matrices. This can be seen as a template, which any matrix operation can use to retrieve the relevant information. The matrix addition operation uses that template twice, as shown in the Query 2.1, since it expects two matrices as input.

r

a	b	c	d
6	1	8	m
5	3	6	i
1	3	6	j
5	2	5	k
2	3	3	l

application part — order

$$\rightarrow \quad R = \begin{pmatrix} 5 & 3 \\ 1 & 3 \\ 5 & 2 \\ 2 & 3 \\ 6 & 1 \end{pmatrix}$$

Figure 3.4.: Relation r and corresponding matrix R

Consider the relations r and s, as illustrated in Figure 3.3, as a running example. The relation r is used as matrix R, as shown in Figure 3.4 and according to the Query 2.1, and relation s is used as matrix S correspondingly. Using the above matrix definition syntax, we implement matrix addition on relations, so that the query in Listing 2.1 can be processed by the DBMS. We expect a result relation with the application part attributes a, and b, as well as both sub relations' descriptive part attributes c, d, g and h, as illustrated in Figure 3.5.

result

a	b	c	d	g	h
7	7	6	i	1	2
4	6	6	j	1	9
9	5	5	k	2	8
3	5	3	l	4	3
7	5	8	m	4	9

Figure 3.5.: Result relation for query 2.1

3.3. Implementation of Addition

In this section we look at the extensions we have made in MonetDB to provide the matrix addition functionality. We consider the different types of trees that are built and used by MonetDB, which we use in the next section when we consider optimizations. The relevant parts of the code can be found in Appendix A.

3.3.1. Symbol Tree

We extend the grammar for the parser generator to recognize our new operation. For that, we add a new token that is associated with it. The token is used to add a new node in the symbol tree, and other information from the query is then attached to it. Figure 3.6 shows the resulting symbol tree that is produced for the given matrix addition query.

```
SELECT * FROM (r ON a, b ORDER BY d) ADD (s ON e, f ORDER BY g, h);
```

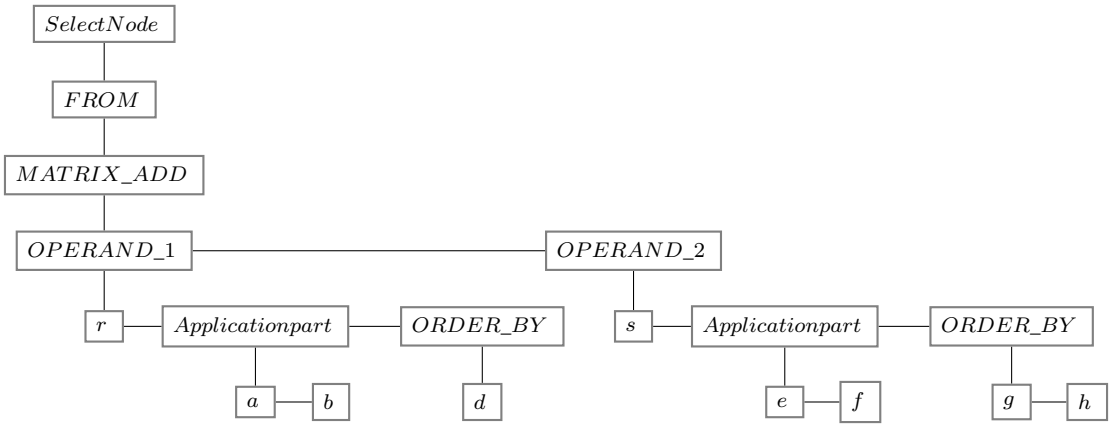

Figure 3.6.: Symbol tree for matrix addition query

In the SQL query, the brackets with the definition of a matrix do not refer to a new datatype or construct. They are part of the `MATRIX_ADD` query and merely provide parameters to

the matrix operation. We decided to use this syntax to provide a consistent way of declaring matrices.

The root node of the symbol tree is a special node, that depends on the type of the query and holds the main tokens of the SQL syntax, like `FROM`, `WHERE`, `GROUP_BY`, etc. In our case, the matrix addition token `MATRIX_ADD` is added as child node to the `FROM` node, which defines the input relation of the query. The remaining information, like the relation names, the application parts and the order specification, is appended to the list that is associated to the node.

3.3.2. Relation Tree

In the relation tree we created a new type of node for the matrix addition. The new type holds all the necessary information that we get from the symbol tree. Formally, the matrix addition node has four parameters as shown in figure 3.7: Left relation's order specification (d) and application part (a, b), right relation's order specification (g, h) and application part (e, f). Note that the result relation also contains descriptive part attributes that are not part of the order specification, for example the attribute c from relation r, which is included in the projection as a demonstration.

```
SELECT * FROM (r ON a, b ORDER BY d) ADD (s ON e, f ORDER BY g, h);
```

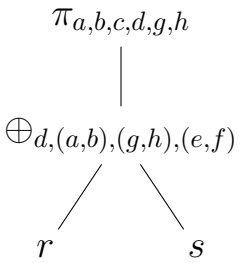

Figure 3.7.: Relation tree for matrix addition query

To store information in the relation tree node, we can use a field of the structure of a relation that can hold a list of expressions. However, we have to store four different lists of expressions, namely the application part attributes and the order specification, for each of the two child relations. We decided to extend the relation tree node with four new fields, that hold lists of expressions. We chose this variant, since it is the less likeliest to break existing functionality and since it does not alter the meaning of the expressions.

Using the names of the relations, that we have been provided with through the symbol tree, we fetch actual relation tree nodes, which are then attached to the corresponding fields of our matrix addition node. Similarly we resolve the attributes for the application parts and the order specifications. This generates expressions that refer to attributes, which we add to the corresponding list of expressions of the relation tree node.

3.3.3. Statement Tree

After the relation tree has been built, it is transformed into the statement tree. For that, a function traverses the relation tree and calls the function that is associated to the type of the relation tree node. The function is provided with the relation tree node, which contains all needed information to create statement tree nodes. In our case, we create a function that expects a matrix-addition node from the relation tree and transforms it into statements. The matrix-addition relation node has references to the child relations, and it contains the information of the application parts and the order specifications.

The process of transforming statements of the input relation to the output statement list is visualized in Figure 3.8, where each node is either a statement after performing some operation, or a statement list that consists of multiple statements. That statement tree is equivalent to the relation tree shown in Figure 3.7.

We first retrieve the lists of statements for the left and right child relations. By using the expressions that describe the application parts, provided by the relation tree node, we split the statements into four lists, application part and descriptive part, for each child relation. This is depicted in Figure 3.8 as a phase called Split.

We then sort the attributes of the order specification, and save the resulting ordered OIDs in a statement. This is depicted as node o_d and $o_{g,h}$ in Figure 3.8, and we refer to the ordering operation as $O()$ there. Using those two statements, we align the OIDs of the left and right relation respectively. This is done by iterating over the OIDs of the ordered order specification and fetching the values of the target statement. That procedure is internally called *projection* and we refer to the operation as $P()$. Note that this is a BAT algebra operation and does not refer to the projection in the relational algebra. Ordering the order specification and aligning the OIDs of a BATs using projection is illustrated in Figure 3.9 with the attributes d and a of relation r.

The next step is the preparation of the result statement. The descriptive part statements are no longer modified, so they are inserted into the result list of statements. For the matrix addition, we created a new type of statement. It expects two child statements and produces one column as result. We iterate over the two application part lists of statements simultaneously, generate the matrix addition statements and append them to the result list. We refer to this operation as $Add()$, in Figure 3.8. The matrix addition statements are in fact translated to the BAT algebra operation that performs vector addition on two BATs.

The statements in the statement tree are translated into the MAL-plan as soon as the statement tree is fully built. However, here MonetDB uses the aforementioned lazy evaluation. Starting at the result statement of the whole statement tree, only those statements that are required for the creation of the result BATs are translated. If we for example added a relational projection on a in the query shown in Figure 3.8, the statement $\hat{b}$ is created anyways in the statement tree, however it is never included in the translated MAL-plan, since it is not required in the final relation and the corresponding statement.

```
SELECT * FROM (r ON a, b ORDER BY d) ADD (s ON e, f ORDER BY g, h);
```

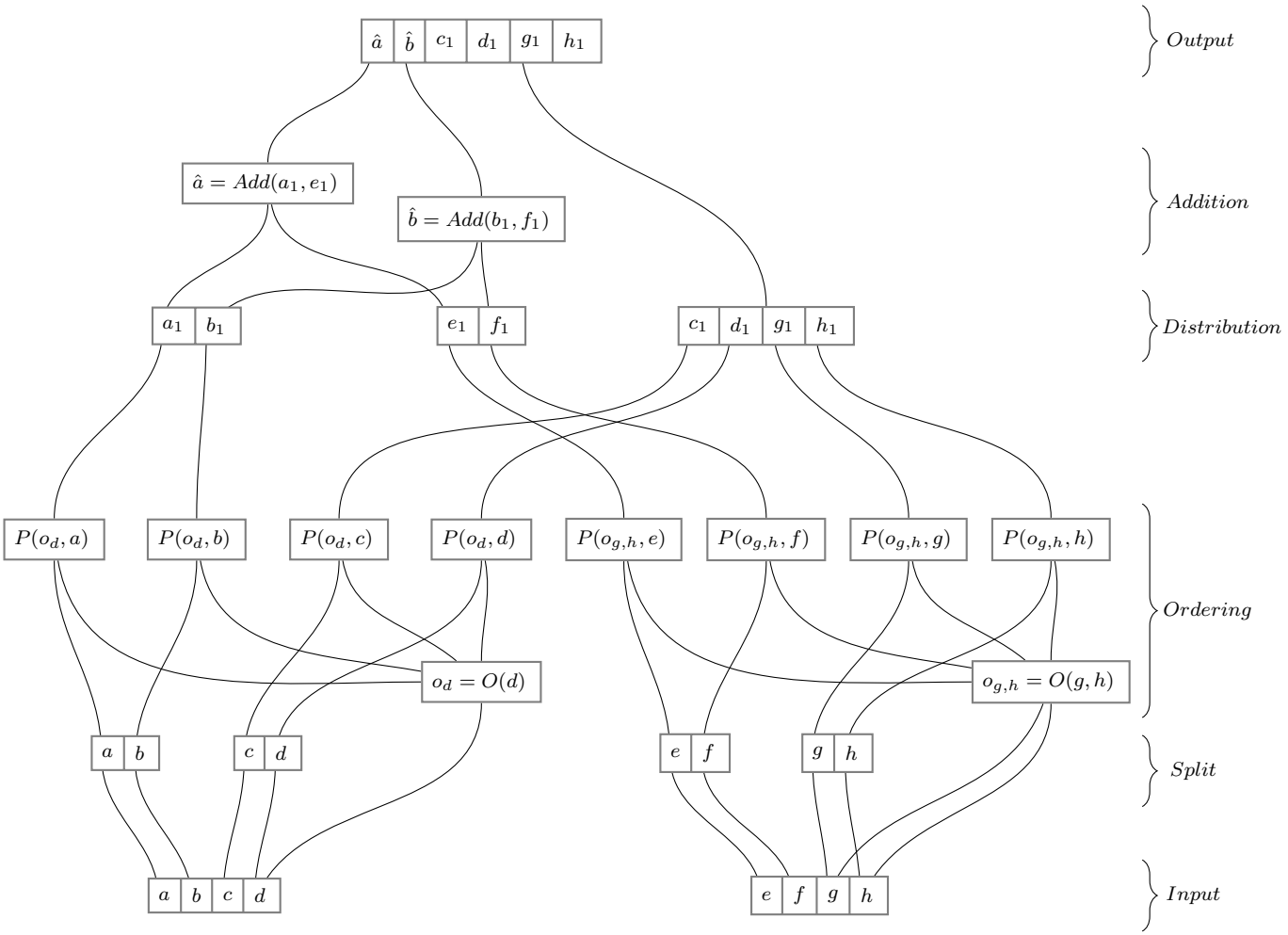

Figure 3.8.: Statement tree for matrix addition

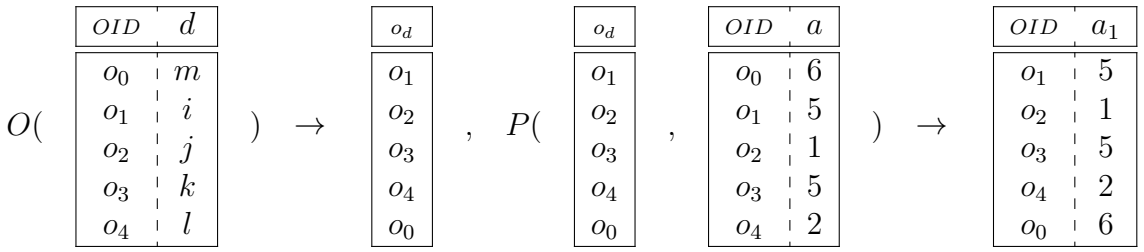

Figure 3.9.: Ordering and projection

3.4. Optimization

Query optimization is usually done by reordering the relational tree, that has been built based on the input query. The condition for such reordering is that the result relation for the optimized relation tree must be the same as for the original relation tree. Relation trees can be written as relational algebra expressions, which allows us to formulate equivalence rules more easily. We focus on optimizing selections in mixed queries. Projections, for example, are already optimized by MonetDB through the lazy evaluation it does during the transformation of the statement tree into the MAL-plan.

```
SELECT * FROM r NATURAL JOIN s WHERE a = 1;
```

$$\sigma_{a=1}(r \bowtie s) \quad \equiv \quad \sigma_{a=1}(r) \bowtie s$$

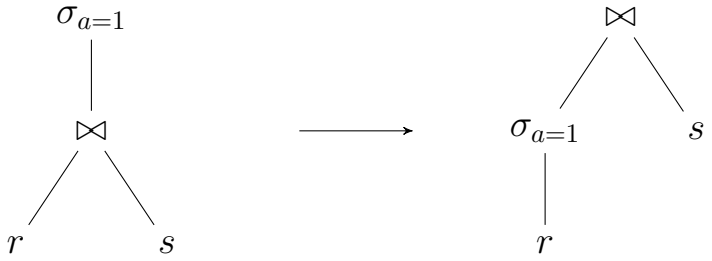

Figure 3.10.: Optimization of a join in a relation tree

Optimizing joins can be done by reordering the tree as shown in Figure 3.10 (where r and s are the relations of the running example), such that the selection is performed before the join, so that the intermediary relation becomes smaller. It is easy to see that such an optimization is not possible in the case of matrix addition. According to the constraints of matrix addition, we expect two input relations that have the same amount of tuples. An optimization as shown for the join operation will possibly reduce the numbers of tuples in one relation, which leads to a broken constraint.

For the optimization of matrix operations, we have to consider different approaches depending on the constraints of the operation. Consider the matrix multiplication operation for example,

where one constraint is that the number of columns in the left matrix must be equal to the number of rows in the right matrix. Due to that, we can optimize a matrix multiplication query by performing selections, that only involve descriptive part attributes of the left relation, before the matrix operation. However, we cannot optimize selections that involves attributes of the right relation, since that would break the constraints of the matrix multiplication operation.

In the following part, we will only consider selections on descriptive part attributes of relations. To perform selections on an application part attribute, we have to perform the calculation beforehand, which leaves no room for optimizations regarding that attribute. It is possible to first perform the additions for those application part attributes, which are referenced in the selection, then preform the selection and finally perform the remaining additions. A model of how that optimization looks like is shown in Chapter 5, however, in this thesis we focus on optimizing selections with predicates that only refer to descriptive part attributes.

3.4.1. Optimization of Matrix Addition

In the case of matrix addition (but also matrix subtraction and element-wise multiplication), the constraint is that both input matrices must have the same dimensions. Thus, for optimization, if we want to remove rows before the calculation, we must remove them correspondingly in both input matrices.

```
SELECT *
FROM (r ON a, b ORDER BY d) ADD (s ON e, f ORDER BY g, h)
WHERE g > 1;
```

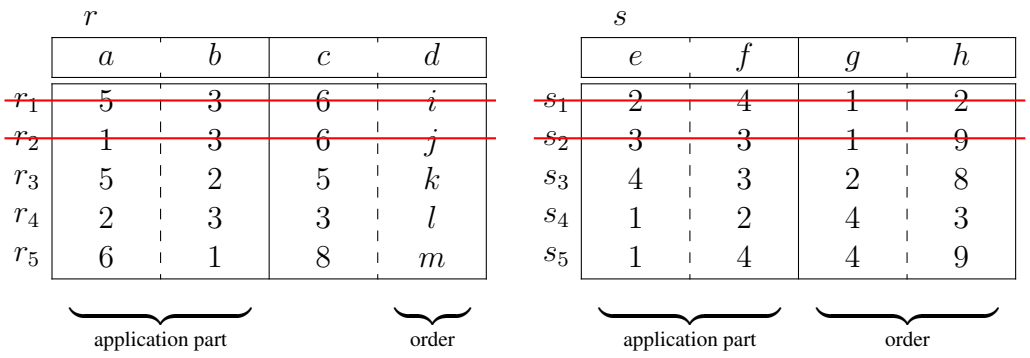

Figure 3.11.: Ordered relations r and s, with selection

The selection in the query of Figure 3.11 involves only attributes of the relation s. It removes two records of the right relation. We must remove the corresponding two rows of the ordered left relation in order to preserve the correctness of the matrix addition, as illustrated in Figure 3.11. Concretely, if we remove the tuple s_1 in the right relation, we have to remove the corresponding tuple r_1 in the left relation too, and to remove the tuple s_2, we have to remove the tuple r_2 as well.

```
SELECT * FROM (r ON a, b ORDER BY d) ADD (s ON e, f ORDER BY g, h)
WHERE g > 1;
```

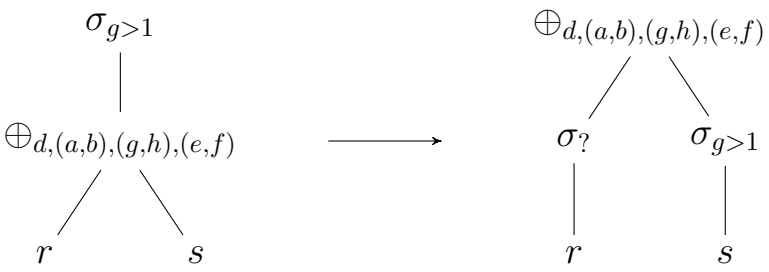

Figure 3.12.: Naive optimization of matrix addition in the relation tree

Figure 3.12 illustrates the naive approach for the optimization of matrix addition, which would be to push the selection down below the matrix addition operation, similar to what we do in the optimization of joins in Figure 3.10. One problem that arises is how to formulate the selection on relation r in a way, that it removes the tuples corresponding to the selection on the relation s. Another problem is that this approach assumes an ordering of the relations r and s, which we cannot guarantee. And such an approach would also hinder us from optimizing selections, that involve attributes from both sub relations, as shown in the Query 3.3.

```
1  SELECT *
2  FROM (r ON a, b ORDER BY d) ADD (s ON e, f ORDER BY g, h)
3  WHERE c = g;
```

Listing 3.3: Selection that refer to both sub relations' attributes

A simpler approach is to perform the optimization itself on a lower layer, in the statement tree. Consider Figure 3.13, which conceptually illustrates how our optimization is done. We make a statement list that refers to all attributes of both input relations. This forms a new relation, on which we perform the selection. After performing the selection, we split the statement list back into two statement lists, forming two relations that represent r and s again.

From the perspective of the relation tree, this optimization cannot be performed before or after the matrix addition operation. It can only be done during the matrix addition operation, and thus we add an optional parameter θ to the matrix addition relation. The new relational algebra notation for the matrix addition is shown in Equation 3.2. The predicate θ is used to denote the selection during the matrix addition operation as explained.

$$r \ \oplus^{\theta}_{O_1,A_1,O_2,A_2} \ s \tag{3.2}$$

The predicate θ is not assigned to the matrix addition node directly. When creating the relation tree, we will still have a selection node on top of the matrix addition node. It is the query

```
SELECT *
FROM (r ON a, b ORDER BY d) ADD (s ON e, f ORDER BY g, h)
WHERE g > 1;
```

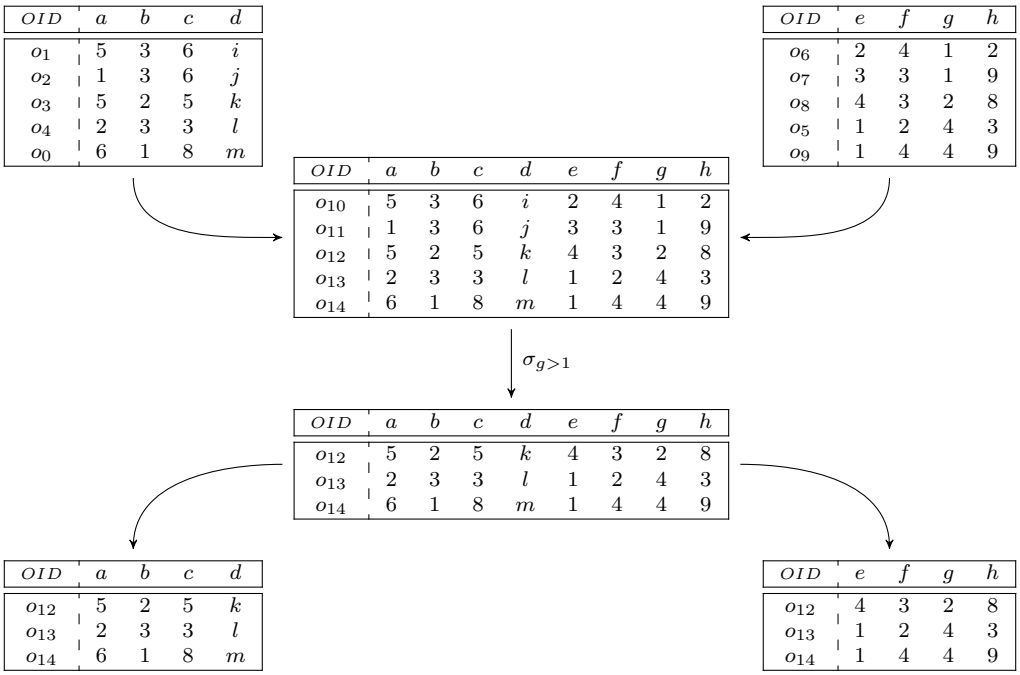

Figure 3.13.: Optimization in Statement Tree, illustrating BATs

optimizer's job to identify predicates, that can be evaluated during the matrix addition operation, and move them to the matrix addition node accordingly. The following section explains, which kind of predicates are considered for the optimization.

In the statement tree layer, the additional steps are only performed if the relation tree node for the matrix operation has predicates attached to it. This approach is explained in Section 3.5 for the matrix addition operation.

3.4.2. Equivalence Rules

We exclusively consider predicates of a selection that only refer to descriptive part attributes for optimization, that is to move the predicate from the selection node to the matrix addition node, since predicates that contain application part attributes cannot be checked until the matrix operation has been performed.

$$\sigma_{\theta_1 \wedge \theta_2}(r) \quad \equiv \quad \sigma_{\theta_1}(\sigma_{\theta_2}(r)) \quad \equiv \quad \sigma_{\theta_2}(\sigma_{\theta_1}(r)) \tag{3.3}$$

For a selection, conjunctions of predicates can be separated into multiple consequent individual selections, as shown in Equation 3.3. Also, selections are commutative, the order of consequent selections does not matter. Both of those rules do not apply to disjunctions. Using that and the rules of the first-order predicate logic, we can formulate equivalence rules for our optimization.

For the following cases, consider θ_D as a predicate that doesn't refer to any application part attributes, and θ_A as a predicate that refers to at least one application part attribute. Also note that the subscripts of the matrix addition operator, that define the order specifications and the application parts, are omitted for the sake of clarity.

In case we have two predicates in a conjunction, where both predicates refer to only descriptive part attributes, can move the conjunction with both predicates to the matrix addition node. Note that if all predicates of a selection were moved to the matrix addition operation, the selection node becomes redundant and can be removed. The equivalence rule is shown in Equation 3.4.

$$\sigma_{\theta_{D1} \wedge \theta_{D2}}(r \ \oplus \ s) \quad \equiv \quad r \ \oplus^{\theta_{D1} \wedge \theta_{D2}} \ s \tag{3.4}$$

If two predicates in a disjunction only refer to descriptive part attributes, we can move the disjunction with both predicates to the matrix addition node, as shown in Equation 3.5.

$$\sigma_{\theta_{D1} \vee \theta_{D2}}(r \ \oplus \ s) \quad \equiv \quad r \ \oplus^{\theta_{D1} \vee \theta_{D2}} \ s \tag{3.5}$$

Since disjunctions cannot be split like conjunctions, we can only consider moving the whole disjunction. In case one of the predicates in a disjunction refers to an application part attribute, for example $\sigma_{\theta_D \vee \theta_A}(r \oplus s)$, we cannot optimize the selection of the disjunction.

In the case of a selection with a conjunctions of two predicates, where one predicate refers to only descriptive part attributes, whereas the other predicate refers to an application part attribute, we can partially optimize the query. Since conjunctions can be split up into consecutive selections, we move the predicate that only refers to descriptive part attributes to the matrix addition node, while we leave the predicate that refers to application part attributes in the selection node. This is illustrated in Equation 3.6

$$\sigma_{\theta_A \wedge \theta_D}(r \oplus s) \quad \equiv \quad \sigma_{\theta_A}(r \oplus^{\theta_D} s) \tag{3.6}$$

Consider the expression subtree in Figure 3.2 as a concrete example, and Equation 3.7 as the formal notation for the optimization. Remember that attributes a, b, e and f are application part attributes, and the remaining attributes belong to the descriptive part. We cannot push the expression $a = 3$ down to the matrix addition node, since a is an attribute of the application part. The disjunction cannot be split up and due to the expression $b < 9$, which refers to an application part attribute, it cannot be pushed down to the matrix addition node. The expression $c < 5$ can be pushed down to the matrix addition node, since it doesn't refer to application part attributes.

$$\begin{aligned} & \sigma_{a=3 \,\wedge\, ((d=h \,\wedge\, (c>5 \,\vee\, b<9)) \,\vee\, c=g) \,\wedge\, c<5}(r \oplus s) \\ \equiv \quad & \sigma_{a=3 \,\wedge\, ((d=h \,\wedge\, (c>5 \,\vee\, b<9)) \,\vee\, c=g)}(r \oplus^{c<5} s) \end{aligned} \tag{3.7}$$

3.4.3. Trade-off

The proposed optimization has a trade-off, which, depending on the input data and the predicate of the selection, can even lead to an increased query time compared to the unoptimized form of the matrix addition. For the explanation, let m be the number of application part attributes, that is, the number of columns of the matrices. Let $|D_1|$ be the number of descriptive part attributes of the left input relation, and $|D_2|$ be the number of descriptive part attributes of the right input relation. Let n be the total number of tuples, which is equal for both input relations and the output relation. And let s be the selectivity, i.e. the number of tuples, that are not removed by the selection, divided by the total number of tuples.

If we apply the optimization, the selection is performed after the ordering of the input relations, but before the addition operations. Since we haven't performed the addition yet, the number of operations we perform for the selection is shown in Equation 3.8.

$$\sigma_{\text{optimized}} \quad = \quad (|D_1| + |D_2| + 2m) \cdot n \tag{3.8}$$

The number of tuples in the input relations for the matrix additions is now $s \cdot n$, where $0 \leq s \leq 1$, and thus the input relations for the addition are potentially smaller now. The number of additions we perform is displayed in Equation 3.9.

$$\oplus_{\text{optimized}} \quad = \quad (n \cdot s) \cdot m \tag{3.9}$$

If we do not consider our optimization, we first perform the addition on the whole input relations, thus we tend to perform more additions. Equation 3.10 presents the number of additions we perform in this case.

$$\oplus_{\text{non-optimized}} \quad = \quad n \cdot m \tag{3.10}$$

The selection is then performed on the result relation of the matrix addition operation. The number of operations we perform for the selection is shown in Equation 3.11.

$$\sigma_{\text{non-optimized}} \quad = \quad (|D_1| + |D_2| + m) \cdot n \tag{3.11}$$

In the experimental evaluation in Chapter 4, we want to see if this trade-off has a real, measurable effect. If there is an effect, we want to determine up to which point it makes sense to use this optimization.

3.5. Implementation of Optimization

In this section we look at the extensions we have made in MonetDB to implement optimization for the matrix addition operation. We look at the changes and extensions that are made on the different types of trees of MonetDB's query processing. The relevant parts of the code can be found in Appendix A.

Figure 3.14 depicts a matrix addition query and the according statement tree, including the proposed optimization, using concrete BATs. For the sake of illustration, we use smaller input relations than the relations we use as running example. The corresponding query shows how the input relations are formed.

3.5.1. Relation Tree

On the relation tree layer, we create a new function for the optimizer. The function is executed if the optimizer finds in the relation tree a selection node, which has as child node a matrix addition node.

The expressions that are attached to a selection node in the relation tree represent the predicate, that is used for the selection. We iterate over the expressions to identify the ones that can be

```
SELECT * FROM
    ((SELECT a, d FROM r) AS r1 ON a ORDER BY d)
  ADD
    ((SELECT e, g, h FROM s) AS s1 ON e ORDER BY g, h)
WHERE g > 1;
```

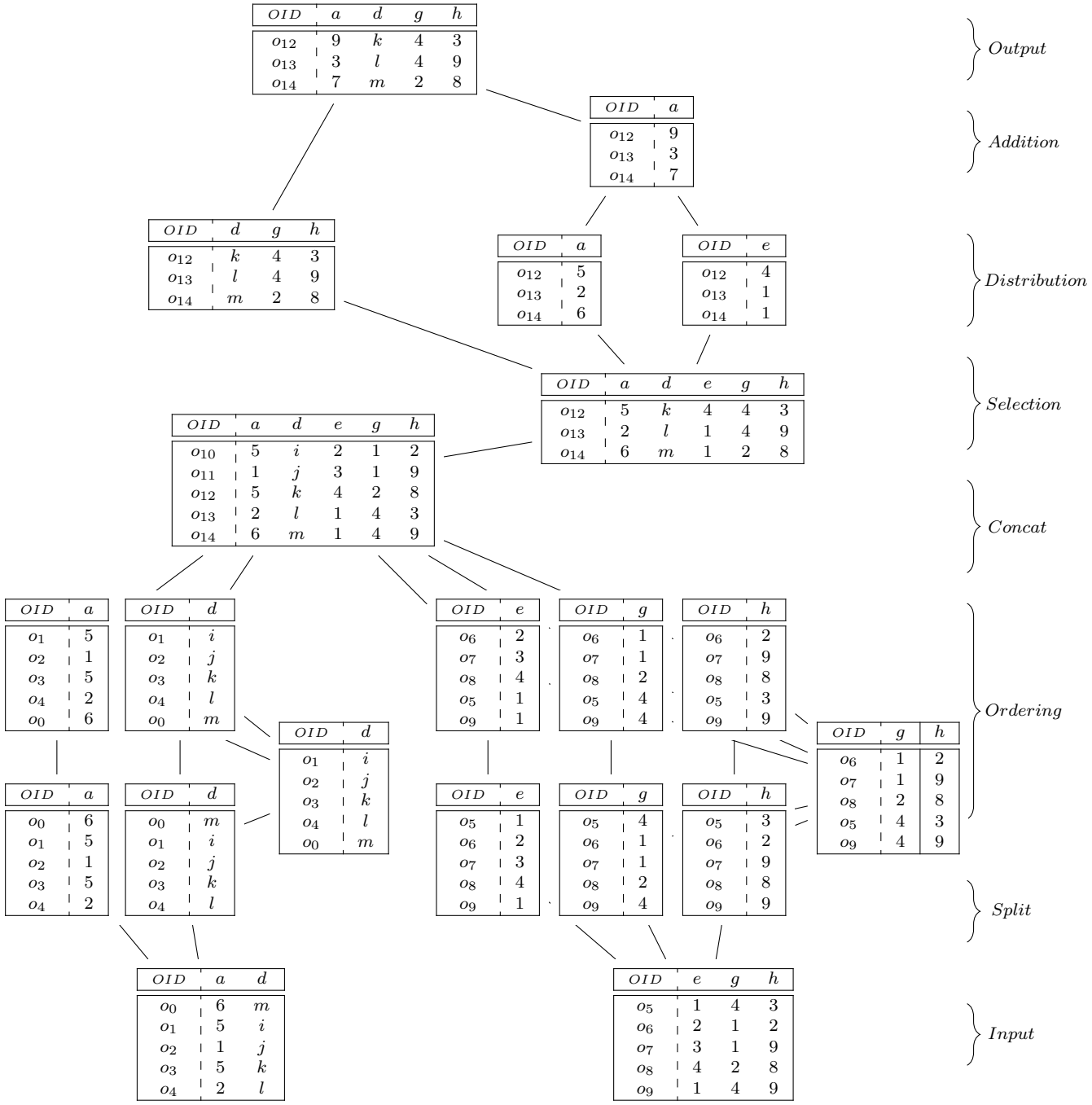

Figure 3.14.: Optimization in Statement Tree, illustrating BATs

```
SELECT * FROM (r ON a, b ORDER BY d) ADD (s ON e, f ORDER BY g, h)
WHERE g > 1 AND a > 4;
```

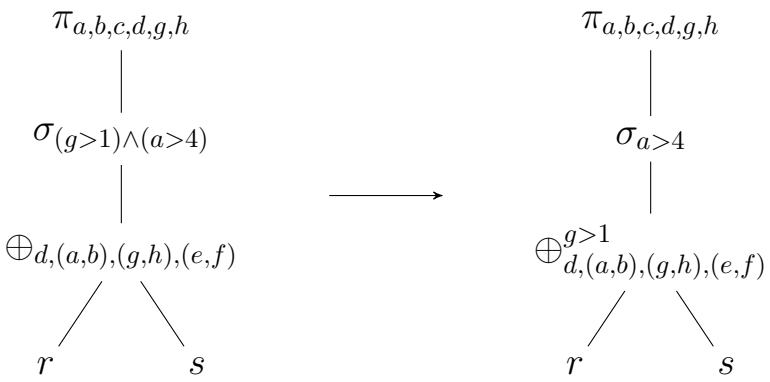

Figure 3.15.: Transformation of the relation tree for optimization

pushed down to the matrix addition, as described in Section 3.4.2. Pushing down an expression concretely means removing an expression from the parent relation and adding it to its child relation. The movement of expressions is illustrated in Figure 3.15. If the selection node has no expressions left, it is removed by the query optimizer.

To check if an expression can be pushed down to the matrix addition, we use two helper functions. The first is meant to check one single expression, that is not an OR expression. It takes as input one expression and a list of application part attributes, and returns false if the input expression refers to one of the attributes.

The second function is recursive, expects as input an OR expression and a list of application part attributes. It traverses the expression tree by starting at the OR expression and walking over both child lists of expression. If there are further OR expressions, those are checked too recursively. On each expression in the expression tree, we use the first helper function to check if the expression can be optimized. If any expression in the whole expression cannot be optimized, the whole OR expression is considered to be unoptimizable.

3.5.2. Statement Tree

In the statement tree, we proceed as described in Section 3.3.3, until we created the ordered statements for the input BATs. If the input relations has any selection expressions attached to it, we perform two additional steps to enable the selection.

First we concatenate descriptive part and application part attributes by creating a new statement list and appending statements, that refer to said attributes. This is not a cross join or a join on some attribute, but rather a concatenation of BATs on a lower level than where relational operations are formulated.

Now we create a selection statement on the temporary relation, represented by the statement list. For that, we use a slightly modified version of the existing function, that is used to create a selection statement based on a selection node from the relation tree. The function returns a statement list, which represents the result relation that is produced by the selection.

The result statement list is split into lists of statements again, namely two lists for the application part attributes and one for the descriptive part attributes. Those lists are then processed as described in Section 3.3.3. The resulting statement tree after the optimization is illustrated in Figure 3.16.

```
SELECT * FROM (r ON a, b ORDER BY d) ADD (s ON e, f ORDER BY g, h)
WHERE g > 1;
```

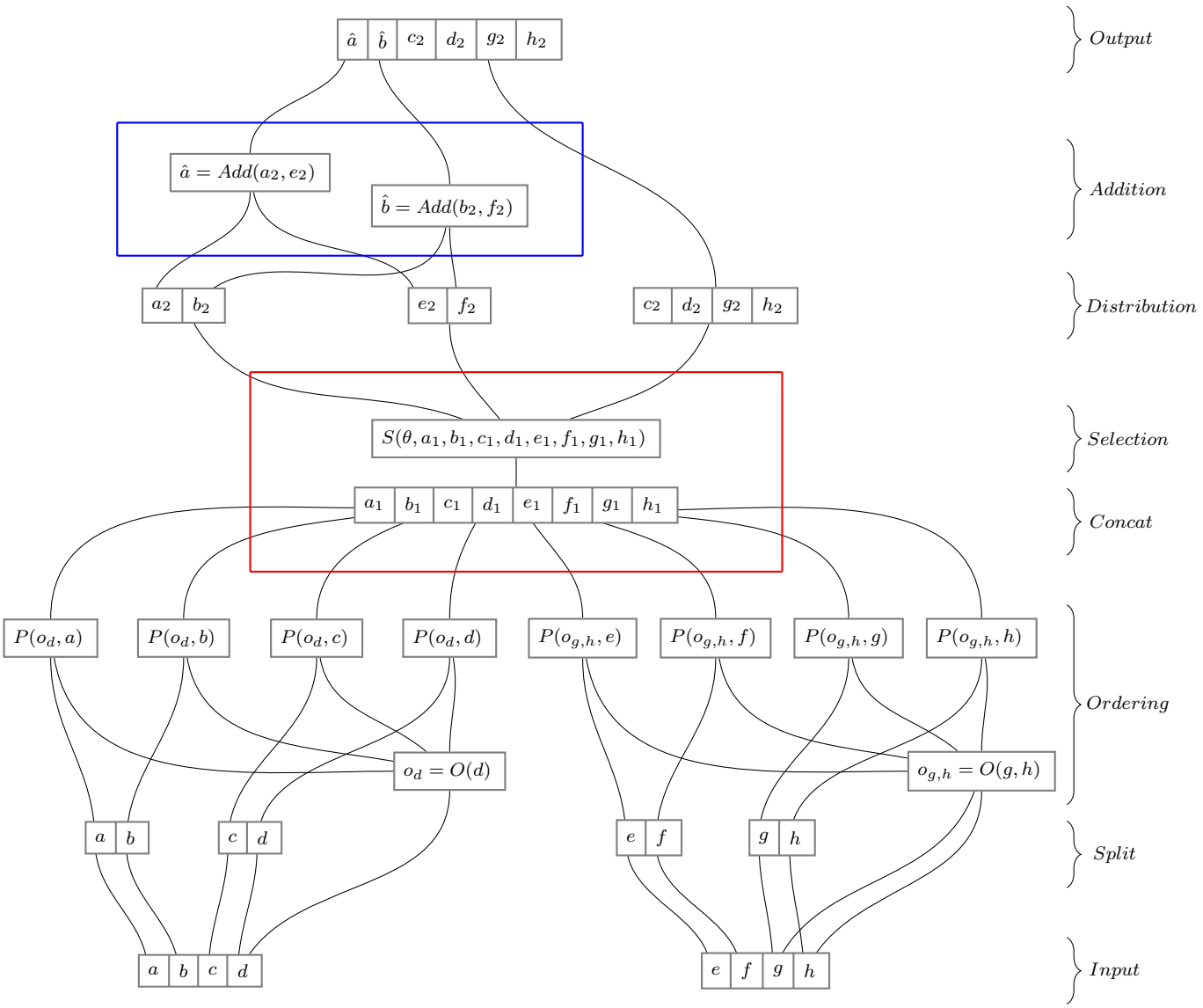

Figure 3.16.: Optimized statement tree, selection done before addition

4. Experimental Evaluation

In this chapter we conduct the experimental evaluation and we discuss the results. That includes the description of the setup we used for the tests, an explanation of the test cases and a description of how the corresponding data sets are generated. The experimental evaluation aims to analyze how the optimization we discussed in Section 3.4 affects the runtime of different queries with different properties. In particular, we want to see if the trade-off we discussed in Section 3.4.3 has a measurable effect, and if that trade-off makes our optimization counterproductive.

4.1. Setup

We carry out the experiments on a Dell XPS 13, that was running Arch Linux (with Linux 4.18.5). The system used an Intel Core i7-7560U CPU @ 2.40GHz, 16GB of LPDDR3 RAM @ 1866MHz and 512GB PCI SSD as hard drive.

For the experimental evaluation, we extended the matrix addition operation with a keyword `NOOPTIMIZE`, which works as a flag to turn off the optimization. It is a Boolean variable attached to the relation tree, which is checked by the optimization function. In case the flag is set, the optimization is cancelled and no expressions are moved from the selection node of the relation tree to the matrix addition node.

The MonetDB client was also slightly modified, so that the client drops the output, instead of writing them to the terminal or to a file. This change causes the query evaluation time to be printed as soon as the server processed the query and starts producing the output. This was done to reduce the slack time that arises due to the generation of the output.

4.2. Test Cases

With the experimental evaluation, we want to determine how the parameters for the matrix addition and the selection influence the query time, with and without our optimization.

Matrix addition has a time complexity of $\mathcal{O}(n \cdot m)$, where the input matrices have n rows and m columns. To perform the matrix addition operation on relations, the input relations must be sorted beforehand. Sorting has a complexity of $\mathcal{O}(n \log n)$, and recall that even if we use the optimization and perform the selection before the addition, the sorting must be performed first.

Selection has a complexity of $\mathcal{O}(n)$, if the attributes referenced in the predicates are unsorted. If the number of columns m is low, the complexity of the addition operation drops to $\mathcal{O}(n)$, in which case, performing the selection before the addition operations might be redundant. To test this, we conduct an experiment that varies the number of columns, that is, the number of application part attributes.

To test the trade-off described in Section 3.4.3, we conduct experiments where we vary the selectivity of the selection. In one experiment, we test a selection on an unordered attribute, in another experiment we test a selection on an ordered attribute, to check if that has any effect on the performance. The complexity of a selection decreases to $\mathcal{O}(\log n)$ if its predicates only refer to ordered attributes decreases, whereas the complexity of the addition is $\mathcal{O}(n)$ if m is small. This suggests that performing the selection before the additions is sensible even if the number of columns m is low, in case the selection is done on ordered attributes.

Our optimizations only affect the addition operations. In order to verify that our optimization does not influence the query time if we vary the number of descriptive part attributes, we conduct an experiment for that case too.

As mentioned, in case the number of columns, that is, the number of application part attributes, is low, the complexity of our matrix addition operation degenerates to the complexity of sorting. To test this, and see if and how a selection affects the performance in this case, we conduct another experiment that varies the number of tuples.

4.3. Data Sets

The test data sets are created using a small program written in C, and consist of random numbers equally distributed between 0 and 999. Each test is executed 4 times, where the first run is not counted. We do this due to the fact that the query plan is cached for consequent runs of the same query. However, difference in time it takes to produce the unoptimized and optimized execution plans is not measurable ($< 0.1s$). The mean values of the 3 measurements is used as data points in the plots.

4.4. Results

In this selection, we look at the results of the experiments. We consider the four different experiments mentioned in Section 4.2, where in each test we vary one parameter of the selection or the matrix addition operation. The detailed setup of each test, the results as plots, as well as the evaluation and explanations are provided in each subsection. The detailed results for the tests can be found in Appendix C.

4.4.1. Selectivity

In this experiment, we fix all parameters related to the matrix addition and vary the selectivity of the selection. Selectivity is defined as number of tuples, that are not filtered by the selection, divided by the total number of tuples. A selection that filters out many tuples has a low selectivity.

In this experiment, we expect that the optimized variant is faster if the selectivity is low, but slower than the unoptimized variant if the selectivity is high. We expect this result mainly due to the trade-off mentioned in Section 3.4.3. We vary the selectivity, while the remaining parameters are fixed in the following way:

- 100 application part attributes
- 1 descriptive part attribute
- 1 order specification
- $10M$ tuples

Since the data set we use for this experiment contains random numbers equally distributed between 0 and 999, the predicate for this experiment is formulated as $< p \cdot 1000$, where p is the selectivity we are testing.

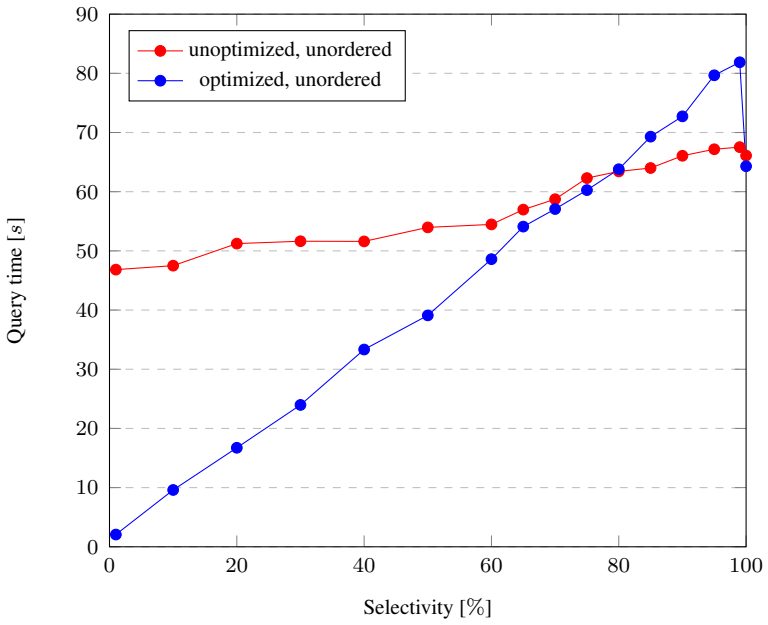

Figure 4.1.: Query time depending on selectivity, selection on unordered attribute

Looking at the result, we can see that at approximately 70% and higher selectivity, the optimized matrix addition operation is slower than the unoptimized version, as expected. However, at 100% selectivity, both variants are faster than at a selectivity close to 100%, where the

decrease in time is significantly high in the optimized version of the operation. This behavior can probably be tied to how the MAL operation *projection* is implemented. My hypothesis is that the projections are omitted if it notices that all OIDs are included. Note that the MAL-plan for both queries, 99% and 100% selectivity, are identical in terms of MAL instructions. The MAL-plans for two equivalent queries, can be found in Appendix B.1.

In the second experiment, we want to see if there is any difference in the query time in case the selection is done on an ordered attribute, thus on an attribute, that is part of the order specification of one of the input matrices. For this experiment, the expectation was similar, since the trade-off explained in Section 3.4.3 is still valid.

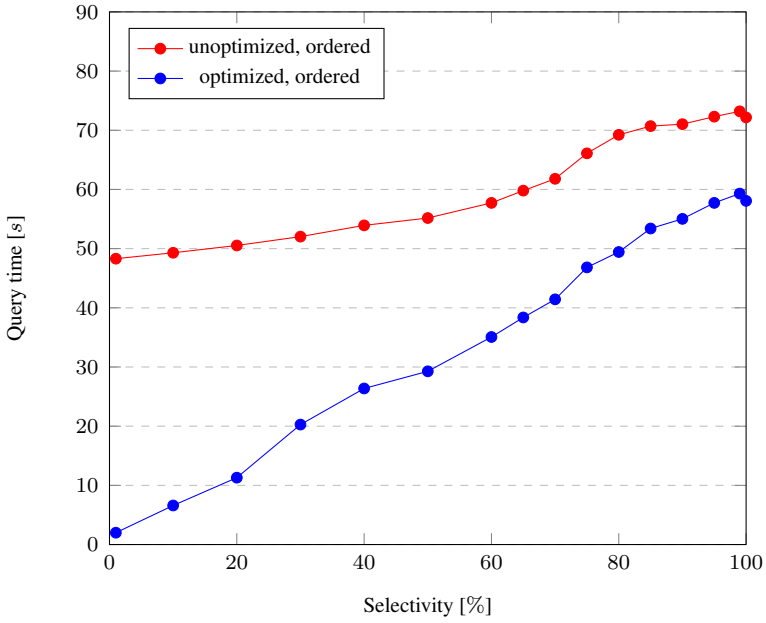

Figure 4.2.: Query time depending on selectivity, selection on ordered attribute

In contrast to the previous experiment, here the optimized variant of the matrix addition is always faster than the unoptimized variant. Looking at the MAL-plan reveals that MonetDB does further optimization by combining two steps, namely the two calls to the BAT algebra operation *projection* by the ordering and the selection operations, in the optimized variant. This is an optimization that MonetDB can't do in the unoptimized case, where ordering and selection is separated by the matrix addition operation. Two MAL-plans to compare the selection on and unordered and an ordered attribute are included in Appendix B.2. According to the MAL-plans in Appendix B.3, this optimization is not done for the unoptimized variant of the experiment.

We can conclude, that our optimization is sensible whenever the selectivity of a selection is lower than 70%. The optimizer could be extended to use histograms to determine if the selectivity is going to be below 70%, and then only apply our optimization in that case. However, in case the predicates only refer to ordered attributes, we can state that our optimization is always efficient.

4.4.2. Application Part

In this experiment, we fix all parameters besides the number of attributes in the application part. We expect the relationship between query time and number of application part attributes to be linear, since the matrix addition query is of linear complexity regarding the number of columns, for a fixed number of rows.

We vary the number of application part attributes from 1 to 1000, in steps of 50. In the first experiment for the selection, we set a selectivity of 50%. The remaining parameters were fixed in the following way:

- 1 descriptive part attribute
- 1 order specification
- $1M$ tuples
- selection on unordered attribute

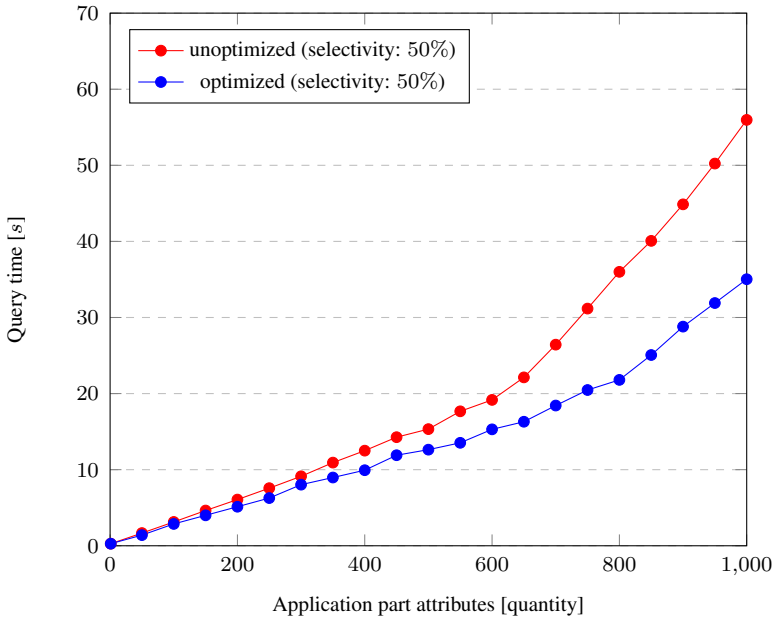

Figure 4.3.: Query time depending on number of application part attributes, 50% selectivity

Since the selectivity is the most important variable, that affects how well the optimization works, we performed additional tests with two more values for selectivity, 10% to cover low selectivity and 90% to cover high selectivity.

As we can see, there is a kink visible in the plots. We suspect that this can be associated with a feature of MonetDB, which is to use memory-mapped files in case the main memory runs out of space [2]. This phenomenon increases the query time of the unoptimized query so much, that in Figure 4.5 the optimized variant becomes faster, even though the selectivity is 90%.

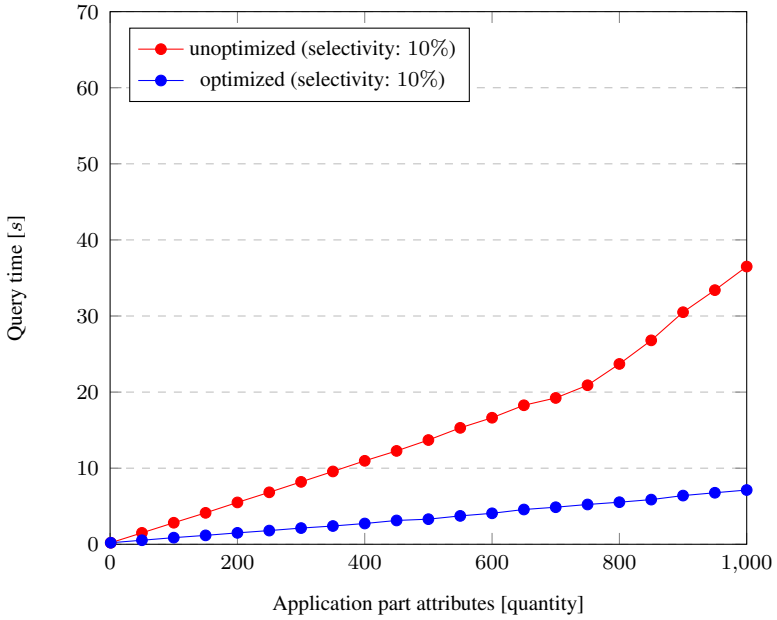

Figure 4.4.: Query time depending on number of application part attributes, 10% selectivity

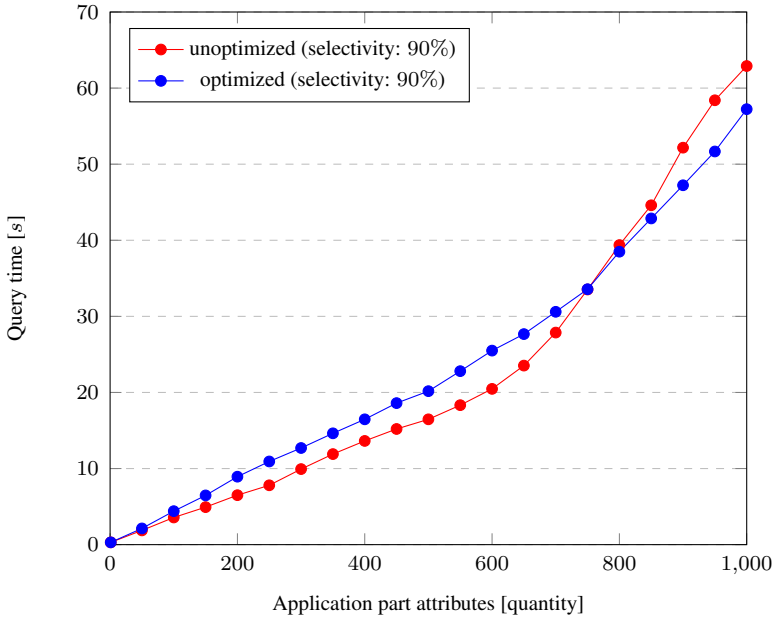

Figure 4.5.: Query time depending on number of application part attributes, 90% selectivity

We can conclude that if we increase the number of application part attributes, our optimization becomes even more valuable, assuming the selectivity is below 70%. In Figure 4.5, we can't say with certainty if our optimization is responsible for the increased performance due to the aforementioned phenomenon.

4.4.3. Descriptive Part

In this experiment, we fix all parameters besides the number of attributes in the descriptive part. We do not expect any difference between the optimized and the unoptimized variants. If any difference occurred, it should be in cases where the number of descriptive part attributes is lower than the number of application part attributes.

We vary the number of descriptive part attributes, that are not part of the order specification, from 1 to 1000, in steps of 50. Note that we count the number of descriptive part attributes that are included in the result relation of the matrix addition operation. The remaining parameters were fixed in the following way:

- 10 application part attributes
- 1 order specification
- $1M$ tuples
- 50% selectivity

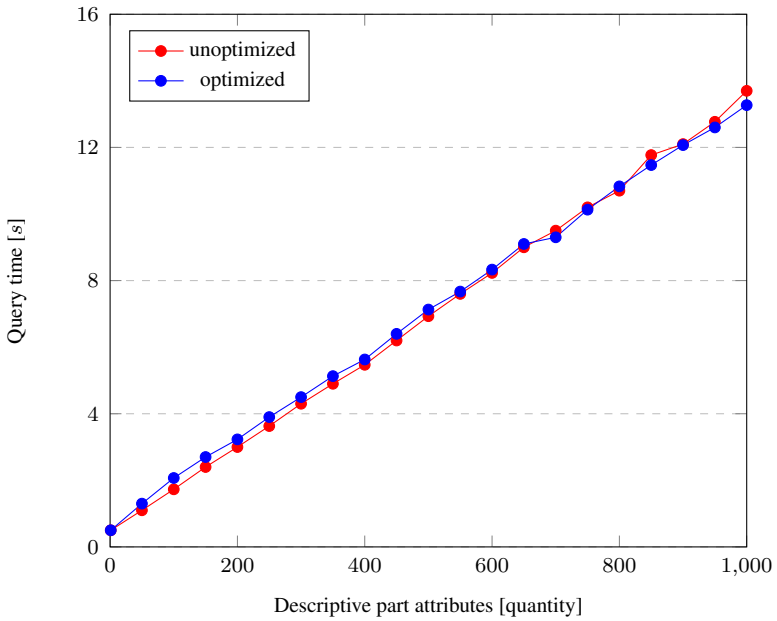

Figure 4.6.: Query time depending on number of descriptive part attributes

The result shows that there is no significant difference between the optimized and the unoptimized variants when we change the number of descriptive part attributes. This is to be expected, since the effect of the trade-off only becomes visible if the number of application part attributes is high and the selectivity is high as well. A quick test for 10% and 90% selectivity shows that there is no significant difference there either, between the optimized and unoptimized variants.

4.4.4. Tuples

In this experiment, we fix all parameters besides the number of tuples in the input relations. Both input relations have the same number of tuples, as required by the matrix addition operation. We expect the optimized variant to be faster than the unoptimized variant, since the number of addition operations increases as we increase the number of tuples.

We vary the number of tuples from 1 million to 80 million. The remaining parameters are fixed in the following way:

- 10 application part attributes
- 1 descriptive part attribute
- 1 order specification
- 50% selectivity

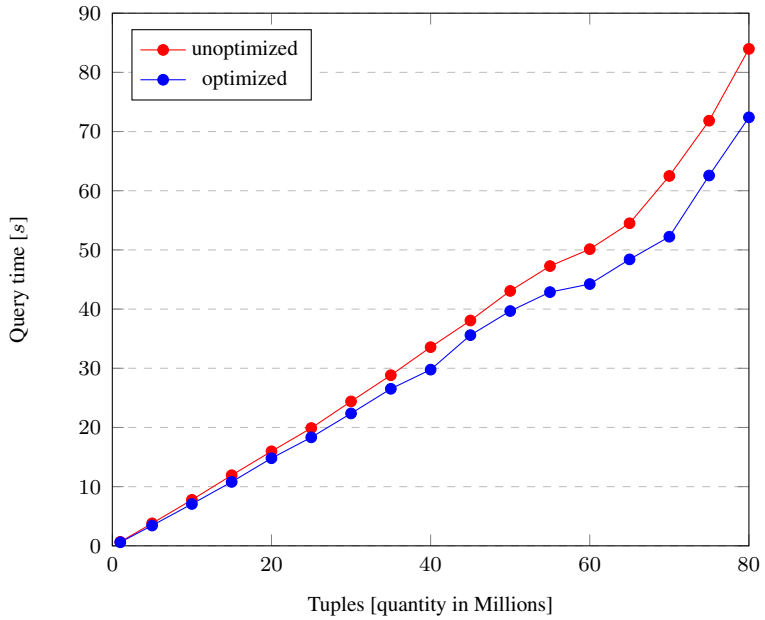

Figure 4.7.: Query time depending on number of tuples

We can see that the optimized variant is slightly faster than the unoptimized case. Though, both query times are close together, since we only used 10 application part attributes and a selectivity of 50%. A quick test for 10% selectivity shows that the optimization increases the performance significantly, whereas for 90% selectivity, the optimized variant is slower than the unoptimized variant. The shape of the curve in Figure 4.7 slightly resembles $n \log n$. This is caused by the sorting, which, since m is very small, constitutes the biggest factor to the asymptotic complexity.

5. Summary and Future Work

The goal of this thesis was to extend a DBMS with a matrix addition operation, that works on relations. Then we investigated on the optimization of queries, that contain both matrix and relational operations, and extend the query optimizer of the DBMS to implement that optimization.

After defining the problem, we gave an overview of the architecture of MonetDB, that is relevant for the processing and optimization of queries. We then describe how matrix operations can be defined on relations, and we look at the implementation by considering a running example. We looked at the constraints of optimizing mixed queries, some equivalence rules for the optimization of matrix addition and selections, the trade-offs and the implementation.

In the experimental evaluation we saw the effects of the trade-offs we make in the optimization, in some cases the optimization improves the query time by up to 95%, whereas, if the selectivity is below 70%, the query time increases the query time significantly.

We can conclude, that MonetDB is well suited for the integration of further matrix operations. MonetDB already includes a BAT algebra module, which provides many arithmetic operations on columns. Even though the matrix addition operation only uses one such BAT algebra operation, other more complex matrix operations can be built by using and combining the existing BAT algebra operations.

Operations, that are complicated to do on the relation tree layer, can be done easily in a lower level layer, the statement tree, while still working in an abstract and declarative way. We used this facility and extended the matrix addition operation to be capable of performing an optional selection between some steps of the matrix operation. The query optimizer can be extended by simply defining how to detect the new case, and defining a function that is called by the optimizer if that case occurs.

Further research can be done to improve the optimization of selections on matrix addition. First, the results of the experimental evaluation could be integrated into the query optimizer, so that the optimization is only carried out when the optimizer can identify a low enough selectivity for a selection. Further, the optimization could be extended to also consider the case, where the selection refers to only a part of the application part attributes. In that case, the evaluation of those attributes by performing the additions, could be done before the selection, as shown in Figure 5.1. Moreover, other matrix operations and corresponding optimizations could be implemented in MonetDB. Even things like consecutive matrix operations, e.g. consecutive matrix multiplications, can possibly be optimized using the query optimizer.

```
SELECT * FROM (r ON a, b ORDER BY d) ADD (s ON e, f ORDER BY g, h)
WHERE b > 5;
```

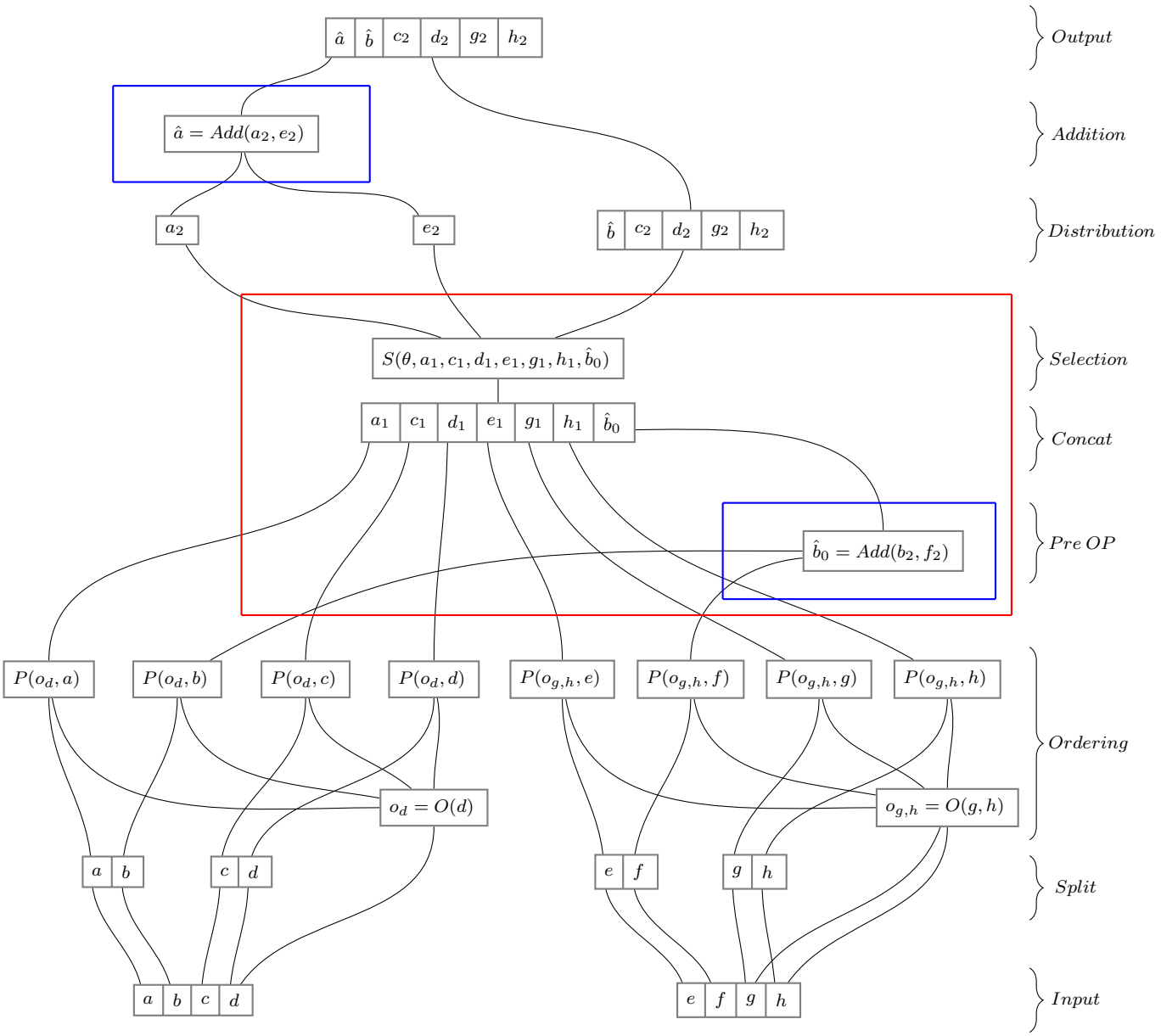

Figure 5.1.: Push down addition statements, if referenced in select condition

Note that in Figure 5.1, the predicate θ of the selection refers to the application part attribute b, but not to the application part attribute a.

Bibliography

[1] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten *MonetDB: Two Decades of Research in Column-oriented Database Architectures*, IEEE Data Engineering Bulletin, *35*(1), 40-45, 2012.

[2] Database Architectures research group (CWI), MonetDB *https://www.monetdb.org*, September 2018.

A. Code

```
1 //  ...
2
3 matrix_ref:
4     '(' table_ref ON selection opt_order_by_clause ')'
5         { dlist *l = L();
6           append_symbol(l, $2);
7           append_symbol(l, $5);
8           append_list(l, $4);
9           $$ = _symbol_create_list( SQL_MATRIX, l); }
10  ;
11
12        /* query expressions */
13
14 opt_no_optimize:
15     /* empty */          { $$ = FALSE; }
16  |  OPTIMIZE             { $$ = FALSE; }
17  |  NOOPTIMIZE           { $$ = TRUE; }
18  ;
19
20 joined_table:
21     '(' joined_table ')'
22         { $$ = $2; }
23  |  table_ref CROSS JOIN table_ref
24         { dlist *l = L();
25           append_symbol(l, $1);
26           append_symbol(l, $4);
27           $$ = _symbol_create_list( SQL_CROSS, l); }
28
29  |  matrix_ref ADD matrix_ref opt_no_optimize
30         { dlist *l = L();
31           append_symbol(l, $1);
32           append_symbol(l, $3);
33           append_int(l, $4);
34           $$ = _symbol_create_list( SQL_MATRIXADD, l); }
35
36 //  ...
```

Listing A.1: sql_parser.y

```
// ...
static list *
append_desc_part(mvc *sql, sql_rel *t, list *ap, list **outexps) {
  if (!*outexps)
    return NULL;

  list *exps = rel_projections(sql, t, NULL, 1, 0);
  if (!exps)
    return NULL;

  node *n;

  for (n = exps->h; n; n = n->next) {
    sql_exp *te = n->data;
    const char *rnm = te->rname;
    const char *nm = te->name;
    sql_exp *e = exps_bind_column(ap, nm, NULL);

    if (!e)
      append(*outexps, te);
  }

  return *outexps;
}

static list *]b
append_appl_part(mvc *sql, list *apl, list *apr, list **outexps) {
  if (!*outexps)
    return NULL;

  int nr = ++sql->label;
  char name[16], *rnm;
  rnm = number2name(name, 16, nr);
  node *n, *en;

  // append only min(list_length(lexps), list_length(rexps))
    expressions
  for (n = apl->h, en = apr->h; n && en; n = n->next, en = en->next) {
    sql_exp *te = n->data;
    const char *nm = te->name;

    exp_setname(sql->sa, te, rnm, sa_strdup(sql->sa, nm));
    append(*outexps, te);
  }

  return *outexps;
}
// ...
```

Listing A.2: rel_select.c, helper functions

```
// ...

static sql_rel *
rel_matrixaddquery(mvc *sql, sql_rel *rel, symbol *q)
{
  dnode *en, *n = q->data.lval->h;

  // read data from symbol tree
  symbol *tab1 = n->data.sym->data.lval->h->data.sym;
  symbol *tab2 = n->data.sym->data.lval->h->next->data.sym;
  dlist  *tab3 = n->data.sym->data.lval->h->next->next->data.lval;
  symbol *tab4 = n->next->data.sym->data.lval->h->data.sym;
  symbol *tab5 = n->next->data.sym->data.lval->h->next->data.sym;
  dlist  *tab6 = n->next->data.sym->data.lval->h->next->next->data.lval
    ;

  // resolve table refs
  sql_rel *t1 = table_ref(sql, rel, tab1);
  sql_rel *t2 = table_ref(sql, rel, tab4);
  if (!t1 || !t2)
    return NULL;

  rel = rel_matrixadd(sql->sa, t1, t2);

  // set no-optimization flag
  rel->noopt = n->next->next->data.i_val;

  list *lobe = NULL;
  list *robe = NULL;
  int lnrcols = 0;
  int rnrcols = 0;

  // set orderby for left relation
  if (tab2) {
    lobe = rel_order_by(sql, &rel, tab2, 0);
  }

  // set orderby for right relation
  if (tab5) {
    sql_rel *t = rel->l;
    rel->l = rel->r;
    rel->r = t;
    robe = rel_order_by(sql, &rel, tab5, 0);
    t = rel->l;
    rel->l = rel->r;
    rel->r = t;
  }

  rel->lord = lobe;
  rel->rord = robe;

```

```
51   // set application part of left relation
52   for (en = tab3->h; en; en = en->next, lnrcols++) {
53     sql_exp *ce = rel_column_exp(sql, &t1, en->data.sym, sql_sel);
54
55     if (ce)
56       append(rel->lexps, ce);
57   }
58
59   // set application part of right relation
60   for (en = tab6->h; en; en = en->next, rnrcols++) {
61     sql_exp *ce = rel_column_exp(sql, &t2, en->data.sym, sql_sel);
62
63     if (ce)
64       append(rel->rexps, ce);
65   }
66
67   if (lnrcols != rnrcols) {
68     sql_error(sql, 02, "MATRIX ADD: number of selected columns from
       tables '%s' and '%s' don\'t match", rel_name(t1)?rel_name(t1):"",
      rel_name(t2)?rel_name(t2):"");
69     rel_destroy(rel);
70     return NULL;
71   }
72
73   // set number of attributes in the result relation
74   rel->nrcols = t1->nrcols + t2->nrcols - lnrcols;
75
76   // project necessary attributes for result relation
77   list *exps = new_exp_list(sql->sa);
78   append_desc_part(sql, t1, rel->lexps, &exps);
79   append_desc_part(sql, t2, rel->rexps, &exps);
80   append_appl_part(sql, rel->lexps, rel->rexps, &exps);
81   rel = rel_project(sql->sa, rel, exps);
82   return rel;
83 }
84
85 // ...
```

Listing A.3: rel_select.c, relation tree

```
1 //  ...
2
3 static void
4 split_exps_appl_desc(mvc *sql, stmt *p, list *exps, list **a, list **d)
5 {
6   node *n, *en;
7
8   for (n = p->op4.lval->h; n; n = n->next) {
9     stmt *c = n->data;
10    stmt *s;
11    bool desc = true;
12    const char *rnme = table_name(sql->sa, c);
13    const char *nme = column_name(sql->sa, c);
14
15    for (en = exps->h; en; en = en->next) {
16      sql_exp *exp = en->data;
17
18      if (exp->l && exp->r) {
19        char *rname = exp->l;
20        char *name = exp->r;
21
22        if (rnme && nme && strcmp(nme, name) == 0
23            && strcmp(rnme, rname) == 0) {
24
25          s = column(sql->sa, c);
26          s = stmt_alias(sql->sa, s, rnme, nme);
27
28          if (a)
29            list_append(*a, s);
30          desc = false;
31          break;
32        }
33      }
34    }
35
36    if (desc) {
37      s = column(sql->sa, c);
38      s = stmt_alias(sql->sa, s, rnme, nme);
39      if (d)
40        list_append(*d, s);
41    }
42  }
43 }
44
45 //  ...
```

Listing A.4: rel_bin.c, helper function

```
// ...

static stmt *
gen_orderby_ids(mvc *sql, stmt *s, list *ord)
{
  if (!ord)
    return NULL;

  node *n;
  list *p;
  stmt *psub = NULL;
  stmt *orderby_ids = NULL;
  stmt *orderby_grp = NULL;

  p = sa_list(sql->sa);
  p->expected_cnt = list_length(s->op4.lval);
  psub = stmt_list(sql->sa, p);

  stmt_set_nrcols(psub);

  // ordering of the order specification columns to know the final
    order of OIDs for
  for (n = ord->h; n; n = n->next) {
    stmt *orderby = NULL;
    sql_exp *orderbycole = n->data;
    stmt *orderbycolstmt = exp_bin(sql, orderbycole, s, psub, NULL,
   NULL, NULL, NULL);

    if (!orderbycolstmt) {
      assert(0);
      return NULL;
    }

    /* single values don't need sorting */
    if (orderbycolstmt->nrcols == 0) {
      orderby_ids = NULL;
      break;
    }

    if (orderby_ids)
      orderby = stmt_reorder(sql->sa, orderbycolstmt, is_ascending(
   orderbycole), orderby_ids, orderby_grp);
    else
      orderby = stmt_order(sql->sa, orderbycolstmt, is_ascending(
   orderbycole));

    const char *tname = table_name(sql->sa, orderbycolstmt);
    const char *cname = column_name(sql->sa, orderbycolstmt);

    orderby_ids = stmt_result(sql->sa, orderby, 1);
    orderby_grp = stmt_result(sql->sa, orderby, 2);
```

```
48    }
49    return orderby_ids;
50  }
51
52  //  ...
```

Listing A.5: rel_bin.c, helper function

```
1  //  ...
2
3  static void
4  align_by_ids(mvc *sql, stmt *orderby_ids, list *l, list **ol)
5  {
6    node *n;
7
8    for(n = l->h; n; n = n->next) {
9      stmt *c = n->data;
10     stmt *s;
11
12     const char *tname = table_name(sql->sa, c);
13     const char *cname = column_name(sql->sa, c);
14
15     if (orderby_ids)
16       s = stmt_project(sql->sa, orderby_ids, c);
17     else
18       s = column(sql->sa, c);
19     s = stmt_alias(sql->sa, s, tname, cname);
20
21     if (ol)
22       list_append(*ol, s);
23   }
24 }
25
26 //  ...
```

Listing A.6: rel_bin.c, helper function

```
// ...

static stmt *
rel2bin_matrixadd(mvc *sql, sql_rel *rel, list *refs)
{
  // list of all statements (result)
  list *l;

  // application part and description part columns
  list *al, *ar, *dl, *dr;

  // ordered application part columns (desc part is directly appended
   to l)
  list *oal, *oar;

  // iterators
  node *n, *en;

  stmt *left = NULL;
  stmt *right = NULL;
  stmt *orderby_idsl = NULL;
  stmt *orderby_idsr = NULL;

  left = subrel_bin(sql, rel->l, refs);
  right = subrel_bin(sql, rel->r, refs);
  assert(left && right);

  // construct list of statements
  l = sa_list(sql->sa);
  al = sa_list(sql->sa);
  ar = sa_list(sql->sa);
  dl = sa_list(sql->sa);
  dr = sa_list(sql->sa);
  oal = sa_list(sql->sa);
  oar = sa_list(sql->sa);

  // split into application and descriptive part lists
  assert(rel->lexps && rel->rexps);
  split_exps_appl_desc(sql, left, rel->lexps, &al, &dl);
  split_exps_appl_desc(sql, right, rel->rexps, &ar, &dr);

  // generate the orderby ids
  orderby_idsl = gen_orderby_ids(sql, left, rel->lord);
  orderby_idsr = gen_orderby_ids(sql, right, rel->rord);

  // align lists according to the orderby ids
  align_by_ids(sql, orderby_idsl, dl, &l);
  align_by_ids(sql, orderby_idsr, dr, &l);
  align_by_ids(sql, orderby_idsl, al, &oal);
  align_by_ids(sql, orderby_idsr, ar, &oar);

```

```
51   // perform selection on concatenated matrices
52   if (rel->exps && rel->exps->h) {
53     list *sl = sa_list(sql->sa);
54     list_merge_destroy(sl, l, NULL);
55     list_merge_destroy(sl, oal, NULL);
56     list_merge_destroy(sl, oar, NULL);
57
58     l = sa_list(sql->sa);
59     oal = sa_list(sql->sa);
60     oar = sa_list(sql->sa);
61
62     stmt *sel = select_on_matrixadd(sql, stmt_list(sql->sa, sl), rel->
      exps, refs);
63
64     // application part of right relation will be in l
65     split_exps_appl_desc(sql, sel, rel->lexps, &oal, &l);
66     split_exps_appl_desc(sql, sel, rel->rexps, &oar, NULL);
67   }
68
69   // create matrixadd stmts, which perform addition between two stmts
      each
70   for (n = oal->h, en = oar->h; n && en; n = n->next, en = en->next) {
71     stmt *s = stmt_matrixadd(sql->sa, n->data, en->data);
72     list_append(l, s);
73   }
74
75   return stmt_list(sql->sa, l);
76 }
77
78 //  ...
```

Listing A.7: rel_bin.c, statement tree

```
// ...
    case st_matrixadd:{
      int  l, r;

      l = _dumpstmt(sql, mb, s->op1);
      r = _dumpstmt(sql, mb, s->op2);
      assert(l >= 0 && r >= 0);

      q = newStmt(mb, batcalcRef, "+");

      q = pushArgument(mb, q, l);
      q = pushArgument(mb, q, r);

      s->nr = getDestVar(q);
      return s->nr;
    }
      break;
// ...
```

Listing A.8: sql_gencode.c

```
// ...
  typedef struct relation {
    sql_ref ref;

    operator_type op;
    void *l;
    void *r;
    list *exps;
    list *lexps;
    list *rexps;
    list *lord;
    list *rord;
    int noopt;
    int nrcols; /* nr of cols */
    char flag;  /* EXP_DISTINCT */
    char card;  /* 0, 1 (row), 2 aggr, 3 */
    char processed;
    char subquery;
    void *p;
  } sql_rel;
// ...
```

Listing A.9: sql_relation.h, modified relation struct

```
// ...

/* forward ref */
int exp_match_one_col_exp(sql_exp *exp, list *l);

int
exps_match_one_col_exp(list *exps, list *l)
{
  node *n;

  for (n = exps->h; n; n = n->next) {
    sql_exp *e = n->data;
    sql_exp *el = e->l;
    sql_exp *er = e->r;

    switch (e->type) {
      case e_func:
      case e_aggr:
      case e_psm:
        return 1;
    }

    if (e->type == e_cmp && e->flag == cmp_or &&
        (exps_match_one_col_exp(e->l, l) ||
         exps_match_one_col_exp(e->r, l)))
        return 1;

    if (er->type == e_convert)
      el = el->l;
    if (el->type == e_convert)
      er = er->l;
    if (el->type == e_column &&
        exp_match_one_col_exp(el, l))
      return 1;
    if (er->type == e_column &&
        exp_match_one_col_exp(er, l))
      return 1;
  }
  return 0;
}

// ...
```

Listing A.10: rel_optimizer.c, helper function

```
// ...

int
exp_match_one_col_exp(sql_exp *exp, list *l)
{
  node *n;

  for (n = l->h; n; n = n->next) {
    sql_exp *e = n->data;

    if (exp_match_exp(exp, e))
      return 1;
  }
  return 0;
}

// ...
```

Listing A.11: rel_optimizer.c, helper function

```
// ...

static sql_rel *
push_select_exps_to_matrix(int *changes, mvc *sql, sql_rel *rel)
{
  if (!is_select(rel->op))
    return rel;

  sql_rel *p = rel->l;
  list *exps = rel->exps;
  node *n;

  if (!p || !is_matrixadd(p->op))
    return rel;

  if (p->noopt)
    return rel;

  for (n = exps->h; n; n = n->next) {
    sql_exp *e = n->data;
    sql_exp *l = e->l;
    sql_exp *r = e->r;

    switch (e->type) {
      case e_func:
      case e_aggr:
      case e_psm:
        continue;
```

```
29      }
30
31      if (e->type == e_cmp && e->flag == cmp_or) {
32        if (exps_match_one_col_exp(e->l, p->lexps) ||
33            exps_match_one_col_exp(e->l, p->rexps))
34          continue;
35        if (exps_match_one_col_exp(e->r, p->lexps) ||
36            exps_match_one_col_exp(e->r, p->rexps))
37          continue;
38      } else {
39        if (l->type == e_convert)
40          l = l->l;
41        if (r->type == e_convert)
42          r = r->l;
43        if (l->type == e_column &&
44            (exp_match_one_col_exp(l, p->lexps) ||
45             exp_match_one_col_exp(l, p->rexps)))
46          continue;
47        if (r->type == e_column &&
48            (exp_match_one_col_exp(r, p->lexps) ||
49             exp_match_one_col_exp(r, p->rexps)))
50          continue;
51      }
52
53      list_append(p->exps, e);
54      list_remove_node(rel->exps, n);
55    }
56
57    return rel;
58  }
59
60  //  ...
```

Listing A.12: rel_optimizer.c, optimization

```
// ...

/* Pushing projects up the tree. Done very early in the optimizer.
 * Makes later steps easier.
 */
static sql_rel *
rel_push_project_up(int *changes, mvc *sql, sql_rel *rel)
{
  /* project/project cleanup is done later */
  if (is_join(rel->op) || is_select(rel->op)) {
    node *n;
    list *exps = NULL, *l_exps, *r_exps;
    sql_rel *l = rel->l;
    sql_rel *r = rel->r;
    sql_rel *t;

    /* Don't rewrite refs, non projections or constant or
       order by projections  */
    if (!l || rel_is_ref(l) ||
       (is_join(rel->op) && (!r || rel_is_ref(r))) ||
       (is_select(rel->op) && l->op != op_project) ||
       (is_join(rel->op) && l->op != op_project && r->op != op_project)
     ||
      ((l->op == op_project && (!l->l || l->r || project_unsafe(l))) ||
       (is_join(rel->op) && (is_subquery(r) ||
        (r->op == op_project && (!r->l || r->r || project_unsafe(r))))))
    ))
      return rel;

    /* Don't rewrite projection of matrixadd */
    if (is_project(l->op) && l->l && is_matrixadd(((sql_rel*)l->l)->op)
    ) {
      fprintf(stderr, ">>> [rel_push_project_up] no action\n");
      return rel;
    }

// ...
```

Listing A.13: rel_optimizer.c, modified projection optimization

```
// ...

  case op_matrixadd:
    if (rel->op == op_matrixadd)
      r = "matrix add";
    print_indent(sql, fout, depth, decorate);
    if (need_distinct(rel))
      mnstr_printf(fout, "distinct ");
    mnstr_printf(fout, "%s (", r);
    if (rel_is_ref(rel->l)) {
      int nr = find_ref(refs, rel->l);
      print_indent(sql, fout, depth+1, decorate);
      mnstr_printf(fout, "& REF %d ", nr);
    } else
      rel_print_(sql, fout, rel->l, depth+1, refs, decorate);
    mnstr_printf(fout, ",");
    if (rel_is_ref(rel->r)) {
      int nr = find_ref(refs, rel->r);
      print_indent(sql, fout, depth+1, decorate);
      mnstr_printf(fout, "& REF %d  ", nr);
    } else
      rel_print_(sql, fout, rel->r, depth+1, refs, decorate);
    print_indent(sql, fout, depth, decorate);
    mnstr_printf(fout, ")");
    exps_print(sql, fout, rel->lexps, depth, 1, 0);
    exps_print(sql, fout, rel->rexps, depth, 1, 0);
    exps_print(sql, fout, rel->lord, depth, 1, 0);
    exps_print(sql, fout, rel->rord, depth, 1, 0);
    exps_print(sql, fout, rel->exps, depth, 1, 0);
    break;

// ...
```

Listing A.14: rel_dump.c, print relation tree with all expressions

B. MAL-Plan

B.1. 100% vs. 99% Selectivity

```
function user.s2_1(A0:int):void;
  X_131:void := querylog.define("explain select * from (x on a,b order by d) add (y on e,f order by g,h) where c < 9;",
      "default_pipe",103);
barrier X_150 := language.dataflow();
  ...
  X_2 := sql.mvc();
  C_3:bat[:oid] := sql.tid(X_2,"sys","x");
  X_6:bat[:str] := sql.bind(X_2,"sys","x","d",0);
  (C_9,r1_9) := sql.bind(X_2,"sys","x","d",2);
  X_12:bat[:str] := sql.bind(X_2,"sys","x","d",1);
  X_14 := sql.delta(X_6,C_9,r1_9,X_12);
  X_15 := algebra.projection(C_3,X_14);
  (X_16,r1_16,r2_16) := algebra.subsort(X_15,false,false);
  X_20:bat[:int] := sql.bind(X_2,"sys","x","c",0);
  (C_22,r1_23) := sql.bind(X_2,"sys","x","c",2);
  X_24:bat[:int] := sql.bind(X_2,"sys","x","c",1);
  X_25 := sql.delta(X_20,C_22,r1_23,X_24);
  X_26:bat[:int] := algebra.projectionpath(r1_16,C_3,X_25);
  C_27 := algebra.thetasubselect(X_26,A0,"<");
  X_29 := algebra.projection(C_27,X_26);
  X_30:bat[:str] := algebra.projectionpath(C_27,r1_16,X_15);
  C_31:bat[:oid] := sql.tid(X_2,"sys","y");
  X_33:bat[:int] := sql.bind(X_2,"sys","y","h",0);
  (C_35,r1_39) := sql.bind(X_2,"sys","y","h",2);
  X_37:bat[:int] := sql.bind(X_2,"sys","y","h",1);
  X_38 := sql.delta(X_33,C_35,r1_39,X_37);
  X_39 := algebra.projection(C_31,X_38);
  X_40:bat[:int] := sql.bind(X_2,"sys","y","g",0);
  (C_42,r1_46) := sql.bind(X_2,"sys","y","g",2);
  X_44:bat[:int] := sql.bind(X_2,"sys","y","g",1);
  X_45 := sql.delta(X_40,C_42,r1_46,X_44);
  X_46 := algebra.projection(C_31,X_45);
  (X_47,r1_51,r2_51) := algebra.subsort(X_46,false,false);
  (X_50,r1_56,r2_56) := algebra.subsort(X_39,r1_51,r2_51,false,false);
  X_53:bat[:int] := algebra.projectionpath(C_27,r1_56,X_46);
  X_54:bat[:int] := algebra.projectionpath(C_27,r1_56,X_39);
  X_55:bat[:int] := sql.bind(X_2,"sys","x","a",0);
  (C_57,r1_65) := sql.bind(X_2,"sys","x","a",2);
  X_59:bat[:int] := sql.bind(X_2,"sys","x","a",1);
  X_60 := sql.delta(X_55,C_57,r1_65,X_59);
  X_61:bat[:int] := algebra.projectionpath(C_27,r1_16,C_3,X_60);
  X_62:bat[:int] := sql.bind(X_2,"sys","y","e",0);
  (C_64,r1_74) := sql.bind(X_2,"sys","y","e",2);
  X_66:bat[:int] := sql.bind(X_2,"sys","y","e",1);
  X_67 := sql.delta(X_62,C_64,r1_74,X_66);
  X_68:bat[:int] := algebra.projectionpath(C_27,r1_56,C_31,X_67);
  X_69 := batcalc.+(X_61,X_68);
  X_70:bat[:int] := sql.bind(X_2,"sys","x","b",0);
  (C_72,r1_84) := sql.bind(X_2,"sys","x","b",2);
  X_74:bat[:int] := sql.bind(X_2,"sys","x","b",1);
  X_75 := sql.delta(X_70,C_72,r1_84,X_74);
  X_76:bat[:int] := algebra.projectionpath(C_27,r1_16,C_3,X_75);
  X_77:bat[:int] := sql.bind(X_2,"sys","y","f",0);
  (C_79,r1_93) := sql.bind(X_2,"sys","y","f",2);
  X_81:bat[:int] := sql.bind(X_2,"sys","y","f",1);
  X_82 := sql.delta(X_77,C_79,r1_93,X_81);
  X_83:bat[:int] := algebra.projectionpath(C_27,r1_56,C_31,X_82);
  X_84 := batcalc.+(X_76,X_83);
  ...
exit X_150;
  sql.resultSet(X_126,X_127,X_128,X_129,X_130,X_29,X_30,X_53,X_54,X_69,X_84);
end user.s2_1;
```

Listing B.1: MAL-Plan, optimized, unordered, 100% selectivity

```
function user.s2_1(A0:int):void;
  X_131:void := querylog.define("explain select * from (x on a,b order by d) add (y on e,f order by g,h) where c < 8;",
      "default_pipe",103);
barrier X_150 := language.dataflow();
  ...
  X_2 := sql.mvc();
  C_3:bat[:oid] := sql.tid(X_2,"sys","x");
  X_6:bat[:str] := sql.bind(X_2,"sys","x","d",0);
  (C_9,r1_9) := sql.bind(X_2,"sys","x","d",2);
  X_12:bat[:str] := sql.bind(X_2,"sys","x","d",1);
  X_14 := sql.delta(X_6,C_9,r1_9,X_12);
  X_15 := algebra.projection(C_3,X_14);
  (X_16,r1_16,r2_16) := algebra.subsort(X_15,false,false);
  X_20:bat[:int] := sql.bind(X_2,"sys","x","c",0);
  (C_22,r1_23) := sql.bind(X_2,"sys","x","c",2);
  X_24:bat[:int] := sql.bind(X_2,"sys","x","c",1);
  X_25 := sql.delta(X_20,C_22,r1_23,X_24);
  X_26:bat[:int] := algebra.projectionpath(r1_16,C_3,X_25);
  C_27 := algebra.thetasubselect(X_26,A0,"<");
  X_29 := algebra.projection(C_27,X_26);
  X_30:bat[:str] := algebra.projectionpath(C_27,r1_16,X_15);
  C_31:bat[:oid] := sql.tid(X_2,"sys","y");
  X_33:bat[:int] := sql.bind(X_2,"sys","y","h",0);
  (C_35,r1_39) := sql.bind(X_2,"sys","y","h",2);
  X_37:bat[:int] := sql.bind(X_2,"sys","y","h",1);
  X_38 := sql.delta(X_33,C_35,r1_39,X_37);
  X_39 := algebra.projection(C_31,X_38);
  X_40:bat[:int] := sql.bind(X_2,"sys","y","g",0);
  (C_42,r1_46) := sql.bind(X_2,"sys","y","g",2);
  X_44:bat[:int] := sql.bind(X_2,"sys","y","g",1);
  X_45 := sql.delta(X_40,C_42,r1_46,X_44);
  X_46 := algebra.projection(C_31,X_45);
  (X_47,r1_51,r2_51) := algebra.subsort(X_46,false,false);
  (X_50,r1_56,r2_56) := algebra.subsort(X_39,r1_51,r2_51,false,false);
  X_53:bat[:int] := algebra.projectionpath(C_27,r1_56,X_46);
  X_54:bat[:int] := algebra.projectionpath(C_27,r1_56,X_39);
  X_55:bat[:int] := sql.bind(X_2,"sys","x","a",0);
  (C_57,r1_65) := sql.bind(X_2,"sys","x","a",2);
  X_59:bat[:int] := sql.bind(X_2,"sys","x","a",1);
  X_60 := sql.delta(X_55,C_57,r1_65,X_59);
  X_61:bat[:int] := algebra.projectionpath(C_27,r1_16,C_3,X_60);
  X_62:bat[:int] := sql.bind(X_2,"sys","y","e",0);
  (C_64,r1_74) := sql.bind(X_2,"sys","y","e",2);
  X_66:bat[:int] := sql.bind(X_2,"sys","y","e",1);
  X_67 := sql.delta(X_62,C_64,r1_74,X_66);
  X_68:bat[:int] := algebra.projectionpath(C_27,r1_56,C_31,X_67);
  X_69 := batcalc.+(X_61,X_68);
  X_70:bat[:int] := sql.bind(X_2,"sys","x","b",0);
  (C_72,r1_84) := sql.bind(X_2,"sys","x","b",2);
  X_74:bat[:int] := sql.bind(X_2,"sys","x","b",1);
  X_75 := sql.delta(X_70,C_72,r1_84,X_74);
  X_76:bat[:int] := algebra.projectionpath(C_27,r1_16,C_3,X_75);
  X_77:bat[:int] := sql.bind(X_2,"sys","y","f",0);
  (C_79,r1_93) := sql.bind(X_2,"sys","y","f",2);
  X_81:bat[:int] := sql.bind(X_2,"sys","y","f",1);
  X_82 := sql.delta(X_77,C_79,r1_93,X_81);
  X_83:bat[:int] := algebra.projectionpath(C_27,r1_56,C_31,X_82);
  X_84 := batcalc.+(X_76,X_83);
  ...
exit X_150;
  sql.resultSet(X_126,X_127,X_128,X_129,X_130,X_29,X_30,X_53,X_54,X_69,X_84);
end user.s2_1;
```

Listing B.2: MAL-Plan, optimized, unordered, 99% selectivity

B.2. Optimized, Selection on Ordered vs. Unordered Attribute

```
function user.s2_1(A0:int):void;
  X_131:void := querylog.define("explain select * from (x on a,b order by d) add (y on e,f order by g,h) where h < 9;",
      "default_pipe",103);
barrier X_150 := language.dataflow();
  ...
  X_2 := sql.mvc();
  C_3:bat[:oid] := sql.tid(X_2,"sys","y");
  X_6:bat[:int] := sql.bind(X_2,"sys","y","h",0);
  (C_9,r1_9) := sql.bind(X_2,"sys","y","h",2);
  X_12:bat[:int] := sql.bind(X_2,"sys","y","h",1);
  X_14 := sql.delta(X_6,C_9,r1_9,X_12);
  X_15 := algebra.projection(C_3,X_14);
  X_16:bat[:int] := sql.bind(X_2,"sys","y","g",0);
  (C_18,r1_18) := sql.bind(X_2,"sys","y","g",2);
  X_20:bat[:int] := sql.bind(X_2,"sys","y","g",1);
  X_21 := sql.delta(X_16,C_18,r1_18,X_20);
  X_22 := algebra.projection(C_3,X_21);
  (X_23,r1_23,r2_23) := algebra.subsort(X_22,false,false);
  (X_27,r1_28,r2_28) := algebra.subsort(X_15,r1_23,r2_23,false,false);
  X_30 := algebra.projection(r1_28,X_15);
  C_31 := algebra.thetasubselect(X_30,A0,"<");
  C_33:bat[:oid] := sql.tid(X_2,"sys","x");
  X_35:bat[:str] := sql.bind(X_2,"sys","x","d",0);
  (C_37,r1_39) := sql.bind(X_2,"sys","x","d",2);
  X_39:bat[:str] := sql.bind(X_2,"sys","x","d",1);
  X_40 := sql.delta(X_35,C_37,r1_39,X_39);
  X_41 := algebra.projection(C_33,X_40);
  (X_42,r1_44,r2_44) := algebra.subsort(X_41,false,false);
  X_45:bat[:int] := sql.bind(X_2,"sys","x","c",0);
  (C_47,r1_49) := sql.bind(X_2,"sys","x","c",2);
  X_49:bat[:int] := sql.bind(X_2,"sys","x","c",1);
  X_50 := sql.delta(X_45,C_47,r1_49,X_49);
  X_51:bat[:int] := algebra.projectionpath(C_31,r1_44,C_33,X_50);
  X_52:bat[:str] := algebra.projectionpath(C_31,r1_44,X_41);
  X_53:bat[:int] := algebra.projectionpath(C_31,r1_28,X_22);
  X_54 := algebra.projection(C_31,X_30);
  X_55:bat[:int] := sql.bind(X_2,"sys","x","a",0);
  (C_57,r1_63) := sql.bind(X_2,"sys","x","a",2);
  X_59:bat[:int] := sql.bind(X_2,"sys","x","a",1);
  X_60 := sql.delta(X_55,C_57,r1_63,X_59);
  X_61:bat[:int] := algebra.projectionpath(C_31,r1_44,C_33,X_60);
  X_62:bat[:int] := sql.bind(X_2,"sys","y","e",0);
  (C_64,r1_72) := sql.bind(X_2,"sys","y","e",2);
  X_66:bat[:int] := sql.bind(X_2,"sys","y","e",1);
  X_67 := sql.delta(X_62,C_64,r1_72,X_66);
  X_68:bat[:int] := algebra.projectionpath(C_31,r1_28,C_3,X_67);
  X_69 := batcalc.+(X_61,X_68);
  X_70:bat[:int] := sql.bind(X_2,"sys","x","b",0);
  (C_72,r1_82) := sql.bind(X_2,"sys","x","b",2);
  X_74:bat[:int] := sql.bind(X_2,"sys","x","b",1);
  X_75 := sql.delta(X_70,C_72,r1_82,X_74);
  X_76:bat[:int] := algebra.projectionpath(C_31,r1_44,C_33,X_75);
  X_77:bat[:int] := sql.bind(X_2,"sys","y","f",0);
  (C_79,r1_91) := sql.bind(X_2,"sys","y","f",2);
  X_81:bat[:int] := sql.bind(X_2,"sys","y","f",1);
  X_82 := sql.delta(X_77,C_79,r1_91,X_81);
  X_83:bat[:int] := algebra.projectionpath(C_31,r1_28,C_3,X_82);
  X_84 := batcalc.+(X_76,X_83);
  ...
exit X_150;
  sql.resultSet(X_126,X_127,X_128,X_129,X_130,X_51,X_52,X_53,X_54,X_69,X_84);
end user.s2_1;
```

Listing B.3: MAL-Plan, optimized, ordered

```
function user.s2_1(A0:int):void;
  X_131:void := querylog.define("explain select * from (x on a,b order by d) add (y on e,f order by g,h) where c < 9;",
      "default_pipe",103);
barrier X_150 := language.dataflow();
  ...
  X_2 := sql.mvc();
  C_3:bat[:oid] := sql.tid(X_2,"sys","x");
  X_6:bat[:str] := sql.bind(X_2,"sys","x","d",0);
  (C_9,r1_9) := sql.bind(X_2,"sys","x","d",2);
  X_12:bat[:str] := sql.bind(X_2,"sys","x","d",1);
  X_14 := sql.delta(X_6,C_9,r1_9,X_12);
  X_15 := algebra.projection(C_3,X_14);
  (X_16,r1_16,r2_16) := algebra.subsort(X_15,false,false);
  X_20:bat[:int] := sql.bind(X_2,"sys","x","c",0);
  (C_22,r1_23) := sql.bind(X_2,"sys","x","c",2);
  X_24:bat[:int] := sql.bind(X_2,"sys","x","c",1);
  X_25 := sql.delta(X_20,C_22,r1_23,X_24);
  X_26:bat[:int] := algebra.projectionpath(r1_16,C_3,X_25);
  C_27 := algebra.thetasubselect(X_26,A0,"<");
  X_29 := algebra.projection(C_27,X_26);
  X_30:bat[:str] := algebra.projectionpath(C_27,r1_16,X_15);
  C_31:bat[:oid] := sql.tid(X_2,"sys","y");
  X_33:bat[:int] := sql.bind(X_2,"sys","y","h",0);
  (C_35,r1_39) := sql.bind(X_2,"sys","y","h",2);
  X_37:bat[:int] := sql.bind(X_2,"sys","y","h",1);
  X_38 := sql.delta(X_33,C_35,r1_39,X_37);
  X_39 := algebra.projection(C_31,X_38);
  X_40:bat[:int] := sql.bind(X_2,"sys","y","g",0);
  (C_42,r1_46) := sql.bind(X_2,"sys","y","g",2);
  X_44:bat[:int] := sql.bind(X_2,"sys","y","g",1);
  X_45 := sql.delta(X_40,C_42,r1_46,X_44);
  X_46 := algebra.projection(C_31,X_45);
  (X_47,r1_51,r2_51) := algebra.subsort(X_46,false,false);
  (X_50,r1_56,r2_56) := algebra.subsort(X_39,r1_51,r2_51,false,false);
  X_53:bat[:int] := algebra.projectionpath(C_27,r1_56,X_46);
  X_54:bat[:int] := algebra.projectionpath(C_27,r1_56,X_39);
  X_55:bat[:int] := sql.bind(X_2,"sys","x","a",0);
  (C_57,r1_65) := sql.bind(X_2,"sys","x","a",2);
  X_59:bat[:int] := sql.bind(X_2,"sys","x","a",1);
  X_60 := sql.delta(X_55,C_57,r1_65,X_59);
  X_61:bat[:int] := algebra.projectionpath(C_27,r1_16,C_3,X_60);
  X_62:bat[:int] := sql.bind(X_2,"sys","y","e",0);
  (C_64,r1_74) := sql.bind(X_2,"sys","y","e",2);
  X_66:bat[:int] := sql.bind(X_2,"sys","y","e",1);
  X_67 := sql.delta(X_62,C_64,r1_74,X_66);
  X_68:bat[:int] := algebra.projectionpath(C_27,r1_56,C_31,X_67);
  X_69 := batcalc.+(X_61,X_68);
  X_70:bat[:int] := sql.bind(X_2,"sys","x","b",0);
  (C_72,r1_84) := sql.bind(X_2,"sys","x","b",2);
  X_74:bat[:int] := sql.bind(X_2,"sys","x","b",1);
  X_75 := sql.delta(X_70,C_72,r1_84,X_74);
  X_76:bat[:int] := algebra.projectionpath(C_27,r1_16,C_3,X_75);
  X_77:bat[:int] := sql.bind(X_2,"sys","y","f",0);
  (C_79,r1_93) := sql.bind(X_2,"sys","y","f",2);
  X_81:bat[:int] := sql.bind(X_2,"sys","y","f",1);
  X_82 := sql.delta(X_77,C_79,r1_93,X_81);
  X_83:bat[:int] := algebra.projectionpath(C_27,r1_56,C_31,X_82);
  X_84 := batcalc.+(X_76,X_83);
  ...
exit X_150;
  sql.resultSet(X_126,X_127,X_128,X_129,X_130,X_29,X_30,X_53,X_54,X_69,X_84);
end user.s2_1;
```

Listing B.4: MAL-Plan, optimized, unordered

B.3. Unoptimized, Selection on Ordered vs. Unordered Attribute

```
function user.s2_1(A0:int):void;
  X_133:void := querylog.define("explain select * from (x on a,b order by d) add (y on e,f order by g,h) nooptimize
      where h < 9;","default_pipe",105);
barrier X_152 := language.dataflow();
  ...
  X_2 := sql.mvc();
  C_3:bat[:oid] := sql.tid(X_2,"sys","y");
  X_6:bat[:int] := sql.bind(X_2,"sys","y","h",0);
  (C_9,r1_9) := sql.bind(X_2,"sys","y","h",2);
  X_12:bat[:int] := sql.bind(X_2,"sys","y","h",1);
  X_14 := sql.delta(X_6,C_9,r1_9,X_12);
  X_15 := algebra.projection(C_3,X_14);
  X_16:bat[:int] := sql.bind(X_2,"sys","y","g",0);
  (C_18,r1_18) := sql.bind(X_2,"sys","y","g",2);
  X_20:bat[:int] := sql.bind(X_2,"sys","y","g",1);
  X_21 := sql.delta(X_16,C_18,r1_18,X_20);
  X_22 := algebra.projection(C_3,X_21);
  (X_23,r1_23,r2_23) := algebra.subsort(X_22,false,false);
  (X_27,r1_28,r2_28) := algebra.subsort(X_15,r1_23,r2_23,false,false);
  X_30 := algebra.projection(r1_28,X_15);
  C_31 := algebra.thetasubselect(X_30,A0,"<");
  C_33:bat[:oid] := sql.tid(X_2,"sys","x");
  X_35:bat[:str] := sql.bind(X_2,"sys","x","d",0);
  (C_37,r1_39) := sql.bind(X_2,"sys","x","d",2);
  X_39:bat[:str] := sql.bind(X_2,"sys","x","d",1);
  X_40 := sql.delta(X_35,C_37,r1_39,X_39);
  X_41 := algebra.projection(C_33,X_40);
  (X_42,r1_44,r2_44) := algebra.subsort(X_41,false,false);
  X_45:bat[:int] := sql.bind(X_2,"sys","x","c",0);
  (C_47,r1_49) := sql.bind(X_2,"sys","x","c",2);
  X_49:bat[:int] := sql.bind(X_2,"sys","x","c",1);
  X_50 := sql.delta(X_45,C_47,r1_49,X_49);
  X_51:bat[:int] := algebra.projectionpath(C_31,r1_44,C_33,X_50);
  X_52:bat[:str] := algebra.projectionpath(C_31,r1_44,X_41);
  X_53:bat[:int] := algebra.projectionpath(C_31,r1_28,X_22);
  X_54 := algebra.projection(C_31,X_30);
  X_55:bat[:int] := sql.bind(X_2,"sys","x","a",0);
  (C_57,r1_63) := sql.bind(X_2,"sys","x","a",2);
  X_59:bat[:int] := sql.bind(X_2,"sys","x","a",1);
  X_60 := sql.delta(X_55,C_57,r1_63,X_59);
  X_61:bat[:int] := algebra.projectionpath(r1_44,C_33,X_60);
  X_62:bat[:int] := sql.bind(X_2,"sys","y","e",0);
  (C_64,r1_71) := sql.bind(X_2,"sys","y","e",2);
  X_66:bat[:int] := sql.bind(X_2,"sys","y","e",1);
  X_67 := sql.delta(X_62,C_64,r1_71,X_66);
  X_68:bat[:int] := algebra.projectionpath(r1_28,C_3,X_67);
  X_69 := batcalc.+(X_61,X_68);
  X_70 := algebra.projection(C_31,X_69);
  X_71:bat[:int] := sql.bind(X_2,"sys","x","b",0);
  (C_73,r1_81) := sql.bind(X_2,"sys","x","b",2);
  X_75:bat[:int] := sql.bind(X_2,"sys","x","b",1);
  X_76 := sql.delta(X_71,C_73,r1_81,X_75);
  X_77:bat[:int] := algebra.projectionpath(r1_44,C_33,X_76);
  X_78:bat[:int] := sql.bind(X_2,"sys","y","f",0);
  (C_80,r1_89) := sql.bind(X_2,"sys","y","f",2);
  X_82:bat[:int] := sql.bind(X_2,"sys","y","f",1);
  X_83 := sql.delta(X_78,C_80,r1_89,X_82);
  X_84:bat[:int] := algebra.projectionpath(r1_28,C_3,X_83);
  X_85 := batcalc.+(X_77,X_84);
  X_86 := algebra.projection(C_31,X_85);
  ...
exit X_152;
  sql.resultSet(X_128,X_129,X_130,X_131,X_132,X_51,X_52,X_53,X_54,X_70,X_86);
end user.s2_1;
```

Listing B.5: MAL-Plan, unoptimized, ordered

```
function user.s3_1(A0:int):void;
  X_133:void := querylog.define("explain select * from (x on a,b order by d) add (y on e,f order by g,h) nooptimize
      where c < 9;","default_pipe",105);
barrier X_152 := language.dataflow();
  ...
  X_2 := sql.mvc();
  C_3:bat[:oid] := sql.tid(X_2,"sys","x");
  X_6:bat[:str] := sql.bind(X_2,"sys","x","d",0);
  (C_9,r1_9) := sql.bind(X_2,"sys","x","d",2);
  X_12:bat[:str] := sql.bind(X_2,"sys","x","d",1);
  X_14 := sql.delta(X_6,C_9,r1_9,X_12);
  X_15 := algebra.projection(C_3,X_14);
  (X_16,r1_16,r2_16) := algebra.subsort(X_15,false,false);
  X_20:bat[:int] := sql.bind(X_2,"sys","x","c",0);
  (C_22,r1_23) := sql.bind(X_2,"sys","x","c",2);
  X_24:bat[:int] := sql.bind(X_2,"sys","x","c",1);
  X_25 := sql.delta(X_20,C_22,r1_23,X_24);
  X_26:bat[:int] := algebra.projectionpath(r1_16,C_3,X_25);
  C_27 := algebra.thetasubselect(X_26,A0,"<");
  X_29 := algebra.projection(C_27,X_26);
  X_30:bat[:str] := algebra.projectionpath(C_27,r1_16,X_15);
  C_31:bat[:oid] := sql.tid(X_2,"sys","y");
  X_33:bat[:int] := sql.bind(X_2,"sys","y","h",0);
  (C_35,r1_39) := sql.bind(X_2,"sys","y","h",2);
  X_37:bat[:int] := sql.bind(X_2,"sys","y","h",1);
  X_38 := sql.delta(X_33,C_35,r1_39,X_37);
  X_39 := algebra.projection(C_31,X_38);
  X_40:bat[:int] := sql.bind(X_2,"sys","y","g",0);
  (C_42,r1_46) := sql.bind(X_2,"sys","y","g",2);
  X_44:bat[:int] := sql.bind(X_2,"sys","y","g",1);
  X_45 := sql.delta(X_40,C_42,r1_46,X_44);
  X_46 := algebra.projection(C_31,X_45);
  (X_47,r1_51,r2_51) := algebra.subsort(X_46,false,false);
  (X_50,r1_56,r2_56) := algebra.subsort(X_39,r1_51,r2_51,false,false);
  X_53:bat[:int] := algebra.projectionpath(C_27,r1_56,X_46);
  X_54:bat[:int] := algebra.projectionpath(C_27,r1_56,X_39);
  X_55:bat[:int] := sql.bind(X_2,"sys","x","a",0);
  (C_57,r1_65) := sql.bind(X_2,"sys","x","a",2);
  X_59:bat[:int] := sql.bind(X_2,"sys","x","a",1);
  X_60 := sql.delta(X_55,C_57,r1_65,X_59);
  X_61:bat[:int] := algebra.projectionpath(r1_16,C_3,X_60);
  X_62:bat[:int] := sql.bind(X_2,"sys","y","e",0);
  (C_64,r1_73) := sql.bind(X_2,"sys","y","e",2);
  X_66:bat[:int] := sql.bind(X_2,"sys","y","e",1);
  X_67 := sql.delta(X_62,C_64,r1_73,X_66);
  X_68:bat[:int] := algebra.projectionpath(r1_56,C_31,X_67);
  X_69 := batcalc.+(X_61,X_68);
  X_70 := algebra.projection(C_27,X_69);
  X_71:bat[:int] := sql.bind(X_2,"sys","x","b",0);
  (C_73,r1_83) := sql.bind(X_2,"sys","x","b",2);
  X_75:bat[:int] := sql.bind(X_2,"sys","x","b",1);
  X_76 := sql.delta(X_71,C_73,r1_83,X_75);
  X_77:bat[:int] := algebra.projectionpath(r1_16,C_3,X_76);
  X_78:bat[:int] := sql.bind(X_2,"sys","y","f",0);
  (C_80,r1_91) := sql.bind(X_2,"sys","y","f",2);
  X_82:bat[:int] := sql.bind(X_2,"sys","y","f",1);
  X_83 := sql.delta(X_78,C_80,r1_91,X_82);
  X_84:bat[:int] := algebra.projectionpath(r1_56,C_31,X_83);
  X_85 := batcalc.+(X_77,X_84);
  X_86 := algebra.projection(C_27,X_85);
  ...
exit X_152;
  sql.resultSet(X_128,X_129,X_130,X_131,X_132,X_29,X_30,X_53,X_54,X_70,X_86);
end user.s3_1;
```

Listing B.6: MAL-Plan, unoptimized, unordered

C. Detailed Results

C.1. Selectivity, Unordered

Selectivity (%)	Optimized (s)				Unoptimized (s)			
	1	2	3	Mean	1	2	3	Mean
0	2.00	2.00	2.20	2.07	44.80	49.30	46.40	46.83
10	9.60	9.60	9.60	9.60	46.60	47.70	48.20	47.50
20	16.80	16.70	16.70	16.73	52.90	49.90	50.90	51.23
30	24.00	23.90	24.00	23.97	52.20	49.70	53.00	51.63
40	32.90	33.70	33.40	33.33	53.40	50.20	51.20	51.60
50	39.30	39.10	38.90	39.10	53.10	52.80	56.00	53.97
60	50.10	47.20	48.50	48.60	55.90	54.00	53.50	54.47
65	55.30	50.10	56.90	54.10	57.10	56.60	57.20	56.97
70	55.40	57.20	58.60	57.07	60.00	60.10	56.10	58.73
75	60.50	61.50	58.80	60.27	62.80	63.20	60.90	62.30
80	61.90	63.00	66.50	63.80	63.90	65.30	61.10	63.43
85	70.50	70.50	66.90	69.30	64.40	62.50	65.10	64.00
90	72.50	73.10	72.60	72.73	61.70	65.30	71.20	66.07
95	78.20	80.70	80.10	79.67	63.10	67.50	70.90	67.17
99	83.80	79.60	82.20	81.87	63.00	68.90	70.70	67.53
100	65.70	65.00	62.20	64.30	69.40	62.60	66.30	66.10

Table C.1.: Query time depending on selectivity, selection on unordered attribute

C.2. Selectivity, Ordered

Selectivity (%)	Optimized (s)				Unoptimized (s)			
	1	2	3	Mean	1	2	3	Mean
0	2.00	2.00	2.00	2.00	48.60	48.20	48.10	48.30
10	6.70	6.50	6.60	6.60	50.30	48.70	48.90	49.30
20	11.70	11.00	11.20	11.30	47.80	52.50	51.30	50.53
30	18.80	20.80	21.20	20.27	52.10	51.40	52.60	52.03
40	26.00	25.90	27.20	26.37	55.90	51.60	54.30	53.93
50	29.40	29.40	29.00	29.27	55.60	55.50	54.40	55.17
60	36.00	33.80	35.40	35.07	56.70	57.60	58.90	57.73
65	39.60	38.60	36.90	38.37	59.10	60.40	59.90	59.80
70	39.30	43.00	42.00	41.43	61.50	61.90	62.00	61.80
75	46.90	46.20	47.40	46.83	64.50	64.00	69.80	66.10
80	48.50	49.80	50.00	49.43	70.00	69.10	68.60	69.23
85	53.50	52.30	54.40	53.40	70.60	71.30	70.20	70.70
90	52.90	55.50	56.70	55.03	69.50	71.20	72.40	71.03
95	59.30	54.20	59.70	57.73	73.90	72.10	70.90	72.30
99	58.20	61.50	58.20	59.30	72.00	73.20	74.40	73.20
100	57.10	58.20	58.90	58.07	70.50	72.10	73.90	72.17

Table C.2.: Query time depending on selectivity, selection on ordered attribute

C.3. Application Part, 50% Selectivity

# Attributes	Optimized (s)				Unoptimized (s)			
	1	2	3	Mean	1	2	3	Mean
1	0.30	0.30	0.20	0.27	0.30	0.20	0.30	0.27
50	1.50	1.30	1.40	1.40	1.60	1.70	1.70	1.67
100	2.80	2.70	3.10	2.87	3.10	3.10	3.20	3.13
150	3.90	4.10	4.00	4.00	4.60	4.60	4.70	4.63
200	5.30	5.10	5.00	5.13	6.00	6.10	6.10	6.07
250	6.30	6.30	6.20	6.27	7.50	7.60	7.60	7.57
300	7.60	8.10	8.40	8.03	9.10	9.10	9.20	9.13
350	8.70	9.10	9.10	8.97	10.90	10.90	11.00	10.93
400	9.70	9.80	10.30	9.93	12.20	12.30	13.00	12.50
450	12.10	11.80	11.80	11.90	14.30	14.50	14.00	14.27
500	12.80	12.60	12.50	12.63	15.50	15.30	15.20	15.33
550	13.80	13.30	13.50	13.53	18.20	17.40	17.40	17.67
600	15.10	15.50	15.30	15.30	19.20	19.00	19.30	19.17
650	16.00	16.50	16.40	16.30	22.20	22.20	22.00	22.13
700	17.70	19.00	18.60	18.43	26.90	26.00	26.40	26.43
750	19.40	20.40	21.60	20.47	29.50	32.50	31.50	31.17
800	20.50	22.10	22.80	21.80	36.80	35.50	35.70	36.00
850	23.70	25.90	25.60	25.07	39.80	40.30	40.10	40.07
900	29.60	28.50	28.30	28.80	44.70	44.90	45.00	44.87
950	32.60	31.10	32.00	31.90	50.00	48.60	52.10	50.23
1000	34.50	34.20	36.40	35.03	53.60	59.20	55.10	55.97

Table C.3.: Query time depending on number of application part attributes, 50% selectivity

C.4. Application Part, 10% Selectivity

# Attributes	Optimized (s)				Unoptimized (s)			
	1	2	3	Mean	1	2	3	Mean
1	0.20	0.20	0.20	0.20	0.20	0.20	0.20	0.20
50	0.60	0.50	0.50	0.53	1.50	1.50	1.50	1.50
100	0.90	0.90	0.80	0.87	2.80	2.90	2.80	2.83
150	1.20	1.20	1.10	1.17	4.10	4.20	4.10	4.13
200	1.50	1.40	1.60	1.50	5.50	5.50	5.50	5.50
250	1.80	1.80	1.80	1.80	6.80	6.80	6.90	6.83
300	2.20	2.10	2.10	2.13	8.20	8.20	8.20	8.20
350	2.40	2.40	2.40	2.40	9.50	9.60	9.60	9.57
400	2.80	2.70	2.70	2.73	11.00	10.90	11.00	10.97
450	3.10	3.10	3.20	3.13	12.30	12.30	12.20	12.27
500	3.30	3.30	3.30	3.30	13.60	13.80	13.70	13.70
550	3.70	3.80	3.70	3.73	15.60	15.50	14.80	15.30
600	4.10	4.10	4.00	4.07	16.80	16.40	16.70	16.63
650	4.50	4.50	4.70	4.57	18.40	18.50	17.90	18.27
700	4.90	4.90	4.80	4.87	19.10	19.40	19.20	19.23
750	5.30	5.20	5.20	5.23	21.00	21.10	20.60	20.90
800	5.50	5.50	5.60	5.53	23.30	24.10	23.70	23.70
850	5.80	6.00	5.80	5.87	26.90	26.90	26.60	26.80
900	6.40	6.40	6.40	6.40	30.50	30.30	30.70	30.50
950	6.80	6.80	6.70	6.77	33.30	33.80	33.10	33.40
1000	7.10	7.10	7.20	7.13	36.20	35.20	38.10	36.50

Table C.4.: Query time depending on number of application part attributes, 10% selectivity

C.5. Application Part, 90% Selectivity

# Attributes	Optimized (s)				Unoptimized (s)			
	1	2	3	Mean	1	2	3	Mean
1	0.30	0.30	0.30	0.30	0.30	0.30	0.30	0.30
50	1.90	2.00	2.50	2.13	1.80	1.90	1.90	1.87
100	4.60	4.70	3.90	4.40	3.90	3.50	3.30	3.57
150	6.30	6.10	7.00	6.47	5.10	4.90	4.80	4.93
200	9.30	8.20	9.30	8.93	6.50	6.50	6.50	6.50
250	10.90	10.80	11.10	10.93	7.30	8.30	7.80	7.80
300	13.00	12.70	12.40	12.70	9.90	10.20	9.70	9.93
350	14.70	14.20	15.00	14.63	11.20	12.20	12.30	11.90
400	16.40	16.50	16.50	16.47	13.60	13.80	13.50	13.63
450	18.70	19.00	18.10	18.60	14.60	15.10	15.90	15.20
500	20.70	19.90	19.90	20.17	16.90	16.80	15.70	16.47
550	23.20	22.90	22.30	22.80	18.70	18.60	17.70	18.33
600	25.30	25.90	25.30	25.50	20.10	20.60	20.70	20.47
650	27.10	28.30	27.60	27.67	24.10	23.30	23.20	23.53
700	29.70	30.00	32.10	30.60	27.30	28.30	28.00	27.87
750	31.60	32.20	36.90	33.57	33.20	32.40	35.00	33.53
800	39.50	38.20	37.80	38.50	40.20	38.20	39.70	39.37
850	43.90	40.90	43.80	42.87	45.10	43.10	45.60	44.60
900	46.10	46.90	48.70	47.23	52.70	52.50	51.30	52.17
950	52.40	50.90	51.70	51.67	52.70	60.70	61.80	58.40
1000	57.20	58.00	56.50	57.23	65.60	59.20	63.90	62.90

Table C.5.: Query time depending on number of application part attributes, 90% selectivity

C.6. Descriptive Part

# Attributes	Optimized (s)				Unoptimized (s)			
	1	2	3	Mean	1	2	3	Mean
1	0.50	0.50	0.50	0.50	0.50	0.50	0.50	0.50
50	1.30	1.30	1.30	1.30	1.10	1.10	1.10	1.10
100	2.00	2.10	2.10	2.07	1.70	1.70	1.80	1.73
150	2.40	2.80	2.90	2.70	2.40	2.40	2.40	2.40
200	3.20	3.20	3.30	3.23	3.00	3.00	3.00	3.00
250	3.90	3.90	3.90	3.90	3.60	3.70	3.60	3.63
300	4.50	4.50	4.50	4.50	4.30	4.30	4.30	4.30
350	5.10	5.10	5.20	5.13	4.90	4.90	4.90	4.90
400	5.70	5.60	5.60	5.63	5.50	5.50	5.40	5.47
450	6.40	6.40	6.40	6.40	6.20	6.20	6.20	6.20
500	7.10	7.20	7.10	7.13	6.90	6.90	7.00	6.93
550	7.70	7.70	7.60	7.67	7.60	7.60	7.60	7.60
600	8.40	8.30	8.30	8.33	8.10	8.30	8.30	8.23
650	9.10	9.00	9.20	9.10	9.00	8.90	9.10	9.00
700	9.30	9.30	9.30	9.30	9.40	9.60	9.50	9.50
750	10.00	10.20	10.20	10.13	10.10	10.20	10.30	10.20
800	10.70	10.90	10.90	10.83	10.70	10.70	10.70	10.70
850	11.40	11.30	11.70	11.47	11.80	11.90	11.60	11.77
900	12.30	11.90	12.00	12.07	12.10	12.20	12.00	12.10
950	12.60	12.60	12.60	12.60	12.60	13.00	12.70	12.77
1000	13.30	13.30	13.20	13.27	13.70	13.60	13.80	13.70

Table C.6.: Query time depending on number of descriptive part attributes

C.7. Tuples

# Tuples	Optimized (s)				Unoptimized (s)			
	1	2	3	Mean	1	2	3	Mean
1 M	0.60	0.60	0.60	0.60	0.60	0.70	0.70	0.67
5 M	3.40	3.50	3.40	3.43	3.80	3.80	3.80	3.80
10 M	7.00	7.00	7.20	7.07	7.60	7.80	7.90	7.77
15 M	10.90	10.70	10.80	10.80	12.10	11.90	11.80	11.93
20 M	14.60	15.00	14.80	14.80	16.00	15.90	16.00	15.97
25 M	18.60	18.20	18.20	18.33	20.00	19.80	19.90	19.90
30 M	22.60	22.20	22.30	22.37	24.60	24.40	24.20	24.40
35 M	26.70	26.50	26.40	26.53	28.90	28.90	28.70	28.83
40 M	29.70	30.00	29.60	29.77	33.00	34.30	33.40	33.57
45 M	35.80	35.80	35.20	35.60	38.10	37.70	38.40	38.07
50 M	39.70	40.00	39.30	39.67	43.20	42.70	43.30	43.07
55 M	42.90	43.00	42.70	42.87	46.10	48.30	47.40	47.27
60 M	45.70	44.40	42.60	44.23	50.90	48.20	51.30	50.13
65 M	48.90	47.90	48.40	48.40	53.40	54.90	55.20	54.50
70 M	54.20	50.70	51.80	52.23	65.60	63.10	58.80	62.50
75 M	61.70	62.00	64.00	62.57	71.60	69.50	74.40	71.83
80 M	70.10	71.30	75.80	72.40	84.60	85.10	82.20	83.97

Table C.7.: Query time depending on number of tuples

List of Figures

List of Tables