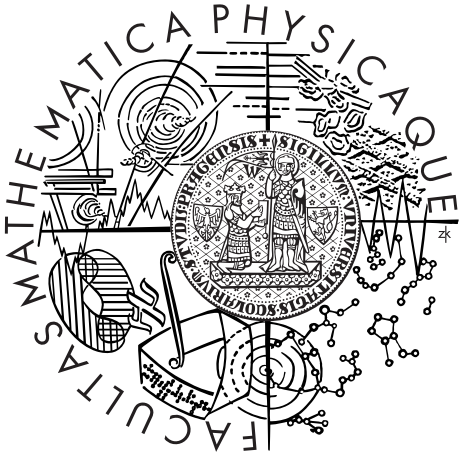
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Marcel Kikta

Evaluating relational queries in pipeline-based environment

Department of Software Engineering

Supervisor of the master thesis: David Bednárek

Study programme: Software systems

Specialization: Software engineering

Prague 2014

I would like to thank my parents for supporting me in my studies and my supervisor David Bednárek for his advice and help with this thesis.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Vyhodnocování relačních dotazů v proudově orientovaném prostředí

Autor: Marcel Kikta

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. David Bednárek, Ph.D.

Abstrakt:

Klíčová slova: SQL, Překladač, Relační algebra, Optimalizator, Bobox

Title: Evaluating relational queries in pipeline-based environment

Author: Marcel Kikta

Department: Department of Software Engineering

Supervisor: RNDr. David Bednárek, Ph.D.

Abstract: This thesis deals with the design and implementation of an optimizer and a transformer of relational queries. Firstly, the thesis describes the database compiler theory. Secondly, we present the data structures and algorithms used in the implemented tool. Finally, the important implementation details of the developed tool are discussed. Part of the thesis is the selection of used relational algebra operators and design of an appropriate input. Input of the implemented software is a query written in a XML file in the form of relational algebra. Query is optimized and transformed into physical plan which will be executed in the parallelization framework Bobox. Developed compiler outputs physical plan written in the Bobolang language, which serves as an input for the Bobox.

Keywords: SQL, Compiler, Relational algebra, Optimizer, Bobox

Contents

1. Introduction

Current processors have multiple cores and their single core performance is improving only very slow because of physical limitations. On the other hand, the number of cores is still increasing and we can assume that this trend will continue. Therefore, development of parallel software is crucial for improvement of the overall performance.

Parallelization can be achieved manually or using some framework designed for it. For example, there are frameworks like OpenMP or Intel TBB. Department of Software Engineering at Charles University in Prague developed its own parallelization framework called Bobox[1].

Bobox is designed for parallel processing of large amounts of data. It was specifically created to simplify and speed up parallel programming of certain class of problems - data computations based on non-linear pipeline. It was successfully used in implementation of XQuery, SPARQL[9], and TriQuery[8] engines.

Bobox consists from runtime environment and operators. These operators are called boxes and they are C++ implementation of data processing algorithms. Boxes use messages called envelopes to send processed data to each other.

Bobox takes as input execution plan written in special language Bobolang[2]. It allows to define used boxes and simply connect them into directed acyclic graph. Bobolang specifies the structure of whole application. It can create highly optimized evaluation, which is capable of using the most of the hardware resources.

Most used databases are relational. They are based on the view of data organized in tables called relations. An important language based on relational databases is Structured query language (SQL[10]) which is used for querying data and modifying content and structure of tables.

Architecture of planned SQL compiler is displayed in Figure 1.1. SQL query is written in text parsed into parse tree, which is transformed into logical query plan (Relational algebra). Relational algebra is then optimized and this form is used for generating physical query plan. Physical plan written in Bobolang is input for Bobox for execution. Besides physical plan, we need to provide implementation of physical algorithms (Bobox operators), as well.

Since SQL is a rather complicated language, the aim of this thesis is only implementing optimization and transformation of logical plan into physical plan. This part is displayed as physical plan generator in Figure 1.1.

The main goal of this thesis is to implement part of SQL compiler. The input is a query written in XML format in form of relational algebra. Program reads input and builds relational algebra tree, which is then checked for semantic errors.

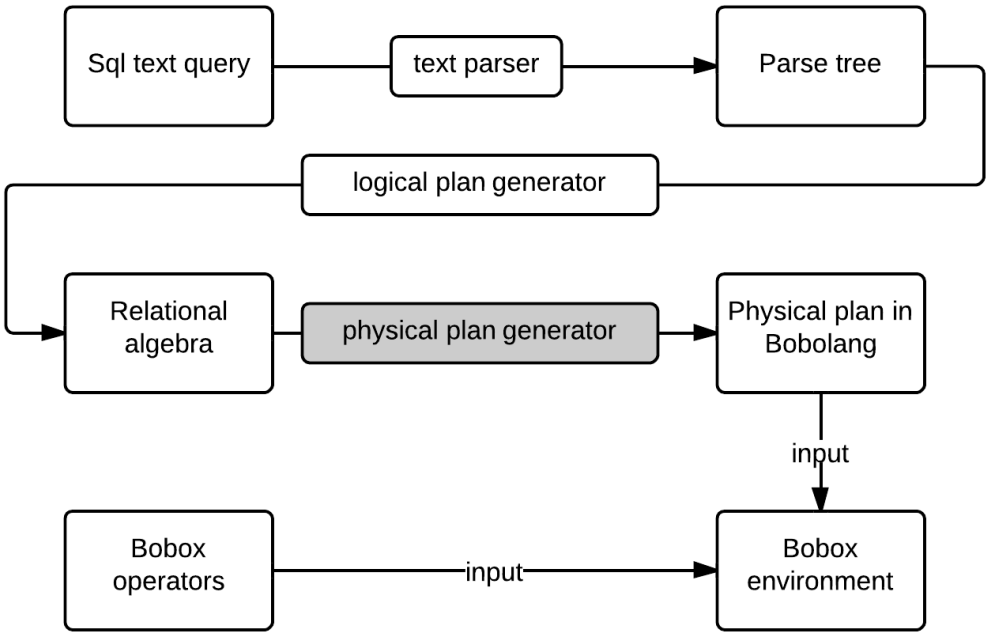


Figure 1.1: SQL compiler architecture.

Then, we improve logical plan by pushing selections down the tree. We generate physical plan from improved relational algebra. In this phase, we assign physical algorithm for every logical plan operator and we also choose the order of joins. The output is an execution plan for Bobox written in Bobolang. The query will be executed in pipeline–based environment[11, 12] of Bobox.

2. Bobox architecture

The purpose of this chapter is description of Bobox framework and Bobolang language.

2.1 Bobox

The purpose of this chapter is description of basic architecture of Bobox. The main source of information for this chapter is the doctoral thesis by Falt [4].

Overall Bobox architecture is displayed in Figure 2.1. Framework consists of Boxes which are C++ classes containing implementation of data processing algorithm. Boxes can be also created as a set of connected boxes. Boxes can have arbitrary number of inputs and outputs. All boxes are connected to a directed acyclic graph.

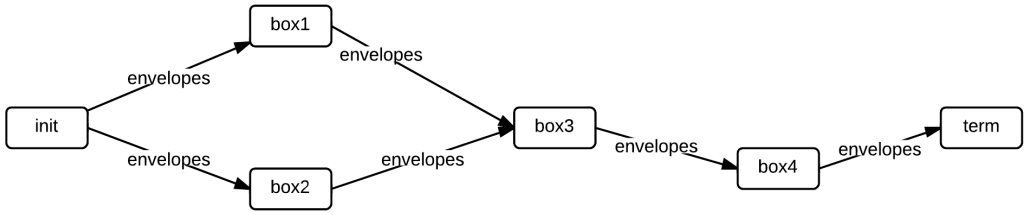


Figure 2.1: Bobox architecture.

Data streams are implemented as streams of data units called envelopes. Envelope structure is displayed in Figure 2.2. It consists of sequence of tuples but internally data are stored in columns. Envelope contains sequence of columns and its data is stored in separate list. In order to read all attributes of the i-th tuple, we have to access all column lists and read its i-th element. Special type of envelope contains a poisoned pill which is sent after all valid data, thus indicating the end of data stream.

There are two special boxes, which have to be in every execution plan:

- *init* - first box in topological order indicating starting box of execution plan.
- *term* - last box in topological order indicating that plan has been completely evaluated.

Evaluation starts with scheduling *init* box, which sends poisoned pills to all of its output boxes that will be scheduled. They can read data from the hard drive or network, process it and send it to other boxes for further processing. Other

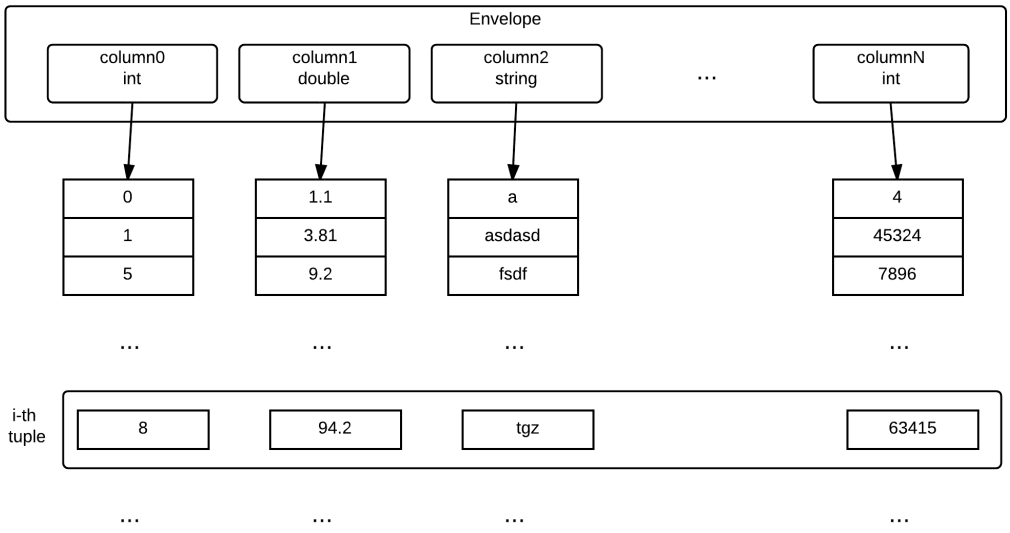


Figure 2.2: Envelope structure.

boxes usually receive data in envelopes in their inputs. Box *term* waits to receive for its every input to receive poisoned pill and then The evaluation ends when the box *term* receives poisoned pill from each of its inputs.

2.2 Bobolang

Syntax and semantics of Bobolang language is explained in this section. The work Falt et al. [2] served as source of information for this text.

Bobolang is a formal description language for Bobox execution plan. Bobox environment provides implementation of basic operators (boxes). Bobolang allows programmer to choose which boxes to use, what type of envelopes are passed between boxes and how the boxes are interconnected.

We can define whole execution plan using operator `main` with empty input and output. An example of complete Bobolang plan:

```
operator main()->()
{
    source()->(int) src;
    process(int)->(int,int,int) proc;
    sink(int,int,int)->() sink;

    input -> src -> proc;
    proc -> sink;
    sink -> output;
```

```
}
```

In the first part, we declare operators and define type of input and output. We provide identifier for every declared operator. Second part specifies connection between declared operators. Code `op1 -> op2` indicates that output of `op1` is connected to input of operator `op2`. In this case, the output type of `op1` has to be equal to the input type of `op2`. Bobolang syntax also allows to create chains of operators like `op1 -> op2 -> op3` with following semantics: `op1 -> op2` and `op2 -> op3`.

There are explicitly defined operators called `input` and `output`. The line `input -> src;` means that input of the operator `main` is connected to the output of operator `src`.

Bobolang also allows to declare operators with empty input or output with the type `()` meaning that they do not transfer any data. These operators transfer only envelopes containing poisoned pills. The box starts working after receiving poisoned pill. Sending the pill means that all data has been processed and the work is done.

Structure of example execution plan can be seen in Figure 2.3. Operators `init` and `term` are added automatically. Operator `init` sends poisoned pill to `source`, which can read data from hard drive or network. Data is sent to the box `process`. Operator `sink` stores data and sends poisoned pill to the box `term` and the computation ends.

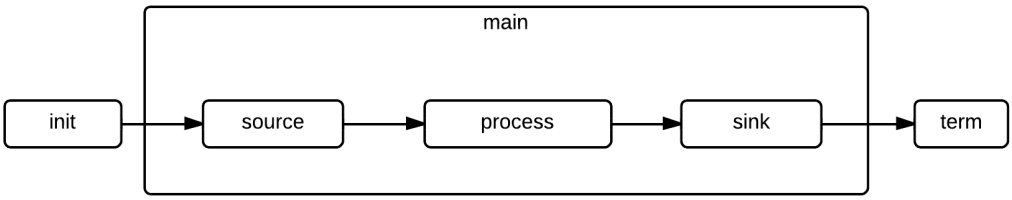


Figure 2.3: Example of execution plan.

3. Related work

The basics of the theory of relational algebra along with its optimizations and physical plan generation are introduced in this chapter. Information for this chapter and compiler implementation was taken from Database systems [3].

3.1 Relational algebra

In this section we introduce and describe relational algebra[3]. Following definitions of relational model can vary depending on used literature.

Definition 1. *Relation is a two dimensional table.*

Definition 2. *Attribute is a column of a table.*

Definition 3. *Schema is the name of the relation and the set of its attributes. For example:* $Movie\ (id, name, length)$.

Definition 4. *Tuple is a row of a relation.*

Relational algebra has atomic operands:

- Variables that are relations.
- Constants which are relations.

In classical relational algebra all operators and expression results are sets. All these operations can be applied also to bags(multi sets). We used relational algebra based on bags in the implemented compiler.

Classical relational algebra operators are:

- Set operations – union, difference, intersection.
- Removing operators – selection, which removes rows and projection that eliminates columns from given relation.
- Operations that combine two relations: all kinds of joins.
- Renaming operations that do not change tuples of the relation but only its schema.

Expressions in relational algebra are called *queries*.

3.1.1 Classical relational algebra operators

Set operations on relations

Sets operations are:

- Union $R \cap_S S$ is a set of tuples that are in R or S.
- Intersection $R \cup_S S$ is a set of tuples that are both in R and S.
- Difference $R -_S S$ is a set of tuples that are in R but not in S.

Considering two relations R and S, if one wants to apply some set operation, both relations must have the same set of attributes. We can also use renaming operations if relations do not have same attribute names.

Projection

Projection operator π_S produces new relation with reduced set of attributes from relation R. Result of an expression $\pi_{(S)A_1,A_3,A_4,...,A_N}(R)$ is relation R with attributes $A_1, A_3, A_4, ..., A_N$. Set version of this operator also eliminates duplicate tuples.

Selection

If the operator selection σ is applied on relation R with condition C, a new relation with the same attributes and tuples, which satisfy given condition, is obtained, for example $\sigma_{A_1=4}(R)$.

Cartesian product

Cartesian product of two sets R and S creates a set of pairs by choosing the first element of pair to be any element from R and second element of pair to be any element of S. Cartesian product of relations is similar. We pair tuples from R with all tuples from S. Relations S and R cannot have attributes with the same name because some columns of expression $R \times S$ could have the same name.

Natural joins

The simplest join is called natural join of R and S ($R \bowtie S$). Let schema of R be $R(r_1, r_2, .., r_n, c_1, c_2, ...c_n)$ and schema of S be $S(s_1, s_2, .., s_n, c_1, c_2, ...c_n)$. In natural join we pair tuple r from relation R to tuple s from relation S only if r and s agree on all attributes with the same name (in this case, $c_1, c_2, ...c_n$).

Theta joins

Natural join forces us to use one specific condition. In many cases we want to join relations with some other condition. The theta join serves for this purpose. The notation for joining the relations R and S based on the condition C is $R \bowtie_C S$. The result is constructed in the following way:

1. Make Cartesian product of R and S.
2. Use selection with condition C.

In principle, $R \bowtie_C S = \sigma_C(R \times S)$. Relations R and S have to have disjunct names of columns.

Renaming

In order to control name of attributes or relation name we have renaming operator $\rho_{A_1=R_1,A_2=R_2,...,A_n=R_n}(R)$. Result will have the same tuples as R and attributes $(R_1, R_2, ..., R_n)$ will be renamed as $(A_1, A_2, ..., A_n)$.

3.1.2 Relational operations on bags

Commercial database systems are almost always based on bags (multiset). The only operations that behave differently are intersection, union, difference and projection.

Union

Bag union of $R \cup_B S$ adds all tuples from S and R together. If tuple t appears m-times in R and n-times in S, then in $R \cup_B S$ t will appear $m + n$ times. Both m and n can be zero.

Intersection

Assume we have tuple t that appears m-times in R and n-times in S. In the bag intersection $R \cap_B S$ t will be $min(m, n)$-times.

Difference

Every tuple t, that appears m-times in R and n-times in S, will appear $max(0, m-n)$ times in bag $R -_B S$.

Projection

Bag version of projection π_B behaves like set version π_S with one exception. Bag version does not eliminate duplicate tuples.

3.1.3 Extended operators of relational algebra

We will introduce extended operators that proved useful in many query languages like SQL.

Duplicate elimination

Duplicate elimination operator $\delta(R)$ returns set consisting of one copy of every tuple that appears in bag R one or more times.

Aggregate operations

Aggregate operators such as sum are not relational algebra operators but are used by grouping operators. They are applied on column and produce one number as a result. The standard operators are SUM, AVG(average), MIN, MAX and $COUNT$.

Grouping operator

Usually, it is not desirable to compute aggregation function for the entire column, i.e. one rather computes this function only for some group of columns. For example, average salary for every person can be computed or the people can be grouped by companies and the average salary in every company is obtained.

For this purpose we have grouping operator $\gamma_L(R)$, where L is a list of:

1. attributes of R by which R will be grouped
2. expression $x = f(y)$, where x is new column name, f is aggregation function and y is attribute of relation. When we use function $COUNT$ we do not need to specify argument.

Relation computed by expression $\gamma_L(R)$ is constructed in the following way:

1. Relation will be partitioned into groups. Every group contains all tuples which have the same value in all grouping attributes. If there are no grouping attributes, all tuples are in one group.
2. For each group, operator produces one tuple consisting of:

(a) Values of grouping attributes.

(b) Results of aggregations over all tuples of processed group.

Duplicate elimination operator is a special case of grouping operator. We can express $\delta(R)$ with $\gamma_L(R)$, where L is a list of all attributes of R.

Extended projection operator

We can extend classical bag projection operator $\pi_L(R)$ introduced in Chapter 3.1.1. It is also denoted as $\pi_L(R)$ but projection list can have following elements:

1. Attribute of R, which means attribute will appear in output.
2. Expression $x = y$, attribute y will be renamed to x.
3. Expression $x = E$, where E is an expression created from attributes from R, constants and arithmetic, string and other operators. The new attribute name is x, for example $x = e * (1 - l)$.

The sorting operator

In several situations we want the output of query to be sorted. Expression $\tau_L(R)$, where R is relation and L is list of attributes with additional information about sort order, is a relation with the same tuples like R but with different order of tuples. Example $\tau_{A_1:A,A_2:D}(R)$ will sort relation R by attribute A_1 ascending and tuples with the same A_1 value will be additionally sorted by their A_2 value descending. Result of this operator is not bag or set since there is no sort order defined in bags or sets. Result is sorted relation and it is essential to use this operator only on top of algebra tree.

Outer joins

Assuming join $R \bowtie_C S$, we call tuple t from relation R or S *dangling* if we did not find any match in relation S or R. Outer join $R \bowtie^{\circ}_C S$ is formed by creating $R \bowtie_C S$ and adding dangling tuples from R and S. The added tuples must be filled with special *null* value in all attributes they do not have but appear in the join result.

Left or right outer join is an outer join where only dangling tuples from left or right relation are added, respectively.

3.2 Optimizations of relational algebra

After generation of the initial logical query plan, some heuristics can be applied to improve it using some algebraic laws that hold for relational algebra.

3.2.1 Commutative and associative laws

Commutative and associative operators are Cartesian product, natural join, union and intersection. Theta join is commutative but generally not associative. If the conditions make sense where they were positioned, then theta join is associative. This implies that one can make following changes to algebra tree:

- $R \oplus S = S \oplus R$
- $(R \oplus S) \oplus T = R \oplus (S \oplus T)$,

where $\oplus$ stands for $\times$, $\cap$, $\cup$, $\bowtie$ or $\bowtie_C$.

3.2.2 Laws involving selection

Selections are important for improving logical plan. Since they usually reduce the size of relation markedly we need to move them down the tree as much as possible. We can change order of selections:

- $\sigma_{C_1}(\sigma_{C2}(R)) = \sigma_{C2}(\sigma_{C1}(R))$

Sometimes we cannot push the whole selection but we can split it:

- $\sigma_{C_1 \ AND \ C_2}(R) = \sigma_{C_1}(\sigma_{C_2}(R))$
- $\sigma_{C_1 \ OR \ C_2}(R) = \sigma_{C_1}(R) \cup_S \sigma_{C_2}(R)$

Last law works only when R is a set.

When pushing selection through the union, it has to be pushed to both branches:

- $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$

When pushing selection through the difference, we must push it to the first branch. Pushing to the second branch is optional. Laws for difference are:

- $\sigma_C(R - S) = \sigma_C(R) - \sigma_C(S)$
- $\sigma_C(R - S) = \sigma_C(R) - S$

The following laws allows to push selection down the both arguments. Assuming the selection σ_C, it can pushed to the branch, which contains all attributes used in C. If C contains only attributes of R, then

- $\sigma_C(R \oplus S) = \sigma_C(R) \oplus S$,

where $\oplus$ stands for $\times$, $\bowtie$ or $\bowtie_C$. If relations S and R contain all attributes of C, the following law can be also used:

- $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie \sigma_C(S)$

3.2.3 Laws involving projection

We can add projection anywhere in the tree as long as it only eliminates attributes, which are not used anymore and they are not in query result.

3.2.4 Laws involving joins and products

We have more laws involving selection that follow directly from the definition of the join:

- $\sigma_C(R \times S) = R \bowtie_C S$
- $R \bowtie S = \pi_L(\sigma_C(\pi_X(R) \times \pi_Y(S)))$. $\pi_A(B)$ renames all attributes of relation B from *attributename* to B_*attributename*, where $A \in \{X, Y\}$ and $B \in \{R, S\}$. C is condition that equates each pair of attributes of R and S, which had the same name before renaming. π_L keeps all columns not used in condition C and renames them back by removing prefix $S\_$ or $R\_$. It also keeps all columns used in C which came from relation R and renames them by removing prefix $R\_$.

3.3 Physical plan generation

Physical plan will be created from optimized logical plan. We generate many physical plans and choose the one with the least estimated cost (run time). This approach is called cost–based enumeration.

For every physical plan we select:

1. an order of grouping and joins.
2. an algorithm for each operator, for example, usage of join based on hashing or sorting.

3. additional operators which are not present in logical plan, for example, we can sort relation in order to use faster algorithm which assumes that its input is sorted.

4. the way in which arguments are passed between operators. We can use iterators or store the result on the hard drive.

3.3.1 Size estimations

Estimates used in this section are taken from Database systems [3]. The costs of evaluation of physical plan are based on the estimated size of intermediate relations. Ideally, we want our estimation to be accurate, easy to compute and logically consistent (size of relation does not depend on how relation is computed). We will present simple rules, which will give us good estimates in most situations. The goal of estimation of sizes is not to predict the exact size of relation as even an inaccurate size will help us with plan generation.

In this section we will use the following conventions:

- $T(R)$ is the number of tuples in relation R.
- $V(R, a)$ is the number of distinct values in attribute a.
- $V(R, [a_1, a_2, ..., a_n])$ is the number of tuples in $\delta(\pi_{a_1,a_2,...,a_n}(R))$

Estimating the size of a projection

Bag projection is the only operator for which the size of its result is computable. It does not change the number of tuples but only their lengths.

Estimating the size of a selection

Selection usually reduces the number of tuples. For selection $S = \sigma_{A=c}(R)$, where A is an attribute of R and c is a constant, we can use the following estimation:

- $T(S) = T(R)/V(R, A)$

The selection involving inequality comparison, $S = \sigma_{A<c}(R)$, is more problematic. On average, half of the tuples satisfy the condition $A < c$ but usually, queries select only a small fraction of them from the relation. Typical inequality will return about a third of the original tuples. Therefore, the recommended estimate is:

- $T(S) = T(R)/3$

Selection with condition in the form C_1 *and* C_2 *and ... and* C_N can be treated as a cascade of simple selections and we can use the estimated size from simpler conditions to compute original selection estimate.

In case we have the condition $S = \sigma_{not(C)}(R)$, the following estimate is recommended:

- $T(S) = T(R) - T(\sigma_C(R))$

Estimates for selections with conditions involving logical disjunction are more complicated. Recommended estimated size of relation $S = \sigma_{C_1 \ or \ C_2}(R)$ is:

- $T(S) = T(R)(1 - (1 - \frac{m_1}{T(R)})(1 - \frac{m_2}{T(R)}))$

In order to use this estimate, we assume that conditions C_1 and C_2 are independent. Variable m_i equals $T(\sigma_{C_1}(R))$, where $i \in \{1, 2\}$. Expression $1 - m_1/T(R)$ is the fraction of tuples which do not satisfy the condition C_1 and $1 - m_2/T(R)$ is the fraction of tuples which do not satisfy the condition C_2. Product of these numbers is the fraction of tuples from R which are not included in the result. The fraction of tuples in S is given by the subtraction of the product from unity.

Estimating the size of a join

Considering the case of the natural join $A = R(X, Y) \bowtie S(Y, Z)$, we use the following estimate:

- $T(A) = \frac{T(R)T(S)}{max(V(R, Y), V(S, Y))}$

We can generalize this rule for joining with multiple attributes. For join $R \bowtie S$, where we join R and S using the attributes $(a_1, a_2, ...a_n)$, we use this estimate:

- $T(R \bowtie S) = \frac{T(R)T(S)}{\prod_{k=1}^{n} max(V(R, a_k), V(S, a_k))}$

When we join using multiple attributes, then the result of this estimate can be smaller than one. Such estimate indicates that the relation will be very small or possibly empty.

If we consider other types of join, e.g. theta join, we can use the following rules for their estimation:

1. The size of the Cartesian product is the product of sizes of relations involved.
2. Equality conditions can be estimated using techniques presented for natural joins.
3. An inequality comparison can be handled like expression $\sigma_{A<c}(R)$. We assume that one third of the tuples will satisfy the condition.

Estimating the size of an union

Estimated size of expression $R \cup_B S$ is a sum of sizes of the relations R and S. Size of the set union $R \cup_S S$ comes from the interval $\langle max(T(R), T(S)), T(R) + T(S) \rangle$. Recommended estimate is the midpoint of given interval.

Estimating the size of intersection

Size of relations $R \cap_S S$ and $R \cap_B S$ can be between 0 and $min(T(R), T(S))$. Recommend estimate is to take half of the size of smaller relation:

- $min(T(R), T(S))/2$.

Estimating the size of a difference

Result of expression $R - S$ can be as big as $T(R)$ and as small as $T(R) - T(S)$. Following estimate can be used for bag or set version of difference operator:

- $T(R - S) = T(R) - \frac{T(S)}{2}$.

Estimating the size of a grouping

Size of the result of expression $\gamma_L(R)$ is $V(R, [g_1, g_2, ..., g_n])$, where g_x are grouping attributes of L. This statistic is almost never available and for this case we need another estimate. Size of $\gamma_L(R)$ can be between 1 and $\prod_{k=1}^{n} V(R, g_k)$. Suggested estimate is:

- $T(\gamma_L(R)) = min(\frac{T(R)}{2}, \prod_{k=1}^{n} V(R, g_k))$

Estimation of duplicate elimination can be handled exactly like grouping.

3.3.2 Enumerating plans

For every logical plan there is an exponential number of physical plans. This section presents ways to enumerate physical plans such that the plan with the least estimated cost can be chosen. There are two broad approaches:

- Top–down: We work down the algebra tree from the root. For each possible implementation for algebra operation at the root, we consider best possible algorithms to evaluate its subtrees. We compute costs of every combination and choose the best one.

- Bottom–up: We proceed up the algebra tree and we enumerate plans based on the plans generated for its subexpressions.

These approaches do not differ considerably. However, one method eliminates plans that other method cannot and vice versa.

Heuristic selection

We use the same approach for generating physical plan as we used to improve logical plan. We make choices based on heuristics, for example:

- If a join attribute has an index, we can use joining by this index.
- If one argument of join is sorted, then we prefer using the merge join algorithm to the hash join algorithm.
- If we compute intersection or union of more than two relations, we process smaller relations first.

Branch–and–bound plan enumeration

Branch–and–bound plan enumeration is often used in practice. We begin by finding physical plan using heuristics and denote its cost C. We can consider other plans for sub–queries. We can eliminate any plan for sub–query with cost greater than C. If we get plan with lower estimated cost than C, we use this plan instead.

The advantage is that we can choose when to stop searchings for a better plan. If C is small, then we do not continue looking for other plan. On the other hand, when C is large, it is better to invest some time in finding a faster plan.

Hill climbing

First, we start with heuristically selected plan. Afterwards, we try to do some changes, for example, change of the order of joins or replacement of the operator using hash table for sort–based operator. Provided that such a plan is found, that no small modification results in a better plan, we choose that physical plan.

Dynamic programming

Dynamic programming is a variation of bottom–up strategy. For each subexpression, we keep only the plan with the least estimated cost.

Selinger–Style Optimization

Selinger–Style Optimization is an improvement of dynamic programming approach. For every subexpression, we keep not only the best plan, but also other plans with higher costs, the results of which are sorted in some way. This might be useful in the following situations:

1. The sort operator τ_L is on the root of tree and we have a plan, which is sorted by some or all attributes in L.

2. Plan is sorted by attribute used later in grouping.

3. We are joining by some sorted attribute.

In this situations, either we do not sort the input or we use only partial sort. This way, we can use faster algorithms which takes advantage of the sorted input.

3.3.3 Choosing the join order

We have three choices how to choose the order of joins of multiple relations:

1. Consider all possible options.

2. Consider only a subset of join orders.

3. Pick one heuristically.

Algorithms, that can be used for choosing join order, are presented in this section.

Dynamic programming algorithm

Dynamic programming algorithm requires the use of a table for storage of the following information:

1. Estimated size of a relation.

2. Cost to compute current relations.

3. Expression of the way how the current relation was computed.

Every input relation with estimated cost 0 is stored in the table. For every pair of relations, we compute their estimated size and cost of join and store it in the table, see Section 3.3.1.

The next step is the computation of join trees of sizes $(3, 4, ...n)$. If we want to consider all possible trees, we need to divide relations in current join R into two non–empty disjoint sets. For each pair of sets, we compute the estimated size and cost of their join to get the join R. We use data already stored in our table for this purpose. The join tree with least estimated cost will be stored in the table. When we are estimating joins of k relations, then all joins of $k-1$ relations must have been already estimated.

Example: for estimating join $A \bowtie B \bowtie C \bowtie D$ we try to join the following trees:

1. A and $B \bowtie C \bowtie D$
2. B and $A \bowtie C \bowtie D$
3. C and $A \bowtie B \bowtie D$
4. D and $A \bowtie B \bowtie C$
5. $A \bowtie B$ and $C \bowtie D$
6. $A \bowtie C$ and $B \bowtie D$
7. $A \bowtie D$ and $B \bowtie C$

Dynamic programming algorithm does not have to enumerate all possible trees, but only left–deep trees. A tree is called left–deep if all of its right children are leafs. Right–deep tree is tree, where all left children are leafs.

Computation differs only in division of processed relation R in two non–empty disjoint subsets. One of the subsets can contain only one relation.

In order to estimate join $A \bowtie B \bowtie C \bowtie D$, we try to join the following subsets:

1. A and $B \bowtie C \bowtie D$
2. B and $A \bowtie C \bowtie D$
3. C and $A \bowtie B \bowtie D$
4. D and $A \bowtie B \bowtie C$

Greedy algorithm

Time complexity of using dynamic programming to select an order of joins is exponential. Thus, we can use it only for small amounts of relations. If less time consuming algorithm is desired, one can use faster greedy algorithm. However, the disadvantage of greedy algorithm is that in some cases it outputs slower plan then dynamic programming algorithm.

Greedy algorithm stores the same information as the dynamic programming algorithm for every relation in the table. We start with a pair of relations R_i, R_j, for which the cost of $R_i \bowtie R_j$ is the smallest. We denote this join as our current tree. For other relations, that are not yet included, we find a relation R_k, such that its join with the current tree gives us the smallest estimated cost. We continue until all relations in our current tree are included. Greedy algorithm will create left–deep or right–deep join tree.

3.3.4 Choosing physical algorithms

To complete a physical plan we need to assign physical algorithms to operations in the logical plan.

Choosing selection algorithms

We try to use index for scanning table instead of reading the whole table. We can use the following heuristics for picking selection algorithm:

- For selection $\sigma_{A \oplus c}$, relation R which has index on attribute A, and constant c, we scan by index instead of scanning the whole table and filtering the result. Sign $\oplus$ denotes $=$, $<$, $\leq$, $>$ or $\geq$.
- More generally, for selections containing condition $A \oplus c$ and selected relation with index on columns A, we can scan by index and filter the result by other parts of condition.

Choosing join algorithms

We recommend to use the sort join when:

1. One or both join relations are sorted by join attributes.
2. There are more joins on the same attribute. For join $R(a, b) \bowtie S(a, c) \bowtie T(a, d)$ we can use sort based algorithm to join R and S. If we assume that $R \bowtie S$ will be sorted by attribute a, then we use second sort join for $R \bowtie S$ and relation T.

If we have join $R(a, b) \bowtie S(b, c)$ and we expect the size of relation to be small and S has index on attribute b, we can use join algorithm using this index. If there are no sorted relations and we do not have any indices on any relations, it is probably the best option to use the hash based algorithm.

Choosing scanning algorithms

Leaf of algebra tree will be replaced by one of scanning operators:

- $TableScan(R)$ – operator reads the entire table.
- $SortScan(R, L)$ – operator reads the entire table and sorts by attributes in the list L.

- $IndexScan(R, C)$ – C is a condition in form $A \oplus c$, where c is a constant, A is an attribute and $\oplus$ stands for $=$, $<$, $\leq$, $>$, $\geq$. Operator reads tuples satisfying condition C using an index on A and the result is sorted by columns on the used index.

- $TableScan(R, A)$ – A is an attribute. Operator reads the entire table using an index on A and the result will be sorted by columns on the used index.

We choose the scan algorithm based on the need of sorted output and availability of indices.

Other algorithms

Usually, sort and hash versions of algorithm are used. The following rules can be used for replacement of algebra operator with physical algorithm:

- We use the hash version of algorithm if the input is not sorted in a way we need or if the output does not have to be sorted.

- We use the sort version of algorithm if we have the input sorted by some of requested parameters or we need sorted output. In case the input is only sorted by some of the needed attributes, we can still use the fast partial sort and apply sort based algorithm.

4. Analysis

Data structures and algorithms used in the implemented compiler are discussed in this chapter.

4.1 Format of relational algebra

In this section, we present relational algebra operators which are the input of the compiler. Our relational algebra is based on bags contains the following operators:

1. Projection – we used extended projection π_L which removes columns, computes new ones using expressions and renames attributes.

2. Table reading operator which is a leaf of the algebra tree. The following arguments need to be provided for this operator:

   - table name
   - information about indices (name, columns and sort order)
   - read columns.

3. Join - we used theta join $\bowtie_C$ operator where C is a condition having the following format:

   - Condition can be empty and in this case join represents Cartesian product.
   - $a_1 = b_1$ *and* $a_2 = b_2$ *and* $a_3 = b_3$ *and...and* $a_n = b_n$, where a_k belongs to the first relation and b_k belongs to the other relation.
   - $a_1 \oplus b \ominus a_2$, where a_1 and a_2 belong to one input and b belongs to other input. Signs $\oplus$, $\ominus$ mean $<$ or $\leq$.

   Apart from the condition C, we need to specify output attributes of the join. These attributes can come from both inputs and we can optionally assign them a new name. Assigning new attribute name is useful when some of the attributes have the same name.

   The other types of joins are not directly supported but they can be replaced with the cross join followed by selection.

4. Anti–join operator which was not presented with other algebra join operators. Output of the expression $R \ltimes_C S$ is a relation from R for which no tuple from S satisfying the condition C exists. We can use join and anti–join to express outer join.

The anti–join can replace the difference operator. The expression $R - S$ equals $R \ltimes_C S$, where C is a condition that equates each pair of attributes of R and S with the same name.

The presence of the anti–join in our relational algebra eliminates the need for the outer join and the difference, thus making the algebra simpler.

Condition C of anti–join $R \ltimes_C S$ has the following format:

- $a_1 = b_1$ *and* $a_2 = b_2$ *and* $a_3 = b_3$ *and...and* $a_n = b_n$, where a_k belongs to first the relation and b_k belongs to the other relation.

In every anti–join, we need to specify its output attributes with optional new name. The anti–join can output only columns from the first relation.

5. Group operator γ_L, where L is a list of group attributes and aggregate functions. Supported aggregate functions are *min*, *max*, *sum* and *count*. The function *avg* is not supported but it can easily computed using *sum* and *count*. All mentioned functions take one input, except for *count* which has an empty input.

   As mentioned before, group operator is a more general version of the duplicate elimination which is not included in our algebra.

6. Sort operator τ_L, where L is a non empty list of attributes with sort directions.

7. Bag union $\cup$. Both input relations have to have the same names and types of attributes. The set union can be computed using bag union and grouping operator for duplicates elimination.

8. Selection used in our algebra does not differ from selection from classical relational algebra.

4.2 Physical algorithms

Enumeration and description of the output algorithms of the compiler is given in the following. We assume that execution environment has enough memory and physical operators do not have to store intermediate results on the hard drive.

The following algorithms are generated by the compiler:

- `Filter` - this algorithm reads input tuples and output tuples satisfying given condition. Output does not have to be sorted in the same way as input.

- `Filter keeping order` - this algorithm reads input tuples and outputs tuples satisfying given condition. Output has to be sorted in the same way as input.

- `Hash group` - operator groups tuples using hash table and for every group of tuples aggregate functions are computed.

- `Sorted group` - operator groups sorted tuples and computes aggregate functions. The input has to be sorted by group attributes.

- `Column operations` - this is an implementation of extended projection algebra operator.

- `Cross join` - operator computes Cartesian product of two relations.

- `Hash join` - operator uses hash table to compute join of two relations R and S with condition C, where C has the following format: $r_1 = s_1$ *and* $r_2 = s_2$ *and* ... $r_n = s_n$. Attributes $(r_1, r_2, ..., r_n)$ belong to the relation R and $(s_1, s_2, ..., s_n)$ are from the relation S.

- `Merge equi--join` - algorithm takes advantage of sorted inputs to compute join with condition C, where C has the same format as the condition in `Hash join`.

- `Merge non equi--join` - operator computes theta join with condition $a_1 \oplus b \ominus a_2$, where a_1 and a_2 belong to the first input and b belongs to the second input. Signs $\oplus$ and $\ominus$ denote $<$ or $\leq$. Input relations have to be sorted by the attributes in the join condition.

- `Hash anti--join` - algorithm computes anti–join with condition C using hash table. Condition C has the same format as the condition in `Hash join`

- `Merge anti join` - operator takes advantage of sorted inputs to compute anti–join with condition C, where C has the same format as the condition in `Hash join`.

- `Table scan` - operator scans the whole table from the hard drive.

- `Scan and sort by index` - operator scans the whole table from the hard drive an using index. Output will be sorted by columns on the used index.

- `Index Scan` - this algorithm uses an index to read tuples from table satisfying given condition.

- `Sort` - this algorithm sorts input. Input can be presorted and in such case, operator does only partial sorting.
- `Union` - an implementation of bag union.

4.3 Architecture

The architecture of implemented tool is displayed in Figure 4.1.

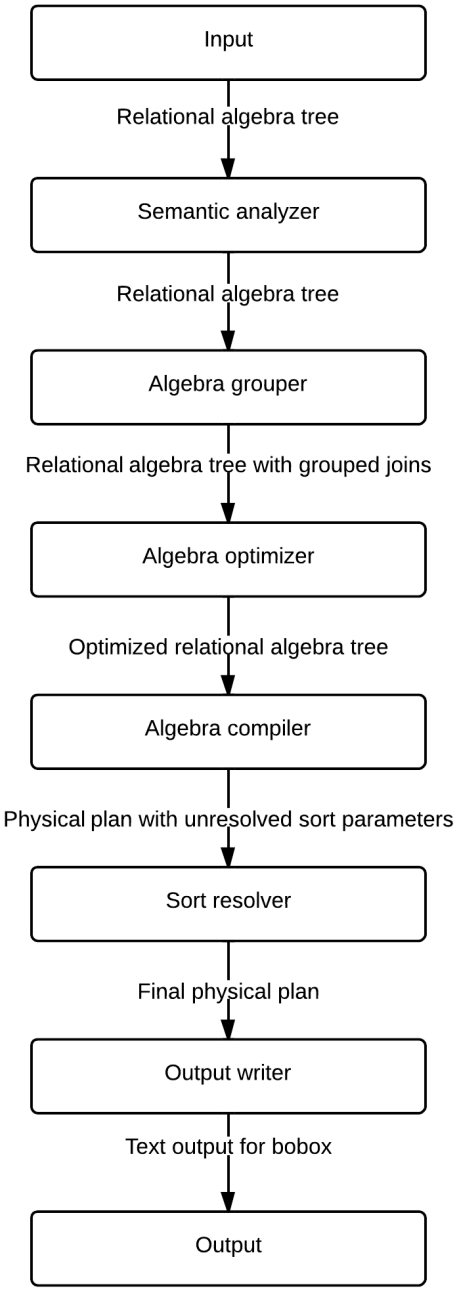


Figure 4.1: Compiler architecture.

The relational algebra tree is read from XML file. We decided for this format for the following reasons:

- XML has a tree–like structure.
- Only the schema needs to be written for the validation.
- There are already implemented tools for parsing.
- There is no need to write an input parser.

The relational algebra tree is checked in the `Semantic analyzer`. This component checks if all of the used attributes are in the input relation. `Semantic analyzer` searches for duplicate named attributes and reports them as an error.

Semantically correct tree is processed by component `Algebra grouper` that groups neighboring joins into one. Thanks to this operation, we can later choose the fastest way to join multiple relations.

Algebra tree with grouped joins is optimized. `Algebra optimizer` pushes the selections down the tree. This component also applies the following operation:

- $\sigma_C(R \bowtie_D S) = R \bowtie_{D\ and\ C} S$, where C has the following format: $r = s$, r belong to R and s belongs to S.

Optimized algebra tree is processed by `Algebra compiler` which generates physical plan. However, this physical plan is not final at this stage. Parameters of its sort operators can represent multiple ways of sorting relation. When sorting relation before grouping, we have multiple possibilities how to sort current relation and we can decide later which way is the best.

Final plan is an output of the component named `Sort resolver`. This component resolve unknown sort order in sort operators and produces final plan which is converted to Bobolang language.

Implemented tool will be the back end of the compiler and it does not check the types of columns. We assume that types will be handled by the front end which parses the text. Types of columns are only copied to the output and we assume that the types of columns do not contain any errors.

4.4 Data structures

Data structures used in the implemented tool are presented in this section.

Relational algebra is stored in the polymorphic tree. Every node stores its parameters, pointer on the parent in the tree and pointers on its children node. No other structure was considered for this representation since this is an efficient

way to store logical plan. It allows to easily add and remove relational algebra operators. The example of this representation can be found in Figure 4.2. It is representing simple query reading the whole table. Read relation is grouped and aggregation functions are computed. The result is sorted at the end. Leaf of the tree stores the following information:

- List of indices on current table.
- List of columns with their names, types and number of unique values.
- Size of the relation.

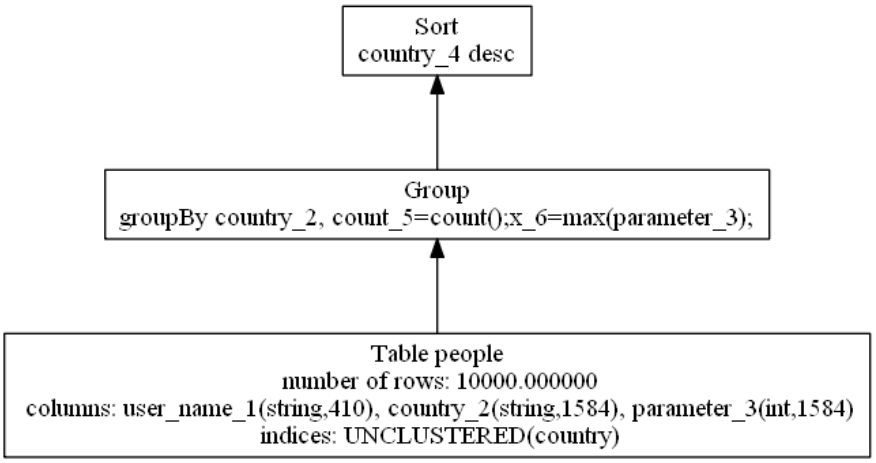


Figure 4.2: Example of relational algebra structure.

We chose the same structure for physical plan. The advantage of storing physical plan into polymorphic tree is the ability to easily add a new root node. The example of this representation can be found in Figure 4.3. This Figure contains one of the possible physical plans for relational algebra shown in Figure 4.2. We used the algorithm `Table` scan for reading the table. Afterwards, the input is hashed by requested columns. Result is sorted in the `Sort` operator. All of the node store additional information like output attributes, estimated run time and size of the output relation.

Physical and logical plans contain expressions, as well. The expressions are stored in polymorphic expression tree. Example of this structure can be found in Figure 4.4.

More complicated structure was used for storing sort parameters. This structure is stored in every physical sort operator to determine how the relation can be sorted.

If one wants to use sort based group operator grouping by two columns, multiple possibilities for sorting relation are available. In order use sort based algorithm for evaluating expression $\gamma_{x,y}(R)$, the relation can be sorted in four possible ways:

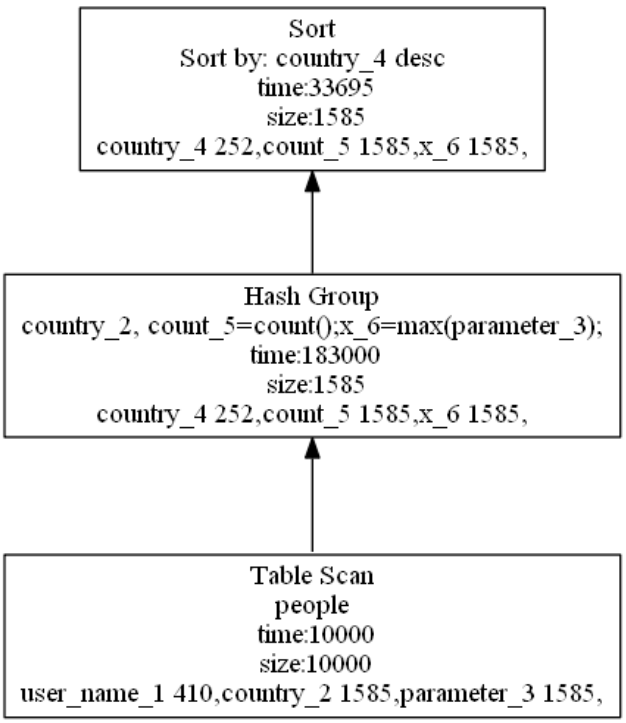


Figure 4.3: Example of physical plans structure.

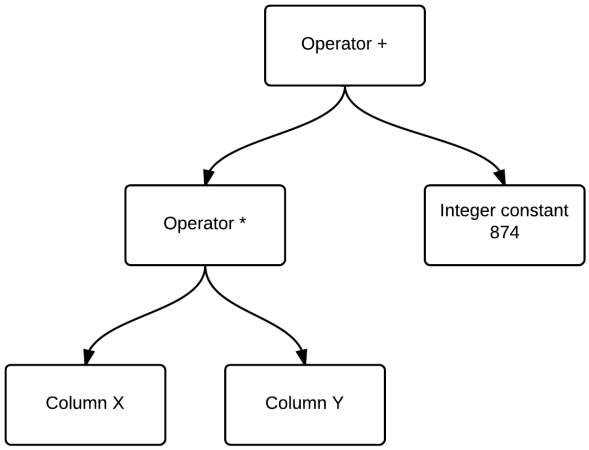


Figure 4.4: Example of expression tree representing the expression $X * Y + 874$.

- $x : A, y : A$
- $x : A, y : D$
- $x : D, y : A$
- $x : D, y : D$

A means ascending and D is the abbreviation for descending.

There are multiple possibilities of sorting relation R before applying $R \bowtie_{r_1=s_1 \ and \ r_2=s_2} S$. Relation R can be sorted in the following ways:

- r_1, r_2
- r_2, r_1

The order for sorting columns is arbitrary.

We also want to store information about equality of sort column. After the application of merge join $R \bowtie_{r_1=s_1} S$, the result can be sorted by r_1 or s_1. All these requirements were used to design structure for storing sort parameters without enumerating all possible sort orders.

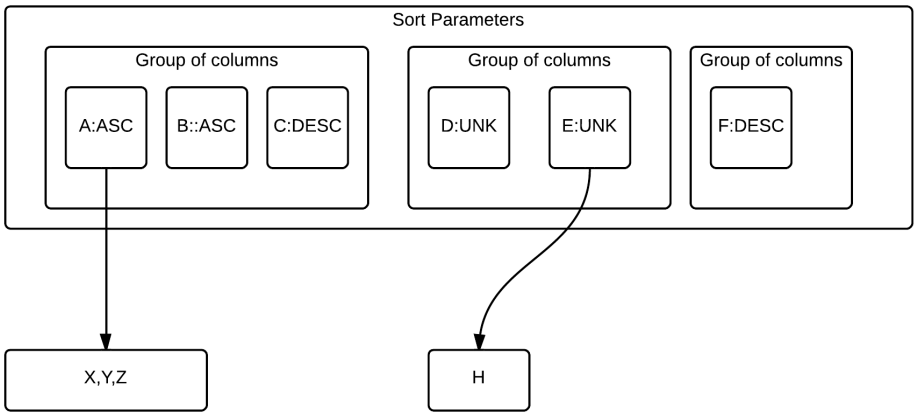


Figure 4.5: Structure for storing parameters for sort.

In Figure 4.5, we display an example of sort parameters which sort by 6 columns. The structure contains one or more groups of columns. The order of the groups of columns cannot be changed. The order of columns in groups is arbitrary. For example, the column F has to be on sixth place but column E can be on forth or fifth place. Every column contains information about sort order: ASC (ascending), $Desc$ (descending) or UNK (unknown – can be ascending or descending). We store list of equal attributes for every column. In case a projection operator removes columns A from relation, we can replace removed attribute with attributes X, Y or Z in sort parameters. The Figure 4.5 represents many sort order possibilities. Here we enumerate only some of them:

1. $A : ASC, C : DESC, B : ASC, H : DESC, D : ASC, F : DESC$
2. $C : DESC, B : ASC, Z : ASC, H : DESC, D : DESC, F : DESC$
3. $B : ASC, C : DESC, A : ASC, E : ASC, D : DESC, F : DESC$
4. $C : DESC, B : ASC, Y : ASC, D : ASC, H : ASC, F : DESC$

4.5 Optimization

In this section we describe algebra optimization, which was implemented to improve logical plan.

Logical plan needs to be prepared before one proceeds with optimizations. We group joins algebra nodes and expressions connected with *and* and *or*. We work down the algebra tree. If we find join we convert to it the grouped join node. If one of its children is a join, then we merge it with grouped join. This representation is used for choosing faster order join. Conditions are grouped in the same way. We create $AND(a = 2, b = 2, c = 2)$ from expression tree $a = 2$ *and* $(b = 2$ *and* $c = 2)$. This representation is useful for splitting condition into simpler conditions.

An important optimization was implemented: pushing selection down the tree. Every selection is split into selections with simpler conditions. Every selection is moved down the tree as much as possible. In this phase, we used the following rules (σ_C is being pushed down):

1. $\sigma_C(\sigma_D(R)) = \sigma_D(\sigma_C(R))$
2. $\sigma_C(\pi_L(R)) = \pi_L(\sigma_C(R))$, it works only if C does not contain new computed columns in extended projection. We also need to rename columns in condition C in case projection renamed some of the condition columns.
3. $\sigma_C(R \bowtie_D S)$ can be rewritten as
   (a) $\sigma_C(R) \bowtie_D S$ if C contains only columns from R.
   (b) $R \bowtie_D \sigma_C(S)$ if C contains only columns from R.
   (c) $R \bowtie_{D\ and\ C} S$ if C is in form $a = b$ where a belong to the relation R and b comes from the relation S.
4. $\sigma_C(R \ltimes_D S) = \sigma_C(R) \ltimes_D S$
5. $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$

4.6 Generating physical plan

We tried to choose the easiest method for generating plans. The two options were the heuristic method and dynamic programming which, we assume, would probably need the same amount of the source code.The dynamic programming was chosen since it is expected to give better results. Using this method, we generated all possible plans for each node and chose the fastest plan.

We process logical plan from leafs. For every leaf, we generate all possible physical algorithms and we insert resulting plans into heap, where we keep c fastest plans for current node. Variable c is a constant set in compiler. For every algebra node we use plans generated in its children to generate new plan. This way we continue up the algebra tree to the root.

Physical plans are compared based on estimated run time. Every operator stores its estimated run time. The sum of run times of all the operators is estimated run time of the whole physical plan. Among the most important are those equations which compute estimated time for physical algorithm. These equations depend on size of input relation. Modifying them can result in getting better physical plans. For example, if physical algorithm `Hash join` takes too much time due to accessing random parts of memory, we can modify estimated times so sort with merge join will be preferred.

Crucial is the information about sizes of tables. If they are not provided in the input, we use default values and physical plan will be probably worse. Other important parameter is the number of unique values of attribute. Size of join depends on it and since joins usually take significant amount of time, it is important to have as precise values as possible.

4.6.1 Join order selecting algorithm

In section 3.3.3 we presented algorithm for choosing the join order. We should choose order of joins and then assign join algorithms. These operations are done in one phase for the following reason: in case we do not have information about the sizes of tables, we cannot determine join order because all orders have the same estimated run time. In this situation, we can start by joining relations which are sorted to get a faster plan.

We use two algorithms, dynamic programming and greedy algorithm. The version of used dynamic programming algorithm enumerates all possible trees. This algorithm can provide us very good plans but it has exponential complexity. We use it only if number of joined relations in grouped join node is small. For joining more relations, we use faster greedy algorithm which generates only left or right deep trees. Contrary to the dynamic programming algorithm, time complexity of greedy algorithm is only polynomial.

Input in both algorithms is a set of plans for every input join relation.

Dynamic programing for selection join order

We use a variation of algorithm described in section 3.3.3. Input relations are numbered from 1 to n. We used table for storing plans with key that is non–empty

subset of the set $1..n$.

We only store k best plans in every table cell, where k a is constant set in the compiler. It represents the best plans that were created by joining the inputs stored in the key of the table entry.

We begin by storing input plans into table entry identified by the set containing the number of the input relation. In the first iteration, we fill entries of tables with key that has two values by combining plans from entries with the key size 1.

We continue by computing plans for entries with the key sizes $3, 4...n$. The key of the current entry is spit in all possible pairs of non empty disjunctive subsets. We take plans from table entries identified with subsets and we generate new plans by combining them. We only store k fastest plans in the current table cell. We continue until we compute plans for table entry identified by the key $1..n$. These plans are our result.

Time complexity is at least exponential since we generate all subsets of n relations, this value is 2^n.

Greedy algorithm for selection join order

This is a variation of algorithm described in section 3.3.3. We begin by creating joins for every pair of relations. We choose the fastest k trees from created pairs.

In every iteration, we generate new trees by adding new relation to every tree. Only k best trees are allowed to continue to the following iteration. We iterate until we create join tree containing every input relation.

Time complexity is $O(n^2)$. At most n new trees are generated from every tree in each iteration. However, we keep only k of the for them for next iteration. The number of iterations is $n-1$ because all trees grow by one in every iteration. The number k is a constant and it does no have any effect on time complexity.

4.6.2 Resolving sort parameters

After the physical plan has been generated, a decision has to be made about the sort operator parameter that shall be used. We work down the tree and store the information about the sort order of the relation. Based on that, we adjust sort parameters or just choose the arbitrary sort order if possible.

5. Implementation

Here, we present implementation details in the developed software. Its functionality is described on a selected example. More implementation details can be found on CD in the generated doxygen [7] documentation. We will use the following example to describe optimizations and plan generation:

```
select
    l_orderkey,
    sum(l_extendedprice*(1-l_discount)) as revenue,
    o_orderdate,
    o_shippriority
from
    customer,
    orders,
    lineitem
where
    c_mktsegment = '[SEGMENT]'
    and c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate < date '[DATE]'
    and l_shipdate > date '[DATE]'
group by
    l_orderkey,
    o_orderdate,
    o_shippriority
order by
    revenue desc,
    o_orderdate;
```

This example was taken from TPC benchmark TM H [5]. Parameters $[DATE]$ and $[SEGMENT]$ are constants. Tables do not contain any indices in this benchmark. Columns starting with `o_` are from the table `order`, columns beginning with `l_` are from the table `lineitem` and columns with prefix `c_` belong to the table `customers`.

5.1 Input

The input is XML file containing logical query plan. In this section, we describe its structure.

5.1.1 Sort

The root of every algebra tree contains the sort operator, even if the output does not have to be sorted. In this case, sort has empty parameters. The following example displays the sort operator structure:

```
<?xml version="1.0" encoding="utf-8"?>
<sort xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="algebra.xsd">
  <parameters>
    <parameter column="revenue" direction="desc" />
    <parameter column="o_orderdate" direction="asc" />
  </parameters>
  <input>
  ...
  <input>
</sort>
```

The sort is the root element of XML file. Inside the <parameters> element, we can find sort parameters specifying columns and the sort order. This example represents sort $\tau_{revenue:desc,o\_orderdate:asc}(...)$. The element <input> should contain other algebra tree node.

5.1.2 Group

Next example shows the group node:

```
<group>
  <parameters>
    <group_by column="l_orderkey"/>
    <group_by column="o_orderdate" />
    <group_by column="o_shippriority"/>
    <sum argument="x" output="revenue"/>
  </parameters>
  <input>
  ...
  <input>
</group>
```

This node represents expression $\gamma_{l\_orderkey,o\_orderdate,o\_shippriority,x=sum(x)}(...)$. Group element must have at least one <group_by> parameter or at least one aggregate function. Another algebra operator should be inside the element <input>.

5.1.3 Selection

The selection is presented in the following example:

```
<selection>
  <parameters>
    <condition>
      <lower>
        <constant type="date" value="today"/>
        <column name="l_shipdate"/>
      </lower>
    </condition>
  </parameters>
  <input>
   ...
  </input>
</selection>
```

This example represents the following expression: $\sigma_{today<l\_shipdate}$. The element <condition> can contain multiple conditions connected by <and> or <or> elements. Input algebra supports operators =, < and ≤. Only <column> or <constant> element can be present in the leafs of the expression tree. Possible call of a boolean function from the selection is presented in the following example:

```
<condition>
  <boolean_predicate name="like">
    <argument>
      <column name="x"/>
    </argument>
    <argument>
      <constant type="int" value="445" />
    </argument>
  </boolean_predicate>
</condition>
```

Used boolean predicate has to be supported by runtime (Bobox operators). The compiler does not check if called predicate does exist.

5.1.4 Join

The join operator without condition represents cross join. We can use join with multiple equal conditions or with one simple unequal condition. The first example

contains equal conditions:

```
<join>
  <parameters>
    <equal_condition>
      <equals>
        <column name="a" />
        <column name="b" />
      </equals>
      <equals>
        <column name="c" />
        <column name="d" />
      </equals>
    </equal_condition>
    <column name="a" input="first" />
    <column name="b" input="second" />
    <column name="c" input="first" />
    <column name="d" input="second" newName="e" />
  </parameters>
<input>
...
</input>
```

This example represents join with condition $a = b$ *and* $c = d$. In join equal condition, the first column has to be from the first relation and the second column comes from the second relation. The columns a and c are from the first input and b and d are from the other one. Joins do not copy all columns to the output. We have to specify non–empty sequence of output columns. For every column we have to provide the name and the input number. We can also rename the join output column by using the attribute *newName*. In the last example we renamed the output column d to e.

The next example shows join with inequality condition:

```
<join>
  <parameters>
    <less_condition>
      <and>
        <lower_or_equals>
          <column name="a1" />
          <column name="b" />
        </lower_or_equals>
```

```
        <lower_or_equals>
          <column name="b"/>
          <column name="a2"/>
        </lower_or_equals>
      </and>
    </less_condition>
    <column name="a1" input="first"/>
    <column name="b" input="second"/>
    <column name="a2" input="first"/>
</parameters>
<input>
...
</input>
</join>
```

This example represents join with the condition $a1 \leq b \leq a2$. First element <lower_or_equals> has to contain the column from the first relation followed by the column from the second relation. On the other hand, the second element <lower_or_equals> must have these columns in the reversed order. The element <lower_or_equals> can be replaced by the element <lower>. The rules for specifying the output column are the same as in the join with equal conditions. The element <input> should contain two algebra operators.

5.1.5 Anti–join

```
<antijoin>
  <parameters>
    <equal_condition>
      <equals>
        <column name="d"/>
        <column name="b"/>
      </equals>
    </equal_condition>
    <column name="d"/>
  </parameters>
<input>
...
<input>
</antijoin>
```

This is an example of anti–join with the simple condition $d = b$. The structure is almost the same as join with equal conditions. We can output columns only from the first relation and these columns can be renamed.

5.1.6 Table

Table operator is the leaf of the algebra tree. We specify here the name of read table, its columns and indices. The number of rows can be given in order to get a better plan. If we do not have that information, we assume that a table has 1000 tuples. For every column we have to specify its name and type. Other optional parameter is *number_of_unique_values* which is important for estimating the size of join. If this information is missing, we will assume that *number_of_unique_values* is the size of the table to the power of $\frac{4}{5}$. This assumption is only experimental since the number of unique values comes from the interval $\langle 0, table\ size \rangle$. There are two types of indices: clustered and unclustered. Every table can have only one clustered index. For each index, we have to provide the attribute names and the sort order. The following example contains the table algebra node:

```
<table name="orders" numberOfRows="1500000">
  <column name="o_orderdate" type="int"/>
  <column name="o_shippriority"
  type="int" number_of_unique_values="30000"/>
  <column name="o_orderkey" type="int"/>
  <column name="o_custkey" type="int" />
  <index type="clustered" name="index">
    <column name="o_orderdate" order="asc" />
    <column name="o_shippriority" order="asc" />
  </index>
</table>
```

5.1.7 Union

The union operator does not have any parameters and the columns from both inputs must have the same names. Example:

```
<union>
  <input>
  ...
  </input>
</union>
```

5.1.8 Extended projection

The following example of an extended projection represents expression $\pi_{l\_orderkey,o\_orderdate,o\_shippriority,x=l\_extendedprice*(1-l\_discount)}(...)$:

```
<column_operations>
  <parameters>
    <column name="l_orderkey"></column>
    <column name="o_orderdate"></column>
    <column name="o_shippriority"></column>
    <column name="x">
      <equals>
        <times>
          <column name="l_extendedprice" />
          <minus>
            <constant type="double" value="1" />
            <column name="l_discount" />
          </minus>
        </times>
      </equals>
    </column>
  </parameters>
  <input>
  ...
  </input>
</column_operations>
```

The extended projection contains a list of columns. Newly computed columns contain element <equals> with expression. Expression tree can contain arbitrary function call which has to be supported by Bobox operators. Function call is explained in the following example:

```
<column_operations>
  <parameters>
    <column name="x">
      <equals>
        <aritmetic_function name ="sqrt"
        returnType="double">
          <argument>
            <constant type="double" value="2" />
          </argument>
        </aritmetic_function>
```

```
        </equals>
      </column>
    </parameters>
    <input>
      ...
    </input>
</column_operations>
```

Last example computes new column named x with values $\sqrt{2}$.

5.2 Building relational algebra tree

In this section, we describe the construction of logical plan as well as the structure for its storage. Relational algebra operators are represented by children of the abstract class `AlgebraNodeBase`. It has the following abstract subclasses:

- `UnaryAlgebraNodeBase` – abstract class for algebra operator with one input.
- `BinaryAlgebraNodeBase` – abstract class for algebra operator with two inputs.
- `GroupedAlgebraNode` – abstract class for algebra operator with the variable number of inputs.
- `NullaryAlgebraNodeBase` – abstract class for algebra tree leafs.

All algebra operators are children of one of the mentioned classes. Every operator has pointer to its parent and smart pointers to its children, if they exist.

Expressions in algebra nodes are represented by polymorphic trees. All expression nodes are children of the class `Expression`.

We used the visitor pattern for manipulating and reading expression and algebra tree. All nodes (algebra and expression) contain the method `accept` which calls visitor method on the class `AlgebraVisitor`/`ExpressionVisitor`. All classes, which manipulate the algebra tree, are children of class `AlgebraVisitor`.

In Figure 5.1 we presented the example of algebra tree of the query presented in the beginning of this chapter. We have transformed it to cross joins of three tables. After cross joins we apply selection with condition located in the `where` clause. The obtained result is used to compute new column and to apply grouping operator computing aggregate functions. Output of the query is sorted. In table reading operator, we can see additional information like the name of the table. Every attribute has the suffix $\_-1$ which is the unique identifier of the column.

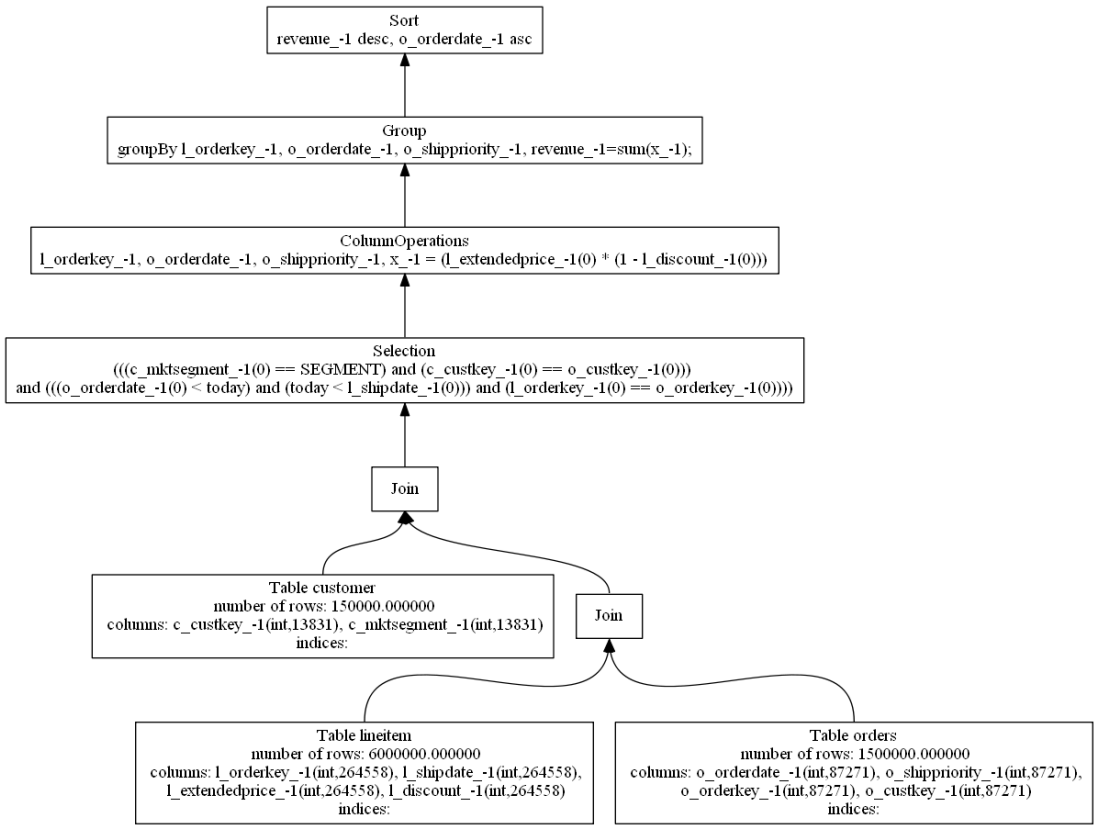


Figure 5.1: Example of algebra tree.

After the algebra tree is built from XML, the unique identifiers are not assigned and they contain default value -1. The inputs, from which the columns originate, are represented by the numbers in parentheses. Input operators are numbered from 0. All columns in used selection come from the zeroth input.

We used library Xerces version 3.1.1 [6] for parsing and validating the input XML file. It parses the input file and creates the DOM tree. This tree has to be validated against XML schema. Parsing and validation of XML is performed in the class `XmlHandler`.

Every algebra tree has to have the sort operator on the top. We call `Sort` constructor with root element as an argument. This method copies all the information from DOM tree and calls the method `AlgebraNodeBase::constructChildren` which decides what constructor to call on children of processed node. This way we recursively build the algebra tree.

5.3 Semantic analysis and node grouping

Semantic checking is processed by the class `SemanticChecker`, which checks if columns used in expressions exist and if the output columns of each operator have unique name. During this checking we assign unique identifier to every column.

After this phase, we do not need attribute names anymore, we use only the unique identifiers.

Logical plan is visited by `GroupingVisitor`. In this phase, joins represented by the class `Join` are replaced by the grouped join represented by the class `GroupedJoin` with two or more input relations. We apply `GroupingExpressionVisitor` on every expression. The `GroupingExpressionVisitor` groups expressions with *and* and *or* operators. This step simplifies splitting condition into sub conditions.

5.4 Algebra optimization

We need to prepare logical tree for optimizing it by pushing down selections. To do this, we split selection into smaller conditions using the rule:

- $\sigma_{A\ and\ B}(R) = \sigma_A(\sigma_B(R))$

A chain of selections is created from every selection. This operation performed by `SelectionSpitingVisitor`.

Afterwards, we apply `SelectionColectingVisitor` which stores pointers of all selections in the relational algebra tree. These pointers are input into `PushSelectionDownVisitor`. It pushes all selections down the tree as much as possible and also converts cross joins into regular joins if we have selection with equal condition. At this moment, we have optimized tree but it still contains the selection chains. To resolve this problem, we apply `SelectionFusingVisitor` that applies the following rule to the tree:

- $\sigma_A(\sigma_B(R)) = \sigma_{A\ and\ B}(R)$

Optimized algebra tree is depicted in Figure 5.2. Contrary to the case of Figure 5.1, this tree has grouped join with three input relations. The selection above joins was split and moved down the tree. Some parts of the condition became parts of the join condition, others were pushed down to one of the branches of grouped join. It can be seen that new tree has columns with assigned unique identifiers which eliminate the need of the input number for each column.

This output is optimized algebra tree. Needless to say, more optimizations can be implemented in order to improve logical plan.

5.5 Generating plan

Final logical plan will be processed by `AlgebraCompiler` which outputs n best plans. Parameter n is a constant in `AlgebraCompiler` represented by variable `NUMBER_OF_PLANS`.

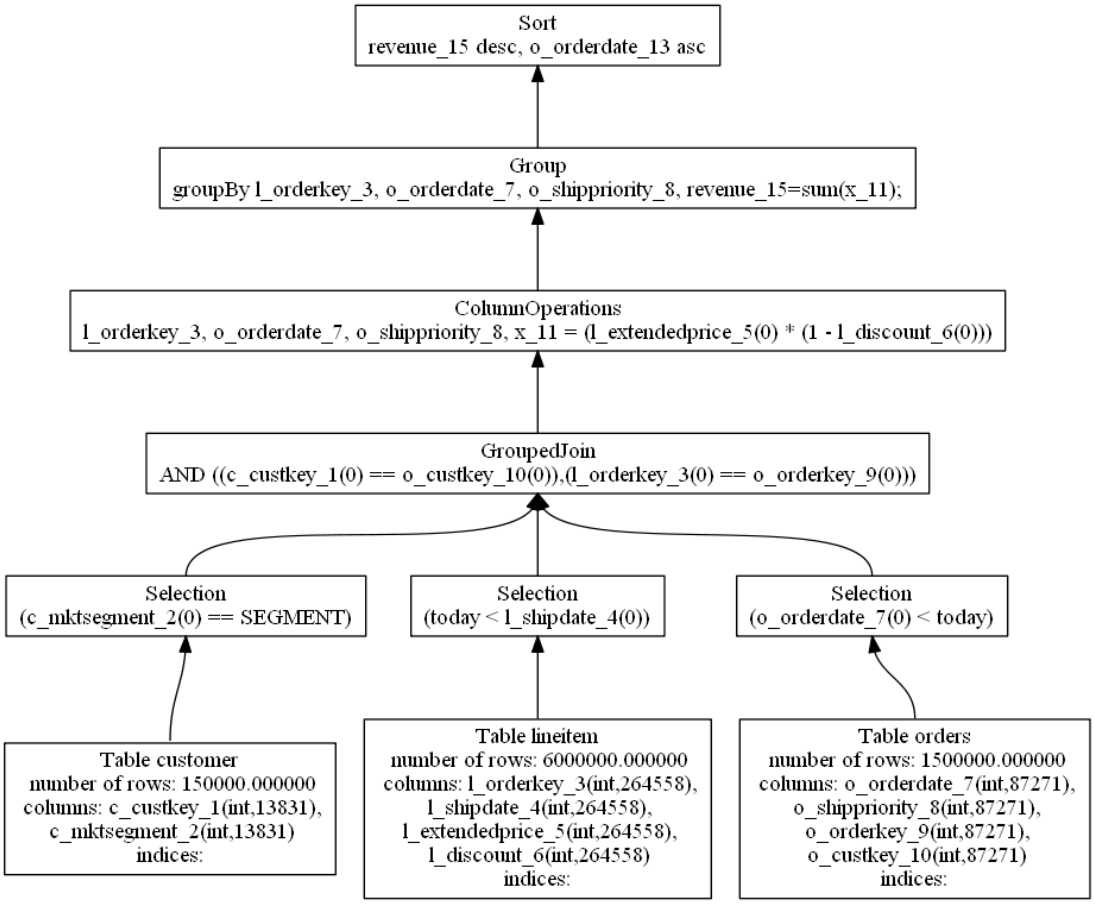


Figure 5.2: Example of optimized algebra tree.

This visitor visits node of algebra tree, then it calls itself on its children. We use generated plans for child nodes to create plans for the current node. After that, we store the best plans in the variable `result`, relation size in the variable `size` and output columns in the variable `outputColumns`.

For each algebra node we generate all possible algorithms. Generated plans are stored in the variable `result`. This variable stores a max–heap, where plans are compared by their overall time complexity. This time complexity is computed as a sum of time complexities in all physical operators in the current plan. Maximal size of the heap is n. If there are more than n plans in the heap, we remove plan in the root of the heap.

Both join order algorithms can be found in the method `visitGroupedJoin`. If the number of join relations is smaller than k, we use the dynamic programing algorithm to estimate order of join. If we have more relations to join, we use the greedy algorithm. Constant k is represented in the variable `LIMIT_FOR_GREEDY_JOIN_ORDER_ALGORITHM`. Both join algorithms call the method `join` which combines plans and generates all possible plans.

Physical plan is represented as polymorphic tree.

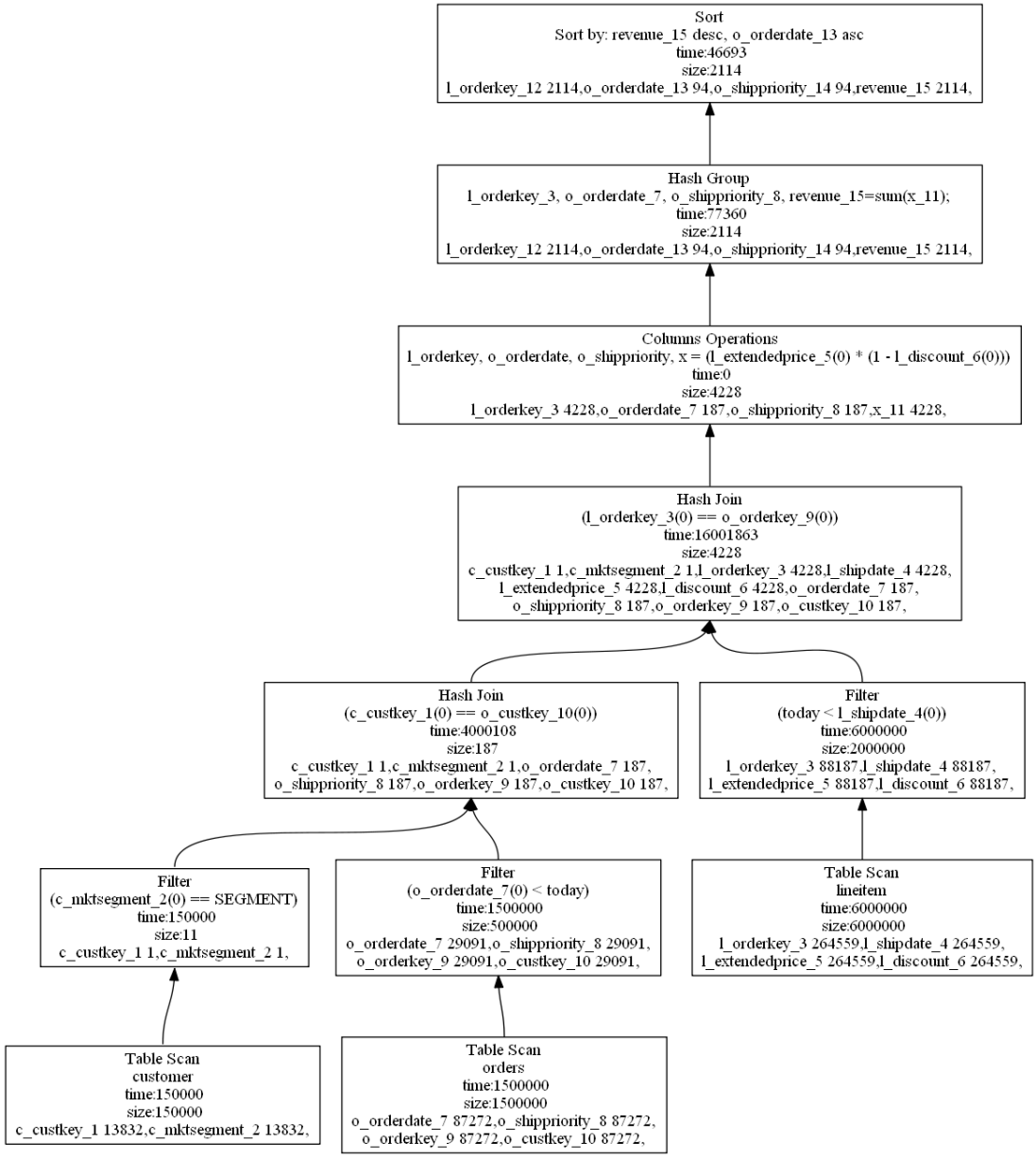


Figure 5.3: Example of physical plan.

The best generated physical plan for query presented in the beginning of this chapter is drawn in Figure 5.3. Each operator contains estimated size and run time. Output columns with their unique identifiers and estimated number of unique values are given on the bottom of the physical operators. Since read tables do not contain any indices we have to read all the tables and filter the results. After that, we can only use hash join because sorting relations for merge join would be to expensive and nested loop join is not supported by runtime or compiler. From the result we compute new columns and use the hash group algorithm. We did not use sorted group as input is not sorted and output has to be sorted by other than group column.

Physical operators are chosen according to their estimated time complexity

which is computed from the estimated size of relation. In the class `TimeComplexity`, we have static functions which compute time complexity for each operation. It also contains constants used in these functions. We assume that this constants or whole functions need to be improved. This improvement can be achieved in the future if the tests and measurements of the evaluation queries in Bobox are performed. At the time of submitting this thesis, the runtime environment is not fully functional.

5.6 Resolving sort parameters

Sort parameters structure in Figure 4.5 is represented by the class `PossibleSortParameters`. Every group of columns is stored in the class `SortParameters`. The class `SortParameter` is used to store column name and sort direction.

We take generated plans from the class `AlgebraCompiler`. Sort parameters of sort nodes need to be resolved. Two plans also can contain the same physical operator. Hence, we need to clone plans in order to assure that no algorithm object is used in two or more plans.

We used `CloningPhysicalOperatorVisitor` for cloning and we resolve generated plans in `SortResolvingPhysicalOperatorVisitor`. Physical plan with unresolved sort parameters can be seen in Figure 5.4. It contains two sort algorithms. Left one has the following possible parameters:

- $a : both, c : both$
- $c : both, a : both$

Right sort algorithm has these sort parameters:

- $b : both, d : both$
- $d : both, b : both$

At the time of generating this algorithm, we did not know what order was the best to choose. After merge join, the plan has to be sorted by $d : desc, a : asc$. At the top of the tree, we generated partial sort which does not do anything because relation is already sorted. It only indicates that we choose $d : desc, a : asc$ from all sort parameter possibilities and we do not have to do any additional sorting.

`SortResolvingPhysicalOperatorVisitor` works down the tree. Variable `sortParameters` is used for storage of information how the input of current node was sorted. We adjust sort parameters of the sort algorithm using this variable. The adjusted plan is drawn in Figure 5.5.

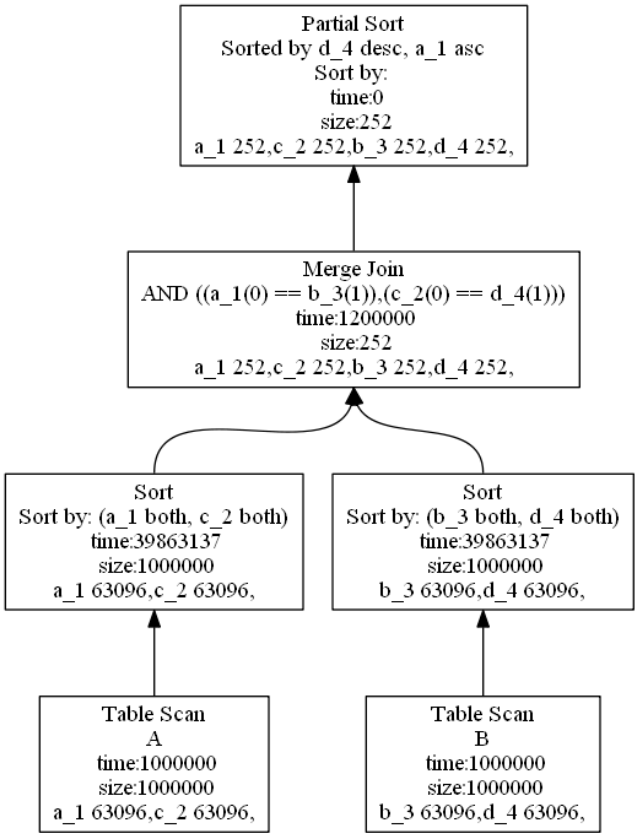


Figure 5.4: Example of physical plan.

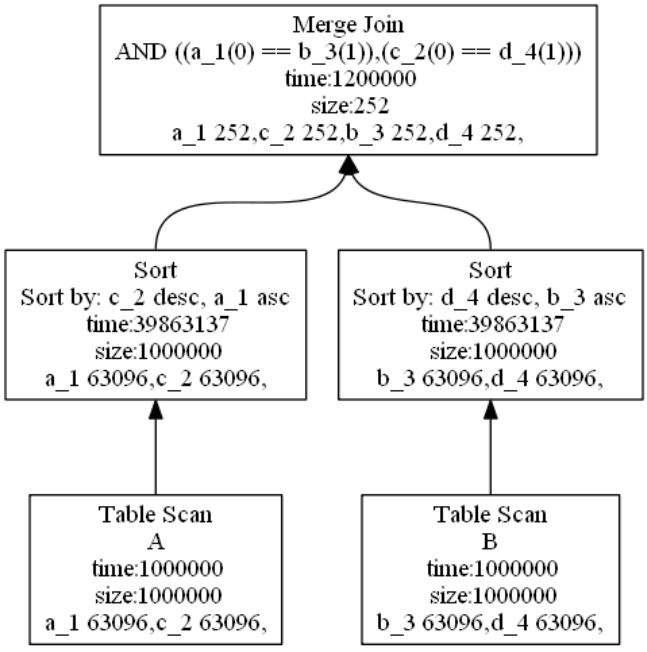


Figure 5.5: Example of final physical plan.

Output of the query has to be sorted by $d : desc, a : asc$. Visitor sets that input of partial sort has to be sorted by $d : desc, a : asc$. In the merge join we compute that left input has to be sorted by $c : desc, a : asc$ and right input has

to be sorted by $d : desc, b : asc$. We choose correct sort parameters in the sort algorithms using this information. There can be situations where we use sort based algorithm but output does not have to be sorted and thus we can choose arbitrary order of sort parameters.

5.7 Output

Output in Bobolang is generated by `BoboxPlanWritingPhysicalOperatorVisitor`. We can also generate output from the algebra tree. Visitor `GraphDrawingVisitor` can generate output in dot language. Output of physical plan can be written in dot language using `PhysicalOperatorDrawingVisitor`. `PhysicalOperatorDrawingVisitorWithouSorts` provides dot output without partial sorts with empty sort parameters. In the following sections, we present text output generated by the implemented compiler.

5.7.1 Filters

Example:

```
Filter(double,double,int)->(double,double,int)
f(condition="OP_LOWER(OP_double_CONSTANT(4.8),1)");
```

Input and output columns are the same and they are both numbered from 0. This filter operator takes input of two double streams and integer stream and it filters by condition $4.8 < (column\ 1)$. Columns 0 and 1 are streams of doubles and column 2 is stream of integers. We have also another version of this operator which guarantees that the input and the output are sorted in the same way. In order to use it, we write $FilterKeepingOrder$ instead of $Filter$ in the operator declaration.

5.7.2 Group

Example:

```
HashGroup(string,string,int)->(string,int,int)
g(groupBy="1",functions="count(),max(2)");
```

Input columns are numbered from 0. Output columns consist of grouped columns and computed aggregate functions in the same order as specified in parameters. This example groups by column 1 and computes aggregate functions $COUNT$ and MAX. The parameter of the function MAX is column 2.

We have also sorted version of this operator. It assumes that input is sorted by group columns. In order to use it, we write *SortedGroup* instead of *HashGroup* in the declaration.

5.7.3 Column operations

Example:

```
ColumnsOperations(int,int,int,int,int)->
(int,int,int,double)
c(out="0,3,4,
OP_TIMES(2,OP_MINUS(OP_double_CONSTANT(1),2))");
```

Input columns are numbered from 0. Output is specified in the parameter *out*. If it contains the number of column, it copies input to output, otherwise it computes a new column. This example copies columns labeled with $0, 3, 4$ to output and computes a new column with the expression: $(column\ 2) * (1 - (column\ 2))$.

5.7.4 Cross join

Example:

```
CrossJoin(string,int),(int,string)->(string,string)
c(left="0,1",right="2,3",out="0,3");
```

Parameters *left* and *right* specify how the columns are numbered from the first and second input, respectively. The join outputs only the columns given in the *out* argument.

The other join operators have the same *left*, *right* and *out* parameters. The last mentioned rule applies to the following operators:

- `Hash join`
- `Merge equi-join`
- `Hash anti-join`
- `Merge anti-join`

5.7.5 Hash join

Example:

```
HashJoin(int,int),(int,int,int,int)->
(int,int,int,int,int,int)
h(left="0,1",right="2,3,4,5",out="0,1,2,3,4,5",
```

```
leftPartOfCondition="0,1","rightPartOfCondition="5,2");
```

This operator works only with equal condition which is given in the parameters *leftPartOfCondition* and *rightPartOfCondition*. Relation in the first input should be stored in a hash table since its estimated size is smaller than the size of the second relation. This example computes join with the condition: (*column* 0 = *column* 5) *and* (*column* 1 = *column* 2).

5.7.6 Merge equi–join

Example:

```
MergeEquiJoin(int),(int)->(int,int))
m(left="0",right="1",out="0,1",leftPartOfCondition="0:D",
rightPartOfCondition="1:D");
```

Condition is given in the parameters *leftPartOfCondition* and *rightPartOf-Condition*, and these parameters contain information about the sorting of the inputs. This example computes join with condition (*column* 0 = *column* 1). The first input is sorted by the column 0 descending and the second input is sorted by the column 1 descending.

5.7.7 Merge non equi–join

Example:

```
MergeNonEquiJoin(date,date),(date)->(date,date,date)
m(left="0,1",right="2",out="0,1,2",
leftInputSortedBy = "0:A,1:A",rightInputSortedBy = "2:A",
condition="OP_AND(OP_LOWER_OR_EQUAL(0,2)
,OP_LOWER_OR_EQUAL(2,1))");
```

This operator joins sorted relations. Numbering of columns from the first and second input is specified in the *left* and *right* parameter, respectively. Parameters *leftInputSortedBy* and *rightInputSortedBy* store information about the sorting of the input relations. Join condition is in the parameter *condition*. Operator in this example joins by condition *column* 0 ≤ *column* 2 ≤ *column* 1. The first input is sorted by the column 0 ascending and the column 1 ascending and the second input is sorted by the column 2 ascending.

5.7.8 Hash anti–join

Example:

```
HashAntiJoin(int),(int)->(int)
h(left="0",right="1",out="0",leftPartOfCondition="0",
rightPartOfCondition="1");
```

Parameter *out* can only contain columns from the first input. Condition is given in the parameters *leftPartOfCondition* and *rightPartOfCondition*. Relation in the first input should be stored in the hash table since its estimated size is smaller than the size of the second relation. This example computes the anti–join with the condition (*column* 0 == *column* 1).

5.7.9 Merge anti–join

Example:

```
MergeAntiJoin(int),(int)->(int)
m(left="0",right="1",out="0",leftPartOfCondition="0:D",
rightPartOfCondition="1:D");
```

The operator copies to the output only the rows from the first input for which no row in the second input satisfying given condition exists. Condition is given in the parameters *leftPartOfCondition* and *rightPartOfCondition* and they also contain information about the sorting of inputs. This example computes join with the condition (*column* 0 == *column* 1). The first input is sorted by the column 0 descending and the second input is sorted by the column 1 descending.

5.7.10 Table scan

Example:

```
TableScan()->(int,int,int,int)
t(name="lineitem",
columns="l_orderkey,l_shipdate,l_extendedprice,l_discount");
```

This operator scans the table specified in the parameter *name* and reads only columns given in the parameter *columns*.

5.7.11 Scan And Sort By Index

Example:

```
ScanAndSortByIndexScan()->(string,string,int)
s(name="people",index="index",
columns="user_name,country,parameter");
```

Scan And Sort operator reads the whole table given in the *name* using the *index* and reads the columns specified in the attribute *columns*.

5.7.12 Index Scan

Example:

```
IndexScan()->(int,int)
i(name="customer",index="index2",columns="c_custkey,
c_mktsegment",
condition="OP_EQUALS(1,OP_string_CONSTANT(SEGMENT))");
```

Index Scan operator reads part of the table given in the *name* using the *index* and reads the columns specified in the attribute *columns*. Operator reads only the rows satisfying the condition given in the attribute *condtion*.

5.7.13 Sort

Example:

```
SortOperator(int,int)->(int,int)
s(sortedBy="0",sortBy="1:D");
```

Input and output columns are the same and they are numbered from 0. Parameter *sortedBy* specifies by which columns is the table sorted and parameter *sortBy* specifies by which columns should the table be sorted. Example is already sorted by the column 0 and will be sorted by the column 1 descending.

5.7.14 Union

Example:

```
Union(int,string)(string,int)->(int,string)
u(left="0,1",right="1,0",out="0,1");
```

Numbering of columns from the first input is given in the $left$ parameter. Second input uses the same number of columns as the first output. This information is specified in the parameter *right*. Operator appends the rows from the input 1 to the rows from input 0. The order of the output columns is specified in the parameter *out*. Operator unites columns with the same numbers.

6. Conclusions

We described Bobox architecture and Bobolang language in Chapter 2. Chapter 3 contains theory used to implement the query transformer. Chapter Analysis 4 deals with description of used algorithms and important data structures used in the implemented tool. Final chapter 5 presents some implementation details of created program.

The aim of this thesis was to implement part of the SQL compiler. Created program reads input relational algebra which is optimized. We implemented very effective optimization of logical plan: pushing selections down the tree. Possible physical plans were enumerated using Selinger–Style Optimization method. In this phase, we replaced algebra operators with physical plan. We implemented two different algorithms for choosing the order of joins. Asymptotically slower algorithm based on dynamic programming is used for estimated order of joins on smaller amount of relations. A faster greedy algorithm, which can generate less optimal join tree is provided for larger amount of joined relations. While choosing order of joins, we assigned physical algorithms. Merging of the assignment of physical algorithms and choice of join order can result in faster physical plan, in case we do not have information about sizes of input relations. Physical plan is written to output in the Bobolang language. Implemented compiler provides possibility to write algebra tree and physical plan to language Dot for debugging purposes.

Created software is a first part of planned SQL compiler. Front end, which transforms text query to the relational algebra, has not yet been implemented. At the time of submitting this thesis, compiler was successfully connected to Bobox. However, not all of the physical operators have been implemented. Therefore we were not able to evaluate any queries to prove that generated plans were correct.

We tested software by transforming some simple queries and queries from TPC benchmark TM H [5] to physical plans. We are able to check generated plans by looking at generated debug outputs. Based on this results we include that generated plans are correct and optimal.

Implemented tool can be improved by adding more logical plan optimizations. After queries are run and their run time is measured, compiler time estimations, used for selection of physical algorithms, can be improved. We can also add support for more physical algorithms like nested loop joins.

Bibliography

[1] Bednárek D., Dokulil J., Yaghob J., Zavoral F.: *Bobox: Parallelization framework for data processing. Advances in Information Technology and Applied Computing*, 2012.

[2] Falt Z., Bednárek D., Martin K., Yaghob J., Zavoral F.: *Bobolang - a language for parallel streaming applications.* In 23rd international sym- posium on High-Performance Parallel and Distributed Computing. ACM, 2014.

[3] Garcia-Molina H., Ullman J. D., Widom J.: *Database Systems The Complete Book*. Prentice Hall, 2002, ISBN 0-13-031995-3.

[4] Falt Z.: *Parallel Processing of Data - Doctoral thesis*. Prague, 2013.

[5] *TPC BENCHMARK TM H*, Standard Specification, Revision 2.15.0

[6] *Xerces-C++*, http://xerces.apache.org/xerces-c/

[7] *Doxygen*, www.doxygen.org/

[8] Bednárek D., Dokulil J.: *TriQuery: Modifying XQuery for RDF and Relational Data.* In 2010 Workshops on Database and Expert Systems Applications, Bilbao, Spain, ISBN: 978-0-7695-4174-7, ISSN: 1529-4188

[9] Falt Z., Čermák M., Dokulil J., Zavoral F.: *Parallel SPARQL Query Processing Using Bobox.* In International Journal On Advances in Intelligent Systems, Vol. 5, Num. 3, ISSN: 1942-2679, pp. 302-314, 20121

[10] Silberschatz, A., Korth, H., Sudarshan, S.: *Database System Concepts (6th ed.)*, McGraw-Hill, 2010. ISBN 978-0-07-352332-3

[11] Zou Q., Wang H., Soulé R., Hirzel M., Andrade H., Gedik B., Wu K.: *From a Stream of Relational Queries to Distributed Stream Processing*, 2010, ISSN: 2150-8097

[12] Harizopoulos S., Shkapenyuk V., Ailamaki A.: *QPipe: A Simultaneously Pipelined Relational Query Engine* In Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, 2005, ISBN: 1-59593-060-4

Attachments

Attachment on CD

The genereted documentation can be found in the folder `RelationalQueryEvaluator/Doxygen/html/html`. THe folder `Thesis` contains text of this thesis.

Software was developed in Visual studio and its solution can be found in the folder `RelationalQueryEvaluator`. Recommended tool for compiling source codes is Visual Studio 2013. We used external library $XercesC++$ version 3.1.1 which is located in the folder `RelationalQueryEvaluator/externals`.

Test queries can be found in folder `RelationalQueryEvaluator/RelationalQueryEvaluator/data`. Files with the suffix .xml contain test queries. Compiler generates the following outputs for every processed file (for example named `query.xml`) containing query:

1. `query.xml._1.txt` – Dot representation of the input algebra tree.
2. `query.xml._2.txt` – Dot representation of the input algebra tree after semantic analysis and grouping phase.
3. `query.xml._3.txt` – Dot representation of the optimized algebra tree.
4. `query.xml._4.txt` – Dot representation of the best physical plans with unresolved sort parameters.
5. `query.xml._5.txt` – Dot representation of the best final physical plans.
6. `query.xml._6.bbx` – Bobolang representation of the best physical plan.

The generated PNG images from outputs of queries can be found in the folder with test queries. Description of image output can be found in Chapter 5. Provided Windows binaries are located in the folder `Binaries`.

Program usage:

- `RelationalQueryEvaluator.exe inputfile`

Every line of the input file should contain the name of the file with relational query.