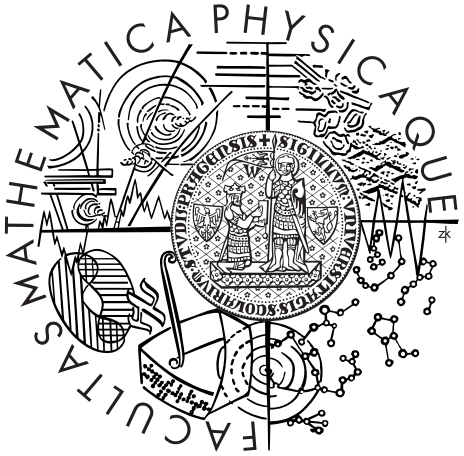
Charles University in Prague

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 thesis 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:

Author: Marcel Kikta

Department: Název katedry či ústavu, kde byla práce oficiálně zadána

Supervisor: RNDr. David Bednárek, Ph.D.

Abstract:

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

Contents

Introduction

Today's processors have multiple cores and it's single core performance is improving only very slow because of physical limitations. On the other hand number of cores is still increasing and we can assume that it will continue. That's why developing parallel software is crucial for improving 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 it's own parallelization framework called Bobox[1].

Bobox is designed for parallel processing 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 created to evaluate queries over relational data but it was succesfully used in implementation of XQuery and TriQuery engines.

Bobox contains from runtime environment and operators. Theses operators are called boxes and they are C++ implementation of data processing algorithm. 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 then into directed acyclic graph. Bobolang specifies the structure of whole application and also the inner structure of each box. It can create highly optimized evaluation, which is capable of using the most of the hardware resources. The language has been tested in several applications and it turned out to be very powerful tool in data processing massive parallel application.

Most used databases are relational databases. They are based on the view of data organized in tables called relations. SQL[3] ("Structured query language") is very important language based on relation databases. It is used for queering data, modifying content of tables and also the structure of tables. When we want to evaluate query we need to parse query text input into parse tree. This form will be transformed to relational algebra, which we call logical query plan. It will be optimized and physical plan is generated. Physical plan indicates not only operation performed, but also which order are they performed and what kind of algorithms are used for execution.

The main goal of this thesis is to implement part of SQL compiler. The input is query written in XML format in from of relational algebra. Program validates input, optimizes and transforms it to physical plan of given query. The output is execution plan for Bobox written in Bobolang.

1. Architecture

1.1 Bobox

In the section we describe basic architecture of Bobox. Information source for this chapter is Doctoral thesis Parallel Processing of Data[4].

Overall Bobox architecture is displayed in figure 1.1. Framework contains of Boxes. Box is basically a C++ class containing implementation of data processing algorithm or it can be set of connected boxes. Box can have arbitrary number of inputs and outputs. All boxes are connected to a directed acyclic graph.

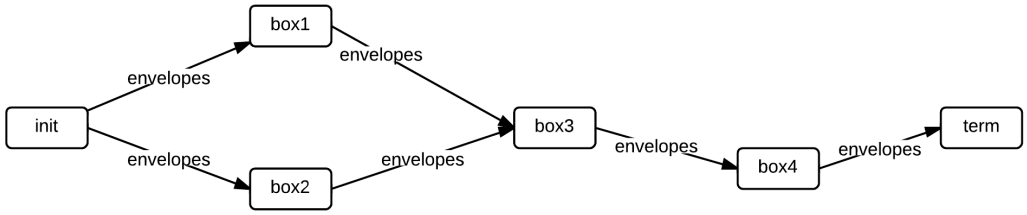


Figure 1.1: Bobox architecture.

Data streams are implemented as data units called enveloped. Envelope structure is displayed in figure 1.2. It consists of sequence tuples, but internally data are stored by columns, that means envelope contains from sequence of columns and it's data is stored in separate list. So to read all attributes of the i-th tuple we have to access all column lists and read it's i-th element. There is special type of envelope having poisoned pill. It is send after all valid data indicating end of data stream.

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

- *init* - first box in topological order and it indicates starting box of execution plan
- *term* - last box in topological order and indicates that plan has been completely evaluated

Evaluation starts with scheduling *init* box, which sends poisoned pills to all of its output. All of it's output boxes will be scheduled. They can read data from hard drive or network, process it and sent it to other boxes for further processing. Other boxes usually receives data in envelopes in their inputs. Box *term* waits for every it's input to receive poisoned pill and then evaluation ends.

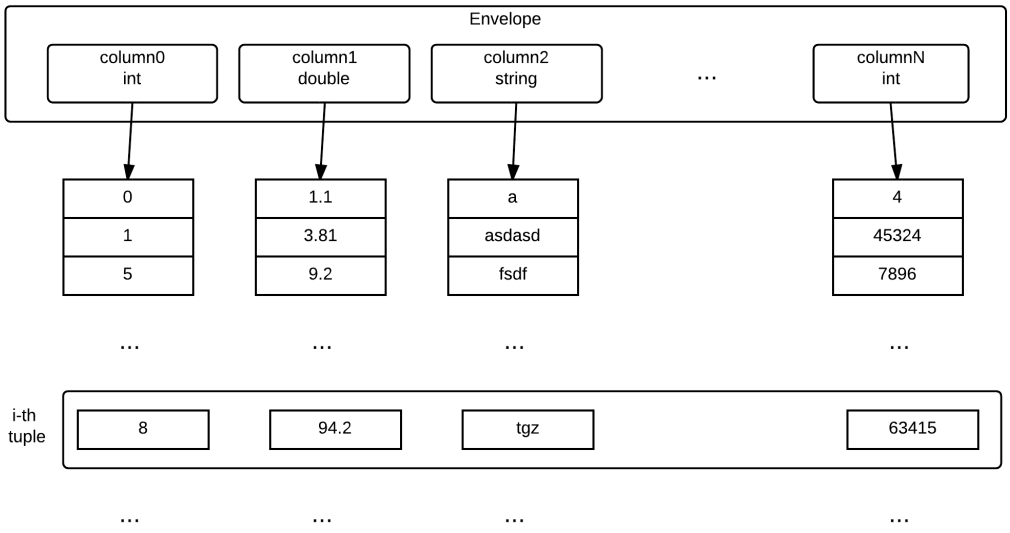


Figure 1.2: Envelope structure.

1.2 Bobolang

In this section we describe syntax and semantics of Bololang language. We used paper Bobolang - a language for parallel streaming applications[2] as information source.

Bobolang is a formal description language for Bobox execution plan. Bobox environment provides implementation of basic operators (boxes). Bobolang let's programmer choose which boxes to used, what boxes to use, what type are passed and how the boxes are interconnected. Bobolang also provides possibility to create operators connecting existing ones.

In following example we show a definition using other operators:

```
operator process (int)->(int,int,int)
{
    preproc(int)->(int,int) pre;
    post(int,int)->(int,int,int) post;

    input -> pre;
    pre -> post -> output;
}
```

Code specifies that we are creating new operator called `process`. It takes one stream of integers as input and outputs one stream of triplets integers.

In the first part we declare sub operators, define type of input and output. For every declared sub operator we provide identifier. Second part specifies connec-

tion between declared operators. Code `op1 -> op2` indicates that output of `op1` is connected to input of operator `op2`. In this case output type of `op1` has equal to input type of `op2`. Bobolang syntax also allows to create chains of operators like `op1 -> op3` which has semantics like `op1 -> op2` and `op2 -> op3`.

There are explicitly defined operators called `input` and `output`. They represents input and output of declared operator `process`. The line `input -> pre;` represents that input of the operator `process` is connected to operator `pre`.

Boblang also allows to declare operators with empty input or output. They have type `()` that means it doesn't transfer any data. Only data allowed is to transfer poisoned pill. When box receives poisoned pill, it means that it should start working, Sending it means that it's work is done.

We can define whole execution plan using operator main with empty input and output. Example of whole Bobolang plan:

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

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

In figure 1.3 we can seen structure of example execution plan. Operators `init` and `term` are added automatically. Operator `init` sends poisoned pill to `source`, which can read data from hard drive or network. These data are send to box `process`. Operator sink stores data and sends poisoned pill to box `term` and the computation ends.

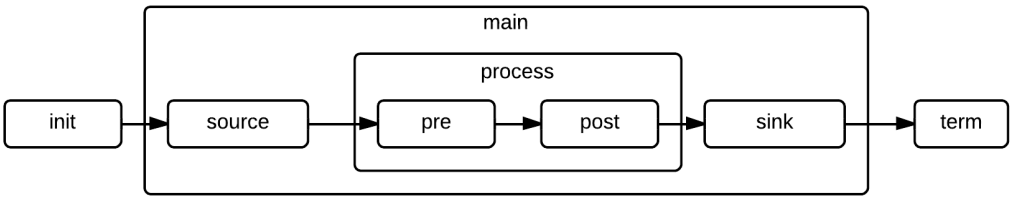


Figure 1.3: Example of execution plan.

1.3 SQL compiler architecture

In this section we describe planed SQL compiler. It's architecture is displayed in figure 1.4.

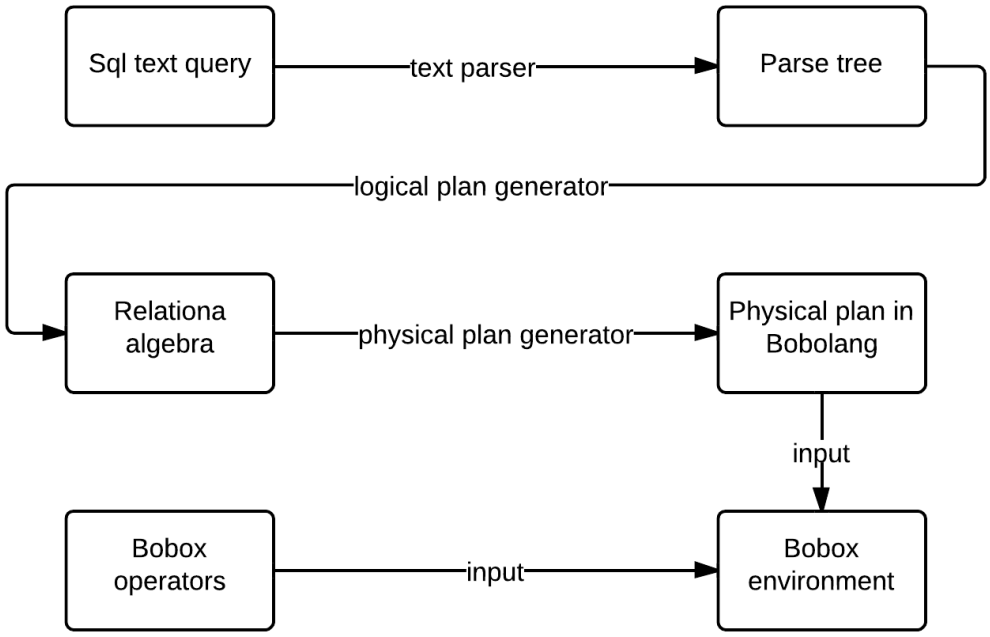


Figure 1.4: SQL compiler architecture.

SQL query is written in text. This text is 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. Physical plan is not enough, we need to also provide implementation of physical algorithms (Bobox operators).

Since SQL is a pretty complicated language, this thesis aim is only implementing optimization and transformation of logical plan into physical plan.

2. Related work

I the chapter we introduce some theory of relational algebra, it's optimizations and physical plan generation. This informations were used for tool implementation.

2.1 Relational algebra

In this chapter we introduce and describe relational algebra[3]. We start with some basic definitions of relational model.

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

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

Definition 3. *Schema name of the relations and a set of attributes. For example:* $Movie(id, name, lenght)$.

Definition 4. *Tupple of a relation is a row other than header row.*

An algebra in general consist of operators and atomic operands. For example in arithmetic algebra variables like x or constant like 15 and operators are addition multiplication, subtraction and division. We can build expression by applying operators on operands or other expressions. Example of an expression in arithmetic algebra is $(15 + x) * x$.

Relational algebra has atomic operands:

- Variables, that are relations.
- Constants, that are finite relations.

In classical relational algebra all operates and expression results are set. All this operations can be applied also to bags. Relation 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 relation, all kinds of joins.
- Renaming operations, that doesn't change tuples of the relation bud changes schema.

Expressions in relational algebra are called *queries*.

2.1.1 Classical relational algebra operators

Set operations on relations

Sets operations are:

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

Lets have relations R and S. If we want to apply some set operation both relations must have the same set of attributes. If we want to compute set theoretic union, difference or intersections the oder of columns must be the same in both relations. We can also use renaming operations if relations doesn't have same number of attributes.

Projection

Projection operator π produces from relation R new Relations with reduced set of attributes. Result of a expression $\pi_{A_1,A_3,A_4,...,A_N}(R)$ is relation R with attributes $A_1, A_3, A_4, ..., A_N$.

Selection

If we apply operator selection σ on Relation R with condition C we get a new relation with same attributes and tuples, which satisfy given condition. 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 similar. We pair tuples from R with all tuples from S.

Natural joins

We usually don't want to pair all of the tuples from R to all tuples from S. We can par tuple in some other way. The simples 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 same name (in this case $c_1, c_2, ...c_n$).

Theta joins

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

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

Basically $R \bowtie_C S = \sigma_C(R \times S)$

Renaming

In order to control name of attributes or relation name we have renaming operator. We can use operator $\rho_{S(A_1,A_2,...,A_n)}(R)$. Result will have the same tuples as R but relation will be called S and attributes will be renamed to $(A_1, A_2, ..., A_n)$.

2.1.2 Relational operations on bags

Commercial database system almost never are based purely on bags. *Bag* is a multi-set. Only operation that behave differently are intersection union and difference.

Union

Bag union of $R \cup S$ we just add all tuples from S and R together. If tuple t appears in R m-times and in S n-times then in $R \cup S$ will t appear $m + n$ time. Both m and n can be zero.

Intersection

Lets have tuple t that appears in R m-times and S n-times. In the Bag intersection $R \cap S$ will be t $min(m, n)$-times.

Difference

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

2.1.3 Extended operators of Relational algebra

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

Duplicate elimination

This 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 operator but are used by grouping operator. They apply on column and produce one number as result. The standard operators are SUM, AVG(average), MIN, MAX and $COUNT$.

Grouping operator

We often doesn't want to compute aggregation function for entire column. We rather compute this function on for some group of columns. For example we can compute average salary for every person in database, or we can group them by companies and get every salary in every company.

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

1. Attribute of R by which R will be grouped.
2. Aggregation operator applied on a attribute of relation.

Relation computed by expression $\gamma_L(R)$ is constructed:

1. Relation will be partitioned into groups. Every group contains all tuples which have same value in all grouping attributes. If there is no grouping attributes, all tuples will be in one group.
2. For each group operator produces one tuple consisting of:
   (a) Grouping attributes values for group.
   (b) Results of aggregations over all tuple 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 projection operator $\pi_L(R)$ introduced in chapter 2.1.1. We denote it also $\pi_L(R)$ 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, arithmetic, string and other operators. x is new name. 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, L is list of attributes with additional information about sort order, is relation with same tuples like R but 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 same A_1 value will be additionally sorted by their A_2 value descending.

Outer joins

Lets have join $R \bowtie_C S$. We call tuple t from relation R or S *dangling* if we didn't 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 don't have but appear in join result.

Left/right outer join is outer join but we only add dangling tuples from left-/right relation.

2.2 Optimizations of relational algebra

After initial logical query plan is generated, we can apply some heuristics to improve it, using some algebraic laws that hold for relational algebra.

2.2.1 Commutative and associative laws

Commutative and associative operators are Cartesian product, natural join, union and intersection. Theta join is commutative but generally is not associative. But if the conditions makes sense where they where positioned, then theta join is associative. That means we can make following changes to algebra tree:

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

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

2.2.2 Laws involving selection

Selection are very important for improving logical plan. They usually reduce size of relation markedly so that's why we need to move them down the tree as far as possible. We can change order of selections:

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

Sometimes we cannot push whole condition 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. $\cup_S$ stands for set union. We can push selection down 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 difference we must push it to first branch. Pushing to second branch optional. Laws for difference:

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

Following laws allow to push selection down both arguments. Let's have selection σ_C. We can push it to the branch, which contains all attributes used in C. If C contains only attributes of R:

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

$\oplus$ stands for $\times$, $\cap$, $\cup$, $\bowtie$ or $\bowtie_C$. If relation S and R contains all attributes of C we can also use following law:

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

2.2.3 Laws involving projection

Principle for manipulation with projections that we can add projection anywhere in the tree as long as it only eliminates attributes which are not used anymore and don't appear in query result.

2.2.4 Laws involving joins and products

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

- $\sigma_C(R \times S) = R \bowtie_C S$
- $R \bowtie S = \pi_L(\sigma_C(R \times S))$, C is condition that equates each pair of attributes of R and S, which have the same name and L is a list of attributes of relation R.

2.3 Physical plan generation

After we optimized logical plan, we need to create physical plan. We generate many physical plans a choose one with least estimated cost to run it. This approach is called cost-based enumeration.

For each physical plan we select

1. An order of grouping and joins.
2. An algorithm for each operator. For example if we use join based on hashing or sorting.
3. Additional operators which are not presented in logical plan. For example we can sort relation in order to user faster algorithm which assumes that it's input is sorted.
4. The way in which arguments are pass to between operators. We can use iterators for it or store result on hard drive.

2.3.1 Size estimations

The costs of evaluating physical plan are based of estimated size of intermediate relations. Ideally we want out estimation to be accurate, easy to compute and logically consistent(size of relation doesn't depend on how relation is computed). We will present simple rules, which will give us give us good estimations in most situation. Goal of estimating sizes is not predict exact size of relation, even an inaccurate sizes will help us with plan generation.

In this section we will use following conventions:

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

Estimating the size of projection

Projection is only operator which size of result is computable. It doesn't change number ot tuples, only their lengths change.

Estimating the size of selection

Selection usually reduces number of tuples. Lets have $S = \sigma_{A=c}(R)$, where A is a attribute of R and c is a constant. Recommended estimation is:

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

More problematic estimation is when selection involves inequality comparison Lets have $S = \sigma_{A<c}(R)$. In average half the tuple satisfies condition, but usually queries select only a small fraction from all tuples. Therefore the estimation is:

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

For selection where condition is in form C_1 *and* C_2 *and ...* *and* C_N we can treat selection as a cascade of simple selections and estimate size for every simpler condition.

In case we have following condition: $S = \sigma_{not(C)}(R)$, we use this estimation:

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

Little more complicated is when condition involved an *or* of conditions. Lets have expression $S = \sigma_{C_1 \ or \ C_2}(R)$. We assume that C_1 and C_2 are independent. Size of S is:

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

Expression $1 - \frac{m_1}{T(R)}$ is fraction of tuples which doesn't satisfy condition C_1 and $1 - \frac{m_2}{T(R)}$ is fraction of tuples which doesn't satisfy condition C_2. Product of these numbers are the fraction of tuples from R which are not in result. One minus the product gives us fraction of tuples in S.

Estimating the size of join

We start with natural join. Let's have expression $X = R(X, Y) \bowtie S(Y, Z)$. We join here by one attribute and we use estimation:

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

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

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

If we have other type of join (theta join), we can use following rules for it's estimation:

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

Estimating the size of union

If we have bag union, then size of resulting relation is sum of sizes of input relations. Size of set union $R \cup_S S$ can be between $max(T(R), T(S))$ and $T(R) + T(S)$. Recommended estimate is average of this numbers.

Estimating the size of intersection

Size of relation $R \cup S$ can be between 0 and $min(T(R), T(S))$. We recommend to take half of size of smaller relation:

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

Estimating the size of difference

Result of expression $R - S$ can be as big as $T(R)$ and as small as $T(R) - T(S)$. Suggested estimate is:

- $T(R - S) = T(R) - T(S)$.

Estimating the size of 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 statistics is almost never available so we need another estimate. $T(\gamma_L(R))$ can be between 1 and $\prod_{k=1}^{n} V(R, g_k)$. Upper bound can larger than number of tuple in relation R. That's why we suggest:

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

Duplicate elimination can be handled exactly like grouping.

2.3.2 Enumerating plans

For every logical plan there is exponential many physical plans. in this section we present way to enumerate physical plans so we can choose plan with least estimated cost. There are two broad approaches:

- Top-down: We go down the algebra tree. For each possible implementation of operation, we consider best possible algorithms for it's subtrees. We compute costs of every combination and take the best.
- Bottom-up: We go up the tree and for every subexpression we enumerate plans based on plans generated for it's subexpressions.

There is not much difference between this approaches but one method eliminates plans that other method can't and vice versa.

Heuristic selection

We use same approach for generating physical plan as we used to improve logical plan. We make choices based on heuristics. Here are some heuristics that can be used:

- If an join attribute has an index, we can use joining by index.
- If one argument of join is sorted, then we prefer merge join to hash join.
- If we compute intersection of union of more than 2 relations, we perform algorithm on smaller relations first.

Branch-and-bound plan enumeration

Branch-and-bound plan enumeration is ofter use in practice. We begin by finding physical plan using heuristics. We denote cost of this plan C. Then 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 grate advantage is we can choose when to cut search for better plan. If C is low then we don't continue searching other plan. On the other hand when C is large, it is better to invest some time in finding a better plan.

Hill climbing

First we start with heuristically selected plan. Then we try to do some changes, for example changing order of joins or replacing operator using hash table for

sort-based operator. When we find a plan where no small modification give up better plan, then we make this plan out chosen physical plan.

Dynamic programming

Dynamic programming is variation of bottom-up strategy. For each subexpression we keep only one plan of least cost.

Selinger-Style Optimization

This is a improvement of dynamic programing approach. We keep for every subexpression not only best plan, but also other plans with higher cost, which result is in some way sorted. This might be useful in following situation:

1. The sort operator τ_L is on the root of tree and we have 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 situation we don't have to sort input or we need only partial sort and we can use faster algorithm, which takes advantage of sorted input.

2.3.3 Choosing join order

We have three choices how to choose order of join of multiple relation:

1. Consider all possible options.
2. Consider only subset of join orders.
3. Use heuristic to pick one.

In this section we present algorithm that can be used for choosing join order.

Dynamic programing to select a join order

For this algorithm we need a table, where we store information following information:

1. Estimated size of relation.
2. Cost to compute current relations.
3. Expression how was current relation computed.

We store every single relation R with estimated cost 0. For every pair of relations we compute their estimated size of join and store it into table. Fo that we use estimation describe in section 2.3.1.

After that we compute entry of all subset of sizes $(3, 4, ...n)$. If we want to consider all possible trees, we need to partition relations in join R into two non empty disjoint sets. For each pair of sets we compute estimated size and cost of their join to get join R. For this we use data already in our table. We are estimating join of k relations and all joins of $k-1$ relations are already estimated. The join tree with least estimated cost will be stored in the table.

Example: for estimating join $A \bowtie B \bowtie C \bowtie D$ we try to join 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 $A \bowtie B \bowtie C$
5. $A \bowtie B$ and $C \bowtie D$
6. $A \bowtie C$ and $B \bowtie D$
7. $A \bowtie D$ and $B \bowtie C$

We don't have to consider all trees but only so called left-deep join tree. Tree is called left-deep tree if all of it's right children are leafs. Right-deep tree is tree, where all left children are leafs.

We do it the same way like we wanted to include all possible trees, but with one exception. When partitioning R into two non empty disjoint sets, we make sure that one set has only one relation.

Example: for estimating join $A \bowtie B \bowtie C \bowtie D$ we try to join 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 for selecting a join order

Time complexity of using dynamic programming to select an order of join is exponential. We can us it for small amount of relations (maybe maximal 5 or 6).If we don't want to invest time we can use greedy algorithm. This algorithm has to store same information as dynamic programing algorithm for every relation.

We start with a pair fo relations R_i, R_j, for which size of $R_i \bowtie R_j$ is smallest. This join is our current tree.

For other relation that are not yet included we find relation R_k, so that it's join with current tree give us smallest estimated size. We continue until we include all relations into tree.

This algorithm will also create left-deep or right-deep join tree.

There are examples where dynamic algorithm finds plans with lower estimated cost than greedy algorithm.

2.3.4 Choosing physical algorithms

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

Choosing selection algorithms

Basic approach is to use index on relation instead of reading whole relation. We can usel following heuristic for picking selection algorithm:

- If there is selection $\sigma_{A \oplus c}$ and relation R has index on attribute A, c is constant and $\oplus$ is $=$, $<$, $\leq$, $>$ or $\geq$, then we use scanning by index instead of scanning table with filtering result.
- More general rule if selection contains condition $A \oplus c$ and selected relation contains index of A, then we use scanning by index and filtering result based on other parts of condition.

Choosing join algorithms

We recommend to use sort join when:

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

If we have join $R(a, b) \bowtie S(b, c)$ and we expect size of relation to be small and S have index on attribute on b, we can use join algorithm using this index.

If there are no sorted relation and we don't have any indexes on relations, it is probably best option to use hash base algorithm.

Choosing scanning algorithms

Leaf of relation tree will be replaced by scanning operator:

- $TableScan(R)$ - operator reads entire table.
- $SortScan(R, L)$ - operator reads entire table and sort by attributes in list L.
- $IndexScan(R, C)$ - C is a condition in form $A \oplus c$, where c is constant, A is attribute and $\oplus$ is $=$, $<$, $\leq$, $>$, $\geq$. Operator reads tuples satisfying condition using index on A, result will be sorted by columns on used index.
- $TableScan(R, A)$ - A is an attribute. Operator reads entire table using index on A, result will be sorted by columns on used index.

We choose scan algorithm based of need of sorted output and availability of indexes.

Other algorithms

Basically for other operators we usually have sort and hash version of algorithm. For example for processing grouping we can create groups using hash table or we can sort relation by grouped operators. We can choose:

- Hash version of algorithm if the input is not sorted in way we need or if the output doesn't have to be sorted.
- Sort version of algorithm if we have sorted input by some of requested parameters, or we need sorted input. In case the input is only sorted by some needed attributes we can still use pretty fast partial sort and apply sort based algorithm.

3. Analysis

3.1 Format of relational algebra

In this section we present relation algebra operators which are used as input of compiler:

1. Projection - we used extended projection ϕ_L, which remove columns, compute new columns using expressions and rename columns

2. Table operator, which is leaf or algebra tree. For this operator we need to provide arguments like:

   - table name
   - information about index, like name and columns
   - columns to read

3. Join - we used theta join $\bowtie_C$. Condition C can be in following format:

   - Empty and in this case join represent 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 belong to first relation and b_k belongs to other relation.
   - $a_1 \oplus b \ominus a_2$, where a_1 and a_2 belong one input and b belongs to second input. $\oplus$ and $\ominus$ can be $<$ or $\leq$.

   In addition to condition we need to specify output attributes of join. They can be from both input and we can optionally assign them new name, in case we need to work with two attributes but they have same name.

   Other types or joins are not directly supported, but can be replace with cross join with following selection.

4. Anti join wasn't presented with other join algorithms. We denote it $\ltimes_C$ where C is anti join condition. Output of expression $R \ltimes_C S$ is relation with tuples from R, for which doesn't exist any tuple is S that satisfy condition. We can us join and anti join to express outer join:

   - $R \bowtie^{\circ}_C S = (R \bowtie_C S) \cup (R \ltimes_C S)$

   To be precise we need to add columns contain *null* to result of anti join.

   Other use is to compute difference $R - S$. This can be rewritten as $R \ltimes_C S$, where C equates attributes from R with same called attributes in S.

Advantages of using this attribute is, that we don't need outer join and difference, which will make working with algebra a little easier.

In implemented tool condition C of anti join can be in 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 belong to first relation and b_k belongs to other relation.

In addition to that, we also need to specify output attributes of anti join and optionally assign them a new name. They can be only from first input relation.

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

   As we mentioned before, group operator is more general version of duplicate elimination. That's why we don't include duplicate elimination in our algebra.

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

7. Union - $\cup$ is set union. In case we want to bag union we can compute set union and eliminate duplicate using grouping operator. Requirement is that both relations have same number of columns and they have same name.

8. Selection - we used selection as described in classic relational algebra.

3.2 Physical algorithms

In this section we enumerate and describe algorithms which are generated to output. We assume that queries have enough memory and physical operators doesn't have to store intermediate result on hard drive.

Here is a list on algorithms:

- *Filter* - this algorithm reads input tuples and outputs tuple satisfying given condition. Output doesn't have to be sorted same way as input.

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

- *Hash group* - operator group tuples and computes aggregate functions. Grouping is performed using hash table.
- *Sorted group* - operator groups tuples and computes aggregate functions. Input has to be sorted by group attributes.
- *Column operations* - this is an implementation of extended projection algebra node.
- *Cross join* - this operator computes product of two relations.
- *Hash join* - computes join with equal conditions using hash table.
- *Merge equijoin* - this algorithm computes join with equal conditions. Input relations has to be sorted by join attributes. Algorithm creates join result merging sorted relations.
- *Merge non equijoin* - operator computes theta join with condition $a_1 \oplus b \ominus a_2$, where a_1 and a_2 belong one input and b belongs to second input. Signs $\oplus$ and $\ominus$ can be $<$ or $\leq$. Input relations has to be sorted by join attributes. Algorithm computes join merging relations.
- *Hash anti join* - algorithm computes anti join with equal conditions of two relations using hash table.
- *Merge anti join* - algorithm computes anti join with equal conditions. Input relations has to be sorted by join attributes.
- *Table scan* - operator scans whole table from hard drive.
- *Scan and sort by index* - operator scans whole table from hard drive using index. Output will be sorted by columns on given index.
- *Index Scan* - this algorithm uses index to read only tuples satisfying given condition.
- *Sort* - this algorithm sorts input. Input can be presorted, in this case operator uses this information and sorts only by not yet sorted attributes.
- *Union* - this operator is bag union, it only append tuples from one relation to another.

Nested loop joins are not supported, because it's implementation in Bobox can create cycles.

3.3 Architecture

The architecture of implemented tool is displayed in figure 3.1.

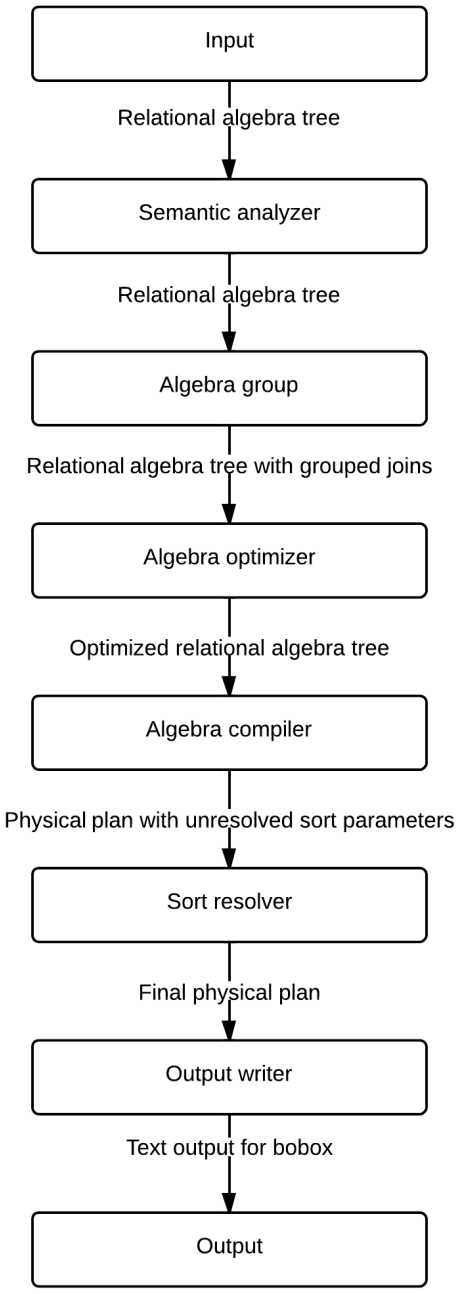


Figure 3.1: Compiler architecture.

Relational algebra is read from input. the input is in XML format. For this format we decided for following reasons:

- XML has tree like structure exactly like relational algebra.
- For validation we only need to write schema.

- There are already implemented tools for parsing.
- There is no need to write input parser.

After that relational algebra tree is checked in component called semantic checker. Semantic checker checks if all of used attributes are on input of operator or if there are no duplicate operators.

Semantically correct tree goes to component that groups neighboring joins into one. This is done so we can choose fastest way to join multiple relations.

Algebra tree with grouped joins is optimized. We implemented one most important optimizations pushing selections down the tree. This component also pushes selection to join if selection contains equal condition, where one argument is from first and second argument is from second input.

Optimized algebra tree is processed by compiler, which generates physical plan which. This plan is not final. It's sort operator's parameters doesn't have to be final. For example if we want to sort relation before grouping we can sort it in different directions and than later decide what direction is better.

Final plan is output of component named Sort resolver. This component decides unknown sort order of sort operators.

Final plan is then converted to Bobolang in Output writer.

Implemented tool doesn't check types. Since it will be back end of compiler, the assumption is that front end parsing text will handle types. Types are only copied to output and we assume that there are no mistakes in types.

3.4 Data structures

In this chapter we describe data structures used in implementing tool.

Relational algebra si stored in polymorphic tree. Every node stores it's parameters pointer on parent in the tree and zero or more pointer on children node. No other structure was considered for this representation since this is efficient way to store logical plan. It allows easily to add new types of relational algebra operators and it is not had to manipulate with the tree. We can remove or add new node very easily. Example of this representation can be found in figure 3.2. It's representing simple query reading whole table, then grouping it and computing some aggregation functions. The result is sorted at the end. Leaf of the tree also stores some information about indexes on read table, list of columns with their type and number of unique values. Other important parameter is size of relation which is displayed in number of rows parameter.

We choose same structure for physical plan. Physical plan usually doesn't have to changes. The advantage of storing it into polymorphic tree is to ability

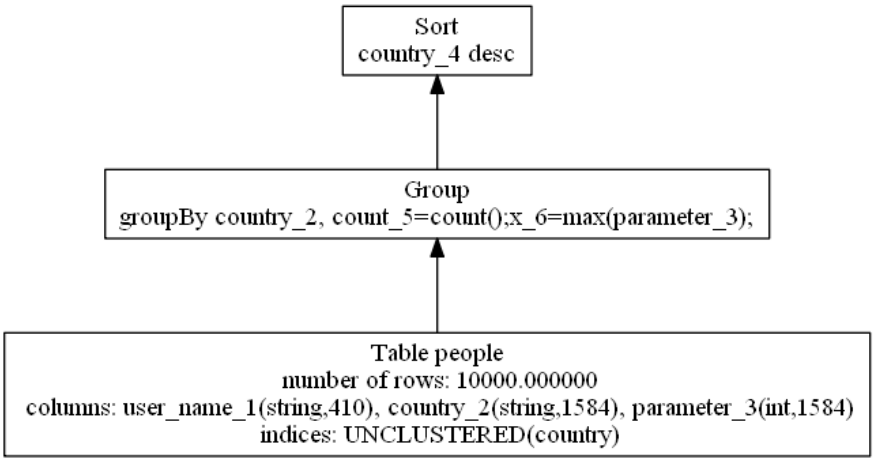


Figure 3.2: Example of relational algebra structure.

to easily add new root node. Example of this representation can be found in figure 3.3. This figure contains one of possible physical plans for relational algebra in figure 3.2. For reading we used algorithm table scan, then we hashed input by requested columns and at the end we sorted it using sort operator. Every nodes stores additional information like output attributes, estimated time it's going to need and size of output relation.

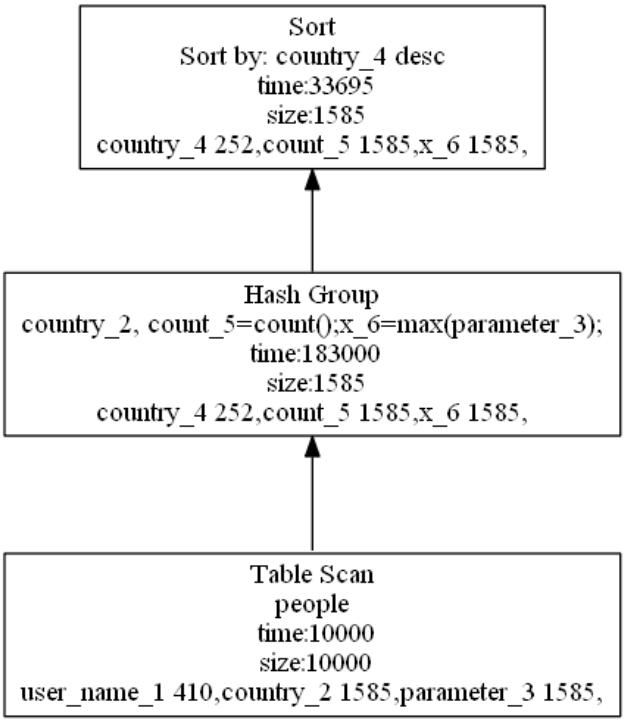


Figure 3.3: Example of physical plans structure.

Physical and logical plan also contain expressions. Expressions are stored in polymorphic expression tree. We have example of this structure in figure 3.4. This structure represents expression $X * Y + 874$.

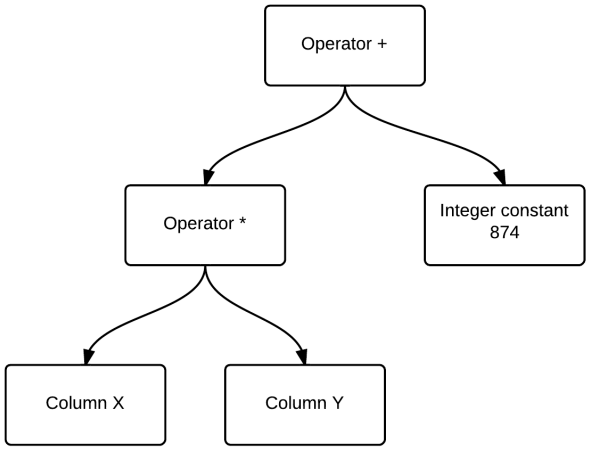


Figure 3.4: Example expression tree.

More complicated structure was used to storing join parameters. This structure is stored in every sort physical operator to determine what columns should relation be sorted by.

If we want to use group operator based on sort and it groups by three columns, we don't know which sort direction to use. Let's have expression $\gamma_{x,y}(R)$. There is four way to sort expression before calling group operator. This ways are:

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

A means ascending and D is abbreviation for descending.

If we want to use merge join, joining on two attributes, we don't know direction and also which column should be first and which second. For example let's have $R \bowtie_{r_1=s_1 \ and \ r_2=s_2} S$. In this case we can sort relation R following way:

- r_1, r_2
- r_2, r_1

Order, how to sort columns is also unknown.

We also want to store information about equality of sort column. After merge join $R \bowtie_{r_1=s_1} S$ is result sorted by r_1 or s_1.

All this requirements were use to design structure to store sort parameter without enumerating all possible sort orders.

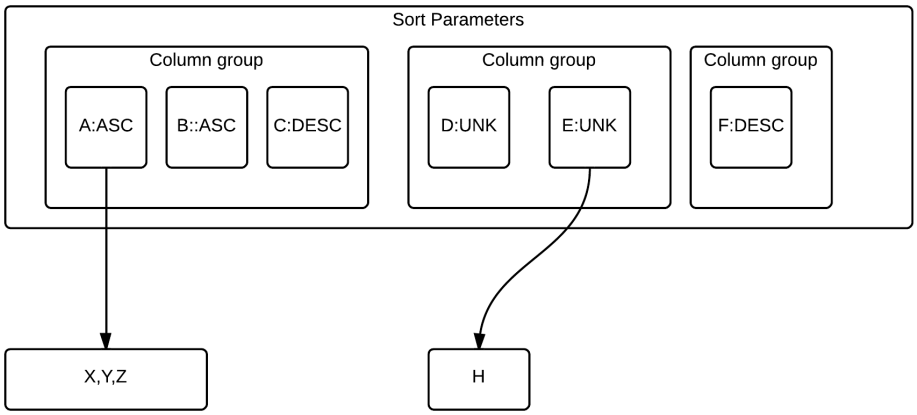


Figure 3.5: Structure storing parameters for sort.

In figure 3.5 we display an example of sort parameters, which sorts by 6 columns. It usually contains from 1 or more columns group. The meaning is that order of columns groups is set. Order of columns in groups is arbitrary. It means that F has to be on sixth place, but column E can be on forth of fifth. Every column contains information about sort order: ASC (ascending), $Desc$ (descending) or UNK (unknown - can be ascending or descending). Every column also can be list of attributes which are equal to it. If we for example in projection remove attribute A, we still have attributes X, Y and Z which are equal to it, so one can take it's place.

Figure 3.5 represents many sort order possibilities 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. $C : DESC, B : ASC, Y : ASC, D : ASC, H : ASC, F : DESC$

3.5 Optimization

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

Before we start with optimizations we need to prepare logical plan for it. We group joins algebra nodes and expressions connected with *and* and *or*.

Basically we go from to of the tree. If we find join we convert to it grouped join. If on of it's children is join we merge it. This representation is used for

choosing faster way to order join. We do the same thing for conditions. From expression tree $a = 2$ *and* $(b = 2$ *and* $c = 2)$ we create $AND(a = 2, b = 2, c = 2)$. This representation is useful for splitting condition into simpler conditions.

We implemented very important optimization: pushing selection down the tree. Every selection is spitted into selection with simpler conditions. For every such selection we try to move it down the tree as much as possible. In this phase we used 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 doesn't contains new computed columns in extended projection. We also need to rename columns in condition C in case there was some renaming.
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 is from R and b is from S
4. $\sigma_C(R \ltimes_C S) = \sigma_C(R) \ltimes_C S$ if C contains only columns from R, which is always because output of $R \ltimes_C S$ can contain only columns R
5. $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$

3.6 Generating physical plan

We try to choose easiest method for generating plans. The decision was between heuristic method and dynamic programming. From amount of code it was probably the same. We chose dynamic programming, because it can give better results. We just generate all possible plans for each node and choose the fastest.

We process logical plan from leafs. For every leafs we generate all possible physical algorithms and we insert resulting plans into heap, where we keep c fastest plans for current node, where c is constant set in compiler.

For every node we use plans generated in it's children to generate new plan. This way we go up the logical plan tree up to the root.

For comparing physical plans we estimated run time. For every node we store how long it will run. Estimated time for plan is sum of estimated times of all nodes.

Very important are equations, which compute estimated time for nodes. They depend from size of input relation. Modifying them can resolve in getting better

physical plans. For example if physical algorithm hash join takes too much time because of lot random accessing memory, we can modify it so sort with merge join will be preferred.

Crucial are informations about size of tables. If they are not in input we use default value and physical plan will be probably worse. Other important parameter is number of unique values in table column. Size of join depends on it and since joins usually takes significant amount of time, it is important to have as precise values as possible.

3.6.1 Join order selecting algorithm

In section 2.3.3 we presented algorithm choosing join order. We should choose this order and then assign join algorithms. We do it in one phase for following reason. In case we don't have information about table sizes, we cannot determine join order because all are the same. In this situation we can first join relations which are sorted some way to get a faster plan.

In this section we describe algorithm for selecting join order. We are using Two algorithms dynamic programming and greedy algorithm. For dynamic programming we use version where we enumerate all possible trees. This algorithm can give us very good plans but has exponential complexity and that's why we use it only if number of joined relations in grouped join node is small. If we join more relations we use greedy algorithm, which only generated left or right deep trees but it's complexity is not exponential only polynomial.

Input in both algorithm are set of plans for every input join relation.

Dynamic programing for selection join order

We use a variation of algorithm described in section 2.3.3. We number then input relations from 1 to n. For actual computation we use table, where we store plans. Key of the table is non empty subset of set $1..n$.

For every table entry we only store k best plans. It represents best plans for, that were created by joining input in key of table.

In begin we store input plans into table entry identified by set containing input number. In first iterator we fill tables entries, which key have two values by combining plans from entries with key size 1.

Then we compute plans for entries, which have key size $3, 4...n$. We do so by spiting set in key of the entry in all possible pairs of non empty disjunctive subsets. We take plans from this subsets and we generate new plans combining them. We only store k fastest plans.

We do this until we compute plans for table entry $1..n$. This is our result.

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

Greedy algorithm for selection join order

This is also a variation of algorithm described in section 2.3.3. It beginning we create joins of every pair of relations. From them we choose best k trees.

In every following iteration for every tree we generate new trees by adding new relation to tree. In following iteration we continue only with best k join trees. At the end we have maximal k plans.

Time complexity is $O(n^2)$. In every iteration we generate from every tree maximal n new trees, but we keep only k of them for next iteration. Number of iteration is $n - 1$, because all trees grow by one in every iteration. Number k is a constant so it doesn't change time complexity.

3.6.2 Resolving sort parameters

After we generated physical plan we need to decide what sort parameters in sort operator to use.

To do this we goes down the tree and store information about how relations is sorted. Based on that we adjust sort parameters or just choose arbitrary order if possible.

4. Implementation

In this chapter we describe implementation details in developed software and describe it's functionality on examples. We will use following example to show optimizations and compiling:

```
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 is taken from Tpc benchmark [5]. $[DATE]$ and $[SEGMENT]$ are constants. In this benchmark there are no indexes on tables. Columns which starts wit o_ are from table order, columns which beginning with l are from table lineitem and columns starting with _c belongs table customers.

4.1 Input

As mentioned input is XML containing logical query plan. In this section we describe it's structure.

4.1.1 Sort

On root of every tree is sort, even if output hasn't been sorted. in this case it has empty parameters. This is an example of sort in algebra tree:

```
<?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>
```

Sort is a root element of XML file. Inside parameters is specified how to sort relation. In this example we have sort $\tau_{revenue:desc,o\_orderdate:asc}(...)$. In element input there should be one other algebra tree node.

4.1.2 Group

Next example display 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 has to have at least one group by parameter or at least one aggregate function. Inside element input there should be one other operator.

4.1.3 Selection

This is an example of selection:

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

This example represent following expression: $\sigma_{today<l_shipdate}$. In condition element we can have multiple conditions connected by *and* or *or* elements. Input algebra supports operators $=, <$ and $\leq$. In the leafs of expression tree there can be only column or constant element. We also can call a boolean function from condition, which is represented in following example.

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

While using boolean predicate it has to be supported by runtime(Bobox operators). Compile doesn't check it's existence.

4.1.4 Join

Join without condition is considered to be cross join. We can use join with multiple equal conditions or with simple unequal condition. 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 conditions has to be first column from first relation and second column from second relation. In example a and c are from first input and b and d are from the other one. Joins doesn't copy to output all column from both input relations. After condition we have to specify non empty sequence of columns. In every column we specify it's name and number of input. We can also rename join output column by using attribute *newName*. In example we renamed column d to e.

Next example shows also join but 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 condition $a1 \leq b \leq a2$. In first sub condition first column has to be from first input, but in second sub condition first column has to be from second input. Also instead *lower_or_equals* we can use just *lower* condition. Rules for output column are same like in join with equal conditions.

In element *input* of join there has to be two operators.

4.1.5 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 antijoin with simple condition $d = b$. Structure is the almost same like join. Output columns can be only from first relation and we can

also rename this columns.

4.1.6 Table

This is a leafs of algebra tree. It specifies name of read table, its columns and indexes. We can specify number of rows in the table to get better plans. If it is not specified we will assume that table has 1000 tuples. For every column we have to specify name and it's type. Other optional parameter is *number_of_unique_values*. This number is important for estimating size of join. If it is not given, we will assume, that *number_of_unique_values* is size of table to power of $\frac{4}{5}$. This assumption is only experimental, since number of unique values can be from 0 to size of table. Index can be clustered or unclustered. Table can have only one clustered index. In every index we specify on what attribute it is created. Here is an example or 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>
```

4.1.7 Union

Union doesn't have any parameters, but columns from both input have tt have the same name. Here is an example:

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

4.1.8 Extended projection

Following example of 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>
```

Extended projection contains list of columns. If columns is new computed values it contains elements representing expression tree. It cal also contain function call, which has to be supported by Bobox operators. Following example displays function call:

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

```
    ...
  </input>
</column_operations>
```

4.2 Building relational algebra tree

In this section we describe in more details structure storing logical plan and it's building.

Relational algebra operators are represented by children of abstract class `AlgebraNodeBase`. It has 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 variable number inputs
- `NullaryAlgebraNodeBase` - abstract class for algebra tree leafs

All operators are children of one of mentioned classes. Every operator has pointer to it's parent in tree and smart pointers to it's children if it has any.

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

For manipulating and reading expression and algebra tree we used visitor pattern. All nodes (algebra and expression) contains method `accept`. This method calls visitor method on class `AlgebraVisitor`/`ExpressionVisitor`. All classes, which manipulate algebra tree, are children of class `AlgebraVisitor`.

In figure 4.1 we have example of algebra tree of query presented in beginning in this chapter. We translated it as cross join of tree tables. After that we apply selection with condition in where clause. From the result we compute new column, group and sort result. In table reading operator we can see that they store additional information, like name of table. Every attribute has $\_ - 1$, behind name. It is it's unique identifier but after building tree from XML it is not assigned yet and that's why it contains default value -1. Columns in expressions also contains number in parenthesis. This number stores information from which input this columns it. Inputs are numbered from 0. Here we can see that all columns in selection are from 0^{th} input. This information is useful mainly in joins.

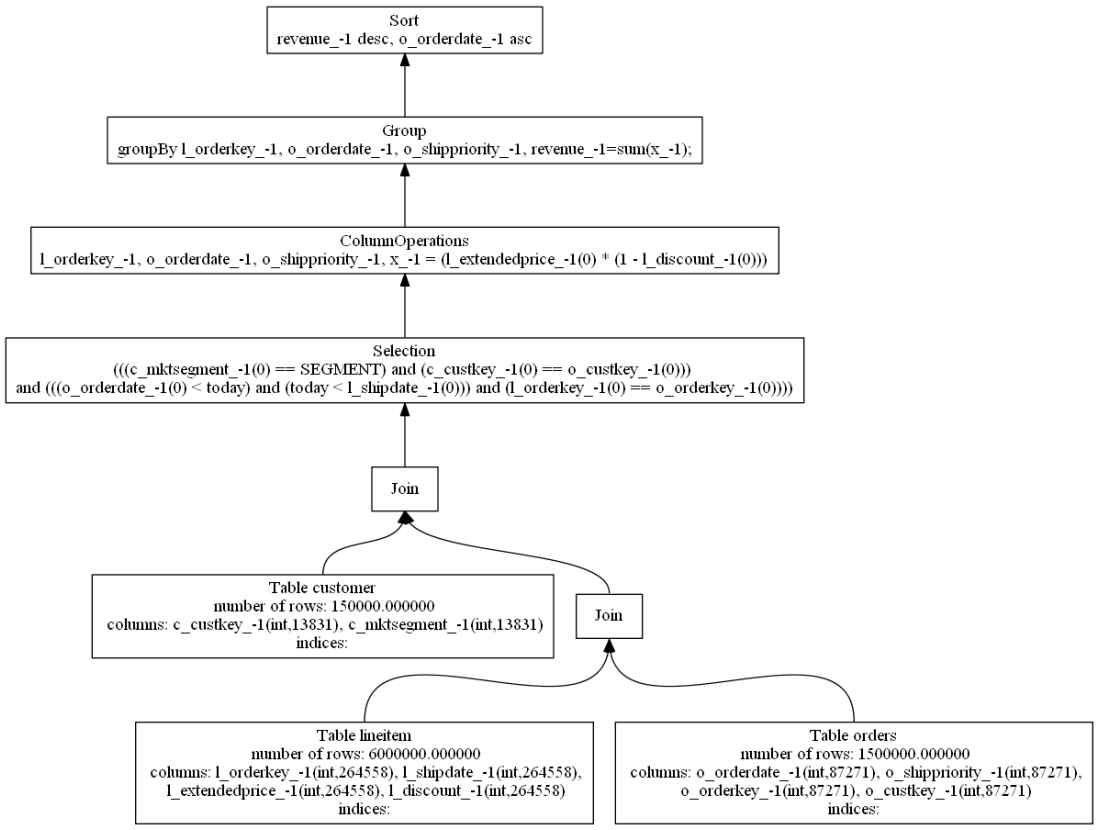


Figure 4.1: Example of algebra tree.

For parsing and validating input XML file we used library Xerces version 3.1.1[6]. It parses input file and create DOM tree. This tree has to be validated so we know it contains valid relational algebra tree. All this functionality is in class `XmlHandler`.

We know that every tree has to have sort operator on the top. We call `Sort` constructor on it. This method takes all information from DOM tree and call method, which decide that constructor to call next on it's children. This way we recursively build algebra tree.

4.3 Semantic analysis and node grouping

This phase in done in class `SemanticChecker`. It checks columns used in expression exist. It also checks if output columns of operator has unique name. During this checking we assign unique identifier to every column. After this phase we don't need names only this id.

Logical plan is after that visited by `GroupingVisitor`. In this phase are replaced joins represented by class `Join` by grouped join with two or more input relations. This node is represented by class `GroupedJoin`. Also in every expression we apply `GroupingExpressionVisitor`. It groups expression with *and* and

or operators. This is done for simplifying splitting condition into sub conditions.

4.4 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 rule:

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

From every selection we created chain of selections. This operation is done by `SelectionSpitingVisitor`.

After that we call `SelectionColectingVisitor`. This visitor stores pointer of all selection in relational algebra tree. This 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, we can find selection chains

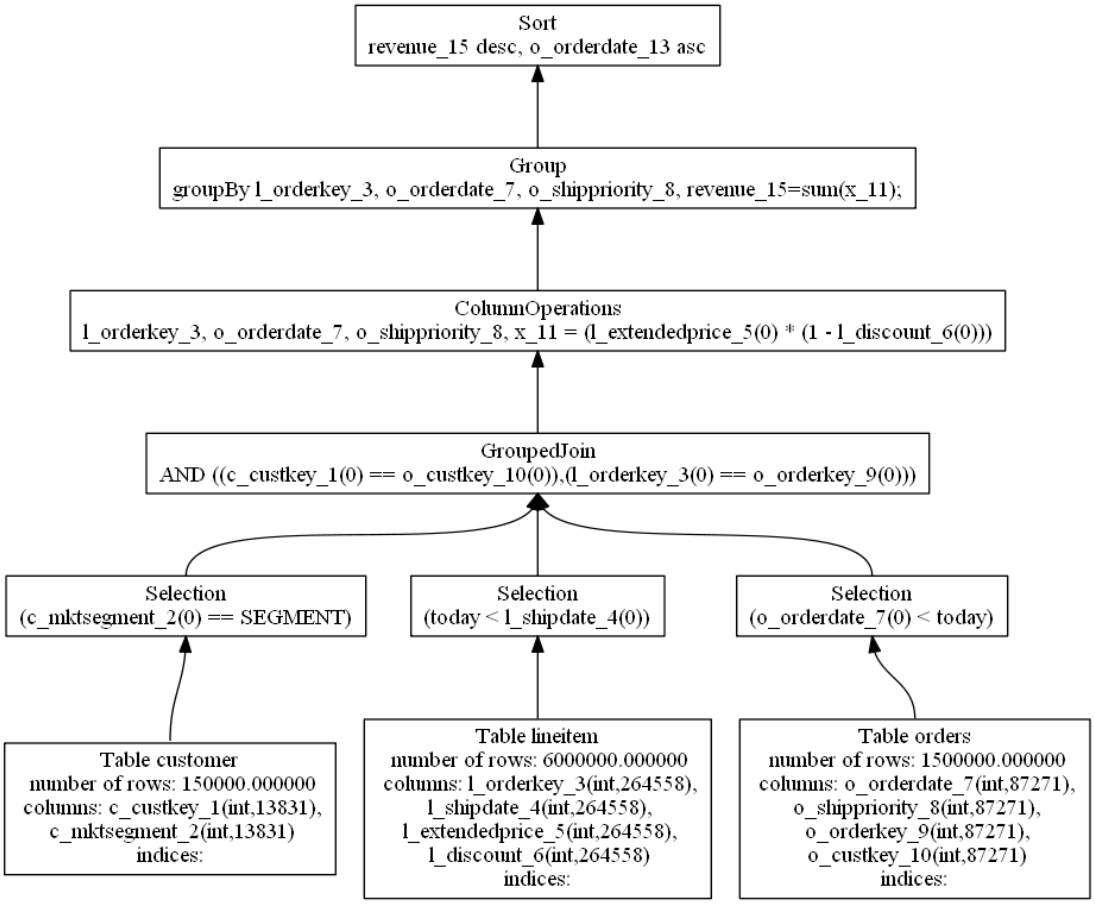


Figure 4.2: Example of optimized algebra tree.

in it. To resolve this problem we apply `SelectionFusingVisitor`. This visitor applies following rule to tree:

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

In figure 4.2 we can see optimizes algebra tree. In comparison figure 4.1, this tree has grouped join with three input relations. Also big selection about joins has been split and moved down the tree. Some part of condition became join condition other were pushed down on of branches of grouped join. Then we can see that new tree has columns with assigned unique identifiers. Because we have this identifiers we don't need to know number of input for each column.

This input is optimized algebra tree. We can of course implement more optimizations to improve logical plan.

4.5 Generating plan

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

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

In every node we generate all possible plans. Structure store in variable `result`, which is a heap. After inserting plan we remove slowest plan.

Physical plan is represented as polymorphic tree.

In figure 4.3 we can see best generated physical plan for query presented on the beginning of this chapter. Every operator contains estimated size and time. Below that we can see output columns with their unique identifiers. Since read tables doesn't contain any indexes, we have to read all the tables and filter results. After that we can only use hash join, because sorting relations for merge join would be to expressive and nested loop join is not supported by runtime or compiler. From the result we compute new columns and use hash group algorithm. We didn't use sorted group, because input is not sorted and output has to be sorted by other than group column.

4.6 Output

In this section we describe text output generated by implemented compiler.

4.6.1 Filters

Example:

Figure 4.3: Example of physical plan.

```
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 numbered from 0. This operator takes input of two double streams and integer stream and it filters by condition $4.8 <$ (*column number* 1). We have also another version of this operator, which guaranteers that input and output are sorted the same way. To use it we write *FilterKeepingOrder* instead of *Filter* in operator declaration.

4.6.2 Group

Example:

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

Input columns are numbered from 0. Output columns consists from grouped columns and computed aggregate functions in the same order as in parameters. This example groups by column number 1 and computes aggregate function

COUNT and *MAX*. *MAX* has as parameter column number 2.

We have also sorted version of this operators. It assumes that input is sorted by group columns. To use it we write *SortedGroup* instead of *HashGroup* in declaration.

4.6.3 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 parameter *out*. It if contains number operator, copies input to output, otherwise it computes new column. This example copies columns number $0, 3, 4$ to output and computes new column with expression: $2 * (1 - \textit{column number } 2)$.

4.6.4 Cross join

Example:

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

Numbering columns from first input is specified in $left$ parameter and numbering columns from second input is specified in *right* parameter. Join outputs only columns given in *out* argument.

4.6.5 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");
```

Numbering columns from first input is specified in $left$ parameter and numbering columns from second input is specified in *right* parameter. Join outputs only columns given in *out* argument. This operators works only with equal condition, which is given in parameters $leftPartOfCondition$ and $rightPartOfCondition$. This example computes join with condition:
$(\textit{column } 0 = \textit{column } 5)$ *and* $(\textit{column } 1 = \textit{column } 2)$.

4.6.6 Merge equijoin

Example:

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

Numbering columns from first input is specified in *left* parameter and numbering columns from second input is specified in *right* parameter. Join outputs only columns given in *out* argument. Condition is given in parameters *leftPartOfCondition* and *rightPartOfCondition* and they also contain information how are inputs sorted. This example computes join with condition (0 == 1). First input is sorted by column number 0 descending and the second input is sorted by column 1 descending.

4.6.7 Merge non equijoin

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 from left(first) and right(second) input is specified in parameters *left* and *right*. Parameters *leftInputSortedBy* and *rightInputSortedBy* store information about how are input relations sorted. Join condition is in parameter *condition*. Operator in this example joins by condition *column* 0 $\leq$ *column* 2 $\leq$ *column* 1. First input is sorted by column 0 ascending and column 1 ascending and second input is sorted by column 2 ascending.

4.6.8 Hash anti join

Example:

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

Column number from first input is specified in *left* parameter and columns numbers from second input is specified in *right* parameter. Join outputs only columns given in *out* argument. Parameter *out* can only contains columns from

first input. And specifies columns, which goes to output. Condition is given in parameters *leftPartOfCondition* and *rightPartOfCondition*.This example computes antijoin with condition (*column* 0 == *column* 1).

4.6.9 Merge anti join

Example:

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

Numbering columns from first input is specified in *left* parameter and numbering columns from second input is specified in *right* parameter. Join outputs only columns given in *out* argument. Operator copies to output only rows from first input for which doesn't exist row in second input satisfying given condition. Condition is given in parameters *leftPartOfCondition* and *rightPartOfCondition* and they also contain information how are inputs sorted. This example computes join with condition (*column* 0 == *column* 1). First input is sorted by column number 0 descending and the second input is sorted by 1 descending.

4.6.10 Table scan

Example:

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

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

4.6.11 Scan And Sort By Index

Example:

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

Operator reads whole table given in *name* using *index* and reads columns specified in attribute *columns*.

4.6.12 Index Scan

Example:

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

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

4.6.13 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 table sorted and parameter *sortBy* specifies by which columns should table be sorted. Example is already sorted by *colum* 0 and will be sorted by *column* 1 descending.

4.6.14 Union

Example:

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

Numbering columns from the first input is given in the $left$ parameter, from second input is given in the *right* parameter and from the output is given in the *out* parameter.

Conclusion

Bibliography

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

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

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

[4] Zbyněk Falt. *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/

Attachments