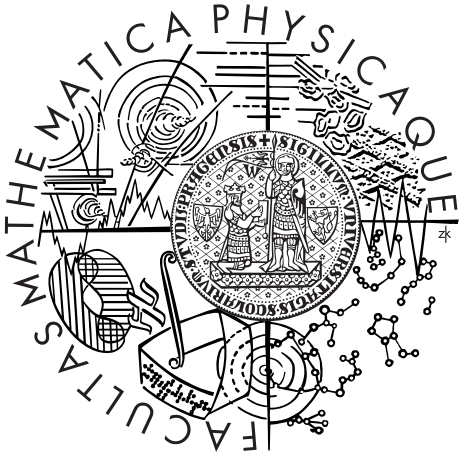
Charles University in Prague

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: Název katedry či ústavu, kde byla práce oficiálně zadána

Supervisor: RNDr. David Bednárek, Ph.D.

Abstract: This thesis deals with the design and implementation of optimizer and transformer of relational queries. Firstly, the thesis describes database compiler theory. Following chapter presents data structures and algorithm used in implemented tool. Final chapter specifies important implementation details. Part of the thesis was the selection of used relational algebra operators and design of appropriate input. Input of implemented software is query written in XML file in form of relational algebra. Query is optimized and transformed into physical plan which will be executed in parallel framework Bobox. Developed compiler outputs physical plan written in Bobolang language, which serves as input for 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 and TriQuery 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[3]) 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.

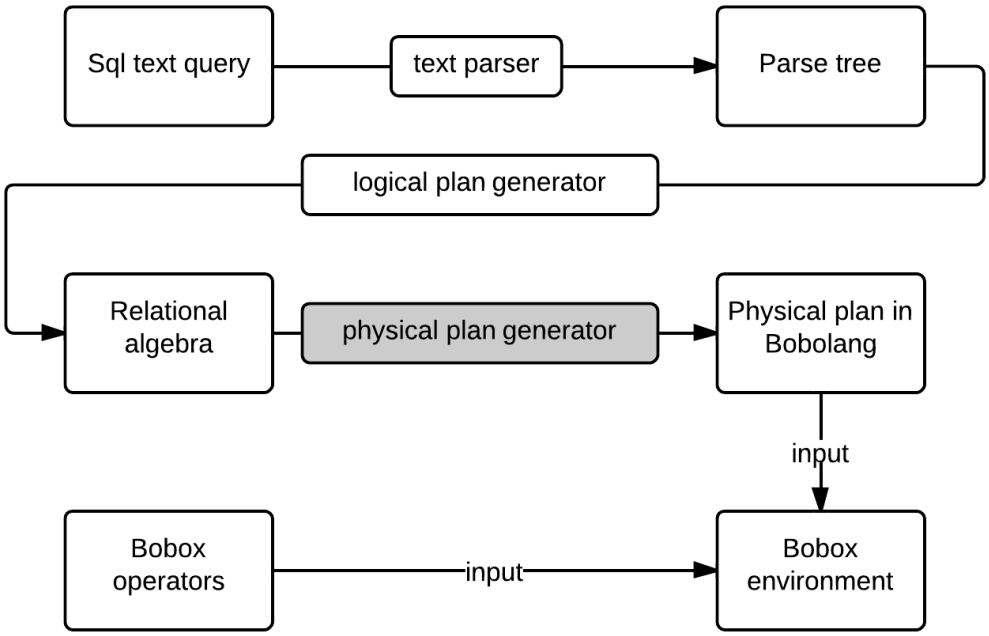


Figure 1.1: SQL compiler architecture.

Then, we improve logical plan by pushing selection 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.

2. Architecture

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.

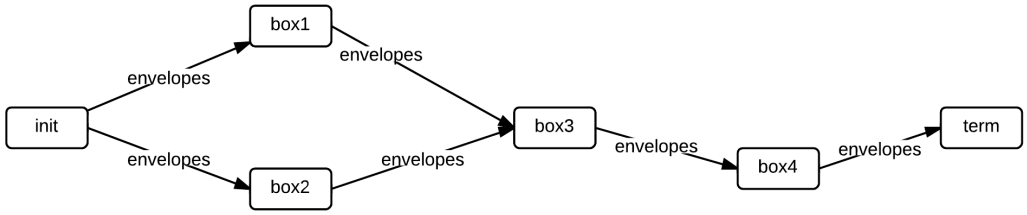


Figure 2.1: Bobox architecture.

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 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.

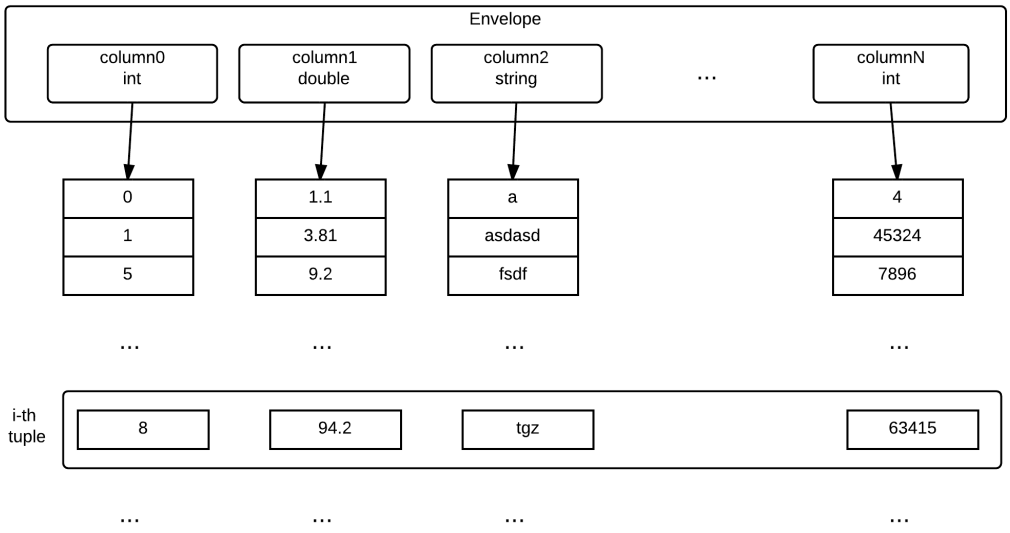


Figure 2.2: Envelope structure.

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.

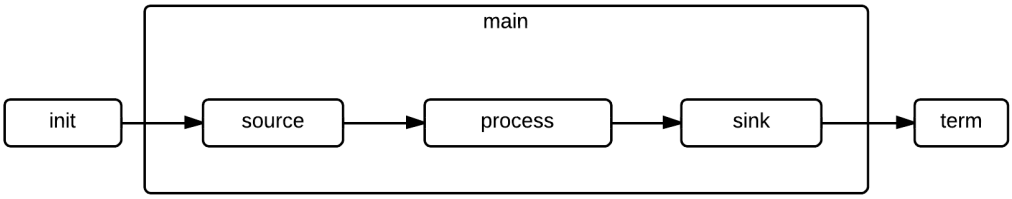


Figure 2.3: Example of execution plan.

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 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 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 or store result on hard drive.

3.3.1 Size estimations

Estimates used in this section are from Database systems [3]. The costs of evaluating physical plan are based of estimated size of intermediate relations. Ideally, we want our 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 situations. Goal of estimating sizes is not predict exact size of relation, even an inaccurate size will help us with plan generation.

In this section we will use the 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

Bag projection is only operator which size of result is computable. It doesn't change number of tuples but only their lengths.

Estimating the size of selection

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

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

More problematic estimation is for selection involving inequality comparison: $S = \sigma_{A<c}(R)$. In average half the tuple satisfies condition, but usually queries select only a small fraction from relation. Typical inequality will return about third of the original tuples. Therefore the recommended estimation is:

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

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

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

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

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

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

To use this estimate we assume that condition C_1 and C_2 are independent. Variable m_i equals $T(\sigma_{C_1}(R))$, where $i \in \{1, 2\}$. 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 $A = R(X, Y) \bowtie S(Y, Z)$. We join here by one attribute and we use estimation:

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

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

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

When we join using multiple attributes, than result of this estimation can be smaller than 1. This estimation indicates that relation will be very small or possibly empty.

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

1. Size of Cartesian 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 expression $\sigma_{A<c}(R)$. We assume that 1/3 of the 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 relations $R \cup_S S$ and $R \cup_B S$ can be between 0 and $min(T(R), T(S))$. Recommend estimate is 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 can be used for bag or set version of difference:

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

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 be larger than number of tuples in relation R. That's why we suggest:

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

Estimation of duplicate elimination can be handled exactly like grouping.

3.3.2 Enumerating plans

For every logical plan there is exponential many physical plans. This section presents ways to enumerate physical plans, so plan with least estimated cost can be chosen. 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 great 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 faster 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.

3.3.3 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 following informations:

1. Estimated size of relation.

2. Cost to compute current relations.

3. Expression how was current relation computed.

We store every single input relation 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 3.3.1.

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.

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. 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.

3.3.4 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.

4. Analysis

4.1 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 π_L, which remove columns, compute new columns using expressions and rename columns

2. Table reading operator - 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 replaced 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 form S containing *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 from *sum* and *count*. 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.

Used relational algebra works with bags.

4.2 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 operator.
- *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.

4.3 Architecture

The architecture of implemented tool is displayed in figure 4.1.

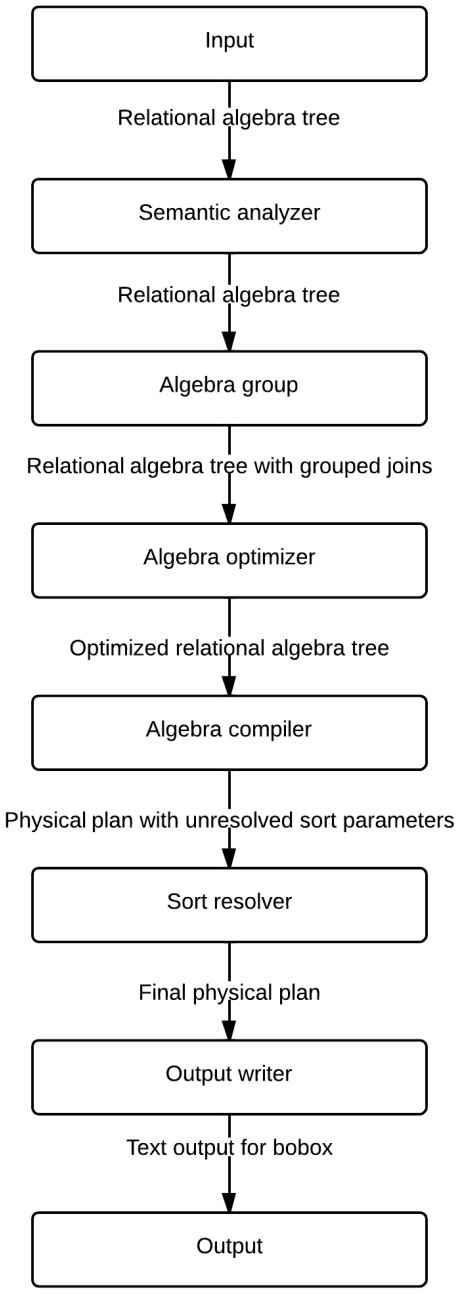


Figure 4.1: Compiler architecture.

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 is processed by 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 errors in types.

4.4 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 pointers 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 hard to manipulate with the tree. We can remove or add new node very easily. Example of this representation can be found in figure 4.2. 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 types 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 change. The advantage of storing it into polymorphic tree is to ability to

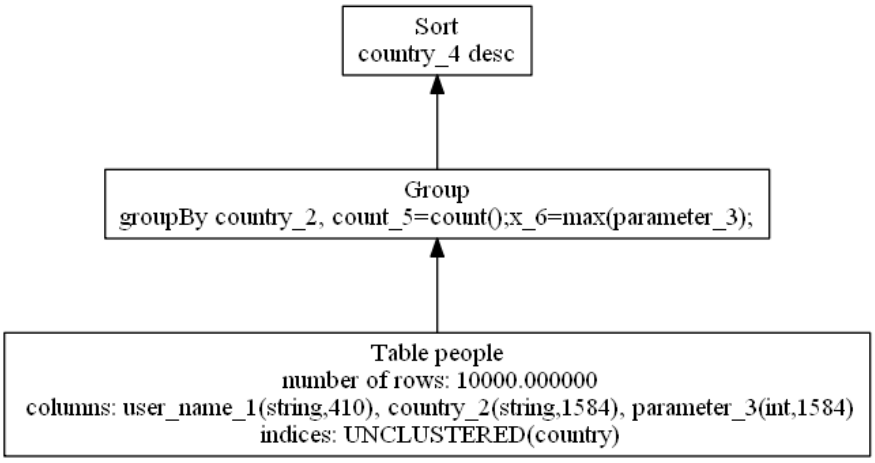


Figure 4.2: Example of relational algebra structure.

easily add new root node. Example of this representation can be found in figure 4.3. This figure contains one of possible physical plans for relational algebra in figure 4.2. 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 and size of output relation.

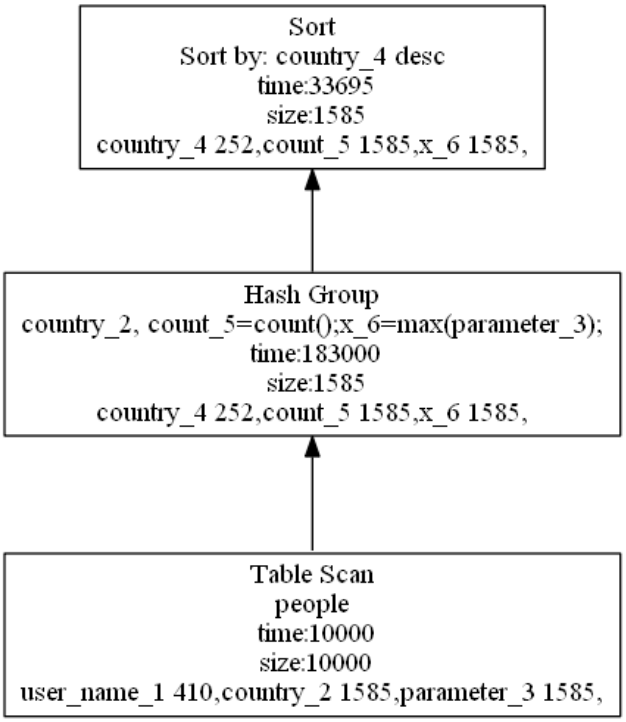


Figure 4.3: Example of physical plans structure.

Physical and logical plan also contain expressions. Expressions are stored in polymorphic expression tree. We have example of this structure in figure 4.4. This structure represents expression $X * Y + 874$.

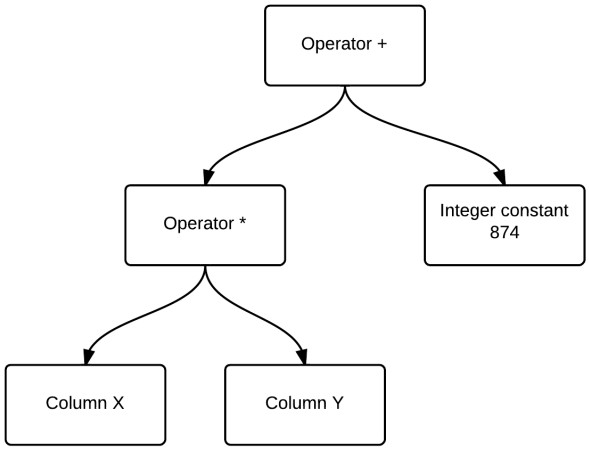


Figure 4.4: Example expression tree.

More complicated structure was used to storing 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.

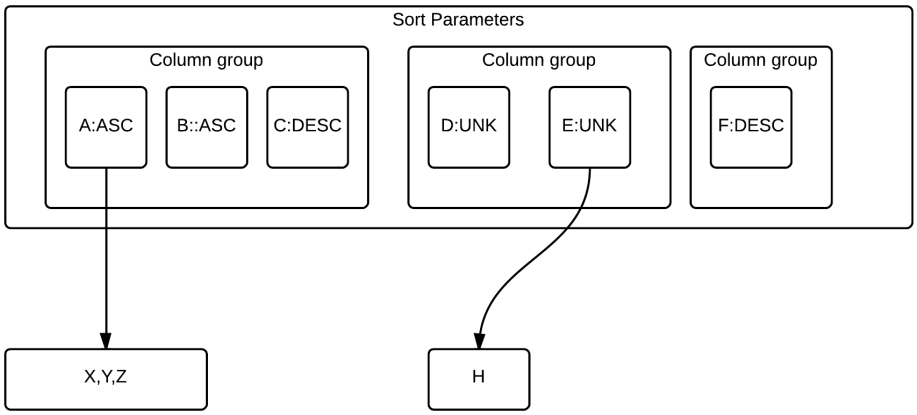


Figure 4.5: Structure storing parameters for sort.

In figure 4.5 we display an example of sort parameters, which sorts by 6 columns. It usually contains from 1 or more columns group. The order of columns groups cannot be changed. 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 4.5 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. $C : DESC, B : ASC, Y : ASC, D : ASC, H : ASC, F : DESC$

4.5 Optimization

In this section we describe algebra optimization, which wes 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 top 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 selections 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. $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$

4.6 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 estimated time 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.

4.6.1 Join order selecting algorithm

In section 3.3.3 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 3.3.3. 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 3.3.3. 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.

4.6.2 Resolving sort parameters

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

To do this we go 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.

5. Implementation

In this chapter we describe implementation details in developed software and describe it's functionality on examples. More implementation details can be found in generated doxygen[7] documentation in attachment on CD. We will use 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 is taken from 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.

5.1 Input

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

5.1.1 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.

5.1.2 Group

Next example contains 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.

5.1.3 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>
```

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

5.1.4 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* there have to be two 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 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.

5.1.6 Table

This is a leaf 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 of 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

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

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

5.1.8 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 can also contain function call, which has to be supported by Bobox operators. Following example displays function call:

```
<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>
```

We compute new column named x with values $\sqrt{2}$.

5.2 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 5.1 we have example of algebra tree of query presented in beginning in this chapter. We have rewritten it as cross join of tree tables. After that we apply selection with condition in where clause. From the result we compute new column. After that we apply grouping compute aggregate functions and sort the 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.

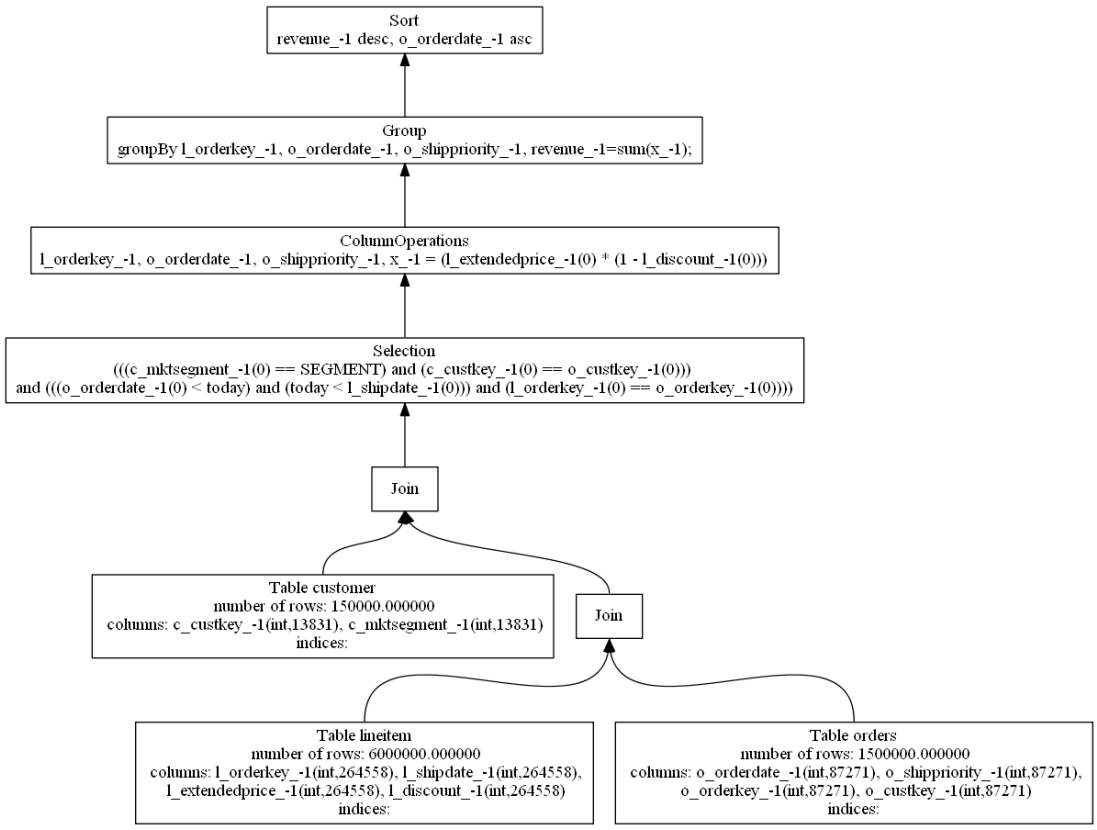


Figure 5.1: Example of algebra tree.

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.

5.3 Semantic analysis and node grouping

This phase in processed by 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 attribute names only this identifier.

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.

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 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, but we can find selection

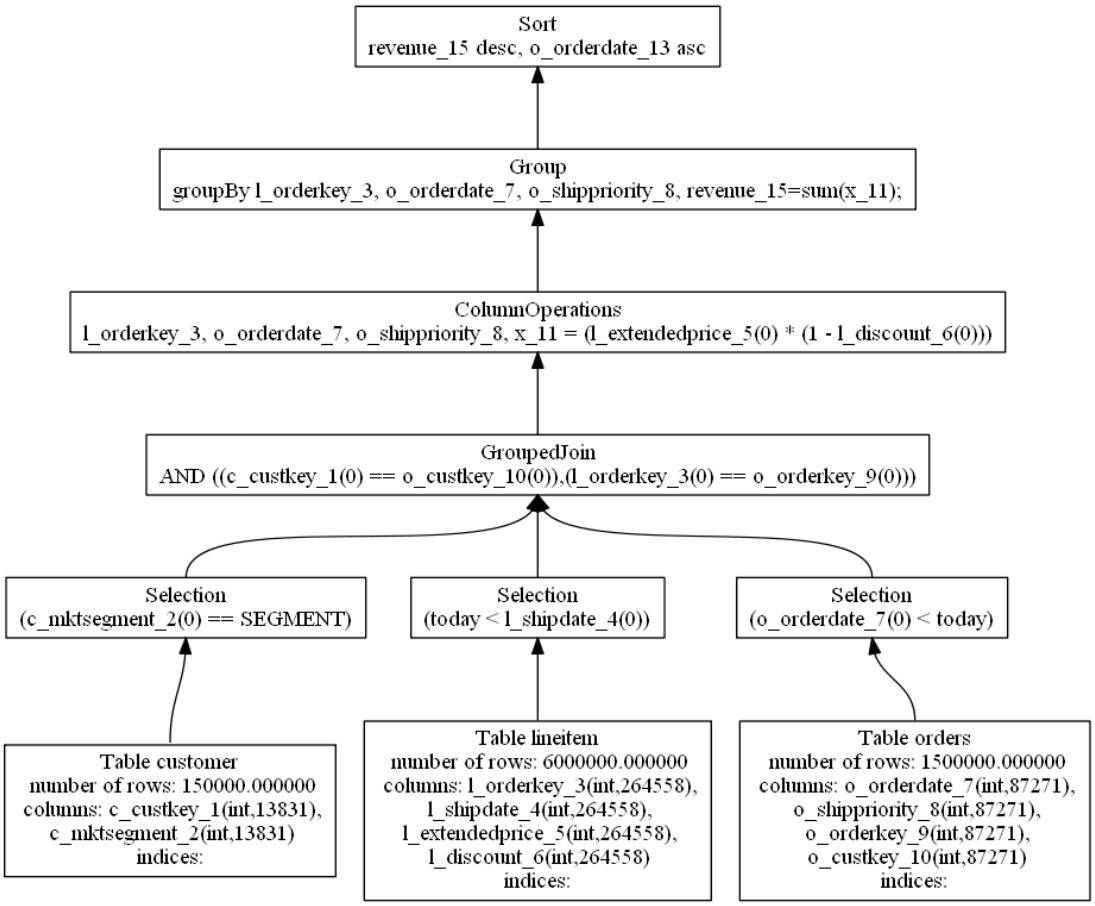


Figure 5.2: Example of optimized algebra tree.

chains 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 5.2 we can see optimizes algebra tree. In comparison figure 5.1, 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 output is optimized algebra tree. We can of course implement more optimizations to improve logical plan.

5.5 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 for 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`.

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

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

Physical plan is represented as polymorphic tree.

In figure 5.3 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 and estimated number of unique values for each column. 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

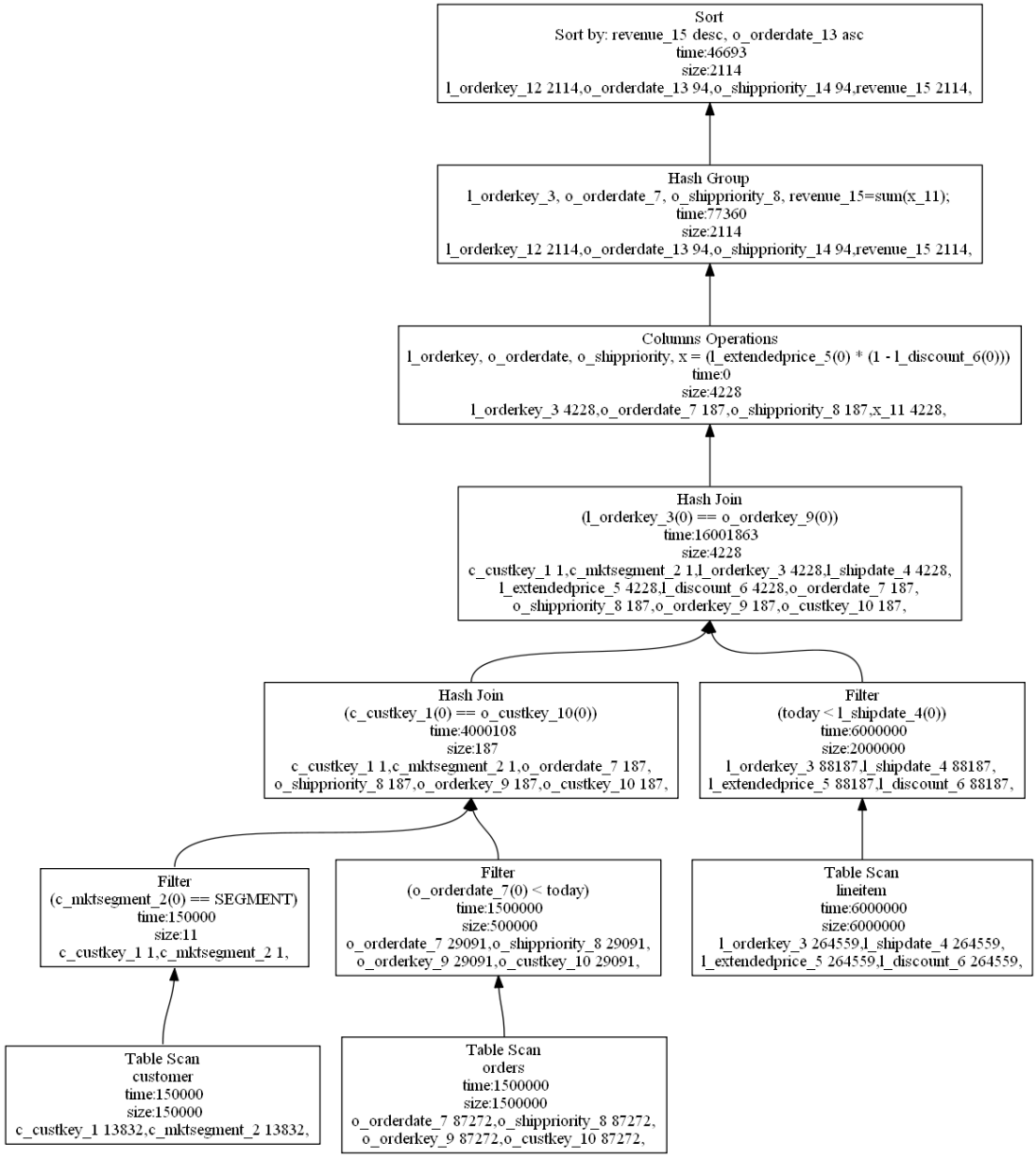


Figure 5.3: Example of physical plan.

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.

Physical operators are chosen based on their estimated time complexity. It is computed from estimated size. In class `TimeComplexity` we have static functions which compute time complexity for each operation. It also contains constants used in this functions. We assume, that this constants or while function need to be improved. This improvement can be done base on testing and measuring evaluation queries in Bobox. At the time of submitting thesis runtime environment is not fully functional.

5.6 Resolving sort parameters

We take generated plans from class `AlgebraCompiler`. Sort parameters of sort nodes need to be resolved. Two plans also can contain same physical operator. That's why need to clone plans, to assure that no algorithm representing object are used in two or more plans.

For cloning we used `CloningPhysicalOperatorVisitor`. Than we resolve generated plans in `SortResolvingPhysicalOperatorVisitor`. In figure 5.4 we can see physical plan with unresolved sort parameters.

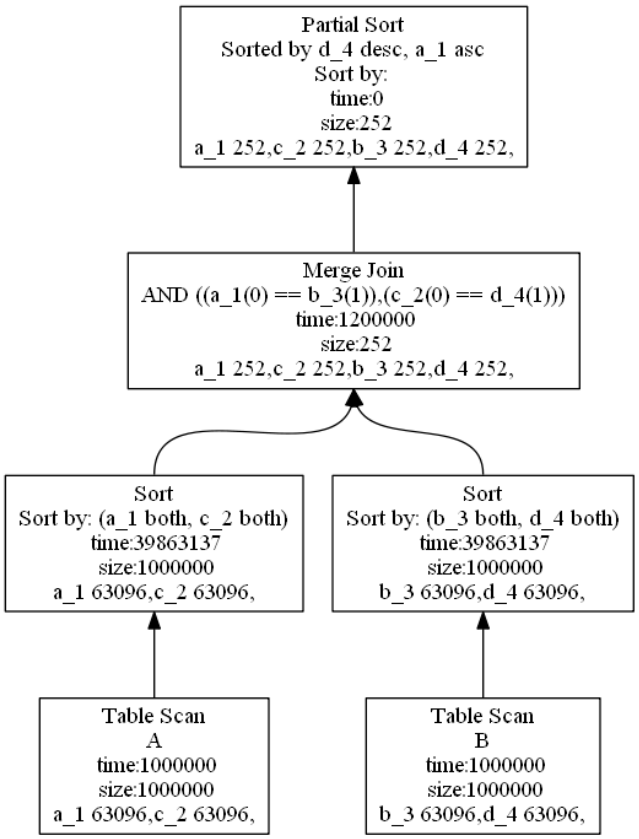


Figure 5.4: Example of physical plan.

It contains two sort algorithms. Left one has following possible parameters:

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

Right sort algorithm has sort parameters:

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

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

`SortResolvingPhysicalOperatorVisitor` goes down tree. It uses variable `sortParameters`. We store there information how the input has been sorted before input of visited node. Using this variable we adjust sort parameters of sort algorithm. Adjusted plan is in figure 5.5.

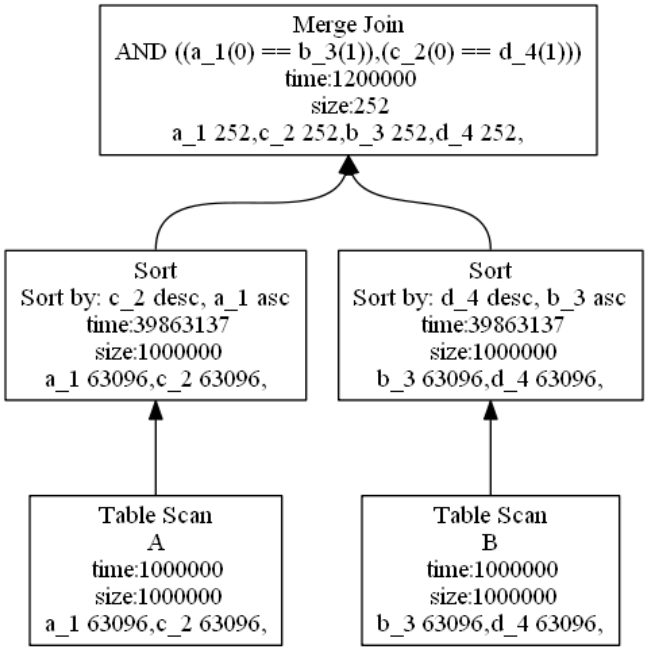


Figure 5.5: Example of final physical plan.

Output od the query has to be sorted by $d : desc, a : asc$. Visitor sets that input of partial sort has to be sorted by $d : desc, a : asc$. After that we resolve in merge join that left input has to be sorted by and right input has to be sorted by $c : desc, a : asc$ $d : desc, b : asc$. Using this information we choose correct sort parameters in sort algorithms.

There can be situations where we use sort based algorithm but output doesn't been sorted, we can choose arbitrary order of sort parameters.

5.7 Output

Output in Bobolang is generated by `BoboxPlanWritingPhysicalOperatorVisitor`. We can also generate output from algebra tree. Visitor `GraphDrawingVisitor` can generate output in dot language. Physical plan's output can be generated

in dot language using `PhysicalOperatorDrawingVisitor`. `PhysicalOperatorDrawingVisitorWithouSorts` provides dot output without partial sorts with empty sort by parameters. In following chapters we text output generated by 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 same. Both 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). Columns 0 and 1 are streams of doubles and column 2 is stream of ints. 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.

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 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 operator. It assumes that input is sorted by group columns. To use it we write *SortedGroup* instead of *HashGroup* in 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 parameter *out*. If it contains number operator, it 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: $2 * (1 -$ (*column number* 2)).

5.7.4 Cross join

Example:

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

left parameter specifies how are columns from first input numbered. *right* parameter specifies numbering columns from second input. Join outputs only columns given in *out* argument.

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");
```

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*. Relation in first input should be stored in hash table, because it's estimated size is smaller. This example computes join with condition:
(*column* 0 = *column* 5) *and* (*column* 1 = *column* 2).

5.7.6 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.

5.7.7 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.

5.7.8 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. Condition is given in parameters *leftPartOfCondition* and *rightPartOfCondition*. Relation in first input should be stored in hash table, because it's estimated size is smaller. This example computes anti join with 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");
```

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.

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 table specified in parameter *name* and reads only columns given in 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");
```

Operator reads whole table given in *name* using *index* and reads columns specified in 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))");
```

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*.

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 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.

5.7.14 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. Second input uses same number of columns like first output. This information is specified in parameter *right*. Operator appends columns from input 1 to columns from input 0. Order of output columns is specified in parameter *out*. Operator unites columns with same numbers.

6. Conclusion

The aim of this thesis was to implement part of the SQL compiler. Created program reads input relational algebra, optimizes it and generated physical plan from it. Output is written in Bobolang.

After the introduction we described Bobox and Bobolang. Next chapter contains theory used to implement query transformer. Chapter analysis contains description of used algorithms and important data structures used in implemented tool. Final chapter describe some implementation details of created program.

Created software is a first part of planed SQL compiler. Front end, which transforms text query to relational algebra, is not yet implemented. At the time of submitting this thesis, not all Bobox runtime operators implemented. That's why we couldn't evaluate any queries to prove that generated plans are correct.

We tested software by transforming some simple queries and queries from benchmark[5] to physical plans. We can only check generated plans by looking generated debug outputs. From the results we can say that generated plans look correct and also optimal. Based on this result we can say, that this thesis fulfilled it's aim.

Implemented tool can be improved by adding more logical plan optimizations. We can add also support for more algorithms like nested loop joins.

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/

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

Attachments

Attachment on CD