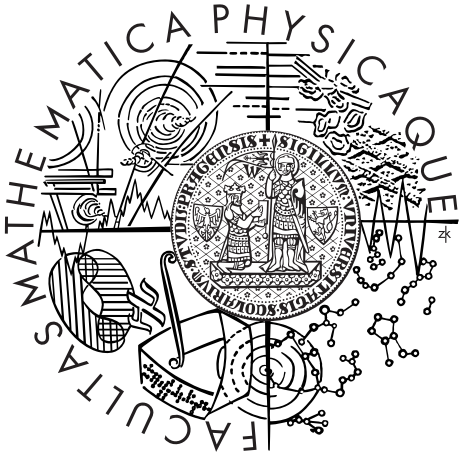
Charles University in Prague

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

Dedication.

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.

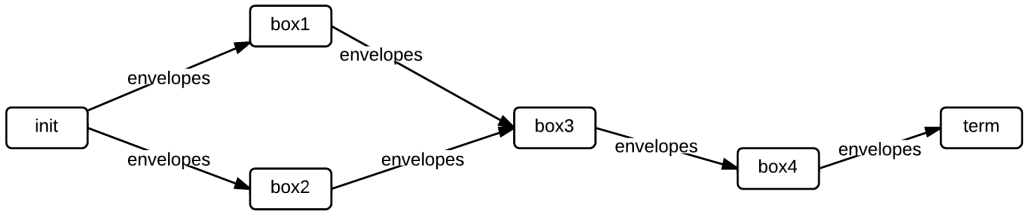


Figure 1.1: Bobox architecture.

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.

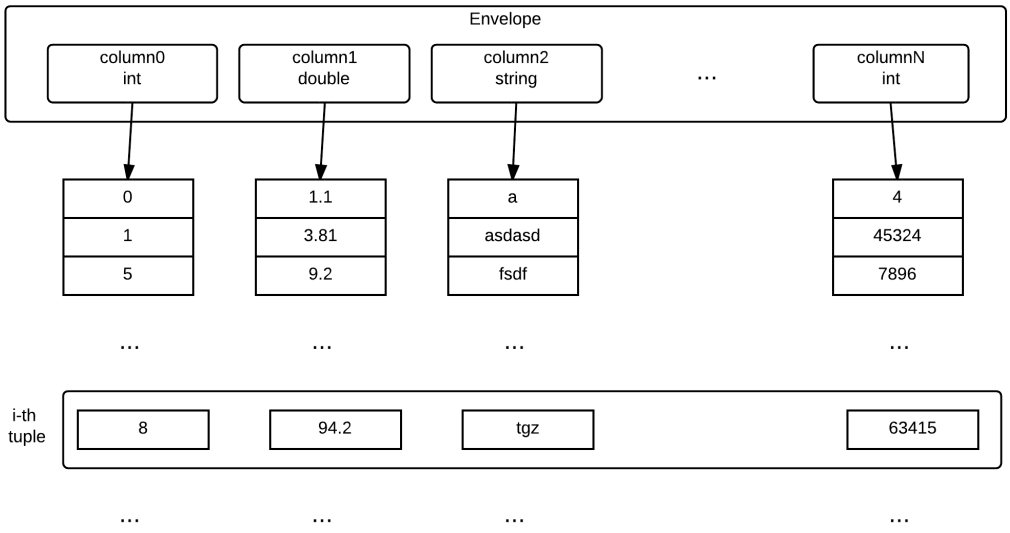


Figure 1.2: Envelope structure.

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.

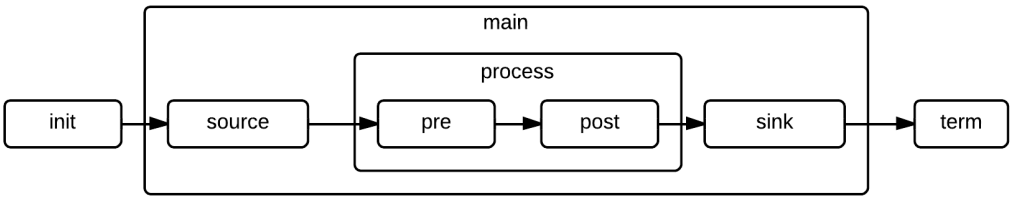


Figure 1.3: Example of execution plan.

1.3 SQL compiler architecture

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

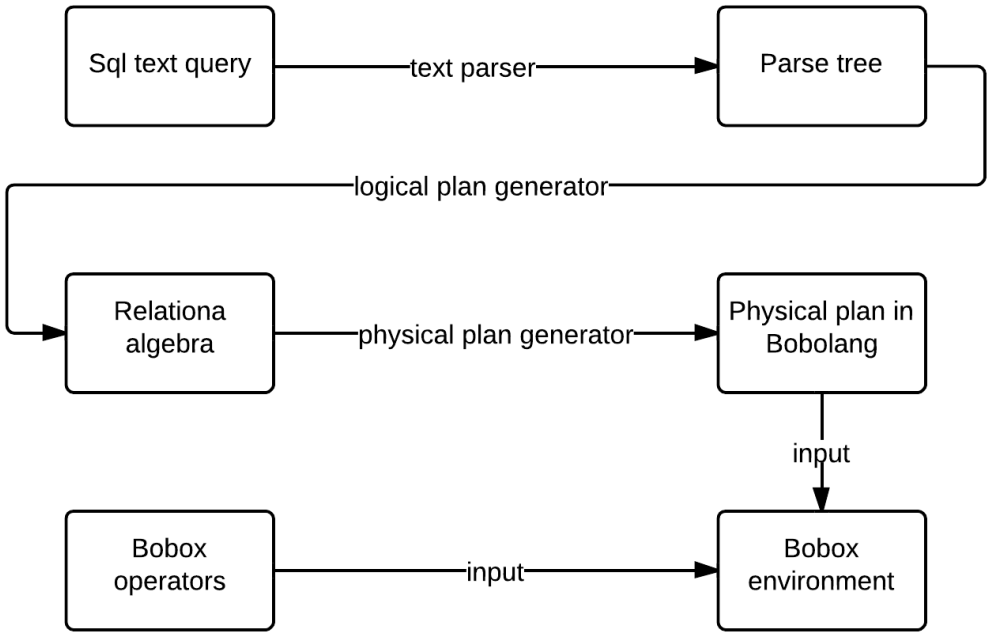


Figure 1.4: SQL compiler architecture.

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 compatible. It doesn't change number ot tuples, their lengths change.

Estimating the size of selection

Selection 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

When selection involves *not* and we have $S = \sigma_{not(C)}(R)$ we can use following 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 can 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

a

Estimating the size of union

Estimating the size of intersection

Estimating the size of difference

Estimating the size of grouping

2.3.2 Enumerating plans

2.3.3 Choosing join order

2.3.4 Choosing physical algorithms

table select join

3. Analysis

4. Implementation

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.

List of Tables

List of Abbreviations

Attachments