VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Savioptimizavimas reliacinėse duomenų bazių sistemose

Magistro baigiamasis darbas

Atliko:
Artūras Lapinskas ....
(parašas)

Darbo vadovas:
lekt. I. Radavičius ....
(parašas)

Recenzentas:
prof. R. Vaicekauskas ....
(parašas)

Vilnius
2016

VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
DEPARTMENT OF COMPUTER SCIENCE

Self-optimization in relational database management systems

Master's thesis

Author:
Artūras Lapinskas ....
(signature)

Supervisor:
lect. I. Radavičius ....
(signature)

Reviewer:
prof. R. Vaicekauskas ....
(signature)

Vilnius
2016

Santrauka

Dauguma šiuolaikinių užklausų planuotojų yra sukurti lipdant specializuotus algoritmus pasinaudojant įvairiomis euristikomis. Tokių planuotojų kūrimas reikalauja daug laiko ir dalykinės srities žinių. Šie planuotojai, dėl palyginus mažo paieškos erdvės tyrimo, taipogi gali sukurti neoptimalius planus. Šiame darbe mes parodome kaip galima užkoduoti *DB* užklausas *SMT* logikomis formulėmis. Toks kodavimas leidžia atlikti užklausų ekvivalentumo patikrinimą. Vėliau, pasinaudodami šiuo kodavimu, sukuriame bendro pobūdžio užklausų planuotoją. Šis planuotojas atlieka savo darbą foniniu režimu, taip nuolatos gerindamas dinamiškai kintančią užklausų aibę. Galiausiai su šiuo planuotoju mes atliekame keletą standartizuotų testų, iš kurių rezultatų padarome tris išvadas. Visų pirma daugelių atvejų planuotojas pasirodo panašiai kaip ir įprastiniai planuotojai. Keletai užklausų šiam planuotojų pavyksta rasti geresnius planus. Tačiau taip pat egzistuoja užklausos planų, kurių palyginimas apribotame *DB* modelyje neveikia, dėl to planuotojas sukuria nekorektiškus planus.

Raktažodžiai Superoptimizacija, *SMT*, Kompiliatoriai, Užklausų Vykdymo Planas.

Summary

Most query planners are implemented as a set of specialized algorithms combined by hand written heuristics. Creating such planner requires an enormous amount of time and expertise, not to mention that due to a rather small exploration of search space planners can return suboptimal results. In this project, we present a way to encode query plans in *SMT* for equivalency checking. We use this to create general purpose planner which does its work in a background, continuously producing better plans for a dynamically created set of queries. We also experimentally test our planner with conventional one under a few standard benchmarks. From these benchmarks, we conclude that most of the time our planner performs as well as a state of the art planner. For some queries, it may also find better plans or, if we can not prove equivalency under the restricted *DB* model, may end up at plans that do not produce expected results.

Keywords Superoptimization, *SMT*, Compilers, Query Execution Plan.

Išplėstinė santrauka

Šiame darbe mes nagrinėjame kaip galima pagerinti duomenų bazių valdymo sistemų (toliau *DBVS*) atliekamą *SQL* užklausų konvertavimą į užklausos vykdymo planus. Užklausos planas – tai medis, sudarytas iš žemo lygio instrukcijų. Šios instrukcijos tiksliai ir be jokių dviprasmybių apibrėžia iš kur ir kaip *DBVS* gali paimti vartotoją dominančius duomenis. Kiekvieną užklausą gali atitikti daugiau nei vienas planas, todėl *DBVS* stengiasi iš visų galimų planų rasti tą, kuris kiek įmanomo labiau sutrumpintų duomenų išgavimo laiką. Šią paiešką *DBVS* atlieka pasinaudodama įvairiomis, iš anksto apibrėžtomis, euristikomis ir algoritmais, atliekančiais pilną perranką tam tikrose paieškos erdvės poaibiuose (pavyzdžiui, optimalų santykių išdėstymą galima rasti pasinaudojant dinaminio programavimo algoritmu, apibrėžtu *IBM System R* duomenų bazių valdymo sistemoje [SAC$^+$79, p. 28]). Deja, tokios paieškos realizavimas *DBVS* reikalauja daug laiko bei dalykinės srities žinių. Tokia paieška taipogi ne visada duoda optimalius rezultatus.

Visam užklausos kompiliavimo procesui retai bandoma pritaikyti bendro pobūdžio optimizavimo algoritmus, kurie išspręstų anksčiau aptartas problemas. Vietoje to nagrinėjama kaip būtų galima perrašyti pirminę užklausą, kad užklausų plano generatoriui būtų lengviau rasti optimalų planą (pavyzdžiui, semantinis užklausų perrašymas tą pasiekia perrašydamas užklausas atsižvelgiant į įvairias duomenų bazių vientisumo sąlygas [CFM86, SSS92, KK07]). Taipogi užklausų veikimo greitis gerinamas ieškant būdų automatiškai kurti indeksus [ACN00, CN98, CBC93].

Užklausų plano paieškos situaciją mes bandome pagerinti *DBVS* viduje pritaikant superoptimizaciją. Superoptimizacija – tai procesas, kurį atlieka programavimo kalbų kompiliatoriai. Šio proceso metu kompiliatorius išsirenka mažą programinio kodo gabalėlį. Tada šiam kodui sugeneruoja atitinkamą instrukcijų seką. Galiausiai kompiliatorius perrenka visas ekvivalenčias instrukcijų sekas, ieškodamas sekos, kurią kompiuteris galėtų įvykdyti greičiausiai.

Darbo metu mes, pristatydami superoptimizacijos raidą, detaliai aptariame kiekvieną iš anksčiau nurodytų žingsnių:

- pirmasis superoptimizaciją pradėjo taikyti Massalin'is [Mas87];
- Denali projektas [JNR02, JNZ$^+$03] praplėtė Masalin'io darbą pradėjęs koduoti instrukcijų ekvivalentumą *SMT* logikos formulėmis (Massalin'is instrukcijų sekas lygino atlikdamas atsitiktinius jų vykdymo testus);
- kadangi visos programos superoptimizacija praktiniais tikslais yra per lėta, S. Bansal'is ir A. Aiken'as parodė kaip galima taikyti superoptimizaciją iš anksto pasiruošiant kodo konvertavimo į instrukcijas šablonus [BA06].

- *STOKE* projektas superoptimizacijai pradėjo naudoti mašininio mokymo algoritmus [SSA13] bei pristatė kaip galima atlikti ekvivalentumo tikrinimą kodui su sąlyginiais sakiniais [SSCA15].

Šiame darbe mes adaptavome aptartą superoptimizacijos strategiją *DBVS* tokiu būdu:

- sukūrėme matematinį modelį, leidžiantį atlikti dviejų planų ekvivalentumo patikrą; Ši patikra atliekama užkoduojant planus ir ekvivalentumo sąlygą *SMT* formulės. Vėliau šios formulės atiduodamos *SMT* sprendžiančiai aplikacijai.

  Stengdamiesi pagreitinti šią patikrą mes apribojame visų santykių dydį iki trijų kortežų. Toks apribojimas leidžia aprašyti visas formules be kvantorių, taip išvengiant su jų naudojimu kylančias problemas;
- pasinaudoję šiuo ekvivalentumo tikrinimu, sukūrėme modulį, atliekantį superoptimizacijas. Šiame modulyje planų paieška vykdoma pasinaudojant darbe apibrėžtomis plano mutacijos operacijomis. Planai lyginami pasitelkus įverčio funkciją, kuri skiria prioritetą žemo aukščio indeksuotiems planams;
- siekdami patikrinti kaip šis modulis elgiasi praktikoje, sukūrėme užklausų perėmėją. Šis perėmėjas dirba perimdamas paketus, keliaujančius tarp vartotojo ir duomenų bazės. Iš šių paketų perėmėjas pasiima jį dominančias užklausas, konstruoja užklausų abstrakčius sintaksės medžius ir leidžia atitinkamam moduliui vykdyti medžio pakeitimus testinėje duomenų bazėje. Vėliau šis perėmėjas lygina modifikuotos ir originalios užklausos grąžintus rezultatus – esant nesutapimams perspėja vartotoją.

  Šio perėmėjo kūrimo etape mes taip pat praplėtėme *PostgreSQL DBVS* papildoma komanda `execplan`. Šios komandos pagalba planai buvo vykdomi *DBVS*.

Galiausiai mes ekperimentiškai ištyrėme sukurto superoptimizatoriaus veikimą vykdydami *OLTBench* testus. Pagal šių eksperimentų rezultatus mes padarėme tokius pastebėjimus:

- sukurtas matematinis modelis yra tinkamas daugumai praktinių užduočių – visi superoptimizatoriaus sukurti planai grąžino tokius pačius duomenis kaip ir *PostgreSQL* sugeneruoti planai;
- 12 iš 21 *OLTBench* naudotų užklausų superoptimizatoriaus planai sutapo su *PostgreSQL* rastais;
- dėl per paprastos įverčio funkcijos ir nepakankamos mutacijų įvairovės, 8 iš 21 užklausų nepavyko suoptimizuoti iki galo;
- vienai užklausai nepavyko rasti optimalaus plano dėl nepilnos plano viršūnių realizacijos.

Darbo metu taip pat pristatėme tris užklausų grupes, kurioms mūsų sukurtas superoptimizatorius pajėgė rasti geresnius planus nei *PostgreSQL*:

- gebėjimas pasinaudoti santykyje apibrėžtomis `NULL` leidžia superoptimizatoriui suprastinti kai kurias užklausas iki vienos išraiškos. Konkrečiu, darbe pateiktu atveju, šis gebėjimas leido gauti planą, kuris veikė 100 kartų greičiau nei *PostgreSQL* sukurtas planas;
- mūsų sukurtas superoptimizatorius, pasinaudodamas indeksų savybėmis, gali išprastinti plane naudojamas `sort` viršūnes. Užklausai, kuriai *PostgreSQL* nepadarė šios optimizacijos, superoptimizatoriaus planas veikė 200 000 kartų greičiau;
- užklausoje aprašytos tranzityvios priklausomybės tarp atributų kartais leidžia panaudoti indeksą, kurio nepavyktų panaudoti kitu atveju. Šiame darbe mes pateikėme pavyzdinę užklausą, kuriai, dėl aptartos savybės, superoptimizatoriui pavyko rasti planą, veikiantį 10 000 kartų greičiau nei *PostgreSQL* rastas planas.

Contents

Introduction

From the early days, humans transferred information on how to benefit from surrounding environment to other generations. Initially, this was done by storytelling, but later, with the invention of written word, it was done by writing valuable information on clay tablets and paper. This data preservation was one of the factors of such immense human expansion. As we collected more and more data, specialized institutions, called libraries, dedicated only for data housing were born. Though libraries were perfect for data catalogization, they had one problem – they tended to be a scarce and centralized source of knowledge. That is, in order to read something about an interesting topic you had to go to a library. What is more, as note copying was a tedious, long and costly process, libraries tended to have the only original copy of work saved on fragile paper. This, combined with aging factor, meant that some rarely read works became unreadable and were lost (at extreme we have one of the most famous libraries „The Royal Library of Alexandria", in which all of the saved works were destroyed due to fire). This problem was solved by the invention of the Gutenberg's press machine. Press machine boomed the number of libraries to the point where almost all cities had one and made personal copies of various works accessible to the general public. Later on, because of digitalization, books and other types of media could be saved in more compact digital form. As such, millions of works could be saved on a simple *USB* storage device which fits in one's pocket.

The easier it gets to store data, the more of it will be collected. The more data one has, the harder it is to find needed bits and process them to get useful information. The easiest way to speed up a search of information is giving your data a fixed structure. That is what databases (*DB*s for short) are – a structured collection of interdepended data. Libraries are *DB*s as books and paper are stored in shelves sorted lexicographically, shelves may be categorized by the first letter of all books in them, and an array of shelves could be put inside of rooms depending on what kind of literature they contain – all this is needed to simplify search procedures. In digital world *DB*s are represented as specialized file formats. There are a plethora of different formats, and different formats save data by using different objects with which one can do particular set operations. Depending on these objects and operations (data model) *DB*s are put into different groups. There are graph *DB*s that are optimized to manipulate graph-like structures, object *DB*s that try to look similar to object oriented languages, relational *DB*s that work with mathematical objects called relations, etc. Obviously, manipulating data in *DB* directly is a hard task. As such, most interfacing with data occurs through *DB* management system (*DBMS* for short) – „software system that manages, and in particular handles all access to, some database or collection of databases" [Dat08, p. 40].

In this paper, we will be looking at relational *DBMS*s – a *DBMS* which handles access to relational *DB*s. When, working with *RDBMS*s, we will be manipulating table-like structures – relations. Relation consists of two parts [Dat11, p. 12]: head and body. A Head is similar to table's header, defining its columns, and consists of the attribute set. Here an attribute is a pair consisting of user chosen name and domain of possible values instances of this attribute can get. A Body, on the other hand, consists of data saved in relation. More specifically, a body is a set of tuples (list of rows). Each tuple has as many elements as there are elements in the head. These elements contain corresponding attribute name and domain (so that it would be possible to map element to attribute) with an addition of value from attribute domain.

Relations within *RDBMS* are manipulated through *SQL* language. This language allows writing down, at a high level, transformations that allow getting needed information as a relation from the initial set of data relations (relations that contain persisted data). Each transformation may have various implementations from which *RDBMS* can choose from and each mentioned attribute can be retrieved going through various places (contrast to a book, where you can search for a particular phrase by reading a book from beginning to the end, or by going to an index and looking only at those pages that contain all words in phrase). As such, for each *SQL* query, *RDBMS* tries to find the fastest access path and transformation combination called an execution plan. Most of the time, *RDBMS* does a great job and finds an optimal plan which allows fast data retrieval, but sometimes, due to compilation time constraints imposed by large search space and anxious user, it is forced to use suboptimal plans.

Assuming that queries are not changing frequently, it would seem that it should be possible to look at a query plan generation as a single global optimization problem which could be solved by using slower offline techniques. However, this is usually not what people are doing. Instead, most constrain themselves in only looking into queries, like doing semantic query rewriting, in which queries are rewritten to something which is closer to the optimal solution by taking into account various integrity constraints [CFM86, SSS92, KK07]. Others try to direct a query plan by automatically creating indexes [ACN00, CN98, CBC93]. As such, we will try to fill this niche spot by trying to apply superoptimization, an offline optimization technique used by programming language compilers, to the query planning.

The goal of this thesis – a creation of query plan superoptimzer. We will achieve this goal by doing following tasks:

- an overview of query processing strategies;
- a literature review of possible offline compilation strategies;
- a definition of the most common query plan nodes;
- creation of a framework which seamlessly enables users to intercept, alter and check results of queries flowing to *DB*;

- creation and an experimental testing of the offline query planning module.

This paper will consist of 4 sections. In the first section, we discuss how *RDBMS*s process their queries and how planning step could be improved by using offline compilation tactics used in programming language compilers. The second section is dedicated to detailing the architecture of the created query interception framework. After that, in the third section, we present a particular module of query interceptor which is used to continuously compile given queries. The main technical contribution of this module is mathematical formalism which allows comparing two plans for equivalency, under the bounded *DB* model. Finally, we show that, in our experiments, *SMT* based query plan superoptimizer performs similarly to the state of the art *PostgreSQL* planner.

1 Background

In this section, we will give all the necessary background needed to understand how queries are processed. We will also discuss superoptimization in greater detail and will briefly explain how it can be included into *RDBMS*s.

1.1 Query processing

User interaction with *RDBMS* is started by the user typing a *SQL* query at the terminal and pressing *enter* key. After that, a query is transferred over the network to *RDBMS*. Inside of *RDBMS*, this query goes through a pipeline of variuos transformations [HSH07]. First of all, a given query is transformed from string representation into internal *RDBMS* structure. This structure is a simple parsing tree, sometimes called abstract syntax tree (*AST* for short). It is needed primarily to simplify further query processing. While doing this transformation, a parser may also try to find as many syntax errors inside of a query and report them to the user. The parser may also do other things, like *view* inlining (replaces *view* names inside of a query to actual definitions), some primitive constant folding, and so on – but, in general, these things are done by other modules, e.g. rewrite engine.

On the second pass, query planner kicks in. Its responsibility is to transform *AST* to the execution plan. This plan is low-level instructions (contrast with high level *SQL* query), once again, represented as a tree that can be easily executed. This transformation is not one-to-one, that is, there are multiple equivalent plans that represent a given query. Here equivalency means that plans produce same output tuples, but plans can actually be different in other aspects, like specific data crunching algorithms used in planner's nodes. So, though these plans will be equivalent in the final result, they will be different in how this result is achieved. As such, some of these equivalent plans will work faster than others and most planners will try really hard to produce the fastest one.

After planner, the plan goes over some other steps like cache manager, *MVCC* management, etc. Finally, it hits executor which executes all instructions encoded in the plan. Though they are interesting in themselves, we are not going to look into these steps in greater detail in this paper, instead, we will concentrate on the planner aspects.

As stated before, everything starts with a user. Previously, this user was a person, but nowadays we are working with large business applications. So, it is no surprise that some of the requirements for the process described above have changed.

When working with humans, *RDBMS*'s put a big stress on a feedback loop. This means that *RDBMS*'s communicates with their users in a human readable language, instead of relying on a user to use more efficient binary format. *RDBMS*'s are also trying to keep users attention by producing at least the first few tuples of a query execution, as fast as possible. If a user is not happy with this result, execution can be stopped no matter in which state it is. If there are any errors, *RDBMS*'s return them to the user as fast as possible, in a form understandable by humans.

Enterprise behemoths, on the other hand, do not really care about fast feedback. They care only about how fast they will receive full output. First thing which hampers query execution is a network. It is not cheap to transfer large queries over the network to *RDBMS*. Human readability also does not help with this problem as it tends to promote usage of excess words (like `fetch next 2 rows only`). As such, applications started to encode their queries as small procedures inside of *RDBMS*. This allowed applications not to clutter the network with large queries, and instead, use short aliases.

*RDBMS*s also introduced a similar feature called precompiled queries. This feature allows applications to send a full query to *RDBMS* only once. On this first call, *RDBMS* puts a query into an internal cache and returns user query id. Later on, instead of sending the query, a user can use this small query id. In addition to that, *RDBMS* vendors allow the initial query to have placeholders which later on can be filled in with appropriate data. Such feature almost replaces procedure use, as described previously.

The only problem with precompiled queries is that in order to use them at full, users have to compile the initial query only once. This headache of keeping track of what is compiled and what is not is resolved by using query pools on the application's side. These pools are created by compiling all the queries that might be needed by the application, and saving their ids on the application's startup (this also has a side benefit: errors in the queries are resolved at the beginning of the application's life cycle instead of an arbitrary time at runtime).

Query pools used with precompiled queries solved a lot of problems with *RDBMS* usage from application's perspective, but they introduced a big problem to the planner – how to make a plan for the query which contains placeholders? The simplest answer – you can not. This is because, depending on concrete values used in a query, the planner can choose different plans. For example, if inside of the query we are using a filter condition `attribute = 1` (part of the `where` clause) which filters almost all tuples (if, for example, all tuples contained `attribute = 2`), then the query planner most probably will implement this condition as an index scan. However, if our condition will leave out almost all the tuples – a sequence scan usage is better because a random heap access incurred by the index scan will be too costly.

So, in general, *RDBMS*s can not compile queries into an execution plan until it knows all the values used in the query. However, a compilation is considered one of the query execution bottlenecks (in the same way the network is), so most *RDBMS* vendors still try to do that. They can do that because a generated plan depends only on the query and the attribute value distributions (as showed in the previous example). So, if we fixate a distribution, we can compile a query plan in advance. However, what distribution should one use for a filter with an unknown value? The most conservative choice, used by most vendors, is to use the worst distribution possible [Win12, p. 32].

Using a worst distribution means that the planner will tend to generate slower plans than it could. For example, a query with a parameterized filter condition `attribute1 = $1 or $1 is null` will always use a sequence scan over the whole relation as *RDBMS* does not know what filter will be needed in advance (whether instantiated parameter will not be `null`). Obviously, this is not what the user expects. As such, *RDBMS*s try to use various tricks. For instance, *PostgreSQL* for the first 5 executions of a query will compile plan only at the very end – then all values are instantiated. On the sixth execution, *PostgreSQL* will create a plan disregarding actual values, and will check if its cost is close to the average cost of the first 5 plans. If so, this general plan will be used for all the upcoming requests (for a some predetermined time) skipping the compilation stage. If general cost of the plan is too large, *PostgreSQL* will continue to create a plan, for each request, individually [Pos15b].

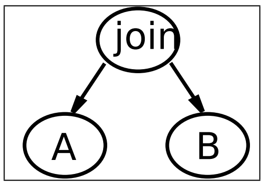


Figure 1: Usual join order

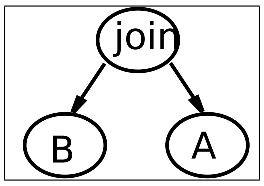


Figure 2: Reversed join order

Figure 3: Two possible ways to join two relations

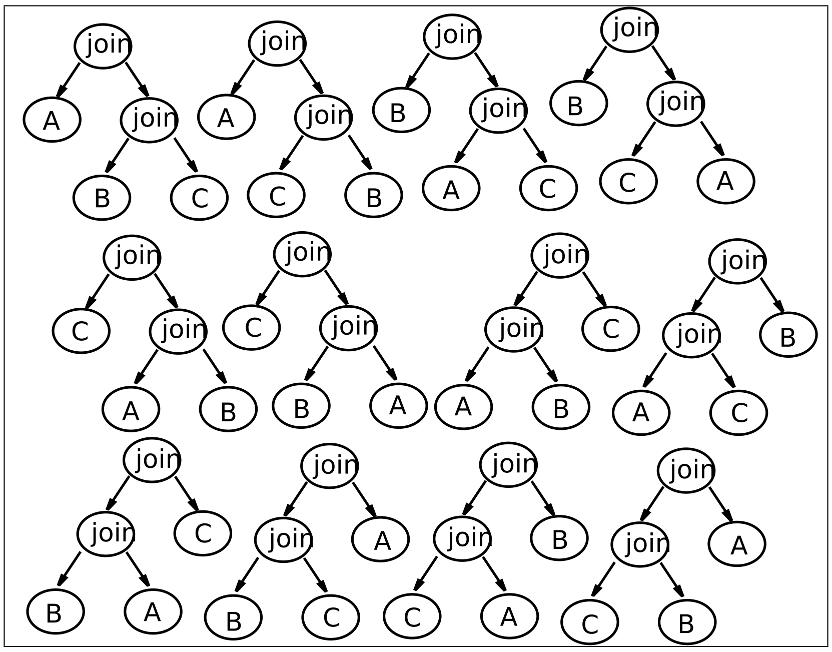


Figure 4: All possible ways to join three relations

Other *RDBMS*s use similar techniques, like bind peeking and parameter sniffing, but no matter what kinds of tactics *RDBMS*s choose to use, one thing remains – they all agree that this additional complexity is worth it, as the query compilation is slow. But how slow could it be? The answer to this question can be found by looking at simple query `select A.*, B.* from A, B`. In this query, planner does not have a lot of choices or, to be exact it, has only one choice (assuming that *RDBMS* supports only nested loop join) – what is better, to join `A` with `B` (Figure 1) or `B` with `A` (Figure 2)? Joining `A` with `B` will be more beneficial when `B` is small enough to fit into memory, as then executor could do a slow sequential disk read of `A` tuples combined with a fast repeated read of the relation `B` from memory. The reverse is also true – if `A`, is small then it is better to join `B` with `A`. So, for two tables we have two possibilities. For three tables we have 12 possibilities – all of them are enumerated in Figure 4. For n tables, we would need to test $n! \times C_n$ – a number of all the possible leaf permutations times the number of full binary trees[1]. So, even with a moderate number of relations, it is impractical to do a full search.

What most *RDBMS*s do with join order problem, is some alternation of the dynamic programming solution proposed in IBM's System R database [SAC$^+$79, p. 28]. Though this solution pushes the limit of a number of joined relations, for which it is possible to find an optimal solution, it is still computationally too expensive for a large number of relations. For queries containing many join clauses, *RDBMS*s fall to different imprecise algorithms like timed genetic algorithms, used in *PostgreSQL* [Pos14a].

The compilation of a general query is an even tougher problem than join ordering. Here we have to check all the possible algorithms that can be used in implementing the join clauses, which indexes to use, different orderings of the filter predicates, predicate propagation, constant folding, etc. So, the real search space is beyond anyone's reach. That is why, if *RDBMS* founds itself in a place where it can skip compilation, even if it introduces some problems, it will do that.

It would seem that a search space is so large that compilation should almost never terminate, but in reality, it finishes in seconds (seconds for human beings sitting behind the terminal, for which *RDBMS* there initially created, looks like ages). This is because no one searches for an optimal plan. Instead, *RDBMS* planners are using a number of small algorithms (like a dynamic programming solution for join order) glued together with hard-coded heuristics. This means that for each planner we can find optimizations that it can not perform.

This can be even seen from the *SQL* queries. For example, in Figure 5, we can see a simple query which takes a union of two relations, sorts them and takes top 10 results. An execution of this query on test environment took $3.9s$. We can help the planner to find a better plan for this query by giving a new query whose direct translation (without optimizations) yields a plan closer to optimal (Figure 6). In this new query, we do everything exactly the same, but, instead of taking relations as is, we pre-sort them and

[1]here C_n stands for n'th Catalan number

```
drop table if exists rel1;
create table rel1 (attr1 float);
insert into rel1 (attr1)
  select random() from generate_series(1, 10000000, 1) as gen;

explain analyze ((select * from rel1)
                  union all
                 (select * from rel1))
                order by attr1 limit 10;
-- QUERY PLAN
-- Limit (cost=720687.36..720687.38 rows=10 width=8)
--       (actual time=3976.888..3976.890 rows=10 loops=1)
-- -> Sort (cost=720687.36..770687.24 rows=19999954 width=8)
--         (actual time=3976.887..3976.887 rows=10 loops=1)
--    Sort Key: rel1.attr1
--    Sort Method: top-N heapsort  Memory: 25kB
--    -> Append (cost=0.00..288495.54 rows=19999954 width=8)
--              (actual time=0.019..2484.937 rows=20000000 loops=1)
--       -> Seq Scan on rel1 (cost=0.00..144247.77 rows=9999977 width=8)
--                           (actual time=0.019..675.376 rows=10000000 loops=1)
--       -> Seq Scan on rel1 (cost=0.00..144247.77 rows=9999977 width=8)
--                           (actual time=0.171..798.876 rows=10000000 loops=1)
-- Total runtime: 3976.905 ms
```

Figure 5: Execution of suboptimal query with limit clause

take first 10 tuples. Plan for this query is executed in under $2.9s$ – a win in logarithmic parameter of sorting complexity (instead of having asymptotic sorting complexity of $O(2n \log(2n))$ we have $O(2n \log(n))$), [Pos15a].

To sum up, initially *RDBMS*s were created to assist users in data management by hiding a lot of complicated processes behind human readable and editable *SQL* language. One of these processes is *SQL* compilation to machine readable and fast executable plan. It does this translation by doing a search over a vast search space of the equivalent plans. Because of the computational expensiveness of full search and user impatience, *RDBMS*s employ various heuristics that most of the time give good enough plans in a timely fashion. After some time, human user faded away and was replaced by the enterprise applications. These applications do not use dynamic queries as humans do – they prepare large precompiled query pools in advance. One of the many advantages of these pools is that sometimes when right conditions are met they allow *RDBMS* to skip the compilation process altogether by reusing old plans.

1.2 Superoptimization

As stated before, our goal will be to look on how to improve the query plan generation process, but before describing a general idea lets look how a similar problem – producing an optimal executable format from an abstract form – is solved in programming languages that are transformed into machine code by the compiler. Though similar in nature, this problem is somewhat simpler than creating an optimal plan for the *SQL* query. This is primarily due to the two facts:

- the cost for executing a particular piece of machine code is well defined – most

```
drop table if exists rel1;
create table rel1 (attr1 float);
insert into rel1 (attr1)
  select random() from generate_series(1, 10000000, 1) as gen;

explain analyze ((select * from rel1 order by attr1 limit 10)
                  union all
                 (select * from rel1 order by attr1 limit 10))
                order by attr1 limit 10;

-- QUERY PLAN
-- Limit (cost=687123.93..687124.18 rows=10 width=8)
--       (actual time=3234.362..3234.367 rows=10 loops=1)
-- -> Merge Append (cost=687123.93..687124.43 rows=20 width=8)
--                 (actual time=3234.360..3234.365 rows=10 loops=1)
--    Sort Key: rel1.attr1
--    -> Limit (cost=343561.96..343561.99 rows=10 width=8)
--             (actual time=1794.350..1794.351 rows=6 loops=1)
--       -> Sort (cost=343561.96..367234.64 rows=9469072 width=8)
--               (actual time=1794.350..1794.351 rows=6 loops=1)
--          Sort Key: rel1.attr1
--          Sort Method: top-N heapsort  Memory: 25kB
--          ->  Seq Scan on rel1 (cost=0.00..138938.72 rows=9469072 width=8)
--                               (actual time=0.020..1020.129 rows=10000000 loops=1)
--    -> Limit (cost=343561.96..343561.99 rows=10 width=8)
--             (actual time=1440.008..1440.009 rows=5 loops=1)
--       -> Sort (cost=343561.96..367234.64 rows=9469072 width=8)
--               (actual time=1440.006..1440.006 rows=5 loops=1)
--          Sort Key: rel1_1.attr1
--          Sort Method: top-N heapsort  Memory: 25kB
--          -> Seq Scan on rel1 (cost=0.00..138938.72 rows=9469072 width=8)
--                              (actual time=0.036..695.884 rows=10000000 loops=1)
-- Total runtime: 2847.414 ms
```

Figure 6: Execution of improved query with limit clause

CPU come with the manuals detailing the execution cost of each of its instructions in terms of *CPU* cycles [GA05]. We can contrast this with evaluating the score of the query plan, where we have to take into consideration a speed of the hard drive's sequential read, a speed of the hard drive's random page read, a speed of the memory, whether some data subset will fit into memory without knowing an exact amount, etc.;

- machine code instructions are known by the community, can be written and tested directly. Query execution plans, on the other hand, are internal *RDBMS* structures that can not be easily manipulated without changing *RDBMS* itself.

As such, there is a lot more scientific work put into improving the compilation process.

At the core, most modern compilers nowadays use peephole optimizers as their primary source of the machine code optimization. A peephole optimizer works by going through already generated machine code, searching for the code segments that can be replaced with the instruction sequence which is either shorter or can be executed faster [McK65]. These replacements are predefined inside of compiler itself in simple template language and are constantly evolving with each new release of a compiler as better and better replacements are found.

It is obvious from the description that peephole optimizer can only do local changes. As such, they are not enough to produce an optimal sequence of instructions. That is why modern compilers do not limit themselves to using only one single optimizer. Most of them use other optimization passes and techniques, like complex data flow analysis tools that are able to detect and optimize out the dead code or resolve constant expressions, loop specific optimizers (unrolling, splitting, unswithcing, motion, etc.), static analysis passes, profile guided optimizations [GMZ02] (that may not produce same instruction set after several compilations), and so on.

Though very useful in practice, these techniques are also incapable of producing something that we could call optimal – only way better than original. That is why some people tend to refer to these techniques as improvers and not as optimizers (this oxymoron is best described in [BA06, p. 1]).

The first true optimizer (from now on we will call optimizer that produces optimal code a superoptimizer) was written by Massalin [Mas87]. Massalin's superoptimzer is based on a very simple brute force approach. It iteratively generated all possible instruction sequences starting with a sequence containing single instruction, then two, then three and so on. Whenever an instruction sequence equivalent to optimization target is found, superoptimizer stops and returns it as a final optimization result. Such approach allowed superoptimizer to produce shortest instruction sequence carrying the same procedure as the original one (all work was done on Motorola's 68020 instruction set, in which all instruction would take a single *CPU* cycle – there, shortest meant fastest).

Though such superoptimzer worked well in practice and could work with an instruction set with up to 13 instructions, it could return incorrect results. The problem was the equivalency checking part. In his original paper, Massalin describes an equivalency checking procedure based on a mathematical logic that would be able to guarantee that his superoptimizer would always return correct results, but it was computationally expensive. As such, his superoptimizer used a simple unit testing approach. For each generated sequence, it generated a few random data sets and checked whether running input sequence on these data sets produced the same result as running generated sequence – if so, both sequences were considered to be equivalent.

The next major step in superoptimization was Denali [JNR02], [JNZ$^+$03]. Instead of iteratively trying all possible sequences, Denali used templates and E-graphs to generate possible sequences. It tested generated instruction sequences by encoding them in the format used by *SMT* solver (this kind of solvers will be discussed later in this paper). This allowed Denali to have strong equivalence guarantees.

Though Denali's project showed that it is possible to have 100% correct superoptimizer, it was still infeasible to use it in practice while compiling whole programs. First of all, Denali was not able to encode complex interactions inside of the instruction sequences – mainly the loop constructs – with *SMT*. Secondly, generating an optimal instruction sequence containing only a few instructions was still a complex and, what is more important, a very slow task.

The first project which was able to improve compilation of general programs was described in [BA06]. Instead of trying directly superoptimize generated machine code, authors were using superoptimizer to automatically generate replacement rules for peephole optimizer. That is, they were detecting potentially optimizable instruction sequences on the fly and were pushing them to the optimization queue. Later on, the superoptimizer would take those pieces and would try to produce faster replacements off-line (after the actual compilation). These replacements would be later added to the compilers rule list – so improving the quality of the generated machine code for upcoming compilations. This technique proved to be very valuable in practice, and now compilers from *GCC* and *LLVM* include rules found by superoptimizers.

Since automatic peephole generation, superoptimizers have improved a lot. They started using machine learning techniques, like Monte Carlo tree search, to minimize search spaces [SSA13], and even started to work with some looping constructs [SSCA15].

Going back to *RDBMS*s and queries, we are in a similar position. A search space is too large, so, *RDBMS* used specific algorithms (like compiler using specific loop optimizations for loops and integer arithmetic expansion to optimize division by a constant) and heuristics, to come up with a decent suboptimal solution. Now, knowing that queries in pools are changing rarely, it would seem that *RDBMS*s could use more exhaustive search algorithms in the background. That is, *RDBMS* on the first query access could generate any sub-optimal plan, put it to internal cache and return it to the user. On any of the subsequential calls for the same query, *RDBMS* would skip compilation and return the cached plan. In addition to that, *RDBMS* could start searching for a better plan in a separate thread. Whenever a better plan is found, it could replace the cached version. Such process would imply that user would get continuous improvements on his query execution times. This is the idea that we are going to develop in this paper.

2 Architecture

In this section, we will start presenting a created optimizer (similar to superoptimizers discussed in the previous section) by beginning with its architecture. The project is split into three actors: a client, a proxy and *RDBMS*. These actors are implemented as services that can run on different machines. While working on this project locally, these services are run on *VM*s managed by *Vagrant*. The exact layout of *VM* machines can be seen in Figure 7.

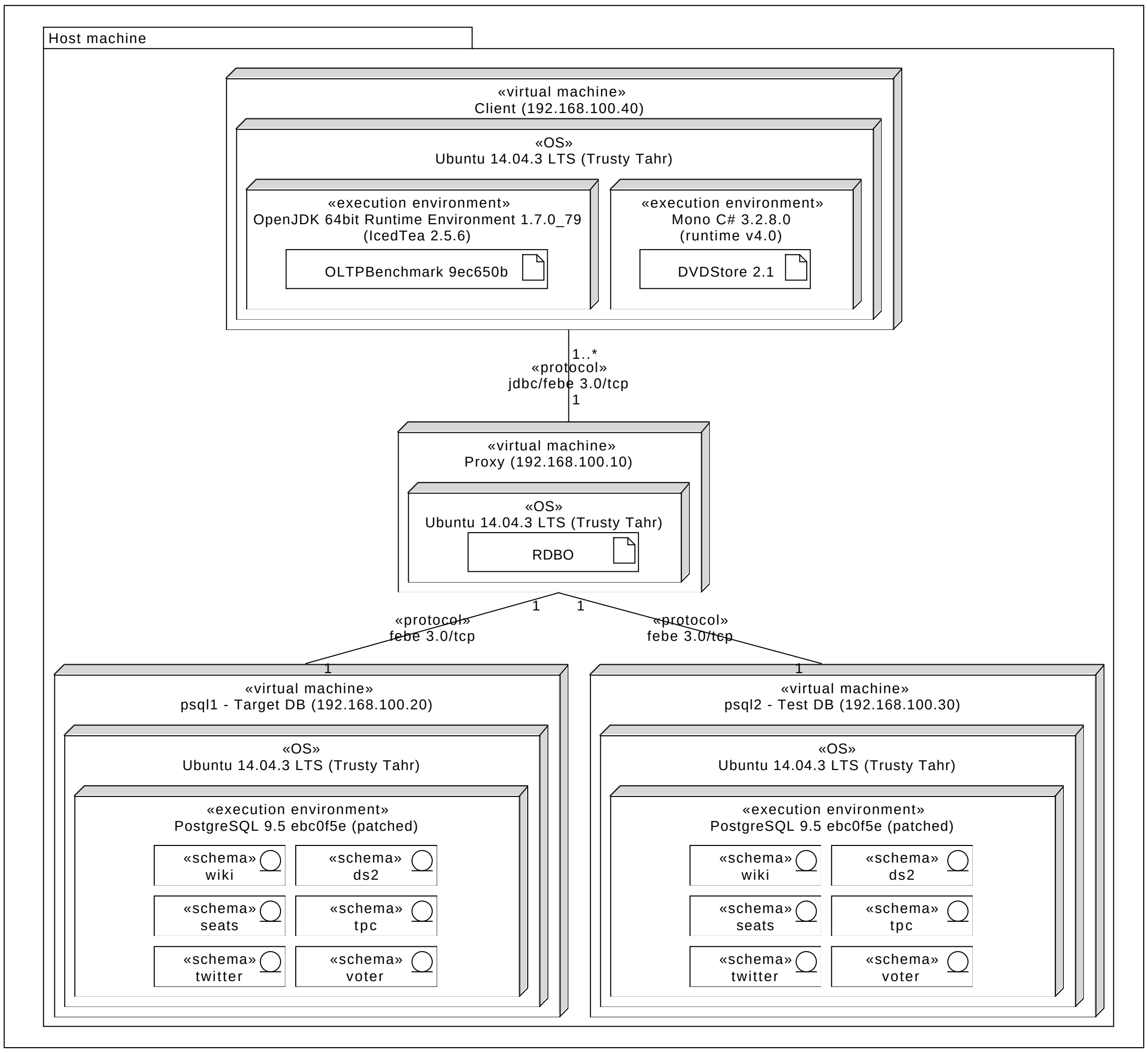


Figure 7: Deployment on *VM*

We can see from this figure that client is running on the machine with *IP* `192.168.100.40` . It consists of two applications: *OLTBench* and *Dells DVD stores simulation*. Each of these applications connects to the proxy server and executes various *SQL* queries over *JDBC* or *ADO.NET* connectors.

The primary application of this project (called *RDBO*) run inside of proxy server `192.168.100.10` on port `8123`. It intercepts all queries, parses them to internal representation and generates query execution plans. *RDBO* then executes these plans on *PostgreSQL* instance, running on `192.168.100.20` and the original query on *PostgreSQL*, sitting inside of machine with *IP* address of `192.168.100.30`. Results of both executions are compared against each other and results of the query plan are returned to the appropriate client application.

In the upcoming section we are going to discuss responsibilities and implementation of these actors in greater detail.

2.1 Proxy

As mentioned before, the proxy is one of the main actors and is responsible for quite a few things. In this section we will discuss what tasks it has to do, before going to its main loop in detail. We will also show how proxy deals with the query interception, parsing and plan execution. Finally, we'll see what kind of precautionary measures it uses to assure quality of its work.

The proxy is also responsible for the query plan generation, but being one of the most integral topics of this project, the plan generation, deserves separate section and will not be discussed here.

2.1.1 Initialization

Before processing, any query *RDBO* has to do some initialization procedures.

First of all, it has to know the connection parameters for the target and test databases. These parameters are passed to the *RDBO* executable on its start, via command line arguments by the user.

Secondly, *RDBO* has to connect to one of these *PostgreSQL* instances in order to load schema information from the appropriate database into the memory. This is needed primary for two reasons:

- a parser component of *RDBO* needs to have the information about schema, to resolve some more complicated structures, before converting query into an internal format. For example, a resolution of the natural joins requires finding an intersection of the attribute sets of joined the relations. The only place from which such information can be acquired is schema itself;
- the planner uses powerful deduction mechanisms to compare two query execution plans. With every bit of information we give to it, the probability of getting false negative results decreases. As such, it is very beneficial to feed these mechanisms with information like inclusion dependencies, unique constraints, etc.

Such schema caching allows all these operations to get appropriate information with ease and, what is more important, fast. The only restriction which is imposed by this process is schema's immutability. Data within schema can not change while *RDBO* is running.

Finally, after all the caching work, *RDBO* can start busy loop on which it listens for incoming connections.

2.1.2 Proxying

When *RDBO* gets a connection from client, it immediately makes an appropriate connection to the test database on a separate thread. After that it enters listen/execute cycle on which it tries to proxy the whole interaction.

Overall, proxying the interaction is not the only thing it has to do as it also has to intercept all queries. As such, simple getting a *TCP* packet from the client and sending to the server (or vice versa) is not sufficient. What it actually does is getting packet from the client, parsing its contents as defined by the *PostgreSQL* message formats [Pos14d]. Then, it tries to understand the meaning of this message and do appropriate action. For example, if *RDBO* receives a *SSL* encoded protocol request from the client, it sends a message saying „unsupported" back to the client (as we do not support this feature) and waits for next step from the client – no message goes to the server.

As it can be seen, modelling all the possible interactions directly might be too complicated. That is why proxying is implemented as a specialized automata. This automata is encoding by lifting *RUST* macro system in special purpose *DSL* and its whole specification can be found in `proxy/src/protocols/postgres.rs`. One big advantage of using this kind of automata is that it is easy to visualize all possible states and transitions in it. Such visualizations are outputed on every client connection by *RDBO* logging module as *Graphviz* graphs and are presented in figures 8 and 9[2].

That we can see from these graphs is that *RDBO* automata closely imitates subset of *PostgreSQL*s *FEBE* protocol [Pos14c]. All the edges of these graphs contain two parts – input and output, separated by the forward slash character (which somewhat reassembles Mealy machines):

- input is encoded as a direction (can be either `c` indicating client or `s` – server), message type pair (elements are separated by colon). So, for example, `c:StartupMessage` indicates that transition happens when we get `StartupMessage` from client.

  As *RUST* does not provide proper selection mechanism, detection of the side which is sending the message is implemented as *FFI* to `pool(2)` which has one problem – on an idle connection it starts a busy loop (this does not really matter for *RDBO* as clients are always attacking *RDBMS* with queries and properly end their connection by sending `Terminate` message);

[2]These figures lack edges for various completion messages like `CommandCompleted`, `BindComplete`, `ParseComplete` and `NoData` as they do not have any useful meaning and just clutter visualization.

- output – an action which will be executed. In general, this action can be any function implemented as `proxy/src/protocols/postgres.rs:StateMachinesEdge` , but in most cases we use default action which is to relay packet to other direction. An encoding of such default actions inside of graphs is exactly the same as input – direction and message type tuple.

As an edge representing simple packet proxying appears quite a few times, we sometimes use a shorthand notation in which input and output directions are written together within curly braces. For example, `{c/s}:StartupMessage` means that on an event of `StartupMessage` message from the client, we pass exactly the same packet to the server.

Knowing how automatas represent their data, we can continue by looking what they represent.

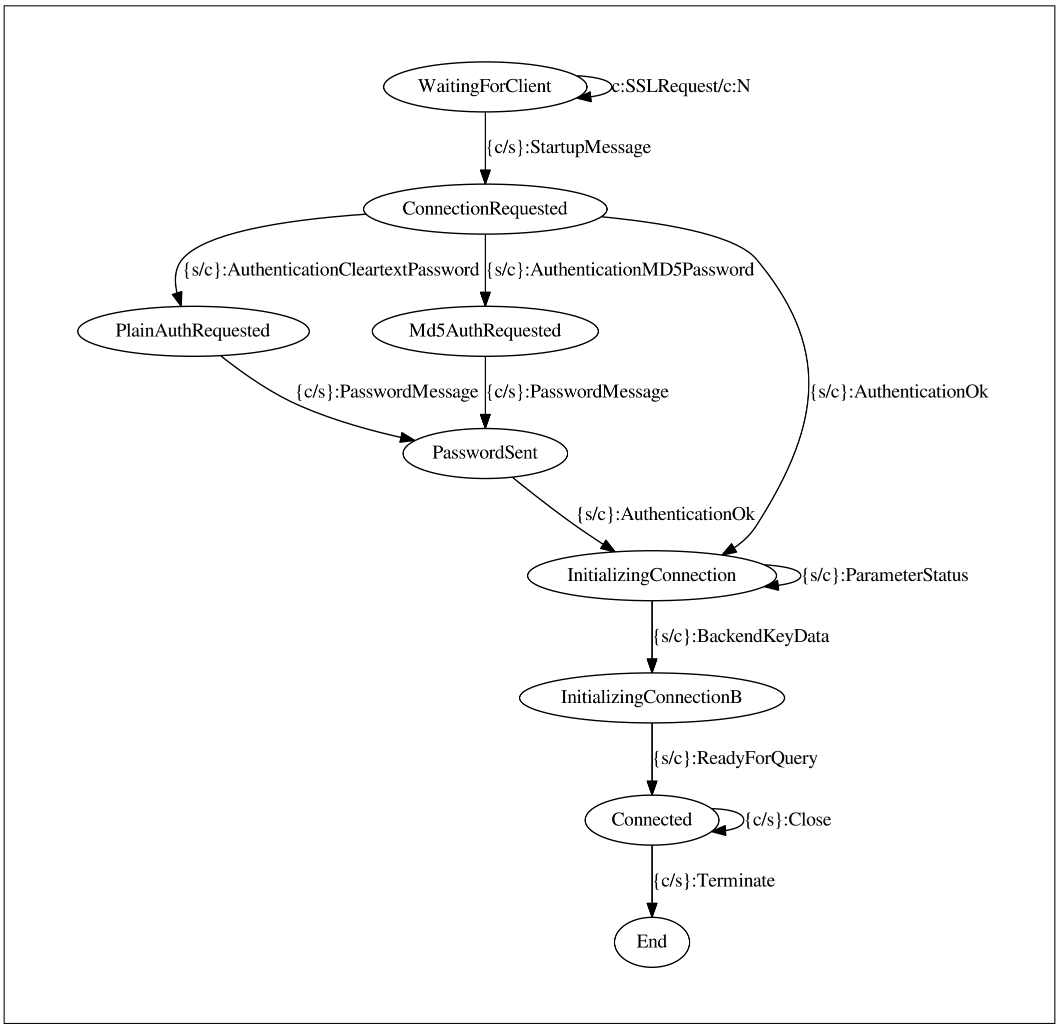


Figure 8: State machine for connection

First automata (Figure 8) handles a client connection which includes:

- denying *SSL* request if any;

- handling authentication (please note that in this phase all passwords are logged as part of the protocol message logging mechanism – use in production environment is highly disadvised);
- collecting authentication parameters, which are later used to connect to target *RDBMS*;
- handling of *GUC* variables – taking care of things like query encoding and parameter formats (this information is used in later parts of interceptor).

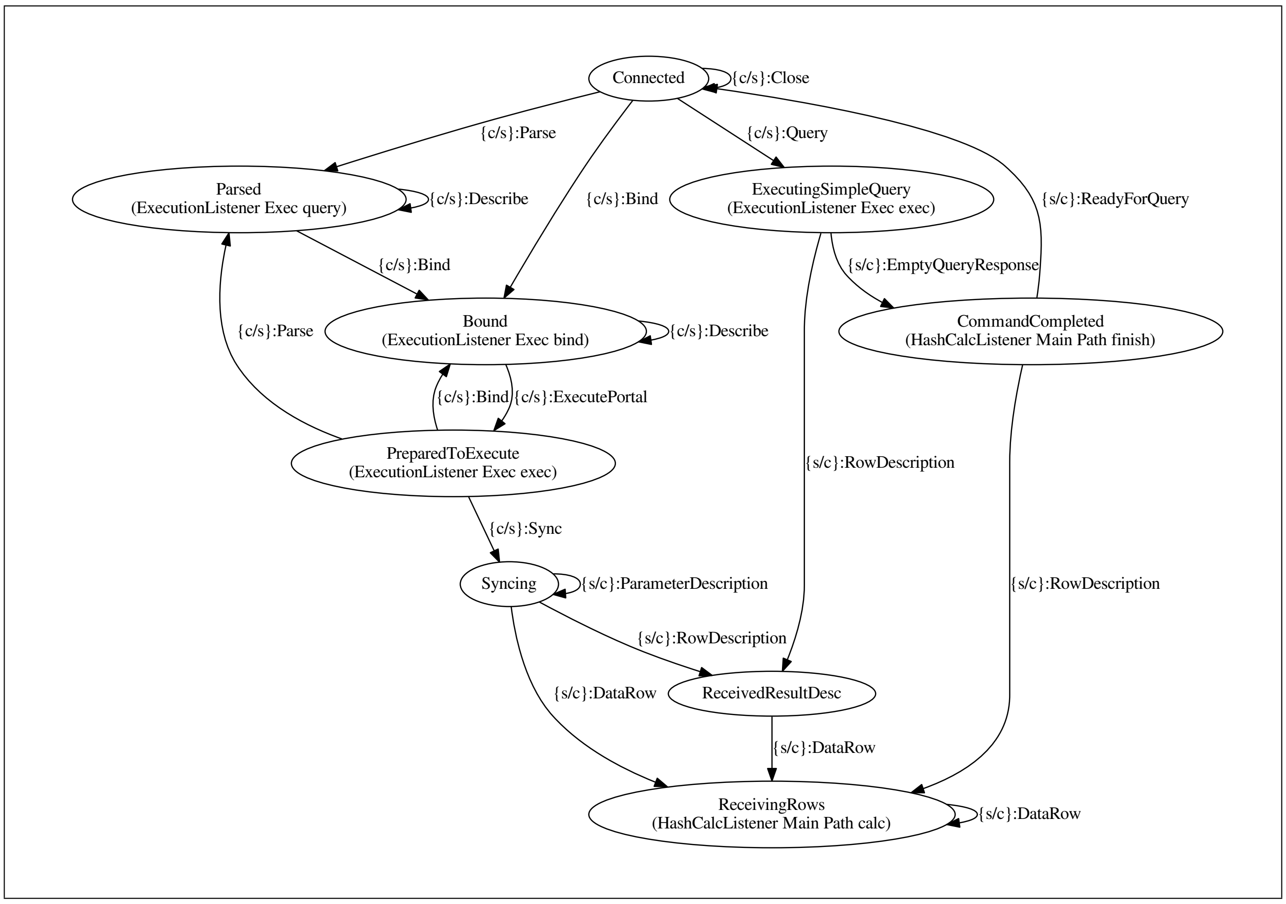


Figure 9: State machine for query execution

Second automata is more complicated and represents two types of queries:

- Simple query – is executed when a user supplies a query as a string;
- Extended query – in this mode a user first compiles a query, giving it some sort of name (and actually creates an associated *PostgreSQL* execution portal). After that the user can bind concrete values to it (marked with the dollar signs in the original query) by giving the parsed queries name and all values in a single message. Finally, the user can execute a query and receive resulting tuples from *RDBMS*. Binding and execution can happen at any time so *RDBO* has to collect and keep all the necessary data inside of the memory.

2.1.3 Query interception

The next thing which needs to be discussed is query parsing, but before that we have to get queries. Though automata described in the previous section allows us to intercept all messages and convert them to internal *RDBO* format, it does not carry any other responsibilities. Instead, it allows an arbitrary number of external listeners to attach to its nodes. These listeners are fired when communication between client and server reaches certain state and get all data needed for further processing (edge by which we came to the state, stored *GUC* parameters, etc.). These listeners are also shown between brackets inside of automata's states. Their names are split into three parts:

- a name of the internal *RDBO* structure[3] as it is written in *RUST* code which is listening on a particular state;
- an arbitrary name designated for a particular instance of the structure;
- a name of the method which will be called.

A query interceptor is implemented as one of these listeners and is named `Execution listener`. Its implementation can be found in `proxy/src/protocols/execution_listener.rs`.

The first time this listener is fired, it creates a connection to target *RDBMS*. Then, it creates exactly the same automata as was used for its call and associates it to this new connection. After that, it replies authentication procedure on this automata – this pushes target *RDBMS* to a state in which it is able to accept queries.

After that, the query interceptor processes queries. For simple queries, this is easy – we just take it and pass to other modules. However, for extended queries there, is a little bit of additional work that has to be carried out:

- inside of the `Parsed` state, the query interceptor gets a query which may contain placeholders. As such, it has to collect and wait for other user actions;
- inside of the `Bound` state, a particular query (each compiled query get a dedicated name, unique to the execution portal) gets all values that should be filled in, instead of placeholders. The query interceptor takes these values, encodes them as strings, and pushes to the copy of original query.

  There are also a few limitations that need to be pointed out here:

  – bound values can come in two forms: binary and text. Currently, we support only a former format and throw exception on any binary value (*JDBC* uses text format, so here such limitation does not have a large impact);

[3]Structures are used as classes in other languages.

  - each value comes with its type *OID*. In order to properly process values, we have to have knowledge of these *OID*s. Instead of asking *PostgreSQL* backend about these *OID*s, we include a subset of mappings between *OID* to actual name inside of *RDBO* itself. These mappings are taken from *PostgreSQL* source code file `src/include/catalog/pg_type.h` and, while they could change between versions, they are pretty stable in practice. A listing of the supported types and their *OID*s are shown in Figure 10.

- inside of the `PreparedToExecute` state, a query, which is now fully encoded as a string, is processed as a simple query.

```
#[derive(Debug, Copy, Clone, PartialEq, Eq, Hash)]
pub enum ParameterType {
    Unknow = 0,
    Bool = 16,
    Int8 = 20,
    Int2 = 21,
    Int4 = 23,
    Text = 25,
    Float8 = 701,
    Varchar = 1043,
    Timestamp = 1114,
    Numric = 1700,
    BPCHAR = 1042,
}
```

Figure 10: Supported value types and their *OID*'s

The first module which is hit by query interceptor, when it tries to process captured query, is the parser. This module takes a query represented as a string and transforms it to the internal *AST*.

There are a lot of ways how this module could be created. One of the most popular decisions is to *PostgreSQL* parser itself by statically linking it to executable. This is what projects like *pg_query* and *rust-postgres* do.

In this project, we followed a different path – we are using a parser generator *rust-peg* to generate a parser from grammar (the grammar is stored inside `proxy/src/parser /grammar.peg`). This enables to have a more general way to parse *SQL* queries (for example, it would be easier to switch to different *SQL* dialect) and, what is more important, easily gives native structures to work with. The exact way a query is represented in *AST* will not be discussed, as it is of little importance.

After parsing queries, *AST* is given to the planer which produces a query execution plan. It is this plan which will be serialized and sent to target *RDBMS* for the execution.

2.1.4 Result validation

Generating and executing plans is a hard task. As such, a possibility to err is very high.

The responsibility for capturing errors is pushed to validation module. This module works similarly to the query interceptor. That is, it is implemented as an automata listener called `HashCalcListener` and is attached to two states:

- When data from a new query passes `ReceivingRows` state, this listener creates a new hasher. Then with all subsequent enters to `ReceivingRows`, this hasher is updated by hash of received tuples (after stripping some metadata);
- Finally, at `CommandCompleted` state, the final value of the hash is output to internal *RDBO* log.

en This procedure is repeated for all the queries that are passed to both target and test *RDBMS*. Later on, we run a separate script `/scripts/parse_proxy_log.py` on *RDBO* log. This script collects all queries and final hashes and checks whether bag of hashes generated by target *RDBMS* matches the hash bag from test *RDBMS*. Any mismatches are reported to the caller [4].

There might be a question why these checks are done off-line and disregard order of queries. The answer to both concerns is quite simple – *RDBO* does not give any guarantees on the execution of the queries inside of test *RDBMS*. In addition to that, both automatas that were used to capture queries may also go out of sync.

Having said all that we can finally talk how the hash calculation is actually performed. A Hasher used in this project is implemented inside of `proxy/src/hash.rs`. As hash calculation will be executed millions of times per benchmark, it was important to use a fast hash function. As such, we are not using conventional hash functions as *SHA256*, but instead use a noncryptographic function due to Jenkins [Jen] (which was shown to be one of the best of its kind [HSZ08][5]).

2.2 Database

As stated in the architecture overview, we have two instances of *PostgreSQL* running on two separate machines (then using both *VM* and *Digital Ocean*). First one is used to execute custom generated plans and the second one for a result set validation. In this section we will define changes made to both of them as well as discuss some of the used helper scripts.

2.2.1 Changes to *PostgreSQL* configuration

If we would check *PostgreSQL* code under file `psql/code/src/include/nodes/plannodes.h` we would see what *PostgreSQL* plan may consist of roughly 40 different plan nodes.

[4]In order to work correctly, it is very important that both *RDBMS*s are of the same version and compiled with same options, otherwise queries like „SHOW ALL“ will be caught to be invalid by validation module

[5]There is actually a newer version of this function called SpookyHash that might be used in future versions of *RDBO*.

Some of these are used for advanced optimization techniques (like bitmap indexes – index which allows going over indexed tuples in heap order) and features (like window aggregates). Due to various factors, these nodes are not used in this project.

In order not to give *PostgreSQL* unfair advantage, these advanced nodes are disabled – see Figure 11.

```
enable_bitmapscan = off
enable_hashjoin = off
enable_indexonlyscan = off
enable_mergejoin = off
enable_tidscan = off
```

Figure 11: Disabled *PostgreSQL* planner capabilities

In addition to these changes, there are also changes in a way *PostgreSQL* logs are written, such as enabling of vacuum and checkpoint messages, changing log format, etc. All changes can be found in `/psql/scripts/postgresql_ex.conf`.

2.2.2 Changes to *PostgreSQL*

One of the primary things needed for this project was the query plan execution. Unfortunately, *PostgreSQL* does not give such facility. This is understandable as a query plan generation is considered *RDBMS* driver's responsibility. For this reason, under this project, *PostgreSQL* was extended with a new command `execplan e_json`[6]. Full implementation of this command can be found in `/psql/scripts/psql.patch`

As an argument, this command takes a string containing query plan in *JSON* format. A query plan given for `execplan` tries to be simplified abstraction of a *PostgreSQL* structure `PlannedStmt`. This abstraction handles such details as relation and its attribute *OID*s resolution, range table handling, junk variable management, additional node creation (for example, aggregation node requires that its actions would contain additional flag attribute showing its origin), function and operation resolution, data type correlation, etc.

The data format of the query plan used by `execplan` is defined in following sections.

2.2.2.1 Query plan node At the top of the query plan sits a plan node. It is a *JSON* object consisting of the three keys:

- `plan_id` – this should hold any string which could be considered to be unique per plan. In general, `plan_id` is *SHA256* hash of query, and is primary used for logging purposes.
- `action_tree` – a plan node which will be used to generate the resulting tuples;

[6]There is also `execplan e_parse`, but its mostly used for internal testing and is not discussed in this paper.

- `output` – array of strings. Each of a them represents attribute name from tuples returned from `action_tree`. These attributes will be returned to the user as a final output.

Action tree by itself consists of two types of nodes: expressions and actions.

2.2.2.2 Expression nodes

Expression nodes are the simplest type of objects and define a certain type of computation. That is, they take some values inside of processed tuples and produce new ones. There are several simple expressions:

- Integers – represent 32 bit signed integer and are used in *JSON* as is;
- Booleans – truth value which can be either `true` or `false`;
- Attribute references – these represent a concrete value of an attribute in processed tuple and are encoded as strings in *JSON*;
- Typed values – *PostgreSQL* has more primitive values than *JSON*. To express those that are not available, we use a special typed value object. This object contains two keys: `typ` – defines the type which we want to get (can be any type supported by *PostgreSQL*); `val` – value encoded as string. One of the most used typed values in plans is `str`, which autocorrelates to the appropriate *RDBMS* `varchar` constant;
- Functions – a final type of an expression that may occur is a function expression. Functions are represented as an object containing two keys: `func` – a name of the function or an operation; `args` – an array of expressions which will be given to the appropriate function as its argument.

In Figure 12 we can see the expression generated for the query `select (1 < 5 and 'abc' like 'a%c')or false;`

```
{ "func": "or",
  "args": [
    { "func": "and",
      "args": [
        {"func": "<", "args": [1, 5]},
        {"func":"like","args":[{"typ":"str","val":"abc"}, {"typ":"str","val":"a%c"}]}]},
  false ]}
```

Figure 12: Expression node

2.2.2.3 Action nodes

While expressions change values inside of tuples, actions work with tuples themselves.

```
{ "action": "seq scan",
  "output": [ "attr1" ],
  "relation": "rel2",
  "filter": { "func": "<",
              "args": [ "attr2", 2 ] } }
```

Figure 13: Node representing sequence scan

2.2.2.3.1 Sequence scan node A Sequence scan node is the primary tuple generator in a query plan. This node goes through a relation in heap order one disk page at a time. While doing so, it extracts all valid tuples from a page (here valid means tuples that belong to the current transaction in terms of *MVCC* and are accepted by the filter). All these tuples are pushed up the tree as input for other nodes.

Simple sequence scan's node is given in Figure 13. This node contains these keys:

- `action` – defines type of an action node. In the sequence scan's case, this field should always be `seq scan`;
- `relation` – a name of the relation of which tuples ought to be returned;
- `filter` – an optional expression on which all returned tuples should agree (expression can access attributes that are defined in `output` of the node below it);
- `output` – an array of attributes of a relation to return. This array can contain a literal attribute name without any restrictions (names can repeat or be given in any order) or named expressions of form `{"alias": "var1", "expr": /* expr */}`.

```
{ "action": "rtn_result",
  "output": [
    { "alias": "pow",
      "expr": { "func": "power",
                "args": [2, 3]}},
    { "alias": "const",
      "expr": 8 }]}
```

Figure 14: Result node

2.2.2.3.2 Result node Result node is another tuple generator. It always generates a single tuple containing values from evaluating all expressions from its `output` key. For example, result node shown in Figure 14 generates a tuple containing two values: a floating point number 8.0 and an integer 8.

Additionally, a result node can contain `outer_action` key. This key represents another action's node whose tuples will be processed by the result node. That is, a result node's output can have expressions containing attribute references to tuples below it. In this case, the result node would produce as many tuples as `outer_action` generates. Such processing is rarely used in practice as there are better ways to apply additional expressions over tuples (like using a subquery node).

2.2.2.3.3 Nested loop join Nested loop join takes two actions: outer and inner. It then takes the first tuple from an outer action and concatenates it to all of the tuples in inner action. After that it does the same with the second, the third and other tuples of the outer action (in order they are produced).

All tuples that do not match filter condition are not returned if and only if `join_type` is set to type `inner`. For `left` join, nested loop join, generates additional tuple for all tuples inside of its outer action that was unmatched by any tuple inside of the inner action.

There are plenty of other join types like `natural`, `natural inner`, `cross`, `left outer`, `natural left`, `natural left outer`, `right`, `right outer`, `natural right`, `natural right outer`, `full`, `full outer`, `natural full outer`. All of these can be simulated by `inner` and `left` joins on the syntax level – and that is done by the *RDBO* parser and the query plan generator.

```
{ "action": "join by nested loop",
  "output": [ "attr1",  "attr2" ],
  "join_type": "left",
  "filter": { "func": "<=",
              "args": [ "attr1", "attr2" ] },
  "outer_action" : { /* action */ },
  "inner_action" : { /* action */ } }
```

Figure 15: Node for nested loop join

A full example of the nested loop join is given in Figure 15.

2.2.2.3.4 Materialization node On the first access of materialization node, it takes all tuples from its `outer_action` and passes them through up the tree. While doing so it also stores a snapshot of these tuples on disk. It is this snapshot which is used for subsequent calls – `outer_action` is not traversed anymore.

It is very beneficial to add a materialization node above a node which slowly generates (subtree at outer action is very large and complicated) just a few tuples and is called multiple times. One common example of such node is anything that is inside of the nested loop join inner's action. This is due to the fact that such node will be iterated as many times as there are tuples in the outer action.

```
{ "action": "materialize",
  "output": [ "attr1" ],
  "outer_action" : { /* action */ } }
```

Figure 16: Node for result materialization

An example of materialization node can be seen in Figure 16. This node contains only three keys: action's name (always `materialize`), child node inside of `outer_action` and a list of variables that will be passed up the tree from node's outer action. This list will be read verbatim. That is, it can not contain any expressions – only the variable references (and, in general, this list matches the output of outer action).

2.2.2.3.5 Limit node Limit node's action ignores first `offset` tuples from its action node. After that, it takes next `count` tuples and passes them up the tree. Finally, if possible, it tries to terminate its action node's work as no other tuples will be needed by its parent node.

```
{ "action": "limit",
  "output": [ "attr1" ],
  "count": 1,
  "offset": 1,
  "outer_action" : { /* action */ } }
```

Figure 17: Node limiting number of output tuples

Node itself is shown in Figure 17. Most of its fields should be self-explanatory, the only thing which should be taken with care is node's `output` field. This field should fully match node action's output.

```
{ "action": "concat",
  "output": [ "attr1" ],
  "actions": [
    { /* action 1 */ },
    { /* action 2 */ },
    { /* ... */ },
    { /* action n */ } ] }
```

Figure 18: Node representing concatenation of inner actions

2.2.2.3.6 Concatenation node In order to accommodate *SQL*'s need for various set operations *PostgreSQL* has two action nodes: concatenation and set operation. A concatenation node which is simpler of the two and can be seen in Figure 18 takes a list of actions and executes them in order thus creating a combined stream of tuples. All actions, including concatenation node itself, have to agree on output attribute names and their types (expression nodes in the concatenation node are disallowed).

This enables to write query plans that handles `union all` and `union` (needs additional actions for unique guarantees) binary operations.

```
{ "action": "setop",
  "output": [ "attr1" ],
  "operation": "intersect",
  "leave_unique": false,
  "op_type": "hash",
  "groups": 5,
  "outer_action": { /* action */ },
  "inner_action": { /* action */ } }
```

Figure 19: Node representing various set operations

2.2.2.3.7 Set operation node Set operation node (Figure 19) is the second action node used for implementing the set operations. It takes a name of an operation which it will be doing (`intersect` or `minus`) and two actions on which this operation will be carried out.

Set operations can be carried out in two ways, determined by `op_type` key:

- `hash` – before applying an operation, all the tuples are put to a predetermined number of bins, by applying some hashing operation on them. After this operation, the actual implementation of intersection and minus becomes trivial;
- `sort` – assuming that both child actions are sorted by all output attributes, set operation node can use modified merge sort to implement intersection and minus.

Additionally, set operation node is capable by itself, without any help from additional nodes, leave only unique tuples. The only thing which is needed by the user is to set a value of `leave_unique` to `true`. This is due to the fact, that both implementations of a set operation carry the uniqueness information by themselves. This enables faster repeated tuple removal compared to the usage of an additional aggregate node.

```
{ "action": "sort",
  "output": [ "attr1" ],
  "by": [ { "col": "attr1", "dir": "asc", "nulls_first": false } ],
  "outer_action" : { /* action */ } }
```

Figure 20: Node representing sorting

2.2.2.3.8 Sort Sort node (Figure 20) sorts tuples from its child action before moving them up to its parent. The sorting can be done in either the memory or the disk – whichever is better is determined by the executor of a plan.

In order to distinguish how to sort, sort node contains a key `by`. Under this key, the user should put a list of attributes he/she wants to sort by. This list can contain either literal attribute names from the output or *JSON* object containing a specification for the sort order. This specification can contain:

- `col` – a mandatory field containing a name of an attribute;
- `dir` – indicates whether to sort in ascending (`asc` – default) or descending (`desc`) order;
- `nulls_first` – determines where `NULL` values should be put in accordance to full output.

2.2.2.3.9 Aggregate node An aggregate node is used to collect the tuples that agree on some values and apply some function over the groups. This node consists of a two additional keys compared to general action node:

- agg_type – indicates a tuple grouping method. This can be plain in order to group nodes by doing a nested loop; hash – grouping is done by applying hash function on specified values and putting tuples with same hash in same buckets; sort – similar to plain way, but short circuits second loop on the first failure (works only with data which is already pre-sorted).
- by – a list of attribute references from the aggregate nodes output. Tuples will have to agree precisely on a value determined by these attributes.

```
{ "action": "aggregate",
  "output": [
    { "alias": "sum",
      "expr": { "func": "sum",
                "args": [ "attr1" ] } },
    "attr2" ],
    "agg_type": "hash",
    "by": [ "attr2" ],
    "filter": true,
    "outer_action" : { /* action */ } }
```

Figure 21: Aggregate node

Functions that work with tuple set are specified as any other function – by including them in node as an expression. For example, an aggregate node shown in Figure 21 uses sum function to sum over var1 attribute values. As we are also grouping by this attribute, the sum will not be different from an attribute's value.

```
{ "action": "sub view",
  "output": [ "attr1" ],
  "name": "sub",
  "filter": { "func": ">",
              "args": [ "attr1", 1 ] },
  "outer_action" : { /* action */ } }
```

Figure 22: Node representing subquery

2.2.2.3.10 Subquery node A simple node (Figure 22) which is used in *PostgreSQL* primary as an optimization blocker (for example, it is not possible to bubble out filter conditions up from this node).

Inside of *RDBO*, this node is used as an intermediate node which gives support to nodes that could not do filtering or are unable to further process its attributes with expression nodes.

In addition to general values, this node also contains a name. This name has no meaning accessible from *JSON* interface and is used primarily for the logging purposes.

2.2.2.3.11 Correlated subquery This node is not really an action node, but rather a connector between the expressions and the nodes. That is, this node can be used in all the same places as any other expression, but, instead of manipulating values, it executes an action node which is saved within it.

This action node should produce exactly one attribute and at most one row of values (actually, it may produce more rows and what is useful for a thing like `in` expressions, but in *RDBO* this feature is not implemented):

- If it returns one value, then it is used as a result of an expression;
- If none – this node is considered to be equal to the `NULL` value.

```
{ "action": "seq scan",
  "output": [ "attr1" ],
  "relation": "rel2",
  "filter": { "func": "=",
              "args": [
                { "exec_sub_plan": {
                  "action": "rtn_result",
                  "output": [ { "alias": "inner",
                                "expr": { "func": "+",
                                "args": [1, { "param": "p" }]}}]},
                  "output": "inner",
                  "args": [ {
                    "set_param": "p",
                    "to_var": "attr1" }]},
                2]}}
```

Figure 23: Node representing a correlated subquery

A full usage of this node is shown in Figure 23. Here we are simulating a sequence scan over relation `test`. All tuples are filtered according to the rule `2 = (select 1 + var1)`.

As it can be seen from the example correlated, a subquery besides being able to execute arbitrary action nodes is also able to give them arbitrary values. These values are put inside of `args` array. This array contains *JSON* objects consisting of two keys:

- `set_param` – this is a name of a parameter which is accessible from anywhere inside of `exec_sub_plan`'s action. It is accessed by using a special `param` expression node;
- `to_var` – variable which is described inside of action node containing correlated subquery. This variable can not be an expression – only an attribute reference – a restriction which can be easily overcome by inserting an additional subquery node.

2.2.2.3.12 Index scan An index scan is very similar to the sequence scan – it takes a relation and outputs some expression applied on its attributes. Unlike sequence scan, it does this by iterating over the index. An index is a special structure which contains a subset of relation's attributes sorted in some order (plus a pointer to the original tuple). As such, search for these attributes is very fast. In order to use this optimized search, index scan's filter has to adhere to the specific set of rules:

- It has to be right recursive;

- Only a simple comparison operator like `=`, `<=`, `>`, . . . can appear in it;
- It can use only the attributes defined in the index. These attributes should appear in the filter in the same order as they are saved in the index, without gaps.

Additionally, index scan takes `scan_dir` key which can be either `forward` or `backward`. This key is used to directly implement `ORDER BY` clause (without a need of a separate sorting node).

```
{ "action": "index scan",
  "output": [ "attr1", "attr2", "attr3" ],
  "relation": "rel1",
  "index": "rel1_attr2_attr3_key",
  "scan_dir": "forward",
  "filter": {"func": "=", "args": [ "attr2", 1 ] }}
```

Figure 24: Node representing index scanning over relation

An example of an index scan can be seen in Figure 24. Here we are reading attributes `attr1`, `attr2`, `attr3` from a relation `rel1` via an index `rel1_attr2_attr3_key`. All tuples returned from this node will be sorted lexicographically by `attr2`, `attr3` tuple.

2.2.3 *RDBMS* structure overview

All code which is relevant *RDBMS* is stored in `/psql/`. Under this directory there are 4 subdirectories;

- `code/` – code of the patched *PostgreSQL*. This directory is created in provisioning phase;
- `logs/` – we put *PostgreSQL* log files under this directory. It is these logs that are later parsed to get all the benchmark results;
- `cache/` – a temporal directory containing various packages downloaded from the internet that were needed by provisioning;
- `scripts/` – a primary directory containing scripts and helper files for creating *RDBMS* instances. These files include:
  - `debug.bash` – a script which is used, as name suggests, to start a *PostgreSQL* instance in the debug mode. That is, it starts `postmaster`, then client which detaches from it and finally the `gdb` debugger. In addition to that, this file is also able to compile *PostgreSQL* from the source, run several static checkers on modified parts of the code base, run *RDBO* specific regression tests;
  - `setup.bash` – a script which is run by the provisioning system. It is this script that downloads the *PostgreSQL* code and all the libraries needed to build it, patches the code, compiles it, creates data a directory, changes and creates the appropriate configuration files, etc.;
  - `postgresql_ex.conf` – changes to vanilla *PostgreSQL* configuration concerning the query plan generation;
  - `psql.patch` – a patch which implements the `execplan` command.

2.3 Client

The primary concern of a client is a creation of a workload for the database. In most cases, the client is a simple script that takes a list of template files with attached weights. This script repeatedly selects template file at random (taking into account its weight), replaces some general parameters with random values and executes its contents.

Obviously, this scheme is not the best as the uniform random values in *SQL* queries are quite rare. Also, most values have some interdependencies that disallow certain permutations. As such, more elaborate template languages were created, such as *RAGS* [SS98] or *QGEN* [PS04].

Still, this system allows defining and creation of complicated queries – finding a good set of templates is still a hard task. This is mainly due to the fact that goodness of a set depends on what we are trying to benchmark. That is why most benchmark suits consist of multiple workloads, modeling different real world scenarios.

A most well-known benchmark suit for the databases is *TPC* [Kleb]. Currently, it contains 10 benchmarks – from ones for testing simple online transaction processing to those for stress testing databases, running on virtual machines.

There are a few other benchmarking solutions like *Dells DVD stores simulation* [Klea], *PostgreSQL* specific `pg_bech`[Pos14b][7] and recently *OLTBench* [DPCCM13]. Out of these, only *OLTBench* gives multiple workloads and, as such, was used as a primary transaction generator for this project.

In this project, we used `seats`, `tpcc`, `twitter`, `voter` and `wikipedia` benchmarks from *OLTBench*. Each of them represents some real-world scenario (for example, `seats` models airline ticketing system), and as such, gives a rough view of *SQL* queries that we might expect in the wild. Exact description of all benchmarks is given in [DPCCM13, p. 280] and workload parameters used in this spece ific project can be found in the configuration files under the `/client/scripts/oltpbench_configs/` directory.

2.3.1 Limitations and changes to *OLTBench*

Though *OLTBench* covered almost all our benchmarking needs, there were a few changes that need to be discussed.

First of all, some queries inside of *OLTBench* contained `LIMIT` clause. `LIMIT` usage in queries is not limited by this project, but it introduces some nondeterminism to the query result if not treated carefully. For example, in `twitter` benchmark there is a query which gives at most 20 tweets from following people. The problem here is that no one defines which tweets to return. Because of this ambiguity, there are multiple query plans that describe same query, but return a different set of results. This gave a lot of trouble to th`proxy`'s data validation mechanism (discussed in 2.1) and needed to be fixed by

[7]To a certain level simulates `TPC-B` workload

introducing a total order into result set with `ORDER BY` clause (Figure 25).

```
-- Before
SELECT f2 FROM "follows" WHERE f1 = ? LIMIT 20
-- After
SELECT f2 FROM "follows" WHERE f1 = ? order by f2 LIMIT 20
```

Figure 25: Change in `getFlights` workload

Secondly, some queries contained row level locking mechanisms like `FOR UPDATE` clause. These mechanisms are needed to enforce the data consistency when one query depends on the results of the previous one. Due to the fact that locking is out of the scope of this project, these clauses were ignored when building a query plan in `proxy`. As such, results of *PostgreSQL* and `proxy` query plans might go out of sync. To mitigate this problem, it was decided to disallow execution of multiple workloads from the same benchmark in parallel.

Finally, some workloads were too complicated to be executed with a naive plan and locked down the whole system. One prominent example of this is stock level query inside of `tpcc` benchmark (Figure 26). Naive plan for this query plan joins relations `ORDER_LINE`, `STOCK` before doing any filtering. Because of that, we get intermediate relation which is too large to process in any reasonable amount of time.

```
SELECT COUNT(DISTINCT (S_I_ID)) AS STOCK_COUNT FROM ORDER_LINE, STOCK
  WHERE OL_W_ID = 2 AND OL_D_ID = 2 AND OL_O_ID < 3007 AND OL_O_ID >= 3007 - 20
        AND S_W_ID = 2 AND S_I_ID = OL_I_ID AND S_QUANTITY < 11;
```

Figure 26: Query occurring stock level from database

One way to solve this problem is to interrupt naive query when a considerably better plan is found. Though this solution seems very promising, due to technical complications it was not implemented in this work. Instead, all workloads containing misbehaving queries were removed. A full listing of removed workloads can be seen in Table 1.

Table 1: Excluded workloads

Benchmark	Workload
seats	FindFlights
	NewReservation
tpcc	StockLevel

2.3.2 Clients structure overview

All client code is stored under `/client`. This directory contains three subdirectories:

- `scripts` – this directory contains the primary scripts for executing client related procedures and will be discussed later in this section;

- `impl` – after provisioning this directory will contain patched instances of *Dells DVD stores simulation* and *OLTBench*;
- `cache` – saves various remote resources improving provisioning speed. This directory will also contain copies of all *OLTBench* created benchmark databases. Not only such local database copy improves test running speed, but is also vital for testing. This is due to the fact that *OLTBench* before its benchmarks pre-fills databases with random data. As we need to have two identical databases, we can not allow *OLTBench* to fill second one randomly. Instead, we use a cached version of the first one.

The most important directory for us is `scripts`. As stated before, scripts under this directory are responsible for creating and running all the tests.

Although there are quite a few scripts in directory, here we briefly discuss just the usage of a few most important ones:

- `setup.bash` – this script is used to initialize a clear machine (whether it is a local *VM* or remote *Digital Ocean* instance). It installs various prerequirements for *OLTBench* and *Dells DVD stores simulation*, like *Java*, *C#*, *PostgreSQL*, ... It also checkouts the appropriate versions of *OLTBench* and *Dells DVD stores simulation*, patches them when appropriate and prepares correct binaries.
- `oltpbenchmark.bash` – a script for interacting to *OLTBench*– running a specified benchmark or initializing database (maybe from cache). This script is quite complicated so there are also a few simpler wrappers: `recreate_default.bash`, `run_default.bash`. These, as name suggests, call `oltpbenchmark.bash` by first adding all needed arguments to recreate or run some benchmark under *VM*.
- `dvdstore.bash` – script similar to `oltpbenchmark.bash`, but used to interact with *Dells DVD stores simulation*.
- `run_all.bash` – runs all prepared benchmarks.

3 Query plan generation

In order to execute parsed and intercepted query, we have to do several things. First of all, we have to generate any plan which would get data defined by the query and execute it on *RDBMS*. Then we can try to generate other plans similar to given one by using simple mutation procedure. After getting a new plan, we have to evaluate it – check if it is better than the original one. Finally, we have to check whether two plans are equivalent by encoding them in *SMT* (introduction to the concept can be found in [DMB11]) and running appropriate solver (in this project we use *Z3* solver). If so we can replace original plan with a better one and try improvement cycle once again.

In later sections, we will discuss all these tasks in greater detail.

3.1 Naive plan

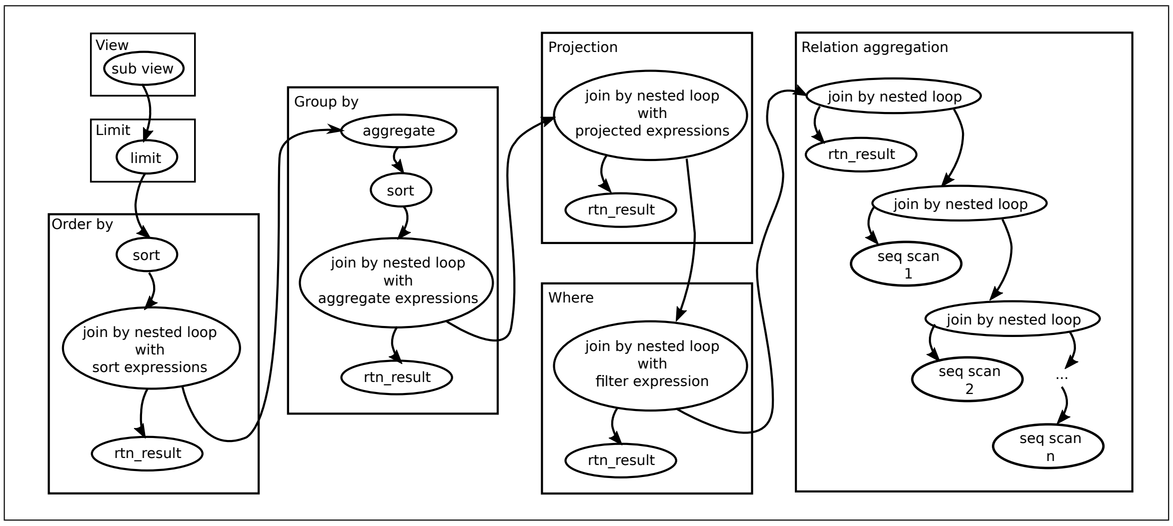


Figure 27: Outline of naive plan generation

In order to start plan optimization, we have to have a starting plan. This plan is generated by procedures inside of *RDBO*'s planning module. These procedures take as an input query's *AST* and produces an equivalent plan. As plan optimization is done in different part of the system *RDBO*'s planner does not try to apply any sophisticated transformations. Instead, it assumes that query is made of 8 different clauses/parts and for each of them directly generates appropriate plan nodes. Explanation of these clauses as seen in Figure 27 follows:

View – this is a root node which sits on top of every naive plan. Its role is to rename internal attribute names into what is needed by the user. We can look at this node as something which implements `SELECT` keyword;

Limit – for limit clause planner generates appropriate `limit` node as is. If query does not contain limit we insert dummy node with `offset` set to be 0;

Order by – ORDER BY clauses are translated into a sort node. Additionally, we insert join by nested loop child node. The job of this node is to evaluate all expression needed by sort node. Such preparation is needed as sort node's output can not contain arbitrary expression – only attribute references. It would seem that such evaluation could have been carried in lower levels, but ORDER BY clauses can have arbitrary expression not only attributes between SELECT and FROM clauses and lower levels do not have access to these sorting expressions;

Group by – before sorting the data set we have to resolve GROUP BY and HAVING clauses. As aggregate node can support arbitrary expressions we do not need to insert additional join by nested loop, but what we have to do is to take into account aggregation type. For plain we do not need to do anything special. For sort we have to insert additional sort node which orders input data set on all attributes inside of GROUP BY.

As grouping by empty set inside of aggregation node means that whole data set will be collapsed into single tuple, aggregation node is the only optional node – we insert it only if query has grouping expressions or if it has HAVING clause;

Projection – here, by using join by nested loop, we encode all expressions that appear between SELECT and FROM keywords;

Where – again by lifting the properties of join by nested loop we encode queries filter condition;

Relation aggregation – finally, at the very bottom of the encoded plan, we put all relations referenced inside of query. We do this by a simple iterative process. Initially, we assume that final subtree will contain only the rtn_result. Then for each joined relation (we assume that relation mentioned after the FROM keyword is implicitly joined) we create a join by nested loop node with current subtree as either left or right child – this join be nested loop will be our new result. While encoding we also try to put expression after ON (or WHERE for top relation) keyword in closest join by nested loop node.

By doing such encoding we end up with a correct, but a slow plan. For example, encoding query SELECT 1 will end up with plan shown in 28. This plan contains all nodes described above, though the optimal plan would have only a single rtn_result node.

3.2 Plan exploration

A naive plan by definition is not a final solution. What we will do in order to get what we would call optimal plan is depicted in Figure 29. Here we are taking a plan which we want to improve from job queue and then for some predetermined time run something

```
{
  "errors": 0,
  "score": 128.00,
  "plan": {
    "plan_id": "rnd-635278556",
    "output": [ "?column?" ],
    "action_tree": {
      "action":"sub view",
      "output": [ { "alias": "?column?", "expr": "?gen-1?" } ],
      "name": "?gen-2?",
      "outer_action": {
        "action": "limit",
        "output": [ "?gen-1?" ],
        "offset": 0,
        "outer_action": {
          "action": "join by nested loop",
          "output": [ { "alias": "?gen-1?", "expr": 1 } ],
          "join_type": "inner",
          "outer_action": {
            "action": "join by nested loop",
            "output": [],
            "join_type": "inner",
            "filter": true,
            "outer_action": { "action": "rtn_result", "output": [] },
            "inner_action": { "action": "rtn_result", "output": [] }},
          "inner_action": {"action": "rtn_result", "output": [] }}}}}}
```

Figure 28: Naive plan for query SELECT 1

```
loop {
    let global_plan = jobs_queue.take();
    while we did not run out of time {
        plan = global_plan;
        while true {
            mutate(rand.choose_node(plan));
            if rand.choose([true, false]) {
                break;
            }
        }

        if score(plan) < score(global_plan) {
            global_plan = plan;
        }
    }
    jobs_queue.push(global_plan);
}
```

Figure 29: Plan improvement cycle

similar to hill climbing algorithm [Luk13, p. 17] – pick a random tree node, apply random mutation on it, if the final result is better than previous then update current global solution. To improve exploration aspects of mutation, with exponentially decaying probability, we apply mutation multiple times. After we finish current improvement cycle, we put the plan back to the job queue. This ensures that even if we run several plan improvers in parallel (in actual experiments we are using 3 improver threads) we will give all plans similar attention time. So to fully understand how plan improver works we only need to explain how we are choosing a random node in a tree and what is done by mutation procedure.

Node selection is actually a simple expansion of linear random selection algorithm used by Unix v7's *fortune* utility. That is, we are running depth first search from root keeping candidate solution. We set our candidate to currently visited node with probability $\frac{1}{d}$, where d is node's position according to depth first search traversal. This ensures that each plan action has an equal probability in being picked.

Definition of mutation is a bit trickier as it consists of multiple smaller mutation sub-procedures executed with a predetermined probability. The probabilities and sub-procedures in order are (P will stand for an independent call probability):

- $P = \frac{1}{10}$ create an new node and put it above current one. Type of this node is chosen at random and can be either `limit` or `sub view`;
- $P = \frac{1}{3}$ take random expression from node's output and push it to child's action (if there are multiple child actions, pick any of them at random);
- $P = \frac{1}{3}$ choose a child at random. Take a random subset of current nodes output expressions and a random subset of child nodes expression. Finally swap them – child will get parents' expressions and parent – child's.
- $P = \frac{1}{3}$ remove current node and put one of its children in its place;
- $P = \frac{1}{10}$ convert node's filter into list of conjuncts, delete one of these conjuncts at random and put back what is left into a new filter;
- $P = \frac{1}{3}$ go through all aliases defined in current node's output and rename some of them (renaming to particular alias happens with probability $\frac{1}{5}$) to any string used in plan;
- $P = \frac{1}{10}$ change expression of a random alias to a 0;
- $P = \frac{1}{5}$ delete random subset of node's output expressions;
- $P = \frac{1}{3}$ convert current node, if it is a sequence scan, into an index scan. Which index to use choose at random from all indexes attached to appropriate relation;

Obviously, such improvement procedure is highly biased towards creating small (in terms of node count) index based plans. However, this is enough to prove that the actual idea is correct and anyone is welcome to use a better exploitation/exploration algorithm.

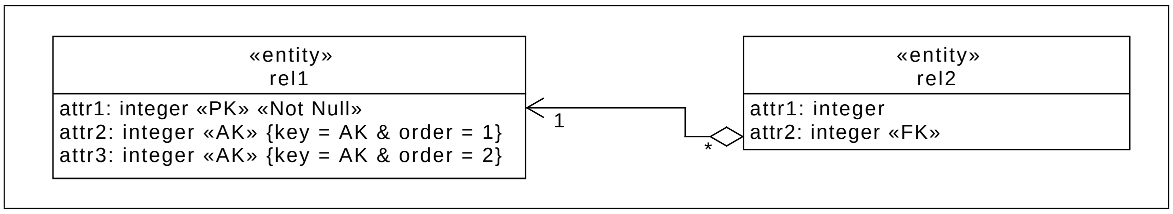
3.3 Query plan equivalency checking

Improvement procedure defined in section 3.2 ensures that generated plans are syntactically valid, but that is not enough. In order to have a plan that could be used instead of naive one we have to check if it is semantically corrected. Here semantic correction means that both plans (generated and naive one) will always return the same, possible ordered, data set for any instance of *RDBMS*.

Even the simpler problem of checking whether there exist *RDBMS* for which given query returns non empty data set is an undecidable problem (it is easy to see that knowing that non linear integer arithmetic is undecidable problem [Gö31]). Even with restricted *SQL* fragments problem of finding appropriate *RDBMS* is computationally hard [DV97]. As such, we have to relax equivalence definition a bit. In this project it means that we compare two plans only against *RDBMS*'s having relations with bounded number of tuples[8].

Equivalency checking itself is done by first encoding schema and plans as theory under *QF_UFNIA* logic. Here *QF_UFNIA* stands for *SMT* logic consisting of union of:

- *QF* – formulas contain boolean variables and arithmetics, but can not contain quantifiers;
- *IA* – formulas can contain arithmetic and comparison operations. That is, we are using theory of integer numbers;
- *N* – integer arithmetic may be non-linear – containing operations like multiplication;
- *UF* – indicates that there might be free sorts and function symbols inside of functions.



«entity»
rel1
attr1: integer «PK» «Not Null»
attr2: integer «AK» {key = AK & order = 1}
attr3: integer «AK» {key = AK & order = 2}
1
*
«entity»
rel2
attr1: integer
attr2: integer «FK»

Figure 30: Schema used for testing generated theories

This kind of encoding combined with efficient solver allows to ask various questions about underlying plan and *RDBMS*. For example, lets take schema shown in Figure 30. This schema contains two tables `rel1` and `rel2`. First relation contains three integer attributes out of which first one is primary key and two others form unique key. Second relation has only two attributes and its second attribute depends on `rel1`'s primary key (there is an inclusion dependency between them).

[8]In this project we are using bound of 3 tuples per relation, but this can be easily changed.

```
select attr1 from rel2 where 1 <= attr2 and attr2 <= 3
```

Figure 31: Simple query for *RDBMS* data generation testing

If we encode this schema and naive plan generated for query shown in Figure 31 in described logic, we can ask for *RDBMS* instance which would contain tuples matched by this query. This is done by additionally adding constrains that `rel1` and relation constructed from query plan should not be empty and executing appropriate test from `./proxy/src/knowledge/convert.rs`. Result of this can be seen in tables 32. In this Figure we see data inside of `rel2` under „Table 1“ (information about actual names is lost while performing various data translations) – first column of this table is an internal id, second corresponds to `attr1` and third to `attr2`. We also get selected data under „Table 16“ (again only the second column matters).

```
########### Table 1 ###########
+---+----+---+
| 0 | 9  | 1 |
+---+----+---+
| 2 | 10 | 1 |
+---+----+---+


########### Table 16 ###########
+---+----+
| 0 | 9  |
+---+----+
| 2 | 10 |
+---+----+
```

Figure 32: Possible „selectable“ data for query

We can also ask a different question – find *RDBMS* which contains data not selected by query. This is achieved by adding constraint that relation representing result of query plan for query should be empty. Running satisfiability proof for such theory gives as tables shown in 33

```
########### Table 0 ###########
+---+----+---+---+
| 0 | 4  | 1 | 2 |
+---+----+---+---+
| 1 | 3  | 5 | 6 |
+---+----+---+---+
| 2 | 45 | 7 | 8 |
+---+----+---+---+

########### Table 16 ###########
++
++
```

Figure 33: Tuples that are not „selected“

This is what various automatic test generation tools do. For example, *QEX* uses *Z3 SMT* solver to find tests by generating formulas using bag theory [VGdHT09]. Its next version [VTdH10] uses similar techniques and theory of algebraic data types. These tools also blend with other tools like *REX* [VTdH10] to support string operations.

RDBO's knowledge module is heavily influenced by ideas from these tools. Though it uses different way of encoding queries (actually *RDBO* works with plans not queries, though distinction for data generation is not really important). *RDBO* also adds additional formulas for `limit` and `order by` clauses and uses *SMT* not only for test generation, but equivalency checking.

In the next sections we will detail conversion from query plan to *SMT*. We will do that by first overviewing schema's encoding. Then we will introduce some basic utility transformation for relations. After that, we will see formulas for simulating most of discussed query plan nodes. Finally, definition for equivalent relations will be given as it is used by *RDBO* procedure `prove_are_equivalent`.

3.3.1 Representing schema

First thing which is encoded is schema. Or to be more precise – its relations.

If we bound schema by relations of size of at most k we can represent each of its relations r_i as k variables $r_{i,j}, 1 \leq j \leq k$. Each of these variables will belong to $\mathbb{T} < \mathbb{B}, \sigma_0, \sigma_1, \ldots, \sigma_{a_{i-1}} >$ (a_i – number of attributes within relation i) tuple sort. First element of this tuple belongs to boolean sort and represents whether tuple exists or not. All other elements represent sorts agreeing with appropriate attribute sort.

Number of variables (which we sometimes may call number of tuples) inside of relation will be represented as $|r_i|$ and call its virtual cardinality. Please note that virtual cardinality is not the same as physical one, as some of tuples may have marked as non existing.

Now we can try to look how `rel1` would like after encoding:

$$r_{0,0} \in \mathbb{T} < \mathbb{B}, ? < \mathbb{Z} >, ? < \mathbb{Z} >, ? < \mathbb{Z} >> \qquad (1)$$

$$r_{0,1} \in \mathbb{T} < \mathbb{B}, ? < \mathbb{Z} >, ? < \mathbb{Z} >, ? < \mathbb{Z} >> \qquad (2)$$

$$r_{0,2} \in \mathbb{T} < \mathbb{B}, ? < \mathbb{Z} >, ? < \mathbb{Z} >, ? < \mathbb{Z} >> \qquad (3)$$

Here $? < \sigma >$ represents extension of σ with special symbol `NULL` which represents undefined value. This special sort also contains two special functions:

- $is - null :? < \sigma > \rightarrow \mathbb{B}$ – given value from sort with `NULL`s returns whether value is `NULL` or not;
- $get - value :? < \sigma > \rightarrow \sigma$ – extracts actual value from nullable sort.

Now observer reader could have notices that first attribute of `rel1` is marked as `NOT NULL`. As such, it should not belong to $? < \mathbb{Z} >$, but to $\mathbb{Z}$. Nevertheless we encoded it as nullable as it helps to keep some consistency in other parts of formulas. We fix this „inconsistency" by adding additional `NOT NULL` constraints into encoding:

$$\bigwedge_{0 \leq j < |r_0|} \neg is - null(r_{0,j}.1) \tag{4}$$

The next thing which we are going to consider for encoding is unique constraints. Inside of `rel1` we have two of them: a primary key and unique constraint *AK*. We can encode *AK* by saying that for tuple j there is no tuple with higher index agreeing on attributes defined by *AK*:

$$\bigwedge_{0 \leq j < |r_0|-1} \left(r_{0,j}.0 \implies \bigwedge_{j < l < |r_0|} \neg r_{0,l}.0 \vee r_{0,l}.2 \neq r_{0,j}.2 \vee r_{0,l}.3 \neq r_{0,j}.3 \right) \tag{5}$$

It is easy to see how this would be changed to other unique constraints – we would simply need to check equality on different set of attributes. In addition to forcing uniqueness, we also create additional relation for each of unique constraints. This relation is exactly the same as the one with defined index, but is sorted (sorting will be explained in section 3.3.3.4) by attributes defined by constraint. This is done, because most of constraints that enforce uniqueness are implemented by sorted structures inside of *RDBMS*. As such, planer can choose going over these structures instead of sorting relation on the fly and safe a few cycles – this translates to *SMT* solver going over sorted relation.

Inclusion dependency between `rel2` and `rel1` is encoded by formula stating that for each non `NULL` attribute `attr2` inside of `rel2` there should be a tuple inside of `rel1` with same value:

$$\bigwedge_{0 \leq j < |r_1|} \left(r_{1,j}.0 \wedge \neg is - null(r_{1,j}.2) \implies \bigvee_{0 \leq l < |r_0|} r_{0,l}.0 \wedge r_{0,l}.1 = r_{1,j}.2 \right) \tag{6}$$

Finally, we add two types of formulas for improving solver's work time. First one asserts that each non existing tuple should contain only `NULL` values as its attribute values – such constraint prunes *Z3*'s search tree. Such formula for general relation r_i can be encoded as:

$$\bigwedge_{0 \leq j < |r_i|} \left(\neg r_{i,j}.0 \implies \bigwedge_{0 \leq l < a_i} is - null(r_{i,j}.l) \right) \tag{7}$$

Second formula called symmetry breaking formula is similar to what is defined in [VTdH10, p. 10]. The idea behind it is to force strict ordering between tuples inside of relations of schema. Such ordering helps solver not to try all $k!$ possible permutations for candidate interpretations and so improves its efficiency.

This formula for general relation r_i is defined below:

$$\begin{aligned}
\bigwedge_{0 \le j < |r_i| - 1} & ite(r_{i,j}.0 \neq r_{i,j+1}.0, r_{i,j}.0 \land \neg r_{i,j+1}.0, && (8)\\
& ite(r_{i,j}.1 \neq r_{i,j+1}.1, r_{i,j}.1 < r_{i,j+1}.1, && (9)\\
& ite(r_{i,j}.2 \neq r_{i,j+1}.2, r_{i,j}.2 < r_{i,j+1}.2, && (10)\\
& \ldots && (11)\\
& ite(r_{i,j}.(a_i - 1) \neq r_{i,j+1}.(a_i - 1), && (12)\\
& r_{i,j}.(a_i - 1) < r_{i,j+1}.(a_i - 1), && (13)\\
& true) \ldots))) && (14)
\end{aligned}$$

Here we use a built-in function ite. This function takes three parameters: a condition, formula which will be evaluated if given condition evaluates to `true` and formula for `false` condition.

Symmetry breaking formula works by adding constraint between two consequent tuples:

- If they differ in existence, then existing tuple should go first;
- If not, it checks whether first attributes of these tuples are different and if so it asserts that tuple with smaller attribute comes first;
- If tuples agreed on everything so far it checks second attribute, if they are equal third one and so on;
- If tuples agreed on all attributes then we assert that everything is correct, as relation can contain two identical tuples.

3.3.2 Node encoding helpers

After a talk about schema's encoding, we continue by presenting three utility transformations: *filter*, *projection*, *unique*. First two are used by node encoders and help them to encode part of node – *projection* is used for encoding node's `output` key and *filter* is used for `filter` encoding. Last one is used as intermediate step for encoders like aggregation node encoder.

In next sections we will describe these operations in greater details.

3.3.2.1 Projection As mentioned before *projection* operation is primarily used to encode any node's `output` key. *Projection* as its arguments takes a relation r_i and a list of functions $\phi_0, \phi_1, \ldots, \phi_{a'_i - 1}$ (each of these functions maps single tuple from r_i to attribute value and has signature $\phi_j : \mathbb{T} < \mathbb{B}, ? < \mathbb{Z} >_0, ? < \mathbb{Z} >_1, \ldots, ? < \mathbb{Z} >_{a_i - 1} > \rightarrow ? < \mathbb{Z} >)$ and produces a new relation r'_i. This relations will have a'_i attributes and $|r_i|$ tuples. It is generated by encoding *projection* as *SMT* formula. In this formula we first loop over all existing tuples of r_i (formula 15). For each of these tuples r'_i will have a tuple produced by applying all of *projection* functions on appropriate r_i tuple (formula 16).

$$\bigwedge_{0 \leq j < |r_i|} ite(r_{i,j}.0, \tag{15}$$

$$r'_{i,j}.0 \wedge \bigwedge_{0 \leq l < a'_i} r'_{i,j}.(l+1) = \phi_l(r_{i,j}), \tag{16}$$

$$\neg r'_{i,j}.0) \tag{17}$$

Projection is also responsible for creation of ϕ function. As such, to finish discussion of *projection* we also have to look how it does that. The encoding of any expression inside of `output` key into *SMT* formula is actually trivial. Essentially we have two objects that we have to consider: functions and attribute references.

Functions are encoded into formulas by a direct transformation. That is, `{"func": "+", "args": [1, 2]}`, becomes formula $1 + 2$ (though inside of *SMT* this formula would be written by using *S-expressions*). The only thing that we have to remember is that most of operations in *SMT* are nullable – if at least one of their arguments evaluates to `NULL` (note that argument has to be evaluated to be considered, that is, `true or null` will produce `true` as right side of `or` will not be evaluated) then whole operation should also produce `NULL`. For that we have to overload most of builtin *SMT* functions ourself. Currently inside of *SMT* we define and support such nullable operations with usual semantics: $? <$, $? <=$, $? >$, $? >=$, $? =$, $?not$, $?and$, $?or$, $?+$, $?-$, $?*$, $?div$, $?mod$, $? <>$, $?! =$, $?is - null$.

Encoding of attribute reference is also carried by a simple transformation. References are transformed into a list of relation's tuple accessors. For example, accessing attribute `attr1` of `rel1` could be transformed to a list $r_{0,0}.1, r_{0,1}.1, r_{0,2}.1$.

```
prove_are_equivalent(
    r#"select 1"#,
    r#"select (5 + 5) / 10"#);
```

Figure 34: Proving a simple math identity

With this we finish review of *projection* operation. Even having only this simple tool we already can start proving some equalities between queries. For example, we can check whether arithmetic formula $(5+5)/10$ can be reduced into 1. We can prove this by writing down formulas into singleton *SQL* queries and giving them to `prove_are_equivalent` function – see Figure 34. It is this function which is responsible for transforming queries into query execution plans, execution plans into *SMT* formulas and doing actual equivalence check.

3.3.2.2 Filter Similarly to *projection filter* takes relation r_i as its argument. As second argument *filter* takes not a list of functions producing attribute values, but single function ϕ. This function is constructed from nodes `filter` (in the same as any other expression – see section 3.3.2.1) key and is in charge of mapping single tuple from r_i to nullable boolean value (its signature is $\phi : \mathbb{T} < \mathbb{B}, ? < \mathbb{Z} >_0, ? < \mathbb{Z} >_1, \ldots, ? < \mathbb{Z} >_{a_i - 1} > \rightarrow ? < \mathbb{B} >$). By having these two arguments *filter* produces a new relation r'_i containing all tuples from r_i on which ϕ returned truthful value.

Filter does its work by iterating over all existing tuples of relation r_i (formula 18). For each tuple $r_{i,j}$ *filter* applies function ϕ. If it produces non `NULL` value which is equal to `true` (formula 19) it can copy tuple from r_i to r'_i (formula 20). If ϕ produced `NULL` or `false` value, we assume that appropriate tuple inside of r'_i does not exist (formula 21).

$$\bigwedge_{0 \leq j < |r_i|} ite(r_{i,j}.0 \tag{18}$$

$$\wedge\, ite(is - null(\phi(r_{i,j})), false, get - value(\phi(i_{n,j}))), \tag{19}$$

$$r'_{i,j} = r_{i,j}, \tag{20}$$

$$\neg r'_{i,j}.0) \tag{21}$$

```
prove_are_equivalent(
    r#"select 1 where false"#,
    r#"select 8 where false"#);
```

Figure 35: Queries returning no tuples are equivalent

As with *projection* we can prove a few simply facts by using *filter* and schema's encodings in *SMT*. The simplest thing that we could check, is equivalency of queries `select 1 where false` and `select 8 where false`. They are equivalent due to fact that it does not matter what kind of data could be returned by *SQL* query if all of it is filtered out. This proof can be executed inside of *RDBO* by using function call show in Figure 35.

```
prove_are_equivalent(
    r#"select attr1 from rel1 where attr1 is null"#,
    r#"select 1 where false"#);
```

Figure 36: Scanning over relation and filtering all tuples

Similarly we can check that some expressions are always evaluated to `false`. One such expression is a checking if attribute `attr1` of relation `rel1` is `NULL`. It will always produce `false` as this attribute is marked as `NOT NULL`. Execution of this proof is shown if Figure 36.

3.3.2.3 Unique The final utility transformation which we will discuss is ***unique***. This operation takes relation r_i and attribute set s. It produces a new relation r'_i which contains tuples from r'_i. The only difference between two relations is that r'_i will not include more than one tuple agreeing on set s. Such relation can be created by iterating all tuples of r_i (formula 22). If we can not find tuple with larger index and agreeing on attributes within set s (formula 23), we can insert it to relation (formula 24).

$$\bigwedge_{0 \leq j < |r_i|} ite(r_{i,j}.0 \tag{22}$$

$$\wedge \bigwedge_{j < l < |r_i|} \left(\neg r_{i,l}.0 \vee \bigvee_{t \in s} r_{i,j}.t \neq r_{i,l}.t \right), \tag{23}$$

$$r'_{i,j} = r_{i,j}, \tag{24}$$

$$\neg r'_{i,j}.0) \tag{25}$$

3.3.3 Node encoding

Finally, by having encoded schema and knowing how to encode some simple transformations we can actually look in how to encode plan nodes at full. Representation of all encodings used for plan nodes are presented below.

3.3.3.1 Sequence and index scan nodes Sequence and index scan nodes are one of the simplest to encode. Encoding of both these nodes start by first applying *filter* operation on appropriate relation (or index). And is finished by taking *projection*.

For index we do not check whether filtering is actually done by using appropriate attributes. Node encoder's caller is responsible for such check.

3.3.3.2 Nested join Nested join node takes two relations r_n and r_m and produces relation r'_i. This new relation will contain a tuple for all pairs of r_n and r_m tuples.

Simulating nested join for relations r_n and r_m is actually a hard task. The problem arises when we have to consider outer joins. These joins produce additional tuples (with `NULL` values for either r_n or r_m attributes) if filter condition is not matched. As such, we have to apply *filter* operation before knowing which additional tuples needs adding. However, in order to know whether we actually need these tuples we also have to run *filter* on them – this somewhat complicates formulas.

So instead of running *filter* multiple times we create r'_i in three simple steps. In the first step we create relation r_o representing full outer join of relations r_n and r_m. This is done by iterating by all pairs of tuples from these relations (formula 26). If both tuples from such pair represents existing tuples (formula 26), r_o will contain additional tuple (formula 27) composed of their concatenation (formulas 28 and 29):

$$\bigwedge_{0\leq j<|r_n|, 0\leq l<|r_m|} ite(r_{n,j}.0 \wedge r_{m,l}.0, \tag{26}$$

$$r_{o,j\cdot l}.0 \tag{27}$$

$$\wedge \bigwedge_{0\leq t<a_n} \left(r_{o,j\cdot l}.(t+1) = r_{n,j}.(t+1)\right) \tag{28}$$

$$\wedge \bigwedge_{0\leq t<a_m} \left(r_{o,j\cdot l}.(a_n+t+1) = r_{m,l}.(t+1)\right), \tag{29}$$

$$\neg r_{o,j\cdot l}.0) \tag{30}$$

We also consider existing tuples from r_n separately (formula 31). For each of these tuples we add tuple to r_o (formula 32) which consist of attributes taken from relation r_n (formula 33). r_m part is filled with `NULL` values (formula 34):

$$\bigwedge_{0\leq j<|r_n|} ite(r_{n,j}.0, \tag{31}$$

$$r_{o,|r_n|\cdot|r_m|+j}.0 \tag{32}$$

$$\wedge \bigwedge_{0\leq t<a_n} \left(r_{o,|r_n|\cdot|r_m|+j}.(t+1) = r_{n,j}.(t+1)\right) \tag{33}$$

$$\wedge \bigwedge_{0\leq t<a_m} \left(r_{o,|r_n|\cdot|r_m|+j}.(a_n+t+1) = null\right), \tag{34}$$

$$\neg r_{o,|r_n|\cdot|r_m|+j}.0) \tag{35}$$

In similar fashion we also append tuples only from r_m (formula is symmetric to formula used for r_n relation and is not shown here). After all this, r_o will have cardinality of $|r_n| \cdot |r_m| + |r_n| + |r_m|$.

Second step of join simulation is *filter* operation. This operation is applied as is on relation r_o and produces relation r_f.

On the third step we create final relation r'_i (disregarding obligatory *projection*). This relation will consist of first $|r_n| \cdot |r_m|$ tuples from r_f. The next $|r_n|$ tuples will be copied from unfiltered r_o relation (formula 37). This happens if and only if appropriate tuple from $|r_n|$ exists and all instances of this tuple were filtered inside of cross join part of r_f (formula 36):

$$\bigwedge_{0 \leq j < |r_n|} ite(r_{n,j}.0 \wedge \bigwedge_{j \cdot |r_m| \leq l < (j+1) \cdot |r_m|} \neg r_{f,l}.0, \tag{36}$$

$$r'_{i,|r_n| \cdot |r_m| + j} = r_{o,|r_n| \cdot |r_m| + j} \tag{37}$$

$$\neg r'_{i,|r_n| \cdot |r_m| + j}.0) \tag{38}$$

After applying symmetric assertion for r_m tuples, we finalize creation of r'_i.

```
prove_are_equivalent(
    r#"select rel1.attr1 as a, rel1.attr2 as b, rel1.attr3 as c,
              rel2.attr1 as d, rel2.attr2 as e
        from rel1
        left join rel2 on
          rel1.attr2 = rel2.attr2"#,
    r#"select rel1.attr1 as a, rel1.attr2 as b, rel1.attr3 as c,
              rel2.attr1 as d, rel2.attr2 as e
        from rel2
        right join rel1 on
          rel1.attr2 = rel2.attr2"#);
```

Figure 37: `left` and `right` joins differ only in relation order

With nested loop join node implemented we can check that `right join` is not really necessary. This is because it can be implemented only from `left join`. In order to do that we only have to swap order of its arguments. Prove of such assertion in *RDBO* can be seen in Figure 37.

3.3.3.3 Aggregate node

Within aggregate we will begin with relation r_i and will want to get relation r'_i. Inside of this relation all tuples agreeing on attribute set s should be collapsed into single tuple. This collapsing is defined by aggregate functions used inside of `filter` and `output` defined in aggregation node.

In order to give more information to *SMT* solver we encode these aggregate functions not by using theory of undefined functions, but directly writing down their semantics. As such, we support only a small fraction of possible aggregate functions:

- `count` – counts number of tuples within agreed group if its argument is a star symbol. If its argument is an expression, then it evaluates this expression on all agreeing tuples and counts how many of them gave non `NULL` result;
- `sum` – single argument function which indicates that collapsed tuple will contain sum of attributes that appeared inside of agreeing tuples;
- `min` – will leave only attribute with minimum value;
- `max` – will leave only attribute with maximum value.

These functions are actually done not by separate transformation, but as part of *filter* and/or *projection* transformation (section 3.3.2.2). For example, `count(*)` for relation

r_i on attribute set s can be implemented as *projection*. In this *projection* we will go over all existing rows of r_i (formula 39). For each of them we create a new tuple inside of r'_i (formula 40). This tuple will contain only one attribute (if *projection* would have more expressions, they would appear as additional attributes as usual). In order to get value of this attribute we have to do an additional scan of r_i. By doing so we would need to increment attribute in question by one for each found tuple agreeing on set s (formula 41).

$$\bigwedge_{0 \leq j < |r_i|} ite(r_{i,j}.0, \tag{39}$$

$$r'_{i,j}.0 \tag{40}$$

$$\wedge\, r'_{i,j}.1 = \sum_{0 \leq l < |r_i|} ite(r_{i,l}.0 \wedge \bigwedge_{t \in s} r_{i,j} = r_{i,l}.t, 1, 0), \tag{41}$$

$$\neg r'_{i,j}.0) \tag{42}$$

Obviously, after running this projection modification in such a way, we will be left with a lot of repeated tuples. These tuples can be filtered out by running *unique* operation on relation r'_i with attribute set s.

We can implement and other supported aggregate functions in similar fashion as shown above. As such, their definitions are not included here.

To finalize aggregation node, after applying *projection*, we also have to run *filter*. As stated before *filter* may also contain aggregate functions, but their calculation is exactly the same as in projection. The only difference is that we no longer run additional *unique* operation.

```
prove_are_equivalent(
    r#"select sum(attr2 + 1) from rel2"#,
    r#"select sum(attr2) + count(*) from rel2"#);
```

Figure 38: Associativity of aggregate functions

Having implemented aggregate node we can check a few equivalences. The easiest thing to see is that most of defined aggregate functions are associative. That is, we can extract part of expression inside of aggregate function and put it inside into other aggregate function. For example, inside of Figure 38 we can see that `sum(attr2 + 1)` can be safely replaced by `sum(attr2)+ count(*)`. That is, we extracted expression `+ 1` and replaced it by `count(*)` (which is a more direct way of saying `sum(1)`).

Another simple fact that can be checked is that `having` clause is not really necessary. It can always be replaced by simple `where` clause – if we push current *SQL* query into subquery (Figure 39).

Aggregation is also one of the procedures which can trick the prover. For example,

```
prove_are_equivalent(
    r#"select sum(attr2) from rel2 group by attr1 having max(attr2 + 1) = 5"#,
    r#"select a.s from
         (select sum(attr2) as s, max(attr2) as m from rel2 group by attr1) as a
       where a.m = 4"#);
```

Figure 39: Having clause is just syntactic sugar for temporal table

```
prove_are_equivalent(
    r#"select attr1 from rel1 having count(*) > 1000"#,
    r#"select 1 where false"#);
```

Figure 40: Proving equivalency for non equivalent plans

queries shown in Figure 40 are clearly non equivalent, but *RDBO* considers them as equivalent. This is due to the fact that all equivalency checks are done on a bounded model. As such, queries that have to work with more tuples will always return an empty result set.

3.3.3.4 Sort Sort node takes an input relation r_i and creates relation r'_i. This new relation will contain exactly the same tuples as r_i, but in the different order. This order is defined by attributes defined under by key inside of sort node. We do not really care about these attributes as long as we can compare two tuples. This comparison is created in exactly the same way as any other expression and is not discussed here. Here we are going to look only into sorting procedure itself.

As with all the other operations we need to implement sorting without using any loops. Simplest way to do that is to create a new relation r'_i. Then we could assert that physical cardinalities of both original r_i and created one r'_i are the same. Furthermore we should say that each of original tuples belongs to created relation. Finally, applying symmetry breaking formula on appropriate attributes of r'_i would force it to be sorted (there would be problems with repeated tuples as there is no way for solver only from this information to Figure out which ones has to be repeated and which not).

The problem with this approach is that it would take quite some time to find appropriate assignment. As such, it is more beneficial to implement sorting by taking any sorting algorithm and unrolling its loops.

Such idea is pretty similar to concept called sorting networks [Bat68]. For example, by introducing 8 temporal variables $s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}$ we can sort relation having 3 tuples with only 6 comparison operation.

We can achieve this by assigning s_4 minimal and s_5 maximum of $r_{i,0}$ and $r_{i,1}$ tuples by some comparison operator. Then s_6 gets minimal and s_7 maximum of s_5 and $r_{i,2}$. And so on – we repeat this process as defined by sorting network shown in Figure 41 and at the end get tuples in sorted order.

This is what was done in *RDBO*. In addition to that we add additional metadata variable to all relations. This variable, lets call it S_i, indicates whether relation is sorted

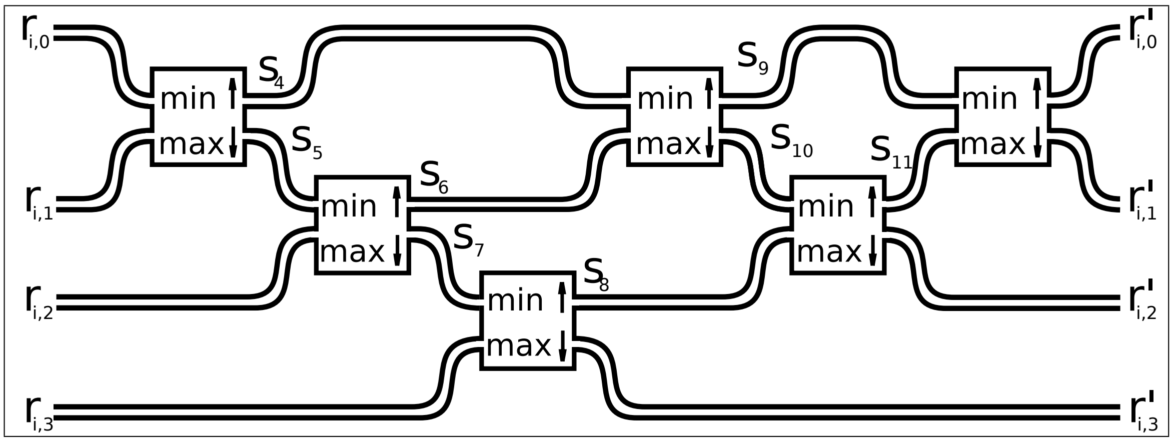


Figure 41: Sorting network implementing bubble sort for relation having 3 tuples

in user meaningful manner. Later on, when checking relational equivalency, S_i is used to chose appropriate comparison procedure between two relations.

This variable can take three different values:

- `NOT_SORTED` – indicates that relation is not sorted in any user meaningful way;
- `SORTED` – order of tuples inside of relation matters;
- `CAN_BE_SORTED` – relation is sorted by an intermediate operation, but user may not care about that.

In order to understand why two states are not sufficient we can check query plans show in Figure 42. Here we prove that going over sorted sequence scan – user expects to get sorted tuples – is same as index scan. Similarly we could prove that going over index is same as going over simple scan. In this case user does not care that index scan provides sorted tuples, he/she only cares that set of returned tuples would be the same. For both proves to be correct (and for sorted scan to be not equivalent to sequence scan) at the same time we have to have third state.

```
prove_actions_are_equivalent(
    r#"
        { "action": "index scan",
          "output": [ "attr1" ],
          "relation": "rel1",
          "index": "prim1",
          "filter": { "func": "<", "args": [ "attr1", 10 ] } }
    "#,
    r#"
        { "action": "sort",
          "output": [ "attr1" ],
          "by": [ { "col": "attr1" } ],
          "outer_action": { "action": "seq scan",
                            "output": [ "attr1" ],
                            "relation": "rel1",
                            "filter": { "func": "<", "args": [ "attr1", 10 ] } } }
    "#);
```

Figure 42: There is no need to sort by an indexed column

S_i for various elements is defined like this:

- For all relations generated from scheme we assign `NOT_SORTED` to this variable. Though these relations are sorted by symmetry breaking formula this is done only for internal use and user may not get tuples in such order while executing appropriate plan;
- For indexes generated from scheme we assign `CAN_BE_SORTED`. This is due to fact that most indexes are implemented by using ordered structures and scanning them produces tuples in sorted order;
- For relations generated by applying sorting procedure we set this variable to `SORTED` ;
- Relation created by joining two other relations inherits its S_i from left relation.

  There is an exception for this – if this relation has virtual cardinality of 1 we inherit S_i from right relation.

  We do this because joining is done by doing nested loop which preserves order of its left side tuples;
- Aggregation erases sorting information, that is it gets S_i value of `NOT_SORTED`, unless we are using `"type" = "sort"`. In such case we assign `CAN_BE_SORTED`.
- All other relations retrieved by doing operation like *filter*, *projection*, etc. inherits their S_i value from original relations.

```
prove_are_equivalent(
    r#"
        select r1.attr1 from rel2 as r1
        cross join rel2 as r2
        order by r1.attr1
    "#,
    r#"
        select r1.attr1 from
            (select attr1 from rel2 order by attr1) as r1
        cross join rel2 as r2
    "#);
```

Figure 43: Non equivalent queries may produces equivalent plans

Interestingly, introduction of sorting information may prove equivalency for some queries that are not equivalent. For example, lets check queries shown in Figure 43. We know that first query will produce tuples in `r1.attr1` order, but order of tuples retrieved by second query is undefined. This is due to the fact that *SQL* does not talk about order of tuples after using join clause (this is done intensionally to allow wider array of algorithms implementing this operation). As such, these queries should be considered different. However, knowing what kind of join implementation we are using under the hood – nested join – we can imply that plan generated from second query will also return ordered set of tuples – proving equivalency of plans.

Sort may also introduce interesting results when interacting with other encoded parts. For example, we may prove that order of `NULL` values does not matter – Figure 44. However, that happens only if appropriate attribute is set to be `NOT NULL`.

```
prove_are_not_equivalent(
    r#"select attr2 from rel1 order by attr2 nulls first"#,
    r#"select attr2 from rel1 order by attr2 nulls last"#);

prove_are_equivalent(
    r#"select attr1 from rel1 order by attr1 nulls first"#,
    r#"select attr1 from rel1 order by attr1 nulls last"#);
```

Figure 44: `NULL` order matters unless we are ordering by column with `NOT NULL` constraint

3.3.3.5 Limit node Within limit node we have relation r_i and want to generate relation r'_i containing only first b tuples (`count` part of limit node) starting from tuple numbered a (`offset` part). Problem arises when we realize that not all tuples inside of r_i represents physical tuples. That is, some of them do not exist and should not be included into final count.

This problem is solved by first creating relation r_n containing $|r_i|+1$ tuples and single attribute. For the first tuple we set this attribute to -1 (43 formula). All other tuples (44 formula) of this relation will get this attribute to be set to increment of previous value (formula 46) if appropriate tuple from r_i exists (formula 45). If tuple does not exists this attribute will be copied untouched (formula 47).

$$(r_{n,0}.0 \land r_{n,0}.1 = -1) \tag{43}$$

$$\land \bigwedge_{1 \le j \le |r_i|} r_{n,j}.0 \tag{44}$$

$$\land\, ite(r_{i,j-1}.0, \tag{45}$$

$$r_{n,j} = r_{n,j-1} + 1, \tag{46}$$

$$r_{n,j} = r_{n,j-1}) \tag{47}$$

After all this, values saved in r_n will correspond to physical tuple number of relation r_i. This enables us to implement limit operation by just removing tuples from r_i that are not within interval $[a; a+b)$. Final definition of r'_i is shown below:

$$\bigwedge_{0 \le j < |r_i|} ite(r_{i,j}.0 \land a \le r_{n,j+1} \land r_{n,j+1} < a + b, \tag{48}$$

$$r'_{i,j} = r_{i,j}, \tag{49}$$

$$\neg r'_{i,j}.0 = r_{i,j}) \tag{50}$$

Limit is particularly useful node in terms of optimization. This is true because limit stops tuple processing as fast as possible – when it reads tuple with number $a+b-1$. As such, it is always beneficial to have limit node near nodes producing a lot of tuples. In addition to that we may sometimes want to copy limit multiple time. First copy would be used for correctness and other copies for performance reason.

```
prove_are_equivalent(
    r#"select attr1 from (
            select attr1 from rel2
            union all
            select attr1 from rel2 where attr2 > 3
        ) as a
        limit 1
    "#,
    r#"select attr1 from (
            (select attr1 from rel2 where true limit 1)
            union all
            (select attr1 from rel2 where attr2 > 3 limit 1)
        ) as a
        limit 1
    "#);
```

Figure 45: Pushing limit inside of union

One such optimization is shown in Figure 45. Here we have an union of two queries after which we leave only single tuple. Obviously, we do not need to calculate whole union to get only single tuple – it is enough to take only single element from both queries.

3.3.3.6 Set operation and concatenation nodes Starting with two relations r_n and r_m we wish to apply some set operation and get relation r'_i. We have to support three set operations union, intersection and difference[9]. Implementing the first one is pretty trivial – we create a relation of size $|r_n| + |r_m|$. First $|r_n|$ tuples will come from relation r_n and next ones from r_m:

$$\bigwedge_{0 \leq j < |r_n| + |r_m|} ite(j < |r_n|, r'_{i,j} = r_{n,j}, r'_{i,j} = r_{m,j-|r_n|}) \tag{51}$$

Next operation that we are going to consider is intersection. This operation should produce relation r'_i which would contain only those tuples that are in both r_n and r_m. Simulation of intersection is a bit more difficult compared to union. The difficulty comes from fact that for each new tuple we consider adding from r_n to r'_i we have to know whether it has corresponding unused entry inside of r_m. In order to follow used tuples we create $|r_n|$ additional relations $r_{m+0}, r_{m+1}, r_{m+2}, \ldots, r_{m+|r_n|-1}$. First relation is equal to original r_m, that is $r_{m+0} = r_m$. Other relations on line will be exactly the same as previous ones if no tuple was added to r'_i at appropriate time frame or will be smaller by one tuple (the one which was added to r_i).

Having said that we can actually see formula generating intersections. We start by iterating over tuples of r_n (formula 52). Having concrete tuple $r_{n,j}$ we try to see if we

[9]Here we are considering these operation as they would be applied to multisets – repeated elements are preserved

will need to add it to r'_i. We first check if this tuple corresponds to physical one and is equal to first tuple of r_{m+j+0} (formula 53). If so, we move it to relation r'_i (formula 54) and create next temporal relation r_{m+j+1}. This relation will contain all, but first tuples of r_{m+j+0} (formula 55).

If we do not match $r_{n,j}$ with first tuple of r_{m+j+0} we try matching with second tuple (formula 56). If again we do not find equal tuples we move to third tuple, fourth tuple and so on until we hit tuple numbered $|r_m| - 1$ (formula 60). If we fail to match this tuple we conclude that tuple $r_{n,j}$ should not go to r'_i (formula 63) and that next temporal relation r_{m+j+1} will be exactly the same as current one (formula 64).

$$\bigwedge_{0 \le j < |r_n|} \tag{52}$$

$$ite(r_{n,j}.0 \wedge r_{n,j} = r_{m+j+0,0}, \tag{53}$$

$$r'_{i,j} = r_{n,j} \tag{54}$$

$$\wedge \bigwedge_{0 \le l < |r_m|} ite(l = 0, \neg r_{m+j+1.l}.0, r_{m+j+1,l} = r_{m+j,l}), \tag{55}$$

$$ite(r_{n,j}.0 \wedge r_{n,j} = r_{m+j,1}, \tag{56}$$

$$r'_{i,j} = r_{n,j} \tag{57}$$

$$\wedge \bigwedge_{0 \le l < |r_m|} ite(l = 1, \neg r_{m+j+1.l}.0, r_{m+j+1,l} = r_{m+j,l}), \tag{58}$$

$$\ldots \tag{59}$$

$$ite(r_{n,j}.0 \wedge r_{n,j} = r_{m+j,|r_m|-1}, \tag{60}$$

$$r'_{i,j} = r_{n,j} \tag{61}$$

$$\wedge \bigwedge_{0 \le l < |r_m|} ite(l = |r_m| - 1, \neg r_{m+j+1.l}.0, r_{m+j+1,l} = r_{m+j,l}), \tag{62}$$

$$\neg r'_{i,j}.0 \tag{63}$$

$$\wedge\, r_{m+j+1} = r_{m+j}) \tag{64}$$

Final operation – set difference – is very similar to intersection. This operation defines new relation r'_i which will contain all tuples from r_n that are not inside of r_m. Such operation can be implemented by changing a few lines of intersection formula:

- If we do not find a matching tuple inside of r_{m+j+0} we insert it to r'_i. That is, formula 63 is changed to $r'_{i,j} = r_{n,j}$;
- If we find the match we zero out corresponding tuple of r'_i. So formulas 54, 57, . . ., 61 become $\neg r'_{i,j}.0$

Having implemented set operations we check simple fact – if we start from relation

```
prove_are_equivalent(
    r#"(select attr2 from rel2)
       union all
       (select attr1 from rel2)
       except all
       (select attr1 from rel2)
    "#,
    r#"select attr2 from rel2"#);
```

Figure 46: We can add and remove tuples without changing original data set

`rel2` and add to it some tuples, then removing same tuples will produce unchanged relation `rel2`. Equivalency checking of such behaviour is shown in Figure 46.

```
prove_are_not_equivalent(
    r#"((select attr2 from rel2)
       union all
       (select attr1 from rel2))
       except
       (select attr1 from rel2)
    "#,
    r#"select attr2 from rel2"#);
```

Figure 47: Uniqueness operation is destructive

It is also possible to see that it is very important not to mix repeated tuple count preserving set operations and the ones that produce only unique tuples while proving such fact. An example of such mix is shown in Figure 47. Two shown queries are definitely not equivalent. To see that we can take relation `rel2` containing repeated value for attribute `attr2`. After doing `except`, these repeated values will collapse into single one. As such, final result will have one tuple less compared to original relation `rel2`.

3.3.4 Checking for relation equivalence

To finalize description of query plan encoding we introduce the way we are checking if two encoded relations are equivalent. Two relations r_n and r_m are equivalent if *SMT* solver can not find *RDBMS* interpretation in which these relations return different result set. That is, we extend our formula set with formulas asserting that these two relations are not equal. If solver infers that such extended formula set is unsatisfiable then we can assume equivalency.

There are a few simple checks that can be used to check if these relations are not equal:

- $a_n \neq a_m$ – if relations differ in number of attributes they definitely can not be equal;
- $\sum_{0 \leq j < |r_n|} ite(r_{n,j}.0, 1, 0) \neq \sum_{0 \leq j < |r_m|} ite(r_{m,j}.0, 1, 0)$ – difference in physical cardinality is also a sufficient proof;
- $(S_n = SORTED \wedge S_m = NOT\_SORTED) \vee (S_n = NOT\_SORTED \wedge S_m = SORTED)$ – relations should also agree on whether they are sorted or not.

If at least one of these properties is true we are finished. If not, check procedure splits in two cases:

- In first case both relations are not sorted – $S_n = NOT\_SORTED \vee S_m = NOT\_SORTED \vee (S_n = S_m \wedge S_n = CAN\_BE\_SORTED)$.

  As such, we just have to check whether where exists tuples from r_n which does not belong to r_m or vice verse. That is, we are trying to check if r_n is not subset of r_m or r_m is not subset of r_n. The needed check is similar to inclusion dependency checking which was shown in formula 6 and is not repeated here.
- We consider second case if both relations are sorted – $S_n = SORTED \vee S_m = SORTED$.

  Here we are interested not only in existence of tuples, but also in their order. To simplify comparison process we first sort both relations by tuple existence by applying sorting procedure defined in section 3.3.3.4 and get relations r'_n and r'_m. In these relations all existing tuples should go before non existing. In addition to that, relative order of other tuples is the same as in original relations as described sorting procedure is stable. Finding mismatches in these relations is as trivial as going through relations in parallel and searching for first difference:

$$\bigvee_{0 \leq j < min(|r'_n|, |r'_m|)} r'_{n,j} \neq r'_{m,j} \tag{65}$$

3.4 Calculating score of a plan

We already know how to generate a stream of plans, we also can check which of these plans are equivalent to initial one, but what we lack is a check which of these plans is the fastest one. That is, we do not know how to implement `score` function used in improver procedure (Figure 29) and that is what is going to be detailed in this section.

This function is going to return a triple consisting of:

errors – the number of errors inside of given plan. Errors are parts of a plan that would fail to execute. For example, one of the most common errors which is produced by mutation is attribute references that mention nonexisting attributes. Plans that contain at least one error are broken to the level that it is even impossible to convert them to *SMT* format. As such, *errors* component is the first one – if new plan has more errors than the original one, there is no need to check other parts.

equivalency – a flag variable showing whether given plan is equivalent to original one (so equivalency check is actually backed into score value – this simplifies a few places inside of *RDBO* code base);

quality – quality is a scalar value which tries to represent how fast plan would be executed. In reality *DBMS*s use very complicated functions to represent executed plan that depends on various characteristics of the host machine, attribute value distributions, used algorithms, etc. In this project, we are using a function which uses only a fraction of the possible surrounding information. This function is defined as:

- each action node gets a score which is equal to a sum of level at which this node is, quality values of its children, quality of its output;
- quality of `output` is number of different attributes used in all expressions inside of output aliases plus its size;
- index scan gets an additional of 10 points;
- sequence scan gets an additional of 100 points;
- if action node contains a filter then we subtract from its quality its depth multiplied by a number of unique variables inside of the filter.

Having score tuples for two plans we can compare them by finding first score component on which they do not agree – plan having smaller value on this component will be cheaper. By doing so we should generate equivalent simple plans without errors (as the initial plan does not have errors and we accept plans only having less or an equal number of them). Here simple means that quality function prefers plans having a smaller number of nodes and attribute references. It also rewards index scans and penalizes sequence scans. Finally, it prefers plans that contain their filters' conjuncts pushed down the tree.

4 Experiments

In order to test how well the *RDBO* system acts under a stress, we run *OLTBench*'s benchmarks. As stated in the section 2.3, these benchmarks are: `seats`, `tpcc`, `twitter`, `voter` and `wikipedia`. We ran each individual benchmark for approximately 6 hours (as the `tpcc` contains more queries than any other benchmark, we let it run for 44 hours) which ensured 95% level of confidence for an interval 5% around the mean. This time also gave the *RDBO* enough time to converge to the reasonable plans. For each benchmark we also cut 2 minutes worth of the data from the beginning of an experiment and 2 minutes from the end – this is needed to exclude the warm up and the cold down times.

All the results of the experiments will be shown as a charts showing the average time in milliseconds it took to execute plans for the corresponding queries. All of the graphs will be showing two items: how fast were *RDBO*'s generated plans (shown in the red) and how on the same conditions performed *PostgreSQL*'s plans (the green color). Besides the average response time, all of the graphs will include a variance shown as the error bars for a bar charts. Additionally, instead of the queries, we will be presenting only their ids generated by using a *SHA256* on a parametrized query and taking the last 5 characters (mapping between the queries and their ids is given in Appendix 1).

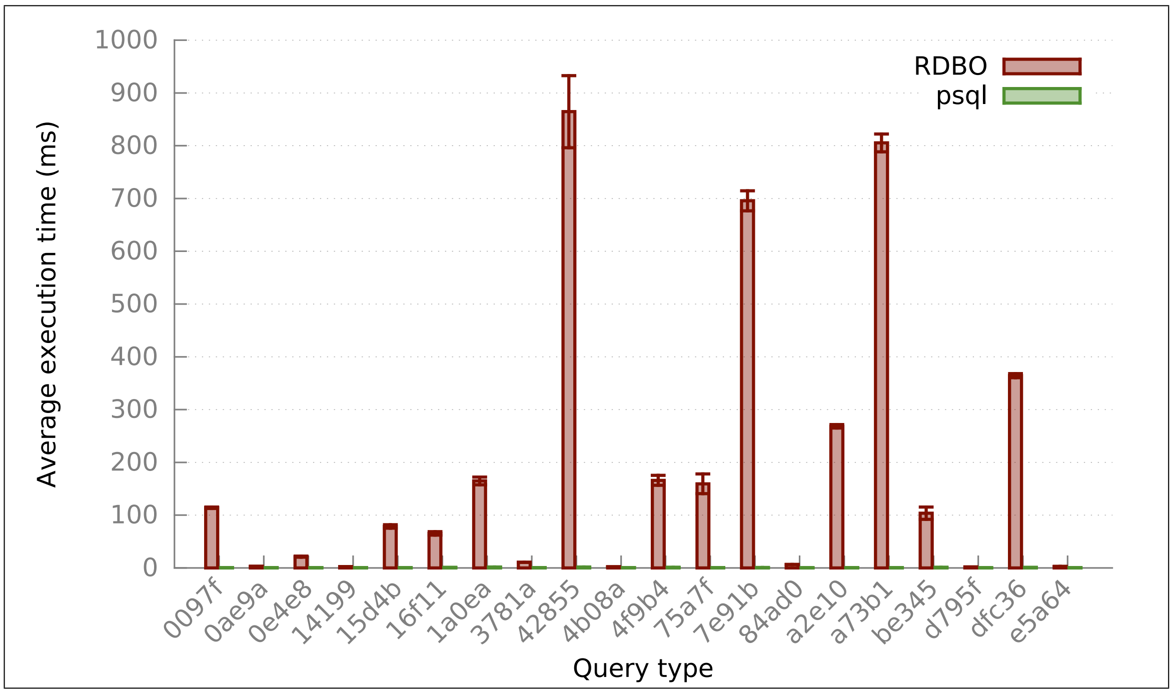


Figure 48: Comparison of naive and *PostgreSQL* execution plans

All of the experiments were ran on the *Digital Ocean* cloud's platform (the second New York's data center region). The overall architecture is very similar to the local one, which was shown in Figure 7. That is, we have 4 machines all running *ubuntu* 14.04 $x64$. First one contains the clients code and has a 512 MB of the *RAM*, a single 1.75 GHz *CPU* and a *SSD* disk. The second machine hosts the proxy, its brief hardware specification: a 1*GB* of the *RAM*, a 1.75 GHz *CPU* and a *SSD* disk. The third one and the fourth one run *DB*s and have the same hardware as the client's machine.

In the following sections, we will present the results of these experiments.

4.1 Performance of a naive plan

We will start the discussion by looking at the simplest experiment possible – performance of a naive plan. Here we disable all *RDBO*'s optimizations. This forces *RDBO* to always use naive plans. Results of such experiment can be observed in Figure 48. As expected, naive plan performed miserably. In the worst case, for the query `42855`, it took naive plan about 594 times more time to find the result compared to an appropriate *PostgreSQL* plan. However, the most important thing holds – naive plans generated the same results as *PostgreSQL* ones. This is an indicator, showing that *RDBO*'s translation mechanism (discussed in section 3.1) and our `execplan` command (section 2.2.2) work as expected.

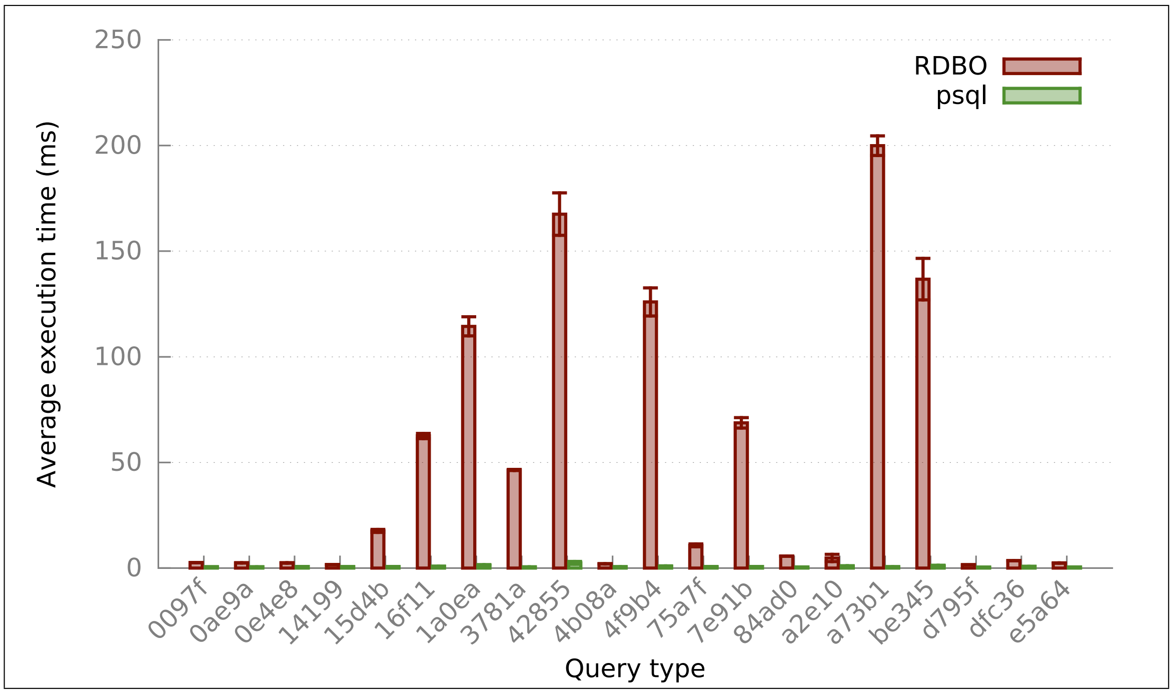


Figure 49: Performance comparison of execution plans generated by *RDBO* and *PostgreSQL*

4.2 Performance of an optimized plan

Having a baseline from which to improve, we can enable improver's threads and rerun the experiments. A bar chart of the response times over the last hour of the experiments (in this period, plans either stopped improving or improvement was very slow) can be seen in Figure 49. From it, we can see that *RDBO*'s performance increased for many, if not all, queries. However, that was not enough to catch up the *PostgreSQL* and at the query 42855 *RDBO* performed about 67 times worse.

```
{
  "plan_id": "sha1-917185b4f00cf531840017ce9064fc906f40e4e8",
  "output": ["no_o_id"],
  "action_tree": {
    "action": "limit",
    "offset": 0,
    "count": 1,
    "output": ["no_o_id"],
    "outer_action": {
      "action": "index scan",
      "relation": "new_order",
      "index": "new_order_pkey",
      "output": ["no_o_id"],
      "filter": {
        "func": "and",
        "args": [
          {"func": "=", "args": ["no_w_id", {"const_param": "$2"}]},
          {"func": "=", "args": ["no_d_id", {"const_param": "$1"}]}]}}}}
```

Figure 50: The final *RDBO*'s output for the query with id 0e4e8

```
explain analyze select no_o_id from new_order where no_d_id = 1 and no_w_id = 2 order by
-- QUERY PLAN
-- Limit (cost=0.29..0.75 rows=1 width=4)
--        (actual time=0.050..0.051 rows=1 loops=1)
-- -> Index Scan using new_order_pkey on new_order (cost=0.29..197.99 rows=426 width=4)
--                                                  (actual time=0.048..0.048 rows=1 loop
--     Index Cond: ((no_w_id = 2) AND (no_d_id = 1))
```

Figure 51: *PostgreSQL*'s generated plan for the query with id 0e4e8

.

To understand better what happened, we can categorize the queries into three groups. In the first group, we have the queries that were optimized fully. That is, *RDBO*'s generated plans for these queries matched, or were very close, to the *PostgreSQL*'s plans. One of the queries that lies in this group is 0e4e8. *RDBO*'s generated plan for this query can be seen in Figure 50 and *PostgreSQL*'s in Figure 51. Despite the equivalency of these plans, *RDBO*'s plan was still slower, running at the average speed of 2.4 ms per query (compared to the *PostgreSQL*'s 0.7 ms). Actually, all the plans in this group were slightly slower than we expected. This is due to the way performance of these plans is measured. For *RDBO*, a response time consists of these parts:

- a plan parsing – in this step, patched *PostgreSQL* translates given plan's *JSON* string into a query plan. This step involves converting *JSON* into an internal object, resolving attribute and relation names, adding additional nodes to a plan, etc.;
- a portal managing – in order to execute a plan inside of the *PostgreSQL*, we have to create a portal (can be viewed as some sort of an execution environment), assign a plan to it, execute it (not included in a portal management timing), and finally, destroy it;

- a data retrieval – this is the primary part in which we are executing a plan by traversing the heap and producing output tuples;
- an other – before a plan hits patched code it goes through the several general *PostgreSQL*'s layers. This involves *PostgreSQL*'s parser (parsing the `execplan` part), transaction cop, etc.

Table 2: Inspection of query execution parts

	Query `0e4e8`			Overall		
Part	Cumulative s	Avg. ms	Percentage	Cumulative s	Avg. ms	Percentage
Plan parsing	0.05	0.19	7.71%	119.85	3.16	2.74%
Data retrieval	0.10	0.37	15.33%	3199.72	84.27	73.06%
Portal management	0.04	0.13	5.41%	81.65	2.15	1.86%
Other	0.49	1.74	71.55%	978.14	25.76	22.34%
Total	0.68	2.43		4379.37	115.34	

Run times of each of these parts separately for the query `0e4e8` and overall commutative impact of all queries in the optimized query group (includes queries: `0ae9a`, `4b08a`, `7e91b`, `e5a64`, `0e4e8`, `15d4b`, `14199`, `d795f`, `0097f`, `84ad0`, `a2e10`, `dfc36`) can be seen in Table 2. As expected, a data retrieval is either the most or the second most demanding part. If we would take only this value and compare it to the *PostgreSQL*, we could observe that now for most of the plans we get faster run times. This is due to the fact that *PostgreSQL*'s query execution life cycle is different from ours. For actual execution, *JDBC* driver is not sending a full query, instead, it sends only the prepared statement's id and the bind parameters. These values are assigned to an in advance created portal. As such, *PostgreSQL* execution includes little of that we have under „Plan parsing“, „Portal management“ and „Other“ parts. On the other hand, *PostgreSQL* may re-plan prepared statements – so increasing the query execution time. As such, making a fair comparison is actually a hard task and for simplicity sake, we assume that anything within 10 ms should be considerate equal.

Second group consists of queries whose optimized versions contain nodes that are not supported by the `execplan` command. As such, it was impossible for the *RDBO* to find an optimal plan. This group contains only a single query `16f11` for which an optimal plan contained an indexed array expression.

What is left for the third group, are queries whose plans suffered from nonmonotonicity of a score function and poor exploration capabilities of the improver. That is, a proper scoring function should produce smaller values for faster plans, but our function does not have such property. One prominent example where not having such property can lead to bad results is the query `SELECT SUM(OL_AMOUNT)AS OL_TOTAL FROM ORDER_LINE WHERE OL_O_ID = $1 AND OL_D_ID = $2 AND OL_W_ID = $3` (query id `a73b1`). A

```
insert into rel1 (attr1, attr2, attr3)
  select generate_series(1, 10000000),
         generate_series(1, 10000000),
         generate_series(1, 10000000);
explain analyze
  select attr3 from rel1
  where attr2 + 1 = attr3 + 1 and attr3 = 500000;
-- QUERY PLAN
-- Seq Scan on rel1 (cost=0.00..254055.00 rows=250 width=4)
--                  (actual time=185.203..3155.304 rows=1 loops=1)
--                  Filter: ((attr3 = 500000) AND ((attr2 + 1) = (attr3 + 1)))
--                  Rows Removed by Filter: 9999999
-- Execution time: 3155.344 ms

execplan e_json '{
  "plan_id": "sha1-51c7fd35b14cf4c64ddcd6e8ce98716ff3551dd3",
  "output": [ "attr3" ],
  "action_tree":{
    "action": "join by nested loop",
    "join_type": "inner",
    "output": [ { "alias": "attr3", "expr": "?gen-3?" } ],
    "filter": { "func": "=", "args": [ { "func": "+", "args": [ "?gen-2?", 1 ] },
                                       { "func": "+", "args": [ "?gen-3?", 1 ] }]},
    "outer_action":{
      "action": "index scan",
      "relation": "rel1",
      "index": "uniq1",
      "output": [ { "alias": "?gen-2?", "expr": "attr2" },
                  { "alias": "?gen-3?", "expr": "attr3" }],
      "filter": { "func": "=", "args": [ "attr2", 500000 ] }},
      "inner_action": { "action": "rtn_result", "output":[] }}}';
-- Execution time: 0.297 ms
```

Figure 52: Query for which *RDBO* finds better plan than *PostgreSQL*

proper plan for this query should consist of a sorted aggregate node, followed by an index scan and filtered by all the three conjuncts. *RDBO* started the search for such plan by going from the naive one containing an aggregate node, a sequence scan, and some other excess nodes. The score of this initial plan was 245. After some time, *RDBO* managed to drop most of the useless nodes and produce a plan with the score of 113. Then it started to use an index scan, filtering by the first attribute – `ol_w_id`. This led to a plan having the score of 22. The problem with this new indexed scan is that it is actually slower than the one with the score of 113. This is because selectivity of the `ol_w_id` is rather small so, the index does not filter out a lot of tuples and instead give a penalty for jumping between indexed structure and heap. Thankfully, after some more time, *RDBO* managed to move the second attribute to an index's filter which gave a physically faster plan (score – 17). Not all queries were so lucky, and some of them stuck in the places that gave a great score but were actually bad. Queries that were stuck because of the scoring function or were not optimized fully because of the timing constraints: `42855`, `4f9b4`, `75a7f`, `a73b1`, `be345`, `e5a64`, `1a0ea`, `3781a`.

4.3 Cutom queries

To conclude the experimental section, we provide several queries (not from *OLTBench*) for which *RDBO* finds better plans than *PostgreSQL*.

The first query and its execution statistics can be observed in Figure 52. Here we are doing a simple search over the relation `rel1` from our test *DB* (Figure 30). In this search, we are filtering out all the tuples on which `attr2` and `attr3` are not equal (with excessive `+ 1` expression) or `attr3` is not equal to 500000. As the attribute `attr3` is not used in any indexe's prefix, *PostgreSQL* concludes that it is impossible to optimize the filter condition any better than using a simple sequence scan[10]. *RDBO*, on the other hand, notices that there is an index on `attr2` and, because `attr2 = attr3`, it can use it to improve plan's execution performance. As such, *RDBO*'s generated plan runs over 10 thousand times faster.

```
insert into rel1 (attr1, attr2, attr3)
  select generate_series(1, 100000000),
         generate_series(1, 100000000),
         generate_series(1, 100000000);
explain analyze select attr1 from rel1 where attr1 is null;
-- QUERY PLAN
-- Index Scan using rel1_attr1_pk on rel1
--              (cost=0.57..1381758.57 rows=500000 width=4)
--              (actual time=0.041..0.041 rows=0 loops=1)
--              Index Cond: (attr1 IS NULL)
-- Execution time: 0.106 ms

execplan e_json '{
  "plan_id": "sha1-0f08efa67ae2070ff975ceaefaece6894d718611",
  "output": [ "attr1" ],
  "action_tree":{
    "action": "rtn_result",
    "output": [ { "alias": "attr1", "expr": 0 } ],
    "filter": false }}';
-- Execution time: 0.001 ms
```

Figure 53: Query whose optimal plan depends on `check` constraint

The second query can be seen in Figure 53. This query finds all tuples from `rel1` in which `attr1` is equal to `NULL`. The only problem with such search is that `attr1` is marked as `NOT NULL` as such no tuples will ever be returned. *RDBO* can easily prove such fact and, after several minutes of optimization, comes up with a very simple plan which always returns an empty set of tuples. On the other hand, *PostgreSQL* never looks at `CHECK` and have to to use index scan which, for given data sample, is 100 times slower.

The final query is shown in Figure 54. In this query, we are searching for a tuple with the smallest `attr3` value and in which `attr2` is undefined. The best plan that we can generate for it contains a single index scan over attributes `attr2` and `attr3`. As index is already sorted there is no need to add any additional `sort` node, but *PostgreSQL* inserts one. The problem here is the `IS NULL` operator. In *PostgreSQL*, this operator is not an equity, but a set membership check. This small difference is enough for *PostgreSQL*'s planner not to consider a `uniq1` index as a viable access path[11]. Using simple sequence scan resolves in a plan which is 200 thousand times faster compared to the plan generated

[10]Actually, *PostgreSQL* is able to deduce transitive aliases when they are created by simple equity operator without any additional expressions.

[11]If we would use `attr2 = 1` instead of `attr2 is null`, we would get a desired plan

by *RDBO*.

```
insert into rel1 (attr1, attr2, attr3)
  select generate_series(1, 10000000),
         case when random() < 0.5 then null else 1 end,
         generate_series(1, 10000000);
explain analyze select attr1 from rel1 where attr2 is null order by attr3 limit 1;
-- QUERY PLAN
-- Limit (cost=174223.02..174223.02 rows=1 width=8)
--       (actual time=4593.337..4593.338 rows=1 loops=1)
--   Sort (cost=174223.02..186805.52 rows=5033001 width=8)
--        (actual time=4593.336..4593.336 rows=1 loops=1)
--        Sort Key: attr3
--        Sort Method: top-N heapsort  Memory: 25kB
--     Seq Scan on rel1 (cost=0.00..149058.01 rows=5033001 width=8)
--                      (actual time=0.635..3182.984 rows=5002203 loops=1)
--                      Filter: (attr2 IS NULL)
--                      Rows Removed by Filter: 4997797
-- Execution time: 4593.380 ms

execplan e_json '{
  "plan_id":"sha1-83254ad6156486b395cb96daea5e7db6f14a7751",
  "output": [ "attr1" ],
  "action_tree": {
    "action": "limit",
    "output": [ "attr1" ],
    "count": 1,
    "outer_action": {
      "action": "index scan",
      "output": [ "attr1" ],
      "relation": "rel1",
      "index": "uniq1",
      "filter": { "func": "is null", "args": [ "attr2" ] }}}}';
-- Execution time: 0.021 ms
```

Figure 54: Query sorted by attribute tuple containing NULL

Results and conclusions

In this paper, we present a problem of converting a human-readable *SQL* query into the executable rules expressed as an execution plan. By introducing a simple subproblem of optimal join ordering, we show how complicated and slow such conversion could be in practice. In order to speed up this process, most *RDBMS*s provide several ways to avoid at least some parts of it. We detail most of the common ways of doing so, in particular, stressing a usage of precompiled queries.

We also examine another problem of program compilation into executable machine instructions and conclude that this compilation problem is inherently similar to a query planning. This is due to the fact that currently most compilers, similarly as *RDBMS*s, use special purpose algorithms (most prominently peephole optimization) together with various heuristics to do their job. The problem with these techniques is that they try to solve the computationally expensive problem fast which leads to suboptimal results.

There is also a new wave of general purpose optimizers that are used in compilation process and can actually find truly optimal machine code. In the beginning of this paper we present a history of these so called superoptimizers. We also explain how these superoptimizers work – they look at a compilation as a complex optimization problem which can be solved by a brute force (trying out all the possible candidate sequences) or by using randomized algorithms (like the Monte Carlo search or the genetic programming). Obviously, in order to apply these methods, superoptimizer has to be able to prove that generated sequences are equivalent to the initial one. To do that, superoptimizers represent input instructions as mathematical formulas in *SMT* logic – a distinguished superoptimizer tactic – which allows usage of state of the art solvers (like *Z3*) to do actual equivalence proves for them. Though, at least in theory, superoptimizer can produce optimal code, they tend to be way too slow. This problem is solved by using them offline in preparation of conversion templates used by peephole optimizers in advance.

In the paper, we note that query planners are not using anything similar to superoptimizers, though having rarely changed query pools of precompiled queries seems to be a great target for long running optimizations, and try to apply examined techniques ourselves. We start an exploration of superoptimization application in *RDBMS*s by outlining an implementation of general purpose framework for query interception written as a side product of this thesis. This framework consists primarily of proxy service, which sits between client and *DB*. This service captures all packets flowing through it, combines them to query execution commands (query and its bind parameters) and parses these commands by producing easily malleable query *AST*. Any module can hook to this process and do query modifications at will. Proxy also allows to branch execution to other *DB*s executing on them altered queries. This allows one *DB* to act as a control mechanism which always produces correct results, as we are executing unchanged queries on it, and other *DB* as a testing service on which altered queries have to produce same results as unchanged ones on control *DB*.

As part of this framework, we also extend *PostgreSQL* to support plan execution. This plan execution is carried by a new command called `execplan` which as its second argument (first argument is always a constant `e_plan`) takes query execution plan as *JSON* object. In the paper, we also give a detailed explanation of this object. That is, we describe what kind of plan nodes this object can contain and what they do.

After that, we overview a created superoptimization module which was plugged into a previously mentioned framework. We start this overview by detailing query plan conversion to set of *SMT* formulas and how these formulas were used to prove equivalency between two plans. We show that direct plan encoding is too slow for practical use (direct conversion would require a use of quantifiers for which no general purpose prover could guarantee termination). To mitigate this problem, we make a restriction on maximum relation size – while converting we assumed that no initial relation will have more than 3 tuples. This allowed us to manually ground all formulas and define any plan as a pipeline of simple transformations on a finite set of free variables, defined from the *DB*'s schema. We also show that such model bounding may sometimes have a negative impact on equivalency checking – any plan which has to touch more tuples than predefined bound will be equated to an empty query.

As part of superoptimization module's overview, we also define a scoring fuction used in comparing two plans. We define this score function as a triple consisting of a number of errors inside of a generated plan, a flag showing whether plan is equivalent to the target plan and a scalar value which tries to estimate plans qualitative properties (in theory, this estimate should be a normalized representation of plans executions speed). We define a third, and most important, component of this triple to be a simple additive function on nodes children which tries to promote small index based scans. In the paper, we note that this is a naive definition as it does not take into its account things like environment on which superoptimizer runs and time changing data variability, but is good enough for basic optimization needs.

We use this score function inside of created superoptimizer's exploratory search. In this search an improver thread takes a plan that was determined to be needing optimizations from the job queue. Later on, improver for some predetermined amount of time tries to find a better plan by using an algorithm similar to hill climbing algorithm. That is, for a particular plan a randomized mutation procedure (which we explained in details) is applied several times. Current plan is changed to a mutated one if its score is lower, otherwise, improvement search continues from a copy of the original plan. After improvement time runs out, the plan is put back to queue – all appropriate queries from that time on will use this improved plan.

Finally, this module was tested by running *OLTBench* benchmarks on it. The results of these benchmarks showed that:

- our created mathematical model is sufficient to work with queries used in a practice as all plans – optimized, *PostgreSQL* and naive ones – returned matching result sets;
- our optimized query plans compared to *PostgreSQL* were:
  - as fast as *PostgreSQL* generated plans – happened to 12 plans;
  - not optimized due to us not implementing some of the plan nodes – a single plan;
  - not optimized fully due to nonmonotonicity of scoring function and poor query plan exploration – 8 plans.

We also showed that there are queries for which our superoptimizer performed better than *PostgreSQL* one:

- ability to look at `NULL` constraints allowed superoptimizer to collapse one of the queries to a single expression. Empirically, this gave a plan which is 100 times faster than a *PostgreSQL*'s created plan;
- knowledge that indexes are ordered structures allows our superoptimizer to remove unnecessary `sort` nodes in a plan. For one particular query this knowledge gave 200 thousand times faster plan;
- deduction of transitive attribute aliases allows superoptimizer to use indexes indirectly. In the paper, we presented a query for which this ability allowed to generate a query plan which is 10 thousand times faster.

From all these benchmarks we conclude that superoptimization could be a viable solution for *RDBMS*'s in a near future, but before that, following tasks/problems should be solved:

- a detection of query plan instances for which mathematical model can produce a false positive result;
- a creation of a better scoring function which could work ofline;
- an improvement of a query plan exploration.

Investigation of possible solutions for these problems could be a topic for future works.

References

[ACN00] S. Agrawal, S. Chaudhuri and V. R. Narasayya. *Automated selection of materialized views and indexes in sql databases*. In *Proceedings of the 26th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000, VLDB '00, pp. 496–505

[BA06] S. Bansal and A. Aiken. *Automatic generation of peephole superoptimizers*. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 2006, ASPLOS XII, pp. 394–403

[Bat68] K. E. Batcher. *Sorting networks and their applications*. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. ACM, New York, NY, USA, 1968, AFIPS '68 (Spring), pp. 307–314

[CBC93] S. Choenni, H. M. Blanken and T. Chang. *On the selection of secondary indices in relational databases*, 1993

[CFM86] U. S. Chakravarthy, D. H. Fishman and J. Minker. *Semantic query optimization in expert systems and database systems*. In *Proceedings from the First International Workshop on Expert Database Systems*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1986, pp. 659–674

[CN98] S. Chaudhuri and V. R. Narasayya. *Autoadmin 'what-if' index analysis utility*. In L. M. Haas and A. Tiwary, eds., *SIGMOD Conference*. ACM Press, 1998, pp. 367–378

[Dat08] C. Date. *The Relational Database Dictionary, Extended Edition*. Apressus Series. Apress, 2008

[Dat11] C. Date. *SQL and Relational Theory: How to Write Accurate SQL Code*. O'Reilly Media, 2011

[DMB11] L. De Moura and N. Bjørner. *Satisfiability modulo theories: Introduction and applications*. In *Commun. ACM*, 54(9), pp. 69–77, 2011

[DPCCM13] D. E. Difallah, A. Pavlo, C. Curino and P. Cudre-Mauroux. *Oltp-bench: An extensible testbed for benchmarking relational databases*. In *Proc. VLDB Endow.*, 7(4), pp. 277–288, 2013

[DV97] E. Dantsin and A. Voronkov. *Logical Foundations of Computer Science: 4th International Symposium, LFCS'97 Yaroslavl, Russia, July 6–12, 1997 Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, chap. Complexity of query answering in logic databases with complex values, pp. 56–66, 1997

[GA05] T. Granlund and S. Ab. *Instruction latencies and throughput for amd and intel x86 processors*, 2005

[GMZ02] R. Gupta, E. Mehofer and Y. Zhang. *Profile guided compiler optimizations*, 2002

[Gö31] K. Gödel. *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*. In *Monatshefte für Mathematik und Physik*, 38(1), pp. 173–198, 1931

[HSH07] J. Hellerstein, M. Stonebraker and J. Hamilton. *Architecture of a Database System*. Foundations and trends in databases. Now Publishers, 2007

[HSZ08] C. Henke, C. Schmoll and T. Zseby. *Empirical evaluation of hash functions for multipoint measurements*. In *SIGCOMM Comput. Commun. Rev.*, 38(3), pp. 39–50, 2008

[Jen] B. Jenkins. *lookup3*. `http://burtleburtle.net/bob/c/lookup3.c`. *[online; accessed 2015-04-12]*

[JNR02] R. Joshi, G. Nelson and K. Randall. *Denali: A goal-directed superoptimizer*. In *SIGPLAN Not.*, 37(5), pp. 304–314, 2002

[JNZ+03] R. Joshi, G. Nelson, Y. Zhou, R. Joshi, G. Nelson and Y. Zhou. *The straight-line automatic programming problem*, 2003

[KK07] N. Kerdprasop and K. Kerdprasop. *Semantic knowledge integration to support inductive query optimization*. In *Proceedings of the 9th International Conference on Data Warehousing and Knowledge Discovery*. Springer-Verlag, Berlin, Heidelberg, 2007, DaWaK'07, pp. 157–169

[Klea] Kle11. *Dell dvd store*. `http://linux.dell.com/dvdstore/`. *[online; accessed 2015-11-21]*

[Kleb] Kle11. *Transaction processing performance council*. `http://www.tpc.org/default.asp`. *[online; accessed 2015-11-21]*

[Luk13] S. Luke. *Essentials of Metaheuristics*. Lulu, second edn., 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/

[Mas87] H. Massalin. *Superoptimizer: A look at the smallest program*. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1987, ASPLOS II, pp. 122–126

[McK65] W. M. McKeeman. *Peephole optimization*. In *Commun. ACM*, 8(7), pp. 443–444, 1965

[Pos14a] PostgreSQL Global Development Group. *Genetic Query Optimizer*, 2014. PostgreSQL documentation. [Online; accessed 2015-03-14] `http://www.postgresql.org/docs/9.4/static/geqo.html`

[Pos14b] PostgreSQL Global Development Group. *pgbench*, 2014. PostgreSQL documentation. [Online; accessed 2015-11-22] `http://www.postgresql.org/docs/9.4/static/pgbench.html`

[Pos14c] PostgreSQL Global Development Group. *PostgreSQL FEBE flow*, 2014. PostgreSQL documentation. [Online; accessed 2015-01-12] `http://www.postgresql.org/docs/9.4/static/protocol-flow.html`

[Pos14d] PostgreSQL Global Development Group. *PostgreSQL Message Formats*, 2014. PostgreSQL documentation. [Online; accessed 2015-01-12] `http://www.postgresql.org/docs/9.4/static/protocol-message-formats.html`

[Pos15a] PostgreSQL Global Development Group. *Pushing order by + limit to union subqueries*, 2015. Pgsql-performance (Mailing list). [Online; accessed 2015-02-28] `http://www.postgresql.org/message-id/CAP=2L=FRza_catqP9LRfJOMybzoorpxSBxSP=J1o9WZnFp1USg@mail.gmail.com`

[Pos15b] PostgreSQL Global Development Group. *SELECT slows down on sixth execution*, 2015. Pgsql-performance (Mailing list). [Online; accessed 2015-10-14] `http://www.postgresql.org/message-id/561E068F.1040702@socialserve.com`

[PS04] M. Poess and J. M. Stephens, Jr. *Generating thousand benchmark queries in seconds*. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB Endowment, 2004, VLDB '04, pp. 1045–1053

[SAC$^+$79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price. *Access path selection in a relational database management system*. In

Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data. ACM, New York, NY, USA, 1979, SIGMOD '79, pp. 23–34

[SS98] D. Slutz and D. Slutz. *Massive stochastic testing of sql*. In *In VLDB*. Morgan Kaufmann, 1998, pp. 618–622

[SSA13] E. Schkufza, R. Sharma and A. Aiken. *Stochastic superoptimization*. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 2013, ASPLOS '13, pp. 305–316

[SSCA15] R. Sharma, E. Schkufza, B. Churchill and A. Aiken. *Conditionally correct superoptimization*. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA, 2015, OOPSLA 2015, pp. 147–162

[SSS92] M. Siegel, E. Sciore and S. Salveter. *A method for automatic rule derivation to support semantic query optimization*. In *ACM Trans. Database Syst.*, 17(4), pp. 563–600, 1992

[VGdHT09] M. Veanes, P. Grigorenko, P. de Halleux and N. Tillmann. *Symbolic query exploration*. In *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, 2009, pp. 49–68

[VTdH10] M. Veanes, N. Tillmann and J. de Halleux. *Qex: Symbolic SQL query explorer*. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, 2010, pp. 425–446

[Win12] M. Winand. *SQL Performance Explained: Everything Developers Need to Know about SQL Performance*. M. Winand, 2012

Glossary

A | C | D | F | G | I | J | M | R | S | T | U | V

A

Abstract syntax tree

A tree representing structure of compiled program or query. 13, 27, 41, 72

ADO.NET

Part of Microsoft .NET Framework that enables comunication with *RDBMS*. 21

C

Central processing unit

Circuit used for machine level instruction execution. 17–19, 65, 66

D

Database

Collection of structured data. 10–12, 66, 70, 72, 73

Database management system

Software which is used to manipulate *DB*. 10, 64

Domain specific language

A language created for modeling particular domain problem. 23

F

Foreign function interface

A mechanism that allows two languages to call each other functions. 23

Frontend/Backend protocol

Protocol used by by various applications to to communicate with *PostgreSQL* backend. 23

G

Grand unified configuration

Mechanism used by *PostgreSQL* to manage its configuration variables on various levels at runtime. 25, 26

I

Internet protocol

Internet layer protocol used for packet relaying based on *IP* address. 21, 22

J

Java Database Connectivity

Programming interface that allows to communicate with *RDBMS* from *Java*. 21, 26, 68

JavaScript object notation

General purpose data encoding format. 29, 30, 34–36, 67, 73

M

Multiversion concurrency control

A concurrency control implementation based on concept of dublicated values. 13, 27, 29, 31

R

Random Access Memory

Volatile data storage device that allows to access data within the same speed despite its physical location. 65, 66

Relational database management system

DBMS which uses relational data model. 11–18, 20, 21, 23, 25–30, 37, 41, 45, 46, 48, 62, 72, 74

Relational database optimizer

Package of programs created while writing master thesis. 21–23, 25–28, 32, 35–37, 41, 47, 51, 54, 56, 63, 65–71, 76

S

Satisfiability modulo theories

First order logic formula encoded with a help of specialized predicates from a variety of underlying theories. 12, 19, 41, 45–48, 50, 51, 54, 62, 63, 72, 73

Secure Sockets Layer

Application layer protocol which uses a set of cryptographic primitives to ensure secure communication over network. 23, 24

Solid-state drive

Persistance data storage device. 66

Structured query language

Most popular language for writing *RDBMS* queries. 11, 13, 16, 17, 21, 27, 33, 38, 45, 50, 51, 55, 58, 72

T

Transaction Processing Performance Council

Non-profit organization defining *RDBMS* benchmarks and openly publising their results. 38

Transmission control protcol

Transport layer protocol used for reliable session based packet transportation. 23

U

Universal Serial Bus

Standard which defines cables and protocols used in communication between various digitilized systems. 10

V

Virtuam machine

Software implementation of machine that enables user to run run operating systems in isolation. 21, 28, 40

Acronyms

A | C | D | F | G | I | J | M | O | R | S | T | U | V

A

AST

Abstract syntax tree. 13, 27, 41, 72

C

CPU

Central processing unit. 17–19, 65, 66

D

DB

Database. 10–12, 66, 70, 72, 73

DBMS

Database management system. 10, 64

DSL

Domain specific language. 23

F

FEBE

Frontend/Backend protocol. 23

FFI

Foreign function interface. 23

G

GUC

Grand unified configuration. 25, 26

I

IP

Internet protocol. 21, 22

J

JDBC

Java Database Connectivity. 21, 26, 68

JSON

JavaScript object notation. 29, 30, 34–36, 67, 73

M

MVCC

Multiversion concurrency control. 13, 31

O

OID

Multiversion concurrency control. 27, 29

R

RAM

Random Access Memory. 65, 66

RDBMS

Relational database management system. 11–18, 20, 21, 23, 25–30, 37, 41, 45, 46, 48, 62, 72, 74

RDBO

Relational database optimizer. 21–23, 25–28, 32, 35–37, 41, 47, 51, 54, 56, 63, 65–71, 76

S

SMT

Satisfiability modulo theories. 12, 19, 41, 45–48, 50, 51, 54, 62, 63, 72, 73

SQL

Structured query language. 11, 13, 16, 17, 21, 27, 33, 38, 45, 50, 51, 55, 58, 72

SSD

Solid-state drive. 66

SSL

Secure Sockets Layer. 23, 24

T

TCP

Transmission control protcol. 23

TPC

Transaction Processing Performance Council. 38

U

USB

Universal Serial Bus. 10

V

VM

Virtuam machine. 21, 28, 40

Applications

D | F | G | L | O | P | Q | R | U | V | Z

D

Dells DVD stores simulation

`http://linux.dell.com/dvdstore/`

Benchmark based on ecommerse site simulation. 21, 38, 40

F

fortune

`http://minnie.tuhs.org/cgi-bin/utree.pl?file=V7/usr/src/games/fortune.c`

Unix V7 program that display randomized message. 44

G

GCC

`https://gcc.gnu.org/`

A collection of programmin language compilers. 20

Graphviz

`http://www.graphviz.org/`

A software suite used in graph visualization. 23

L

LLVM

`http://llvm.org/`

A collection of compiler specific technologies. 20

O

OLTBench

`http://oltpbenchmark.com/`

A framework consisting of multiple benchmarks created for *DB* testing. 21, 38, 40, 65, 69, 73, 76

P

pg_query

`https://github.com/lfittl/pg_query`

PostgreSQL style query parser for Ruby. 27

PostgreSQL

`http://www.postgresql.org/`

Opensource *RDBMS*. 12, 15, 16, 22, 23, 25, 27–30, 33, 35, 37–40, 65–70, 73, 74

Q

QEX

Testing data generator for *SQL* queries. 46

QGEN

Query generator. 38

R

RAGS

Randomized query generator. 38

REX

Testing data generator for regular expressions. 46

RUST

`https://www.rust-lang.org/`

A system programming language. 23, 26

rust-peg

`https://github.com/kevinmehall/rust-peg`

Parser generator for *RUST*. 27

rust-postgres

`https://github.com/sfackler/rust-postgres`

PostgreSQL driver for *RUST*. 27

U

ubuntu

`http://www.ubuntu.com/`

Linux operating system. 65

V

Vagrant

`https://www.vagrantup.com/`

VM management system. 21

Z

Z3

`http://rise4fun.com/z3`

Automatic theorem prover. 41, 46, 48, 72

Appendix 1. Used queries

All queries used in experiments are shown in the table below (queries are taken from *OLTBench* [DPCCM13]). In this table column „Opt.“ represents how well queries were optimized by *RDBO* and can contain following symbols:

- ✓– *RDBO* found optimal plan;
- ✗– found plan is suboptimal;
- ➘– plan uses too many optimizations (mostly using index scan then simple sequence scan is faster);
- ➚– plan for given query is optimal, but contains up to 3 excess nodes.

Query id	Opt.	Query
		`tpcc`
0097f	➚	`SELECT I_PRICE, I_NAME , I_DATA FROM ITEM WHERE I_ID = $1`
0ae9a	➘	`SELECT D_STREET_1, D_STREET_2, D_CITY, D_STATE, D_ZIP, D_NAME FROM DISTRICT WHERE D_W_ID = $1 AND D_ID = $2`
0e4e8	✓	`SELECT NO_O_ID FROM NEW_ORDER WHERE NO_D_ID = $1 AND NO_W_ID = $2 ORDER BY NO_O_ID ASC LIMIT 1`
15d4b	✓	`SELECT O_C_ID FROM OORDER WHERE O_ID = $1 AND O_D_ID = $2 AND O_W_ID = $3`
1a0ea	✗	`SELECT C_FIRST, C_MIDDLE, C_ID, C_STREET_1, C_STREET_2 , C_CITY, C_STATE, C_ZIP, C_PHONE, C_CREDIT, C_CREDIT_LIM, C_DISCOUNT, C_BALANCE, C_YTD_PAYMENT, C_PAYMENT_CNT, C_SINCE FROM CUSTOMER WHERE C_W_ID = $1 AND C_D_ID = $2 AND C_LAST = $3 ORDER BY C_FIRST`
42855	✗	`SELECT OL_I_ID, OL_SUPPLY_W_ID, OL_QUANTITY, OL_AMOUNT, OL_DELIVERY_D FROM ORDER_LINE WHERE OL_O_ID = $1 AND OL_D_ID =$2 AND OL_W_ID = $3`
4b08a	➘	`SELECT D_NEXT_O_ID, D_TAX FROM DISTRICT WHERE D_W_ID = $1 AND D_ID = $2 FOR UPDATE`
4f9b4	✗	`SELECT C_Fsql, C_MIDDLE, C_LAST, C_STREET_1, C_STREET_2 , C_CITY, C_STATE, C_ZIP, C_PHONE, C_CREDIT, C_CREDIT_LIM, C_DISCOUNT, C_BALANCE, C_YTD_PAYMENT, C_PAYMENT_CNT, C_SINCE FROM CUSTOMER WHERE C_W_ID = $1 AND C_D_ID = $2 AND C_ID = $3`
75a7f	✗	`SELECT C_DATA FROM CUSTOMER WHERE C_W_ID = $1 AND C_D_ID = $2 AND C_ID = $3`

7e91b	↘	SELECT C_DISCOUNT, C_LAST, C_CREDIT, W_TAX FROM CUSTOMER, WAREHOUSE WHERE W_ID = $1 AND C_W_ID = $2 AND C_D_ID = $3 AND C_ID = $4
a73b1	✗	SELECT SUM(OL_AMOUNT)AS OL_TOTAL FROM ORDER_LINE WHERE OL_O_ID = $1 AND OL_D_ID = $2 AND OL_W_ID = $3
be345	✗	SELECT O_ID, O_CARRIER_ID, O_ENTRY_D FROM OORDER WHERE O_W_ID = $1 AND O_D_ID = $2 AND O_C_ID = $3 ORDER BY O_ID DESC LIMIT 1
dfc36	↗	SELECT S_QUANTITY, S_DATA, S_DIST_01, S_DIST_02, S_DIST_03, S_DIST_04, S_DIST_05, S_DIST_06, S_DIST_07, S_DIST_08, S_DIST_09, S_DIST_10 FROM STOCK WHERE S_I_ID = $1 AND S_W_ID = $2 FOR UPDATE
e5a64	↘	SELECT W_STREET_1, W_STREET_2, W_CITY, W_STATE, W_ZIP, W_NAME FROM WAREHOUSE WHERE W_ID = $1
wikipedia		
84ad0	↗	SELECT * FROM page WHERE page_namespace = $1 AND page_title = $2 LIMIT 1
voter		
14199	✓	SELECT state FROM AREA_CODE_STATE WHERE area_code = $1
3781a	✗	SELECT COUNT(*)FROM VOTES WHERE phone_number = $1
d795f	✓	SELECT contestant_number FROM CONTESTANTS WHERE contestant_number = $1
seats		
a2e10	↗	SELECT R_ID, R_F_ID, R_SEAT FROM "RESERVATION" WHERE R_F_ID = $1
cfab8	✗	SELECT * FROM "CUSTOMER" WHERE C_ID = $1
d07f5	✓	SELECT C_ID FROM "CUSTOMER" WHERE C_ID_STR = $1
twitter		
16f11	✗	SELECT uid, name FROM "user_profiles" WHERE uid IN ($1 , $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13, $14, $15, $16, $17, $18, $19, $20)