Finding the Pitfalls in Query Performance

M.L. Kersten*
Centrum Wiskunde & Informatica
martin.kersten@cwi.nl

P. Koutsourakis
MonetDB Solutions
panagiotis.koutsourakis@monetdbsolutions.com

Y. Zhang
MonetDB Solutions
ying.zhang@monetdbsolutions.com

ABSTRACT

Despite their popularity, database benchmarks only highlight a small part of the capabilities of any given system. They do not necessarily highlight problematic components encountered in real life or provide hints for further research and engineering.

In this paper we introduce *discriminative performance benchmarking*, which aids in exploring a larger search space to find performance outliers and their underlying cause. The approach is based on deriving a domain specific language from a sample query to identify a query workload. SQLsCALPEL subsequently explores the space using query morphing, and simulated annealing to find performance outliers, and the query components responsible. To speed-up the exploration for often time-consuming experiments SQLsCALPEL has been designed to run asynchronously on a large cluster of machines.

ACM Reference Format:
M.L. Kersten, P. Koutsourakis, and Y. Zhang. 2018. Finding the Pitfalls in Query Performance. In *DBTest'18: Workshop on Testing Database Systems , June 15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3209950.3209951

1 INTRODUCTION

Each new version of a database system ideally shows a better performance. Commercial providers have a large quality assurance team that can study and leverage the workload of key customers to guard against dissatisfied users. In research, the predominant approach is to rely on benchmarks such as TPC-H to measure progress.

Designing a new high-performance database system from scratch is even more cumbersome. A proof of superior performance on a subset of a TPC-H may be countered by lack of functionality in many other parts. Or worse, a biased use of TPC-H to show how a new system excels or to invent one's own (micro) benchmark to prove a point thereby neglecting the often richer functionality of the systems compared with.

In this paper we describe a different approach to the problem. Consider two systems A and B, which may be different altogether or merely two versions of the same system. System B may be considered an overall better system, beating system A on all benchmarked TPC-H queries. This does not imply that there aren't queries that can not be handled more efficiently in A. These queries might simply not be part of the benchmark. Or the improvement is obtained in the restricted cases covered by the benchmark.

Therefore, the key questions to consider are "what queries perform relatively better on A"? and "what queries run relatively better on B"? Such queries give clues on the side-effects of new features or identify performance cliffs. We coin the term *discriminative benchmark queries* to describe such queries. For any pair of systems there may be just a few such queries, or there may be a large collection if the systems are targeted at different application domains.

Unfortunately, even for a fixed database schema and data distribution there is an infinite number of queries to consider. This puts the challenge on designing good heuristics to finding discriminative queries with minimal resources. Since we are primarily interested in finding performance pitfalls situations we assume that the database schema and data are available upfront. This can be a snapshot/sample of a real-world database. For research purposes it can be a TPC-H database at a given scale-factor.

In the remainder of this paper we introduce SQLsCALPEL, a tool to aid in finding performance pitfalls using discriminative queries, i.e. query pairs with a small edit-distance that exhibit substantially different performance behavior. To steer the process, we start with sample SQL queries that are automatically transformed into a large search space of related queries. In this way, we stay close to the intended application semantics. The system explores this space using a guided random walk to find the discriminative queries. This leads to the following contributions:

- We extend the state of the art in grammar based database performance benchmarking.
- We provide an algorithm to find query pitfalls as embodied by discriminative queries.
- We use simulated annealing to explore the practical infinite space of experiments.
- We deploy asynchronous experimentation on a cluster of target machines to reduce the time needed to perform the experiments.

SQLsCALPEL takes comparative performance assessment to a new level. It is inspired by a tradition in grammar based testing of software [8]. It assumes that the systems being compared understand more-or-less the same SQL dialect and that a large collection of similar queries can conveniently be described by a grammar. Minor differences in syntax are accommodated using *dialect* sections in the test suite.

One of the main problems is to cope with the combinatorial explosion of queries even in a simple grammatical description of the search space. For example, one of our driving test cases involved 43 sample queries, which brought together in a grammar leads to a search space of 10^{22} different queries. Evidently they

*Also with MonetDB Solutions.

DBTest'18, June 15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-5826-2/18/06…$15.00
https://doi.org/10.1145/3209950.3209951

can not reasonably all be checked, but also it is not clear if the 43 queries properly captures the performance profile of the system in all critical areas.

The second problem of performance measurement is scale. The performance on TPC-H SF-1 often has no predictive power for SF-100 exercised on the same machine. This means that a performance analysis tool should be built for long running experiments at multiple scales. An individual experiment may run for hours before reporting a result. This immediately leads to relying on a cluster of machines and asynchronous processing of the experiments.

Another challenge comes from the database system itself. Minor edits to a query formulation may trigger a different path to be taken in the optimizer. This leads to a rough performance landscape of seemingly identical queries, as shown by the Picasso project [5, 6]. This property makes a hill climbing tour through the search space cumbersome. However, it is also known that many queries have comparable performance, simply because they touch data with a similar data distribution, index support and data type. For example, the performance of `SELECT sum(C1) FROM T1` doesn't change much if we switch to a column `C2` of the same data type. This means that a controlled random picking of candidates queries becomes a valid option to locate the performance pitfalls.

A slight twist to the problem is to consider a single complex query from which we would like to isolate the critical performance components. In this case, a grammatical description of the query tells how it can sensibly be decomposed into its building blocks. Then SQLscalpel can run it against a dummy reference target to isolate the expensive predicates or sub-queries.

Outline. In the remainder of this paper we focus on the design of SQLscalpel. A short synopsis of the research background is given in Section 2. In Section 3 we give an architectural over view and introduce the query search spaces. The derivation process for new queries is presented in Section 4. Section 5 explains how we explore it to find discriminative queries. A preliminary functional evaluation is presented in Section 6.

2 BACKGROUND

Since the early days of database management systems, researchers have made it a standard practice to compare the performance of their solution to that of their peers using e.g. TPC-H. Its successor TPCH-DS [10] has been under development for over a decade and covers a wide range of application requirements. A decade where we have also seen a switch from traditional database applications towards NOSQL systems, main-memory systems, and massive distributed systems in the Cloud. Some recent developments on benchmark design, such as the LDBC benchmark [2], are geared at graph-like database performance evaluations. It can be used to analyse social networks. In this paper we merely take the benchmarks as a starting point to study their query space.

Grammar based testing has a long history in software engineering, in particular in compiler validation, but it also remained a small niche in database system testing. In grammar-based testing [8, 12] the predominant approach is to annotate a grammar with probabilistic weights on the productions. It is primarily used to generate test data geared at improved coverage tests for the target system, e.g. compiler [4], or to capture a user interaction with a web-based application. These approaches can be considered static and labor intensive, as they require the test engineer to provide weights and hints up front.

Another track pursued is based on genetic algorithms. A good example is the open-source project SQLsmith[1], which provides a tool to generate random SQL queries by directly reading the database schema from the target system. It has been reported to find a series of serious errors, but often using very long runs. Unlike randomized testing and genetic processing we guide the system through this search space by morphing the queries in a stepwise fashion.

Unlike work on compiler technology [9], the grammar based experimentation in the database arena is hindered by the relatively high cost of running a single experiment. Some preliminary work has been focused on generating test data with enhanced context-free grammars [7] or based on user defined constraints in the intermediate results [1]. A seminal work is [11], where massive stochastic testing of several SQL database systems was undertaken to improve their correctness.

The problem of large execution times can be alleviated somewhat nowadays, by using distributed experimentation platforms like ACTiCLOUD[2], where differentiation of database-as-a-service is one of the key targets.

3 SYSTEM OVERVIEW

In this section we provide an overview of the SQLscalpel architecture, the specification language for the query benchmarks, and the details to run them on a target platform.

3.1 System architecture

The SQLscalpel architecture is a straight-forward server/client approach.

SQLscalpel server This component is the heart of the system. It can be controlled from the command line and offers a web interface (See 5). A parser reads a specification file or a sample SQL query to initialize the project. A domain specific grammar and template queries are derived from them to form a pool of candidate experiments. The state of the workflow is kept in a MonetDB instance.

SQLscalpel client This component is run on each target system. It connects with the server to obtain outstanding (or unfinished) tasks. It translates this into a call to a local program to actually perform the task. The client program can be used to shield sensitive information, e.g. method and authentication information to actually run the experiment. Upon completion of the task, a REST call is made to the server to update the state. Each client is free to determine where and how the task should be executed. The predominant factor retained is the best-of time of a repetitive run of a single query against a target DBMS and a reference value for the result.

3.2 Query space

Query spaces are specified using a domain-specific language grammar G. All sentences in the language derived, i.e. $L(G)$, are candidate experiments to be ran against the target system(s). The sample grammar in Figure 2 illustrates a query space grammar with seven

[1] https://share.credativ.com/~ase/sqlsmith-talk.pdf
[2] http://www.acticloud.eu/

${Tag}	a sentence derived from Tag
${Tag}*	an optional repeated sentence
${Tag}+	a repeated sentence
$[Tag]	an optional sentence

Figure 1: Tag syntax and semantics

```
query:
   SELECT ${project} FROM ${l_tables} $[l_filter]
project:
   ${l_count}
   ${l_column} ${columnlist}*
l_tables:
   nation
columnlist:
   , ${l_column}
l_column:
   n_nationkey
   n_name
   n_regionkey
   n_comment
l_count:
   count(*)
l_filter:
   WHERE n_name= 'BRAZIL'
```

Figure 2: Sample SQLscalpels grammar

rules. Each grammar rule is identified by a name and contains a number of alternatives, i.e. free-format sentences with *embedded references* to other rules using an EBNF-like encoding (Table 1).

The SQLscalpel syntax is geared at a concise, readable description for the user. Internally, the grammar is normalized by making a clear distinction between rules producing lexical tokens, only governing alternative text snippets, and all others. called a literal classes in the sequel. Furthermore, the validity of the grammar is checked by looking for missing and non-used rules. They terminate the process.

Generation of concrete sentences from the grammar is implemented with a straight-forward recursive descend algorithm. This process stops when the parse tree only contains text and references to lexical tokens (e.g. `${l_column}`). They will form the templates for a final step, injection of tokens that embody predicates, expressions, and other text snippets to be derived for a concrete SQL query.

A naive interpretation of the grammar as a language generator easily leads to a (extreme) large set of queries. Especially, when a literal token would be defined using a value producing function, a.k.a. token generator, or when a recursive grammar rule is provided. To control this explosion somewhat we enforce that the literal tokens are used at most once in a query. This does not rule out that the same text snippet can be used multiple times. They are differentiated by their line number in the grammar. Furthermore, we ignore the different orders literal tokens may appear in a query. This heuristic is inspired by the observation that most query optimizers normalize the expressions internally first and look at terms as sets.

```
query:
  SELECT ${l_column_n} ${plist_n}* FROM nation
  SELECT ${cols} ${plist}* FROM nation,region \
         WHERE ${l_join}
plist_n:
  , ${l_column_n}
plist:
  , ${cols}
l_column_n:
  n_nationkey
  n_name
  n_regionkey
  n_comment
cols:
  r_regionkey
  ${l_column_n}
l_join:
   nation.n_regionkey = region.r_regionkey
```

Figure 3: Sample SQLscalpels join space

This way it suffice to count the lexical tokens during template generation and cap sentence expansion by taking this into account.

For example, the template `SELECT ${l_column} FROM ${tables}` is derived from the grammar above and can be finalized by choosing a table and a column literal. The number of query templates is limited compared to all possible queries. The grammar shown in Figure 2 leads to 10 templates, which can be used to generate 32 different queries.

3.3 SQL to SQLscalpel translation

A SQLscalpel grammar can be used to generalize existing benchmarks to cover a much larger spectrum of valid queries. For example, we can take TPC-H and morph the individual queries into a query grammar. This can be done in multiple ways. One extreme is to merely take the 22 queries of TPC-H as alternative literals under a single rule identifier. The other extreme is to break its parse tree and making all sub-trees optional as a basis for a SQLscalpel grammar.

We have implemented a SQL parser that turns a query into a SQLscalpel grammar. The heuristic is to split the query along project-list elements, table-expressions, and/or expressions, group-by and order-by terms. The remainder are considered literal tokens. The language derived encompasses all possible substructures. Of course, then dropping a table expression would leave the query semantically invalid.

A good practice for manual construction of a SQLscalpel grammar is to gradually increase its complexity . This way the explosion, and subsequent work, can be controlled. Furthermore, a prudent grammar construction can avoid erroneous queries to a large extend. For example, Figure 3 illustrates a grammar with a join term where the projection attributes only make sense in the context of `l_join`.

3.4 Running experiments

If SQLscalpel is used for a proof-of-concept project or to assess the performance of a given production database solution, we can not assume that the complete schema and database are handed over for experimentation. Instead, we should be happy to be given access to a test system, possibly loaded with a representative sample of the database. But even in this case, as little data as possible should be leaked. This is largely captured by the SQLscalpel client program, which grabs tasks from a task pool, executes them, and reports the findings back. It is under full control of the local DBA.

The client program requires the definition of one or more driver scripts, i.e. local programs to run an experiment. The user credentials, driver name and query id are centrally retained to make analysis and comparisons possible. Optionally, the user can release database and hardware product information for completion of the analysis reports. This information is not shared unless explicitly allowed.

A script per target system is the way to go. In particular, it controls how to set up the target system for an experiment, how to massage the performance results into a message understood by SQLscalpel, and how to cope with failing experiments. In most cases the environment should be reset to guarantee repeatable results, e.g. by flushing system caches. The experiment can be ran multiple times to obtain the best response time over a hot database. Some post processing is often needed to gather the performance date from the DBMS.

The expected output from each experiment is a simple JSON dictionary structure, e.g. {"system":"sf1", "tag":1, "time":43.5, "row":1242, "checksum":52814}. It should at least include the query tag, its runtime in millisecond precision, the number of rows in the result set count and a checksum over the result set for consistency analysis between the targets. The remainder of the JSON structure can be used to pass system specific key-value pairs for post analysis.

A complicating factor in running SQL experiments against multiple DBMS is that the SQL syntax understood may slightly differ. This is accommodated using literal class dialects. The class `${l_column}` can be extended to the dialect `${l_column@N}` where N is the name of a DBMS driver script. The sole requirement is that the order and number of elements in both classes are semantically identical. Although dialects solve most of the problems encountered, it can not solve all syntax/lexical differences. In that case the query grammar could be used to produce e.g. an abstract syntax tree and let the driver script convert it into the concrete syntax required by the target system.

4 MORPHING QUERIES

The query templates derived from the grammar provide a nice starting point of our search. Each template is a precise query up-to, but excluding final selection of the lexical tokens, i.e its parameters. Once we fix those with a collection of tokens into a concrete query task we can start a guided walk through the space by morphing the query. There are three morphing steps to consider: alter, expand and a prune strategy. This guided generation of concrete queries based on previous versions, allows us to attribute performance differences to specific changes in terms of the SQL text, in contrast to systems such as RAGS [11] that randomly generates queries.

Alter strategy. When we start SQLscalpel the query pool is empty, which means we can not morph a query already seen. The easiest way out is then to pick randomly a query template and then pick a random subset of all the literal classes mentioned. If, however, we can start with a randomly picked previously executed query, then one literal class is chosen at random and within that class we change one literal token. Since we collect a large number of queries in the pool over time, such an edit action may lead to a query that has already been executed. These are obviously ignored.

Expand strategy. The next strategy is to take a query from the pool and search for a template that is slightly larger. The metric here is that the number of literal classes is minimally expanded. It ensures also semantic cohesion of the queries. To illustrate, the query derived from template

```
SELECT ${l_column} FROM ${l_table}
```

can be expanded into a query satisfying

```
SELECT ${l_column}, ${l_column} FROM ${l_table}.
```

To complete the SQL query we need to add one more choice from `${l_column}`.

Prune strategy. The reverse operation for expanding a query is to search for the template with slightly fewer lexical classes. A prune strategy can be quite effective for queries with a lot of predicates. They are likely to produce small results and will be relatively fast to execute. With each pruning step the query becomes less complex, but probably also less selective. This increases the processing time. As such, a good starting position for a search could also be the most complex query. It is also the preferred method when the user wishes to identify the contribution of sub-queries.

Error handling. SQLscalpel is agnostic to the semantics imposed by the target systems. This means that an experimental run may report a syntax or semantic error. It is still kept in the query history as a basis for generating derived queries. The reason is that a morphing step can convert an erroneous query into a valid one.

5 QUERY SPACE EXPLORATION

Our prime target is to identify queries that separate two systems based on their relative performance differences. Consider for this a query Q exercised against both system A and B with timings $T_A(Q)$ and $T_B(Q)$. The performance ratio of such a single query does not provide much insight, except for the revelation that one system might be significant faster then the other. Surveying the ratios of many queries forming a benchmark set is already more informative to gain understanding the differences. It provides hints on what queries are relatively expensive/cheap. But, given the size of query space, it is not feasible to gather all ratios up front.

To understand the impact of queries we should learn what component in a query, e.g. sub-queries, expressions or predicates, is causing a major change in the ratio observed. It is already widely known that the performance space of $L(G)$ is ragged, but also that many queries have similar performance [5, 6]. For example, when keeping the selectivity factor and the data distribution constant for two identically typed columns, their performance is identical. Especially if we take the best result of a sequence of runs, because that would nullify the effect of start up cost and variance induced be

concurrent operating system load. With this information in mind we can reduce the complexity of the query grammar and have a heuristic to escaped a brute force evaluation of all queries in the space.

Without such background knowledge, we need to compare the ratio and structure of two almost identical queries. What we can do, however, is to take query instance Q and morph it with a small edit change into a query Q'. If we then compare the ratio $T_A(Q)$ / $T_B(Q)$ against $T_A(Q')$ / $T_B(Q')$ a statistical significant difference might become visible.

If the divergence is equal to 1.0 we can conclude that both systems are equally affected by the morphing to Q'. This may be relatively good or bad, but does not provide much further insight. However, if the ratio becomes greater then 1.0 it indicates that the performance of system A is relatively better then B. Conversely, if the ratio is less then 1.0 then system B is relatively better.

In all cases not equal to 1.0 we may conclude that the difference is likely to be attributed to the edits made. We have found a *discriminative query*. This gives clues on where a system developer should invest his time. Beware, a divergence not necessary indicates a malfunctioning, it may also show that a system hits a resource limit, e.g. flushing intermediates to a slow disk instead of keeping them in RAM cache. Or, that some supportive data structure, e.g. a join index, can be used to speed up execution which couldn't be used before.

Definition 5.1. The divergence between two queries Q and Q' on system A and B is defined as:
$(T_A(Q')$ / $T_B(Q'))$ / $(T_A(Q)$ / $T_B(Q))$.

5.1 Simulated annealing

The baseline strategy is to start with a random query from the space of alternatives. The SQLscalpel specification gives you an easy handle on this task. We merely pick a template, a collection of lexical tokens from each of the sub-languages and glue them together to create a syntactically valid SQL query.

After the query has been executed we know the performance ratio between the two systems. Subsequently, we apply the morphing steps to search for more prominent differences using a randomized selection of tokens needed and execute the queries. For example, let $\{A, B, C\}$ be the tokens chosen in a query task Q_0 and we are left with a collection of non-used tokens $\{D, E\}$. We can derived 6 direct morphing steps by replacing a single token. Without further knowledge, each of these replacements may tip the point of divergence.

If, however, we know that the task Q_0 is derived from another task already performed $Q_1(\{A, B\})$ and $Q_2(\{A, C\})$, respectively. Can we then concluded that B and C are 'identical' in terms of divergence? e.g. considering only the result of replacing B with D. Probably not. From the SQLscalpel perspective they are simple sentences without semantics. The underlying cause is not known. {D} can represent a data source with a distribution that has different effect on {A,C} and {A,B}. It also can represent an operator with completely different complexity. The challenge then turns into a guided walk where best guesses, hopefully, lead to insights as quickly as possible.

Such a strategy is known as simulated annealing and heavily used in cost-based query optimizers [3]. The cost metric to decide what direction to take in the space of alternatives is the performance ratio. The number of queries considered during a simulated annealing step is bounded by a beam size, which caps the number of new experiments. Furthermore, at every step we take the top N of prospective discriminative queries as the starting point to morph them into new queries. With an increasing part of the space being explored, a trail of morphing steps ends in a local minimum. To escape this, we can continue picking a random query and restart the process.

6 EVALUATION

SQLscalpel is a prototype under active development. In this section we report on our initial experiences using it. The purpose was not to engage into a deep performance analysis of specific target systems, but on assessment of the design and heuristics itself to steer us into the right direction.

6.1 The platform

scalpel is built as a web-based software platform for developing, management, and sharing experimental results. The GUI is built around Python and the Flask and Bokeh libraries, which already provides a rudimentary set of visual data analytics functions.[3] Figure 4 illustrates the interface after running experiments on TPC-H query 1. It shows the execution time of a system for a series of queries. The dashed lines between the queries illustrate the morphing action taken. The color coding for {alter, expand, prune} morphing is {purple, green, blue}. Queries that result in an error, i.e. are semantically incorrect, are shown as yellow dots. Note that they can be morphed into valid queries later on. The node size illustrates the number of components in the query. Hovering over a node shows the details of the experiment.

Running experiments against MonetDB, the user would quickly notice the dominant term in Q1:

```
sum(l_extendedprice*(1 - l_discount)*(1 + l_tax))
            as sum_charge
```

as shown in Figure 4. It is by far the most expensive component. The underlying reason stems from the way MonetDB evaluates such expressions, which include type casts to guard against overflow and creation of fully materialized intermediates. The component breakdown illustrates this more clearly as shown in Figure 5.

6.2 TPC-H

The TPC-H benchmark was revisited to assess how large the search space becomes when the SQL queries are converted automatically into a SQLscalpel grammar.The results are shown in Figure 6. We can observe a wide variation of sizes. Nested queries with compound predicates, e.g. Q2, Q7, Q19, leads to an explosion of alternatives. This is to be expected, because the grammar produced contains sets of literal classes from which all subsets are considered for template construction. This results in a combinatorial explosion of templates. They form a nice target to study robustness of relational optimizers.

[3] Space limitations prohibits an explanation of all its functional components

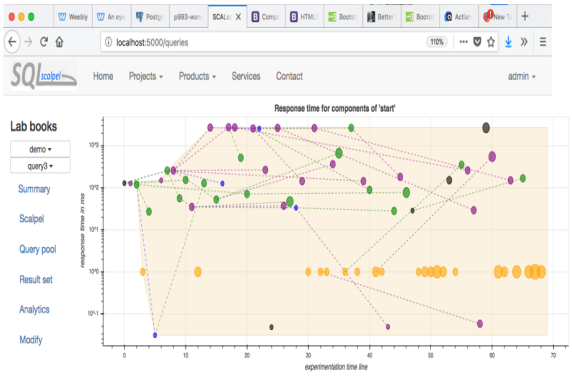

Figure 4: Query space provenance

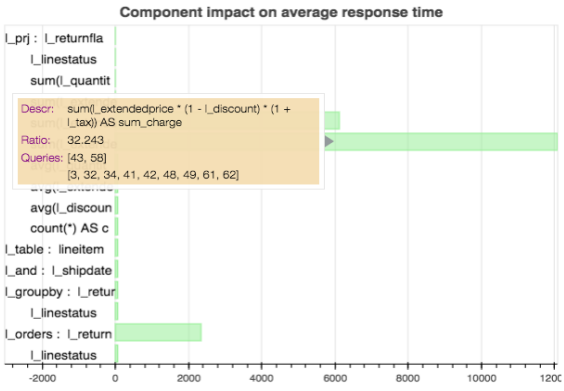

Figure 5: Component analysis

tag	templates	space	tag	templates	space
Q1	40	9207	Q12	8484	162918
Q2	58160	6354837405	Q13	16	81
Q3	240	29295	Q14	6	21
Q4	28	81	Q15	40	372
Q5	108	96579	Q16	608	25515
Q6	4	15	Q17	26	81
Q7	>100K	–	Q18	576	43659
Q8	480	5478165	Q19	>100K	–
Q9	1512	3528441	Q20	320	3339.0
Q10	384	722925	Q21	18464	4255065
Q11	162	7203	Q22	156	777

Figure 6: TPC-H query space

To control such explosion the user can take the scalpel grammar and fuse rules (manually) to reduce the search size. Alternatively, the SQL to SQLscalpel compiler can be instructed to limit the size of lexical token classes considered or focus on conjunctive predicates only.

6.3 The real world

Another driving example for SQLscalpel was a problematic customer query. It was a typical BI query, generated by a front-end tool and heavily relying on expanding (inline) views. The result was a single SQL statement of ~2K lines (~ 110 K characters), which included 68 subqueries, > 100 (dis/con)junctive terms and much of the SQL functionality, i.e. group-by, order-by, case-expressions, window-based aggregates, casting, etc.

The query complexity is large enough to make it hard for a human to identify what portion is detrimental to the performance. The SQLscalpel parser inferred a grammar with 992 rules, 685 literals, and >100K query templates. The complexity of this query drove query morphing towards *pruning*, where a binary dissection of the query into its components should provide the answer a.s.a.p.

7 SUMMARY AND CONCLUSIONS

In this paper we have introduced *discriminative performance benchmarking* as a next major step in database performance analysis. We provided a progress report on the SQLscalpel project, a sharp tool in the hands of system architects. It allows them to describe in a concise manner a large query workspace and rely on guided randomized walks to localize performance pitfalls quickly.

Acknowledgments

This research has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement no. 732366 (ACTiCLOUD).

REFERENCES

[1] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. QAGen: Generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 341–352, New York, NY, USA, 2007. ACM.

[2] Orri Erling et al. The ldbc social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 619–630, New York, NY, USA, 2015. ACM.

[3] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[4] Hai-Feng Guo and Zongyan Qiu. Automatic grammar-based test generation. In Hüsnü Yenigün, Cemal Yilmaz, and Andreas Ulrich, editors, *Testing Software and Systems*, pages 17–32, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[5] Jayant R. Haritsa. The picasso database query optimizer visualizer. *PVLDB*, 3(2):1517–1520, 2010.

[6] Jayant R. Haritsa. Query optimizer plan diagrams: Production, reduction and applications. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 1374–1377. IEEE Computer Society, 2011.

[7] Johannes Härtel, Lukas Härtel, and Ralf Lämmel. Test-data generation for xtext. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering*, pages 342–351, Cham, 2014. Springer International Publishing.

[8] Ralf Lämmel and Wolfram Schulte. Controllable combinatorial coverage in grammar-based testing. In M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko, editors, *Testing of Communicating Systems*, pages 19–38, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[9] William M. McKeeman. Differential testing for software. *DIGITAL TECHNICAL JOURNAL*, 10(1):100–107, 1998.

[10] Meikel Poess, Tilmann Rabl, and Hans-Arno Jacobsen. Analysis of tpc-ds: The first standard benchmark for sql-based big data systems. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 573–585, New York, NY, USA, 2017. ACM.

[11] Donald R. Slutz. Massive stochastic testing of SQL. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 618–622. Morgan Kaufmann, 1998.

[12] Kamal Z. Zamli, Mohammad F.J. Klaib, Mohammed I. Younis, Nor Ashidi Mat Isa, and Rusli Abdullah. Design and implementation of a t-way test data generation strategy with automated execution tool support. *Information Sciences*, 181(9):1741 – 1758, 2011.