Bridging the Chasm between Science and Reality

Martin Kersten, Panagiotis Koutsourakis, Niels Nes, Ying Zhang
MonetDB Solutions
Amsterdam, The Netherlands
<firstname>.<lastname>@monetdbsolutions.com

ABSTRACT

When a research-prototype of a database management system becomes a product in the market, then more insight is required from the actual workloads to steer its industrial hardening. Unfortunately, few customers are willing to share their database schema, data samples, query load and execution traces for business and legal reasons. The technical challenge then becomes, how to enable customers to share their workloads in such a way that it will not leak sensitive information, while still provide sufficient information to allow DBMS researchers and developers to assess and improve their technology.

In this paper, we report on ongoing research to address this challenge in the context of MonetDB[1], as it is increasingly adopted in the enterprise market. This paper sheds light on the importance of a good profiling tool during system construction and the techniques deployed to gain permission from customers' legal departments to share profiling traces captured on live production systems.

1. INTRODUCTION

With over a hundred relational database management systems (DBMS) listed in the DB-Engines Ranking [6], one might think that the space of solutions is well covered. However, the functionality, performance, stability and maturity of these systems differ widely. Many more DBMS prototypes and enhancements among the open-source offerings are fighting for their places in this pack or geting credits from the science community.

In all cases, DBMS researchers and developers are facing an insurmountable problem, namely what one should optimize for. It is either impact of new hardware trends or the requirements posed by new application domains. In all cases, the lifeline for decades has been to fall back on standard (industrial) benchmarks, such as the TPC benchmark suite, to showcase the innovations in perspective to the competitions. A "short cut" is often taken through the already limited benchmark use cases to highlight only those aspects covered by the proposed scientific innovation.

The game changes as soon as a system reaches the maturity level of being used as a product in the market. Then (industrial) benchmarks are not necessarily the best representation of what is required, as has often been elaborated upon in the major DBMS conferences and, in particular, the ACM DBTest workshops series [13, 4, 11]. For example, [15] reports on the transition of Hyper into the eco-system of Tableau, in which databases are as small as a few megabytes, users encode too much information in strings, and most queries are simple.

The MonetDB team has also been facing similar challenges. We would love it if a customer could share with us their schema, data and queries to study. This would allow us to assess the impact of the choices we make during the design and implementation of the DBMS on their applications. Even further, we would like to have the full knowledge of when and how concurrent database users interact with the DBMS to understand a full production system. Evidently, this is hardly ever possible for obvious business and legal reasons.

This problem became more prominent when MonetDB has been chosen as the backbone analytics engine by a cloud provider for the enterprise market. The workload of thousands of applications running on MonetDB is shielded from our eyes while we are charged to improve MonetDB further.

Instead of requesting samples of the workload of a production system and vendor benchmarks, we focus on what is needed for our analysis under the boundary condition that no sensitive information about the application can be given. This analysis involves understanding what corners of the DBMS cause performance degradations, what statistical properties can be derived for improved algorithmic choices, what is the memory footprint of a query including its intermediates, do we exploit all possible parallelism, which concurrent transactions are we dealing with, etc.

The approach taken is based on the performance monitoring tool Stethoscope [12], which can be attached at any moment in time to a running MonetDB database server to capture details on what is actually happening in the database kernel. The output is a detailed trace of all queries being executed during the monitoring period, including for each algebraic operator its system resource consumption, the type and size of its input and output, the algorithm and indices used, execution time, etc.

The Stethoscope traces help DBMS researchers to advance their technology innovations, application developers and DBAs to make more efficient use of the DBMS, and cloud providers to make better decisions about resource provisioning. However, those traces can contain sensitive data. When applied to production systems, the research challenge is *how to remove all sensitive data while retaining sufficient information to allow the Stethoscope users to gain valuable insights*.

The *contribution* of this paper is twofold. First, we present the

[1] https://www.monetdb.org

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2021.
11th Annual Conference on Innovative Data Systems Research (CIDR '21) January 10-13, 2021, Chaminade, USA.

obfuscation technique "phase spaces" adopted in Stethoscope, with which we have been able to both sufficiently blur the production data such that the trace is approved to be shared with us by the legal department of our customer, while retaining all the important properties of the data and queries. Then, we discuss the results of an in-depth analysis of the traces conducted using our other in-house tool "Holmes": how different is a real-life application than the standard benchmarks, which lessons can we learn from these traces, and which inspiration can we draw from them to guide the further innovative development of an analytical DBMS.

The approach taken is incremental, seeking how we can bridge the chasm between the database developers and the legal department, who has to protect the sensitive data. Although the paper is centered around MonetDB and its tools, it should inspire DBMS researchers to think along this line in an early phase of their system development. This is a major step beyond the query monitoring information gathered in most database systems.

The paper is further organised as follows. In Section 2 we give a quick overview of some inner working of the MonetDB DBMS and its Stethoscope profiler. In Section 3, we describe the obfuscation techniques adopted in Stethoscope. In Section 4, we present the results of an in-depth analysis of the Stethoscope traces obtained from a live data warehouse application, which include both our findings and future outlook. Finally, we discuss related work in Section 5 before concluding in Section 6.

2. DATABASE PROFILING

The context of our work is the MonetDB database management system[5, 9]. It is open-source, has been under active development and deployment since 1993, and has been recognized for its innovations [1, 14].

2.1 Query execution plans

MonetDB is a column-store like the ones we have seen emerging over the last decade, with an emphasis on lean data structures, in-memory processing, and exploitation of multi-core architectures. However, it also takes an offbeat approach to query processing, which is best illustrated with a small example. Consider the following query:

```
SELECT count(*) FROM gym.agenda
WHERE visit BETWEEN '2020-08-02' AND '2020-08-05'
  AND weight >100
```

When users request the execution plan for this query, they will get something similar to what is shown in Figure 1. It is not an annotated operator tree of an execution pipeline as found in many database engines, but a program is written in the MonetDB Assembler Language (MAL)[2]. This abstract machine language is interpreted and supported by thousands of highly optimized and specialized algorithms.

The execution plan of this SQL query is a series of MAL statements, which can be divided into three main blocks. In the first block, we locate the storage location of the columns (ln. 4–7). Each MAL variable is annotated with its type (here we only show a few for reasons of readability). The annotation `:bat` indicates an array instead of a scalar value. The second block contains the relational operators to compute the query (ln. 8–10). It is what most database system will expose to the user. Finally, in the third block, we construct the result set of the query (ln. 12). Each MAL statement is executed by a single worker thread to completion. A `barrier...exit` block (ln. 3–11) indicates statements that can be executed in parallel in a dataflow manner. In Section 4.2 we will discuss in more details how the partial results of parallel executions are combined into the final result set.

The approach to materialize all intermediate results in MonetDB comes at a cost. Often, the memory footprint is larger than an equivalent Volcano-style interpreter [8]. The benefits, however, outweigh this memory waste, because each relational operator will be given the complete operands to work with. This way, we can not only exploit the highly efficient positional array lookups but also use the runtime properties of the arguments to pick the optimal algorithm just-in-time, e.g. merge-join versus hash-join. This approach simplifies the work of a query optimizer, because it does not have to guess, instead it can inspect the actual arguments to make a locally optimal decision.

MonetDB uses dynamically sized vectors to break up large tables into chunks. Each chunk comfortably fits in memory and avoids resource competition. The net result is that a database portion is horizontally split up dpending on the needs and resource availability at no cost. It leads to better use of the multi-core platforms.

2.2 Profiling a database kernel

Profiling the performance of a database kernel is an essential step in its development. A common approach is to keep track of the queries being executed and a number of performance counters in a (light-weight) system table. The DBMS users can inspect those for post-execution analysis. A drawback of this approach is that a query running amok will not be noticed until it has ceased to exist.

The key performance monitoring tool for MonetDB is *Stethoscope* [3]. It works like the medical instrument that lends it, its name, i.e. it can be attached at any moment to a running MonetDB server to listen in on what is going on deep inside the system. One of the benefits of Stethoscope is that it allows us to see if a query is running amok, e.g. because it causes the computation of unnecessarily complex joins. The query can subsequently be safely aborted.

Once connected to a MonetDB server, Stethoscope will spit out profiling events from the MAL interpreter over an IO stream. This way, we can avoid the overhead incurred by keeping internal performance tables up to date. Subsequent trace analysis is detached from the DBMS kernel.

Two events are produced for each MAL instruction at its 'start' and 'done' times. Stethoscope has an overhead of ~30 microseconds per instruction. As a columnar database, each MAL instruction operates on columns or large chunks. The MAL execution trace generally has 10s to a few 100s of events. The profiling overhead only occurs during the lifetime of stethoscope being active. It can be attached for a short time on a regular or a need to know basis.

Figure 2 illustrates a JSON object Stethoscope produced when the `aggr.count` operator in Figure 1 is 'done' (ln 9)[4]. Lines 1–4 uniquely identify this JSON object, i.e. to which MonetDB server `version` and `session`, which database `user` and which MAL `program` it belongs. Lines 5–7 contain information of system times and the worker thread. Lines 8–9 identify the operator `aggr.count`, and its `state` and execution time in `usec`. Finally, the `args` array contains information about all input and output variables of this function (ln. 10–20). For a `:bat` variable (i.e. an array), we also know if it is persistent, sorted, dense, etc.

These traces are essential for gaining performance insights. We can not only find the bottleneck instructions in a query, but also

[2]https://www.monetdb.org/Documentation/Manuals/MonetDB/MALreference

[3]https://www.monetdb.org/blog/pystethoscope

[4]Full documentation: monetdb-pystethoscope.readthedocs.io

```
1  function user.s40():void;
2    querylog.define("select count(*) from gym.agenda
                      where visit between '2020-08-02' and '2020-08-05' and weight >100");
3    barrier X106 := language.dataflow();
4      X7 := sql.mvc();
5      X25:bat[:int] := sql.bind(X7, "gym", "agenda", "weight", 0);
6      X20:bat[:date] := sql.bind(X7, "gym", "agenda", "visit", 0);
7      C8:bat[:oid] := sql.tid(X7, "gym", "agenda");
8      C35:bat[:oid] := algebra.select(X20:bat[:date], C8:bat[:oid], "2020-08-02", "2020-08-05", true,true,false,true);
9      C41:bat[:oid] := algebra.thetaselect(X25:bat[:int], C35:bat[:oid], 100, ">");
10     X45:lng := aggr.count(C41:bat[:oid]);
11   exit X106;
12   sql.resultSet("gym.\%1", "\%1", "bigint", 64, 0, 7, X45);
13 end user.s40;
```

Figure 1: A simplified MAL execution plan of the example query in Section 2.1

.

```
1  {"version": "11.37.7",
2   "session": "b7ed4557-b202-4116-94d0-44163771b271",
3   "user": 3,
4   "program": "user.s40",
5   "clk": 1597681144673082,
6   "mclk": 2138257488,
7   "thread": 4, "pc": 17, "tag": 37150,
8   "module": "aggr", "function": "count",
9   "state": "done", "usec": 43,
10  "args": [
11   {"ret": 0, "var": "X45", "type": "lng",
12    "const": 0, "value": "11", "eol": 137, "used": 1,
13    "fixed": 1, "udf": 0},
14   {"arg": 1, "var": "C41",
15    "alias": "gym.agenda.weight", "type": "bat[:oid]",
16    "persistence": "transient", "sorted": 1,
17    "revsorted": 0, "nonil": 1, "nil": 0, "key": 1,
18    "file": "tmp_1112316", "bid": 300238, "count": 11,
19    "size": 88, "eol": 127, "used": 1, "fixed": 1,
20    "udf": 0} ]}
```

Figure 2: an example Stethoscope JSON object

deduce if the parallelism has been applied to the max. Ideally, we would like customers to use Stethoscope and share the traces with us for analysis. The catch is that traces can contain sensitive information.

3. EXECUTION TRACE OBFUSCATION

The needs for data sharing and privacy often conflict [2]. A key concern is that even snippets of information shared can be framed to an individual or disclose confidential business data. Aside from access control schemes, the predominant technique to tackle this problem is based on data obfuscation by *data masking*, a.k.a. data scrambling and data anonymization. It is the process of replacing sensitive information copied from production databases to a test system with realistic, but scrubbed, data based on masking rules. It is also a concern when the user is free to formulate queries to engage in a *statistical tracker* [7] to breach data security.

Data protection is also relevant for database system architects beyond its evident strive for data privacy[3]. They need detailed information about the query execution to understand what is happening and how the engine can be tweaked further. However, a naive execution log may leak sensitive information.

A significant difference with data masking of result sets is that 1) the system architect has no access to the production system to issue a statistical tracker query sequence, 2) the system architect is not exposed to any result set what-so-ever, 3) the query arguments in a trace is not necessarily part of the protected database, and 4) the analysis is performed off-line using execution traces approved by the DBA.

We started out using a simple masking strategy by replacing every literal value in a MAL program with asterisks. This has the advantage that no information from the database is leaked whatsoever, while it already allows us to gain valuable insights (See Section 4).

However, one of the main drawbacks of this simplistic and highly restrictive approach is that we lose sight of some critical information for the understanding of the inner workings of MonetDB. Consider an algorithmic choice that depends on the selectivity estimation derived from the predicates of a query; in such cases, a masked value does not reveal any useful information. Therefore, a more sophisticated approach is needed.

Hiding schema information When obfuscating a database, there are two dimensions to consider: schema- and data- obfuscation. For a given query, the combination of customer identity, schema naming and column names are the first point to leak information. They are also the easiest to mask. We can simply replace the *schema.table.column* with randomly generated identifiers or words picked from a dictionary to elicit readability.

Hiding operator arguments Obfuscation of the data values in a trace is a little more involved because the obfuscated variant should still reveal the domain properties. The approach taken is inspired by *phase spaces* in mathematics and physics[5]. In physics, for example, a phase space transformation maps Euclidean coordinates to some other generalized coordinate system. Such a transformation makes it possible to retain the structure of the problem at hand.

The key insight here is that we do not need the actual data, but several properties that are computed by the engine at the execution time. Essential properties are, e.g., the storage type, domain order, sort order, number of unique values, min and max values and the data distribution. Such *phase space* mappings can be readily encoded in the Stethoscope to provide an acceptable level of masking.

4. CROSSING THE BRIDGE

MonetDB was built as an open-source research project and focused on developing technology for data analytics. An implicit assumption in this research area is that most queries would be complex and time- and resource-consuming. Moreover, the concurrent workload is considered limited to just a few power users. So, how different is this from the real-world?

[5]https://en.wikipedia.org/wiki/Phase_space

MonetDB version	11.37.8	
Total number of queries	5573	(4851 r. / 722 w.)
Number of unique queries	498	
No. times a query is reused	1646	(max)
	294	(95th percentile)
Number of unique schemas	3	
Number of unique tables	48	
Number of unique columns	1835	
No. times a column is reused	5167	(max)
	73	(avg)
	84	(95th percentile)
Query response times (msec)	0.06	(min)
	253.25	(avg)
	24452.27	(max)
	50.96	(90th percentile)
	1313.63	(95th percentile)
Footprint persistent data	25.2	(GB)
Footprint transient data	450	(GB)
Number of concurrent clients	603	
Peak concurrent queries	24	

Figure 3: General statistics of the customer application

Recently, we received some Stethoscope traces from one of our customers. The trace considered here covers a 5-minute log of one of their most active production systems using MonetDB. Next to the traces, we have no information about the application other than that it is a typical data warehouse application with bulk updates and concurrent user queries.

Subsequently, we analysed the traces with an in-house tool, called *Holmes*, to extract statistics and query patterns, run a simulation of the queries to assess the impact of different scheduling strategies on a multi-core machine, and locate common query patterns as the basis for automated multi-query optimization. In short, an effective tool in the hands of a DBMS architect.

In the remainder of this section, we report on our findings, a few 'war'-stories and some lessons learned.

4.1 Looking through a magnifying glass

Running the traces through Holmes gives us some interesting statistics of the application as shown in Figure 3.

Query mix The workload is a mix of 4851 read-only queries and 722 updates. The rule of thumb in DBMS research to focus on an 85-15 read/write mix seems to apply here.

Furthermore, the query response times show that most queries are very short running, e.g. 90% of them took only ~50 milliseconds. This is somewhat surprising, and we will address it more in Section 4.2.

Common queries In a data analytics production setting, the front-end application is expected to support a limited number of query patterns. This is confirmed by the "number of unique queries" in Figure 3, i.e. only 489 out of >5.5K queries are unique. Consequently, most of the 1.8K unique columns were reused many times which increase the memory footprint.

This statistic strongly indicates that the *recycler* technique [10], which caches partial intermediates to speed up processing significantly, and which we have successfully applied on an astronomy data set in the past, can be beneficial in this type of commercial applications as well.

Storage footprint Temporary tables and intermediates took roughly 20 times more space than the persistent data (i.e. "footprint transient data" vs "footprint persistent data"), which is a lot more than we have expected. However, from the >600 users who were logged-in in MonetDB during the monitoring period, there are only 24 concurrent queries at the maximum (i.e. "peak concurent queries"). For a cloud provider, this information is extremely valuable for resource provisioning. We will come back to this topic in Section 4.3.

Operator statistics Developing a relational database kernel for analytics is commonly driven by running the TPC-H benchmark. To measure progress, we keep a summary of where the time spent on solving all 22 queries. Figure 4(a) shows the top 10 most expensive MAL instructions for SF-10[6] in the MonetDB Jun2020 release: the number of times they are called, their total execution time in microseconds and the percentage of time they took in the execution of the whole query set. Together, these operators consumed more than 80% of the total execution time. The list is dominated by the relational operators join, grouping and selection. For years, such overviews have been pivotal in improving MonetDB with more efficient operator implementations. It has brought international recognitions to MonetDB as a leading open-source columnar DBMS [1, 14]. However, the big question is if this workload summary reflects what the market is calling for.

Figure 4(b) shows the top 10 most expensive MAL instructions in the Stethoscope traces on our client trace. The pregnant observation is that the distribution pattern of where the time was spent in the MonetDB kernel is completely off from our TPC-H workload. Overall, the reliance on the relational operators join, grouping and project is way down the list.

The `intersect` operator is noteworthy. It is regularly used to perform the intersection between two lists of OIDs, which represent the positions of the tuples in a table. Under the hood, it is implemented using a variant of the join algorithm. The "ttl. time" of this operator here demonstrates a case missing in its implementation. We could also have noticed this in the TPC-H overview, but it has been pushed below the radar by more expensive operators[7]. Congrats to Holmes for finding this algorithmic glitch.

4.2 Uncover clues

Optimizer improvements The MonetDB optimizer consists of two parts. The first part manipulates the SQL query tree and derives a logical query plan. It only uses semantic rewriting rules without any statistics derived from the database to arrive at a more efficient plan to process.

The second part consists of a series of plan rewriters, inspired by compiler technology and organised in a linear pipeline. Each rewriter takes the resulting MAL plan from its predecessor as its input and produces another MAL plan, in which the MAL plan snippets are rephrased, e.g. constant expression evaluation and common expression elimination. Together, the rewriters morph a logical query plan into a physical execution plan. In this pipeline two steps are important to consider: *Mitosis* and *Mergetables*.

The Mitosis rewriter locates the largest table referenced in the query and applies a zero-cost horizontal partitioning to separate this table into multiple *logical* merge tables. Then the subsequent relational operators are applied as much as possible on the partitions in parallel. Each partition should fit comfortably in memory so as not to cause contention over memory resources with concurrent worker threads. After all parallel computations are finished, the result partitions are glued back together by `mat.packIncrement`, a blocking operator.

[6]Operators in this list are fairly stable for other scale factors because the TPC-H data set scales nicely up and down.

[7]In TPC-H, the total cost of the `intersect` operator is even outside the top 20.

no.	command	#calls	ttl. time (μs)	percentage
1	algebra.join	249	16391781	26.23%
2	aggr.subavg	20	7698648	12.32%
3	batcalc.*	239	6022505	9.64%
4	algebra.projectionpath	374	5146479	8.24%
5	algebra.projection	825	4929295	7.89%
6	aggr.subsum	93	3883472	6.21%
7	algebra.select	105	3258274	5.21%
8	algebra.thetaselect	205	2737517	4.38%
9	group.groupdone	59	2042968	3.27%
10	group.subgroupdone	36	1978706	3.17%
				sum: 86.56%

(a) TPC-H

command	#calls	ttl. time (μs)	percentage
mat.packIncrement	90684	919527662	52.07%
algebra.intersect	704	533178902	30.19%
algebra.thetaselect	7539	88713655	5.02%
sql.delta	529	65079102	3.69%
algebra.join	862	45180010	2.56%
algebra.select	206	30995947	1.76%
algebra.likeselect	313	26616183	1.51%
algebra.projection	41400	12314027	0.70%
sql.projectdelta	181	11160017	0.63%
sql.append	75961	6180143	0.35%
			sum: 99.48%

(b) customer application

Figure 4: Top 10 most expensive relational operators of TPC-H vs. customer application

As an example, the MAL plan below (simplified for readability) shows that Mitosis has split the column `gym.agenda.weight` used by our example query in Section 2.1 into two partitions, each containing 1000 tuples and starting from positions 0 and 1000, respectively:

```
X162:= sql.bind('gym','agenda','weight', 0, 1000 )
X163:= sql.bind('gym','agenda','weight', 1000, 1000 )
X268:= mat.packIncrement(X162,X163)
...
C134:= algebra.thetaselect(X268, '>', 14);
```

Then, when the Mergetables rewriter comes in, it should take the partitioned table and pushes the `thetaselect` onto each partition, and modify the `mat.packIncrement` statement to glue together the results of the `thetaselect`-s instead. This should lead to a rewritten MAL plan similar to the following:

```
X162:= sql.bind('gym','agenda','weight', 0, 1000)
X163:= sql.bind('gym','agenda','weight', 1000, 1000)
C200:= algebra.thetaselect(X162, '>', 14);
C201:= algebra.thetaselect(X163, '>', 14);
C268:= mat.packIncrement(C200,C201)
```

At this point, we have prepared a MAL plan for dataflow driven parallel processing with two worker threads. Note that the speed-up in the rewritten plan is twofold: the `thetaselect`-s are computed in parallel and `mat.packIncrement` often has significantly fewer data to process.

However, the analysis of the customer traces illustrates an exorbitant processing time in `mat.packIncrement`. A drill down into the MAL programs quickly revealed that the Mergetables rewriter bailed out in the middle of the process, leaving behind a potentially very expensive MAL plan, because it might unnecessarily first reconstruct a big persistent column and then apply selections on the whole column. Holmes helped us to uncover an important clue to achieve some considerable improvements.

Parallel processing The parallel processing of analytical queries is the prime target for most database architects. Whether aimed at a multi-core modern processor or a distribute cloud setting, it should lead to a significant performance boost. The public TPC-H benchmarks stress this point to the extreme.

MonetDB uses worker threads to process the queries in parallel. The level of parallelism achieved strongly depends on the opportunities offered by the queries. For a full-table scan, linear speed-up with the number of cores can often be achieved. However, simple queries and blocking operators in the plan jeopardize the potential gain. In a multi-user system, the remaining resources can be used for concurrent queries.

The reality check against the customer log was again surprising. Although MonetDB was mainly put to work on the data analytics queries for the customer, it appears that only 208 out of the 5.5K queries (i.e. <5%) used parallel processing. What is going on here?

Further assessment confirmed that most of the queries are indeed simple queries, such as point queries on small tables, and therefore, applying inter-parallel execution would not be beneficial. This calls for a parallel code optimizer which would simplify the execution plan into a sequential pipeline, thus avoiding all the overhead that comes with parallel task scheduling. Furthermore, in the cloud market (since our customer is a cloud vendor), a parallel processing analysis would be particularly useful for the resource provisioning decisions often need to be made. Holmes showed that reality may favor a workload quite the opposite to what its design is aimed at.

Copy-into stability The customer revealed that there is a front-end application which regularly ships update batches. This is reflected in the 258 `COPY...FROM` statements. A rumor was a perceived update instability in this bulk-update process. The analysis of the trace confirmed 27 outliers (of ~0.3 sec). It may well be explained by resource fights over CPU cores caused by these updates, even though the update sizes are less than 1000 tuples each. All cores are assigned to work in parallel for analytical query processing. However, a `COPY...FROM` operation also tries to grab all cores to speed up the process. This resource contention is relatively easy to fix. Holmes, thanks for reporting, we were a little too enthusiastic in assigning CPU cores to tasks.

4.3 Future potentials

What-if scenarios Having a customer trace makes it relatively easy to study some *what-if* scenarios, such as what is the impact on query execution time of reducing the number of worker threads, and what happens if we trim the high-water mark of memory consumption.

This is facilitated by a simple simulator for the MAL interpreter. The task scheduler can be replaced to study the effect of increasing/reducing the number of worker threads. It can also be used as an alternative for a cost-based optimizer because it has precise reference data of the duration of the relational operators.

Holmes is going to speed-up innovations at the DBMS kernel level by exposing information hitherto not accessible to us.

Resource summary The obfuscated trace can be used to infer a 'storage schema', which elicits all tables/columns with their properties such as type, size, uniqueness, sort order, and indices to support fast access. It will not reveal any actual content.

Since MonetDB materializes the intermediates, they consume space as well. The storage schema provides a good indication of

the amount of RAM needed to keep the persistent tables in-memory for the duration of a DBMS session. Especially in applications with many complex queries to process, this can lead to a significant increase in the storage footprint. The "footprint transient data" in Figure 3 shows the maximum storage needed to keep intermediates of *all* concurrent queries around in RAM or SSD. Both are valuable hints for a cloud provider to consider their resource provisioning and a target for database kernel developers.

5. RELATED WORK

This work is related to operating and database system monitoring. Most database systems have a method to emit both the query execution plan and its timing. The major difference from our approach is that it is often a graphical tool, which hides all information except for the prime relational algebra operators, the cardinalities involved, the algorithms chosen and the timing. The lack of obfuscation means it can only be used by the DBA.

This work is also related to workload analysis. Workload analysers have been around for a long time. They typically gather (long-running) queries from a production system and simulate those against variants of the database. Their main objective is to find a sweet spot in query processing using auxiliary structures, such as indices and materialized views. These workload analyzers are tools in the hand of a DBA because they often do not obfuscate. It is not needed to obfuscate because workload re-runs are the focus. In our case, Stethoscope and Holmes are the tools in the hand of a performance expert on MonetDB. They provide fine-grained information on the actual relational operators involved.

The authors of Hyper have also stumbled on the problems sketched in their papers [13, 15]. However, in their case, the application emphasis is on preparing data for visualisation, where techniques to improve processing hundreds of GBs fell short when confronted with simple queries over just several MBs of data. In their case, the publicly available dashboards can be used to gain insight into the real-world requirements. It has resulted in the Public-BI benchmark[8]. Our use-case is closer to a more traditional data warehouse setting with hundreds of concurrent users and a widely diverse database workload.

6. CONCLUSION

In this short paper, we addressed the challenge to obtain real-world query execution traces from customers, while obeying their strong desire to avoid leakage of sensitive data. We crossed the chasm with a suspension bridge built from obfuscated kernel performance trace events produced by MonetDB's Stethoscope and analysed with a home-brew tool Holmes. The initial runs shed light on both the application behavior and its impact on the software. A 30% performance improvement on this workload with better resource provisioning seems a low hanging fruit harvested soon.

The important lesson learned again for DBMS architects is to make sure the proper low-level performance tracers are available from the outset. Real-world traces show corners of the system that call for attention, both from a performance and system stability point of view.

7. REFERENCES

[1] ACM Sigmode Systems award 2016. www.sigmod.org/sigmod-awards/people/martin-kersten-2016-sigmod-systems-award.

[2] D. E. Bakken, R. Rarameswaran, D. M. Blough, A. A. Franz, and T. J. Palmer. Data obfuscation: anonymity and desensitization of usable data sets. *IEEE Security Privacy*, 2(6):34–41, 2004.

[3] K. Barker, M. Askari, M. Banerjee, K. Ghazinour, B. Mackas, M. Majedi, S. Pun, and A. Williams. A data privacy taxonomy. In A. P. Sexton, editor, *Dataspace: The Final Frontier*, pages 42–54, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[4] A. Böhm and T. Rabl, editors. *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*. ACM, 2018.

[5] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *Proc. VLDB Endow.*, 2(2):1648–1653, 2009.

[6] DB-Engines Ranking of Relational DBMS. https://db-engines.com/en/ranking/relational+dbms.

[7] D. E. Denning and J. Schlörer. A fast procedure for finding a tracker in a statistical database. *ACM Trans. Database Syst.*, 5(1):88–102, Mar. 1980.

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

[9] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[10] M. Ivanova, M. Kersten, N. Nes, and R. Goncalves. An Architecture for Recycling Intermediates in a Column-store. In *Proc. of the ACM SIGMOD*, 2009.

[11] M. L. Kersten, P. Koutsourakis, and Y. Zhang. Finding the pitfalls in query performance. In A. Böhm and T. Rabl, editors, *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, pages 3:1–3:6. ACM, 2018.

[12] PyStethoscope, a system monitor for MonetDB. https://www.monetdb.org/blog/pystethoscope.

[13] P. Tözün and A. Böhm, editors. *Proceedings of the 8th International Workshop on Testing Database Systems, DBTest@SIGMOD 2020, Portland, Oregon, June 19, 2020*. ACM, 2020.

[14] VLDB 10-year Best Paper Award 2009. www.vldb.org/archives/website/2009/q=node%252F50.html.

[15] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Mühlbauer, T. Neumann, and M. Then. Get real: How benchmarks fail to represent the real world. In A. Böhm and T. Rabl, editors, *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, pages 1:1–1:6. ACM, 2018.

[8] https://github.com/cwida/public_bi_benchmark