# Get Real: How Benchmarks Fail to Represent the Real World

Adrian Vogelsgesang, Michael Haubenschild,
Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, Manuel Then

*Tableau Software*

{avogelsgesang, mhaubenschild, jfinis, akemper, vleis, tmuehlbauer, tneumann, mthen}@tableau.com

## ABSTRACT

Industrial as well as academic analytics systems are usually evaluated based on well-known standard benchmarks, such as TPC-H or TPC-DS. These benchmarks test various components of the DBMS including the join optimizer, the implementation of the join and aggregation operators, concurrency control and the scheduler. However, these benchmarks fall short of evaluating the "real" challenges imposed by modern BI systems, such as Tableau, that emit machine-generated query workloads. This paper reports a comprehensive study based on a set of more than 60k real-world BI data repositories together with their generated query workload. The machine-generated workload posed by BI tools differs from the "hand-crafted" benchmark queries in multiple ways: Structurally simple relational operator trees often come with extremely complex scalar expressions such that expression evaluation becomes the limiting factor. At the same time, we also encountered much more complex relational operator trees than covered by benchmarks. This long tail in both, operator tree and expression complexity, is not adequately represented in standard benchmarks. We contribute various statistics gathered from the large dataset, e.g., data type distributions, operator frequency, string length distribution and expression complexity. We hope our study gives an impetus to database researchers and benchmark designers alike to address the relevant problems in future projects and to enable better database support for data exploration systems which become more and more important in the Big Data era.

## 1 INTRODUCTION

To evaluate the performance of database systems, researchers and system vendors use standardized benchmarks like TPC-H and TPC-DS for analytical workloads and TPC-C for transactional workloads. Unfortunately, there are only few such benchmarks and the public availability of real-world database workloads is limited. This especially puts academia but also practitioners at risk of overfitting their implementations to the few available benchmarks, leading to bad design choices if the benchmarks lack relevance and broad applicability [6].

*DBTest'18, June 15, 2018, Houston, TX, USA*
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5826-2/18/06…$15.00
https://doi.org/10.1145/3209950.3209952

In this paper we focus on the evaluation of real-world analytical query processing systems. In recent years, visual-based analytics tools such as Tableau have disrupted the business intelligence (BI) market. These modern BI tools allow the user to interact with and understand their data on a higher level, while traditional SQL queries are automatically generated in the background. These machine-generated queries introduce new challenges for query processing systems. In contrast to "hand-written" queries, interactive high-level interfaces allow users to easily express very complex queries. Even a simple calculation in the BI frontend can lead to a large query size due to expression unrolling. There is also an impedance mismatch for some concepts in the BI tools that cannot be mapped directly to a SQL query. Consequently, query generation in a BI tool has to restructure the query, which can lead to complicated structures in SQL. Further, taking analytics to a higher level not only changed queries, but also data set characteristics. Users now expect to be able to easily analyze data of all sizes and of all types. Standardized benchmarks are relevant, but given the observed long tail in both, query and data set complexity, we challenge the broad applicability of standardized benchmarks, especially to modern BI workloads.

In particular, the main contributions of this paper are:

- We offer insights into real-world BI workloads at scale, based on monitoring and sampling 60k data repositories together with their query workloads generated by a modern BI tool.
- We point out the high-level differences between a modern BI workload and standard industry benchmarks.

While we observed a significant discrepancy between standardized benchmarks and the observed real-world usage, we do not claim that existing benchmarks are not relevant. Instead, we hope that this paper will augment existing benchmarks that focus on hand-crafted queries and give an impetus for other database researchers and vendors to make their systems more versatile and robust for the challenges imposed by modern BI tools.

## 2 METHODOLOGY

Tableau Public[1] is a free offering by Tableau for sharing BI visualizations on the web. It hosts user-generated Tableau workbooks, each containing one or more visualizations. A workbook comprises the logical description of a visualization and the visualized data, in the following referred to as *extract*. From this logical description, Tableau generates SQL queries and uses their result to render the final visualization. The generated SQL queries are executed using Hyper, Tableau's query processing backend for extracts.

The workbooks hosted on Tableau Public give us a comprehensive and fairly representative overview of a modern BI workload. We

[1] https://public.tableau.com

collected statistics on a random sample containing approx. 62k workbooks using two methods: First, we obtained metadata about the data stored in the extracts such as data types, sample values, domain sizes, and the length of strings. Second, we collected all SQL queries and query execution plans from Hyper's log files.

In total, we collected over one million generated SQL queries, covering a large variety of real-world scenarios. 75% of those queries are metadata queries that do not reference relations from the dataset but instead, e.g., retrieve the schema information or the current server time. This surprisingly high number of metadata queries is partially caused by our non-interactive measurement method which nullifies the benefits the metadata cache would provide in interactive sessions. Nevertheless, the number of metadata queries is still remarkable, which leads us to our first insight:

INSIGHT 1. *BI tools issue many queries to retrieve information about the schema and other metadata.*

Querying the system catalogs therefore needs to be efficient. Since it is usually straightforward to provide sufficient performance for meta data queries, we ignore them for the remainder of this paper and only show results for the remaining 250k queries.

## 3 DATASETS

In this section we focus on the dataset characteristics before delving into the query workload characteristics in Sec. 4.

### 3.1 Strings are Everywhere

The TPC-H benchmark uses integer keys for all relations. In contrast, our real-world dataset mostly features strings as keys: ISO country codes are used to identify countries, IANA codes for airports and ISBNs for books. UUIDs or other alphanumeric identifiers are also common choices where pre-established keys are not available. All those different flavors of "surrogate" keys have one thing in common: They are stored as strings in the database, i.e., either as `VARCHAR`, `CHAR` or `TEXT` depending on the DBMS and administrator.

Similarly, other non-key columns that a DBA would normally specify as `INTEGER` or even as a boolean are also commonly stored as strings. Our dataset shows that more than 60% of the single-character strings are `0` or `1`. With a combined frequency of 4.5%, the characters "Y" and "N" are also very popular to represent"yes" and "no", respectively. In a cleanly designed schema, those columns would be represented as booleans. Another common pattern is to store fiscal years as strings in the form of "2017/18". In general, while the schema of TPC-H and other benchmarks is carefully engineered and designed by an experienced DBA, most real-world queries work on a schema with a very pragmatic design to just "get the job done". This leads us to:

INSIGHT 2. *Most data is stored as strings. BI data is often represented in the application domain, not in a format efficient for processing by database systems.*

Benchmarks should therefore not only work on a perfectly designed schema, but also on relations with sub-optimal data type choices. There are multiple reasons for these. One, as hinted above, is that there are generally no corresponding types to store, e.g., UUIDs and fiscal years. While most DBMS offer the user the possibility to create custom datatypes, this feature is seldom used. In that sense, strings are used as a "dumping yard" for all types other than the most common SQL types which are supported by all major DBMS.

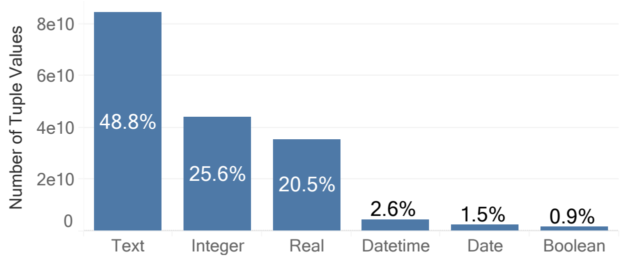

**Figure 1: Distribution of data types used**

Even if a specific datatype is supported, it might not be possible to use it due to dirty values that cannot be parsed cleanly. Their existence leaves two choices during data ingestion: either store them as `NULL`, or fall back to store the whole column as strings. Our dataset indicates that the latter usually prevails, probably to prevent loss of information. A human can still extract some meaning out of the textual representation of, e.g., a date containing a '\' instead of a '/', or a number containing an accidental character are examples.

Given all those different reasons to store data as strings, the distribution shown in Fig. 1 is not surprising. It shows that 48% of all values are stored as strings. Integers are slightly more common than floating point numbers. Dates and booleans are pretty rare. This does not mean that people are not using them. It rather means that dates and booleans are stored as strings and only interpreted during query time by calling functions such as `TO_TIMESTAMP`. Users tend to import data into Tableau and only afterwards use the calculation language to clean it up. The calculation language enables users to refine and clean their data while they are visualizing it. At the same time, this also means that the data initially imported and stored in the database files is far from clean.

In contrast to this, industry benchmarks use strings mostly as ballast. They are either directly presented to the user, used as group key, or as restrictions. However, this allows one to mostly ignore the string content itself and instead use, e.g., dictionary tokens or indices during processing. In the few cases where string modifications are applied (such as `SUBSTRING` from TPC-H query 22), those modifications are cheap to execute.

Fixed-precision numbers, such as `NUMERIC` or `DECIMAL`, are missing from Fig. 1 completely. While TPC-H and TPC-DS make extensive use of fixed-precision numbers, Tableau made the design choice not to support fixed-point data types and instead always uses floating point values. It is not alone with this choice: Google's Cloud Spanner does not support numerics, either.

### 3.2 Collation and Unicode Support

In our international world, support for plain ASCII strings is no longer sufficient. Indeed, 0.64% of the strings contain non-ASCII characters. While those characters are uncommon, a DBMS still needs to be able to store those Unicode characters and process them correctly. Doing so influences performance as it complicates string processing. However, this performance factor is not covered by existing benchmarks.

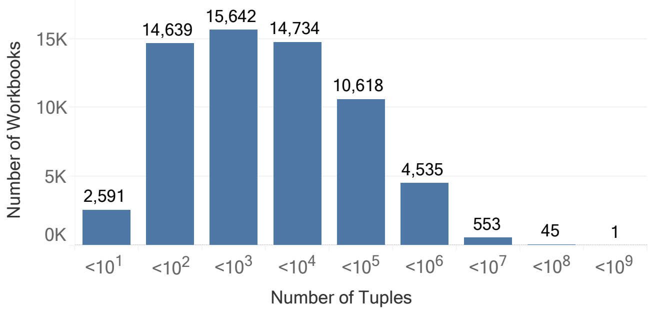

**Figure 2: Distribution of dataset sizes measured as the overall number of tuples**

Similarly, standard benchmarks completely ignore collations, although many database systems support them. Microsoft SQL Server even uses a locale-dependent, case-insensitive collation by default. PostgreSQL offered collations for more than a decade now, and it recently introduced unified collation support by relying on ICU, a library providing unicode and globalization support. Before, its behavior depended on the underling operating system. Although now unified, it is still not complete, as it is currently not possible to sort case or accent insensitive. By covering collation support, benchmarks could stimulate research in efficient collation support.

Case-insensitive collations are particularly challenging, as comparisons and hashing become much more expensive. Nevertheless, case insensitive ordering is something that appears very commonly in our workload. Over 85% of the string columns in our dataset have a collation. For more than 70% of the columns the collation is either case- or accent-insensitive. In more general terms:

Insight 3. *Collations play a large role when working with international datasets and highly impact query performance.*

Benchmarks should therefore feature more non-English collations to test how a database system performs when working on international data. Even machine-readable identifiers, such as ISO country or MAC addresses, are often stored with case-insensitive collations. In most cases, the collation does not actually influence the query result, but query evaluation has to take it into account nevertheless in order to guarantee the correctness of the result.

## 3.3 Dataset Size

To get an impression of the datasets uploaded to Tableau Public, Fig. 2 shows a histogram of the number of tuples per table. This histogram shows that most datasets are rather small. Only approximately 600 out of the 62 thousand workbooks contain more than a million tuples and thereby reach the order of magnitude of TPC-H scale factor one. In comparison to both TPC-H and TPC-DS which already deal with tables containing hundred thousands of tuples at the smallest scale factor, this is outright tiny. The majority of the datasets on Tableau Public contain less than a thousand tuples.

The heavy skew towards tiny datasets might be due to the target audience of Tableau Public, as it is a free service (and thus also contains a considerable amount of "toy" workbooks) and it limits the maximum workbook size. Consequently, workbooks of paying customers will be larger on average. Nevertheless, tiny data sets with a long tail of huge data sets are also common in industry.

| # | Query text (bytes) | Operators | Expressions |
|---|---|---|---|
| 1 | 6.7 MB | 3 | 287, 926 |
| 2 | 1.9 MB | 2 | 29, 786 |
| 3 | 1.5 MB | 2 | 20, 024 |
| 4 | 1.1 MB | 2 | 15, 847 |
| 5 | 602 KB | 2 | 9, 549 |

**Table 1: The top-5 largest queries. There is an extremely long tail with one query having over 200k expressions.**

Insight 4. *Many BI datasets are tiny, while few are quite large.*

Thus, a benchmark simulating the workload of a BI server should consist of a large number of queries on small datasets and some on larger datasets. Current analytic benchmarks instead work on a single, large dataset, which does not reflect the reality of BI systems. To get more insights of why most BI datasets are rather small, we looked at those small tables in more detail and found the following reasons: In most cases, these small tables contain pre-aggregated data. However, most of the time that data was not actively pre-aggregated by the user. Instead, the raw data is just not available at all. E.g., the population count of a country is only available as a pre-aggregated number. The underlying list of individual citizens is not existent or at least not publicly accessible.

If they do not contain some form of pre-aggregated measure, those small tables usually serve as dimension tables similar to the `region` table from TPC-H. They are used, e.g., to map ISO country codes to their respective full country names, airport codes to full airport names and currencies to their corresponding exchange rates.

## 4 QUERY WORKLOAD

As we mentioned earlier many datasets are rather small. Similarly, most queries are also fairly small and only 0.5% of the queries consist of more than 5KB of query text. There are, however, huge outliers of up to multiple megabytes of SQL text. Table 1 shows the five largest queries we observed in our test runs. Outside of our test set, we encountered even larger queries with up to 27MB query text.

The numbers also show that the queries are not huge just due to some large inlined string constants or comments but are actually complex: The largest query contains more than 280 thousand nodes in our internal representation of the expression tree – even after applying logical optimizations such as constant folding and partial evaluation. In other words:

Insight 5. *Benchmarks should also scale query complexity, not only data size.*

This is in contrast to most benchmarks: Benchmarks only scale data sizes but the queries are rather small. The largest queries from TPC-H only consist of approximately 1.4 KB of query text and do not cover the long tail of huge queries we observed.

The sheer query size puts stress on parts of the database system which are not covered by common benchmarks: The network layer needs to be optimized to receive the query text. The SQL parser has to process megabytes instead of kilobytes of query text. The logical optimizer has to deal with expressions containing hundred thousands of nodes, such that each additional traversal over the tree should be considered carefully. For compiling databases, during code

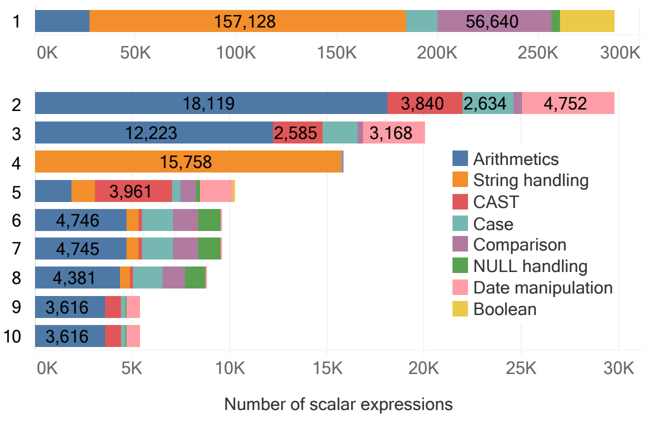

**Figure 3: The 10 queries with most expressions, grouped by expression type**

generation the goal is no longer only to produce high-performance code to reduce query execution time. Instead, it becomes equally important to optimize query compilation time. For example, if LLVM is used to generate machine code, some parts of it do not scale linearly with the code size [9].

While those queries are still the exception, those large and often slow queries are the ones that impair user experience the most. Taking multiple days to compile a query which is then evaluated on a few thousand tuples is just not acceptable.

## 4.1 Scalar Expressions

As shown in Table 1, the largest 5 queries contain a lot of expressions but only a small number of relational operators. Most of them even only contain a TableScan and a GroupBy operator, the largest one contains an additional Map operator. None of them contain a Join. Hence, join reordering, the focus of most research on query optimization, will not help at all to handle those queries. The actual complexity is contained in the expressions. Hence, we claim:

Insight 6. *Optimization and evaluation of complex scalar expressions should be covered by BI benchmarks.*

The largest query belongs to a dataset of only 41k tuples, but the data itself consists of string columns representing integers on which heavy calculations are done. Thus all the 79k references to columns in the base table are parsed as an integer first. In addition, the query contains 7.3k CASE expressions. If all of those would be evaluated per tuple, the query takes multiple days in Hyper just to compile. By employing common subexpression elimination and other techniques, we were able to reduce compilation and execution time to seconds. It now takes longer for the client application to generate the SQL query than for the DBMS to answer it.

By adopting techniques known from compilers such as common subexpression elimination, the removal of unreachable branches of CASE statements and combining subsequent equivalent branches, we were able to simplify most queries. Furthermore, optimizing expressions allows us to trigger traditional query optimization techniques in more cases. One example here is that we can cull a join only after removing all unreachable branches which reference columns from the joined table.

From Fig. 3 we can see the different purposes for which expressions are used. It shows the ten queries with most expressions representing the expression type by color. As before, the numbers shown are measured after optimizing the query which includes constant folding. Expressions were grouped together based on their purpose into the following groups: *Arithmetics* such as addition, multiplication, square roots and logarithms; *String handling* which includes both string manipulation, string parsing and regular expressions; Type *Casts*; CASE expressions; *Comparisons*, also including IN clauses with a constant list of values; expressions for *NULL handling* such as NULLIF, COALESCE and IS NULL; *Date manipulation* such as extracting years or months of timestamps; and *Boolean* expressions such as AND,OR and NOT

As expected, arithmetics and string handling are the two most prevalent expression types. Given the small number of date/datetime values in the underlying data (see Fig. 1), the large number of operations on date values is unexpected. It stems from the fact that dates are stored as strings and only parsed as part of the query.

The high number of CASTs is not because users type those casts into their calculations. In most cases, they are inserted automatically during query generation. This is done to avoid overflow errors in computations. For example, the day extracted from a date has datatype INT. Tableau always wraps this in a cast to BIGINT, so that users can apply computations on that result without having to worry about overflows. In this way, complex expressions are used to bridge the gap between SQL semantics and end user expectations.

The high number of expressions related to NULL handling are also due to discrepancies between SQL semantics and user expectations: One such instance is the division operator. In most database systems, a division by zero raises an exception and aborts query execution. For data exploration this behavior is not helpful: You do not want a complete query to fail just because division for a single tuple failed. Instead, the user wants to see at least the result tuples for which no such error occurred. To achieve this, NULLIF is used to ensure that each division by zero returns NULL instead of raising an error. In general, Tableau tries to return NULL in all error cases for expressions and injects custom error handling into the queries to achieve this behavior. Since NULL is the common error value, users use IS NULL and COALESCE in order to handle such errors.

Furthermore, BI systems try to provide consistent semantics across database systems: One such example is the SPLIT function which allows users to split a string and select the *n*-th group. Postgres offers this functionality under the name SPLIT_PART. However, Postgres throws an exception for negative *n* while there are other systems that return the *n*-th last group in this case. To achieve the desired semantics, the expression REVERSE(SPLIT_PART(REVERSE(input), REVERSE(delimiter), n*-1)) is used. Of course, the performance of such an expression is suboptimal.

All those examples for expensive expressions are due to the same root cause: BI applications want semantics different from the SQL standard.

## 4.2 Relational Operators

Table 2 gives a rough impression of the relational operators that appear in the workload. As expected, the vast majority of queries contains at least one Table Scan. Most queries are requesting aggregated results and hence contain a Group By. At the same time,

| Operator | Percentage of queries | Max operators in single query |
|---|---|---|
| Table Scan | 97.8% | 273 |
| Group By | 80.7% | 253 |
| Sort | 17.7% | 254 |
| Inner-Join | 4.5% | 164 |
| Temp. Table Creation | 2.2% | - |
| Outer-Join | 1% | 247 |
| Percentiles | 0.5% | 132 |
| Group-Join | 0.3% | 36 |

**Table 2: Frequency of individual relational operators illustrated by 1) the percentage of queries containing at least one such operator 2) the maximum number of times an operator occurred in a single query**

a large percentage of queries contains at least one Sort operator. Note that those Sort operators are not only used in order to sort the result set. They are also frequently used in combination with a `LIMIT` clause to implement a Top-N selection.

Joins are a bit less common in our workload than one would expect. This is due to the fact that most datasets on Tableau Public are single-table datasets. At the same time, it is an interesting observation that, although most queries target a single table, they still contain a significant number of joins. Those joins are typically self joins, although not in their most naive form: They join a base table with an aggregated version of that same base table. Thereby, this combination of a Group By with a Join adds additional columns containing aggregated measures but without reducing the number of tuples. The same result could be achieved using window functions of the form `SUM(col1) OVER (PARTITION BY col2,col3 UNBOUNDED PRECEDING UNBOUNDED FOLLOWING)`.

This is also the reason why our workload does not contain any window functions: For lack of comprehensive support among different data sources, Tableau uses the combination of the commonly available GroupBy and Join operator to emulate most use cases for window functions. For the rest, e.g. running sums, Tableau computes the window functions on the client side. Hence, the numbers we obtained do not contradict statistics obtained in previous work [8] which identified 4% of the queries to contain window functions. In our statistics, those 4% just show up as joins instead of window functions. This leads us to

Insight 7. *Benchmarks should cover advanced SQL functionality, in particular window functions.*

Another example of advanced SQL functionality are "Percentiles". Percentile operators represent a special type of aggregate function which calculate percentiles, i.e., an aggregate which in contrast to COUNT, SUM, MIN and all other basic aggregates requires materializing and sorting the complete input. While database systems might implement this functionality as part of the normal Group By operator, calculating this result involves effort similar to a Sort operator. Hence, we decided to account for it separately.

Group-Joins are a special operator which fuses a Group By and a Join operator together[11]. The combined Group-Join operator achieves better performance by sharing the hash table between Group By and Join. Again, the small number of Group-Joins is caused by the small number of joins to begin with.

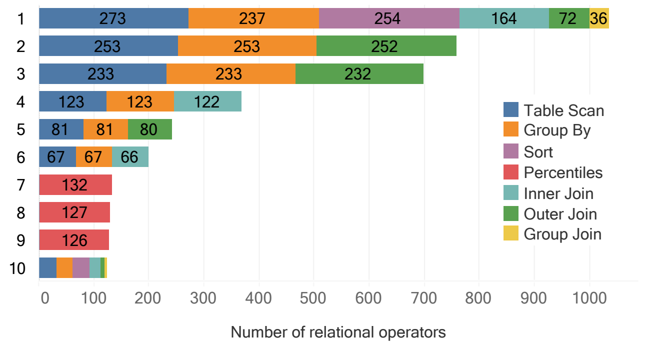

**Figure 4: The 10 queries with the most relational operators. Color indicates operator type**

Although the data we collected represents a read-only OLAP benchmark, still 2% of the queries are used to create and populate temporary tables. These tables serve two purposes: First, they are part of a technique to avoid gigantic `IN` clauses in the query text. Since `IN` clauses represent an invisible join [2] anyway, and to allow reuse across multiple queries, Tableau expresses them by populating a temporary table first with the qualifying values and then joining on them in subsequent queries. Second, temporary tables are used to represent ad-hoc lookup tables. Remapping a set of values to new values can be expressed in SQL using a `CASE` expression. To avoid overly large `CASE` expressions, a join with a temporary table is used instead of an otherwise huge `CASE` expression. Due to this a conceptually read-only workload still issues write statements. In case a database system does not support the creation of temporary tables, query generation falls back to issuing large `IN` or `CASE` expressions. Hence, it is crucial for performance to support creation of temporary tables, although the workload is actually a read-only workload.

The second column in Table 2 illustrates that some queries contain a much larger number of operators than covered by any benchmark. While other database practitioners already observed a similar order of magnitude of operators per query [4], we see a large number of operators not only for joins and table scans, but also for Group By, Sort and Percentile operators. Furthermore, Fig. 4 reveals that those different types of operators do not occur independently of each other. Rather the opposite is true: The query with the most operators contains a lot of table scans, aggregations, sort operators and joins, all at the same time. For benchmarks this means that

Insight 8. *Benchmarks should scale to a much larger number of relational operators.*

While such large numbers of operators are rather the exception than the rule, those large queries are most demanding for the optimizer and the execution engine.

## 4.3 Incomplete/Underspecified Queries

Another interesting observation is that some queries are incomplete or underspecified. As the users are exploring their data and refining their queries, the intermediate queries tend to be incomplete: Crucial parts of join conditions might still be missing, turning joins into de-facto cross products; `WHERE` clauses might not contain all filter conditions yet, leading to much larger scans than intended; `ORDER BY` clauses are still missing expressions, triggering undefined behavior due to ties in the sort order.

Insight 9. *Benchmarks simulating interactive data exploration should also include underspecified queries to test system robustness.*

Although we collected statistics for static, finished workbooks only, we came across a few instances of such incomplete queries. In one workbook, for example, a table with multiple million rows is joined with itself on a column containing only 7 distinct values. This obviously leads to a de-facto cross product.

One example of such a bogus query would be the interactive exploration of the TPC-H dataset: While the user is still editing the join conditions, a query is already issued in order to display an intermediate results. This query joins the `lineitem` table with the `partsupp` table only on the `suppkey` but not on the `partkey` column. It would be desirable if such an incomplete and probably meaningless query would still complete. At least, in combination with a `LIMIT`, the query should be fast so the user can see which mistake he made. The bare minimum expectation for such incomplete or bogus queries is that they do not impede other concurrent queries and that they can be cancelled quickly as soon as the user realizes that he posed an incorrect query and refined his join condition.

However, underspecified queries are not only issued due to interactive exploration. Sometimes, a query does not need to be well-defined: To get a quick overview of the data contained in a table, displaying a few thousand records from that table is usually a good start. In order to retrieve those records, Tableau uses a `LIMIT` clause without a corresponding `ORDER BY` clause. Strictly speaking, this kind of query is non-deterministic. Nevertheless it is sufficient for the purpose of previewing a dataset.

## 5 RELATED WORK

While considerable work has been done in developing challenging benchmarks, none of them stress the dimensions we found in our data, e.g., string dominant processing, a long tail of expression complexity and large queries.

The most relevant related work is probably the evaluation of **SQLShare** [8], a database-as-a-service project intended to be used by scientists. They found many of the same characteristics we observe, including many small datasets with few tables, high skew in query complexity, data cleansing & transformation directly in SQL, and calculation-heavy queries. The main difference is that their queries are handwritten, which manifests itself in the size of the workload (25k vs. 1M queries), and the absence of metaqueries.

In contrast to this, Martin Boissier et al. [1] analyzed machine-generated real-world queries, just like we do in this paper. However, they investigate machine-generated workloads from a different perspective. As they report different metrics, the results can not be directly compared.

**TPC-H** is perhaps the most widespread and well-understood benchmark for decision support systems. It consists of complex analytical queries with focus on efficient join processing. This is in contrast to our analysis of BI workloads, which shows that calculation heavy aggregation on a single denormalized table often dominates query performance. Boncz et al. [2] did a thorough investigation of TPC-H, in which they identify the tested features of the DBMS, or in other words, the features a DBMS must optimize for in order to reach competitive results. Some of their findings, in particular choke point 3.2, are very specific to TPC-H. The importance of others, such as common subexpression elimination (4.2a), we observe in our dataset as well, even to a much larger extent than they are stressed in TPC-H. We can also confirm the relevance of choke point 4.2c (large IN clauses) and 4.3 (string matching performance), as `IN` clauses appear in our dataset in 46k queries (up to 81 instances in a single query), and 10.8k queries do string manipulation.

The more recent **TPC-DS** improves upon TPC-H by adding a more diverse workload with complex queries. However, it still does not adequately represent the amount of skew we found in our data regarding query size and complexity. Also, it does not put enough pressure on string processing. Each table in TPC-DS has a surrogate (integer) key which is used for join processing, while most of the joins we observed are on collation aware string values. We note that all queries that appear in TPC benchmarks depict crafted SQL written by a skilled database administrator. They vastly differ from machine generated queries, which in general are more verbose and can contain unnecessary and duplicated expressions (see Sec. 4.3).

While TPC-H and TPC-DS are probably the most widespread analytical benchmarks, there are more specialized benchmarks focusing on specific aspects like hybrid transactional/analytical processing (HTAP) [3], join order optimization [10], graph analytics [7], or various OLTP workloads [5].

## 6 CONCLUSION

In this paper we presented the analysis of a large body of real-world BI workloads and extracted 9 core insights from analyzing these workloads. As modern BI tools can connect to most DBMSs on the market, our findings are not confined to Hyper, but are immediately applicable to all major database systems. They depict the actual challenges that database systems face these days when being connected to a BI tool.

We showed the discrepancy between current benchmarking practice and the real world workloads emanating from BI systems. As these database frontends gain more and more importance, we hope that future benchmark designers and, in particular, system builders get an impetus for enabling better database support for such system-generated workloads.

## REFERENCES

[1] Martin Boissier, Carsten A. Meyer, Timo Djürken, et al. 2016. Analyzing Data Relevance and Access Patterns of Live Production Database Systems. In *CIKM*.

[2] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPCTC*.

[3] Richard L. Cole, Florian Funke, Leo Giakoumakis, et al. 2011. The mixed workload CH-benCHmark. In *DBTest*.

[4] Nicolas Dieu, Adrian Dragusanu, Françoise Fabret, et al. 2009. 1,000 Tables Inside the From. *PVLDB* (2009).

[5] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, et al. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013).

[6] Karl Huppler. 2009. The art of building a good benchmark. In *TPCTC*.

[7] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, et al. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *PVLDB* (2016).

[8] Shrainik Jain, Dominik Moritz, Daniel Halperin, et al. 2016. SQLShare: Results from a Multi-Year SQL-as-a-Service Experiment. In *SIGMOD*.

[9] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *ICDE*.

[10] Viktor Leis, Andrey Gubichev, Atanas Mirchev, et al. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015).

[11] Guido Moerkotte and Thomas Neumann. 2011. Accelerating Queries with Group-By and Join by Groupjoin. *PVLDB* 4, 11 (2011).