Tractor Pulling on Data Warehouses

Martin L. Kersten
CWI Amsterdam
mk@cwi.nl

Alfons Kemper
TU München
kemper@in.tum.de

Volker Markl
TU Berlin
Volker.Markl@tu-berlin.de

Anisoara Nica
Sybase, An SAP Company
Waterloo, Canada
anica@sybase.com

Meikel Poess
Oracle Corporation
Redwood Shores, California
meikel.poess@oracle.com

Kai-Uwe Sattler
Ilmenau Univ. of Technology
kus@tu-ilmenau.de

ABSTRACT

Robustness of database systems under stress is hard to quantify, because there are many factors involved, most notably the user expectation to perform a job within certain bounds of the user requirements. Nevertheless, robustness of database system is very important to end users. In this paper we develop a database benchmark suite, inspired by tractor pulling, where robustness is measured as a system's ability to process data despite a continuous increase in system load, as defined in terms of data volume, query volume and complexity. A functional evaluation is performed against several systems to highlight the benchmark capabilities.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*; H.2.7 [**Database Management**]: Database Administration—*Data warehouse and repository*

General Terms

Experimentation, Measurement, Performance, Reliability

Keywords

Benchmark, Robustness, SQL, TPC, Data Warehousing, Tractor Pulling, Dagstuhl

1. INTRODUCTION

Its roots dating back to the 1860's, tractor pulling is a contemporary motorsport competition that requires a farm tractor to pull a heavy sledge along a 100 meter (about 300 feet) track. As the sledge is pulled down the track over a prepared soil the sledge weight, which is linked to the sledge wheels, is transferred from its rear axles towards its front axles. A pan, which essentially is a metal plate, is located in front of the rear wheels of the sledge. As the weight moves toward the front pan the resistance builds. The further the tractor pulls the sledge, the harder it gets. The winner is the tractor that can pull the sledge the farthest without blowing up. Besides the pure horse power of the tractor the success of a tractor pulling team depends on the team mechanics to prepare the tractor for a specific track, and on the driver to steer the tractor.

Database Management Systems (DBMSs) are like tractors when it comes to supporting large information systems. Users demand database systems to lift a heavy weight for a long time without breaking down. As tractors they come in several brands, allow for adjustments of the software and hardware to deal with the soil specifics, and are expected to provide proper service for a long time. However, not all DBMSs are the same in performing their task. The service provided differs considerably in many details and user satisfaction is at stake due to hickups in the engine itself. As the driver of a tractor often has to go under the hood to remove dust, stones, dirt, and use the oil can to make sure the system is up to the job, database administrators have to maintain the data warehouse continuously to guarantee service level agreements.

These differences make the selection of a DBMS for a particular task difficult. In many cases this is tackled by calling for proof-of-concept implementations at the client site by vendors, to rerun user workloads, or to extrapolate from public benchmarks. Such benchmarks, such as TPC-H [2] for analytical workloads and TPC-C [1] for transactional workloads, have been used for decades to stress test products as well as to compare systems and system configurations. However, it is a well-known fact that either approach tells little about the robustness of the system for the task at hand. A slight change in environment may render a system useless, or at least not behave as expected.

System robustness is a leading criteria in the design of modern information systems. It comes in many flavors, which share the semantics that a) a system behaves as expected, and b) minor changes in the system parameters, the query workload, or the database content do not lead to large response time variances among identical jobs. The former is often hard to quantify in practice, unless one focuses on a system component for which a theoretical model could be identified to capture the intended behavior. For example, checking the robustness of a cost-based optimizer can be described by a mathematical model.

To improve our knowledge about system robustness we designed a novel way to exercise database systems against the functional requirements of large data warehouses inspired by the tractor pulling game. Compared to simple stress tests and performance benchmarks the Tractor Pulling suite takes a more holistic approach, where the robustness follows the variance analysis to quantify objectively robustness. Therefore, an essential component in the setup is to identify a metric to assess the performance evolution during a single run, and to amortize the information obtained from several runs using different system brands or slightly adjusted parameter settings. The main contributions of this paper can be summed up by:

- The Tractor Pulling suite provides a framework for defining different track characteristics representing different models for testing robustness.
- The suite is formulated to systematically evaluate a system against an increasingly complex workload in an automatic way.
- The parameter space for comparison of intra- and extra-solutions is defined with a metric to enable relative comparisons of the solutions provided with a particular focus on robustness.
- The suite is run against several platforms for sanity checks and as a frame of reference for taking it further.

The suite is designed for portability as a set of scripts using simple SQL queries. The provided SQL queries are merely examples of queries that could be used in a robustness benchmark rather than claiming to be the best and only queries. Future work might identify a representative set of queries or it might reveal that specific instantiations of the tractor suite need to be tailored for specific application domains. Hooks are provided to turn the suite into brand specific versions, which may include gearing advice and DBA advice in general.

2. TRACTOR PULLING SUITE

In this section we will describe the Tractor Pulling suite in more details. The tracks are paved with an increasing workload using a particular database soil. The DBMS to support the workload can be fine-tuned to derive best-practice advice to increase robustness of the solution.

2.1 The Workload Track

The tractor suite is built around tracks composed of a series of workloads $\{W(i)\}_{i=0,N}$ where the length of the workload sequence is limited to an a priori defined parameter N. Each workload $W(i)$ is a tuple $W(i) = \langle S(i), L(i), Pre(i), qry(i), tQ(i), Q(i), Post(i), db(i)\rangle$ which contains a list of operations to change the database schema $S(i)$, perform the (bulk)- load $L(i)$ and run a query batch $Q(i)$. The database size is increased at the beginning of each track by creating a new table, which constitutes the largest table in the database thus far, its size being defined by the function $db(i)$. The function is expected to be monotonically increasing.

Likewise the global function $Q(i)$ determines the query load at the track i, i.e. the queries to be processed. We ensure that workload query set $Q(i)$ of $W(i)$ is fully contained in the query set $Q(i+1)$ of the track workload W_{i+1}, and that each query in $Q(i)$ will produce the same answer when run during the track $i+1$. This is achieved by gradually increasing the domain range from which values are drawn. It creates several strata covering different key values. This way, we can quantify robustness of individual queries in light of an increasingly complex world they run into at each track. The queries are described below in Section 2.3.

The components $Pre(i)$ and $Post(i)$ encapsulate the DBA knowledge, such as gathering statistics and building indexes before queries $Q(i)$ are executed, and post actions such as garbage collecting temporary storage or performing information feedback into the optimizer based on observed behavior. By using $Pre(i)$ we are closer to the real world case, where the system needs steering to keep performing as before, once negative deviations from expected behavior occur.

2.2 The Data Warehouse Soil

The database at the time of the track W_i is derived from the workload W_{i-1} incrementally using the global functions $db(i)$. The initial size $db(0)$ ensures that the table $R_0(K_0, B_0)$, created at the track 0, contains a few hundred thousand tuples that comfortably fit in main memory of the system, but the database size will quickly leave this performance-wise preferred resource setting as a new table is added at each step. The size for the newly created table at track i can be obtained in various ways with widely different performance and robustness issues. In the initial approach each workload step starts with creation of a new table using bulk loads. This reflects a common use in business intelligence applications, where a daily bulk load of transactional data precedes a lengthly analysis and reporting phase. For simplicity of the database soil, we assume all columns to be defined over the same ordered domain. Values for all columns $K_i, B_0, \ldots, B_i$ are taken from {integer, float, timestamp, varchar}. Switching between these scalar types is likely to demonstrate different behavior, as more complex functions are called in the inner loop of most algorithms. Focusing on one type at the time helps identifying the culprit for degradation.

The key range for the column $R_i.K_i$ is a dense sequence from $[l(i), h(i))$ which mimics the growing trans"-actional database. A stress test can be performed by shuffling the key values before the bulk load, but that would merely add a constant cost for the DBMS to sort the load files before actual loading commences.

The database instance at track i is obtained by adding a new table $R_i(K_i, B_0, \ldots, B_i)$. The new table R_i is loaded with the following data (see Equations 2 and 3): (1) all tuples from the table R_{i-1} which are extended with an extra column $R_i.B_i$ filled by randomly selecting values from $R_{i-1}.K_{i-1}$ following a Zipfian distribution; and (2) $db(i)$ new tuples where unique key values for $R_i.K_i$ are taken from the range $[l(i), h(i))$, with $l(i) = h(i-1)$. This key distribution leads to a database built around independent strata.

This extension process is iterated, to create a growing number of tables with a growing size for each database instance at track i. By construction, each new payload column $R_i.B_j$ can be used to join with the key column $R_j.K_j$ with $j < i$. Thus each database instance can be considered to model a data warehouse with R_i as the fact table and its columns $R_i.B_j, j = 0, ..., i-1$, as foreign keys to the dimension table $R_j.K_j$.

Formally, we construct a new table R_i for the database instance of the workload W_i as follows: Let $l(i)$ and $h(i)$ denote the minimum and maximum key ranges for $R_i.K_i$. Let $key(R_i, l(i), h(i))$ denote drawing of a unique integer number in the range $[l(i), h(i))$ that does not yet exist in the key column $R_i.K_i$. Let $zipf(R_j)$ and $uniform(R_j)$ denote a Zipfian and a uniform draw of numbers, respectively, from the values of the column $R_j.K_j$. Equations 1, 2, and 3 describe how the new table is being populated at each track.

$$R_0(K_0, B_0) = \bigcup_{j=1}^{db(0)} (key(R_0, l(0), h(0)), uniform(R_0)) \qquad (1)$$

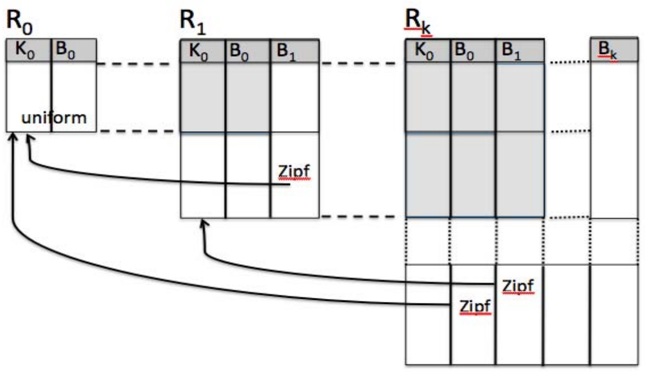


Figure 1: The Tractor Data Soil at Track k

$$R_1(K_1, B_0, B_1) = \{(k, b, \mathit{zipf}(R_0))|(k, b) \in R_0\} \bigcup \bigcup_{j=1}^{db(1)} \{(\mathit{key}(R_1, h(1), l(1)), \mathit{zipf}(R_0), \mathit{zipf}(R_1))\} \quad (2)$$

$$R_i(K_i, B_0, ..., B_i) = \{(k, b_0, ..., b_{i-1}, \mathit{zipf}(R_{i-1}))|(k, b_0, ..., b_{i-1}) \in R_{i-1}\} \bigcup \bigcup_{j=1}^{db(i)} \{(\mathit{key}(R_i, l(i), h(i)), \mathit{zipf}(R_0), \ldots, \mathit{zipf}(R_i))\} \quad (3)$$

Figure 1 illustrates the database evolution and the common parts between each instance.

2.3 The Query Load

Pulling the sledge over the field is comparable to running queries against a database and observing their behavior. Our test suite consists of a query mix $Q(i)$ derived from a limited class of simple yet representative data warehouse queries. With each step in the workload we augment the query set such that all tables including all their data are accessed at that time. While the benchmark progresses, the complexity of the query mix increases subject to the size of the query load $|Q(i)|$. The construction of the database content ensures that we don't need an exhaustive list of queries.

The robust query processing can be defined as the ability of a system to maintain constant response time in the face of slightly changing query parameters [3]. In particular, selectivity estimation underlying most cost-based optimizers are considered a major source for robustness failures. This phenomenon is represented by parameterizing the queries with a Zipfian distributed range constraint, one that favors small ranges over large ones.

The second complicating factor for query optimization is the complexity structure of a query, e.g. how many join predicates are being used, subqueries, and sort-based grouping and aggregation. For each database a seemingly unbounded collection of query templates can be conceived. In practice, however, just a few dozen of structural different queries are characteristic for a DBMS application. Moreover, the query complexity is inversely related to their use as simple queries are much more prevalent then large and complex queries.

The Tractor Pulling suite is based on a slowly increasing collection of queries to be run. At each track $W(i)$ we extend the previous query batch $Q(i-1)$ with new query instances following the templates defined in $tQ(i)$. This way we can observe the stability of a query batch as the database and query load increases. The Tractor Pulling query template $tQ(i)$ is based on the following generic i-way join queries:

```
SELECT   $R_0.B_0, ..., R_i.B_i$, count(*), avg($R_0.B_0$),
         avg($R_1.B_0$), avg($R_1.B_1$),. . ., avg($R_i.B_0$), . . .
FROM $R_0, \ldots, R_i$
WHERE $\mathit{selectpattern}(R_0, \ldots, R_i)$ AND $\mathit{joinpattern}(R_0, \ldots, R_i)$
GROUP BY $R_0.B_0, \ldots, R_i.B_i$
ORDER BY $R_0.B_0, \ldots, R_i.B_i$
```

The select pattern is a sequence of range bounds over the key attributes, e.g. $R_i.B_j >= X$ AND $R_i.B_j < Y$ where $X <= Y$ and forms a Zipfian distribution range. The join pattern is a conjunct of equijoin predicates referencing the key columns $R_i.K_i$. We consider the following four classical query templates defined in $tQ(i) = \{lq_i, cyq_i, sq_i, clq_i\}$: linear, cycle, star, and clique queries. The order of the equijoin predicates in the WHERE clause is purposely randomized to detect the query optimizer's ability to guard against predicate order. Care should be taken in the generation of the selection predicates. Taking random ranges over all attributes would render most queries empty. Therefore, the range is picked from the last strata added, e.g., for $Q(4)$ the range is taken from $[l(4), h(4))$ which ensures that all other tables can be access through the foreign key relationship $R_4.B_j \rightarrow R_j.K_j$ with $j < 4$. The key ranges are again Zipfian distributed over the range $[l(4), h(4))$. The aggregate operations in the templates provide an upper bound. For each query instance we select a few by Zipfian sampling the possible aggregate function terms.

At each track i, the set of queries to be run is defined by $Q(i) = Q(i-1) \bigcup (\bigcup_{j=1}^{qry(i)} \{q|q$ a query instance from $tQ(i)\}$, i.e., the previous workload $Q(i-1)$ is augmented with $qry(i)$ more queries instantiated from the template $tQ(i)$; the number of new queries at each step is defined by the function $qry(i)$ which is an input to the benchmark.

3. TRACK CHARACTERISTICS

A DBMS is a multi-layer resource system with widely different performance characteristics. Databases as small as the CPU L-2 cache are not common, yet, their behavior will largely be determined by the software coding robustness. At the other extreme, we have multi terabytes database occupying hundreds of disks, where the bandwidth to access relevant data becomes a hindrance. Evaluation of the systems calls for a specification of the workload tracks and its relationship with the tractor's capabilities. This is done by varying the global functions *db*, defining the size of the new table, and *qry*, defining the number of new queries, added at each step i. We introduce the notion of a landscape to demonstrate different database workloads which can be created from this general workload description $W(i)$.

db, the number of new rows for the new tables, is varied by the initial table size $db(0)$ and its growth rate constant g. The function $db(i) = g \times i \times db(0), \forall i > 0$ determines the number of rows in the new table at step i (see Equation 3): $|R_i| = |R_{i-1}| + g \times i \times |R_0| = db(0) \times (1 + g \times \sum_{j=1}^{i} j) = = db(0) \times (1 + g \times (\frac{i \times (i+1)}{2}))$.

The size, in bytes, of the table R_i is $|R_i|\times$ the size of its row which depends of the chosen domain for the columns $K_i, B_0, \ldots, B_i$. We will use the notation *dom* to denote the size of the domain of the columns used in the benchmark.

The total database size at step i, $DB(i)$, can be calculated as
$DB(i) = \sum_{j=0}^{i} |R_j| \times ((j+1) \times dom) = \sum_{j=0}^{i}(db(0) \times (1 + g \times (\frac{j\times(j+1)}{2}))) \times ((j+1) \times dom)$.

Assessing system robustness under increasing query workload can be achieved by controlling the function qry. The query batch $Q(i)$ for each track should be large enough to exercise the system in many ways. Typically, the query load within each step should be hundreds of query instances, which are fired against the system in single- or multi-user mode. We foresee that even single user behavior will highlight robustness issues. A multi-user load merely creates excessive competition for resources which is a major source for performance variances. The query workload can be characterized by the number of queries and joins executed per query in each step i.

A workload W_i is run in sequence against a DBMS without interruption until it finishes, get stalled due to limitations in its configuration, or blows up the engine. Using the workload for a performance characterization, albeit interesting for an entertainment perspective, is not our prime objective. In terms of robustness, a slow but predictable system and a fast one are considered the same.

We describe below a set of possible scenarios which can be defined using our benchmark framework. These landscapes were used to run the Tractor Pulling benchmark against some database systems (Section 5 presents the results).

Hills. The Hills scenario models a data warehouse that grows with a modest growth rate of $g \in (0,1)$ (e.g., $g = 0.2$). It starts out from a main-memory focus until it overflows into a few disks. It will highlight a system's robustness to deal with the memory-disk performance chasm. Starting from a small table R_0 of the size $d\%$ of the tractor's RAM, it increases linearly with an ascend of g:

$$\begin{array}{l} d \in (0\%, 100\%),\ g \in (0,1) \\ \text{Number of connections at track } i:\ 1 \\ db(0) = (d \times RAM) \times (\frac{1}{2\times dom}) \\ db(i) = g \times i \times db(0) \\ qry(0) = 1,\ qry(i) = 4 \\ |Q(i)| = 1 + 4 \times i \end{array} \tag{4}$$

For an initial $d = 30\%$ and $g = 0.2$, already at step $i = 1$ the database size $DB(1)$ is closed to 50% of the tractor's RAM. The query load is modestly growing as well. At each track, one query instance for each template in $tQ(i)$ is added (in total 4 more queries), and the new query batch $Q(i)$ is executed serially using one connection. Hence, at track i, $|Q(i)| = 1 + 4 \times i$. Although it is a relatively slow slope, it emphasizes the capabilities of the system to handle rather complex queries for modest database sizes.

Meadows. The Meadows scenario stresses the system behavior focusing on a modest increase in complexity of the query workload, while keeping the new tables number of rows constant.

$$\begin{array}{l} d \in (0\%, 100\%),\ g = 0,\ C > 1 \\ \text{Number of connections at track } i:\ C \\ db(0) = (d \times RAM) \times (\frac{1}{2\times dom}) \\ db(i) = 0 \\ qry(0) = 0,\ qry(i) = C \\ |Q(i)| = 1 + C \times i \end{array} \tag{5}$$

The new queries for each track may be chosen to be biased towards a certain type of queries, e.g., clique queries (see Figure 2(b) experiment where only clique queries were used).

Rockies. The Rockies illustrates the system behavior of steep climbing of the database size, emphasis complex queries, and an increased number of connections at each new track. At each track i, $4 \times i$ queries are added, i query instances for each template in $tQ(i)$.

$$\begin{array}{l} d \in (0\%, 100\%),\ g \in (0,10) \\ \text{Number of connections at track } i:\ i \\ db(0) = (d \times RAM) \times (\frac{1}{2\times dom}) \\ db(i) = g \times i \times db(0) \\ qry(0) = 1,\ qry(i) = i \times 4 \\ |Q(i)| = 1 + 4 \times (\frac{i(i+1)}{2}) \end{array} \tag{6}$$

4. ROBUSTNESS METRICS

In this section we introduce the Tractor Pulling robustness metrics, which leads to a n-way characterization focused on different system components. Robustness of a DBMS can describe several qualities, e.g., it does not break down easily by a single application failure, it recovers quickly from system failures, or faults in the code do not bring the system to a grinding halt. Robustness depends on many parameters (hardware, DBMS, etc.) and, therefore, cannot be expressed in a single number. Furthermore, it could have different meanings, e.g., constant elapsed times for identical queries while the database grows or a degraded response time while data volume are increased. Because it is hard to quantify the degree of expected degradation the smoothness of the gradient maybe used as an indicator for robustness.

Robustness is strongly related to the variance from user expectations. This implies that robustness is primarily a relative factor of the system against past behavior. In this context, we consider robustness as predictable load and query behavior of the system under changing environments. For data loading it means that we expect a linear behavior as more data is added to the database. For query processing – which is the main focus of the tractor pulling suite – this means that repetitive execution of a query should show little response time variations.

4.1 Performance Characteristics

The basis for our robustness metrics is formed by the wall-clock response times observed for the components of W_i. In particular, we are interested in the stability of the system under growth. The workload sequence is organized such that the same query is run against increasingly larger database. The database is set up in such a way that the same answer set is produced regardless the workload step. This means that we can check robustness against pure errors in query processing, but, more importantly, illustrates possible degradation beyond our expectation that a good system would provide consistent response time while its state changes.

4.2 Robustness Fingerprint

For ease of comparison amongst different tractor property settings and pulls, we propose a possible robustness vector

$$\mathit{Robust}(N) = \langle L, S, QO, QE, \{QO_k\}_{k=0,N}, \{QE_k\}_{k=0,N}, H \rangle$$

which depicts the performance characteristics for an N-way tractor pulling. Its components are defined as follows:

- *Load* $L(N)$: the sample standard deviation of the load times.
- *Storage* $S(N)$: the sample standard deviation of the disk storage size.

Statistic	Description
$t_l(i),\ i \geq 0$	Elapsed time to complete load operation at track i
$b_s(i),\ i \geq 0$	the total storage increased, in bytes, after the load operation at track i
$t_{qo}(i), i \geq 0$	optimization time for all queries at track i
$t_{qe}(i), i \geq 0$	execution time for all queries at track i
$t_{qo}(i,k), i \geq k, k \geq 0$	optimization time at track i, for the queries added at step k: $\Delta Q(k) = Q(k) \setminus Q(k-1)$.
$t_{qe}(i,k), i \geq k, k \geq 0$	execution time at track i, for the queries added at step k: $\Delta Q(k) = Q(k) \setminus Q(k-1)$.
$t_w(i), i \geq 0$	Elapsed time to complete the workload W_i at track i

Table 1: Statistics collected during Tractor Pulling Suite

Metrics	Description
$\Delta t_l(i) = \lvert t_l(i) - t_l(i-1)\rvert,\ i > 0$	Elapsed time difference of load time between track i and $i-1$
$\mu_l(N) = \frac{1}{N}\sum_{i=1}^{N} \Delta t_l(i)$	Mean of all load elapsed time differences
$L(N) = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(\Delta t_l(i) - \mu_l(N))^2}$	Load robustness metrics
$\Delta t_{qe}(i) = \lvert t_{qe}(i) - t_{qe}(i-1)\rvert,\ i > 0$	Elapsed time difference of query execution between track i and $i-1$
$\mu_{qe}(N) = \frac{1}{N}\sum_{i=1}^{N} \Delta t_{qe}(i)$	Mean of all query execution elapsed time differences
$QE(N) = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(\Delta t_{qe}(i) - \mu_{qe}(N))^2}$	Query execution robustness metrics
$\Delta t_w(i) = \lvert t_w(i) - t_w(i-1)\rvert,\ i > 0$	Elapsed time difference of between track i and $i-1$
$\mu_w(N) = \frac{1}{N}\sum_{i=1}^{N} \Delta t_w(i)$	Mean of all elapsed time differences
$H(N) = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(\Delta t_w(i) - \mu_w(N))^2}$	Holistic robustness metrics

Table 2: Robust(N) Metrics Formulas: sample standard deviation of delta vectors

- *Query optimization* $QO(N)$: the sample standard deviation of query optimization times.
- *Query Execution* $QE(N)$: the sample standard deviation of query execution times.
- *Query optimization* $QO_k(N)$: the sample standard deviation of query optimization times for the queries added at track k.
- *Query execution* $QE_k(N)$: the sample standard deviation of query execution times (excluding the optimization time) for the queries added at track k. This robustness measure and $QO_k(N)$ depict the behaviour of the new queries added at track k, i.e. $Q(k)\backslash Q(k-1)$, when run as part of the later workload $W(i)\ i >= k$.
- *Holistic* $H(N)$: the sample standard deviation of total execution time of the tracks. The total execution time of the track W_i includes any *Pre(i)* and *Post(i)* activities.

The Holistic robustness is calculated from the observed wall-clock times over all the steps in the tractor pulling. The variance from the trend is derived to capture the robustness. Ideally, the system is not susceptible to increasing workload. While running the tractor pulling benchmark with N tracks, the statistics shown in Table 4.1 are collected in order to compute $Robust(N)$ metrics. Each robustness metrics is then computed as the sample standard deviation of the statistics vector as defined in Table 4.2.

5. EVALUATION

As an example implementation of the benchmark we provide an open source package that can be downloaded from *sourceforge*[1]. The package consists of the data and query generator, `tpgen`, as well as a set of system-specific run scripts `tprun`. The workload generator `tpgen` creates, for given input benchmark parameters, the SQL DDL to create the schema, the data files to be loaded into the database, the scripts to load the data, and the SQL queries to be executed against the schema. The `tprun` script plays the role of the tractor's driver: loads the data, runs the queries, and collects the benchmark statistics (e.g., execution times). The open source package is already customized for major database systems and it can be further supplemented by instance-specific maintenance tasks such as index creation.

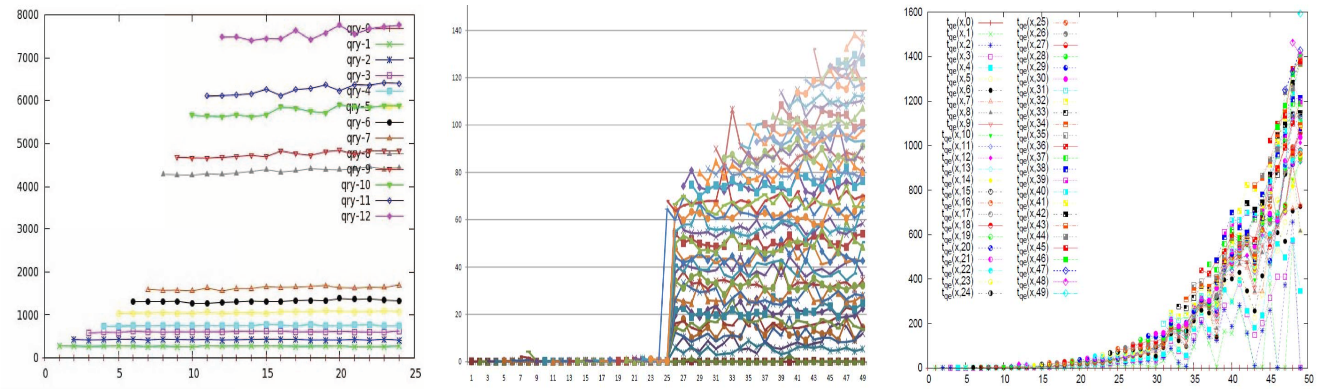
5.1 Tractor Pulling Experiments

In this section, we describe preliminary experiments done using Tractor Pulling benchmark for a set of database systems, where the Tractor Pulling implementation was used to define interesting landscapes. Our goal is to obtain first-use experiences with this initial implementation and draw some conclusions of how to further evolve the benchmark and develop new robustness metrics.

Figure 2 plots three benchmark runs on three different systems, using landscapes described in Section 3. All three systems experience sudden drops in performance at track 8, track 25, and track 40, respectively, which can be explained by how the landscapes are defined.

The Tractor Pulling suite with the Rockies landscape was run on System X with d = 0.06, g = 0.2, once for N = 49 and second run for N = 63. The original database size was very small comparing to the available RAM such that the database size reached the size of RAM around the track 40. The experiment stressed the query execution under increased query workload and number of connections: there are $1 + 4 \times (\frac{i(i+1)}{2})$ queries run at track i, where the number of concurrent connections is i. Figure 2(c) plots the execution times for N = 49, $\{t_{qe}(x,k)\}_{k=0,49}$. The computed robustness metrics for these two runs are:
$Robust(49) =$
$\langle L(49) = 11.14, QE(49) = 50.001, H(49) = 51.43\rangle$

[1]`https://sourceforge.net/projects/tractorpulling/`



(a) System Z on Hills $d = 20$, $g = 0.2$, $C = 1$, $N = 24$ (b) System W on Meadows $d = 20$, $g = 0$, $C = 4$, $N = 49$ (c) System X on Rockies $d = 0.06$, $g = 0.2$, $C = i$, $N = 49$

Figure 2: X-axis: the track number; Y-axis: the elapsed time for $\{t_{qe}(x,k)\}_{k=0,N}$

where $t_{qe}(i) \in [0.188, 1593.59]$ and $t_w(i) \in [1.73, 1698.525]$, $0 \leq i \leq 49$.
$Robust(63) =$
$\langle L(63) = 3.54, QE(63) = 12758.65, H(63) = 12758.36\rangle$
where $t_{qe}(i) \in [0.175, 154386.83]$ and $t_w(i) \in [0.723, 154441.453]$, $0 \leq i \leq 63$.
These preliminary experiments show that the sample standard deviation metrics must be interpreted in the context of the range of values observed during the benchmark, and also that a larger N is expected to give larger metrics.

6. SUMMARY AND FUTURE WORK

The writing of this paper was triggered by intense discussions at the Dagstuhl 2010 workshop on *Robust Query Processing* [3]. Although many seem to agree that robustness of a DBMS is a qualitative measure for deviance against expected behavior, there is no formal definition of DBMS robustness to date. Consequently there are little concrete techniques on how to measure it in practice. With the Tractor Pulling suite, developed in this paper, we provide a framework for defining robustness benchmarks. Its parameterized approach allows for exploring various dimensions of system robustness, such as database size, query workload and multiuser access. The database size can be varied in terms of number of rows, tables and columns per table. The workload can be increased in terms of the number of queries, complexity of queries and various level of multi-user access. Hence, the Tractor Pulling suite can be seen as a major step towards a standard way for quantitative assessment of DBMS robustness. As part of the tractor pulling suite we defined 4 metrics, Load, Storage, Query Optimization and Query Execution. Each of these metrics use the standard deviation as a relative measure between tracks. They are indicative for monitoring how robust systems are with increases in incremental data load, the overall database size and the query workload complexity.

To demonstrate the Tractor Pulling suite, we defined four sample instantiations of it: Hills, Meadows, and Rockies and conducted experiments on three systems, System X, Z and W. Each of these landscapes stresses different abilities of a system. The Hills landscape highlights a system's robustness to deal with the memory-disk performance chasm. It starts from a database that resembles a fraction of a system's main memory and increases the database size linearly at a modest rate. The Meadows landscape stresses a system's ability to deal with a modest increase in complexity of the query workload while increasing the database size considerably. The results of our experiments reveal that each system shows significant performance drops at a certain number of tracks, which can be observed with the metrics defined in our paper.

In future work we will concentrate on two key areas. Firstly, we will further investigate which Tractor Pulling Suite parameters define the most representative landscapes in terms of customer usage of current DBMSs. Secondly, we will investigate the possibility of a single robustness metric. The individual metrics, defined in this paper, are very indicative for how systems behave under increased workloads. In order to judge whether a system is robust in a more holistic approach, we need to combine these metrics into one single metric. This metric can also be used to quantitatively compare the robustness of multiple releases of the same system or different systems or even take monetary costs into account, which are needed to achieve a given robustness. The definition of a set of landscapes and one metric will, ultimately, lead to the definition of a benchmark for measuring DBMS robustness with concrete parameters for the database soil, the tracks and the queries and also formal execution rules and metrics.

Acknowledgements.
The authors wish to thank the participants, especially the workshop organizers, of the Dagstuhl 2010 workshop on *Robust Query Processing* [3] for their inspiration and advice.

7. REFERENCES

[1] Transaction Processing Performance Council. *TPC-C – On-line Transaction Processing Benchmark.*, `http://www.tpc.org/tpcc/`, 2011.
[2] Transaction Processing Performance Council. *TPC-H – Ad-hoc, Decision Support Benchmark.*, `http://www.tpc.org/tpch/`, 2011.
[3] G. Graefe, A. C. König, H. A. Kuno, V. Markl, and K. Sattler. *Robust Query Processing – Summary and Abstracts Collection*, number 10381 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2011.