DBMS Metrology: Measuring Query Time

SABAH CURRIM, University of Arizona
RICHARD T. SNODGRASS, University of Arizona
YOUNG-KYOON SUH, University of Arizona
RUI ZHANG, University of Arizona

It is surprisingly hard to obtain accurate and precise measurements of the time spent executing a query, as there are many sources of variance. To understand these sources, we review relevant per-process and overall measures obtainable from the Linux kernel and introduce a structural causal model relating these measures. A thorough correlational analysis provides strong support for this model. We attempted to determine why a particular measurement wasn't repeatable, and then to devise ways to eliminate or reduce that variance. This enabled us to articulate a timing protocol that applies to proprietary DBMSes, that ensures the repeatability of a query, and that obtains a quite accurate query execution time, while dropping very few outliers. This resulting query time measurement procedure, termed the Tucson Timing Protocol Version 2 (TTPv2), consists of the following steps: (1) perform sanity checks to ensure validity of the data, (2) drop some query executions via clearly motivated predicates, (3) drop some entire queries at a cardinality, again via clearly motivated predicates, (4) for those that remain, compute a single measured time by a carefully justified formula over the underlying measures of the remaining query executions, and (5) perform post-analysis sanity checks. The result is a mature, general, robust, self-checking protocol that provides a more precise and more accurate timing of the query. The protocol is also applicable to other operating domains in which measurements of multiple processes each doing computation and I/O is needed.

CCS Concepts:•**Information systems** → **Database query processing;**•**Software and its engineering** → **Software performance;**

General Terms: Performance

Additional Key Words and Phrases: accuracy; database ergalics; repeatability; Tucson Timing Protocol

ACM Reference Format:
Sabah Currim, Richard T. Snodgrass, Young-Kyoon Suh, and Rui Zhang, 2016. DBMS Metrology: Measuring Query Time. *ACM Trans. Datab. Syst.* V, N, Article XXXX (December 2015), 42 pages.
DOI: 0000001.0000001

1. INTRODUCTION

A common approach for at least the last forty years to measure database management system (DBMS) query time is as follows.

> "We used the UNIX *time* command to measure the elapsed time and CPU time. All queries were run 10 times. The resultant CPU usage was averaged." [HWANG, H.-Y. and YU, Y.-T. 1987]

Consider the measured times in Table I (below) for a single query repeated ten times, for which considerable care (to be described in detail later) was taken to get

Author's addresses: S. Currim, Alumni Office, University of Arizona; R. T. Snodgrass, Department of Computer Science, University of Arizona; Y.-K. Suh, (corresponding author, current address) Korea Institute of Science and Technology Information (KISTI); R. Zhang, (current address) Dataware Ventures.

DOI: 0000001.0000001

Table I. Measured time of ten executions of a query

	1	2	3	4	5	6	7	8	9	10	*Avg*	*Std Dev*
$Time_{meas}$ (msec)	9321	9210	9964	**13442**	9310	9470	**9206**	9394	9280	9398	9800	1298

repeatable results, and for which many sources of time variation (also to be discussed) were eliminated.

Even after taking those many proactive steps to improve the measurements, they still vary quite a lot, from 9,206 msec to 13,442 msec (with the lowest and highest numbers in **bold**), a range of 4,236 msec, which is over 30% of the highest time. As we will see, this variability arises from necessary daemon processes and their I/O and other interactions with the DBMS.

The *accuracy* of any measurement system is the "closeness of agreement between a measured quantity value and a true quantity value of a measurand" while the *precision* of that system is the "closeness of agreement between ... measured quantity values obtained by replicate measurements on the same or similar objects under specified conditions" [WORKING GROUP 2 OF THE JOINT COMMITTEE FOR GUIDES IN METROLOGY (JCGM/WG 2) 2008]. (In some contexts, accuracy is termed *external validity* and precision, *repeatability*.)

In the following, we address the central question just raised: how to achieve (more) precise and accurate measurements of query execution time? In considering the approach that is the norm, averaging ten runs, one asks, *why average?* (The average, while statistically well-founded, makes assumptions about the extraneous factors; we will look much more deeply into these factors.) In fact, *why not minimum?* (Using minimum assumes that all variations due to extraneous factors are additive; we will show that that is not the case.) *Should all ten times be used?* (We will show that some executions are outliers for quite specific reasons, as so should be dropped. We will then better motivate using ten executions.) *If some are dropped, how many should remain?* (We will provide a detailed rationale for dropping executions with specific properties.) *Could additional information from the operating system help?* (We will identify which information is useful in refining the determination of query processing time.)

We previously proposed a structural causal model and timing protocol [CURRIM, S. et al. 2013], which we term the *Tucson Timing Protocol Version 1* (TTPv1). The main limitation of that protocol is that it dropped about 20% of the timings because of *phantom processes*, those whose presence the protocol could detect but could not collect their measures. The longer the query ran, the higher the chance of a phantom process, effectively limiting the protocol to fast and moderately fast queries, and biasing the resulting measurements to such queries.

This paper presents a refined protocol, termed *Tucson Timing Protocol Version 2* (TTPv2), that (a) drops many fewer query executions, (b) adds many more sanity checks, (c) is based on an elaborated structural causal model, and (d) estimates process I/O in a more sophisticated manner, thereby providing a more comprehensive, robust, defensible measurement of query evaluation time and of process time generally.

The protocol chooses the median query time across the repeated query executions, for several reasons that we articulate in Section 8.5. As an example, examine query execution #4 in Table I. That one query execution raises the average to above all nine of the other executions. The median of 9357 is not affected by that one skewed value.

In this paper, we address both accuracy and precision in detail, along the way developing a comprehensive, carefully motivated query time measurement protocol. This protocol is much more accurate, given that it retains 96% of the query measurements, as compared with 76% for TTPv1 and more than doubles the number of sanity checks. (We emphasize that we do not have ground truth, and so are only stat-

ing that by not throwing away some of the longer-running measured queries, the refined protocol removes some of the systematic bias of the original protocol.) The refined protocol is also more precise, with a relative error for time measured of 2.3% and of time calculated of 2.1%, as compared to 4.5% and 2.2% for TTPv1.

In the next section we provide a taxonomy of time measurements. Section 3 considers many subtle aspects of measuring time: mitigating disk, main memory, and DBMS cache effects, ensuring data and plan repeatability, contending with O/S daemons, utilizing the wide range of Linux per-process and overall measures, and understanding the intricacies of time measurement resolution. We then present a comprehensive structural causal model for how the measures relate and test this model, finding that it is strongly supported by the experimental results. Section 7 first looks at the way we collect the measures and how we ensure that we haven't missed any process, then examines in detail the most complex part, that of computing the I/O time used by the DBMS. (Interestingly, Linux provides no direct measure of this highly relevant aspect, and even indirect approaches are quite complex.) We present the protocol in detail, emphasizing the many sanity checks along the way. Section 10 evaluates our approach against the existing protocol, showing that is more precise and accurate. We end with a summary and an examination of future work.

2. BACKGROUND

There is a spectrum of granularities with regard to what is being measured and how it is measured, as summarized in the taxonomy shown in Figure 1 (with the darker blocks being measured by our protocol). (We note in passing that there are alternatives to the root node of this tree, Query Time, such as DBMS startup time and query planning time. We don't consider those further.) The first decision is whether to consider time as an *independent variable* (that is, specified when the experiment is run) or as a *dependent variable* (that is, measured). The TPC-C benchmark is run for a user-specified length of time (minutes to hours), along with the transaction mix (ratio of read and update transactions) and the number of completed transactions is measured, yielding a measured *transactions per minute* [TPC 2010b]. We focus in this paper on measured query time, that is, as a dependent variable.

The second decision is, what is to be measured? This could be of a mix of transactions, each with one or more queries and updates, or a single transaction, or a single SQL statement (query, insert, delete, or update). We focus here on measuring the total time of a single query. Some of these measurements could also be made of databases in which the queries are run over many distributed computers, say in the cloud [COOPER, B. F. et al. 2010] or at a smaller scale, on a local distributed system [FORMAN, R. F. et al. 2001].

When measuring how much time an individual query running on a single server requires, one can look again at wall-clock time, which will include all the DBMS process(es), including those not actually evaluating the query, as well as operating system daemons and processes invoked by other users. The TPC-H benchmark [TPC 2010a] runs a host of queries over a prescribed database and measures total time for each, as do the XBench [YAO, B. B. et al. 2004] and τBench [THOMAS, S. W. et al. 2014] benchmarks. Or one can look more closely, restricting oneself to just those DBMS processes actually executing the query, or even to the time required for JDBC interaction, CPU execution, I/O, or network. One can measure I/O time, or obtain counts, such as the number of blocks read or written, perhaps differentiating between random and sequential disk I/O. One could also study the *pattern* of accesses, including differentiating synchronous from asynchronous I/O. For computation, one can also measure time or counts (such as number of CPU ticks). The same differentiation applies to measuring network activity. JDBC activity is generally composed of network ac-

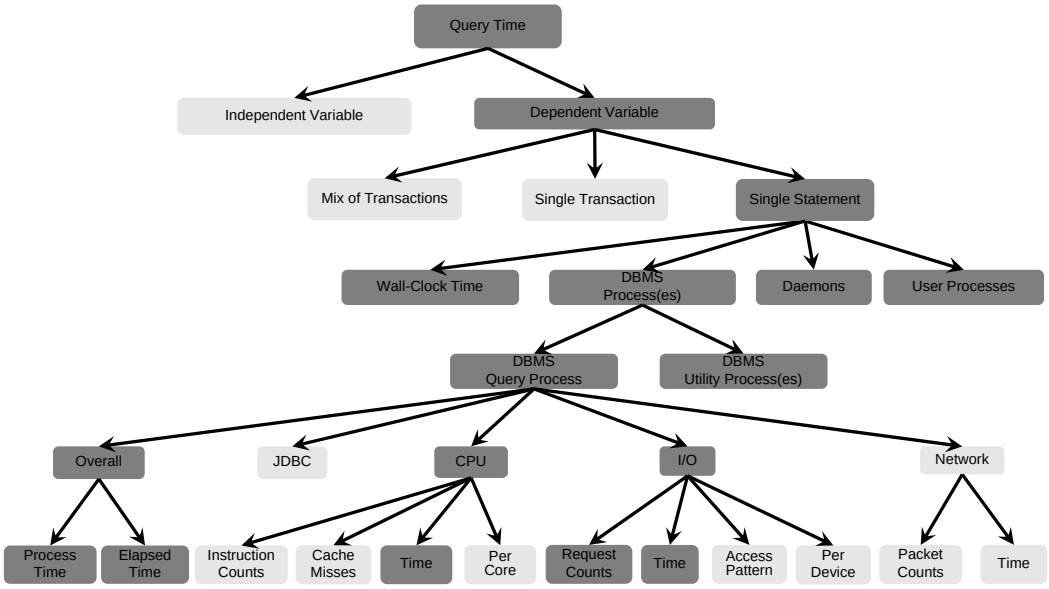


Fig. 1. DBMS time taxonomy

tivity (if the SQL statement initiator is running off-server) and computation. Finally, one can delve into the specifics of the CPU performance of a DBMS, examining for example processor cache effects [AILAMAKI, A. G. et al. 1999] using profiling tools like Valgrind and Callgrind [VALGRIND DEVELOPERS 2010], which provide instruction and cache hit metrics. Counts are generally collected either through the operating system or instrumented DBMS [ZHANG, R. et al. 2012] or by running a disk or cache simulation on the instrumented DBMS. Because they are gathered using different approaches, counts and time measurements can be compared to ensure that they consistent.

There are two types of overall time (the left-most measure in Figure 1, as contrasted with the *component* times to the right: JDBC, CPU, I/O and Network). The *process time*, abbreviated as PT, is defined as the sum of CPU, IO, JDBC, and Network time. The query time of our interest is the process time (PT) of the query process. The time difference between the start and the completion of the process is called *elapsed wall-clock time*, abbreviated as ET. Table I provides ET, which is by its very nature highly variable, because of all the other things going on while the query executes.

The present paper will consider how to more accurately and precisely measure the time required to execute a single SQL statement, examining (a) overall elapsed wall-clock time, (b) overall DBMS process time, (c) CPU time, and (d) I/O time. For understanding DBMS behavior, wall-clock time is highly variable due to extraneous operating system daemons and user processes, which is why we focus in this paper on the harder problem: finer-grained measurements of DBMS process time and its CPU and I/O components. (Doing so can then provide insight into the additive effect of daemons and user processes.) We will extract counts from the operating system but as we are measuring proprietary DBMSes, we will not consider approaches that require that the DBMS itself be instrumented. Thus, we do not consider CPU or I/O simulation to obtain detailed counts and measures of cache performance nor of random versus sequential I/O (we *will* consider overall I/O time). We do not focus on measuring network time; instead, we reduce network time to the absolute minimum by mounting the

Scenario (e.g., Exhaustive[1], OnePass[2])
Experiment (query set (Q1-Q11), data spec, scenario parameter(s))
DBMS (DB2, MySQL, Oracle, PostgreSQL)
Machine (`sodb8`–`sodb12`)
Experiment Run (on a particular date)
Query Instance (query number within query set)
Query at Cardinality (*Q@C*) (10K–2M)
Q@C Execution (*QE*) (1–10)

[1] Executing a plan at each cardinality as cardinality changes.
[2] Executing a plan only when the current plan is different from the previous one as cardinality changes.

Fig. 2. Hierarchy of Query-at-Cardinality (Q@C) executions

disks on the server (not using a network file server). We minimize JDBC activity by returning a minimal result.

We show that it is possible to deduce DBMS query process I/O and computation times, which when summed provides a much more stable measure of DBMS processing than wall-clock time. This allows us to isolate the contribution of DBMS query processing in terms of computation and I/O time, within the context of realistic execution. By comparing these measures to those of non-DBMS processes, we can also characterize the contributions of those other processes, thereby achieving a more comprehensive picture. Indeed, our approach allows one to accurately measure all of the variables in dark boxes in Figure 1.

Such measurements can be an initial step in broader studies, with these measures to be used as input to create more efficient query evaluation algorithms, to refine the query optimizer, such as its cost model (e.g., [GIKOUMAKIS, L. and GALINDO-LEGARIA, C. 2008; STILLGER, M. et al. 2001]), to predict query performance (e.g., [AKSWEW, M. et al. 2012; GANAPATHI, A. et al. 2009; HWANG, H.-Y. and YU, Y.-T. 1987]), to characterize workloads (e.g., [YU, P. et al. 1992]), or to do provisioning and capacity planning (e.g., [ZHANG, N. et al. 2011]).

Figure 2 presents a simple representation of the structure of our experiments, as a hierarchy of eight levels, ending at a particular query execution of a particular query at a particular cardinality for the underlying table(s), as part of a particular experiment run started on a stated date and time on a designated machine using a specified release of a specific DBMS, in the context of a specified experiment setup (stating the set of queries, the characteristics of the data, and various other parameters), of a selected experiment scenario. Once the scenario and experiment have been stated, the rest of the levels are instantiated when the experiment is run.

As an example of this hierarchy, we previously presented in Table I measurements of 10 executions for a query-at-cardinality (*Q@C*, that is, a particular query at a specified cardinality for a specified DBMS running on a particular machine within the context of a particular scenario and experiment conducted with a DBMS-centric research infrastructure called AZDBLAB [SUH, Y.-K. et al. 2014]. For the data in this table, we utilized the OnePass Scenario, an Experiment specifying a set (Q1) of 390 queries, specifying data with a maximum cardinality of 2M rows, decreasing by 10K rows at a time, and specifying the scenario parameter of 10 executions per Q@C. We ran this query on a proprietary DBMS, on the `sodb8` machine, from an experiment run started April 6, 2014 at 4:07pm, for query number 19, thus identifying a particular query instance running on a variable table with a cardinality of 1,010,000, and examining all ten Q@C executions. This paper concerns a total of 72 experiment runs, each taking a

Table II. Adjusting Number of QEs per Q@C

Number of Original QEs	Percentage Dropped	Number of Q@Cs Dropped	Number of QEs Retained
10	3.97%	2,207/55,585	9.5
9	4.55%	2,531/55,585	8.5
8	5.86%	3,260/55,585	7.6
7	10.31%	5,731/55,585	6.6

few days to a week on a single machine (with the disk drive humming the entire time), involving a total of 909,480 query executions (90,948 Q@Cs) over the DB2, MySQL, Oracle, and PostgreSQL DBMSes running on the Linux operating system, totaling over 9100 hours (over one year, 24x7) of cumulative time.

3. MEASURING QUERY TIME

We now turn our attention to the central problem: measuring in a defensible manner the execution of a Q@C. These considerations inform the development of our protocol.

As we will see shortly, there are many other activities in addition to the strict query time that is the focus here. Once a more precise and accurate query time is determined, those other aspects can also be measured either separately or in conjunction with the query (say, by introducing them individually). In that way, the full context and scope of a query can be quantified.

We execute in quick order, for a single query at a single cardinality, a certain number of Q@C executions (also termed *QEs*). In our protocol, we execute each Q@C 10 times, which is sufficient for timing purposes. As we'll see, Step 3-(iv) of the protocol, discussed in Section 8.4, drops Q@Cs with fewer than six query executions after many checks on these executions. (Why six QEs? Because that is the smallest sample size for which means and standard deviations make sense.) This step retained about 96% of the Q@Cs indicating that 10 is about the right number of query executions to start with. But this number of original QEs per each Q@C is a knob available to the experimenter. Table II shows the relationship that we observed between the number of original QEs, the number of Q@Cs dropped because they went below the threshold of six QEs after the protocol was applied, the number of Q@Cs actually dropped, and finally, for the remaining Q@Cs, how many QEs each retained, on average. It seems that for many situations, 8 QEs per Q@C would work fine.

From various low-level measurements gathered during these remaining multiple executions, we then compute a query execution time for that Q@C.

3.1. Cache Effects

Query evaluation normally involves intensive disk I/O, and so disk reads figure heavily in timings. However, if the DBMS cache is *warm*, say because a previous query read in data pages that are part of the next query, then those data pages in main memory will not need to be read back in, reducing overall I/O time.

If the DBMS main memory buffer is large enough to hold all the base tables and intermediate results, then the second time the same query is run, there should be no I/O at all. This can be termed *pure warm cache*. (As we will see, there are still issues in doing such timings, but at least no I/O is involved to complicate the timing further.) If the DBMS main memory cannot hold everything, then there may be some I/O required to evaluate the query, depending on the details of the DBMS and file system cache manager, which we term *partial warm cache*.

A further complication is that there are *multiple* caches involved: (a) most disk drives have a several MB cache, (b) many disk controllers also have a cache, which is typically larger than the disk cache, (c) the network file server generally has a large cache in main memory, (d) the operating system the DBMS runs on has a large cache

in main memory, (e) the DBMS buffer is generally large, (f) the JDBC driver may have a cache, and (g) the processor has a medium-sized L2 and a smaller L1 cache. We won't consider the processor L2 and L1 caches nor the JDBC cache further, as they are generally utilized in both warm cache and cold cache situations.

We focus in this paper on the cold cache scenario, as that is the most challenging, although of course our protocol can also be used to measure warm cache scenarios. (Many other DBMS papers mention cold-cache measurements. We discuss the details for two reasons: to be explicit on exactly how cold cache measurements can be done and to enable discussion of the efficacy (accuracy and precision) of our protocol.)

We eliminate the caching and buffering artifacts via four general steps.

First, we eliminated the problem of network file system caches by installing the disk directly on the machine that also runs the DBMS. For the other caches, we considered the Linux `hdparm -W` command, which allows the drive's write-caching feature to be set; the `-f` flag flushes the buffer cache. Note though that this feature only works on some drives. Instead, we filled both the disk and disk controller caches by reading a data file of size 64MB. The size of the data file was chosen such that it is bigger than the hard-drive buffer, which is 32MB. Third, we flushed the OS-related caches by using the Linux `drop_caches` facility to discard cached clean pages from the O/S file cache. As no dirty pages would be created during query execution, clearing the clean pages is sufficient.

We verified that these three steps were sufficient by timing the repeated reading of another file. Before the caches were flushed, the second read would take a much shorter time than the first read. After the cache was flushed, the second read took almost exactly the same time (modulo other timing variations, to be discussed shortly) as the first read.

Fourth, we cleared the DBMS buffer via the provided API, to discard cached content residing inside the DBMS. (In some cases, there were multiple caches to be cleared in the DBMS, each with a different kind of command. These caches were often not clearly documented; the commands were also sometimes hard to find.) For DB2, it is necessary to deactivate the DBMS cache for the `research` database with `sudo /opt/ibm/db2/v9.7/bin/db2 deactivate database research` followed by a similar `activate database research`. For Oracle, the SQL command is `ALTER SYSTEM FLUSH BUFFER_CACHE` followed by `ALTER SYSTEM FLUSH SHARED_POOL` followed by `ALTER SYSTEM CHECKPOINT`. For MySQL the command is simply `FLUSH TABLES`. PostgreSQL supports no such mechanism to clear all of its caches. PostgreSQL has two kinds of caches: `shared_buffer` (default 24MB, depending on the `shmmax parameter` under the kernel) and `work_mem` (default 1MB). The only way to clear these buffers is by restarting PostgreSQL. That said, because the shared buffer is so small, the effect of a warm cache on that DBMS is minimal.

There is an important philosophical point to be made here, one with implications in practice. Flushing the buffer cache may do other things, like close tables or reset other data structures. Doing so may require that table be opened, etc., when the query is (re-)executed, and so can include components that were not in the original query. Thus, by flushing, our approach is made more precise (repeatable, starting from the same state), but less accurate, in that we are no longer measuring just the query, but rather the query plus some additional management activities. In effect, our cold start is somewhat artificially cold.

In the end, the question comes down to, specifically what is desired to be measured? Our stance is that (a) a protocol should be thoroughly documented so that it is clear what is being measured, and how, (b) any deviations to the protocol should be explicitly stated in any paper that uses the protocol (we return to this in Section 9), and (c) it is

ultimately up to the researcher to utilize the (version of the) protocol that best applies to the issue at hand.

We tested each DBMS by running the query twice, verifying that its warm cache time was much smaller. We then cleared the O/S cache and found that the third time was still quite close to the second time, because of the DBMS cache. After then clearing the O/S and DBMS cache, using the facilities of each DBMS, we found that the fourth time was very close (modulo other vagaries) to the first time.

These steps in sequence, performed before each query is run, ensure a *pure cold cache*. All the pages needed to evaluate the query must be read from disk, according to the OS's and DBMS's specific buffer management policies. As we will see in Section 5, there remains a complex interaction between CPU execution and I/O that is challenging to tease out.

We now turn to two other, completely distinct caches that we must also contend with.

The second is *query plan caching*, in which a previously-determined plan is used when the query reappears. The first time the query is optimized, the DBMS may store the resulting plan in its plan cache, along with the original query. Then, if the query is given again, the DBMS just grabs the plan from its cache, rather than spending time reoptimizing. In contrast to data caching, query plan caching is a *good* thing, as we want to repeatedly time the same query.

A third type of caching is *query result caching*, in which the result (possibly many rows) of a query is stored, in case that same query is requested shortly thereafter. If query result caching is implemented in a DBMS, we would see it as a query that initially took longer than subsequent tries. (We run each query ten times at each desired cardinality.) However, we have to be careful: a simple query that requires a scan of the rows of a table may do the same I/O as one just reading the result from its cache. However, the computation time would definitely be somewhat less on all subsequent reads from the cache. Our protocol would select one of the faster QEs in this case, thus (unfortunately) reflecting the salutary effect of query result caching. Step 1 of the protocol (indicated in the second row of Table XI in Section 8.2) checks for this explicitly and then Step 3-(i) of the protocol (indicated in the fourth row in Table XIII in Section 8.4) discards any sequence of query executions that appears to follow this particular pattern.

3.2. Data and Plan Repeatability

Here we consider data repeatability, within-run plan repeatability, and between-run plan repeatability. All concern *precision*, the closeness of agreement between quantity values obtained by replicate measurements.

Below, we refer to "across-run" which indicates different "runs" of executions of the same query while "within-run" means different "executions" of the same query within the same run.

3.2.1. Data Repeatability. Queries are evaluated on several tables. Some of these tables are of fixed cardinality; for the rest, the cardinality is varied by the experiment scenario. In the following, we assume a single variable table.

Although the experiment tables are populated with randomly generated data, we ensure the same numbers are generated from a particular experiment on subsequent runs, by retaining the random number seeds in the collected data (as scenario parameters).

There are various ways to vary a table's cardinality. One is to start from an empty table and insert rows. Another is to start from a max-sized table and delete rows. It is important to configure things so that each time a cardinality is specified, the resulting variable table is identical in all respects, such as in terms of tuple order and page

packing. For example, when deleting rows from a table, the deleted rows are sometimes simply flagged without being physically removed. This can imply that different cardinalities have the same number of occupied pages, which is a factor considered by the cost model employed by the optimizers as well as the observed IO, which breaks an expected correlation between the number of rows and the number of pages. It is also helpful that the table loading be efficient, as it will be done repeatedly.

For the experiments discussed here, we wanted to start with a variable of maximal size, because inserting rows can be slow. But instead of deleting rows, we utilize the `SELECT ... INTO ...` command to populate the newly created table to the target cardinality. This approach effectively ensures that the cardinality of the table is coupled with the number of pages the table occupies while making table creation quite efficient.

3.2.2. Across-Run Plan Repeatability. When we ran each experiment multiple times to collect running time samples for statistical analysis, we observed that different plans resulted on the same query at the same cardinality (Q@C). We then tried a simple experiment: invoking the `EXPLAIN PLAN` facility with the same query at different times, sometimes one minute apart and sometimes hours apart. The underlying tables were of course not changed whatsoever. We found that the produced plans at various times were often different. We conjecture that this observed plan inconsistency is due to the randomness introduced by the heuristics adopted by DBMSes to compute the table statistics.

Given that it is difficult to ensure the across-run plan repeatability, we altered our approach. Instead of executing multiple runs for each experiment, each plan, once identified, is executed multiple times in close succession such that the samples are guaranteed to be collected on the same plan. We run the optimizer only once, using JDBC's `PreparedStatement` for each part of query and cardinality. We then execute the same query multiple times to obtain time values, such as those given in Table I.

3.2.3. Within-Run Plan Repeatability. We want to ensure that the query, at a particular cardinality, is run with the same plan. Ensuring so turns out to be impossible for extant DBMSes, so we need to approximate that check.

The standard approach in JDBC to execute a query is to use the `Statement.executeQuery()` method, which takes a query as a string argument. The central difficulty is that whenever a query is executed by the DBMS, it is free to reoptimize the query. Some query optimizers seem to be quite stable, but others, MySQL in particular, utilize stochastic optimization, or heuristic join orderings (e.g., used in PostgreSQL). Therefore, it is difficult to measure the *same* plan multiple times.

To ensure within-run plan repeatability, we use the `PreparedStatement` class in JDBC. The documentation [ORACLE CORP. 2014] states, that a `PreparedStatement` is "An object that represents a precompiled SQL statement. A SQL statement is precompiled and stored in a `PreparedStatement` object. This object can then be used to efficiently execute this statement multiple times." This class allows one to state the query in the constructor then call the `execute()` method, perhaps multiple times. It also permits placeholders within the query, which we didn't use.

So our plan was to instantiate a `PreparedStatement` object at the beginning of the Q@C and then reuse it ten times, using `execute()` within each QE.

The challenge was that to determine which plan was used, we needed to execute the `EXPLAIN PLAN FOR SELECT...` SQL statement. One includes the query in this statement and the DBMS returns a table that encodes the query plan as rows, generally as a preorder traversal of the tree structure of the query plan. That can be run within a `Statement` or `PreparedStatement`, but gives the plan resulting from the optimization at the time the statement was called.

Table III. Linux, Java, and processor clocks

Function	*Resolution*
`time` system call	second
Java `System.currentTimeMillis()`	millisecond
`gettimeofday` system call	microsecond
`clock_gettime` system call	nanosecond
`rdtsc` instruction	CPU cycles

What we would have liked to do is run `EXPLAIN PLAN` within the `PreparedStatement`, but that is not possible: the `execute()` function in that class takes no arguments! So it seems impossible to accurately determine the query used by any given execution.

So what we do is use a single `PreparedStatement` for each Q@C as well as a separate `Statement` to execute the `EXPLAIN PLAN` immediately after each query is run. When a DBMS reveals a different plan for a QE within the same Q@C, we restart the Q@C. In this way, we accomplish self-recovery without a user's intervention.

Another self-recovery is done when any exception (e.g., network disconnection or a runtime error) occurs during query execution. In this case, exponential backoff is performed, each successively doubling the backoff time. If the exception happens more than the ten-time threshold, then the ongoing experiment is automatically paused, and later can be unpaused by a user.

3.3. Operating Systems Daemons and User Processes

There are three other sources of I/O activity during query execution: other processes invoked by the DBMS process (we call these *utility processes*, operating system daemons, and user processes. As an optional step that can reduce measurement error, many of these processes can be stopped, as long as the process is not critical to the system's functionality. We discuss the details in an online Appendix A accessible in the ACM Digital Library. Later we'll see how to remove most of the timing artifacts from the remaining daemons and utility processes. (Note that the other daemons could instead be retained, at a slight increase in variability of the DBMS measures.)

Along with daemons, user processes can also be reduced. We isolated the DBMS server machines so that there is no keyboard nor mouse nor monitor attached. We used *VNC* to attach to the server machine remotely, and start an EXECUTOR process (our process doing the timing), which reads jobs from an external file (actually, a database accessible on a different machine via JDBC). This process records its progress to this *lab* database [SUH, Y.-K. et al. 2014], thereby minimizing interference from user jobs.

3.4. Wall-Clock Query Time

The Linux kernel provides several system calls that return the current time. (This method of measuring time is called *software monitoring*; it "is most suited for program-level measurements" [KANT, K. 1992].) The major difference among these functions is their measurement resolution, as shown in Table III. Note that the the `rdtsc` assembly-language instruction provided by the processor returns the number of CPU cycles since the last system reset. By using `rdtsc`, we are actually not measuring the execution time of the the operation, but the CPU cycles spent on this operation.

We use the Java method `currentTimeMillis()` which is based on the Linux `gettimeofday` system call. As we will see, milliseconds is actually a finer granularity than we will be able to achieve in the end, given all that is going on in a DBMS query.

Table I given on the second page of this paper is a pure cold cache measurement of query time using `currentTimeMillis` for 10 executions on a proprietary DBMS of the following query.

```
SELECT t2.id4, t3.id1, SUM(t2.id1)
FROM ft_HT1 t2, ft_HT1 t0, ft_HT1 t1, ft_HT4 t3
WHERE (t2.id2=t0.id1 AND t0.id1=t1.id1 AND t1.id1=t3.id4)
GROUP BY t2.id4, t3.id1
```

This query is on four correlation names, three of which reference the same table. `ft_HT1` (the variable table) contains 1,010,000 tuples, each with four integers; `ft_HT4` (one of the constant tables) contains 1 million tuples, also each with four integers. It turns out that `ft_HT1` has a primary key while `ft_HT4` does not.

In these measurements we have taken the following steps: (a) stopped as many operating system daemons as possible, (b) eliminated network delays by running the EXECUTOR process on the same machine as the DBMS and by using a local disk, (c) eliminated user interactions by having the EXECUTOR interact with an external lab DBMS to obtain the queries to be run and disallowing other user access, (d) ensured that the exact same query plan was being executed, on exactly the same database content, in exactly the same environment, to achieve data and within-run repeatability, and (e) ensured repeatability of I/O by clearing the many caches involved. Steps (a)–(c) improve accuracy while all five steps improve precision. (As we saw in Section 3.2, it is generally not possible to achieve across-run repeatability for a particular Q@C, because the query plan may change. Such variability must be accommodated in the experimental design and analysis.)

Most machines now have multiple cores, from 2 to 8 cores. As will be discussed in Sections 5 and 7, it is very difficult to get precise measurements for even a single core. Multiple cores are more complicated, as execution can continue as long as there are more unblocked processes as there are cores. Otherwise, one or more cores will be in an IOWait status for that tick. So we configured the Linux kernel to enable just one core, by adding `maxcpus=1` to the kernel arguments and verifying with `cat /proc/cpuinfo`. We applied this configuration to all the experimental machines. We note in passing that for all the DBMSes we measured, their default configuration limits query evaluation to just one process, so this is not a significant limitation, as for the great majority of the time the DBMS query process was the only one executing on the system.

As mentioned at the beginning of the paper, even when all of these other interactions have been eliminated to the extent possible, the measured times still vacillate a lot. For the Q@C shown in Table I, the measured query time varies from 9,206 msec to 13,442 msec, a range of 4,336 msec, which is 40% of the smallest time (and over 30% of the highest time). To reduce this variability, we need to use other information, specifically per-process and processor-wide measures provided by the O/S.

3.5. Per-Process Measures

The measures available to us from `/proc/`*pid*`/stat`, `/proc/`*pid*`/status`, and `/proc/`*pid*`/io` are (a) *min flt* [BOWDEN, T. et al. 2009], the number of minor page faults, those which do not require loading a memory page from disk, and so do not incur I/O, (b) *maj flt*, the number of major page faults, which cause the process to be blocked while that page is swapped in, thus incurring I/O, (c) *utime*, the number of ticks in which that process was running in user mode, (d) *stime*, the number of ticks in which a request from that process was being handled by the operating system, (e) *guest time*, the number of ticks in spent running a virtual CPU for a guest operating system (always 0 for our protocol), (f) *cguest time*, guest time of the process's children (always 0 for our protocol), (g) *number of voluntary context switches*, (h) *number of involuntary context switches*, (i) *blockio delay time*, the total time the process has been blocked on synchronous IO, (j) *number of read system calls*, and (k) *number of write system*

calls. These values are cumulative for each process, counting from 0 when the process was instantiated/forked; hence, these values are also accessed both before and after the query. We retrieve these statistics with a `getPerProc()` method within the protocol that itself takes about 80 milliseconds (on average, since the time is linear in the number of processes, with the the cost per process of about 485 microseconds, due to the large number of measures gathered, and with average QE having 171 processes). Note that thread synchronization costs are not recorded by the O/S, and thus cannot be measured in this way. There are also some other measures not relevant to timing, such as *pid*: the process id.

We will discuss in Section 7.1 the need for and use of the Linux NetLink facility (a system allowing user-space applications to communicate with Linux kernel), which is an alternative to `/proc/`*pid*`/stat` when a process terminates before the query finishes. (We will see a detailed example of the terminating process(es) in Section 7.1.) This facility provides a C struct called `taskstats` [ROSEN, R. 2014] (see also `https://www.kernel.org/doc/Documentation/accounting/taskstats-struct.txt`) which includes extensive per-process timing measures, most as counts in units of milliseconds, although some are given with nanosecond precision. This struct aligns fairly well with `/proc/`*pid*`/stat`. It omits many `/proc` measures that are no longer relevant, as the process has already stopped (e.g., state, session, tpgid, tty_nr, priority, num_threads, itreal_value, rsslim, startcode, endcode, startstack, kstkesp, kstkeep, signal, blocked, sigignore, sigcatch, wchan, exitsignal, processor, rt_priority, vsize, and policy) and some measures that are already included in minflt (but cmin flt in `/proc/`*pid*`/stat`), in majflt (reps. cmaj flt), rss (coremem), cutime (utime), cstime (stime), and cnswap (swapin_cnt). Finally, it drops two other measures (guest_time and cguest_time) which would be helpful to have in `/proc`, but only when virtual machines are involved (and thus, are always zero for our protocol).

The `taskstats` add a good many per-process statistics, such as number of IO requests. However, since we do not have IO requests for non-stopped processes, in particular, the query process, these are not useful. The other ones added by `taskstats` are not relevant to timing (e.g., hiwater_rss, which provides the high-watermark of RSS usage), so we don't discuss them here.

To conclude, our protocol uses nine per-process measures (three in both the model and protocol, four in just the model, and two just in sanity checks in the protocol), out of a total of 71 per-process measures available (most of which are not relevant to timing).

3.6. Overall Measures

There are a variety of overall measures that are available and related to per-process measures. The per-processor cumulative counts include (a) *utime*, the number of ticks in which a user process was executing, (b) *user mode with low priority* in ticks (in our data, always 0), (c) *stime*, the number of ticks in which the operating system was servicing a system call or interrupt, (d) *IOWait time*, the number of ticks in which the system had no processes to run because all were waiting for I/O (only available across processes), (e) *guest*, the number of ticks spent running a virtual CPU for guest operating systems under the control of the Linux kernel, and (f) *guest_nice*, the number of ticks spent running a niced guest (virtual CPU for guest operating systems under the control of the Linux kernel). (Note that the last, *guest_nice*, was added in Linux 2.6.33 and so our protocol captures this, though the experiments reported in this paper were run on Linux 2.6.32 (cf. Section 9) and so don't include that measure.)

There are other overall measures not related to per-process measures. The overall measures include (g) *idle time*, the number of ticks when the processor has nothing to do, (h) *IRQ* (interrupt requests) handled by the system, (i) *SoftIRQ*, the number of soft

interrupt requests (these requests can be handled with further interrupts enabled, to allow high-priority interrupts to still get in a timely manner), (j) *steal time*, the number of ticks spent in other operating systems when running in a virtualized environment (always 0 in our protocol), and (k) *processes*, the number of forks.

Note that all of these values are cumulative, counting from 0 since the last system rebooted. In our protocol, these values are accessed before and after the query is evaluated (by a `getOverall()` method within the protocol), with the first value subtracted from the second value to obtain the number performed during query execution. Each `getOverall()` invocation takes about 0.13 milliseconds, because there is only one set of these measures. (We note that `taskstats` mentioned earlier for processes that stopped during the query does not include any overall measures.) There are also other overall measures that we don't use (e.g., *page*, the number of pages the system paged in and the number that were paged out to disk).

We discovered that for some versions of the Linux operating system (e.g., Redhat Enterprise Linux 5.8), the *IRQ* overall measure is always zero. For a newer version, Redhat Enterprise Linux 6.4, the *IRQ* measure increases very slowly, just a few per hour, even when experiments are run continuously. This rate is sufficiently low that we don't consider such interrupts further. Hence, our protocol uses the eight overall measures (three needed in both the causal model and protocol, one used only in the model, and four used only for sanity checks in the protocol), out of a total of 20 overall measures available (many of which were not relevant). After discussing time resolution, we will use some of the 17 most relevant per-process and overall measures within a proposed causal model, to help ensure that we understand how these measures interact, and later (in Step 1, Section 8.2) use other of those measures within the protocol in a suite of 25 sanity checks, to ensure that everything went fine during data collection.

3.7. Time Measurement Resolution

Section 2 differentiated the related concepts of elapsed time (ET), CPU time (CT), and query time (CT + IO). Taking into account the available per-process and overall timing measures enumerated above, we use a respectively different resolution for CT and ET, depending on whether the process is in *mixed* or *pure-computation* mode,

With respect to the CT measurement of a process in mixed mode, the highest resolution we can use is tick. We could consider the resolution of microsecond available through the `taskstats` facility. However, there is a constraint on the use of `taskstats`. To get statistics of a process via `taskstats`, userspace needs to communicate with kernel space via a netlink socket [ROSEN, R. 2014]. The receive buffer for the netlink socket, however, is not robust when a number of processes' statistics are delivered. This concern leads us to limit the use of `taskstats`, only for (the few) processes that terminate in the middle of measuring query time. For the CT measurement of a non-terminating process, we, therefore, rely on the `proc` file system's tick-based per-process timing measures.

Regarding the ET measurement of the mixed process, the highest resolution we can use might be nanosecond, in that we can utilize a Java API, called `System.nanoTime()`, which returns the current time in nanoseconds. However, our timing protocol, to be discussed in Section 8, uses the `proc` file system's tick-based per-process timing measures. Note that one tick is one hundredth of a second (10 msecs), so nanosecond resolution is overkill. Therefore, we instead use millisecond resolution via `System.currentTimeMills()`, which returns the current time in milliseconds.

4. THE PRINCIPAL CHALLENGE

We have seen that there are a variety of per-process measures: user time, system time, and guest time measured in ticks and minflt and majflt measured in counts,

and a variety of overall measures: user time, user mode with low priority time, system time, idle time, IOWait time, and steal time measured in ticks and in number of interrupt requests (SoftIRQ) and processes measured in counts. Per-process measures are measured at individual process level, extracted from the output of `getPerProc()`. We define *per-type measures* as the aggregation of per-process measures for the DBMS query process(es) (this category is termed *query*), for the other DBMS processes (this category is termed *utility*), and for the remaining, operating system daemon processes (this category is termed *daemon*). (We also occasionally lump the second and third categories into what we call the *non-query processes*.) The goal is how to infer/calculate the total time used by the query process(es) to actually execute the query. The task before us is to allocate the appropriate portion of overall time to the query processes, using if possible the per-process and overall times and counts in this allocation. (We assume that the query is executed in exactly one process, the query process, and that DBMS utility processes perform activities unrelated to the query.)

As noted above, steal time and user model with low priority time are always 0. We already have counts for user and system time for those DBMS process(es). The challenge is to estimate the portion of (overall) IOWait time directly or indirectly caused by activities of the DBMS processes.

In order to get a handle on what is happening inside the operating system as processes compete for resources, we previously proposed a simple structural model [CURRIM, S. et al. 2013] over some of the measures just discussed. We now considerably elaborate that model to provide extended coverage of factors that weren't considered in that earlier paper. (Foreshadowing: we will use the insights from this model in Section 8.5 to come up with a better means of calculating query I/O time.)

5. AN ELABORATED CAUSAL MODEL

As with the original model, this new, elaborated model differentiates the DBMS query processes, which dominate the time and which is known to have significant I/O and CPU components, and is given in Figure 3(a), from the non-query processes (utility, daemon, and user processes), which contribute much shorter run times and for which it is not known whether they have significant I/O or CPU components, given in Figure 3(b).

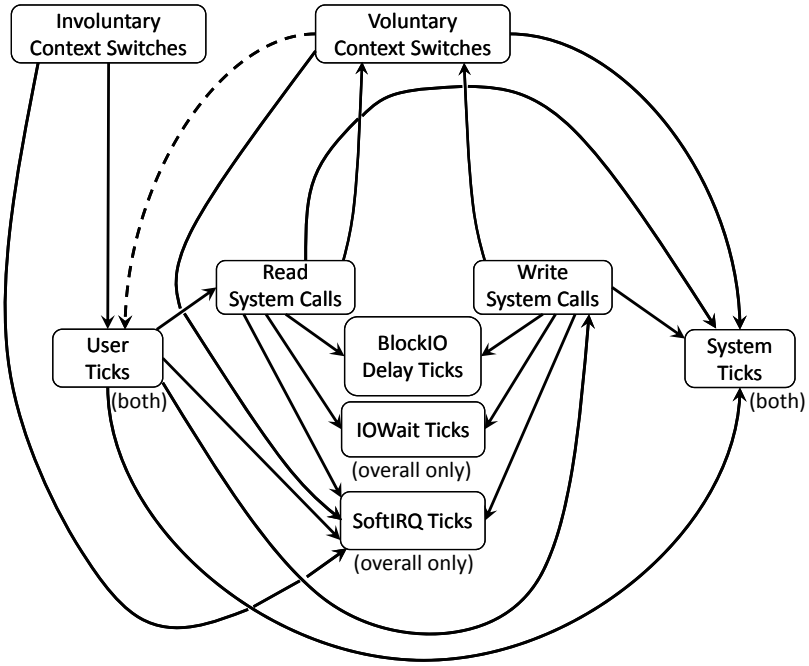
5.1. Considered Variables

Within this composite model, nodes are variables to be measured and directed arcs hypothesize causal relationships. Tucson Timing Protocol Version 1 (TTPv1) [CURRIM, S. et al. 2013] included just 13 measures in all. Because we have expanded considerably the number of per-process measures, we include in the elaborated model almost twice as many factors, in order to better understand how the DBMS process interactions with other processes and with the operating system.

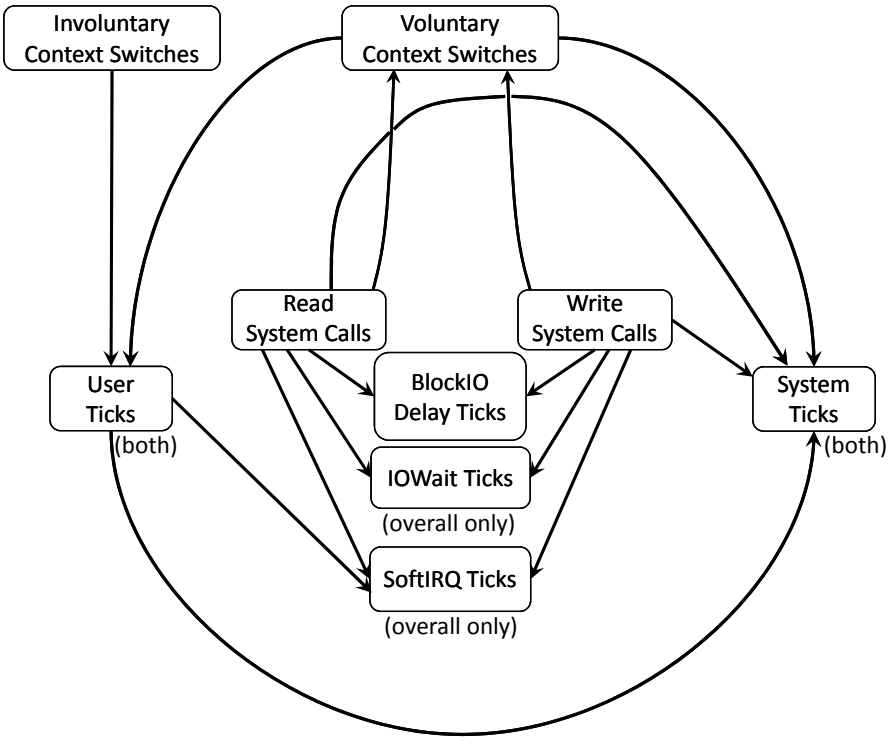
Most variables express a number of ticks, that is, time (e.g., "User Ticks," "SoftIRQ Ticks"); the rest express counts (e.g., "Involuntary Context Switches," "Read System Calls"). Two variables, SoftIRQ Ticks and IOWait Ticks, are overall measures. These are marked "(overall only)", and so represent the same values in both sub-models. User Ticks and System Ticks are both per-process and overall measures and so are marked "(both)". For the other five measures (involuntary and voluntary switches, read and write system calls, and BlockIO Delay ticks), we have only per-process measures.

5.2. Relationships

Directed arcs between nodes denote causal relationships. The box labeled "User Ticks" in Figure 3 is a construct with two variables, User Query Ticks (that for the particular DBMS process) and Overall User Ticks (an overall measure), and similarly for



(a) Model for the DBMS query process



(b) Model for the non-query (utility, daemon, and user) processes

Fig. 3. Structural Causal Model

"System Ticks." So the line from User Ticks to System Ticks in Figure 3 actually concerns three relationships: (i) overall user ticks to overall system ticks, (ii) query user ticks to overall system ticks, and (iii) query user ticks to query system ticks. In Figure 3(b), this construct has *four* variables: Daemon User Ticks (sum of user ticks for all daemon processes), Utility Process User Ticks (sum of user ticks for all DBMS utility processes), User Process User Ticks (sum of user ticks for all user processes), and Overall User Time (an overall measure). Figure 3(b) concerns many relationships: daemon user ticks to daemon system ticks, utility user ticks to utility system ticks, user user ticks to user system ticks. daemon user ticks to overall system ticks, utility user ticks to overall system ticks, and user user ticks to overall system ticks. We enumerate all the relationships implied by this causal model below (73 in all, over 25 separate measures).

The intuition behind this model is that the DBMS in its normal processing (measured by user ticks) reads data from the database, expressed as a system call to read in the block. That system call incurs its own system time, additional BlockIO Delay ticks, and additional interrupt requests, some of which are soft IRQs. If all processes are blocked on I/O, that request could add to the (overall) IOWait ticks (we delve into the intricate details in Section 7.3). These predicted causal relationships are all in the positive direction. For example, if user ticks increase for a particular query, it is predicted that the number of IO requests (query process Read or Write System Calls) might increase, which could itself increase (overall) SoftIRQ ticks, (overall) IOWait ticks, query BlockIO Delay ticks, and amount of system time (query and overall). The DBMS presumably does other things that require system time.

As timer interrupts are in fact soft interrupts, user ticks directly affects SoftIRQ ticks as well.

Each read or write system call that requires data to be read from disk, will likely induce a context switch if there is another process scheduled, hence the number of context switches is directly correlated with such system calls. The number of voluntary context switches directly increases the time spent in system mode, hence increasing the system time.

We also predict a causal relationship between the number of context switches (voluntary and involuntary) and user ticks. Consider the following situation: the DBMS process blocks on an IO request between two timer interrupts and the scheduler decides to run a daemon process. When the next timer interrupt comes in, the entire tick will be attributed to that daemon process. When the daemon process finishes its work and goes to sleep, the processor will switch back to the DBMS process. Again, the entire tick will be attributed to the DBMS process, even though it may run for less than one tick. If this happens often, then user ticks will accumulate spurious time. Hence, we have a direct correlation between the number of voluntary context switches and user ticks and between the number of involuntary context switches and user ticks.

Finally, since an involuntary context switch preempts the query process and goes to the interrupt handler, that may result in SoftIRQ tick(s). A voluntary context switch could have the same result. Hence, we include both causal relationships for query processes. Note that we don't consider relationships between two different groups of processes, nor from query Voluntary Context Switches to query User Ticks, as that induces a loop: query User Ticks to query Read System Calls to query Voluntary Context Switches, to query User Ticks.

Figure 3(b) proposes a similar causal model for all but the query process (that is, utility, daemon, and user processes), with six causal relationships omitted: Daemon User Ticks, Utility User Ticks, and User User Ticks to Read System Calls and to Write System Calls. Even though most of the obvious I/O-bound daemons are turned off,

some I/O-bound daemons cannot be eliminated, such as `kjournald` and `pdflush`. On the other hand, some Linux kernel daemons are non-I/O bound. The same observations hold for utility processes and user processes. Hence, we don't know whether these non-query processes cause system time or context switches. For instance, it is unclear if they require memory to be allocated or not. Similarly, we don't know whether these processes cause significant I/O requests, soft or otherwise. We cannot summarize in general what execution pattern these non-query processes have, and therefore, in Figure 3(b), there is no causal link predicted between any per-process User Ticks and any measure other than overall and per-process System Ticks. (It might be possible to utilize Read and Write System calls to characterize a non-query process as heavily I/O, thereby allowing it to be included in the query model rather than the non-query causal model.) There is no causal interconnection between user ticks and read and write system calls because we don't know whether such processes even do much I/O.

5.3. Predicted Correlations

Because we did not include user processes in our study, this causal model relates 25 specific measures. The overall model makes 73 specific predictions of correlations between pairs of measures, which we will now enumerate, motivating the strength of the correlation we expect to find for each. (Recall that the TTPv1 causal model made only 27 specific predictions; this refined model is thus much more comprehensive.) We identify an expected correlation of 0.7 or greater as a "high" level of correlation, from 0.3 to 0.7 (exclusive) as "medium" and those below that as a "low" level of correlation. Each correlation is positive: when one variable increases in value, we expect the other variable to increase in value.

We group these expected correlations that act in similar ways in Table IV. In each box is a relationship between two measures and a predicted level. (In this table, "Vol. Con. Sw" is an abbreviation of Voluntary Context Switches.) We designate such pairs by group and letter within group. An example is correlation (*Ia*), the box at the top left, which concerns the relationship between query user ticks and query system ticks.

In general, we predict correlations between two different directly-linked measures for the same type of process (e.g., (*Ia*)) to be medium to high (as will be explained shortly). We predict overall correlations based on the strength of the associated per-type correlations. For relationships involving a per-type and overall of different directly linked measures (e.g, (*VIIa*)), as well as part-whole relationships (between a per-type and the overall operationalization of the same construct, e.g., (*XIVa*)), we differentiate between query and non-query measures, because query processing dominates the computation. If the measure is only overall (that is, IOWait Ticks and SoftIRQ Ticks, e.g., (*VIIa*)), we do a case-by-case analysis.

The first set (Groups I–VI) involves correlations between two different measures, but for the same type of process which can be either query (Groups I and II), utility (Groups III and IV), or daemon (Groups V and VI). All such pairs are expected to be highly correlated for query relationships, but only of medium correlation for the utility and daemon relationships, because we know much less about what those processes actually do. Recall that the query processes do not include one relationship: voluntary context switches to User Ticks and that the utility and daemon processes do not include two relationships: User Ticks to Read System Calls and User Ticks to Write System Calls. This group includes 31 predicted correlations.

The next set (Groups VII–XII) involves correlation between the per-type measure and overall of a directly-linked different measure. The query process dominates the overall measure, but there are other things going on, so there will be some noise in overall; hence we expect medium correlation for such relationships. For utility and

Table IV. Hypothesized Relationships and Hypothesized Levels

	a	*b*	*c*	*d*	*e*	*f*	*g*
I	query UTicks/ query STicks *high*	query UTicks/ query Read SysCalls *high*	query UTicks/ query Write SysCalls *high*	query Read SysCalls/ query BlockIO Delay *high*	query Read SysCalls/ query Vol. Con. Sw *high*	query Read SysCalls/ query STicks *high*	query Write SysCalls/ query Vol. Con. Sw/ *high*
II	query Write SysCalls/ query STicks *high*	query Write SysCalls/ query BlockIO Delay *high*	query Vol. Con. Sw/ query STicks *high*	query Invol Cont./ query UTicks *high*			
III	utility UTicks/ utility STicks *med*			utility Read SysCalls/ utility BlockIO Delay *med*	utility Read SysCalls/ utility Vol. Con. Sw *med*	utility Read SysCalls/ utility STicks *med*	utility Write SysCalls/ utility Vol. Con. Sw/ *med*
IV	utility Write SysCalls/ utility STicks *med*	utility Write SysCalls/ utility BlockIO Delay *med*	utility Vol. Con. Sw/ utility STicks *med*	utility Invol Cont./ utility UTicks *med*	utility Vol. Con. Sw/ utility UTicks *med*		
V	daemon UTicks/ daemon STicks *med*			daemon Read SysCalls/ daemon BlockIO Delay *med*	daemon Read SysCalls/ daemon Vol. Con. Sw *med*	daemon Read SysCalls/ daemon STicks *med*	daemon Write SysCalls/ daemon Vol. Con. Sw *med*
VI	daemon Write SysCalls/ daemon STicks *med*	daemon Write SysCalls/ daemon BlockIO Delay *med*	daemon Vol. Con. Sw/ daemon STicks *med*	daemon Invol Cont./ daemon UTicks *med*	daemon Vol. Con. Sw/ daemon UTicks *med*		
VII	query Read SysCalls/ overall IOWait Ticks *med*	utility Read SysCalls/ overall IOWait Ticks *low*	daemon Read SysCalls/ overall IOWait Ticks *low*	query Write SysCalls/ overall SoftIRQ Ticks *med*	utility Write SysCalls/ overall SoftIRQ Ticks *low*	daemon Write SysCalls/ overall SoftIRQ Ticks *low*	
VIII	query Write SysCalls/ overall IOWait Ticks *med*	utility Write SysCalls/ overall IOWait ticks *low*	daemon Write SysCalls/ overall IOWait Ticks *low*	query Read SysCalls/ overall STicks *med*	utility Read SysCalls/ overall STicks *low*	daemon Read SysCalls/ overall STicks *low*	
IX	query Read SysCalls/ overall SoftIRQ Ticks *med*	utility Read SysCalls/ overall SoftIRQ Ticks *low*	daemon Read SysCalls/ overall SoftIRQ Ticks *low*	query Write SysCalls/ overall STicks *med*	utility Write SysCalls/ overall STicks *low*	daemon Write SysCalls/ overall STicks *low*	
X	query UTicks/ overall STicks *med*	utility UTicks/ overall STicks *low*	daemon UTicks/ overall STicks *low*	query Vol. Con. Sw/ overall STicks *med*	utility Vol. Con. Sw/ overall STicks *low*	daemon Vol. Con. Sw/ overall STicks *med*	
XI	query Invol. Cont./ overall UTicks *med*	utility Invol. Cont./ overall UTicks *low*	daemon Invol. Cont./ overall UTicks *low*	query Vol. Con. Sw/ overall UTicks *med*	utility Vol. Con. Sw/ overall UTicks *low*	daemon Vol Con. Sw/ overall UTicks *low*	
XII	query UTicks/ overall SoftIRQ Ticks *med*	utility UTicks/ overall SoftIRQ Ticks *low*	daemon UTicks/ overall SoftIRQ Ticks *low*	query Vol. Con. Sw/ overall SoftIRQ Ticks *med*	query Invol Cont./ overall SoftIRQ Ticks *med*		
XIII	overall UTicks/ overall STicks *med*						
XIV	query UTicks/ overall UTicks *high*	utility UTicks/ overall UTicks *low*	daemon UTicks/ overall UTicks *low*	query STicks/ overall STicks *high*	utility STicks/ overall STicks *low*	daemon STicks/ overall STicks *low*	

daemon processes, we expect only low correlation, for the same reason: the query process dominates the overall measure. This group contains 35 predicted correlations.

The next set (Group XIII) also involves a correlation between two different measures, but for overall, expected to be correlated if all three per-type correlations in the first

group are predicted: (*XIIIa*) is predicted to be medium due to (*Xa*), (*Xb*), and (*Xc*) and because query dominates.

(We previously investigated correlations between a per-type measure and an *indirectly-linked* different per-type measure [CURRIM, S. et al. 2013]. An example is query user ticks to voluntary context switches. However, those correlations are mathematically implied by the direct relationships, as we have no latent variables, and so we don't include those here. The same holds for indirect correlations between two overall measures.)

The final set (Group XIV) involves part-whole relationships, that is, between the query component and the overall operationalization of the same construct. As we expect the query process dominate the overall, the correlations involving the query process will be high and otherwise low. This group contributed six expected relationships, so the total is 73.

6. TESTING THE CAUSAL MODEL

Table IV hypothesizes a total of 73 testable correlations. Our experiments provide a large amount of data that can be used to test this model, which we do now. Then, in Sections 7 and 8, we will use this model to apportion I/O wait time to the DBMS processes and other processes.

Each testable correlation in Table IV lists two measures and an expected correlation. Let's examine the very first relationship, *Ia*, in the top left corner, between query user ticks and query system ticks, with a high correlation expected. When we look at the data, each DBMS exhibited a different range of possible user ticks (some DBMSes are faster than others) and thus a different range of possible system ticks, for the queries we ran at the cardinalities we used (Q@C). For example, in the experiment, for one DBMS, the query user ticks went to a maximum of 704 ticks while another experienced a maximum of 80,760 ticks. Examining query system ticks, one DBMS experienced at most 2 ticks while another experienced in one case 3,107 ticks.

Within each DBMS, even when there was high correlation, the combined data set sometimes exhibited a much lower correlation due to a difference in scaling across both variables for each DBMS. To enable a more appropriate analysis, we transformed each variable in the data set of each DBMS using a common feature scaling transformation: $(x - min)/(max - min)$. This transformation has the property that the transformed variable ranges over $[0, 1]$ with the same per-DBMS correlation between them, thereby allowing us to compute a more representative overall correlation.

6.1. Exploratory Model Analysis

We tested the model in two phases. In the first, *exploratory model analysis*, we ran a correlation analysis on a small number of query runs, eight in all, requiring a total of 1188 hours of cumulative time; see Table V. We then examined our assumptions against the results of this analysis and revised the model where needed.

Table V. Statistics for the Exploratory and Confirmatory Experiment Runs

	Exploratory	Confirmatory
Number of Experiment Runs	8	36
Number of Query Instances	1200	4,800
Number of Q@Cs	8,740	55,585
Number of QEs	87,400	555,850
Number of cumulative hours	1,188	6,184

The main changes suggested by this phase were to separate consideration of the query process, which we felt we understood much better, from the non-query processes, which we understand poorly. (Note that this paper is focused on measuring the query time, that is, the execution time of just the query process.) As a result, an initial single causal model was refined to the two-part model depicted in Figure 3.

Another aspect highlighted by the exploratory analysis was the role that major faults play in the model. The number of major faults generally is quite low, and almost non-existent for the query process. This made sense in retrospect, as the query code will have been swapped in at the beginning of the experiment and repeated executed. So we removed this measure in our model.

Note however that although we can measure the number of chars and bytes read and written through the `proc` file system [FAULKNER, R. and GOMES, R. 1991], we didn't include those variables in our model. In our exploratory analysis, we found that these measured values of the variables were very different from the expected ones. Specifically, we found that for query processes, the number of bytes read (`qp_rbytes`) per QE was almost zero across DBMSes, an unexpected result. (Note that as previously discussed, we flush the disk drive, OS, and DBMS caches before running the query, to effect a cold cache.) According to the Linux kernel documentation, the number of bytes read (`read_bytes`) is defined as an "attempt to count the number of bytes which this process really did cause to be fetched from the storage layer" [BOWDEN, T. et al. 2009] (A similar definition is given to the number of bytes written.)

For one DBMS, the number of chars read (`rchar`) of its query process was always zero. The kernel documentation defines `rchar` as "the number of bytes which a process has caused to be read from storage (although the read might have been satisfied from pagecache)" [BOWDEN, T. et al. 2009]. The documentation also says that this number is simply the sum of bytes which the process passed to `read()` or `pread()`. (A similar definition is given to the number of chars written, too.)

We wondered if the sum is actually equal to the number passed to `(p)read()`. To see if this is the case, we ran an experiment: a simple Linux C program, which opens a 200-byte file containing 200 characters exactly, reads the whole file, and subsequently writes the read 200 characters on console. We ran this program with our protocol. We expected this simple program's measured values of the number of chars and bytes read and written to be all positive numbers, that is 200. Indeed, the measured number of chars read was 1956, but the other three variables' values were all zeros. Furthermore, we also tried to check the validity of `rchar`'s values on one of the DBMSes by running an aggregate self-join query on a table loaded with 30K tuples. As a result, the query process' number of chars read was 575, and the other three variable's values were all zeros as well.

In summary, we doubt that these particular `proc` file variables are accurate so we do not include them in our model.

Finally, we observed within exploratory model analysis an unexpectedly high correlation of query Read System Calls with query Write System Calls. In thinking about this further, it seems that this could be due to DBMS reads being immediately followed by writes as the DBMS writes out rows to a temporary table on disk and then reads back that temporary table. We then looked at very simple queries such as a scan of one table, where that would not happen; for the few such queries we examined, that correlation was not present. We decided not to add this relationship to the model, because it was somewhat query specific. We did not find any other unexpected high correlations in exploratory analysis.

Table VI. Hypothesized Relationships and Observed Levels

	a	*b*	*c*	*d*	*e*	*f*	*g*
I	*high*	*high*	*high*	*high*	*high*	*high*	*high*
	0.372	0.402	0.357	0.430	0.480	0.920	0.549
II	*high*	*high*	*high*	*high*			
	0.898	0.579	0.647	0.651			
III	*med*			*med*	*med*	*med*	*med*
	0.775			0.685	0.843	0.813	0.502
IV	*med*	*med*	*med*	*med*	*med*		
	0.458	0.871	0.924	0.776	0.875		
V	*med*			*med*	*med*	*med*	*med*
	0.462			0.216	0.401	0.612	0.434
VI	*med*	*med*	*med*	*med*	*med*		
	0.747	0.283	0.537	0.699	0.303		
VII	*med*	*low*	*low*	*med*	*low*	*low*	
	0.442	0.784	0.138	0.692	0.206	0.323	
VIII	*med*	*low*	*low*	*med*	*low*	*low*	
	0.582	0.472	0.579	0.913	0.415	0.401	
IX	*med*	*low*	*low*	*med*	*low*	*low*	
	0.698	0.385	0.237	0.944	0.223	0.482	
X	*med*	*low*	*low*	*med*	*low*	*med*	
	0.417	0.415	0.300	0.665	0.445	0.901	
XI	*med*	*low*	*low*	*med*	*low*	*low*	
	0.653	0.135	0.286	0.518	0.180	0.439	
XII	*med*	*low*	*low*	*med*	*med*		
	0.325	0.368	0.140	0.459	0.122		
XIII	*med*						
	0.437						
XIV	*high*	*low*	*low*	*high*	*low*	*low*	
	0.991	0.165	0.193	0.953	0.432	0.608	

6.2. Confirmatory Model Analysis

The result of exploratory data analysis described above was the structural causal model presented in Figure 3 in Section 5, which engenders the specific set of 73 predicted correlations among these measures discussed in Section 5.3 and summarized in Table IV. As discussed in Section 5.3, we identify the expected strength of each correlation by three rough levels: an expected correlation of 0.7 or greater was identified as high, from 0.3 to 0.7 (exclusive) as medium, and those below that as a low level of correlation.

We then transitioned to *confirmatory model analysis*, in which we did a correlational analysis of a comprehensive, independent data set of 36 query runs, over five times as large as the exploratory analysis, requiring 6184 hours of cumulative time; see Table V. With this empirical data, we performed a comparison of the actual level of correlation (based on the Pearson correlation coefficient) for each of the relationships in question with their level predicted by our model. The results are given in Table VI. It is important to note that these correlations are over four disparate relational DBMSes, implemented independently by separate teams of developers.

All the relationships were significant (having $p < 0.05$) and all were in the predicted direction (a positive correlation). In general, we found that the level predicted by our model either exactly matched that of the actual level (e.g., predicted high and actual high, for (*If*)) for 35 relationships, indicated with green, or was close (e.g., predicted high and actual medium, for (*Ia*)) for 37 relationships, indicated with yellow. (Of those 37 relationships, most, 25, had correlations *higher* than predicted.) For (*VIIb*), the one

red cell, which we expected a low correlation, due to a relationship between a per-type measure and overall of a directly-linked different measure, we actually observed a high correlation, which is not of much concern.

Our conclusion is that the model is strongly supported by these experimental results.

We now delve into the details of the timing protocol, which is fundamentally informed by this causal model, in that we now know how intricately related the various measures are, as well as which time measures are most related to I/O activity.

7. TIMING CONSIDERATIONS

In the following, we discuss general data collection, then consider the intricacies of process interaction, and finally show how the data that was collected, in concert with the causal model shown in Figure 3, can be used to ascribe the portion of I/O time utilized by DBMS query evaluation. We then turn to the protocol itself, in Section 8.

7.1. Process Interactions

We now consider how individual processes interact with each other and with the measurement protocol. These interactions significantly complicate the timing of queries.

Let's start by examining the process that collects the measures, termed the EXECUTOR. (The EXECUTOR process is a part of AZDBLAB.)

Because the time required to collect the per-process metrics (via `getPerProc()`) is much longer than that taken to collect the overall metrics (via `getOverall()`), the EXECUTOR (a Java program) performs measurements in the following order: `getPerProc()`→`getOverall()`→`getTime()`→*execute query*→`getTime()`→`getOverall()`→`getStopProc()`→`getPerProc()`, after which EXECUTOR computes and records measured time along with the collected process information into database. Later, the protocol machinery retrieves the stored process information from the database, parses it, and then computes and stores back into the database the differences in the cumulative statistics.

Figure 4 provides a timeline (moving left to right) of both the protocol and various process interactions with the data collection. We mention in passing that we include a WATCHER java process. EXECUTOR sends heartbeat messages to WATCHER every time a task (a query) is completed. WATCHER alerts us by email if such heartbeats go silent. Hence, such heartbeats are totally outside the figure and the WATCHER process is just a utility process that is in a blocked state for the entire query evaluation.

In the figure, the EXECUTOR process is the top horizontal line; it runs the entire time. The first thing that EXECUTOR does is invoke the PROCMONITOR process (a C program), denoted by a downward arrow from the top line to the second horizontal line. The last interaction at the far right is a message sent by PROCMONITOR process back to EXECUTOR shown as an upward arrow from the second line to the top line, followed by the EXECUTOR storing data about this execution in the protocol DBMS.

Processes $P0$-$P13$ and Px, Py, and Pz are illustrative to show how other processes may co-exist (or not) with the query execution process.

The solid vertical lines signify relevant events. In order, they are as follows.

(1) EXECUTOR starts the PROCMONITOR process, which collects `taskstats` data.
(2) PROCMONITOR starts the *ProcListen* thread, which will receive `taskstats` data.
(3) PROCMONITOR sends a message back to the EXECUTOR: ready to start.
(4) EXECUTOR invokes the `getPerProc()` Java function, which collects per-process measures in /`proc`/*pid*/`io`, /`proc`/*pid*/`stat` and /`proc`/*pid*/`status` for each process identified by *pid*.
(5) `getPerProc()` returns.

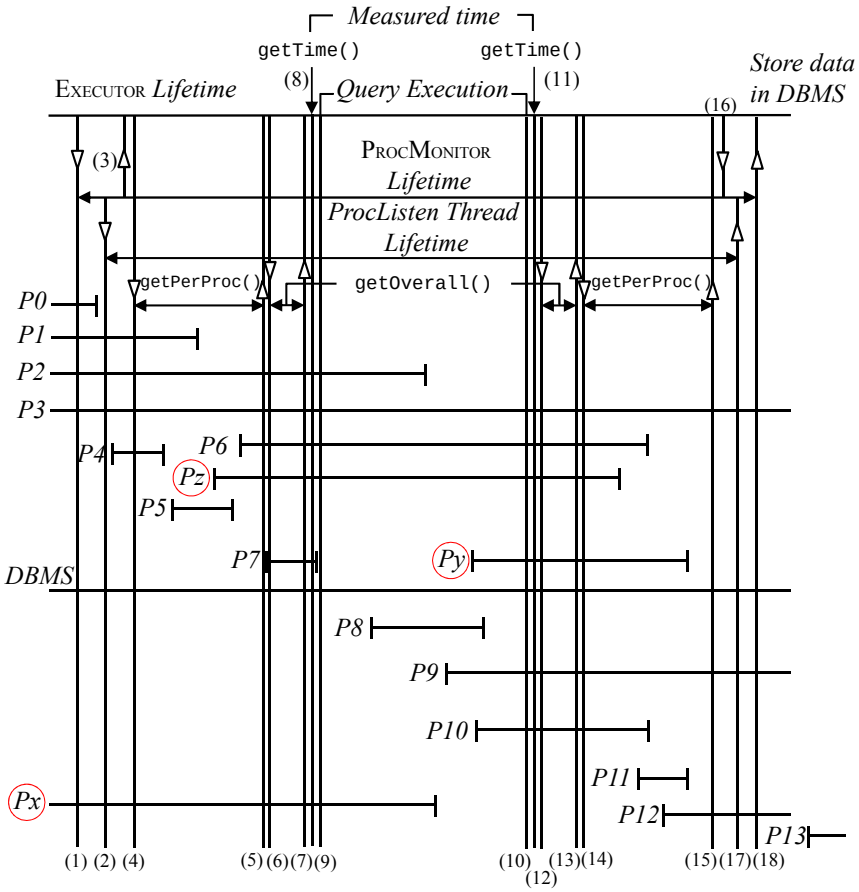


Fig. 4. Processes considered when timing a query

(6) EXECUTOR invokes the `getOverall()` Java function, which collects overall measures in `/proc/stat`.
(7) `getOverall()` returns.
(8) EXECUTOR invokes the `getTime()` function, which takes a few microseconds.
(9) EXECUTOR starts the query.
(10) The query completes.
(11) EXECUTOR invokes the `getTime()` function, which takes a few microseconds.
(12) EXECUTOR invokes the `getOverall()` Java function, which collects overall measures in `/proc/stat`.
(13) `getOverall()` returns.
(14) EXECUTOR invokes the `getPerProc()` Java function, which collects per-process measures for each process.
(15) `getPerProc()` returns.
(16) EXECUTOR sends a message to PROCMONITOR, which stops the *ProcListen* thread: get stopped `taskstats` data.
(17) The *ProcListen* thread sends the `taskstats` data back to PROCMONITOR and terminates.

Algorithm timeSingleQE(*iterNum*, *query*, *prevPlan*, *card*):
plan ← $getQueryPlan$(*query*)
if *plan* ≠ *prevPlan* **then**
  Error: two different plans are observed at the same cardinality.
end if
{Only if cold cache:}
Clear disk, OS and DBMS caches.
$perProcStat_1$ ← `getPerProc()`
$ovrProcStat_1$ ← `getOverall()`
$time_1$ ← `getTime()`
`executeQuery(`*query*`)` {using `PreparedStatement`}
$time_2$ ← `getTime()`
$ovrProcStat_2$ ← `getOverall()`
stopProcStat ← `getStopProcs()`
$perProcStat_2$ ← `getPerProc()`
$qeProcStat \leftarrow perProcStat_1 \cup ovrProcStat_1 \cup perProcStat_2 \cup ovrProcStat_2 \cup stopProcStat$
$Time_{meas} \leftarrow time_2 - time_1$
`recordQueryExecution(`*iterNum*, *query*, *card*, $Time_{meas}$, *qeProcStat*`)`

Fig. 5. Algorithm for the single query execution time measurement

(18) PROCMONITOR terminates, at which time the EXECUTOR stores the data in the protocol DBMS.

Online Appendix B explains how to discover *ephemeral* processes, those that were somehow not seen by the *ProcListen* thread. We expect that for each query execution, ephemeral processes almost never are encountered, as will be discussed in Section 8.2 (QE sanity check 1).

Figure 5 provides the algorithm for the single query execution time measurement. In the algorithm, the input arguments, specifically *iterNum*, *query*, *prevPlan* and *card*, denote the current iteration number for repeatability check, a given query, the previously detected query plan and the current cardinality at which the query gets executed, respectively (cf. Figure 2).

This data needs to be carefully analyzed to understand how the processes interact when scheduled in an interleaved fashion, especially for allocating I/O among the processes. We found it to be highly useful to retain *all* of this data for *every* Q@C execution, as detailed in online Appendix C.

7.2. Calculating the CPU Time

The CPU time is easy to calculate because we have per-type system and user time, once we determine which DBMS process was actually running the query. For MySQL, there is only one DBMS process. For DB2, we can uniquely identify the query process (`db2sysc`). Oracle and PostgreSQL have multiple processes of the same name, one of which could be the one actually running the query. For such cases, we select the process that has the highest total user plus system time (across all executions for a single Q@C) calculated for those processes occurring in every QE as the one that is performing the query. Hence, if there is no such process, we will drop that Q@C in Step 2, below.

7.3. Calculating the I/O Time

Viewing the system from the processor point of view, every tick of the processor can be in one of the following modes [BOWDEN, T. et al. 2009]: running a user process (*user*

	Tick 1	*Tick 2*	*Tick 3*	*Tick 4*	*Tick 5*	*Tick 6*	*Tick 7*
Query Process	User Mode	BlockIO Delay Mode (disk IO)	BlockIO Delay Mode	BlockIO Delay Mode (disk I/O)	System Mode	BlockIO Delay Mode (disk IO)	System Mode
Daemon Process	BlockIO Delay Mode (disk I/O)	User Mode	BlockIO Delay Mode (disk I/O)	BlockIO Delay Mode	(waiting)	System Mode	BlockIO Delay Mode (disk I/O)
Processor Status	User Mode	Low Priority Mode	IOWait Mode	IOWait Mode	System Mode	SoftIRQ Mode	System Mode

Fig. 6. Seven ticks in the lives of two interleaved, running processes

mode), running a user process in low priority (*low priority mode*), running on behalf of user processes in the operating system (*system mode*), running on behalf of user processes in soft interrupt processing (*SoftIRQ mode*), waiting for I/O, when all user processes are blocked (*IOWait mode*), or waiting for a user process, as there is no I/O or computation to be done (*idle mode*). We record the number of ticks in each of these modes as overall measures, listed in Section 3.6.

Viewing the system from the *process* point of view, every tick in each existing process can be in one of the following modes: running the process' code (*user mode*), running in the operating system in behalf of the operating system (*system mode*), or blocked on synchronous I/O (*BlockIO Delay mode*). We record the number of ticks in each of these modes as per-process measures, listed in Section 3.5. One mode not captured by the operating system and thus not recorded is when the process is ready to run but is waiting to be scheduled, as another process is currently running.

The relevant modes are illustrated in Figure 6, for two processes, the DBMS query process and another (daemon) process. Let's examine this, tick by tick.

(1) The DBMS query process starts in user mode, while the daemon process waits for synchronous I/O (the disk is performing the I/O in parallel with the processing). The processor is in user mode, executing the query process.
(2) The DBMS query process then performs some I/O, while the daemon process runs in low priority mode (not differentiated from user mode on an per-process basis). The processor is thus in low priority (user) mode.
(3) Now the disk is performing I/O for the daemon process, while both that process and the query process wait on synchronous I/O. As all the processes are in BlockIO Delay mode, the processor status is IOWait mode.
(4) Now the disk is performing I/O for the query process, while both it and the daemon process wait on synchronous I/O. Again, the processor status is IOWait mode.
(5) The query process is running, but has made some system calls, the last for a disk read, so it is in system mode, as is the processor. The daemon process is waiting for the processor (this is not actually recorded anywhere).
(6) The CPU scheduler switches to the daemon process, which promptly is interrupted by a soft interrupt request (SoftIRQ). The per-process mode doesn't differentiate between system mode and SoftIRQ mode. The query process waits on its I/O.
(7) The scheduler once again switches to the query process, which makes a system call, while I/O is performed by the daemon process.

The overall measures for this QE are summarized in Table VII. Here, the idle time is 0 and the total number of ticks is 7.

Table VII. Overall measures for the example QE

Measure	*Value*
User Mode	1
Low Priority Mode	1
System Mode	2
SoftIRQ Mode	1
IOWait Mode	2
Idle Mode	0
Total	7

Table VIII. Per-Process measures for the example QE

Measure	*Value*	
	Query Process	*Daemon Process*
User Mode	1	1
System Mode	2	1
BlockIO Delay Mode	4	4
Total	7	6
Actual I/Os	3	3
Actual I/Os (alternative)	4	2

One might suppose then that the measured time ($Time_{meas}$) will be equal to the sum of other times in ticks, multiplied by the tick duration (10msec).

$$Time_{sum} = (U_{total} + L_{total} + S_{total} + Q_{total} + Idle_{total} + IO_{total}) \times 10$$

where U, L, S, Q, *Idle*, and *IO* denote time in user, low priority, system, SoftIRQ, idle, and IOWait mode, respectively, all in ticks (which again are 10msec). That is indeed the case for our example, cf. Table VII.

Returning to our running example in Figure 6, Table VIII provides the per-process measures for our query and daemon processes. Note that user mode doesn't differentiate low-priority ticks and that system mode doesn't differentiate SoftIRQ ticks. The total differs because the daemon process was waiting to be scheduled in tick 5. While both processes were in BlockIO Delay mode for 4 ticks, each process did 3 ticks worth of I/O. (We'll return to the last line of this table shortly.)

Our error model is that in the absence of non-query processes (utility and daemon), the measured time should be close to the true time, though it will vary somewhat with the vagaries of disk head position, so the number of user time plus system time and number of IOWait ticks will trade-off in slightly different ways for the Q@C executions. (We note in passing that this error model argues against taking the average mentioned at the start of this paper.) The utility processes when present will increase the total time through *their* user and system ticks, and will also probably increase the IOWait ticks, as there are generally only a few processes running at any one time. (On average, there are two or three additional processes running sometime during a Q@C execution, though there is a long tail to 18 processes.)

For our query example given in Table I, the per-type metrics shown in Table IX support this model. (In this table, the lowest and highest values for each row are in **bold**.) Note how stable the user times (U_{query}) and system times (S_{query}) are for the DBMS process performing the query ($SU = S + U$), with SU_{query} varying by only 3 ticks. Note the large number of idle ticks, which occurred for only a few Q@Cs in our data (8.4% in all). Finally, observe that the variance in $Time_{meas}$ is contributed by the daemon processes that we cannot turn off, specifically SU_{daemon}, $SoftIRQ$, and $MajFlt_{daemon}$ and by the interaction of those processes and the DBMS query process in *IOWait* ticks.

Table IX. Breaking out the per-type metrics

	1	2	3	4	5	6	7	8	9	10	*Avg*	*Std Dev*
$Time_{meas}$ (msec)	9321	9210	9964	**13442**	9310	9470	**9206**	9394	9280	9398	9320	1298
U_{query} (ticks)	148	**147**	**152**	—	150	148	—	149	149	151	149	1.7
S_{query} (ticks)	**15**	14	12	—	**11**	13	—	13	13	11	13	1.4
SU_{query} (ticks)	163	**161**	**164**	—	161	161	—	162	162	162	162	1.1
$SU_{utility}$ (ticks)	19	21	**153**	—	36	27	—	27	**18**	23	25	45.8
SU_{exp} (ticks)	4	3	4	—	4	**5**	—	**3**	4	3	4	0.7
SU_{daemon} (ticks)	**24**	24	26	—	24	24	—	26	**27**	24	24	1.2
$BlkIODelay_{query}$ (ticks)	57	58	**69**	—	**56**	62	—	58	59	61	58	4.1
$BlkIODelay_{utility}$ (ticks)	**1**	1	**6**	—	3	2	—	2	2	1	2	1.7
$BlkIODelay_{exp}$ (ticks)	0	0	0	—	0	0	—	0	0	0	0	0
$BlkIODelay_{daemon}$ (ticks)	15	12	**18**	—	**7**	14	—	12	12	13	12	3.1
$MajFlt_{query}$ (ticks)	0	0	0	—	0	0	—	0	0	0	0	0
$MajFlt_{utility}$ (ticks)	0	0	0	—	0	0	—	0	0	0	0	0
$MajFlt_{exp}$ (ticks)	0	0	0	—	0	0	—	0	0	0	0	0
$MajFlt_{daemon}$ (ticks)	0	0	0	—	0	0	—	0	0	0	0	0
$IOWait_{meas}$ (ticks)	40	43	43	—	**37**	45	—	46	44	**47**	43	3.3
$SoftIRQ_{meas}$ (ticks)	6	4	**7**	—	**3**	6	—	5	5	5	5	1.2
$Idle_{total}$ (ticks)	**672**	659	**602**	—	658	681	—	662	663	668	660	23.9

What is of great interest to us, and quite surprising, is that in all of this data, there is *no concrete measure* of the amount of time any process (our focus is the query process) expended doing I/O.

First, note that I/O can be occurring during ticks in any of the processor modes except for idle mode. In the first tick of the running example, the disk is doing I/O but the processor is in user mode, executing the query process. In tick 2, the processor is executing the daemon process in low priority model. In tick 3, the system is doing I/O but the processor is idle, as all processors are blocked (indicated by a processor status of IOWait mode). In tick 4, the situation is reversed. In tick 6, the processor is in SoftIRQ mode and in tick 7 the processor is in system mode.

So the challenge is in apportioning the *IOWait* ticks to the various processes. In our example, we know that the query and daemon processes were in BlockIO Delay mode for 4 ticks each and that the processor was in IOWait mode for 2 ticks. In reality, the query and daemon processes were each doing I/O for 3 ticks, as shown in Table VIII.

Now consider the alternative history in Figure 7, in which three ticks of the daemon process were changed, shown in **bold**. The processor statistics given in Table VII remain as before. The per-process statistics in Table VIII also remain the same. Surprisingly, however, the number of I/Os actually performed by each process, shown in the last row of Table VIII (a measure not actually available) differs from the original history. The reason is that the measures do not identify which process is actually doing the I/O, when more than one process is waiting on I/O. This example demonstrates that

	Tick 1	*Tick 2*	*Tick 3*	*Tick 4*	*Tick 5*	*Tick 6*	*Tick 7*
Query Process	User Mode	BlockIO Delay Mode (disk IO)	BlockIO Delay Mode (disk IO)	BlockIO Delay Mode (disk I/O)	System Mode	BlockIO Delay Mode (disk IO)	System Mode
Daemon Process	**(waiting)**	User Mode	BlockIO Delay Mode	**BlockIO Delay Mode**	**BlockIO Delay (disk I/O)**	System Mode	BlockIO Delay Mode (disk I/O)
Processor Status	User Mode	Low Priority Mode	IOWait Mode	IOWait Mode	System Mode	SoftIRQ Mode	System Mode

Fig. 7. An alternative history of seven ticks in the lives of two interleaved, running processes

we cannot always derive the amount of time any process is doing I/O whenever there are any overall IOWait ticks.

If there are only two processes executing, if one of those processes (in our case, the query process) is running the entire time, and if there are no overall IOWait ticks, the per-process BlockIO Delay ticks should be correct. If there exist IOWait ticks, then we don't know which process to charge them to. For our two IOWait ticks in our example histories, in the first case (Figure 6) one was performed by the query process and one by the daemon process and in the second case (Figure 6) both were performed by the query process.

If there are three or more processes executing, then IOWait mode occurs only when *all* of the currently running processes are waiting on I/O. As an illustration, if we added a compute-bound process to our history, we would get no IOWait ticks: ticks 4 and 5 would both be User mode ticks for that third process.

Online Appendix D examines the possibility of dropping all QEs with non-zero IOWait ticks, concluding that there is no easily-identifiable criteria for rejecting QEs with non-zero IOWait ticks.

So we instead utilize a logical argument: the IOWait time violations indicate when the overall IOWait ticks are greater than the BlockIO Delay ticks for the query process. This check assumes that the query process runs for the entire measurement period, which is the case (see Figure 4). In that case, the query process must be delayed for block I/O for *every* overall IOWait tick, by definition: during an IOWait tick, every active process is waiting on I/O; see Section 7.3. (Note that not every query BlockIO Delay tick results in an IOWait tick.) The advantage of this option is that it drops many fewer QEs; the challenge is that we need to now estimate the query I/O for those QEs with IOWait ticks, which we do in Step 4 of the protocol.

8. TIMING PROTOCOL

Recall that we desire an operationalization of the query execution time for a Q@C, which is a single time in milliseconds computed from the data from ten QEs.

We now have the components for a more accurate and precise timing protocol that can correctly utilize the measures that Linux provides, building on the intricacies of time management and of computing the I/O time used in evaluating a query.

Our general protocol (1) performs sanity checks to ensure the validity of the data, (2) perhaps drops some query executions via clearly motivated predicates, (3) perhaps drops some entire Q@Cs, again via clearly motivated predicates, (4) for those Q@Cs that remain, computes for each a single measured time by a carefully-justified formula

over the underlying measures of the remaining query executions, and (5) performs post-analysis sanity checks.

8.1. Step 0: Set Up the System and Run the Queries

To run an experiment, perhaps with thousands of queries on experiment subject DBMSes, a user needs to do the following steps.

(1) Prepare three machines: one (Linux) for an experiment subject DBMS, another for a dedicated DBMS server, and the third for starting and monitoring an experiment.
(2) Start the dedicated DBMS server, to collect query execution data (query plans, cardinalities, measured times, and process snapshots at query time).
(3) Perform the following setup activities, on each machine that will be running queries on a subject DBMS:
    (a) Enable only one CPU core.
    (b) Switch off turbo mode [INTEL CORP. 2015] on the CPU.
    (c) Ensure that the kernel is up-to-date, specifically, later than 2.6.32 (or the release of Red Hat Enterprise Linux server is after 6.4 (Santiago)).
    (d) Deactivate unnecessary Linux daemons (see list in online Appendix A).
    (e) Ensure that the Network Time Protocol (NTP) daemon is running.
(4) Write experiment specifications in XML:
    (a) one (approximately 30 lines) providing the experiment description, scenario name (e.g. OnePass in Fig. 2), table configuration, and query generation,
    (b) another (a few lines associated with populating the tables),
    (c) and another (perhaps a few to hundreds of lines) with predefined queries, only if automatic query generation is not desired.

    Writing these specifications won't take more than a few hours, as several examples are available.
(5) If a measurement scenario other than the ones already provided is desired, write code (about 200∼300 LOC) in Java to implement the scenario.
(6) Build the project via `ant`. One jar file will be then created.
(7) Copy the jar over a remote client machine, run it as an *Observer* to control the experiment, and install the data collection schema into the dedicated server (by a mouse click).
(8) Copy the jar over the machine running the experiment subject DBMS and run the jar as *Executor* (cf. Section 7.1) to perform requested experiments.
(9) Load the written experiment specification (by mouse click) on Observer.
(10) Run the experiment (by mouse click) on Observer. Executor will immediately begin the experiment, running each Q@C (see Figure 2) of each query ten times and collecting for each QE 91 separate measures, most of which are per-process, and so collecting approximately 170 values per QE, and sending those measures to the lab DBMS (running on a different machine).
(11) Monitor the started run (by mouse click) on Observer to check if it goes well. (If something untoward happens to the run, the Executor will catch it, and then Observer will automatically move the run under the paused run category. Later, a user can resume the paused run (by mouse click) on Observer.)
(12) Once the run is finished, create a study node (by mouse click) on Observer and add the completed run to the study node.
(13) Run the TTPv2 protocol over the added run(s) in the study node (by mouse click) on Observer.
(14) Check the protocol analysis results (by scrolling down) on Observer.

It may take some time for users to understand the entire system (comprising about 45K SLOC), but once the user is familiar with what parts are related to experiment running, it will be enough for her to take from tens of minutes to a couple of hours—depending on how many queries are timed—to complete these steps.

We now examine each step of this protocol in detail and provide an example of applying the protocol to the data used in the confirmatory analysis (see Section 6.2), consisting of a total of 555,850 query executions (55,585 Q@Cs) over the four DBMSes.

8.2. Step 1: Perform Sanity Checks

Before computing the query execution time for each Q@C in the completed runs constituting the experiment, we assess the validity of our data by performing nine *sanity checks*. These sanity checks serve only to indicate possible systematic errors across the entire experiment that need to be examined carefully before proceeding through the rest of the steps. (Sanity checks are always recommended within an experimental methodology to ensure data quality before analyzing the data.)

In prior testing, we encountered violations of several of these sanity checks, which suggested refinements to our protocol, thereby ensuring that there were no such violations for the runs examined in this paper.

We have three different classes of such sanity checks, comprising 25 sanity checks in total. These are given in online Appendix E.

Table X lists the first class: the experiment-wide cases for which not a single violation should occur in a run: eight such sanity checks. The second class of sanity checks concerns query executions; see Table XI. Each of these thirteen sanity checks could encounter a few isolated violations. However, we expect the violation percentage to be low. The final class involves four checks over each Q@C; see Table XII. Now that we see that the original data from our experiment passes all of these sanity checks, we can proceed onto the next step.

8.3. Step 2: Drop Query Executions

In this step, we drop query executions that exhibit specific problems, in order to increase accuracy and precision. Specifically, we drop those query executions identified as problematic in Table XI.

Table XIII shows how many query executions remained after this step. In TTPv1, this step dropped a full 30% of the QEs, whereas TTPv2 only dropped 7.2% of the query executions in all, with 99% of the Q@Cs remaining, with each Q@C retaining an average of 9.35 query executions. In other words, TTPv2 retained four times more query executions and almost all Q@Cs, compared with TTPv1, and removed a DBMS-dependent bias.

8.4. Step 3: Drop Selected Q@Cs

In this step, we look at the query executions for each Q@C, and determine if these query executions *in concert* exhibit specific problems. If so, we drop the entire Q@C, to increase accuracy and precision.

All time metrics except for measured time via `getTime()` are in ticks. For very quick queries, which take only a few ticks, a single tick or two of additional IOWait ticks for a daemon process can throw off the total by a large percentage factor. Also, as noted in Section 7.2, very short execution times result in the wrong query process being chosen. Hence, we (i) drop all Q@Cs identified in the first two rows of Table XII, (ii) drop any Q@C for which the identified query process does not appear in every query execution for that Q@C, (iii) drop Q@Cs with $Time_{meas}$ (average measured time across remaining executions) of 2 or less ticks, and (iv) drop those Q@Cs with less than

Table X. Experiment-wide sanity checks

1	*Number of Missing Queries*	0
2	*Number of Unique Plan Violations*	0
3	*Number of Underlying Measure Violations*	0
4	*Number of Derived Measure Violations*	0
5	*Number of Steal Time Violations*	0
6	*Number of Guest Time Violations*	0
7	*Number of Other Query Process Violations*	0
8	*Number of Other Utility Process Violations*	0

Table XI. Query execution sanity checks

1	*Percentage of Ephemeral Process Violations*	0.01%
2	*Percentage of DBMS Time Violations*	0.06%
3	*Percentage of Zero Query Time Violations*	0.26%
4	*Percentage of Query Time Violations*	0.70%
5	*Percentage of Query User Tick Violations*	1.59%
6	*Percentage of Overall Computation Time Violations*	2.65%
7	*Percentage of Computation Time Violations*	0.14%
8	*Percentage of BlockIO Delay Time Violations*	0.01%
9	*Percentage of IOWait Time Violations*	1.84%
10	*Percentage of Context Switch Violations*	0%
11	*Percentage of Ambiguous Query Process Violations*	0.10%
12	*Percentage of No Query Process Violations*	0.05%
13	*Percentage of Timed-Out QE Violations*	0.02%

Table XII. Q@C sanity checks

1	*Percentage of Excessive Var. in Query Comp. Time*	0.14%
2	*Percentage of Possible Query Result Cache Violations*	0.17%
3	*Percentage of Strict Monotonicity Violations*	0.59%
4	*Percentage of Relaxed Monotonicity Violations*	0.32%

Table XIII. The number of Query Executions (QEs) and Q@Cs remaining after each sub-step and for the entire protocol

At Start of the Protocol	555,850 QEs 55,585 Q@Cs
After Step 2	515,892 QEs (7.18% dropped)
At Start of Step 3	55,181 Q@Cs (0.73% dropped)
After Step 3-(i)	55,023 Q@Cs (0.29% dropped)
After Step 3-(ii)	55,023 Q@Cs (0% dropped)
After Step 3-(iii)	54,458 Q@Cs (1.03% dropped)
After Step 3-(iv)	53,385 Q@Cs (1.97% dropped)
At End of the Protocol	53,385 Q@Cs (4.0% dropped in all)

six valid query executions. For our running query, none of these predicates dictated dropping this Q@C.

Table XIII shows how many Q@Cs are dropped after each sub-step in Step 3. As indicated by the last row for Step 2, we initially had 55,181 Q@Cs, and no Q@Cs were dropped at Step 3-(i), thanks to previously dropping query executions violating no query process sanity check. We dropped 158 Q@Cs at Step 3-(ii), 565 Q@Cs at Step 3-(iii), and 1,073 Q@Cs at Step 3-(iv). Throughout Step 3, a total 3.25% of Q@Cs were discarded; each remaining Q@C had an average of 9.5 query executions. (This increased slightly primarily because we dropped Q@Cs with less than six QEs.)

We expect that cardinality of the result of each each successive step to monotonically decrease. As shown in Table XIII, the cardinalities behave as expected.

Anecdotally, when we studied some previous runs and computed these cardinalities, we observed a dramatic drop in the number of Q@Cs computed in Step 3-(i) from the query executions retained by Step 2-(iii). Recall that each Q@C initial contains ten query executions. After Step 2 eliminated some of these executions, we expected perhaps eight executions per Q@C. Previously, however, we observed that a much greater percentage of query executions were dropped. In examining our data more closely, we determined that in some runs the query process was missing. This was because the name was different (due to the use of a symbolic link in Linux). In other words, no query process appeared to be present in those query executions, when the query process did in fact *exist*. To catch this, we added the sanity check of obtaining the percentage of query executions having no query processes, listed in Table XI. Furthermore, from those runs, we found that for some runs some utility processes from another DBMS were present, which should have not happened. Recall that a physical machine is allowed to run an experiment on only one DBMS at a time, turning off other DBMSes if any. We then added another sanity check to compute the percentage of query executions having other DBMS query or utility processes, as shown in Table X ((7) and (8)).

8.5. Step 4: Calculate Query Time

In TTPv1, we included the following components in computing query time: (a) DBMS user time, (b) DBMS system time, and (c) the portion of IOWait time due to DBMS user time. (Recall that in that protocol we didn't have the benefit of having per-process BlockIO Delay ticks.) For the last component, we used a less elaborate causal model than that given Figure 3, in which there were four factors that cause IOWait ticks: query user time and query, utility, and daemon major faults. (All do so via the number of IO requests, which is a latent, and hence unmeasured, factor.) Of these three influences, only query user time and query major faults involve the query process. We performed a regression, examining how much these three factors each contributed to IOWait ticks. The TTPv1 protocol then used the regression coefficient associated with the query process to determine the contribution of the IOWait time to the total query time.

This approach is problematic, for three reasons. First, it doesn't take into account the environment of the particular QEs of the Q@C for which the query IO time is being estimated. A second concern is that as query computation time can be estimated in a much more accurate fashion than query I/O time, for the reasons given in Section 7.3, this computation provides an unrealistically stable, and thus suspect, I/O time estimate. The third reason this approach is problematic is that it uses a globally-computed regression coefficient, across a potentially large number of QEs, to estimate the contribution of query I/O to query time.

For TTPv2, we have additional measures to utilize in this calculation. That said, our analysis in Section 7.3 shows that this is challenging. It turns out that with care we can get a very close estimate of I/O time directly from these other measures, without resorting to a global regression coefficient.

We assume (as verified in Section 7.3) that there is relatively little utility and daemon I/O, and thus for each IOWait tick, there are only two processes waiting on I/O: the query process and some other process. In that case, there is a 50-50 chance that the I/O is not being performed for the query process, and thus the query I/O time must adjust for this over-counting.

$$IO_{\boldsymbol{calc}} = BlockIO_{\boldsymbol{query}} - 0.5 \times IOWait \quad (1)$$

$$Time_{\boldsymbol{calc}} = (S_{\boldsymbol{query}} + U_{\boldsymbol{query}} + \boldsymbol{IO}_{\boldsymbol{calc}}) \times 10\boldsymbol{msec} \quad (2)$$

As context, our validated structural causal model (Figure 3) articulates the following components of the run time of any process: user ticks and system ticks (both per-process and overall), BlockIO Delay ticks (per-process), IOWait ticks (overall only), and SoftIRQ ticks (also overall only), along with causal links between them (oversimplifying: user ticks impact number of read and write system calls, which influence BlockIO Delay, IOWait, and SoftIRQ ticks). Upon further investigation of the Linux documentation and examination of our data, we realized that SoftIRQ's are intermediate to IOWait ticks and are few in number with a small variance, and so can be safely omitted. This resulted in the above equations, which utilize the relevant per-process and overall measures in a disciplined and justified manner.

In Section 1, we questioned whether taking the average of the time from multiple QEs is appropriate. We now have much more information in which to decide between minimum, average, median, and maximum.

Taking the maximum assumes that the extraneous factors that contribute to the variance of the time all or predominately subtract from the measure of execution time. As we discussed in Section 5.2, when a scheduler timer interrupt occurs, the tick (10msec) is entirely attributed to the currently executing process. Thus, if a lot of IO is going on, there is a good chance that that process is the DBMS, and also a chance that the DBMS had only received a portion of that tick, implying that in that situation the recorded DBMS execution time was *higher* than the actual time. This particular extraneous factor then could cause the time to be *over-counted*. Also, as we saw in our causal model of Figure 3 (along with Table VI), there are many now-known factors, in particular, from other processes operating during the DBMS process, that can increase the calculated I/O time of the DBMS process. Even more important is that these factors have a different proportional effect on the I/O time. We conclude that taking the maximum may not be indicated.

Taking the minimum assumes that the extraneous factors all or predominately add to the measure of execution time. However, we have no a priori evidence that there are no factors that can *decrease* the calculated I/O time and even the measured CPU time. For that reason, we were drawn to the mean and median measures instead of the minimum.

Taking the average value assumes that factors extraneous to the actual running time are distributed equally on both the positive and negative side. To get a handle on this distribution as well as whether there was some correlation between successive QEs, we ran only Q@C for 800 QEs in one of the DBMSes, then plotted adjacent pairs (hence, 400 points in all) as the x and y axes, in Figure 8. This plot doesn't show a discernable correlation between successive QEs, which is good, but also shows some non-uniformity in the distribution (a uniform distribution would look more like a circular cloud).

Given the above considerations, the TTPv2 protocol takes the *median* value of the QEs surviving Step 3 within the Q@Cs surviving Step 3 of the protocol.

This completes our original task: calculating query time more accurately and more precisely, while dropping many fewer Q@Cs.

The emphasis throughout this paper has been measuring query time, which experiences significant computation and I/O as compared with utility and daemon processes in the pristine configuration described in Section 8.1. However, the protocol is applicable much more broadly, as suggested in Figure 1. Let's examine how the other items in that taxonomy can be accommodated in our protocol.

. **Wall-clock time** The protocol captures wall-clock time as an overall measure (see Section 3.4), while increasing the precision and accuracy via a raft of sanity checks on the QEs and Q@Cs.

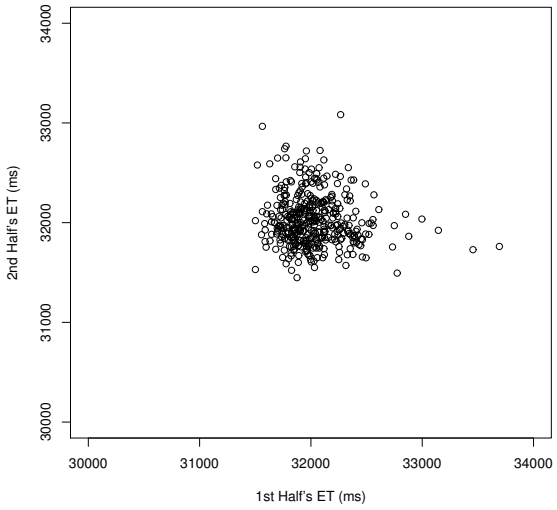


Fig. 8. QE pairs of a single Q@C for one DBMS

. **Daemon processes** Such processes can be measured in the same manner as the query process. If the assumption that daemon processes are rare does not hold up, then the portion of I/O for each daemon process as well as the total portion can be estimated. The fraction $\frac{1}{2}$ in equation 1 is replaced by an expression estimating the number of processes on average competing for an I/O during an *average* IOWait tick.

$$IO_{calc} = BlockIO_{daemon} - \frac{BlockIO_{daemon} \times IOWait}{BlockIO_{query} + BlockIO_{utility} + BlockIO_{daemon}} \quad (3)$$

. **User processes** Such processes can be measured in the same way as daemon processes.
. **DBMS utility processes** Such processes can be measured in the same way as daemon processes.
. **I/O request counts** Section 3.5 includes the following per-process measures: *number of characters read*, *number of characters written*, *number of bytes read*, *number of bytes written*, *number of read system calls*, and *number of write system calls*. However, as discussed in Section 5, the first four of these measures are suspect.

In summary, this protocol, by focusing on both compute-bound processes and I/O time measurement (Section 7.3), accommodates a wide variety of measurement needs, including many of the measurements mentioned in Figure 1.

8.6. Step 5: Post Sanity Checks

As post sanity checks, we re-examine excessively varying query time on the refined data, and check monotonicity violations based on the calculated query times. We add three post sanity checks:

(4) how different are measured and calculated time for the *retained* Q@Cs (relative to measured),

Table XIV. Post (Q@C) sanity checks

1	*Percentage of Excessive Variation in Calculated Query Time*	0.25%
2	*Percentage of Strict Monotonicity Violations*	0.68%
3	*Percentage of Relaxed Monotonicity Violations*	0.35%
4	*Relative Difference For Measured and Calculated Time for Retained*	2.3%
5	*Relative Difference For Measured and Calculated Time for Dropped*	7.5%
6	*Number of Non-Varying Model Measures*	0

(5) how different are measured and calculated time for the *dropped* Q@Cs (again, relative to measured), and

(6) number of non-varying model measures, that is, those measures used in the model in Figure 3 that did not vary over the QEs across the entire experiment, or equivalently, have the exact same value for every QE.

Table XIV shows our post sanity check results. After our protocol was applied, about 0.25% of the retained Q@Cs revealed excessively-varying calculated query time. The rates of monotonicity violations changed slightly.

We also checked the relative difference between measured and calculated query time for retained Q@Cs and for dropped Q@Cs, with the differences small. (As we'll see shortly, for our one running example query, the difference between measured and calculated was large, but this particular query is unusual. For most queries the differences were very small.) These small percentages are an indication of quality of measurement, suggesting that dropping the small percentage of Q@Cs did not skew the times.

Finally, while some measures (daemon cguest ticks, daemon guest ticks, overall guest ticks, overall IRQ ticks, overall low user priority ticks, overall steal ticks, query process cguest ticks, query process guest ticks, utility process cguest ticks, utility process guest ticks, and utility process major faults) were non-varying, none of these measures were utilized in our causal model. (Most of these are guest ticks, which are those running a virtual CPU for a guess operating system.) We only apply this sanity check to the measures that we use in the model (hence the term "model measure"); the sanity check identifies which of those model measures are non-varying, that is, have the same value for *every* QE. In the confirmatory data, we had no such violations.

9. REPORTING THE RESULTS

Other sciences have standardized ways of reporting that a measurement methodology has been used. For example, in chemistry experiments, the following information should be included: quantities of all reagents in grams or mL, for extractions: solvent, volume, and number of extractions; for distillation: type and temperature; for IR spectroscopic data: method of sample prep and one or two diagnostic frequencies.

We suggest that the following nine pieces of information should be presented in papers that utilize this methodology for measuring query time (or process time in general).

(1) *Protocol:* The name of the protocol and its citation (there are currently two protocols in this class: Tucson Timing Protocol Version 1 (TTPv1) [CURRIM, S. et al. 2013] and Tucson Timing Protocol Version 2 (TTPv2) [this paper]).
(2) *Hardware:* Relevant information on the hardware (e.g., chipset, motherboard, main memory, disk drive, network connection).
(3) *Operating System:* The specific version of the operating system used, as well as which specific kernel version.
(4) *Number of QEs per Q@C:* How many QEs were run at each Q@C.
(5) *Resulting Computed Measures:* The measure(s) that were ultimately provided by the protocol, and their unit(s).

(6) *Any Deviations:* List any and all deviations from the stated protocol, including O/S daemons retained or dropped.
(7) *Experiment-Wide Sanity Checks:* Number of runs that violated any of the sanity checks listed in Table X.
(8) *QE and Q@C Sanity Checks:* The percentages of query executions and Q@Cs dropped in Steps 2 and 3 of the protocol, to two significant digits.
(9) *Post-Sanity Checks:* The observed values of the first three post-sanity checks, also to two significant digits. (We don't require stating the values for the last three post (Q@C) sanity checks only because we don't have a well-motivated recommended cutoff.)

In this manner, the study in this paper can be described in two crisp paragraphs.

> The measurements were collected using Tucson Timing Protocol Version 2 (TTPv2) [CURRIM, S. et al. 2016] on a suite of five machines, each an Intel Core i7-870 Lynnfield 2.93GHz quad-core processor on an LGA 1156 95W motherboard with 4GB of DDR3 1333 dual-channel memory and Western Digital Caviar Black 1TB 7200rpm SATA hard drive, running Red Hat Enterprise Linux Server release 6.4 (Santiago) with the 2.6.32-358.18.1 kernel. The protocol executed the query ten times at each cardinality and provided calculated query evaluation time, including computation and I/O time, in msec. The protocol was utilized exactly as specified.
>
> No runs violated the experiment-wide sanity checks. Approximately 7.2% of the query executions and 4.0% of the queries-at-cardinality (Q@Cs) were dropped due to query execution and Q@C sanity checks. As a result, excessive variation in calculated query time was observed in 0.25% of the Q@Cs. A total of 0.35% of Q@C adjacent pairs violated relaxed monotonicity and 0.68% of the Q@Cs violated strict monotonicity, which is acceptable.

Using an established protocol enables a paper to concisely yet thoroughly specify the methodology and procedures utilized in the experiments reported in the paper, while allowing the reader to ascertain the quality of the measurements.

10. EVALUATION

Our approach requires significant additional time and effort to more accurately and precisely measure query execution time. Why could we not use approximations like average query time, dropping outliers, or using the minimum query time? The main problem with these approximations is that they cannot systematically separate the actual query execution time from other process times.

We started with the times in Table I. The lowest and highest numbers (in **bold**) exhibited a range of 4,236 msec, which is over 30% of the highest time. As we have seen, this variability arises from necessary daemons and DBMS utility processes and their I/O and interactions with the DBMS.

The execution times ($Time_{calc}$) computed by our TTPv2 protocol are shown in Table XV, with the median, 1993 msec, being the computed query time for this Q@C (in this case, there are an even number of remaining query executions, and so the median is the average of the two central times, with three times on each side). The last column indicates that this time is within one standard deviation of ± 4.6 ticks, or $\pm 2.3\%$. Across all runs, the computed time is within one average standard deviation of ± 4.2 ticks, or $\pm 2.1\%$. In addition, we can report (see Figure 1) that the overall wall-clock time ranged from 9.2 to 13.4 seconds, with an average of 9.8 sec $\pm$1.3 sec. The DBMS CPU time is 1620 $\pm$11 msec (SU_{query}) and the DBMS I/O time is 370 $\pm$39 msec (IO_{calc}).

Table XV. Final computed times

	1	2	3	4	5	6	7	8	9	10	*Median*	*Std Dev*
$Time_{meas}$ (msec)	9321	9210	9964	**13442**	9310	9470	**9206**	9394	9280	9398	9800	1298
SU_{query} (tick)	163	**161**	**164**	—	161	161	—	162	162	162	162	1.1
$MajFlt_{Qdbms}$ (msec) (msec)	0	0	0	0	—	0	0	0	0	—	0	0
$BlkIODelay_{query}$ (ticks)	57	58	**69**	—	**56**	62	—	58	59	61	59	4.1
$IOWait_{meas}$ (ticks)	40	43	43	—	**37**	45	—	46	44	**47**	44	3.3
IO_{calc} (ticks)	37	36.5	**47.5**	—	37.5	39.5	—	**35**	37	37.5	37	3.9
$Time_{calc}$ (msec)	2000	1975	**2115**	—	1985	2005	—	**1970**	1990	1995	1993	46.2

Table XVI. Experiment Details

	TTPv1	TTPv2
Cumulative Experiment Hours	3,000	6,184
Number of Query Instances	3,080	4,800
Number of Q@Cs	35,363	55,585
Number of QEs	353,630	555,850

Compare the $Time_{meas}$ with $Time_{calc}$ in Table XV. The $Time_{calc}$ have a much smaller range than $Time_{meas}$. This difference is caused by variations in SU_{daemon} and $MajFlt_{daemon}$ across runs. Therefore, if we used average query time, we would obtain 9800 msec, with a standard deviation of 1298 msec, which is significantly higher, less accurate and less precise than our average $Time_{calc}$ of 1993 msec, with a standard deviation of 46.2 msec. As mentioned in Section 7.3, the large disparity between measured and computed time in this case was primarily due to a large number of idle ticks, which occurred for only a few Q@Cs in our data; the point here is that our protocol can handle such unusual events.

It is important to find the root cause of an outlier before deciding how to handle it; therefore, a systematic approach like ours is needed to separate times into query related time and other time, before excluding or adjusting for outliers. The minimum $Time_{meas}$ is also influenced by other processes, so it is not as accurate nor precise. In our example, the maximum $Time_{calc}$ is 2115 msec (in the third run), while the minimum $Time_{meas}$ is 1970 msec (the eighth run). The third calculated time of 2115 msec includes extra irrelevant time associated with $SU_{utility}$ (1530 msec) and $BlkIODelay_{daemon}$ (180 msec), see Table IX.

This example, and the discussion throughout this paper, emphasize that it is important to separate the overall time into its component parts and exclude query-irrelevant components in calculating actual query execution time, instead of using coarse approximations like average or minimum, as exemplified at the very beginning of the paper.

The result is version 2 of the Tucson Timing Protocol (TTPv2), a significant refinement of Version 1 (TTPv1) [CURRIM, S. et al. 2013]. The following lists the major changes made to produce this refined version.

— Precision was discussed throughout in more detail.
— Section 3.1 - A discussion on cold cache versus warm cache measurements was added.
— Section 3.2 - We now ensure data and query plan repeatability.
— Section 3.3 - A thorough discussion of which daemons were possible to turn off and which were not was included
— Section 3.4 - The choice of various Linux, Java, and processor clocks was discussed.

Table XVII. Protocol Outcome Comparison

	TTPv1	TTPv2
Number of Sanity Checks	18	34
Percentage of Retained Q@Cs	76%	96%
Avg. Number of QEs per Retained Q@C	8.5	9.4
Relative error for Measured Time	4.5%	2.3%
Relative error for Calculated Time	2.1%	2.2%

— Section 3.5 - Several important per-process measures were added: *number of context switches* (voluntary and involuntary), for a sanity check in Step 1 of the protocol and *BlockIO Delay time*, for dropping query executions in Step 2.
— Section 5 - The structural causal model utilized in the previous protocol was elaborated to include BlockIO Delay time and voluntary and involuntary context switches, increasing the number of measures from 13 to 25 and the number of causal relationships from 27 to 73. (These additional measures were included to better understand the context of a query execution.) A greater number of hours (cf. Table XVI) were required to test this new causal model thoroughly. The increase in number of hours is due to four factors: (i) the increased number of QEs run, (ii) retaining longer runs, (iii) for some reason many of the MySQL queries were slower in the new protocol, and (iv) an upgrade in the operating system added some new daemon processes.
— Section 6 - The exploratory and confirmatory evaluation of the elaborated structural causal model required testing many more predicted correlations, yet was still strongly supported.
— Section 7.1 - This is perhaps the most significant change: the *ProcListen* thread was added to gather `taskstats` for those processes that stopped during the timing. This allows us to accommodate such processes; TTPv1 had no way to recording measures for processes that started and stopped during a query; the detection of such an *ephemeral* process causes that QE to be dropped, approximately 35% of the QEs, and 25% of the Q@Cs. Even more concerning is that such QEs were encountered mainly on long-running queries, thereby introducing a bias into the definition of query time. Also, what to record and why is an important component; TTPv2 records a great deal more information on each QE than TTPv1.
— Section 7.3 - The analysis of query I/O time is quite a bit more elaborate compared with TTPv1, by virtue of the per-process BlockIO Delay ticks, which give us a finer (though still not entirely accurate) view of how many I/Os each process performs.
— Section 8 - The TTPv2 protocol extends TTPv1 in several important areas, summarized in Table XVII.

Step 0. Setting up the system is a new step.
Step 1. The overall sanity checks (now termed “experiment-wide”) grew from 3 to 8, the QE sanity checks from 6 to 13, and the Q@C checks from 3 to 4.
Step 2. QEs are now dropped using a fairly sophisticated analysis of IOWait ticks. However, the *ProcListen* thread ensures that we have no phantom processes, a major advance over TTPv1. (We also check now for ephemeral processes.)
Step 3. The no query process sanity check was added.
Step 4. The calculation of query time is much simpler, as it doesn’t require a regression analysis. We can also time a much broader range of processes, including user processes and daemon processes.
Step 5. Three sanity checks were added.
Reporting. The TTPv2 protocol includes a standard means of reporting the use of the protocol in a consistent and comprehensive fashion.

11. SUMMARY

We desire to know how long a query took to execute. This simple question is surprisingly challenging to answer, as a DBMS interacts with other processes and with the operating system in quite involved ways.

This paper has considered these interactions in detail. We first discussed the spectrum of granularities with regard to both *what* is being measured and *how* it is measured (cf. Figure 1). We emphasize that once the approach espoused here is used to obtain a quite precise and accurate query time, those other aspects can also be measured to understand the full context of a query.

We then considered many subtle aspects of measuring time: mitigating the many cache effects, ensuring data and within and across run plan repeatability, showing that between-run repeatability is not possible for some DBMSes, and understanding the intricacies of time measurement in Linux. We articulated a structural causal model relating these measures. A thorough correlational analysis provided strong support for this model. We examined in detail the most complex part, that of computing the I/O time used by the DBMS.

Interestingly, Linux provides no direct measure of this important aspect, and even indirect approaches are quite complex. We showed that the most relevant, albeit indirect, measure of I/O activity is a combination of BlockIO Delay ticks and IOWait ticks (the former per-process and the latter, overall). We also showed that there is no easily-identifiable criterion for rejecting query executions with non-zero IOWait ticks.

These investigations provided the specific mechanisms for our proposed timing protocol that comprises *Isolation*: eliminating as many extraneous factors, including network delays, cache effects, and daemons; *Measurement*: specific metrics collected before and after the query execution, in a carefully prescribed order; and *Analysis*: a sequence that (i) performs initial sanity checks over the entire data, (ii) perhaps drops some query executions, (iii) perhaps drops some entire Q@Cs, (iv) for those Q@Cs that remain, for each computes a single query time using the underlying measures of the remaining query executions, and finally, (v) performs some final sanity checks.

The result is a mature, robust, self-checking protocol that provides a more precise and more accurate timing of the query, reducing variance from 2.2% for TTPv1 to 2.1% for calculated time (which comes out to 29 msec over all the confirmatory data (56K Q@Cs). TTPv2 also discards only 4.0% of original Q@Cs, while TTPv1 discarded 24% of the Q@Cs, in a biased way, in that longer-running queries were much more likely to be discarded, because of the increased chance of ephemeral processes, which are handled better here.

The *Tucson Timing Protocol* Version 2 (TTPv2) is quite general, applicable to most versions of Linux that support `/proc` and `taskstats`, and is also applicable to other operating domains in which measurements of multiple processes each doing computation and I/O is needed.

12. FUTURE WORK

The experimental analysis of the structural causal model in Figure 3, discussed in Sections 6.1 and 6.2, raised some interesting issues that could potentially be used to increase the quality of the model and thus perhaps of the query time measurement. First, capturing I/O effectively turned out to be very difficult, given both the lack of adequate measures and the fact that they do not in concert capture all of the I/O activity. We found that the measured number of bytes read and written are not accurate, due perhaps in part to the availability of other system calls that perform I/O, such as `mmap()`, which unfortunately are not included in available measures. Delving into the source code of the specific DBMSes (which differed considerably for some of

the relationships in Table VI), might shed some light. Similarly, we did not explore further individual variations for the other relationships, because those variations did not affect the overall strength of correlation. It would be useful to look at the individual variations for the DBMS(es) for which one had source code access.

It would also be useful to look into (i) why per-process user mode ticks do not exactly line up with the overall user ticks reported by Linux, as well as (ii) for overall system ticks, and (iii) how a process could exhibit no user mode ticks yet produce a significant number of system mode ticks (see Section 8.2, specifically QE sanity check 5 in Table XI).

Delving even further into the Linux system, it would be helpful to add a measure of per-process waiting mode ticks (cf. Section 7.3) or perhaps two per-process BlockIO measures, one counting the BlockIO Delay ticks incurred by *that* process performing disk I/O and another one counting those ticks incurred by *another* process performing disk I/O. That would make it *much* easier to ascribe I/O to the appropriate process. And while we are recommending Linux source revisions, it would be helpful to record user and system time in the `proc` file system in microseconds, as does the `taskstats` facility (cf. Section 3.7) and to record the number of `mmap()`, etc., function calls.

It would also be useful to understand the variance that remains (for our protocol, $\pm 2.1\%$). Our conjecture is that unmeasurable factors such as thread synchronization contributed to this variance. Also useful would be to explicate the interaction between multiple threads, some potentially doing I/O and some doing computation, and BlockIO Delay ticks and to look into O/S prefetching of data while a process was performing computation in advance of requesting that data.

It would be useful to extend this query time measurement protocol to (a) incorporate network delays for a remote disk (which necessitates clearing the network file server cache for cold cache timings), (b) utilize number of bytes (chars) read and written, (c) accommodate multiple disks, connected by a single or distinct channels, (d) accommodate multiple processor cores, (e) accommodate VMs and DBMSes in VMs while eliminating their impact on the computed time, (f) extend PostgreSQL to clear its cache, (g) ensure repeatability of file fragmentation, (h) understand asynchronous I/O, which is also supported by Linux [LINUX 2015] (but might not be used by DBMSes), (i) extend the protocol across various data caching approaches. It would be desirable to extend the protocol to handle multiple queries in a single transaction (this should be straightforward) as well as a mix of transactions (again, this changes only the identification of the start and stop of the timing), though of course the variance of the timing will probably increase, and (j) extend the protocol to measure other important aspects of DBMS execution, such as transaction throughput (cf. [SUH, Y.-K. 2015]).

The protocol can also be extended to support the Windows operating system, which has substantially different per-process metrics, and thus might require an altered causal model and a different causal model and calculation of query time.

This protocol uses 17 of the 91 underlying overall and per-process measures available in Linux. It would be helpful to utilize additional measures (within both the causal model and the timing protocol), should they further improve the precision and accuracy of the protocol and to generalize the protocol to (a) measure single transactions that incorporate multiple statements and (b) measure a mix of transactions. We believe our model should remain validated in different Linux-based environments. However, we expect that it should work across hardware changes, though the correlations might change. As long as the setup is the same (few daemons and utility processes) and as long as Linux is not changed radically, we would expect the model to hold. Future hardware changes are a different matter. In such cases, model discovery can be manual or automated, and machine-learning-based model discovery and derivation may be useful.

Finally it would be useful to extend the protocol to measure aspects outside of the taxonomy of query time in Figure 1, such as DBMS startup time, query planning time (the latter, say by timing the "prepared_query" operation), and resource consumption, e.g., buffer space utilization, in general. Indeed, the approach espoused here could be taken as a case study on a general methodological approach to more precisely and accurately measuring anything within a DBMS, or even within a computer program generally.

13. ELECTRONIC APPENDIX

The online electronic appendix to this article can be accessed in the ACM Digital Library. That appendix lists stopped daemons, explains how we detect ephemeral processes, explains how the collected data is retained, explains how to handle IOWait ticks (which are problematic in the presence of three or more executing processes, as discussed in Section 7.3), and lists the three classes of sanity checks used in Step 1 in Section 8.2.

14. WEBSITE

More information may be found at `http://www.cs.arizona.edu/projects/TTP` .

15. ACKNOWLEDGMENTS

We thank Benjamin Dicken, Preetha Chatterjee, Pallavi Chilappagari, Jennifer Dempsey, David Gallup, Kevan Holdaway, Matthew Johnson, Andrey Kvochko, Derek Merek, Lopamudra Sarangi, Cheng Yi, and Haziel Zuniga for their contributions to AZDBLAB and Tom Buchanan, Phil Kaslo, Tom Lowry, and John Luiten for constructing and maintaining our experimental instrument, a laboratory of ten machines and associated software. Finally, we thank Nikolaus Augsten and the referees of the previous conference version and of this journal version for their many helpful comments. This research has been supported in part by NSF grants IIS-0415101, IIS-0639106, CNS-0838948, IIS-1016205, and EIA-0080123, and by partial support from a grant from Microsoft Corporation.

REFERENCES

AILAMAKI, A. G., DEWITT, D. J., HILL, M. D., and WOOD, D. A. 1999. *DBMSes on Modern Processors: Where Does Time Go?* Technical Report UW-CS-TR-1394. Computer Sciences, University of Wisconsin.

AKSWEW, M., CETINTEMEL, U., RIONDATO, M., UPFAL, E., and ZDONIK, S. B. 2012. Learning-based Query Performance Modeling and Prediction. In *Proceedings of the IEEE International Conference on Data Engineering*. IEEE, 390–411.

BOWDEN, T., BAUER, B., NERIN, J., FENG, S., and SEIBOLD, S. 2009. Linux Kernel Documentation - The `/proc` File System. https://www.kernel.org/doc/Documentation/filesystems/proc.txt. (2009).

COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., and SEARS, R. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 143–154.

CURRIM, S., SNODGRASS, R. T., SUH, Y.-K., ZHANG, R., JOHNSON, M., and YI, C. 2013. DBMS Metrology: Measuring Query Time. In *Proceedings of the 39th ACM SIGMOD International Conference on Management of Data*. ACM, New York, USA, 421–432.

FAULKNER, R. and GOMES, R. 1991. The Process File System and Process Model in UNIX System V.. In *USENIX Winter*. USENIX, 243–252.

FORMAN, R. F., PECHACEK, J. M., and SCHWANE, W. H. 2001. Apparatus and Method for Measure Transaction Time in a Computer System. IBM Patent. (January 2001).

GANAPATHI, A. , KUNO, H. A., DAYAL, U. , WIENER, J. L., FOX, A. , JORDAN, M. I., and PATTERSON, D. A. 2009. Predicting Multiple Performance Metrics for Queries: Better Decisions Enabled by Machine Learning. In *Proceedings of IEEE International Conference on Data Engineering*. IEEE, 592–603.

GIKOUMAKIS, L. and GALINDO-LEGARIA, C. 2008. Testing SQL Server's Query Optimizer: Challenges, Techniques and Experiences. *IEEE Data Eng. Bull.* 31, 1 (Aug. 2008), 36–43.

HWANG, H.-Y. and YU, Y.-T. 1987. An Analytical Method for Estimating and Interpreting Query Time. In *Proceedings of VLDB International Conference*. Morgan Kaufmann Publishers Inc., 347–358.

INTEL CORP. 2015. http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html. (2015). [Online; accessed 12-August-2015].

KANT, K. 1992. ***Introduction to Computer System Performance Evaluation***. McGraw-Hill.

LINUX. 2015. http://lse.sourceforge.net/io/aio.html. (2015). [Online; accessed 12-August-2015].

ORACLE CORP. 2014. PreparedStatement in Java 7. http://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html. (2014). [Online; accessed 21-March-2014].

ROSEN, R. 2014. ***Linux Kernel Networking***. Springer.

STILLGER, M., LOHMAN, G. M., MARKL, V., and KAQNDIL, M. 2001. LEO - DB2's Learning Optimizer. In *Proceedings of the VLDB International Conference*. Morgan Kaufmann Publishers Inc., 621–632.

SUH, Y.-K. 2015. *Exploring Causal Factors of DBMS Thrashing*. Ph.D. Dissertation. Department of Computer Science, University of Arizona.

SUH, Y.-K., SNODGRASS, R. T., and ZHANG, R. 2014. AZDBLAB: A Laboratory Information Systems for Large-Scale Empirical DBMS Studies. *Proceedings of the VLDB Endowment (PVLDB)* 7, 13 (Aug. 2014), 1641–1644.

THOMAS, S. W., SNODGRASS, R. T., and ZHANG, R. 2014. Benchmark Frameworks and τBench. *Software—Practice and Experience* 44, 9 (April 2014), 1047–1075.

TPC. 2010a. TPC Transaction Processing Performance Council—TPC-H. http://www.tpc.org/tpch/. (2010). [Online; accessed 29-August-2010].

TPC. 2010b. Transaction Processing Performance Council—TPC-C. http://www.tpc.org/tpcc/. (2010). [Online; accessed 29-August-2010].

VALGRIND DEVELOPERS. accessed October 27, 2010. Callgrind: A Call-Graph Generating Cache and Branch Prediction Profiler. http://valgrind.org/docs/manual/cl-manual.html. (accessed October 27, 2010).

WORKING GROUP 2 OF THE JOINT COMMITTEE FOR GUIDES IN METROLOGY (JCGM/WG 2). 2008. International Vocabulary of Metrology—Basic and General Concepts and Associated Terms. (2008).

YAO, B. B., ÖZSU, M. T., and KHANDELWAL, N. 2004. XBench Benchmark and Performance Testing of XML DBMSs. In *Proceedings of the International Conference on Data Engineering*. IEEE, 621–632.

YU, P., CHEN, M.-S., HEISS, H.-U., and LEE, S. 1992. On Workload Characterization of Relational Database Environments. *IEEE Transactions on Software Engineering* 18, 4 (April 1992), 347–355.

ZHANG, N., TATEMURA, J., PATEL, M., and HACGUMUS, H. 2011. Towards Cost-Effective Storage Provisioning for DBMSs. *Proceedings of the VLDB Endowment (PVLDB)* 5, 4 (Sept. 2011), 274–285.

ZHANG, R., SNODGRASS, R. T., and DEBRAY, S. 2012. Micro-Specialization in DBMSes. In *Proceedings of the IEEE International Conference on Data Engineering*. IEEE, 690–701.

Received September 2015; revised May 2016; accepted September 2016

Online Appendix to: DBMS Metrology: Measuring Query Time

SABAH CURRIM, University of Arizona
RICHARD T. SNODGRASS, University of Arizona
YOUNG-KYOON SUH, University of Arizona
RUI ZHANG, University of Arizona

A. STOPPED DAEMONS

The following table lists the daemons that were disabled in Red Hat Linux 6.4. The remaining daemons either belong to the kernel or are managed by the *root* user. Due to their association with the kernel, it is necessary to not to turn them off, as doing so may jeopardize the proper function of the OS.

Table XVIII. List of eliminated system daemons

Daemon Name	Description
`abrtd`	Automated bug reporting tool's daemon. A daemon watching for application crashes.
`acpid`	Advanced Configuration and Power Interface event daemon. A daemon designed to notify user-space programs of ACPI events.
`atd`	`atd` runs jobs queued by `at`
`auditd`	The Linux Audit daemon. Responsible for writing audit records to the disk, as the userspace component to the Linux Auditing System.
`certmonger`	A daemon that monitors certificates for impending expiration, and is capable of optionally refreshing soon-to-be-expired certificates with the help of a certificate authority (CA).
`crond`	The daemon to execute scheduled commands (ISC Cron V4.1)
`cups`	Common Unix Printing System
`haldaemon`	A Hardware Monitoring System. Auto-recognizes various kinds of hardware and mountable media.
`hplip`	HP Linux Imaging and Printing Project
`ip6tables`	IPv6 packet filter administration
`iptables`	Administration tool for IPv4 packet filtering and NAT
`postfix`	Postfix control program. Used to submit mail.
`sendmail`	An electronic mail transport agent
`vmware`	VMware SVGA video driver. An Xorg driver for VMware virtual video cards.
`vmware-USBArbitrator`	VMware USB Arbitration Service daemon. Allows USB devices plugged into the HOST to be usable by the guest.
`xinetd`	The extended Internet services daemon

B. DETECTING EPHEMERAL PROCESSES

Given the timeline of Figure 4, let's now examine how to detect ephemeral processes, those that were somehow not seen by the *ProcListen* threat.

Consider what is stored for processes that start and stop at various times. Measures for processes *P2* and *P3* are recorded into a map, M_1, when the first `getPerProc()` is

DOI: 0000001.0000001

invoked (event (4)). This map contains the value up to that point of each of the per-process measures for that process. Note that M_1 does not mention *P0*, which ended before the first call to `getPerProc()`. Processes *P5* and *P6* might appear in M_1, depending on exactly when they started (as `getPerProc()` takes a while to execute (around 80msec), going through the processes one by one). *P1* and *P4* may also be in M_1, depending on exactly when they stopped. In our analysis below, we assume that all of the processes were in M_1, and processes were also similarly in other sets mentioned below, for simplicity.

Another map, M_2, is built after the query has executed by the second call to `getPerProc()` (event (14)). M_2 records processes *P3* and *P9*, and it might also have *P6*, *P10*, and *P11*, depending on exactly when they stopped, as explained above. *P12* may be in M_2, depending on when it started. M_1 and M_2 will of course contain the DBMS query process (which waits for the query, then executes it, then waits for the next query to arrive) and EXECUTOR. *P13* won't be in M_2, as it started after event (15).

There are some processes that terminate before, during or after the query execution and while the *ProcListen* thread is still active. We call these *stopped processes*. For instance, *P1*, *P4*, *P5* and *P7* all stop before the query execution. *P2* and *P8* stop during the query execution (event (9)). *P6*, *P10* and *P11* stopped after query execution (event (10)). Among these stopped processes, *P1*, *P2*, *P4*, *P5*, *P7* and *P8* are not in M_2, as again they terminated before the second call to `getPerProc()` (event (14)). Likewise, *P8*, *P9*, *P10*, *P11*, and *P12* are not in M_1, as again they started after the first call to `getPerProc()` completed. *P7* appeared after the first call to `getPerProc()` completed and exited before the query execution, and *P8* started after query execution began and ended before the query execution finished. Thus, processes *P7* and *P8* are neither in M_1 nor M_2.

Continuing processes persist after the *ProcListen* thread (event (17). In this case, these are *P3*, *P9*, *P12*, *P13* and of course the DBMS process itself, as well as EXECUTOR.

Figure 9 provides a Venn diagram of all the processes illustrated in Figure 4. Our protocol doesn't see processes *P0* and *P13*, as these occur outside of the measurement regime. The rest all reside in one or more of the following sets: 1) live processes recorded (into a map called M_1) before query execution (M_1 processes), 2) live processes recorded (into another map called M_2) after query execution (M_2 processes), 3) stopped processes, and 4) continuing processes. Note that the area of N/A indicates that there is no case that a process is in M_1 and continuing processes but not in M_2 processes. There are some *ephemeral* processes, to be discussed shortly, such as processes *Px*, *Py*, and *Pz* in the diagram. These three processes were somehow not seen by the *ProcListen* thread; we want to be able to detect when we have one or more ephemeral processes, so that we can discard that QE.

What is important is that for the stopped processes not in M_2, we cannot ascertain from that map the cumulative per-process measures for these processes. That said, we can detect the presence of the stopped processes through the *ProcListen* thread, started at event (2) and stopped at event (17). Each time a process terminates, this thread receives notifications from the `taskstats` NetLink interface that is discussed in Section 3.5. The thread then records the cumulative per-process measures of the terminating process. Therefore, processes *P1*, *P2*, *P4*, *P5*, *P6*, *P7*, *P8*, *P10* and *P11* are recorded by the *ProcListen* thread. As the thread provides us with the cumulative per-process measures for the stopped processes, we do not have to be concerned about losing their process information. *ProcListen* sends their measures to PROCMONITOR, which subsequently reports them to EXECUTOR.

Finally, terminating processes not captured by the *ProcListen* thread, as well as processes that the second call to `getPerProc()` just missed, are termed ephemeral

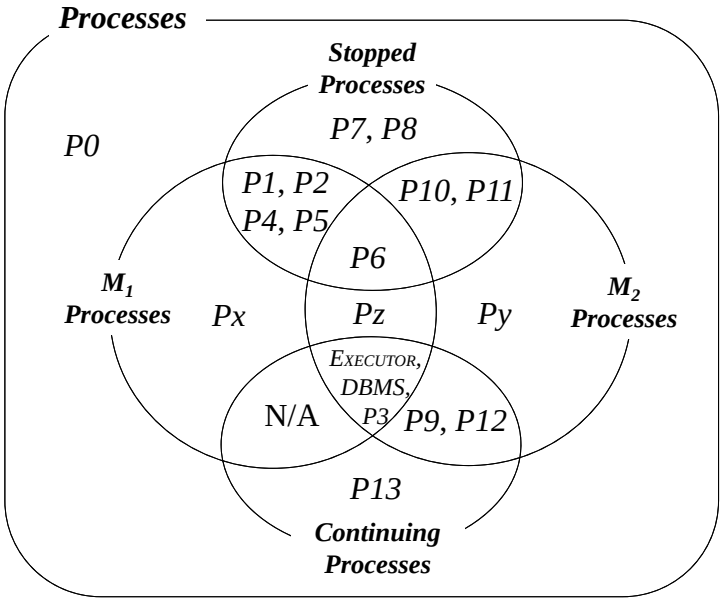


Fig. 9. Venn diagram of processes in Figure 4

processes, as there are no measures available for those processes. We show three such processes, *Px*, *Py* and *Pz*, whose termination during the query execution was not detected by the *ProcListen* thread.

The number of ephemeral processes is derived as follows. Four sets of processes are involved and illustrated in the Venn diagram in Figure 4: (i) M_1 processes computed by `getPerProc()` in event (5), (ii) M_2 processes computed by `getPerProc()` in event (15), (iii) *Stopped Procs* computed by the *ProcListen* thread in event (17), and (iv) *Continuing Procs* computed as follows.

$$\textit{Continuing Procs} = M_2 - \textit{Stopped Procs}$$

There are three kinds of ephemeral processes, identifies as *Px*, *Py*, and *Pz*. We can catch ephemeral process *Px* via the following.

$$(M_1 - \textit{Stopped Procs}) - M_2$$

We catch ephemeral process *Pz* in the following way.

$$((M_1 \cap M_2) - \textit{Stopped Procs}) - \textit{Continuing Procs}$$

Finally, we catch ephemeral process *Py* the following way.

$$((M_2 - M_1) - \textit{Stopped Procs}) - \textit{Continuing Procs}$$

C. RETAINING THE COLLECTED DATA

We store three text files: a *before image* of the overall and per-process measures, an *after image* of the overall and per-process measures, and a `taskstats` *image* just of the processes that stopped during the measured time. It is important to note that we collect all available measures, a total of 20 overall measures, 53 per-process measures from `/proc/`*pid*`/stat`, from `/proc/`*pid*`/status`, and from `/proc/`*pid*`/io`, and 42 per-process measures from `taskstats` (of which 24 overlap with `/proc/`*pid*`/stat` or `/status` or `/io` and 18 are unique to `taskstats`). Of those 91 separate measures that are collected, our

protocol uses just some of those, the eight overall and nine per-process measures listed in Sections 3.5 and 3.6. The protocol uses these seventeen measures (either directly, or indirectly within sanity checks) to obtain the ultimate objective: the calculated time (in msec) for that Q@C. (The rationale is that disk space is cheap, but running all the queries again if the protocol is refined is very expensive.) The text files total about 430KB on average, after compression only 33KB. Thus the protocol analyzes for each Q@C 430KB $\times$ 10 = 4.3GB to finally compute a single four-byte (!) derived measure.

Later, during analysis, we uncompress the raw data for each Q@C execution and parse these files to extract the value for each measure. The per-process and overall measures are all *cumulative*, in that they increase during the query execution. Hence, we can subtract the value of each in the before image from that in the after image. Some of the remaining measures are relevant just in the before image (e.g., `PID`). For some (e.g., `priority`) we just store both values in a string. We locate all child processes that are forked from other processes, and fold up the measures to the parent. We can then identify the query process via a heuristic: having the relevant name and having the most computation time. The end result is exactly 91 property-value pairs per QE, which are stored in a relational database. For a 100-query run, on average we store about 1.1M property-values per DBMS subject.

D. HANDLING IOWAIT TICKS

In Section 7.3, we saw how problematic IOWait ticks are in the presence of three or more executing processes, as an IOWait model occurs only when *all* of the currently running processes are waiting on I/O.

All is not lost, however. In Section 3.3 we showed how to significantly reduce the number of user processes (to none) and system daemons. Figure 10 provides the cumulative distribution of BlockIO Delay ticks of (a) utility processes (recall that these are DBMS processes performing tasks other than query evaluation), (b) daemon processes, and (c) the sum of BlockIO Delay ticks for utility and daemon processes, over the QEs with greater than zero ticks. For the exploratory study, involving 87,223 QEs (absent 177 QEs that never completed), most (82,408 QEs) had no BlockIO Delay ticks for any utility processes, 54,901 QEs had no such ticks for any daemon processes, and 54,265 QEs had no such ticks for any utility or daemon processes.

In this graph we plot for x-axis value n the number of QEs having a number of ticks $\leq n$. We show values at x-axis values 1, 2, ..., 10, 20, ..., 100, 200, ..., 1000, 2000, 3000, and 4000. (For that reason, there is a small jump at 20 and an even smaller jump at 200.) We see a gentle rise for utility processes and an elbow around 4 ticks for daemon and total (non-query) processes.

We thus see that the non-query processes contribute a small amount of I/O in most cases, though there do exist QEs with a substantial amount of non-query I/O. Sometimes this I/O occurs when the query process is doing computation, so no IOWait tick occurs. In that case, we can get a more accurate measure of query I/O through the query BlockIO Delay ticks.

One possibility we considered is to drop all QEs with non-zero IOWait ticks. The calculation then of query I/O time is query BlockIO Delay ticks $\times$ 10msec. The advantage of this option is that it results in very clean QEs for which we know precisely how much I/O time was expended by the query process. The primary concern is that this injects a bias against long-running queries, which have a greater chance of experiencing IOWait ticks. It is likely that *all* QEs in a Q@C for a long-running query will be dropped, thus dropping the entire Q@C.

How many QEs would be dropped by this approach? Figure 11 provides the cumulative distribution of non-zero overall IOWait ticks. (Again, the jump at an x-value of 20

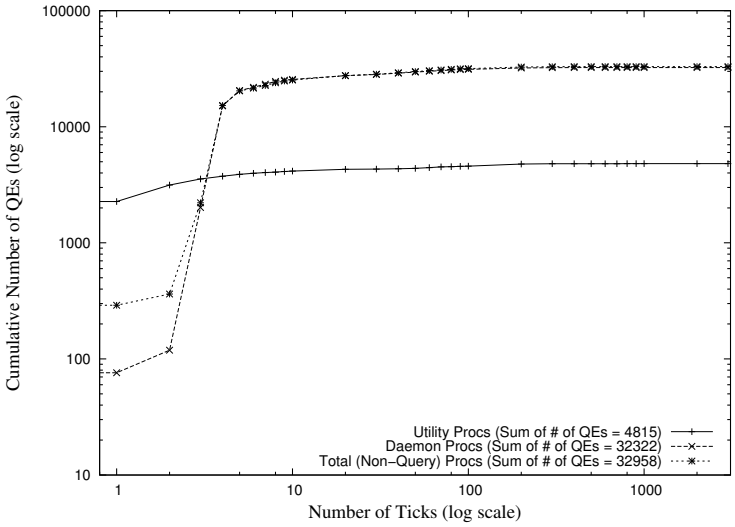


Fig. 10. Cumulative distribution of non-query BlockIO Delay

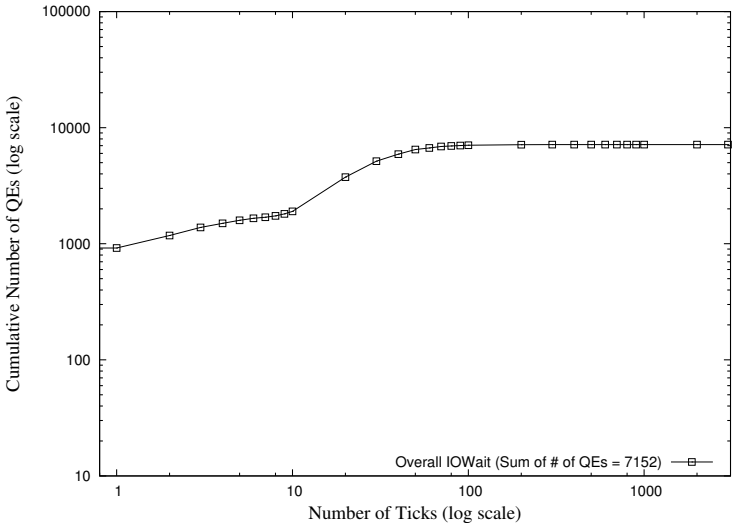


Fig. 11. Cumulative distribution of overall IOWait ticks

is due to the spacing of the x-values of the points.) Out of 87,223 QEs, 80,071 QEs had no IOWait ticks and 7,152 QEs (8.2%) had at least one such tick.

We considered several possibilities for reducing the number of violations. One would be to drop those with a low number of IOWait ticks. But as we see from that figure, there is no identifiable elbow. Dropping those with > 10 IOWait ticks retains 81,972 QEs, or 93.97%; dropping those with say > 30 IOWait ticks retains 85,211 QEs, or 97.69%. This does not seem to be worth the less certainty that results.

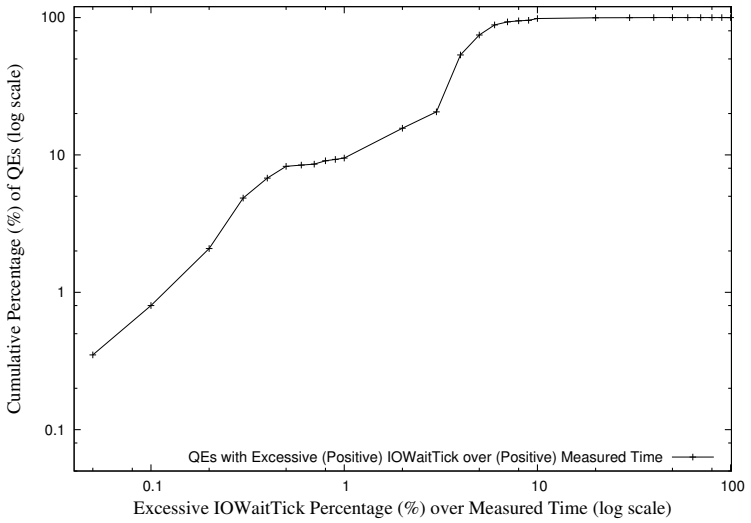


Fig. 12. Cumulative QEs with *Percent-IOWait* $\leq$ x=value

Another way to look at this is to observe that each IOWait tick adds a little uncertainty. We want a criterion that does not penalize long-running queries. Consider the following value.

$$\textit{Percent-IOWait} = \frac{IOWait \times 10msec}{Time_{meas}} \times 100\% \tag{4}$$

A *Percent-IOWait* of 100 would indicate that all the measured time was in terms of IOWait ticks; a *Percent-IOWait* of 1 would indicate that 1% of the measured time was in terms of IOWait ticks. We would like to retain those QEs with a low *Percent-IOWait*.

Figure 12 shows, for those QEs with non-zero IOWait ticks, the cumulative percentage of those QEs with a stated *Percent-IOWait*. This value climbs fairly smoothly, covering most of the offending QEs at around 10% of measured time. Again, there is no convenient elbow. Even at a 1% of measured time, we retain only 10% of the offending QEs, which is less than 1% of the total collection of QEs.

As we see, there is no easily-identifiable criteria for rejecting QEs with non-zero IOWait ticks.

E. SANITY CHECKS

There are three classes of sanity checks used in Step 1 in Section 8.2.

The first class are the experiment-wide cases for which not a single violation should occur in a run: eight such sanity checks.

(1) The number of missing queries indicates how many queries were not executed, for whatever reason.

(2) The number of unique plan violations counts the Q@Cs that have more than one query plan. All query executions of Q@C must have a single, identical query plan. This should be ensured by our automated experiment runner.

(3) The number of underlying measure violations counts the QEs that have at least one process that is missing (i.e., has a null value) one or more of the expected per-process or overall measures discussed in Sections 3.5–3.6 for that process.

(4) The number of derived measure violations counts the QEs that are missing (i.e., has a null value) for a derived overall measure (i.e., the number of ephemeral processes).

(5) The steal time violation counts the QEs that have non-zero steal time in overall measures.

(6) The guest time violation counts the QEs that have non-zero guest time or guest nice time in overall measures, or more than one process with non-zero guest time or cguest time.

(7) We ensure that no query process(es) from another DBMS are running. We thus check how many query executions include another query processes.

(8) We also check for utility processes from other DBMSes.

The second class of sanity checks concerns query executions; see Table XI. Each of these thirteen sanity checks could encounter a few isolated violations. However, we expect the violation percentage to be low.

(1) The ephemeral process violation identifies the percentage of QEs with such a process, as described in Section 7.1 and exemplified in Figure 4. There are only 12 violations observed in our data, about 0.01%.

(2) The DBMS time violation check identifies the percentage of QEs with the time taken by DBMS processes (query and utility) in total is less than the time taken by daemon processes, since we expect query execution time to dominate. Fortunately, only about 0.06% DBMS time violations were observed across all query executions in our data. (Note that in this and the next two sanity checks, process time is defined as the sum of user, system, and BlockIO Delay time.)

(3) The *zero* query time violation check indicates how many QEs have a query time of 0 ticks. (Fast-running queries are hard to measure.) Our data showed only 0.26% violations as well.

(4) The query time violation check identifies the QEs in which the query time is greater than measured time. This could happen if one was off by just one tick. In our data, we uncovered 0.70%, which is a very low rate.

(5) The query user tick violations indicates the percentage of QEs in which the number of query process user mode ticks is greater than number of overall user time (normal and low priority) ticks plus 1 tick. The one tick is a fudge factor, due to how the scheduler charges processes for partial ticks. Note that this sanity check reveals that per-process user tick measurements are somewhat problematic; this should be investigated further. We considered checking user ticks for other processes and found that statistics of in particular stopped processes—the query process never stops—are themselves inaccurate, in terms of per-process user time and system time. We also considered doing the same test for overall system mode ticks, as well as for overall system mode plus overall user mode ticks. However, we found that some processes that exhibited no user mode ticks still exhibited significant process-level system ticks, which doesn't make sense. This too indicates a need for further investigation. We observed 1.59% query user tick violations across our data, which was very low rate.

(6) The overall computation time violations identify the number of QEs where the sum of overall system and overall user time is greater than the measured time (we don't need to add ticks here because overall measurement is very fast, unlike per-process measurement). Only 2.65% overall computation time violations were detected in our data.

(7) The computation time violations identify those QEs for which the sum of system and user time for all processes is greater than the measured time plus 10 ticks (again, because these per-process measures are collected in `getPerProc()`, which takes around 100 msec). We saw only 0.14% computation time violations across our data.

(8) The BlockIO Delay time violations identify QEs for which the maximum (over all processes of that QE) of that process' BlockIO Delay time is greater than the measured time. Intuitively, the measured time must include all the I/Os done individually by each process. The rate of BlockIO Delay time violations was only about 0.01%.

(9) The IOWait time violations indicate when the QE exhibits a worrisome number of overall IOWait ticks. As we saw in Section 7.3, it is difficult to accurately time queries having IOWait ticks. So instead, we check for cases where the number of overall IOWait ticks is greater than the BlockIO Delay ticks for the query process. About 1.84% overall IOWait violation rates were observed.

(10) The context switch violations indicate those QEs in which the query process had an abnormally high number of context switches (more than three standard deviations away from the average, that is, it is definitely an outlier). Since we expect the query process to be dominant, such executions indicate a problem. We observed no context switch violations.

(11) The ambiguous query process violations indicate those QEs for which a utility process had a greater computation time than the identified query process for that Q@C, indicating that our heuristic for identifying the query process (having the relevant name and having the most computation time; see Section C) was flawed in some way. We saw only about 0.10% ambiguous query process violations. This low rate indicates that our heuristics for query process identification were effective.

(12) The no query process violation check indicates the number of QEs that do not have any query process. We got very low percentage (0.05%), with most of these query executions at the minimum cardinality (10K) or with a very short measured time, less than 10ms.

(13) The timed-out query violations indicates the number of QEs that timed out (our timeout limit is 20 minutes). Only about 0.02% of queries timed out, which was very low. These timed-out queries were from only one DBMS.

The final class involves four checks over each Q@C; see Table XII.

(1) The first detects excessive query computation time variations, those in which the standard deviation of the query computation time (computed as in Equation 2, to be discussed in Section 8.5) is greater than 20% of the average query process computation time. (We don't include timed-out query executions in this percentage.) We observed only few violations (0.14%).

(2) To test for possible query result cache effects (see Section 3.1), we look for Q@Cs that might imply the use of such a cache: the query computation time for the first QE is significantly longer (by ten times the average standard deviation of query computation time over all QEs within the experiment) than all of the remaining QEs. Since we observed a very low rate (0.17%) of possible query result cache violations, we think that there was little chance of a query result cache being used.

(3) Monotonicity Violation identifies two Q@Cs for the same query plan having different cardinalities, for the query time of the Q@C's at a lower cardinality is greater than that of the Q@C at a higher cardinality. (For this and the next sanity check, the query time is calculated using Equation 3 on page 37.) The query over the larger cardinality should take more time, so such instances indicate a problem; we term this *strict* monotonicity violation. We expect a small number of violations due to the small variance of query time at each query execution. As expected, only about 0.59% of strict monotonicity violations were detected.

(4) Due to the unavoidable variation of query times, we also consider *relaxed* monotonicity violation in which half a standard deviation of query time is subtracted from the lower cardinality and half a standard deviation is added to the upper cardinality, to account for some of this variation. Again, we observed only about 0.32% of relaxed monotonicity violations.