DBMS Metrology: Measuring Query Time

Sabah Currim
University of Arizona
scurrim@email.arizona.edu

Richard T. Snodgrass
University of Arizona
rts@cs.arizona.edu

Young-Kyoon Suh
University of Arizona
yksuh@cs.arizona.edu

Rui Zhang
Teradata
Rui.Zhang@teradata.com

Matthew Wong Johnson
Univ. of California, San Diego
mwj@email.arizona.edu

Cheng Yi
University of Arizona
yic@cs.arizona.edu

ABSTRACT

It is surprisingly hard to obtain accurate and precise measurements of the time spent executing a query. We review relevant 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. Using this model, we developed a timing protocol, which (1) performs sanity checks to ensure validity of the data, (2) drops some query executions via clearly motivated predicates, (3) drops some entire queries at a cardinality, again via clearly motivated predicates, (4) for those that remain, for each computes 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. The resulting query time measurement procedure, termed the *Tucson Protocol*, applies to proprietary and open-source DBMSes.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*

Keywords

accuracy; repeatability; Tucson Protocol

1. INTRODUCTION

A common approach for at least the last twenty-five years to measure database management system 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." [8]

Consider the measured times in Table 1, for which considerable care (to be described in detail later) was taken to get repeatable results, and for which many sources of time variation (also to be discussed) were eliminated.

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

Even after taking these many proactive steps to improve the measurements, they still vary quite a lot, from 6290 msec to 8510 msec (with the lowest and highest numbers in **bold**), a range of 2220 msec, which is over 25% of the highest time. As we will see, this variability arises from necessary daemons and DBMS utility processes and their I/O and 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" [9]. (Accuracy is termed *external validity* and precision, *repeatability*, in some contexts.)

In the following, we address the central question just raised: how to achieve precise and accurate measurements of query execution time? In considering the approach that is the norm, averaging ten runs, one asks, why average? In fact, why not minimum? Should all ten times be used? If some are dropped, how many should remain? Can additional information from the operating system help refine the determination of query processing time?

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 precise, with a measurement resolution for many queries averaging ±1.2%. Due to the many external vagaries that have been eliminated, this protocol is also much more accurate.

While many (most?) database papers contain measurements, this is the first detailed time measurement protocol to be presented, providing a concrete and carefully specified sequence of steps to arrive at a defensible measurement of query evaluation time. Along the way, we present the first detailed causal model of the ways in which per-processor and overall measures are related and examine support for that model by a correlational analysis over hundreds of thousands of program runs.

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. 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 [12] benchmark is run for a user-specified length of time (minutes to hours),

	1	2	3	4	5	6	7	8	9	10	*Avg*	*Std Dev*
$Time_{meas}$ (msec)	**6530**	6571	**8764**	7961	8427	7829	8246	8506	8239	6991	7806	818

Table 1: Measured time of ten executions of a query on PostgreSQL

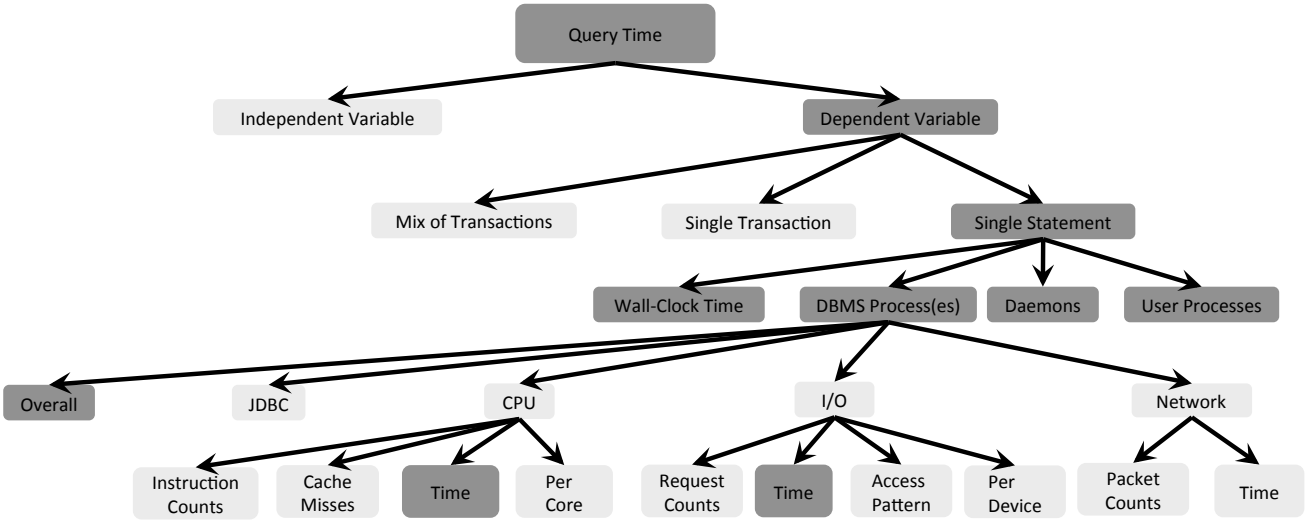


Figure 1: DBMS Time Taxonomy

long with the transaction mix (ratio of read and update transactions) and the number of completed transactions is measured, yielding a measured *transactions per minute*. We focus in this paper on measured query time. This could be of a mix of transactions, each with one or more queries and updates, of a single transaction, or of a single SQL statement (query or update). We focus here on measuring the total time of a single query. Some of these measurements can also be made of cloud databases, in which the queries are run over many distributed computers [4] or at a smaller scale, on a local distributed system [5].

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 [13] benchmark runs a host of queries over a prescribed database and measures total time for each, as do the XBench [16] and τBench [15] 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, I/O, network, or CPU execution. 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 activity (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 [1] using profiling tools like Valgrind and Callgrind [14], which provide instruction and cache hit metrics. Counts are generally collected either through the operating system or instrumented DBMS or by running a disk or cache simulation on the instrumented DBMS [19].

The present paper will consider how to more accurately and precisely measure the time required to execute a single SQL statement, examining (a) overall 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 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 measure DBMS query process I/O and compute 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 compute 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.

Scenario (e.g., Exhaustive[1], OnePass[2])
Experiment (query set (Q1-Q6), data spec, scenario parameter(s))
DBMS (DB2, Oracle, PostgreSQL)
Machine (`sodb6`–`sodb12`)
Experiment Run (on a particular date)
Query Instance (query number within query set)
Query at Cardinality (10K–2M)
Q@C Execution (1–10)

Figure 2: Hierarchy of Query-at-Cardinality (Q@C) executions

Indeed, our approach allows one to accurately measure all of the shaded variables 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., [7, 11]), to predict query performance (e.g. [2, 6, 8]), to characterize workloads (e.g., [17]), or to do provisioning and capacity planning (e.g., [18]).

3. TERMINOLOGY

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.

As an example of this hierarchy, we previously presented in Table 1 measurements of 10 *Q@C Executions*. For the data in this table, we utilized the OnePass Scenario, an Experiment specifying a second set (Q2) of 20 queries (out of six such sets of 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 the PostgreSQL DBMS, on the `sodb10` machine, from an experiment run started January 20, 2012 at 3:31pm, for query number 17, thus identifying a particular query instance running on a variable table with a cardinality of 1,770,000, and examining all ten Q@C executions. This paper concerns a total of 36 experiment runs, each taking a few days to a week on a single machine, involving a total of 353,630 query executions (35,363 Q@Cs) over the DB2, MySQL, Oracle, and PostgreSQL DBMSes running on the Linux operating system, totalling almost 3000 hours (4 months) of cumulative time.

4. MEASURING QUERY TIME

We now turn our attention to the central problem: measuring in a defensible manner the execution of a Q@C: a particular query at a specified cardinality on a particular machine for a specified DBMS within the context of a particular scenario and experiment.

[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.

We execute in quick order, for a single query at a single cardinality, a certain number of Q@C executions. In our protocol, we execute each Q@C 10 times, which is sufficient for timing purposes. (As we'll see, Step 3-(iii) of the protocol drops Q@Cs with fewer than 6 query executions after many checks on these executions. This step retained 85% of the Q@Cs, indicating that 10 is about the right number of query executions to start with.) From various low-level measurements gathered during these multiple executions, we then compute a query execution time for that Q@C.

4.1 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" [10].) The major difference among these functions is their measurement resolution. 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 1 given on the second page of this paper is a pure cold cache measurement of query time using `currentTimeMillis` for 10 executions on PostgreSQL of the following query.

```
SELECT t0.id3, t1.id2
FROM ft_HT3 t3, ft_HT1 t1, ft_HT3 t2, ft_HT1 t0
WHERE t3.id3=t1.id2 AND t1.id2=t2.id1
      AND t2.id1=t0.id4
```

This query is on two tables. `ft_HT1` (the variable table) contains 177,000 tuples, each with four integers; `ft_HT2` (one of the constant tables) contains 2 million tuples, also each with four integers.

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 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 (d) and (e) address precision.

Most machines now have multiple cores, from 2 to 8 cores. As we will see, 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 configure 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, by over 25% 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.

4.2 Per-Process Measures

By examining the measures in `/proc`, a pseudo-file system maintained by Linux which provides an interface to kernel data structures, the measurement protocol can obtain through file reads valuable *per-process* statistics for each active process identifier. Most are counts in units of *ticks*, which for our processors are exactly 10msec. The measures of interest to us are (a) *minflt* [3], 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) *majflt*, 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). 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 `getProcs` method within the protocol that itself takes about 16 milliseconds. Note that thread synchronization costs are not recorded by the O/S, and thus cannot be measured in this way.

4.3 Overall Measures

There are a variety of other overall measures available. 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 (always 0, because lower-priority processes have already been eliminated), (c) *stime*, the number of ticks in which the operating system was servicing a system call or interrupt, (d) *idle time*, the number of ticks when the processor has nothing to do (almost always 0), (e) *IOwait time*, the number of ticks in which the system had no processes to run because all were waiting for I/O, (f) *irq* (interrupt requests) handled by the system, (g) *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 in a timely manner), (h) *steal time*, the number of ticks spent in other operating systems when running in a virtualized environment (always 0 in our protocol) and (i) *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 `getProcStat()` method within the protocol), with the first value subtracted from the second value to obtain the number performed during query execution. Each `getProcStat` takes about 0.13 milliseconds, because there is only one set of these measures.

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 the newest version, Redhat Enterprise Linux 6.3, 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.

5. A PROPOSED CAUSAL MODEL

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 ticks, and steal time measured in ticks and number of irq, softirq, and processes measured in counts. Per-process measures are measured at individual process level, extracted from the output of `getProcs()`. 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.

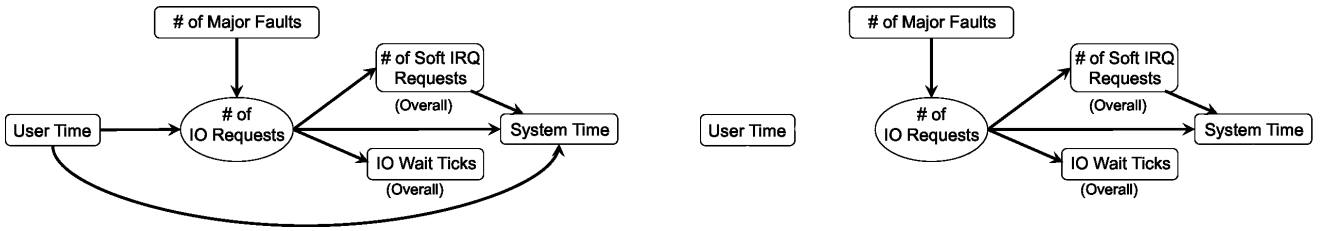
As noted above, steal time and user model with low priority time are always 0. We already have counts for user and system time for those DBMS process(es). The challenge is to estimate the portion of overall IOwait ticks directly or indirectly caused by activities of the DBMS processes.

5.1 Structural Causal Model

Figure 3 provides an initial causal model of these measures. The 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 3a, from the non-query processes (utility and daemon 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 3b.

Within this composite model, nodes are variables to be measured and directed arcs hypothesize causal interactions. The variable denoted "# of IO Requests", a count of such requests, cannot be measured, and thus is considered a *latent variable*, depicted with an oval. (`/proc/`*pid*`/IO` provides bytes read, not I/O requests in counts of blocks read; `/proc/`*pid*`/mountstats` provides a count of `READ` and `WRITE` requests, but only to the network file server, not to the local disk [3]. Several of the DBMSes include page reads and writes in the statistics they maintain; we have not yet pursued that source of information.)

Some variables, specifically the number of Soft IRQ Requests and the number of IO Wait Ticks, are overall measures, and so represent the same values in both sub-models. For the other three measures, user time, number of major faults, and system time, we have both per-type and overall measures. For Figure 3a, "User Time" is a construct with four different operationalizations: user time of the DBMS query process(es), total user time of the utility DBMS process(es) (if any), user time of the daemon process(es) (if any), and total user time, aggregated over all processes. The other two variables, system time and major faults, have analogous operationalizations.



(a) Model for the DBMS Query Processes

(b) Model for Non-Query (Utility and Daemon) Processes

Figure 3: The Causal Model

The intuition behind this model is that the DBMS in its normal processing (measured by user time) reads data from the database, each read which induces an IO request to read in the block. That IO request incurs its own system time and interrupt requests, some of which are soft IRQs. If all processes are blocked on I/O, that request could add to IOwait ticks. All of these causal interactions are positive. For example, if user time increases for a different query, it is predicted that the number of IO requests might increase, which could itself increase the number of Soft IRQ requests, number of IO Wait ticks, and amount of system time. The DBMS presumably does other things that require system time. We don't expect to see a significant number of major faults caused by the DBMS query process, because after just a few query executions the entire query evaluator will have been faulted in, though if there were any major page faults, each would incur an IO Request to read the page in.

For all but the query process (that is, both utility and daemon processes), 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. In either case, we don't know whether these non-query processes cause system time or major faults and therefore, in Figure 3b, there is no causal link predicted between user time and anything.

5.2 Predicted Correlations

This causal model relates 14 measures (four for each of user time, number of major faults, and system time plus single overall measures of number of soft IRQ requests and IOwait ticks), or a total of 182 interactions. For some pairs of measures, such as between daemon user time and daemon system time, there is no causal interaction predicted. For some other pairs of measures, such as daemon major faults and daemon system time, there is no direct causal link but there is an indirect causal link, in this case, along two paths, on involving number of soft IRQ requests. The overall model thus makes 45 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. We identify expected an correlation of 0.7 or greater as a "high" level of correlation, from 0.3 to 0.7 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 2. In each box is an interaction between two measures and a predicted level. We designate such pairs by group and letter within group. An example is correlation (*Ia*), the box at the top left, which concerns the interaction between query user time and query system time.

In general, we predict correlations between two different directly-linked measures for the same type of process (e.g., (*Ia*)) to be high. We predict overall correlations based on the strength of the associated per-type correlations. Similarly, indirectly-linked interactions are a sequence of directly-linked interactions: the correlation for the entire path generally won't be greater than that of any interaction along the way. An example is query major faults and query system time. For interactions involving a per-type and overall of different directly linked measures, as well as part-whole relationships (between a per-type and the overall operationalization of the same construct), we differentiate between query and non-query measures, because query processing dominates the computation. If the measure is only overall (that is, number of soft IRQ requests and number of IOwait ticks), or if there is a common ancestor, we do a case-by-case analysis.

The first group (Group I) involves correlations between two different measures, but for the same type of process (which can be either query, utility, or daemon). All such pairs are expected to be highly correlated, except for those involving major faults, because major faults are rare, and so their contribution to IO requests is minor. This group includes 11 predicted correlations.

The second group (II) 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, so we expect medium correlations. For utility and daemon processes, we expect only low correlation, for the same reason: the query process dominates the overall measure. Finally, concerning between IO requests and soft IRQ, we expect medium correlation, because prior investigation implied that IO requests generate soft IRQs.

The third group also involves correlations between two different measures, but for overall, expected to be correlated if all three per-type correlations in the first group are predicted. Also included in this group are such correlations involving measures that are only overall.

The fourth group involves correlation between a per-type measure and overall of an *indirectly-linked* different measure. Such correlations are expected to be of medium strength because they involve a sequence of other correlations.

The fifth group involves indirect correlations between two overall measures, because they don't differentiate the type of process. There are two cases. The first case is correlations involving user time. The interactions between user time and any other variable for non-IO bound processes are not in the model (Figure 3b).

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*
I	query UTime/ query STime *high*	query I/O req/ query STime *high*	query UTime/ query I/O req *high*	query MajFlts/ query I/O req *low*	query MajFlts/ query STime *low*	utility MajFlts/ utility I/O req *low*	utility MajFlts/ utility STime *low*	utility I/O req/ utility STime *high*	daemon I/O req/ daemon I/O req *low*	daemon MajFlts/ daemon STime *low*	daemon I/O req/ daemon STime *high*
II	query I/O req/ overall IO wait *med*	query I/O req/ overall Soft IRQ *med*	query Soft IRQ/ query STime *med*	utility I/O req/ overall I/O wait *low*	utility STime/ overall Soft IRQ *low*	daemon I/O req/ overall I/O wait *low*	daemon STime/ overall Soft IRQ *low*	utility I/O req/ overall Soft IRQ *med*	daemon I/O req/ overall Soft IRQ *med*		
III	overall MajFlts/ overall I/O req *low*	overall I/O req/ overall STime *high*	overal IO req/ overall I/O wait *low*	overall I/O req/ overall Soft IRQ *med*	overall Soft IRQ/ overall STime *med*						
IV	query UTime/ overall Soft IRQ *med*	query UTime/ overall IO wait *med*	query MajFlts/ overall Soft IRQ *med*	query MajFlts/ overall IO wait *med*	utility MajFlts/ overall Soft IRQ *low*	utility MajFlts/ overall IO wait *low*	daemon MajFlts/ overall Soft IRQ *low*	daemon MajFlts/ overall IO wait *low*			
V	overall UTime/ overall Soft IRQ *med*	overall UTime/ overall IO wait *med*	overall MajFlts/ overall Soft IRQ *med*	overall MajFlts/ overall IO wait *low*	overall MajFlts/ overall STime *low*						
VI	query UTime/ overall UTime *high*	query STime/ overall STime *high*	query IO req/ overall I/O req *high*	query MajFlts/ overall MajFlts *low*	utility MajFlts/ overall MajFlts *med*	daemon MajFlts/ overall MajFlts *med*					
VII	overall IO wait/ query STime *med*										

Table 2: Hypothesized Interactions and Hypothesized Levels

The sixth group involves part-whole relationships, that is, between the query component and the overall operationalization of the same construct. If the query process dominates the overall, the correlation will be high. If query process does not dominate, the correlation will be low. Query dominates for user, system and IO requests. Query does not dominate for major faults, which means that either or both utility and other contribute.

The seventh and final group involves implied relationships, that is, between variables that share a common ancestor, though there is no path between them. That said, we do so only for query measures, as less is known about non-query processes. Since there is no direct path, we expect the strength of the correlation to be smaller of the incoming arcs from the common ancestor.

6. TESTING THE CAUSAL MODEL

As mentioned, the seven group contribute 45 expected correlations. However, many involve the latent variable, number of IO requests, which by definition cannot be measured. We have a remaining 27 correlations not involving the latent variable. Our experiments provide a large amount of data that can be used to test this model, which we do now. Then, in Section 7, we will use this model to apportion I/O wait time to the DBMS processes and other processes.

We tested the model in two phases. In the first, *exploratory model analysis*, we ran a correlation analysis on a small portion of the query runs. We then examined our assumptions against the results of this analysis, and revised the model. 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.) The result is that a 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 nonexistent for the query process (which made sense in retrospect, as the query code will have been swapped in at the beginning of the experiment and repeated executed).

The result to this point is a set of 27 correlations, each with an expected level: low, medium, or high. There are other interactions that are not predicted by our model. We then transitioned to the *confirmatory model analysis* stage of our testing, in which we did a correlational analysis of all 36 query runs, followed by a comparison of the actual level of correlation for each of the interactions in question with their level predicted by our model, for each DBMS.

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) or was close (e.g., predicted high and actual medium), for the majority of predicted interactions. For some interactions, the significance of the correlation was greater than 0.05, which means that there was insufficient data to determine if that interaction was present.

We now examine those few interactions where the prediction did not match the actual observed level of correlation, for each DBMS.

For DB2, there are five interactions of concern. Considering overall major faults, while we predicted the correlation with query major faults (interaction (*VId*)) would be low and with utility major faults (*VIe*) and with daemon major faults (*VIf*) would be medium, the actual correlations were query, high, and utility and daemon, not correlated with overall major faults. Looking into this further, we discovered that the *first* execution of a query in DB2 often experienced a few major faults and that these were often the only major faults in the execution, and so query dominated the total major faults.

The other discrepancy for DB2 was with the query user time and overall IO Wait ticks (*IVb*), which we predicted would have medium correlation but in fact were not correlated at all. We discovered that less than 2% of the DB2 query executions had *any* IO Wait ticks. That number of samples was too low to do a separate correlational analysis. This also explains the fifth interaction of concern, that of overall user time and overall IO Wait ticks (*Vb*).

For MySQL, there were three interactions of concern. The first is between overall soft IRQ and daemon system time (*IIg*), which was predicted to be low but whose actual correlation was high. Most queries in mySQL took a long time to complete. This resulted in both query and daemon processes causing a large number of major faults, so that both dominated the correlation between soft IRQ and system time and high correlations and observed correlations of high for both.

The second is *IVb*, discussed above. For MySQL, two things are going on. First, for many queries, there are no overall IOwait ticks. But there are also some long-running MySQL queries, in which the IOwait ticks accumulate, though still in low numbers: perhaps one or two dozen, representing less than 1% of the query user time. In both situations, the IOwait ticks is such a small component of the run time that no correlation was observed. Again, the exceptions, 0.97%, were so infrequent as to disallow a separate correlational analysis.

Finally, our model predicts a medium correlation for overall user time and overall IO Wait time (*Vb*), but no such correlation was observed. It seems that again, this is because of the very low amount of IO Wait ticks present in most of the query executions. For Oracle, we found that interaction (*IVf*), utility major faults and overall IO Wait ticks, was predicted to be low but was actually high. This is probably because the Oracle utility processes were being run intermittently, causing major faults which added to IO Wait ticks when contended with the Oracle query process. This also explains interaction (*Vd*), overall major faults and overall IOwait ticks, from expected low to actual high. For PostgreSQL, there was only one discrepancy (*IIg*), already discussed.

In summary, for the 108 testable interactions (27 interactions for each of four DBMSes), we encountered only eleven that were of concern, none of which presents a serious challenge to the model. Our conclusion is that the model is strongly supported by these experimental results. As we will see in Section 7.5, after cleaning up the query executions in Steps 2 and 3 of our protocol, the number of interactions of concern reduced dramatically.

We now delve into the details of the timing protocol.

7. TIMING PROTOCOL

In the following, we first 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.

7.1 Data Collection

Since the time to collect the per-type metrics is much longer than that taken to collect the overall metrics, we perform measurements in the following order: `getProcs()`→ `getProcStat()`→ `getTime()`→*execute query*→`getTime()`→ `getProcStat()`→`getProcs`, after which we compute the differences in the cumulative statistics.

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 useful to retain all of this data for *every* Q@C execution, but recording only those processes for which some activity was observed during the query execution.

7.2 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.

As illustrated in Figure 4, processes *P1*, *P2* and *P3* are recorded into a map, M_1, when the first `getProcs()` is invoked. Process *P4* might also appear in M_1, depending on exactly when it started (as `getProcs()` takes a while to execute, going through the processes one by one). Another map, M_2, built after the query has executed, records processes *P3*, *P6* and *P7*, and it might also have *P8*. We term process *P5* a *phantom process*, since it appears and completes its task before the query execution is finished, and thus will appear in neither M_1 or M_2. However, we can detect the presence of phantom process(es) by comparing the total number of executed processes (forks) extracted by `getProcStat()` before and after the query evaluation. Process *P4*, even if it is recorded in M_1, will not be considered to be a phantom process, because it did not start after the first `getProcStat()` was invoked.

The comparison between M_1 and M_2 reveals that processes *P1* and *P2* are present in M_1 but not in M_2. We term these processes *stopped processes*. It is evident that process *P2* overlaps the query execution, but it is not easy to differentiate process *P1* from *P2*. For example, both could have user ticks, even though only process *P2* actually overlapped with the query execution, contributing time measured by `gettime()`. If process *P4* was recorded in M_1, then it will also be considered a stopped process.

Processes *P6* and *P7* are located in M_2 but not in M_1. Process *P8* may also be in this situation as well, if it was caught by `getProcs()`. If caught, its ticks will be captured

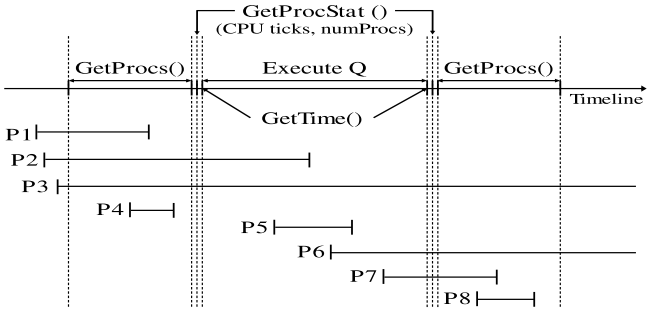


Figure 4: Processes considered when timing a query

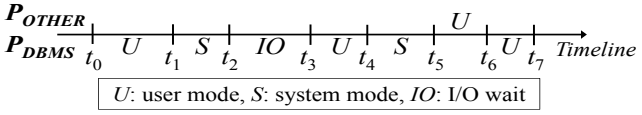


Figure 5: Two interleaved, running processes

in the per-type metrics, but not in the processor metrics.) We term these *started processes*. Note however that we know the amount of user and system time these processes incurred, because that information will be in M_2, unlike that for phantom processes.

7.3 Calculating the CPU Time

Generally, the CPU time is easy to calculate because we have per-type system and user time. However, we need to 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 as the one that is performing the query. Note that this could select the wrong process, if the query process ran very quickly and a DBMS utility process performed a lot of work during that same time frame. Hence, if the selected process does not appear in every query execution for a Q@C, we will drop that Q@C in Step 2-(i), below.

7.4 Calculating the I/O Time

Every tick can be in one of several modes [3]: running a user process (*user 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 softirq (*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*). The first three modes are illustrated in Figure 5, for two processes, the DBMS process and another process. Here the DBMS process starts in user mode for one tick, then switches to system mode, then to IOwait mode.

Our error model is that in the absence of non-query processes (utility and daemon), the measured time should be accurate, 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, with one execution each with 20, 27, and 53 processes, out of the 350,000 executions examined.)

Algorithm measurementProtocol(*query*, *card*):

{turn off non-essential O/S daemons}
plan ← *GetQueryPlan*(*query*)
for *iterNum* ← 1 **to** 10 **do**
  timeSingleQuery(*iterNum*, *query*, *plan*, *card*)
end for
for *iterNum* ← 1 to 10 **do**
  dropQueryExecutions()
end for
if *dropQ@C*() **then**
  return null {Drop some Q@C's}
else
  calculateQueryTime() {Over remaining Q@C's}
end if

Figure 6: Measurement Protocol

For our running example, the per-type metrics shown in Table 4 support this model. (In this table, the lowest and highest values for each row are in **bold**.) Note how stable the user times (U_{Qdbms}) and system times (S_{Qdbms}) are for the DBMS process performing the query ($SUquery = S+U$), with $SUquery$ varying by only 7 ticks (less than 2%). For PostgreSQL, there is only one DBMS process, so we don't see any $SU_{utility}$ ticks (nor major faults, $MajFlt_{utility}$). For these executions, PostgreSQL has already faulted in all of the code to run the query, so $MajFltquery$ is zero (though we do see non-zero values for this metric). The variance 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.

So the challenge is in apportioning the $IOwait$ ticks to the query process. The model shown in Figure 3 and validated in Section 6 provides guidance on how to do so.

7.5 Query Execution Time Computation

Recall that we start with a fixed number (10) of query executions for each Q@C. We desire an operationalization of the query execution time for that Q@C, which is a single time in milliseconds.

Our general protocol is to (1) perform sanity checks to ensure the validity of the data, (2) perhaps drop some query executions via clearly motivated predicates, (3) perhaps drop some entire Q@Cs, again via clearly motivated predicates, (4) for those Q@Cs that remain, for each 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.

We now detail each step of this protocol, shown in Figure 6. We apply the protocol to our data, consisting of a total of 353,630 query executions (35,363 Q@Cs) over the four DBMSes.

	1	2	3	4	5	6	7	8	9	10	*Avg*	*Std Dev*
$Time_{meas}$ (msec)	**6530**	6571	**8764**	7961	8427	7829	8246	8506	8239	6991	7806	818
U_{total} (ticks)	303	315	**308**	300	—	303	301	**299**	309	—	305	5.1
L_{total} (ticks)	4	**0**	5	2	—	2	3	**26**	24	—	9	9.8
S_{total} (ticks)	171	**166**	171	183	—	176	176	**188**	185	—	177	7.7
Q_{total} (ticks)	**24**	27	44	42	–	39	43	**63**	48	–	41	11.4
$Idle_{total}$ (ticks)	0	0	0	0	—	0	0	0	0	—	0	0
IO_{total} (ticks)	150	**149**	**349**	269	—	262	301	275	258	—	253	65.1
$Time_{sum}$ (msec)	**6520**	6570	**8770**	7960	—	7820	8240	8510	8240	—	7829	845.4

Table 3: Measured time versus computed sum

	1	2	3	4	5	6	7	8	9	10	*Avg*	*Std Dev*
$Time_{meas}$ (msec)	**6530**	6571	**8764**	7961	8427	7829	8246	8506	8239	6991	7806	818
U_{query} (ticks)	289	293	**294**	286	—	287	288	**283**	290	—	289	3.6
S_{query} (ticks)	166	**161**	164	171	—	168	168	**177**	166	—	168	4.8
SU_{query} (ticks)	455	454	458	457	—	**453**	456	460	**460**	—	457	2.6
$SU_{utility}$ (msec)	0	0	0	0	—	0	0	0	0	—	0	0
SU_{daemon} (msec)	33	**37**	41	42	—	41	43	79	**84**	—	50	19.7
$MajFlt_{query}$	0	0	0	0	—	0	0	0	0	—	0	0
$MajFlt_{utility}$	0	0	0	0	—	0	0	0	0	—	0	0
$MajFlt_{daemon}$	17	**0**	7	**18**	—	0	0	0	0	—	5	7.9
$IOwait_{meas}$ (ticks)	150	**149**	**349**	269	—	264	301	275	258	—	252	69.5
$SoftIrq_{meas}$ (ticks)	**24**	27	44	42	—	39	42	**63**	48	—	40	14.2

Table 4: Breaking out the per-type metrics

Step 1: Perform Sanity Checks.

Before starting the query execution time computation, we assess the validity of our data by performing several *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.)

We have three different classes of such sanity checks, comprising twelve sanity checks in total.

Table 5 lists the first class: the overall cases for which not a single violation should occur in our data. The number of missing queries indicates how many queries were not executed, for whatever reason. The number of process info failures represents how many query executions do not have data stored about its processes, specifically, overall query execution information: overall ticks, number of started/stopped/executed processes, phantom presence, total faults, and total daemon time. Finally, the unique plan violation checks how many Q@Cs have more than one query plan. All query executions of Q@C must have a single, identical query plan.

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.

The second class of sanity checks concerns query executions; see Table 6. Each of these six sanity checks could encounter a few isolated violations. However, we expect the violation percentage to be very low.

The DBMS time violation check identifies the percentage of query executions with the time taken by DBMS processes (query and utility) in total less than the time taken by daemon processes, since we expect query execution time to dominate. Fortunately, only about 0.01% DBMS time violations were observed across all query executions in our data. Second, the *zero* query time violation check indicates how many query executions have a query time of 0 ticks. We would not expect such query executions. Our data showed only 0.01% violations as well. Third, the query time violation check identifies the query executions in which the query time is greater than measured time. In our data, we uncovered 0.52%, which is a very low rate.

For the last three sanity checks in this class, one could be detected but should be very low and the other two should not be violated by any query execution. The no query process violation check indicates how many query executions do not have any query process. We got very low percentage (0.19%), with most of these query executions at the minimum cardinality (10K) or with a very short measured time, less than 10ms. Finally, we ensure that (query or utility) processes from another DBMS should not be running. We thus check how many query executions include other query or utility processes. There were no such query executions.

The final class involves three checks over each Q@C; see Table 7. The first detects excessive query process time variations, those in which the standard deviation of the query time (user mode ticks plus system mode ticks) is greater than 20% of the average query process time. We observed few violations. 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. 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 Due to the unavoidable variation of query times, we also

Number of Missing Queries	0
Number of Process Info Failures	0
Number of Unique Plan Violations	0

Table 5: Overall sanity checks

Percentage of DBMS Time Violations	0.01%
Percentage of Zero Query Time Violations	0.01%
Percentage of Query Time Violations	0.52%
Percentage of No Query Process Violations	0.19%
Percentage of Other Query Process Violations	0%
Percentage of Other Utility Process Violations	0%

Table 6: Query execution sanity checks

Percentage of Excessive Var. in Query Time	0.05%
Percentage of Strict Monotonicity Violation	0.29%
Percentage of Relaxed Monotonicity Violation	0.073%

Table 7: Q@C sanity checks

At Start of Step 2	353,630 QEs
At Start of Step 2	35,363 Q@Cs
After Step 2-(i)	350,830 QEs
After Step 2-(ii)	251,270 QEs
After Step 2-(iii)	249,110 QEs
After Step 2-(iv)	247,867 QEs
At Start of Step 3	31,658 Q@Cs
After Step 3-(i)	31,658 Q@Cs
After Step 3-(ii)	31,374 Q@Cs
After Step 3-(iii)	26,841 Q@Cs

Table 8: The number of Query Executions (QEs) and Q@Cs remaining after each sub-step

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.

Now that we see that the original data from our experiment passes all of these sanity checks, we can proceed onto the next step.

Step 2: Drop Query Executions.

In this step, we drop query executions that exhibit specific problems, in order to increase accuracy and precision.

First, we drop (i) the query executions identified as problematic in Table 6 as well as the first row of Table 7 to increase the precision of our measured query time. Table 8 shows how many query executions are valid at the beginning of Step 2 and after Step 2-(i): about 0.79% query executions were dropped.

Referring back to Figure 4 showing the interleaved processes, the total time (from the algorithm, $time_2 - time_1$) includes the time taken by the DBMS, the time taken by *P5*, and portions of processes *P2*, *P3*, *P5*, *P6*, and *P7*. For most of these processes, we have per-process information from `getProcs()`. Not so for the stopped process *P2* nor for the phantom process *P5*, because neither of those processes show up in map M_2. Due to this potentially large uncertainty, we (ii) drop any query execution that contains a stopped or phantom process.

Percentage of Excessive Var. in Query Time	0%
Percentage of Strict Monotonicity Violations	0.38%
Percentage of Relaxed Monotonicity Violations	0.076%

Table 9: Post (Q@C) sanity checks

If *P4* was included in M_1, then it will also be (conservatively) considered to be a stopped process (we know it started before the query and we only know it ended before we could get its per-process metrics). If that process was not included in M_1, then we don't need to worry about it.

Started processes (in this case, *P6* and *P7*) do not cause a problem, because we have per-process information for these processes in M_2. Even if process *P8* is included in M_2 that process will be ignored, as it is not a DBMS process.

We also (iii) drop query executions where the IOwait ticks is greater than two times the median IOwait ticks for the Q@C, when the median IOwait ticks is greater than zero. The rationale is that such a large number of IOwait ticks over the median could have only be caused by other processes. Because we allocate IOwait ticks in our computation of $Time_{calc}$, we need to be conservative.

When the median IOwait ticks is zero, a minority of query executions have a non-zero IOwait ticks. In such cases, we drop query executions for that Q@C whose number of IOwait ticks is greater than 2, because such query executions will be outliers (again, to be conservative).

The discussion in Section 6 identified two places where the overall IOwait ticks was problematic: when it was non-zero for DB2 (for a very small percentage of queries) and when it was significant for MySQL (also a very small percentage). Thus we (iv) drop DB2 query executions with non-zero IOwait ticks and drop MySQL query executions for non-zero IOwait ticks for which this time represents more than 1% of the query user time. The net take away is that for these two DBMSes, we need not concern ourselves with an adjustment to apportion IOwait time.

For our running example of query 17, execution 5 had one phantom and execution 10 had one phantom and one stopped process, so we drop those two query executions in Step 2-(ii). No other query executions from this query were dropped, as shown in Table 3.

Table 8 shows how many query executions remained after each sub-step in Step 2. As mentioned, 0.79% were dropped in Step 2-(i). 28.38% query executions were dropped due to stopped or phantom processes at Step 2-(ii). After Step 2-(iii), 0.86% query executions were dropped due to the high IOwait ticks. After Step 2-(iv), 0.50% query executions were dropped due to problematic IO Wait ticks from DB2 and MySQL. In summary, Step 2 in concert dropped 29.91% query executions, and based on the remaining query executions (247,867), a total of 31,658 Q@Cs were left, with each Q@C retaining an average of 7.83 query executions.

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 7.3, very short execution times result in the wrong query process being chosen. Hence, we (i) drop any Q@C for which the identified query process does not appear in every query execution for that Q@C, (ii) drop Q@Cs with $Time_{meas}$ (average measured time across remaining executions) of 2 or less ticks, and (iii) drop those Q@Cs with less than six valid query executions. For our running example, none of these predicates dictated dropping this Q@C.

Table 8 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 31,658 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 0.90% Q@Cs at Step 3-(ii) and 14.5% at Step 3-(iii). Throughout Step 3, a total 15.2% of Q@Cs were discarded; each remaining Q@C had an average of 8.52 query executions.

We expect that cardinality of the result of each each successive step to monotonically decrease. As shown in Table 8, the cardinalities behave as expected.

Step 4: Calculate Query Time.

In this calculation, we include (a) DBMS user time, (b) DBMS system time, and (c) the portion of IOwait time due to DBMS I/O requests. Due to Step 2-(iii), the data from DB2 and MySQL will not have a meaningful number of IOWait ticks so component (c) does not pertain to these DBMSes.

According to the model shown in Figure 3 and validated in Section 6, there are 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. We ran the following regression over the Q@C executions.

$$\begin{aligned} IOwait_{meas} &= a + b \times U_{query} \\ &\quad + c \times MajFlt_{query} \\ &\quad + d \times MajFlt_{utility} \\ &\quad + e \times MajFlt_{daemon} \end{aligned} \tag{1}$$

This regression fits quite well with our error model. The variance explained is 67% for Oracle and 38% for PostgreSQL (due to the different characteristics of various DBMSes, we compute regression coefficients for each individual DBMS). None of the factors on the right dominate or appear to be collinear with $IOwait$. However, the $MajFlt_{query}$ factor is not a significant contributor; thus we do not use this factor.

The y-intercept for Oracle is 67 ticks and for PostgreSQL is 38 ticks. (Note that the model predicts that this intercept should be 0, though exogenous factors such as process startups and the latent variable contribute to this value).

We can then use the regression coefficient associated with the query process to determine the contribution of the IOwait time to the total query time. (As noted before, there is no reason to scale the IOwait time for DB2 nor for MySQL.)

$$IOwait_{calc} = b \times U_{query} \tag{2}$$

$$\begin{aligned} Time_{calc} &= (S_{query} + U_{query} + IOwait_{calc}) \\ &\quad \times 10 msec \end{aligned} \tag{3}$$

The regression (evaluated over the retained query executions) produced the needed regression coefficients of b (the multiplier for U_{query}) of 1.916 for Oracle and 0.259 for PostgreSQL. The resulting computed execution times ($Time_{calc}$) are shown in Table 10, with the median, 5308.3 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 ± 1.7 ticks, or $\pm 0.3\%$. Across all runs, the computed time is within one average standard deviation of ± 2.9 ticks, or ± 1.2. Our conjecture is that unmeasurable factors such as thread synchronization are contained in this variance. In addition, we can report (see Figure 1) that the overall wall clock time ranged from 6.5 to 8.8 seconds, with an average of 7.38 sec ± 1.2 sec. The DBMS CPU time is 4570 ± 35 msec and the DBMS I/O time is 750 ± 1.4 msec. We could also report similar measures for the utility and daemon processes.

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.

Table 9 shows our post sanity check results. After our protocol was applied, no excessively varying query times were detected. Also, the rates of monotonicity violations increased just slightly. The total number of violations itself was reduced but relatively more Q@Cs were dropped by our protocol.

We reran the confirmatory correlation analysis discussed in Section 6 on the refined data. For DB2, four of the five interactions of concern in that prior analysis are now closer to the predicted, at most one level different, with three matching the predicted level exactly,. For MySQL, the number of interactions of concern went from three to one, for Oracle, two to one, and for PostgreSQL, the one concern was resolved. In sum, the refined data reduced the number of interactions of concern from 11 to 3, lending even stronger support for the causal model in Figure 3.

8. SUMMARY AND FUTURE WORK

Measuring query time is complex, 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 articulated a structural causal model relating these measures. A thorough correlational analysis provided strong support for this model. Using this model, we developed a 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 Q@C executions, (iii) perhaps drops some entire Q@Cs, (iv) for those Q@Cs that remain, for each compute a single query time using the underlying measures of the remaining query executions, and finally, (v) does some final sanity checks. This protocol results in a more precise and more accurate timing of the query, reducing variance significantly.

	1	2	3	4	5	6	7	8	9	10	*Avg*	*Std Dev*
$Time_{meas}$ (msec)	**6530**	6571	**8764**	7961	8427	7829	8246	8506	8239	6991	7806	818
SU_{query} (ticks)	455	**454**	458	457	—	453	456	**460**	460	—	457	2.6
$MajFlt_{Qdbms}$ (msec)	0	0	0	0	—	0	0	0	0	—	0	0
$IOwait_{meas}$ (ticks)	150	**149**	**349**	269	—	264	301	275	258	—	252	69.5
$IOwait_{calc}$ (ticks)	74.9	75.9	**76.1**	74.1	—	74.3	74.6	**73.3**	75.1	—	74.8	0.9
$Time_{calc}$ (msec)	5298.5	5298.9	**5341.5**	5310.7	—	**5293.3**	5305.9	5333.0	5311.1	—	5311.6	17.1

Table 10: Final computed times

The *Tucson Protocol* is quite general, applicable to most versions of Unix that support `/proc`, and is also applicable to other operating domains in which measurements of multiple processes each doing computation and I/O is needed. While many of the specifics, such as clearing caches before executing a query, are well-known (though not well-documented), this is the first general query evaluation time measurement protocol to be articulated.

In subsequent work we plan to refine 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 block read and write statistics available from the DBMSes and bytes read and written from the O/S, (c) accommodate multiple disks, connected by a single or distinct channels, (d) accommodate multiple processor cores, (e) accommodate phantom processes while eliminating their impact on the computed time, (f) extend PostgreSQL to clear its cache, (g) ensure repeatability of file fragmentation, (h) support the Windows operating system, which has different per-process metrics, and thus might require an altered causal model and a different regression model and calculation of query time, and (i) accommodate multiple disks. We also want to extend the protocol to (a) measure single transactions that incorporate multiple statements and (b) measure a mix of transactions.

9. ACKNOWLEDGMENTS

This research was supported in part by NSF grants IIS-0639106, IIS-0415101, and EIA-0080123. We thank Benjamin Dicken, Preetha Chatterjee, Pallavi Chilappagari, David Gallup, Kevan Holdaway, Andrey Kvochko, and Lopamudra Sarangi for their contributions to the AZDBLab and Phil Kaslo, Tom Lowry, and John Luiten for constructing and maintaining our experimental instrument, a laboratory of ten machines and associated software. Phil was particularly helpful in tracking down irq behavior. Finally, we thank Nikolaus Augsten for many helpful comments.

10. REFERENCES

[1] A. G. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSes on modern processors: Where does time go?" University of Wisconsin Computer Sciences Technical Report 1394, February 25, 1999.

[2] M. Akswew, U. Cetintemel, M. Riondato, E. Upfal, and S. B. Zdonik, "Learning-based Query Performance Modeling and Prediction," in *ICDE'12*, pp. 390–411.

[3] D. Bovet, and M. Cesati, "Understanding the Linux Kernel, Third Edition," O'Reilly, 2003.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the ACM Symposium on Cloud Computing*, New York, June 2010, pp. 143–154.

[5] R. F. Forman, J. M. Pechacek, and W. H. Schwane, "Apparatus and Method for Measure Transaction Time in a Computer System," IBM Patent, 2001.

[6] A. Ganapathi, H. A. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. I. Jordan, and D. A. Patterson, "Predicting multiple performance metrics for queries: Better decisions enabled by machine learning," in *ICDE'09*.

[7] L. Gikoumakis and C. Galindo-Legaria, "Testing SQL Server's Query Optimizer: Challenges, Techniques and Experiences, *IEEE Data Eng. Bull.* 31(1): 36-43 (2008).

[8] H.-Y. Hwang and Y.-T. Yu, "An Analytical Method for Estimating and Interpreting Query Time," in *VLDB'78*.

[9] Working Group 2 of the Joint Committee for Guides in Metrology (JCGM/WG 2), **International Vocabulary of Metrology—Basic and General Concepts and Associated Terms**, 2008.

[10] K. Kant, **Introduction to Computer System Performance Evaluation**, McGraw-Hill, 1992.

[11] M. Stillger, G. M. Lohman, V. Markl, and M. Kaqndil, "LEO - DB2's Learning Optimizer," in *VLDB'01*.

[12] TPC, "Transaction Processing Performance Council—TPC-C," `http://www.tpc.org/tpcc/`. (accessed August 29, 2010)

[13] TPC, "TPC Transaction Processing Performance Council—TPC-H," `http://www.tpc.org/tpch/`. (accessed August 29, 2010)

[14] Valgrind Developers, "Callgrind: A Call-Graph Generating Cache and Branch Prediction Profiler," `http://valgrind.org/docs/manual/cl-manual.html`. (accessed October 27, 2010)

[15] S. W. Thomas, and R. T. Snodgrass and R. Zhang, "τBench: Extending XBench with Time," TimeCenter TR-92, 2010.

[16] B. B. Yao, M. T. Özsu, and N. Khandelwal, N., "XBench benchmark and performance testing of XML DBMSs," in *ICDE'04*.

[17] P. Yu, M.-S. Chen, H.-U. Heiss, and S. Lee, "On workload characterization of relational database environments," *IEEE Transactions on Software Engineering*, 18:347–355, 1992.

[18] N. Zhang, J. Tatemura, J. M. Patel, and H. Hacõgumus, "Towards Cost-Effective Storage Provisioning for DBMSs," *PVLDB*, Vol. 5, No. 4, pp. 274-285 (2011)

[19] R. Zhang, R. T. Snodgrass and S. Debray, "Micro-Specialization in DBMSes," in *ICDE*, pp. 690–701, April 2012.