Sentinel: Universal Analysis and Insight for Data Systems

Brad Glasbergen, Michael Abebe, Khuzaima Daudjee, Amit Levi
Cheriton School of Computer Science, University of Waterloo
{bjglasbe,mtabebe,kdaudjee,amit.levi}@uwaterloo.ca

ABSTRACT

Systems continue to grow in complexity in response to the need to support vast quantities of data and a wide variety of workloads. Small changes in workloads and system configuration can result in significantly different system behaviour and performance characteristics. As a result, system administrators and developers spend many hours diagnosing and debugging performance problems in data systems and the applications that use them. In this paper, we present Sentinel, an analysis tool that assists these users by constructing fine-grained models of system behaviour and comparing these models to pinpoint differences in system behaviour for different workloads and system configurations. Importantly, Sentinel's insights are derived from built-in debug logging libraries without necessitating that their log messages be written to disk, thereby generalizing to all systems that use debug logging without incurring its overheads. Our experiments demonstrate Sentinel's superiority in analyzing the execution behaviour and performance characteristics of database systems, client applications, and web servers compared to prior approaches.

PVLDB Reference Format:
Brad Glasbergen, Michael Abebe, Khuzaima Daudjee and Amit Levi. Sentinel: Universal Insight and Analysis for Data Systems. *PVLDB*, 13(11): 2720-2733, 2020.
DOI: https://doi.org/10.14778/3407790.3407856

1. INTRODUCTION

To meet the demands of many different workloads, data systems have risen in complexity from both a configuration and behavioural standpoint [21, 33]. Database systems in particular are well-known to have a large number of configuration knobs that play a significant role in performance [33] and thousands of recorded metrics with which to analyze it [37]. When a workload performs poorly, a highly-trained administrator spends hours combing through execution logs and fine-grained metrics to determine whether the workload contains characteristics that cause poor performance (e.g., lock contention, high I/O overhead, poorly written queries) or the database is misconfigured (e.g., inadequately sized buffer pool, excessive checkpointing). Similar analysis must often be conducted on the application machines generating the workload when performance degrades to understand *what* the application is doing and *how* it induces the behaviour in the database system.

Making matters worse, each data system and client application can have its own set of debugging tools, recorded metrics, and performance characteristics. Methods to diagnose a performance problem in one data system are likely to differ from those appropriate for another, and again differ from those suitable for client applications. Therefore, administrators and developers must cultivate a deep understanding of multiple systems and their debugging tools because organizations frequently use different types of data systems (e.g., database systems, web servers) for different workloads.

Although previous work has developed specialized analysis tools for individual systems [3, 17, 37], these tools depend on characteristics and features of their targeted system. Porting these tools to other systems is challenging because a developer must determine which characteristics and metrics in the new system are important and relevant, and instrument the code as necessary to extract them. Doing so requires substantial implementation and feature engineering effort. While other analysis tools generalize across systems by exploiting outputted debug logs to compare and contrast system behaviour [12, 14, 24, 34], the overheads of fine-grained debug logging are typically prohibitive [35, 37]. Further complicating matters, these analysis tools often require system-specific preprocessing scripts that apply domain knowledge to transform the logs into a specific format suitable for analysis.

In this paper, we present **Sentinel**, a system behaviour and performance analysis tool that avoids the pitfalls of prior work by integrating with built-in application debug logging libraries. Sentinel intercepts logging library calls, builds models of system behaviour in memory, and provides novel types of analysis over these extracted models. Using Sentinel's reports, administrators and developers can rapidly remedy performance problems and understand shifts in client application behaviour. Sentinel's insights are available to any system or application using debug logging libraries, without significant code modification. As it is the norm to use libraries such as Log4j2 [6], Google logging [11], spdlog [22] to help debug system software Sentinel supports a wide array of systems and applications. Importantly, Sentinel does not require that fine-grained application debug

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407856

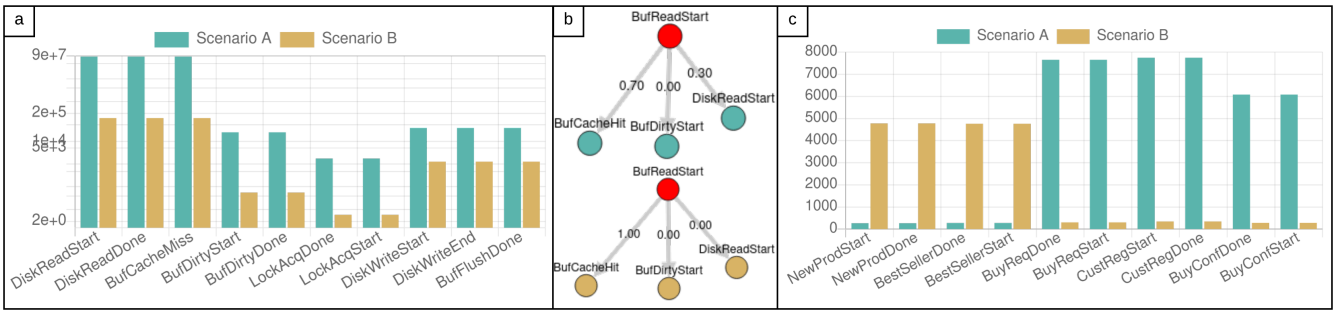


Figure 1: Sentinel presenting differences in PostgreSQL system and application behaviour for Scenario A and Scenario B: (a) contains the largest differences in PostgreSQL behaviour by proportion of messages (b) presents a subset of the transition model relating to buffer accesses (c) presents the largest differences in application behaviour by proportion of messages.

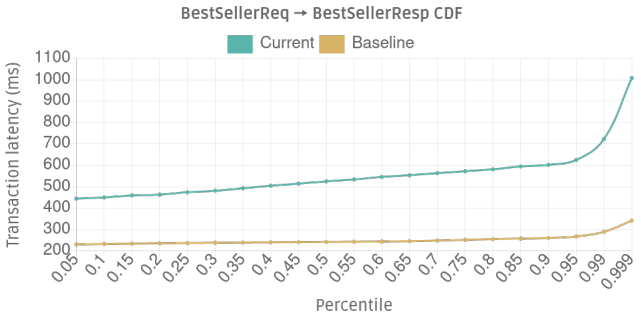


Figure 2: A CDF extracted by Sentinel that compares the BestSellers transaction latency in Scenario A and B.

logs are emitted to disk; by intercepting the logging calls and constructing its models in memory, Sentinel has little overhead. As log calls correspond to system events and execution paths taken in the code [12,14,41], Sentinel's analysis describes differences in *system behaviour*.

To demonstrate Sentinel's usefulness to administrators and developers, we configure PostgreSQL 9.6 to use Sentinel's tracing and analysis library and consider two different scenarios. In Scenario A, we execute the TPC-W [31] ordering mix using the standard PostgreSQL configuration. In Scenario B, we execute the TPC-W browsing mix using a PostgreSQL configuration with a much larger 8 GB buffer pool. We also integrate the TPC-W benchmark clients' logging with Sentinel to provide insight into the benchmark clients. Selected images from Sentinel's user interface pinpointing behavioural differences in both the database and clients are shown in Figure 1.

System Behaviour Differences: Sentinel reports that the largest differences in PostgreSQL execution behaviour between the two workloads relates to disk I/O (Figure 1a). In execution Scenario A, PostgreSQL must read far more pages from disk than in execution Scenario B. This increase in disk I/O is induced by the increase in cache misses (BufCacheMiss in Figure 1a) in Scenario A; when a page needed for query execution is not available in memory, it must be read in from disk. Scenario A also blocks while acquiring locks more often than Scenario B, which (as shown later) is a consequence of an increase in the number of update transactions that acquire table-level exclusive locks.

Event Transition Models: Beyond these event-level comparisons, Sentinel enables deeper analysis by building models of *transitions* between log messages. In particular, for each log message, Sentinel reports which messages are likely to be reached (or transitioned to) next and with what probability. This *event transition model* enables developers to evaluate the performance of more complex system functionality. Consider the first event transition for a buffer page read in PostgreSQL (Figure 1b). When a PostgreSQL process needs to access a page, it checks to see if it is already mapped to a page in memory (i.e., in the buffer pool). If so, the buffer pool page can be reused; this is termed a buffer pool (cache) hit. Otherwise, PostgreSQL finds an available buffer and reads the page into it from disk. In cases where the chosen buffer's contents have been updated (i.e., the page is dirty), the updated contents are first flushed to disk before overwriting the buffer with the desired page's data. Using the transition model, users are able to verify that the system is performing as expected and identify condition-based behaviour differences. In this example (Figure 1b), the buffer pool hit rate for Scenario A is only 70%, while Scenario B's is over 99%. These differences in hit rate manifest in the disk read differences shown in Figure 1a.

Workload Behaviour: As Sentinel's insights are available to all applications using debug logging, we now apply the same techniques we used to analyze PostgreSQL to understand benchmark client application execution. As these application clients generate the workload against PostgreSQL, understanding their behaviour provides insight into the workload and presents optimization opportunities. For example, Sentinel reports that the largest differences in client execution between the scenarios correspond to differences in the mix of transaction types they execute (Figure 1c). Scenario A executes far more transactions related to book orders, while Scenario B executes more read-dominant transactions. These differences explain the increased lock contention we observed in Scenario A.

Cumulative Distribution Functions (CDFs): Sentinel efficiently builds empirical CDFs of the time between different log messages (*transition times*) and encodes these CDFs into the transition model. By using CDFs, Sentinel captures tail-latency effects that frequently dominate performance but that cannot be detected using averages. For example, consider the CDF that Sentinel extracted for BestSeller transaction latency from the benchmark clients (Figure 2). We observe that the transaction's latency is consistently higher for Scenario A than for Scenario B, which indicates that the cause behind this performance degradation is substantial and consistent. Further, the CDF captures the effects of contention at the tail, which are significant. In

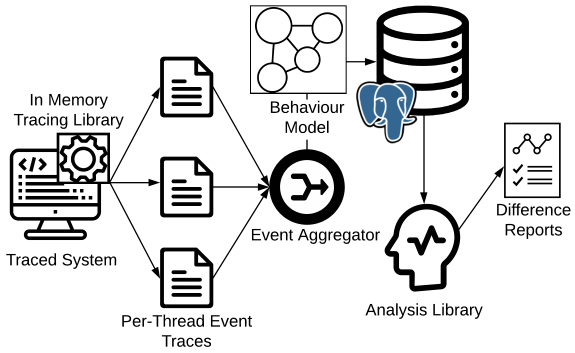


Figure 3: The Sentinel system architecture.

addition to these CDFs for individual transitions, Sentinel can combine CDFs for multiple transitions to highlight differences in performance for system functionalities that span multiple log messages using random walks. For example, in Section 3, we construct and contrast CDFs for total buffer read times in both scenarios.

Our prior demo paper [10] illustrated Sentinel's utility as a workload and system understanding tool for different workloads and system configurations. This research paper presents and evaluates the techniques it uses to provide this utility to administrators. We present Sentinel's lightweight event and transition tracking techniques, as well as its methods for generating models that effectively summarize system execution behaviour. We also highlight Sentinel's techniques to rank behavioural differences, which our experiments show are highly effective and efficient. Importantly, Sentinel's insights are *universally* available in that its techniques can be used on any system using debug logging — which has become ubiquitous [7, 24, 35, 37] — without incurring the overheads of emitting detailed debug logs to disk.

The rest of this paper is organized as follows. Sections 2 and 3 present examples that progressively reveal features of Sentinel's functionality followed by descriptions of the techniques behind them. Section 4 describes details of the Sentinel system and Section 5 presents an experimental comparison of Sentinel's techniques against targeted performance diagnosis frameworks. We discuss the implications of Sentinel's techniques in Section 6, compare Sentinel to related work in Section 7, and conclude in Section 8.

2. UNIVERSAL ANALYSIS

Sentinel's model extraction and analysis techniques are implemented as three distinct components — an *in-memory tracing module*, an *event aggregator*, and an *analysis library* (Figure 3). As all of the analysis components depend on the ability to extract system events from a running system with low overhead, we start by describing Sentinel's in-memory tracing module.

To support the workflow in Section 1 while generalizing across systems, Sentinel's tracing module and associated extraction techniques provide the following key features:

1. *Sentinel easily integrates with a wide range of systems and applications.*

2. *Sentinel extracts all of its data from a running system without significantly degrading performance.*

```
pid = fork_process();
if(pid>0) {
  /* in parent, successful fork */
  log(DEBUG2, "forked new backend, pid=%
    d socket=%d",
  (int) pid, (int) port->sock)));
}
```

Figure 4: A simplified PostgreSQL debug logging call issued while starting a new backend process.

As one of Sentinel's primary goals is to provide behavioural analysis to a broad range of systems, its extraction techniques must not rely on system-specific features or metrics. Furthermore, Sentinel must minimize integration effort; requiring engineers to design and build complex system-specific scripts to integrate with Sentinel defeats its goals of generalizability and hinders its adoption. Finally, Sentinel must not hamper system performance. If Sentinel significantly slows down system performance, then it will not be used regardless of its utility in detecting and explaining system behaviour. Next, we describe how Sentinel meets these challenges.

2.1 Low Overhead Event Extraction

Nearly all systems use debug logging [7, 24, 35, 37]; Sentinel exploits this fact to provide universal behaviour analysis. We start by describing an overview of debug logging libraries and their pitfalls before discussing how Sentinel exploits them without incurring their overheads.

Popular debug logging libraries [6, 11, 22] all use a similar interface to output a debug log message to file:

```
log(LOG_LEVEL, message, format_args)
```

where `LOG_LEVEL` indicates the granularity of the log message, `message` is a string with placeholders for variables, and `format_args` contains the variables to be spliced into the message. For example, the PostgreSQL 9.6 code shown in Figure 4 has a logging call with `LOG_LEVEL` DEBUG2 and splices the variables `pid` and `port->sock` into the log message. `LOG_LEVEL`s range from extremely fine-grained (`DEBUG5`, `trace`) to coarse (`WARNING`, `ERROR`), and the logging library is configured when the system starts to emit messages of coarser granularity than a specified threshold to disk. For example, if the logging library is configured to emit only messages above `WARNING`-level, `ERROR`-level messages would be written to disk, but `DEBUG5`-level messages would not. This feature enables developers to output detailed information during debugging sessions while avoiding the associated performance overheads in live deployments.

It is well-known that detailed logging results in considerable performance overheads [35, 37]. Broadly speaking, there are two issues that lead to performance degradation when using fine-grained logging. First, debug logging libraries often incur synchronization overheads in the presence of multi-processing and multi-threading. With detailed logging enabled, this synchronization may result in considerable overhead. Second, messages emitted by the debug logging library are traditionally persisted to disk for later analysis. Although log messages may be buffered in memory and asynchronously written out to persistent storage as a batch, the costs of writing out detailed logs remain substantial.

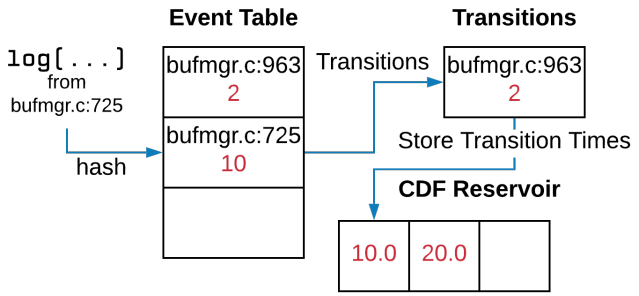


Figure 5: Sentinel's data structures used for tracking log messages, transitions, and empirical CDFs.

Sentinel avoids these overheads by integrating directly with debug logging libraries to track events and event transitions in memory. In particular, when a running system issues a `log` call, the debug logging library first forwards the call to Sentinel's tracing module by calling the module's `record_event` function. Afterward, the library handles the call as usual, discarding the message if it is lower than the pre-configured emission threshold. Note that discarding a message does not preclude Sentinel from including the message's data in its models — Sentinel uses all messages of all granularities regardless of whether the system is configured to write them to disk. This feature enables the system to output coarse-grained logs for auditing purposes while obtaining Sentinel's analysis over logs of all granularities.

Sentinel's tracing module obtains the filename and line number of the position in source code that issued the logging call[1]. It uses this information to uniquely identify each event type. Note that log messages originating from the same line in source code are therefore mapped to the same event. This is by design; log messages are often parameterized by variables but correspond to the same system event [12, 14]. Consequently, prior approaches that mined log *files* for behaviour and anomaly detection typically rely on detailed system-specific preprocessing scripts to map log messages to events [12, 14, 24]. By using the file name and line number to identify events, we avoid the overhead of such scripts while maintaining Sentinel's generalizabilty.

2.2 Event Tracking

After obtaining a logging library call's originating file name and line number, Sentinel uses this information to look up its corresponding event in an in-memory hash map called the *event table*. For example, in Figure 5, the event corresponding to the log message originating in file `bufmgr.c` on line 725 hashes to the second slot. Each event in the event table is associated with a filename, line number, event hit counter and pointers to transition and timing information. Each time `record_event()` is called for a particular event e, e's hit count is incremented and its transition and timing information is updated (described in Section 3). Importantly, each process and thread maintains their own event table as a thread-local data structure. Therefore, there is no contention when an event table is updated.

Periodically, or when the system shuts down, each thread writes its event table to disk in a per-thread file. To catch these shutdown events, developers register `atexit()` hooks or subclass their programming language's `thread` class to submit a `dump_im_tracing` call to Sentinel's tracing library. Before analysis, Sentinel's background *event aggregator* sums the counts for each event over all of these files to determine their overall frequency, and computes their proportion. Along with event transition and timing information, these event proportions are encoded into a *behaviour model* for comparison against those of other workloads and system configurations. Note that although Sentinel internally associates events with only their file names and line numbers, the original log message may be obtained for reports by looking up and reading the message at that event's location in source code. Moreover, Sentinel enables users to "tag" events with custom names to further improve the clarity of its reports.

Event Differences Report: Sentinel compares these behaviour models and reports differences in events in descending order of their proportional differences. Sentinel reports differences in *event proportions* rather than raw event counts. Comparing event counts directly leads to reporting differences in only the most frequently occurring events, while differences in event proportions differentiate contributions to overall system behaviour. The insights shown in Figures 1a and 1c are the top proportional event differences between Scenarios A and B for PostgreSQL and the TPC-W benchmark application clients. These insights are obtained in a fully domain-agnostic way; neither Sentinel's extraction nor ranking of event differences rely on any domain knowledge or system-specific characteristics.

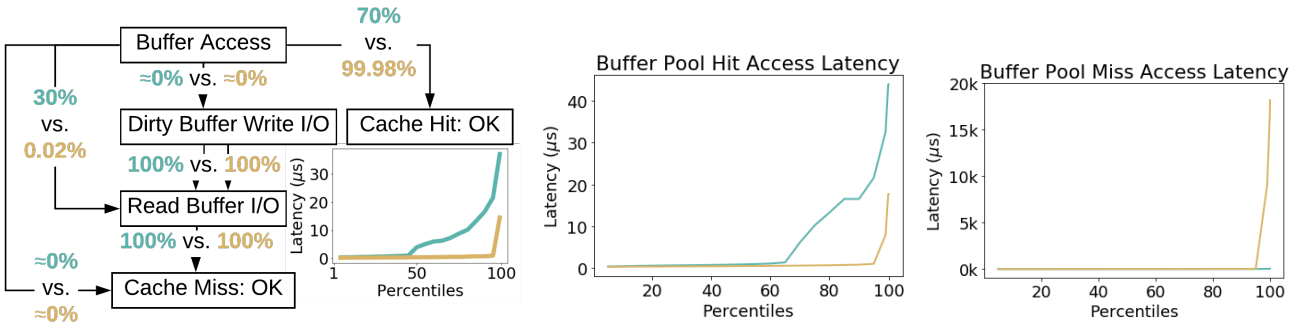
3. EVENT FLOWS AND TIMING

Although event proportion comparisons capture aggregate differences in behaviour, they do not describe event relationships. For example, counts of individual buffer page accesses, buffer pool misses, and dirty buffer writes alone do not express the control flow of operations that comprise PostgreSQL's buffer page access code. By encoding sequences of events into its event transition model, Sentinel *automatically* expresses the target application's control flow. Moreover, Sentinel combines its event transition models with detailed timing information to present performance breakdowns for functionality that spans multiple events.

We demonstrate these benefits using an extracted event transition model of PostgreSQL's buffer accesses for Scenarios A and B, shown in the inset of Figure 6a. The structure of the transition models for both scenarios is the same and reflects the execution path in code for page accesses. Although the buffer pool cache hit rate is relatively high for both scenarios, Scenario A's hit rate of 70% is much lower than Scenario B's 99% hit rate. These differences in execution control flow patterns result in significant differences in buffer access latency.

Beyond these transition probabilities, Sentinel computes empirical cumulative distribution functions (CDFs) of the time to transition between pairs of events. In contrast to the typical approach of instrumenting code to insert timers and computing average latencies, CDFs describe the full range of performance behaviour, including tail latencies, which are a key performance concern [19]. Furthermore, individual CDFs may be combined with event transition probabilities to estimate CDFs of the time spent in functionality that straddles multiple events and event transitions, as we describe below.

[1]For example, file names and line numbers may be obtained via the `__FILE__` and `__LINE__` macros in C/C++.



(a) Buffer Page Access Transition Graph **(b)** Buffer Page Cache Hit CDF **(c)** Buffer Page Cache Miss CDF

Figure 6: Buffer page access transition graph and latency CDFs for Scenario A (blue) and Scenario B (orange).

The combined CDF for buffer access latencies for the Scenarios is shown in Figure 6a. The overall buffer access times for both scenarios are similar until the higher percentiles, where the differences in cache misses manifest in higher latencies for Scenario A. As Figure 6b shows, the access latencies for buffer pool cache hits are far lower than tail buffer pool miss times (Figure 6c), which emphasizes the importance of maximizing cache hits. However, we see that buffer pool miss times are not uniformly high because many cache misses can be resolved by retrieving the page from the operating system's cache that is outside the confines of PostgreSQL's buffer pool. In the rare cases where this is not possible, access latencies increase considerably, as seen in the tail of the buffer pool miss CDF (Figure 6c). Given these CDFs, we can conclude that the buffer pool is too small for Scenario A, even though the operating system's cache can mitigate the cost of cache misses. Thus, this example demonstrates the importance of considering tail latencies and not merely averages; in the worst case, buffer access latencies can increase by orders of magnitude.

To support the above workflow, we next describe how Sentinel provides the following key features:

1. *Sentinel efficiently tracks transitions and the time to transition between pairs of events without significantly degrading performance.*
2. *Sentinel estimates and combines CDFs for individual transitions without excessively degrading their accuracy.*
3. *Sentinel intuitively and effectively ranks differences in transition probability and CDFs of transition time, outputting them in a behavioural differences report.*

3.1 Efficiently Tracking Event Transitions

During system execution, Sentinel stores transition count information in event tables. In addition to the functionality described in Section 2, `record_event()` also looks up the last executed event for the current thread, and increments a transition count from that event to the current event.

Each event in the event table has its own dynamically allocated list of counters. These lists contain one counter for each of the event's transitions (Figure 5). As most events transition to only a handful of others, using a dynamically allocated list of counters reduces memory consumption considerably. These per-thread transition counters are written to disk alongside the event counts. Sentinel's event aggregator uses these transition counters to compute the probability of transition from e_1 to e_2 by dividing the $e_1 \rightarrow e_2$ transition counter by the number of times it has observed e_1. As event transitions are stored in thread-local data structures and the total transition counts are computed in an offline analysis phase using the outputted files, Sentinel avoids introducing contention among threads.

Transition Differences Report: Sentinel compares event transition probabilities in behaviour models similarly to how it compares event probabilities. Concretely, Sentinel ranks differences in transition probability according to the ratio difference between them:

$$max\left(\frac{P(e_2|e_1)}{P'(e_2|e_1)}, \frac{P'(e_2|e_1)}{P(e_2|e_1)}\right)$$

where $P(e_2|e_1)$ and $P'(e_2|e_1)$ represent the probability to transition to event e_2 from event e_1 in the first and second behavioural models, respectively.

3.2 Estimating Transition Time CDFs

For each transition from an event e_1 to an event e_2, Sentinel computes an empirical CDF of the transition time. When Sentinel handles a `record_event()` call, it retrieves a nanosecond-precision timestamp from the operating system. It compares this timestamp to the timestamp of the last event to determine the transition time between events. Transition times are added to an array of previously recorded times for this transition, called a *reservoir*, and used to compute the CDF.

Sentinel uses *adaptable damped reservoir sampling* [1] to store transition time data for building CDFs. Rather than obtaining a timestamp for every event transition, Sentinel samples a subset of the event transition times. For each `record_event()` call, Sentinel samples the transition time for the current event e_1 to next event e_2 with probability $max(\frac{N}{k}, 1)$, where N is the number of samples we may store in the reservoir (*reservoir size*) and k is the number of times we have observed e_1 so far. Thus, as Sentinel observes more instances of e_1, k increases, which reduces the likelihood of sampling e_1 in the future. If there are already N elements in the reservoir, then one of these elements in the reservoir (chosen randomly) is replaced. By reducing the sampling probability as more samples of the transition are observed, Sentinel avoids excessive sampling overheads once it has obtained enough samples. Replacing an element at random enables Sentinel to keep older transition times around rather than just the most recent samples if a least-recently-used policy was employed instead. This strategy gives a more

complete picture of the CDF over the full duration of the experiment. The reservoirs for each thread that correspond to the $e_1 \rightarrow e_2$ transition are written to disk alongside their respective transition counters, and are combined to produce CDFs of transition times.

The size of the reservoirs (and thus the number of samples) determines the accuracy of the estimated CDFs, as well as the memory used for tracking. As the reservoir size increases, the CDF accuracy increases, as does memory consumption. Therefore, we chose reservoir size by accounting for theoretical guarantees on CDF accuracy and considering the associated memory usage trade-off. By modelling transition time latencies as exponential distributions $\exp(\lambda)$ for $\lambda > 0$ (motivated by prior work [25]) and choosing an error bound $\varepsilon > 0$ and probability of failure $\delta > 0$, we have the following theorem:

THEOREM 1. *Let* $\varepsilon, \delta > 0$. *After drawing* $4/\varepsilon^2 \log(\frac{1}{\delta})$ *samples from an unknown exponential distribution* $\exp(\lambda)$, *Sentinel outputs a probability distribution* $\hat{p}$ *such that with probability* $1-\delta$, $\text{dist}_{TV}(\hat{p}, \exp(\lambda)) \leq \varepsilon$, *where* $dist_{TV}$ *is the total variation distance.*

The proof is in Appendix A.

This result indicates that Sentinel's estimated CDFs approximate the true CDFs within a total variation distance of $\varepsilon = 0.1$ with more than 90% probability given a reservoir size of 1000. Using this reservoir size for each unique event transition that occurs in PostgreSQL in our experiments consumes a paltry 1.5 MB per thread.[2] Therefore, we use this reservoir size for all of our experiments. We also show the effects of reservoir size on empirical CDF accuracy in Section 5.6.

3.3 Combining Transition Time CDFs

Sentinel's empirical CDFs provide granular timing information about a *single* transition; often, system functions comprise multiple event transitions. For example, PostgreSQL's buffer page access functionality spans buffer pool cache hits, buffer pool misses, dirty page writes and disk I/O. To obtain a complete picture of time spent in buffer accesses, we must account for the likelihood of transitioning between all of these events *and* their corresponding transition time CDFs. Sentinel supports this functionality by enabling developers to combine transition CDFs together (Algorithm 1). Sentinel uses random walks to recreate the flow of program execution, sampling from the empirical CDFs and adding the sampled times together to form a sample for the overall CDF. These samples are then used to derive the overall empirical CDF.

We demonstrate how Sentinel combines CDFs by computing the total buffer page access latency CDF for a buffer pool miss as an example. In the buffer pool event transition graph (Figure 1b), we specify the start node for the random walk to be the start of a buffer page access and the terminal node to be a buffer pool miss completion event. In the first phase, Sentinel conducts a bounded depth-first search from the start node to end nodes to find paths that lead to the end node (Algorithm 1, line 1). As Sentinel conducts the depth first search (Algorithm 2), it keeps track of the probability of arriving at each node from the start node using the transition probabilities. If the probability reaches

[2]PostgreSQL emits 90 unique events and 150 unique transitions.

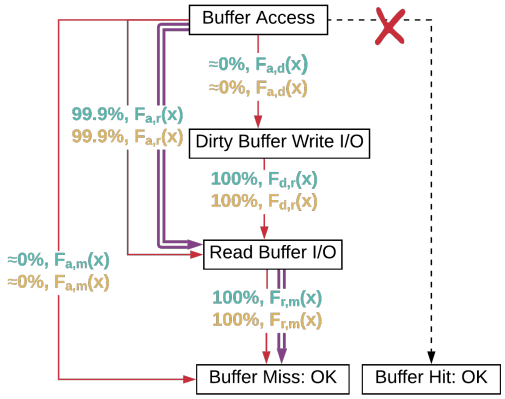


Figure 7: Combining empirical CDFs using random walks for Scenario A (blue) and Scenario B (orange) on a subset of the buffer access transition model. Crossed-out paths indicate those pruned from consideration. Double purple lines indicate the random walk example in the text. $F_{e_1,e_2}(x)$ are CDFs for percentile x of transition times from event e_1 to event e_2.

Algorithm 1 Combining CDFs in Sentinel

Require: (V,E) are vertices/edges in the transition graph, S is the start node, T is a set of terminal nodes, τ is the probability cut-off threshold, n is the number of random walks

1: $E' =$ `bounded_dfs(`S`,`T`,(`V`,`E`),1,`τ`)`
2: $E' =$ `renormalize_probs(`(V, E')`)`
3: $t = \emptyset$
4: **for** i = 0; i < n; i++ **do**
5: $\quad t = t \cup$ `random_walk`$(S, T, (V, E'))$
6: **end for**
7: **return** `convert_to_cdf(`t`)`

a lower-bound threshold (line 4), then Sentinel prunes the path from consideration. Paths that lead to the terminal nodes are recorded as acceptable choices during random walks and have their probabilities renormalized to account for the probabilities removed from pruned edges (Algorithm 1, line 2). After pruning the divergent paths from the transition graph for the buffer pool miss event, Sentinel constructs the renormalized transition graph in Figure 7.

In the second phase, Sentinel conducts random walks from the start node to terminal nodes (Algorithm 1, line 5). At each step, the next transition is chosen using the renormalized transition probability. For example, from the buffer page access start node in Figure 7, Sentinel takes the transition to the read buffer I/O node in both scenarios with nearly 100% probability. Sentinel computes the probability of taking the path at each step, and terminates the walk when it reaches minimum probability threshold or a terminal node. Without loss of generality, assume that Sentinel has taken the transition to the buffer read event and then to the buffer pool miss completion event for one of the random walks. Then at each step, Sentinel samples the CDF of each transition taken and adds the sampled value to a running total. Thus, Sentinel will sample the $F_{a,r}(x)$ distribution and the $F_{r,m}(x)$ distribution (Figure 7) and add those results together to get the cumulative transition time for the random walk. Sentinel conducts multiple random walks, us-

Algorithm 2 Bounded DFS

Require: N is the current node, T is a set of terminal nodes, (V,E) are vertices/edges in the transition graph, p is the probability of getting to N from the start node, τ is the probability cut-off threshold

```
E' = ∅
for (N, v2, p2) ∈ E do
  p_n = p × p2
  if p_n ≥ τ then
    if v2 ∈ T then
      E' = E' ∪ (N, v2, p2)
    else
      next_edges = bounded_dfs(v2, T, (V, E), p_n, τ)
      if next_edges ≠ ∅ then
        E' = E' ∪ (N, v2, p2) ∪ next_edges
      end if
    end if
  end if
end for
return E'
```

ing each of the returned cumulative times to estimate the overall CDF of transition times from the start to terminal events. The accuracy of the combined CDF is determined by the number of random walks and the accuracy of the underlying CDFs. Appendix A shows that a constructed CDF with maximum path length from a source node l has total variation of at most ε from the true CDF given that each CDF along the path has total variation of at most $\frac{\varepsilon}{l}$.

3.4 CDF Differences Report

It is often desirable to measure how event transition latencies have changed between workloads and system configurations. For example, a developer may wish to see how the lock wait time CDF has changed between a workload with low contention and a workload with high contention. Sentinel provides this functionality by reporting the largest CDF transition time differences.

We quantify the differences in CDFs p and q using the earth-mover's distance between them [27]. This distance metric provides an intuitive measure of the differences between CDFs as it quantifies the effort to "push probability mass" in p to make it "look like" q.

For each event transition present in both of the models Sentinel is comparing, Sentinel computes the earth-mover's distance between the transition's CDFs to obtain scores, and ranks CDF differences in order of decreasing scores. Thus, the transitions that differ the most will be presented first. As with differences in event proportions and transition probabilities, differences in transition time CDFs are computed and ranked in a fully domain-agnostic way.

4. THE SENTINEL SYSTEM

We implemented the in-memory tracing module, event aggregator and analysis library (Figure 3) described in Sections 2 and 3 as the Sentinel system.

The in-memory tracing module implements the event and transition tracking algorithms. The tracing module outputs its extracted event counts, transition counts, and timing reservoirs to disk, which are then processed offline by the event aggregator.

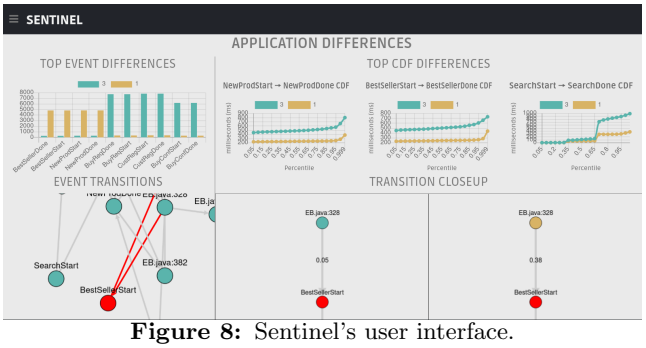


Figure 8: Sentinel's user interface.

The event aggregator reads the per-thread files outputted by the tracing module and combines them to form a cohesive model of overall system behaviour. These models are then loaded into a PostgreSQL database for later analysis.

Given two behavioural models that correspond to a baseline configuration and a situation of interest, the analysis library compares the models using the difference and ranking computations presented in Sections 2 and 3. It outputs the differences in event proportion, transition probability, and transition time, sorted by difference size. These models and differences may be consumed directly via an API, or through a web-based user interface that plots them visually.

Sentinel's web interface (Figure 8) presents intuitive visual rankings of event, transition, and transition time CDF differences. As behavioural models extracted from systems are often complex, the analysis library and UI present the largest differences in events and event transitions to highlight important subsets of the transition graph. In fact, the UI uses brushing and linking [16] so that interactions with one visualization affect what is shown in the others, thereby helping users to effectively explore behavioural differences. For example, when a user selects an event in the event frequency differences pane, the transitions pane will focus on the event and its neighbours in the transition graph. This feature enables users to quickly find the largest ranked differences, and then contextualize them within the transition graph. Users may pan and zoom within the transition graph, and then examine other panes in the UI that directly contrast the selected event and its associated transitions between the behavioural models. Furthermore, Sentinel's UI renders nodes in transition graphs using the CoSE layout, which places them near other nodes to which they are connected via a physics simulation [5]. This rendering results in clusters of nodes corresponding to different behavioural aspects (e.g., checkpointing, vacuuming, or buffer/query management). These aspects are illustrated by example in our demonstration video [9] and paper [10] both of which present these concepts in further detail.

5. EXPERIMENTAL EVALUATION

We now empirically demonstrate Sentinel's superiority in describing and differentiating system behaviour. We highlight the *cross-system* effectiveness of Sentinel by integrating with a typical 3-tiered system consisting of a database (PostgreSQL/SQLite), TPC-W benchmark clients, and Apache Tomcat (web server). We contrast Sentinel with state-of-the-art approaches for system behaviour analysis in terms of precision, performance overheads, analysis time, and ease of integration.

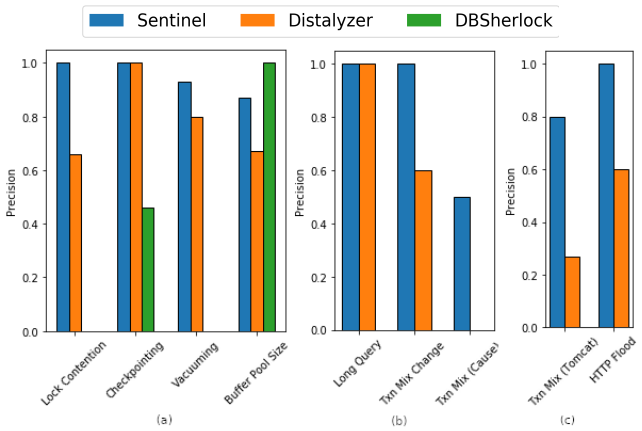


Figure 9: Precision graphs for Sentinel, Distalyzer, and DBSherlock's ability to pinpoint differences in (a) PostgreSQL, (b) TPC-W benchmark client, and (c) Apache Tomcat execution behaviour.

5.1 Experiment Setup

We demonstrate Sentinel's superiority at detecting and reporting system behaviour differences by comparing it against Distalyzer [24] and DBSherlock [37].

Distalyzer [24] is a log analysis tool that describes differences in log files that correspond to normal and anomalous system executions. Distalyzer describes statistically significant differences in log message counts, timestamps at which messages are emitted and logged variable values. Unlike Sentinel, Distalyzer requires that logs be written to disk and preprocessed offline using system-specific scripts before analysis. We needed domain knowledge of each system to develop scripts that extract important variables and latencies from the log files, which Sentinel does not require.

DBSherlock [37] is a state of the art database monitoring and anomaly detection tool. It extracts metrics from a database and the underlying OS and produces predicates (e.g., `dbCurLockWaits > 5`) to describe how a period of anomalous performance differs from the norm. We ported DBSherlock to other database systems by mapping MySQL metrics it relies on to corresponding statistics in the other databases (where available). As DBSherlock is designed for database systems, we only consider its ability to locate behavioural differences in PostgreSQL and SQLite.

The TPC-W [31] benchmark serves as our baseline workload, and we vary workload and database configuration parameters to introduce behavioural differences in benchmark clients and the database. The multiple system components in the benchmark (benchmark clients, database, web server) enable us to demonstrate Sentinel's effectiveness across system domains. By default, our experiments use PostgreSQL 9.6 as the database, though we also demonstrate Sentinel's generality within the database systems domain by differentiating SQLite 3.31.1 behaviour in Section 5.2.4.

Our benchmark machine is configured with 32 GB of RAM, 12 CPU cores with hyperthreading enabled, and a 800 GB HDD. Our experiments use 10 concurrent clients with no think time between transactions.

5.2 Behaviour Difference Validation

To assess each system's ability to surface relevant and useful behavioural insights, we studied controlled scenarios in which we adjust a single system characteristic (e.g., lock contention, query execution time, transaction mix) and determine whether each system reports the expected behaviour change corresponding to this difference. The nine scenarios we studied are shown in Table 1, along with the variations we use to test precision.

In each scenario, Sentinel, Distalyzer, and DBSherlock compare system behaviour between a baseline TPC-W workload and the same TPC-W workload but with the relevant change induced (test workload). Distalyzer is provided with the same level of logging information as Sentinel by configuring the database, benchmark clients, and Apache Tomcat to write all log messages to disk, which comes at the cost of significant performance degradation (Section 5.3). As Distalyzer uses populations of log files to derive its insights, each of its tests rely on log files obtained under three executions of the test configuration. Distalyzer therefore uses three times as much information as the other approaches and takes three times as long to gather it. We used Distalyzer's absolute total difference to rank its reported differences, as in [24]. For DBSherlock, we monitored metrics every second (as in [37]) for the baseline configuration and compare it to metrics recorded for the test configuration. We labelled the metrics recorded during the test configuration as anomalous and the metrics during the baseline configuration as normal. DBSherlock outputs predicates over these metrics that hold for the test configuration but do not hold for the baseline configuration. We rank these predicates according to DBSherlock's normalized difference threshold metric, which describes how different the underlying metric's values are in the baseline and test configurations.

To ensure an apples-to-apples comparison, we compared each analysis system's output from their APIs. If a Sentinel/Distalyzer log event or DBSherlock predicate is surfaced within the top 3 ranked differences in the relevant report category that is indicative of the change (e.g., change in lock wait events for lock contention), we consider the test successful. We repeated each test three times and consider five variations for each test case. We compute precision results for each tool by dividing the number of correct test cases in each scenario over the number of test cases (Figure 9). Next, we discuss each result in turn.

5.2.1 *PostgreSQL Behaviour*

We integrated Sentinel and Distalyzer with PostgreSQL 9.6 and enhanced its logging by configuring relevant built-in DTrace hooks to emit logging information. DBSherlock was provided with relevant database metrics stored by PostgreSQL, equivalent to what it obtained from its targeted database, MySQL, where available. We induced changes in PostgreSQL execution behaviour and evaluated Sentinel, Distalyzer, and DBSherlock's ability to detect these differences (Figure 9a). Unless otherwise stated, these experiments (and those in Sections 5.2.2 and 5.2.3) use the default PostgreSQL configuration with appropriate values for the buffer pool and operating system cache (8 GB and 16 GB, respectively). We used a 50 GB TPC-W database.

Lock Contention: We introduced additional lock contention in the BuyConfirm transaction by holding exclusive locks for longer. Sentinel reports differences in lock wait event proportion in *all* of the situations we considered, demonstrating its ability to surface relevant behavioural insights. Distalyzer identifies lock contention in a majority

Table 1: Scenarios used to evaluate Sentinel, Distalyzer and DBSherlock's ability to pinpoint behavioural differences.

Test Case	Description	Variations
Lock Contention	Increase lock hold time	**PostgreSQL/SQLite:** 10, 25, 50, 75, 100 ms
Buffer Pool Size	Decrease buffer pool size	**PostgreSQL:** 250MB, 500MB, 1GB, 2GB, 4GB **SQLite:** 50KB,100KB,1MB,100MB,1GB
Aggressive Vacuuming	Increase frequency of vacuuming	**PostgreSQL:** 50, 40, 30, 20, 10 s **SQLite:** 5, 10, 20, 40, 80 txns
Aggressive Checkpointing	Increase frequency of checkpointing	**PostgreSQL:** 90, 75, 60, 45, 30 s **SQLite:** 500, 2.5k, 5k, 10k, 20k WAL frames
Long Running Query	Decrease BestSeller transaction selectivity	5x, 10x, 15x, 20x, 25x more tuples accessed
Txn Mix Change	Change likelihood of executing BestSellers after Homepage transaction	-10%, -20%, +10%, +20%, +30% less/more likely
Txn Mix Change (Cause)	As above, but find transition probability difference	-10%, -20%, +10%, +20%, +30% less/more likely
Txn Mix Change (Tomcat)	As above, but using only web server logging	-10%, -20%, +10%, +20%, +30% less/more likely
HTTP Flood	Rapidly issue GET requests on new HTTP connections	0, 0.001, 0.01, 0.1, 1 s think time between requests

of cases, but is susceptible to reporting differences in irrelevant events that happen at different times in test cases (e.g., checkpointing, autovacuum). DBSherlock performs poorly at detecting lock contention on PostgreSQL because PostgreSQL 9.6 does not keep a running tally of lock conflicts. We approximated this statistic by polling how many queries are blocked on locks, but this approximation is not enough to capture all lock conflicts.

Aggressive Checkpointing: We decreased the checkpoint interval from the default (5 minutes) and compared system behaviour to the default configuration. Both Sentinel and Distalyzer correctly identify differences in checkpoint events in all test cases. Distalyzer performs comparably with Sentinel because checkpoint occurrence rates have changed significantly, and Distalyzer ranks changes in occurrence time highly. DBSherlock does not extract checkpoint counts by default, and therefore does not capture these differences. We accommodated an increase in page flush metrics as well for DBSherlock, but these predicates are also infrequently reported compared to predicates over values of unrelated metrics that have changed.

Aggressive Vacuuming: In this scenario, we compare the default autovacuum interval (1 minute) with decreased autovacuum intervals. Both Sentinel and Distalyzer are effective at pinpointing differences in autovacuum events, though Sentinel's precision remains higher. Again, Distalyzer's susceptibility to event occurrence timings affects its precision, an issue from which Sentinel does not suffer. By contrast, DBSherlock does not report these differences as it does not capture metrics related to vacuum behaviour.

Improperly-Sized Buffer Pool: We decreased PostgreSQL's allocated buffer pool size to induce buffer pool cache misses. Sentinel accurately detects differences in buffer cache misses for all configurations where the buffer pool size is less than 4 GB. When comparing system behaviour with a 4 GB buffer pool to an 8 GB buffer pool, the change in buffer misses is not significant enough to be outputted. Similarly, Distalyzer accurately reports cache miss effects for small buffer pool sizes, but remains less accurate than Sentinel. DBSherlock pinpoints differences in buffer pool size as its top reported difference in each experiment because it extracts buffer pool size as a metric. As this metric is constant for the baseline configuration and constant for the test configuration, but these constant values differ, the predicate `dbTotalPagesMB <` X for a configured buffer pool size X perfectly partitions the data observed in the baseline configuration from that of the X configuration, and is thus highly ranked.

From these results, we observe that Sentinel is highly accurate at pinpointing relevant behavioural changes compared to the other approaches. Unlike Sentinel, Distalyzer's precision is hindered because its ranking scores are heavily affected by *when* events occur. DBSherlock's accuracy suffers because its ranking prioritizes predicates that hold for values of metrics in the test configuration but not in the baseline configuration. Sentinel avoids this pitfall as it pinpoints the largest differences in behaviour between the configurations and not predicates that separate values of metrics in one configuration from another.

5.2.2 *Client Application Behaviour*

We now consider changes in application behaviour and evaluate Sentinel and Distalyzer's ability to highlight these differences (Figure 9b). We do not evaluate DBSherlock on benchmark client or web server behaviour (Section 5.2.3) due to its specialization for databases and reliance on a priori knowledge of which system metrics to extract. By contrast, both Sentinel and Distalyzer use information that is available via logging calls.

Long Running Query: We varied the selectivity of the BestSeller transaction to increase the transaction's execution time. In all cases, Sentinel and Distalyzer correctly highlight the BestSeller transaction as exhibiting a large latency change. Sentinel naturally captures this difference as part of its transition time CDF rankings. By contrast, Distalyzer detects this transaction's latency differences because we explicitly extracted each transaction's latency from TPC-W client execution logs as part of the client log preprocessing script we developed for Distalyzer. This result highlights the need for domain knowledge when configuring Distalyzer for each system.

Transaction Mix Change: We modified the probability of executing the BestSeller transaction after the Home page transaction. Decreasing this probability results in an increased rate of executing NewProducts transactions, while increasing it results in more BestSeller transactions. Sentinel correctly detects these transaction mix changes in application behaviour in all cases, which is captured by differences in client event logging about which web pages they will access. Distalyzer correctly detects them in only 60%

of cases. In cases where Distalyzer is incorrect, it is due to the importance it places on event timing differences and because it also highly ranks event differences correlated with the changes in the transaction mix.

Transaction Mix Change (Cause): For the previous scenario, we also assessed whether each system could find the root cause — i.e., the change in transition probability from the home page. *Only Sentinel* detects changes in transition probabilities between log events. For the larger transition probability modifications, Sentinel is highly accurate. When Sentinel is incorrect, it highlights transition differences from *rarely* occurring events; as their transition counts are low, they are more susceptible to variation.

5.2.3 Web Server Behaviour

To demonstrate Sentinel's generalizability to a wide range of systems, we also integrated Sentinel and Distalyzer with Apache Tomcat version 9.0.3, a popular open-source servlet container and web server. As in the other environments, we developed a custom preprocessing script to enable Distalyzer to extract events and timing information from Tomcat's log files. Precision results are shown in Figure 9c.

Transaction Mix Change (Tomcat): For the transaction mix change experiment above, we further evaluated whether we could determine this change in access patterns using *only* models of the web server's behaviour. Sentinel is highly accurate at pinpointing the behavioural change, which is emitted from per-transaction servlet logging. Distalyzer's precision again suffers due to its susceptibility to event timing differences. As Tomcat emits 2.5× more event types than PostgreSQL, this result not only demonstrates Sentinel's generalizability across systems but also its resilience to system complexity.

HTTP Flood: We simulated an HTTP flood attack [30] by rapidly issuing HTTP GET requests to Tomcat while running the TPC-W browsing mix. We tested the system's sensitivity to reporting these events by inserting varying think times between each GET request. Sentinel captures this behaviour by highlighting differences in request type proportion in *every* variation of this test (repeated accesses to the same page), while Distalyzer captures these differences in only 60% of cases. In cases where Distalyzer does not pinpoint the correct behavioural difference, it highlights differences correlated with the attack (e.g., session management) or changes in event timing. Through these results, we observe that Sentinel's techniques are applicable to security-focused behavioural exploration on data systems as well.

5.2.4 SQLite Behaviour

We further demonstrate Sentinel's generality by integrating it with SQLite, a popular embedded SQL database (Figure 10a). To do so, we enabled its debug logging statements by adjusting compiler flags, and configured it to use Sentinel's in-memory tracing library instead of writing logs to the console. For Distalyzer, we configured SQLite to emit these logs to disk and developed a preprocessing script to extract relevant features from them. We provided metrics for DBSherlock by using SQLite's `sqlite3_(db)status` functions. As these metrics do not cover all the test case functionality, we advantaged DBSherlock by providing extra information obtained by outputting and preprocessing only the relevant SQLite log messages (e.g., checkpoints). We configured SQLite to use a write ahead log and used the default configuration unless otherwise mentioned. As SQLite is an embedded database, we used a 1 GB TPC-W database.

Lock Contention: As SQLite processes update transactions serially, i.e., one at a time, we replicated the Lock Contention PostgreSQL test by creating a connection pool of database connections for concurrent readers and a single database writer connection. We added logging statements to the connection pool code and made this information available to each analysis system. Sentinel and Distalyzer obtain high accuracy on this test as obtaining the writer connection is the main bottleneck and is therefore highly reported in Sentinel's transition time CDFs and Distalyzer's state variables. DBSherlock is not effective in this scenario because the lock wait time metric does not separate one scenario's behaviour from the other.

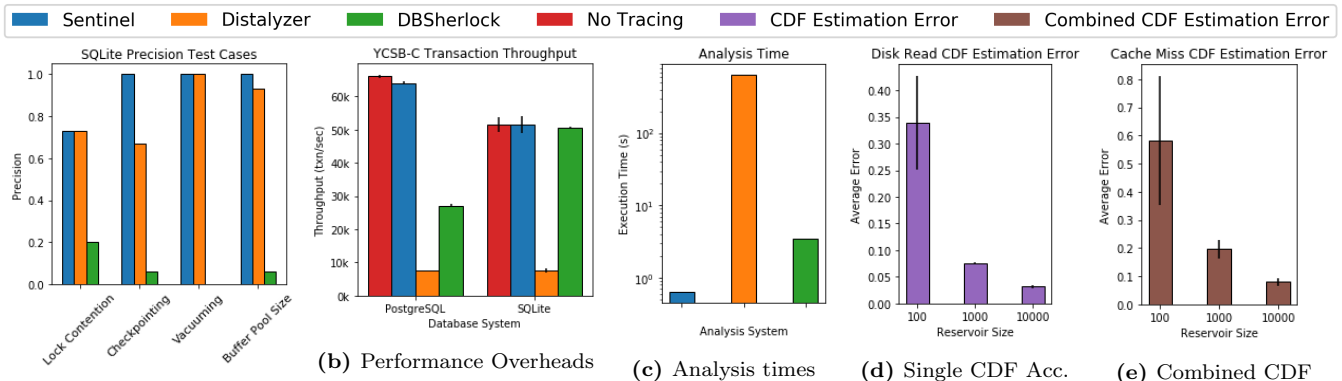
Aggressive Checkpointing: SQLite conducts checkpoints every N frames, in contrast to PostgreSQL's method of every N seconds, so we adjusted our test case values to accommodate this difference (Table 1, default 1000 frames). Sentinel obtains 100% precision on this test case because small changes in checkpoint frequency greatly affect their overall event count proportion, which Sentinel detects. Although Distalyzer is not as accurate as Sentinel, its support for time-based and frequency-based differences enable it to frequently report differences in checkpointing as well. DBSherlock is not effective in this scenario because, as before, its ranking prioritizes unrelated metric differences.

Aggressive Vacuuming: Vacuuming in SQLite is triggered by issuing a `PRAGMA incremental_vacuum` command. Therefore, we configured the write connection to submit this command after every k^{th} committed transaction for varying values of k (Table 1, default every transaction, as in SQLite's full vacuum mode). Both Sentinel and Distalyzer obtain perfect precision for this test for the same reasons they performed well on the Aggressive Checkpointing test, reporting differences in vacuum events. As before, DBSherlock is not effective on this test case.

Improperly-Sized Buffer Pool: Unlike PostgreSQL, update transactions in SQLite invalidate the cache of other concurrent connections, making large buffer pool sizes less effective. We accommodated this behaviour by reducing the buffer pool sizes we used in our tests compared to the values we used for PostgreSQL (Table 1, default 10 MB). Sentinel reports larger numbers of cache misses and page fetches in all cases when the buffer pool size changes, and Distalyzer reports similar characteristics for most of the experiments in this test case. DBSherlock is not effective on this test case because SQLite's memory consumption grows to meet the buffer pool size. Other metrics are therefore prioritized by DBSherlock's ranking.

Distalyzer's accuracy is improved on these test cases compared to their counterparts in PostgreSQL because we disabled time-based events (e.g., checkpoints or vacuuming) if they are not the focus of the test case. Therefore, it is less vulnerable to over-emphasizing the importance of these events. Despite these advantages for its competitors, Sentinel retains its superiority in highlighting behavioural differences on SQLite as well. Furthermore, as our subsequent results show, Sentinel has much lower overhead on both PostgreSQL and SQLite than Distalyzer.

5.3 Monitoring Overheads



(a) SQLite Precision Results **(b)** Performance Overheads **(c)** Analysis times (PostgreSQL) **(d)** Single CDF Acc. **(e)** Combined CDF Acc.

Figure 10: SQLite precision results, YCSB-C throughput, system analysis times, and CDF estimation errors.

To assess the performance overheads of Sentinel, Distalyzer and DBSherlock, we executed the YCSB-C workload against PostgreSQL and SQLite for 5 minutes using OLTP-Bench [4] and measured the throughput. We used YCSB-C as it is a read-only workload with simple operations, and therefore moves the bottlenecks from transaction processing and lock contention to debug logging and monitoring. We computed the average throughput and their 95% confidence intervals, which are shown in Figure 10b.

We observe that Sentinel's monitoring reduces transaction throughput of PostgreSQL and SQLite by less than 3%, while Distalyzer imposes large performance penalties of 90% and 85% respectively because it requires detailed and costly debug logging to disk. Sentinel's improves throughput over DBSherlock by 60% on PostgreSQL because Sentinel does not force logging to disk. On the SQLite database, the throughput is similar because DBSherlock's SQLite configuration does not log as heavily as its PostgreSQL configuration — it relies largely on `sqlite3_(db)status` metrics. These results demonstrate that Sentinel obtains the highest precision with the lowest overhead.

To further understand Sentinel's overheads, we conducted an ablation study in which we measured the throughput of the database systems while Sentinel tracked (*i*) only event counts, (*ii*) events and transition counts, and (*iii*) all tracking enabled. We observed that event count tracking resulted in only 1.5% of the overhead, enabling transition count tracking incurred an additional scant overhead of 0.5%, and enabling the remaining tracking functionality in Sentinel added only 1% overhead.

5.4 Analysis Time

We now contrast Sentinel, Distalyzer, and DBSherlock in terms of the time they take to analyze results from two workloads or system configurations. We average the time it takes to compare PostgreSQL behaviour for the 10 ms variation of the lock contention scenario to the baseline configuration (Section 5.2). Results are shown in Figure 10c.

Distalyzer's analysis time far exceeds that of DBSherlock and Sentinel due to the large size of its preprocessed log files for each test (≈ 4 GB). Both Sentinel's and DBSherlock's analysis phases use summaries of system execution behaviour to determine behavioural differences, which requires much less I/O and computation time. However, Sentinel's analysis phase takes only 1/5 the time of DBSherlock's, a testament to Sentinel's efficiency while yielding useful results (Section 5.2). This low analysis time enables administrators to rapidly identify and respond to system changes.

Table 2: Analysis of system integration efforts.

	Sentinel	Distalyzer	DBSherlock
Per-System Preprocessing	-	✓	✓
Reqd Domain Knowledge	-	✓	✓
Integrate w/ Log Lib	✓	-	-
PostgreSQL LoC changed	53	156	159
SQLite LoC changed	74	317	194
TPC-W LoC changed	13	61	N/A
Tomcat LoC changed	25	189	N/A

5.5 System Integration Efforts

We summarize our experience integrating Sentinel, Distalyzer and DBSherlock with PostgreSQL, SQLite, TPC-W benchmark clients, and the Apache Tomcat web server in Table 2. Integrating Sentinel with these data systems requires less effort than the other approaches, which we quantify using the lines of code (LoC) changed during integration.

Both Distalyzer and DBSherlock require system-specific preprocessing scripts, while Sentinel does not. Distalyzer's preprocessing scripts process log files into a format containing event variables, state variables and relevant latencies. Due to the complexity of parsing a large variety of log messages and coercing them into the correct format, these scripts often require many lines of code to implement (Table 2). Similarly, DBSherlock's preprocessing scripts take the form of customized `dstat` plugins or targeted modifications to an SQLite JDBC driver to obtain status metrics. These scripts rely on *a priori* knowledge to obtain the salient metrics, transforming them and combining them together for later analysis. In contrast to these approaches, Sentinel requires instrumenting only the logging library and wrapping thread logic to output Sentinel's data before terminating, thus requiring fewer code changes to integrate.

Note that the architecture of a system influences the complexity of integrating it with an analysis system. For exam-

ple, integrating Sentinel with PostgreSQL requires less effort than SQLite because PostgreSQL uses a centralized logging library (elog) while SQLite uses compiler-enabled `printf` statements. Furthermore, SQLite's embedded nature necessitates modifying the JDBC driver, which is not necessary in PostgreSQL. As these aspects also increase the complexity of integrating with Distalyzer and DBSherlock, Sentinel's integration efforts remain the lowest in all cases.

5.6 Accuracy of Sampled CDFs

We assessed Sentinel's accuracy in estimating individual CDFs and those it generates using its random walk technique. First, we determined the CDF accuracy for a single transition using three different reservoir sizes (Figure 10d). We measured the deviation of the estimated CDF from the true CDF at every 5^{th} percentile up to the 95^{th} percentile and averaged them. As expected, increasing the reservoir size reduces the estimation error, but also increases the memory consumption. With a reservoir size of 100, we observed a high estimation error of 35%, while increasing the size to 1000 reduces the error to below 10%. Although increasing reservoir size further does slightly improve accuracy, it consumes 10× more memory.

We next considered the accuracy of CDFs constructed via random walks by computing buffer miss latency CDFs (recall Figure 7) and comparing them to the true CDF (Figure 10e). As above, we consider various reservoir sizes and averaged the errors at every 5^{th} percentile after 1 million random walks. Again, we observed that the error decreases significantly from 60% to 20% when increasing reservoir size from 100 to 1000 and that 10000 samples further improves it further to 8%. Given these accuracy and memory trade-offs, Sentinel uses reservoirs of size 1000 in our experiments.

These results complement our theoretical results (Section 3.2) as the theory bounds total variation in probability while the empirical results measure differences in latency at given percentiles. Combined, these results show that Sentinel's CDF estimation techniques are very effective.

6. DISCUSSION AND FUTURE WORK

Sentinel's insights assist administrators and developers in addressing system performance concerns and in understanding system behaviour. Although Sentinel's reports obviate the need for users to sift through vast log files and thousands of metrics to identify behavioural differences, some open issues remain. For example, Sentinel's reports enable an administrator to understand and address performance problems rather than directly remedying them. Furthermore, although past work has focused on placements of log messages [7] and debug logging is widely used, Sentinel cannot surface behavioural differences that are not captured by log messages. While Sentinel has been shown to be effective for a wide range of systems and workloads in this paper, capturing such information and unifying it with Sentinel's transition model is interesting future work.

7. RELATED WORK

Growing application and system complexity has led to increased research effort towards system analysis tools.

Distalyzer [24] compares system debug logs using statistical tests. These debug logs must be processed using system-specific preprocessing scripts before analysis. Similarly, DBSeer [23, 36] and its module DBSherlock [37] provide insight into database behaviour by relying on system metrics and OS details. PerfXplain [17] relies on similar parameters to explain why two MapReduce jobs have different performance characteristics, and Oracle 10g uses carefully placed timers to diagnose performance bottlenecks in Oracle's database [3]. Apollo [15] instruments DBMS code to locate the source code responsible for performance regressions. Other work associates log messages with higher-level system requests using static analysis [42], modeling correct execution of requests [38], or by relying on built-in request identifiers [2] to find failures and performance problems. Sentinel differentiates itself from these approaches by generalizing across systems, obviating the need for system-specific preprocessing, and reducing performance overheads by integrating directly with logging libraries.

Jiang et al. [13] describe how user-provided rules may be used to abstract messages in log files to events and transitions between them. These models are used to determine which log messages are anomalous using Z-tests [14]. Similar ideas have been used to locate memory consumption problems by correlating system counters to log messages [29]. Sentinel differentiates itself from these works through its in-memory tracing library, which avoids logging to disk and the associated overheads, its difference ranking techniques, and its CDF estimation capabilities.

Self-driving and autonomous systems model client workloads to adaptively process requests [8, 20, 26, 28]. These approaches are targeted towards modeling components of system behaviour (e.g., storage accesses) rather than constructing comprehensive behaviour models using log calls.

Pensieve [39] uses system logs to locate and reproduce failures in a distributed system by developing a minimal chain of events that are responsible for the failure. By contrast, Sentinel concisely presents how system behaviour differs under varying system configurations and workloads.

Zhao et al. [40] describe where to place logging calls in system code to obtain information without exceeding a given performance overhead threshold. These techniques are orthogonal to Sentinel, but can be combined to increase the data available to understand the workload while leveraging Sentinel's low-overhead tracing framework.

8. CONCLUSION

This paper presented Sentinel, an analysis tool that uncovers behavioural differences in systems using only built-in debug logging. Sentinel's in-memory tracing model seamlessly integrates with popular logging frameworks and builds models of system behaviour, thereby avoiding the traditional overheads of emitting detailed debug logs to disk. By comparing these extracted models, Sentinel highlights important and useful differences in system behaviour, as evidenced by our experiments using benchmark workloads.

ACKNOWLEGEMENTS

Funding for this project was provided by the Natural Sciences and Engineering Research Council of Canada, the Canada Foundation for Innovation, the Ontario Research Fund and the University of Waterloo-Huawei Joint Innovation Lab.

9. REFERENCES

[1] P. Bailis, E. Gan, S. Madden, D. Narayanan, K. Rong, and S. Suri. MacroBase: Prioritizing Attention in Fast Data. *Proceedings of the 2017 international conference on Management of Data - SIGMOD '17*, pages 541–556, 2016.

[2] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 217–231, Broomfield, CO, Oct. 2014. USENIX Association.

[3] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood. Automatic Performance Diagnosis and Tuning in Oracle. Automatic Performance Diagnosis and Tuning in Oracle. *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 84–94, 2005.

[4] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, Dec. 2013.

[5] U. Dogrusoz, E. Giral, A. Cetintas, A. Civril, and E. Demir. A layout algorithm for undirected compound graphs. *Inf. Sci.*, 179(7):980–994, Mar. 2009.

[6] A. S. Foundation. Apache log4j 2. `https://logging.apache.org/log4j/2.x/`, 2020.

[7] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, page 24–33, New York, NY, USA, 2014. Association for Computing Machinery.

[8] B. Glasbergen, M. Abebe, K. Daudjee, S. Foggo, and A. Pacaci. Apollo: Learning query correlations for predictive caching in geo-distributed systems. *EDBT*, 2018.

[9] B. Glasbergen, M. Abebe, K. Daudjee, D. Vogel, and J. Zhao. Sentinel: Understanding data systems. `https://www.youtube.com/watch?v=mJIv1wX4Dak`.

[10] B. Glasbergen, M. Abebe, K. Daudjee, D. Vogel, and J. Zhao. Sentinel: Understanding data systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2729–2732, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Google. Google logging module. `https://github.com/google/glog`, 2020.

[12] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. An automated approach for abstracting execution logs to execution events. *J. Softw. Maint. Evol.*, 20(4):249–267, July 2008.

[13] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. An automated approach for abstracting execution logs to execution events. *J. Softw. Maint. Evol.*, 20(4):249–267, July 2008.

[14] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *2009 IEEE International Conference on Software Maintenance*, pages 125–134, Sept 2009.

[15] J. Jung, H. Hu, J. Arulraj, T. Kim, and W. Kang. Apollo: Automatic detection and diagnosis of performance regressions in database systems. *PVLDB*, 13(1):57–70, Sept. 2019.

[16] D. A. Keim. Information visualization and visual data mining. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):1–8, Jan 2002.

[17] N. Khoussainova, M. Balazinska, and D. Suciu. Perfxplain: Debugging mapreduce job performance. *PVLDB*, 5(7):598–609, Mar. 2012.

[18] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[19] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

[20] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 631–645, New York, NY, USA, 2018. ACM.

[21] A. Mahgoub, P. Wood, S. Ganesh, S. Mitra, W. Gerlach, T. Harrison, F. Meyer, A. Grama, S. Bagchi, and S. Chaterji. Rafiki: A middleware for parameter tuning of nosql datastores for dynamic metagenomics workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, pages 28–40, New York, NY, USA, 2017. ACM.

[22] G. Melman. Spdlog: Fast c++ logging library. `https://github.com/gabime/spdlog`, 2020.

[23] B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and resource modeling in highly-concurrent oltp workloads. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 301–312, New York, NY, USA, 2013. ACM.

[24] K. Nagaraj, C. Killian, and J. Neville. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. *Nsdi*, pages 353–366, 2012.

[25] M. Nguyen, Z. Li, F. Duan, H. Che, Y. Lei, and H. Jiang. The tail at scale: How to predict it? In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, pages 120–125, Berkeley, CA, USA, 2016. USENIX Association.

[26] A. Pavlo, E. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel OLTP systems. *PVLDB*, 5(2):85–96, oct 2011.

[27] O. Pele and M. Werman. Fast and robust earth mover's distances. In *2009 IEEE 12th International Conference on Computer Vision*, pages 460–467. IEEE, September 2009.

[28] G. Soundararajan, D. Lupei, S. Ghanbari, A. D. Popescu, J. Chen, and C. Amza. Dynamic resource allocation for database servers running on virtual storage. In *Proccedings of the 7th Conference on File*

and Storage Technologies, FAST '09, pages 71–84, Berkeley, CA, USA, 2009. USENIX Association.

[29] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora. Leveraging performance counters and execution logs to diagnose memory-related performance issues. *IEEE International Conference on Software Maintenance, ICSM*, pages 110–119, 2013.

[30] C. C. Team. Capec 488: Http flood. `https://capec.mitre.org/data/definitions/488.html`, February 2020.

[31] TPC. Tpc benchmark w (web commerce). `http://www.tpc.org/tpcw`, 2000.

[32] A. B. Tsybakov. *Introduction to nonparametric estimation*. Springer series in statistics. Springer, New York, 2008.

[33] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1009–1024, New York, NY, USA, 2017. ACM.

[34] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM.

[35] S. Yang, S. J. Park, and J. Ousterhout. NanoLog: A Nanosecond Scale Logging System. *Proceedings of the 2018 USENIX Annual Technical Conference*, pages 335–350, 2018.

[36] D. Y. Yoon, B. Mozafari, and D. P. Brown. Dbseer: Pain-free database administration through workload intelligence. *PVLDB*, 8(12):2036–2039, Aug. 2015.

[37] D. Y. Yoon, N. Niu, and B. Mozafari. Dbsherlock: A performance diagnostic tool for transactional databases. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1599–1614, New York, NY, USA, 2016. ACM.

[38] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 489–502, New York, NY, USA, 2016. Association for Computing Machinery.

[39] Y. Zhang, S. Makarov, X. Ren, D. Lion, and D. Yuan. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 19–33, New York, NY, USA, 2017. ACM.

[40] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 565–581, New York, NY, USA, 2017. ACM.

[41] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm. Non-intrusive Performance Profiling for Entire Software Stacks based on the Flow Reconstruction Principle. *Osdi'16*, pages 603–618, 2016.

[42] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 629–644, Broomfield, CO, Oct. 2014. USENIX Association.

APPENDIX

A. THEORETICAL GUARANTEES

Inspired by previous work [25] and validated by empirical observation, we consider the distribution of latencies for each event transition to follow an exponential distribution $\exp(\lambda)$ for some $\lambda > 0$. We first consider accuracy guarantees on estimated CDFs for a single transition, afterward discussing accuracy guarantees on CDFs constructed via random walks.

Let $\mathbf{X}_1, \ldots, \mathbf{X}_n$ be n independent and identically distributed samples from $\exp(\lambda)$. Define the *empirical estimator* for the mean $\mu = 1/\lambda$ as $\hat{\boldsymbol{\mu}} \stackrel{\text{def}}{=} \frac{1}{n}\sum_{i\in[n]} \mathbf{X}_i$.

LEMMA 1. *Fix $\varepsilon > 0$, and let $n \stackrel{\text{def}}{=} \frac{4}{\varepsilon^2}$. Then, with probability at least $3/4$ it holds that*

$$(1-\varepsilon)\mu \leq \hat{\boldsymbol{\mu}} \leq (1+\varepsilon)\mu.$$

PROOF. For each $i \in [n]$ we have $\mathbf{E}[\mathbf{X}_i] = \mu$ and $\mathbf{Var}[\mathbf{X}_i] = \mu^2$ as properties of $\exp(\lambda)$, thus $\mathbf{E}[\hat{\mathbf{X}}_i] = \mu$ and $\mathbf{Var}[\hat{\mathbf{X}}_i] = \mu^2$.

By Chebyshev's inequality:

$$\mathbf{Pr}\left[|\hat{\boldsymbol{\mu}} - \mu| \leq 2 \cdot \sqrt{\frac{\varepsilon^2\mu^2}{4}}\right] \leq 1/4,$$

which concludes the proof. □

Now, define the total variation distance for two probability distributions p and q over the same support Ω to be

$$\text{dist}_{TV}(p,q) \stackrel{\text{def}}{=} \sup_{x\in\Omega} |p(x) - q(x)|$$

The following shows how accurate our estimate $\hat{\mu}$ must be to approximate $\exp(\lambda)$ within ε total variation distance.

LEMMA 2. *In order to learn the distribution $\exp(\lambda)$ up to error ε in total variation distance, it suffices to approximate μ within a multiplicative factor of $(1+2\varepsilon)$.*

PROOF. By Pinsker's inequality [32]:

$$\text{dist}_{TV}(\exp(\lambda), \exp(\lambda'))^2 \leq \frac{1}{2} \cdot \text{dist}_{KL}(\exp(\lambda) || \exp(\lambda'))$$

where dist_{KL} is the Kullback–Leibler divergence [18]. In addition, assuming an estimate μ' such that

$$\mu/(1+2\varepsilon) \leq \mu' \leq (1+2\varepsilon)\mu,$$

we get that

$$\lambda/(1+2\varepsilon) \leq \lambda' \leq (1+2\varepsilon)\lambda.$$

By letting $k = 1 + \frac{\lambda'}{\lambda}$ we have that

$$\text{dist}_{KL}(\exp(\lambda) || \exp(\lambda')) \leq -\log(\lambda'/\lambda) + \lambda'/\lambda - 1$$

$$k - \log(1+k) \leq k^2/2 = 2\epsilon^2.$$

Therefore,

$$\text{dist}_{TV}(\exp(\lambda), \exp(\lambda')) \leq \sqrt{\frac{1}{2} \cdot 2\epsilon^2} \leq \varepsilon.$$

□

By combining the above lemmas, we show that with repeated experiments we can guarantee at most ε error with $1-\delta$ probability, for any $\delta > 0$.

THEOREM 2. *Let $\epsilon, \delta > 0$. There exists an algorithm that draws $4/\varepsilon^2 \log(\frac{1}{\delta})$ samples from an unknown exponential distribution $\exp(\lambda)$, such that with probability at least $1-\delta$ outputs a probability distribution $\hat{p}$ such that $\text{dist}_{TV}(\hat{p}, \exp(\lambda)) \leq \varepsilon$.*

The proof follows by combining Lemma 1 and Lemma 2 and repeating the experiment $\log(1/\delta)$ many times and taking the median $\hat{\mu}$ value.

Note that by construction, the total variation error of a constructed distribution obtained via bounded random walks for maximum path length of at most l is also bounded by ε provided that each estimated distribution along the path has total variation of at most $\frac{\varepsilon}{l}$.