Author manuscript, published in Proceedings of the ACM SIGMOD Conference (Demo), 2020.
DOI: 10.1145/3318464.3384691

Sentinel: Understanding Data Systems

Brad Glasbergen, Michael Abebe, Khuzaima Daudjee, Daniel Vogel, Jian Zhao
{bjglasbe,mtabebe,kdaudjee,dvogel,jianzhao}@uwaterloo.ca
Cheriton School of Computer Science, University of Waterloo

ABSTRACT

The complexity of modern data systems and applications greatly increases the challenge in understanding system behaviour and diagnosing performance problems. When these problems arise, system administrators are left with the difficult task of remedying them by relying on large debug log files, vast numbers of metrics, and system-specific tooling. We demonstrate the Sentinel system, which enables administrators to analyze systems and applications by building models of system execution and comparing them to derive key differences in behaviour. The resulting analyses are then presented as system reports to administrators and developers in an intuitive fashion. Users of Sentinel can locate, identify and take steps to resolve the reported performance issues. As Sentinel's models are constructed online by intercepting debug logging library calls, Sentinel's functionality incurs little overhead and works with all systems that use standard debug logging libraries.

CCS CONCEPTS

• **Information systems → Database administration**; **Database utilities and tools**.

KEYWORDS

performance diagnosis; system behaviour; debug logging

ACM Reference Format:
Brad Glasbergen, Michael Abebe, Khuzaima Daudjee, Daniel Vogel, Jian Zhao. 2020. Sentinel: Understanding Data Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3318464.3384691

1 INTRODUCTION

Modern data management systems are expected to process a wide range of workloads, which has resulted in considerable system complexity [10, 15]. To diagnose performance problems, administrators rely on system-specific metrics and debug logs. Unfortunately, the large size of these debug log files and the sheer number of recorded metrics in these systems [16] greatly increase the difficulty of debugging; administrators spend hours reading system logs, understanding them, and acting on insights gleaned from their contents. Further complicating matters, the cause of a performance degradation may be due to characteristics of the workload (e.g. I/O overheads or poorly-written queries), or due to issues in the data system's configuration (e.g. inadequately sized buffer pool). Therefore, the administrator must analyze both the application and the data system to locate the source of the problem and remedy it.

Although debug logging is ubiquitous in both data systems and applications [17], emitting logs at a fine-grained level results in significant performance degradation and vast log files that administrators must comb through for relevant details. Consequently, administrators rely on system-specific metrics and analysis tools to diagnose problems. As modern companies deploy many different types of data management systems and applications (e.g. database servers, data lakes), administrators must possess a robust understanding of all of these systems and their associated analysis tools.

It would be highly desirable if administrators could use a *single* tool for performance debugging and analysis rather than having to learn, understand and use different suites of tools for each system and application. To meet this need, we present **Sentinel**, a system analysis tool that integrates with (standard) debug logging libraries (e.g., Log4j2 [3]) to efficiently build models of system behaviour and performance. Concretely, Sentinel captures debug log messages and models the relationships between them. As log messages correspond to system events and system execution logic [5, 6], Sentinel's models capture *system behaviour*. Sentinel uses these models to detect and pinpoint underlying performance problems without relying on system-specific features or characteristics. Notably, Sentinel's insights are available to *all* applications

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3384691

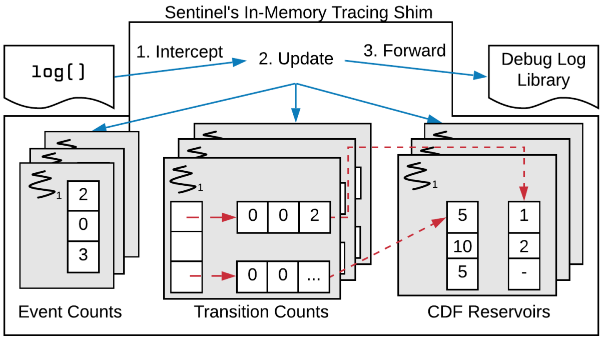


Figure 1: Sentinel intercepts debug logging library calls, using them to build per-thread models of system behaviour before forwarding the calls along to the logging library.

and systems using standard logging libraries *without* requiring significant code modifications.[1]. Unlike other approaches [8, 11], Sentinel does not require fine-grained debug logs to be emitted to disk to abstract them and model behaviour. Instead, Sentinel intercepts logging calls and constructs its models in-memory with little overhead.[2]

We discuss the Sentinel system and its features next. In Section 3, we describe the scenarios we present in our demonstration and highlight the utility of Sentinel's techniques.

2 THE SENTINEL SYSTEM

Sentinel extracts models of behaviour from applications and data systems, performing comparisons among these extracted models to highlight important differences in behaviour and performance. We start by discussing how Sentinel extracts these models with low overhead, followed by how Sentinel efficiently compares these models to pinpoint important performance differences. We then discuss extensions to these models to include detailed timing information.

2.1 Extracting Behaviour Models

Applications and data systems use debug logging libraries for both auditing and bottleneck analysis purposes. Debug logging libraries tend to expose a similar interface, which Sentinel uses to extract its models.

When the system issues a `log(LEVEL, msg, args...)` call, Sentinel intercepts it and translates the message to an event ID using its file name and line number in source code. For example, suppose that we are handling the first `log()` call in Figure 2 and that it corresponds to a GetLock event with ID 3. This event ID is used as an offset into per-thread arrays of data structures (Figure 1). After determining the event's ID, Sentinel increments its occurrence count and the number of times the system has transitioned from the previously observed event ID to the current ID. For example, if we are moving from a prior event with ID 1 to the current event with ID 3, we increment the count in the third slot of the event count array (3 in Figure 1) and then follow the pointer from the first slot in the transition count array to another counter array, incrementing the third slot (2 in Figure 1) there. These simple operations on per-thread data structures are very efficient and avoid cross-thread synchronization.

```
LockResult LockAcquireExtended( Lock *lock ) {
    ...
    log( INFO, "Get Lock: \%p", lock );
    WaitAcquire( lock );
    log( INFO, "Lock acquired." );
    ...
}
```

Figure 2: Simplified lock acquisition code adapted from PostgreSQL.

After updating these statistics for the log message, Sentinel forwards the message to the logging library. The logging library may write the message to disk depending on the log's severity LEVEL. However, Sentinel captures statistics among events of *all* granularities without requiring that the logging library emit everything to disk.

Sentinel's per-thread data structures are written to disk periodically or when the system shuts down. Per-thread counters for event frequencies and event transition counts are summed to give the total event and transition counts, thereby deriving an *event transition model* for the whole system. Sentinel compares these transition models to highlight differences in behaviour and performance.

2.2 Comparing Transition Models

Sentinel analyzes transition models, which correspond to system behaviour under a given configuration or workload, to find and present the most significant differences among events and event transitions to the user (Figure 3).

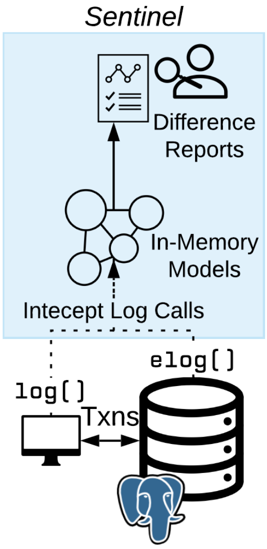
Sentinel compares extracted event frequencies to determine which events vary the most in popularity among extracted models. That is, for transition models m_1 and m_2, Sentinel reports the top k differences in event frequency:

$$\max\left(\frac{freq_{m_1}(e)}{freq_{m_2}(e)}, \frac{freq_{m_2}(e)}{freq_{m_1}(e)}\right) \tag{1}$$

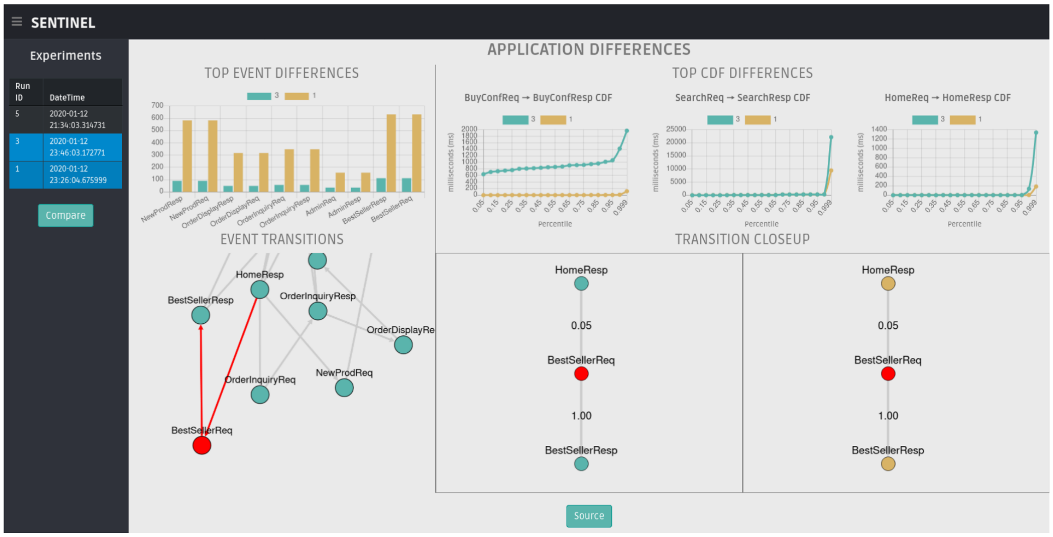
where $freq_m(e)$ returns the frequency of event e per model m. By using ratios of frequencies, Sentinel precludes popular

[1] Integrating Sentinel with PostgreSQL and the TPC-W [14] clients required changing less than 60 and 15 lines of code in each, respectively.

[2] In our experiments with 25 clients executing a YCSB-C workload, we observed that Sentinel reduces PostgreSQL transaction throughput by a mere 5% while fine-grained debug logging degrades throughput by 90%.



a)



Run ID	DateTime
5	2020-01-12 21:34:03.314731
3	2020-01-12 23:46:03.172771
1	2020-01-12 23:26:04.675999

b)

Figure 3: (a) Demo system architecture.

Figure 3: (b) Sentinel's User Interface.

events from always being reported as the top differences. Sentinel similarly compares event transition probabilities in extracted models to determine and report the k largest differences in transition probability.

2.3 Extracting CDFs

Although the techniques presented in the previous section highlight differences in event occurrences and transition probability, the time it takes to transition between system events is also important. For example, it is not enough to know how many times a data system acquires locks to detect lock contention; it is also important to know how long it takes to acquire the locks. To this end, Sentinel captures transition times between events, presenting them as cumulative distribution functions (CDFs).

As with the event frequencies and transition counts, transition times are captured and stored on a per-thread basis. When the system issues a `log()` function call, Sentinel obtains the current time using the `clock_gettime` system call. It compares the obtained time to the one it obtained for the previous log call; the difference between them is the time to transition between `log` calls. As it is impossible to store all of the transition times between events due to memory constraints, Sentinel employs adaptive damped reservoir sampling [2] to store samples for each unique transition. These samples are used to accurately estimate CDFs.

2.4 Comparing CDFs

In addition to the differences between event frequencies and event transition probabilities, Sentinel also reports the top differences in transition time CDFs. To do so, it computes the earth-mover's distance between two CDFs. The earth-mover's distance provides an intuitive measure of the differences between CDFs as it quantifies the minimum effort needed to transform one probability distribution into another [13]. Transition time CDFs with the largest differences are presented in Sentinel's UI (Figure 3b).

2.5 Sentinel Interface

Sentinel stores its extracted transition models in a database and presents detailed comparisons between them in a web application. The core Sentinel system is written in Python and uses numpy and pyemd [12] modules to perform model comparisons. The Sentinel user interface uses PHP and Javascript, rendering its charts and transition graphs with Chart.js [1] and Cytoscape.js [4] respectively.

Sentinel supports *brushing and linking* [7] across multiple visualizations. For example, selecting an event in the event frequency differences chart will zoom in on the event in the event transitions pane and present detailed information in the transition close-up panes. By combining the information from multiple visualizations, Sentinel presents a detailed and holistic picture of system behaviour differences.

3 DEMONSTRATION SCENARIOS

Our demonstrations use Sentinel to investigate behavioural and performance differences in PostgreSQL and TPC-W [14] benchmark clients under a variety of transaction mixes and system configurations (Figure 3a). Importantly, Sentinel does not need *a priori* knowledge of events or their relative importance; all insights are provided using black-box model comparisons. *Attendees will pose as analysts using Sentinel* to investigate performance issues by comparing PostgreSQL and benchmark client behaviour in these scenarios to that of a known good (baseline) configuration (TPC-W *ordering mix*). Two such scenarios are described next. A video of these demonstration scenarios is available online.

3.1 Lock Contention

Our first demonstration provides an overview of Sentinel's features and shows how Sentinel can be used to detect and investigate the common case of database lock contention. We induce lock contention by inserting a short `sleep()` in the BuyConfirm transaction while holding database locks. We compare this poorly-performing (current) workload's behaviour to that of the baseline configuration to determine the source of the problem.

Using Sentinel's reports, we determine that there are far fewer benchmark client events of *all* types in the current workload compared to the baseline, indicating widespread performance degradation (Figure 3b). The largest CDF difference in client behaviour highlights a large increase in BuyConfirm transaction execution time. We observe that the largest differences in database event frequencies and CDFs of transition time pinpoint increased lock wait events and lock wait times.

Given this investigation, the analyst concludes that lock contention is the primary cause of the performance degradation in the current workload. This contention affects the BuyConfirm transaction while hampering the performance of the workload as a whole.

3.2 TPC-W Transaction Mixes

Workload access patterns often change [9]; our second demonstration investigates complex behavioural differences in such a case by supposing the transaction mix has changed from the baseline *ordering* mix to the TPC-W *browsing* mix. As analysts, we observe that performance has degraded and must determine what has changed.

The largest event differences in client behaviour pinpoint differences in transaction popularity between the two workloads. Read-only transactions like the NewProduct and BestSellers transactions are more popular in the browsing mix than in the ordering mix, enabling the analyst to conclude that the browsing mix is more read-heavy than the ordering mix. The transition time CDFs for these popular read-only transactions indicate that they take longer to execute than the update transactions, which affirms that these more expensive queries are impacting system throughput.

The largest behaviour differences in the database reveal that the browsing mix has lower query throughput than the baseline ordering mix despite the baseline workload exhibiting more lock contention and disk writes. Sentinel also highlights increased checkpoint throttling behaviour in the browsing mix as fewer dirty pages are written to disk. Thus, Sentinel indicates that the read-only queries in the current workload are more complex, resulting in lower overall performance. Given these insights, the analyst optimizes the highlighted, popular, long-running transactions in the current workload to improve performance.

REFERENCES

[1] 2020. Chart.js Github. https://www.chartjs.org/. (2020).

[2] Peter Bailis, Edward Gan, et al. 2016. MacroBase: Prioritizing Attention in Fast Data. (2016), 541–556.

[3] Apache Software Foundation. 2020. Apache Log4j 2. https://logging.apache.org/log4j/2.x/.

[4] Max Franz et al. 2015. Cytoscape.js: a graph theory library for visualisation and analysis. *Bioinformatics* 32, 2 (09 2015), 309–311.

[5] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. 2008. An Automated Approach for Abstracting Execution Logs to Execution Events. *J. Softw. Maint. Evol.* 20, 4 (July 2008), 249–267.

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

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

[8] Kamdem Kengne et al. 2013. Efficiently Rewriting Large Multimedia Application Execution Traces with Few Event Sequences. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '13).* 1348–1356.

[9] Lin Ma et al. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems *(SIGMOD '18).* ACM, New York, NY, USA, 631–645.

[10] Ashraf Mahgoub, Paul Wood, et al. 2017. Rafiki: A Middleware for Parameter Tuning of NoSQL Datastores for Dynamic Metagenomics Workloads *(Middleware '17).* ACM, New York, NY, USA, 28–40.

[11] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. *Nsdi* (2012), 353–366.

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

[13] Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. 2000. The Earth Mover's Distance as a Metric for Image Retrieval. *International Journal of Computer Vision* 40, 2 (01 Nov 2000), 99–121.

[14] TPC. 2000. TPC Benchmark W (Web Commerce). http://www.tpc.org/tpcw.

[15] Dana Van Aken, Andrew Pavlo, et al. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning *(SIGMOD '17).* ACM, New York, NY, USA, 1009–1024.

[16] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. DBSherlock: A Performance Diagnostic Tool for Transactional Databases *(SIGMOD '16).* ACM, New York, NY, USA, 1599–1614.

[17] Xu Zhao, Kirk Rodrigues, et al. 2017. Log20: Fully Automated Optimal Placement of Log Printing Statements Under Specified Overhead Threshold *(SOSP '17).* ACM, New York, NY, USA, 565–581.