Universal Database System Analysis for Insight and Adaptivity

by

Brad Glasbergen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2022

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Tilmann Rabl Professor, Digital Engineering Fakultät, University of Potsdam, Hasso Plattner Institute
Supervisor:	Khuzaima Daudjee Research Associate Professor, Cheriton School of Computer Science, University of Waterloo
Internal Member:	Tamer Özsu University Professor, Cheriton School of Computer Science, University of Waterloo
Internal Member:	Semih Salihoglu Associate Professor, Cheriton School of Computer Science, University of Waterloo
Internal-External Member:	Patrick Lam Associate Professor, Department of Electrical and Computer Engineering, University of Waterloo

Author's Declaration

Statement of Contributions

Some portions of this thesis are based on peer-reviewed joint work with Prof. Khuzaima Daudjee, Prof. Daniel Vogel, Prof. Jian Zhao, Michael Abebe, Amit Levi, and Fangyu Wu [49, 50, 52]. I am the first author and the primary contributor in each work.

Abstract

Database systems are ubiquitous; they serve as the cornerstone of modern application infrastructure due to their efficient data access and storage. Database systems are commonly deployed in a wide range of environments, from transaction processing to analytics.

Unfortunately, this broad support comes with a trade-off in system complexity. Database systems contain many components and features that must work together to meet client demand. Administrators responsible for maintaining database systems face a daunting task: they must determine the access characteristics of the client workload they are serving and tailor the system to optimize for it. Complicating matters, client workloads are known to shift in access patterns and load. Thus, administrators continuously perform this optimization task, refining system design and configuration to meet ever-changing client request patterns.

Researchers have focused on creating next-generation, natively adaptive database systems to address this administrator burden. Natively adaptive database systems construct client-request models, determine workload characteristics, and tailor processing strategies to optimize accordingly. These systems continuously refine their models, ensuring they are responsive to workload shifts. While these new systems show promise in adapting system behaviour to their environment, existing, popularly-used database systems lack these adaptive capabilities. Porting the ideas in these new adaptive systems to existing infrastructure requires monumental engineering effort, slowing their adoption and leaving users stranded with their existing, non-adaptive database systems.

In this thesis, I present Dendrite, a framework that easily "bolts on" to existing database systems to endow them with adaptive capabilities. Dendrite captures database system behaviour in a system-agnostic fashion, ensuring that its techniques are generalizable. It compares captured behaviour to determine how system behaviour changes over time and with respect to idealized system performance. These differences are matched against configurable adaption rules, which deploy user-defined functions to remedy performance problems. As such, Dendrite can deploy whatever adaptions are necessary to address a behaviour shift and tailor the system to the workload at hand. Dendrite has low tracking overhead, making it practical for intensive database system deployments.

Acknowledgements

First, I would like to thank my advisor, Khuzaima Daudjee. Khuzaima's steadfast support, encouragement, and advice were indispensable throughout my PhD. Thank you for the many hours of research and life advice, for indulging my research ideas, and for teaching me to be an independent researcher.

I am thankful to my committee members, Tamer Özsu, Semih Salihoglu, Patrick Lam, and Tilmann Rabl for their valuable suggestions and feedback. I appreciate the time they took to read and comment on this thesis, which helped to refine its content. I am also thankful to Reza Ramezan, who provided insightful feedback on the proofs.

I am endlessly grateful to my wife, Rebecca Mayers. Thank you for your ardent support throughout graduate school, for advocating for my success, and for propelling this thesis to completion.

I am thankful to my friends in the Data Systems Group at the University of Waterloo. I am particularly thankful to Michael Abebe, who served as an inspiring role model and key collaborator throughout graduate school.

I am thankful to my parents for their encouragement and assistance in completing my graduate studies.

Finally, I am grateful to the Province of Ontario, the Natural Sciences and Engineering Council of Canada, and the Cheriton School of Computer Science for funding this research.

Dedication

For my beloved wife, Rebecca Mayers. Thank you for always believing in me.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Relational database systems are a critical backbone of modern application infrastructure [107]. They are used to store and access data across a variety of application domains, ranging from transaction processing to analytics.

Different applications have different access characteristics. For example, online transaction processing workloads (OLTP) tend to be update-intensive, but have relatively simple transactions. By contrast, online analytical processing workloads (OLAP) rarely update data but use long-running, complex operations.

Due to these differences in access patterns, different data structures and algorithms perform better for different workload types. For example, updates are more efficiently conducted on row-oriented storage than column-oriented storage, making row-orientation more suitable for transaction processing workloads. On the other hand, column-oriented storage is preferable for analytics-intensive workloads [1]. Similarly, optimistic concurrency control techniques are superior for highly-concurrent workloads executing on commodity hardware with low contention, while locking-based concurrency control protocols are preferable in the highly-contended case [112]. Generally, it is not possible to select a single system configuration — encompassing physical design and processing techniques — that performs well across all workloads and environments [106, 108].

As database systems are a vital component in application infrastructure [107], the storage and processing algorithms deployed by the system must be tailored to the workload at hand. To this end, administrators vigilantly monitor client workloads, analyzing them to determine access and load characteristics. To do so, they comb through large debug-logging files and hundreds of system metrics [125] exported by the database system to determine how the system behaves in response to the workload. This information is used

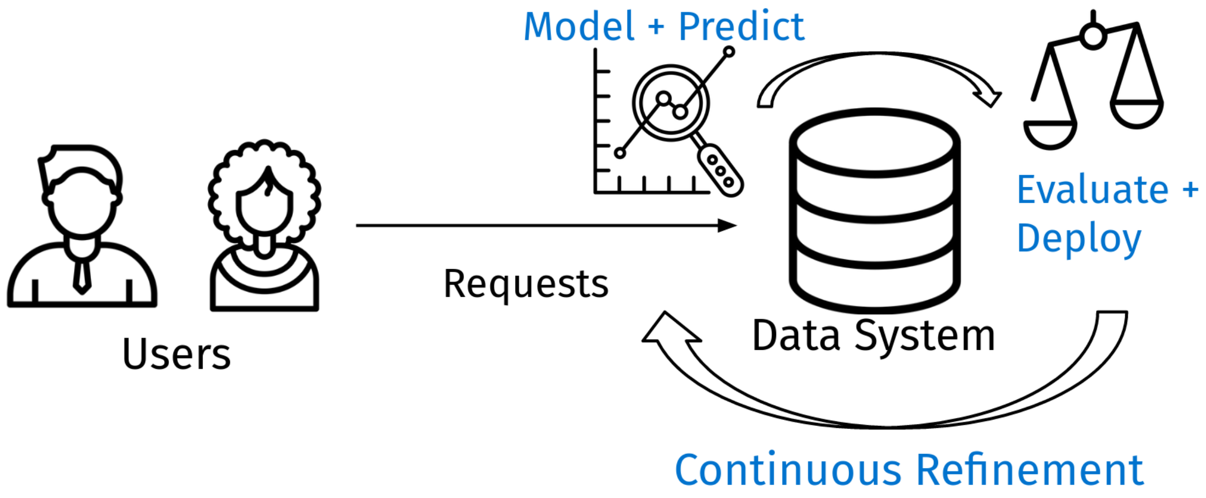


Figure 1.1: A natively adaptive database system architecture.

to determine the amount of resources to be provisioned to the system and to tailor its request processing strategies.

Furthermore, client workloads are known to shift in access patterns and demand over time [3,81,110,111]. Therefore, administrators must continuously monitor the workload to determine how it evolves over time, updating the database system's processing strategies to reflect the workload's new characteristics.

In recent years, researchers have proposed next-generation, natively *adaptive* systems [34, 81, 93, 95] as a remedy to this administrator burden (Figure 1.1). These systems build models of the client workload to determine its access characteristics, evaluate the benefits and trade-offs of different storage and query processing strategies, and deploy the choices expected to maximize benefits for client requests. In particular, these systems *adapt* their physical design, processing decisions, or resource allocation according to their environment. When the workload changes, the system updates its models to reflect the workload's new behaviour, in turn refining the adaptions chosen for the workload.

While these natively adaptive systems have shown promise in automatically tailoring themselves to client workloads, their techniques are not yet widespread. Popular, open-source database systems, such as PostgreSQL [59], either have limited adaption capabilities or lack them entirely. Unfortunately, retrofitting systems with adaptive components requires immense developer effort [94]. As every database system is unique, determining the circumstances under which adaptions should be deployed is currently system-specific,

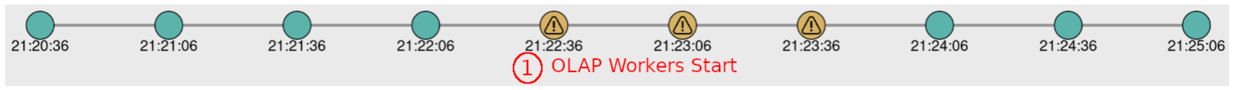


Figure 1.2: Dendrite's user interface highlighting a behaviour change.

time-intensive and error-prone.

Natively adaptive systems are often research prototypes under active development and far less battle-tested than existing industrial systems. It is therefore unreasonable for users of popular industrial systems to switch to using a recently proposed adaptive system; running mission-critical infrastructure on prototypes is ill-advised. Therefore, users are stranded with database systems lacking adaptive capabilities. Administrators continue to shoulder the Sisyphean task of system optimization, despite the promise of adaptive systems on the horizon.

Dendrite addresses this challenge by "bolting" onto existing database systems, automatically detecting system behaviour differences, and deploying adaptions to address them. Database developers use Dendrite's intuitive adaption rules to codify the appropriate response to behaviour differences, which it executes when rule conditions are satisfied. In doing so, Dendrite enables otherwise unequipped database systems to respond to changes in workload and system behaviour. To integrate with Dendrite, developers: (*i*) modify their system's logging library and link it against Dendrite, or (*ii*) inject Dendrite directly into the system's executable code via binary instrumentation, requiring no source code modification. As such, popular database systems can obtain Dendrite's adaptive benefits with little engineering effort.

1.1 Motivational Example

To demonstrate Dendrite's utility, I will show how it can respond to workload changes in PostgreSQL 14.1. Figure 1.2 shows Dendrite's behaviour differences timeline for an experiment in which a 10-client OLTP workload executes against PostgreSQL, with 2 clients joining the system to execute OLAP queries a few minutes later. Each circle on the timeline indicates a 30-second time interval, where teal circles represent expected behaviour, and gold circles represent a behaviour difference. Dendrite is configured to monitor PostgreSQL's execution behaviour.

Once the OLAP clients join the system, Dendrite reports a significant difference in PostgreSQL's execution behaviour compared to its behaviour on the previously executing

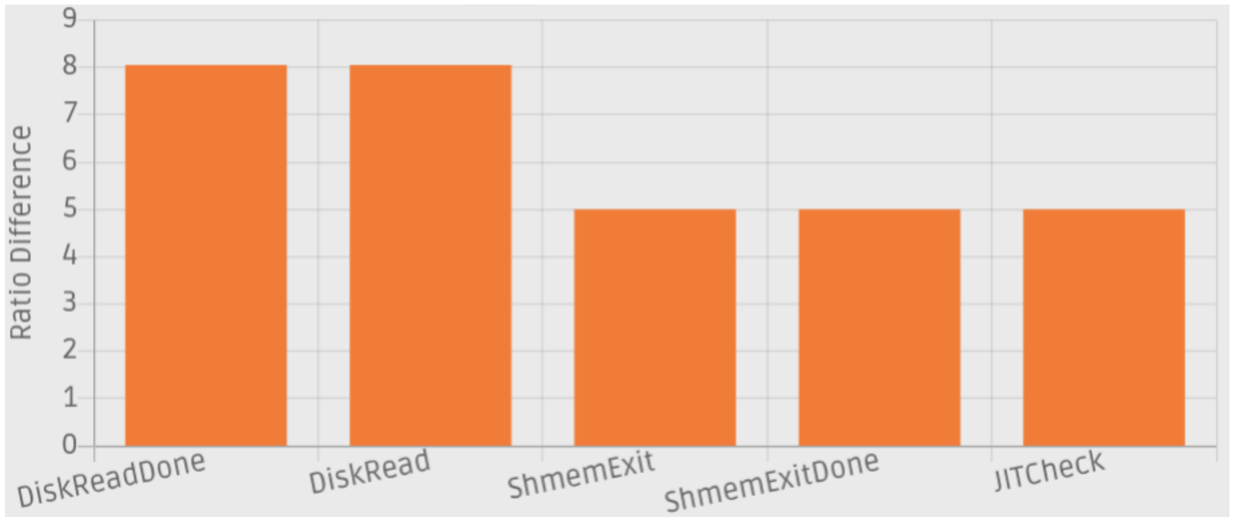


Figure 1.3: Dendrite's user interface showing the largest proportional differences between current and expected system behaviour.

OLTP workload. It marks this shift in its user interface with a gold circle (1). Consulting Dendrite's reported behaviour differences (Figure 1.3), we observe a large increase in DiskRead events, which corresponds to a disproportionate rise in page reads from disk. There is also a marked increase in shared memory exit events, which are part of the termination procedure for the background parallel scan processes that PostgreSQL uses for the analytics queries. By characterizing each database process's behaviour, Dendrite determines that two analytics workers are executing.

Dendrite matches this active process and behaviour difference information against a set of rules that determine how it should respond. Using these *adaption rules*, Dendrite responds by re-routing the analytics queries to a more suitable column-store database, MonetDB [11], which processes the queries more efficiently and reduces analytic query latency by 25×. Afterward, Dendrite returns system behaviour to expected, as indicated by the subsequent teal circles on the timeline in Figure 1.2.

1.2 Contributions

Dendrite captures its behaviour information in a system-agnostic way and therefore integrates easily with any database system. It extracts system events, such as page flushes and checkpoints, from ubiquitously used debug-logging calls and encodes them alongside resource-consumption metrics in *behaviour models*. These models capture how the system moves between events, enabling rich behaviour analysis and difference detection. If the target system uses logging sparingly (or not at all), Dendrite retrieves the information it requires through binary instrumentation. As Dendrite's adaption rules support user-defined functions as responses, it can deploy a wide range of augmentations to adapt the running system in response to a behaviour shift. Dendrite consumes little memory and has low overhead, making it suitable for intensive, high-performance database system deployments.

Dendrite provides these features to deliver the following key contributions:

1. Dendrite uses novel, system-agnostic techniques to efficiently extract events and fine-grained resource consumption from a running database system.
2. Dendrite encodes system behaviour in expressive models that capture complex system-behaviour patterns while automatically minimizing model memory consumption.
3. Dendrite enriches database systems with adaptive capabilities through its adaption rule framework. Adaption rules are intuitive and can respond to disparate behaviour changes to improve system performance.
4. Dendrite is effective in adapting multiple popular database systems and improving their performance, as evidenced through comprehensive and representative case studies spanning the PostgreSQL, MariaDB, and SQLite database systems. Beyond showcasing Dendrite's capabilities, these studies offer valuable insight for researchers investigating generalized system adaptivity; findings show that models constructed using widely available information sources (debug logging and operating system metrics) are effective in detecting system behaviour changes and that adaption criteria are portable across database systems from the same (relational) domain.

The rest of the thesis describes these contributions and demonstrates Dendrite's effectiveness in practice. Dendrite is fundamentally different from prior approaches in its *generalizability* across systems, comprehensive *behaviour modelling*, and support for a wide range of *online adaptions*.

1.3 Thesis Organization

The layout of this thesis is as follows. In Chapter 2, I present the necessary background and terminology to simplify Dendrite's presentation. In Chapter 3, I describe the core behaviour-extraction techniques and behaviour models underlying the Dendrite system. In Chapter 4, I detail enhancements to these models that capture resource consumption in code and binary instrumentation techniques that extend Dendrite's applicability to an even broader range of systems. This chapter also describes Dendrite's adaption rule framework that effects behaviour changes, along with its user interface and tools that ease deployment. Chapter 5 demonstrates Dendrite's applicability to a wide range of database systems and workloads, presenting insights into the portability of rules between similar database systems and lessons learned for generalized system adaptivity. Related work is covered in Chapter 6. I discuss future work and conclude in Chapter 7. The central parts of the thesis are summarized next.

1.3.1 Universal Behaviour Model Extraction

Behaviour models are at the core of Dendrite's functionality. They provide comprehensive information about system behaviour to enable robust behaviour comparisons, yet remain lightweight enough to avoid degrading system performance during their capture.

I will first present the novel extraction techniques that Dendrite uses to obtain these behaviour models from arbitrary data systems. A key insight is that debug-logging libraries are ubiquitously used, share a common Application Programming Interface (API), and output information about system events. Hence, Dendrite can extract the information it requires from the system by intercepting debug-logging calls and encoding its insights into Markov chains that describe system behaviour evolution.

By building its behaviour models on a per-thread basis using thread-local storage, Dendrite mitigates cross-thread contention during monitoring and model construction. This design is key in capturing expressive models without unduly impairing performance.

1.3.2 Robust Behaviour Capture

While Markov chains are an efficient tool for capturing event transitions, they report the probability of transitioning between system events as being conditional only on the immediately preceding event. This reductionist approach would preclude Dendrite from properly attributing event origins. For example, it would not be able to tell whether buffer pages being flushed to disk in PostgreSQL is due to the background writer process performing its expected flushing duties, or whether the buffer pool must unexpectedly flush dirty pages to disk to accommodate page reads.

To this end, I extend the Markov chains underlying Dendrite's behaviour modelling into comprehensive variable-order Markov models. These models capture the transition complexity of the traced events adaptively, encoding longer-term system-behaviour trends and resource consumption (in terms of disk usage, memory allocation, network traffic) as the system moves through the code.

While debug logging is widely used and developers tend to place logging statements in information-dense places [128], relying on them would preclude Dendrite from use in environments where logging is not used effectively (or at all). Dendrite eliminates this logging dependency by optionally deploying binary instrumentation techniques that extract information directly from targeted function calls.

1.3.3 Adaptivity Framework

Given a representative behaviour model and an extracted model corresponding to the system's current behaviour, Dendrite compares them using a novel event-proportion-based model-comparison technique. This technique determines how different the models are overall and which differences are the most important. If the models are significantly different, Dendrite evaluates a set of adaption rules over these differences to determine how to respond.

Dendrite provides a set of built-in primitives that act as predicates over behaviour differences to ease rule composition. For example, `prob_diff(event)` computes the difference in frequency of `event` in the representative model compared to that of the extracted. If an adaption rule's conjunctive conditions match the behaviour differences, then a user-defined function (UDF) associated with the rule is executed to address the behaviour difference.

Dendrite also provides users with a web interface that describes the system's behaviour evolution over time, along with any executed adaption rules. Administrators can explore the system's behaviour and event relationships in detail, and additional rules can be registered using the interface to respond to behaviour differences in the future.

1.3.4 Case Studies and Lessons Learned

To show Dendrite's effectiveness across database systems, I integrate Dendrite with PostgreSQL, MariaDB, and SQLite. I present case studies for each system that show that Dendrite can both detect behaviour differences induced by the workload or environment, and respond to these differences appropriately through adaption rules.

Beyond per-system efficacy, I also show the generality of Dendrite's adaption rules by executing the same three case studies on both PostgreSQL and MariaDB, open-source database systems that exhibit broadly similar characteristics. Results demonstrate that adaption rules composed for one database system can be ported to another without significant changes through a translation layer that determines equivalent events across the systems. Similarly, while the response UDF is similar overall, some translation is performed to convert PostgreSQL adaption logic to MariaDB adaption logic.

Finally, I discuss key requirements for every adaption tool and future work that further extends Dendrite's scope and applicability.

Chapter 2

Terminology

2.1 System Behaviour

Database systems are complex, containing a large number of components and features that work together to process user requests. Dendrite's goals are to (*i*) determine whether the system's current *behaviour* differs from its idealized behaviour, and, if so, (*ii*) effect changes that restore the system to expected functionality and performance.

In this thesis, system behaviour is defined as the execution patterns of the database system. This definition captures a wide body of research that models database systems in a myriad of different ways. For example, some research relies on database performance metrics, while others require direct instrumentation. In all cases, the goal is to capture a signature of how the database is processing requests and its performance. Dendrite differs from prior work in its ability to capture and model database system behaviour in a system-agnostic fashion using debug logging and `libc` calls.

2.2 Adaption

Dendrite captures database system behaviour to determine how the system's current behaviour differs from expectations. These differences determine how Dendrite will *adapt* the system, such that the system returns to its idealized behaviour.

Database systems accept user input from configuration files, data-definition language (DDL) commands, and data-modification language (DML) commands. Dendrite uses these input vectors to tailor the system's processing behaviour, improving performance for the workload at hand. Fundamentally, these inputs affect the database system's physical design, processing decisions, and resource provisioning [95].

Database system *physical design* refers to the storage formats, auxiliary data structures, and data placement decisions that govern how data is stored and accessed. Physical design decisions involve creating indexes, selecting which views to materialize, and determining partitioning and replication strategies for data. Each decision is associated with cost-benefit trade-offs that must be carefully navigated to optimize for the workload at hand. For example, indexes accelerate access to data items when filtering by indexed fields compared to a full table scan, but require maintenance when data items are updated.

Processing decisions refers to how the database system determines which access strategies to use for client requests and how it manages its background maintenance processes. A database system chooses among available access strategies based on the values of user-configured thresholds, plan hints, and hardware performance. Moreover, database systems feature background processes to maintain system metadata, address table bloat, and flush dirty pages to disk. User-configurable thresholds govern when these background maintenance processes execute and how many resources they consume, in turn significantly affecting system performance.

Resource Provisioning refers to the physical system resources (memory, CPU, network bandwidth) assigned to the database system or its sub-components. While some resources are configurable within the database system (e.g., PostgreSQL's buffer pool size, which stores disk pages in memory), others are external and configured using containers, virtual machines, or cloud-provider interfaces. The amount of resources the database system can use to process requests plays a key role in overall system performance.

Dendrite uses arbitrary user-defined functions to deploy changes and can therefore augment system behaviour along any of these dimensions. For example, Dendrite can deploy new indexes based on the workload, manage the frequency of background checkpointing events, and increase the resources provisioned to the database system's buffer pool. For the rest of this thesis, change in any of these aspects in response to the workload or environment constitutes an *adaption*:

Definition 1. *An **adaption** is a change in physical design, processing decisions, or resource allocation made in response to the workload, database system behaviour, or computing environment.*

A running database system that deploys adaptations to tailor itself to client workloads or the environment is an *adaptive database system.*

Chapter 3

Core Models and Behaviour Difference Detection

This chapter presents the core behaviour extraction and modelling techniques that Dendrite uses to capture database system behaviour and determine how the system has changed over time or with respect to an expected behaviour baseline. Experiments at the end of this chapter show the effectiveness of these techniques in modelling database system behaviour and detecting salient differences.

Dendrite's modelling framework (Figure 3.1) is implemented in two steps. First, the *in-memory tracer* extracts behaviour models on a per-process, per-thread basis and outputs each model into a separate file. The tracer extracts this information by intercepting debug logging calls and encoding it into Markov chain based behaviour models. Next, a background process combines these models to produce an overall model of system behaviour, loading it into a model storage database. Dendrite's control server compares models stored in this database to determine behaviour differences and outputs a behaviour difference report. In this chapter, this reporting process is assumed to be triggered by an administrator; the subsequent chapter builds on the core techniques outlined here to perform behaviour model comparisons on the fly during system execution, using the reported differences to deploy adaptions and improve performance.

As one of Dendrite's primary goals is to provide adaption capabilities to a broad range of database systems, its extraction techniques do not rely on system-specific features or metrics. Furthermore, Dendrite minimizes integration effort; requiring engineers to design and build complex system-specific scripts to integrate their database systems with Dendrite defeats its goals of generalizability and hinders its adoption. Finally, Dendrite does not

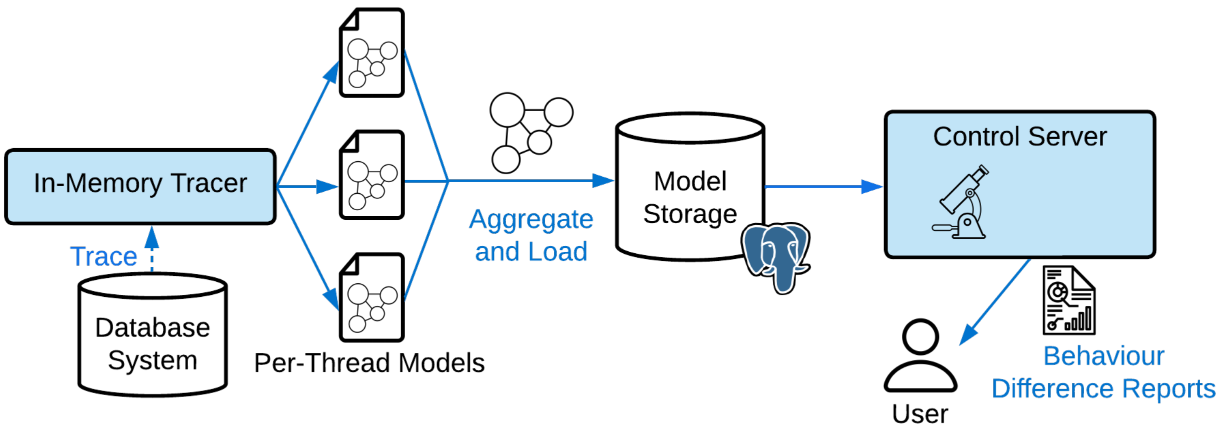


Figure 3.1: Dendrite's architecture for difference detection.

significantly hamper system performance. The following sections describe how the in-memory tracer and control server achieve these objectives.

3.1 Low Overhead Event Extraction

Nearly all systems use debug logging [48,90,122,125]; Dendrite exploits this fact to provide universal behaviour modelling and adaption.

Although there are many different debug logging libraries (e.g., Google Logging [56], Log4j [44], and Spdlog [87]), they all provide a similar interface for database systems to output a debug log message to file:

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

where `message` is a string with placeholders for variables, composing the body of the text to be written to disk, and `format_args` contains the variables to be spliced into the message body. For example, the PostgreSQL code shown in Figure 3.2 has a logging call with `LOG_LEVEL` DEBUG2 and splices the variables `pid` and `port->sock` into the log message.

`LOG_LEVEL` indicates the log level of the message. Log levels communicate the importance of the message to output, and range from critical (FATAL) to fine-grained and informational (DEBUG5) (Figure 3.3). The logging library is configured to emit messages

```
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 3.2: A PostgreSQL debug logging call issued while starting a new backend process.

Level	Name	Description
10-15	DEBUG1-DEBUG5	Fine-grained informational details
16	LOG	Server operational messages
17	INFO	Messages specifically enabled by user (VERBOSE)
19	WARNING	Warnings, unexpected behaviour
21	ERROR	Error, abort transaction
22	FATAL	Fatal error, abort process
23	PANIC	Fatal error, abort all server processes

Figure 3.3: A selection of PostgreSQL log levels and their descriptions. Higher numbers are more coarse-grained and important.

of coarser granularity than a specified threshold to disk. For example, a system configured with log level INFO will emit a message at the high-importance FATAL level to disk, but not a message at the fine-grained DEBUG level.

It is well-known that detailed logging results in considerable performance overheads [122, 125]. Two issues 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 overhead may be considerable. 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. Administrators minimize these overheads by configuring database systems to use a high log level threshold in production and thus reduce the volume of messages logged to disk.

Dendrite avoids these overheads by integrating directly with debug logging libraries to track events and event transitions in memory (Figure 3.4). In particular, when a running database system issues a `log` call, the debug logging library forwards the call to Dendrite's

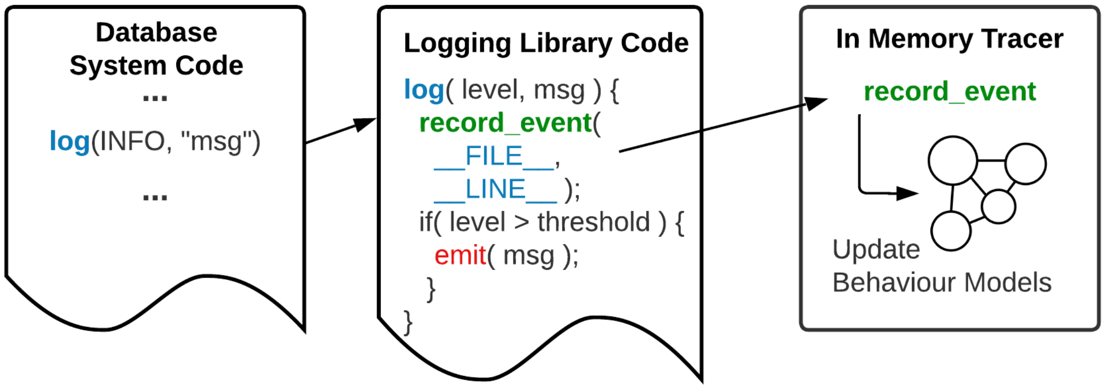


Figure 3.4: A logging call is handled by the logging library, which calls `record_event`. `record_event` is implemented in Dendrite's in-memory tracer and updates system behaviour models.

in-memory tracer by calling the tracer's `record_event` function. Afterward, the logging library handles the call as usual, emitting the message only if its level is higher than the preconfigured emission threshold. Note that this procedure ensures Dendrite captures all messages *regardless* of whether they are ultimately written to disk per the log level threshold. As such, the database system may output coarse-grained logs for auditing purposes as usual while obtaining Dendrite's analysis over log messages of all granularities.

The in-memory tracer obtains the file name and line number of the position in source code that issued the logging call. This information is readily available via programming language primitives, like the `__FILE__` and `__LINE__` macros in C/C++. Dendrite uses this information to uniquely identify each event. Note that log messages originating from the same line in source code therefore map to the same event. This is by design; log messages are often parameterized by variables but correspond to the same system event [67, 68]. 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 [67,68,90]. By using file names and line numbers to identify events, Dendrite avoids the overhead of such scripts while maintaining generalizability.

3.2 Event Tracking

After obtaining a logging library call's originating file name and line number, Dendrite uses this information to look up its corresponding event in an in-memory hash map called

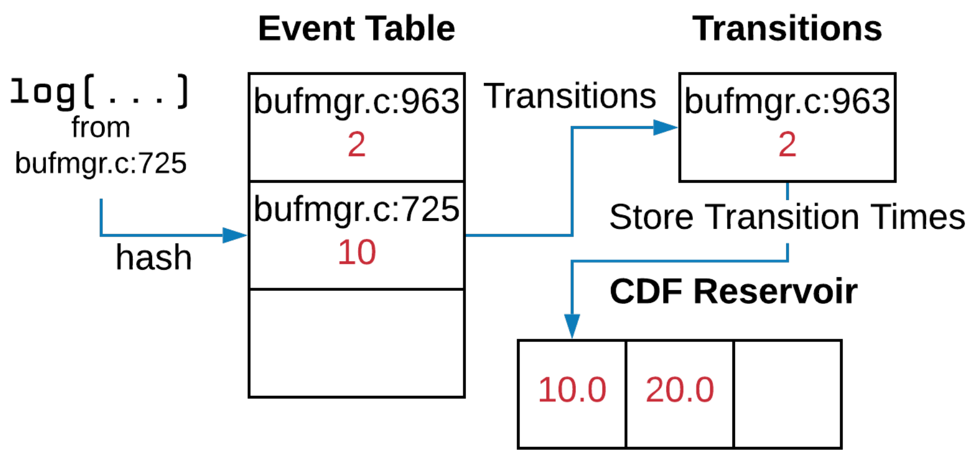


Figure 3.5: Dendrite's data structures used for tracking log messages, transitions, and empirical CDFs.

the *event table*. For example, in Figure 3.5, 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 file name, line number, event counter and pointers to event transition and transition-time 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 Chapter 3.3). 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 database system shuts down, each thread writes its event table to a per-thread file on disk. Before analysis, a background aggregator thread sums the counts for each event over all of these files to determine their overall frequency and compute their proportion. Along with event transition and timing information, Dendrite encodes these event proportions into a behaviour model for comparison against those of other workloads and system configurations. Although Dendrite internally associates events with only their file names and line numbers, the original log message can be obtained by looking up and reading the message at that event's location in source code. Moreover, Dendrite enables users to "tag" events with custom names to further improve the clarity of its reports. These tags are stored along with the models in a model storage database (Figure 3.1).

Event Differences Report: Dendrite's control server retrieves behaviour models from the model storage database and reports differences in events in descending order of their

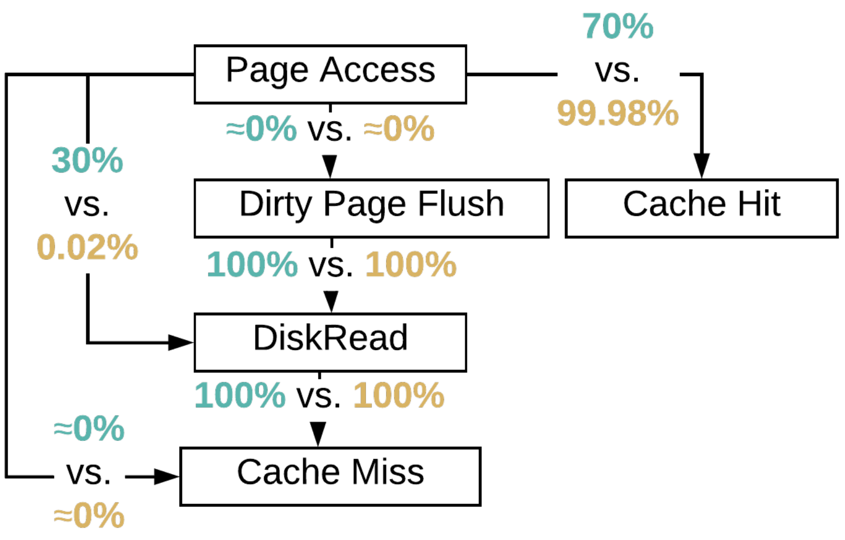


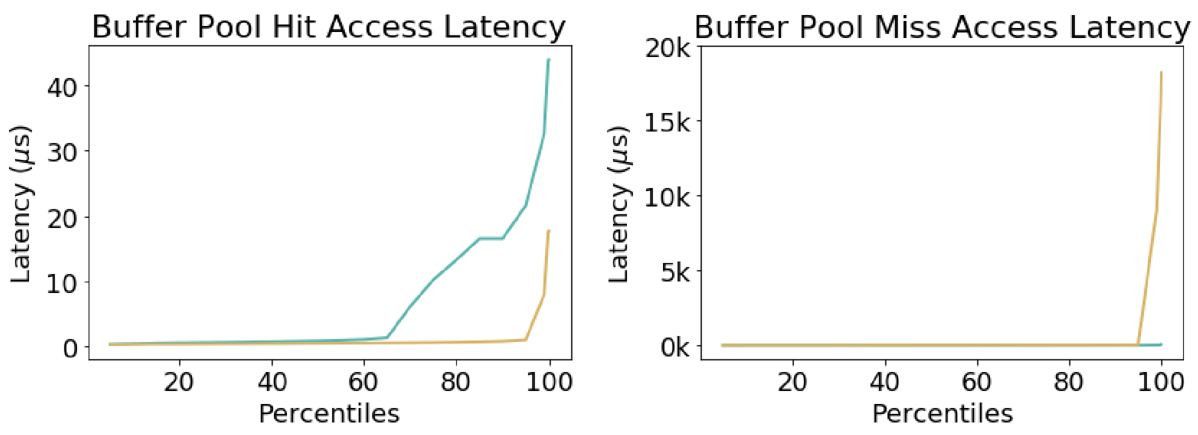
Figure 3.6: PostgreSQL page access transition graph for Scenario A (**blue**) and Scenario B (**orange**).

proportional differences. Dendrite reports differences in *event proportions* rather than raw event counts. Directly comparing event counts leads to high difference rankings in only the most frequently occurring events, while differences in event proportions differentiate the event distribution of overall system behaviour. Note that this comparison is fully system-agnostic; neither Dendrite's extraction nor ranking of event differences rely on any domain knowledge or system-specific characteristics.

3.3 Event Flows

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

To demonstrate these benefits, consider the buffer page access portion of the behaviour models Dendrite extracted for two scenarios, A and B (Figure 3.6). Scenario A uses



(a) PostgreSQL page access CDF. (b) PostgreSQL buffer pool cache miss CDF.

Figure 3.7: Buffer page access transition graph and latency CDFs for Scenario A (**blue**) and Scenario B (**orange**).

PostgreSQL's default buffer pool size of 128 MB, while Scenario B uses 8 GB. The structure of the models for both scenarios is the same and reflects the execution path in code for page accesses. When PostgreSQL needs to access page data (PageAccess), it first checks to see if the page is loaded into the in-memory buffer pool. If so, it retrieves the data directly from the page without further processing (termed a CacheHit event). Otherwise, it determines a page that should be evicted from the fixed-size buffer pool according to a replacement policy so that the requested page can take its place. If the page-to-be-replaced (termed the victim) has been modified relative to its on-disk version, these modifications must first be flushed to the disk (DirtyPageFlush). Afterward, the requested page's data is read into the buffer pool, replacing the victim page's content (DiskRead). This whole process of page replacement is termed a buffer pool cache miss. As cache misses present significant processing overheads, maximizing the proportion of cache hits is desirable.

Consulting Dendrite's behaviour model in Figure 3.6, we observe that although the buffer pool cache hit rate is high for both scenarios, Scenario A's hit rate is significantly lower than Scenario B's. In particular, there is a 70% probability of transitioning from the PageAccess event to the CacheHit event in Scenario A, while there is 99% chance in Scenario B. These differences in execution control flow patterns result in significant differences in buffer access latency, which Dendrite captures.

Dendrite enriches its models by efficiently computing empirical cumulative distribution functions (CDFs) of the time to transition between pairs of events. In contrast to the typical approach of computing average latencies, CDFs describe the full range of performance behaviour, including tail latencies, which are a key performance concern [75]. 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 (Chapter 3.3.3).

The combined CDF for buffer access latencies for the Scenarios is shown in Figure 3.7a. The overall page access times for both scenarios are similar until the higher percentiles, where the differences in cache misses manifest in higher latencies for Scenario A. The access latencies for buffer pool cache hits are far lower than tail cache miss times (Figure 3.7b), which emphasizes the importance of maximizing cache hits. However, note that cache miss times are not uniformly high because many cache misses can be resolved by retrieving the page from the operating system's cache 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 cache miss CDF (Figure 3.7b). Given these CDFs, one can conclude that the buffer pool is too small in 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, page access latencies can increase by orders of magnitude.

To support this functionality, I next describe how Dendrite provides the following key features:

1. Dendrite efficiently tracks transitions and the time to transition between pairs of events without significantly degrading performance.

2. Dendrite estimates and combines CDFs for individual transitions to compute the elapsed time for system functionality that straddles multiple events.

3. Dendrite effectively ranks differences in transition probability and CDFs of transition time, outputting them in a behaviour differences report.

3.3.1 Efficiently Tracking Event Transitions

During database system execution, Dendrite stores transition count information in event tables. In addition to the event frequency tracking described in Chapter 3.2, `record_event` also looks up the last executed event for the current thread, and increments the transition count from that event to the current event.

Each event in the event table is associated with a dynamically allocated list of counters. These lists contain one counter for each of the event's transitions (Figure 3.5). For example, `bufmgr.c:725` in Figure 3.5 transitions to the `bufmgr.c:963` twice out of the total of 10 times the event has been executed. As most events transition to only a handful of others, using a dynamically allocated list of counters reduces memory consumption considerably (Chapter 3.5.4).

These per-thread transition counters are written to disk alongside the event counts. Dendrite's background aggregation thread 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. For example, `bufmgr.c:725` has a 20% probability of transitioning to `bufmgr.c:963` using the values in Figure 3.5. As event transitions are stored in thread-local data structures, and the total transition counts are computed by the background aggregator thread using the outputted files, Dendrite avoids introducing contention among threads.

As noted earlier, since these behaviour models are backed by event and pairwise event transition counts, they are essentially Markov chains. Hence, Dendrite benefits from the rich literature on Markov chain analysis (e.g. random walks) but also inherits the memoryless assumption (it does not account for multiple prior events influencing a transition probability). Even these simple Markov chain models enable rich analysis and difference detection, as evidenced by the experiments in Chapter 3.5. The subsequent chapter extends Dendrite's behaviour models to address these limitations.

Transition Differences Report: Dendrite's control server compares event transition probabilities in behaviour models similarly to how it compares event probabilities. Concretely, Dendrite 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 an event e_2 from event e_1 in the first and second behaviour models, respectively.

3.3.2 Estimating Transition Time CDFs

While transition probabilities provide insight into the likelihood of moving from an event e_1 to event e_2, it is important to know how *long* this transition takes. For example, a

LockAcquire event is highly likely to transition to a LockAcquired event, but the time it takes to perform this transition corresponds to lock contention.

Dendrite supports this analysis by computing an empirical CDF of transition time for each event transition. When the in-memory tracer handles a `record_event()` call, it uses the `clock_gettime` system call to retrieve a nanosecond-precision timestamp from the operating system. It compares this timestamp to the timestamp obtained for the last event to determine the transition time between events. Transition times are stored in a fixed-size array of recorded times for this transition, called a *reservoir*, and used to compute the CDF.

Dendrite reduces sampling overheads by using *adaptable damped reservoir sampling* [6] to manage the transition time data that powers its CDFs. Rather than obtaining a timestamp for every event transition, Dendrite samples a subset of the event transition times. For each `record_event()` call, Dendrite 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 that can be stored in the reservoir (*reservoir size*) and k is the number of times Dendrite has observed e_1 so far. Thus, as Dendrite 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. Dendrite avoids excessive sampling overheads by reducing the sampling probability once it has obtained enough samples. Replacing an element at random enables Dendrite to keep older transition times around rather than just the most recent samples if it employed a least recently used policy instead. The reservoirs for each thread corresponding to the $e_1 \rightarrow e_2$ transition are written to disk alongside their respective transition counters and combined to produce CDFs of transition times. This strategy gives a fuller picture of transition times.

To produce an empirical CDF for a given reservoir, Dendrite uses the `numpy` analysis library's linear approximation technique [64]. The reservoirs' size (i.e., the number of stored samples) determines the accuracy of the estimated CDFs and the memory used for tracking. As the reservoir size increases, the CDF's accuracy increases, as does memory consumption. Therefore, the reservoir size is chosen by accounting for theoretical guarantees on CDF accuracy and considering the associated memory usage trade-off.

Theoretical Guarantees

Assume (as in previous work [91]) that the distribution of latencies for each event transition follows an exponential distribution $\exp(\lambda)$ for some $\lambda > 0$.[1]

[1] This assumption is validated through empirical accuracy results in Chapter 3.5.7.

Let $\mathbf{X}_1, \dots, \mathbf{X}_n$ be n independent and identically distributed samples from $\exp(\lambda)$. The mean of the $\exp(\lambda)$ distribution is $\mu = 1/\lambda$.

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

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

Proof. Each sample X_i has expectation value $\mathbf{E}[\mathbf{X}_i] = \mu$ and variance $\mathbf{Var}[\mathbf{X}_i] = \mu^2$ since it is drawn from $\exp(\lambda)$.

Let $\bar{X} \overset{\text{def}}{=} \frac{1}{n}\sum_{i=1}^n X_i$. Then:

$$\begin{aligned}
\mathbf{E}[\bar{X}] &= \mathbf{E}[\frac{1}{n}\sum_{i=1}^n X_i] \\
&= \frac{1}{n}\mathbf{E}[\sum_{i=1}^n X_i] \\
&= \frac{1}{n}n\mu \\
&= \mu
\end{aligned}$$

And:

$$\begin{aligned}
\mathbf{Var}[\bar{X}] &= \mathbf{Var}[\frac{1}{n}\sum_{i=1}^n X_i] \\
&= \frac{1}{n^2}\mathbf{Var}[\sum_{i=1}^n X_i] \\
&= \frac{1}{n^2}n\mu^2 \\
&= \frac{\mu^2}{n}
\end{aligned}$$

Chebyshev's inequality [113] states that for a random variable X with mean μ and non-zero variance σ^2:

$$\mathbf{Pr}\left[|X-\mu| \geq k\sigma\right] \leq \frac{1}{k^2}$$

Since we desire $(1-\varepsilon)\mu \leq \hat{\mu} \leq (1+\varepsilon)\mu$, we want $|\hat{\mu}-\mu| \leq \varepsilon\mu$. So, setting $k\sigma = \varepsilon\mu$, we find:

$$\begin{aligned}
k\sigma &= \varepsilon\mu \\
k &= \frac{\varepsilon\mu}{\sigma} \\
k &= \frac{\varepsilon\mu}{\sqrt{\sigma^2}} \\
k &= \frac{\varepsilon\mu}{\sqrt{\frac{\mu^2}{n}}} \\
k &= \frac{\varepsilon\mu}{\frac{\mu}{\sqrt{n}}} \\
k &= \sqrt{n}\varepsilon \\
k &= \sqrt{\frac{4}{\varepsilon^2}}\varepsilon \\
k &= 2
\end{aligned}$$

So, by Chebyshev's inequality:

$$\mathbf{Pr}\left[|\bar{X} - \mu| \geq \mu\varepsilon\right] \leq \frac{1}{4}$$

which concludes the proof. □

Next, bounds are placed on the maximum error ε for the learned approximation of $\exp(\lambda)$.

The total variation distance between two probability distributions p and q that share the same sample space Ω is defined as:

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

where sup is the supremum (or least upper bound).

Lemma 2. *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. As our estimate $\hat{\mu}$ is within a multiplicative factor of $(1+2\varepsilon)$ of the true μ value:

$$\mu/(1+2\varepsilon) \leq \hat{\mu} \leq (1+2\varepsilon)\mu$$

$$\lambda/(1+2\varepsilon) \leq \hat{\lambda} \leq (1+2\varepsilon)\lambda$$

The Kullback-Leibler divergence [73] for two probability distributions P and Q with density functions $p(x)$ and $q(x)$ is defined as:

$$\mathrm{dist}_{KL}(P,Q) = \int_{-\infty}^{\infty} p(x) log(\frac{p(x)}{q(x)})dx$$

Recall that:

$$\exp(x;\lambda) = \lambda \exp^{-\lambda x}$$

So:

$$\mathrm{dist}_{KL}(exp(x,\lambda), exp(x,\hat{\lambda})) = \int_{-\infty}^{\infty} \lambda \exp^{-\lambda x} log(\frac{\lambda \exp^{-\lambda x}}{\hat{\lambda}\exp^{-\hat{\lambda} x}})dx$$

$$= \int_{-\infty}^{\infty} \lambda \exp^{-\lambda x} log(\frac{\lambda}{\hat{\lambda}} \cdot \exp^{-\lambda x} \cdot \frac{1}{\exp^{-\hat{\lambda} x}})dx$$

$$= \int_{-\infty}^{\infty} \lambda \exp^{-\lambda x} \big(log(\frac{\lambda}{\hat{\lambda}}) + log(\exp^{-\lambda x}) - log(\exp^{-\hat{\lambda} x})\big)dx$$

$$= \int_{-\infty}^{\infty} \lambda \exp^{-\lambda x} \big(log(\frac{\lambda}{\hat{\lambda}}) + (\hat{\lambda}x - \lambda x)\big)dx$$

$$= log(\frac{\lambda}{\hat{\lambda}}) \int_{-\infty}^{\infty} \lambda \exp^{-\lambda x} dx + (\hat{\lambda} - \lambda) \int_{-\infty}^{\infty} \lambda \exp^{-\lambda x} x\, dx$$

Note that $\int_{-\infty}^{\infty} \lambda \exp^{-\lambda x} dx = 1$, since it is a probability distribution. Moreover,

$$\int_{-\infty}^{\infty} \lambda \exp^{-\lambda x} x dx = \mathbf{E}[\exp(\lambda)] = \frac{1}{\lambda}$$

So:

$$= log(\frac{\lambda}{\hat{\lambda}}) + \frac{\hat{\lambda} - \lambda}{\lambda}$$

$$= log(\lambda) - log(\hat{\lambda}) + \frac{\hat{\lambda}}{\lambda} - 1$$

$$= -1(log(\hat{\lambda}) - log(\lambda)) + \frac{\hat{\lambda}}{\lambda} - 1$$

$$= -log(\frac{\hat{\lambda}}{\lambda}) + \frac{\hat{\lambda}}{\lambda} - 1$$

Let $k = \frac{\hat{\lambda}}{\lambda} - 1$. Then:

$$= k - log(k+1)$$

Conducting a Taylor Series expansion of $\log(k+1)$, we find that:

$$\log(k+1) \approx \sum_{n=0}^{\infty} \frac{f^{(n)}}{n!} k^n = 0 + k - \frac{k^2}{2} + \dots$$

So:

$$\leq k - (k + \frac{k^2}{2})$$
$$\leq \frac{k^2}{2}$$

Expanding out:

$$\leq \frac{(\frac{\hat{\lambda}}{\lambda} - 1)^2}{2}$$

Since $\frac{\hat{\lambda}}{\lambda} \leq (1 + 2\varepsilon)$, we have:

$$\leq \frac{(1 + 2\varepsilon - 1)^2}{2}$$
$$\leq \frac{(2\varepsilon)^2}{2}$$
$$\leq 2\varepsilon^2$$

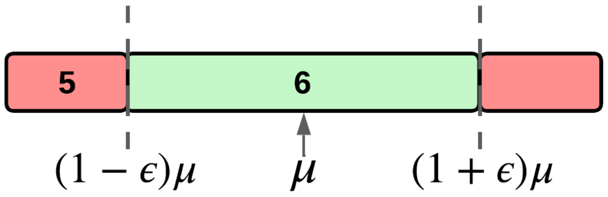


Figure 3.8: $(1-\varepsilon)\mu \le \hat{\mu} \le (1+\varepsilon)\mu$ with probability 3/4, per Lemma 1. If more than half of our estimates $\hat{\mu}$ are within ε of μ, then the median $\hat{\mu}$ is within ε of μ.

Pinsker's inequality [116] states that:

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

So:

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

concluding the proof. □

Combining the above lemmas reveals that with repeated experiments, at most ε error with $1-\delta$ probability for any $\delta > 0$ can be guaranteed.

Theorem 1. *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 $1-\delta$, it outputs a probability distribution $\hat{p}$ where $\text{dist}_{TV}(\hat{p}, \exp(\lambda)) \le \varepsilon$.*

Proof. Lemma 1 says that $(1-\varepsilon)\mu \le \hat{\mu} \le (1+\varepsilon)\mu$ with probability 3/4 when using $\frac{4}{\varepsilon^2}$ samples. Hence, there is a small probability that the estimate $\hat{\mu}$ will be outside of the acceptable range.

Suppose the estimation procedure for μ is repeated k times. If more than $k/2$ of the estimates $\hat{\mu}$ fall within ε of the true μ value, then the median necessarily falls within ε as well. This argument is presented visually in Figure 3.8. The worst case is that all of the unacceptable estimates for μ fall on one side of the acceptable range. Assume without loss of generality that they are below $(1-\varepsilon)\mu$ — here, five of the 11 total estimates. However, since more than half (six) of the estimates are in the acceptable range, the median is also within the acceptable range.

Let $X_1, X_2, \ldots, X_k$ be independent random variables where $X_i = 1$ if the i^{th} estimate $\hat{\mu}$ is within ε of μ and 0 otherwise. Then if $\sum_{i=1}^{k} X_i > 0.5k$, the median $\hat{\mu}$ value is within ε of μ.

Hoeffding's inequality [63] states that:

$$\mathbf{Pr}[|\sum_{i=1}^{k} X_i - \psi| \geq t] \leq 2\exp^{\frac{-2t^2}{\sum_{i=1}^{k}(b_i - a_i)^2}}$$

where $\psi = \mathbf{E}[\sum_{i=1}^{k} X_i]$ and a_i and b_i are the lower and upper bounds of each X_i variable. Since the range is [0,1], we have:

$$\mathbf{Pr}[|\sum_{i=1} X_i - \psi| \geq t] \leq 2\exp^{\frac{-2t^2}{k}}$$

We want:

$$\mathbf{Pr}[\sum_{i=1}^{k} X_i \geq 0.5k] \geq \delta$$

Simplifying the left:

$$\mathbf{Pr}[\sum_{i=1}^{k} X_i \geq 0.5k] = \mathbf{Pr}[\sum_{i=1}^{k} X_i - 0.75k \geq -0.25k]$$

$$\leq \mathbf{Pr}[|\sum_{i=1}^{k} X_i - 0.75k| \geq 0.25k]$$

Choose $\psi = \mathbf{E}[\sum_{i=1}^{k} X_i] = 0.75k$, so $k = \psi/0.75$. Thus:

$$= \mathbf{Pr}[|\sum_{i=1}^{k} X_i - \psi| \geq \frac{0.25\psi}{0.75}]$$

Applying Hoeffding's inequality with $t = \frac{\psi}{3}$ gives us:

$$\mathbf{Pr}[|\sum_{i=1} X_i - \psi| \geq \frac{\psi}{3}] \leq 2\exp^{\frac{-2(0.75)k^2}{9k}}$$

$$\leq c \exp^{-k}$$

for $c \in \mathbb{R}$.

Finding k:

$$\delta \leq c \exp^{-k}$$

$$\Rightarrow \frac{1}{\delta} \leq c' \exp^{k}$$

$$\Rightarrow O(log(\frac{1}{\delta})) \leq k$$

Thus, repeating the estimation procedure $O(log(\frac{1}{\delta}))$ times for μ yields a $\hat{\mu}$ value within ε of the true μ value with probability $1 - \delta$. Since $(1-\varepsilon)\mu \leq \hat{\mu} \leq (1+\varepsilon)\mu$ is a tighter bound than $\mu/(1+2\varepsilon) \leq \hat{\mu} \leq (1+2\varepsilon)\mu$ when $0 < \varepsilon \leq \frac{1}{2}$ and $\mu > 0$, applying Lemma 2 constrains the total variation distance as required.

□

Concretely, these theoretical results show that Dendrite's estimated probability distributions approximate the true distribution within a total variation distance of $\varepsilon = 0.1$ with more than 90% probability, given its default reservoir size of 1000. Using this reservoir size for each unique event transition in PostgreSQL in the experiments consumes only 1.5 MB per thread.[2] Therefore, Dendrite uses this reservoir size for all experiments. Chapter 3.5.7 shows the effects of reservoir size on empirical CDF accuracy.

3.3.3 Combining Transition Time CDFs

Dendrite's empirical CDFs provide granular timing information about a *single* transition. Database system functionality often spans multiple event transitions. For example, PostgreSQL's buffer page accesses span buffer pool cache hits, cache misses, dirty page flushes and disk reads (Figure 3.6). To obtain a complete picture of time spent in buffer accesses, one must account for the likelihood of transitioning between all of these events *and* their corresponding transition time CDFs. Dendrite supports developers by enabling them

[2]PostgreSQL 9.6 emits 90 unique events and 150 unique transitions.

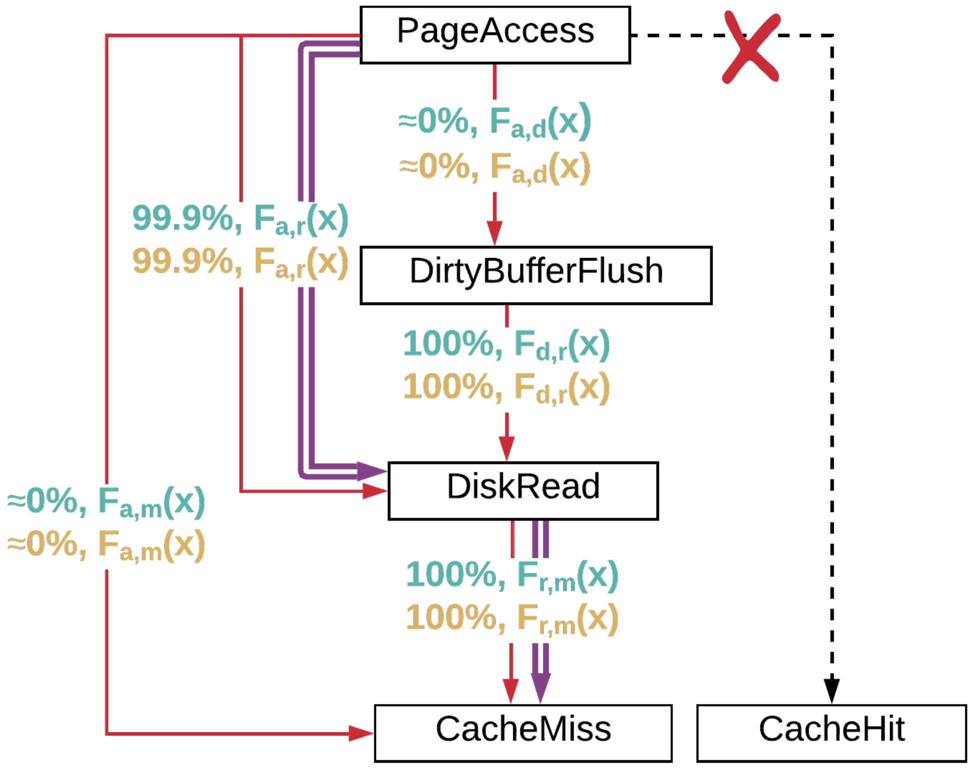


Figure 3.9: Combining empirical CDFs using random walks for Scenario A (blue) and Scenario B (orange) using a renormalized subset of the page access transition graph. Crossed-out paths indicate those pruned from consideration. Double purple lines indicate the random walk example in the text. $F_{e_1,e_2}(x)$ are CDFs for percentile x of transition times from event e_1 to event e_2.

to combine transition CDFs and thus explore the performance characteristics of complex system functionality (Algorithm 1). This functionality is particularly desirable when investigating further after receiving Dendrite's behaviour difference report. For example, suppose Dendrite has reported an increase in the time to transition from the DiskRead event to the CacheMiss event (corresponding to the time to read a page from disk); a developer can then use this feature to determine the effect on overall page access time, a particular code path of interest, or query performance more broadly. Dendrite 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.

Algorithm 1 combine_cdf((V,E), S, T, τ, n)

Input: (V,E) are vertices/edges in a model's transition graph, S is the start event vertex, T is a set of terminal events, τ is the probability cut-off threshold, n is the number of random walks

Output: Unified transition time CDF from event S to an event in T.

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

To demonstrate how Dendrite combines CDFs, consider the example in Figure 3.9, where it computes the total page access latency CDF for a buffer pool cache miss. The initial node for the random walk is the start of a buffer page access (PageAccess), and the terminal node is a buffer pool cache miss completion event (CacheMiss). In the first phase, Dendrite conducts a bounded depth-first search from the start node to the end nodes to find paths that lead to the end node (Algorithm 1, line 1). As Dendrite conducts the depth-first search (Algorithm 2), it computes the probability of arriving at each node from the start node using the tracked transition and event frequency counts. If the probability reaches a lower-bound threshold (line 4), then Dendrite 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). That is, if a node has i unpruned paths with probabilities $p_1, p_2, \ldots, p_i$, then $p_i = \frac{p_i}{\sum_j p_j}$. After pruning the divergent paths from the transition graph for the buffer pool miss event, Dendrite constructs the renormalized transition graph shown in Figure 3.9.

In the second phase, Dendrite conducts random walks from the start node to terminal nodes (Algorithm 1, line 5). The next transition is chosen at each step using the renormalized transition probability. For example, from the PageAccess start node in Figure 3.9, Dendrite takes the transition to the DiskRead node in both scenarios with nearly 100% probability. Dendrite computes the probability of taking the path at each step and terminates the walk when it reaches the minimum probability threshold or a terminal node.

Algorithm 2 bounded_dfs(N, T, (V,E), p, τ)

Input: N is the current event, T is a set of terminal events, (V,E) are vertices/edges in the model's transition graph, p is the probability of getting to N from the start node, τ is the probability cut-off threshold

Output: The set of edges that can lead from N to T with probability $\geq \tau$

```
1: E' = $\emptyset$
2: for (N, v2, p2) $\in$ E do
3:     pn = p * p2
4:     if pn >= $\tau$ then
5:         if v2 $\in$ T then
6:             E' = E' $\cup$ (N, v2, p2)
7:         else
8:             next_edges = bounded_dfs(v2, T, (V,E), pn, $\tau$)
9:             if next_edges != $\emptyset$ then
10:                E' = E' $\cup$ (N, v2, p2) $\cup$ next_edges
11:            end if
12:        end if
13:    end if
14: end for
15: return E'
```

Assume without loss of generality that Dendrite has taken the transition to the DiskRead event and then to the CacheMiss completion event for one of the random walks. Then at each step, Dendrite samples the CDF of each transition taken and adds the sampled value to a running total. Thus, Dendrite will sample the $F_{a,r}(x)$ distribution and the $F_{r,m}(x)$ distribution (Figure 3.9) and add those results to obtain the cumulative transition time for the random walk.

Dendrite conducts multiple random walks, using each of the returned cumulative times to estimate the overall CDF of transition times from the start to terminal events. The number of random walks and the accuracy of the underlying CDFs determine the combined CDF's accuracy. It follows from Theorem 1 that a constructed CDF with maximum path length from a source node l has a total variation of at most ε from the true CDF, given that each CDF along the path has a total variation of at most $\frac{\varepsilon}{l}$.

Algorithm 3 random_walk(N, T, (V,E), t)

Input: N is the current event, T is a set of terminal events, (V,E') are vertices/edges in the transition graph that lead to terminal events (Algorithm 2), t is the current total transition time.

Output: A sample of the time to transition from the start event to an event in T

```
1: r = random_float(0, 1)
2: for (N, v2, p)∈ E' do
3:     r = r - p
4:     if r < 0 then
5:         tn = sample_cdf(N, v2)
6:         t = t + tn
7:         if v2 ∈ T then
8:             return t
9:         else
10:            return random_walk(v2, T, (V,E'), t)
11:        end if
12:    end if
13: end for
```

3.3.4 CDF Differences Report

Measuring how event transition latencies have changed between workloads and system configurations is often desirable. 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. Dendrite provides this functionality by reporting the largest CDF transition time differences.

Dendrite quantifies the differences in CDFs p and q using the earth-mover's distance between them [96]. 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. That is:

$$EMD(p,q) = \frac{\sum_{i=1}^{100}\sum_{j=1}^{100} f_{i,j} d_{i,j}}{\sum_{i=1}^{100}\sum_{j=1}^{100} f_{i,j}} \tag{3.1}$$

where $\mathbf{F} = [f_{i,j}]$ defines the optimal flow of probability mass from percentile i in p to percentile j in q (computed by solving the min-cost-flow problem [96]), and $\mathbf{D} = [d_{i,j}] =$

$|i-j|$ defines the distance (and cost) of moving one unit of probability mass from percentile i in p to percentile j in q.

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

3.4 Difference Detection

The in-memory tracer and control server (Figure 3.1) work together to enable Dendrite's behaviour difference detection.

The in-memory tracer extracts per-process, per-thread behaviour models using the event and transition tracing approaches described in Chapters 3.2 and 3.3. Periodically, and when the database system being traced shuts down, these behaviour models are written into separate files.

Asynchronously, Dendrite's background aggregation thread sweeps through these model files and combines them into an overall model of system behaviour. To do so, it sums the counts for every event and event transition across all of the model files, subsampling values in the CDF reservoirs according to the frequency contributed by each model. For example, if model m_1 has observed event transition $e_1 \rightarrow e_2$ 40 times and m_2 has observed it 20 times with a fixed-size reservoir of 10 samples, then the combined reservoir for $e_1 \rightarrow e_2$ will have 7 samples from m_1 and 3 from m_2 because m_1 comprises $\frac{2}{3}$ of the total samples. Since this model merging process is performed in the background using simple counter additions and lightweight subsampling, it is efficient. The combined model of system behaviour is then loaded into a model database for later analysis.

Dendrite's control server provides a comprehensive suite of tools for developers and administrators to compare models and explore system behaviour differences. Given two behaviour models that correspond to an expected baseline and a situation of interest, the control server compares them using the difference and ranking procedures described in Chapters 3.2 and 3.3. It outputs a three-part report containing the previously mentioned event differences report, transition differences report, and CDF differences report. Each subreport is sorted by the largest contributing differences. These subreports are available in text or through the control server's intuitive user interface.

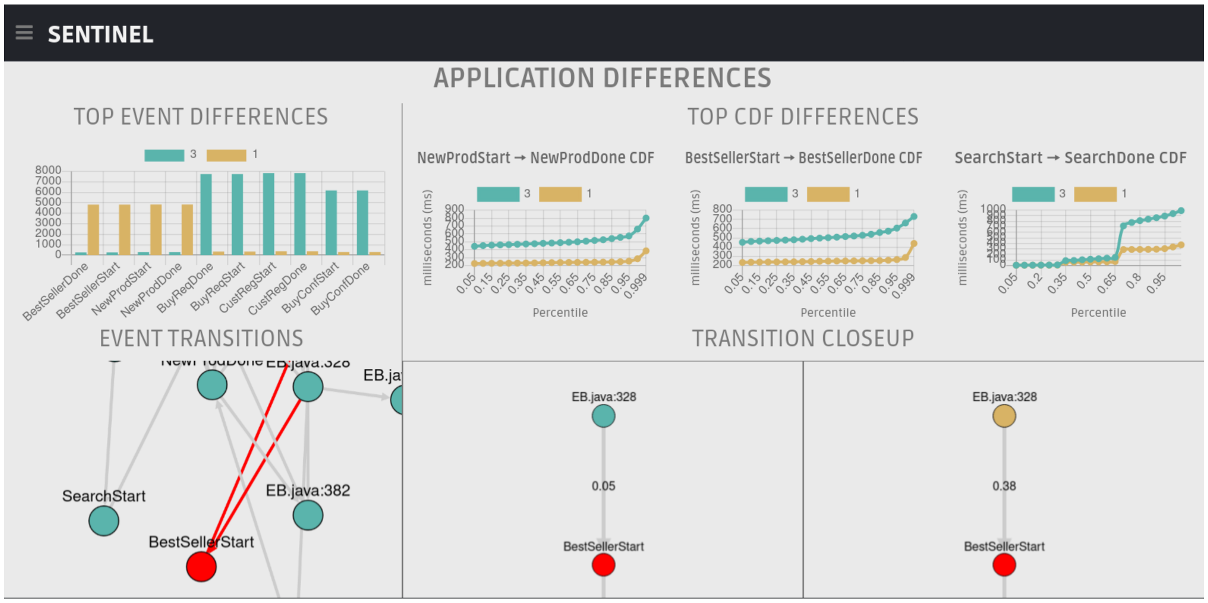


Figure 3.10: Dendrite's difference detection interface.

Beyond these tools' investigative value, the core techniques that power them are extended in the following chapter into an online difference detection and adaption deployment framework.

3.4.1 Difference Monitoring User Interface

The control server's user interface is implemented as a web application that operates over the model database. The interface consists of four logical components (Figure 3.10): the top reported event differences (upper left); the largest CDF differences (upper right); the event transition graph that shows which events transition to the current event of interest, and which events are likely to follow (bottom left); and an event transition comparison for the current event between the two models that presents execution flow differences (bottom right).

Each of these components employs brushing and linking [71]; an interaction with one component affects what is shown in the other, thereby helping users to explore behaviour differences effectively. For example, when the user selects an event in the top event differences component, the transition graph will pan and zoom to the event of interest and the transition comparison panes will adjust to display the event and its neighbours in the

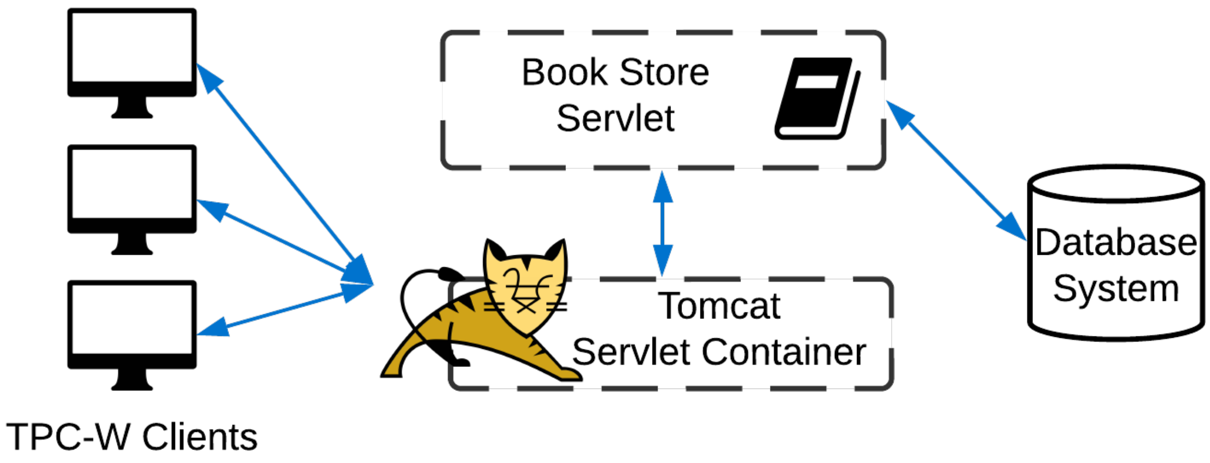


Figure 3.11: The experiment architecture used to evaluate Dendrite's difference detection.

transition graph. This feature lets users quickly find the most significant ranked differences and 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 behaviour models. Furthermore, Dendrite'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 [36]. This rendering results in clusters of nodes corresponding to different behaviour aspects (e.g., checkpointing, vacuuming, or buffer/query management). These aspects are illustrated through representative examples in a demonstration video highlighting Dendrite's difference detection capabilities [50].

3.5 Experimental Evaluation

The behaviour modelling and difference detection techniques presented in this chapter are fundamental building blocks for the extensions and adaption framework presented in the next chapter. Therefore, I now present a robust empirical evaluation of Dendrite's modelling and detection capabilities that demonstrates their utility, efficiency, and applicability.

3.5.1 Experiment Setup

To demonstrate Dendrite's *cross-system* effectiveness, I integrated it with a typical 3-tiered system consisting of a database system (PostgreSQL or SQLite), TPC-W benchmark clients, and Apache Tomcat (web server) (Figure 3.11). These components work together to execute the popular TPC-W benchmark [115].

TPC-W is a transactional web benchmark simulating a book store e-commerce environment. Clients submit HTTP requests to a web server, which issues requests in turn to a database system that stores persistent application state. The information in the database is used to generate web page responses to client requests.

I used the popular University of Minho implementation of TPC-W [97]. This implementation deploys Apache Tomcat, a Java servlet container, as the web server. The book store application is implemented as a Java servlet that Tomcat executes. Hence, the TPC-W clients issue requests to Tomcat, which forwards the requests to the book store servlet. The book store servlet interacts with the database system as necessary to implement the search, browsing, and product order features necessary for the benchmark.

I integrated Dendrite with each of the benchmark's system components: the benchmark clients, Apache Tomcat, the book store servlet, and the database system (PostgreSQL or SQLite). These components vary in implementation language (Java vs. C++), functionality (client vs. server), and complexity (application vs. database). Hence, the following experiment results demonstrate Dendrite's cross-system difference detection effectiveness and low-effort integration.

The benchmark machines are configured with 32 GB of main memory, 12 CPU cores with hyperthreading enabled, and an 800 GB HDD. The experiments use 10 concurrent clients with no waiting (i.e., think time) between transactions.

3.5.2 Evaluation Methodology

I contrasted Dendrite with state-of-the-art approaches for system behaviour analysis in terms of precision, performance overheads, analysis time, and ease of integration.

Distalyzer [90] is a log analysis tool that describes differences in log files that correspond to normal and abnormal system executions. Distalyzer describes statistically significant differences in log message counts, timestamps at which messages are emitted and logged variable values. Unlike Dendrite, Distalyzer requires that logs be written to disk and preprocessed offline using system-specific scripts before analysis. I used domain knowledge of the systems integrated with Distalyzer to develop scripts that extract important variables and latencies from the log files, which Dendrite does not require.

DBSherlock [125] is a state-of-the-art database monitoring and anomaly detection tool. It extracts metrics from system-specific sources (e.g., PostgreSQL statistics collector), the operating system, and debug log files. It generates predicates over these metrics (e.g., `dbCurLockWaits > 5`) that predominantly hold for the period of anomalous performance but not under normal performance. I ported DBSherlock to other database systems by mapping the MySQL metrics it relies on to corresponding statistics in the other databases (where available). As DBSherlock is designed for database systems, I consider only its ability to locate behaviour differences in PostgreSQL and SQLite.

By default, these experiments use PostgreSQL 9.6 as the database system. However, I also showcase Dendrite's difference detection generality within the database systems domain by highlighting behaviour differences in SQLite 3.31.1 behaviour in Chapter 3.5.3.

3.5.3 Behaviour Difference Validation

To assess each analysis tool's ability to surface relevant and useful behaviour insights, I studied controlled scenarios in which a single system characteristic is adjusted (e.g., lock contention, query execution time, transaction mix) and determined whether each tool reports the expected behaviour change corresponding to this difference. Table 3.1 shows the nine scenarios I studied, along with the variations I used to test precision.

In each scenario, Dendrite, Distalyzer, and DBSherlock compare the behaviour of the integrated system on the baseline TPC-W workload to the same workload but with the relevant change induced (test workload). To ensure an apples-to-apples comparison, I provided Distalyzer with the same level of logging information as Dendrite by configuring the database system, benchmark clients, and Apache Tomcat to write all log messages to disk, which comes at the cost of significant performance degradation (Chapter 3.5.4). As Distalyzer uses populations of log files to derive its insights, each of its tests relies 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. I used Distalyzer's absolute total difference to rank its reported differences, as in [90].

For DBSherlock, I monitored metrics every second (per [125]) for the baseline configuration and compared it to metrics recorded for the test configuration. I labelled the metrics recorded during the test configuration as anomalous and the metrics during the baseline

Table 3.1: Scenarios used to evaluate Dendrite, Distalyzer and DBSherlock's ability to pinpoint behaviour 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

configuration as normal. Hence, DBSherlock outputs predicates over these metrics that hold for the test configuration but not for the baseline configuration. I 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.

While each analysis tool has a different output format, they all output a ranked list

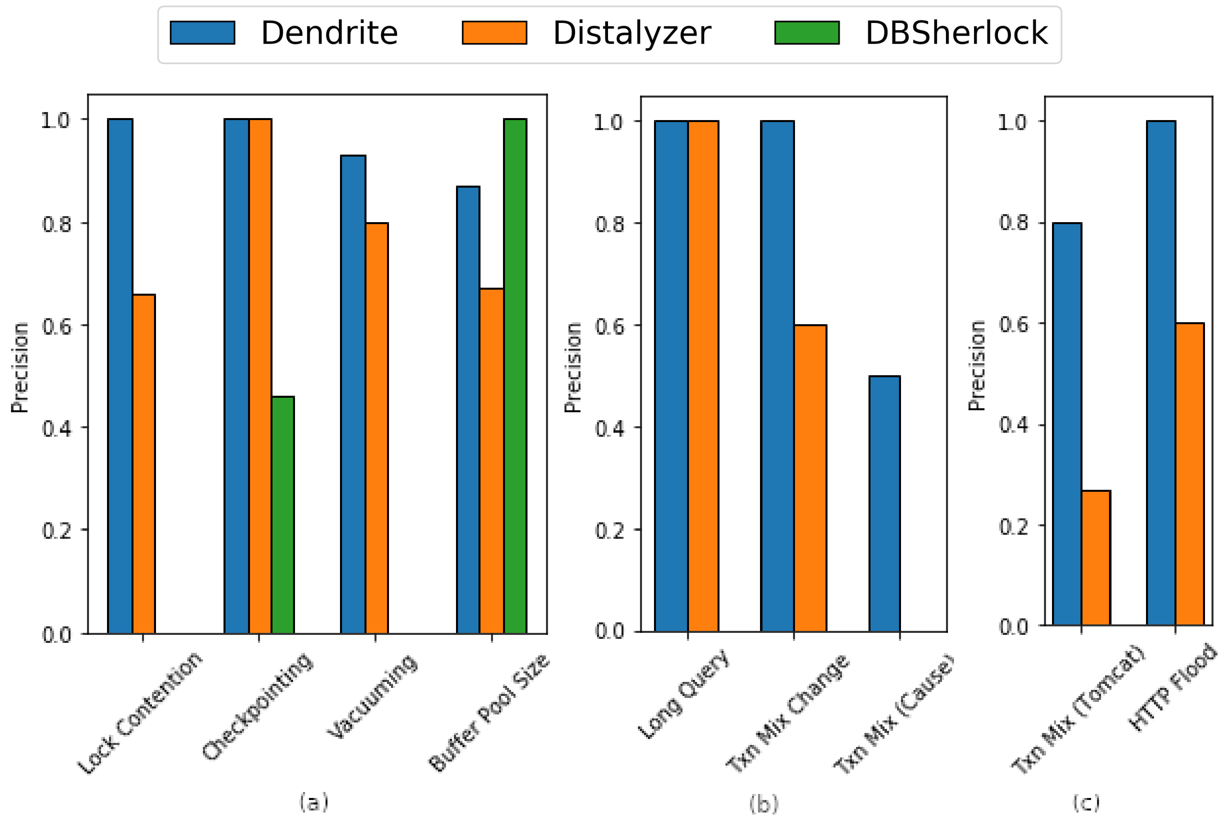


Figure 3.12: Precision graphs for Dendrite, Distalyzer, and DBSherlock's ability to pinpoint differences in (a) PostgreSQL, (b) TPC-W benchmark client, and (c) Apache Tomcat execution behaviour.

of system behaviour differences. I compared the top 3 ranked differences in an outputted category and determined if the information surfaced with them was indicative of the behaviour change. For example, in the lock contention scenario, I looked for a DBSherlock predicate or Dendrite/Distalyzer event that indicated more locking or increased lock time within the ranked list, considering such a test successful. I repeated each test three times and considered five variations of each test case. Precision results for each tool are computed by dividing the number of correct test cases in each scenario over the number of test cases (Figure 3.12). Next, I discuss each result in turn.

PostgreSQL Behaviour

I integrated Dendrite and Distalyzer with PostgreSQL 9.6 and enhanced its logging by configuring relevant built-in DTrace hooks to emit logging information. I induced changes in PostgreSQL's execution behaviour and evaluated Dendrite, Distalyzer, and DBSherlock's ability to detect these differences (Figure 3.12a). Unless otherwise stated, these experiments use the default PostgreSQL configuration with appropriate values for the buffer pool and operating system cache (8 GB and 16 GB, respectively) and a 50 GB TPC-W database.

Lock Contention: I introduced additional lock contention in the TPC-W BuyConfirm transaction by holding exclusive locks for longer. Dendrite reports differences in lock wait event proportion in *all* of the test cases, demonstrating its ability to surface relevant behaviour insights. Distalyzer identifies lock contention in most cases but is susceptible to reporting spurious differences in irrelevant events that occur at different times in the baseline and test configuration (e.g., checkpointing, autovacuum). DBSherlock does not accurately detect lock contention in PostgreSQL because PostgreSQL 9.6 does not keep a running tally of lock conflicts. I approximated this statistic by polling the number of queries that are blocked on locks, but this approximation does not capture all lock conflicts.

Aggressive Checkpointing: I decreased the checkpoint interval from PostgreSQL's default (5 minutes) and compared system behaviour to the default configuration. Both Dendrite and Distalyzer correctly identify differences in checkpoint events in all test cases. Distalyzer performs comparably with Dendrite 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. I accommodated an increase in page flush metrics 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, I compared PostgreSQL's behaviour when using its default autovacuum interval (1 minute) and with decreased autovacuum intervals. Both Dendrite and Distalyzer effectively pinpoint differences in autovacuum events, though Dendrite's precision remains higher. Again, Distalyzer's susceptibility to event occurrence timings affects its precision, an issue from which Dendrite 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: I decreased PostgreSQL's allocated buffer pool size to induce additional buffer pool cache misses. Dendrite 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 is less accurate than Dendrite. 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.

These results show that Dendrite is highly accurate at pinpointing relevant behaviour changes compared to the other approaches. Unlike Dendrite, Distalyzer's ranking scores hinder its precision because they are heavily affected by *when* events occur. DBSherlock's accuracy suffers because its ranking prioritizes predicates over metrics that hold for the test configuration but not for the baseline, which can happen spuriously. For example, if the operating system had a different number of files open during the test period than what was observed in the baseline, a predicate describing this unimportant difference was ranked highly. Dendrite avoids this pitfall as it pinpoints the most significant differences in behaviour between the configurations and not predicates that separate values of metrics in one configuration from another.

Client Application Behaviour

I now consider changes in application behaviour and evaluate Dendrite and Distalyzer's ability to highlight these differences (Figure 3.12b). I did not evaluate DBSherlock on benchmark client or web server behaviour due to its specialization for databases and reliance on a priori knowledge of which system metrics to extract. By contrast, both Dendrite and Distalyzer use information that is available through debug logging.

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

Transaction Mix Change: I 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. Dendrite 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, but not the change in transaction mix directly.

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

Web Server Behaviour

To demonstrate Dendrite's generalizability to a wide range of systems, I also integrated Dendrite and Distalyzer with Apache Tomcat version 9.0.3, a popular open source Java servlet container and web server. As in the other environments, I developed a custom preprocessing script to enable Distalyzer to extract events and timing information from Tomcat's log files, which Dendrite does not require. Precision results are shown in Figure 3.12c.

Transaction Mix Change (Tomcat): For the transaction mix change experiment above, I further evaluated whether we could determine this change in access patterns using *only* models of the web server's behaviour. Dendrite is highly accurate at pinpointing the behaviour change, which is emitted from per-transaction servlet logging. In particular, it notes that clients' web page access patterns differ significantly when the change is induced. 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 demonstrates Dendrite's generalizability across systems and its resilience to system complexity.

HTTP Flood: I simulated an HTTP flood attack [114] by rapidly issuing HTTP GET requests to Tomcat while running the TPC-W browsing mix. I tested the analysis tools' sensitivity to reporting these events by inserting varying think times between each

GET request. Dendrite 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 behaviour difference, it highlights differences correlated with the attack (e.g., session management) or changes in event timing. These results show that Dendrite's techniques also apply to security-focused behaviour exploration on data systems.

SQLite Behaviour

To further demonstrate Dendrite's generality, I integrated it with SQLite, a popular embedded SQL database (Figure 3.13a). I enabled SQLite's debug logging statements by adjusting its compiler flags and configured it to use Dendrite's in-memory tracer instead of writing logs to the console. For Distalyzer, I configured SQLite to emit these logs to disk and developed a preprocessing script to extract relevant features from them. I provided metrics for DBSherlock by using SQLite's `sqlite3_(db)status` functions. As these metrics do not cover all the test case functionality, I also advantaged DBSherlock by providing extra information obtained by outputting and preprocessing only the relevant SQLite log messages (e.g., checkpoints). I configured SQLite to use a write-ahead log and used the default configuration unless otherwise mentioned. As SQLite is an embedded database, I used a 1 GB TPC-W database.

Lock Contention: SQLite supports only a single concurrent writer; concurrent updates either block or are handled by a retry busy-loop. Therefore, I replicated the PostgreSQL lock contention test by creating a connection pool of database connections for concurrent readers and a single database writer connection. I added logging statements to the connection pool code and made this information available to each analysis system.

Dendrite and Distalyzer obtain high accuracy on this test as obtaining the writer connection is the main bottleneck and is therefore highly reported in Dendrite's transition time CDF differences 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 I adjusted the test case values for this scenario to accommodate this difference (Table 3.1, default 1000 frames). Dendrite obtains 100% precision on this test case because even small changes in checkpoint frequency greatly affect their overall event count proportion, which Dendrite detects. Although Distalyzer is not as accurate as Dendrite, 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.

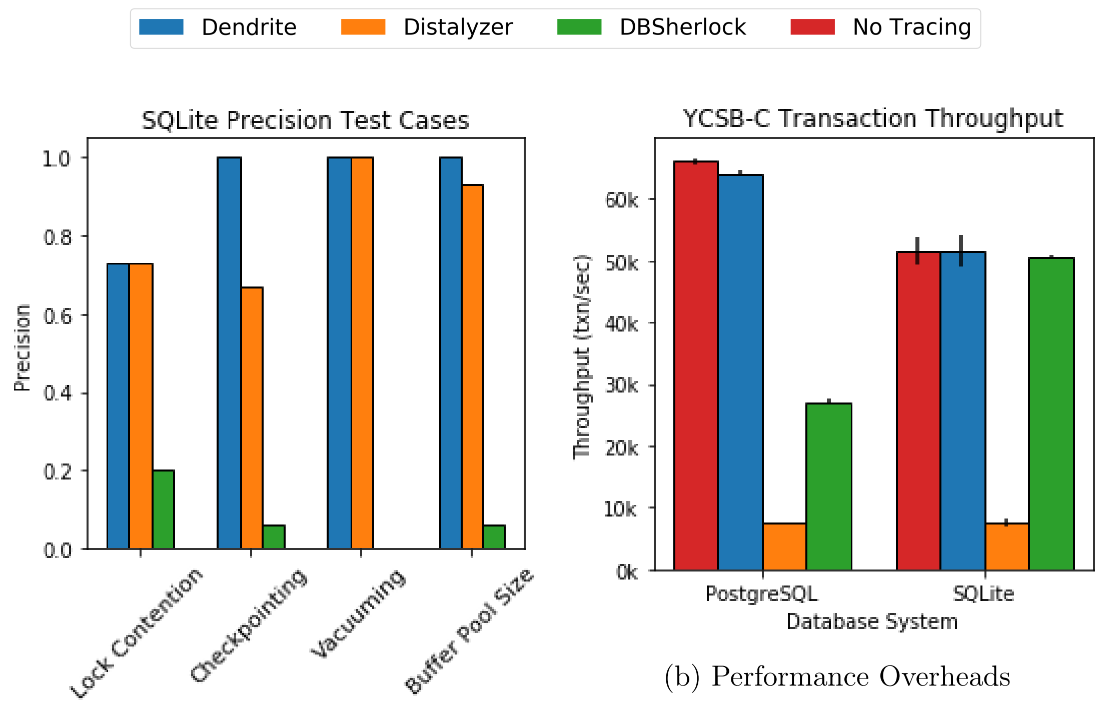
Aggressive Vacuuming: Clients issue `PRAGMA incremental_vacuum` commands to SQLite to trigger vacuuming. Therefore, I configured the write connection to submit this command after every k^{th} committed transaction for varying values of k (Table 3.1, defaulting to every transaction, as in SQLite's full vacuum mode). Both Dendrite and Distalyzer obtain perfect precision for this test for the same reasons they perform 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. I accommodated this behaviour by reducing the buffer pool sizes I used in this test compared to the values we used for PostgreSQL (Table 3.1, default 10 MB). Dendrite 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 ineffective on this test case because SQLite's memory consumption grows to meet the buffer pool size. DBSherlock's ranking, therefore, prioritizes other metrics.

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

3.5.4 Monitoring Overheads

To assess the performance overheads of Dendrite, Distalyzer and DBSherlock, I used the YCSB-C benchmark [22]. The YCSB-C workload exclusively uses single record lookups by primary key, transferring workload processing bottlenecks from transaction execution to the debug logging and monitoring of the analysis tools. I executed this workload against PostgreSQL and SQLite for 5 minutes using OLTPBench [35] and measured the throughput. Figure 3.13b shows the average throughput over three experiments for each database-analysis tool pair, along with their 95% confidence intervals.



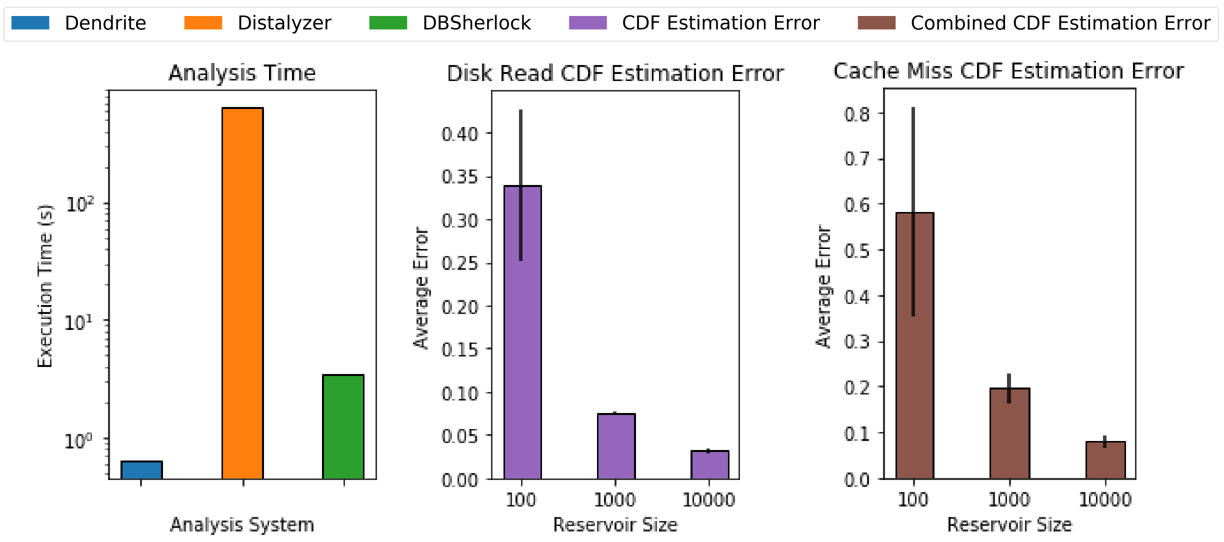
(a) SQLite Precision Results

(b) Performance Overheads

Figure 3.13: SQLite precision results and YCSB-C throughput.

Observe that Dendrite's monitoring reduces the transaction throughput of PostgreSQL and SQLite by less than 3%, while Distalyzer imposes large performance penalties of 90% and 85%, respectively. These overheads are induced by the detailed and costly debug logging it requires, whereas Dendrite intercepts logging calls in memory and does not need log messages to be materialized on disk. Similarly, Dendrite improves throughput over DBSherlock by 60% on PostgreSQL because DBSherlock requires some logging to complement its obtained system metrics. On the SQLite database, the throughput discrepancies are 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, when combined with those from the prior section, demonstrate that Dendrite obtains the highest difference detection precision with the lowest overhead.

To further understand Dendrite's overheads, I conducted an ablation study in which I measured the throughput of the database systems while Dendrite tracked (*i*) only event counts, (*ii*) events and transition counts, and (*iii*) with all tracking enabled. I observed



(a) Analysis times (PostgreSQL) (b) Single CDF Accuracy (c) Combined CDF Accuracy

Figure 3.14: Analysis times for each of the analysis tools and Dendrite's CDF accuracies.

that event count tracking resulted in only 1.5% of the overhead, enabling transition count tracking incurred a scant additional overhead of 0.5%, and enabling the remaining tracking functionality in Dendrite added only 1% overhead.

3.5.5 Analysis Time

I now contrast Dendrite, Distalyzer, and DBSherlock in terms of the time they take to analyze results from two workloads or system configurations. I average the time it takes to compare PostgreSQL behaviour for the 10 ms variation of the lock contention scenario to the baseline configuration (Chapter 3.5.3). Results are shown in Figure 3.14a.

Distalyzer's analysis time far exceeds that of DBSherlock and Dendrite due to the large size of its preprocessed log files for each test ($\approx$ 4 GB). By contrast, Dendrite's and DBSherlock's analysis phases use summaries of system execution behaviour to determine behaviour differences, which requires much less I/O and computation time. However, Dendrite's analysis phase takes only 1/5 the time of DBSherlock's, a testament to Dendrite's efficiency while yielding useful results (Chapter 3.5.3). This low analysis time enables administrators to identify and respond to system changes rapidly.

Table 3.2: Analysis of system integration efforts.

	Dendrite	Distalyzer	DBSherlock
PostgreSQL Lines of Code Changed	53	156	159
SQLite Lines of Code Changed	74	317	194
TPC-W Lines of Code Changed	13	61	N/A
Tomcat Lines of Code Changed	25	189	N/A

3.5.6 System Integration Efforts

I estimate the efforts of integrating Dendrite, Distalyzer and DBSherlock with PostgreSQL, SQLite, TPC-W benchmark clients, and the Apache Tomcat web server in Table 3.2. Integrating Dendrite with these data systems requires less effort than the other approaches, which I quantify using the lines of code (LoC) changed during integration.

Both Distalyzer and DBSherlock require system-specific preprocessing scripts, while Dendrite does not. Distalyzer's preprocessing scripts transform 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 Distalyzer-interpretable format, these scripts often require many lines of code to implement (Table 3.2). Similarly, DBSherlock's preprocessing scripts take the form of customized `dstat` plugins or targeted modifications to the 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, Dendrite requires instrumenting only the logging library and wrapping thread logic to output behaviour models before terminating, thus requiring fewer code changes to integrate.

The architecture of a system influences the complexity of integrating it with an analysis tool. For example, integrating Dendrite 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 unnecessary for PostgreSQL. As these aspects also increase the complexity of integrating with Distalyzer and DBSherlock, Dendrite's integration efforts remain the lowest in all cases.

3.5.7 Accuracy of Sampled CDFs

I assessed Dendrite's accuracy in estimating individual CDFs and those generated using its random walk technique. First, I determined the CDF accuracy for a single transition using three different reservoir sizes (Figure 3.14b). I 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, I observed a high estimation error of 35%, while increasing the size to 1000 reduces the error to below 10%. While further increases to the reservoir size marginally improve accuracy, returns on the additional memory consumed diminish significantly when using reservoir sizes above 1000.

I next considered the accuracy of CDFs constructed via random walks by computing buffer miss latency CDFs (recall Figure 3.9) and comparing them to the true CDF (Figure 3.14c). As above, I considered various reservoir sizes and averaged the errors at every 5^{th} percentile after 1 million random walks. Again, I observed that the error decreases significantly from 60% to 20% when increasing reservoir size from 100 to 1000 and that 10000 samples reduce it further to 8%. Given these accuracy and memory trade-offs, Dendrite uses reservoirs of size 1000 in the above experiments.

These results complement the theoretical results (Chapter 3.3.2) as the theory bounds total variation in probability while the empirical results measure differences in latency at given percentiles. Combined, these results show that Dendrite's CDF estimation techniques are effective and accurate.

3.6 Summary and Discussion

This chapter presented Dendrite's core behaviour extraction, modelling, and difference detection techniques. In brief, Dendrite intercepts debug logging calls made by the database system and encodes the frequency and execution control flow of these logging events into Markov chain based behaviour models. Dendrite compares the extracted models to determine how behaviour has shifted in terms of event proportion, event transition probabilities, and cumulative distribution functions of event transition times.

In Chapter 3.5, I showed the effectiveness of Dendrite's difference detection techniques on a variety of popular data systems in representative scenarios. The results demonstrate that Dendrite's difference detection has higher precision than existing approaches, and that it achieves these results with low overhead.

Despite these results, a few challenges remain to make Dendrite a generalizable adaptivity framework. This chapter presented model extraction and behaviour comparison techniques for post-hoc analysis, but adaption frameworks operate online. The next chapter enhances Dendrite to extract models from live database systems and perform online behaviour difference detection. Online difference detection enables rapid responses to system behaviour changes, which is infeasible with offline analysis. I also discuss Dendrite's adaption rule framework that codifies and deploys responses to behaviour differences; this framework supports a wide variety of adaptive use cases while remaining intuitive to use.

Lastly, the behaviour models used in this chapter cannot capture all behaviour differences. In particular, they do not encode resource-consumption metrics — for example, memory consumption or network bandwidth utilization. Determining when database system resources are nearing exhaustion and responding appropriately comprises an important class of adaption scenarios. Moreover, these models do not handle cases where event control flow is dependent on more than one preceding event, precluding it from disambiguating the contexts in which an event occurs. This drawback leads to event-relationship conflation when, for example, DirtyPageFlush events occur in a loop in PostgreSQL's checkpointer but not in its page access logic. In the following chapter, I present enhancements to these core techniques, along with Dendrite's adaption rule framework, that address the above challenges.

Chapter 4

Enhanced Modelling and Adaptivity Framework

In the last chapter, I described the core behaviour modelling and difference detection techniques that power the Dendrite system. While these techniques are effective in pinpointing behaviour differences in system events and transitions, they cannot attribute system resource consumption to system code paths, a critical feature for understanding overall system performance and improving it. For example, the prior modelling approach cannot be used to determine if a database system's performance has degraded because disk I/O bandwidth is saturated.

A chief objective of Dendrite is to tailor database system processing in response to workload or environmental changes, such as the changes in transaction mixes shown in the last chapter. After registering an expected baseline of system behaviour, Dendrite could employ the techniques described in the previous chapter to determine when behaviour has shifted and output a behaviour differences report. However, an administrator would need to digest the report, use their domain knowledge to decipher the root cause of the behaviour difference, and address the underlying issue. Relying on administrators to effect changes increases their heavy system-maintenance burden. Hence, these techniques alone do not meet Dendrite's stated goals of a bolt-on, generalized adaptivity framework.

In this chapter, I present enhancements to Dendrite's behaviour models that encode fine-grained resource consumption metrics. I also describe Dendrite's online adaptivity framework, which uses the behaviour models enhanced by this chapter's techniques to detect behaviour differences that warrant system augmentation and automatically deploy changes to improve performance.

4.1 Model Enhancement Motivation

Before explaining how Dendrite improves upon the previous chapter's behaviour modelling technique, I will describe its limitations in more detail. The improvements in this chapter address two key shortcomings: resource-consumption tracking and higher-dimensional modelling.

4.1.1 Resource Consumption and Behaviour Differences

Database systems are resource-intensive. Traditional disk-based database systems rely heavily on disk bandwidth to store data, access it quickly, and process requests transactionally. A portion of the system's stored data is cached in memory to avoid expensive disk accesses, posing a significant main memory requirement. As these database systems often receive many concurrent client requests, they also induce high network traffic. Processing complex queries also requires a powerful CPU to meet stringent client latency requirements.

If any of these system resources are exhausted, performance will degrade. Requests for resources may be queued or denied, increasing client request latency. Therefore, administrators carefully monitor the resource usage of their database systems to ensure the resources needed to meet client demands are available. When these requirements cannot be satisfied, techniques such as re-routing and deferred processing can be used to reduce resource requirements.

Detecting and remedying resource-contention situations are common tasks for system administrators. Dendrite addresses this administrative burden by capturing system resource consumption and enabling automatic, consumption-based responses to behaviour differences. Automated responses to these scenarios are valuable because every second of downtime and performance reduction becomes expensive for companies [4, 42, 77].

Resource consumption can also indicate the importance of a system behaviour change and whether a reaction is warranted. For example, suppose Dendrite has detected that a database system it is monitoring has encountered a sudden increase in client load. If the system can handle the increased demand given the available resources, then no adaption is required. On the other hand, if query latency significantly increases due to the additional load, then Dendrite may elect to deploy a load-shedding strategy to ameliorate it.

It is common to capture system resource consumption in aggregate, using operating system or database system counters. For example, administrators may use `dstat` or `top` to determine the resource consumption of various system processes and threads. However, finer-grained metrics are required to support the types of adaptions outlined in Dendrite's objectives. Returning to the running example of a load spike, if the system's performance has degraded due to contention on disk bandwidth, it is necessary to determine which areas of the system's code are responsible for the contention. If background processes are executing that contribute to disk I/O saturation, such as a checkpoint or background page flushing, then throttling those processes may alleviate the contention. If these processes are not active, throttling them will be ineffective.

To this end, I enrich Dendrite's behaviour models to support fine-grained resource consumption attribution. These enhanced models report the total resources consumed for each resource type and which parts of the system's code are consuming them. In line with Dendrite's goals of generalizability, this extraction process is fully system-agnostic. The following section describes how Dendrite meets these objectives.

4.1.2 Higher-Dimensional Modelling

The modelling techniques described in the previous chapter exploit the memoryless property of Markov models to reduce complexity. In particular, they assume that the probability of moving to a subsequent event e' is conditional only on the current event e (i.e., none of the previous events). This assumption of conditional independence may not hold for all event transitions in practice. While the prior models are effective for behaviour difference detection, behaviour models powering adaption frameworks must eliminate this assumption to ensure appropriate responses.

As a concrete example, consider a DirtyPageFlush event in PostgreSQL. This event occurs for one of three main reasons: (i) a modified page held in the buffer pool must be evicted from memory and written to disk to make room for another page's data, (ii) PostgreSQL's background writer process (`bgwriter`) is flushing page modifications to disk to reduce the overheads of a future checkpoint, or (iii) a checkpoint is taking place, which flushes a large number of dirty pages to disk.

In case (i), only a single page is flushed, after which transaction processing resumes as normal. In case (ii) and (ii), dirty pages will be flushed in a loop, resulting in a very different series of events than that of case (i). The previous modelling approach does not disambiguate these cases as its event transitions use only the immediately preceding event. When models from the background writer, checkpoint process, and transaction processing threads are merged, event transition probabilities will be combined and conflated. This conflation happens whenever a system event E appears in multiple code paths and these paths have different event-transition characteristics for E.

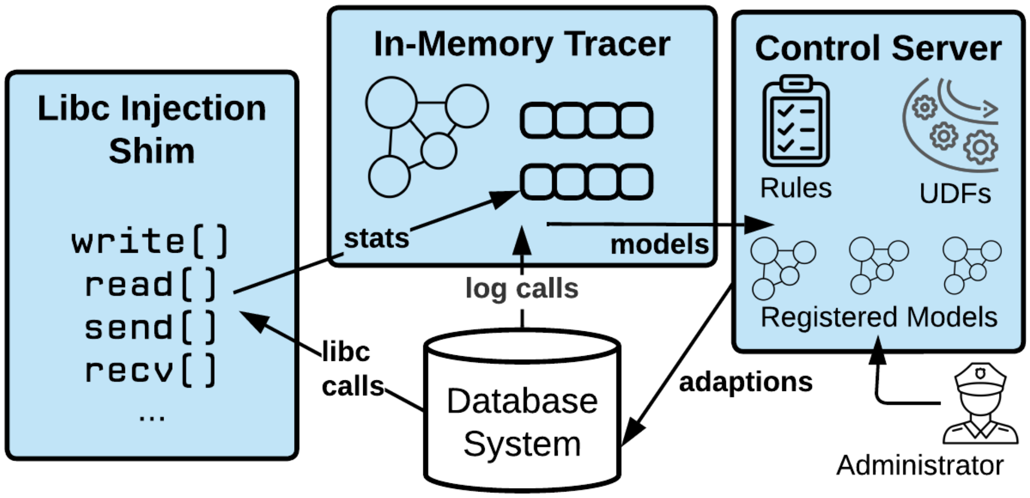


Figure 4.1: Dendrite's architecture when deployed in online adaption mode. Components from the previous chapter are repurposed and extended to support adaption-rule responses.

The correct response to a high number of dirty page flushes depends on the cause. Suppose the background writer is impairing performance through periodic page flushes. In that case, Dendrite can defer the background writes to a period of lower load to alleviate these effects, and similarly for a checkpoint. However, if the impetus for dirty page flushes is eviction during page reads, then the solution is to increase buffer pool size or background flushes to move costly page flushes off the main path of transaction execution. Dendrite's enhanced behaviour models disambiguate such situations, ensuring an appropriate adaption response.

4.2 Behaviour Model Enhancements

To support enhanced behaviour tracking and autonomous adaptions, Dendrite's architecture is augmented as shown in Figure 4.1. The in-memory tracer is extended to support higher-dimensional modelling and resource capturing by collaborating with the *injection shim*, described below. The control server periodically issues commands to the in-memory tracer, compelling it to write its per-thread models to disk. As before, a background thread aggregates the models into a combined behaviour model, but this model is now passed to the control server directly. The control server detects behaviour differences and responds to them using system adaptions. Presently, we restrict our focus to the injection shim and

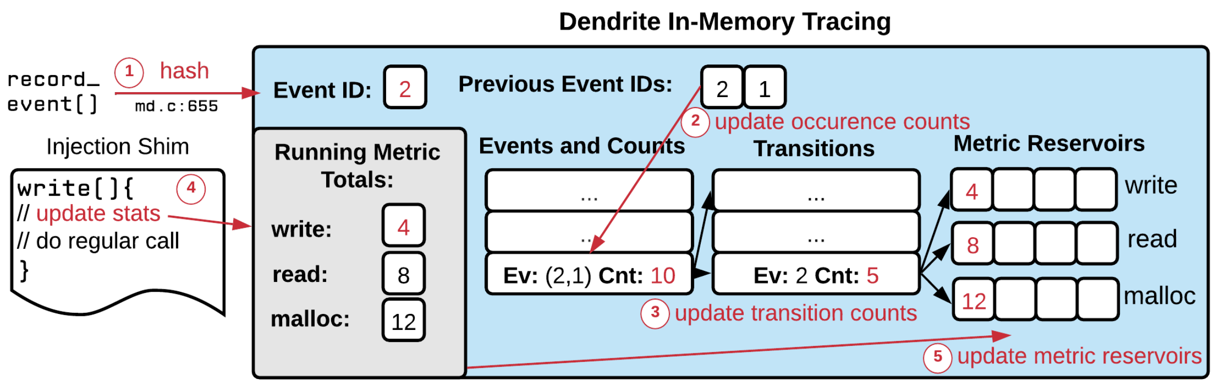


Figure 4.2: Dendrite's in-memory tracer handling an example `record_event` call.

the in-memory tracer; Chapter 4.3 covers the control server's enhanced functionality in detail.

Dendrite's enhanced behaviour models obtain their core information in a manner similar to the models presented in the previous chapter. The target database system's logging library is integrated with Dendrite by linking against the in-memory tracer and having the `log()` function call Dendrite's `record_event` function (recall Figure 3.4). However, `record_event` is modified to account for sequences of prior events.

As before, `record_event` obtains the file name and line number of the originating `log` call in source code (Figure 4.2). Dendrite hashes these values to determine a unique ID for the event. In this example, the log message from file `md.c` on line 655 (corresponding to a BufferReadDone event) obtains a hash of 2.

In contrast to the prior models, Dendrite obtains the event IDs of the k previous events (in Figure 4.2 example, $k = 2$), which are stored in a ring-buffer by the tracer. Dendrite hashes the previous event IDs (corresponding to DiskReadDone, DiskRead) together to determine an offset into an in-memory hash table that tracks the number of times those events have occurred consecutively. In Figure 4.2 ②, the event sequence (2,1) hashes to the third slot and its count is incremented to 10.

Afterward, Dendrite records that the prior event sequence has transitioned to the current event (Figure 4.2, ③); it increments the transition count from event sequence (2,1) to the current event ID 2. Dendrite captures how the database system moves between events by tracking event transitions. If transition patterns change (e.g., page accesses now lead to more cache hits), Dendrite recognizes this difference and responds appropriately.

Dendrite uses these event and transition counts to determine the most popular events and compute transition probabilities. In Figure 4.2, the probability of moving from event sequence (2,1) to event 2 is computed by dividing the transition count by the prior sequence frequency: $P(2|[2,1]) = \frac{5}{10}$. These enhanced event frequencies and transition probabilities are the core of Dendrite's behaviour models; next, I describe how Dendrite enriches them with resource metrics.

4.2.1 Fine-grained Resource Metric Collection

It is often desirable to deploy adaptions based on the resource usage of database system components and processes. For example, suppose a PostgreSQL database exhausts disk I/O resources due to checkpointing during heavy system load. In that case, postponing the checkpoint to a less workload-intensive period may be appropriate. By contrast, if disk resources are exhausted due to query processing, postponing checkpoints will have little impact on performance. To this end, Dendrite's behaviour models encode both aggregate and fine-grained system resource consumption metrics. Extracting these metrics while conforming to the outlined goals of generalizability and low overhead is challenging, requiring novel techniques.

To capture aggregate resource consumption information, Dendrite uses `dool`, a fork of the popular (now deprecated) `dstat` [120] resource monitor. `dool` is a Python 3 tool that obtains resource information from the `/proc` filesystem on Linux, outputting it to a comma-separated values (CSV) file for later analysis. Dendrite configures `dool` to output aggregate CPU utilization, memory consumption, network traffic, and disk I/O to file. As part of behaviour model merging (Chapter 3.2), the background process reads these metrics and incorporates them into the combined behaviour models.

Dendrite exploits dynamic linking to intercept targeted `libc` calls and capture their resource usage. On Linux systems[1], the `LD_PRELOAD` environment variable specifies shared object files to be loaded ahead of any others the executable needs. As the symbols in these files are resolved first, functions implemented within them override functions with the same name defined in the files loaded afterward. Dendrite uses this feature to override C library functions with custom implementations in a preloaded shared object file called the *injection shim*.

The injection shim contains custom implementations of `libc` functions (e.g., `pwrite`, `send`) that allocate or use system resources. These custom versions extract the `size` argument from the function call, which dictates how much data is to be operated on, adding

[1]Other operating systems like Windows and macOS support similar functionality.

Function Name	Description	Tracked Parameter and Type
`malloc(sz)`	Allocates `sz` bytes from the heap	Sum `sz`, allocation
`free(ptr)`	Frees `ptr` memory	count, deallocation
`read(fd,buf,sz)`	Reads `sz` bytes from file `fd` into buffer `buf`	Sum `sz`, disk read I/O
`write(fd,buf,sz)`	Write `sz` bytes into file `fd` from buffer `buf`	Sum `sz`, disk write I/O
`pwrite(fd,buf,sz,off)`	Write `sz` bytes into file `fd` at offset `off` from buffer `buf`	Sum `sz`, disk write I/O
`io_submit(ctx, n, iocb)`	Submit n asynchronous I/O calls found in `iocb` ptr.	Sum of I/O sizes in `iocb`, disk read/write I/O.
`send(fd,buf,sz,flags)`	Send `sz` bytes in buffer `buf` over socket `fd` using flags `flags`.	Sum `sz`, network write I/O
`recv(fd,buf,sz,flags)`	Receive `sz` bytes into buffer `buf` over socket `fd` using flags `flags`.	Sum `sz`, network read I/O

Figure 4.3: The C library functions Dendrite's injection shim overrides to track resource utilization.

it to a running metric total for the current event transition (Figure 4.2, ④). Afterward, they call the default `libc` implementation of the function, preserving the original system behaviour. When Dendrite receives the next `record_event` call, it stores the total in a fixed-size array (*reservoir*) for the given metric on the given transition (Figure 4.2, ⑤). Afterward, it sets the counters for each metric to zero so that resource consumption can be tabulated for the next event transition.

This resource tracking feature enables developers and administrators to determine which parts of their database system's code consume the most resources. As an example from Figure 4.2, they can determine that the transition from $(2, 1) \rightarrow 2$ reads 8 bytes, and map the event IDs back to source code locations ((`md.c:655`, `md.c:640`) $\rightarrow$ `md.c:655`) to determine the responsible code. Moreover, Dendrite's adaption rules can adapt system behaviour in response to targeted resource consumption — for example, a case study in Chapter 5.2.1 shows that Dendrite can detect excessive disk I/O *due to checkpointing* and then reduce this process' aggressiveness.

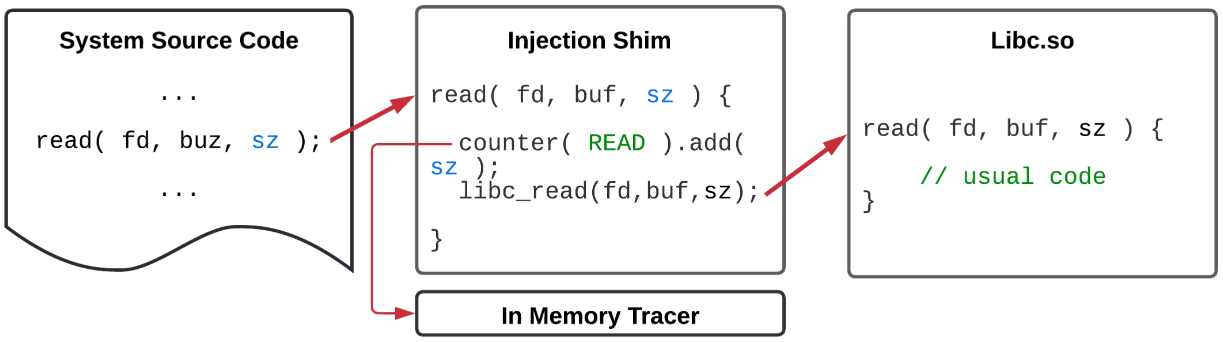


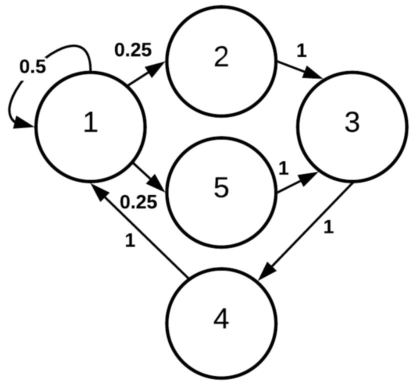
Figure 4.4: Dendrite's injection shim.

Figure 4.3 shows the functions the injection shim tracks by default. These functions were determined empirically by executing a variety of popular database systems and ensuring that Dendrite captured expected resource utilization. Note that not every function call is used by every database system — for example, PostgreSQL 14.1 uses `pwrite` to write data to disk, PostgreSQL 9.6 uses `write`, and MariaDB 10.5 uses asynchronous `io_submit` calls. Metrics are recorded as long as the system's resource-consuming calls are defined in the injection shim. If developers or administrators wish to capture additional classes of resource usage not shown in Figure 4.3, they can trivially add new interceptors to the shim. Doing so requires only a few lines of code.

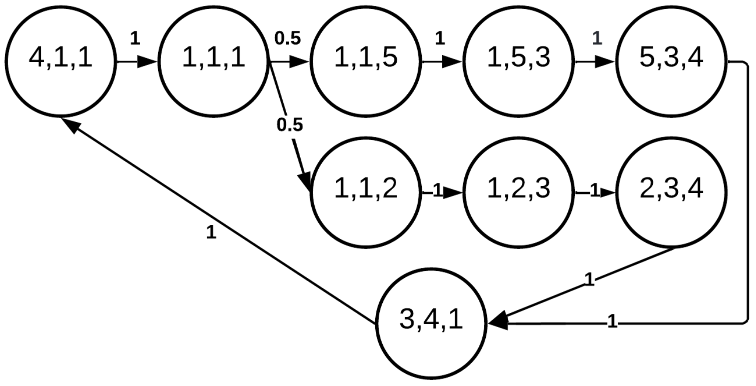
4.2.2 Minimizing Modelling Overheads

The higher-dimensional behaviour modelling techniques proposed in the previous sections assume a fixed constant k that defines the number of prior events to account for when computing transition probabilities. However, the choice of k presents trade-offs that must be carefully navigated.

The value of k affects the accuracy of Dendrite's behaviour models and their memory consumption. For example, setting $k = 1$ leads to a conflation in DirtyPageFlush transition probabilities because the models do not account for whether the events arose from a checkpoint/background write or an LRU page eviction (Chapter 4.1.2). On the other hand, setting $k = 2$ consumes more memory than required for any event transition that does not depend on two preceding events. In the worst case where every k-length sequence of n events transitions to each of the n events, an order-k model consumes $O(n^{k-1})$ more memory than an order-1 model.



a) Order-1 Markov Model



b) Order-3 Markov Model

Figure 4.5: Behaviour models for $k = 1$ and $k = 3$ capturing event sequence $1, 1, 1,\ 2, 3, 4,\ 1, 1, 1,\ 2, 3, 5 \ldots$

To demonstrate the trade-offs imposed by the parameter k, consider the following example. Suppose Dendrite observes the following sequence of events IDs: $(1, 1, 1, \{2 \text{ or } 5\}, 3, 4)$, repeating infinitely. Figure 4.5a shows the model constructed for $k = 1$, while Figure 4.5b shows the model for $k = 3$. The $k = 3$ model consumes more memory as it contains a node for each triplet of event occurrences, while the $k = 1$ model has only one node for each event type. Moreover, the $k = 3$ model contains nodes for both $(5, 3, 4)$ and $(2, 3, 4)$ transitioning to 1, even though the system always moves to event 1 after event 4 regardless of the events that came before — a relationship captured succinctly by the $k = 1$ model. However, the $k = 3$ model correctly captures that the sequence $(4, 1, 1)$ is always followed by 1 (Figure 4.5b), unlike the $k = 1$ model.

It would be ideal if nodes in the model were created only when necessary to capture event transition relationships. For example, the model in Figure 4.6 does not contain nodes to disambiguate $(1, 5, 3)$ from $(1, 2, 3)$ or $(5, 3, 4)$ from $(2, 3, 4)$ because the transitions from 3 or 4 do not depend on any prior events. Such a model would consume less memory and communicate event transition relationships more clearly.

Constructing this idealized model is challenging because Dendrite operates online. Dendrite discovers the transition independence relationships on-the-fly because it receives information about event executions only while the system is running. Moreover, Dendrite continuously refines its behaviour models, meaning that events that at one time appeared not to influence subsequent transitions may do so as Dendrite observes more of the database system's behaviour.

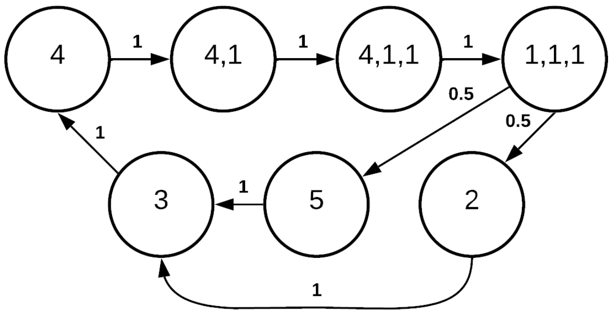


Figure 4.6: Variable-order behaviour model capturing event sequence $1, 1, 1,\ 2, 3, 4,\ 1, 1, 1,\ 2, 3, 5\ldots$

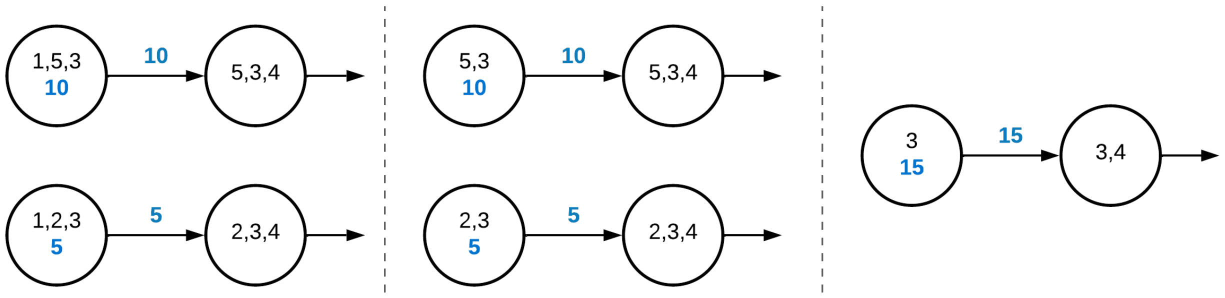


Figure 4.7: Steps in dynamically reducing an order-3 transition to order-1.

Dendrite dynamically determines how many prior events each transition depends upon and eliminates nodes accordingly (as in Figure 4.6). At the same time, it provides strong statistical guarantees when reducing sequence lengths to ensure its models remain accurate.

To achieve these objectives, Dendrite performs the following detection step when updating event transition counts while handling a `record_event` call (recall Figure 4.2, ③). Assume without loss of generality that the current k-length event sequence is $(E_1, E_2, \ldots, E_k)$, and the subsequent event is E_{k+1}. As before, Dendrite will update the counters for the occurrences of event sequence $(E_1, E_2, \ldots, E_k)$ and event transition $(E_1, E_2, \ldots, E_k) \rightarrow E_{k+1}$. Afterward, it will check if the transition probability of moving from $(E_1, E_2, \ldots, E_k)$ to any subsequent event E'_{k+1} is equivalent to the transition probability without accounting for E_1. In other words, it determines whether the transition's probability is independent of E_1:

$$P(E'_{k+1}|E_1, E_2, \ldots, E_{k-1}, E_k) = P(E'_{k+1}|E_2, \ldots, E_{k-1}, E_k) \tag{4.1}$$

As a concrete example, consider Figure 4.7, which follows our running example. The

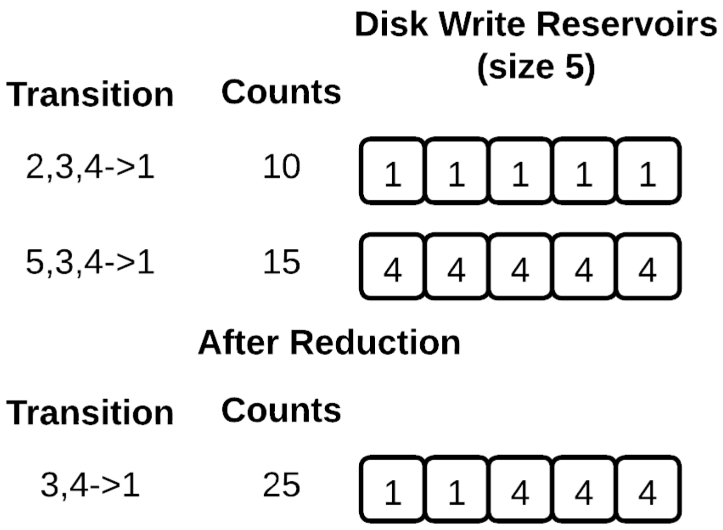


Figure 4.8: Merging prior event sequences and metric reservoirs.

blue numbers indicate event sequence or transition counts. Both the $(1, 5, 3)$ and $(1, 2, 3)$ event sequences transition to subsequent event 4 with 100% probability, yielding event sequences $(5, 3, 4)$ and $(2, 3, 4)$ respectively. Since $P(E_{k+1}|1, 5, 3) = P(E_{k+1}|5, 3)$, for all E_{k+1} to which $(1, 5, 3)$ transitions (here only 4), Dendrite reduces the sequence to the middle panel. The same holds for $(1, 2, 3)$, so the sequence length is reduced accordingly. On a subsequent transition count update for $(5, 3)$, Dendrite determines that $\forall E_{k+1}$, $P(E_{k+1}|5, 3) = P(E_{k+1}|3)$, so it reduces the sequence length again. To determine this fact, it finds all event sequences ending in 3 that transition to E_{k+1}, thereby accounting for the $(2, 3)$ path to 4. It then merges these sequences to produce a single event sequence, 3, that transitions to 4.

Merging event sequences requires careful consideration since Dendrite must account for the resource-consumption reservoirs. As each merged transition requires combining resource-consumption samples, Dendrite subsamples each of the original transitions' samples according to their proportion of the overall transition count. That is, if, as a consequence of reducing an event sequence length, Dendrite needs to merge two event sequences, A and B, and A transitions to E_{k+1} 100 times but B transitions only once, then the samples from A should be represented 100-fold more in the merged reservoir than the samples from B.

This merging procedure (Algorithm 4) is illustrated by merging two reservoirs for the disk write I/O metric in Figure 4.8. Suppose Dendrite is reducing prior event sequence

Algorithm 4 reduceEventSequenceLength(m, prior_seq)

```
Input: Behaviour model m, prior event sequence to reduce prior_seq
 1: reduced_seq = prior_seq[1:]
 2: seqs = m.findPriorSeqsEndingIn(reduced_seq)
 3: reduced_seq_count = 0
 4: next_ev_transition_counts = hashmap()
 5: for seq ∈ seqs do
 6:     reduced_seq_count += m.getCount(seq)
 7:     for ev ∈ m.eventsSeqTransitionsTo(seq) do
 8:         next_ev_transition_counts[ev] += m.getTrCount(seq, ev)
 9:     end for
10: end for
11: for seq ∈ seqs do
12:     for (ev, count) ∈ next_ev_transition_counts do
13:         prob = count/reduced_seq_count
14:         res = m.sampleReservoirs(seq, m.getTrCount(seq, ev), count)
15:         reduced_seq.addTransition(ev, prob, res)
16:     end for
17:     m.remove(seq)
18: end for
19: m.add(reduced_seq)
```

(2,3,4) to (3,4), as in the example (Figure 4.5b, 4.6). First, it finds all of the prior event sequences that have the suffix (3,4) — in the example, (2,3,4) and (5,3,4). Next, Dendrite computes the reduced sequence's frequency count and transition counts by iterating through the transitions for each of the found sequences (Algorithm 4, lines 5-10). In the example, it finds only one unique transition for each of the found sequences (to event ID 1), with a total count of 25. Next, Dendrite determines the transition probabilities for the reduced sequence by dividing these transition counts from the total reduced sequence count (line 13). There are 25 transitions to event ID 1 in the example, with a total count of 25 transitions, yielding a probability of 1.0. Finally, Dendrite subsamples the metric reservoirs for the found transitions according to the proportion of the transition total they comprise. Here, (2,3,4) → 1 makes up 2/5 of the total transitions, so 2/5 of each metric reservoir in the reduced sequence will be made up of samples from its reservoirs (Figure 4.8). The remaining samples come from $(5,3,4) \to 1$.

In practice, performing a simple equality check to verify independence before sequence reduction is insufficient. Dendrite's computed event transition probabilities are inherently estimates of the true event transition probabilties; each event transition it observes is a sample of the true event transition distribution. With few observations of an event sequence transition, Dendrite cannot be confident that its estimate of the transition's probability is accurate. As the number of observations increases, it becomes increasingly confident about the transition probability's true value.

As a concrete example, suppose Dendrite has captured 10 samples of event sequence (E_1, E_2, E_3) and observed that a particular transition $(E_1, E_2, E_3) \to E'$ occurred 5 times. Dendrite computes the maximum likelihood estimate $\hat{p}$ for the transition's probability, 0.5. But the true transition probability p may be 0.3 — Dendrite just happened to observe otherwise in its 10 samples. Dendrite exploits theoretical guarantees on the true probability's deviation from the maximum likelihood estimate, as shown in the subsequent section, to ensure that it is confident in its transition probability estimates before considering the associated transitions for merging.

As a further consideration, the number of samples involved in computing transition probabilities may affect whether Dendrite chooses to reduce sequence lengths if relying on strict-equality independence checks. For example, suppose that Dendrite wishes to reduce $(1, 2, 3)$ to $(2, 3)$, but the transition from $(1, 2, 3) \to 4$ has taken place $999/1000$ times, and $(2, 2, 3) \to 4$ has taken place $9991/10000$ times. As these probabilities marginally differ, the strict-equality procedure above would preclude these sequences from merging. However, it could be the case that the true transition probability from $(1, 2, 3) \to 4$ is 0.9991 — it simply cannot be represented with that level of precision using 1000 samples. This result is undesirable because it decreases the scope of sequence merging.

Dendrite overcomes these challenges by instead applying the following check before reducing a prior event sequence $E = (e_1, e_2, \ldots, e_k)$ to $E' = (e_2, \ldots, e_k)$:

$$\forall X, Y : |P(Y|e_1, e_2, \ldots, e_k) - P(Y|X, e_2, \ldots, e_k)| < \epsilon \tag{4.2}$$

with probability $1 - \delta$. That is, if all other event sequences with the same ending events have true transition probabilities within ϵ of each other with probability at least $1 - \delta$, then Dendrite merges the sequences together. Dendrite computes the number of samples required to satisfy $\epsilon, \delta = 0.05$ (by default), and simplifies the prior sequence for the transition only when this criterion is met. Next, I show how Dendrite computes this sample count.

4.2.3 Model Accuracy Guarantees

As Dendrite computes transition probabilities using *observed* frequencies of event sequences and transitions, these probabilities are necessarily estimates of the true transition probabilities and thus subject to error. Therefore, Dendrite quantifies the error bounds on its estimates based on the number of samples it has. Concretely, Dendrite computes the number of samples required to ensure that an estimated transition probability $\hat{p}$ is within ϵ of the true value p with probability $1 - \delta$.

Observe that the number of times prior event sequence $(e_1, e_2, ...e_k)$ transitions to subsequent event e' can be modelled by a binomial distribution with some true probability p. Thus, the *empirical estimator* for p is $\hat{p} \overset{\text{def}}{=} \frac{x}{m}$ where m is the number of times we have observed event sequence $(e_1, e_2, \ldots, e_k)$ and x is the number of times it has been followed by e'.

Lemma 3. *Fix $\epsilon > 0$, $\delta \in (0, 1)$. Then for $m \geq \frac{2+\epsilon}{\epsilon^2} ln(\frac{2}{\delta})$, it holds with probability $1 - \delta$ that $|\hat{p} - p| \leq \epsilon$.*

Proof. Let $X \sim Binomial(m, p)$ represent the probability distribution of the observed transition count. By applying the two-sided Chernoff inequality:

$$P(|x - E[X]| \geq \epsilon' E[X]) \leq 2\exp(-\frac{\epsilon'^2}{2+\epsilon'} E[X]) \text{ (for any } \epsilon' > 0)$$

Therefore,

$$P(|x - mp| \geq \epsilon' mp) = P(|\hat{p} - p| \geq \epsilon' p) \leq 2\exp(-\frac{\epsilon'^2}{2+\epsilon'} mp)$$

Choose $\epsilon' = \frac{\epsilon}{p}$, then

$$P(|\hat{p} - p| \geq \epsilon' p) \leq 2\exp(-\frac{\frac{\epsilon^2}{p^2}}{2+\frac{\epsilon}{p}} mp) = 2\exp(-\frac{\epsilon^2}{2p+\epsilon} m)$$

since $p \leq 1$, $\frac{\epsilon^2}{2p+\epsilon} \geq \frac{\epsilon^2}{2+\epsilon}$, so

$$P(|\hat{p} - p| \geq \epsilon) \leq 2\exp(-\frac{\epsilon^2}{2+\epsilon} m)$$

then since $m \geq \frac{2+\epsilon}{\epsilon^2} ln(\frac{2}{\delta})$

$$P(|\hat{p} - p| \geq \epsilon) \leq 2\exp(-\frac{\epsilon^2}{2+\epsilon} m) \leq \delta$$

□

Lemma 4. *If two independent event transition probabilities $\hat{p_1}, \hat{p_2}$ are within at most ϵ of their true probabilities p_1, p_2 respectively with probability $1 - \delta$ and $|\hat{p_1} - \hat{p_2}| \leq \epsilon$, then $|p_1 - p_2| \leq 3\epsilon$ with probability at least $(1 - \delta)^2$.*

Proof. By definition, $\hat{p_1} = p_1 + \Delta_{p_1}, p_2 = \hat{p_2} + \Delta_{p_2}$ where $|\Delta_{p_i}| \leq \epsilon$.
So, we have

$$|p_1 - p_2| = |\hat{p_1} + \Delta_{p_1} - \hat{p_2} - \Delta_{p_2}|$$

Using the triangle inequality:

$$|p_1 - p_2| \leq |\hat{p_1} - \hat{p_2}| + |\Delta_{p_2} - \Delta_{p_1}|$$

$$\Rightarrow |p_1 - p_2| \leq \epsilon + |\Delta_{p_2} - \Delta_{p_1}| \leq 3\epsilon$$

with probability $(1 - \delta)^2$. □

Thus, from Lemmas 3 and 4, we have the reduction rule:

$$\forall X, Y : |\hat{P}(Y|e_1, e_2, ..., e_k) - \hat{P}(Y|X, e_2, ..., e_k)| \leq \epsilon$$

$$\Rightarrow \forall X, Y : |P(Y|e_1, e_2, ..., e_k) - P(Y|X, e_2, ..., e_k)| \leq 3\epsilon$$

with probability $(1 - \delta)^2$.

Practically, these results mean that for Dendrite's default settings of $\epsilon, \delta = 0.05$, Dendrite ensures $P(Y|e_1, e_2, ..., e_k)$ differs from $P(Y|X, e_2, ..., e_k)$ by at most 0.15 with probability 90.25% by collecting at least 9075 samples before reducing prior event sequence length in its models.

4.2.4 Concurrency

Recall that Dendrite constructs its enhanced behaviour models on a per-process, per-thread basis. Each thread creates a behaviour model independently of the others, storing the counters and reservoirs in thread-local storage to avoid contention on shared data structures.

When a database system linked against Dendrite starts executing, the first thread to issue a logging call is responsible for setting up a network socket, a thread to monitor the socket, and shared-memory communications structures. The control server (discussed in the following section) periodically issues a request to this network socket, indicating that each thread should write its model to disk and begin construction of a new model for the

Algorithm 5 record_event(fname,line)

```
Input: File name and line number of the logging event (fname, line)
 1: if shm_ptr == NULL then
 2:    global_sem = sem_open("/lock.sem", O_CREAT, 1)
 3:    global_rdy_sem = sem_open("/rdy.sem", O_CREAT, 0)
 4:    if sem_trywait(global_sem) then
 5:       shmid = shmget(ftok( "shm_buff"))
 6:       shm_ptr = shmat(shmid, sizeof(ShmData), IPC_CREATE)
 7:       shm_ptr->epoch = 0
 8:       t_epoch = 0
 9:       create_socket_and_monitoring_thread()
10:       sem_post(global_rdy_sem, INT64_MAX)
11:    else
12:       sem_wait(global_rdy_sem)
13:       shmid = shmget(ftok("shm_buff"))
14:       shm_ptr = shmat(shmid, sizeof(ShmData))
15:       t_epoch = shm_ptr->epoch
16:       sem_post(global_rdy_sem, 1)
17:    end if
18: end if
19: if t_epoch < shm_ptr->epoch then
20:    dump_model()
21:    t_epoch = shm_ptr->epoch
22:    reset_model()
23: end if
24: update_counters(fname, line)
```

upcoming time period. The interval at which this command is sent is configurable (with a default of 30 seconds), and termed an *epoch*.

The setup process operates as follows. A `log()` call in turn executes `record_event()` (Algorithm 5). However, `record_event` will determine that the shared-memory control structures for the current thread have not yet been allocated and initialized (line 1). Therefore, it races with other threads to create a named SYSV semaphore in the `/dev/shm` filesystem called `lock.sem` with initial value 1 and a `rdy.sem` semaphore with value 0 (lines 2-3). This race is benign as only one thread will succeed in creating the semaphores — the other threads will open the created semaphores. Afterward, the threads race to acquire the lock semaphore (line 4). The thread that wins the race creates a named shared-memory segment (line 7), initializing an atomic `uint64_t` epoch counter. Each thread uses this counter to determine when to write their models to disk (line 8). It then creates the network socket and a monitoring thread, incrementing the ready semaphore to `INT64_MAX` (lines 9-10). The other threads block on the ready semaphore until this increment occurs (line 11), after which they will attach to the initialized shared memory structures (line 12). Note that this setup process is conducted only once per thread. On subsequent executions of `record_event` the shared-memory structures are detected, so execution proceeds directly to writing out models if necessary (line 19) and updating the thread-local event and transition counters as previously described (line 24).

When the monitoring thread receives the control server's command to output per-thread models to disk, it atomically increments the epoch counter in shared memory (`shm_ptr->epoch`). When other threads handle a subsequent `record_event` call, they check this counter and compare it against a thread-local copy of the counter's value (line 19). If the values of these copies differ, then the control server has advanced the epoch; the thread writes its current model to disk in a per-thread file (line 20), tagging the model with the epoch value stored in the local counter. It then updates its local counter to match the shared epoch counter (line 21). Finally, the thread deletes the memory for the prior model and constructs a new model (line 22), adding information for the event occurrence associated with the `record_event` call (line 24).

4.2.5 Combining Variable-Order Behaviour Models

After the per-thread models have been written to disk, Dendrite uses a background process to read the models for each thread within a given epoch. This process combines these models into a single model that describes overall system behaviour. Unlike the merge process described in the previous chapter, the enhanced models from different threads may now record information for the same event transition with different sequence lengths. For example, the transition $(E_1, E_2) \to E_3$ in one model may be represented as $(E_2) \to E_3$ in another. Dendrite's combined models account for these different representations and unify their information correctly.

Dendrite respects model differences by first transforming each thread's model to a fixed-order k model, where k is the maximum prior event sequence length present in any thread's model. This process is called *model expansion*.

Expanding a model m to order k from its existing order is done one order at a time. The first step is to find the smallest-order transition — the minimum length of a prior event sequence for any recorded transition in m (Algorithm 6, line 1). Next, Dendrite preprocesses the existing transitions, building a map from prior event sequences to their next events (line 5). Then, for each transition t in the existing model, Dendrite performs the following sequence of steps (line 6):

- If t is already of the desired order, it is copied over to the expanded model directly (line 10).
- Otherwise, Dendrite finds all other transitions that involve the prior event sequence of t. That is, if $t = (E_1, \ldots E_{i-1}, E_i) \to E'$, it finds all transitions $t2 \in m$ with prior event sequence $(E_1, \ldots, E_{i-1}, E_i)$ (line 15).
- It finds all of the events to which $(E_1, \ldots, E_i)$ transitions (line 16), e.g., E'.
- It determines all transitions in m that could lead to t. For example, transition $(E_0, E_1, \ldots, E_{i-1}) \to E_i$ leads to $(E_1, \ldots, E_{i-1}, E_i) \to E'$ because the the last portion of the former's prior event sequence appended with its transition matches the prior event sequence of the latter. Informally, one can interpret this relationship as "the former transition may be directly followed by the latter." Dendrite constructs the expanded event sequence by combining the former's prior event sequence and its transition into an expanded prior event sequence, recording it in `unique_prior_seqs` (line 18).
- For each of the (`next_event`, `unique_prior_seq`) pairs returned by the previous two steps, Dendrite generates an expanded transition (line 21).
- Dendrite determines the number of times the expanded prior event sequence occurred by summing the counts of transitions that would have led to it. Note that this is not the number of transitions from the expanded prior sequence to a particular `next_event`, but rather the number of times the sequence transitioned to *any*

Algorithm 6 expand_model_one_order(m)

```
Input: Behaviour model m
Output: Behaviour model with transitions of at least order min_order(m)+1
 1: min_order = min([t.order for t in m.transitions])
 2: transitions_to_skip = map()
 3: new_transitions = set()
 4: new_transitions_metrics = map()
 5: prior_index = build_index_from_priors()
 6: for t ∈ m.transitions do
 7:    if t ∈ transitions_to_skip then
 8:       continue
 9:    end if
10:    if  t.prior_events.length >= min_order + 1  then
11:       new_transitions[t] = m.transitions[t]
12:       new_transitions_metrics[t] = m.transitions_metrics[t]
13:       continue
14:    end if
15:    all_other_transitions =
          prior_index.find_all_transitions_with_same_priors(t)
16:    next_events = unique([t2.next() for t2 in all_other_transitions])
17:    transitions_that_end_in_start =
          [t2 for t2 in m.transitions if t2.could_transition_to(t)]
18:    unique_prior_seqs = unique(
          [t2.construct_before(t) for t2 in transitions_that_end_in_start] )
19:    for  next_event in next_events  do
20:       for  unique_prior_seq in unique_prior_seqs do
21:          expanded_transition = (unique_prior_seq) → next_ev
22:          total_transition_count = m.sum_matching_trs(
                transitions_that_end_in_start, unique_prior_seq)
23:          subsampled_metrics = m.subsample_reservoirs(
                transitions_that_end_in_start, unique_prior_seq)
24:          new_transitions[expanded_transition] = total_transition_count
25:          new_transitions_metrics[expanded_transition] = subsampled_metrics
26:          transitions_to_skip.add((t) → next_event)
27:       end for
28:    end for
29: end for
30: return  model(new_transitions, new_transitions_metrics)
```

subsequent event. Thus, Dendrite exploits its knowledge that sequence lengths are reduced only if the reduced sequence's transition probability for any transition is close (within ϵ with high probability $1-\delta$) to the original sequence's probability. Hence, the expanded sequence's probability must be close to that of the reduced sequence's probability. So, Dendrite multiplies the probability of the reduced sequence's transition to `next_event` by the total number of times the expanded prior event sequence occurred to determine the transition count (line 22).

- The same process as above is used to expand out metric reservoirs for all `next_event`, `unique_prior_seq` pairs, except that subsampling is applied to merge multiple reservoirs that contribute to an expanded transition (line 23).
- Since (`next_event`, `unique_prior_seq`) pairs account for all transitions that share t's prior event sequence, Dendrite records this fact and skips over them when processing subsequent transitions (line 26).
- Finally, Dendrite returns the model defined by transition counts and transition metrics.

The `expand_order_by_one` function is repeatedly executed until the desired model order is achieved. This process is applied to each per-thread model, after which the merge procedure described in the previous chapter is executed. Metric reservoirs are subsampled so that fixed-size reservoirs do not overflow. Once the models are combined, Dendrite applies the order reduction procedure over each transition to simplify the combined model. This merging procedure is performed in the background by the control server (described next) in a process external to the database system; thus, Dendrite avoids cross-thread synchronization on the main path of execution that would impair performance. The resulting combined model is the chief means of Dendrite's behaviour comparisons and adaptions.

4.3 Control Server and Enabling Adaptivity

Dendrite's in-memory tracer constructs comprehensive behaviour models in collaboration with the injection shim using the procedures described above. Dendrite's *control server* (Figure 4.9) retrieves behaviour models from the tracer when an epoch has completed, combines the models, and compares the combined model against a registered model

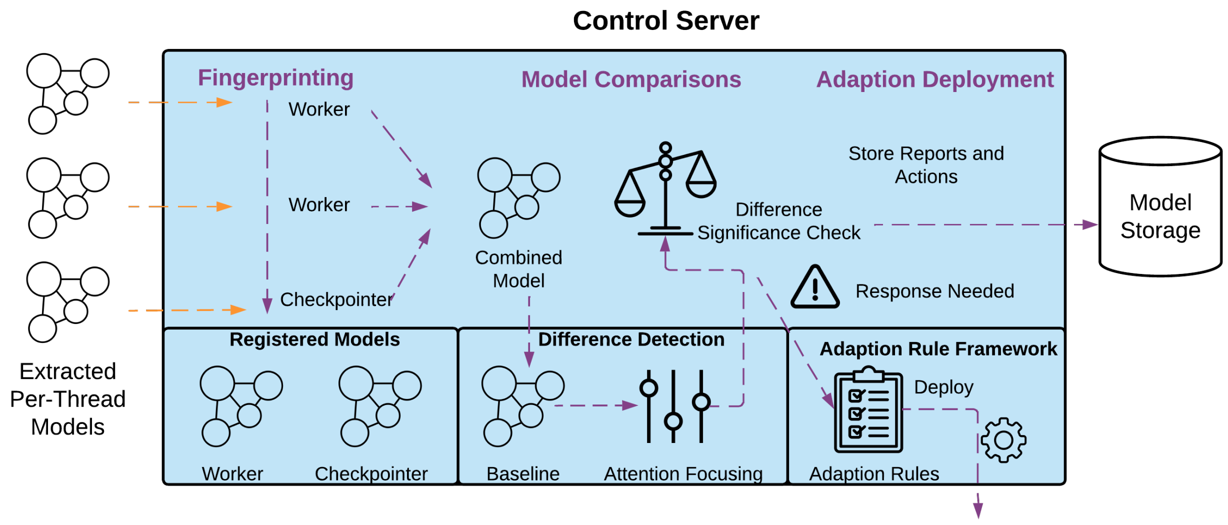


Figure 4.9: Dendrite's control server.

of expected system behaviour (termed the baseline). If the differences are significant, Dendrite evaluates adaption rules to return system behaviour to normal. The control server can also optionally exploit domain knowledge to enhance its difference detection and adaption rule selection through *fingerprinting* and *attention focusing* techniques, described later in this chapter.

4.3.1 Overview

Although Dendrite's in-memory tracer enables system-agnostic behaviour extraction and modelling, the control server that selects and deploys adaptions requires some knowledge of database system details. For example, if Dendrite determines that an index should be created to accelerate a client workload, it must know whether the underlying database system is PostgreSQL, MariaDB, or otherwise, to select the proper driver to communicate with the system. The in-memory tracer's system-agnostic nature is still of great value because its bolt-on techniques to extract behaviour models apply to arbitrary relational database systems. This feature eliminates the need for developers to create their own behaviour modelling framework for every database system and provides a consistent behaviour difference format over which adaption rules operate.

Adaption-rule creation is a collaborative process between developers and administrators. Developers create template rules as part of system development, and administrators refine these templates for their environment.

The development process already includes adaption-rule creation; existing rules are simply static and hard-coded in algorithms and configuration files. For example, consider the `shared_buffers` configuration parameter in PostgreSQL, which defines the number of disk pages the database system can store in shared-memory buffers. PostgreSQL uses a default value of 128 MB, but the documentation says that using 25% of the system's total memory pool is a "reasonable starting value" for parameter selection. This is an example of a *static* rule — the system provides a default and advice on adjusting the setting, but the setting is changed only through administrator intervention. Enriching a database system with the ability to dynamically change such values on the fly through adaption rules is a natural evolution of this process.

Dendrite's control server enables adaptivity for arbitrary relational database systems by:

- enabling and leveraging system-agnostic behaviour comparisons to detect and quantify the database system's deviation from expected behaviour. These comparisons can be enriched with domain-knowledge to account for the importance of various system events and to identify running system processes.
- enabling developers to intuitively specify system adaptions and their criteria as adaption rules.
- automatically matching adaption-rule conditions against behaviour differences and deploying adaptions in response to system behaviour changes.

The rest of this section describes these features and the novel techniques enabling them. The design choices powering these features are validated through a robust suite of representative case studies in Chapter 5.

4.3.2 Detecting Behaviour Differences

Once the control server obtains the per-thread behaviour models for the current epoch, it combines them into the current overall behaviour model m_c using the procedure outlined in Chapter 4.2.5. It then compares this model against the registered baseline model, m_b. If the behaviour differences are significant, as defined below, Dendrite will evaluate adaption rules to restore/improve system performance. While Dendrite could perform this comparison using the techniques described in Chapter 3.4, that approach suffers from two key issues.

Recall that the previously described technique compares the proportion of each event e between the two models:

$$diff\_score(m_c, m_b) = \sum_{e \in m_c \cup m_b} event\_score(m_c, m_b, e) \tag{4.3}$$

$$event\_score(m_c, m_b, e) = \max\left(\frac{Prop_{m_c}(e)}{Prop_{m_b}(e)}, \frac{Prop_{m_b}(e)}{Prop_{m_c}(e)}\right) \tag{4.4}$$

where e is an event and $Prop_m(e)$ is the proportion of the total event count in model m attributable to event e (e.g., 40% of the total events in m). If $diff\_score(m_c, m_b) \geq \tau$, for some $\tau > 1.0$, we say that the behaviour difference between these models is significant. Dendrite evaluates adaption rules (Chapter 4.3.4) to react to significant differences and improve system performance.

The first issue with this approach is that not every event is equally important. For example, the appearance of single checkpoint event in PostgreSQL may warrant a behaviour change due to the high overheads it induces, but a slight difference in lightweight query-parse events does not. If the checkpoint count during the current epoch has doubled from the baseline, the prior approach assigns the same weight to this difference as a doubling in query-parse events.

Second, the technique's handling of events present in one model but not in the other is problematic. Suppose an event e occurs in the current model, m_c, but not in the baseline model, m_b. The technique would compute `event_score`$(m_c, m_b) = \infty$, meaning that this difference alone would be sufficient to mark the models as significantly different, regardless of the configured τ threshold. While this large score may be warranted for failure or crash events, it is not for one-off informational events like log-file maintenance.

These issues present challenges for Dendrite's adaption rule framework, since significant behaviour differences would be frequently reported in situations for which adaptions are not warranted. Though adaption-rule conditions can filter out spurious reports without invoking unnecessary adaptions, such reports result in administrator confusion as Dendrite will detect behaviour differences but decide not to respond to them. It would be ideal if Dendrite reported a behaviour difference only in situations where an adminstrator would judge the behaviour difference significant; otherwise, administrators may determine that Dendrite is excessively sensitive and ignore its reports.

Note that reporting a behaviour difference due to these two issues is *not* incorrect; database system behaviour does differ, and the differences are significant in terms of event proportion. However, an administrator's determination of significance is a value judgment based on their domain knowledge and the effects of the difference on database system performance. Therefore, Dendrite enables developers and administrators to optionally signify event importance, as described next.

Attention Focusing

Dendrite enables developers to signal the importance of database system events by specifying a configurable importance-mapping function. This function increases the relative weight of target events in the overall difference score:

$$\mathit{diff\_score\_mapper}(\mathit{fname}, \mathit{line}, m_c, m_b, \mathit{level}, \mathit{raw\_score}) \rightarrow \mathit{score}$$

where *fname* is the event's file name in source code, *line* is its line number, m_c has object fields describing event frequency and proportion of the given event in the current epoch's model (and similarly for baseline m_b), *level* is the event's log level, and *raw_score* is the computed difference score for the event as above.

As a `log()` call is parameterized with a *level* parameter that describes its importance, `diff_score_mapper` can be used to down-weight informational debug messages and more heavily weight error events. However, the mapper can also be used to change the weighting of individual events as required. For example, the mappers used in the case studies in the following chapter clamp the maximum scores for disk page accesses and statistics collector events that on their own are not indicative of behaviour changes requiring adaption.

As the `diff_score_mapper` uses domain knowledge about the importance of events in a data system's life cycle, devising this function is a task most suited for developers for the system. Developers already know which events in their system are the most important, as they use these insights to answer user questions, diagnose performance problems, and alleviate system bottlenecks for their systems in production.

Some system events occur infrequently and may therefore not be present in both of the models Dendrite is comparing. The default `event_score` function would assign this difference a score of infinity, and therefore Dendrite would always indicate that the models are significantly different. As discussed above, this is not always desirable. Therefore, developers can also optionally provide a `missing_event_score_mapper`, which takes the same arguments as the `diff_score_mapper` and similarly outputs a difference score. By default, events present in only one model are assigned a large event difference score of 5.0.

Dendrite's difference detection is effective without specifying these functions, as shown by Chapter 5.3.2. Leveraging domain knowledge to encode the importance of a few targeted events improves Dendrite's resiliency to fluctuating workload environments.

4.3.3 Fingerprinting

It is useful to know which data system processes (or threads) are active when determining how to respond to a behaviour change in the data system. For example, an increase in CPU consumption and query execution count may indicate that more clients are connecting to the system or that the existing clients are submitting more queries. In the former case, an appropriate adaption rule may redirect future client connections to another system node to balance query load. In the latter case, query-level load balancing is more suitable. Dendrite can differentiate between such cases by accounting for the number of active client processes or threads in the system.

An easy way for Dendrite to determine which processes or threads are running is to use the operating system's process/thread list. For example, the `ps` tool on UNIX systems outputs a list of running process IDs and their human-readable process names. Unfortunately, for Dendrite to extract this information the database system must use consistent naming conventions for each executing process/thread. Moreover, Dendrite must know these conventions *a priori* to be able to group executing threads/processes by type (e.g., client workers, checkpointer, autovacuumer) so that adaption rules can operate over them (e.g., worker thread count has doubled in the current epoch relative to baseline). As different database systems can use different naming conventions (or none at all), it is difficult to generalize any technique relying on thread names across data systems.

Dendrite avoids this limitation by identifying running threads through *fingerprinting*. Developers can optionally supply Dendrite with pre-registered models of behaviour indicative of database system threads (e.g., checkpointer). After obtaining the per-thread behaviour models for an epoch, Dendrite compares them to these pre-registered models to determine using the *diff_score* function mentioned above. If the models are similar, Dendrite records that the corresponding thread ID is of the registered model's type and makes this information available to adaption rules to better tailor a response (Chapter 4.3.4). Obtaining these representative models is simple: developers can simply request that Dendrite export a thread's behaviour model during testing and register it with the control server alongside the adaption rules (Chapter 4.3.6).

4.3.4 Adaption Rules

Dendrite uses conjunctive adaption rules to determine how it should respond to changes in workload or database system behaviour. These rules specify the conditions to match against behaviour changes, along with a user-defined function (UDF) to augment the system and improve performance.

Adaption rule syntax is as follows:

```
boolean_expression (AND boolean_expression)* -> PYTHON_UDF( func_args* )
```

A `boolean_expression` is a boolean predicate over the current epoch's differences from the baseline, `PYTHON_UDF` is an arbitrary Python user-defined function (UDF) supplied to Dendrite, and `func_args` are arguments passed to the UDF.

To ease rule composition, Dendrite provides a set of built-in functions that operate over model differences (Figure 4.10). These functions retrieve differences in event frequencies, event proportion, transition probabilities, metric statistics for `libc` calls (e.g., average resource consumption), aggregate system resource-consumption, the elapsed time between events, active processes, and new processes. Developers combine the functions with suitable operators and thresholds to create boolean predicates that form the backbone of adaption rules.

As a representative example, consider an adaption rule used to reduce the aggressiveness of checkpointing in PostgreSQL when the system is under load (Chapter 5.2.1):

```
get_cur_aggregate_metric_value('write') > 25 MB
AND get_transition_cur_metric_sum( ['BufferFlush';'DiskWrite'], 'DiskWriteDone',
'write') > 2 MB AND active_models('checkpointer') = 1 ->
'dial_back_chkpt_and_autovac'()
```

That is, if the total amount of disk writes during this epoch exceeds 25 MB, at least 2 MB of the writes are due to buffer flushes, and the checkpointer process is active, then execute the `'dial_back_chkpt_and_autovac'` UDF.

Event names in adaption rules are specified as strings, while event transitions are specified using a list of prior and subsequent event names. For example, 'BufferFlush' identifies the buffer flush event, and ['BufferFlush';'DiskWrite'], 'DiskWriteDone' describes an order-2 transition from (BufferFlush, DiskWrite) to DiskWriteDone. Metrics are expressed as strings in the domain of intercepted `libc` calls (e.g., 'write', 'malloc').

Function Name	Description
`get_prob_diff`(e)	Probability difference score for event e
`get_[cur|prev]_prob`(e)	Current/previous epoch probability for event e
`get_count_diff`(e)	Frequency difference score for event e
`get_[cur|prev]_count`(e)	Current/previous epoch probability for frequency event e
`get_transition_diff`(e_1, e_2)	Transition probability difference score from e_1 to e_2
`get_transition_[cur|prev]_prob`(e_1, e_2)	Transition probability for current/previous epoch from e_1 to e_2
`get_agg_metric_diff`(m)	Aggregate metric difference for metric m (e.g., `write`)
`get_transition_metric_diff`(e_1, e_2, m)	Transition metric difference (`libc`) for metric m between e_1 and e_2
`get_cdf_diff`(e_1, e_2, m)	The earth-mover's distance between the CDFs of metric m between e_1 and e_2
`get_[cur|prev]_metric_mean`(e_1, e_2, m)	The mean for metric m between events e_1 and e_2 for the current/previous epoch
`get_[cur|prev]_metric_sum`(e_1, e_2, m)	The sum for metric m between events e_1 and e_2 for the current/previous epoch
`active_models`(*type*)	The number of models of type *type* active in the current epoch. If *type* is not provided, matches an process (as with other functions below).
`new_models`(*type*)	The number of models of type *type* newly active in the current epoch

Figure 4.10: A sample of Dendrite's built-in functions for composing adaption rules.

Dendrite allows developers to tag events with human-readable names (e.g., mapping the event from `md.c` on line 655 to BufferReadDone) to simplify rule composition and adjustments. These tags improve the readability of rules and enable administrators that may be less familiar with the database system's source code to customize them for their environment. Tagging also allows the porting of rules between different database system versions as events may change their position in source code between versions.

The `PYTHON_UDF` argument supports arbitrary Python 3 UDFs, which may call shell scripts or other executables as desired. These functions accept a list of function arguments, `func_args`, to which the current and baseline models are prepended. As such, the functions can access characteristics of the models to determine how to respond. As examples of adaption UDFs, one scenario in the following chapter's case studies employs a UDF that changes the system configuration for PostgreSQL (Chapter 5.2.1), and another detects and routes OLAP queries away from a PostgreSQL database to a MonetDB database that more efficiently supports them (Chapter 5.2.3).

Dendrite evaluates adaption rules whenever it observes a significant behaviour difference (Chapter 4.3.2). However, adaption rules can take time to effect the desired change; for example, reducing the frequency of checkpointing may take a few epochs to reduce I/O consumption because a checkpoint is ongoing (and should not be interrupted). If so, subsequent epoch behaviour differences may also match a given rule's conditions, resulting in the same rule being fired multiple times in a row. Dendrite eliminates this concern by executing a UDF only when its conditions are unsatisfied in the previous epoch but are satisfied in the current one.

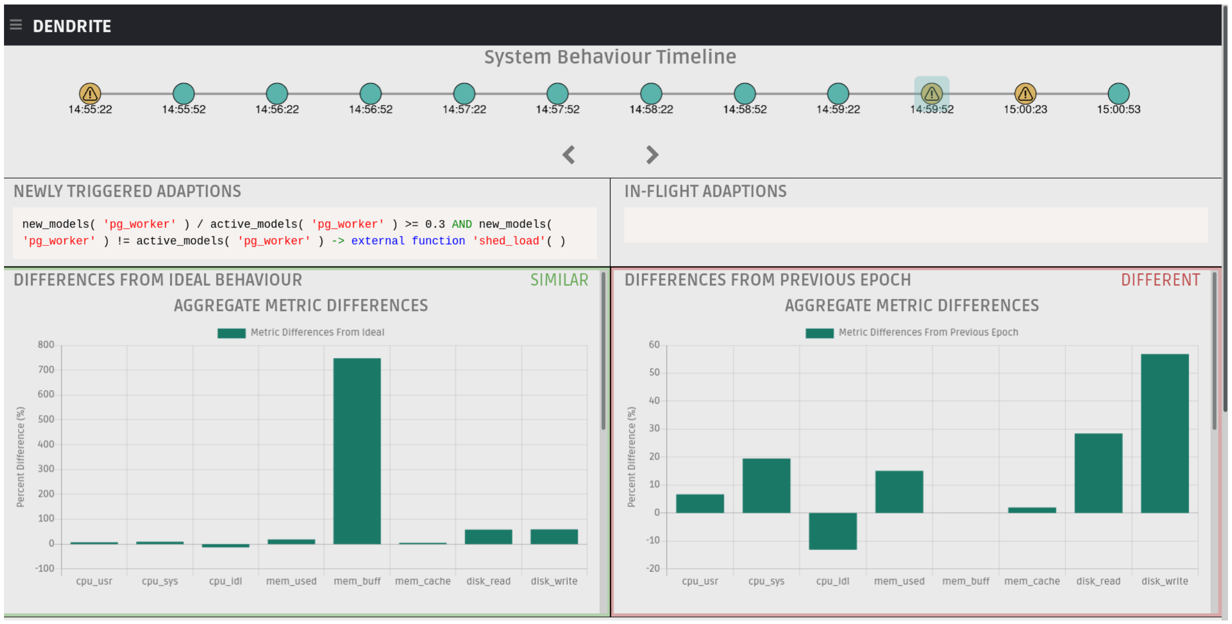
The control server deploys an HTTP server that can be used to dynamically register rules in response to the ongoing workload. The control server will evaluate these additional rules in subsequent epochs. This feature supports administrators interacting with Dendrite's web interface (Chapter 4.3.5), which highlights behaviour changes and alerts administrators when significant changes have occurred.

Rule Evaluation Discussion

Although adaption rules are individually conjunctive, Dendrite's execution of any matching adaption rule over behaviour differences naturally forms a disjunctive condition. Consider the following adaption rule condition:

$$A \wedge (B \vee C)$$

where A, B, and C are boolean subexpressions, $\wedge$ is the logical AND operator and $\vee$ is the logical OR operator.


DENDRITE
System Behaviour Timeline
14:55:22
14:55:52
14:56:22
14:56:52
14:57:22
14:57:52
14:58:22
14:58:52
14:59:22
14:59:52
15:00:23
15:00:53
NEWLY TRIGGERED ADAPTIONS
IN-FLIGHT ADAPTIONS
DIFFERENCES FROM IDEAL BEHAVIOUR
SIMILAR
AGGREGATE METRIC DIFFERENCES
Metric Differences From Ideal
Percent Difference (%)
cpu_usr
cpu_sys
cpu_idl
mem_used
mem_buff
mem_cache
disk_read
disk_write
DIFFERENCES FROM PREVIOUS EPOCH
DIFFERENT
AGGREGATE METRIC DIFFERENCES
Metric Differences From Previous Epoch

```
new_models( 'pg_worker' ) / active_models( 'pg_worker' ) >= 0.3 AND new_models(
'pg_worker' ) != active_models( 'pg_worker' ) -> external function 'shed_load'( )
```

Figure 4.11: Dendrite's user interface.

By the distributive property, the condition can be rewritten as:

$$(A \wedge B) \vee (A \wedge C)$$

i.e., a logical disjunction of conjunctions. This disjunctive adaption condition can be represented using two rules in Dendrite — $A \wedge B \rightarrow UDF$ and $A \wedge C \rightarrow UDF$. In other words, rules in Dendrite that share a UDF reaction are in disjunctive normal form (DNF) as Dendrite will execute any rule that matches the behaviour differences. As all boolean logic formulae may be converted into DNF [29], Dendrite supports a large variety of database system adaption conditions.

Currently, Dendrite evaluates each rule's conditions sequentially. If an administrator has registered rules $R_1, R_2, \ldots R_k$, Dendrite will evaluate R_1 first, then R_2, and so forth. Dendrite could exploit more advanced rule evaluation algorithms (e.g., the Rete algorithm [43]) and work on active databases [13]. These optimizations, however, are orthogonal to Dendrite's contributions of enabling online adaptivity in generalized database systems.

4.3.5 User Interface

In addition to the interface described in the previous chapter for exploring behaviour differences (Chapter 3.4.1), Dendrite presents a web interface that shows ongoing database system behaviour and any adaptions it has deployed (Figure 4.11). This user interface consists of three panels: the timeline panel (top), the differences relative to the baseline panel (bottom left), and the differences compared to the prior epoch panel (bottom right).

The timeline panel presents database system behaviour over time, where each circle describes the behaviour of an epoch. A teal circle indicates that system behaviour during that epoch is comparable to both the expected baseline and the prior epoch, while a gold circle with a warning symbol means that system behaviour differs. Clicking on a circle shows these differences in detail in the differences panels below; the panels show differences in terms of aggregate system resource consumption and event proportion. Dendrite can also attribute resource consumption to particular event transitions using `libc` metric reservoirs in a pop-up panel.

When database system behaviour changes, a pop-up window summarizes the details of the behaviour change and lists any adaptions executed in response. Users may register new rules in this panel using the existing ones as a template. Dendrite considers these rules if system behaviour changes in subsequent epochs.

This user interface is demonstrated in greater detail in a demonstration video [52]. This video highlights Dendrite's utility in remedying load spikes and reducing excess disk I/O due to checkpointing and autovacuuming in PostgreSQL.

4.3.6 System Tools and Deployment

Dendrite provides a robust suite of management tools for developers and administrators to work with behaviour models and adaption rules. These management tools include a PostgreSQL database that indexes historical and current behaviour models and scripts that automate common system management tasks.

When Dendrite outputs per-thread behaviour models to disk at the end of an epoch, it writes them into per-thread files identified by process and thread IDs. New models are appended to these files when an epoch completes. Offline, the control server retrieves these models and loads them into a PostgreSQL database. Dendrite associates each model with its timestamp, epoch, process ID and thread ID.

Administrators use the provided tools to simplify their interactions with the control server:

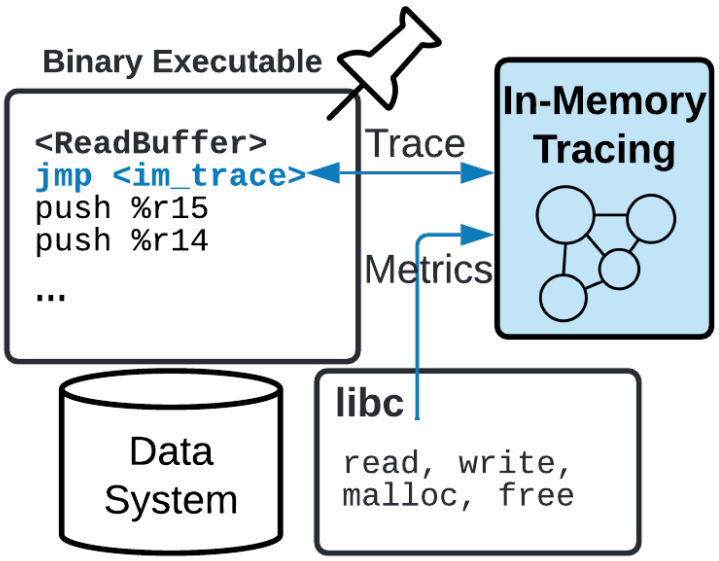
- `pg_get_model`: Retrieves the combined behaviour model from PostgreSQL for the specified epoch, serializing it to disk in a local file.
- `diff_models`: Determines the difference score for the two specified behaviour models (with or without attention focusing), outputting the top differences.
- `read_single_model_at_epoch`: Returns the behaviour model for the requested process/thread ID for the given epoch.

The first two tools enable a detailed offline analysis of a behaviour anomaly. While Dendrite automatically responds to these events, administrators may wish to perform a postmortem analysis of discovered problems to better understand the problem source.

The last tool is useful for registering representative behaviour models. After launching Dendrite and executing the representative workload, a developer or administrator can determine the process/thread ID of an execution unit of interest and export its behaviour model. These exported models are then provided to Dendrite to enable fingerprinting (Chapter 4.3.3).

Building on the approach used in the case studies (Chapter 5), the Dendrite deployment process is envisioned as follows. First, developers integrate the target database system with Dendrite. They modify the system's logging library to forward information to Dendrite's in-memory tracing library (Chapter 5.3.3). Developers use domain knowledge of common system problems that require adaption and reproduce them in test environments with benchmark workloads, using Dendrite to output behaviour differences. As Dendrite reports the top differences when system behaviour changes, these differences provide a convenient starting point for developers to compose template adaption rules. The Dendrite-equipped version of the database system is made available to administrators, along with the template adaption rules, to deploy in their environments.

Next, administrators deploy the database system as usual, using system documentation or tuning tools to obtain good initial performance for their workload. They have Dendrite output a model of system behaviour and register it as the baseline. They augment the provided template adaption rules for their environment, adjusting the thresholds and user-defined functions if necessary. Finally, they monitor Dendrite's user interface and use it to define and register new responses to unforeseen situations that benefit from adaption.



```
<ReadBuffer>
jmp <im_trace>
push %r15
push %r14
...
```

Figure 4.12: Dendrite-Pin extracting events from executables.

4.4 Beyond Logging-based Models

The previous sections have shown how Dendrite constructs behaviour models by intercepting logging messages and system metrics. While debug logging is widely used in practice, a dependence on logging would reduce Dendrite's effectiveness when logging is used sparingly or eschewed entirely.

In this section, I present an alternative version of Dendrite's in-memory tracer built on top of Intel's dynamic binary instrumentation framework, Pin [80]. This version, **Dendrite-Pin**, eliminates the need for logging and can inject adaption logic directly into binary code, unlike the default implementation (**Dendrite-Log**). Developers can choose among these Dendrite versions depending on the availability of source code, their willingness/ability to modify the system's code to improve logging coverage if needed, and their performance requirements.

4.4.1 Overview

Developers use Pin to create instrumentation tools (pintools) that are injected into binary executable code. These instrumentation tools can modify functions in the executable's code, enabling developers to trace and change the underlying system's behaviour. Dendrite-Pin exploits this functionality to extract the information it needs from the system without logging. Deploying Dendrite-Pin is simple — administrators download the Pin framework on the database machine. They then execute the system using Pin and specify Dendrite-Pin's tracer as the pintool to deploy.

By default, Dendrite-Pin overrides same `libc` calls as Dendrite-Log. It also accepts a configuration file specifying a list of additional function names to override in the executable. When the system starts, Dendrite-Pin will locate these functions in the system's binary code, inserting calls to the in-memory tracer's `record_event` function that extract events and encode them into its models (Figure 4.12). In doing so, it captures tracing information that may not be available through logging.

Moreover, Dendrite-Pin overrides same `libc` calls as Dendrite-Log. However, instead of using `LD_PRELOAD` to supply a shared object file with the tracking shim versions of these calls before `libc.so` is loaded, Dendrite-Pin directly overrides the function calls using Pin.

Beyond inserting tracing information, Dendrite-Pin can also entirely replace functions in the executable with custom implementations. The original implementation of a replaced function remains available to the override, enabling it to call the original while modifying function inputs and outputs. As such, Dendrite-Pin can endow systems with adaptive processing strategies they cannot support with solely system-external UDFs. For example, we show how Dendrite-Pin can improve SQLite's performance on an update-intensive workload by adaptively batching queries in Chapter 5.2.4.

4.4.2 Trade-offs between Dendrite Versions

Dendrite-Log and Dendrite-Pin pose different benefits and drawbacks, making them suitable for different environments. We now discuss these trade-offs in more detail, providing insight into when each version should be used.

Dendrite-Pin offers three key advantages over Dendrite-Log. First, it enables systems to integrate with Dendrite without code recompilation. Second, it extracts behaviour models without logging and thus supports systems in which logging is sparse or unused. Third, it can inject new functionality into the system using binary modification, enabling adaption rules to augment internal system logic. If these features are required in the target environment, then the Dendrite-Pin version can be deployed.

However, Dendrite-Log's ability to access source code has advantages. Logging libraries may use one method to test if a message should be written to disk, and a different one to write it out. This mechanism is less suited for Dendrite-Pin because the test function may not use the event's file name and line number, precluding Dendrite-Pin from determining the event's origin by intercepting this function. However, the test function determines whether the write-back function — that contains the needed information — will be executed. This pattern is used in PostgreSQL 14.1 and means that Dendrite-Pin captures slightly different events than Dendrite-Log. Whereas Dendrite-Log obtains its information from logging calls, Dendrite-Pin obtains information by intercepting targeted function calls that may or may not have logging calls in them. Finally, Dendrite-Pin can have slightly higher overhead than Dendrite-Log (Chapter 5.3.1). For these reasons, Dendrite-Log should be deployed when binary instrumentation is not required, using Dendrite-Pin when it is.

The next chapter shows the effectiveness of both Dendrite versions in a variety of representative use cases, examining these trade-offs in more detail.

Chapter 5

Generalized Database System Adaptivity: Case Studies

Dendrite is unique in its *cross-system* ability to model database system behaviour and *enrich these systems with adaptive responses through its adaption-rule framework*. Thus, this chapter focuses on exercising, evaluating and analyzing these capabilities through four representative case studies on several popular database systems.

The case studies answer the research questions posed in the introduction. Namely, they show that:

1. Dendrite efficiently extracts database system events and resource-consumption metrics in a system-agnostic fashion, enabling its modelling framework to integrate easily with popular, open-source database systems.
2. Dendrite's behaviour models are lightweight and capture the information necessary to detect system behaviour shifts. Its difference detection techniques surface the most important behaviour changes.
3. Dendrite's adaption rules are expressive and handle a wide variety of behaviour changes. Adaption rules are portable across database systems with similar characteristics.

To obtain these findings, I integrated Dendrite with PostgreSQL 14.1 [59], MariaDB 10.5 [47], and SQLite 3.36.0 [32]. These databases are well-known and widely deployed; they consistently rank among the most popular open source database systems [31, 103,

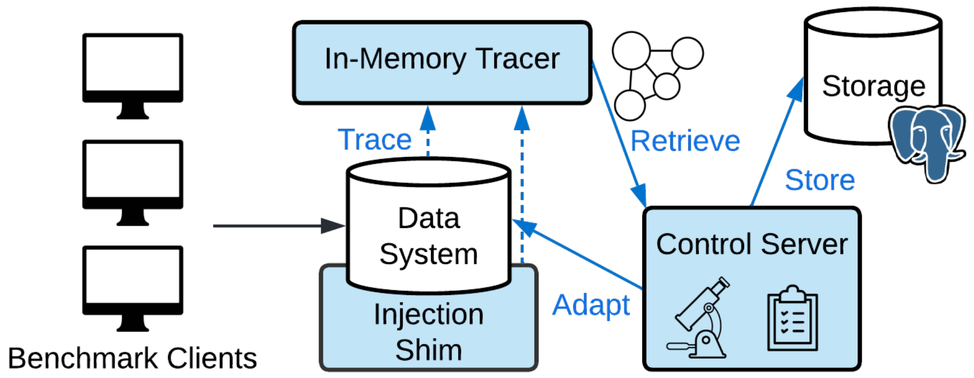


Figure 5.1: Experiment architecture for Dendrite's case studies.

123]. I evaluated Dendrite on three workloads with disparate characteristics: TPC-C [24], CHBenchmark [20], and YCSB [22].

In the case studies, Dendrite:

1. detects disk I/O overheads due to checkpointing and augments configuration parameters to improve performance;
2. reduces query latency by determining when a client workload would benefit from a secondary index and constructing one appropriately;
3. detects clients that are executing analytical queries on a row-oriented database system that would be better served by a column-oriented database system and re-routes their queries accordingly to reduce query-response time;
4. improves the performance of the SQLite embedded database system in an update-intensive environment by dynamically batching updates, reducing processing overheads.

5.1 Experiment Setup

The case studies use a single-node deployment (Figure 5.1), except for the hybrid transaction/analytic processing experiment. The control server is deployed on the same machine as the database system, issuing a command to the database-integrated in-memory tracer every 30 seconds (the configured epoch length) that triggers writing the current epoch's models to disk. To avoid repeatedly polling the files on disk to determine when the models have been written, the control server uses `inotify` [66]. The Linux kernel signals the control server's `inotify` handlers when a model file is created or new data has been written, prompting the server to process the new information. The control server then conducts behaviour comparisons and deploys adaptions as described in the previous chapter.

Clients send requests to the database system to execute each workload using the standard Java-based JDBC or C drivers. The way these systems execute the requests depends on their architectures and the processing techniques they support. I first present a brief overview of each of the systems used in the case studies below.

5.1.1 PostgreSQL

PostgreSQL [59] is a relational database management system (RDBMS) that uses a traditional client-server architecture. Clients use a language-specific driver (e.g., JDBC for Java, `libpq` for C/C++), to send queries through the network to the standalone PostgreSQL database server to execute.

PostgreSQL uses a process-centric model. When a client connects to the system, PostgreSQL forks a worker process to handle the client's requests. Background workers, such as the bgwriter that asynchronously flushes dirty pages or the checkpointer that conducts periodic system checkpoints, also operate in independent processes. PostgreSQL supports parallel query execution by forking new background processes to execute query plans concurrently.

The client, background worker, and coordinator processes communicate with each other using shared memory and signals. The buffer pool resides in shared memory and is typically configured to use 25% of overall system memory. The remaining memory serves as the operating system's file cache.

PostgreSQL uses checkpoints to flush dirty pages to disk and reduce recovery time in case of a system crash. Checkpoints in PostgreSQL are expensive because they flush a large number of pages to disk and consume significant I/O bandwidth. As processing query requests involves writing updates to the write-ahead log (WAL) and reading pages from disk, the extra resource consumption of checkpoints conflicts with normal query processing and can impair performance. The bgwriter offsets this cost by asynchronously writing dirtied pages to disk in batches, reducing the number of pages that must be flushed during a checkpoint. Alternatively, checkpoints can be spread over a longer interval using the `checkpoint_completion_target` configuration parameter, thus avoiding an I/O burst.

Checkpoints are triggered when the WAL reaches a specific size (1 GB by default) or when a configured timeout has elapsed (5 minutes by default).

PostgreSQL tables use multi-version concurrency control to avoid conflicts between concurrent clients that wish to read and update the same record. Updates create new versions of records, and PostgreSQL tracks which record versions should be visible to executing clients to preserve isolation requirements. However, old versions of records that are no longer visible to clients must be periodically removed from the table to prevent tables from growing in size and degrading performance. Background autovacuum workers scan the tables for such "dead tuples" and remove them. Although this process is necessary, it is also disk-bandwidth intensive and impairs database system performance.

PostgreSQL uses `elog` as its debug logging library. PostgreSQL's debug logging is sparse by default, so I configured its important DTrace probes [58] to emit debug logs to improve logging coverage. Integrating PostgreSQL with Dendrite-Log chiefly involves changing a debug-logging macro (`ereport_domain`) to call `record_event` (Chapter 5.3.3).

5.1.2 MariaDB

MariaDB [47] is a popular, open-source fork of the MySQL [23] relational database management system. Although the two systems share a similar codebase, they are increasingly distinct as they grow to support different features.

MariaDB uses a highly asynchronous, threading-based architecture. Clients are assigned threads when they connect to the system, enabling them to share the same address space and buffer pool with other clients. Requests for page data on disk are performed through asynchronous I/O requests (`libaio`), unlike PostgreSQL's use of `read` and `pwrite`.

Like PostgreSQL, MariaDB uses checkpointing to reduce recovery time in case of a system crash. However, MariaDB implements *fuzzy checkpointing* [102]. When a checkpoint starts, it flushes dirty pages in least-recently-used (LRU) order, skipping any latched pages. Of the remaining dirty pages, that with the earliest change according to transaction commit order defines the checkpoint low watermark; all earlier changes have been flushed to disk, so recovery can start from this low watermark in the WAL.

As MariaDB implements its WAL differently than PostgreSQL, the conditions under which checkpointing is triggered also differ. Unlike PostgreSQL's append-based WAL, MariaDB uses a fixed-size, circular, disk-backed buffer. WAL records are written to this buffer in the typical circular fashion. If a write to the WAL would overwrite a record that has changes in the buffer pool as of yet unflushed to disk, the system must block to flush the page to disk before the record would be overwritten. This condition triggers an aggressive round of page flushing that blocks transactions, leading to significant performance degradation. The other condition for checkpointing is more benign — if 90% of the pages in the buffer pool are dirty, MariaDB will trigger a lightweight checkpointing round that does not block.

MariaDB's tables do not require autovacuuming. Before a record is updated, its old version is copied to a temporary buffer called the rollback segment. Afterward, the transaction directly updates the record's values in the table. Records in the rollback segment are purged when they are no longer visible to ongoing transactions, which incurs less overhead than the scans required by PostgreSQL's autovacuuming. Thus, MariaDB incurs more overhead on updates to maintain this rollback segment but avoids the cost of vacuums, while PostgreSQL is the opposite.

MariaDB does not provide support for parallel query execution, unlike PostgreSQL. While PostgreSQL aggressively uses parallel plans to reduce the latency cost of expensive analytics queries, MariaDB cannot benefit from such strategies.

To output debug logging information, MariaDB uses the DBUG library [41]. This library provides `DBUG_ENTER` and `DBUG_EXIT` macros that are called at the entrance and exit of most functions, along with `DBUG_PRINT` for outputting information in the middle of function execution. These macros feature group arguments so that logging statements may be enabled and disabled according to their group name (e.g., `lock`, `ib_buf`). Integrating MariaDB with Dendrite-Log requires changing only these `DBUG_*` macros to call `record_event` as part of their execution (Chapter 5.3.3).

5.1.3 SQLite

SQLite is a popular, embedded relational database management system, in contrast to the client-server models of PostgreSQL and MariaDB. SQLite stores the database in a single file, unlike the complex file architectures of the other database systems. Clients interact with SQLite using function calls provided by SQLite's client-library instead of issuing requests over network sockets.

Although SQLite supports concurrent clients, it executes update transactions serially. Clients that wish to modify data while another client is updating it must block or use a retry loop to repeatedly submit their transaction. Moreover, an update from one client connection invalidates the local caches of the other clients, causing significant performance degradation. Thus, SQLite targets read-mostly workloads.

Like the other RDBMSs, SQLite supports checkpointing dirty pages to minimize recovery time. Automatic checkpointing executes after a configurable number of log frames have been written to SQLite's WAL, defaulting to 1000. SQLite also supports autovacuuming, but differs from PostgreSQL in that it packs the database into the smallest file size. Autovacuuming is disabled by default but can be configured to execute on every transaction commit or by user command.

SQLite does not output debug logs by default. Internally, SQLite uses `TRACE` macro statements to output information, but compiler flags typically disable them. While it is a simple matter to adjust the compiler flags and adjust the `TRACE` macro to call `record_event` (as in Chapter 3.5), the case studies in this chapter use this situation to demonstrate the benefits of Dendrite-Pin.

5.1.4 MonetDB

MonetDB [11] is a relational database management system that uses column-oriented data storage and excels at processing analytics queries. Like PostgreSQL and MariaDB, it employs a client-server architecture; clients submit queries to MonetDB over a network socket.

MonetDB automatically creates indexes to optimize for the workload [65], while PostgreSQL, MariaDB, and SQLite necessitate that administrators hand-select them. MonetDB's queries are automatically compiled into parallelizable, vectorized execution plans to improve performance.

5.1.5 Workloads

The case studies consider three workloads with disparate characteristics.

TPC-C [24]: TPC-C is a popular order-entry benchmark. Its update-intensive nature is representative of online transaction processing (OLTP) workloads.

CHBenCHmark [20]: While analytics (OLAP) and transaction processing (OLTP) workloads have traditionally been considered disjoint, there has recently been increased interest in executing complex analytical queries over the most up-to-date transactional data. Workloads exhibiting such characteristics are called hybrid analytics/transaction processing workloads (HTAP), and are among the most challenging for databases to efficiently serve.

CHBenCHmark unifies a representative OLTP workload, **TPC-C**, with a popular OLAP workload, **TPC-H** [25]. CHBenCHmark executes a configurable number of TPC-C clients in parallel with a configurable number of TPC-H-like workers in the background. The TPC-H style workers operate over an extended TPC-C schema to support TPC-H queries while still incorporating the data updated by the OLTP clients.

YCSB [22]: The Yahoo Cloud Serving Benchmark is a key-value style benchmark that features a variety of workload types. The case studies and microbenchmarks use variations of YCSB-B and YCSB-C benchmarks. By default, the YCSB-B workload uses a 95%/5% read-only/update transaction mix, while the YCSB-C workload uses a 100% read-only transaction mix. All queries select the records to be retrieved or updated based on the primary key of the relation and thus use index scans. These workloads are used to test automatic index creation, evaluate dynamic query batching, and examine Dendrite's overheads.

5.2 Case Studies

The following case studies comprehensively evaluate Dendrite's generality and adaption capabilities. The first three case studies show Dendrite's ability to adapt system behaviour in the popular PostgreSQL and MariaDB database systems. These database systems have broadly similar characteristics but distinct architectures (process vs. thread-based), features (e.g., intra-query parallelism in PostgreSQL, which is not supported in MariaDB), and code bases. As such, these case studies demonstrate that Dendrite is effective in a variety of scenarios and across database systems. Moreover, they show that adaption rules composed for one database can be ported to other similar systems.

The last case study uses the SQLite database system, which unlike PostgreSQL and MariaDB uses limited debug logging. Even in this different and challenging environment, Dendrite is effective in detecting behaviour changes and deploying adaptions to remedy them through its use of binary instrumentation.

Each experiment's performance metric of interest is averaged over at least three runs. Each machine uses the Ubuntu 20.04 operating system and is equipped with 12 CPU cores with hyperthreading enabled, 32 GB of memory, a 1 GB/s network card, and a 1 TB HDD.

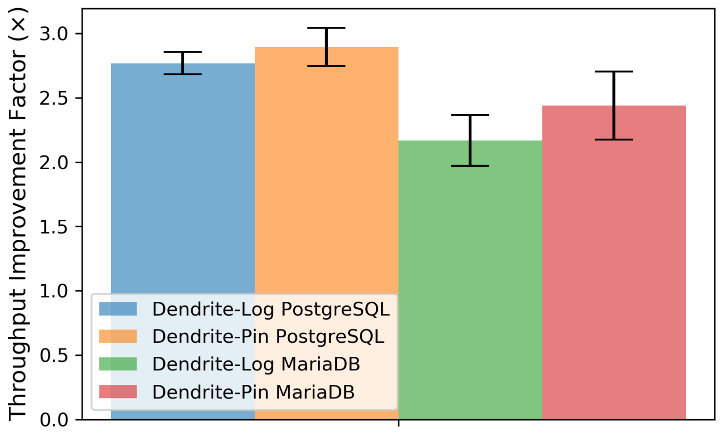


Figure 5.2: Checkpoint adaption throughput improvements.

5.2.1 Reducing Checkpoint Frequency

Checkpointing is a critical component of database maintenance. Checkpoints execute periodically, flushing dirty pages to disk to reduce recovery time if the database crashes. Unfortunately, checkpoints are known to have high overheads as they consume significant CPU and disk resources that could otherwise be used for query processing. In this scenario, Dendrite reduces an aggressive checkpointing threshold to ameliorate checkpointing overheads and improve system performance.

This case study uses the default configurations for both PostgreSQL and MariaDB, except that the `checkpoint_completion_target` configuration parameter in PostgreSQL is adjusted to 0.0 to improve checkpoint completion speed.

Both Dendrite versions are supplied with a behaviour model of the checkpointer process/thread in PostgreSQL and MariaDB for identification through fingerprinting (Chapter 4.3.3). I registered a baseline behaviour model extracted from a period when checkpointing was not taking place. All models were obtained during a dry run of this experiment.

Dendrite-Log is configured with the following adaption rule for both PostgreSQL and MariaDB:

```
get_cur_aggregate_metric_value('disk_write') > 25 MB
AND get_transition_cur_metric_sum( ['BufferFlush'; 'DiskWrite'], 'DiskWriteDone',
'write') > 2 MB AND active_models('checkpointer') = 1 ->
'dial_back_chkpt_and_autovac'()
```

That is, if the aggregate disk writes during an epoch exceed 25 MB, 2 MB of these writes are due to flushing pages to disk in a checkpoint, and the checkpointer is active, then Dendrite executes the `dial_back_chkpt_and_autovac` user-defined function.

This UDF reduces the frequency of checkpointing and autovacuuming. In PostgreSQL, this function modifies the PostgreSQL configuration to set `max_wal_size` to 100 GB and `checkpoint_timeout` to 6 hours, which act as the thresholds for when checkpointing is triggered (Chapter 5.1.1). It also reduces the background autovacuum worker count to 0 and cancels ongoing autovacuuming.

As MariaDB supports dynamically resizing its WAL size using the `SET` SQL command[1], Dendrite uses this approach to increase the WAL file size to 8 GB. Since checkpointing in MariaDB on the update-intensive TPC-C workload using the default configuration is primarily triggered by overwriting unflushed WAL records in the small 96 MB WAL (Chapter 5.1.2), this change dramatically improves system performance by alleviating blocking and I/O overheads. Dendrite-Pin uses the same adaption rule for both MariaDB and PostgreSQL, simply adjusted to use the names of the relevant intercepted functions.

I executed 20 TPC-C workers against a scale factor 10 database using OLTPBench [35]. The write-intensive nature of this workload rapidly fills the WAL and triggers checkpointing in PostgreSQL and MariaDB. Dendrite recognizes the behaviour difference from expected behaviour in the first few epochs for both PostgreSQL and MariaDB, matching the provided adaption rule and reducing checkpoint frequency. Dialing back checkpoint frequency takes place immediately in PostgreSQL, but does not cancel ongoing checkpoints (which take 5 minutes to complete). After the checkpoint completes, Dendrite reports that the system has converged to expected system behaviour. Similarly, reducing checkpoint frequency for MariaDB necessitates online resizing of the WAL but provides significant throughput improvements once the resizing completes (5 minutes). Afterward, MariaDB converges to expected system behaviour.

As a result of these system adaptions, Dendrite improves system throughput by more than 2.5× in PostgreSQL and 2× in MariaDB (Figure 5.2). Note that the same adaption rule conditions work for both systems; porting rules requires only a simple developer-provided abstraction layer to determine which events are equivalent across the systems.[2]

[1]As dynamic WAL resizing is supported in MariaDB 10.9+, this version is used for this experiment.

[2]e.g., DiskWrite = `buf0flu.cc:835` in MariaDB and `md.c:710` in PostgreSQL.

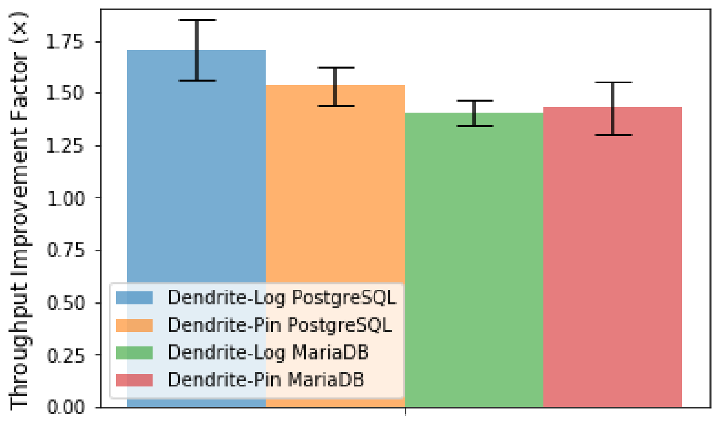


Figure 5.3: Index-creation adaption throughput improvements.

5.2.2 Automatic Construction of Secondary Indexes

Indexes are widely used in database systems to accelerate lookup operations on relations with a large amount of data. However, they must be updated when new data items are inserted or updated, leading to a trade-off between search and update operation performance. Administrators must determine which indexes are worth the maintenance cost, which depends on client access patterns in the workload. This case study shows how Dendrite detects circumstances in which a workload would benefit from a secondary index, determines which index to create, and consequently improves system throughput.

When a database system needs to find a record in a relation for which indexed information is unavailable, it resorts to a sequential scan. While PostgreSQL uses parallel background workers to improve the performance of large scans, MariaDB lacks this capability. As PostgreSQL does not have debug logging events for sequential scans, I targeted logging events for the creation of background scan workers in Dendrite's adaption rules for this system. The MariaDB rules operate over sequential scans directly.

Dendrite-Log is equipped with the following adaption rule for PostgreSQL:

```
get_prob_diff( 'BackgroundWorkerStart' ) > 2.0 AND
get_count_diff( 'CommitTransaction' ) > 1.5 AND
get_cur_count( 'CommitTransaction' ) < get_prev_count( 'CommitTransaction' ) ->
'create_appropriate_sec_index'()
```

That is, if the proportion of background workers registered has increased by 2×, but the number of committed transactions has decreased by 50%, then execute the `create_appropriate_sec_index` user-defined function. This UDF retrieves the set of frequently executed queries that are using sequential scans, and constructs a secondary index over the fields on which the queries have a predicate.

In PostgreSQL, `create_appropriate_sec_index` uses the `pg_stat_user_tables` view to find the tables that have the largest number of sequential scans. The UDF determines that the sequential scans primarily execute over the `USERTABLE` relation and uses the `pg_stat_statements` view to retrieve slow queries operating on `USERTABLE` with an execution count greater than 10. It parses the query and finds the columns it accesses, constructing an index over them to improve query performance.

In MariaDB, `create_appropriate_sec_index` retrieves the slowest-to-execute queries from MariaDB's `slow_log` relation, and parses them as in PostgreSQL. It extracts the relevant columns and relation, and constructs an index over these columns for the specified relation.

The adaption rule used for MariaDB is similar to that of PostgreSQL, but the 'BackgroundWorkerStart' $>$ 2.0 condition is substituted for a 'SequentialScan' $>$ 2.0 check. The rest of the rule is the same. The Dendrite-Log rule for a given database system is also used for its Dendrite-Pin experiments. Again, observe that a simple abstraction layer mapping events of interest between systems is sufficient to port a rule.

I executed a 25-client YCSB-C workload against PostgreSQL and MariaDB. The default configuration is used for both database systems while disabling checkpointing and autovacuuming to remove background performance effects and ensure consistency across results. At 90 seconds into the 300 second workload, 15 clients switch to executing a record lookup operation using an unindexed field. These queries require sequential scans and are significantly slower compared to the original queries that use indexes.

Dendrite detects a behaviour shift in both PostgreSQL and MariaDB in the first epoch in which queries requiring sequential scans execute (epoch 3). Both systems take less than 30 seconds to create the necessary index, after which performance improves and behaviour rapidly converges to expectations. As a result of this adaption, Dendrite improves performance by between 1.4× and 1.7× (Figure 5.3).

While all adapted systems provide significant gains compared to their unadapted defaults, the PostgreSQL configurations enjoy the largest benefits since PostgreSQL incurs a larger penalty than MariaDB for a lack of indexing on this read-intensive workload. This effect is due to its parallel sequential scans amplifying resource contention. Dendrite-Pin is marginally slower for PostgreSQL because Pin has higher overhead than log interception

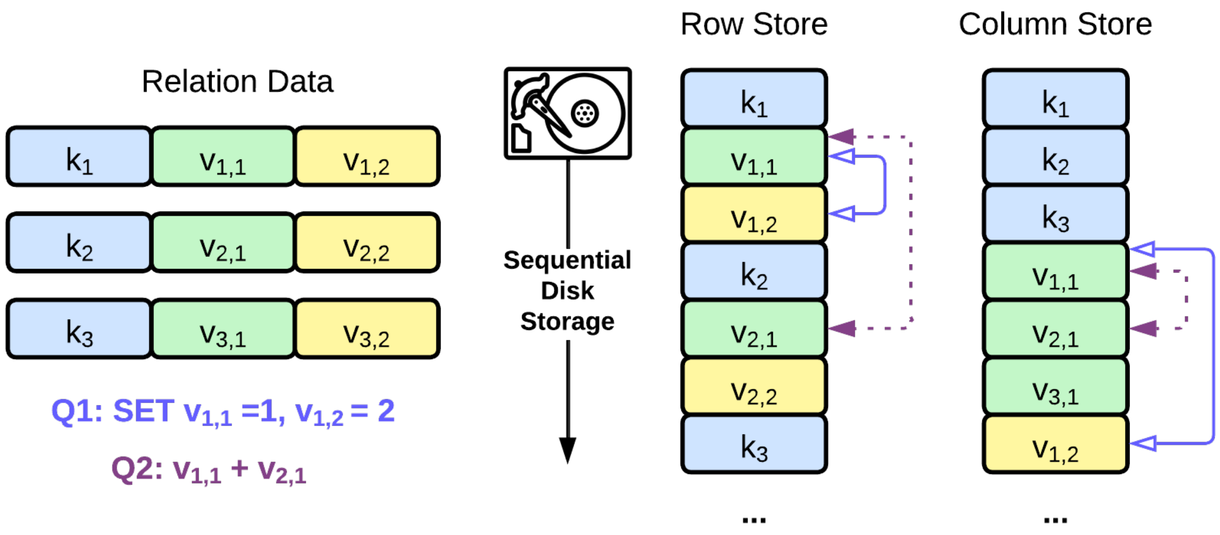


Figure 5.4: Row-oriented storage vs. column-oriented storage. Updates that span multiple record fields are best served by row-stores, while queries that aggregate the same field across records are best served by column-stores.

(Chapter 5.3.1), and both configurations capture similar numbers of events. By contrast, Dendrite-Pin is marginally faster than Dendrite-Log on MariaDB since it intercepts fewer function calls than the events captured by Dendrite-Log.

5.2.3 Handling an HTAP Workload

Relational database management systems select between column-oriented and row-oriented storage. Column-oriented storage packs a relation's data *columns* sequentially on disk, whereas row-oriented storage packs *records* sequentially (Figure 5.4). Column-oriented storage is read-optimized [1,5,11], in that only the columns accessed by submitted queries need to be read from disk. In contrast, row-oriented storage must skip over unused fields during a sequential scan (purple query in Figure 5.4). The downside of column-oriented storage is that transactions which update multiple record fields need to access data spread across non-sequential disk locations, whereas row-oriented storage places these fields closer together (blue query in Figure 5.4). Hence, read-heavy analytics workloads (OLAP) are best served by column stores, while row-stores best serve update-intensive workloads (OLTP). Hybrid workloads (HTAP) benefit from a hybrid storage approach.

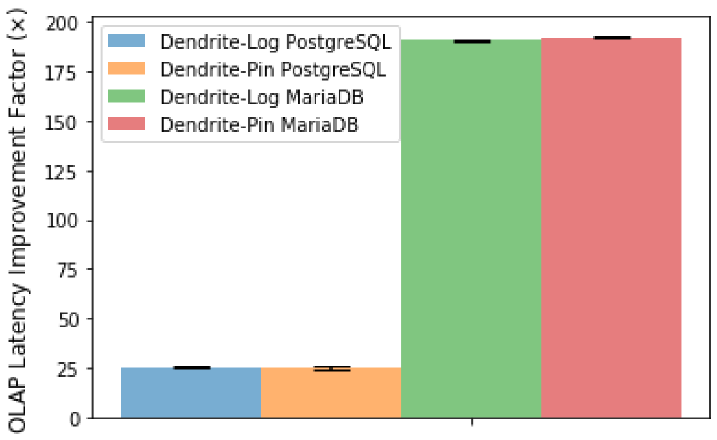


Figure 5.5: CHBenchmark query latency improvements.

In this case study, Dendrite automatically detects OLAP worker threads/processes executing within a hybrid workload. It changes these workers' query routing strategies, routing their queries to a column store instead of the primary row store. Doing so reduces the latency of these expensive analytics queries significantly.

I used OLTPBench to create a scale factor 10 CHBenCHmark database in PostgreSQL (or MariaDB), and replicate the data to MonetDB. As PostgreSQL and MariaDB are row-oriented database systems, they serve as the primary row-store system. MonetDB uses a column-oriented data storage and therefore serves as a standby node to which analytics queries will be redirected.

Both PostgreSQL and MariaDB are equipped with the same configurations used in the prior case study, and MonetDB is equipped with its default configuration. To obtain a representative model of desired workload behaviour, I executed the OLTP portion of CHBenCHmark against the PostgreSQL/MariaDB database, and exported a behaviour model during normal system execution. I then enabled the OLAP portion of the workload and repeated this experiment, exporting models of the PostgreSQL processes and MariaDB threads used to execute the analytical queries. I registered the OLTP-only model as the baseline and provided Dendrite with OLAP worker models of the row-store databases for use in fingerprinting.

I equipped Dendrite-Log with the following adaption rule for both PostgreSQL and MariaDB:

```
get_cur_count('DiskRead') > get_prev_count('DiskRead') * 5.0 AND
active_models('olap_worker') >= 2 -> 'route_olap_to_monet'()
```

That is, if the count of DiskRead events has increased by more than 5× and at least two `olap_worker` processes/threads are active, then Dendrite will re-route the OLAP queries to MonetDB. For both systems, Dendrite-Pin is equipped with an equivalent rule that operates over relevant function execution counts.

In PostgreSQL, the `route_olap_to_monet` UDF determines which processes correspond to OLAP workers using the provided fingerprint and the per-thread models. It obtains their process IDs and identifies the client connections using the metadata on JDBC connections. These clients are transparently adjusted to interact with MonetDB instead of the default row store, improving performance. The MariaDB UDF is similar, except that it uses thread IDs as MariaDB uses a threading-based architecture.

I executed 10 OLTP CHBenCHmark workers to execute against the row-store database. After 180 seconds, 2 OLAP workers began executing analytical queries as well.

During the initial epochs, Dendrite reports that MariaDB/PostgreSQL behave as expected, since there are no OLAP queries running. In the first epoch in which OLAP queries appear, Dendrite detects a behaviour difference in both systems. Dendrite matches the new processes'/threads' behaviour against the registered model using fingerprinting and determines that they are OLAP workers. As the DiskRead condition also matches, Dendrite deploys the `route_olap_to_monet` UDF, which obtains the process/thread IDs of the running OLAP workers and forwards them to the connection pool manager for rerouting. The connection pool transparently swaps the underlying row-store connection for a connection to MonetDB, rerouting the queries.

As as result of Dendrite's adaption, average OLAP query latency is reduced by a factor of 25× compared to PostgreSQL, and by 200× compared to MariaDB (Figure 5.5). These gains are due to MonetDB's superior execution strategy and column-oriented data storage layout compared to the row-store systems. The performance difference between the PostgreSQL and MariaDB results is due to PostgreSQL's parallel query execution support and superior query plans.

5.2.4 Batching Updates in SQLite

SQLite is an embedded database system suitable for read-intensive workloads, but it suffers from contention on update-intensive workloads. Unlike PostgreSQL and MariaDB,

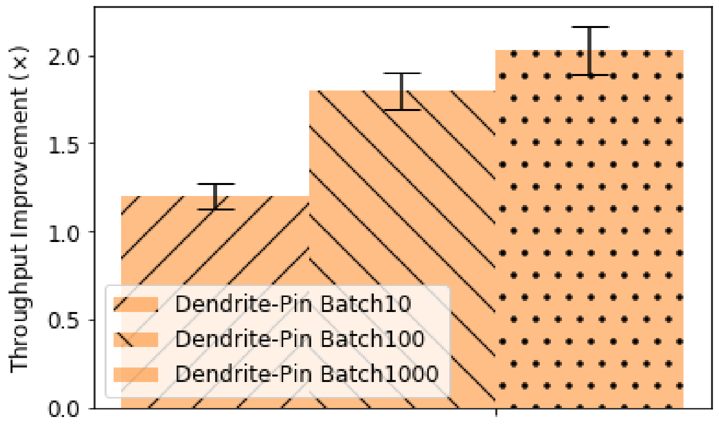


Figure 5.6: SQLite batched query adaption throughput improvements.

it supports only serial, i.e., one-transaction-at-a-time, execution of updates. In this case study, Dendrite improves SQLite's update performance through binary instrumentation. Dendrite detects an increase in submitted update transactions, dynamically batches them into a single larger transaction and executes them as a unit. In doing so, it improves performance by mitigating the overheads of processing only a single update operation at a time.

In contrast to PostgreSQL and MariaDB, SQLite provides limited debug logging. While enhanced logging can be enabled by adjusting compilation flags, this necessitates rebuilding the software with a custom configuration. Therefore, this study focuses on Dendrite-Pin, demonstrating its ability to model systems *without* recompilation or reliance on logging. I configured Dendrite-Pin to trace the `sqlite3Select` and `sqlite3Update` functions, which execute select and update queries respectively. Dendrite-Pin is provided with the following adaption rule:

```
get_prob_diff( 'sqlite3Update' ) > 1.5 AND
get_cur_prob( 'sqlite3Update' ) > get_prev_prob( 'sqlite3Update' ) ->
'start_batching'()
```

That is, if Dendrite determines that the proportion of update queries has increased by 50% then it executes the `start_batching` UDF.

`start_batching` creates a `/hdd1/override` file. The file is used by `sqlite3ExecWrapper`, a function injected by Dendrite-Pin to override the `sqlite3Exec` function responsible for executing SQL strings:

```
int SQLite3ExecWrapper(sqlite3 *db, const char *sql, int (*callback)(void *,
     int, char**, char**), void *first, char *errmsg) {
    struct stat statbuf;
    int rc = stat( "/hdd1/override", &statbuf );
    bool not_exists = (rc == -1);
    if( not_exists || strncmp( sql, "UPDATE usertable", 16 ) != 0 ) {
        // Can't guarantee this is our UDPATE, so don't buffer it. Pass
   through.
        return (*sqliteExecFunc)( db, sql, callback, first, errmsg );
    }

    // Assume we can buffer or execute it at will.
    std::lock_guard<std::mutex> lk( batch_mutex );

    if( batch_updates.size() == 99 ) {
        drain_buffered_queries( db, sql );
    } else {
        batch_updates.push_back( strdup( sql ) );
    }
    return SQLITE_OK;
}
int SQLite3CloseWrapper( sqlite3 *db ) {
    {
        std::lock_guard<std::mutex> lk( batch_mutex );
        drain_buffered_queries( db, NULL );
    }
    return (*sqliteCloseFunc)(db);
}
```

The `SQLite3ExecWrapper` function checks to see if a file called `/hdd1/override` exists on disk, and if so, batches updates. To do so, it adds received `UPDATE` queries on the `usertable` relation to the `batch_updates` queue. If the queue reaches a fixed length, the queue is drained alongside the current query. If the query is not an update, or it is not an update that should be batched (e.g., background maintenance queries performed by SQLite), then the query batch is bypassed and executed as usual. This adaption temporarily trades off ACID semantics of the batched update queries for improved performance; Dendrite's adaption rules can codify and exploit cases where developers/administrators deem this trade-off appropriate, as in this case study. Although this adaption shows the possibilities of Dendrite-Pin's responses, it could be enhanced to provide ACID-compliant group-commit semantics [33] if desired.

```
void drain_buffered_queries( sqlite3 *db, const char *last_sql ) {
    char *errmsg;

    (*sqliteExecFunc)( db, "BEGIN", NULL, NULL, errmsg );
    while( !batch_updates.empty() ) {
        char *sql_ptr = batch_updates.back();
        batch_updates.pop_back();
        (*sqliteExecFunc)( db, sql_ptr, NULL, NULL, errmsg );
        free( sql_ptr );
    }
    if( last_sql ) {
        (*sqliteExecFunc)( db, last_sql, NULL, NULL, errmsg );
    }
    (*sqliteExecFunc)( db, "COMMIT", NULL, NULL, errmsg );
}
```

The query buffer is drained using the `drain_buffered_queries` function. Dendrite executes all of the queries in the queue as a single update transaction, wrapping them in `BEGIN` and `COMMIT` SQLite commands.

I executed the YCSB-B workload against SQLite, starting with an initial mix of 90% read and 10% update transactions. A model of this workload's behaviour from SQLite was exported using Dendrite and registered as the expected model of system behaviour. After 150 seconds, the workload switches to an 80% read, 20% update mix. Dendrite rapidly detects this behaviour shift, matches it against the provided adaption rule, and processes update requests in batches for the remainder of the 300 second workload.

I evaluated batch sizes of 10, 100, and 1000 to determine which yielded the most significant benefits (Figure 5.6). As shown, even a modest batch size of 10 yields benefits over default SQLite (1.2×), but a larger batch size of 100 provides a larger throughput advantage (1.8×). The relative gains become smaller as the batch size grows to 1000 (2.0×). These results demonstrate the utility of Dendrite-Pin in integrating with systems that use limited logging, as well as the power of modifying system code through binary injection.

5.3 Microbenchmarks

To quantify Dendrite's overheads, I conducted targeted assessments through microbenchmarks. This section analyzes these overheads in terms of throughput reduction and memory consumption.

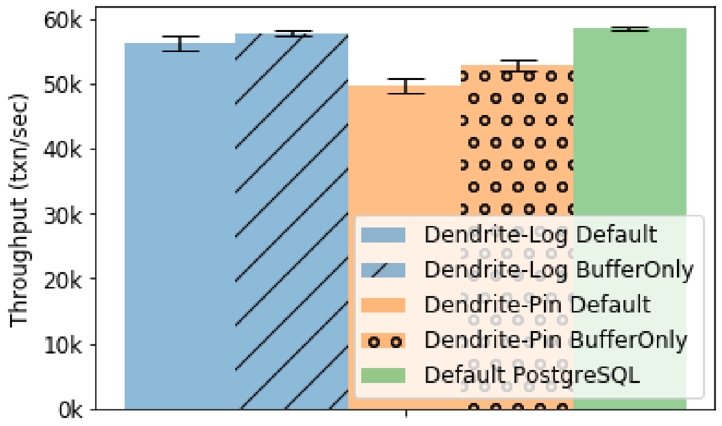


Figure 5.7: PostgreSQL throughput on a YCSB-C workload.

5.3.1 Quantifying Overheads

To assess Dendrite's overheads, I executed a 25-client YCSB-C workload against PostgreSQL; YCSB-C's simple queries move the bottleneck from query execution to Dendrite's tracing. As Dendrite-Log and Dendrite-Pin capture a different number of events using their default configurations (Chapter 4.4.2), I also contrast each version when they capture only the popular BufferPageAccess event. This approach ensures an apples-to-apples comparison.

The default configurations of Dendrite-Log and Dendrite-Pin reduce system throughput with respect to default PostgreSQL by 4% and 15%, respectively. Dendrite-Pin has higher overhead than Dendrite-Log because of the increased performance impact of binary instrumentation compared to log-based integration (Chapter 4.4.2).

When both versions use an equivalent tracing configuration (BufferOnly), Dendrite-Pin has only 8% higher overhead than Dendrite-Log. Again, this difference is attributable to the overheads of binary instrumentation. These results show the efficiency of Dendrite's tracing; capturing information for an event requires updating only a few counters in hash maps. Moreover, model comparisons and rule evaluation are performed off the main path of execution and therefore have little effect on overall performance.

The default configurations of Dendrite-Log and Dendrite-Pin consume less than 2 MB of memory per process for tracking events, event transitions, and metric reservoirs. Overwhelmingly, this memory comes from tracking metric consumption reservoirs on a per event-transition basis. PostgreSQL exhibits few unique transitions, which is expected due to the consistent way in which it moves between events. As Dendrite allocates memory for reservoirs only if required, it minimizes memory consumption.

5.3.2 Attention Focusing and Fingerprinting

Dendrite's attention-focusing techniques enable developers and administrators to encode domain knowledge about which database system events are the most important and should be weighed more heavily when determining system behaviour differences. Similarly, by supplying Dendrite with behaviour models corresponding to database system processes (e.g., the checkpointer), Dendrite can identify when these processes are active and exploit this information in adaption rules.

To examine the sensitivity of Dendrite's fingerprinting (Chapter 4.3.3) and attention-focusing techniques (Chapter 4.3.2), I compared PostgreSQL worker behaviour models obtained from different epochs, experiments and workloads. I obtained a YCSB-C worker model from the overheads microbenchmark above and compared it against the other clients' models obtained during the same epoch and during database system startup. I also compared this model against models obtained from a different execution of the same YCSB-C workload, against a different workload mix (50/50 read/read-modify-write YCSB) and against a different workload, TPC-C. I tested model similarity with and without attention-focusing mappers enabled (denoted by 'AF' and 'no AF', respectively), to show that although the mappers improve robustness, Dendrite remains effective without them.

The distribution for difference scores among models for each setting is shown as boxplots in Figure 5.8. As in the case studies, Dendrite is configured to treat models with a difference score less than 3.0 (the dotted line) as similar when fingerprinting. Hence, scores below the blue dotted line indicate that Dendrite considers the models to be of the same type, with scores above indicating significant differences. Dendrite considers worker models obtained from the same YCSB-C configuration to be similar to the registered default model, except for models obtained while the system was starting with attention-focusing techniques turned off. The largest differences in these cases correspond to startup events not present in the registered YCSB-C worker model and in disk read probabilities since the buffer pool is not yet warm. Attention focusing lessens the importance of these differences as it recognizes their transient nature, enabling Dendrite to detect active YCSB-C workers correctly during startup when these techniques are enabled.

By contrast, Dendrite considers PostgreSQL worker models obtained under different workload mixes or workloads to be significantly different from the YCSB-C worker model, regardless of whether attention-focusing techniques are enabled. These large scores are

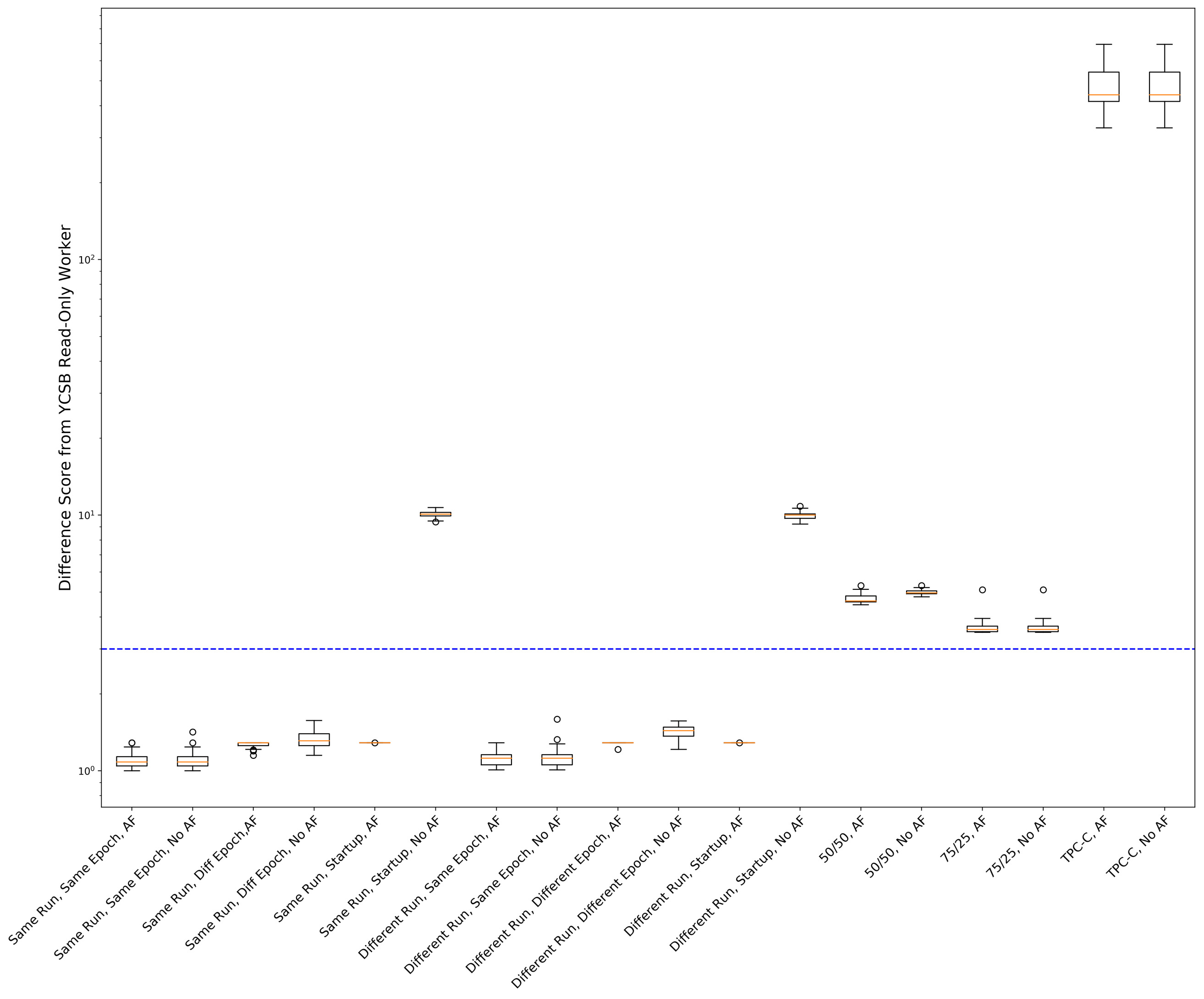


Figure 5.8: Model difference scores for a PostgreSQL YCSB 100% read-only worker compared to worker models obtained from other environments.

due to significant differences in disk access and transaction management events. This distinction is desirable because it shows that Dendrite can recognize processes by their type (e.g., PostgreSQL worker) and differentiate them based on their execution characteristics (e.g., transaction mix, workload).

5.3.3 System Integration

The case studies required integrating Dendrite with PostgreSQL 14.1, MariaDB 10.5, and SQLite 3.36.0. I now describe the integration process with each of these database systems.

Integrating Dendrite-Log with PostgreSQL requires few targeted changes. First, I modified PostgreSQL's logging macro, `ereport_domain`, to call Dendrite's `record_event` function, and modified PostgreSQL's Makefiles to link against Dendrite's shared libraries. Next, I configured targeted DTrace probes in PostgreSQL to call `record_event` so that this information is available to Dendrite. Then, I changed PostgreSQL's `pg_ctl` binary to set the `LD_PRELOAD` environment variable to point to Dendrite's injection shim before starting the PostgreSQL server, so that it can obtain resource-consumption metrics. Finally, I adjusted the exit handlers of processes to emit their behaviour models to disk. Altogether, these changes required just 35 lines of code. Note that this integration requires even fewer lines of code than the difference detection experiments from Chapter 3.5.6. This difference arises because the PostgreSQL versions differ and because the full Dendrite system does not need signal handlers to catch system shutdown; the control server has the running database system emit models at the end of every epoch.

Integrating Dendrite-Pin with PostgreSQL is even simpler. I provided Dendrite-Pin with a set of functions called around the DTrace probes configured for Dendrite-Log, as well as the `errfinish` logging function (as the `ereport_domain` macro is not accessible in the compiled binary). As with Dendrite-Log, I intercepted process exit-handler functions and configured them to write captured behaviour models to disk. Altogether, these changes intercept a set of 12 functions and require no changes to PostgreSQL source code.

To extract behaviour information from MariaDB with Dendrite-Log, I intercepted MariaDB's `DBUG` macros. MariaDB uses these macros liberally throughout the codebase, including at the entrance and exit of many function calls. Macros are grouped by the area of the code they operate in or their functionality — Dendrite uses these group names to extract information from only the performance-relevant groups. To match PostgreSQL, I inserted tracing during buffer writebacks and sequential scan initialization. Altogether, these changes modified only 29 lines of code in MariaDB. Dendrite-Pin used a similar approach to the PostgreSQL integration, intercepting 10 functions and requiring no changes

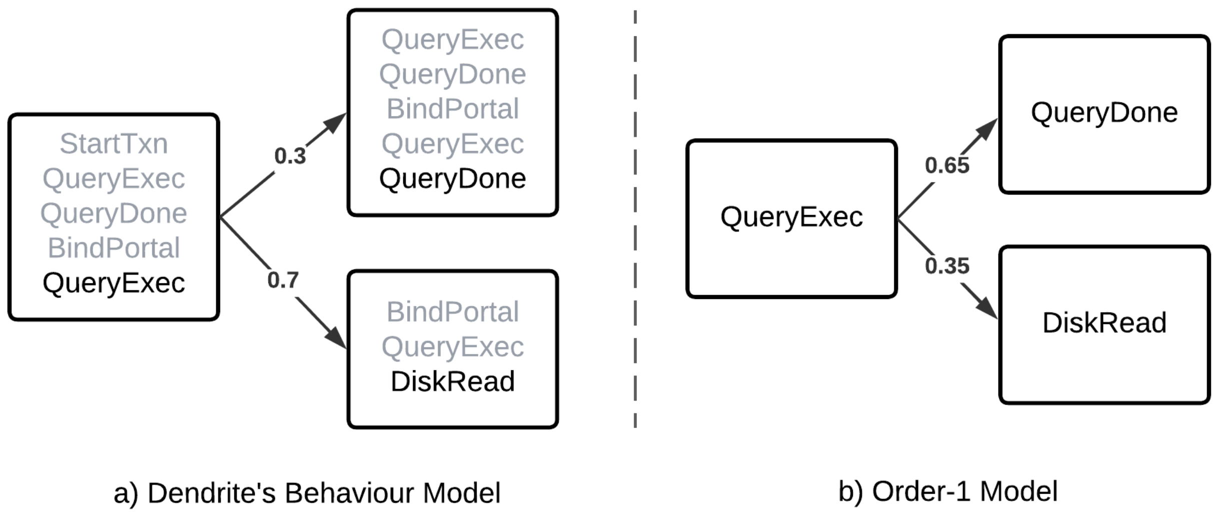


Figure 5.9: a) the disk read-portion of Dendrite's behaviour model for a YCSB-C PostgreSQL worker, and (b) the relevant portion of an order-1 behaviour model constructed from the same data.

to source code.

Integrating Dendrite-Pin with SQLite required little effort; I simply configured Dendrite-Pin to intercept the `sqlite3Select` and `sqlite3Update` functions. As with PostgreSQL and MariaDB, Dendrite-Pin requires no code changes to integrate with SQLite.

5.3.4 Behaviour Model Microbenchmarks

Dendrite's behaviour models capture complex database system behaviour patterns (Chapter 4.2.2). Dendrite detects when fewer prior events affect an event transition's probability and simplifies the model, reducing its memory consumption. I now show these models' utility in accurately representing system behaviour and their automatic simplification property.

Behaviour Model Representativeness

As Dendrite's behaviour models encode prior event sequences, they capture more detailed event context than single-order behaviour models. To demonstrate this property, consider the following PostgreSQL case study.

I obtained a model of YCSB-C PostgreSQL worker behaviour by executing a 5 client workload against a 1.75 GB PostgreSQL database. The database is equipped with a 500 MB buffer pool, thereby inducing buffer pool cache misses. Since the entire database fits in the machine's 32 GB of main memory, buffer pool cache misses are handled by the operating system's filesystem cache. Hence, I emulated a machine with 500 MB of main memory to match the buffer pool size by adding latency proportional to disk seek time to cache misses.

Figure 5.9a shows the cache miss portion of Dendrite's behaviour model for a PostgreSQL worker. The nodes describe the prior event sequence, with the current event shown in black and the previous events shown in gray. In the left node, the current event is QueryExec, which follows the previous events StartTxn, QueryExec, QueryDone, and BindPortal. There is a 30% chance of directly moving to the QueryDone event, forming the new event sequence QueryExec, QueryDone, BindPortal, QueryExec, QueryDone, and a 70% chance of incurring a CacheMiss.

Note that the prior event sequence contains a transition from StartTxn, QueryExec to QueryDone without a cache miss; this transition is guaranteed because the first query in a transaction is always a BEGIN command, which does not access data and therefore cannot incur a cache miss. As the last query is a COMMIT command, cache misses may only occur on queries between these commands. Dendrite's enhanced behaviour models capture this fact, while the equivalent single-order model in Figure 5.9b fails to do so. This lack of context leads to poor estimates of cache miss rates, reducing the effectiveness of adaption rules.

I equipped Dendrite with the following adaption rule:

```
get_transition_prob_diff([‘StartTxn’; ‘QueryExec’, ‘QueryDone’ ‘BindPortal’,
‘QueryExec’], ‘DiskRead’) > 2.0 -> ‘increase_buffer_pool’()
```

Specifically, if the probability of a cache miss on a YCSB-C query has increased by a factor of 2 compared to when a buffer pool is adequately provisioned, Dendrite executes the `increase_buffer_pool` UDF. This UDF alters PostgreSQL's configuration and increases the buffer pool to 8 GB, thereby increasing the proportion of cache hits and improving performance. Dendrite immediately detects that system behaviour differs from expected, matches this adaption rule, and deploys the UDF. After restarting the database and executing the workload for 10 minutes, performance is improved by 5×.

Dendrite's models provide increased model accuracy by better capturing event patterns.

In doing so, Dendrite enables simplified adaption-rule composition and expressivity in responding to database system behaviour differences.

Reduction Experiments

I now empirically evaluate Dendrite's behaviour model reduction properties through a targeted microbenchmark with event-transition probability distributions.

In this microbenchmark, Dendrite captures 10 distinct events that have uniform event transition probability between each pair of events. Dendrite is configured to capture a maximum prior sequence length of 3 in its behaviour models. Initially, Dendrite tracks event transition probabilities from each triplet of events (E_i, E_j, E_k) to subsequent event E', yielding a total of $10,000$ unique transitions. As Dendrite obtains samples from the running database system, it determines that transition probabilities between pairs of events are independent of any prior events and simplifies its model accordingly. Since these checks require $\approx 10,000$ samples to obtain high confidence per the ϵ, δ requirements outlined in Chapter 4.2.3, *all* sequences are reduced to $(E_k) \rightarrow E'$ once Dendrite has observed 100M `record_event` samples.

In practice, it is unlikely that each system event transitions to every other event. Thus, I also tested Dendrite capturing 10 distinct events with Zipf-distributed event transition probabilities; event ID i was far more likely to be be followed by event ID $i+1$ than any other event.

Using a Zipf($\rho = 10$) distribution of event transition probabilities, Dendrite reduced triplets of the form $(E_{j-1}, E_j, E_{j+1}) \rightarrow E'$ to $(E_j, E_{j+1}) \rightarrow E'$ for all E' with 125,000 samples of event transitions over 10 unique events. Dendrite performs these reductions with fewer `record_event` calls because it obtains more samples for the popular event transitions from event ID i to event ID $i+1$, giving it greater confidence in the captured transition probabilities. Similarly, Dendrite is able to reduce the popular event sequence transitions to order-2 over 100 unique events with 2 million samples. Notably, Dendrite does not reduce the event sequence lengths to order-1 because it does not have enough samples for rare transitions, e.g., $(E_i, E_j, E_{j+1}) \rightarrow E$ where $E_i \neq E_{j-1}$. Obtaining enough samples for these rare transitions requires a large number of overall system samples given the Zipf distribution.

I also considered a Zipf($\rho = 2$) distribution of event transition probabilities. As Dendrite acquires fewer samples of the popular event transitions than with the more skewed Zipf($\rho = 10$) distribution for the same number of event transitions, it reduces sequence lengths of popular transitions to order-2 at 500,000 samples for 10 unique events and 1B samples for 100 unique events.

5.4 Discussion

The case studies in this chapter show that Dendrite effectively adapts multiple database systems to optimize for their circumstances, improving performance. These results lead to interesting insights about the nature of system adaptivity and its generalization across database systems, which I now discuss.

As noted in Chapter 5.1, PostgreSQL, MariaDB, and SQLite exhibit significantly different characteristics. For example, PostgreSQL and MariaDB differ in how they support client connections (processes vs threads), their support for parallel query execution, and their management of the write-ahead log through checkpointing. Dendrite's logging-call interception and behaviour difference detection techniques are effective across database systems despite these dissimilarities, showing the generality of Dendrite's techniques.

Interestingly, the case studies show that an adaption rule composed for one database system can inform the adaption rules for another. In fact, the criteria of each PostgreSQL adaption-rule in the case studies can be used in MariaDB by translating PostgreSQL events to MariaDB. Reusing rule criteria significantly decreases the effort of rule development.

Thus, transferring an adaption rule from one database system to another requires an event equivalence layer between the systems and translating the source system's UDFs to support the destination system. In the case studies, transferring rules was simple as it required only information easily obtainable through system documentation.

Through this discussion and the results, we observe that there are three key components for database system adaptivity, which Dendrite provides:

1. A means to monitor system behaviour and detect differences
2. A means to match behaviour differences to adaption strategies
3. A means to deploy adaption strategies on the targeted system

For a case study to be successful, each of these components must work effectively in tandem with each other. If a behaviour difference is not detected, then no adaption will be deployed. If an inappropriate adaption strategy is selected, the response may worsen matters. Finally, there must be an effective way to deploy the chosen strategy to the underlying database system rapidly. Dendrite's success in these case studies demonstrates the effectiveness of its design choices in each of these areas.

5.5 Summary

This chapter demonstrates the effectiveness of Dendrite in bolting adaptive capabilities onto popular, industrial-strength database systems through a robust suite of case studies. Dendrite integrates easily with each system, despite their distinct characteristics. Adaption rules are expressive and handle a wide variety of behaviour differences while remaining portable between database systems that exhibit similar characteristics. Microbenchmarks show that Dendrite's overheads are low in terms of both memory consumption and performance degradation, making it suitable for intensive database system deployments.

Chapter 6

Related Work

Dendrite is unique in its ability to (*i*) capture a live database system's behaviour in a system-agnostic way, and (*ii*) respond to behaviour differing from expectations to improve system performance. Thus far, the research community's focus has been limited to system behaviour modelling [58, 83, 90, 124, 125] and next-generation, natively adaptive database systems [2, 3, 39, 93, 110, 111]. This chapter provides an overview of prior work and its limitations, with a focus on how it differs from Dendrite.

Given the challenges that administrators face while managing database systems and the time-sensitive nature of remedying performance problems, there has been substantial research effort into alleviating this burden through system-management assistance tools and adaptive database systems. I now present related work in these areas and their differences from Dendrite (Figure 6.1).

6.1 System Management Assistance Tools

This first category consists of research efforts that *assist* administrators in monitoring their systems and configuring them. While these tools provide helpful output to guide administrator decisions, administrators ultimately diagnose, remedy, and configure the underlying system.

This category may be further subdivided into tools that gather and expose simple statistics, those that capture higher-level system behaviour and alert administrators when changes occur, and those that directly recommend system modifications. Importantly, none of these tools directly effect changes, which is one of Dendrite's core contributions.

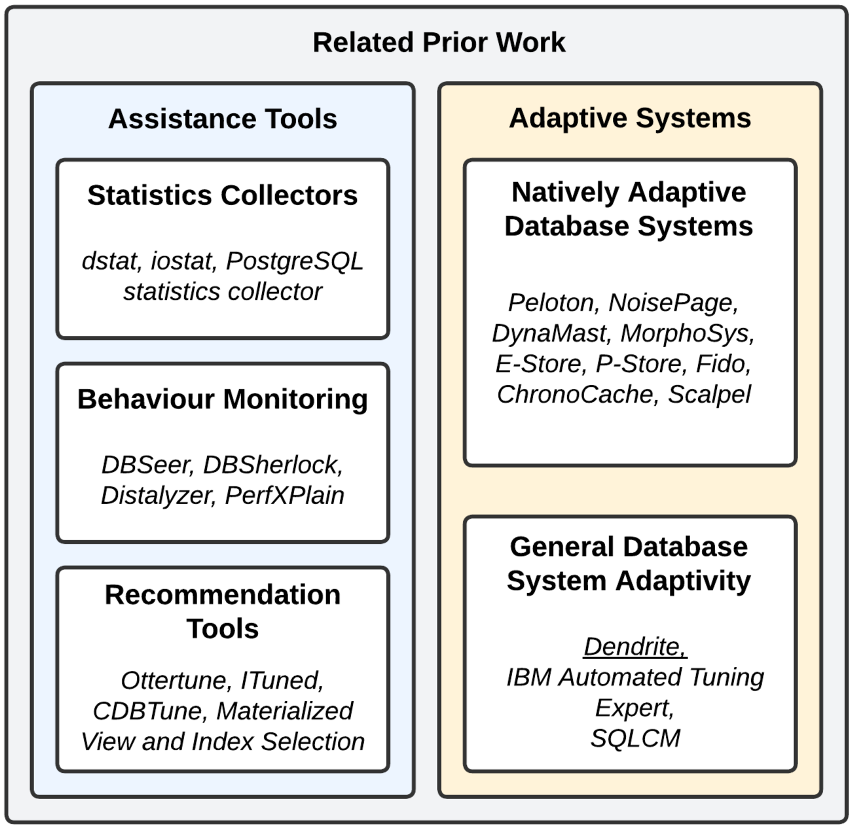

Related Prior Work
Assistance Tools
Statistics Collectors
dstat, iostat, PostgreSQL statistics collector
Behaviour Monitoring
DBSeer, DBSherlock, Distalyzer, PerfXPlain
Recommendation Tools
Ottertune, ITuned, CDBTune, Materialized View and Index Selection
Adaptive Systems
Natively Adaptive Database Systems
Peloton, NoisePage, DynaMast, MorphoSys, E-Store, P-Store, Fido, ChronoCache, Scalpel
General Database System Adaptivity
Dendrite,
IBM Automated Tuning Expert,
SQLCM

Figure 6.1: A taxonomy of prior work related to Dendrite.

6.1.1 Simple Statistics Collection

As resource contention is well-known to degrade the performance of database systems and applications, many tools have been developed to extract and expose operating system and database resource consumption metrics. On Linux systems, `vmstat` [119], `iostat` [53], and `netstat` [9] are commonly used to retrieve memory, disk I/O, and network I/O statistics respectively. Administrators use these statistics to determine when system capacity is exceeded, and either provision additional resources or reduce load to restore normal performance by eliminating resource bottlenecks. Since it is desirable to monitor the usage of multiple resources simultaneously, `dstat` [120] and `sar` [54] capture several key operating system resource statistics using a single application and present them in a unified output stream.

As database systems are heavy consumers of system resources, they often provide their own tools to explain their resource utilization to administrators. For example, PostgreSQL's statistics collector [60] records the number of pages read from and written to disk, page access cache hit ratios, index usage counts, and other pertinent information useful for administrators to monitor and optimize their database. Other popular data systems, from databases [23, 47] to message brokers [46] to streaming systems [45] similarly expose key metrics.

While these aggregate system metrics are helpful in obtaining a high-level perspective of overall resource utilization, it is desirable to attribute this usage to individual system components. Therefore, developers often use profiling tools (e.g., gprof [57], `perftools`) to find the slow or resource-intensive sections of their code and optimize them. These tools add significant overhead and hamper the performance of the profiled system, which relegates them to post-hoc use when investigating a known problem offline.

Dendrite uses `dool` [120] (formerly `dstat`) to extract operating system metrics from an online system but goes beyond `dool`'s capabilities by encoding these metrics in comprehensive behavioural models. These models include low-level system tracing and enable code-level resource-consumption attribution with very low overhead. Dendrite uses this tracing and resource usage information to effect appropriate system adaptions in response to changes in workloads or deployment environments. The case studies in Chapter 5 demonstrate the effectiveness of Dendrite's approach; however, Dendrite can be easily enhanced to capture additional metrics from sources other than `dool` if desired.

6.1.2 System Behaviour Analysis

It is difficult for administrators to determine whether a system is performing well using metric statistics alone; resource consumption naturally fluctuates during system execution, so administrators must determine whether a change in these metrics is *unexpected*. Therefore, researchers have developed tools that capture higher-level data system behaviour to more easily detect and remedy system anomalies.

DBSeer [88, 124] builds white-box and black-box models of database system processing techniques, such as disk I/O, to predict the system resources required to process a client workload at a target throughput rate. Specifically, DBSeer builds statistical models of memory consumption, disk flushing, CPU utilization in the MySQL and PostgreSQL RDBMSs, and uses them to predict (i) the throughput of the system under a fixed set of resources and a given workload mix, (ii) the resources required of each modelled type to process the target workload at the desired throughput rate. Like Dendrite, DBSeer uses debug logs and operating system metrics. However, DBSeer assumes knowledge of the underlying database system in its white-box model construction, extracting DBMS-specific counters not available in other systems and relying on domain knowledge in its model construction, while Dendrite does not. Furthermore, as DBSeer's focus is on resource prediction, it does not detect system behaviour differences nor effect system adaptions in response.

DBSherlock [125] associates periods of abnormal system behaviour with predicates constructed over captured debug logs, operating system metrics, and database system metrics. Given a period of anomalous or poorly-performing system behaviour, DBSherlock determines the ranges of values in its extracted metrics most associated with the anomaly. For example, DBSherlock may determine that poor performance is associated with an increase in CPU waits, query latency, and disk writes. While Dendrite also uses predicates over system behaviour differences, these predicates are used by Dendrite's adaption rules to effect changes that remedy the underlying issue, which DBSherlock does not support. Moreover, Dendrite captures behaviour in a fully system-agnostic way, as it does not rely on DBMS-specific counters or log preprocessing, unlike DBSherlock. These features ensure Dendrite's approach is generalizable and keep its overheads low (Chapter 5.3.1).

Distalyzer [90] preprocesses log files to obtain information about the frequency and occurrence times of system events, alongside value ranges of variables outputted in the files. Given a population of log files obtained over multiple system executions that are labelled according to good and bad performance, Distalyzer determines which events, occurrence times, and values are statistically associated with the poorly performing executions. Although Distalyzer operates over system logs from arbitrary systems, Dendrite has key differences. First, Distalyzer requires that log files are emitted to disk, while Dendrite intercepts log calls in memory to avoid the performance overheads of detailed log file materialization. Second, Distalyzer requires multiple log files to determine statistical behaviour differences, while Dendrite detects these differences online during a single system execution. Lastly, Dendrite also uses these behaviour differences to determine how to adapt the system to the workload at hand, while Distalyzer focuses solely on behaviour difference detection.

PerfXplain [72] uses debug logs and performance counters obtained from Ganglia [30] to determine why the performance of a pair of MapReduce jobs differed. It provides a query language to look over previous runs of MapReduce jobs with similar characteristics to the pair of interest, and then outputs explanatory information to show what features, constructed over the counters and logs, are most correlated with the difference. PerfXplain assumes that the tracing information it requires is readily available, unlike Dendrite, which obtains it in a system-agnostic, low-overhead fashion.

Lprof [129] reconstructs the control flow of distributed applications. Using static analysis of Java bytecode, Lprof determines the format of various log messages. It interprets request identifiers from outputted messages and links messages from different system components. At runtime, log files are analyzed to determine the performance characteristics of requests that span the system, storing them in a queryable database. Dendrite differs from Lprof as it intercepts the information it needs without static analysis. and enables adaptive responses to behaviour differences.

Dendrite's behaviour extraction techniques operate on live database systems rather than depending on offline static analysis. Offline analysis determines only which code paths and systems events the system *may* access, not which ones are popularly exercised in the deployment environment. Dendrite's online techniques enable it to characterize system behaviour in situ, which static analysis alone cannot support.

PerfAugur [101] detects anomalies in system telemetry using robust statistics [100]. It assumes that this telemetry is encoded in a single relational table and finds predicates on the table for which an aggregate over a subset differs significantly from the aggregate computed over the whole relation. By contrast, Dendrite extracts all of the information it requires *online* from a running data system.

Pivot Tracing [83] allows developers to dynamically enable system tracing by expressing queries over the output of pre-defined tracepoints. These queries can group and filter based on request flow, i.e., what events happened before other events, which relies on metadata propagation between the tracepoints. The supplied queries determine which tracepoints are "switched on," such that the information needed to answer the query is extracted with low overhead. Tracepoints must be declared, however, by expert users or developers *a priori*, whereas Dendrite extracts the information it requires from built-in debug logs.

CloudSeer [126] is an anomaly detection tool that operates over outputted system debug logs. It constructs an automaton of logging offline and detects when the log-ordering patterns it observed are violated. Dendrite goes beyond by enabling broader behaviour comparisons using statistics of captured system events. For example, Dendrite can determine if there is a lower proportion of cache hits or a change in how clients issue requests [49], which is beyond CloudSeer's capabilities. Moreover, Dendrite builds its models in memory by intercepting log calls and can therefore construct finer-grained models with lower overhead than CloudSeer.

The goal of anomaly detection is to find patterns in subsets of data that do not match the overall behaviour of the whole [14]. These techniques are popularly used to detect credit card fraud and network intrusions. Hence, behaviour difference detection may be viewed as a particular instance of anomaly detection on a time series of metrics extracted from the operating system and debug logs. However, anomaly detection techniques are highly workload-specific and vary significantly according to their domain [14]; therefore, while Dendrite shares high-level similarities with other tools in this space, it is fundamentally novel in how it captures system behaviour and compares it.

Given the widespread challenges system administrators face in managing their data systems, there has also been significant industry focus on system monitoring tools [7,28,38,70,98,104,105]. These tools rely on instrumentation in the core libraries used by distributed applications to trace the performance and characteristics of requests as they move through the system. For example, thread management operations and remote procedure call (RPC) libraries are commonly instrumented to attach identifiers to requests. These identifiers are used to trace requests that span computing nodes in a distributed system. Metrics characterizing the performance of these requests are sampled and exported, displaying them to developers and users in rich visualizations called dashboards. These systems deploy sophisticated visualization techniques to present information to developers to more easily detect performance problems and remedy issues. However, unlike Dendrite, these tools do not adapt the system according to the exported data.

The Mystery Machine [19] unifies logging events from multiple sources and determines causal relationships among them (e.g., happens-before [74], mutually exclusive) using large-scale mining over event samples. These causal models are used to perform common investigation tasks such as determining resource slack and critical-path performance analysis. While Dendrite also uses log files as a key information source, it differs from the Mystery Machine in both how it extracts its information and the types of analysis it conducts.

Statistical debugging techniques [40, 78] inject code into an application to evaluate and report boolean predicates at points of interest. The application is then evaluated repeatedly offline to determine which predicates are statistically associated with a fault or failure. While Dendrite may also be used to investigate system failure conditions, it does so using logs and metrics and not by injecting predicates into the application for offline evaluation. As Dendrite's low-overhead analysis is executed on live production systems, it does not require multiple system executions to locate failures.

Amoeba [79] locates performance bugs in database systems by generating semantically equivalent SQL queries and determining if there is a significant difference in their performance. Apollo [69] similarly detects performance regressions across different versions of the same DBMS using SQL fuzzing techniques. Dendrite differs from these systems in that they aim to capture performance regressions in generated SQL queries, while Dendrite captures system behaviour to detect performance degradation in live systems.

The software engineering research community has developed tools that use debug logs to pinpoint load-testing and memory-consumption problems in code [67,68,109]. These tools mine outputted debug log files for control-flow relationships and correlate them with higher-level system metrics (e.g., memory usage). If the control flow or memory consumption is abnormal relative to other parts of the log file, it may indicate a bug in the code. Unlike Dendrite, which intercepts logging *function calls*, these systems require log files to be materialized on disk which vastly increases their performance overheads. Moreover, these tools require administrator intervention to remedy discovered issues, while Dendrite enables adaptions that address them.

Despite the prior work on database system behaviour analysis tools, Dendrite differs from this class of work in (*i*) its system-agnostic information extraction techniques, and (*ii*) its ability to respond to database system behaviour changes through adaption rules.

6.2 System Recommendation Tools

All of the prior approaches are limited to providing only high-level views of system behaviour and explanations of discrepancies; they do not suggest steps for an administrator to take that would improve system performance nor take action independently. Instead, administrators must use the output of these tools to devise an appropriate response, a daunting task given the complexity of database systems [117,125]. Therefore, researchers have developed recommendation tools that monitor database system performance and suggest materialized views [62,86], indexes [17,61], and parameter configurations [37,117,127] to optimize performance. I will first present an overview of each tool's functionality and then describe how Dendrite fundamentally differs from them.

Much research has focused on selecting views to materialize within databases [16,62,86]. As queries can access pre-computed results directly from materialized views, they can avoid processing the underlying data tables entirely if the right views are materialized, leading to significant performance improvements. However, these views must be updated whenever the data over which they are built is modified; a view selection strategy must carefully balance maintenance cost against query processing time improvements for the workload. Thus, view selection tools monitor the data system to obtain a representative sample of client query workloads, calculate the potential improvements of candidate views, consider their maintenance costs, and deploy the views most likely to improve system performance.

Similarly, using appropriate database indexes for query workloads can significantly improve query latency [16,17,61]. Like materialized views, indexes are an auxiliary structure that must be updated if the data they index is modified. Thus, a similar balance must be struck, but with an additional challenge; there are many types of database indexes, each of which is specialized for workloads with different access characteristics.

Database systems are notorious for having vast numbers of configuration parameters that govern many aspects of how the system processes client requests [117]. These parameters play a critical role in system performance. Unfortunately, there is no single set of configuration values that optimizes performance across all workloads, and the sheer scope and complexities of parameter relationships push the parameter selection problem beyond human reasoning capabilities [94, 117]. To meet these challenges, researchers have developed many different parameter selection (colloquially, "knob tuning") tools [37,55,84,85,117,118,127], of which iTuned [37], Ottertune [117], and CDBTune [127] are representative works.

ITuned [37] optimizes database system parameters using an iterative, experiment-based, trial-and-error approach: it selects values for each configuration parameter, executes a provided, representative workload against the database system, and then chooses the next set of parameter values based on the observed performance and that of the prior experiments. As ITuned models the performance-parameter relationship using a Gaussian process [99], it selects parameter values based on their expected improvement over the current parameters using the model.

Like ITuned, Ottertune [117] uses a Gaussian process to predict how a database will perform on a given workload under various parameter settings. However, Ottertune improves on ITuned by maintaining a global repository of training results from other users and databases, which it bootstraps to reduce training time. Ottertune further mitigates tuning time by reducing the number of knobs that must be considered using dimensionality reduction.

A key limitation of Ottertune is that Gaussian process regression is susceptible to selecting parameters that optimize performance within a subregion of the parameter space (local optimum) instead of overall (global optimum) [127]. CDBtune [127] addresses this limitation through deep reinforcement learning, which intrinsically balances exploring the parameter space and exploiting the knowledge it has gleaned. Moreover, CDBTune directly accounts for changes in the database over time and in response to parameter changes, unlike Ottertune.

Knob tuning tools are complementary to Dendrite's behaviour modelling and adaptions. While these tools can excel at selecting configuration parameter values for a representative workload, they do not provide behaviour difference detection to determine *when* to change parameter values. Concisely, these tools define *what* should be changed (and to which values), but not *when*. Hence, these knob tuning tools could be exploited to generate adaption rules for Dendrite, while Dendrite performs the difference detection and generalized adaption deployment. This idea is explored further in Chapter 7.

These recommendation tools specialize in a single stratum of database system behaviour tracking and optimization (parameter, view, or index optimizations). In contrast, Dendrite employs a *universal* approach for capturing database system behaviour and responding accordingly. Each area should be viewed as a particular type of adaption that Dendrite can deploy.

6.3 Adaptive Database Systems

Every monitoring and recommendation tool mentioned thus far shares a common limitation; unlike Dendrite, they all require administrators to deploy appropriate responses. There are two key branches of work that address this limitation: next-generation, natively adaptive database systems and generalized database adaption techniques.

6.3.1 Natively Adaptive Database Systems

Natively adaptive database systems are new, next-generation database systems with novel architectures specifically designed for adaptivity [93, 94]. As such, retrofitting existing database systems with the adaptivity capabilities proposed by this line of research requires tremendous engineering effort [94].

Prior work by Pavlo et al. [95] outlines the components a natively adaptive database system requires for complete autonomy. In particular, a fully autonomous database system must have a *workload forecasting* component that predicts the upcoming workload, a *performance modelling*[1] component that predicts the performance of the system under various configurations and workloads, and an *action planner* that determines which adaptions the system will deploy given the forecast and performance models. A fully autonomous system does not require administrator intervention of any kind, other than to set high-level optimization criteria (e.g., maximize system throughput, minimize cloud infrastructure costs). While fully database autonomous systems are years of research away, this work also enumerates increasing levels of autonomy along the way:

[1]In the paper, this component is called "behaviour modelling", but it focuses on predicting the performance and resource consumption of small, isolated system components (e.g., hash-join initialization time). To avoid confusion with Dendrite's behaviour modelling, this work is differentiated using the *performance modelling* label.

1. **Level 0:** human insight and direct intervention are required to tailor the system to its workload and environment.
2. **Level 1:** The database system provides recommendations to the administrator on how it should be configured based on the workload, but administrators must accept these changes and determine when they should be deployed.
3. **Level 2:** The database system collaborates with the administrator to adaptively configure its parameters and subsystems.
4. **Level 3:** Individual database system components are autonomous, responding to their environment without any administrator involvement (e.g., autoscaling on cloud providers). There is no long term planning, or holistic decision-making.
5. **Level 4:** The administrator provides high-level direction to the database system on what to optimize, and small hints (e.g., anticipated workload spike) to influence system decision making. Otherwise, the database system self-manages its components holistically and responds automatically to the workload and environment.
6. **Level 5:** The administrator sets an optimization objective and the database system self-manages to achieve it. No other administrator intervention is required.

In this thesis, any database system operating at autonomy level 2 is considered adaptive, in line with Definition 1. Concretely, a database system is adaptive if, while running, it changes its system configuration, physical design, or resource allocation without administrator intervention (i.e., it deploys adaptions). Dendrite's goal is to enrich existing non-adaptive database systems with adaptive capabilities, empowering them to reach level 2 autonomy through its bolt-on functionality.

Pavlo et al. [93] propose a natively adaptive database architecture, Noisepage[2], that captures query workload characteristics and changes the data layout of tuples in tables to either row or column orientations to best suit the query workload. NoisePage captures query arrival rates and forecasts upcoming queries to preemptively change table layouts in anticipation of future workload shifts.

MB2 [82] is the performance-modelling component of the Noisepage database system [21]. MB2 decomposes database system functionality into small *operating units* (OUs), whose resource consumption and performance characteristics are modelled with machine learning. A grid-search technique is used to sweep the input features of each OU to

[2]Previously called Peloton.

learn these performance characteristics. This performance prediction framework, when combined with a workload forecasting module [81] is intended to act as the building blocks for a fully autonomous database system. While Dendrite also models database system performance characteristics, it does not require developers to demarcate or declare OUs manually. Dendrite's focus in on determining which aspects of overall system behaviour have changed and responding to these changes.

E-Store [111] uses operating system statistics to determine when a database cluster's load is imbalanced and migrates partitions between database nodes to restore balance. If the system is overloaded, E-Store elastically adds database nodes to the system, reducing node count when the added capacity is no longer required. E-Store is reactionary and does not preemptively move partitions, which may lead to poor performance in the face of predictable patterns. P-Store [110] extends E-Store and addresses this limitation by building workload models to predict upcoming workload patterns and move partitions accordingly.

Distributed transactions in database systems are well-known to cause severe overheads and performance degradation due to two-phase commits [26]. Therefore, there has been significant research focus on enabling online partition migration within database systems to minimize [26] or eliminate [2, 3, 27, 76] distributed transactions.

Database cracking [65] creates database indexes by automatically breaking data tables into pieces based on the columns accessed in the predicates of executed queries. These pieces are labelled according to the range of values they contain for the accessed column. When subsequent queries with predicates operating over the same columns are submitted, their performance is improved as they can be answered using the smaller pieces containing values of interest rather than scanning the whole table. The key advantage of cracked indexes over traditional indexes is that they are constructed on-demand in response to the workload and do not require a priori knowledge of access patterns.

As database caching systems play a significant role in overall performance, researchers have developed predictive caching systems to improve cache hit rates [12, 51, 92]. These systems build models over submitted client queries and predict which queries are likely to arrive next. In doing so, they can ensure that those queries' results are placed into the cache ahead of time, providing a cache hit and improving performance.

The adaptivity enabled by each of these systems cannot be "bolted-on" to other database systems, unlike Dendrite's adaptive framework. By relying solely on debugging logging and operating system metrics, Dendrite easily generalizes to arbitrary database systems, as shown through integrations with PostgreSQL, MariaDB, and SQLite (Chapter 5).

6.3.2 Generalized Database Adaptivity

Prior proprietary work has also recognized the dearth of adaptive capabilities in existing database systems and proposed adaptive techniques that can generalize across database systems. Unlike Dendrite's bolt-on integration, these techniques rely on per-system behaviour extraction scripts and instrumentation, which greatly increases the effort required to integrate with them.

IBM's Automated Tuning Expert [121] relies on custom adaptors over database information sources (metrics, debug logs, catalog) to extract system events. These events are matched against tuning plans, which codify the tuning expertise of administrators and are used to adapt the system to the workload. In contrast to this work's reliance on system-specific adaptors, Dendrite bolts-on to database systems to intercept logging calls in-memory, thereby mitigating overheads. Dendrite also differs from this closed-source work in the form its adaption rules take, its powerful, context-capturing behaviour modelling, and its support for binary instrumentation.

SQLCM [15] instruments SQL Server to extract database metrics and match them against configurable event-condition action rules that tailor database processing. Unlike the proprietary SQLCM, Dendrite extracts all of the information it requries from system-agnostic debug logging calls and `libc` metrics, ensuring that it integrates easily with a broad range of database systems.

Both of these works are proprietary and closed source, precluding comparison with Dendrite.

Beyond database systems, Dendrite shares some high-level similarities with runtime verification systems [8, 10, 18]. Dendrite differs from this work through its adaption rules that use proportion-based behaviour differences from an expected baseline model and its automatic extraction of live system events from logging calls, which obviates instrumentation.

Chapter 7

Conclusion and Future Work

Adaptive database systems are a promising approach to alleviate the heavy administrator burden of maintaining and optimizing database systems. However, many popularly used, industrial-strength database systems lack robust adaption capabilities, trapping administrators in a Sisphyean monitoring and optimization cycle. This thesis provides techniques that bolt-on to existing database systems and empower them with adaptive capabilities. In this chapter, I summarize the core contributions of the thesis (Chapter 7.1) along with limitations and opportunities for future research (Chapter 7.2).

7.1 Contributions

Developing a bolt-on adaption framework for general database systems requires addressing several challenging research objectives. In line with the challenges outlined in Chapter 1.2, I presented a system-agnostic behaviour extraction framework with low overhead that detects important differences and enables adaptive responses through its comprehensive support for a wide range of database system augmentations.

The modelling techniques presented in Chapters 3 and 4 are system-agnostic and do not rely on system-specific knowledge to capture database system behaviour. By intercepting debug logging, operating system metrics, and `libc` calls, Dendrite builds robust behaviour models that capture fine-grained behaviour information. Dendrite is also able to operate in environments with limited debug logging due to its innovative use of binary instrumentation, ensuring that its techniques are broadly applicable.

Chapters 3 and 4 also describe behaviour model comparison techniques that highlight significant differences in the extracted models. Although Dendrite can perform these comparisons in a system-agnostic fashion, it can also exploit domain knowledge provided in the form of attention-focusing mappers to further improve its ranking of difference importances.

Dendrite uses adaption rules to respond to detected behaviour differences. Adaption rules are expressive and handle a wide variety of system behaviour differences, while remaining intuitive to compose due their conjunctive nature, built-in data extraction functions, and tagging functionality. Chapter 5 shows the power of these rules in action, demonstrating their applicability through realistic case studies with representative workloads and multiple database systems. Adaption rules may be ported between database systems that exhibit similar characteristics using a simple abstraction layer that tracks equivalent events in the systems, further decreasing the challenge in composing these rules.

Importantly, Dendrite achieves all of the above techniques while incurring low overhead. As Dendrite builds its models in memory by intercepting debug logging and function calls, it avoids the traditional modelling overheads associated with writing fine-grained information to disk. Dendrite's models are compact and consume little memory, ensuring that system resources are dedicated to the critical task of query processing.

The future for adaptive database systems is bright, with further interesting work to be done. This thesis is a stepping stone towards realizing fully autonomous, generalized database system adaptivity.

7.2 Future Work

While Dendrite is a promising step towards generalized system adaptivity, future research can further increase its scope and utility.

First, Dendrite relies on developers and administrators to provide it with adaption rules to intelligently respond to detected behaviour differences. While some adaption rules are common knowledge among wizened administrators, e.g., increasing buffer pool size to improve cache hits or avoiding checkpointing during periods of high load, other adaption scenarios require more consideration. Hence, while Dendrite reduces administrator burden, it does not eliminate it entirely. It remains interesting future work to determine if adaption rules can be automatically generated, perhaps using an augmented knob-tuning or reinforcement learning tool. Similarly, it would be desirable to have Dendrite automatically learn event importances that enhance its behaviour comparisons.

Second, future research could target Dendrite's footprint to further reduce its overheads. One direction could further mitigate tracing overhead to improve database system throughput. Another could focus on ensures that Dendrite's tracing does not alter the behaviour of the executing system. Although Dendrite's per-thread models avoid synchronization primitives that may lead to heisenbugs [89], the scant overheads it induces could lead to differences in externally observed system behaviour. Thus, future research may investigate techniques to deterministically "replay" system execution given Dendrite's behaviour observations.

Third, Dendrite builds its models on a per-thread basis to eliminate synchronization overheads, but database systems can collaboratively process client requests across threads, processes or distributed machines. Improving Dendrite's models by "stitching" together these distinct models into a combined model that captures collaborative work patterns would increase its applicability, particularly in the popular distributed database systems space.

Fourth, although Dendrite's adaptions tailor database system processing, they cannot fundamentally change system functionality. For example, Dendrite cannot feasibly convert the row-oriented PostgreSQL database system to a column store. Binary instrumentation *can* induce novel adaptions, but injecting large-scale functionality is infeasible due to the deep knowledge of database system internals it requires. Note that adding a layer of indirection using a load balancer or query router widens the scope of Dendrite's adaptions. For example, Dendrite can adapt query router decisions based on the workload (e.g., Chapter 5), transferring targeted queries to another database system with the desired functionality. Hence, while Dendrite supports a wide variety of adaptions, the nature of UDFs and database systems have important considerations in designing an adaptive solution. These considerations may be addressed through future work.

References

[1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 967–980, 2008.

[2] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. Dynamast: Adaptive dynamic mastering for replicated systems. In *Proc. 36th Int. Conf. on Data Engineering*, pages 1381–1392, 2020.

[3] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. Morphosys: Automatic physical design metamorphosis for distributed database systems. *Proc. VLDB Endowment*, 13(13):3573–3587, 2020.

[4] Akamai. New study reveals the impact of travel site performance on consumers. https://www.akamai.com/us/en/about/news/press/2010-press/new-study-reveals-the-impact-of-travel-site-performance-on-consumers.jsp, 2010.

[5] Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Janus: A hybrid scalable multi-representation cloud datastore. *IEEE Trans. Knowl. and Data Eng.*, 30(4):689–702, 2018.

[6] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. MacroBase: Prioritizing attention in fast data. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 541–556, 2016.

[7] Paul Barham, Rebecca Isaacs, and Dushyanth Narayanan. Magpie: online modelling and performance-aware systems. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, pages 85–90, 2003.

[8] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. *Introduction to Runtime Verification*, chapter 1, pages 1–33. Springer International Publishing, Cham, 2018.

[9] Fred Baumgarten, Matt Welsh, Alan Cox, Tuan Hoang, and Bernd Eckenfels. Netstat manual page. https://linux.die.net/man/8/netstat, 2021.

[10] Eric Bodden, Patrick Lam, and Laurie Hendren. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In *Proc. 10th Int. Conf on Runtime Verification*, pages 183–197, 2010.

[11] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Proc. 2nd Biennial Conf. on Innovative Data Systems Research*, pages 225–237, 2005.

[12] Ivan T. Bowman and Kenneth Salem. Optimization of query streams using semantic prefetching. *ACM Trans. Database Syst.*, 30(4):1056–1101, 2005.

[13] Sharma Chakravarthy, Vidhya Krishnaprasad, Eman Anwar, and Seung-Kyum Kim. Composite events for active databases: Semantics, contexts and detection. In *Proc. 20th Int. Conf. on Very Large Data Bases*, pages 606–617, 1994.

[14] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):1–58, 2009.

[15] Surajit Chaudhuri, Arnd Christian König, and Vivek Narasayya. SQLCM: a contiuous monitoring framework for relational database engines. In *Proc. 20th Int. Conf. on Data Engineering*, pages 473–484, 2004.

[16] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: A decade of progress. In *Proc. 33rd Int. Conf. on Very Large Data Bases*, pages 3–14, 2007.

[17] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *Proc. 23th Int. Conf. on Very Large Data Bases*, VLDB '97, page 146–155, 1997.

[18] Feng Chen and Grigore Roşu. Mop: An efficient and generic runtime verification framework. In *Proc. 22nd ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages & Applications*, pages 569–588, 2007.

[19] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proc. 11th USENIX Symp. on Operating System Design and Implementation*, pages 217–231, 2014.

[20] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. The mixed workload CH-BenCHmark. In *Proc. 4th Int. Workshop on Testing Database Systems*, pages 1–6, 2011.

[21] NoisePage contributors. Noisepage: Self-driving database management system. https://noise.page/, 2022.

[22] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symp. on Cloud Computing*, pages 143–154, 2010.

[23] Oracle Corportation. MySQL. https://www.mysql.com/, 2021.

[24] Transaction Processing Council. TPC-C. http://www.tpc.org/tpcc/, 2018.

[25] Transaction Processing Council. TPC-H. http://www.tpc.org/tpch/, 2018.

[26] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endowment*, 3(1–2):48–57, 2010.

[27] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: A scalable data store for transactional multi key access in the cloud. In *Proc. 1st ACM Symp. on Cloud Computing*, pages 163–174, 2010.

[28] DatadogHQ. Cloud monitoring as a service, datadog. https://www.datadoghq.com, 2021.

[29] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, Massachusetts, 2002.

[30] Ganglia Developers. Ganglia monitoring system. http://ganglia.sourceforge.net/.

[31] SQLite Developers. Most widely used and deployed database engine. https://www.sqlite.org/mostdeployed.html, 2022.

[32] SQLite Developers. SQLite. https://www.sqlite.org/index.html, 2022.

[33] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, page 1–8, 1984.

[34] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. Automatic performance diagnosis and tuning in oracle. automatic performance diagnosis and tuning in oracle. In *Proc. 2nd Biennial Conf. on Innovative Data Systems Research*, pages 84–94, 2005.

[35] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endowment*, 7(4):277—288, 2013.

[36] Ugur Dogrusoz, Erhan Giral, Ahmet Cetintas, Ali Civril, and Emek Demir. A layout algorithm for undirected compound graphs. *Inf. Sci.*, 179(7):980–994, 2009.

[37] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proc. VLDB Endowment*, 2(1):1246–1257, 2009.

[38] Elastic. ElasticStack: ElasticStash, Beats, LogStash and Kibana. https://www.elastic.co/elastic-stack.

[39] Aaron J Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 299–313, 2015.

[40] Anna Fariha, Suman Nath, and Alexandra Meliou. Causality-guided adaptive interventional debugging. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 431–446, 2020.

[41] Fred Fish. The DBUG package. https://dev.mysql.com/doc/refman/8.0/en/dbug-package.html, 2022.

[42] Brady Forest. Bing and Google Agree - Slow Pages Lose Users. http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html, 2009.

[43] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *J. Artificial Intelligence*, 19(1):17–37, 1982.

[44] Apache Software Foundation. Apache Log4j2. https://logging.apache.org/log4j/2.x/, 2020.

[45] Apache Software Foundation. Apache Flink: Stateful computations over data streams. https://flink.apache.org/, 2021.

[46] Apache Software Foundation. Apache Kafka. https://kafka.apache.org/, 2021.

[47] MariaDB Foundation. MariaDB server: The open-source relational database. https://mariadb.org/, 2021.

[48] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Proc. 36th Int. Conf. on Software Eng.*, pages 24–33, 2014.

[49] Brad Glasbergen, Michael Abebe, Khuzaima Daudjee, and Amit Levi. Sentinel: Universal analysis and insight for data systems. *Proc. VLDB Endowment*, 13(12):2720–2733, 2020.

[50] Brad Glasbergen, Michael Abebe, Khuzaima Daudjee, Daniel Vogel, and Jian Zhao. Sentinel: Understanding data systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 2729–2732, 2020.

[51] Brad Glasbergen, Kyle Langendoen, Michael Abebe, and Khuzaima Daudjee. Chronocache: Predictive and adaptive mid-tier query result caching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 2391–2406, 2020.

[52] Brad Glasbergen, Fangyu Wu, and Khuzaima Daudjee. Dendrite: Bolt-on adaptivity for data systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 2726–2730, 2021.

[53] Sebastien Godard. Iostat manual page. https://linux.die.net/man/1/iostat, 2021.

[54] Sebastien Godard. SYSSTAT. http://sebastien.godard.pagesperso-orange.fr/, 2021.

[55] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *Proc. 23rd ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 1487–1495, 2017.

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

[57] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. An execution profiler for modular programs. *Software: Practice and Experience*, 13(8):671–685, 1983.

[58] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, New Jersey, 2011.

[59] PostgreSQL Global Development Group. PostgreSQL. https://www.postgresql.org/, 2018.

[60] The PostgreSQL Global Development Group. PostgreSQL: Documentation: 9.6: The statistics collector. https://www.postgresql.org/docs/9.6/monitoring-stats.html, 2021.

[61] H. Gupta, V. Harinarayan, A. Rajaraman, and J.D. Ullman. Index selection for OLAP. In *Proc. 13th Int. Conf. on Data Engineering*, pages 208–219, 1997.

[62] Alon Y. Halevy. Answering queries using views: A survey. *Proc. VLDB Endowment*, 10(4):270–294, 2001.

[63] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *J. American Statistical Association*, 58(301):13–30, 1963.

[64] Rob J. Hyndman and Yanan Fan. Sample quantiles in statistical packages. *The American Statistician*, 50(4):361–365, 1996.

[65] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *Proc. 3rd Biennial Conf. on Innovative Data Systems Research*, pages 68–78, 2007.

[66] iNotify. INotify man page. https://man7.org/linux/man-pages/man7/inotify.7.html, 2022.

[67] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. An automated approach for abstracting execution logs to execution events. *J. Softw. Maint. Evol.*, 20(4):249–267, 2008.

[68] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. Automated performance analysis of load tests. In *Proc. 25th Int. Conf. on Software Maint.*, pages 125–134, 2009.

[69] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. Apollo: Automatic detection and diagnosis of performance regressions in database systems. *Proc. VLDB Endowment*, 13(1):57–70, 2019.

[70] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An end-to-end performance tracing and analysis system. In *Proc. 26th ACM Symp. on Operating System Principles*, pages 34–50, 2017.

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

[72] Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu. PerfXplain: Debugging MapReduce job performance. *Proc. VLDB Endowment*, 5(7):598–609, 2012.

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

[74] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[75] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proc. 5th ACM Symp. on Cloud Computing*, pages 1–14, 2014.

[76] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. Towards a non-2PC transaction management in distributed database systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1659–1674, 2016.

[77] Greg Linden. Make data useful. http://www.gduchamp.com/media/StanfordDataMining.2006-11-28.pdf, 2006.

[78] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and S.P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Trans. Softw. Eng.*, 32(10):831–848, 2006.

[79] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. Automatic detection of performance bugs in database systems using equivalent queries. In *Proc. 44th Int. Conf. on Software Eng.*, pages 225–236, 2022.

[80] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. ACM SIGPLAN 2005 Conf. on Programming Language Design and Implementation*, pages 190–200, 2005.

[81] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 631—-645, 2018.

[82] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. MB2: Decomposed behavior modeling for self-driving database management systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1248–1261, 2021.

[83] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proc. 25th ACM Symp. on Operating System Principles*, pages 378–393, 2015.

[84] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud. In *Proc. USENIX 2020 Annual Technical Conf.*, pages 189–203, 2020.

[85] Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh, Subrata Mitra, Wolfgang Gerlach, Travis Harrison, Folker Meyer, Ananth Grama, Saurabh Bagchi, and Somali Chaterji. Rafiki: A middleware for parameter tuning of NoSQL datastores for dynamic metagenomics workloads. In *Proc. ACM/IFIP/USENIX 18th Int. Middleware Conf.*, pages 28–40, 2017.

[86] Imene Mami and Zohra Bellahsene. A survey of view selection methods. *ACM SIGMOD Rec.*, 41(1):20–29, 2012.

[87] Gabi Melman. SpdLog: Fast c++ logging library. https://github.com/gabime/spdlog, 2020.

[88] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. Performance and resource modeling in highly-concurrent OLTP workloads. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 301—-312, 2013.

[89] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proc. 8th USENIX Symp. on Operating System Design and Implementation*, pages 267–280, 2008.

[90] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proc. 9th USENIX Symp. on Networked Systems Design & Implementation*, pages 26–26, 2012.

[91] Minh Nguyen, Zhongwei Li, Feng Duan, Hao Che, Yu Lei, and Hong Jiang. The tail at scale: How to predict it? In *Proc. 8th USENIX Workshop on Hot Topics in Cloud Computing*, pages 120–125, 2016.

[92] Mark Palmer and Stanley B. Zdonik. Fido: A cache that learns to fetch. In *Proc. 17th Int. Conf. on Very Large Data Bases*, pages 255–264, 1991.

[93] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. Self-driving database management systems. In *Proc. 8th Biennial Conf. on Innovative Data Systems Research*, 2017.

[94] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. External vs. internal: An essay on machine learning agents for autonomous database management systems. *IEEE Data Eng. Bull.*, 42(2):32–46, 2019.

[95] Andrew Pavlo, Matthew Butrovich, Lin Ma, Prashanth Menon, Wan Shen Lim, Dana Van Aken, and William Zhang. Make your database system dream of electric sheep: Towards self-driving operation. *Proc. VLDB Endowment*, 14(12):3211–3221, 2021.

[96] Ofir Pele and Michael Werman. Fast and robust earth mover's distances. In *Proc. 12th Int. Conf. on Computer Vision*, pages 460–467, 2009.

[97] Jose Pereira. TPC-W implementation. https://github.com/jopereira/java-tpcw, 2016. University of Minho's implementation of TPC-W.

[98] Prometheus. Monitoring system and time-series database. https://prometheus.io/.

[99] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, Cambridge, Massachusetts, 2005.

[100] Peter J Rousseeuw and Annick M Leroy. *Robust regression and outlier detection*. John Wiley & Sons, Hoboken, New Jersey, 2005.

[101] Sudip Roy, Arnd Christian König, Igor Dvorkin, and Manish Kumar. Perfaugur: Robust diagnostics for performance anomalies in cloud services. In *Proc. 31st Int. Conf. on Data Engineering*, pages 1167–1178, 2015.

[102] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. In *Proc. 5th Int. Conf. on Data Engineering*, pages 452–462, 1989.

[103] Amazon Web Services. Open source databases. https://aws.amazon.com/products/databases/open-source-databases/, 2022.

[104] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010. https://research.google.com/archive/papers/dapper-2010-1.pdf.

[105] Splunk. Cloud-based data platform for cybersecurity, it operations, and dev ops. https://www.splunk.com/, 2021.

[106] Michael Stonebraker and Ugur Cetintemel. "One Size Fits All": An idea whose time has come and gone. In *Proc. 21st Int. Conf. on Data Engineering*, pages 2–11, 2005.

[107] Michael Stonebraker, Sam Madden, and Pradeep Dubey. Intel "big data" science and technology center vision and execution plan. *ACM SIGMOD Rec.*, 42(1):44–49, 2013.

[108] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proc. 33rd Int. Conf. on Very Large Data Bases*, pages 1150–1160, 2007.

[109] Mark D. Syer, Zhen Ming Jiang, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Leveraging performance counters and execution logs to diagnose memory-related performance issues. In *Proc. 29th Int. Conf. on Software Maint.*, pages 110–119, 2013.

[110] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Andrade. P-store: An elastic database system with predictive provisioning. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 205–219, 2018.

[111] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endowment*, 8(3):245–256, 2014.

[112] Dixin Tang, Hao Jiang, and Aaron J Elmore. Adaptive concurrency control: Despite the looking glass, one concurrency control does not fit all. In *Proc. 8th Biennial Conf. on Innovative Data Systems Research*, pages 1–9, 2017.

[113] P. Tchébychef. Des valeurs moyennes (traduction du russe, n. de khanikof.). *Journal de Mathématiques Pures et Appliquées*, 2(12):177–184, 1867.

[114] CAPEC Content Team. Capec 488: Http flood. https://capec.mitre.org/data/definitions/488.html, February 2020.

[115] TPC. TPC benchmark W (web commerce). http://www.tpc.org/tpcw, 2000.

[116] Alexandre Tsybakov. *Introduction to nonparametric estimation.* Springer, New York, 2008.

[117] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1009–1024, 2017.

[118] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. UDO: universal database optimization using reinforcement learning. *Proc. VLDB Endowment*, 14(13):3402–3414, 2021.

[119] Henry Ware and Fabian Frédérick. Vmstat manual page. https://linux.die.net/man/8/vmstat, 2021.

[120] Dag Wieers. DStat. http://dag.wiee.rs/home-made/dstat/, 2021.

[121] David Wiese, Gennadi Rabinovitch, Michael Reichert, and Stephan Arenswald. Autonomic tuning expert: A framework for best-practice oriented autonomic database tuning. In *Proc. Conf. of the IBM Centre for Advanced Studies on Collaborative Research*, pages 1–15, 2008.

[122] Stephen Yang, Seo Jin Park, and John Ousterhout. NanoLog: A Nanosecond Scale Logging System. In *Proc. USENIX 2018 Annual Technical Conf.*, pages 335–350, 2018.

[123] Matt Yonkovit. The state of the open source database industry in 2020: Part three. https://www.percona.com/blog/2020/04/22/the-state-of-the-open-source-database-industry-in-2020-part-three/, 2020.

[124] Dong Young Yoon, Barzan Mozafari, and Douglas P. Brown. DBSeer: Pain-free database administration through workload intelligence. *Proc. VLDB Endowment*, 8(12):2036–2039, 2015.

[125] Dong Young Yoon, Ning Niu, and Barzan Mozafari. Dbsherlock: A performance diagnostic tool for transactional databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1599—-1614, 2016.

[126] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. In *Proc. 21st Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 489–502, 2016.

[127] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 415–432, 2019.

[128] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. Non-Intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *Proc. 12th USENIX Symp. on Operating System Design and Implementation*, pages 603–618, 2016.

[129] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *Proc. 11th USENIX Symp. on Operating System Design and Implementation*, pages 629–644, 2014.