Dendrite: Bolt-on Adaptivity for Data Systems

Brad Glasbergen, Fangyu Wu, Khuzaima Daudjee
{bjglasbe,f49wu,kdaudjee}@uwaterloo.ca
Cheriton School of Computer Science, University of Waterloo

ABSTRACT

Client application workloads for data systems are known to vary in load and access patterns over time. This variability can place undue stress on data systems, tying up resources and degrading performance. To meet this challenge, systems must adapt by adjusting resource allocation and processing techniques to ameliorate contention and to deliver stable performance. We demonstrate Dendrite, a system designed to *bootstrap adaptivity* for data systems through its widely-applicable approach for extracting metrics, developing adaption rules, and applying them through user-defined functions to effect system behaviour changes. We highlight Dendrite's features and capabilities through a proof-of-concept implementation with the popular PostgreSQL database system.

CCS CONCEPTS

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

KEYWORDS

adaptivity, system behaviour, debug logging, self-driving

ACM Reference Format:
Brad Glasbergen, Fangyu Wu, Khuzaima Daudjee. 2021. Dendrite: Bolt-on Adaptivity for Data Systems. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event, China.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3448016.3452755

1 INTRODUCTION

Data systems are expected to handle fluctuations in client workloads, resource availability, and deployment environments. Typically, this burden falls to database administrators, who must appropriately configure the system, monitor it, and resolve performance problems. In a bid to reduce this burden, self-driving systems [1, 6, 7, 9–11] have recently been proposed that extract metrics while deployed to determine how the system is performing, adaptively adjusting the system's storage and processing techniques to optimize for the workload at hand. Unfortunately, many popular data systems such as PostgreSQL do not support robust adaption capabilities. Moreover, enhancing existing systems to support such adaption capabilities requires significant design and development effort, as each self-driving data system uses a custom monitoring and response approach [8].

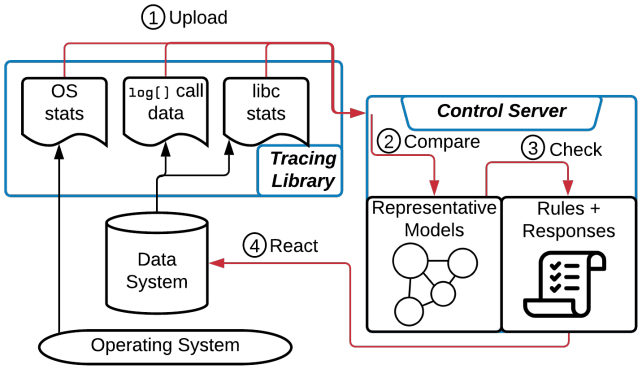

Figure 1: Dendrite's system architecture. The in-memory tracing library extracts system behaviour metrics and uploads them to Dendrite's control server. The server compares system behaviour, and evaluates adaption rules to determine which action to use to remedy behaviour differences.

Our system, *Dendrite*, eliminates this large design and development effort and democratizes self-driving database systems by providing a widely-applicable framework to extract metrics, intuitively declare adaption rules to determine if a reaction is warranted, and specify appropriate responses. Dendrite automatically extracts metrics from application debug logs, `libc`, and the operating system, enabling system developers/administrators to specify conjunctive adaption rules over these metrics. If a rule's conditions are satisfied, Dendrite executes a user-defined function (UDF) configured by the administrator as a response. As Dendrite is directly linked into debug logging libraries (e.g., log4j [3], glog [5]) and does not rely on any system-specific functionality, any data system using such a library can *automatically* obtain Dendrite's functionality to deploy their own self-driving components.

The next section describes how Dendrite extracts fine-grained metrics without requiring significant code modification, as well as its rule-action framework. In Section 3, we show how Dendrite can be used to empower the popular open source PostgreSQL database system, allowing it to automatically respond to system behaviour and workload differences on-the-fly.

2 THE DENDRITE SYSTEM

To detect changes in system and workload behaviour, Dendrite extracts metrics from debug logging calls, `libc` functions, and the operating system using its in-memory tracing library (Figure 1). These metrics are compiled into a model of system behaviour and uploaded to Dendrite's control server after a configurable time interval that we call an *epoch*, after which Dendrite will begin

SIGMOD '21, June 18–27, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452755

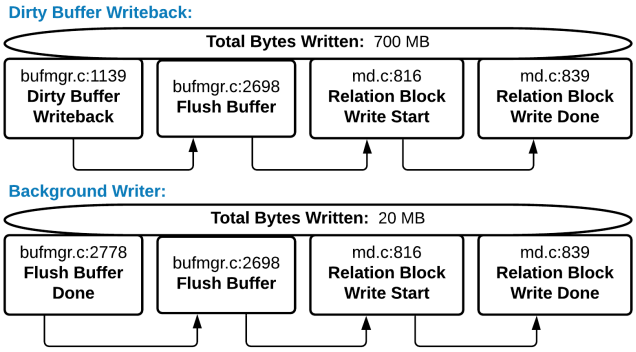

Figure 2: Dendrite stratifies metrics according to their transition path. Here, dirty buffer flushes during cache reads results in 700 MB of writes, while the background writer flushes only 20 MB.

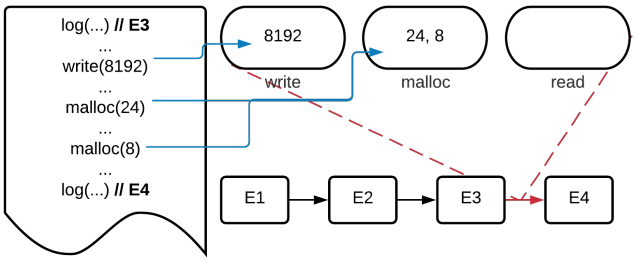

Figure 3: Dendrite overrides targeted `libc` functions to capture resource usage for each transition path. When a log call is issued, Dendrite associates the captured statistics with the current transition path.

building new models for the next epoch. When the control server receives a new behavioural model, it compares this model against the model obtained from the previous epoch to determine how system behaviour has changed over time. Dendrite also compares this epoch's model against a model of ideal behaviour (the *baseline* behaviour model) that is representative of expected system performance[1] as determined and uploaded by an administrator. If the current epoch's behaviour deviates significantly from the baseline, Dendrite checks whether any adaption rule's conditions are satisfied. If so, Dendrite executes the corresponding response UDF to remedy the underlying cause of the differences.

2.1 Extracting Metrics

Dendrite integrates directly with debug logging libraries and `libc` to extract metrics about system behaviour using its in-memory tracing library. As data systems commonly use debug logging libraries (e.g., glog, spdlog, log4j) and `libc`, this integration provides comprehensive insight into behaviour without requiring Dendrite to specialize for a particular system.

Debug logging libraries all use a similar interface, which Dendrite exploits to extract its behavioural models. When a system calls the library's `log(level, msg)`, the library checks to see if `level` exceeds a preconfigured logging threshold. If so, it writes the message (`msg`) to disk for later analysis. We modify this `log()` function to first call Dendrite's `record_event(file, line)` function before handling the call as usual, where file and line are the position in source code of the caller. Dendrite uses these file names and line numbers to uniquely identify *log events*. As this integration process requires only a simple change to the logging library, it is easy to configure data systems to use Dendrite.

Dendrite uses logging calls to capture the frequency of events and determine how the system moves (or transitions) between them. It encodes these events and event transitions into Markov models of system behaviour, as in the Sentinel system it builds upon [4]. Dendrite's event tracing is far more robust and extensive; while Sentinel captures only transitions between pairs of events, Dendrite determines which previous event transitions led to the current transition (*transition path*) and stratifies metrics accordingly. As an example using PostgreSQL, Figure 2 shows how Dendrite traces back from a RelationBlockWriteStart log event to the events that caused it by tracking the k previous event transitions. In doing so, it determines that PostgreSQL is writing back 700 MB of dirty buffers (pages) when trying to read from its page cache, while its background page writer is writing back only 20 MB of pages in a loop. Dendrite can use this information as a signal to increase the cache size, or increase background flushing of pages to avoid flushing pages on the critical path. By contrast, Sentinel can speculatively present only events that might have led to the RelationBlockWriteStart event and is unable to capture resource usage metrics.

To obtain these fine-grained resource usage metrics, Dendrite intercepts `libc` calls by configuring the dynamic linker (`ld.so`) to load custom versions of targeted `libc` functions, such as `malloc`, and `write`. It uses the `LD_PRELOAD` environment variable to load a shared object file with custom implementations of these functions before the main `libc` library and override their defaults. These custom implementations record statistics about call parameters and then execute the corresponding `libc` function as usual. Thus, the overheads of calling the custom functions is low, and system behaviour is unchanged. Dendrite adds these metrics into its behavioural models during `record_event()`; it looks up the last k log events that have occurred and stores each function's recorded statistics in a fixed-size reservoir associated with the transition path between them (Figure 3).

Dendrite also captures aggregate system resource usage metrics using *dstat* for each epoch. These metrics are useful for determining when a system is reaching the capacity of a resource (such as CPU or disk). By combining these details with fine-grained, transition resource usage metrics and information about system event popularity and event transitions, Dendrite provides powerful tools with which to define adaption rules.

At the end of each epoch, Dendrite outputs all of the data it has collected for each thread and encodes them into behavioural models. These models are used to identify which threads are active during the epoch, and are then combined together into a single model representing the system's overall behaviour during the epoch. Dendrite's control server contrasts this combined model with the

[1] Such a model can be trivially obtained by having Dendrite export a model of system behaviour when the system is performing well.

baseline model and that of the previous epoch to detect behaviour differences, evaluate adaption rules, and respond accordingly.

2.2 Process and Thread Fingerprinting

In addition to events, event transitions and metrics, it is often useful to know which processes and threads are active in the data system. For example, PostgreSQL maintains one process for each client connection; identifying and tracking processes therefore reveals how many clients are connected to PostgreSQL. It also enables Dendrite to stratify resource consumption by process. For example, Dendrite may determine that captured page flushes are due to the checkpointer process instead of dirty writes, and thus modify checkpointer parameters accordingly.

It is futile to merely consult the operating system's process/thread list to determine active threads, as not every thread is named and relying on specific names would stymie Dendrite's universality goals. Instead, administrators can register representative models for the data system's threads. When Dendrite obtains the per-thread models for an epoch, it compares their behaviour to that of the registered models. If the behaviour is similar, Dendrite records a mapping from its process/thread ID to the registered model's name. We denote threads that have outputted a model in a given epoch as *active* and threads we have not yet observed as *new*. Administrators can define adaption rules that operate on these active and new thread lists.

2.3 Model Differences and Rule-based Reactions

After identifying the system's active processes and threads, Dendrite compares the combined model for the current epoch against that of the previous epoch and the registered baseline/ideal model. Dendrite calculates a score of model similarity based on differences in the models' event frequency distributions. If the current epoch's behaviour differs from the ideal baseline model by more than a preconfigured score threshold, then Dendrite checks to see if these behaviour differences match the conditions of any registered reaction rules. If a rule's conditions are satisfied, then that rule's response UDF is invoked.

System administrators register adaption rules in Dendrite to adjust system behaviour according to workload and system behaviour changes. These rules are conjunctive expressions over metrics, events, and events transitions, and can be registered before the system starts or added during execution. To simplify rule composition, Dendrite provides a library of data extraction functions that administrators can use with rules to retrieve metric, event, and process information. Each rule is associated with a response action UDF, which is encoded as an arbitrary Python script. This rule-action formulation enables administrators to specify system adaptations intuitively and expressively. For example, we defined the following rule to ameliorate load spikes in one of our demonstration scenarios:

```
new_models('pg_worker')/active_models('pg_worker')
 >= 0.3 → function 'shed_load'()
```
(1)

Concretely, if Dendrite has observed an increase in active PostgreSQL worker threads of more than 30% then Dendrite will call the `shed_load` UDF. This UDF aborts client transactions and closes connections to reduce client load.

Consider another example of a rule in Dendrite that we used to reduce disk write load when disk throughput is saturated:

```
cur_aggregate_metric('write') > 20 MB/s ∧
transition_cur_metric_sum(['FlushBuffer';
'RelationBlockWriteStart'],'RelationBlockWriteDone',
'write') > 2 MB ∧ active_models('checkpointer') = 1
→ function 'dial_back_chkpt_and_autovac'()
```
(2)

Per this rule, if Dendrite has observed more than 20 MB/s of disk writes in the current epoch, at least two megabytes of which is due to flushing buffers (corresponding to the event transition FlushBuffer → RelationBlockWriteStart → RelationBlockWriteDone[2]), and the checkpointer process is active (as identified by fingerprinting from Section 2.2), then execute the `dial_back_chkpt_and_autovac` UDF. This UDF connects to PostgreSQL and adjusts the checkpointing and autovacuum intervals to reduce write pressure.

2.4 Dendrite User Interface

Details of Dendrite's model comparison and executed adaption rules are presented visually to administrators as a web application (Figure 4). The top panel shows the data system's behaviour over time, with a circle for each epoch. If the system behaves similarly to the configured ideal/baseline model in the previous epoch, then the circle is **blue**; otherwise, it is **gold** with a ⚠warning symbol. As the system executes, the interface updates to reflect newly finished epochs and their associated behaviour comparisons and responses. If a recently completed epoch diverges from expected behaviour, a warning panel pops up to notify the administrator of the change, as well as any actions taken in response to these behavioural differences. This panel also presents the option to register new rules, with a suggestion based on the rule-actions that were executed to remedy the behaviour divergence.

On the main page, the "Newly Triggered Adaptions" panel shows any adaptions triggered as a result of behaviour differences for the current epoch. Previously triggered adaptions that have not yet restored behaviour to the baseline are presented in the "In-Flight Adaptions" panel. The bottom panels show the behaviour differences between the current epoch and baseline models (left) and between the current and previous epochs (right). Dendrite presents their differences in terms of aggregate metrics and event frequencies, the latter of which are ranked in decreasing order of their differences. Clicking on a metric bar brings up a panel that shows the event transitions that most heavily contributed to this metric. For example, clicking on the `disk_writes` metric bar shows which event transitions wrote the most data to disk. For details on which processes are new in this epoch or currently active, the user can click the "processes" button.

[2]These are human-readable tags defined over event locations in source code; e.g., FlushBuffer corresponds to bufmgr.c:2698.

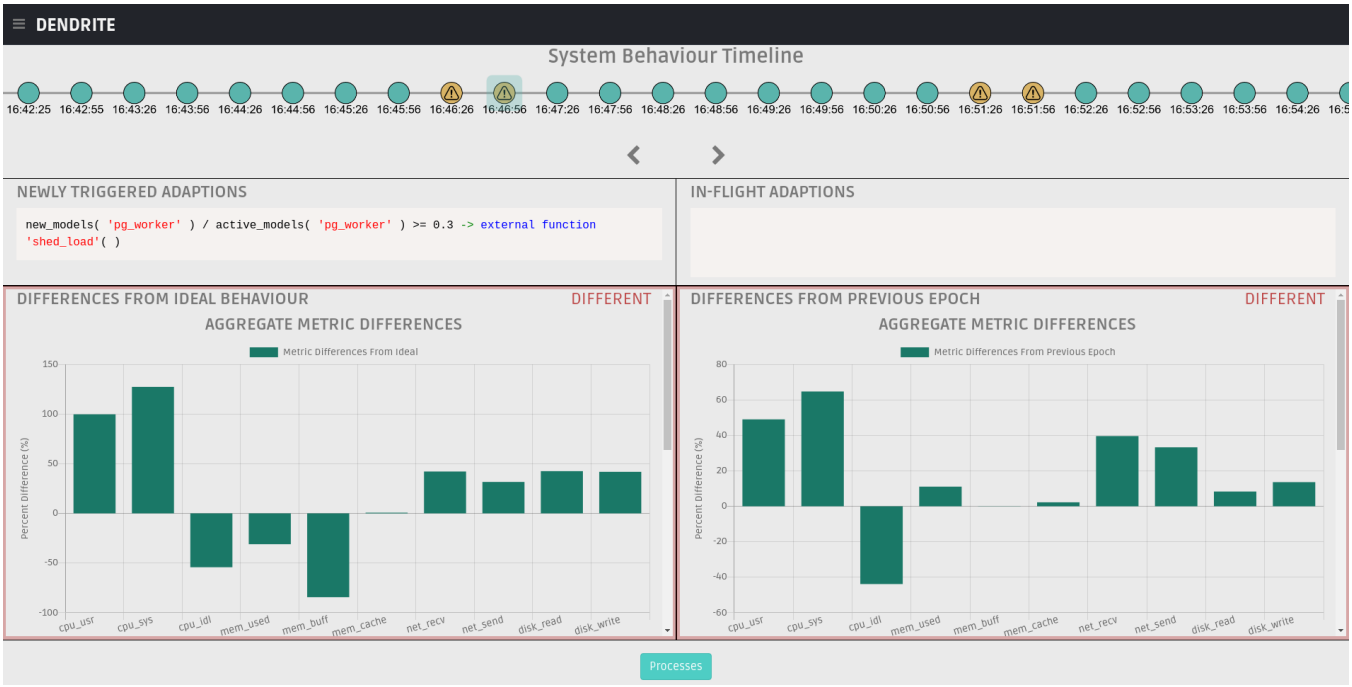

Figure 4: The main page of Dendrite's user interface, showing detailed system behaviour differences and responses for the load spike scenario.

3 DEMONSTRATION SCENARIOS

To demonstrate Dendrite's utility in detecting system behaviour differences and adapting to changing workloads, we present two scenarios using PostgreSQL 9.6. In both scenarios, we take on the role of an administrator using Dendrite's user interface to monitor PostgreSQL, detect behaviour changes, and respond to them appropriately. Both scenarios use a 12 core machine with 32 GB of RAM, so we configure PostgreSQL to use a 12 GB buffer pool. We register representative models for PostgreSQL workers and the checkpointer process, which Dendrite uses during fingerprinting (Section 2.2) to identify running PostgreSQL processes. We also register a model of ideal overall system behaviour (baseline) in which the buffer pool is warmed and there are no ongoing checkpoints or autovacuum tasks. These scenarios use 30-second epoch.

3.1 Load Spike Scenario

In the first scenario, we use a 10 client TPC-C workload [2] and configure Dendrite to use Rule 1 from Section 2.3. Once the system has warmed up and converged to the expected behaviour, we induce a load spike by doubling the number of TPC-C worker clients.

Dendrite rapidly identifies the load spike and alerts the administrator using the pop-up interface panel. The panel shows the behavioural effects of the load spike — more contention and serialization (concurrency) failures — and that Dendrite responded to this change using the `shed_load` UDF to restore normal system behaviour. Consulting the processes panel, we note that the number of PostgreSQL worker processes had doubled, which matches the induced behaviour change. The administrator will register a new rule in response to this change, which will send them an e-mail notification on future occurrences. We then induce another load spike by again doubling the number of clients. We show that the `shed_load` UDF is again deployed and that the administrator now receives an e-mail message warning them of the behaviour change. As a result of this response, Dendrite again returns system processing behaviour to normal.

3.2 Write Pressure Scenario

In the second scenario, we use a 20 client TPC-C workload. We start the system using the default parameters, but adjust the checkpoint completion target to 0.0 to increase checkpointing speed. We configure Dendrite to use Rule 2 from Section 2.3.

Dendrite rapidly alerts the administrator to a significant behaviour difference from the registered ideal behaviour. It determines that the system is behaving unsatisfactorily – PostgreSQL is writing a large amount of data due to the TPC-C workload and the checkpointer process. The pop-up panel shows these behaviour differences and that they satisfy Rule 2. Thus, Dendrite deploys the `dial_back_chkpt_and_autovac` UDF to reduce disk write pressure. Once the ongoing and revised schedule of checkpoints complete, the system returns to the ideal behaviour. As a result of this adaption, Dendrite significantly increases system throughput.

ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Canada Foundation for Innovation (CFI) and Ontario Research Fund (ORF).

REFERENCES

[1] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. MorphoSys: Automatic Physical Design Metamorphosis for Distributed Database Systems. *Proc. VLDB Endow.* 13, 13 (2020), 3573–3587.

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

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

[4] Brad Glasbergen, Michael Abebe, Khuzaima Daudjee, and Amit Levi. 2020. Sentinel: Universal Analysis and Insight for Data Systems. *Proc. VLDB Endow.* 13, 12 (2020), 2720–2733.

[5] Google. 2020. Google Logging Module. https://github.com/google/glog.

[6] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-Based Workload Forecasting for Self-Driving Database Management Systems. In *SIGMOD '18*. 631–645.

[7] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems. In *CIDR'17*.

[8] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. 2019. External vs. Internal: An Essay on Machine Learning Agents for Autonomous Database Management Systems. *IEEE Data Eng. Bull.* 42, 2 (2019), 32–46.

[9] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Andrade. 2018. P-Store: An Elastic Database System with Predictive Provisioning. In *SIGMOD '18*.

[10] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning. In *SIGMOD '17*. 1009–1024.

[11] 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. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *SIGMOD '19*. 415–432.