Doppler: Understanding Serverless Query Execution

Thomas Bodner, Tobias Pietz, Lars Jonas Bollmeier, Daniel Ritter
Hasso Plattner Institute, University of Potsdam
{thomas.bodner,daniel.ritter}@hpi.uni-potsdam.de,{tobias.pietz,lars.bollmeier}@student.hpi.de

ABSTRACT

Analyzing and understanding query execution dynamics in distributed cloud-based databases is difficult and requires laborious system designs. Serverless query execution, with its massive amount of small, short-lived, and stateless query workers, is even more challenging. To meet this challenge and materialize the economic benefits of serverless computing, all system components have to be serverless themselves. We demonstrate Doppler, a serverless toolkit designed to trace serverless data processing systems with minimal performance and cost overhead and to provide a deep understanding of their query execution. We highlight Doppler's features and capabilities through a proof-of-concept implementation with the serverless data processing system Skyrise.

CCS CONCEPTS

• **Information systems** → *Database utilities and tools*; *Database query processing*; *Relational parallel and distributed DBMSs*; • **Computer systems organization** → *Cloud computing*.

KEYWORDS

Debug Logging, Performance Diagnosis, System Behavior, Serverless Computing, Serverless Functions.

ACM Reference Format:
Thomas Bodner, Tobias Pietz, Lars Jonas Bollmeier, Daniel Ritter. 2022. Doppler: Understanding Serverless Query Execution. In *Big Data in Emergent Distributed Environments (BiDEDE'22), June 12, 2022, Philadelphia, PA, USA*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3530050.3532919

1 INTRODUCTION

Recent work on serverless data processing systems [8, 14, 15] developed sufficient solutions for cost-efficient, interactive, analytical query processing on large amounts of data that is required for current enterprise and analytical applications. Therefore, these systems leveraged serverless function-as-a-service (FaaS) technology like AWS Lambda [3] or Google Cloud Functions [10] for interactive, in-situ query processing and ad-hoc, massive scaling, as illustrated in Fig. 1. However, working with massive amounts of cloud functions / SERVERLESS WORKERS that communicate via a shared, SERVERLESS STORAGE and complex query plans (i. e., STAGE 1 to STAGE N), among others, many issues like delayed execution during LOAD / STORE (i. e., stragglers) [15]) can occur.

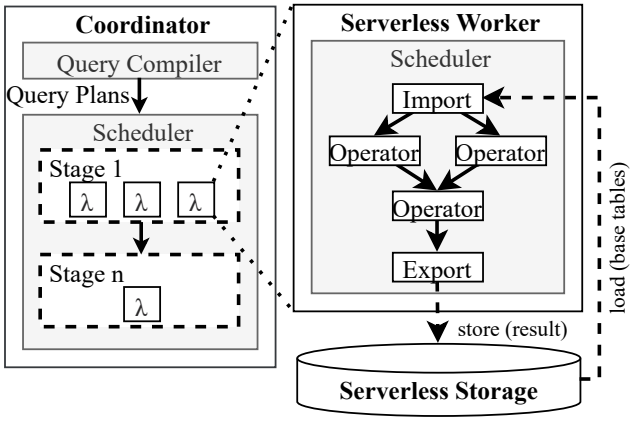


Figure 1: Serverless query execution engine.

Moreover, the ad-hoc, short-lived, and ephemeral nature of these functions, used for data processing, render the analysis and understanding of such distributed query processing difficult. Additionally, to materialize the economic benefits of serverless computing, systems have to be serverless themselves. Unfortunately, current distributed tracing systems (e. g., Dapper [16]), database observability approaches (e. g., [7]), or serverless application tracing (e. g., [9, 12]) do not cover these challenges.

To make serverless query execution understandable, we developed Doppler, a serverless toolkit designed to trace serverless data processing systems with a minimal cost and performance overhead and provide a deep understanding of their query execution. Doppler uses a post-mortem approach to cover a wide variety of use cases. We highlight Doppler's features and capabilities through a proof-of-concept implementation with the serverless data processing system Skyrise [8].

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

2 THE DOPPLER DEBUGGING AND PERFORMANCE PROFILING TOOLKIT

Doppler instruments query processing systems to generate and disseminate traces with query context at little overhead.

Doppler then collects the traces resulting from distributed queries "post mortem" to present multiple analysis techniques and visualizations. This process is depicted in Fig. 2 along the involved Doppler components. Subsequently, we describe each process step in detail.

BiDEDE'22, June 12, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9346-1/22/06…$15.00
https://doi.org/10.1145/3530050.3532919

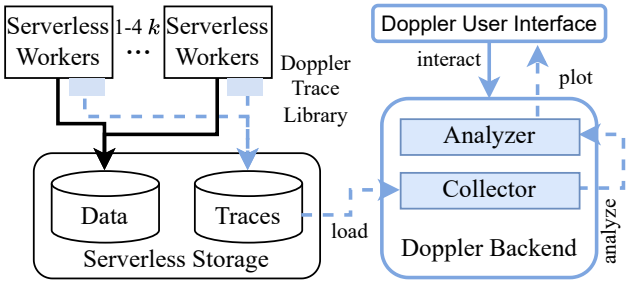


Figure 2: Doppler's system architecture.

2.1 Putting Query Context into Traces

A serverless query processor may run many queries, each over a multitude of serverless query workers. The workers each execute a query stage's physical sub-plan containing multiple operators (cf. Fig. 1). In order to be able to match the resulting execution traces back to a specific query run, every trace is enriched with context information: A coordinator and a worker context (cf. Fig. 3).

The coordinator context includes the software version, user ID, cloud provider region, query arrival time, a hash of the query SQL string, and the active query stage. It allows linking a trace to a query and stage. The worker context is provided by every individual worker function and changes with the actively executed operator. To allow for inter-operator parallelism, there are sub-contexts per thread, such that traces of the threads are not intermixed. The worker context links a trace to a cloud function invocation and a query operator. Both the coordinator and the worker contexts can be extended with additional user-defined attributes to allow further contextualization of the traces.

A trace may represent metrics of the query execution or error log messages. The metrics required by Doppler for its analyzes include the start and end times of cloud function and operator executions and the row counts and bytes consumed and produced by the operators, amongst others. Errors include the HTTP request status messages from third-party services, such as cloud storage systems. We also install signal handlers, e. g., for out-of-memory events and segmentation faults to report on these types of errors.

For our integration with Skyrise, we pass the coordinator context to the worker as part of the JSON body of the AWS Lambda function invocation request. On the worker side, the AWS SDK's logging facility is initialized with this context as prefix to every trace.

2.2 Collecting Traces from Serverless Functions

Doppler is targeted at engineers to conduct a posteriori analysis of slow and failed serverless query executions. To help investigating such pathological scenarios without the need for reproduction, tracing has to be active by default. As such, the overhead of tracing must be low in terms of both performance and operational cost.

For this reason, Doppler's tracing library is built on top of the base logging service of the cloud provider. In AWS, this is CloudWatch Logs [2]. Other major cloud platforms offer similar services at similar price points [11, 13]. From experiments not presented in detail for brevity, the performance and cost overhead of this approach is negligible, even when running queries with thousands

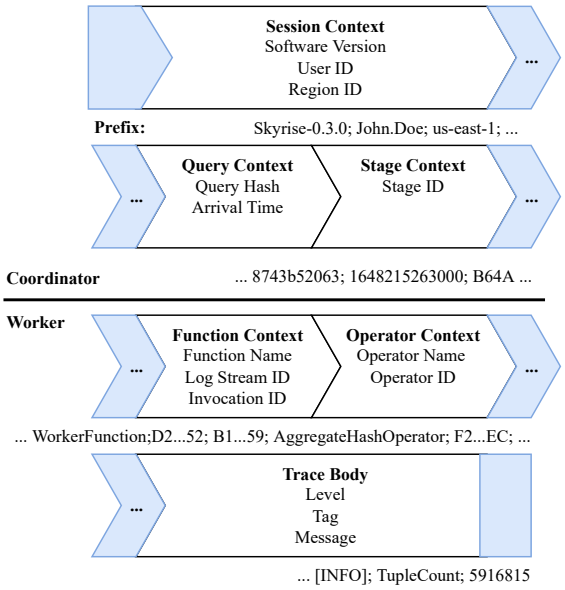


Figure 3: Context information for linking traces and queries.

of serverless functions over terabyte-scale datasets. The additional request and storage cost incurred by the logging service is about three orders of magnitute lower than the actual query cost.

After a query run ended with all related cloud functions shut down, the query's traces are typically available for retrieval from the logging service within seconds (although this may take up to 10 minutes [1]). On the Doppler backend, the trace collector takes the context information uniquely identifying a query run and fetches all of the corresponding traces. The traces are parsed and combined into a structured file (e. g., CSV) for subsequent analysis.

2.3 Analyzing Distributed Traces

The Doppler backend is written in Python and comprises a set of interdependent modules for trace analysis. The modules have been built out to study the pathological scenarios that we have encountered in developing the serverless query processor Skyrise.

The core modules include functionality to correlate the traces generated by the distributed query workers. For this, we rely on the timestamps in the traces. In practice, we have not seen clock skew between the workers to be an issue. All workers run in geographical proximity to one another in a single cloud provider region. They also employ the cloud provider's clock synchronization service [6]. This results in clock skew in the range of milliseconds, which is negligible for analyzing query traces spanning seconds to minutes.

To investigate straggler and concurrency issues common for distributed systems, one module matches the start and end timestamps of the cloud functions and query operators to generate respective runtime intervals. Another module intersects these intervals to compute the degree of concurrency for every point in time during the query run. These modules can be combined or extended to cover further scenarios.

2.4 Doppler User Interface

The results of Doppler's analysis algorithms are presented visually, e. g., as `Matplotlib` plots embedded into Jupyter notebooks, as shown in Fig. 4.

The out-of-the-box set of plots from Doppler already provide a holistic picture of a behavior a serverless query processing system.

In the notebooks, engineers can further refine these analyzes as part of their profiling or debugging workflows. They can document and share the notebooks, and work on them collaboratively, e. g., via a Jupyter server.

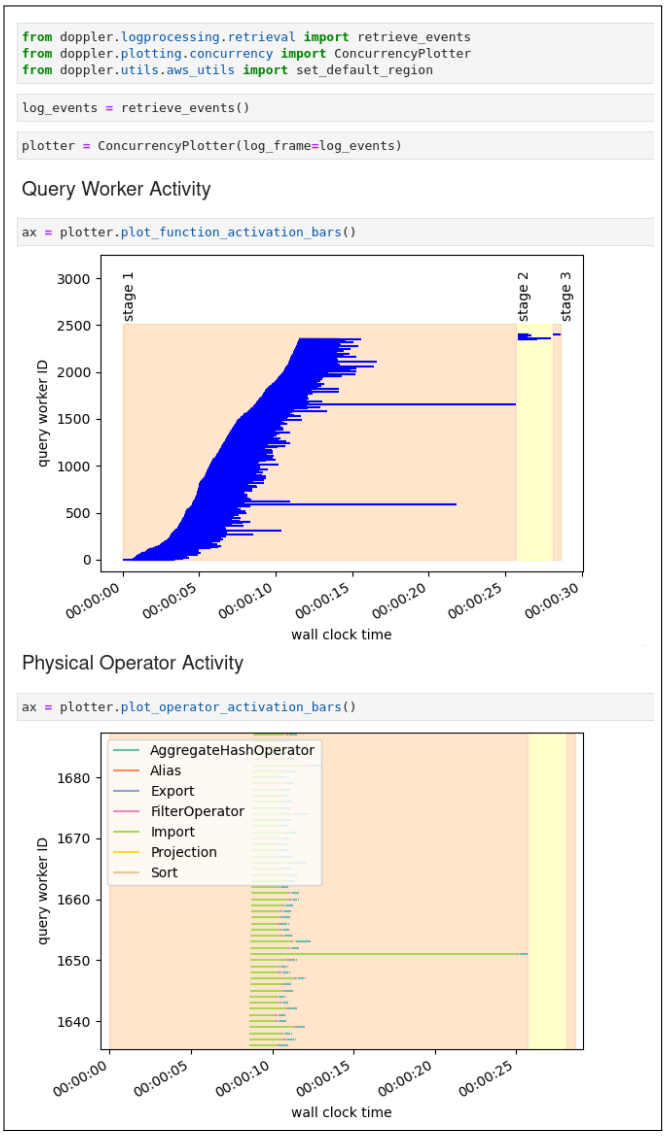


Figure 4: Doppler's notebook-based user interface.

3 DEMONSTRATION SCENARIOS

Our demonstrations employ Doppler to diagnose the root causes of slow and failed queries in Skyrise running TPC-H workloads [17]. Attendees take on the role of a database system engineer using the Doppler user interface to profile or debug the serverless query processor step-by-step.

In the following three scenarios, we run TPC-H queries on the scale factor 1, 000 dataset with an uncompressed size of ~750 GB. Skyrise is configured to use at maximum 2, 400 concurrent workers, each having 2 vCPUs and 2 GB of main memory. Skyrise is further instrumented to trace the information described in Sect. 2.1.

The scenarios differ in the impediment that we introduce and in how we configure AWS service quotas and Skyrise. From our experience developing Skyrise, we consider them highly relevant, since they occur frequently or are cumbersome to investigate manually.

3.1 Straggling Query Worker

Our first scenario shows how Doppler can be used to detect and investigate the common case of straggling query workers. Stragglers taking much longer to run than other workers may severely impact overall query performance. Stragglers can occur due to a variety of reasons, spanning from the underlying hardware and software stack to unevenly sized data partitions and different data distributions between partitions.

In this scenario, we induce data skew by assigning few selected workers many more input partitions than their peer workers. To identify stragglers, the runtime intervals of the workers can be visualized in a bar chart as in the top half of Fig. 4. This chart provides an intuitive overview of the individual worker runtimes, showing that the slowest workers finish about 10 seconds later than the majority of workers in the respective query stage.

The database engineer goes on to investigate by drilling down from the worker runtimes to the runtimes of the operators executed by workers, as can be seen in the bottom half of Fig. 4. There she sees that the import operator executed by the straggling worker takes significantly longer than in the other workers.

Given this investigation, the engineer compares the lists of input partitions to the workers and concludes that the problem is rooted in uneven input sizes.

3.2 Fatal Node Failure

In the second scenario, we use Doppler's dataframe-based tabular data processing capabilities (e. g., using `pandas`) to identify and understand errors in a failed query's traces. In the serverless setting, some types of errors are more prevalent, while others are more difficult to debug in the absence of assertions and debug symbols. For example, serverless functions are subject to tight limits on memory capacity and runtime that can be quickly exceeded [5]. Cloud functions may also be aborted halfway and reexecuted on another physical host at the will of the FaaS platform's control plane. In such cases, the function's process receives a kill signal.

To demonstrate how Doppler can help with fatal errors, we inject a randomly occurring segmentation fault in Skyrise's import operator code. After this segmentation fault caused a query to fail, the database engineer starts to get an overview by filtering the failed query's traces for error log messages. As can be seen in Fig. 5, the error logs are enriched with the earlier discussed query context. The engineer notices multiple occurrences of erroneous import operator executions.

	timestamp	stage_id	operator_name	message
0	1647196394783	1	Import	Memory Access Violation took place. Exiting with a failure.
1	1647196394783	1	Import	Error, operator did not finish
2	1647196394790	1	Import	Memory Access Violation took place. Exiting with a failure.
3	1647196394790	1	Import	Error, operator did not finish
4	1647196394845	1	Import	Memory Access Violation took place. Exiting with a failure.
...	...	...	...	...
23	1647196396238	3	Import	Read failed with message: No response body.
24	1647196396241	3	Import	HTTP response code: 404

Figure 5: List of all error log messages.

	timestamp	stage_id	operator_name	message
963	1647196394688	1	Import	FE517E10-1CE3-479D-9712-846884FDD20A|end|S3:GetObject
964	1647196394688	1	Import	FE517E10-1CE3-479D-9712-846884FDD20A|success|S3:GetObject
995	1647196394790	1	Import	Creating data structures...
996	1647196394790	1	Import	Memory Access Violation took place. Exiting with a failure.
1242	1647196394790	1	Import	Error, operator did not finish

Figure 6: List of last error log messages from the worker that ran into the marked error in Fig. 5.

To investigate further, the engineer filters for all the traces from the worker that failed due to the error marked in Fig. 5. The result is a list of the last messages logged by this worker (cf. Fig. 6, indicating the root cause to be a memory access violation. The engineer can use this information to isolate the error locally.

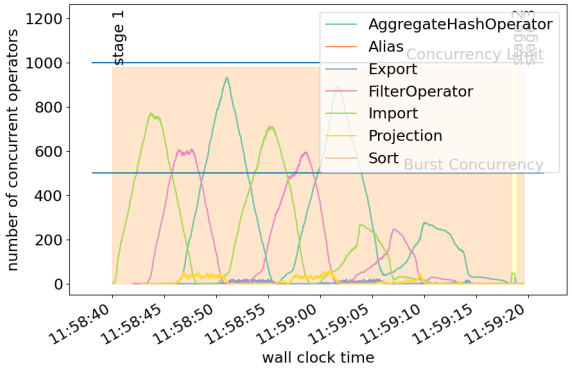
3.3 Serverless Function Concurrency Limit

In our third scenario, we migrate from the AWS default region us-east-1 to us-west-1. In this region, the FaaS platform provider's function concurrency limits are lower [4]. The burst concurrency limit is 500 and the default maximum concurrency is 1,000 functions. We run one of TPC-H's scan-heavy queries (Q1 and Q6) and observe a query runtime about three times as long as in us-east-1.

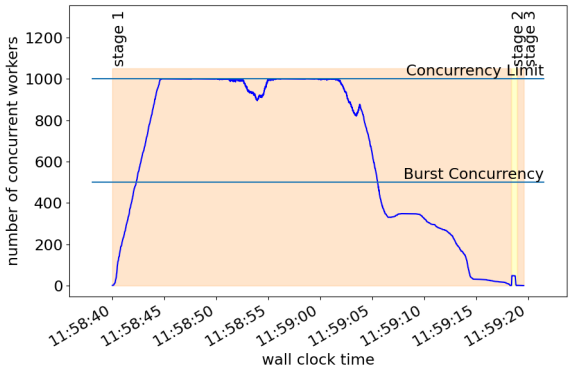
Suspecting straggling query workers, the engineer inspects the worker runtime intervals as in Fig. 4. Not recognizing any particular outliers, the engineer moves on to look into the operator-level traces and plots the operator concurrency shown in Fig. 7a.

The engineer sees that each physical operator in query stage 1 is being executed in three distinct waves. Since this is not dictated by the query plan, the engineer also plots the concurrency of the compute resources as depicted in Fig. 7b. The concurrency limits are shown by the horizontal lines. The burst concurrency limit above which function invocations may get throttled is not an issue. However, the overall concurrency limit does prevent the query's ephemeral cluster from growing to 2,400 function workers. Thus, the set of requested workers is being scheduled in three waves.

Given this insights, the engineer can configure AWS Lambda with a higher concurrency limit. Alternatively, she can instruct Skyrise to not generate query plans for more than 1,000 workers.



(a) Physical query operator instances.



(b) Worker nodes in the ephemeral cluster.

Figure 7: Concurrency over the course of the query run.

REFERENCES

[1] Amazon Inc. 2022. Accessing Amazon CloudWatch logs for AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/monitoring-cloudwatchlogs.html (visited 03/2022).

[2] Amazon Inc. 2022. Amazon CloudWatch. https://aws.amazon.com/cloudwatch/ (visited 03/2022).

[3] Amazon Inc. 2022. AWS Lambda. https://aws.amazon.com/lambda/ (visited 03/2022).

[4] Amazon Inc. 2022. Lambda function scaling. https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html (visited 03/2022).

[5] Amazon Inc. 2022. Lambda quotas. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html (visited 03/2022).

[6] Amazon Inc. 2022. Set the time for your Linux instance. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/set-time.html (visited 03/2022).

[7] A. Beischl, T. Kersten, M. Bandle, J. Giceva, and T. Neumann. 2021. Profiling Dataflow Systems on Multiple Abstraction Levels. In *EuroSys*. 474–489.

[8] T. Bodner. 2020. Elastic Query Processing on Function as a Service Platforms. In *VLDB 2020 PhD Workshop*.

[9] M. C. Borges, S. Werner, and A. Kilic. 2021. FaaSter Troubleshooting - Evaluating Distributed Tracing Approaches for Serverless Applications. In *IC2E*. 83–90.

[10] Google Inc. 2022. Cloud Functions. https://cloud.google.com/functions/ (visited 03/2022).

[11] Google Inc. 2022. Writing, Viewing, and Responding to Logs. https://cloud.google.com/functions/docs/monitoring/logging/ (visited 03/2022).

[12] J. Manner, S. Kolb, and G. Wirtz. 2019. Troubleshooting Serverless Functions: A Combined Monitoring and Debugging Approach. *SICS* 34, 2 (2019), 99–104.

[13] Microsoft Corp. 2022. Monitor Azure Functions. https://docs.microsoft.com/en-us/azure/azure-functions/functions-monitoring (visited 03/2022).

[14] I. Müller, R. Marroquín, and G. Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *ACM SIGMOD*. 115–130.

[15] M. Perron, R. Castro Fernandez, D. DeWitt, and S. Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *ACM SIGMOD*. 131–141.

[16] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. 2010. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. In *Google Technical Report*.

[17] Transaction Processing Performance Council. 2022. TPC-H. https://www.tpc.org/tpch/ (visited 03/2022).