A Cockpit for the Development and Evaluation of Autonomous Database Systems

Jan Kossmann, Martin Boissier, Alexander Dubrawski*, Fabian Heseding*,
Caterina Mandel*, Udo Pigorsch*, Max Schneider*, Til Schniese*, Mona Sobhani*,
Petr Tsayun*, Katharina Wille*, Michael Perscheid, Matthias Uflacker†, Hasso Plattner
Hasso Plattner Institute, University of Potsdam, Germany
{firstname.lastname}@hpi.de, *{firstname.lastname}@student.hpi.de, †{firstname.lastname}@sap.com

***Abstract*—Databases are highly optimized complex systems with a multitude of configuration options. Especially in cloud scenarios with thousands of database deployments, determining optimized database configurations in an automated fashion is of increasing importance for database providers. At the same time, due to increased system complexity, it becomes more challenging to identify well-performing configurations. Therefore, research interest in autonomous or self-driving database systems has increased enormously in recent years. Such systems promise both performance improvements and cost reductions.**

In the literature, various fully or partially autonomous optimization mechanisms exist that optimize single aspects, e.g., index selection. However, database administrators and developers often distrust autonomous approaches, and there is a lack of practical experimentation opportunities that could create a better understanding. Moreover, the interplay of different autonomous mechanisms under complex workloads remains an open question. The presented cockpit enables an interactive assessment of the impact of autonomous components for database systems by comparing (autonomous) systems with different configurations side by side. Thereby, the cockpit enables users to build trust in autonomous solutions by experimenting with such technologies and observing their effects in practice.

I. Autonomous Database Systems

Tuning database management systems (DBMS) for optimal performance is an increasingly challenging task for human database administrators (DBA). Nowadays, DBMSs offer an ever-growing number of options for physical database design and configuration; often, hundreds of knobs are available [1]. These knobs cannot be tuned independently as the performance impact of a particular knob might depend on the current setting of other knobs. Besides, workloads are often complex, combine transactional as well as analytical patterns, and are difficult to anticipate because of seasonal effects and unexpected events.

Self-tuning or even autonomous database systems can support DBAs in such complex optimization tasks. However, interviews with DBAs and customers show that autonomous solutions are often distrusted [2]: they believe that such solutions only work for synthetic benchmarks and are not robust enough to handle the challenges originating from real-world scenarios.

An evaluation of the actual benefit on workload performance, as well as the introduced resource and processing overheads by autonomous techniques, is necessary to build trust and demonstrate the advantages of autonomous approaches. Furthermore, creating an understanding of the reasoning behind the decisions taken by autonomous systems (cf. explainability) facilitates the adoption process [3]. Thus, a system that enables validation of and experimentation with autonomous database systems should be provided to database administrators and engineers.

An interactive system that provides the possibility to directly compare autonomous with conventional systems regarding their performance during operation could create the necessary trust and opportunities for practical experimentation. For example, in their recent work, Das et al. [4], describe an approach to this problem employed for Microsoft Azure SQL databases: the same customer workload is sent to two systems, system A and B. System A is conventional. In contrast, system B is a copy of system A that is used for autonomous administration experiments. Thereby, the performance of both systems can be compared in a real-world scenario, while it is guaranteed that the customer system's performance is not affected in any way.

Contributions. In this paper, we present our cockpit for autonomous database systems that builds on the aforementioned concept of directly comparing multiple systems. Workloads and system configurations can be modified to evaluate database performance in specific situations and scenarios. Our cockpit builds on the following key concepts:

- Monitoring: Important metrics to inspect performance (e.g., throughput or latency), system resources (e.g., CPU utilization and memory footprint), and the current workload (e.g., expensive statements and access counters) are continuously monitored and stored in a time-series database for convenient analysis.
- Interactivity: Users can activate autonomous components, observe their impact on workload performance as well as on the system's resources.
- Explainability: We provide the opportunity to inspect the reasoning behind autonomously taken decisions to facilitate a better comprehension of such decisions.
- Variable workloads and load intensity: Users can provide their own workloads and datasets or choose from a variety of synthetic ones (e.g., the TPC-H, TPC-DS, TPC-C, and JoinOrder Benchmark) and regulate the pressure, in terms of parallel queries per second, that is put on the system.

Thereby, our system allows us to evaluate the impact of autonomous components, supports building trust, and facilitates further development of autonomous database technologies.

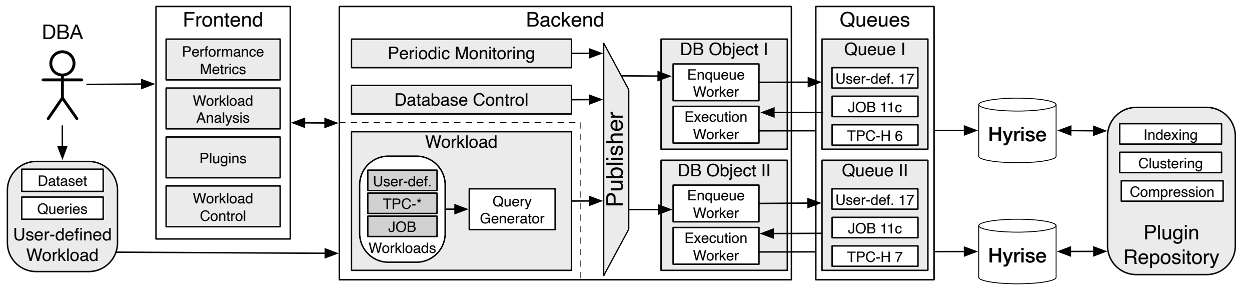


Fig. 1. Architecture of our cockpit for autonomous database systems. The currently active workload represents a mixture of a used-provided workload trace (*User-def.*), TPC-H and Join Order Benchmark based queries.

II. System Overview

We first give an overview of Hyrise[1], the database system that is currently supported by our cockpit. Afterward, we present the cockpit's architecture and provide implementation details.

A. Hyrise

The current foundation of the presented cockpit is the research database system Hyrise [5] and its plugin approach to handling autonomous components. However, other database systems could be easily integrated as long as the relevant metrics can be obtained via SQL by the cockpit.

Architecture. Hyrise is a relational, open-source, main memory, columnar database system for hybrid transactional and analytical transaction processing (HTAP). It employs a vectorized execution model, follows an insert-only approach, and utilizes multi-version concurrency control (MVCC) to ensure correct concurrent transaction handling.

Columns can be encoded with various techniques, e.g., dictionary, run-length, or LZ4 encoding, to reduce the memory footprint and improve memory bandwidth utilization. Tables are implicitly divided into horizontal partitions (65 535 tuples) that are called chunks. Chunking serves two primary purposes: increased opportunities for data access avoidance by pruning and more flexible and fine-granular configuration and tuning decisions. For instance, decisions upon indexes to create or which encoding to apply can be taken on chunk level and do not have to be made for a complete column. Thereby, more precise decisions can be made, and the overhead of applying new configurations is reduced.

Hyrise implements the PostgreSQL wire protocol[2] to provide network access via widely available clients and libraries. Metrics, for instance, the current process CPU utilization, chunk access counts, or the memory footprint of a table, can be easily accessed via SQL. For more details on Hyrise, we refer the interested reader to [5] and the source code.

Plugins. Hyrise offers a plugin interface that allows adding functionality without affecting the code base of the database core to keep it as self-contained as possible. We use plugins to implement capabilities for autonomous database configuration. Plugins are implemented as dynamic libraries in C++, thus enabling direct access to the database core without providing specialized interfaces while guaranteeing native speed.

[1]Hyrise repository: https://github.com/hyrise/hyrise

[2]https://www.postgresql.org/docs/12/protocol.html

B. Cockpit

In the following, we discuss the architecture and implementation details of the presented cockpit[3]. The cockpit was developed with nine undergraduate students as part of their final project.

Our cockpit allows for interactive evaluation of autonomous database configuration techniques. Besides, it enables a better understanding of the decisions taken by the autonomous approaches by providing their decision logs and insights about the processed workload and stored data.

The components of the cockpit (see Figure 1) solve three main tasks: (i) the frontend displays performance metrics, allows the user to interactively examine the processed workload as well as current system configuration, and provides access to the plugins of the DBMS, (ii) the cockpit backend handles the communication between the frontend and the investigated systems, periodic monitoring and storing of the displayed metrics. (iii) The workload generator is responsible for creating queries and putting pressure on the evaluated systems.

While the frontend is implemented in Vue.js, we use Python3 throughout all other components. One of the main challenges during implementation was the harmonization of the, sometimes contrary, design requirements.

- High load generation: For insightful evaluations, the ability to stress the systems at hand with high system loads is mandatory because trust in autonomous approaches is only achieved if extreme situations can be handled well.
- Fair load distribution: Even though it is technically impossible to generate the same workload, including equal arrival times for queries, for all evaluated systems, the workload generator must ensure that differences are kept as small as possible.
- Simple workload extension: The cockpit should serve as an evaluation platform for DBAs and developers. Thus, providing synthetic queries and data is not sufficient. Instead, users need to be able to add their own workloads and data in a simple fashion.

To achieve a fair load distribution, generate a sufficient number of queries, and follow a clear separation of concerns, we decided to decouple query generation from actually sending the queries. Thereby, these tasks can operate independently in different processes and on different CPU cores to avoid that

[3]Source code: https://git.io/AutonomousDBMSCockpit

the workload generation is becoming a bottleneck. The *query generator* implements the logic for creating a configurable number of queries per second from a predefined query set. By default, all queries of a workload are chosen with the same probability. The cockpit user can modify the query distribution to emphasize specific queries and reproduce realistic scenarios.

Afterward, the *workload generator* passes the queries to the backend database objects, which maintain query queues (implemented with Python's multiprocessing library) and handle the actual query sending with psycopg2. We utilize ZeroMQ's[4] publisher-subscriber pattern for efficient inter-process communication. The task of enqueueing new queries also has to be uncoupled from sending them, because otherwise, a poorly performing database instance could affect the fair and even query distribution. Replays of existing real-world workloads are supported by simply providing the necessary table data and queries as CSV.

Flask handles the communication between the backend and frontend. Furthermore, the time series database InfluxDB is used to store the relevant metrics permanently and allows for analyzing historical performance data to comprehend particular system and plugin behavior. We facilitate reproducibility and the cockpit's setup process by providing a docker setup for all components.

III. Autonomous Plugins in Hyrise

In this section, we explain the high-level functionality of three exemplary autonomous database configuration approaches that are implemented as Hyrise plugins.

A. Automated Compression Configuration

Hyrise supports various encoding schemes, e.g., dictionary, frame of reference, and run-length encoding. Further, also heavy-weight compression such as LZ4 is supported.

The compression plugin (cf. [6]) aims to reduce the footprint of main memory-resident data. However, data access costs could be elevated during query processing caused by expensive decoding operations. Therefore, the plugin also minimizes potential performance degradations.

Given a user-defined memory budget, the plugin analyzes the current workload and optimizes the compression configuration in fixed intervals accordingly: The plugin applies a compression configuration that results in the best runtime performance and still complies with the given main memory budget.

B. Automated Clustering

Clustering or sorting a table can significantly impact the performance of several operators, e.g., joins or table scans. With clustered data, Hyrise can exploit its chunked storage concept for data access avoidance. Filters or small materialized aggregates [7], can be utilized to determine prunable chunks during the plan optimization phase, cf. partition pruning.

The clustering plugin aims to improve the system's performance by clustering the table along frequently accessed dimensions. The goal is to (i) improve the pruning of chunks during plan optimization time and (ii) improve the scan runtime as the resulting chunks are sorted to allow binary searches. As of now, the plugin clusters a table in the background and swaps the whole table atomically.

C. Indexing

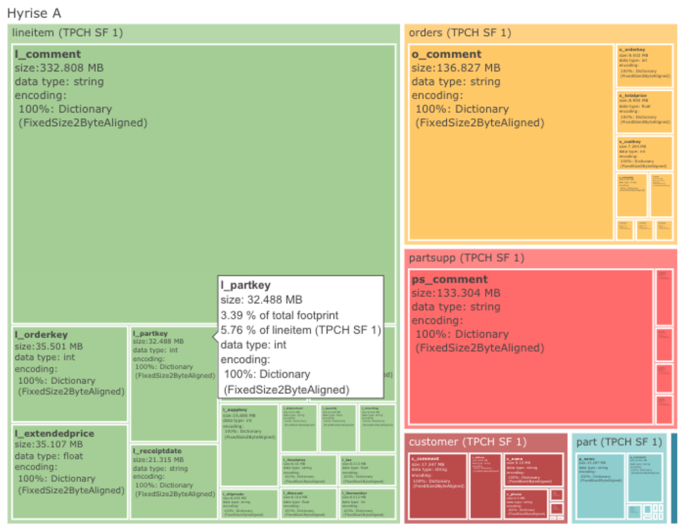
In general, secondary indexes are one possible measure to enhance the performance of a DBMS. However, the positive effects on the throughput and latency results have to be balanced with an increased memory footprint.

The indexing plugin utilized in the cockpit is an implementation of the index selection approach proposed by Schlosser et al. [8]. Similar to the compression plugin, it allows to set a user-defined memory budget. In addition, the maximum number of attributes per index can be adjusted. For this budget, the plugin identifies the best secondary indexes for the current workload based on the query plan cache.

IV. Demonstration Overview

In this section, we first explain which information is displayed and which options regarding interaction are offered in the presented cockpit. Afterward, we sketch a possible scenario for using our cockpit in practice with the help of an example.

A. User Interface Overview



l_comment
size:332.808 MB
data type: string
encoding:
100%: Dictionary
(FixedSize2ByteAligned)

o_comment
size:136.827 MB
data type: string
encoding:
100%: Dictionary
(FixedSize2ByteAligned)

ps_comment
size:133.304 MB
data type: string
encoding:
100%: Dictionary
(FixedSize2ByteAligned)

l_partkey
size: 32.488 MB
3.39 % of total footprint
5.76 % of lineitem (TPCH SF 1)
data type: int
encoding:
100%: Dictionary
(FixedSize2ByteAligned)

l_orderkey
size:35.501 MB
data type: int
encoding:
100%: Dictionary
(FixedSize2ByteAligned)

l_partkey
size:32.488 MB
data type: int
encoding:
100%: Dictionary
(FixedSize2ByteAligned)

l_extendedprice
size:35.107 MB
data type: float
encoding:
100%: Dictionary
(FixedSize2ByteAligned)

l_receiptdate
size:21.315 MB
data type: string
encoding:
100%: Dictionary
(FixedSize2ByteAligned)

Fig. 2. Treemap as displayed by the cockpit for the TPC-H dataset (scale factor 1). This diagram allows to quickly identify the largest tables and columns and offers detailed information about the currently used encoding schemes.

The browser-based cockpit's main visual components are three monitoring views and three configuration panels. The three monitoring views give an overview of the currently evaluated database instances and the current workload. The database views include charts depicting general metrics, such as latency, throughput, or CPU and memory utilization. Users can choose to either show the metrics for all instances (*Overview*) in a single graph or to display each instance separately side by side (*Comparison*, Figure 3). The *Comparison* view also allows for showing diagrams that cannot be aggregated, e.g., treemaps

[4]ØMQ project website: https://zeromq.org

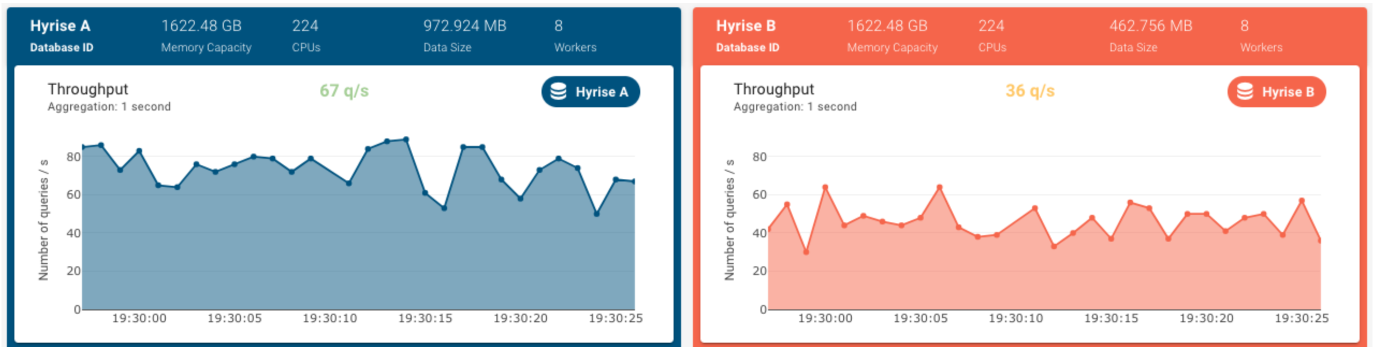


Fig. 3. Crop of the throughput and system details part of the comparison view as displayed by the cockpit. Both Hyrise instances have the same hardware configuration and run an identical workload, but the left was tuned by the autonomous clustering plugin and shows a better throughput.

(Figure 2) for per-attribute memory consumption or heatmaps (Figure 4) displaying access patterns per chunk and attribute. We display the most expensive queries per database instance and the runtime share of the different database operators in the *Workload Analysis* view to enable quick analyses of the currently processed workload.

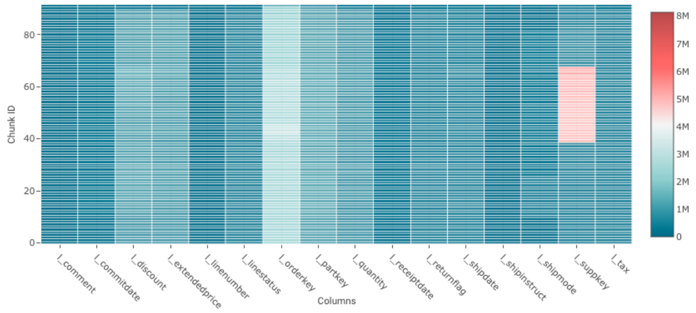


Fig. 4. Access heatmap as displayed by the cockpit for TPC-H's `lineitem` table. Cells represent segments, i.e., an attribute's part of a chunk (Section II-A). Warmer colors indicate frequent accesses; such information can be utilized for comprehending clustering decisions.

Configuration panels allow the user to adjust the workload and the plugins. The workload configuration allows loading and unloading of table data for the supported benchmarks, starting single or multiple workloads, and modifying the number of queries generated per second. Plugins can be (de)activated, and their settings, e.g., the memory budget, can be modified via the plugin panel. Further, the panel displays the log messages of the plugins. These log messages are also displayed in the aforementioned charts to enable the user to directly relate performance changes with plugin activity.

The flexible interaction with plugins per database instance enables users to exchange and compare plugins with each other. Thereby, not only the interplay of different plugins (cf. [9]) can be evaluated, but also alternative approaches to the same problem (e.g., various index selection approaches) can be easily compared.

B. Demonstration Scenario

Marilena is a database administrator of a large company. The company uses the relational database system Hyrise, which provides various means to self-optimize. This includes optimizations regarding, e.g., the main memory footprint, secondary indexes, or data clustering. While Marilena is interested in the advantages of all these approaches, she usually refrains from automated approaches as (i) their impacts are hard to predict, (ii) she has experienced devastating performance impacts when using heavy compression in database systems, (iii) she fears that these approaches fail when unexpected events, e.g., load spikes, occur, and (iv) she is afraid that combining multiple approaches might lead to undesired behavior. However, by using our cockpit for autonomous database systems, Marilena utilizes a recorded query trace of the production system to replay the workload of last year's Black Friday and the surrounding days to evaluate the impact of the offered plugins in different scenarios. She gains confidence in applying compression selections as she learns how much the system can be compressed without affecting the performance of the real system. Simultaneously, she learns that modern index selection approaches provide valuable configurations in an instant and that the improved pruning rates caused by automated table clustering quickly set off the initial costs. Combining the provided heatmap displaying access counts with the plugin's output, she comprehends its actions. As the cockpit also allows us to examine the correlation and interplay of autonomous components, Marilena balances the memory reduced via compression with the memory invested into additional secondary indexes.

REFERENCES

[1] D. V. Aken *et al.*, "Automatic database management system tuning through large-scale machine learning," in *SIGMOD*, 2017, pp. 1009–1024.
[2] A. Pavlo *et al.*, "Self-driving database management systems," in *CIDR*, 2017.
[3] Q. Meteier *et al.*, "Workshop on explainable AI in automated driving: a user-centered interaction approach," in *Adjunct Proceedings of AutomotiveUI*, 2019, pp. 32–37.
[4] S. Das *et al.*, "Automatically indexing millions of databases in microsoft azure SQL database," in *SIGMOD*, 2019, pp. 666–679.
[5] M. Dreseler *et al.*, "Hyrise re-engineered: An extensible database system for research in relational in-memory data management," in *EDBT*, 2019, pp. 313–324.
[6] M. Boissier *et al.*, "Workload-driven and robust selection of compression schemes for column stores," in *EDBT*, 2019, pp. 674–677.
[7] G. Moerkotte, "Small materialized aggregates: A light weight index structure for data warehousing," in *VLDB*, 1998, pp. 476–487.
[8] R. Schlosser *et al.*, "Efficient scalable multi-attribute index selection using recursive strategies," in *ICDE*, 2019, pp. 1238–1249.
[9] J. Kossmann *et al.*, "Self-driving database systems: a conceptual approach," *Distributed Parallel Databases*, vol. 38, no. 4, pp. 795–817, 2020.