Self-Driving: From General Purpose to Specialized DBMSs

Jan Kossmann
Supervised by: Prof. Dr. Hasso Plattner

Hasso Plattner Institute
August-Bebel-Str. 88
Potsdam, Germany

Jan.Kossmann@hpi.de

ABSTRACT

Large data sets, variable workloads of high complexity, and flexible cloud infrastructure make the administration of database systems at the same time more challenging and more important. In the future, self-driving database systems will utilize workload-driven optimization and machine learning techniques to generate forecasts of future workloads, decide upon which actions to take to process workloads most efficiently and to incorporate knowledge from past decisions into future ones without human intervention. But database systems are typically not designed having such capabilities in mind. We propose the architecture of a generalized framework that enables seamless integration of self-driving capabilities into database systems. Thereby, general-purpose database systems can transform themselves into systems tailored to a specific use case. Furthermore, we present our scalable approach to finding solutions for large problem instances of a physical design challenge, i.e., the index selection problem. Both concepts have been implemented and partly evaluated with real-world data on the research database system Hyrise.

1. INTRODUCTION & MOTIVATION

Today's relational database systems are general purpose systems. They are designed to handle most real-world use cases and workloads sufficiently. To achieve this, such systems rely on generic configurations and algorithms. However, to achieve cost efficiency, fully utilize existing hardware resources, and achieve optimal performance such systems need to be optimized. Database administrators (DBAs) are responsible for manually tuning and configuring systems in order to meet the specified Service Level Agreements (SLAs) and deliver optimal performance. Tuning and configuration can include decisions upon the available physical resources (e.g., available network bandwidth, CPUs, main memory), knob configuration (e.g., buffer pool size, number of concurrent threads, different concurrency mechanisms), and physical database design (e.g., partitioning or indexes). The recent work of Kraska et al. [7] on learned index data structures goes one step further. The authors do not only present very interesting results by outperforming traditional data structures that were tuned for decades. Also, the demonstrated solution of replacing core components of database systems with machine learning models opens up opportunities to rethink the tuning and configuration of database systems in general. This might lead to a point where database systems could autonomously optimize themselves and adjust their configuration for the currently processed workload. In Section 4 of this work we will detail how we equipped the relational database system Hyrise with self-driving capabilities and preliminarily evaluated our approach with real-world data.

There are three main trends which strengthen and motivate the need for such *self-driving* database systems: (i) more variable and complex workloads, (ii) the shift from on-premise to cloud deployments, and (iii) an increased number of available options and (hardware) configuration dimensions. We will detail these trends in the following paragraphs.

First, today's relational database systems have to handle a broad variety of complex, volatile, and combined workloads. HTAP workloads contain transactional and analytical queries. The recent increase of machine learning applications on all sorts of data sets increases the workload complexity even further. Data is not necessarily extracted to process data- and computation-intensive machine learning tasks, but the computations are executed directly on a database system [9]. Thereby, workloads are more variable, harder to assess and, in the end, manual tuning gets increasingly challenging. Simultaneously, the need for high performance to fulfill these complex workloads makes proper configuration and tuning even more important.

Second, the shift from on-premise to cloud database solutions is another motivation. Nowadays, many cloud providers offer the opportunity to deploy (traditional) relational database systems on cloud infrastructure. There is a variety of reasons to choose cloud over on-premise database deployments. Flexibility caused by scalability and elasticity, cost-effectiveness because of better resource utilization, and the avoidance of the need to own and maintain physical infrastructure to name a few. Both parties cloud providers, and cloud customers have an interest in self-driving database systems. Cloud providers maintain a large number of systems. Cost-efficiency and optimal resource utilization are a necessity to be competitive. Hence, proper tuning and configuration of their systems are of high importance. Manual tuning of these systems by DBAs would be cumbersome, time-consuming, and, hence very expensive. On the other hand, cloud customers do also have an interest in optimized systems. They usually pay for units of computation time per machine and they also pay more for larger machines.

Proceedings of the VLDB 2018 Ph.D. Workshop, August 27, 2018. Rio de Janeiro, Brazil.

Therefore, they have an interest in renting fewer machines, smaller machines and renting these for the shortest possible periods. Thus, utilizing the rented resources in the most performant way without investing manual administration effort is necessary to avoid unnecessary costs.

Furthermore, database systems as well as the underlying hardware offer increasingly more configuration and tuning options. The number of knobs of database systems is growing with every release of a new version. In 2016, MySQL offered more than 500 tunable knobs. In addition, changing the state of one knob might affect the impact of other knobs [2]. Hence, they cannot be considered independently. Some new hardware mechanisms can be dynamically configured which adds further tuning dimensions, e.g., Intel's Cache Allocation Technology [1]. To exploit such mechanisms to their full extent they need to be used taking the currently processed and soon to be processed workload into account.

All the above-mentioned aspects demand specialized configurations for each database deployment, but the effort to build those would be too high. If we leverage workload-driven optimization and machine learning techniques (ranging from simple techniques, for example, linear regression and decision trees to complex neural networks), we can transform general purpose database systems into database systems tailored to a specific use case. This might cause computationally intensive calculations. However, the computation power of new CPUs, GPUs, and TPUs [6] enables efficient processing of large problem instances.

2. RESEARCH ISSUES

In this section, we want to highlight research problems originating from self-driving databases. Developing self-driving database systems is a complex problem involving multiple (database) components. Therefore, there is no single research problem, but a whole variety of problems arise. We see three main areas that must be mastered to enable self-driving database systems.

1. Tuning: The core problem of self-driving database systems. The main task of such systems is to take certain actions to increase performance with regards to, e.g., latency, memory consumption, or energy consumption. Manifold reasons make tuning difficult. First, the sheer complexity and problem size for large database deployments. Furthermore, tuning decisions might depend on other tuning decisions and considering them separately leads often to suboptimal results [2]. Also, heavyweight operations to put the tuning choices into place, for example, repartitioning of large tables, can easily negatively affect system performance.

2. Feedback loop: Self-driving database systems have to evaluate themselves and learn based on their past tuning decisions in order to be independent of human oversight. This is particularly hard with volatile workloads that change frequently. The system's actions need to be assessed even though the new workload might look completely different as previously anticipated.

3. Forecasting: Tuning choices for the current workload can be affected if the system's workload is going to shift in the next few minutes. Workloads can change, e.g., based on seasonal effects or load peaks caused by high usage. Therefore, detailed forecasts are necessary to anticipate future workloads.

In addition, today's database systems were not designed with self-driving capabilities in mind. Thus, we want to investigate how a database design could support the above presented main requirements without sacrificing performance or well-established functionalities.

Furthermore, we work on finding efficient and robust[1] solutions, even for large problem instances, for tuning problems. With more volatile and diverse workloads [8] the requirements for high-performance operation of the database system are constantly changing. Finding the most promising set of actions to achieve the best performance for the currently processed workload is of high importance. The amount of time it takes to find such a set of actions is not less important. If the solution of a problem instance takes large amounts of time, the workload could have changed, thus rendering the solution out-dated.

3. RELATED WORK

This section should demonstrate why the currently existing solutions are not sufficient. The beginnings of self-driving databases can be seen as early as in the 1970s where self-adaptive databases could automatically tune parts of their physical design [5]. Later, commercial database systems [3, 13, 12] introduced tools and advisors that supported DBAs in tuning tasks. In most cases, these tools do not apply foresight to anticipate future workloads but mostly take reactive measures. In addition, advisors still involve manual human effort to take the final tuning decision.

The modern term of self-driving databases [11] targets a much wider and more holistic goal. These systems should be completely self-sufficient and maintain all aspects that are necessary to make database systems run permanently and efficiently without human intervention. The recent work of Aken et al. [2] presents a system that combines multiple machine learning techniques to automatically manage a database system by tuning its knob configuration. Tuning decisions are based on a large repository of previously tuned systems that processed a similar workload. This is a great step forward, but we see multiple open challenges here. For example, the approach cannot yet work holistically. It only handles knob configurations. Physical database design is not targeted at all. Instead, it expects a reasonable physical design to be in place. This is a fair assumption at this point of time, but in our opinion separating knob configuration and physical design might, in the end, lead to suboptimal results. Therefore, we favor a holistic approach. Furthermore, in order to fulfill this tuning process, the system's workload is characterized. Their system characterizes workloads based on the database system's internal runtime metrics, e.g., acquired locks or written/read pages and not on the logical query level. We argue that only because workloads show the same runtime metrics they cannot necessarily be classified as equal or similar. Especially high variable, complex workloads that combine, for example, HTAP and machine learning queries will be hard to characterize with such approaches.

There are state-of-the-art solutions for many tunable aspects, e.g., indexes [4], views [10] or knob configuration [2].

[1]Robust solutions offer not necessarily the best performance, but aim to provide acceptable performance in most cases.

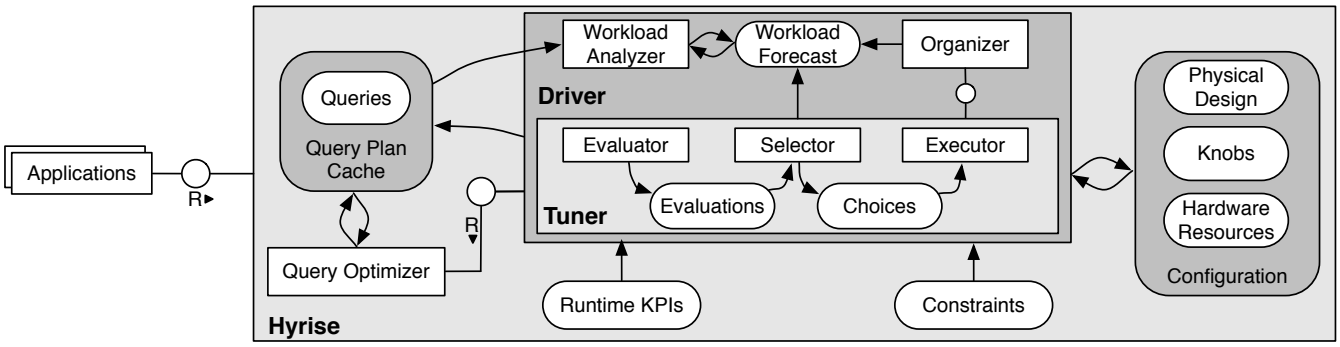

Figure 1: Diagram of the Generalized Self-Driving Framework

However, for large problem instances, these solutions show unacceptable runtimes or limit candidate sets a priori resulting in far from optimal results. If database systems should become fully autonomous, minutes of runtime for tuning a single aspect is not acceptable if dozens of tuning decisions have to be taken. We identify this as an opportunity for further research especially when problem instances grow through ever-increasing database sizes.

4. RESEARCH PLAN

In this section, we describe the approach we pursue, which work is currently conducted and planned for the future, and how we plan to evaluate the proposed solution.

4.1 Approach

We have developed the architecture of a generalized self-driving framework and implemented this in the new version of the Hyrise[2] database system. Hyrise is a relational main-memory database system that stores tables in column-major format. The recent rewrite of the Hyrise codebase gave us the opportunity to take all design decisions with self-driving capabilities in mind and plan the system's architecture accordingly.

Figure 1 depicts how we integrate the necessary components to enable a self-driving database system into Hyrise. For simplicity reasons not all interfaces and components can be visualized in this figure. The Query Optimizer generates efficient query plans from SQL strings. These plans are fed into the Query Plan Cache. In case a query plan has already been cached, the planning phase can be omitted. The query plan cache plays an important role because it holds a representation of current and past workloads. This is necessary to generate forecasts for future workloads. The central component responsible for self-driving is the driver. It is responsible for ensuring that the database system most efficiently processes the incoming workload while considering specified constraints and at the same time not wasting resources. The Workload Analyzer generates workload forecasts of future workloads based on the data from the Query Plan Cache. These forecasts have two main purposes. First, they serve as input for the Organizer. The Organizer controls and supervises the tuning process: it determines when to start or abort tuning based on workload forecasts. It also enforces the time constraints to ensure that the tuning process itself does not consume too many resources. Second, they are used by the tuner. It finds its decisions based on workload forecasts. The Tuner itself consists of three components: The Evaluator component evaluates all possible actions and assigns a negative or positive desirability expressing the expected benefit for the system, a confidence that is associated with the desirability, and a cost. We expect one evaluator to exist for each tunable entity, for example, indexes, partitions, knobs. Evaluators might consult the query optimizer for (hypothetical) cost estimations. Its output is consumed by a selector which selects the most-beneficial actions while considering a cost budget. There could be multiple exchangeable selectors each with a different strategy, e.g., greedy, heuristic-based or solver-based strategies, which show their strengths in different scenarios. In the end, the Executor takes care of executing the proposed actions by changing the configuration. This architecture offers the necessary flexibility while avoiding redundant components at the same time. The selector can be exchanged by the organizer based on past tuning experiences or current needs. While we need one evaluator implementation per tunable aspect, multiple instances of a single selector implementation can be used for multiple tunable aspects. The driver does also draw conclusions from past tuning runs. Therefore, it continuously monitors the database system's runtime KPIs and observes whether the desired effects occurred. It might also change the query plan cache to trigger the eviction of plans that became out-dated after a tuning run. Constraints contain hardware resource restrictions and service level agreements.

Our self-driving framework can benefit from Hyrise's flexible architecture. Tables are physically partitioned into small (consisting of around 100 000 rows) chunks. Small chunks offer high flexibility regarding data placement/movement, indexing, compression and re-encoding. All of the above could be triggered by tuning decisions of a self-driving database system. Hyrise's architecture shows further benefits. Normally, for systems storing the data in large monolithic blocks containing dozens of millions of tuples the time necessary for automatic tuning can be hard to estimate. Also, applying changes, e.g., re-encoding a large table, is a heavyweight, time-consuming operation. Chunks divide the problem in multiple smaller problems, allowing more lightweight operations and better estimations of the necessary effort. In addition, chunks are append-only containers. Once they reach their capacity they become immutable. Therefore, the self-driving database systems can make assumptions about the

[2] `https://github.com/hyrise/hyrise`

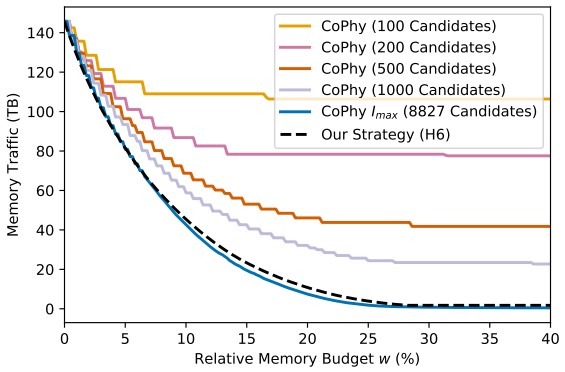

Figure 2: Preliminary evaluation. Lower memory traffic is better. The state-of-the-art approach (CoPhy [4]) is in few cases slightly better than our strategy. Cophy's runtime is either similar for few candidates or orders of magnitude higher for many candidates.

underlying data with a higher certainty which leads in the end to better tuning results.

4.2 Current and Future Work

The concept of the generalized self-driving framework is currently implemented in Hyrise and under evaluation. We have conducted first experiments on real-world enterprise data and the results look promising.

Furthermore, we have started to investigate how large problem instances of physical design challenges can be solved efficiently without restricting the candidate set or relying on external solvers. In the beginning, we studied the well-researched index selection problem. For a given workload and memory budget the best index configuration is searched. The preliminary results, also depicted in Figure 2, indicate that we outperform current state-of-the-art approaches in both scalability and solution quality for real-world as well as synthetic workloads. Our approach relies on a generalized recursive solution principle utilizing the query optimizer's cost model. It is completely integrated with the above presented generalized self-driving framework. However, the number of optimizer calls is greatly reduced by the approach's recursive structure.

The next step is to create reliable workload forecasts on the basis of the query plan cache data. Precise workload forecasts do not only enable the tuner to find the best solutions for a certain workload, but also to find robust solutions that work acceptably well in cases where workloads that were assumed to be less likely need to be processed. Afterwards, we want to investigate how a feedback loop could be established. The driver needs to monitor and assess the effect of past tuning decisions and influence the tuner's behavior for future decisions.

4.3 Planned Evaluation

Current evaluations of our approach were conducted with data and workloads from a productive real-world Enterprise Resource Planning application of a Fortune Global 500 company as well as with synthetic benchmarks. At the moment, we are in the process of getting access to several productive cloud database systems. These systems would offer both, the possibility to learn based on data from real-world systems and the opportunity to evaluate our approach in practice for a large number of systems. We envision to additionally take more complex synthetic benchmarks, e.g., the TPC-DS benchmark into account to provide reproducible examples.

5. CONCLUSION

We have presented our architecture of a generalized self-driving framework that integrates with typical database components. The framework is capable of handling the three main challenges for self-driving database systems: forecasting of future workloads, adjusting the configuration accordingly, and learning based on these adjustments. In addition, we have demonstrated our concept and evaluations for the efficient solution of large problem instances of physical design challenges. Upcoming tasks include the implementation of workload forecasting, assessing the impact of past configuration decisions, and an extensive evaluation with more real-world systems.

6. REFERENCES

[1] Introduction to Cache Allocation Technology. `https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology`. Accessed: 2018-04-10.

[2] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the SIGMOD Conference 2017*.

[3] S. Chaudhuri and V. R. Narasayya. Self-tuning database systems: A decade of progress. In *Proceedings of the 33rd VLDB Conference 2007*.

[4] D. Dash, N. Polyzotis, and A. Ailamaki. Cophy: A scalable, portable, and interactive index advisor for large workloads. *PVLDB*, 4(6):362–372, 2011.

[5] M. Hammer. Self-adaptive automatic data base design. In *American Federation of Information Processing Societies: 1977 National Computer Conference*.

[6] N. P. Jouppi, C. Young, and N. P. et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th ISCA 2017*.

[7] T. Kraska, A. Beutel, and E. H. C. et al. The case for learned index structures. *CoRR*, abs/1712.01208, 2017.

[8] J. Krüger, C. Tinnefeld, M. Grund, A. Zeier, and H. Plattner. A case for online mixed workload processing. In *Proceedings of the Third DBTest 2010*.

[9] S. Luo, Z. J. Gao, M. N. Gubanov, L. L. Perez, and C. M. Jermaine. Scalable linear algebra on a relational database system. In *33rd IEEE ICDE 2017*.

[10] I. Mami and Z. Bellahsene. A survey of view selection methods. *SIGMOD Record*, 41(1):20–29, 2012.

[11] A. Pavlo, G. Angulo, and J. A. et al. Self-driving database management systems. In *CIDR 2017*.

[12] K. Yagoub, P. Belknap, and B. D. et al. Oracle's SQL performance analyzer. *IEEE Data Eng. Bull.*

[13] D. C. Zilio, J. Rao, and S. L. et al. DB2 design advisor: Integrated automatic physical database design. In *Proceedings of the Thirtieth VLDB 2004*.