Self-driving Databases: An Overview of Autonomous Database Management Systems

Chengdong Cao, 1336039   Siqi Zhong, 1306166
Taylor Tang, 1323782   Wenquan Zhao, 1296739

May 24, 2023

Abstract

In the era of big data, the exponentially grown complexity and scale of modern data have intrigued an increasing demand for cloud computing and autonomous database management systems. Traditional database systems, which require manual tuning, are struggling to keep pace. This is where the concept of self-driving databases emerges. While the concept has been researched since the 1970s, its partial actualization has only been realized in the recent decade. The advent of advanced algorithms and machine learning techniques has significantly accelerated the development of self-driving databases.

Despite being an emerging research topic, comprehensive reviews on self-driving databases remain scarce. Therefore, there is a pressing need for a survey that explores the various avenues of optimization that self-driving databases are currently undergoing. In this survey, we have investigated the four key areas: (1) Configuration tuning, tracing the evolution OtterTune to CDBTune, and QTune; (2) Data access, encompassing performance optimization strategies for indexing and partitioning; (3) Query plan optimization, including the optimization of query operator execution order and query rewriting; (4) Query execution, focusing on the query processing in the system after the query plan optimization phase, including adaptive operator selection and query scheduling. For each area, we elucidate the motivation behind the development of each technique and the principles underpinning the models and provide a detailed comparison. While the realization of a fully self-driving database is still a distant goal, we conclude that the field of self-driving databases, with its integration of machine learning techniques, is significantly promising.

Keywords- Self-driving Database; Configuration Tuning; Data Access; Query Optimization; Query Execution

1 Introduction

In recent years, self-driving database management systems(DBMS) have gained more attention due to the increasing complexity of modern database environments and escalating data volumes and query workloads. Traditional manual database optimization techniques supported by database administrators have proven inadequate, therefore requiring an architectural shift towards autonomous operations for database management [1].

Pavlo [2] described the progression of DBMS from self-adaptive databases in the 1970s to self-driving databases in the late 2010s illustrating the ongoing drive for automation and optimization in data management. Self-adaptive databases, which automatically selected efficient query execution methods and optimized physical database design (like index selection and partitioning), began the exploration to automate database management. This idea evolved into self-tuning databases in the 1990s, with advisor tools aiding database administrators in choosing optimal indexes, partitioning schemes, and knob configurations. The rise of cloud computing in the early 2010s necessitated further automation due to the scale and complexity of cloud databases, with companies like Microsoft modeling the resource utilization of the cloud database from internal telemetry data and dynamically adapting allocations to meet resource constraints. By the late 2010s, the concept of self-driving databases emerged, aimed at automating all aspects of DBMS, including the selection of actions to improve performance, the timing of action application, and learning from past actions to refine future decisions. Huawei's GaussDB [3] stands as a pioneering example of a self-driving database, demonstrating the vast capabilities of a fully autonomous DBMS that is powered by machine learning. Despite the challenges of continuous environmental change and workload variations, efforts towards building fully autonomous DBMS are advancing, promising future databases that can perform optimally under various workloads, with minimal human intervention.

As we advance further into the era of big data and machine learning, self-driving databases are no longer a concept of the future [4]. The introduction of AI and machine learning to the field of database management has created possibilities for adaptive optimization techniques that are more dynamic and proactive than ever before. Machine learning algorithms enable these databases to learn from past actions and their outcomes, identify patterns, and make informed predictions about future workloads. This predictive capability enables self-driving databases to anticipate potential issues and adjust their operation in advance to prevent problems rather than just reacting to them passively.

In this survey, we have delved into the transformative potential of artificial intelligence and machine learning in the realm of self-driving databases, examining four critical aspects: configuration tuning, data access, query plan optimization, and query execution. For configuration tuning, machine learning algorithms have supplanted the traditional reliance on human expertise, optimizing a multitude of performance-influencing configurations autonomously. In terms of data access, AI-enhanced methods have revolutionized indexing and partitioning, adapting in real-time to maximize performance based on current workload and data distribution. The area of query plan optimization has seen substantial efficiency improvements, with machine learning accurately predicting and selecting cost-effective execution plans. Finally, query execution has been streamlined with adaptive operator selection and scheduling, allowing for real-time adjustments for enhanced efficiency. As AI and machine learning become increasingly embedded within database management, the traditional database are promising shift towards fully autonomous, self-driving databases, marking a paradigm shift in data management practices.

This survey is organised as follows. Section 2 provides a related work of the past relevant research on the four aspects, including configuration tuning, data access, query optimization and query execution. After that, the advantage and the disadvantage of key approaches will be discussed in the section 3. Finally, a brief conclusion and possible future direction will be given in Section 4.

2 Related Work

2.1 Configuration Tuning

The configuration tuning of DBMS is one of the essential parts and challenges of the self-driven database since there are hundreds of configuration ‘knobs’ that impact performance and scalability of the databases, including memory allocation, cache sizes, concurrency settings and so on. Traditionally, configuration ‘knobs’ are tuned based on the experience of database administrators (DBAs). However, these ‘knobs’ are not standardized, not independent and not universal and with the increase in size and complexity of the database, optimizing a DBMS to meet the needs has outstripped human capabilities [5]. Therefore, some automated ‘knobs’ configuration tuning tools are proposed in self-driven databases.

2.1.1 Machine learning based approach

The OtterTune framework, proposed by Aken et al. [5], used the Gaussian Process Regression to tune the ‘knobs’ configurations automatically based on the importance of ‘knobs’ and the system workload. This Gaussian Process Regression was trained by utilizing the previous session of tuning and the workload from the DBMS internal runtime statistics metrics during the execution. In addition, it used Lasso in linear regression to determine which knobs should be tuned based on the correlation between the special knob and the system’s overall performance. Finally, mapping the current workload to previous known workloads, so that it can use the previous experience to reduce the usage of time and resources to tune the important knobs in DBMS for the new application by using Gaussian Process regression. The result showed the configuration generated by OtterTune faster than 94% of expert DBAs, and 58-94% reduction in latency compared to default settings and the other tuning tools.

2.1.2 Deep learning based approach

However, machine learning-based OtterTune is highly dependent on messy high-quality training data which is difficult to collect. In addition, it ignored the dependence of knobs in high-dimensional continuous space. Therefore, the CDBTune, which was introduced by Zhang et al. [6], adopted reinforcement learning with a try-and-error strategy that can learn the knob set with a limited number of training samples. Therefore, it can find the optimal configurations by utilizing the deep deterministic policy gradient method within the high-dimensional continuous space. While conducting online tuning, CDBTune gathers the query workload from the user and other parameters are generated based on the query workload, so that the generated configuration can adapt to the actual workload to achieve high throughput and low latency.

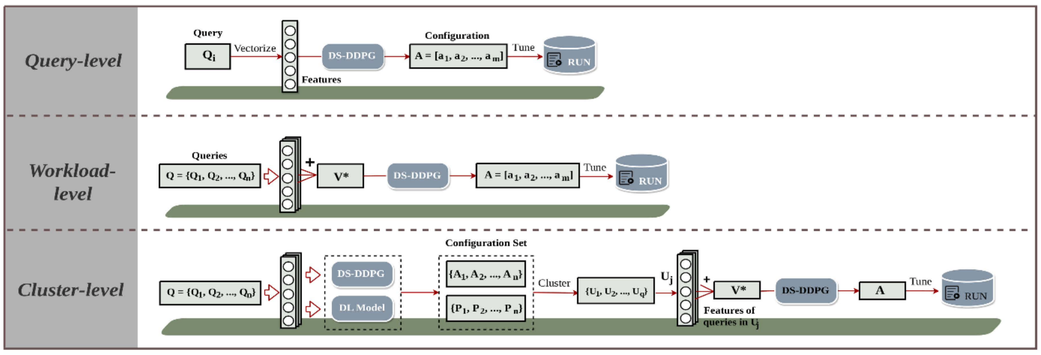


Figure 1: Workflow of QTune [7]

Furthermore, the query-based configuration tuning tool QTune not only focused on coarse-grained tuning, such as workload-level tuning, but also focuses on fine-grained tuning, and provides three tuning granularities: query level, workload-level and cluster-level tuning as shown in the Figure 1 [7]. QTune tunes the configuration by using deep reinforcement learning with the double-state deep deterministic policy gradient model. For the query-level tuning, QTune tunes the configuration based on the SQL query's unique characteristics, at both the session and system levels, which is well-suited for optimizing latency, particularly for handling a limited number of intricate or resource-demanding queries. The workload-level tuning focuses on the whole query workload, which can attain high throughput through the methods like locking and isolation to process multiple queries concurrently. However, it may also result in increased latency as the overall configuration values might not be optimal for different queries. The cluster-level tuning can achieve both low latency and high throughput since it groups similar queries with compatible configuration values and adjusts settings at the group level to facilitate parallelization. As a result, the QTune outperformed the DBAs, BestConfig, CDBTune and OtterTune.

2.2 Data Access

In database management, data access plays a crucial role in ensuring efficient data processing and retrieval. Indexing and partitioning serve as the primary techniques for enhancing data access. Indexing expedites the process of finding specific rows within a database by creating and maintaining data structures, such as B+ trees or LSM trees, that associate key values with corresponding rows. Partitioning, on the other hand, involves dividing a substantial database table into smaller, manageable segments to boost query performance.

With the advent of self-driving databases, these techniques have evolved to become more adaptive and automated. Self-driving databases employ machine learning algorithms to create and manage indexes, as well as determine the optimal partitioning strategies based on the current workload and data distribution. By continually monitoring and adjusting to workload changes, these databases maximize performance and efficiency, reducing the need for manual intervention from DBAs. The integration of machine learning and automation enables self-driving databases to deliver enhanced indexing and partitioning, optimizing data access and overall database performance.

2.2.1 Adaptive indexing

Data indexing is essential for efficient database processing, but it involves storage needs and required updates as the database content changes. Traditionally, this process is manual and requires in-depth knowledge, continuous monitoring, and adaptations to workload shifts, making it time-consuming and error-prone. Given these limitations and the importance of performance, automated index creation and management are crucial for self-driving databases, which cater to current database needs and query patterns while adapting to future workloads.

Adaptive indexing fully automates the process of managing database indexes. Unlike offline indexing switches that rely on a set of predefined indices or online index selection methods that wait for a specific threshold to update indices, adaptive indexing operates dynamically, incrementally adjusting the indexes as needed. This approach allows index creation and optimization to occur as side effects of query execution, increasing efficiency. Database cracking as proposed by Idreos et al. [8] is an adaptive indexing technique designed to minimize index maintenance costs and efficiently adapt to changing query patterns. Instead of pre-sorting and indexing the entire database, database cracking incrementally and adaptively sorts and indexes data based on the incoming queries. In detail, Cracking is the process of breaking the data that is associated with a query into smaller partitions upon receiving a query, It then reorganizes the partitions to create a partially ordered index. As more queries are processed, the database cracking technique continues to refine the partially ordered index by further partitioning and reorganizing the data. This results in improved query performance over time.

Consider a database storing product information for an online shopping business, where customers regularly retrieve data by filtering specific conditions like product specifications. As customer shopping behavior changes, the adaptive indexing algorithm dynamically and incrementally adjusts indexes: the more often a query is executed, the more the index adapts to support query execution. Data columns not frequently used for searching will not be indexed. This technique offers better performance than online indexing systems because it avoids the inefficiencies that arise when the system is burdened with the current indexing but has not yet met the conditions to utilize new indexing.

2.2.2 Learning-based indexing

Machine learning-based indexing searches for the most compact and efficient index structures that best suit a given database. It aims to solve the problem DBAs face by using machine learning techniques such as neural networks, decision trees, and reinforcement learning to manage indexes. Given enough data, machine learning-based indexing explores various indexing strategies while adapting to workload changes and maintaining an optimal hybrid-indexing structure best suitable for the current database. This could involve the use of B+ trees for balanced, sorted data structures, or LSM trees for efficient write-heavy workloads.

Learned index structure as introduced by Kraska et al. [9] is one technique of using machine learning models to replace traditional index structures. Unlike a conventional augmented index, the learned index structure is trained with data in the database, so that when a query is executed, the model can predict the location of a key based on the patterns and distribution of the data. In detail, a new model is trained with existing data inside a database. When a query is received, the learned index structure makes a prediction of the approximate location of the key of the current index. In such way, the search space is narrowed and the amount of time is reduced. As the stored data and query pattern changes over time. The model is continuously trained and adapted, which results in a more accurate prediction.

CrimsonDB as introduced by Idreos et al. [10] at Harvard University uses a Data Calculator to explore a vast design space of data structures. By employing fine-grained design primitives and learned cost models, it explores a wide range of possible data structure designs and computes performance costs without implementing the data structure. These facilities CrimsonDB to adapt its core data structure to varying workloads, hardware and other parameters, thus optimizing read and write performance.

Today's major cloud database service providers have assimilated similar techniques to bolster their service capabilities. For instance, Microsoft Azure SQL Database provides a feature for "index recommendations" in conjunction with an "auto-implementation" option [11]. Upon deployment, the service employs a "Tuning Advisor" to assess the potential benefits and drawbacks of utilizing a new indexing structure. It then selects the most beneficial option and evaluates its effectiveness. This comprehensive automated indexing service significantly alleviates the task of performance tuning, thereby simplifying database management.

2.2.3 Multi-attributes partitioning

Partitioning divides a large database into smaller, more manageable pieces, improving performance, manageability, and availability. Conventional partitioning methods, such as range, list, hash, and composite partitioning, have unique advantages but can be drawbacks in modern DBMS. They require strong domain knowledge and may struggle to adapt to constantly changing workloads. To optimize database performance, recent approaches utilize machine learning algorithms to automatically analyze workload, data distribution, and query patterns, determining the optimal partitioning schema that enhances performance and resource utilization.

One partitioning technique utilized by self-driving databases is multi-attributes partitioning. Compared to the conventional single attributes partitioning, the use case of multiple partitioning is much wider as it can reduce the amount of data that needs to be scanned for a given query, and consequently improve query performance. The complexity of composing multi-attributes partitioning grows with the number of attributes, therefore modern self-drive databases use machine learning to decide how to combine attributes more efficiently.

2.2.4 Adaptive partitioning

Moreover, one could use adaptive partitioning to further maintain the high performance of partitioned data. This technique adjusts the partitioning scheme dynamically based on the current workload and data distribution. This approach ensures that the current partitioning scheme always matches the current needs of the database, maximizing performance and efficiency.

For example, Van et al. [5] proposed a machine learning based self-partitioning technique that gathers performance data and configuration settings from various databases. This collected data is then used to train machine learning models to predict the most effective partitioning schemes for a given workload. If, over time, the workload shifts to another scenario that would benefit more from another partitioning scheme, the system could identify this change and adjust the partitioning scheme accordingly. It might suggest a new partitioning strategy, such as list or hash partitioning, that would better suit the evolved workload.

Today's major cloud database service providers have assimilated similar techniques to bolster their service capabilities. For example, Oracle Autonomous Database offers the feature of "Automatic Partitioning." [11] This service, upon deployment, identifies candidate tables for automatic partitioning through a thorough analysis of the workload associated with those tables. It then evaluates various partition schemes based on this workload analysis and automatically applies partitioning to both the tables and their indexes to enhance performance and facilitate improved management of larger tables. This effectively streamlines the database management process, increasing efficiency and productivity.

2.3 Query Plan Optimization

Query optimization can significantly enhance the performance of database systems. In the context of large-scale data processing tasks, even minor improvements in the execution of each query can have a substantial impact on overall system efficiency. There are numerous critical obstacles to address, including cardinality estimation and cost modelling, handling the organization of tree-based query plan structures, and the creation of a search strategy. Fortunately, exciting opportunities lie ahead in the replacement of human-devised cost models with neural networks, and the application of reinforcement learning methodologies. The potential for continuous improvement and reduction in training time for specific databases further underscores the pivotal role of machine learning and neural networks in the evolution of self-driving databases [12].

Common methods of query optimization include improving the query execution plan and rewriting the query. Plan improvement does not alter the query itself, but rather modifies the planning of the operator execution order within the query. Conversely, query rewriting generates a new query, which is equivalent to the original, but executes in less time. Both plan improvement and query rewriting utilize two key elements of query optimization, cardinality estimation and cost modelling, which involve predicting the cost of various execution or query rewriting plans and selecting the one with the lowest cost.

2.3.1 Physical optimization of query execution plan

Marcus et al. [12], first demonstrated an entirely learned query optimizer in 2019, referred to as Neo (Neural Optimizer). The term "entire" denotes a comprehensive, end-to-end solution encompassing join ordering, physical operator selection, and index selection. The training process employs a conventional query optimizer as a source for expert demonstrations, along with a reward function that promotes the choice of cost-efficient plans. The performance of Neo can match that of leading commercial optimizers, such as Oracle and Microsoft, and occasionally even exceed them.

As an end-to-end learned query optimizer, each component of Neo is supported by a machine learning model.

Firstly, query features are encoded into vectors, followed by the encoding of the query execution plan. A deep neural network (DNN), serving as the cost model, utilizes these two encodings to predict the final latency of query execution plans (QEPs). Secondly, Neo employs tree convolution methods to capture the local query features to conduct a DNN-guided best-first search through the execution plan space. Thirdly, the process of cardinality estimation is rooted in either histograms or a vector embedding scheme that has been learned, paired with a model that has been learned. Lastly, Neo integrates these components into a comprehensive query optimizer using reinforcement learning and learning from demonstration.

Neo surpassed previous open-source query optimizers in performance and matched the standards of commercial optimizers in terms of latency and training time. However, there was room for improvement in its adaptability. Also, the process of independently constructing execution plans could be costly, potentially extending the training duration. After two year of investigation, Marcus and colleagues unveiled the Bandit optimizer, Bao, in 2021 [13].

Instead of building or developing an optimizer from the beginning, Bao utilizes an existing optimizer, such as PostgreSQL's optimizer mentioned in the article, to learn when to enable or disable specific features for each query. The authors propose a finite set of hint sets, which are the query execution strategies that the query optimizer applies to incoming queries. This methodology treats each hint set as an "arm" and the ensemble of query plans as the "context" within the confines of a multi-armed bandit problem, which is contextual in nature. This strategy fuses advanced tree convolutional neural networks with Thompson sampling. The tree convolution model assists in identifying key patterns in query plan trees, which are generated by the underlying query optimizer. Thompson sampling is specifically designed to tackle contextual multi-armed bandit problems.

Combining these techniques, when a query arrives, the system chooses a hint set, performs the resulting query plan, and receives a reward. For each plan selected and query executed, the plan tree and the performance of the execution are stored as a part of the experience E. From the experience E, Bao learns a model M_θ that anticipates which hints will optimize performance for a specific query. Exploration and exploitation are balanced by sampling the prediction parameter theta from the posterior distribution $P(\theta|E)$. As time progresses, Bao continually improves its model to more precisely forecast which hint set will offer the most significant benefits for an incoming query.

2.3.2 Logical query rewrite

In addition to the extensive research on cardinality/cost estimation of query execution plan, there are also a few attempts focusing on rewriting the queries for the best execution performance. Query rewriting means modifying a SQL query into an equivalent form, but with improved performance. Heuristic strategies modifying the query can be significantly influenced by the sequence of rewrites and the characteristics of the query. Determining the optimal rewrite sequence for each query presents various challenges, such as how to represent numerous possible sequences, how to efficiently select the best sequence, and how to quantify the cost-saving potential of a rewrite.

Zhou et al., 2021 [14], proposed a learned query rewrite system that employs Monte Carlo Tree Search (MCTS), called LearnedRewrite. They designed a policy tree to present the numerous possible query rewrites, with the original query acting as the root and each node representing a rewritten version of its parent node. A DNN is employed to estimate the performance of each rewritten query through learning. The training data for this model comprises the queries along with their rewritten formats and their performances. They utilized the upper confidence bound (UCB) to define the node utility, using it as the selection policy for MCTS to balance exploration and exploitation. To achieve high speed of searching, the search process is conducted in parallel to find the optimal node among the leaves.

2.4 Query Execution

Query execution is a program launched after the query optimization phase, where the actual processing of the query is performed. The query optimizer, a critical precursor to this phase, generates an efficient query execution plan. According to this optimized execution plan, the query execution engine gets access to the database system to fetch and process the data. The outcome of this execution process is the final result set, finely tailored to satisfy the user's query. The specific methods of query execution are constantly innovating during the development of self-driving databases. Innovations like adaptive operator selection and adaptive scheduling during query execution push the query execution process into more efficient and intelligent ways, ultimately paving the way to truly self-driving databases.

2.4.1 Adaptive Operator Selection

During query execution, traditional operator selection methods rely heavily on cost models and heuristics. Unfortunately, these approaches often struggle with varying real-time execution environments, leading to sub-optimal query performance. Adaptive databases provide a dynamic solution to this dilemma by employing adaptive operator selection techniques. Adaptive operator selection dynamically selects the most appropriate physical operator for each logical operator in a query plan, continuously adapting to the real-time execution environment. Operator optimization in the query optimization phase of a self-driving database often uses inter-query learning, which learns from lessons learned from past query executions and outputs a better query execution plan. In comparison, the operator selection during the query execution phase of a self-driving database utilizes intra-query learning, which learns from scratch based on the current query execution. Adaptive operator selection using intra-query learning makes autonomous fine-tuning possible by continuously refining and enhancing its policies to optimize query execution. In contrast to the traditional methods based on fixed-cost models and heuristics, adaptive operator selection improves accuracy, reduces errors, and ensures continuous performance improvement. In addition, by automatically adapting query execution to changing data and workload conditions, these techniques significantly improve overall efficiency.

SkinnerDB, a database system proposed by Trummer et al. [15], leverages reinforcement learning to achieve adaptive operator selection, specifically, to uncover near-optimal join orders during query execution. Instead of the conventional approach of basing decisions on static statistics or historical workloads, SkinnerDB begins each query afresh, without any prior assumptions or knowledge. The system's unique approach partitions the query execution into numerous small time slices, with the progress measured after each segment. At the start of each slice, based on the measured progress and the quality of the current join order, SkinnerDB selects the join order that appears most promising. This enables the system to dynamically optimize its performance during query execution rather than during the query optimization phase. Notably, SkinnerDB's reinforcement learning approach allows it to adapt to changing data distributions and workloads, providing a form of 'intra-query' learning - learning from the current query execution to optimize the remaining execution of that same query. Despite the potential overhead introduced by the implementation of reinforcement learning, the practical experiments demonstrate robust performance even for complex queries, given adequate data for processing.

Invented by Kaftan et al. [16], Cuttlefish introduces an adaptive operator selection technology for adaptive tuning for complex data processing applications, including the self-driving database. Beyond the scope of traditional database queries, it extends its application to areas such as convolutional neural networks (CNNs) and regular expression (RegEx) operators. However, its primary prowess becomes evident in query execution, leveraging intra-query learning. This strategy enables Cuttlefish to adapt query execution plans during the process of execution based on the feedback from ongoing executions. It employs a multi-armed bandit reinforcement learning technique to adaptively select the optimal physical operators for logical operations like join, creating a dynamic balance between exploration and exploitation. The multi-armed bandit algorithm probabilistically selects a candidate operator instance for each logical operation, based on its estimated performance advantage over others. This dynamic performance assessment system is updated during the query execution, using a contextual cost model that considers factors like data size, data distribution, and available resources. Cuttlefish demonstrates its effectiveness in an Apache Spark prototype, where it adaptively selected operators for tasks like image convolution, regular expression matching, and relational joins, displaying notable performance even under challenging conditions.

2.4.2 Adaptive Query Scheduling

Traditional scheduling methods rely on pre-defined heuristics manually, and such heuristics typically require manual tuning or configuration to optimize performance, which is time-consuming and labor-intensive. In addition, such traditional schedulers based on manually tuned heuristics may not be able to handle a wide range of workloads and data access patterns, limiting their use in dynamic environments. Adaptive scheduling technology uses machine learning algorithms to dynamically adapt to changing workloads and data access patterns, optimizing query execution and improving performance in real-world work environments. By automatically adjusting the order of query execution based on changing workload conditions, adaptive scheduling techniques can improve query execution performance and resource utilization and reduce the need for manual intervention, which is important for self-driving databases.

Introduced by Zhang et al. [17], the SmartQueue is an innovative query scheduler for self-driving databases that utilize deep reinforcement learning (DRL). The structure of SmartQueue system is displayed in Figure 2. The DRL agent at its core dynamically learns from past decisions and adapts to new data patterns, aiming to maximize buffer hit rates. SmartQueue generates specific strategies tailored to workload, focusing on long-term performance while adapting to new data access patterns. Although still in its prototype stage, SmartQueue has demonstrated significant performance improvements over simpler heuristics, showing promising potential for enhancing adaptive scheduling in self-driving databases.

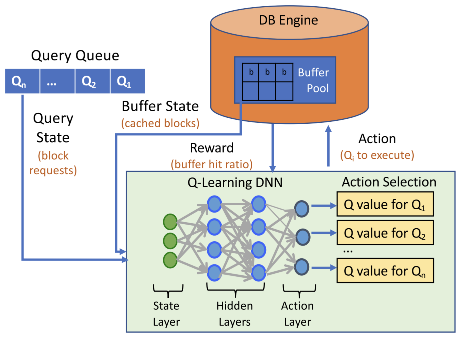


Figure 2: Syetem Structure of SmartQueue [17]

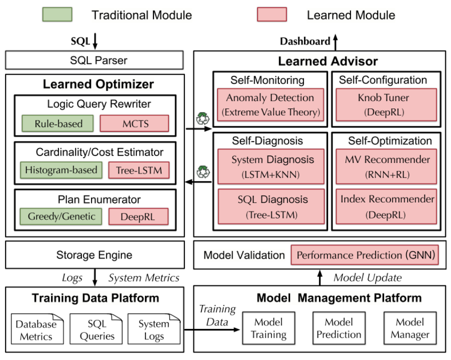


Figure 3: openGauss Architecture [3]

2.5 Integrated System

There have been several endeavors towards integrating all the automated components discussed above to construct a comprehensive self-driving DBMS.

2.5.1 Huawei's GaussDB

Huawei's GaussDB [3] stands as a pioneering example of a self-driving database, demonstrating the vast capabilities of incorporating machine learning into DBMS. Its architecture (Figure 3) integrates learned optimizers and advisors that enable the automation of tasks such as query rewriting, cost estimation, plan generation, knob tuning, database diagnosis, and index advising. The learned components use machine learning models and historical data to autonomously make informed decisions, thus significantly reducing the necessity for human intervention.

For example, the learned knob tuning advisor uses deep reinforcement learning to adjust the database engine settings based on historical performance data, optimizing the overall performance. The learned view/index advisor leverages an encoder-reducer model to recommend effective views and indexes for improved query performance. Through such proactive adaptation to workload changes, GaussDB exemplifies the benefits of applying AI and machine learning in database management, signaling an exciting potential for more autonomous, intelligent, and capable database management, which can contribute to the ideal realization of the self-driving database.

2.5.2 Oracle's Autonomous Database

Oracle claims that its Autonomous Database has reached a state where it can carry out provisioning, tuning, security, updates, backups, and other routine management tasks without human intervention [18]. Noteworthy features encompass automatic indexing, where beneficial indices are added and redundant ones removed; automatic query tuning optimization, where the optimizer dynamically adjusts a plan based on information gathered during execution to enhance query performance [19]; and automatic provisioning, where databases under load can be scaled with requisite compute and storage resources, thereby automatically configuring and tuning for specific workloads.

3 Comparison of Key Approaches

3.1 Configuration Tuning

Latency and throughput are important metrics to measure the performance of self-driving database configuration tuning tools since they intuitively reflect the response time for a single operation in the database and the number of operations that the database can process in a given timeframe.

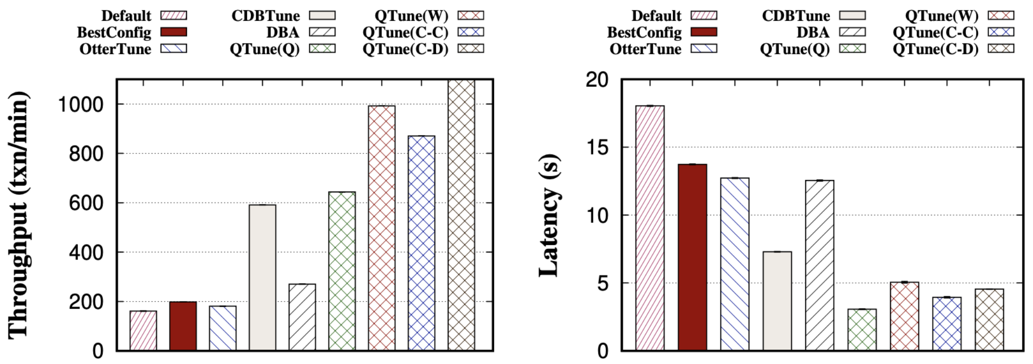


Figure 4: Performance of different tuning tools on MySQL database [7]

As illustrated in Figure 4, default refers to the default setting in the MySQL database which is the baseline of the performance. The BestConfig has higher throughput and also has higher latency compared with the OtterTune. Since resources used to explore optimal configurations are limited, BestConfig usually only gains the configuration with sub-optimal solutions [20]. In addition, the performance of OtterTune is worse than that of CDBTune and QTune in both latency and throughput. OtterTune utilized the Gaussian process regression to generate the best configuration of 'knobs' by mapping the current workload to the past experience' configuration. However, compared with the neural network used in CDBTune and QTune, the Gaussian process regression is not enough to learn more complex patterns of the 'knobs' configuration, such as the nonlinear correlations and dependencies between 'knobs' in the continuous space. In addition, the Gaussian process regression requires messy high-quality training data while the reinforcement learning used in the CDBTune and QTune doesn't since they can explore the new information to refine themselves.

In addition to OtterTune and CDBTune only focusing on tuning the configuration on workload level, the QTune also focuses on other two levels which are query level and cluster level. The query-level QTune, which is QTune(Q), achieved the lowest latency since it provides the best ‘knobs’ value for each query. However, since the query cannot be processed in parallel, the throughput of the query level QTune is a little lower than other levels’ QTune. By contrast, the cluster-level QTune with discrete tuner, which is QTune(C-D), achieved the highest throughput since it clusters the similar pattern’s query into the same group and trades off the tuning time and provides the best configuration for a group of queries.

3.2 Automated Indexing

Database cracking is an adaptive indexing technique designed to minimize index maintenance costs and efficiently adapt to changing query patterns, so that the indexing structure is suitable for the current workload. Learned index structure is a technique of using machine learning models to replace traditional index structures, in which the model can be used to predict the approximate location of a key, thus significantly reducing the search space to improve performance.

Both database cracking and learned index structures offer distinct benefits. Database cracking integrates smoothly with existing DBMS, dynamically updating the augmented traditional index structure alongside query optimization and execution, thus minimizing computational overhead. On the other hand, learned index structures propose a more innovative approach, potentially replacing traditional indexing altogether. Although they involve larger computational overhead, they significantly reduce memory usage.

Each technique provides adaptability to changing data or query patterns, but the way they achieve this differs. Database cracking directly responds to incoming queries, incrementally enhancing performance, whereas learned index structures employ a learning phase to understand data distribution, potentially offering high performance from the outset. Maintenance-wise, database cracking continuously adjusts the index during query processing, while learned index structures might require model retraining if data changes significantly.

In conclusion, the choice between database cracking and learned index structures is largely dictated by the specific requirements of the workload and data distribution. These distinct characteristics are summarized in Table 1.

Method	**Database Cracking**	**Learned Index Structures**
Integration	Easily be fused with traditional indexing structures	Require indexing structure adjustments to deploy trained models.
Overhead	Minimum overhead costs as the index are only updated and refined based on the actual queries being executed	Pre-train model required to initialize the indexing. On-going model adapting is needed to keep performance up.
Storage	Requires additional storage for generated indexing as per conventional indexing.	Provide compact representation of indexing structure, less storage usage.
Performance	Performance is best when the workload consists of on-going and steady shifting.	Performance best when workload access patterns are identifiable.

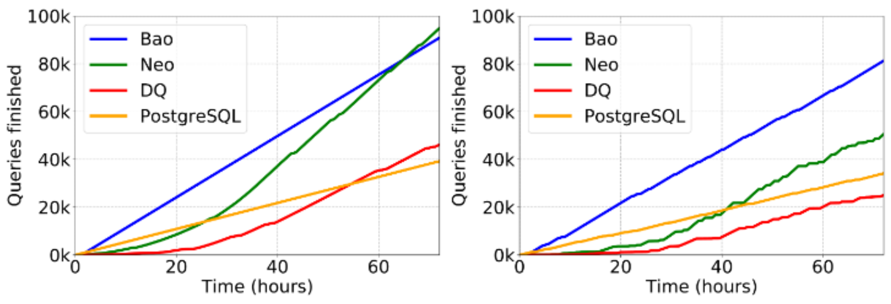
Table 1: Comparison of Database Cracking and Learned Index Structures

3.3 Query Plan Optimization

In the aspect of training time, a noteworthy reduction in training time was observed when comparing Bao and Neo. Neo demonstrated the capability to develop optimization strategies comparable to commercial systems, requiring 24 hours for training. However, Bao, grounded in conventional query optimizers, surpassed these systems within 1 hour of training.

In the aspect of adaptation, Neo utilizes tree convolutional neural networks to capture local features and then evaluate the cost associated with a specific query plan. The tree network allows it to effectively estimate the efficiency of various query plans, subsequently opting for the one with the least cost. Bao adopts a similar approach, using tree convolutional neural networks, but fortifies this network by applying Thompson sampling, a derivative of reinforcement learning. Incorporating Thompson sampling into hint sets significantly improves Bao's ability to discern the most cost-effective plan, enhancing Bao's adaptability to query workloads, data, and schema variations. The tree search algorithm, encapsulated in LearnedRewrite, facilitates the query selection, offering the greatest cost reduction as the outcome of the rewrite. Its estimation model accommodates varying operator numbers of the query by translating the operator features into the rules and metadata features matrix. Consequently, it provides relatively accurate benefit estimates that are dependent on different operators.

Combining adaptation and the query processing speed, we introduced DQ, an optimizer based on reinforcement learning, which primarily focuses on optimizing select-project-join blocks [21]. A comparative analysis was conducted between DQ, Neo, and Bao under both stable and dynamic query workloads (Figure 5). The results revealed that DQ surpassed PostgreSQL, the traditional heuristic optimizer that serves as a baseline, after 60 hours of learning from the stable query workload. However, DQ struggled to adapt to the dynamic query workload. Neo outperformed Bao after 60 hours of adapting to the stable query workload but failed to demonstrate adaptability and superior performance than Bao within 80 hours. Bao, built upon PostgreSQL, exhibited consistent and superior performance in both workloads, underscoring its significance. LearnedRewrite, although not presented in this figure, showed the latency reduction up to two folds of that of heuristic query rewrite such as PostgreSQL when operator number in the query reaches 40 [14]. It indicates that LearnedRewrite is suitable for complex queries with dozens of operators. It is hard to compare LearnedRewrite with Bao, but LearnedRewrite might outperform Neo and DQ according to the current data.



(a) Stable query workload **(b) Dynamic query workloads**

Figure 5: Performance comparison of the quantity of queries completed over 80 hours for Bao, Neo, DQ, and PostgreSQL for both a stable and dynamic query workloads [13]

In the aspect of tail latency, although Neo demonstrated that deep reinforcement learning could be feasibly applied to query latency, it did not exhibit improvements in tail performance. This might be due to its inherent nature of being built from the ground up. Specifically, it did not address how to handle the system under extreme conditions or during the longest latency periods, which could considerably affect the overall system performance and user experience. On the other hand, Bao, which has its foundation in traditional query optimizers, benefits from the intrinsic consideration for outlier data. Bao's capacity for learning from errors, coupled with the system's foundational benefits, enables it to enhance tail performance dramatically, necessitating a short training duration ranging from 30 minutes to a few hours. LearnedRewrite has shown a three-fold query rewrite tail latency as the heuristic PostgreSQL, but the longer rewrite latency can be compensated by the significant reduction in execution latency and overall latency.

3.4 Query Execution

Adaptive operator selection is a crucial part of achieving self-driving database in the query execution. SkinnerDB and Cuttlefish are notable approaches to adaptive operator selection, and each utilizes reinforcement learning to optimize query execution. SkinnerDB concentrates on optimizing join order, dynamically selecting the most promising sequence based on real-time progress assessments during execution. Conversely, Cuttlefish employs a multi-armed bandit algorithm to dynamically choose the optimal physical operator instances, leveraging feedback from past executions and predicting future operator performance.

The dynamic tuning approach of SkinnerDB for join order optimization during query execution is impressive. The system determines the optimal join order for each query on the fly, without relying on prior knowledge or assumptions. This feature can be particularly beneficial for complex joint operations and changing workloads. However, the approach may not be universally effective for diverse query types or workloads, and the computational burden of continuous learning during query execution could pose a constraint.

Cuttlefish's ability to adaptively select the most efficient physical operator during query execution allows it to handle complex data processing tasks beyond the limitations of traditional relational algebra. Nevertheless, the computational cost associated with exploring physical operator instances can be a drawback, especially for less complex or smaller queries.

By comparing the two systems, SkinnerDB and Cuttlefish illustrate the potential and complexity of adaptive operator selection in self-driving database performance enhancement. SkinnerDB's approach can be considered more specialized, focusing primarily on join order optimization, while Cuttlefish's strategy is broader, covering a range of operators beyond the traditional relational algebra. The effectiveness of these two systems may be determined by the workload properties and the specific features of the data, and the choice of system also depends on these factors.

4 Conclusions and Future Direction

In conclusion, the concept of self-driving databases has marked a major shift in the realm of DBMS, providing higher capability of handling tasks with little to no human intervention. This paper has presented an in-depth review of various approaches and techniques that contribute to the evolution of self-driving databases, including configuration tuning, data access, query plan optimization and query execution. Each aspect contributes significantly towards making databases more autonomous.

To automatically tune the configuration of DBMS, the tools like OtterTune, CDBTune and the QTune are introduced in the self-driving database and their performance in the aspects of low latency and high throughput are better than expert DBAs, especially the QTune. However, most of the current automatic configuration tool tunes the knobs based on the current or previous workflow, so the configuration may be sub-optimal for the near future workload. Therefore, for future improvement, it is critical to consider potential future workloads when auto-tuning database configuration.

The data access techniques, namely automated indexing and partitioning, showcased two different paradigms to efficiently handle data access and retrieval. Adaptive indexing involves adaptive sorting and indexing based on incoming queries, contrasts learned index structures that use machine learning models to predict data locations. Adaptive partitioning involved trained machine learning models to make predictions of the most effective partitioning schemes. Both automated data access techniques boosted the data storing and retrieving performance.

The query plan optimization techniques illuminate the potential utility of integrating machine learning algorithms to enhance cost estimation and to strike a balance between the exploration and exploitation of query plans. Besides, the importance of traditional query optimizers has been revealed in Bao, as a foundational element for the development of next-generation optimizers. Among the techniques discussed, query rewriting demonstrated significant potential in query optimization compared with heuristic query rewrite techniques used in PostgreSQL. In the future, these models can be further optimized by adapting them for cloud systems, exploring various machine learning or path searching algorithms, and integrating traditional optimization techniques with these algorithms.

The query execution for self-driving databases, specifically adaptive operator selection and scheduling, is continuously advancing with promising results. Adaptive operator selection, as exhibited by SkinnerDB and Cuttlefish, effectively optimizes query execution and dynamically responds to changing workloads. On the other hand, adaptive scheduling, exemplified by SmartQueue, shows potential in improving query schedules using buffer pool states and data access patterns, indicating opportunities in managing complex workloads and integrating with other autonomous database components.

It is evident that we have come a long way in the field of self-driving databases. Major technology companies such as Oracle, Amazon, and Microsoft have moved in this direction, offering ”automate” features to their cloud DBMS services. Their comprehensive cloud database management services now include various automated features to help reduce needs for human intervention and increase performance.

However, in the quest for fully self-driving databases, the current methods are still limited. Most of the previous studies are focused on only one or a few pieces of DBMS, which may result in reaching the local optimum for the individual self-driving component but not reaching the global optimal for the whole DBMS. To solve this problem, one potential solution is to build a self-driving database from the whole, incorporating machine learning into every aspect of the system like the NoisePage project led by Andy Pavlo [22]. It designs a new architecture and aim to control all aspects of the system by an integrated planning component rather than retrofitting existing systems with automated components. This uncaps the limitations of automated components which enables more effective coordination between different components and optimizes system-wide rather than local performance.

References

[1] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang, “Self-driving database management systems,” in CIDR 2017, Conference on Innovative Data Systems Research, 2017.

[2] A. Pavlo, “What is a self-driving database management system?.” https://www.cs.cmu.edu/ pavlo/blog/2018/04/what-is-a-self-driving-database-management-system.html.

[3] G. Li, X. Zhou, J. Sun, X. Yu, Y. Han, L. Jin, W. Li, T. Wang, and S. Li, “opengauss: An autonomous database system,” Proceedings of the VLDB Endowment, vol. 14, no. 12, pp. 3028–3042, 2021.

[4] G. Li, X. Zhou, and L. Cao, “Ai meets database: Ai4db and db4ai,” in Proceedings of the 2021 International Conference on Management of Data, pp. 2859–2866, 2021.

[5] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, “Automatic database management system tuning through large-scale machine learning,” in Proceedings of the 2017 ACM international conference on management of data, pp. 1009–1024, 2017.

[6] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, et al., “An end-to-end automatic cloud database tuning system using deep reinforcement learning,” in Proceedings of the 2019 International Conference on Management of Data, pp. 415–432, 2019.

[7] G. Li, X. Zhou, S. Li, and B. Gao, "Qtune: A query-aware database tuning system with deep reinforcement learning," Proceedings of the VLDB Endowment, vol. 12, no. 12, pp. 2118–2130, 2019.

[8] S. Idreos and T. Kraska, "From auto-tuning one size fits all to self-designed and learned data-intensive systems," in Proceedings of the 2019 International Conference on Management of Data, pp. 2054–2059, 2019.

[9] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in Proceedings of the 2018 international conference on management of data, pp. 489–504, 2018.

[10] S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, and D. Guo, "The data calculator: Data structure design and cost synthesis from first principles and learned cost models," in Proceedings of the 2018 International Conference on Management of Data, pp. 535–550, 2018.

[11] S. Das, M. Grbic, I. Ilic, I. Jovandic, A. Jovanovic, V. R. Narasayya, M. Radulovic, M. Stikic, G. Xu, and S. Chaudhuri, "Automatically indexing millions of databases in microsoft azure sql database," in Proceedings of the 2019 International Conference on Management of Data, pp. 666–679, 2019.

[12] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A learned query optimizer," arXiv preprint arXiv:1904.03711, 2019.

[13] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," in Proceedings of the 2021 International Conference on Management of Data, pp. 1275–1288, 2021.

[14] X. Zhou, G. Li, C. Chai, and J. Feng, "A learned query rewrite system using monte carlo tree search," Proceedings of the VLDB Endowment, vol. 15, no. 1, pp. 46–58, 2021.

[15] I. Trummer, J. Wang, Z. Wei, D. Maram, S. Moseley, S. Jo, J. Antonakakis, and A. Rayabhari, "Skinnerdb: Regret-bounded query evaluation via reinforcement learning," ACM Transactions on Database Systems (TODS), vol. 46, no. 3, pp. 1–45, 2021.

[16] T. Kaftan, M. Balazinska, A. Cheung, and J. Gehrke, "Cuttlefish: A lightweight primitive for adaptive query processing," arXiv preprint arXiv:1802.09180, 2018.

[17] C. Zhang, R. Marcus, A. Kleiman, and O. Papaemmanouil, "Buffer pool aware query scheduling via deep reinforcement learning," arXiv preprint arXiv:2007.10568, 2020.

[18] Oracle, "What is an autonomous database?." https://www.oracle.com/au/autonomous-database/what-is-autonomous-database/#components-of-an-autonomous-database.

[19] Oracle, "Database sql tuning guide." https://docs.oracle.com/en/.

[20] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: tapping the performance potential of systems via automatic configuration tuning," in Proceedings of the 2017 Symposium on Cloud Computing, pp. 338–350, 2017.

[21] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica, "Learning to optimize join queries with deep reinforcement learning," arXiv preprint arXiv:1808.03196, 2018.

[22] A. Pavlo, "Noisepage self-driving database management system." https://noise.page/.