Adaptive Query Processing

Liumingzi Zhu 727740
Xiaolu Zhang 886161
Luyao Chen 1266572
Muhan Guan 1407870

Abstract

In context of data processing, due to a number of reasons, such as increasingly complex datasets, distributed database systems and scarce resource availability .etc, static assumptions in query processing often result in a sub-optimal performance. As a result, Adaptive Query Processing (AQP) has emerged as one of the most important solutions for handling unpredictability and dynamics of query processing where runtime feedback is considered in query execution and query process will be modified based on feedback. In this survey, we explore a range of AQP techniques and categorize them into four main categories: plan-based techniques, routing-based techniques, continuous-query-based and reinforcement-learning-based techniques. Not only advantages and disadvantages of each technique are examined in this paper, but we also do a vertical comparison within each category based on their effectiveness and suitability in modern data management systems. Furthermore, we analyze the practical use case and application of some modern AQP techniques. Finally, some key challenges and future research directions are discussed, including the scalability, standardized benchmark, reliability of query processing, capability to support multi-modal data source and potential integration with machine learning and artificial intelligence.

1 Introduction

1.1 Traditional Query Processing

Back in 1970s, an optimize-then-execute model was proposed by Selinger et al.[1] in which query processing was divided into four phases: parsing, optimization, code generation and execution. After syntax is checked, the optimizer comes to place which evaluates the cost of different query execution plans. Cost estimation is the core of the optimizer, it is the estimation of query running time based on the statistics of the dataset, estimation of the algorithm performance and related hardware speed. The optimizer will enumerates all possible plans and choose the one with the lowest estimated cost.

This plan-first-execute-next method is still followed by most of the database systems today. However, optimizers, which are critically dependent in this method, do not always produce optimal results, especially when queries become more and more complex today. This is where *Adaptive Query Processing* (AQP) comes into stage as a useful approach to reduce the chance of optimizer failure.

1.2 Motivations for AQP

As mentioned above, database users are experiencing larger and larger datasets and more and more complex queries, which leads to the fact that sub-optimal query plans can be chosen by the query optimizer. There are several factors that can cause query optimizer to produce undesired result:

Unreliable Cardinality Estimation. According to Stillger et al.[2], query optimizer estimates cost and cardinality based on several assumptions. For example, the statistics of the database is assumed to be known and relatively stable. However, in reality, statistics of a database may be hard to obtain and dynamically changing overtime. Additionally, even when the statistics of a database is perfectly predictable, the dependency between predicates can lead to inaccurate cardinality estimation. With these assumptions invalid, significant errors can happen to the optimizer when estimating the cost of query plans and hence lead to a sub-optimal plan being chosen.

Changing Data and System Resources. In some data systems, particularly data stream management systems, queries can be continuous and long-running[3]. As a result, the characteristics of the dataset and the remaining cost of the query can change substantially which leads to the fact that optimal query plan is different from what has been chosen at the start of the query. Moreover, in most scenarios, computational resources such as memory, are limited for query processing, and due to the long-running nature of some queries, the available computational resource can be changing too. Therefore, query execution needs to be adaptive to external environment as well.

Error Propagation in Complex Queries. Ioannidis and Christodoulakis[4] mentioned that when dealing with complex queries where multiple joins are involved, the query optimizer typically needs to estimate various parameters of the intermediate results of the operation, where dynamic programming approach is often used. However, it sometimes leads to error propagation, which means that a small error of the estimation in the previous steps can lead to devastating error of the final result because of the transitivity effect.

Queries in Online Environment. In recently years, internet and web-based query systems has become more and more popular. Two of the main characteristics of online queries are interactive and distributed. Interactive means that the query response should be almost immediate and reasonably accurate but in the meantime, query parameters can be changed very frequently[5]. The traditional query processing method is not always able to generate response very quickly and the pre-calculated query cost estimation of a traditional query optimizer can be significantly far from optimal. The distributed feature suggests that the query optimizer will not have meaningful database statistics such as histograms. Besides, the I/O cost of the query operations is high, unpredictable and variable[6].

1.3 Brief Introduction to AQP

To overcome the above-mentioned challenges of traditional query processing, AQP is introduced. AQP aims to use the feedback from intermediate query steps to adjust the execution plan for the following query steps and subsequently, mitigate the query optimizer mistakes such as missing database statistics, incorrect query cost estimation and unstable database system conditions etc.

Numerous researches has been published in regard of AQP, within which multiple techniques have been proposed against AQP. In this paper, we categorize them into four groups. The first two, namely plan-based techniques and routing-based techniques, are traditional AQP methods, with researches mainly prior to 2005. The latter two components, continuous-query-based and reinforcement-learning-based techniques, are a summary of researches published within the past two decades which includes a couple of cutting-edge AQP techniques.

2 Related Works

In Babu and Bizarro's review [7], AQP techniques can be partitioned into three categories: plan-based, routing-based, and continuous-query-based. In recent years, with the emergence of machine learning and artificial intelligence, reinforcement learning methods are also widely used in AQP. Therefore, we treat reinforcement-learning-based techniques as the fourth category.

2.1 Plan-based Systems

Unlike traditional query processing systems which rely on fixed assumptions made before execution, plan-based systems closely monitor the execution process. As runtime conditions such as data distribution, cardinality, and resource availability deviate significantly from the assumptions, the system will dynamically adjust query plans during execution to improve performance[7]. According to the taxonomy of plan-based AQP systems of Schmidt et al.[8], we classify them into two categories based on implementing styles: mid-query re-optimization and parametric query optimization. We also include a special type of plan-based techniques which utilize competitions between query plans. Table 1 summarises the advantages and disadvantages of different plan-based AQP approaches.

Mid-Query Re-Optimization. Re-optimizing systems interleave execution and optimization during runtime. This concept was first introduced in Re-Opt by Kabra and DeWitt[9]. They insert statistics collectors at specific checkpoints during plan execution to observe intermediate result sizes and execution costs. If the actual statistics observed notably differs from the initial estimate, the system triggers re-optimization for the remainder of the query and switches to a new plan based on the updated statistics. One key challenge in re-optimizing systems is to decide when to materialize the intermediate results and invoke the optimizer. Re-opt[9] materialize results only at pipeline breakers, such as hash join and sort operations, which results in infrequent re-optimizations. POP[10] improves on this by applying forced materialization, aggressively materializing outputs at every nested-loop join. While this provides more opportunities for re-optimization, it introduces significant performance overhead and increased space consumption. To address this issue, QuerySplit[11] offers a more refined solution. Instead of relying on potentially misleading global execution plan, QuerySplit generates subqueries directly from the logical plan and decide proactively when to materialize results before execution. This method provides more precise control over re-optimization.

Parametric Query Optimization (PQO). Parametric systems find a small set of plans that are appropriate for different situations before execution, but postpones certain planning decisions to runtime. During execution, the system can switch between these pre-optimized plans using choose-plan operators based on observed runtime statistics[12]. The Progressive Parametric Query Optimization (PPQO) system[13], further advanced PQO by proposing a specialized interface known as the Parametric Plan. This data structure stores previously optimized plans, allowing the system to avoid redundant optimization processes when similar query conditions arise. By reusing optimal plans from past queries, PPQO significantly reduces the computation overhead associated with real-time re-optimization. However, PQO systems suffer from expensive analysis when identifying sets of optimal candidate plans, as the search space expands exponentially with the complexity of queries.

Competition-based Systems. Another special type of plan-based AQP systems is DEC-Rdb[14] which was originally used in the relational database management system, Oracle Rdb. DEC-Rdb was designed to handle dynamic query conditions by offering multiple competing plan choices during query execution. It runs multiple plans simultaneously and picks the most promising one after a short time. Although this approach helps reduce the performance gap caused by suboptimal initial plans, it only makes one decision per table in a query and does not explore combinations of access methods. Also, it leads to significant overhead due to duplicate elimination[15].

Approach	System	Pros	Cons
Competition	DEC Rdb[14]	(+) High statistics prediction accuracy	(-) High computational overhead due to redundant processing of data (-) Only limited competing plans can be run
Mid-Query Optimization	Re-Opt[9]	(+) High statistics prediction accuracy (+) Low performance overhead	(-) Hard to determine when and how to re-optimize (-) Infrequent re-optimizations
	POP[10]	(+) More re-optimization opportunities	(-) Increased performance overhead (-) Increased space consumption
	QuerySplit[11]	(+) Proactively decide when to re-optimize (+) Near-optimal execution time	(-) Might be myopic as optimizer only operates at the subquery level
Parametric Query Optimization	PQO[12]	(+) Efficient reuse of plans	(-) High start-up cost (-) Low scalability
	PPQO[13]	(+) Lower start-up cost (+) Progressively construct information about the parametric space	(-) Increased space consumption to store previous plans

Table 1: Comparison of plan-based systems

2.2 Routing-based Systems

Compared to plan-based systems, one distinctive feature of routing-based systems is that they eliminate query plans and dynamically optimize the query execution path through a routing mechanism (i.e., tuple router) in real time. Specifically, routing-based techniques aim to ensure that most tuples follow the most efficient route during runtime, while a smaller portion explore alternative routes.

Eddy[16] was first introduced as a highly adaptive technique that continuously reoptimize a query based on changing runtime conditions. It achieves adaptivity by changing the order in which tuples are processed through tuple routing. However, despite its adaptive capabilities, Eddy faces certain limitations. Early routing decisions can constrain its flexibility later in execution, reducing the potential for optimization. Moreover, the uncertainties inherent in runtime conditions, such as variable data arrival rates and incomplete information, can result in suboptimal routing choices, further diminishing its effectiveness in certain scenarios.

An extension of the Eddy architecture, SteMs[17], was then proposed to mitigate the burden of historical operation by only storing tuples at endpoints and discarding intermediate tuples. This design prevents the accumulation of intermediate tuples, meaning that the state inside the SteMs is unaffected by previous routing decisions. While this approach addresses some of the challenges associated with Eddy, it also introduces two potential limitations:

Re-computation of Intermediate Tuples. Since intermediate tuples generated during query execution are not stored for future use, they must be recomputed every time they are needed, leading to inefficiencies.

Constrained Plan Choices. The range of query plans that can be executed for any new tuple becomes limited, even if the Eddy is aware that the current plan is suboptimal. This reduces the overall flexibility of the system, resulting in potential further performance degradation in certain cases.

Following SteMs, another extension of the Eddy architecture is STAIRs[18]. Unlike SteMs, which only stores base tuples, STAIRs chooses to store some intermediate result tuples for future reuse. This design adds flexibility, particularly for executing sliding window queries over streaming data, as it allows the system to efficiently manage and reuse intermediate results instead of recomputing them.

To enhance query execution efficiency, recent approaches attempt to combine routing-based techniques with vectorized database systems and compiling database systems.

Optimization of Compiled Database Systems. The combination of routing-based techniques with compiling database systems improves query execution by reducing interpretation overhead. By partially evaluating query plans into compiled executables, these systems can execute queries more efficiently. For instance, one of the latest works is Dynamic Blocks[8] which extends routing-based techniques to compiling database systems by generating dynamic code fragments at runtime, eliminating the need for recompilation and allowing for both high-level plan adjustments and low-level optimizations.

Optimization of Vectorized Database Systems. In addition to Dynamic Blocks, another representative framework is Micro Adaptivity[19], which demonstrates practical implementation of routing-based techniques on vector-wise system. Micro Adaptivity employs a vw-greedy algorithm to dynamically select the most efficient function implementation based on observed execution costs. By framing the runtime selection of the optimal implementation as a multi-armed bandit problem, this approach ensures performance robustness and reduces development time.

Some recent studies have explored the practical use of routing-based methods combining with other modern database technologies, such as machine learning query optimization and distributed setting. The specifics of these practical applications will be discussed in detail in Section 3.6.

System	Pros	Cons
Eddy[16]	(+) Adaptive query execution (+) Dynamic operator ordering (+) Optimizes at runtime	(-) Sub-optimal routing decisions under constraints on routing history
SteMs [17]	(+) Independence on intermediate state (+) Improved join performance	(-) Overhead on re-computation of intermediate tuples (-) Constrained plan choices
STAIRs[18]	(+) Flexible storage and reuse of intermediate results	(-) Complex implementation

Table 2: Comparison of Routing-based systems

2.3 Continuous-Query-based Systems

Continuous queries are similar to conventional queries but can run continuously over the database[20]. Due to this characteristic, the continuous-query-based(CQ-based) systems care more about runtime change and system conditions over cardinality estimation errors[11].

CQ-based systems can be divided into 2 subcategories: pure CQ-based and CQ-based with Eddy. The key difference is the latter employs Eddy operators for dynamic routing, making it highly responsive to data changes and system conditions. The differences between these CQ-based methods can be found in Table 3.

CAPE, NiagaraCQ, and StreaMon are pure CQ-based systems [7]. CAPE[21] dynamically chooses optimal plans in real time by making use of computational resources and data statistics. NiagaraCQ[22] leverages incremental group optimization and query-sharing techniques. StreaMon[23] achieves adaptive join processing by using Adaptive-Greedy and Adaptive-Caching algorithms.

In the remainder of this section, we illustrate key techniques that focus on adaptive systems and stream processing. All of them were introduced in UC Berkeley's Telegraph Project and also make use of Eddy operators. Continuously Adaptive Continuous Queries(or CACQ) proposed in [24] has four main contributions: dynamic operation reorder via the eddy operator, a fine-grained sharing of computation across multiple queries, grouped-filter indexing, and State Modules utilization. PSoup extended CACQ further by not only "filter" operators but also data, which leads to more flexible query handling. For example, when users need to analyze historical data but also handle continuous streams, PSoup is a good candidate as it treats continuous queries and data as two complementary components[25]. A key feature of these two systems is to take advantage of query commonality [26]. TelegraphCQ was built as a system to process continuous queries over data streams. It is on eddy operator, operator scheduling, and fjords (used for inter-operator communication)[27].

Approach	System	Pros	Cons
CQ	CAPE [21]	(+) Real-time adaptability (+) Efficient resource usage	(-) Higher complexity (-) Not for static data
	NiagaraCQ [22]	(+) High scalability (+) Efficient resource usage	(-) High complexity
	StreaMon [23]	(+) Low-latency	(-) Not for fine-grained adaptivity
CQ Eddy	CACQ [24]	(+) Aggressive computation and operator sharing (+) Fine-grained adaptivity	(-) High complexity (-) High storage overhead
	PSoup [25]	(+) Support historical data (+) Low-memory overhead	(-) Low scalability (-) Not for high-throughput streaming data
	TelegraphCQ [27]	(+) Dynamic operator ordering (+) Fine-grained adaptivity	(-) High memory overhead

Table 3: Comparison of CQ-based systems

2.4 Reinforcement-Learning-based Systems

Tzoumas et al.[28] provided a mathematical framework to model query execution with eddies as a Reinforcement Learning(RL) problem. Empirical studies demonstrate that these new routing policies exhibit the desired adaptivity and asymptotic convergence characteristics as the problem environment stabilizes, and that these novel Eddy routing policies outperform existing ones.

Building on the previous work of Tzoumas et al.[28], SkinnerDB[29] extends the RL framework for query optimization by incorporating regret-bounded query evaluation. In SkinnerDB, the RL problem is similarly modeled, but with enhanced guarantees on the relationship between the expected execution time and the optimal query plan. Unlike the original Eddy-based approaches, which do not formally discard costly intermediate results during query execution, SkinnerDB applies reinforcement learning to actively minimize execution overhead caused by suboptimal intermediate joins. Furthermore, empirical evaluations demonstrate that SkinnerDB reliably identifies near-optimal left-deep plans, offering robust performance compared to traditional bushy plan strategies used in RL with Eddies[28]. This ensures that SkinnerDB not only converges to optimal join orders, but also significantly reduces the inefficiencies presented in prior adaptive query processing systems.

Recently, SkinnerMT[30] was proposed as an improved fork of SkinnerDB to address the limitations of sequential join processing by introducing parallelization strategies. While SkinnerDB executes joins sequentially, leading to bottlenecks in multi-core environments, SkinnerMT enhances the system by exploiting multi-threading to improve performance and scalability. Specifically, SkinnerMT parallelizes query execution in two ways: first, by exploring multiple join orders simultaneously, and second, by processing the same join order across multiple threads. Moreover, it introduces a hybrid strategy that combines both parallel search and parallel processing, optimizing query execution on multicore platforms. These improvements enable SkinnerMT to overcome the execution inefficiencies inherent in the original SkinnerDB, making it more suitable for high-performance adaptive query processing.

System	Pros	Cons
RL with Eddies[28]	(+) Flexibility	(-) Single-threaded solution (-) Less powerful routing policies (-) Not for content-based routing
SkinnerDB[29]	(+) Minimize execution overhead (+) Converge to optimal join order	(-) Sequential joins (-) Limited performance (-) Learning and join order switching overheads (-) Not for nested queries
SkinnerMT[30]	(+) Parallelism from parallel join order search to parallel execution (+) Efficient memory usage (+) Multi-threading	(-) No fault tolerance (-) Less scalability
RouLette[31]	(+) High scalability (+) Scalable multi-query execution (+) Improved hardware utilization (+) Employ eddy and learned policy	(-) Not for non SPJ sub-queries
Cuttlefish[32]	(+) Lightweight (+) Support distributed systems (+) Adapt to workloads (+) Support non-relational operators	(-) Restricted scenario (-) Communication and synchronization overhead

Table 4: Comparison of RL-based systems

As the complexity and diversity of operators in query plans grow, a lightweight model Cuttlefish[32] was proposed, employing multi-armed bandit algorithm to choose the fastest physical operator. Unlike traditional AQP approaches, which require system developers to have in-depth knowledge of the underlying operators and design explicit rules and cost models, Cuttlefish automatically selects between various operator implementations at runtime without the need for explicit optimization rules.

RouLette[31] was developed as a specialized query execution engine to optimize the execution among multiple Select-Project-Join (SPJ) queries by leveraging Q-learning, a model-free RL technique. It takes advantages of symmetric hash joins, eddies, and state modules to achieve operator-level adaptation. Besides, it is designed to suit large workload use cases by applying symmetric join pruning, adaptive projections, range-based grouped filters and locality-conscious routers.

The differences between these RL-based methods can be found in Table 4.

3 Comparison of Key Approaches

In this section, we will present a comparison of different AQP approaches. The section will be structured as follows: we will first conduct a horizontal comparison between the above mentioned general categories of AQP methods. Then an in-depth vertical comparison will be done within each AQP category where various specific AQP techniques will be compared.

3.1 Overview

In terms of intra-query adaptivity, the performance of a query depends on the following aspects: how to periodically or continuously monitor and analyze database statistics, when to re-optimize the query and how to re-optimize the query[15].

In regard of monitoring database statistics, plan-based and pure CQ-based approaches usually perform a prediction of statistics at pipeline stage and some techniques may include algorithms to collect statistics during query processing. On the other hand, routing and RL-based approaches explore alternative paths during execution so that information can be gathered during this time as well. After statistics assumptions are made, plan-based and pure CQ-based approaches will have their cost modeler in query optimizer to estimate the cost of current plan and alternate plans while most routing approaches use greedy algorithms to achieve local optimal plan. With RL-based approach, reinforcement learning strategy is used to analyze the current state and other possible plans. In conclusion, at monitoring and analyzing stage, compared to plan-based approaches, routing-based and RL-based approaches may explore more plans, and use heuristics to find local optimal (not necessarily global optimal) plans which is more suitable to small and simple datasets.

Having the statistics and analysis ready, it's still not straightforward to determine when and how to re-optimize the query for plan-based and pure CQ-based approaches. They need to compare the cost of current plan with alternate plans, taken into consideration of statistic changes, environment changes and possible sunk cost of discarding current execution. On the contrary, re-optimization happens automatically in routing-based approaches as they use the new information to route tuples directly.

3.2 Plan-based Systems

In Table 5, we focus on re-optimization frequency, performance overhead and scalability while comparing different plan-based AQP systems.

System	Re-Optimisation Frequency	Performance Overhead	Scalability
DEC Rdb[14]	Limited dynamic adjustments	High overhead caused by running multiple plans simultaneously and duplicate elimination processes	Less scalable for large queries
Re-Opt[9]	Infrequent re-optimization occurs only at pipeline breakers	Low overhead due to infrequent re-optimizations	Suitable for smaller queries or when deviations are minimal
POP[10]	Frequent re-optimizations due to forced materialization	High memory usage and processing overhead	Low scalability due to excessive memory and computational requirements
QuerySplit[11]	Proactive re-optimizations	Offer more opportunities for re-optimization without the aggressive overhead of POP	More adaptable to larger queries
PQO[12]	Pre-optimize a set of plans and switches at runtime with few real-time re-optimizations	Low overhead during execution but requires extensive planning before execution	Low scalability due to high start-up costs and search space expansion
PPQO[13]	Reuse pre-optimized plans based on past executions, reducing the need for new optimizations	Efficient reuse of past plans minimizes overhead during execution but still requires significant space to store plans	More scalable than PQO due to plan reuse, but still faces space and complexity limitations

Table 5: Deeper comparison of plan-based systems

Competition based systems like DEC Rdb support adaptivity at an intra-operator frequency, but they incur high computational costs by running multiple plans concurrently, making them unsuitable for complex queries. Re-optimizing systems such as Re-Opt, POP, and QuerySplit adapt at an inter-operator frequency, with key differences in memory usage due to their varying materialization strategies. For instance, Re-Opt compiles new plans at checkpoint operators, while POP enforces additional materialization at every nested-loop join. QuerySplit, on the other hand, takes a different approach by proactively re-optimizing based on the logical query plan, allowing it to quickly converge on an optimal solution. This results in better performance with moderate memory usage compared to Re-Opt and POP, which are constrained by their reliance on the initial physical plan during re-optimization. Lastly, parametric optimization approaches like PQO and PPQO, which pre-compute multiple plans for different parameter values, face scalability challenges due to high start-up costs as the number of parameters increases.

3.3 Routing-based Systems

In Table 6, three key techniques, Eddy, SteMs, and STAIRs, are evaluated based on their execution time, state handling, and adaptivity. Eddy demonstrates fast execution with dynamic tuple routing but suffers from state accumulation, which can limit its long-term adaptivity. SteMs, despite slower, avoids storing intermediate results but offers moderate adaptivity focused on managing base relation states. STAIRs excels in scenarios with natural data partitioning, providing faster execution by allowing state migration and correction of earlier routing mistakes, thus achieving a higher level of adaptivity.

System	Execution Time	State Handling	Adaptivity
Eddy[16]	Fast	Accumulates state in operators, limiting future adaptation	Moderate, can be constrained by accumulated state
SteMs[17]	Slow	No storage of intermediate results	Moderate, focuses on base relation states, with less fine-grained control compared to Eddy and STAIRs
STAIRs[18]	Faster when data has natural horizontal partitioning	Storing long-living intermediate tuples, improving flexibility	Higher, allows correction of earlier routing decisions

Table 6: Deeper comparison of Routing-based systems

3.4 Continuous-Query-based Systems

The following section will focus on the in-depth comparison between memory usage, query plan optimization, and join optimization. The details can be found in Table 7.

System	Resource Usage	Query Plan Optimization	Join Optimization
CAPE[21]	Efficient memory usage by shrinking join states	- Periodic plan optimization - Event-driven plan optimization	Component-based adaptive join algorithms by mixing punctuation, sliding window and invalidation operations
NiagaraCQ[22]	Cache group query plans and recently accessed files	Incremental group optimization and dynamic re-grouping	Use join signatures to share join execution across multiple queries
StreaMon[23]	- Reduce memory usage by analyzing arrival patterns - Avoid recomputing results via Adaptive-Caching	Start with a straightforward initial query plan and continues to adapt	Adaptive-Greedy dynamically (re)order join orders for MJoins
CACQ[24]	Use grouped filter to reduce computation	Eddy-based dynamic routing	Uses SteMs to enable pipelined multiway joins
PSoup[25]	Efficient incremental updates and state management	Eddy-based dynamic routing	Uses adaptive N-relation symmetric joins based on Eddy and SteMs
TelegraphCQ[27]	Use shared memory infrastructure	- Eddy-based dynamic routing - Use fjords for inter-operator communication	Involve joins over multiple streams and joins over streams and tables

Table 7: Deeper comparison of CQ-based systems

Performance study[24] shows that while NiagaraCQ works well with overlapped queries, its performance drops with multiple complex queries, especially in user-defined function (UDF) scenarios. Instead, CACQ consistently performed better than NiagaraCQ in the experiments. The root cause is that CACQ avoids expensive operations by applying selections or UDFs before joins. However, NiagaraCQ relies on static query optimization.

Overall, CAPE, NiagaraCQ, and CACQ are less suitable for use as lightweight plug-in solutions for existing streaming data systems. While none of the systems have a clear commercialization plan, TelegraphCQ is likely to be a commercial candidate as it is built on PostgreSQL.

3.5 Reinforcement-Learning-based Systems

The following section will focus on the in-depth comparison between memory usage, query plan optimization, and join optimization. The details can be found in Table 8.

System	Memory Usage	Query Plan Optimization	Join Optimization
RL with Eddies[28]	Introduce Q values to only store meta-data of operators	Query execution is split into: update and improvement phase	Utilize binary SHJs, STAIRs and SteMs
SkinnerDB[29]	- Keep UCT search tree - Keep last execution state - Keep a vector of join result tuples - No any intermediate results	- Introduce a new quality criterion for query evaluation strategies - Propose several adaptive execution strategies based on reinforcement learning	- Divide join execution into small time slices - Use a hybrid algorithm, including reinforcement learning and a traditional query optimizer
SkinnerMT[30]	Space-efficient data structure storing execution states for join orders	Support parallel multiple query plan execution	- Employ multi-way join algorithm - Join order is selected via reinforcement learning - Explore join orders in parallel with multi-threading
RouLette[31]	Use symmetric join pruning and range-based grouped filters	- Online and offline work-sharing - Learned cardinality estimation - Use reinforcement learning to refine ordering and to increase sharing benefits	- Use symmetric hash joins, eddies, and SteMs - Employ multi-way join algorithm
Cuttlefish[32]	Use a constant-memory for each tuner	Automatically adapt a physical query plan based on workload characteristics	Specific tuner can apply join strategy per data partition

Table 8: Deeper comparison of RL-based systems

Overall, RL with Eddies, RouLette, and Cuttlefish have more restrictions than other systems. And they are less likely to be developed as industrial and commercial DB products. SkinnerDB and SkinnerMT are not designed to support distributed systems.

3.6 Practical Use

This section will discuss the how these AQP methods mentioned above solve real world problems. Plan-based systems, being relatively easy to implement, have been used in Apache Spark to optimize the query execution process. They were also proven effective in both SQL Server and PostgreSQL for enhancing query performance. In particular, DEC-Rdb is utilized in Oracle Rdb, which supports demanding database applications and operates across multiple platforms and various environments. Moreover, CQ-based systems are designed for tasks like network monitoring and sensor networks. For example, NiagaraCQ is particularly powerful for web content monitoring and financial applications, such as stock price alerts. Reinforcement learning-based techniques, such as SkinnerDB and SkinnerMT are well-suited for relational database management systems. Cuttlefish has a lightweight and flexible API, while RouLette excels in high throughput and scalable analytical systems.

Compared with other approaches, routing-based techniques have more applications. Particularly, such techniques have been effectively integrated into modern database systems to enhance query execution performance and adaptability. We identify two interesting areas where routing-based techniques has been applied: optimization of machine learning (ML) queries, and extension to distributed environments.

Optimization of ML Queries. In the work by Kakkar et al.[33], the query optimizer leverages the Eddy framework to efficiently handle ML-centric predicates during query execution in video database system (VDBS). By integrating routing-based techniques, the system adaptively selects the most suitable execution path for queries that incorporate complex ML models, enhancing overall performance and resource utilization.

Extension to Distributed Environments. The application of routing-based technique has expanded into distributed settings, accommodating the complexity of data distribution and network communications. The study by Gounaris et al.[34] introduces modifications to the Eddy architecture to suit distributed systems. In this distributed Eddy framework, operators learn statistics during execution and periodically exchange this information with each other - either after processing a certain amount of data or after specific time intervals. Unlike traditional Eddies where routing decisions might occur continuously for individual tuples, the distributed framework applies routing decisions to blocks of tuples. This batching strategy minimizes the overhead associated with routing in a distributed context. By adapting routing decisions at the granularity of tuple blocks, the system maintains efficient execution while effectively managing the additional communication and synchronization required in a distributed environment.

4 Conclusion and Future Direction

In conclusion, this paper presents various adaptive query processing techniques and classifies them into four main categories. Plan-based systems re-optimize query plans at well-defined points during execution, while routing-based systems treat query processing as the routing of tuples through operators, dynamically adjusting the routing order. CQ-based techniques support AQP to continuous queries and are mainly used for stream data. Recently, with rapid advancements in machine learning and artificial intelligence, new approaches integrate these technologies to enhance traditional AQP systems.

Looking ahead, improving the **scalability** of AQP systems will be crucial. As query complexity grows, there's a need to balance optimization granularity and runtime overhead they introduce. Additionally, future research can explore how to adapt AQP techniques to **distributed environments**, ensuring both high effectiveness and efficiency. We also identify the lack of **standard benchmarks** for evaluating AQP systems. Developing a standardized framework will be essential for consistent performance comparisons. Moreover, **disaster recovery and fault tolerance** are often overlooked in current researches. Future studies should consider security and reliability aspects while designing AQP systems. Besides, adaptive query processing can be extended to support **multi-modal databases** that handle diverse data types (e.g., relational, graph, document) by tailoring query processing strategies to each specific data model. Finally, the **evolution of machine learning and artificial intelligence** is likely to drive the development of more hybrid, intelligent query processing systems. In the future, we anticipate the rise of smarter, more scalable, and resilient AQP systems.

References

[1] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, “Access path selection in a relational database management system,” in *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pp. 23–34, 1979.

[2] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil, “Leo-db2’s learning optimizer,” in *VLDB*, vol. 1, pp. 19–28, 2001.

[3] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma, “Query processing, resource management, and approximation in a data stream management system,” in *CIDR 2003*, Stanford InfoLab, 2002.

[4] Y. E. Ioannidis and S. Christodoulakis, “On the propagation of errors in the size of join results,” in *Proceedings of the 1991 ACM SIGMOD International Conference on Management of data*, pp. 268–277, 1991.

[5] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas, “Interactive data analysis: The control project,” *Computer*, vol. 32, no. 8, pp. 51–59, 1999.

[6] Z. G. Ives, A. Y. Levy, D. S. Weld, D. Florescu, and M. Friedman, “Adaptive query processing for internet applications,” 2000.

[7] S. Babu and P. Bizarro, “Adaptive query processing in the looking glass,” in *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR), Jan. 2005*, 2005.

[8] T. Schmidt, P. Fent, and T. Neumann, “Efficiently compiling dynamic code for adaptive query processing.,” in *ADMS@ VLDB*, pp. 11–22, 2022.

[9] N. Kabra and D. J. DeWitt, “Efficient mid-query re-optimization of sub-optimal query execution plans,” in *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pp. 106–117, 1998.

[10] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic, “Robust query processing through progressive optimization,” in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pp. 659–670, 2004.

[11] J. Zhao, H. Zhang, and Y. Gao, “Efficient query re-optimization with judicious subquery selections,” *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–26, 2023.

[12] R. L. Cole and G. Graefe, “Optimization of dynamic query evaluation plans,” in *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pp. 150–160, 1994.

[13] P. Bizarro, N. Bruno, and D. J. DeWitt, “Progressive parametric query optimization,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 4, pp. 582–594, 2008.

[14] G. Antoshenkov and M. Ziauddin, “Query processing and optimization in oracle rdb,” *The VLDB Journal*, vol. 5, pp. 229–237, 1996.

[15] A. Deshpande, Z. Ives, V. Raman, *et al.*, “Adaptive query processing,” *Foundations and Trends® in Databases*, vol. 1, no. 1, pp. 1–140, 2007.

[16] R. Avnur and J. M. Hellerstein, “Eddies: Continuously adaptive query processing,” in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pp. 261–272, 2000.

[17] V. Raman, A. Deshpande, and J. Hellerstein, “Using state modules for adaptive query processing,” in *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, pp. 353–364, 2003.

[18] A. Deshpande and J. M. Hellerstein, “Lifting the burden of history from adaptive query processing,” in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB ’04, p. 948–959, VLDB Endowment, 2004.

[19] B. Răducanu, P. Boncz, and M. Zukowski, “Micro adaptivity in vectorwise,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’13, (New York, NY, USA), p. 1231–1242, Association for Computing Machinery, 2013.

[20] D. Terry, D. Goldberg, D. Nichols, and B. Oki, “Continuous queries over append-only databases,” *Acm Sigmod Record*, vol. 21, no. 2, pp. 321–330, 1992.

[21] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta, "Cape: Continuous query engine with heterogeneous-grained adaptivity," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pp. 1353–1356, 2004.

[22] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "Niagaracq: A scalable continuous query system for internet databases," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pp. 379–390, 2000.

[23] S. Babu and J. Widom, "Streamon: an adaptive engine for stream query processing," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pp. 931–932, 2004.

[24] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman, "Continuously adaptive continuous queries over streams," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pp. 49–60, 2002.

[25] S. Chandrasekaran and M. J. Franklin, "Psoup: a system for streaming queries over streaming data," *The VLDB Journal*, vol. 12, pp. 140–156, 2003.

[26] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An adaptive partitioning operator for continuous query systems," in *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*, pp. 25–36, IEEE, 2003.

[27] S. Krishnamurthy, S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Madden, F. Reiss, and M. A. Shah, "Telegraphcq: An architectural status report," *IEEE Data Eng. Bull.*, vol. 26, no. 1, pp. 11–18, 2003.

[28] K. Tzoumas, T. Sellis, and C. S. Jensen, "A reinforcement learning approach for adaptive query processing," *History*, pp. 1–25, 2008.

[29] 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 Trans. Database Syst.*, vol. 46, sep 2021.

[30] Z. Wei and I. Trummer, "Skinnermt: Parallelizing for efficiency and robustness in adaptive query processing on multicore platforms," *Proc. VLDB Endow.*, vol. 16, p. 905–917, dec 2022.

[31] P. Sioulas and A. Ailamaki, "Scalable multi-query execution using reinforcement learning," in *Proceedings of the 2021 International Conference on Management of Data*, pp. 1651–1663, 2021.

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

[33] G. T. Kakkar, J. Cao, A. Sengupta, J. Arulraj, and H. Kim, "Hydro: Adaptive query processing of ml queries," *arXiv preprint arXiv:2403.14902*, 2024.

[34] A. Gounaris, E. Tsamoura, and Y. Manolopoulos, "Adaptive query processing in distributed settings," *Intelligent Systems Reference Library*, vol. 36, 01 2013.