A Portable Middleware for Plan-Based Adaptive Query Processing

Pei Mu
University of Edinburgh
United Kingdom
pei.mu@ed.ac.uk

Bita Asoodeh
University of Edinburgh
United Kingdom
bita.asoodeh@ed.ac.uk

Anderson Chaves Carniel
Huawei Technologies Research & Development (UK) Limited
United Kingdom
anderson.chaves.carniel@huawei.com

Amir Shaikhha
University of Edinburgh
United Kingdom
amir.shaikhha@ed.ac.uk

ABSTRACT

Inaccurate cardinality estimation is a major performance bottleneck for many database management systems (DBMS). Plan-based adaptive query processing (AQP) has been widely studied to address this issue, but existing solutions are tightly coupled to specific database engines, making them difficult to reuse or extend. This paper presents **AQPHub**, an extensible framework for plan-based AQP that decouples AQP strategies from specific engines. This is achieved by providing a middleware that leverages a SQL-based interface, requiring minimal effort to port to a new DBMS. Users can easily extend AQPHub by integrating different DBMSs and/or designing different AQP strategies. AQPHub shows negligible overhead while preserving the benefits of the state-of-the-art plan-based AQP methods. Our demonstration scenarios enable attendees to interact through a web interface and explore five database systems on four different AQP strategies and integration levels.

PVLDB Reference Format:
Pei Mu, Bita Asoodeh, Anderson Chaves Carniel, and Amir Shaikhha. A Portable Middleware for Plan-Based Adaptive Query Processing. PVLDB, 19(12): 4650 - 4653, 2026.
doi:10.14778/3827998.3828088

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/PeiMu/AQPHub.

1 INTRODUCTION AND RELATED WORK

In real-world analytical workloads, online analytical processing (OLAP) queries over large datasets typically involve heavy and sophisticated joins. A well-known challenge for such queries is inaccurate cardinality estimation, which causes query optimizers to choose poor execution plans and significantly degrade performance.

Over the last three decades, a set of plan-based adaptive query processing (AQP) methods [1, 3, 4, 6–9] have been proposed to address this challenge. Intuitively, plan-based AQP splits the large logical plan with heavy joins into several smaller sub-plans and updates the "real" statistics of each sub-plan as it is executed.

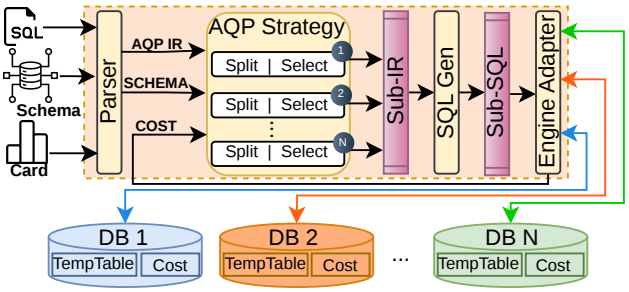

Figure 1: Architecture of AQPHub. The dashed boundary highlights the core middleware modules. AQPHub translates standard SQL into optimized sub-SQLs by different AQP strategies for various DBMS engines. Colored bidirectional arrows represent distinct execution paths and information collection (e.g., cost of sub-SQL candidates) across DBMS engines.

However, all existing plan-based AQP systems are tightly coupled to a specific database engine. This coupling makes it difficult for engineers to apply state-of-the-art (SOTA) AQP strategies to new engines or to compare strategies across systems. In particular, there is often a significant gap in engineering practices between the AQP strategy and the database management system (DBMS). For instance, QuerySplit [9] supports only 64 of the 113 Join Order Benchmark (JOB) queries due to engineering incompatibilities with PostgreSQL's internals. Furthermore, comparing the performance of different AQP strategies across engines currently requires re-implementing each strategy from scratch for each engine [7], which is a prohibitive engineering burden.

This paper introduces a middleware framework, **AQPHub**, that decouples plan-based AQP techniques from specific database engines, as shown in Figure 1. It goes beyond existing related work, as shown in Table 1, by enabling developers to seamlessly integrate existing plan-based AQP methods into their own engines, while also supporting the design of new engine-agnostic AQP algorithms that generalize across different database systems.

Core contributions. To the best of our knowledge, this paper is the first to propose a plan-based AQP middleware framework for bridging the gap in applying different AQP strategies across different DBMS engines. AQPHub offers the following *key benefits*:

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828088

Table 1: Plan-based AQP methods with respective systems.

AQP Strategy	Method	System
checkpoints checking	Dynamic Reoptimization [3]	Paradise
	POP [6]	IBM DB2
fragments checking	SIT [1]	SQL Server
	IEF [8]	SQL Server
sub-plans splitting	DYNO [4]	Hadoop
	QuerySplit [9]	PostgreSQL
	AQP-DuckDB [7]	DuckDB

- Strategy-agnostic: Compatible with different plan-based AQP strategies, thanks to providing a unified intermediate representation (IR), as shown in Section 2.1.
- Engine-agnostic: Applicable to both open- and closed-source DBMSs via a SQL-based interface to interact with DBMSs without requiring intrusive modifications to their codebases, as shown in Section 2.2.
- Community hub: Enables students and researchers to experiment with and compare AQP methods across different engines in a single environment (Section 3). Our three demonstration scenarios show how researchers can engage with AQPHub (Section 4).

2 SYSTEM OVERVIEW

Figure 1 shows the architecture of AQPHub. At a high level, this framework introduces an engine- and strategy-agnostic middleware by decoupling the AQP strategies from the DBMS engines. For each interaction with the engines, including executing the plan or collecting cardinality from the engine, AQPHub processes it using standard SQL without requiring changes to the engine. We use Query 8a from the JOB benchmark as our running example to illustrate the AQPHub workflow.

2.1 Strategy-agnostic Layer

AQPHub achieves AQP strategy-agnostic design by employing a unified IR, called AQP-IR, including a logical plan with the schema metadata and cardinality. In summary, it has an AQP Parser to convert the input SQL to AQP-IR, then applies different AQP strategies on it with the AQP splitter and AQP selector.

AQP Parser. This module translates an input SQL query to the AQP-IR and extracts schema and cardinality information from the underlying base relations. To do this, AQPHub uses the `libpg_query` library to parse the SQL code into a parse tree in `JSON` format, then converts the parse tree to our AQP-IR.

AQP Splitter. This method applies split strategies from the plan-based AQP on top of the AQP-IR, splitting it into a set of sub-IRs (i.e., each sub-IR represents a sub-plan). Figure 2a shows the Application Programming Interface (API) that allows users to implement different split strategies. In this paper, we present four strategies as examples: FK-Center, PK-Center, and MinSubquery from QuerySplit [9], and the node-based split method from AQP-DuckDB [7]. For each split strategy, we reproduce the corresponding algorithm in AQPHub by extending the `AQPStrategy` base class, and overriding `SplitIR` (cf. Figure 2a). Users can easily design and implement their own split strategies by handling the input `AQPStmt` data structure and generating a vector of sub-IR candidates.

```
class AQPStrategy {
public:
 virtual vector<AQPStmt *> SplitIR(AQPStmt *ir) = 0;
 virtual AQPStmt *SelectSubIR(const vector<AQPStmt *> &sub_irs) = 0;
};
```

(a) The API for extending new split strategies.

```
class EngineAdapter {
public:
 virtual void ExecuteSQLandCreateTempTable(const std::string &
       sub_sql, const std::string &temp_table_name) = 0;
 virtual std::pair<double, double> GetEstimatedCost(const std::
       string &sql) = 0;
};
```

(b) The API for extending new DBMS engines.

Figure 2: AQPHub API foundations.

AQP Selector. This method selects the best sub-IR from the AQP splitter as the second step of the AQP strategy. In our current examples, the split strategies from the QuerySplit [9] pick the sub-IR with the smallest estimated cost, while for the node-based split method, we employ the same selector as AQP-DuckDB [7]. AQPHub's API exposes a virtual function `SelectSubIR` for users to design their own select strategies, where its input is a vector of sub-IRs, and the output is the selected sub-IR (cf. Figure 2a).

2.2 Engine-agnostic Layer

AQPHub communicates with DBMS engines through standard SQL by employing an SQL code generator module and manages DBMSs with an engine adapter module. Also, it supports systematic evaluations by enabling/disabling AQP-related components with an AQP integration module.

SQL Code Generator. This module generates SQL code from AQP-IR by parsing the IR nodes, e.g., `ProjectionNode`, `JoinNode`, etc., and constructing the corresponding clauses, e.g., `SELECT`, `WHERE`, etc. In the running example of Query 8a, AQPHub generates three sub-SQLs for both DuckDB and PostgreSQL, as shown in Section 4.

Engine Adapter. This module adapts different DBMS engines for handling engine-related processes. In this paper, we show five DBMS engines as examples, including DuckDB, PostgreSQL, Umbra, MariaDB, and OpenGauss. AQPHub provides a base class `EngineAdapter` for users to easily extend the other DBMS engines. In our design, we override all engine-related functions, including `ParseSQL`, `GetTempTableCardinality`, etc., with engine-specific APIs. For brevity, we highlight two core functions whose implementations differ most across different engines, as shown in Figure 2b. For example, we run `"CREATE TEMP TABLE ... AS (...)"` for executing the sub-SQL and generating a temporary table in PostgreSQL, but executing the `"PreparedStatement"` of DuckDB through its `"Prepare"` API. To get the estimated cost, which is used in the AQP selector, we run SQL code of `"EXPLAIN (FORMAT JSON) ..."` in PostgreSQL, but collect the estimated cardinality immediately through DuckDB's `"ExtractPlan"` API. As the output, in our running example, we compare the final sub-SQL between DuckDB

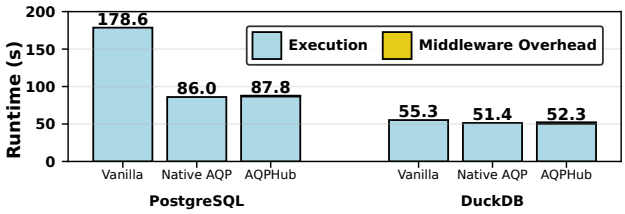

Figure 3: Performance comparison between the vanilla DBMS, native AQP, and AQPHub for the JOB benchmark. The middleware overhead incurred by the AQPHub is negligible.

and PostgreSQL, where both of them use `temp` tables generated from the previous sub-SQLs, as shown in Section 4.

AQP Integration. This module provides a configurable controller over module-level integration, allowing the cardinality updater and sub-plan combiner to be selectively enabled or disabled in accordance with plan-based AQP strategies. By default, the cardinality updater is enabled, and AQPHub obtains exact cardinality by querying the temporary tables' size via SQL. To measure the performance impact of AQP without updating cardinality, users can disable the updater, in which case AQPHub reuses the estimated cardinality as we described for the engine adapter's API. Also, AQPHub supports the configuration of merging the generated sub-SQLs into a whole SQL with specified execution and materialization order. AQPHub achieves this by first processing the AQP pipeline and collecting all sub-SQLs (including executing each of them to get the real cardinality). Then, the code generator encapsulates each sub-SQL, enabling subsequent query fragments to reference these materialized intermediate results.

3 EVALUATION ANALYSIS

In this section, we describe the general setup and the performance evaluation comparing AQPHub with vanilla DBMSs and native AQP systems. A report with performance analysis across five engines with different AQP strategies is available (See Artifact Availability).

3.1 Experimental Setup

AQPHub sets up the backend on a `x86/64` Dell server with a single Intel(R) Xeon(R) E-2236 CPU at 3.40 GHz, 64 GB DDR4 RAM at 2666 MHz, 1 × 931.5 GB SATA HDD (6 GB/s). We use the same configurations for DBMS engines and workloads as the SOTA AQP [7].

3.2 Performance Analysis of AQPHub

In this section, we evaluate the performance of AQPHub relative to vanilla DBMSs and native AQP systems, and provide an end-to-end performance comparison. We measure DuckDB and PostgreSQL with the same queries and settings as the SOTA AQP [7] for the JOB benchmark. We do not consider the other three DBMS systems, as they do not have native AQP implementations.

Overall, as shown in Figure 3, comparing the native AQP system (the middle bar) to AQPHub (the right bar) across each DBMS, we observe a slight increase in runtime due to middleware overhead. This mainly comes from the engine's multiple rounds of parsing for each sub-SQL and from the conversion time between the input SQL, AQP-IR, and generated sub-SQLs in the middleware. However, Figure 3 shows that even with AQPHub's negligible overhead, the key finding that plan-based AQP improves performance over vanilla DBMS across different database engines [7] remains consistent.

4 DEMONSTRATION

We demonstrate AQPHub through an interactive web interface, as shown in Figure 4, in which attendees actively run queries and observe AQPHub's behavior in real time. The interface is organized into two vertical panels: a configuration panel on the left where users specify their dataset and query, and a live output panel on the right where generated sub-SQLs, results, and performance breakdowns are displayed. Below, we describe the interface panels and three structured demonstration scenarios designed to engage attendees with progressively deeper insight into AQPHub.

4.1 Panels

Input Panel. The input panel on the left side is where the users' journey starts. In this step, the user can choose among the predefined queries from different benchmarks or write a custom one. We embed the JOB with IMDB dataset [5] and Decision Support Benchmark (DSB) [2] with the generated dataset into our demonstration environment. In addition to these standard queries, attendees are also encouraged to write their own queries using the provided datasets and immediately observe how different engines and/or AQP strategies handle them. The user can view the updates in the output panel by clicking the `"Run Query"` button.

Output Panel. The output panel on the right side displays AQPHub's output with three demonstration scenarios at the top across different DBMS systems, AQP strategies, and integration levels. For each scenario, the user can compare the generated sub-SQLs and performance breakdowns against different settings.

4.2 Scenarios

In this section, we discuss the different scenarios in which the user interacts with the web interface. For brevity, we only show one representative interface snapshot in Figure 4 for demonstration Scenario 1. Scenarios 2 and 3 share the same input panel as Scenario 1, but display different sub-SQL comparisons in their output panel.

Scenario 1: Varying DBMS. In this scenario, the attendee selects a fixed AQP strategy (FK-Center by default), fixed integration levels (the cardinality updater enabled, as `"CardUpdated"`, and the sub-plan combiner disabled, as `"Splitted"`, by default), and runs the same query against multiple DBMS engines, as shown in Figure 4. After clicking the `"Run Query"` button in the input panel, the user can select the `"DB Systems"` tag to view output for different engines in the output panel. Then the user can choose which sub-SQL to view the query code. The user can also compare the sub-SQLs generated by different engines from the dropdown list. Figure 4 shows an example with the final sub-SQL comparison between DuckDB and PostgreSQL, where DuckDB uses the `temp2` table generated from the `"Sub-SQL 2"`, but PostgreSQL uses both the `temp1` table and the `temp2` table from the previous sub-SQLs. This reveals a key insight: even with the same AQP strategy and query, different engines choose different sub-plans due to differences in their optimizers and cardinality estimators. A comparison without AQPHub would require significant per-system engineering effort.

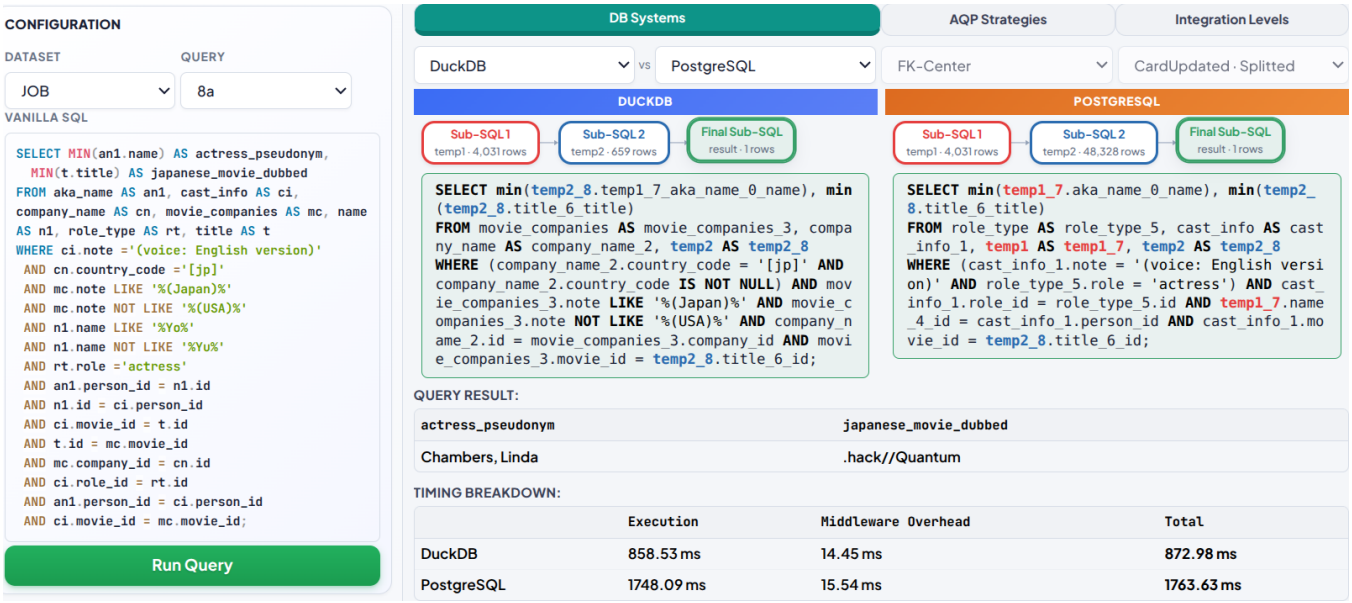

Figure 4: Demo scenario of AQPHub. The left part is the input panel where users can select the dataset and SQL, or have their custom query. The right part is the output panel, where users can view three different demonstration scenario tabs. In this screenshot, we show the example of comparing the sub-SQLs between DuckDB (left) and PostgreSQL (right). The query result and time breakdown are shown at the bottom.

Scenario 2: Varying AQP strategies. In Scenario 2, the attendee selects a fixed engine (DuckDB by default), fixed integration configurations (the same as Scenario 1), and runs the same query against different AQP strategies (with the `"AQP Strategies"` tag selected in the output panel). Similar to Scenario 1, the user can compare sub-SQLs with different AQP strategies from the dropdown list. Scenario 2 reveals a similar insight to Scenario 1: the sub-SQLs differ across AQP strategies, resulting in different performance.

Scenario 3: Varying integration levels. As the final scenario, we show the case with different integration levels of AQPHub, where the attendees select a fixed engine and AQP strategy, and run the same query across different integration levels (with the `"Integration Levels"` tag selected in the output panel). Different integration levels correspond to enabling or disabling the cardinality updater (`CardUpdated/CardEstimated`) and/or sub-plan combiner (`Splitted/Combined`), following the evaluation methodology of [7], allowing users to isolate the contribution of each module. To facilitate comparison of the sub-plan combiner, the web interface presents two columns: with the sub-plan combiner disabled, all sub-SQLs are listed in the `"Splitted"` column; with the combiner enabled, the merged SQL wrapped in `"CREATE TEMP TABLE"` is displayed in the `"Combined"` column. The key insight of this scenario is that researchers can flexibly evaluate the performance impact of individual AQP modules.

5 CONCLUSION

We presented AQPHub, the first plan-based AQP middleware framework that decouples AQP strategies from specific DBMSs through a standard SQL-level interface. AQPHub incurs negligible overhead while preserving the experimental conclusions of the SOTA plan-based AQP [7], by comparing to the vanilla DBMS and native AQP system. We demonstrated it across five different DBMS systems with four different AQP strategies using the JOB and DSB benchmarks. Through three interactive demonstration scenarios, attendees can directly observe the benefits of a portable, strategy- and engine-agnostic AQP middleware framework and explore its behavior across engines, strategies, and integration levels.

ACKNOWLEDGEMENT

The authors thank Huawei for their support of the distributed data management and processing lab. at the University of Edinburgh.

REFERENCES

[1] Nicolas Bruno and Surajit Chaudhuri. 2002. Exploiting statistics on query expressions for optimization. In *SIGMOD'02*. 263–274.
[2] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: A decision support benchmark for workload-driven and traditional database systems. *Proceedings of the VLDB Endowment* 14, 13 (2021), 3376–3388.
[3] Navin Kabra and David J DeWitt. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 106–117.
[4] Konstantinos Karanasos, Andrey Balmin, Marcel Kutsch, Fatma Ozcan, Vuk Ercegovac, Chunyang Xia, and Jesse Jackson. 2014. Dynamically optimizing queries over large scale data platforms. In *SIGMOD'14*. 943–954.
[5] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.
[6] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. 2004. Robust query processing through progressive optimization. In *SIGMOD'04*. 659–670.
[7] Pei Mu, Anderson Chaves Carniel, Antonio Barbalace, and Amir Shaikhha. 2026. Systematic Evaluation of Plan-based Adaptive Query Processing [Experiment, Analysis, and Benchmark]. In *ICDE'26*.
[8] Thomas Neumann and Cesar Galindo-Legaria. 2013. Taking the edge off cardinality estimation errors using incremental execution. In *BTW'13*. 73–92.
[9] Junyi Zhao, Huanchen Zhang, and Yihan Gao. 2023. Efficient Query Re-optimization with Judicious Subquery Selections. *SIGMOD'23* 1, 2 (2023), 1–26.