Towards a Unified Query Plan Representation

Jinsheng Ba
ETH Zurich

Manuel Rigger
National University of Singapore

***Abstract*—In database systems, a query plan is a series of concrete internal steps to execute a query. Multiple testing approaches utilize query plans for finding bugs. However, query plans are represented in a database-specific manner, so implementing these testing approaches requires a non-trivial effort, hindering their adoption. We envision that a unified query plan representation can facilitate the implementation of these approaches. In this paper, we present an exploratory case study to investigate query plan representations in nine widely-used database systems. Our study shows that query plan representations consist of three conceptual components: operations, properties, and formats, which enable us to design a unified query plan representation. Based on it, existing testing methods can be efficiently adopted, finding 17 previously unknown and unique bugs. Additionally, the unified query plan representation can facilitate other applications. Existing visualization tools can support multiple database systems based on the unified query plan representation with moderate implementation effort, and comparing unified query plans across database systems provides actionable insights to improve their performance. We expect that the unified query plan representation will enable the exploration of additional application scenarios.**

***Index Terms*—Case Study, Query Plan, Unified Representation**

I. Introduction

Database Management Systems (DBMSs) are fundamental software systems used to store, retrieve, and run queries on data. They are used in almost every computing device [1], [2], [3]; thus, any bug has potentially severe consequences. Recently, automated testing approaches for DBMSs have gained broad adoption. *Query Plan Guidance* (*QPG*) [4] guides test case generation towards diverse query plans aiming to expose more bugs, *Cardinality Estimation Restriction Testing* (*CERT*) [5] identifies performance bugs by examining query plans, and *Mozi* [6] checks the consistency across query plans of the same query to find incorrect query optimizations.

While these approaches have been effective and have found more than 200 previously unknown and unique bugs, implementing them requires a non-trivial effort, as they require DBMS-specific logic to parse and process query plans. A query plan refers to a series of concrete steps to execute a query written in a declarative language, such as *Structured Query Language* (SQL). DBMSs expose query plans in various formats (*e.g.*, in textual format), in which case we refer to them as *serialized query plans*. Unlike query languages, for which widely-used, standardized [7], and formalized [8] languages exist, such as SQL [9], the formats in which serialized query plans are exposed are non-standardized and DBMS-specific. For example, a predicate in the `WHERE` clause of SQL corresponds to a concrete step to filter data in the query plans of TiDB [10], but corresponds to a property of another step to scan tables in the query plans of PostgreSQL [11]. We refer to the different ways in which serialized query plans are represented as *query plan representations*. Considering that hundreds of DBMSs exist,[1] implementing the above testing methods requires significant effort as they need to account for differences in query plan representations, thus significantly hindering the effectiveness of the above approaches.

We envision that a unified query plan representation would remove the roadblock to implementing the above testing approaches. In this work, we systematically study query plan representations. We present an exploratory case study [12], which is a method to investigate a phenomenon in depth, including both qualitative and quantitative research methods. We collected documents, source code, and third-party applications of the query plans in nine popular DBMSs across five different data models, and summarized the commonalities and differences of query plan representations. Our study shows that query plan representations are based on three conceptual components: operations, properties, and formats. Based on the study, we designed a unified query plan representation, which allows the representation of all conceptual components we studied from DBMS-specific query plans, so that we can easily apply automated testing methods to multiple targets.

To demonstrate the utility of the unified query plan representation, we implemented a prototype called *UPlan* to maintain the representation. Currently, we implemented converters that convert a DBMS-specific serialized query plan to a unified representation; in the long term, we hope that DBMSs will directly expose such a unified representation, rendering such converters unnecessary. We re-implemented *QPG* and *CERT* in a DBMS-agnostic way based on *UPlan*, enabling the large-scale adoption of both testing methods. In our evaluation, we considered three DBMSs: MySQL, PostgreSQL, and TiDB. We found 17 previously unknown and unique bugs, and one bug in the original *QPG* implementation for parsing TiDB's query plans. Additionally, we found that *UPlan* can facilitate other applications based on query plans. First, we modified a visualization tool to support our unified query plan representation. We show that an existing DBMS-specific visualization tool can support five DBMSs through *UPlan* with only moderate implementation effort. Second, the unified query plan representation enables the comparison of query

The first author started the project at the National University of Singapore.

[1] https://dbdb.io/ lists 1001 DBMSs as of August 2024.

plans in different DBMSs, which provides actionable insights for query optimization.

We believe that our unified representation reduces the effort to build applications, including testing, on serialized query plans. We include all study results and the prototype in our supplementary materials.[2] For ease of access, we also provide a comprehensive website,[3] which provides supportive additional information including illustrative examples, explanations of studied query plans, and example applications. We make the following contributions:

- A study of query plan representations and results that allow both practitioners and researchers to study the results.
- A proposal and a reusable library *UPlan* for a unified query plan representation.
- Three applications, including testing, of the unified representations on serialized query plans.

II. Background

Query optimization. DBMSs consider a variety of potential execution strategies for a query written in a declarative language [13], and each execution strategy includes specific steps to execute the query. Query optimization is the process of determining an execution strategy, and typically includes these steps: parsing queries into logical query plans that represent the logical steps to execute the query, including operations such as join, converting them into physical query plans, which represent the executable operations to execute the query, such as hash join, and determining a physical query plan to execute.

Query plans. Query plans, typically short for physical query plans, are organized as a Directed Acyclic Graph (DAG) [14], in which a step outputs data to another step as input followed by directed edges. As query optimizations are specific to the storage engines and query execution engines, query plans are non-standardized. By executing a query with a specific prefix, such as `EXPLAIN` for most relational DBMSs that support SQL, we can obtain a *serialized query plan* in a DBMS-specific *query plan representation*. In this paper, we study various query plan representations. We did not study logical query plans as they usually are not exposed by DBMSs.

III. Query Plan Case Study

We adopted an exploratory case study as the method to investigate query plan representations. The case study, as an empirical method, is used for investigating a contemporary phenomenon in depth and within its real-world context [15] (the "case"). An exploratory case study is a specific case study that generates new questions, propositions, or hypotheses during the study. In this paper, we chose this method, because we wanted to gain an in-depth understanding of how query plans are represented within mature DBMSs. We followed common guidelines of case study research [12] to design and conduct this study.

[2]https://zenodo.org/records/15097905

[3]https://nus-test.github.io/uplan/

TABLE I
The studied nine DBMSs ranging from various data models, development modes, and release dates.

DBMS	Version	Data Model	Release	Rank
InfluxDB [16]	2.7.0	Time-series	2013	28
MongoDB [17]	6.0.5	Document	2009	5
MySQL [18]	8.0.32	Relational	1995	2
Neo4j [19]	5.6.0	Graph	2007	21
PostgreSQL [11]	14.7	Relational	1989	4
SQL Server [20]	16.0.4015.1	Relational	1989	3
SQLite [21]	3.41.2	Relational	1990	10
SparkSQL [22]	3.3.2	Relational	2014	33
TiDB [10]	6.5.1	Relational	2016	79

A. Case Study Design

Objectives and research questions. The goal of this study was to investigate the phenomenon of non-standardized query plan representations within real-world DBMSs. We aim to achieve this goal by answering the following research questions (RQs):

RQ1 How are serialized query plans represented?

RQ2 Do query plan representations share a common conceptual basis?

Case selection. The case study of this paper is characterized as single-case [12]: the query plan representation is the case, while different DBMSs are the units of the analysis. Table I shows the DBMSs that we selected for the study. To conduct a representative and comprehensive study, we made a diverse selection of DBMSs. First, we chose DBMSs of various data models: relational, document, graph, and time-series data models, based on the assumption that the DBMSs of different models might have different query plan representations. The relational data model is the most widely-used one [23], and other non-relational models, which are also called NoSQL models, are widely used for maintaining unstructured or semi-structured data [24]. Then, we chose both classic and new-generation DBMSs [25] ranging from release years from the 1980s to the 2010s. The architectures of the chosen DBMSs include standalone DBMSs and an embedded DBMS, SQLite, which runs in the same process as the application. We included both a commercial DBMS, SQL Server, and open-source DBMSs. Apart from conventional DBMSs, we included an analytics engine for large-scale data processing, SparkSQL, which also optimizes queries. To choose widely-used DBMSs for study, we chose them referring to the DBMS ranking website[4] as shown in the column *Rank*. We chose the latest release version of each DBMS.

Data collection. We collected data from multiple sources: documents, source code, officially integrated development environments (IDEs), and third-party applications based on query plans. The use of multiple data sources allowed us to perform data source triangulation [12], that is, we could confirm the study results from the above four different types of data sources. We included the detailed lists of the data sources in the supplementary materials for reference.

[4]https://db-engines.com/en/ranking as of August 2024.

Listing 1. Query plan examples of PostgreSQL and SQLite in text format. Some properties are truncated by ... due to space constraints.

```
1  CREATE TABLE t0 (c0 INT);
2  CREATE TABLE t1 (c0 INT);
3  CREATE TABLE t2 (c0 INT PRIMARY KEY);
4  INSERT INTO t0 SELECT * FROM generate_series(1,1000000);
5  INSERT INTO t2 SELECT * FROM generate_series(1,100);
6
7  ----------------------PostgreSQL-----------------------
8  EXPLAIN (SUMMARY TRUE) SELECT t1.c0 FROM t0 INNER JOIN
       t1 ON t0.c0 = t1.c0 WHERE t0.c0 < 100 GROUP BY
       t1.c0 UNION SELECT c0 FROM t2 WHERE c0 < 10;
9
10 HashAggregate (cost=62998.82..63009.32 rows=1050...)
11  Group Key: t1.c0
12  ->Append (cost=27150.40..62996.20 rows=1050 width=4)
13    ->Group (cost=27150.40..62949.08 rows=200 width=4)
14      Group Key: t1.c0
15      ->Gather Set (cost=27150.40..62948.08 rows=400...)
16        Workers Planned: 2
17        ->Group (cost=26150.38..61901.89 rows=200...)
18          Group Key: t1.c0
19          ->Set Join (cost=26150.38..56906.48...)
20            Set Cond: (t0.c0 = t1.c0)
21            ->Sort (cost=25970.60..26362.39...)
22              Sort Key: t0.c0
23              ->Parallel Seq Scan on t0 (cost=0.00...)
24                Filter: (c0 < 100)
25            ->Sort (cost=179.78..186.16 rows=2550...)
26              Sort Key: t1.c0
27              ->Seq Scan on t1 (cost=0.00..35.50...)
28    ->Bitmap Heap Scan on t2 (cost=10.74..31.37...)
29      Recheck Cond: (c0 < 10)
30      ->Bitmap Index Scan on t2_pkey (cost=0.00...)
31        Index Cond: (c0 < 10)
32 Planning Time: 0.124 ms
33
34 -------------------------SQLite-------------------------
35 EXPLAIN QUERY PLAN SELECT t1.c0 FROM t0 INNER JOIN ...;
36
37 `--COMPOUND QUERY
38    |--LEFT-MOST SUBQUERY
39    |  |--SCAN t0
40    |  |--SEARCH t1 USING AUTOMATIC COVERING INDEX (c0=?)
41    |  `--USE TEMP B-TREE FOR GROUP BY
42    `--UNION USING TEMP B-TREE
43       `--SEARCH t2 USING COVERING INDEX ...
```

Data analysis. For each DBMS, we first examined its official documents describing its query plan representation. Where the source code was available, we inspected it to gain a better understanding of query plan representations. We also ran official test cases in the official IDEs to observe query plan representations in real-world test cases. For the DBMSs that had open-source query plan applications, we further examined how query plan representations are explained and utilized from a third-party perspective. To answer RQ1, we analyzed the query plan representations from the above data sources qualitatively. To answer RQ2, we qualitatively identified conceptual components in the query plan representations, and quantitatively compared them. To satisfy observation triangulation [12], one author conducted the study, and another author validated the finding against the raw data.

B. Findings Overview

We found that the studied DBMSs share three conceptual components: *operations*, *properties*, and *formats*. As described in Section II, query plans are considered DAGs. In practice, serialized query plans are usually organized in a tree structure, and each node has a key. The keys typically refer to *operations*, which are concrete steps executed by DBMSs to retrieve, process, or output data in response to a query, such as `Full Table Scan`, which refers to the step to scan an entire table. Each *operation* is associated with zero or multiple *properties*, which involve *operation*-related information, such as `row`, which refers to the estimated number of rows returned. Apart from *operation*-associated *properties*, plans also have *properties* associated with them, such as `planning time`, which refers to the time to generate the query plan. A special case is the key *Filter* in the TiDB query plans. It represents the condition that the output data from its child node satisfies, so we deem it as a *property* instead of an *operation*. DBMSs typically allow serializing query plans in various *formats*, such as text, table, JSON, and XML.

Listing 1 shows two examples of query plan representations of PostgreSQL and SQLite in a textual format. Lines 1–5 show the SQL statements that create and populate the tables. The function `generate_series` generates data to populate tables `t0` and `t2`. For PostgreSQL, executing the statement in line 8 outputs the serialized query plan as shown in lines 10–32. The bold texts denote operations, and the non-bold texts denote properties. For example, the operation **HashAggregate** in line 10 has the properties `cost, rows, width, and Group Key`. The tree structure is denoted through hierarchical indents, in which the operation with longer indents is a child of the operation with short indents. Although both DBMSs are based on the relational data model, and support a textual format, their query plan representations are significantly different.

C. Operations

Identification. We identified operations from the source code of all DBMSs, except for SQL Server, whose source code is not public. MongoDB, MySQL, PostgreSQL, and TiDB specify operations in enumeration variables or lists. Neo4j and SparkSQL define each operation as a class or structure. SQLite defines operations as strings that are passed to the query plan generation process. We found that only SQL Server and Neo4j provided detailed documents for operations, while the other DBMSs' documents had incomplete lists of operations, and relied on illustrative examples.

Classification. We classified operations into seven categories, as shown in the left part of Table II. Our classification is based on the well-established relational algebra theory [26], which uses algebraic structures for modeling data and defining queries on it. Relational algebra operates on homogeneous sets of tuples $S = \{(s_{j1}, s_{j2}, ..., s_{jn} | j \in 1...m)\}$, where we commonly interpret m to be the number of tuples in a table and n to be the number of attributes in a tuple. We considered basic operations: selection σ, projection Π, join $\bowtie$, cartesian product $\times$, set operations $\cup$, $\cap$, $-$, aggregation γ, and rename ρ. Although more specific join algebra operators are available, such as left join: $\rtimes$, we used $\bowtie$ to represent all kinds of join operations for simplicity. DBMSs define specific operations in query plans to realize the above algebra except for ρ. On average, every DBMS defines 48 operations in query plans. Neo4j has the most operations, while InfluxDB has no operations. We noticed that Neo4j requires more operations

TABLE II
THE NUMBER OF OPERATIONS AND PROPERTIES IN QUERY PLAN REPRESENTATIONS. RA: RELATIONAL ALGEBRA.

	Operations								Properties				
DBMS	**Producer**	**Combinator**	**Join**	**Folder**	**Projector**	**Executor**	**Consumer**	**Sum**	**Cardinality**	**Cost**	**Configuration**	**Status**	**Sum**
RA Operators	σ	$\cup, \cap, -$	$\bowtie, \times$	γ	Π								
InfluxDB	0	0	0	0	0	0	0	0	5	0	0	1	6
MongoDB	14	9	0	5	3	10	3	44	16	5	18	12	51
MySQL	15	3	2	1	0	2	0	23	3	6	3	10	22
Neo4j	18	11	43	6	3	17	13	111	3	3	12	7	25
PostgreSQL	18	8	3	3	0	9	1	42	8	17	42	40	107
SQL Server	15	3	3	3	0	16	19	59	4	4	7	3	18
SQLite	3	6	3	0	0	5	0	17	0	0	3	0	3
SparkSQL	7	1	2	6	0	43	18	77	11	11	0	0	22
TiDB	19	6	7	5	1	13	5	56	2	5	4	1	12
Avg:	12	5	7	3	1	13	7	48	6	6	10	8	30

Planner COST
Runtime version 5.10

Operator	**Rows**	**...**
+ProduceResults	8	...
+UndirectedRelationshipIndexContainsScan	8	...

Total database accesses: 5, total allocated memory: 184

Fig. 1. An example query plan of the Neo4j operations of the Join category.

on nodes and relationships of the graph data model, such as the operation to set node properties [27], and SQLite is a lightweight DBMS with a limited number of operations, such as lacking the operations for creating tables. Overall, NoSQL DBMSs have more operations than relational DBMSs. An exception is InfluxDB, which does not define operations. InfluxDB's operations are disregarded in query plans due to the limited set of operations supported by the single-tuple time-series data.

Producer. The *Producer* category consists of the operations that retrieve data from storage or return constants instead of from children's operations. These operations realize the algebra operator σ, which selects data that satisfies a given predicate. The operations in the *Producer* category are data sources of queries, so they are typically leaf nodes of query plans. For example, in Listing 1, the operation **SEARCH** in line 43 represents a full table scan, and **Bitmap Heap Scan** in line 28 represents a data scan from bitmaps in heap memory. Six of nine DBMSs define more than ten operations in the *Producer* category, because reading data is usually expensive, and thus reads are customized for different scenarios aiming to improve efficiency. For example, indexes [28], [29] can be used to efficiently read data.

Combinator. The *Combinator* category consists of the operations that change the permutation and combination of tuples, such as sort and union, with no changes to attributes. These operations realize the algebra operators $\cup$, $\cap$, and $-$. In Listing 1, the operation **Append** of PostgreSQL merge tuples from different children operations to a single set by the operations in lines 13 and 28, and is associated with query clause `UNION` in line 8. To execute a similar functionality in SQLite, the operations **COMPOUND QUERY** and **UNION** combine data objects by the operations in lines 37 and 42, and both operations also belong to *Combinator* category.

Join. The *Join* category consists of the operations that generate new tuples by recombining attributes. These operations realize the algebra operators $\bowtie$ and $\times$. In Listing 1, the operation **Set Join** in line 19 combines two ordered data from the operations in lines 21 and 25 based on the common fields `t0.c0` and `t1.c0`. MongoDB has no *Join* operations, because it includes only a single document tuple for querying and lacks support for combining data from multiple documents. The higher number of *Join* operations in Neo4j is because we classified the operations on the edges of the graph data model as belonging to the *Join* category. In the graph data model, edges establish relationships between nodes, and a broader range of operations can be performed on the edges. For example, in Neo4j, executing the simple query `MATCH ()-[r]->() WHERE r.title ENDS WITH 'developer' RETURN r` retrieves the relationships whose properties satisfy `r.title ENDS WITH 'developer'`. The corresponding query plan is shown in Figure 1. Each line in the table represents an operation and associated properties, and the content outside the table is plan-associated properties. The query plan specifies scanning the relationships, which indicates both nodes (*i.e.*, tuples) each relationship connects, so the operation **UndirectedRelationshipIndexContainsScan** belongs to *Join*.

Folder. The *Folder* category consists of the operations that derive new tuples from a set of tuples. These operations realize the algebra operator γ. In Listing 1, the operations **HashAggregate** and **Group** are in the *Folder* category and represent data aggregation and grouping, respectively. SQLite does not define operations in the *Folder* category, but shows similar information in properties together with the operations in the *Producer* category. The operations in the *Folder* denote DBMSs' data transformation capability, so most DBMSs support operations in the *Folder* category.

Projector. The *Projector* category consists of the operations that remove attributes from all tuples. These operations realize the algebra operator Π. No operation in Listing 1 belongs to this category, and 6 of 9 DBMSs have no operations in this category. We observed that these operations correspond to column lists in `SELECT` statements, and they are not explicitly denoted in query plans.

Executor. The *Executor* category consists of the operations that make no change to tuples and attributes. These operations are typically DBMS-specific internal operations, and do not have a clear correspondence with any algebra operators. In

Listing 1, the operation **Gather Set** in line 15 merges the data from the operation **Group** running in other processes for parallel execution. DBMSs define these operations to cater to various designs and goals. For example, PostgreSQL defines the operation **MEMORIZE** to cache the output from node children into memory to speed up processing. We classified these operations into the *Executor* category. The average number of operations in the *Executor* category is 13, and SparkSQL has significantly more operations, 43, in the *Executor* category than others, because it defines multiple operations to interact with other components, such as the Python library *pandas*.

Distributed DBMSs. Within our studied DBMSs, MongoDB, SparkSQL, and TiDB are distributed DBMSs, which allow executing query plans in parallel across multiple computing nodes [30]. Although they have significantly different execution steps than single-node DBMSs, their query plans have no structural difference from other DBMSs' query plans. The operations that can be executed in parallel are not explicitly stated in query plans. A major difference is that distributed DBMSs define operations to exchange data across nodes. For example, TiDB defines the operations **ExchangeReceiver**, **ExchangeSender**, and **Shuffle** to send, receive, and shuffle data across nodes. We classified them into *Executor*. InfluxDB and Neo4j only support the distributed architecture in enterprise editions, while we studied their community editions.

Consumer. The *Consumer* category consists of operations that have no output. These operations correspond to non-query SQL statements, such as `UPDATE`, so they lack a counterpart in relational algebra. Apart from queries, which, in SQL, are `SELECT` statements, DBMSs also support other statements, such as `CREATE` and `UPDATE` in SQL. DBMSs also expose query plans for these statements, and name them as execution plans for wider usage [31]. The operations of the *Consumer* category usually modify stored data or system variables. For example, SparkSQL uses the operation **SetCatalogAndNamespace** to control a particular system variable, and we assign these operations to the *Consumer* category.

D. Properties

Identification. Each property is associated with either an operation or a query plan, and the available properties are statically encoded as strings near the generation processes of associated operations or query plans in the source code. InfluxDB's query plan representation includes only a list of plan-associated properties, while other DBMSs include both plan-associated and operation-associated properties. In Listing 1, PostgreSQL's operations have various general properties enclosed in brackets, along with operation-specific properties in the subsequent lines. At the bottom of the serialized query plan, the property `Planning Time` is plan-associated and represents the time to produce the query plan. In the documentation, similar to operations, only SparkSQL provides a comprehensive list of properties, while other DBMSs only show examples of properties. We explain that it is difficult to maintain the documentation of properties, which are diverse and evolving over versions. To maintain the information of properties, some third-party tools, like pgMustard, maintain a curated list of properties with accompanying explanations, but are usually commercial.

Classification. We identified four categories of properties, as shown in the right part of Table II. On average, every DBMS defines 30 properties. PostgreSQL has the most properties, since it includes many fine-grained properties. For example, it defines three properties to show the status of parallel computations: `worker number`, `worker launched`, and `worker planned`, while other DBMSs provide at most one property for parallel computation. We give a detailed explanation of each property category as follows.

Cardinality. The *Cardinality* category consists of the numeric properties that denote the estimated data size returned by operations. The properties in this category can be associated with operations of any category or the serialized query plan as a whole. In Listing 1, the properties `rows` and `width` belong to the *Cardinality* category and represent the estimated number of returned rows and width. These estimates are derived from statistical information [32] that DBMSs collect, such as the total number of rows and maximum values. Query plans with lower estimated cardinalities are more likely to be selected for execution during cost-based query optimization. Some DBMSs, such as MySQL, provide more fine-grained information about the number of rows that are read and returned. As a lightweight DBMS, SQLite uses simple heuristics to estimate cardinalities, and omits properties in the *Cardinality* category.

Cost. The *Cost* category consists of the numeric properties that denote the estimated resource consumption. The properties in this category can be associated with operations of any category or the serialized query plan as a whole. In Listing 1, the property `cost` is in the *Cost* category, and the two numbers of `cost` denote the cost scores of starting and finishing the associated operation by estimating the total consumption of disk and CPU. As in the *Cardinality* category, SQLite lacks properties in the *Cost* category.

Configuration. The *Configuration* category consists of the properties that configure the operations' parameters, and their values are configuration options which are usually strings or boolean values. The properties in the *Configuration* category can be associated with operations in any category or the serialized query plan, and are typically specific to operations. In Listing 1, PostgreSQL's properties `Group Key`, `Set Cond`, `Sort Key`, `Recheck Cond`, `Index Cond`, `Filter` are in the *Configuration* category and are specific to the associated operations to show the keys used to group, the condition to join, the key to sort, the condition to check, the index condition, and the predicate to exclude data, respectively. SQLite's property `USING COVERING INDEX` denotes the index condition.

Status. The *Status* category consists of the properties of runtime status, and their values are runtime metrics which are usually strings or numbers. These properties can be associated with operations in any category or the serialized query plans, and typically differ depending on the operations they are attached to. In Listing 1, the property `Workers Planned` is in the *Status* category and shows the number of available computing

TABLE III
THE OFFICIALLY SUPPORTED FORMATS OF QUERY PLANS.

	Natural			Structured		
DBMS	**Graph**	**Text**	**Table**	**JSON**	**XML**	**YAML**
InfluxDB		✓				
MongoDB	✓			✓		
MySQL	✓		✓	✓		
Neo4j	✓		✓	✓		
PostgreSQL		✓	✓	✓	✓	✓
SQL Server	✓	✓	✓		✓	
SQLite		✓				
SparkSQL	✓	✓				
TiDB	✓		✓	✓		

TABLE IV
THIRD-PARTY VISUALIZATION TOOLS FOR QUERY PLANS.

Tool	**DBMSs**	**License**
Postgres Explain Visualizer 2 [35]	PostgreSQL	Open-source
pgmustard [36]	PostgreSQL	Commercial
pganalyze [37]	PostgreSQL	Commercial
ApexSQL [38]	SQL Server	Commercial
Plan Explorer [39]	SQL Server	Commercial
Azure Data Studio [40]	SQL Server	Commercial
Dbvisualizer [41]	MySQL, PostgreSQL, SQL Server	Commercial

nodes to execute the associated operation. As another example, due to the distributed architecture, TiDB defines the property `taskType` to show the nodes that the operation is assigned to execute on. The properties in the *Status* category show running status, and are determined by the execution environment, while the properties in the Parameters are usually decided by queries. The properties in both *Status* and *Configuration* categories are customized, and thus are typically different across DBMSs, while the properties in other categories share similar semantics or functionalities across DBMSs.

E. Formats

DBMSs serialize query plans to various formats for different purposes. The formats are typically controlled by a specific configuration in queries, such as for PostgreSQL, the statement `EXPLAIN (FORMAT JSON) SELECT...` serializes the query plan representation to JSON format. We classified all formats into two categories: *natural* formats, which are optimized for readability, and *structured* formats, which are optimized for machine reading. We also consider the graph formats that are supported in official IDEs.

Table III shows the different formats of query plan representations. Overall, DBMSs support more formats in the *natural* category rather than the *structured* category, suggesting that DBMSs prioritize readability over machine processing. Due to the lack of a standard, none of the formats is supported by all DBMSs. For the same DBMS, the formats in the *natural* category usually include less information than the formats in the *structured* category. For example, in Listing 1, the property `Parent Relationship` represents how the associated operation passes data to another operation. This property is ignored in the text format of the *natural* category, but is shown in the JSON format of the *structured* category. We provide more details of each format as follows.

Natural category. The *natural* category includes graph, text, and table formats. Query plans are usually serialized as graphs for DBMSs' IDEs, such as Workbench[5] for MySQL, Compass[6] for MongoDB. Text formats represent query plans as plain text, such as shown in Listing 1. Table formats encode each operation and associated properties in a line, and use line numbers to represent the tree structure of query plans. Graph formats are intuitive to understand, so graph formats are supported by most DBMSs.

Structured category. The *structured* category includes the JSON, XML, and YAML formats. These formats are standardized and widely used for exchanging data [33], [34]. JSON is more widely supported by DBMSs than other *structured* formats, and PostgreSQL supports all *structured* formats. *Structured* formats are not supported by some DBMSs, such as InfluxDB, SQLite, and SparkSQL. SQLite can output a *structured* format of bytecode, which consists of low-level instructions, not operations and properties, so we do not consider it as a *structured* query plan representation.

Visualization. Apart from the graph formats in official IDEs, third-party visualization tools show query plans based on *structured* formats to enhance the readability of query plans. Table IV shows the visualization tools we found for the studied DBMSs. Six of the seven tools are commercial, suggesting the value of understanding query plan representations for developers. Building these tools requires non-trivial effort, because a tool is specific to a DBMS.

IV. UNIFIED QUERY PLAN REPRESENTATION

Our study in Section III shows that query plan representations share the same conceptual basis, which is why we propose a unified query plan representation that is:

1) complete, to include all information of a query plan,
2) general, to support various DBMSs we studied, and,
3) extensible, to support DBMSs we did not study.

A. Design

To define and illustrate the unified query plan representation, we adopted Extend Backus Naur Form (EBNF) [42], which is a metasyntax notation to express context-free grammars. Listing 2 shows the unified query plan representation in EBNF. Following our study in Section III, the unified representation includes the three identified conceptual components of different categories. We define **plan** as a **tree** that can have plan-associated **properties**. Within the **tree**, a **node** is defined as an **operation** and zero or multiple operation-associated **properties**. **operations** and **properties** are key-value pairs including corresponding **categories**, **identifiers**, and **values**.

We use a unified naming convention to denote operations and properties in the unified query plan representation. A

[5]https://dev.mysql.com/doc/workbench/en/wb-performance-explain.html
[6]https://www.mongodb.com/docs/compass/current/query-plan/

Listing 2. The unified query plan representation in EBNF.

```
1  plan ::= ( tree )? properties
2  tree ::= node ( '--children-->' '{' tree (',' tree)*
       '}' )?
3  node ::= operation properties
4  operation ::= 'Operation' ':' operation_category '->'
       operation_identifier
5  properties ::= ( property ( ',' property )* )?
6  property ::= property_category '->' property_identifier
       ':' value
7  operation_category ::= 'Producer' | 'Combinator' |
       'Join' | 'Folder' | 'Executor' | 'Projector' |
       'Consumer'
8  property_category ::= 'Cardinality' | 'Cost' |
       'Configuration' | 'Status'
9  operation_identifier ::= keyword
10 property_identifier ::= keyword
11 keyword ::= letter ( letter | digit | '_' )*
12 value ::= string | number | boolean | 'null'
13 string ::= '"' ( letter | digit )* '"'
14 number ::= '-'? digit+
15 boolean ::= 'true' | 'false'
16 letter ::= [a-zA-Z]
17 digit ::= [0-9]
```

unified naming convention increases the readability and consistency of the representation while avoiding name collisions. Section III shows that various operations and properties share similar semantics, so we mapped DBMS-specific names of operations and properties to unified names. For example, we mapped the operation name **Seq Scan** in PostgreSQL, **Table Scan** in SQL Server, and **TableFullScan** in TiDB to **Full Table Scan**.

B. Analysis

We analyze whether this design achieves the three goals.

Completeness. The unified query plan representation includes the three conceptual components: operations, properties, and formats, which we identified in Section III. Operations and properties are included in the tree of the unified representation, and the unified representation can be serialized into other standard formats, such as JSON and XML, which are used in query plan representations.

Generality. The unified query plan representation supports the query plan representations of the nine DBMSs that we studied. InfluxDB's query plan includes a list of properties without operations, which can be represented by plan-associated properties in the unified representation. For the other DBMSs, we identified operations and properties, mapped them into unified names, and organized them into our unified representation.

Extensibility. The definitions of operations, properties, and categories in the unified query plan representation can be extended or shrunk while keeping forward and backward compatibility. Forward compatibility refers to allowing a system to accept input intended for a later version of itself, and backward compatibility refers to allowing a system to accept input intended for an older version of itself [43]. Both evaluate whether the applications based on the unified query plan representation still work if we update the representation to support more or different DBMSs. To keep forward compatibility, we can add more categories by expanding **operation_category** to include more category names, and add more operations and properties whose names comply with the definition of the keyword at line 11 at Listing 2. The required effort is minimal, as it involves only adding keyword definitions for any new operation or property in a specific DBMS, while the rest of the unified query plan representation remains unaffected. An existing application still can parse the revised representation by ignoring the newly added categories, operations, and properties, or handling them in a generic way (*e.g.*, a visualization tool could represent unknown operations using a generic visual shape). For backward compatibility, similarly, an existing application still can parse the old version of the representation, whose categories, operations, and properties are included in the new version of the representation, so no maintenance effort is required.

An Example of Extensibility. We assume that one of the DBMSs, PostgreSQL, introduces a new operation in query plans to implement Large Language Model (LLM)-based joining [44]. The *UPlan* developers would require minimal effort to accommodate this feature by adding the keyword **LLM Join** for the new operation, without impacting the rest of the unified query plan representation. Similarly, to deprecate this feature, we would remove the keyword in the grammar. These changes would be both forward and backward-compatible to the applications of *UPlan*. Suppose a visualization tool supports *UPlan* to visualize query plans for our studied DBMSs. It is forward-compatible as *UPlan* provides the information of **LLM Join** and the visualization tool can visualize it without any modification. It is also backward-compatible as older versions of *UPlan* would still be supported by the visualization tool, as any older formats would have at most as many operations as the currently-supported format.

V. Applications

We implemented the prototype of a reusable library, *UPlan*, to maintain the unified query plan representation. While our main motivation is facilitating the implementation of automated testing approaches, the unified representation also enables other applications of visualization and benchmarking:

- **A.1 Testing.** The DBMS testing methods *QPG* and *CERT* were implemented in a DBMS-specific way due to DBMS-specific query plan representations, and we show how *UPlan* allows both methods to be implemented in a DBMS-agnostic way.
- **A.2 Visualization.** Visualization tools visually display serialized query plans to ease understanding, but are typically specific to a particular DBMS. We show a general visualization tool based on *UPlan*.
- **A.3 Benchmarking.** Benchmarking is an important method to evaluate the performance of DBMSs. We show a case analysis of a comparison of query plan representations using *UPlan*. We hope that allowing developers to easily compare different DBMSs' query plans enables them to improve their DBMSs' query optimization capabilities.

DBMSs. For A.2 and A.3, we used MongoDB, MySQL, Neo4j, PostgreSQL, and TiDB, because they support the JSON format of query plans. JSON is the most widely supported structural format, and it typically includes more

detailed information than other formats. For example, JSON may include read and optimization costs, but the text format often only includes overall cost, as it is typically optimized for human readability. Within these DBMSs, we used MySQL, PostgreSQL, and TiDB for A.1 as they are supported by *SQLancer*. We used the same versions of DBMSs that we studied in the Table I.

Data set. For A.1, we used *SQLancer* to generate test cases. For A.2 and A.3, we used the benchmarks TPC-H [45] for the five DBMSs, YCSB [46] for MongoDB, and WDBench [47] for Neo4j. TPC-H is one of the most established benchmarks consisting of business-oriented ad-hoc queries and concurrent data modification for evaluating relational DBMSs. It comprises 8 tables and 22 queries for relational DBMSs: MySQL, PostgreSQL, and TiDB in our experiments. For MongoDB and Neo4j, TPC-H requires manual effort to execute as it was not designed for non-SQL languages and non-relational data models. MongoDB is based on a document data model, which lacks support for join operations, so we embedded all entities in one document and rewrote queries 1, 3, and 4 in the MongoDB Query Language (MQL), following a tutorial [48]. For Neo4j, which is based on a graph data model, we mapped nodes into rows of the relational data model and edges into foreign keys of the relational data model following another tutorial [49], and rewrote queries 1–14, 16–19 using the Cypher Query Language (CQL) following an example [50]. WDBench evaluates graph DBMSs and consists of 2623 queries. YCSB is used for NoSQL DBMSs and dynamically generates workloads including updates and queries. We used both benchmarks to further explore the query plans in MongoDB and Neo4j.

Implementation. *UPlan* is a reusable library that consists of around 300 lines of Python code to implement a reusable library, which allows adding or updating operations and properties. The prototype supports serializing query plan representations into JSON as well as text formats, and provides an interface for supporting more formats. We also implemented five customized converters to parse the query plan representations from existing query plan representations to the unified query plan representation, and each parser has around 200 lines of code.

A.1 Testing

We show how *UPlan* allows implementing the testing methods *QPG* [4] and *CERT* [5] in a DBMS-agnostic way. *QPG* is a test case generation approach that is guided by query plans; specifically, it mutates a database if no new query plans have been observed for a specific number of randomly generated queries, aiming to subsequently exercise new query plans, and thus exploring "interesting" behaviors. In terms of implementation, evaluating whether a query plan is structurally different from another requires ignoring unstable information, such as random identifiers and the estimated cost in query plans. *CERT* is a test oracle for finding performance issues by comparing estimated cardinalities, which have to be extracted from query plans. Both methods were implemented

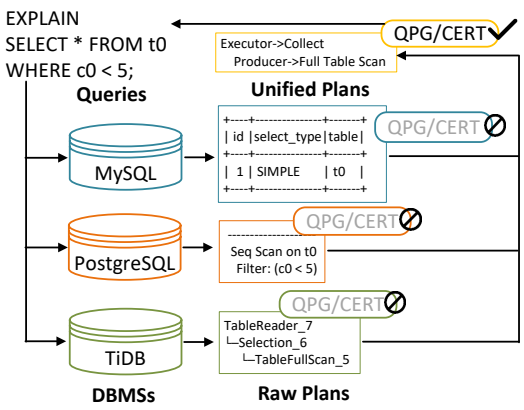

Fig. 2. The architecture of using *UPlan* for *QPG* and *CERT*. Grey font means the implementation without *UPlan*, and ✓ refers to the implementation with *UPlan*. *QPG*/*CERT* stops when observing an unexpected result or query plan.

in *SQLancer*, which is a popular and widely used tool for automatically testing DBMSs. The original implementations used DBMS-specific and error-prone methods, such as string matching and substitution, which had to be implemented for every DBMS that was supported. Both approaches support relational DBMSs, but were not applied to the popular open-source DBMSs MySQL and PostgreSQL, as additional DBMS-specific parsers would have been required. Based on *UPlan*, we implemented general parsers for both methods to support all *UPlan*-compatible DBMSs. We did not consider *Mozi*, because its source code is not public.

Figure 2 shows the architecture of using *UPlan* for *QPG* and *CERT*. Both *QPG* and *CERT* iteratively execute queries in target DBMSs and retrieve query plans for generating the next query and examining cardinalities respectively. In this example, suppose both *QPG* and *CERT* execute the query `EXPLAIN SELECT * FROM t0 WHERE t0<5` in MySQL, PostgreSQL, and TiDB, the query plans returned by three DBMSs are significantly different. Without *UPlan*, *QPG* and *CERT* require DBMS-specific implementation to parse the raw query plans. With *UPlan*, we converted raw query plans into the unified query plan, and *QPG* and *CERT* require only a single implementation based on *UPlan*. In this example, TiDB's plan is converted into two operations, while PostgreSQL's and MySQL's plans are converted into `Producer->Full Table Scan` only as TiDB requires `Executor->Collect` to receive data from other nodes in a distributed system.

To evaluate *UPlan* on *QPG* and *CERT*, we applied the general parser to MySQL, PostgreSQL, and TiDB. We ran our revised versions of *QPG* and *CERT* for 24 hours and found 17 unique and previously unknown bugs, as shown in Table V. Developers confirmed 16 of the 17 bugs and fixed two bugs. The one bug in PostgreSQL was still waiting for the response from developers, so we did not report more bug reports to avoid burdening developers. 11 of 17 bugs are *Critical*, *Serious*, or *Major*, which represent that the bugs can significantly affect the systems. Additionally, *CERT* found hundreds of potential bug-inducing test cases in 24 hours, but it is challenging to distinguish their uniqueness, which requires

TABLE V
PREVIOUSLY UNKNOWN AND UNIQUE BUGS FOUND BY *QPG* WITH *UPlan*.

DBMS	Found by	Bug ID	Status	Severity
MySQL	*QPG*	113302	Confirmed	Critical
MySQL	*QPG*	113304	Confirmed	Critical
MySQL	*QPG*	113317	Confirmed	Critical
MySQL	*QPG*	114204	Confirmed	Serious
MySQL	*QPG*	114217	Confirmed	Serious
MySQL	*QPG*	114218	Confirmed	Serious
MySQL	*CERT*	114237	Confirmed	Performance
PostgreSQL	*CERT*	Email	Pending	Performance
TiDB	*QPG*	49107	Fixed	Major
TiDB	*QPG*	49108	Confirmed	Major
TiDB	*QPG*	49109	Fixed	Major
TiDB	*QPG*	49110	Confirmed	Major
TiDB	*QPG*	49131	Confirmed	Major
TiDB	*QPG*	51490	Confirmed	Moderate
TiDB	*QPG*	51523	Confirmed	Moderate
TiDB	*CERT*	51524	Confirmed	Minor
TiDB	*CERT*	51525	Confirmed	Minor

Listing 3. A bug found by *QPG* with *UPlan*.

```
1  CREATE TABLE t0(c0 INT, c1 INT);
2  INSERT INTO t0(c1, c0) VALUES(0, 1);
3
4  SELECT * FROM t0 WHERE t0.c1 IN (GREATEST(0.1, 0.2));
       -- empty result ✔
5  CREATE INDEX i0 ON t0(c1);
6  SELECT * FROM t0 WHERE t0.c1 IN (GREATEST(0.1, 0.2));
       -- {1|0} 🐞
```

developers' expertise. Listing 3 shows a bug in MySQL found by the test case generated by *QPG* with *UPlan*. Note that we identified this bug by the test oracle Ternary Logic Partitioning (TLP) [51]; however, for presentation, we simplified the bug-inducing test case by demonstrating that the same query returns different results in lines 4 and 6 depending on whether the index exists. The cause of the bug was an incorrect table look-up due to the index inserted by the SQL statement in line 5. Using *UPlan*, we were able to apply *QPG* to MySQL easily, which was previously incompatible and untested by *QPG*. This enabled us to find this bug in MySQL.

We also found that *UPlan* reduces the risk of implementation bugs for DBMS-specific query plan parsers. We identified an implementation bug in *QPG*. Specifically, the query plan parser for TiDB failed to exclude random identifiers due to an incorrect parameter for `EXPLAIN`. With the single implementation for a parser and the unified query plan representation, we have a lower risk of introducing these implementation bugs.

> *UPlan* enables large-scale adoption for testing methods *QPG* and *CERT* in a DBMS-agnostic implementation way.

A.2 Visualization

We implemented a visualization tool for serialized query plans by modifying PEV2 [35], a customized query plan visualization tool for PostgreSQL, to use the unified query plan representation. We modified its parser to support identifying the unified query plan representation, and updated its definitions of visualized elements, such as operation names.

UPlan significantly reduces the effort to build visualization tools on query plans. According to the Git repository, develop-

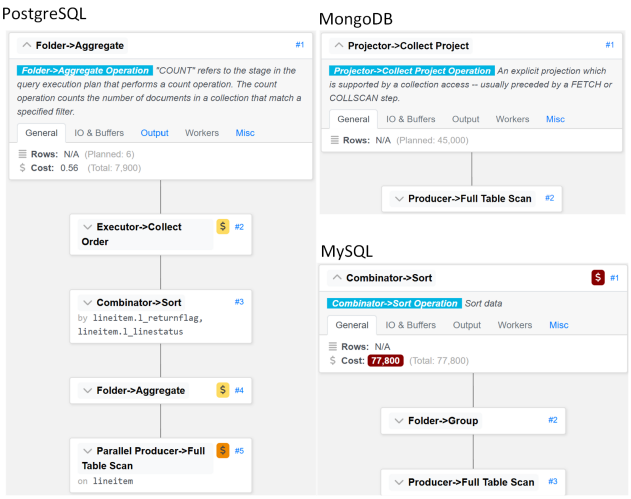

Fig. 3. Visualized unified query plans of query 1 from TPC-H benchmark.

ers of PEV2 committed 24,559 lines of code within the 188 days between the initial commit and the first release. As a naive, but reasonable approximation, we assumed this period reflects the minimal required effort to build a visualization tool. Thus, implementing separate visualization tools for the studied DBMSs would require around $188 * 5 = 940$ days. Using *UPlan*, we modified only around 800 lines of code of PEV2 to support all DBMSs. For PEV2, the average development speed is $24,559/188 \approx 130$ lines of code per day, so the estimated effort for adapting *UPlan* is $800/130 \approx 6$ days. Compared to implementing DBMS-specific visualization tools, which would require 940 days, *UPlan* would take only 194 days $(188 + 6)$ and would reduce the time effort by approximately 80%. Furthermore, the percentage of effort reduction would increase as the number of supported DBMSs grows. Note that this approximation ignores many factors, such as the complexity of query plans and the time to understand visualization tools.

Figure 3 shows examples of visualized unified query plans in PostgreSQL, MongoDB, and MySQL. These query plans stem from query 1 of the TPC-H benchmark. An operation and its associated properties is depicted as rectangle, representing a node. For example, in the first node of MySQL query plan, **Combinator→Sort** represents the operation **Sort**, which belongs to the category **Combinator**. The following is the description and properties. Our unified query plan representations enable a general visualization tool that can visualize query plans from any DBMS that exposes query plans.

> Existing DBMS-specific visualization tools could support more DBMSs if they supported our unified query plan representation.

A.3 Benchmarking

In this application, we show how to find potential optimization opportunities by comparing the serialized query plans across DBMSs using the unified query plan representation. Query optimization is a critical process for DBMSs' perfor-

TABLE VI
AVERAGE NUMBER OF OPERATIONS IN QUERY PLANS FROM TPC-H.

DBMS	Prod.	Comb.	Join	Folder	Proj.	Exec.	Sum
MongoDB	1.00	0.00	0.00	0.00	1.00	0.00	2.00
MySQL	4.55	0.82	2.77	0.86	0.00	0.27	9.27
Neo4j	0.39	0.78	2.89	0.06	0.72	3.06	7.89
PostgreSQL	3.95	1.32	2.64	1.73	0.00	2.45	12.09
TiDB	4.18	0.82	2.73	1.41	1.77	3.73	14.64

TABLE VII
AVERAGE NUMBER OF OPERATIONS IN QUERY PLANS FROM YCSB FOR MONGODB AND WDBENCH FOR NEO4J.

DBMS	Prod.	Comb.	Join	Folder	Proj.	Exec.	Sum
MongoDB	1.00	0.00	0.00	0.00	0.00	0.00	1.00
Neo4j	0.45	0.00	3.73	0.00	0.37	1.44	5.99

mance. To evaluate how effective a query optimization is, existing methods depend on measuring DBMSs' execution time on standard datasets, such as TPC-H [45] and the Join Order Benchmark (JOB) [52]. The execution time shows the overall performance difference between various DBMSs, but cannot provide possible reasons for performance gaps across DBMSs. *UPlan* enables comparing various serialized query plans across DBMSs. Specifically, we collected metrics on the number of operations in DBMSs' query plan representations and show an example of analyzing the difference.

Table VI shows the average number of operations in each category for the query plans of queries from the TPC-H benchmark. We omitted the *Consumer* category as we did not encounter any such operations. Overall, the relational DBMSs, MySQL, PostgreSQL, and TiDB, have more operations than the non-relational DBMSs, MongoDB and Neo4j. It is because relational DBMSs have more operations in the *Producer* category. To further explain the difference in the *Producer* category, we looked into query plans and found that each table in relational DBMSs requires at least one operation to read data, while non-relational DBMSs usually read all data in one or two operations. Apart from MongoDB, the other four DBMSs have a similar number of *Join* operations. The results also show that the query plans of the queries in the TPC-H benchmark suite usually do not cover all categories of operations due to the limited set of queries.

Similarly, Table VII shows the average number of operations for the query plans from the YCSB and WDBench benchmarks respectively. Both benchmarks are designed specifically for NoSQL and graph DBMSs, but mainly consider input diversity instead of internal execution diversity, as query plans are DBMS-specific and challenging to examine. As a result, compared to TPC-H, YCSB and WDBench have a similar distribution of operations. For some categories, the query plans in YCSB and WDBench include even fewer operations than those in TPC-H. For example, YCSB does not expose any operation in the *Projection* category and WDBench does not expose any operation in the *Combinator* and *Folder* categories. This shows that the TPC-H benchmark can efficiently expose operations in query plans.

To find potential optimization opportunities, we analyzed

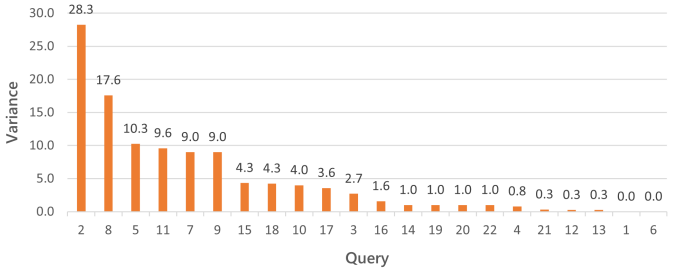

Fig. 4. Variance of the number of Producer operations for each query in TPC-H benchmark across five DBMSs.

Listing 4. The unified serialized query plan representations in the text format for query 11 in TPC-H. Underlines represent table names. Predicates in the query and properties in the query plan representations are ignored for brevity.

```
SELECT ... FROM PARTSUPP, SUPPLIER, NATION WHERE ...
HAVING ... > (SELECT ... FROM PARTSUPP, SUPPLIER,
     NATION WHERE ...) ...;
-------------------------------------------------------
PostgreSQL:                TiDB:
Combinator->Sort           Projector->Project
 Folder->Aggregate          Combinator->Sort
  Join->Hash Join            Folder->Aggregate Hash
   Producer->Full Table       Projector->Project
   name object: partsupp       Join->Index Hash
   Executor->Hash Row         Join->Index Hash
    Join->Hash                 Executor->Collect
     Producer->Full Table       Producer->Full Table
     name object: supplier        name object: nation
     Executor->Hash Row         Executor->Collect Order
      Producer->Full Table       Producer->Index-only...
      name object: nation          name object: supplier
 Folder->Aggregate             Executor->Collect Order
  Join->Hash Join               Producer->Index-only...
   Producer->Full Table         name object: partsupp
   name object: partsupp          Producer->Id Scan
   Executor->Hash Row             name object: partsupp
    Join->Hash Join
     Producer->Full Table
     name object: supplier
     Executor->Hash Row
      Producer->Full Table
      name object: nation
```

the operations of the *Producer*. Across the five DBMSs, the operations of the *Producer* category are typically expensive for performance, and hence, developers typically aim to analyze and reduce the occurrence of the operations in this category. Figure 4 shows the variance of the number of operations in the *Producer* category for each query plan. Among 22 queries, the variances of six queries are more than 5, indicating a significant difference. Queries 2, 8, 5, 7, and 9 have a significant variance due to different data models. For example, for query 2, the DBMSs of relational data models, MySQL, TiDB, and PostgreSQL, have 10, 12, and 9 operations, while the DBMS based on the graph data model, Neo4j, has only 1 operation in the *Producer* category. Query 11 has a significant variance due to a potential optimization issue, and we explain it as follows.

Listing 4 shows query 11 from the TPC-H benchmark suite and the corresponding serialized query plans of PostgreSQL and TiDB using the unified query plan representations in text format. The query references the three tables `PARTSUPP`, `SUPPLIER`, `NATION` twice in the `FROM` and `HAVING` clauses respectively. PostgreSQL uses six table scans, one for each table reference in the original query, while TiDB could optimize the

query to use only three scans. The table `partsupp` is scanned twice in lines 19 and 21, because TiDB reduces the data size of table scanning by retrieving a secondary index before the scan operation. The first scan retrieves indexes only to obtain the row id, which is used for the second scan. Suppose both DBMSs execute the operations in order and the same operation has the same performance overhead in both DBMSs, then the query plan with three table scans is more efficient than the query plan with six table scans.

We quantitatively evaluated the potential performance improvement contributed by our analysis. It is challenging to modify PostgreSQL source code to more effectively optimize this query plan. To estimate the performance improvement, we evaluated the actual execution time for the six table scans in PostgreSQL's query plan using the SQL prefix **EXPLAIN ANALYZE**, which executes the query and returns the execution information of the query plan including the execution time for each operation. For 10 GB data of TPC-H benchmark, the whole query consumes 1503ms, and the six table scans consume 214ms, 5ms, 1ms, 216ms, 2ms, and 189ms, respectively. If PostgreSQL optimizes this query plan to remove the last three table scans, the execution time would be significantly reduced by 407ms ($216+2+189$), which accounts for 27% of the overall execution time of the query. Note that we assumed that eliminating the three operations would not influence the execution time of other operators. *UPlan* enables this analysis and provides actionable insight for DBMS developers to improve query performance by reducing three repeated table scans. We reported this issue to the PostgreSQL developers, who confirmed that this case indicates an unsupported optimization by PostgreSQL. One developer considered how this could be supported—"*[...] maybe there is some easy way to hook this into the same code used by GROUPING SETS [...]*".

Comparing query plans between DBMSs, using the unified query plan representation, provides insights for improving query optimizers.

VI. Discussion

Paths to adoption. Developers can use the unified query plan representation by customized converters to convert original serialized query plans into the unified query plan representation. For all three applications we showed, we implemented five customized converters, each of which has around 200 lines of code, within one week only. If the converters can be implemented by DBMS developers or experts, who are well-versed in the query plans, it is plausible that a higher-quality converter could be developed in a shorter time. We hope that DBMSs will directly expose query plans in the unified representation in the long term, thus avoiding a conversion.

Additional use cases. We envision several additional use cases that are enabled by our unified query plan representations. Toward a comprehensive evaluation of query optimization, additional metrics could be explored, such as similarity on tree structures [53], to compare different DBMSs' query plans using our unified query plan representation. To find issues in query optimization, we can apply differential testing [54] or other methods to compare the unified query plan representations among DBMSs. Query optimization approaches based on machine learning have been proposed that take query plans as input and output suggestions for indexes [55], views [56], and join orders [57], [58], so our unified query plan representation would allow exchanging training data in different DBMSs to improve the performance of models.

Substrait. *UPlan* complements Substrait [59], which provides a DBMS-agnostic specification for logical query plan representation to enable cross-language serialization for relational algebra. First, Substrait mainly focuses on logical query plans, while *UPlan* focuses on physical query plans. Many physical operations are DBMS-specific and are not considered by Substrait. In Section III, our study based on real-world query plans shows that the physical operations of *Executor* and *Consumer* do not have corresponding operators in relational algebra, and Substrait does not support them. Integrating *UPlan* as a backend component for Substrait would allow it to analyze, optimize, and test DBMS-specific query executions comprehensively. Second, *UPlan* enables different application domains compared to Substrait. *UPlan* aims to reduce the effort of building applications on query plans instead of cross-language serialization for relational algebra. The unified logical query plans, provided by Substrait, cannot facilitate building general applications based on physical query plans such as demonstrated in this work. Lastly, *UPlan* offers easier integration with real-world DBMSs. Unlike Substrait, which requires additional support for DBMSs to both output logical query plans in Substrait format and execute those plans, *UPlan* directly parses existing query plans without requiring any modifications to the DBMS. This significantly reduces the implementation effort typically needed by Substrait.

Completeness. We believe our study and implementation are sufficiently complete as we studied query plans from various sources. As we observed in our study, DBMSs lack detailed and standard documents for query plans, so we comprehensively studied query plans from documents, source code, and concrete executions with the workloads from TPC-H, WDBench, and YCSB benchmarks. We also evaluated our implementation of *UPlan* on visualization and benchmarking with three benchmarks. However, the possibility of missing features in query plans still exists. For example, some properties may be dynamically generated without detailed documentation. To observe them, we require specific workloads, which we may overlooked. To alleviate this potential issue, we designed an extensible *UPlan*. Such missing properties can be easily integrated into the unified query plan representation by adding keyword definitions as we explained in Section IV.

Threats to validity. Our study faces several threats to validity, which denotes the trustworthiness [60], [12] of the results, and to what extent they are unbiased. A major concern is the degree to which the data and analysis depend on the specific researchers. We followed the best practice of triangulation [12], which refers to taking multiple perspectives toward

the same object, to increase reliability. For data triangulation, we collected data from multiple sources: documentation, source code, and third-party applications. For observer triangulation, one author conducted the study, and another author validated the findings against the raw data. For methodological triangulation, we used a qualitative method to analyze query plan representations, and a quantitative method to examine and classify the three conceptual components of query plan representations. Furthermore, we have made the process of data collection and analysis publicly available, along with comprehensive study results presented in the supplementary materials of this paper. Another concern is the degree to which our results can be generalized to and across the query plan representations of other DBMSs. We selected representative DBMSs of various data models: relational, document, graph, and time series. The last concern is the degree to which the study really assesses the research questions we aim at. We collected the data ourselves, and our analysis of the semantics may be inconsistent with the intentions of the developers that implemented the query plans.

VII. Related Work

Improving query optimization. Query optimization is a long-standing challenge and a core database research topic with a significant body of related work. We discuss several representative works here. Ioannidis *et al*. [32] proposed to sample data in single columns for estimating the cardinality of a query plan. Ilyas *et al*. [61] proposed to detect correlations across columns, which enables a multi-column sampling [62] for a more accurate estimation of cardinality. Lakshmi *et al*. [63] proposed using machine learning to estimate cardinality. Estimated cardinality, as well as other estimated costs, are considered in cost models. Babcock *et al*. [64] proposed cost distribution to choose a query plan with the lowest cost in a robust way. Apache published a unified framework Calcite [65] to replace existing query optimizers. Compared to them, we studied various query plan representations, rather than improving query optimization.

Benchmark and empirical study of query optimization. Various papers studied query optimizations to understand their performance through empirical studies. Leis *et al*. [52] experimentally studied the contributions of query optimization components to overall performance. Ortiz *et al*. [66] provided an empirical analysis on the accuracy, space, and time trade-off across several machine learning algorithms for cardinality estimation specifically. Harmouch *et al*. [67] conducted an experimental survey on cardinality estimation, focusing on estimating the number of distinct values. While we also present an empirical study in the context of DBMSs, we focus on query plan representations, rather than the performance, algorithms, and internal components of query optimization.

Applications based on serialized query plans. Several applications based on serialized query plans exist. QE3D [68] visualizes distributed serialized query plans for an intuitive understanding and analysis. Machine learning algorithms have utilized serialized query plans for query optimization. Yuan *et al*. [56] used machine learning to select optimal views. Yu *et al*. [58] used reinforcement learning to determine the join order. Marcus *et al*. [69] and Ryan *et al*. [70] applied machine learning algorithms to generate query plans. Zhao *et al*. [71] used machine learning to convert serialized query plan representations to vector representations to facilitate other machine learning algorithms. In this work, we propose a unified query plan representation, which reduces the effort to build the applications based on serialized query plans.

Standardization. Multiple works were proposed to standardize DBMSs. The most related work is Substrait [59], which proposes a unified representation for DBMS-agnostic relational algebra, while we studied DBMS-specific query plans. Feng *et al*. [72] proposed a unified architecture to reduce the implementation effort for in-RDBMS analytics. Mitschang [73] proposed a unified view of design data and knowledge representation when supporting database systems to non-standard applications, such as Computer-Aided Design (CAD). Ginsburg *et al*. [74] proposed a unified approach to query sequenced data. Gueidi *et al*. [75] proposed a unified modeling method for Non-relational DBMSs (NoSQL) to facilitate the applications on NoSQL. Compared to these works, we propose a unified framework for the query plan representations in DBMSs.

VIII. Conclusion

We have presented an exploratory case study to investigate how query plan representations are in nine widely-used DBMSs. Our study has shown that query plan representations share conceptual components among different DBMSs: operations, properties, and formats. Based on the study, we designed the unified query plan representation to reduce the effort to build applications based on query plans. We implemented a reusable library *UPlan* to maintain the unified representation, and evaluated it on five DBMSs. The results show that existing testing methods can be efficiently adopted, finding 17 previously unknown and unique bugs. Additionally, existing DBMS-specific visualization tools could support at least five DBMSs by using *UPlan* with only moderate implementation effort, and *UPlan* enables comparing query plans in different DBMSs, which provides actionable insights. This paper provides a comprehensive study of query plan representations, and can be used as a reference for other research on query plans. We believe the unified query plan representation provides more opportunities to research serialized query plans in the future.

Acknowledgments

This research is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and Cyber Security Agency of Singapore.

References

[1] Website, "Most widely deployed and used database engine," https://www.sqlite.org/mostdeployed.html, 2022, accessed: 2022-06-08.

[2] ——, "Tidb customers," https://en.pingcap.com/customers, 2022, accessed: 2022-06-08.

[3] ——, "Cockroachdb customers," https://www.cockroachlabs.com/customers, 2022, accessed: 2022-06-08.

[4] J. Ba and M. Rigger, "Testing database engines via query plan guidance," in *The 45th International Conference on Software Engineering (ICSE'23)*, May 2023.

[5] ——, "Finding performance issues in database engines via cardinality estimation testing," in *The 46th International Conference on Software Engineering (ICSE'24)*, Apr. 2024.

[6] J. Liang, Z. Wu, J. Fu, M. Wang, C. Sun, and Y. Jiang, "Mozi: Discovering dbms bugs via configuration-based equivalent transformation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[7] P. Guagliardo and L. Libkin, "How standard is the SQL standard?" in *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018*, ser. CEUR Workshop Proceedings, D. Olteanu and B. Poblete, Eds., vol. 2100. CEUR-WS.org, 2018. [Online]. Available: https://ceur-ws.org/Vol-2100/paper16.pdf

[8] ——, "A formal semantics of SQL queries, its validation, and applications," *Proc. VLDB Endow.*, vol. 11, no. 1, pp. 27–39, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol11/p27-guagliardo.pdf

[9] Website. (2023) Iso/iec 9075-1:2023 information technology — database languages sql. [Online]. Available: https://www.iso.org/standard/76583.html

[10] F. Bousquet, R. Lifran, M. Tidball, S. Thoyer, and M. Antona, "Editorial introduction," *J. Artif. Soc. Soc. Simul.*, vol. 4, no. 2, 2001. [Online]. Available: http://jasss.soc.surrey.ac.uk/4/2/0.html

[11] D. Guliato, E. V. de Melo, R. M. Rangayyan, and R. C. Soares, "POSTGRESQL-IE: an image-handling extension for postgresql," *J. Digit. Imaging*, vol. 22, no. 2, pp. 149–165, 2009. [Online]. Available: https://doi.org/10.1007/s10278-007-9097-5

[12] P. Runeson, M. Höst, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering - Guidelines and Examples*. Wiley, 2012. [Online]. Available: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-1118104358.html

[13] E. Mella, M. A. Rodríguez, L. Bravo, and D. Gatica, "Query rewriting for semantic query optimization in spatial databases," *GeoInformatica*, vol. 23, no. 1, pp. 79–104, 2019. [Online]. Available: https://doi.org/10.1007/s10707-018-00335-w

[14] T. Neumann, "Efficient generation and execution of dag-structured query graphs," Ph.D. dissertation, University of Mannheim, Germany, 2005. [Online]. Available: http://bibserv7.bib.uni-mannheim.de/madoc/volltexte/2005/1089/index.html

[15] P. van Rosmalen, E. A. Boyle, J. van der Baaren, A. I. Kärki, and Á. del Blanco Aguado, "A case study on the design and development of minigames for research methods and statistics," *EAI Endorsed Trans. Serious Games*, vol. 1, no. 3, p. e5, 2014. [Online]. Available: https://doi.org/10.4108/sg.1.3.e5

[16] T. J. Anih, C. A. Bede, and C. F. Umeokpala, "Detection of anomalies in a time series data using influxdb and python," *CoRR*, vol. abs/2012.08439, 2020. [Online]. Available: https://arxiv.org/abs/2012.08439

[17] H. Ouyang, H. Wei, H. Li, A. Pan, and Y. Huang, "Checking causal consistency of mongodb," *J. Comput. Sci. Technol.*, vol. 37, no. 1, pp. 128–146, 2022. [Online]. Available: https://doi.org/10.1007/s11390-021-1662-8

[18] M. Egea, C. Dania, and M. Clavel, "Mysql4ocl: A stored procedure-based mysql code generator for OCL," *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, vol. 36, 2010. [Online]. Available: https://doi.org/10.14279/tuj.eceasst.36.445

[19] K. Rabuzin, M. Cerjan, and S. Krizanic, "Supporting data types in neo4j," in *New Trends in Database and Information Systems - ADBIS 2022 Short Papers, Doctoral Consortium and Workshops: DOING, K-GALS, MADEISD, MegaData, SWODCH, Turin, Italy, September 5-8, 2022, Proceedings*, ser. Communications in Computer and Information Science, S. Chiusano, T. Cerquitelli, R. Wrembel, K. Nørvåg, B. Catania, G. Vargas-Solar, and E. Zumpano, Eds., vol. 1652. Springer, 2022, pp. 459–466. [Online]. Available: https://doi.org/10.1007/978-3-031-15743-1_42

[20] Website. (1989) Microsoft sql server. [Online]. Available: https://www.microsoft.com/en-us/sql-server/

[21] D. Pawlaszczyk, "Sqlite," in *Mobile Forensics - The File Format Handbook - Common File Formats and File Systems Used in Mobile Devices*, C. Hummert and D. Pawlaszczyk, Eds. Springer, 2022, pp. 129–155. [Online]. Available: https://doi.org/10.1007/978-3-030-98467-0_5

[22] Q. Meng, X. Ma, W. Lu, and Z. Yao, "A spatial SQL based on sparksql," in *Geo-Spatial Knowledge and Intelligence - 4th International Conference on Geo-Informatics in Resource Management and Sustainable Ecosystem, GRMSE 2016, Hong Kong, China, November 18-20, 2016, Revised Selected Papers, Part I*, ser. Communications in Computer and Information Science, H. Yuan, J. Geng, and F. Bian, Eds., vol. 698. Springer, 2016, pp. 437–443. [Online]. Available: https://doi.org/10.1007/978-981-10-3966-9_50

[23] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970. [Online]. Available: https://doi.org/10.1145/362384.362685

[24] C. Strauch, U.-L. S. Sites, and W. Kriha, "Nosql databases," *Lecture Notes, Stuttgart Media University*, vol. 20, no. 24, p. 79, 2011.

[25] A. Pavlo and M. Aslett, "What's really new with newsql?" *SIGMOD Rec.*, vol. 45, no. 2, pp. 45–55, 2016. [Online]. Available: https://doi.org/10.1145/3003665.3003674

[26] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[27] Website. (2024) Query plan set node properties from map. [Online]. Available: https://neo4j.com/docs/cypher-manual/current/execution-plans/operators#query-plan-set-node-properties-from-map/

[28] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, B. Yormark, Ed. ACM Press, 1984, pp. 47–57. [Online]. Available: https://doi.org/10.1145/602259.602266

[29] T. J. Lehman and M. J. Carey, "A study of index structures for main memory database management systems," in *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, Eds. Morgan Kaufmann, 1986, pp. 294–303. [Online]. Available: http://www.vldb.org/conf/1986/P294.PDF

[30] A. Grove, *How Query Engines Work*. Leanpub, 2022. [Online]. Available: https://leanpub.com/how-query-engines-work

[31] Website. (2024) Execution plans in neo4j. [Online]. Available: https://neo4j.com/docs/cypher-manual/5/execution-plans/

[32] Y. E. Ioannidis, "The history of histograms (abridged)," in *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, Eds. Morgan Kaufmann, 2003, pp. 19–30. [Online]. Available: http://www.vldb.org/conf/2003/papers/S02P01.pdf

[33] E. International, "Ecma-404—the json data interchange format," 2013.

[34] J. Bosak, "Xml, java, and the future of the web," *World Wide Web J.*, vol. 2, no. 4, pp. 219–227, 1997. [Online]. Available: http://metalab.unc.edu/pub/sun-info/standards/xml/why/xmlapps.htm

[35] Website. (2023) Postgres explain visualizer. [Online]. Available: https://explain.dalibo.com/

[36] ——. (2023) pgmustard. [Online]. Available: https://www.pgmustard.com/

[37] ——. (2023) pganalyze. [Online]. Available: https://pganalyze.com/

[38] ——. (2023) Apexsql. [Online]. Available: https://www.apexsql.com/products/sql-tools-bundle-fundamentals/

[39] A. Orthey, B. Frész, and M. Toussaint, "Motion planning explorer: Visualizing local minima using a local-minima tree," *IEEE Robotics Autom. Lett.*, vol. 5, no. 2, pp. 346–353, 2020. [Online]. Available: https://doi.org/10.1109/LRA.2019.2958524

[40] Website. (2023) Azure data studio. [Online]. Available: https://learn.microsoft.com/en-us/sql/azure-data-studio/query-plan-viewer?view=sql-server-ver16

[41] ——. (2023) Dbvisualizer. [Online]. Available: https://www.dbvis.com/

[42] R. E. Pattis, "Teaching EBNF first in CS 1," in *Proceedings of the 25th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 1994, Phoenix, Arizona, USA, March 10-12, 1994*, R. Beck and D. Goelman, Eds. ACM, 1994, pp. 300–303. [Online]. Available: https://doi.org/10.1145/191029.191155

[43] W. Wallace, "Review of "designing with web standards (second edition) by jeffrey zeldman", peachpit press, 2006, ISBN 0321385551," *ACM Queue*, vol. 5, no. 4, p. 56, 2007. [Online]. Available: https://doi.org/10.1145/1255421.1255432

[44] M. Urban and C. Binnig, "Efficient learned query execution over text and tables [technical report]," *arXiv preprint arXiv:2410.22522*, 2024.

[45] Website. (2023) Tpc-h benchmark. [Online]. Available: https://www.tpc.org/tpch/

[46] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[47] R. Angles, C. B. Aranda, A. Hogan, C. Rojas, and D. Vrgoč, "Wdbench: A wikidata graph query benchmark," in *International Semantic Web Conference*. Springer, 2022, pp. 714–731.

[48] Website. (2024) Modelling tpc-h for mongodb. [Online]. Available: https://alronz.github.io/Factors-Influencing-NoSQL-Adoption/site/MongoDB/Examples/DenormalisedModel/

[49] ——. (2024) Modelling tpc-h for neo4j. [Online]. Available: https://alronz.github.io/Factors-Influencing-NoSQL-Adoption/site/Neo4j/Examples/TPC-HQueries/

[50] ——. (2024) Cql queries of tpc-h benchmark suite. [Online]. Available: https://github.com/aiquis/tpch-neo4j/blob/5e4e5c/tpch_queries.cql

[51] M. Rigger and Z. Su, "Finding bugs in database systems via query partitioning," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 211:1–211:30, 2020. [Online]. Available: https://doi.org/10.1145/3428279

[52] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proc. VLDB Endow.*, vol. 9, no. 3, pp. 204–215, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol9/p204-leis.pdf

[53] R. Yang, P. Kalnis, and A. K. H. Tung, "Similarity evaluation on tree-structured data," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, F. Özcan, Ed. ACM, 2005, pp. 754–765. [Online]. Available: https://doi.org/10.1145/1066157.1066243

[54] W. M. McKeeman, "Differential testing for software," *Digit. Tech. J.*, vol. 10, no. 1, pp. 100–107, 1998. [Online]. Available: https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[55] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya, "AI meets AI: leveraging query executions to improve index recommendations," in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds. ACM, 2019, pp. 1241–1258. [Online]. Available: https://doi.org/10.1145/3299869.3324957

[56] H. Yuan, G. Li, L. Feng, J. Sun, and Y. Han, "Automatic view generation with deep learning and reinforcement learning," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 1501–1512. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00133

[57] R. Marcus and O. Papaemmanouil, "Deep reinforcement learning for join order enumeration," in *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, R. Bordawekar and O. Shmueli, Eds. ACM, 2018, pp. 3:1–3:4. [Online]. Available: https://doi.org/10.1145/3211954.3211957

[58] X. Yu, G. Li, C. Chai, and N. Tang, "Reinforcement learning with tree-lstm for join order selection," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 1297–1308. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00116

[59] substrait io, "Substrait: Cross-Language Serialization for Relational Algebra," Sep. 2021. [Online]. Available: https://github.com/substrait-io/substrait

[60] D. Neuman, "Evaluating evolution: Naturalistic inquiry and the perseus project," *Comput. Humanit.*, vol. 25, no. 4, pp. 239–246, 1991. [Online]. Available: https://doi.org/10.1007/BF00116078

[61] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga, "CORDS: automatic discovery of correlations and soft functional dependencies," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, G. Weikum, A. C. König, and S. Deßloch, Eds. ACM, 2004, pp. 647–658. [Online]. Available: https://doi.org/10.1145/1007568.1007641

[62] V. Poosala and Y. E. Ioannidis, "Selectivity estimation without the attribute value independence assumption," in *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, Eds. Morgan Kaufmann, 1997, pp. 486–495. [Online]. Available: http://www.vldb.org/conf/1997/P486.PDF

[63] M. S. Lakshmi and S. Zhou, "Selectivity estimation in extensible databases - A neural network approach," in *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, A. Gupta, O. Shmueli, and J. Widom, Eds. Morgan Kaufmann, 1998, pp. 623–627. [Online]. Available: http://www.vldb.org/conf/1998/p623.pdf

[64] B. Babcock and S. Chaudhuri, "Towards a robust query optimizer: A principled and practical approach," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, F. Özcan, Ed. ACM, 2005, pp. 119–130. [Online]. Available: https://doi.org/10.1145/1066157.1066172

[65] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire, "Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds. ACM, 2018, pp. 221–230. [Online]. Available: https://doi.org/10.1145/3183713.3190662

[66] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi, "An empirical analysis of deep learning for cardinality estimation," *CoRR*, vol. abs/1905.06425, 2019. [Online]. Available: http://arxiv.org/abs/1905.06425

[67] H. Harmouch and F. Naumann, "Cardinality estimation: An experimental survey," *Proc. VLDB Endow.*, vol. 11, no. 4, pp. 499–512, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol11/p499-harmouch.pdf

[68] D. Scheibli, C. Dinse, and A. Boehm, "QE3D: interactive visualization and exploration of complex, distributed query plans," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds. ACM, 2015, pp. 877–881. [Online]. Available: https://doi.org/10.1145/2723372.2735364

[69] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," *SIGMOD Rec.*, vol. 51, no. 1, pp. 6–13, 2022. [Online]. Available: https://doi.org/10.1145/3542700.3542703

[70] R. C. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A learned query optimizer," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1705–1718, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p1705-marcus.pdf

[71] Y. Zhao, G. Cong, J. Shi, and C. Miao, "Queryformer: A tree transformer model for query plan representation," *Proc. VLDB Endow.*, vol. 15, no. 8, pp. 1658–1670, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p1658-zhao.pdf

[72] X. Feng, A. Kumar, B. Recht, and C. Ré, "Towards a unified architecture for in-rdbms analytics," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, Eds. ACM, 2012, pp. 325–336. [Online]. Available: https://doi.org/10.1145/2213836.2213874

[73] B. Mitschang, "Towards a unified view of design data and knowledge representation," in *Expert Database Systems, Proceedings from the Second International Conference, Vienna, Virginia, USA, April 25-27, 1988*, L. Kerschberg, Ed. Benjamin/Cummings, 1988, pp. 133–159.

[74] S. Ginsburg and X. Wang, "Pattern matching by rs-operations: Towards a unified approach to querying sequenced data," in *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1992, pp. 293–300.

[75] A. Gueidi, H. Gharsellaoui, and S. B. Ahmed, "Towards unified modeling for nosql solution based on mapping approach," in *Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 25th International Conference KES-2021, Virtual Event / Szczecin, Poland, 8-10 September 2021*, ser. Procedia Computer Science, J. Watróbski, W. Salabun, C. Toro, C. Zanni-Merk, R. J. Howlett, and L. C. Jain, Eds., vol. 192. Elsevier, 2021, pp. 3637–3646. [Online]. Available: https://doi.org/10.1016/j.procs.2021.09.137