SparkSQL$^+$: Next-generation Query Planning over Spark

Binyang Dai
bdaiab@ust.hk
Hong Kong University of Science and Technology
Hong Kong, Hong Kong

Qichen Wang
qcwang@hkbu.edu.hk
Hong Kong Baptist University
Hong Kong, Hong Kong

Ke Yi
yike@ust.hk
Hong Kong University of Science and Technology
Hong Kong, Hong Kong

ABSTRACT

We will demonstrate SparkSQL$^+$, a SQL processing engine built on top of Spark. Unlike the vanilla SparkSQL that uses classical query plans, SparkSQL$^+$ adopts some of the recently developed new query plans, including generalized hypertree decompositions (GHD), worst-case optimal join (WCOJ) algorithms, and conjunctive queries with comparisons (CQC). SparkSQL$^+$ also provides a platform for users to explore different query plans for a given query through a web-based interface, and compare their performance with classical query plans on the same Spark core.

CCS CONCEPTS

• **Information systems → Query optimization**.

KEYWORDS

conjunctive query, acyclic joins, compilation, visualization

ACM Reference Format:
Binyang Dai, Qichen Wang, and Ke Yi. 2023. SparkSQL$^+$: Next-generation Query Planning over Spark. In *Companion of the 2023 International Conference on Management of Data (SIGMOD-Companion '23), June 18–23, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3555041.3589715

1 INTRODUCTION

Select-Project-Join (SPJ) queries form the backbone of most analytical queries. Evaluation of SPJ queries, especially across multiple relations, is often the bottleneck of modern OLAP systems. The standard technique for evaluating an SPJ query, as implemented in SparkSQL and most traditional query engines, aims at finding an optimal (or near-optimal) query plan, which is a tree where each leaf node corresponds to an input relation and each internal node corresponds to a relational operator: selection (σ), projection (π), or (natural) join ($\bowtie$). However, recent developments in database theory and query evaluation algorithms have shown that, for many queries, no such classical query plans can achieve the optimal running time. We give some examples below to illustrate, while referring the reader to the literature for more details. Let N be the total size of all input relations and OUT the size of the query results.

Example 1.1. For the triangle query

$$Q_1 := R_1(x_1,x_2) \bowtie R_2(x_2,x_3) \bowtie R_3(x_3,x_1),$$

any classical query plan, which joins two of the relations first and then joins the third relation, has a cost of $O(N^2)$ in the worst case. On the other hand, a worst-case optimal join (WCOJ) algorithm [3] can achieve $O(N^{1.5})$ time. In general, WCOJ algorithms achieve a running time of $O(N^{\rho})$ where ρ is the *fractional edge cover number* of the query.

Example 1.2. For the dumbbell query

$$\begin{aligned} Q_2 := {} & R_1(x_1,x_2) \bowtie R_2(x_2,x_3) \bowtie R_3(x_3,x_1) \\ & \bowtie R_4(x_4,x_5) \bowtie R_5(x_5,x_6) \bowtie R_6(x_6,x_4) \bowtie R_7(x_1,x_4), \end{aligned}$$

even an optimal classical query plan has a cost of $O(N^2 + OUT)$ in the worst case. On the other hand, by combining the generalized hypertree decomposition (GHD) [2] and WCOJ algorithms, it is possible to achieve a running time of $O(N^{1.5} + OUT)$. More generally, a running time of $O(N^w + OUT)$ can be achieved where w is the *fractional hypertree width* of the GHD.

Example 1.3. Consider the SJ query:

$$Q_3 := \sigma_{x_1 < x_4}(R_1(x_1,x_2) \bowtie R_2(x_2,x_3) \bowtie R_3(x_3,x_4)).$$

Note that the selection predicate $x_1 < x_4$ crosses relations, so it cannot be pushed down[1]. As such, a classical query plan must evaluate the 3-way join first (using some join order), and then filter the join results through the predicate. This results in a running time of $O(N^2)$ in the worst case. On the other hand, using the recent Conjunctive Queries with Comparisons (CQC) algorithm [4], this query can be evaluated in $O(N + OUT)$ time. Note that here OUT still denotes the query result size after passing the predicate, which can be much smaller than the intermediate 3-way join size. More generally, it has been shown that if the query is *α-acyclic* and the comparisons are *Berge-acyclic*, then $O(N + OUT)$ time can be achieved.

Before these exciting algorithmic developments can really change the classical, 50-year-old query planning paradigm, the following questions must be answered:

(1) Are these algorithmic techniques isolated? Can they all be implemented under a common platform to handle a large class of queries?

[1]If a predicate only involves variables from one relation, it can be pushed down to that relation and then eliminated trivially in linear time (or even sublinear time if indexes are available). Thus, a "predicate" in this paper always refers to one that crosses multiple relations. Furthermore, since an equality predicate can always be rewritten into a natural join by proper renaming attributes, it is without loss of generality to assume that the predicate takes the form of a comparison (i.e., $<$, $\leq$, $>$, or $\geq$) between two functions of the variables.

(2) Similar to classical query planning, these next-generation query processing algorithms also instantiate into multiple plans. For example, there might be different GHDs with the same width, and different join trees and reduction orders for the CQC algorithm. How should we pick the optimal one? Note that the theory says that all plans are equally good up to constant factors, but these constant factors matter a lot in practice.
(3) While these new algorithms have a better *worst-case* running time, are they really better than the classical query plans in the *typical* case, say, on real or benchmark data?

We will demonstrate SparkSQL$^+$, a SQL engine built on top of Spark, which addresses question (1) above, while providing a platform for people to better understand questions (2) and (3). Previously, EmptyHeaded [1] has combined GHD and WCOJ, but not CQC. In terms of addressing questions (2) and (3), SparkSQL$^+$ also provides a fairer platform for comparing the classical query plans with the new ones, since after the query plan has been generated, it is still executed by the same Spark core underlying SparkSQL, which generates a classical query plan. Other benefits of building on top of Spark include fault tolerance, distributed processing, elasticity, and compatibility with a variety of data sources and sinks. Finally, SparkSQL$^+$ is easy to be integrated into the thriving Spark ecosystem, e.g., one may train a machine learning model using SparkML over the results of a SparkSQL$^+$ query. On the other hand, EmptyHeaded is a centralized (with multi-threading) system with its own custom implementation.

2 SPARKSQL$^+$ EXAMPLE QUERY PLANS

The query plans used in SparkSQL$^+$ for Example 1.1 and 1.2 are the same as those in EmptyHeaded [1]. Below, we demonstrate one candidate CQC plan [4] for the query Q_3 in Example 1.3.

We first need to fix a join tree for the query. Suppose we use the one in Figure 1(a). The CQC algorithm makes two passes over the join tree: The first bottom-up pass reduces the join tree, one leaf at a time, until one relation remains, on which the predicate can be evaluated trivially in linear time. Then we perform a top-down pass to unroll these reductions.

Step 1. Both R_1 and R_3 are reducible [4]. Suppose we reduce R_1 first. This reduction involves two operations: (1) Group R_1 by x_2, and sort the tuples by x_1 in ascending order within each group. (2) Convert R_2 to R_2' by evaluating the following query:

```
SELECT R2.x2, R2.x3, MIN(R1.x1) as x5
FROM R1 NATURAL RIGHT OUTER JOIN R2
GROUP BY R2.x2, R2.x3
```

The contents of R_1 and R_2' after this step are shown in Figure 1(b), where $\perp$ denotes an empty group. Note that we have

$$\pi_{x_2,x_3,x_4} Q_3 = \sigma_{x_5<x_4}(R_2' \bowtie R_3),$$

namely, the query has been transformed into $\sigma_{x_5<x_4}(R_2' \bowtie R_3)$.

Step2. Next, we reduce R_3. Similar to **Step 1**, we (1) group R_3 by x_3 and sort x_4 in descending order within each group, and (2) convert R_2' to R_2'' with the query

```
SELECT R2'.x2, R2'.x3, R2'.x5, MAX(R3.x4) as x6
FROM R3 NATURAL RIGHT OUTER JOIN R2'
GROUP BY R2'.x2, R2'.x3, R2'.x5
```

Figure 1(c) shows the result after this reduction. Observe that

$$\pi_{x_2,x_3} Q_3 = \sigma_{x_5<x_6} R_2''.$$

Now, the query has been reduced to just one relation with a predicate, which can be trivially evaluated. All tuples that do not pass the predicate are marked in gray in Figure 1(c).

Step 3. After obtaining $\pi_{x_2,x_3} Q_3$, it remains to augment the results with x_1 and x_4. To do so, we unroll the two reductions. We start by enumerating all results in $R_2'' \bowtie R_3$ such that $x_5 < x_4$. For each tuple t_2 in R_2'', we join it with all t_3 in R_3 such that $t_2.x_3 = t_3.x_3$ and scan the x_4 value from large to small. The scan will stop at the first x_4 that is smaller than $t_2.x_5$, as the remaining results cannot satisfy the comparison. The result is shown in Figure 1(d).

Step 4. Similar to **Step 3**, we further join the result from **Step3** with R_1 and get the final result as in Figure 1(e).

It has been shown that the first two steps take $O(N)$ time while the last two steps take $O(OUT)$ time [4]. These guarantees hold on any join tree and any reduction order, but the hidden constant in the big-O might differ.

SparkSQL$^+$ has also implemented GHD and WCOJ. In combination with CQC, it can support more complicated queries. As example, consider the query Q_2 in Example 1.2 with an additional predicate:

$$Q_4 := \sigma_{x_1 \cdot x_2 + x_3 < x_4 + 2x_5 - x_6} Q_2.$$

Since Q_2 is not acyclic, SparkSQL$^+$ first constructs a GHD with three bags:

$$B_1(x_1, x_2, x_3, y_1 := x_1 \cdot x_2 + x_3) := R_1 \bowtie R_2 \bowtie R_3;$$
$$B_2(x_4, x_5, x_6, y_2 := x_4 + 2x_5 - x_6) := R_4 \bowtie R_5 \bowtie R_6;$$
$$B_3(x_1, x_4) := R_7(x_1, x_4).$$

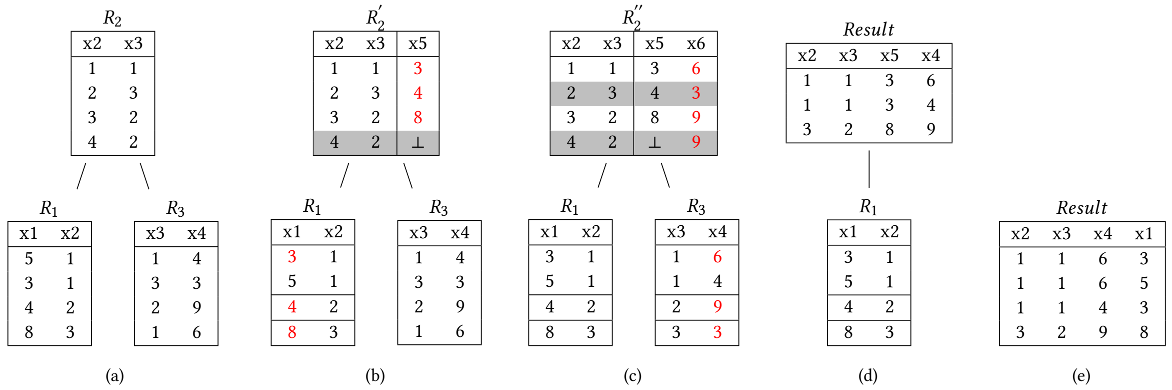
Then it uses WCOJ to compute B_1 and B_2, after which the query transforms to one that is equivalent to Q_3.

3 SYSTEM ARCHITECTURE

Figure 2 depicts the overall structure of SparkSQL$^+$. On a high level, it consists of two parts. The first part translates the query into Scala code, going through various stages including a user interface, the parser, the query planner, and the code generator. The second part compiles the Scala code into a jar, which is then submitted to a Spark cluster with a SparkSQL$^+$ library plus Spark's own necessary dependencies for execution.

User interface. SparkSQL$^+$ provides two user interfaces: a command line, and a web-based interface. The command line interface is easy to interact with and allows the user to integrate SparkSQL$^+$ with other systems. The web-based interface allows the user to explore different query plans (described below) with visualization.

Parser. The parser converts the SQL query to a logical plan. It is implemented based on Calcite. Using Calcite's catalog management and SQL validation, the parser validates the query. We extend the Data Definition Language (DDL) support in Calcite to allow the user to declare more information about the table (e.g., the path to the data source file) in the DDL. After receiving a query from the user interface, it is parsed by the parser and converted into a logical plan as shown in Figure 3(a).



R_2

x2	x3
1	1
2	3
3	2
4	2

R_1

x1	x2
5	1
3	1
4	2
8	3

R_3

x3	x4
1	4
3	3
2	9
1	6

R_2'

x2	x3	x5
1	1	3
2	3	4
3	2	8
4	2	⊥

R_1

x1	x2
3	1
5	1
4	2
8	3

R_3

x3	x4
1	4
3	3
2	9
1	6

R_2''

x2	x3	x5	x6
1	1	3	6
2	3	4	3
3	2	8	9
4	2	⊥	9

R_1

x1	x2
3	1
5	1
4	2
8	3

R_3

x3	x4
1	6
1	4
2	9
3	3

Result

x2	x3	x5	x4
1	1	3	6
1	1	3	4
3	2	8	9

R_1

x1	x2
3	1
5	1
4	2
8	3

Result

x2	x3	x4	x1
1	1	6	3
1	1	6	5
1	1	4	3
3	2	9	8

Figure 1: A running example

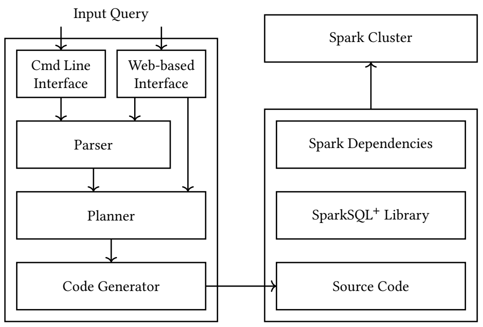
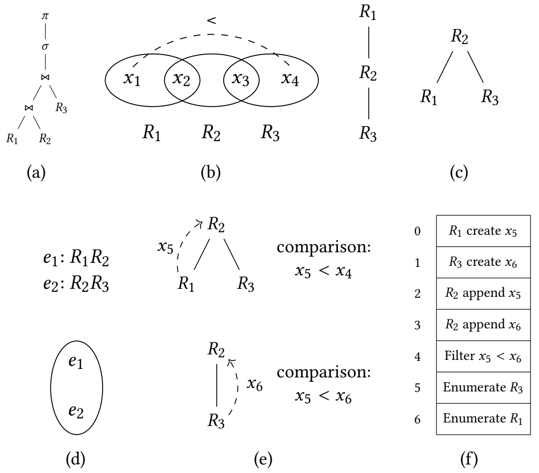


Figure 2: SparkSQL$^+$ system architecture



0	R_1 create x_5
1	R_3 create x_6
2	R_2 append x_5
3	R_2 append x_6
4	Filter $x_5 < x_6$
5	Enumerate R_3
6	Enumerate R_1

Figure 3: Example workflow

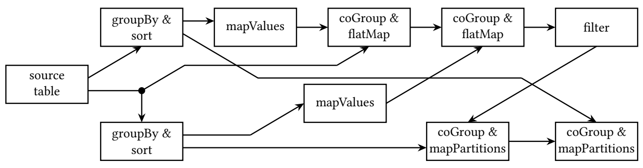


Figure 4: The physical query plan

Query planner. The query planner is the core component of SparkSQL$^+$. After receiving a logical plan from the parser, the planner converts it into a relational hypergraph and a set of comparison predicates as shown in Figure 3(b).

Next, the planner checks if the query is acyclic. If not, it finds a GHD. Then the planner runs the GYO algorithm [5] to find all candidate join trees and their supporting comparison hypergraphs (see [4] for details). For the query in Example 1.3, there are 3 candidates as shown in Figure 3(c). If the user is using the web-based interface, the user can select one of them. Suppose that the join tree in the middle is selected, whose corresponding comparison hypergraph is shown in Figure 3(d).

After a join tree is selected, the planner generates a query plan following [4]. The plan consists of compile actions that tell the code generator what to generate. Figure 3(e) shows the 2 reductions for the given example. When reducing R_1, the planner creates a new column x_5 for R_2', issues a **createExtraColumnAction**, and then updates the comparison to $x_5 < x_4$. Then the planner reduces R_3. Similarly, x_6 is created, a **createExtraColumnAction** is issued, and the comparison is updated to $x_5 < x_6$. Now R_2'' is the only remaining relation, the planner issues **appendExtraColumnAction** and **filterAction**. After that, the planner issues **enumerateAction**s for R_3 and R_1. The issued compile actions are shown in Figure 3(f).

Code Generator. The code generator translates the compile actions into a physical plan, which is a standard Spark lineage graph consisting of RDD operators. For example, the physical plan of Q_3 is shown in Figure 4. The 2 **mapValues** after **groupBy** and **sort** correspond to the creation of x_5 and x_6. The **coGroup** and **flatMap** correspond to appending x_5 and x_6 to R_2. The **coGroup**s and **mapPartitions** in the bottom right corner correspond to enumerations of R_3 and R_1. To eliminate redundancy, SparkSQL$^+$ tries to reuse the RDDs as much as possible when generating the physical plan.

SparkSQL$^+$ library.. To assist the generation of the physical plan, we have also implemented a SparkSQL$^+$ library, which defines a set of APIs such as **sortValuesWith**, **appendExtraColumn**, and **enumerate**. Instead of generating a physical plan, the planner actually generates Scala code that calls these APIs. Then, the SparkSQL$^+$ library translates these APIs into standard RDD operators.

4 DEMONSTRATION

During the demonstration, we will provide an interactive experience for audiences to get a better understanding of these next-generation query plans, as well as experimenting with various

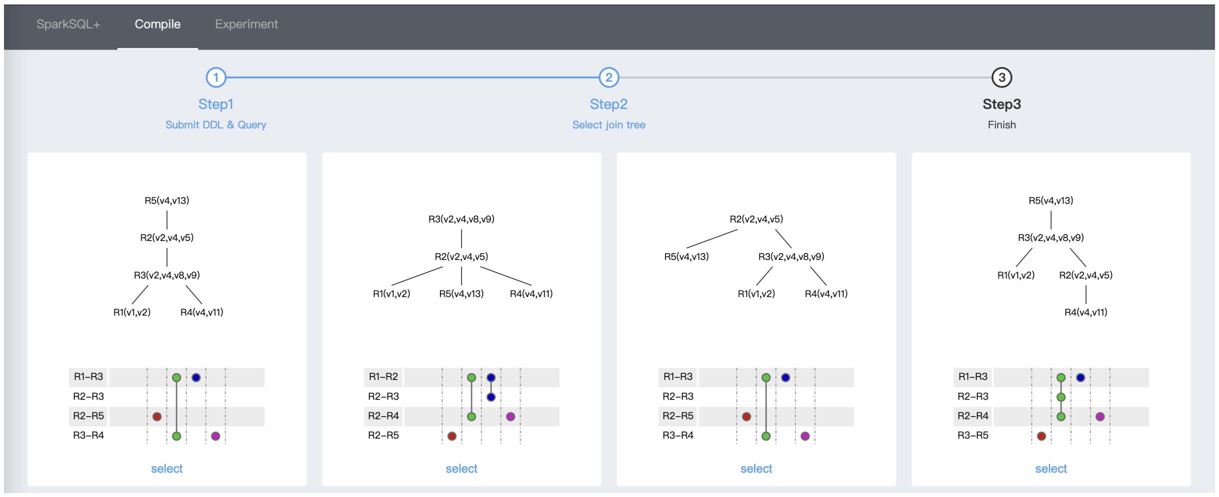
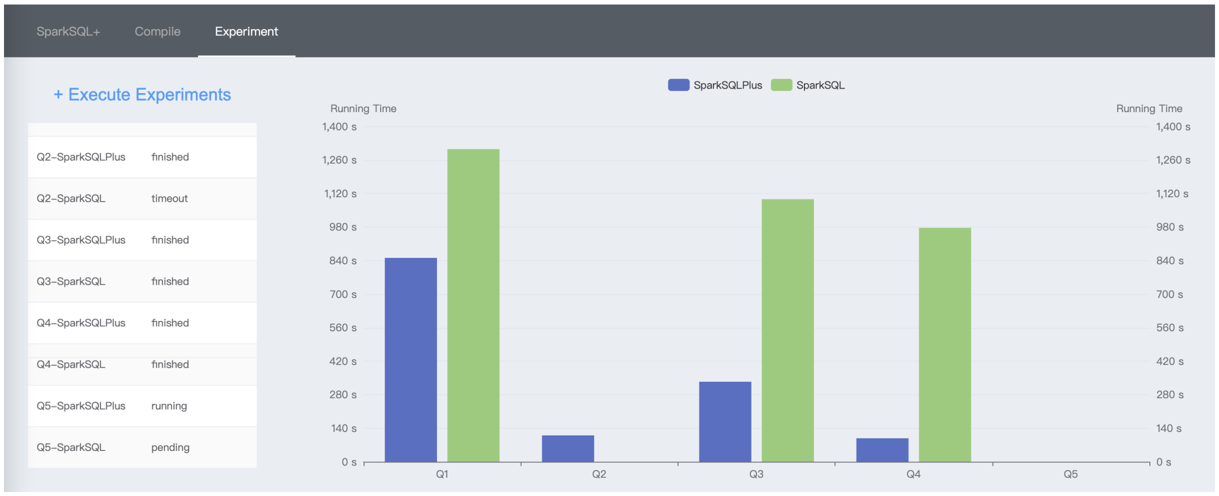


Figure 5: Select candidate view of SparkSQL$^+$



Q2–SparkSQLPlus	finished
Q2–SparkSQL	timeout
Q3–SparkSQLPlus	finished
Q3–SparkSQL	finished
Q4–SparkSQLPlus	finished
Q4–SparkSQL	finished
Q5–SparkSQLPlus	running
Q5–SparkSQL	pending

Figure 6: Experiment view of SparkSQL$^+$

queries and data sets to see if and when these query plans outperform the classical query plans. Specifically, we will guide the audience through the following steps:

Step1. Submit Query. The audience can select one of the predefined queries or write his/her own queries. Then, the DDL and query will be sent to the back-end parser for parsing.

Step2. Select Candidate. After parsing and finding candidate plans, SparkSQL$^+$ will show the plans for the user to select (see Figure 5). The user interface will show both the join tree and the corresponding comparison hypergraph.

Step3. Code generation. The generated Scala code will then be displayed. The user can examine the generated code and then submit it to a back-end Spark cluster for execution.

Step4. Experimental comparison. In addition to running the new plan, one can also use our demonstration system to conveniently run the same query in vanilla SparkSQL, and display the running times side by side (see Figure 6).

ACKNOWLEDGMENTS

This work has been supported by HKRGC under grants 16201819, 16205420, 16209821, and 16205422.

REFERENCES

[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4, Article 20 (oct 2017), 44 pages.

[2] Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. 2005. Hypertree Decompositions: Structure, Algorithms, and Applications. In *Graph-Theoretic Concepts in Computer Science*, Dieter Kratsch (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–15.

[3] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-Case Optimal Join Algorithms: [Extended Abstract]. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Scottsdale, Arizona, USA) *(PODS '12)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/2213556.2213565

[4] Qichen Wang and Ke Yi. 2022. Conjunctive Queries with Comparisons. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 108–121. https://doi.org/10.1145/3514221.3517830

[5] Clement Tak Yu and Meral Z Ozsoyoglu. 1979. An algorithm for tree-query membership of a distributed query. In *COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979.* IEEE, 306–312. https://doi.org/10.1109/CMPSAC.1979.762509