SparkSQL\textsuperscript{+}: 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\textsuperscript{+}, a SQL processing engine built on top of Spark. Unlike the vanilla SparkSQL that uses classical query plans, SparkSQL\textsuperscript{+} 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\textsuperscript{+} 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:

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 ($\sigma$), projection ($\pi$), 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 $\rho$ is the fractional edge cover number of the query.

Example 1.2. For the dumbbell query

$$Q_2 := R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie R_3(x_3, x_1) \bowtie R_4(x_4, x_3) \bowtie R_5(x_5, x_4) \bowtie \cdots \bowtie R_7(x_1, x_4),$$

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:

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

Note that the selection predicate $x_1 < x_4$ crosses relations, so it cannot be pushed down\textsuperscript{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 $\alpha$-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?

\textsuperscript{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 ($i.e., <, \leq, >, \geq$) 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 Q3 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 R1 and R3 are reducible [4]. Suppose we reduce R1 first. This reduction involves two operations: (1) Group R1 by x2, and sort the tuples by x1 in ascending order within each group. (2) Convert R2 to R′ 2 by evaluating the following query:

\[
\text{SELECT } R_{2,x2}, R_{2,x3}, \text{MIN}(R_{1,x1}) \text{ as } x3 \\
\text{FROM } R_{1} \text{ NATURAL RIGHT OUTER JOIN } R_{2} \\
\text{GROUP BY } R_{2,x2}, R_{2,x3}
\]

The contents of R1 and R′ 2 after this step are shown in Figure 1(b), where ⊥ denotes an empty group. Note that we have

\[
\pi_{x2,x3}(Q3) = \sigma_{x2 \leq x3}(R′ 2 \bowtie R1)
\]

namely, the query has been transformed into \(\sigma_{x2 \leq x3}(R′ 2 \bowtie R1)\).

**Step 2.** Next, we reduce R3. Similar to **Step 1**, we (1) group R3 by x3 and sort x4 in descending order within each group, and (2) convert R′ 2 to R′′ 2 with the query

\[
\text{SELECT } R_{2,x2}, R′_{2,x3}, R′_{2,x4}, \text{MAX}(R_{3,x4}) \text{ as } x4 \\
\text{FROM } R_{3} \text{ NATURAL RIGHT OUTER JOIN } R′_{2} \\
\text{GROUP BY } R′_{2,x2}, R′_{2,x3}, R′_{2,x4}
\]

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

\[
\pi_{x2,x3}(Q3) = \sigma_{x3 \leq x4}(R′′ 2 \bowtie R3).
\]

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 \(\pi_{x2,x3}(Q3)\), it remains to augment the results with x1 and x4. To do so, we unroll the two reductions. We start by enumerating all results in R′′ 2 \bowtie R3 such that x3 < x4. For each tuple \(t2 \in R′′ 2\), we join it with all \(t3 \in R3\) such that \(t2.x3 = t3.x3\) and scan the x4 value from large to small. The scan will stop at the first x4 that is smaller than \(t2.x3\), 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 **Step 3** with R1 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 Q4 in Example 1.2 with an additional predicate:

\[
Q4 := \sigma_{x1 \lt x2 + x3 - x4}(Q3).\]

Since Q4 is not acyclic, SparkSQL* first constructs a GHD with three bags:

\[
\begin{align*}
B1 & := (x1, x2, x3, y1 := x1 + x2 + x3) ; := R1 \bowtie R2 \bowtie R3; \\
B2 & := (x4, x5, x6, y2 := x4 + 2x5 - x6) ; := R4 \bowtie R5 \bowtie R6; \\
B3 & := (x1, x4) ; := R7 \bowtie R1.
\end{align*}
\]

Then it uses WCOJ to compute B1 and B2, after which the query transforms to one that is equivalent to Q3.

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).
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 $R'_1$, 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 $R'_3$ is the only remaining relation, the planner issues `appendExtraColumnAction` and `filterAction`. After that, the planner issues `enumerateActions` 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 $Q_1$ 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 $x_6$ to $R_2$. The `coGroups` 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
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**


