Efficient Query Re-optimization with Judicious Subquery Selections (full version)

Anonymous Author(s)

ABSTRACT

Query re-optimization is an adaptive query processing technique that re-invokes the optimizer at certain points in query execution. The goal is to dynamically correct the cardinality estimation errors using the statistics collected at runtime to adjust the query plan to improve the overall performance. We identify a key weakness in existing re-optimization algorithms: their subquery division and re-optimization trigger strategies rely heavily on the optimizer's initial plan, which can be far away from optimal. We, therefore, propose QuerySplit, a novel re-optimization algorithm that skips the potentially misleading global plan and instead generates subqueries directly from the logical plan as the basic re-optimization units. By developing a cost function that prioritizes the execution of less "damaging" subqueries, QuerySplit successfully postpones (sometimes avoids) the execution of complex large joins to maximize their probability of having smaller input sizes. We implemented QuerySplit in PostgreSQL and compared our solution against four state-of-the-art re-optimization algorithms using the Join Order Benchmark. Our experiments show that QuerySplit reduces the benchmark execution time by 35% compared to the second-best alternative. The performance gap between QuerySplit and an optimal optimizer is within 4%.

1 INTRODUCTION

Given a query, a cost-based optimizer in a relational database management system (DBMS) enumerates a subset of valid plans through dynamic programming and computes the cost for each plan by feeding the estimated cardinalities of the intermediate results to the cost model. If such estimations are way off, no matter how precise the cost model is, the optimizer is likely to choose a sub-optimal plan, thus slowing down the query [21]. Unfortunately, it is difficult to get accurate cardinality estimations (CE) consistently, especially for joins because columns are often correlated in real-world data sets [30, 39]. Researchers have proposed new approaches beyond conventional histograms, including multidimensional histogram [10], sketch [6, 31], sampling [22, 39] and machine learning [36] to improve on the CE accuracy. None of them, however, is robust enough to be able to declare victory in solving the problem [21, 23, 30].

The intrinsic difficulty of cardinality estimation calls for alternative approaches to query optimizations. One of such is re-optimization [17, 18, 25, 27, 30]. The idea is straight-forward: if the optimizer cannot make accurate predictions of the cardinalities upfront, we will have to correct its mistakes dynamically at runtime. Therefore, the process of re-optimization is an interleaving of query execution and query optimization: it executes the query partially, obtains some true cardinalities and runtime statistics, and then invokes the optimizer again, hoping to improve the efficiency of the remaining plan. The recent investigation by Perron et al. shows that even a basic re-optimization strategy could remedy a significant portion of the performance losses caused by cardinality mis-estimations [30].

The key problem of designing a re-optimization strategy is to decide (1) which subquery to execute next and (2) when to materialize the intermediate results and re-invoke the optimizer. Existing solutions rely heavily on the optimizer's initial plan [17, 25, 27, 30]. They repeatedly extract subtrees from the complete plan to execute and then use the results to refine the remaining parts. The initial plan, however, can be far away from optimal because of inaccurate cardinality estimations. In this case, the DBMS is likely to choose the "wrong" subplan (e.g., costly itself or generate large results) to execute first, and such a mistake is often unrecoverable by subsequent re-optimization steps.

Meanwhile, these solutions are "reactive" in terms of when to trigger re-optimization, i.e., the re-optimization frequency depends heavily on the initial physical plan. For example, mid-query re-optimization by Kabra et al. only materializes results at pipeline breakers (e.g., a sort operation) [17]. Consequently, for a left-deep join tree where each join is a nested-loop join, re-optimization is never triggered. On the contrary, *Pop* aggressively materializes the output at every nested-loop join, causing a large performance and space overhead because of re-optimization [25].

In this paper, we propose a novel re-optimization algorithm, called QuerySplit, to address the above issues. The key idea is to skip the potentially misleading global plans and instead extract *subqueries* directly from the logical plan as the basic units for re-optimization. Join operators in the logical plan are grouped into subqueries according to heuristics developed from the primary-foreign-key relationships to bound/minimize the output sizes of intermediate results. Such a "query split" algorithm is more robust to balance the gains and costs of re-optimization than those operating on the physical plan. QuerySplit then adopts a greedy algorithm to select a subquery with the smallest cost and output cardinality to execute first. The intuition is that the performance of a complex query is often determined by a few large joins (e.g., fact-fact table join). By executing "simpler" (or "less-damaging") subqueries first and re-optimizing the rest, we increase the probability of delaying the execution of those large joins and thus approaching an optimal plan. Notice that although the cost and output cardinality of a subquery are produced by the optimizer, the subqueries are usually

SIGMOD '23, Seattle, WA, USA,

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

simple enough for existing optimizers to generate reasonably good plans.

We implemented QuerySplit in PostgreSQL and compared our algorithm against the state-of-the-art re-optimization solutions, including mid-query re-optimization (Reopt) [17], Pop [25], incremental execution framework (IEF) by Neumann and Galindo-Legaria [27], and a most recent study by Perron et al. [30], on the Join Order Benchmark (JOB) [21]. Our experiments show that QuerySplit reduces the benchmark execution time by 35% as opposed to the second-best alternative algorithm. Moreover, compared to an optimal optimizer (i.e., an optimizer fed by the true cardinality of each operator), QuerySplit slows down the benchmark execution by less than 4%.

The contributions of this paper are as follows. First, we identified that relying on the sub-optimal initial plan is a key weakness of existing re-optimization strategies. Second, we proposed the QuerySplit algorithm that extracted subqueries directly from the logical plan based on the primary-foreign-key relationships to achieve a robust re-optimization efficiency. Finally, we integrated QuerySplit into PostgreSQL and demonstrated the superiority of our algorithm by comparing it to state-of-the-art solutions on the Join Order Benchmark.

2 BACKGROUND & MOTIVATION

2.1 Cardinality Estimation

Cardinality estimation refers to the process of estimating the number of rows generated by each operator at query optimization. It is used as an input parameter to the optimizer’s cost model. Improved cardinality estimation enables more accurate cost estimation, thus helping the optimizer select an efficient plan. Most DBMSs maintain table/column-level statistics such as histograms and the number of distinct values, from which they derive the selectivity of basic single-column predicates. The more challenging tasks are to estimate the selectivity of conjunctive predicates involving multiple columns and to estimate the join cardinality. Because it is too costly to maintain a relatively complete set of multi-column statistics, the optimizer has to make assumptions about the correlation between columns in these cases.

Most widely-used DBMSs such as PostgreSQL and MySQL assume independent data distributions between columns [21, 26]. It is probably one of the best strategies an optimizer could apply given the lack of statistics. In reality, however, highly correlated columns are common, and this approach is likely to deliver underestimated cardinalities [21]. Although the accuracy of cardinality estimation for complex queries can be improved through sampling [22, 39] and machine learning techniques [36], none of the approaches is robust enough, and their intrinsic overhead is hardly justified in real-world database applications [30]. What makes it worse is error propagation. For an N-way natural join, for example, the cardinality estimation error at each join step could grow exponentially with N [15]. This theoretical result matches what we have observed in practice.

2.2 Re-optimization

Query re-optimization is a technique of adaptive query processing [5, 8] where the optimizer is (re)invoked at execution time to correct potential bad plans. Specifically, the optimizer selects a few operations in the physical plan to materialize the intermediate results. It then compares the true cardinality (i.e., statistics from the actual intermediate results) against the previously estimated value. If those values differ too much, the optimizer would re-plan the remaining part of the query using the true cardinality. Although re-optimization itself brings overheads, the revised plan is almost guaranteed to be at least as good as the original one. Re-optimization, therefore, is a process of interleaving the execution of the query engine and the optimizer, as shown in Figure 1.

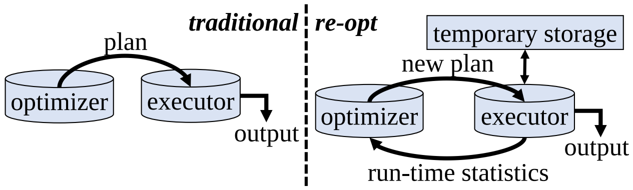
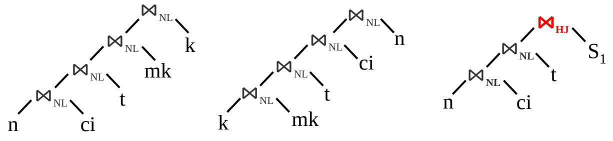


Figure 1: Structure difference between traditional query processing and re-optimization

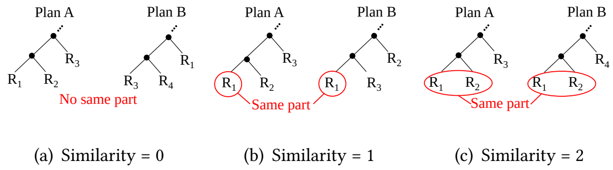


(a) Optimal plan (b) Initial plan (c) Re-Optimized plan after the first join

Figure 2: The example of how a bad global plan influence re-optimization

Existing re-optimization algorithms [17, 25, 27, 30] operate directly on global physical plans. They choose a subtree from the plan obtained from the previous re-optimization cycle and execute that subplan to decide whether further re-optimization is needed. Such a strategy of “selecting a partial query to execute” can be problematic if the referencing global plan deviates largely from an optimal one. And once an undesirable subplan (typically involving large tables but with an underestimated cardinality) is chosen, the damage often propagates through later re-optimization iterations.

Figure 2 shows an example to illustrate such an unrecoverable subplan execution. The query is a 5-way join extracted from the JOB benchmark. There is an index built for each join column. As shown in Figure 2(a), the optimal plan joins table n and ci first and uses the results to probe table t in an index nested-loop join (denoted as NL in the figure). The actual initial plan (Figure 2(b)), however, underestimates the cardinality of $k \bowtie mk$ and thus chooses to execute this subplan first. We use S_1 to denote the intermediate result of $k \bowtie mk$. Once we discover that S_1 is much larger than the prior estimation, we trigger the optimizer to re-plan the rest of the query. However, the best the optimizer can do at this point is shown in Figure 2(c). Because the large temporary table S_1 does not have an index, the DBMS must perform a hash join (highlighted in bold red) for the last step, which could be orders of magnitude slower than probing the existing indexes of the two base tables.



(a) Similarity = 0

(b) Similarity = 1

(c) Similarity = 2

Figure 3: The example of different similarity

Table 1: The ratio of queries whose global plans deviate from the optimal plan with different degrees

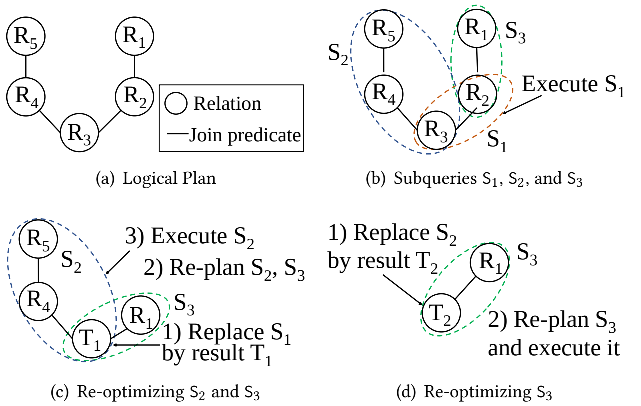
Similarity	0	1	2	> 2
Ratio	13%	12%	32%	43%

To show how much an initial global plan can deviate from optimal, we investigate the query plans in the Join Order Benchmark using the optimizer from PostgreSQL. We define the similarity score of two plans as the number of leaf nodes included in their largest common subtree. For example, as shown in Figure 3, if the first joins of the two plans differ completely, they have a similarity score of 0 (Figure 3(a)); if the probe side scans the same table (but joins a different one), the similarity of the plans is 1 (Figure 3(b)); similarity = 2 means that the plans differ after the first join (Figure 3(c)). In Table 1, we demonstrate how often initial global plans diverge from the optimal ones early in the execution. We observe that more than half of the JOB queries have initial plans whose optimality does not "survive" after one join, among which a quarter of the plans even made mistakes on the first join.

The second problem of existing algorithms is that the re-optimization decision is made reactively according to the types of physical plan nodes. Such a heuristic-based approach often leads to extreme re-optimization frequencies. If the system triggers re-optimization only at pipeline breakers, it never gets a chance to change the ordering of the nested-loop joins in a left-deep plan. On the other hand, if the system invokes re-optimization at every join, the overhead of materializing intermediate results might be intimidating: it essentially converts the execution from the Volcano model to the fully-materialized one.

2.3 A Proactive Strategy

As shown in Section 2.2, a suboptimal global plan can cause irrecoverable damages to the effectiveness of existing re-optimization algorithms. We, therefore, argue that a better strategy is to examine the query's logical plan and decide *proactively* when to materialize results (and re-invoke the optimizer) before execution. We call this scheme QuerySplit. Specifically, we divide the logical plan into subqueries based on the primary-foreign-key relationships and optimize them separately. Because each subquery is relatively simple, it is less likely that the optimizer would make serious mistakes as in a global plan. We then choose one of the subqueries to run and materialize its output. Once the execution is finished, we use the updated statistics (e.g., output size) to re-optimize the remaining relevant subqueries. This process continues until no subquery is left to be executed. The execution order is determined by a "ranking" function (detailed in Section 4.2) where subqueries with small costs and output sizes are prioritized.



(a) Logical Plan

(b) Subqueries S_1, S_2, and S_3

(c) Re-optimizing S_2 and S_3

(d) Re-optimizing S_3

Figure 4: QuerySplit example

Figure 4 shows an example of a 5-way join re-optimized using QuerySplit. We first split the query into three subqueries and optimize them separately: S_1=$R_2 \bowtie R_3$, S_2=$R_3 \bowtie R_4 \bowtie R_5$, and S_3=$R_1 \bowtie R_2$. We then choose S_1 to execute and materialize its output as T_1. Using the statistics of T_1, we trigger re-optimization on the remaining subqueries S_2=$T_1 \bowtie R_4 \bowtie R_5$ and S_3=$R_1 \bowtie T_1$ (Figure 4(c)). S_2 is selected to run next. The result is materialized in T_2, whose statistics is used to re-optimize the final subquery S_3=$R_1 \bowtie T_2$.

Compared to traditional re-optimization algorithms, QuerySplit is more robust at avoiding suboptimal plans, and has more control over the re-optimization cost. First, QuerySplit does not depend on global physical plans. Because the difficulty of the optimization task grows exponentially as the number of joins increases, the optimizer is likely to make mistakes when planning a complex query by entirety. As discussed in Section 2.2, "early mistakes" are common and they cannot be recovered through re-optimization. In QuerySplit, on the other hand, the optimizer only deals with much simpler subqueries, and the probability of generating bad plans is reduced dramatically.

Second, QuerySplit has predictable re-optimization overhead because it determines where to re-invoke the optimizer before executing the query. The overhead is also adjustable by modifying the subquery granularity. In this way, QuerySplit avoids undesirable re-optimization frequencies caused by various physical plan shapes, as described in Section 2.2. The trade-off, though, is that QuerySplit might miss certain optimization opportunities that are only recognizable when examining the query as a whole. QuerySplit could be "myopic" because the optimizer only operates at the subquery level. Our detailed evaluation (Sections 6.3 and 6.5), however, demonstrate that such a trade-off is modest and is outweighed by the aforementioned benefits.

The rest of the paper is organized as follows. Section 3 provides an overview of QuerySplit with correctness proof. Section 4 discusses two critical policies that could largely determine the efficiency of our algorithm. Section 5 briefly describes the integration of QuerySplit into PostgreSQL. An evaluation of

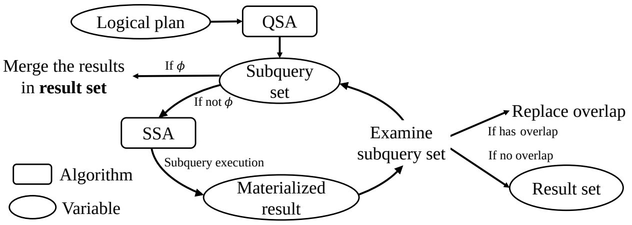


Figure 5: The workflow of QuerySplit

QuerySplit along with detailed case studies is presented in Section 6 followed by the related work in Section 7.

3 QUERYSPLIT

In this section, we present an overview of the QuerySplit algorithm followed by a proof of correctness. Algorithm details and implementation are further discussed in Section 4.

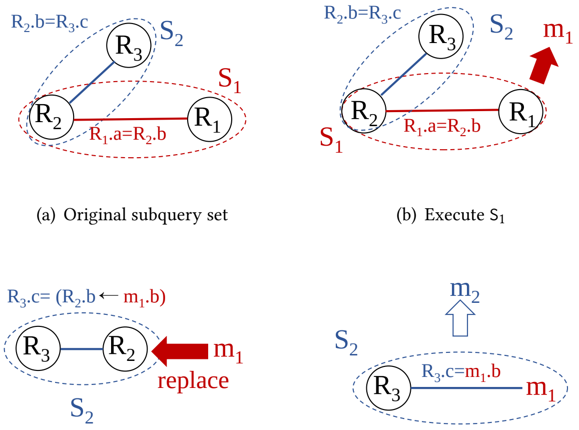
3.1 Algorithm Overview

As shown in Figure 5, QuerySplit takes in a query's logical plan and runs the Query Splitting Algorithm (QSA) against it. For simplicity, we restrict the input to be select-projection-join (SPJ) queries only (i.e., queries involving only select, projection, and join operators). Algorithm extension to support non-SPJ queries is discussed in Section 3.3. The goal of QSA is to generate a set of subqueries where a serial execution of the subqueries would have the same result as executing the original query. We discuss the requirements of such a valid `subquery set` in Section 3.2.

With a valid subquery set, QuerySplit proceeds to enter a loop. At each loop iteration, QuerySplit picks a subquery from the current set to execute according to the Subquery Selection Algorithm (SSA). Although SSA does not affect correctness, it has a significant impact on the efficiency of the entire QuerySplit algorithm. We discuss different subquery ranking strategies in detail in Section 4.2. The selected subquery is then removed from the set, and the execution results as well as the associated statistics are materialized.

Next, QuerySplit examines each of the remaining subqueries in the set: if it overlaps with the just-executed subquery (i.e., they have shared relations), the shared relations are replaced with the corresponding materialized results. If it turns out that the just-executed subquery does not overlap with any of the subqueries in the set, its execution results are pushed to the `result set`. The loop continues until the subquery set becomes empty. Finally, QuerySplit merges the results (through Cartesian product) if there are multiple items in the result set.

Figure 6 shows an example. The original query is $R1 \bowtie_{R1.a=R2.b} R2 \bowtie_{R2.b=R3.c} R3$, where $R1.a$ denotes attribute a from relation $R1$. After running the QSA, we obtain two subqueries in the set: $S1 = R1 \bowtie_{R1.a=R2.b} R2$ and $S2 = R2 \bowtie_{R2.b=R3.c} R3$. Suppose the SSA selects $S1$ to execute first, and the materialized result is denoted by $m1$. We next examine the remaining subquery $S2$. Because $S2$ and $S1$ share the common relation $R2$, we substitute $m1$ for $R2$ and rewrite $S2$ as $m1 \bowtie_{m1.b=R3.c} R3$. We then enter the next iteration and execute $S2$. Because there is no more subquery in the set, we push the execution result $m2$ to the result set and complete the algorithm.



(a) Original subquery set

(b) Execute S_1

(c) Modify S_2

(d) Execute S_2

Figure 6: An example of how QuerySplit works

Notice that the projection operators can be added to the subqueries following the general projection push-down rules. We omit projections in the following discussions for presentation clarity.

3.2 Correctness

As indicated above, the correctness of QuerySplit depends on the output of QSA (i.e., the subquery set). In this subsection, we define the required properties of the subquery set and prove that the QuerySplit algorithm is correct given a valid subquery set.

Given a set of relations $\boldsymbol{R}$ and a set of predicates $\boldsymbol{P}$ over $\boldsymbol{R}$, we define the normal form of an SPJ query as

$$q(\boldsymbol{R}, \boldsymbol{P}) = (\sigma_{\boldsymbol{P}}(r_1 \times r_2 \times ... \times r_m)), r_i \in \boldsymbol{R}$$

A query $q'(\boldsymbol{R'}, \boldsymbol{P'})$ is said to be a subquery of $q(\boldsymbol{R}, \boldsymbol{P})$ if $\boldsymbol{R'} \subseteq \boldsymbol{R}$ and $\boldsymbol{P'} \subseteq \boldsymbol{P}$. A Query Splitting Algorithm (QSA) takes a query q as input and produces a set of subqueries of q. QuerySplit then operates on this subquery set, as described in Section 3.1. Intuitively, in order for QuerySplit to produce the same result as the original query, the subquery set generated by QSA must "cover" all the relations and predicates in the original query. More formally,

Definition 1. *Given an SPJ query $q(\boldsymbol{R}, \boldsymbol{P})$, let $\boldsymbol{Q} = \{q_1(\boldsymbol{R}_1, \boldsymbol{P}_1)$, ..., $q_n(\boldsymbol{R}_n, \boldsymbol{P}_n)\}$ be a set of subqueries of q. $\boldsymbol{Q}$ is said to cover q (denoted as $\boldsymbol{Q} \rightharpoonup_c q$) if the following holds:*
(1) $\cup_{i=1}^{n} \boldsymbol{R}_i = \boldsymbol{R}$
(2) $\cup_{i=1}^{n} \boldsymbol{P}_i$ logically implies $\boldsymbol{P}$. [1]

The above definition guarantees that each base relation r_i in $\boldsymbol{R}$ and each predicate p_i in $\boldsymbol{P}$ must appear at least once in the subquery set $\boldsymbol{Q}$. Notice that the definition allows the same r_i or p_i to be included in multiple subqueries. This does not affect the correctness of the QuerySplit algorithm because the duplicates will

[1] "A logically implies B" means that each predicate from B can be inferred by A.

be removed during the (materialized) result substitution step. In fact, the "coverness" property is sufficient to prove the correctness of the entire QuerySplit algorithm.

Theorem (1). *Let $q(\boldsymbol{R}, \boldsymbol{P})$ be an SPJ query, $\boldsymbol{Q}$ be a set of subqueries of q. QuerySplit produces the same output as q if $\boldsymbol{Q} \rightharpoonup_c q$.*

Proof sketch: We prove the theorem by induction. For the base case, if there is only one query q' in $\boldsymbol{Q}$, and $\boldsymbol{Q} \rightharpoonup_c q$, then q' and q are equivalent queries. Suppose the statement is true for $|\boldsymbol{Q}| = n-1$, we want to prove that it also holds for $|\boldsymbol{Q}| = n$. Let $\boldsymbol{Q} = \{q_1, q_2, ..., q_n\}$. Without loosing generality, suppose the first query executed by QuerySplit is q_1, and the remaining set is $\boldsymbol{Q'} = \{q_2, ..., q_n\}$. If q_1 overlaps (i.e., with at least one shared relation) with a query q_i in $\boldsymbol{Q'}$, we can prove that substituting the materialized view of q_1 into q_i does not change the overall query result. Then, the remaining subquery set (after the substitution) $\boldsymbol{Q''} = \{q_2, ..., q_i', ..., q_n\}$ falls back to the induction hypothesis. On the other hand, if q_1 does not overlap with any of the queries in $\boldsymbol{Q'}$, then executing this "isolated" subquery first and pushing its result to the final buffer does not affect the correctness of the original query. Again, after executing q_1, the remaining subquery set $\boldsymbol{Q'}$ falls back to the induction hypothesis. A detailed proof can be found in Appendix A.

3.3 Extending to Non-SPJ Queries

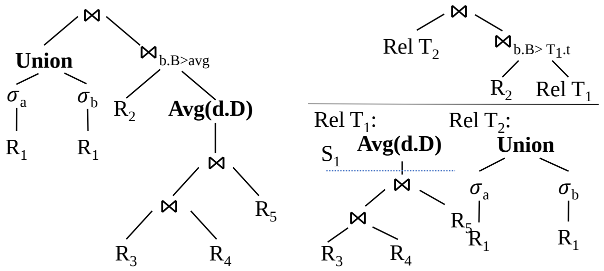
We briefly discuss how to extend QuerySplit to handle Non-SPJ Queries. Given a Non-SPJ operator (e.g., aggregation, semi join) ϕ whose inputs are generated by a set of SPJ subqueries $q_1, q_2, ..., q_n$, we apply the QuerySplit algorithm to each of the subqueries and feed their results to ϕ. Unlike the subquery execution within QuerySplit where the result must be materialized for re-optimization, the data transfer between ϕ and the q_i's can be pipelined.

For a query plan that contains multiple Non-SPJ operators, we segment the plan tree according to these operators and execute them from the bottom up After completing each Non-SPJ operator, we materialize its result and treat it as a base relation in the subsequent QuerySplit invocations.

Figure 7 shows an example. On the left, there is a query plan containing two Non-SPJ operators: a **Union** and an **Avg** aggregation. We first divide the plan tree based on these two operators so that they become the roots of their own subtrees. For each of the subtree, we execute the SPJ part of the plan first. In the **Avg** subtree, for example, the subquery $R_3 \bowtie R_4 \bowtie R_5$ is first executed through QuerySplit, and the result is used as an input to the **Avg** operator. We then materialize the output of **Avg** and **Union** as relation T_1 and T_2, and use those to replace the corresponding subtrees in the root plan. Finally, we invoke QuerySplit again on the root plan to obtain the final result.

4 SUBQUERY CREATION & SELECTION POLICIES

In Section 3, we mainly focused on the correctness of the QuerySplit algorithm. To fully exploit QuerySplit's ability to deliver good query performance, we discuss two critical policies in this section: (1) how to pick a subquery set from a (exponentially) large number of valid choices, and (2) how to select a subquery from an existing set at each QuerySplit iteration.



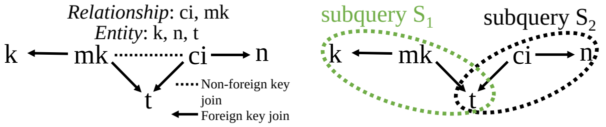
(a) Non-SPJ query

(b) New SPJ query

Figure 7: How to deal with non-SPJ query

```
SELECT MIN(k.keyword) AS movie_keyword
       MIN(n.name) AS actor_name,
       MIN(t.title) AS hero_movie
FROM cast_info AS ci, keyword AS k,
     movie_keyword AS mk, name AS n,
     title AS t
WHERE … filter conditions …
  AND k.id = mk.keyword_id
  AND t.id = mk.movie_id
  AND t.id = ci.movie_id
  AND ci.movie_id = mk.movie_id
  AND n.id = ci.person_id;
```

(a) Original SQL



(b) Join schema

(c) Directed join graph

Figure 8: Join graph split by *RelationshipCenter*

4.1 Generating a Subquery Set

As described in Section 3.2, a subquery set $\boldsymbol{Q}$ output by the Query Splitting Algorithm (QSA) is valid if it "covers" the original query q. The number of the candidate sets, however, grows exponentially with the complexity of q (e.g., the number relations in q). The goal of the QSA is to perform an efficient search in the candidate space and produce a subquery set that best serves the overall QuerySplit algorithm.

Intuitively, we prefer subqueries that return small results. There are two potential advantages. First, the cost of materializing the output of such a subquery is small in each QuerySplit iteration. Second, because the materialized result will become a new base relation to participate in subsequent executions, having a smaller size is beneficial to improving the performance of succeeding subqueries.

Table 2: Cost Functions for SSA

Function Name	Expression
Φ_1	$\mathbf{C}(q)$
Φ_2	$\mathbf{C}(q) \cdot \log(\mathbf{S}(q))$
Φ_3	$\mathbf{C}(q) \cdot \sqrt{\mathbf{S}(q)}$
Φ_4	$\mathbf{C}(q) \cdot \mathbf{S}(q)$
Φ_5	$\mathbf{S}(q)$

We, therefore, propose a subquery generation strategy, called the *RelationshipCenter* strategy (i.e., *RCenter*). *RCenter* leverages the concept of non-expanding operators proposed by Axel et al. [13]. A non-expanding operator is defined as an operator that has an output size smaller than or equal to its input size. A typical example is the primary-foreign-key join operator whose result size cannot exceed the size of the foreign-key relation.

The *RCenter* strategy is based on the non-expanding property of the primary-foreign-key joins. Many relational database schemas follow the classic entity-relationship design [7] (i.e., the star schema) where a "relationship" relation (or the "fact" table) is at the center containing foreign keys referring to the surrounding "entity" relations (or "dimension" tables). Therefore, given a query, we define *R-relation* as a relation that has at least one foreign-key reference to another relation and *E-relation* as a relation whose primary key is referenced by other relations.

We then construct a directed join graph for the query where each vertex represents a relation and each edge represents a join predicate (single table predicates are associated with the vertices). For each edge, its direction is from an R-relation to an E-relation. If the join happens between two relations with the same type, the edge is bidirectional. Redundant join predicates (e.g., those form cycles in the join graph) are deleted with the priority of removing bidirectional edges.

The *RCenter* strategy works by traversing all the vertices in the directed join graph. For each vertex that has at least one outgoing edge, we create a subquery centered at this vertex (i.e., relation) with all the relations it points to. We next demonstrate the process with an example.

Figure 8(a) shows the SQL text of Query 6d from the Join Order Benchmark. Among the five join predicates, four of them ($k \bowtie mk$, $t \bowtie mk$, $t \bowtie ci$, and $n \bowtie ci$) are primary-foreign-key joins. We first build the directed join graph, indicating the primary-foreign-key relationships, as shown in Figure 8(b). Because mk, t, and ci form a join cycle, we remove the redundant bidirectional edge between mk and ci from the graph. We then scan the node list and detect that mk and ci have outgoing edges. We, therefore, create subquery $S_1 = k \bowtie mk \bowtie t$ centered around mk and subquery $S_2 = t \bowtie ci \bowtie n$ centered around ci, as illustrated in Figure 8(c).

4.2 Subquery Execution Order

Given a subquery set generated by the *RCenter* strategy, the second policy decision is how to determine the execution order of the subqueries (i.e., the aforementioned Subquery Selection Algorithm, or SSA). Although this order is irrelevant to the correctness of QuerySplit, it can largely affect the overall query performance. For example, if the largest join in the query is executed early in the re-optimization process, the overhead of materializing its output and scanning it in subsequent executions is overwhelming.

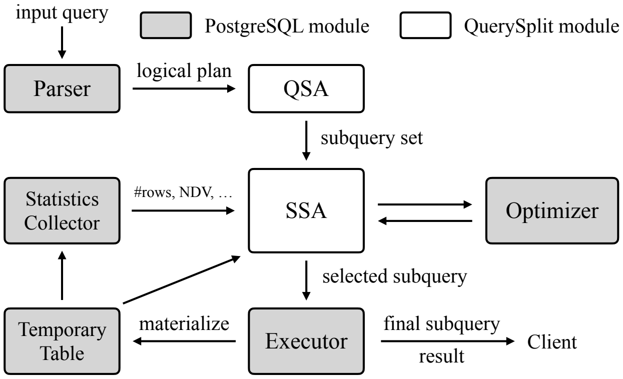


Figure 9: Implementing QuerySplit in PostgreSQL

As mentioned in the introduction, it is beneficial to run the "simpler" subqueries first and delay the execution of large joins by as much as possible. In this way, we increase the probability of reducing the input sizes of those large joins with a modest re-optimization cost. We, therefore, developed a set of cost functions Φ to measure the "simplicity" of the subqueries, as shown in Table 2. At each QuerySplit iteration, we compute Φ for each subquery in the set and select the one with the smallest value to execute.

The two metrics used to compute Φ for a subquery q are the estimated cost generated from the optimizer (denoted by $\mathbf{C}(q)$) and the cardinality estimation of q's output (denoted by $\mathbf{S}(q)$). The intuition is that we want to prioritize a subquery that is fast to execute and has the most potential to speed up later subqueries. Correspondingly, the optimizer-generated cost reveals the complexity of the current subquery, while the output size estimation suggests its potential "burden" on future subqueries. For both metrics, a smaller value is better.

A combination of $\mathbf{C}(q)$ and $\mathbf{S}(q)$ indicates the algorithm's aggressiveness in investing the cost of the present subquery for potential future benefits. A larger factor of $\mathbf{C}(q)$ suggests a more conservative strategy that emphasizes picking the easiest subquery to execute at the moment. On the contrary, putting more emphasis on $\mathbf{S}(q)$ means that the algorithm believes that firing a more complex subquery with a smaller result set is going to pay off in subsequent executions. Φ_1 through Φ_5 defined in Table 2 indicate five strategies with an ascending weight of $\mathbf{S}(q)$. Our evaluation in Section 6.2 shows that a balanced SSA strategy (i.e., $\Phi_4 = \mathbf{C}(q) \cdot \mathbf{S}(q)$) in QuerySplit delivers the best and most robust performance.

5 IMPLEMENTATION

We implemented the QuerySplit algorithm in PostgreSQL 12.3 in C. As shown in Figure 9, we included two new modules: the Query Splitting Algorithm (QSA) module and the Subquery Selection Algorithm (SSA) module. The data flow is redirected accordingly to perform subquery execution and re-optimization.

Specifically, the QSA module receives a logical plan from the built-in parser and runs the RCenter-based algorithm (Section 4.1) to produce a subquery set that covers the original query. The subqueries in the set are logical plans of the same type as the input of QSA. The subquery set is then sent to the SSA module for execution.

Upon receiving the subquery set, the SSA module starts a loop to consume one subquery at each iteration. As described in Section 4.2, the SSA computes the cost function Φ for each subquery and sends the one with the smallest cost to the execution engine. During this process, the SSA invokes PostgreSQL's native optimizer on each subquery to obtain its execution time estimation (i.e., $\mathbf{C}(q)$) and its output size estimation (i.e., $\mathbf{S}(q)$).

The execution result of the selected subquery (except for the last one) is materialized in a memory buffer by setting the output destination to a temporary table in PostgreSQL. The materialized table is then sent to the Statistics Collector, where a set of PostgreSQL's native routines are performed to compute the basic statistics about the table such as the number of rows, number of distinct values, histograms, etc. After the statistical analysis, both the temporary table and its associated statistics are sent back to the SSA module to update the remaining "overlapping" subqueries, preparing for the next iteration.

The source code of QuerySplit-integrated PostgreSQL is available on Github through the anonymous link in [3].

6 EVALUATION

The evaluation of QuerySplit is organized as follows. In Section 6.2, we first examine the policies proposed in Section 4 for the QSA and SSA algorithms. Because the cost function in SSA relies on the optimizer's output, we then investigate the robustness of our algorithm against varying cardinality estimation errors. Next, we show our main results in Section 6.3, where we compare the end-to-end performance of QuerySplit to that of existing baselines using the Join Order Benchmark (JOB) [21]. A follow-up study on whether to collect run-time statistics on the materialized intermediate results is presented in Section 6.4 to show the trade-offs. In Section 6.5, we conduct detailed case studies to provide further insights about the reasons why QuerySplit outperforms the baselines in a majority of situations.

6.1 Workload & Experiment Setup

The Join Order Benchmark (JOB) is a collection of manually-created queries over the IMDB data set [1]. It has been widely used in prior work to evaluate the optimizer in relational database management systems (DBMSs) [6, 13, 30]. JOB is known to have more complex queries than the standard TPC-H [2] and is preferable for stress-testing the optimizer. There are a total of 113 queries in JOB with 91 of them returning non-empty results. We use these 91 queries in our evaluation. By default, we build a B+tree index for each primary key and foreign key appearing in the schema.

All experiments are performed on a server equipped with an Intel®Core®i9-10900K CPU (3.70 GHz) and 128 GB of DRAM. Similar to QuerySplit, all the baseline algorithms are also implemented in PostgreSQL. We use the same parameter configuration in PostgreSQL across all solutions. We set the `max_parallel_workers` to 0 to guarantee a serial execution of the queries in the workload. The `effective_cache_size` is set to 8 GB. Other parameters follow the PostgreSQL default. The execution of each query times out after 1000 seconds. We repeat each experiment three times and report the average measurements.

Table 3: JOB execution time for different combinations of QSA and SSA policies

Time(s) QSA SSA	RCenter	ECenter	MinSubquery
Φ_1: $\mathbf{C}(q)$	421	378	463
Φ_2: $\mathbf{C}(q) \cdot \log(\mathbf{S}(q))$	327	349	428
Φ_3: $\mathbf{C}(q) \cdot \sqrt{\mathbf{S}(q)}$	328	339	418
Φ_4: $\mathbf{C}(q) \cdot \mathbf{S}(q)$	**295**	350	427
Φ_5: $\mathbf{S}(q)$	348	407	474
`global_deep`	356	413	401

6.2 Policies in QSA & SSA

The default policy for subquery generation (i.e., QSA) in QuerySplit is *RCenter*. To show its efficiency, we introduce two alternative strategies, named *ECenter* and *MinSubquery*. As the name suggests, ECenter is the dual of RCenter: the directed join graph in ECenter has all edges reversed (i.e., from a Relationship to an Entity), making the Entity relation at the center of each generated subquery. The other alternative MinSubquery refers to the strategy of dividing the query into minimum-sized subqueries. For each join predicate in the original query, we construct a subquery containing only the involved relations with their corresponding filter conditions, thus creating the smallest join units.

For the cost function Φ used to determine the subquery execution order (i.e., SSA), besides the five candidates proposed in Table 2, we introduce an additional baseline *global_deep* that orders the subqueries according to the global physical plan. At each QuerySplit iteration, we choose the deepest join operator in the global plan tree and obtain the involved relation set $\mathbf{R}$. The algorithm then searches the subquery set and finds the one(s) whose relation set is a superset of $\mathbf{R}$. If multiple subqueries satisfy the requirement, the algorithm randomly picks a subquery to send for execution. Notice that QuerySplit with the *global_deep* SSA policy is different from executing the global plan directly because the subquery set is not derived from the global plan.

Table 3 shows the execution time of the JOB workload using QuerySplit with different combinations of QSA and SSA policies. For QSA policies, RCenter consistently outperforms the others (except for Φ_1). This result confirms that keeping more non-expanding operators in subqueries is beneficial to re-optimization (the USE paper presents similar findings [13]). The RCenter policy achieves the goal by preserving as much primary-foreign-key joins as possible when splitting the original query.

For SSA policies, Φ_3 and Φ_4 outperform the others in general. The results indicate that subqueries with both short execution time and small output cardinality should have execution priority in re-optimization, and a more balanced weight assignment between

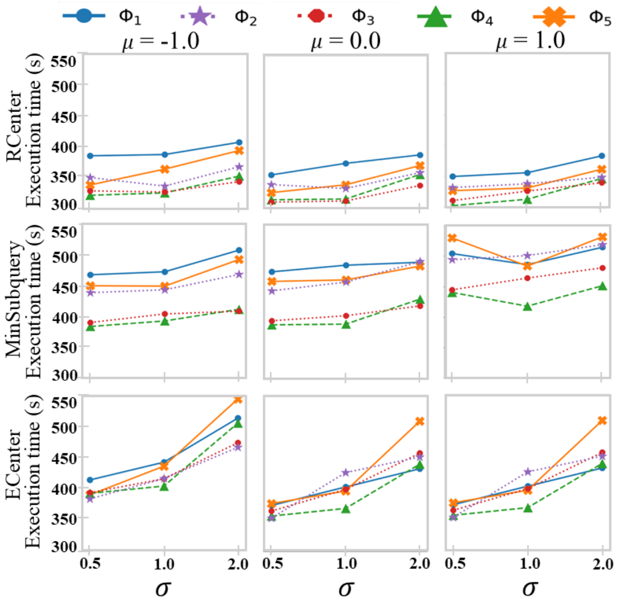


Figure 10: Execution time of QuerySplit on JOB under erroneous cardinality estimations

these two metrics tends to improve the query performance. Overall, a combination of RCenter and Φ_4 in QuerySplit leads to an outstanding performance of the DBMS, as shown in Table 3.

Robustness test Because the cost functions Φ_1 - Φ_5 depend on the output from the optimizer, we next test the robustness of these cost functions against cardinality estimation (CE) errors. The experiments are designed in the following way. For each JOB query, we execute every valid subquery from it and record its true output cardinality (*true_card*). Next, we generate the erroneous cardinality (*err_card*) by adding a controlled noise to the true cardinality:

$$err\_card = 2^{N(\mu,\sigma^2)} * true\_card$$

where $N(\mu, \sigma^2)$ represents a normal distribution with μ as the mean and σ as the standard deviation. We then inject the erroneous cardinalities into PostgreSQL's optimizer via the method proposed by Cai et al. [6] so that we can control the quality of the cardinality estimations (CE).

Figure 10 shows the time of executing the JOB workload under the aforementioned different QuerySplit policies with a varying mean and standard deviation of the injected CE noise. Regardless of the choice of cost functions, errors in cardinality estimation have a small negative impact on the query performance when QuerySplit adopts the RCenter or MinSubquery QSA policy. The ECenter policy is more sensitive to inaccurate cardinalities because an ECenter subquery is more likely to include multiple large "fact" tables, where a miscalculation of the join cardinality could incur an unrecoverable penalty. Because a policy combination of RCenter and Φ_4 outperforms the other pairs consistently, we set both policies as the default in subsequent experiments.

6.3 A Comparison to Baseline Solutions

In this section, we compare QuerySplit against the following baselines, including four re-optimization algorithms and two approaches to improve cardinality estimation.

- **Default**: PostgreSQL with the default optimizer.
- **Optimal**: PostgreSQL with an ideal optimizer. We provide the optimizer with the accurate cardinality of every possible intermediate result so that it generates an optimal plan. This serves as the upper-bound for all the evaluated approaches.
- **Reopt**: A re-optimization algorithm that triggers the optimizer at each pipeline breaker if it detects that the deviation between the true statistics and the estimation exceeds a threshold [17].
- **Pop**: A re-optimization algorithm extending Reopt where the optimizer is triggered aggressively in more situations including at nested-loop join operators [25].
- **IEF**: Incremental Execution Framework (IEF) by Neumann and Galindo-Legaria [27] is an adaptive query processing framework where query executions halt at pre-determined places in the global plan to remove uncertainty in cardinality estimation errors.
- **Perron19**: A most recent study on the effectiveness of re-optimization [30]. In the original paper, the authors compare the true cardinality of each operator with the estimated value obtained from the EXPLAIN command. They then materialize the intermediate results of those operators with large CE errors and re-execute the query to study the performance trade-offs. Because it is impractical to get a global view of the true cardinalities by executing the query in advance, we modified the algorithm by setting a relative threshold (e.g., estimation error is 32× off compared to collected statistics) as the run-time re-optimization trigger.
- **USE, Pessi.**: USE [13] and Pessimistic Cardinality Estimation (Pessi.) [6] are two state-of-the-art approaches to improving cardinality estimation accuracy using sketches. Notice that although USE forms subqueries during query optimization, its execution is non-adaptive.

For baselines that do not provide a PostgreSQL integration, we implemented them according to the paper with the best effort. Implementation details can be found in Appendix B. For each algorithm, we evaluated two index states: (1) indexes are built for primary key (Pk) columns only, and (2) indexes are built for primary key (Pk) and foreign key (Fk) columns.

Figure 11 reports the end-to-end execution time of the JOB workload for QuerySplit and the above baselines. In both *Pk index only* and *Pk + Fk index* cases, QuerySplit achieves the shortest execution time compared to the prior solutions. The performance improvements are more significant when both primary-key and foreign-key indexes are included (which is the default in JOB) because the (sub)plan qualities between different algorithms diverge more with an enlarged optimization space.

Notice that the performance difference between QuerySplit and Optimal is very small (< 4%), indicating that QuerySplit is able to identify and quickly converge to an optimal plan. It also shows that the re-optimization overhead in QuerySplit is modest. The four re-optimization baselines achieve a certain amount of improvement over the Default, but they are still noticeably slower than the

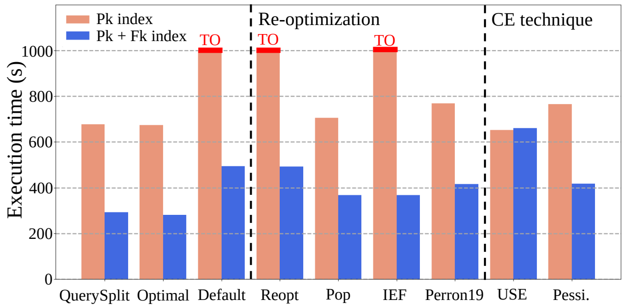


Figure 11: Execution time of JOB for QuerySplit and different baselines (TO = timeout)

Table 4: Materialization frequency and memory usage of re-optimization algorithms

Algorithms	Avg mem per subquery (MB)	Avg mat. freq. per query	Total mem per query (MB)
QuerySplit	5.79	2.66	15.40
Reopt	43.31	0.21	9.09
Pop	7.01	4.62	32.39
IEF	14.59	3.11	45.37
Perron19	10.99	6.59	72.42

optimal. This is because they are all held back by the initial physical plan in the re-optimization process. We provide case studies in Section 6.5 to demonstrate QuerySplit's advantages in detail.

The two improved cardinality estimations (i.e., USE and Pessi.) represent the other route to approach optimal plans. From Figure 11, we observe no superiority of these state-of-the-art methods over re-optimization. This result confirms the finding in Perron et al. [30] that re-optimization is likely to be more effective and efficient than refining CE in improving query performance. Note that USE has the same performance in both index configurations because it disables nest-loop join, and thus ignores indexes in its query planning.

Table 4 shows the materialization frequency and the associated memory usage of each of the re-optimization algorithms. Compared to the baselines, QuerySplit has the lowest memory consumption per subquery (i.e., per re-optimization iteration) because the RCenter-based QSA preserves as many non-expanding operators in the subqueries as possible. QuerySplit also has the second lowest re-optimization frequency per query. Reopt achieves the lowest because it only triggers re-optimization at pipeline breakers with large CE errors. Overall, except for Reopt that adopts an over-conservative strategy, the materialization memory cost of QuerySplit is significantly lower than that of the other competitors.

6.4 Collecting Statistics Or Not?

We continue with a follow-up study on whether collecting statistics on the materialized intermediate results is beneficial for each re-optimization algorithm. The statistics include the number of distinct values, most common values and their frequencies, equal-width/depth histograms, etc. Note that the basic row count is already obtained during the result materialization. Collecting the above statistics requires extra table scans and could incur a high overhead [17]. Nevertheless, all the four re-optimization baselines choose to collect statistics at runtime by default. The reasoning is that collecting these statistics is going to help plan future subqueries because the optimizer has little knowledge of the newly materialized relation(s).

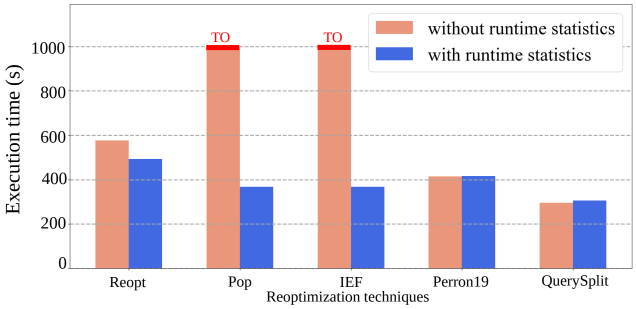


Figure 12: JOB execution time with and without high-level statistics

We repeat the JOB experiments for the re-optimization algorithms in two different settings: (1) collecting the statistics for every materialized intermediate result, and (2) disabling the statistics collector and only passing along the row count to the optimizer.

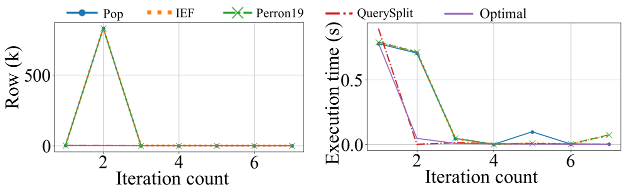
Figure 12 shows the benchmark results. Surprisingly, collecting statistics at runtime has no effect (even a slightly negative effect) on the overall query performance in Perron19 and QuerySplit, despite the performance of other algorithms depending heavily on those statistics. This is because each subquery in Perron19 only involves at most two relations and is, therefore, less likely for the optimizer to make mistakes due to the lack of statistics. For QuerySplit, each subquery mostly contains primary-foreign-key joins. Because PostgreSQL's optimizer does not use any additional statistics other than the row count of the primary-key table to estimate the cardinality of such a join, collecting the statistics provides little benefit for the optimizer to generate a better plan.

The above experiments show that whether to collect statistics during re-optimization should not be a "no-brainer". The decision depends heavily on the re-optimization algorithm and the quality of the system's native optimizer.

6.5 Insights into QuerySplit

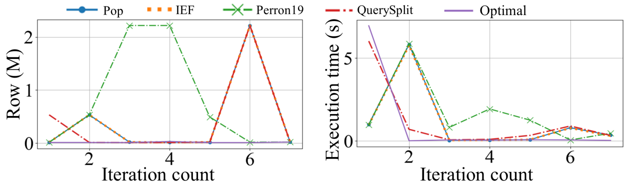
In this section, we provide a deeper analysis on the reasons why QuerySplit outperforms existing re-optimization algorithms. An intuition is that QuerySplit prioritizes the execution of subqueries that produce smaller intermediate results and, thus, postponing potential large joins by as much as possible.

To verify this, we plot two sets of "timelines" for each JOB query, where the X-axis is the count of completed re-optimization iterations. For the first set of timelines, we monitor the size of the intermediate result at each re-optimization iteration for each evaluated algorithm (Reopt is omitted because of poor performance). For the second set of timelines, we plot the execution time for the corresponding subquery at each iteration.



(a) Result size at each iteration (b) Execution time at each iteration

Figure 13: Re-optimization Timelines for the "Avoided Large Join" Category



(a) Result size at each iteration (b) Execution time at each iteration

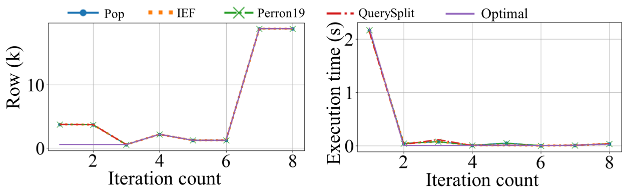
Figure 14: Re-optimization Timelines for the "Delayed Large Join" Category

When comparing QuerySplit against Pop, IEF, Perron19, and Optimal (for reference), we summarized four representative categories out of the 91 JOB queries:

- **Avoided Large Joins**: QuerySplit successfully avoided performing large join(s) that appears in other re-optimization algorithms.
- **Delayed Large Joins**: QuerySplit postponed the large join(s) to later iterations with (hopefully) a smaller input size and a smaller impact on the performance of other subqueries.
- **No Difference**: The timeline patterns and the performance are similar between QuerySplit and other re-optimization algorithms.
- **Worse**: A rare case where QuerySplit unexpectedly produced large intermediate results and performed worse than the others.

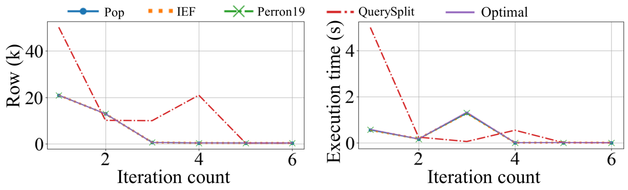
We next provide detailed case studies for each category.

6.5.1 Avoided Large Join. Figure 13 shows the intermediate result size and execution time at each re-optimization iteration for a representative query in this category. We can see that Pop, IEF and Perron19 execute a subquery that generates a large intermediate result close to 1M rows in an early (2nd) iteration. The execution time of this subquery is also large, as shown in Figure 13(b). The reason is that both algorithms rely on a bad initial plan (generated by the PostgreSQL's default optimizer) that decides to execute this large join early because of cardinality estimation errors. On the other hand, QuerySplit successfully avoided the large join by first executing simple subqueries that imposed highly-selective filters on the large input relations. As expected, these wise choices perfectly overlap with the optimal plan.



(a) Result size at each iteration (b) Execution time at each iteration

Figure 15: Re-optimization Timelines for the "No Difference" Category



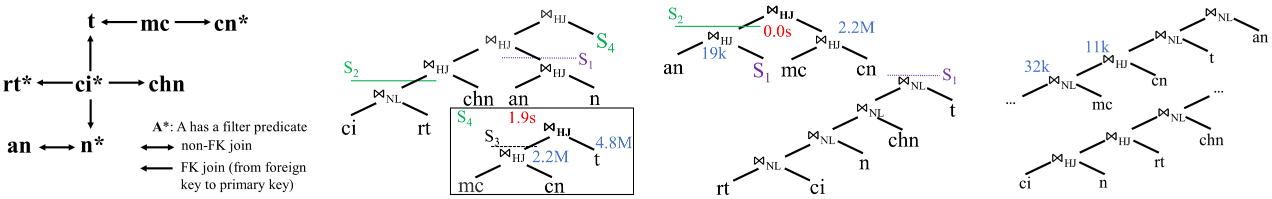
(a) Result size at each iteration (b) Execution time at each iteration

Figure 16: Re-optimization Timelines for the "Worse" Category

6.5.2 Delayed Large Join. A representative set of re-optimization timelines for this category is presented in Figure 14. The observation is that all of the re-optimization algorithms execute at least one subquery that generates a large intermediate result. QuerySplit, however, delays such an execution by as much as possible because the cost function Φ_4 used in the SSA module has a strong preference for subqueries that are "easy" and produce small outputs. Delaying the execution of large joins reduces the probability of letting them slow down additional subqueries. As shown in Figure 14(a), because Perron19 executes a large join as early as in the third iteration, its succeeding iteration suffers from large input and output sizes again due to the ripple effect.

We show the concrete query in Figure 17 for a case study. Figure 17(a) shows the join graph of the JOB query. The (effective) execution plan for Perron19, QuerySplit, and Optimal are illustrated in Figures 17(b) to 17(d), respectively. Both Perron19 and QuerySplit encounter the large join mc $\bowtie$ cn that produces an output relation $\mathbf{x}$ with a size of 2.2M) in one of their subqueries. However, because Perron19 performs this join too early, its subsequent join (i.e., $\mathbf{x} \bowtie$ t) becomes even larger (2.2M × 4.8M) and takes 1.9s to complete. On the contrary, QuerySplit delays the execution of mc $\bowtie$ cn towards the end and gets rewarded by having a much smaller-scale subsequent join (2.2M × 19K) that can be finished in less than 0.1s.

An interesting observation is that the large mc $\bowtie$ cn is completely avoided in the optimal plan. By comparing the plans between QuerySplit and Optimal carefully, we found that the decisive mistake made by QuerySplit is choosing to execute an $\bowtie S_1$ first instead of mc $\bowtie S_1$. This is because an $\bowtie S_1$ has a much smaller estimated cost from the optimizer (i.e., 46K vs. 132K for $\mathbf{C}(q)$) and a similar estimated output size (i.e., 19K vs. 18K for $\mathbf{S}(q)$) compared to



(a) Join Graph of JOB query #9c (b) Perron19 (c) QuerySplit (d) Optimal

Figure 17: (a)The join graph of JOB query #9c and (b)(c) execution processes of the example query in Delayed Large Joins (The blue and red text represents the actual cardinality and the execution time respectively, and the cutting lines represent where the re-optimization is triggered)

`cn` ⋈ `mc` ⋈ S_1. A more sophisticated cost function Φ might be able to further reduce these undesirable decisions. Nevertheless, we notice that the differences in query execution time between QuerySplit and `Optimal` are small despite our algorithm generating a larger intermediate result.

6.5.3 *No Difference.*

As shown in Figure 15, in this category, all the re-optimization algorithms converge to the same (effective) execution plan. This is because the cardinality estimation of these queries is relatively accurate to prevent the optimizer and the re-optimizing process from making mistakes.

6.5.4 *Worse.*

This is a relatively infrequent category where QuerySplit is slower than the other competitors because of a bad decision leading to a large intermediate result. The re-optimization timelines of a representative query are shown in Figure 16. QuerySplit consumes more time and produces larger outputs than the others in the first and fourth iterations.

We found that almost all the bad cases are small queries, each getting split into only two subqueries in QuerySplit. Figure 18 shows an example. Again, Figure 18(a) shows the join graph of the JOB query. We depict the (effective) execution plan for QuerySplit and `IEF` in Figures 17(b) and 17(c), respectively. Other alternative algorithms have the same plan as `IEF`.

We observe that both QuerySplit and `IEF` generate the identical subqueries: $\mathtt{S}_1$ = `ci` ⋈ `rt` ⋈ `t` ⋈ `chn` and $\mathtt{S}_2$ = `mc` ⋈ `cn` ⋈ `t` ⋈ `ct`. The difference is that QuerySplit chose to execute $\mathtt{S}_1$ first, while `IEF` chose to prioritize $\mathtt{S}_2$. This is because PostgreSQL's optimizer makes a huge mistake in estimating the cardinality of $\mathtt{S}_1$. Such a mistake "tricks" QuerySplit into believing that $\mathtt{S}_1$ has a much smaller output size than $\mathtt{S}_2$ (1386 vs. 11K), an advantage outweighing the difference between their execution cost (5.3s vs. 0.71s). Considering the true cardinality for $\mathtt{S}_1$ and $\mathtt{S}_2$ are similar (10K vs. 13K), QuerySplit made a bad decision in executing the heavier $\mathtt{S}_1$ first.

The lesson learned (for future improvements) is that fine-grained subqueries are preferred in re-optimization because they are less likely to cause devastating cardinality estimation errors even with a mediocre optimizer.

6.5.5 *Summary.*

Table 5 presents the query count of each of the above categories out of the 91 JOB queries. The average performance effect refers to QuerySplit's relative performance improvement over the best alternative algorithm. The query counts show that ≈ 70% of the queries belong to the first two categories where QuerySplit outperforms alternative re-optimization algorithms by a sizeable gap. Although queries get slowed down significantly in the `Worse` category, it has a small effect on the overall benchmark performance because such a query is infrequent and the query itself is small.

Table 5: Frequencies and the average performance effect of the four categories of JOB queries

Category	Frequency	Average Perf. Effect
Avoided Large Join	40 / 91	40.5%
Delayed Large Join	23 / 91	21.7%
No difference	18 / 91	3.8%
Worse	10 / 91	-39.5%

7 RELATED WORK

There are two research directions related to our work: (a) adaptive query processing and (b) cardinality estimation techniques. We review existing work in these directions in the following two subsections.

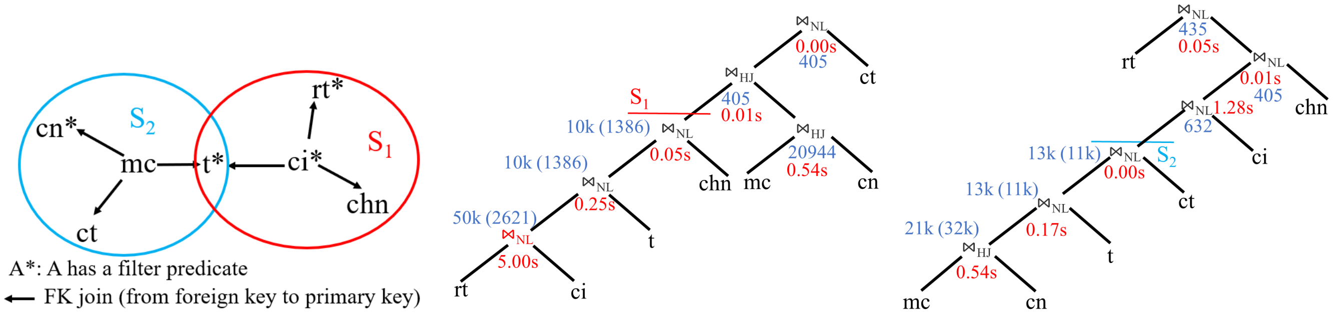
7.1 Adaptive Query Processing

Adaptive query processing is a research direction with a long history. Babu and Bizarro [5] have conducted a comprehensive review of existing works in this direction. According to their investigation, adaptive query processing techniques can be broadly categorized into three families: plan-based system (re-optimization), routing-based system and continuous-query-based system.

7.1.1 *Re-optimization.*

A re-optimization system monitors the execution of the current plan and re-optimizes the plan whenever the actual condition differs significantly from the estimations made by the optimizer.

As far as we know, Reopt is the first research that proposed the idea of re-optimization. Reopt [17] adds statistics collecting operators after pipeline breakers (e.g., hash or sort) in the physical plan. When a deviation is detected, the database calculates the benefit of re-planning the remaining part of the query and compares it with the cost of re-optimization. Pop [25] is very similar to Reopt, except that Pop can trigger re-optimization in more join



(a) Join graph of JOB query #15c

(b) Execution process of QuerySplit

(c) Execution process of IEF

Figure 18: (a) The join graph of JOB query #15c and (b)(c) the execution processes of the example queries in Worse (The blue text outside the bracket and in the bracket represents the actual cardinality and the estimated cardinality respectively, the red text represents the execution time, and the cutting lines represent where the re-optimization is triggered)

nodes, like the outer side of nest-loop join. Instead of deciding when to materialize based on the node type, incremental execution framework (IEF) [27] chose the node in the global physical plan which has the maximal estimation error on cardinality to materialize. The estimation error on cardinality is estimated based on the statistics and assumptions used. Recently, Perron et al. [30] conducts a simulation study to investigate the effectiveness of re-optimization. They use the `EXPLAIN` command to evaluate cardinality estimation error, and materialize the intermediate results that deviate too much from estimation as a temporary table. Their result shows that re-optimization can sharply improve the execution time in PostgreSQL. Moreover, Databricks and Spark extended re-optimization to the Mapreduce background [37]. They re-optimized at shuffle or broadcast exchange, and optimized not only join order and physical operator selection, but also shuffle partitions.

Compared to the QuerySplit framework, the above methods choose the subtree of the global physical plan to execute. Such a strategy can go wrong when the referencing global plan deviates largely from an optimal one. And if a bad subplan is chosen, the damage often influences later execution.

7.1.2 Routing-based system. Routing-based systems behave differently compared to traditional RDBMS. They process queries by routing tuples through a pool of operators. The idea of the routing-based system can be traced back to INGRES [38]. The most representative work is Eddies [4], which adds a new operator called ripple join and can change the join order in ripple joins. Compared to re-optimization, routing-based systems totally abandon the optimizer, making routing algorithms highly dependent on the greedy algorithm and therefore unsuitable for complex queries [16].

7.1.3 Continuous-query-based System. Continuous-Query-based, or CQ-based, systems are used for queries that will run many times or a very long time, which is prevalent in data stream systems. Compared to other adaptive query processing, CQ-base systems pay attention to the runtime change of stream characteristics and system conditions, rather than cardinality estimation errors of a given query.

7.2 Cardinality Estimation

Cardinality estimation techniques are relevant but orthogonal to our work. QuerySplit benefits from the improvement of cardinality estimation on small joins. Making an optimal plan in each subquery can significantly enhance overall performance.

Cardinality estimation techniques can be categorized into traditional methods and learned methods [34], depending on whether machine learning techniques are used.

7.2.1 Traditional Methods. Traditional methods include sketch [6, 13, 31], histogram [10] and sampling [22, 39]. One particularly related work in this part is USE [13], in which the idea of using non-expanding operators (e.g., filter and Pk-Fk join) is independently proposed. In USE, these operators are prioritized to form subqueries (which is similar to subqueries formed in RCenter). Then, sketch-based cardinality estimation techniques are used to decide the join order between subqueries. However, USE is not an adaptive query processing method, and it uses standard query optimization after conducting the above query transformation.

7.2.2 Learned Methods. Learned methods can be further divided into two categories: data-driven cardinality estimator [10, 11, 14, 20, 33, 35, 40, 41] and query-driven cardinality estimator [9, 12, 19, 24, 28, 29, 32, 33]. The data-driven cardinality estimator approximates the data distribution of a table by mapping each tuple to its probability of occurrence in the table. The query-driven cardinality estimator uses some models to learn the mapping between queries and cardinalities. Although learned methods are indeed more accurate than traditional methods, they often suffer from high training and inference costs [34].

8 CONCLUSION

In this paper, we propose QuerySplit, a re-optimization framework which ignores the potentially misleading global plans and instead extracts subqueries directly from the original logical plan. We proposed a cost function that prioritizes the execution of simple subqueries with small output sizes. Experimental results on Join

Order Benchmark showed that QuerySplit outperforms other re-optimization methods and state-of-the-art sketch-based cardinality estimation techniques, and reaches near-optimal execution time.

REFERENCES

[1] 2022. IMDB. https://www.imdb.com.

[2] 2022. TPCH. https://www.tpc.org/tpch/.

[3] Anonymous. 2022. QuerySplit Implementation. https://github.com/274tibjeuw6/querysplit.

[4] Ron Avnur and Joseph M Hellerstein. 2000. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 261–272.

[5] Shivnath Babu and Pedro Bizarro. 2005. Adaptive query processing in the looking glass. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR), Jan. 2005*.

[6] Walter Cai, Magdalena Balazinska, and Dan Suciu. 2019. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *Proceedings of the 2019 International Conference on Management of Data*. 18–35.

[7] Peter Pin-Shan Chen. 1976. The entity-relationship model—toward a unified view of data. *ACM transactions on database systems (TODS)* 1, 1 (1976), 9–36.

[8] Amol Deshpande, Zachary Ives, Vijayshankar Raman, et al. 2007. Adaptive query processing. *Foundations and Trends® in Databases* 1, 1 (2007), 1–140.

[9] Anshuman Dutt, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2020. Efficiently approximating selectivity functions using low overhead regression models. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2215–2228.

[10] Dimitrios Gunopulos, George Kollios, Vassilis J Tsotras, and Carlotta Domeniconi. 2005. Selectivity estimators for multidimensional range queries over real attributes. *the VLDB Journal* 14, 2 (2005), 137–154.

[11] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep learning models for selectivity estimation of multi-attribute queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1035–1050.

[12] Max Heimel, Martin Kiefer, and Volker Markl. 2015. Self-tuning, GPU-accelerated kernel density models for multidimensional selectivity estimation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1477–1492.

[13] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2021. Simplicity Done Right for Join Ordering.. In *CIDR*.

[14] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2019. Deepdb: Learn from data, not from queries! *arXiv preprint arXiv:1909.00607* (2019).

[15] Yannis E Ioannidis and Stavros Christodoulakis. 1991. On the propagation of errors in the size of join results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of data*. 268–277.

[16] Yannis E Ioannidis, Raymond T Ng, Kyuseok Shim, and Timos K Sellis. 1997. Parametric query optimization. *The VLDB Journal* 6, 2 (1997), 132–151.

[17] Navin Kabra and David J DeWitt. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 106–117.

[18] Tomer Kaftan, Magdalena Balazinska, Alvin Cheung, and Johannes Gehrke. 2018. Cuttlefish: A lightweight primitive for adaptive query processing. *arXiv preprint arXiv:1802.09180* (2018).

[19] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[20] Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2019. Estimating cardinalities with deep sketches. In *Proceedings of the 2019 International Conference on Management of Data*. 1937–1940.

[21] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[22] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling.. In *Cidr*.

[23] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal* 27, 5 (2018), 643–668.

[24] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proceedings of the VLDB Endowment* 12, 11 (2019).

[25] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. 2004. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 659–670.

[26] MySQL. 2020. MySQL 8.0 Reference Manual.

[27] Thomas Neumann and Cesar Galindo-Legaria. 2013. Taking the edge off cardinality estimation errors using incremental execution. *Datenbanksysteme für Business, Technologie und Web (BTW) 2019* (2013).

[28] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2018. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. 1–4.

[29] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. 2020. Quicksel: Quick selectivity learning with mixture models. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1017–1033.

[30] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. 2019. How I learned to stop worrying and love re-optimization. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1758–1761.

[31] Florin Rusu and Alin Dobra. 2008. Sketches for size of join estimation. *ACM Transactions on Database Systems (TODS)* 33, 3 (2008), 1–46.

[32] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO-DB2's learning optimizer. In *VLDB*, Vol. 1. 19–28.

[33] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. *Proceedings of the VLDB Endowment* 13, 3 (2019), 307–319.

[34] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned cardinality estimation: A design space exploration and a comparative evaluation. *Proceedings of the VLDB Endowment* 15, 1 (2021), 85–97.

[35] Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. 2011. Lightweight graphical models for selectivity estimation without independence assumptions. *Proceedings of the VLDB Endowment* 4, 11 (2011), 852–863.

[36] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2020. Are we ready for learned cardinality estimation? *arXiv preprint arXiv:2012.06743* (2020).

[37] Fan Wenchen, Hövell Herman van, and Xue MaryAnn. 2020. Adaptive Query Execution: Speeding Up Spark SQL at Runtime. https://www.databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html.

[38] Eugene Wong and Karel Youssefi. 1976. Decomposition—a strategy for query processing. *ACM Transactions on Database Systems (TODS)* 1, 3 (1976), 223–241.

[39] Wentao Wu, Jeffrey F Naughton, and Harneet Singh. 2016. Sampling-based query re-optimization. In *Proceedings of the 2016 International Conference on Management of Data*. 1721–1736.

[40] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. *Proceedings of the VLDB Endowment* 14, 1 (2020), 61–73.

[41] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *Proceedings of the VLDB Endowment* 13, 3 (2019).

A PROOF OF THEOREM 1

This appendix shows the proof of Theorem 1.

THEOREM. *Let $q(\boldsymbol{R}, \boldsymbol{P})$ be an SPJ query, $\boldsymbol{Q}$ be a set of subqueries of q. QuerySplit produces the same output as q if $\boldsymbol{Q} \rightharpoonup_c q$.*

PROOF.

To begin with, let us introduce several notations:

(1) For a set of subqueries $\boldsymbol{Q} = \{q_1(\boldsymbol{R}_1, \boldsymbol{P}_1), ..., q_n(\boldsymbol{R}_n, \boldsymbol{P}_n)\}$, we denote $R(\boldsymbol{Q}) = \cup_{i=1}^{n} \boldsymbol{R}_i$, $P(\boldsymbol{Q}) = \cup_{i=1}^{n} \boldsymbol{P}_i$.

(2) For a set of relations $\boldsymbol{R} = \{r_1, ..., r_n\}$, we denote $\mathsf{X}_{r \in \boldsymbol{R}} = r_1 \times ... \times r_n$.

(3) For a SPJ query $q(\boldsymbol{R}, \boldsymbol{P})$ and a set of subqueries $\boldsymbol{Q}$ of q, we denote the result of q as $E(q) = \sigma_{\boldsymbol{S}}(\mathsf{X}_{r \in \boldsymbol{R}})$, and the output of the QuerySplit algorithm as $E(\boldsymbol{Q})$.

Under these notations, we can rewrite the theorem as: Given a SPJ query $q(\boldsymbol{R}, \boldsymbol{P})$, and a set of subqueries $\boldsymbol{Q}$ of q, such that $\boldsymbol{Q} \rightharpoonup_c q$. Then we have $E(q) = E(\boldsymbol{Q})$.

Without loss of generality, we assume that the names of all attributes in $\boldsymbol{R}$ are unique. Under such assumption, we do not need to consider the rename operation when modifying subqueries, and for simplicity we assume that the rename step is skipped.

Now, we start to prove the rewritten theorem by induction on $|\boldsymbol{Q}|$. First, We prove the statement holds when $|\boldsymbol{Q}| = 1$, in which case

$\boldsymbol{Q} = \{q_1(\boldsymbol{R}_1, \boldsymbol{P}_1)\}$. Apparently the only way that $\boldsymbol{Q} \rightharpoonup_c q$ is $q_1 = q$. So the statement clearly holds for $|\boldsymbol{Q}| = 1$.

Now, assume that the statement holds when $|\boldsymbol{Q}| = n-1$. We consider the case of $|\boldsymbol{Q}| = n$, $\boldsymbol{Q} = \{q_1(\boldsymbol{R}_1, \boldsymbol{P}_1), ..., q_n(\boldsymbol{R}_n, \boldsymbol{P}_n)\}$.

Without loss of generality, we denote the first executed subquery as $q_1(\boldsymbol{R}_1, \boldsymbol{P}_1)$ and discuss two cases: (1) $\forall i > 1, \boldsymbol{R}_1 \cap \boldsymbol{R}_i = \emptyset$ and (2) $\exists i > 1, s.t. \boldsymbol{R}_1 \cap \boldsymbol{R}_i \neq \emptyset$.

***Case 1**: We first execute q_1 and materialize its result as relation $m_1 = E(q_1)$. Then, because $\forall i > 1, \boldsymbol{R}_1 \cap \boldsymbol{R}_i = \emptyset$, according to the algorithm, we have to add m_1 to the subquery result set $\boldsymbol{L}$. After that, we remove q_1 from $\boldsymbol{Q}$ and have a new subquery set $\boldsymbol{Q'} = \{q_2(\boldsymbol{R}_2, \boldsymbol{P}_2), ..., q_n(\boldsymbol{R}_n, \boldsymbol{P}_n)\}$ for the next iteration.*

We construct a SPJ query $q'(\boldsymbol{R'}, \boldsymbol{P'})$, where $\boldsymbol{R'} = \cup_{i=2}^{n} \boldsymbol{R}_i$ and $\boldsymbol{P'} = \cup_{i=2}^{n} \boldsymbol{P}_i$. Apparently, $\boldsymbol{Q'} \rightharpoonup_c q'$ and as $|\boldsymbol{Q'}| = n-1$, by induction hypothesis, $E(q') = E(\boldsymbol{Q'})$.

The final result is the Cartesian product on the elements in $\boldsymbol{L}$, so we have:

$$E(\boldsymbol{Q}) = \mathsf{X}_{r\in \boldsymbol{L}} = m_1 \times \mathsf{X}_{r\in(\boldsymbol{L}\setminus\{m_1\})} = E(q_1) \times E(\boldsymbol{Q'}) = E(q_1) \times E(q')$$

$$= \sigma_{\boldsymbol{P}_1}(\mathsf{X}_{r\in \boldsymbol{R}_1}) \times \sigma_{\boldsymbol{P'}}(\mathsf{X}_{r\in \boldsymbol{R'}}) = \sigma_{\boldsymbol{P}_1 \cup \boldsymbol{P'}}(\mathsf{X}_{r\in \boldsymbol{R}}) = \sigma_{\boldsymbol{P}}(\mathsf{X}_{r\in \boldsymbol{R}}) = E(q)$$

***Case 2**: We denote the set of subqueries that need to be modified after executing q_1 as $\boldsymbol{W} = \{q_k(\boldsymbol{R}_k, \boldsymbol{P}_k) \in \boldsymbol{Q} : k > 1, \boldsymbol{R}_1 \cap \boldsymbol{R}_k \neq \emptyset\}$.*

After we execute q_1 and materialize its result as relation m_1, we modify each $q_i \in \boldsymbol{W}$ and keep $\boldsymbol{L} = \emptyset$. We denote these new-formed subqueries as $q_i'(\boldsymbol{R}_i', \boldsymbol{P}_i')$ and $\boldsymbol{R}_i' = \boldsymbol{R}_i \setminus \boldsymbol{R}_1 \cup \{m_1\}$, $\boldsymbol{P}_i' = \boldsymbol{P}_i$. These new-formed subqueries form a new set $\boldsymbol{W'}$.

Now, $\boldsymbol{Q}$ becomes a new subquery set $\boldsymbol{Q'} = \boldsymbol{Q} \cup \boldsymbol{W'} \setminus \{q_1\} \setminus \boldsymbol{W}$. Because $\boldsymbol{L} = \emptyset$ at this point, when reconstruction finishes, we have $E(\boldsymbol{Q}) = E(\boldsymbol{Q'})$.

We construct a new query $q'(\boldsymbol{R'}, \boldsymbol{P'})$, where $\boldsymbol{R'} = \boldsymbol{R} \setminus \boldsymbol{R}_1 \cup \{m_1\}$ and $\boldsymbol{P'} = \cup_{i=2}^{n} \boldsymbol{P}_i$. We will prove that $E(q') = E(\boldsymbol{Q'})$ and $E(q') = E(q)$, hence finishes the proof. To prove $E(q') = E(\boldsymbol{Q'})$, by induction hypothesis, we only need to show that $\boldsymbol{Q'} \rightharpoonup_c q'$ and $|\boldsymbol{Q'}| = n-1$.

- *Apparently, $R(\boldsymbol{Q'}) = R(\boldsymbol{Q}) \setminus \boldsymbol{R}_1 \cup \{m_1\} = \boldsymbol{R} \setminus \boldsymbol{R}_1 \cup \{m_1\} = \boldsymbol{R'}$. And because $P(\boldsymbol{Q'}) = \cup_{i=2}^{n} \boldsymbol{P}_i = \boldsymbol{P'}$, $P(\boldsymbol{Q'})$ logical implies $\boldsymbol{P'}$, so $\boldsymbol{Q'} \rightharpoonup_c q'$.*
- *Notice that $\boldsymbol{W} \subseteq \boldsymbol{Q}$, $\{q_1\} \subseteq \boldsymbol{Q}$, $\boldsymbol{W'} \cap \boldsymbol{Q} = \emptyset$ and $|\boldsymbol{W}| = |\boldsymbol{W'}|$, so $|\boldsymbol{Q'}| = n-1$.*

By using the induction hypothesis, we get $E(q') = E(\boldsymbol{Q'})$.

At last, we need to prove $E(q') = E(q)$:

$$E(q') = \sigma_{\boldsymbol{P'}}(\mathsf{X}_{r\in \boldsymbol{R'}}) = \sigma_{\boldsymbol{P'}}(\mathsf{X}_{r\in(\boldsymbol{R'}\setminus\{m_1\})} \times m_1) = \sigma_{\boldsymbol{P'}}(\mathsf{X}_{r\in(\boldsymbol{R}\setminus \boldsymbol{R}_1)} \times m_1)$$

Since m_1 is the execution result of q_1, $m_1 = \sigma_{\boldsymbol{S}_1}(\mathsf{X}_{r\in \boldsymbol{R}_1})$, so we have:

$$E(q') = \sigma_{\boldsymbol{P'}}(\mathsf{X}_{r\in(\boldsymbol{R}\setminus \boldsymbol{R}_1)} \times \sigma_{\boldsymbol{P}_1}(\mathsf{X}_{r\in \boldsymbol{R}_1})) = \sigma_{\boldsymbol{P}}(\mathsf{X}_{r\in \boldsymbol{R}}) = E(q)$$

Thus, $E(q) = E(\boldsymbol{Q})$ and the statement holds for $|\boldsymbol{Q}| = n$.

B THE IMPLEMENTATION OF PERRON19'

This appendix describes our implementation of Perron19'. Although Perron et al. [30] compare the performance between PostgreSQL, optimal query execution strategy, and a re-optimization strategy, they only do it in a simulation way. In their simulation, they examine the "EXPLAIN ANALYZE" output of the query and compare the true cardinalities to the PostgreSQL cardinality estimation. For the first join operator in the query plan with a q-error over the threshold, they rewrite this subquery to create a temporary table instead. For the remainder of the query, they replace all the tables in the above join with the temporary table and re-plan. They repeat this procedure until no join operators in the query plan have a q-error over the threshold.

However, it is impossible to calculate the q-error of each join node in the pipeline during execution in practice. This is because each join operator does not produce all its results until the final result is available. Hence, when we know the exact tuple that a join operator produces, the query has been executed, and it is too late to re-optimize.

Hence, to implement the simulation of Perron et al. in practice, we have to break the tuple pipeline in the executor thoroughly. To do so, we materialize the result of each intermediate join operator. When the q-error between the cardinality of materialized result and estimated one is over the threshold, which is set to 32 according to the previous result [30], we first use the routine of the "analyze" command to collect the statistics of the temporary table. And then we use the temporary table to replace the tables that have been used and re-optimize the modified query.