Polynomial Heuristics for Query Optimization

Nicolas Bruno, César Galindo-Legaria, Milind Joshi

Microsoft Corp., USA

{nicholasb,cesarg,milindj@microsoft.com}

Abstract—Research on query optimization has traditionally focused on exhaustive enumeration of an exponential number of candidate plans. Alternatively, heuristics for query optimization are restricted in several ways, such as by either focusing on join predicates only, ignoring the availability of indexes, or in general having high-degree polynomial complexity. In this paper we propose a heuristic approach to very efficiently obtain execution plans for complex queries, which takes into account the presence of indexes and goes beyond simple join reordering. We also introduce a realistic workload generator and validate our approach using both synthetic and real data.

I. INTRODUCTION

Research in query optimization has quickly acknowledged the exponential nature of the problem. While certain special cases can be solved in polynomial time (e.g., chain queries with no cross-products [1] or acyclic join queries under ASI cost models [2]), the general case is NP-hard (see [3], [2]).

Despite the inherent complexity of query optimization, algorithmic research has traditionally focused on exhaustive enumeration of alternatives (see [4] for the classical dynamic programming approach and [5], [6] for a transformation-based approach). As queries become more complex, exhaustive algorithms simply cannot enumerate all alternatives in any reasonable amount of time. For instance, enumerating all join orders for a 15-table star query takes several minutes in commercial systems (but we have seen scenarios with queries that join more than 50 tables together).

To be able to cope with such complex queries, several heuristics have been proposed in the literature (see Section II for more details). However, previous work is limited to joins operators (i.e., they do not consider other relational operators like group-by clauses), do not consider the presence of indexes (which, as we will see, can drastically change the landscape of optimal plans), and can still be inefficient or inadequate in certain scenarios. This is illustrated by the following examples:

Language integrated queries: New advances in programming languages [7] allow expressing declarative queries (similar to those using SQL) in procedural languages like C#. Extensions to compilers and the language themselves make possible to optimize such queries at runtime. In contrast to traditional DBMSs, such declarative queries might be executed in main-memory, are dynamically generated, and usually operate on relatively small amounts of data. For that reason, the latency for optimizing such queries needs to be minimal, and only very efficient optimization strategies are allowed. In fact, any heuristic with more than a quadratic complexity factor risks being longer than a naïve execution of an un-optimized query!

At the same time, availability of indexes and large join graphs present the opportunity for some amount of optimization.

Transformation-based optimization: Some optimization frameworks, like Volcano [6] and Cascades [5], perform a significant amount of exploration, where logical alternatives are discovered, before obtaining physical execution plans that could be evaluated. Note that if we decide to abort optimization (due to time limits or memory pressure) before implementation is well underway, we might not have a single plan to evaluate. To mitigate this problem with complex queries, some implementations of Cascades (e.g., Microsoft SQL Server) use optimization stages, which are performed sequentially. In a first stage, a heuristic join reordering is done, followed by very restricted exploration and implementation phases (e.g., the exploration phase does not consider join reordering, or only uses it in a very restricted manner). A later stage performs the exhaustive enumeration. In this case, if the later (and time-consuming) stage is aborted, the optimizer can always return the result of an earlier phase (which would always be available). It is therefore crucial that the plan found by the first phase is of reasonably good quality (since it will be returned if later phases are aborted), but also very quickly produced (since it is just an initial component of the whole optimization process).

As illustrated in the previous examples, new applications require efficient optimization heuristics that go beyond the traditional scenario that exclusively handles join predicates with no indexes. The main contributions of the paper are:

- We introduce the Enumerate-Rank-Merge framework (or ERM for short), that generalizes and extends previous join reordering heuristics in the literature (Section III).
- We extend previous work on heuristic join reordering to take into account other operators (Section IV).
- We design a workload generator that goes beyond previous proposals, and use it to comprehensively evaluate our techniques on synthetic as well as real data (Section V).

II. RELATED WORK

There has been significant research on techniques to exhaustively enumerate search spaces for relational query optimization. The pioneering work in [4] presents a dynamic programming algorithm to enumerate joins of a given query. This result was extended several times, culminating in a technique in [8] that efficiently enumerates all bushy trees with no cross products, and additionally considers outer-joins. A different approach is considered in [9], which presents
a top-down approach to exhaustively enumerate bushy trees with no cross-products. Finally, reference [10] introduces an exhaustive algorithm to enumerate all bushy trees including cross-products. Reference [11] extends the dynamic programming technique to additionally consider group-by operators. A different line of work in [6], [5] exhaustively enumerates joins by using transformations, and can be more easily extended to explore other relational operators.

Whenever certain query properties hold, there are polynomial algorithms that guarantee optimality. For instance, for chain queries, it is known that the classical dynamic programming algorithm works in polynomial time [1]. Also, if the join graph is acyclic and the cost model satisfies the ASI property [2], the IKKBZ technique in [12] returns the optimal plan in $O(N^2)$ where $N$ is the number of tables.

The join reordering problem has also received significant attention in the context of heuristic algorithms. One line of work adapts randomized techniques and combinatorial heuristics to address this problem. These techniques consider the space of plans as points in a high-dimensional space, that can be “traversed” via transformations (e.g., join commutativity and associativity). Reference [13] surveys different such strategies, including iterative improvement, simulated annealing, and genetic algorithms. These techniques can be seen as heuristic variations of transformation-based exhaustive enumeration algorithms. Another line of work implements heuristic variations of dynamic programming. These approaches include reference [14] (which performs dynamic programming for a subset of tables, picks the best $k$-table join, replaces it with a new “virtual” table, and repeats the procedure until all tables are part of the final plan), reference [15] (which simplifies an initial join graph by disallowing non-promising join edges and then exhaustively searches the resulting, simpler problem using [8]), and references [16], [17] (which greedily build join trees one table at a time).

Except for [16], [17], the above heuristics do not naturally fit the scenarios described in the introduction. Although more efficient than exhaustive search, these approaches are still expensive for low-latency scenarios, do not consider other relational operators beyond joins, and are unaware of indexes.

III. Enumerate-Rank-Merge Approach

In this section we introduce our main approach, which can be seen as a generalization of earlier greedy techniques (e.g., see [17], [16]). When trying to understand the different heuristics, it is useful to refer to Figure 1, which presents a generic pseudocode that is used, in one form or another, by virtually all greedy approaches. The input to the algorithm is a set of tables $T$ and a join graph $J$. Each table is associated with a base cardinality and, optionally, a selectivity of all single-table predicates applied to it. The join graph $J$ has associated a selectivity value to each join edge. In line 1 we initialize a pool of plans $P$, which consists of a Scan operator for each table $t \in T$. We then modify the pool of plans $P$ in lines 2-6, until there is a single plan remaining, which we return in line 7. Each iteration of the main loop starts by enumerating all valid ways to combine two plans in $P$ (enumerate step). Line 4 chooses the pair with the best ranking value (ranking step). Line 5 then combines the highest ranked pair of plans into a new plan $P_n$ (merge step). Finally, line 6 removes the original plans in the highest ranked pair from $P$ and adds the new plan $P_n$ into the pool. The invariant at all times is that any plan $p$ in $P$ is valid for the sub-query that contains all tables of the plans that were merged to obtain $p$. Therefore, at the end, when $|P| = 1$, the only plan in $P$ is valid for the original query.

The generic algorithm ERM is parameterized by three components. First, we need to determine which pairs of plans in $P$ are valid (enumerate). Second, we need to rank each valid pair of plans (rank). Third, we need to combine a pair of plans into a new plan (merge). The complexity of ERM is then $O(N^2R + NM)$, where $N$ is the number of tables, and $R$ and $M$ are, respectively, the complexities of a assigning a score to a single alternative, and merging a pair of plans (if $R$ is constant and $M$ is at most linear, the complexity of ERM is quadratic in the number of tables). Ranking has a factor $N^2$ because we can arrange the computation so that the first time we evaluate $N^2$ pairs, and then at each iteration we only compute the rank of each element in $P$ and the newly added $P_n$ (other ranks stay the same), which is linear per iteration. The following sections explain the components of ERM in detail.

A. Enumerate Step

The enumeration step determines the fundamental characteristics of plans that we consider. If we are only interested in linear trees, the valid function in line 3 only allows a pair $(P_1, P_2)$ if, after merging, there will be exactly one plan in the pool defined over more than one table. It is easy to see that this condition always results in linear trees. Similarly, if we do not want to consider cross products, the valid function would reject any pair $(P_1, P_2)$ for which there is no join predicate in $J$ between a table in $P_1$ and a table in $P_2$. If the valid function succeeds for every possible combination, we end up considering bushy trees with cross products. Previously proposed approaches use different versions of Enumerators: minSel [16] considers linear trees with no cross products, and Goo [17] considers bushy trees and cross products.

Linear vs. Bushy Trees

Since bushy trees include linear trees, the optimal bushy tree is never worse than the optimal linear tree. However,

1In this analysis we assume that the ranking of pairs is done by assigning a score to each pair, and then sorting by such score.
exhaustively enumerating all bushy trees is significantly more expensive than doing so for just the subset of linear trees. For that reason, query engines commonly restrict the search space to consider only linear trees. In the context of our greedy heuristics, the real advantage of bushy trees comes from the additional flexibility and potential to recover from mistakes. A problem with greedy techniques that only consider linear trees is that they are fragile with respect to initial bad choices.

**Example 1:** Suppose that we have a chain query \( R_1 \bowtie R_2 \bowtie R_3 \bowtie \ldots \bowtie R_n \), and suppose that \( R_1 \bowtie R_2 \) is the highest ranked join. Furthermore, suppose that \( R_2 \bowtie R_3 \) is a really bad choice, since it increases 100x the number of tuples in \( R_3 \). Suppose that we initially choose \( R_1 \bowtie R_2 \). Next, we have no choice but choosing \( R_1 \bowtie R_2 \approx R_3 \) to respect linear trees and no cross products (even though \( R_1 \bowtie R_2, R_3 \) is ranked badly, it is the only valid pair). The net effect is an explosion of intermediate results. Alternatively, if bushy trees are allowed, we can still first join \( R_1 \) and \( R_3 \), but we would have the option to choose other join sub-trees before joining the plans containing \( R_2 \) and \( R_3 \) (the final plan could be something like \( (R_1 \bowtie R_2) \bowtie (R_3 \bowtie \ldots \bowtie R_n) \)), therefore delaying the “bad join” as much as possible.

As we show experimentally, including bushy trees in a greedy search strategy greatly improves the quality of results.

**B. Ranking Step**

Consider a pair of plans \((P_1, P_2)\), and denote \( T(P) \) the set of tables referenced in plan \( P \). Further assume that \( J \) is the set of join predicates in the query. There are several ways to assign a score to \((P_1, P_2)\) to enable ranking of pairs. In this section we review some alternatives (we assume that smaller scores are better ranked), and in the next section we refine the ranking function so that it is aware of merging. We first present the rationale behind a good ranking function. The total cost of an execution plan \( p \) is the sum of costs of all operators in \( p \). Consider for simplicity a join tree composed exclusively of hash-based algorithms. Under a simple cost model, the cost of each hash join is proportional to the sizes of both its inputs and output. Since all plans would read the same amount of data from base tables, minimizing the cost of the query is very well aligned with minimizing the total sizes of intermediate results. The different metrics discussed below try to minimize the sum of intermediate results in different ways:

- **MinSel:** The score of pair \((P_1, P_2)\) is defined as \[
\frac{|p_{minSel}(P_1 \bowtie P_2)|}{\sum |p_{minSel}(P)|},
\]
  where the join predicate is the conjunction of all join predicates in \( J \) for which one table is in \( T(P_1) \) and the other table in \( T(P_2) \). In other words, it is the selectivity of the join predicates between \( T(P_1) \) and \( T(P_2) \). This ranking function is used in minSel [16].

- **MinCard:** The score of pair \((P_1, P_2)\) is defined as \[
|p_{minCard}(P_1 \bowtie P_2)|,
\]
  where the join predicate is the same as above. In other words, it is the cardinality of the intermediate result that joins all tables in \( T(P_1) \cup T(P_2) \). This ranking function is used in Goo [17].

- **MinMax:** The score of pair \((P_1, P_2)\) is defined as \[
|p_{minMin}(P_1 \bowtie P_2)|,
\]
  where the join predicate is defined as before, and \( rowLen(P) \) is the size, in bytes, of a row produced by plan \( P \). We slightly generalized MinCard into MinSize, which additionally takes into account the width of tuples produced by different operators, and more closely satisfies the original intent of minimizing intermediate result sizes.

We experimentally validated that MinSize is consistently better and more robust than the alternatives, and this is the ranking function we assume for the rest of the paper. It is important to note, though, that we can construct scenarios for which MinSel results in better results (but this rarely happens in practice, especially when considering bushy trees). The following example illustrates this fact.

**Example 2:** Consider a chain query \( A \bowtie B \bowtie C \bowtie D \), where \( |A| = |B| = 1K, |C| = |D| = 100, |A \bowtie B| = 1K \) (selectivity is \( 1/1K \)), \( |B \bowtie C| = x \) (selectivity is \( x/100K \)), and \( |C \bowtie D| = 200 \) (selectivity is \( 1/50 \)). Further assume, for simplicity, that all tuples have the same size. For \( x = 300 \) (selectivity is \( 3/1K \)), minSel would choose the order \((A \bowtie B) \bowtie C \bowtie D\), and the sum of intermediate results would be \( 1K + 300 + 600 = 1.9K \). In turn minSize would choose the order \((D \bowtie C) \bowtie B \bowtie A\), and the sum of intermediate results would be smaller at \( 200 + 600 + 600 = 1.4K \). Now, if we change \( x = 900 \) (selectivity is \( 9/1K \)), minSel would still choose the same join order, and the sum of intermediate results would be \( 1K + 900 + 1.8K = 3.7K \). When considering minSize, it will first choose \( C \bowtie D \) as before. After the first iteration, we have \( |C \bowtie D| = 200 \) and thus \( |(C \bowtie D) \bowtie B| = 1.8K \), which is larger than \(|A \bowtie B| = 900\). If we are restricted to linear trees, minSize would still end up choosing the same plan as before, and the sum of intermediate results would be \( 200 + 1.8K = 3.8K \) (which is worse than that of minSel). If bushy trees are allowed, however, minSize would choose \( A \bowtie B \) in the second iteration, and would return the final plan \((D \bowtie C) \bowtie (B \bowtie A)\), with a sum of intermediate results of \( 200 + 1K + 1.8K = 3K \) (better than minSel, which returns the same plan for either linear or bushy trees).

**C. Merging Step**

Once we decide in line 4 in Figure 1 the highest ranked pair of plans \((P_1, P_2)\), we have to merge them together in line 5 and replace the \( P_1 \) and \( P_2 \) with the merged plan \( P_n \) in line 6. Traditionally, the merge operation simply combines both execution sub-plans with a join alternative (i.e., \( P_3 = P_1 \bowtie P_2 \)), which always results in a valid plan. Note however that we can consider alternative execution plans, as long as we do it efficiently (as we discussed earlier, since there is a quadratic number of -constant time- ranking invocations, and a linear number of merges, we have a linear budget to implement better versions of merging without increasing the complexity of the algorithm). An interesting aspect of merging is that it is very flexible. If we think of a new way of merging two execution plans, we can add the new alternative and pick the best option.
overall. We next introduce our merging alternatives, assuming that the ranking function is \textit{minSize}.

\textit{Switch-HJ}

This is a very simple merge, motivated by our ranking functions not distinguishing between left and right inputs. In reality, it is better to have the smaller relation in the build side of a hash join. So when merging \( P_1 \) and \( P_2 \), we choose the alternative that uses the smaller input as the build side.

\textit{Switch-Idx}

If we are about to merge \( P_1 \) and \( P_2 \), where \( P_2 \) refers to a single table \( T_2 \), and there is an index on \( T_2 \) that enables an index-join strategy between \( P_1 \) and \( T_2 \), we consider this alternative in addition to the one that does not use indexes, and pick the one expected to be cheaper.

While this is an interesting merge that considers indexes, it still depends on the ranking function to choose the appropriate pair \( P_1 \) and \( P_2 \). In other words, we would only use an index alternative “reactively,” if by chance there was an available index for the highest ranked pair. A more robust alternative would be to bias the ranking function so that it takes into account the presence of indexes “natively”. At the same time, since ranking is done for each pair of plans, this modification to the ranking function needs to be very cheap to compute.

Suppose that we are considering a pair of plans \((P_1, P_2)\) where \( P_2 \) covers a single table \( T_2 \). Further suppose that \( P_1 \) has cardinality \( C \). An index-join alternative needs to perform an index lookup for each of the \( C \) outer tuples, and seek the index for matches (if the index is not covering, for each one of these matches there should be a clustered index lookup to obtain the remaining columns)\(^2\). Suppose that each index lookup fetches \( LT_2 \) tuples that fit in \( LP_2 \) pages from \( T_2 \). Then, the index-join would read \( C \cdot LP_2 \) pages (plus \( C \cdot LT_2 \) pages from the clustered index if the index is not covering), for the benefit of not reading \( T_2 \) altogether. The value \( \mathcal{P}_I \), defined as:

\[
\mathcal{P}_I = \max (0, \text{pages}(T_2) - C \cdot (LP_2 + \delta_{\text{cont}}) \cdot F \cdot LT_2)
\]

approximates the number of pages we would save (if any) by using an index-join alternative with index \( I \), where \( \text{pages}(T_2) \) is the number of pages of table \( T_2 \), \( \delta_{\text{cont}} \) is zero if the index is covering (and one otherwise), and \( F \) is the ratio of a random page read versus a sequential page read (to normalize \( \mathcal{P}_I \) values to “sequential page reads”). The value of \( F \) can be obtained by either using the optimizer’s cost model (which we did) or by a calibration phase. We then refine the \textit{minSize} ranking function as follows:

\[
\text{minSize}'(P_1, P_2) = \text{minSize}(P_1, P_2) - \text{pageSize} \cdot \mathcal{P}_I
\]

This function is still an approximation based on some simplifying assumptions, such as not modeling duplicate outer values or buffer pool effects. While all these could be accommodated by further refining the ranking function, our simple metric already provides good quality results.

\(*\text{The details are slightly more complex since we need to take into account the single-table predicates on table } T_2, \text{ which could be applied before the clustered index lookup. We omit such details for simplicity.}\)

\textit{Push}

The previous merging alternatives are simple and can be implemented in constant time (they only change the root of the merged tree). \textit{Push} is a merging alternative that sometimes is able to correct early “mistakes” that result from using a greedy heuristic. Consider merging \((P_1, P_2)\) and suppose that \( J \) is the set of join predicates between tables in \( P_1 \) and in \( P_2 \). As hinted by its name, the \textit{Push} alternative tries to \textit{push} one plan inside the other. Consider pushing \( P_2 \) into \( P_1 \). Without considering cross products, we can push \( P_2 \) to any subtree in \( P_1 \) that has a non-empty join predicate between the subtree and \( P_2 \) (Figure 2(a) shows the positions where we could push \( P_2 \) to \( P_1 \) if the join predicate between \( P_1 \) and \( P_2 \) is \( J = R_3 \bowtie R_5 \)). For each such sub-tree \( p \) of \( P_1 \), we replace \( p \) with an additional join operator that joins \( p \) and \( P_2 \) (see Figure 2(b) for an example).

Once we determine all subtrees for pushing \( P_2 \), we select the one with the highest aggregated rank (i.e., the sum of sizes for all intermediate results). Note that pushing \( P_2 \) into a sub-tree \( p \) in \( P_1 \) only changes the result size of all ancestors of \( p \) in \( P_1 \). This enables strategies that share most of the work for calculating aggregated ranks of all push alternatives. For instance, if we assume independence among join predicates, each pushing of \( P_2 \) into a sub-tree can be calculated in constant time, by precomputing cardinality information on each plan in the pool incrementally. There are two extensions to this basic approach. First, note that for each push, we can apply the \textit{Switch-HJ} and \textit{Switch-Idx} merges to each intermediate join in the path from the root to the pushed element. This is useful as changes in cardinality might make a index-join alternative much more useful than before the push. Note that, since \( P_2 \) into the root of \( P_1 \) is the same as joining both trees, the \textit{Push} strategy is effectively a generalization of the previous \textit{Switch-HJ} and \textit{Switch-Idx}. At the same time, the complexity of this alternative is larger than that of the original approach.

The benefit of the \textit{Push} technique is twofold. First, it might help correct early mistakes done by the greedy technique. For instance, suppose that \( A \bowtie B \bowtie C \) is better than \( A \bowtie C \bowtie B \) overall, but \( A \bowtie B \) is more expensive than \( A \bowtie C \) (this is the canonical adversarial scenario for a greedy technique). Our greedy heuristic would pick \( A \bowtie B \) first, obtaining a suboptimal plan. However, when later joining with \( B \), we can push \( B \) towards \( A \) and obtain the better plan again. The second (related) benefit of \textit{Push} has to do with join cycles, or residual predicates that operate over a subset of tables. Greedy heuristics are not aware that certain joins are beneficial because they are part of a larger join cycle (or a multi-table predicate) that would significantly reduce cardinality values. Therefore, greedy heuristics sometimes do not close cycles early enough. Whenever the join cycle is eventually closed, the \textit{Push} heuristic will attempt to push it down into the tree, possibly making a substantial reduction in intermediate cardinality appear earlier.
Pull

This strategy tries to correct missed opportunities of index-based joins. After merging \( P_1 \) and \( P_2 \), the resulting cardinality can decrease significantly (either as a result of the join or because of an additional residual predicate or join cycle). Then, some early decisions that discarded index joins because of large outer cardinalities can become interesting again. The Pull strategy operates on \( P_1 \bowtie P_2 \) as follows. It first considers each leaf node in the plan that is not already the inner side of an index-join alternative, and tries to “pull” the leaf node all the way up to the root of the plan to be used as the inner of an index join. Figure 3 illustrates how \( R_3 \) is pulled up to the root of the tree (it also shows that sometimes earlier joins that involved the pulled table must be converted in cross products). As before, every pull operation of node \( p \) changes the cardinality of all operators in the path from root to \( p \). By carefully maintaining cardinality information for all leaf nodes in the pool of plans, the pulling of each node can be done in constant time, and thus the Pull strategy is linear in the size of the tree (we omit details due to space constraints).

D. BSizePP: Heuristic join reordering

Figure 4 shows an instance of the ERM algorithm of Figure 1 (the name BSizePP is explained in Section V). We consider bushy trees and cross products, use the \( \minSize \) ranking function with the extensions in Section III-C to natively consider indexes, considers all valid pull and push merges and picks the one that minimizes the aggregated \( \minSize \) value. We will validate each of this choices experimentally in Section V.

IV. BEYOND JOIN REORDERING

In this section we extend our strategies, which heuristically reorder join graphs, to additionally handle other operators.

**A. Group-by Clauses**

After joins, group-by clauses are the most commonly used operator in SQL. A group-by clause is defined by (i) a set of grouping columns \( c \), (ii) a set of aggregate functions \( a \), and (iii) the relational input expression \( R \). We concisely write a group-by expression as \( \text{GB}(c, a, R) \), or simply \( \text{GB}(c, R) \) if the aggregate functions are not relevant for the discussion.

Group-by clauses can be done before or after joins in certain situations. Specifically, a group-by clause can be pulled up above a join, as long as (i) we add a key of the other join relational input to the set of grouping columns, and (ii) the join predicate is not defined over an aggregated column (otherwise the resulting join is not well-formed). Figure 5 illustrates this transformation (note that \( b_2 \) must include \( b_1 \) so that the original join is well-formed). A group-by clause can be pushed below a join \( R \bowtie S \) (down to \( S \)) whenever (i) the grouping columns include a key of \( R \), (ii) the columns from \( S \) in the join predicate are included in the grouping columns or derived from them via functional dependencies, and (iii) all aggregates are defined in terms of columns in \( S \) [11], [18].

**Canonical representation**

Join graphs are well-known canonical representations of SPJ queries. We now describe how to extend this concept for queries that additionally contain group-by clauses. The canonical representation of a generic query with joins and group-by
clauses is a join graph augmented with a set of canonical group-by clauses. Each canonical group-by clause is obtained from an original group-by in the query, and consists of (i) a set of grouping columns, (ii) a set of aggregate functions, and (iii) an input signature (instead of the relational input expression in regular group-by clauses). An input signature consists of a set of tables and optionally other canonical group-by clauses. Consider a logical operator tree representing a query with joins and group-by clauses. We first push all group-by clauses down joins as much as possible (reordering joins below group-by clauses if required). Since group-by clauses in the same query are either defined over disjoint relational inputs or else one is part of the other’s input, the result of this step is always unique and well defined. We then consider each group-by clause in the resulting tree, and convert it into a canonical group-by. The grouping columns and aggregate functions in the canonical group-by clause are identical to those in the original group-by clause. The input signature contains all tables and group-by clauses that are descendants of the original group-by clause and can be reached from it using a path in the query tree that does not go through another group-by clause. The join graph, in turn, is constructed as if no group-by clauses were present (for simplicity, cases which define joins between aggregated columns are discussed later in this section).

**Example 3:** Consider the query in Figure 6(a) and the corresponding operator tree representation in Figure 6(b). The inner group-by clause initially is defined by (i) columns key(A), B.1, D.3, (ii) the COUNT(\*) aggregate, and (iii) the input relation A \(\bowtie\) B \(\bowtie\) C \(\bowtie\) D. Note that we can push the group-by below A \(\bowtie\) \(A_{1\bowtie\bowtie\bowtie}\) (B \(\bowtie\) C \(\bowtie\) D) since k(A) and B.1 are part of the grouping columns. Then, the group-by is canonically defined as (i) the set of grouping columns [B.1, D.3], (ii) the aggregate, and (iii) the input signature [B, C, D]. We concisely write this as GB\(_1\) = GB([B.1,D.3],[B,C,D]). Using a similar argument, the canonical representation of the outer group-by clause is GB\(_2\) = GB([E.3],[A, GB\(_1\), E]). Figure 6(c) shows the canonical representation of the query above.

It is important to note that we were able to obtain a single join graph even though the original query had two join components “separated” by a group-by clause. This would allow to reorder joins across group-by clauses, which is not possible in previous approaches. The canonical representation of a query allows identifying the locations in a join tree where each group-by clause can be inserted. Specifically, we can insert a group-by G on top of a sub-tree T whenever T contains at least all tables and group-by clauses in G’s input signature. In such case, the set of grouping columns in the resulting group-by clause would additionally include keys for any table or group-by clause in T that is not present in the “transitive closure” of the input signature of G (i.e., not present in G’s input signature, in input signatures in G’s input signature, and so on).

**Example 3:** (continued) Consider a sub-tree T = A \(\bowtie\) B \(\bowtie\) C \(\bowtie\) D \(\bowtie\) E. We can insert a root group-by G\(_1\) (see Figure 6) on top of T because T includes all tables in G\(_1\)’s input signature. We have to include in G\(_1\)’s grouping columns a key for A and E (which are not in the input signature of G\(_1\)). Group-by G\(_2\) can now be placed on top of the resulting sub-tree because G\(_1\), A and E are present in the sub-tree. The grouping columns in G\(_2\) do not need to be augmented, since all tables and group-by clauses in the sub-tree appear in either G\(_2\)’s input signature, or in G\(_1\)’s input signature (which belongs to G\(_2\)’s input signature).

**Selectivity Estimation of Group-By Clauses**

When considering a query that only contains joins, we can associate a selectivity value with each join because this value is independent of the order of the join in an execution tree. No matter where a join is positioned in an execution plan, the selectivity of the join would always be the same\(^3\). We now extend this idea to queries that also contain group-by clauses.

We define the selectivity of a group-by G as the number of output tuples divided by the number of input tuples. In this work we do not take a position on how to calculate this selectivity value (in general it would involve using statistical formulas and exploiting histogram information). Instead, we show that the selectivity of a group-by clause is independent of its position in an execution plan.

Suppose that there are |A| tuples in A and |B| tuples in B. Furthermore, assume that there are dv\(_A\) distinct values of a in A, each one repeating D\(_A\) times, so |A| = dv\(_A\) * D\(_A\) (this is a simplification, but is the information that we have per histogram bucket on A.a, and we can extend the discussion to hold over all buckets). Similarly, there are dv\(_B\) distinct values of b in B, each one repeating D\(_B\) times, so |B| = dv\(_B\) * D\(_B\). Additionally, each distinct value of b in B (i.e., these D\(_B\) tuples) contain distinct values for b’ (recall that the columns in b’ includes those in b, so it tuples that share b values generally vary in columns b’ – b). Let’s say that there are dv\(_{B’}\) distinct values of b’ in these D\(_B\) tuples, each one repeating D\(_{B’}\) times, so D\(_B\) = dv\(_{B’}\) * D\(_{B’}\) (again, there is an implicit assumption about independence of column distributions here, which is typically used during cardinality estimation). With this notation, we can calculate cardinality and selectivity values in presence of both joins and group-by clauses. Specifically:

1) |GB(b’, B)| = dv\(_B\) * dv\(_{B’}\) (the group-by clause removes all D\(_{B’}\) copies).
2) Sel(GB(b’, B)) = |GB(b’, B)|/|B| = 1/D\(_B’\).
3) |A \(\bowtie\) B| = min(dv\(_A\), dv\(_B\)) * D\(_A\) * D\(_B\) (using the containment assumption for joins).
4) |A \(\bowtie\) (GB(b’, B))| = min(dv\(_A\), dv\(_B\)) * D\(_A\) * dv\(_{B’}\) (using the containment assumption again, but now each group of D\(_B\) tuples was reduced to only dv\(_{B’}\) tuples due to the group-by clause).
5) |A \(\bowtie\) (GB(b’, key(A)))| = |GB(b’, key(A), A \(\bowtie\) B)|/|A \(\bowtie\) B| = 1/D\(_B’\).

\(^3\)This statement leverages the standard independence assumption commonly used for cardinality estimation.
virtual group-by nodes are not executable leaf-nodes (as Scan by plan P group-by node for each canonical group-by in the query. These result in a well-formed execution plan). Additionally, we only group-by nodes together (as explained before, it does not top of a valid execution sub-plan. These nodes are handled as before (note that these input plans can have a group-by operator as the root node, but the pairing would result, as before, in a join or cross product).

**Ranking (line 4):** The ranking functions introduced in Section III-B are easily extended to deal with group-by clauses. For instance, the value \( \text{minSize}(G, P_1) \) is the output size of applying the group-by operator \( G \) on top of \( P_1 \). This value can easily be computed based on the cardinality of \( P_1 \) and the selectivity of \( G \), which, as we saw earlier, does not depend on the operator placement in the tree.

**Merging (line 5):** The naïve merging operation is trivially extended so that when combining a virtual group-by \( G \) with a non-virtual plan \( P \), the generated plan consists of the group-by \( G \) (augmenting the grouping columns appropriately) on top of \( P \), *Switch-Idx* and *Switch-Ids* only operate on the root of the tree, so they are not applicable when one of the plans in the pair is a virtual group-by-operator. Push and Pull operations require a bit more care, but are also simple to extend. The main challenge is to prevent invalid trees from being generated through merges. Consider the pull operation of a table \( T \) that is in the input signature of some intermediate group-by. In this case, simply pulling the table to the root of the tree would generate an invalid plan with a group-by that does not operate on its input signature. Instead, when the pulled table appears in the input signature of an intermediate group-by \( G \), we pull, along with the table, any dependent group-by-operator as well to the top of the tree. Similarly, consider the push operation when applied to a virtual group-by node \( G \). In addition to generating the group-by at the root of the new tree, we try to push the group-by down whenever possible, and pick the best alternative. At a high level, the main extensions require us to re-evaluate the ranking function of alternative plans, and pick the one with the best ranking as the resulting merging plan.

**Additional Details**

For simplicity, we have not discussed a special case when handling join predicates defined on aggregate columns of group-by clauses, as illustrated in the example below:

```sql
SELECT *
FROM T1, (SELECT SUM(x) AS SX FROM TX) T2
WHERE T1.X = T2.SX
```

We cannot join \( T1 \) and \( T2 \) until after doing the aggregation on \( T2 \) (otherwise, the join is not well-defined). In general, this situation results in a dependency graph between joins and
group-by nodes, where the application of some operator is blocked until other operators unlock it. Earlier in this section, we discussed the case of group-by operators being blocked by their input signatures. The same ideas and solutions can be applied in this case, where group-by clauses block joins. A suitable extension of the enumerate step in line 3 of Figure 1 is required, but we omit such details due to space constraints.

B. Semi-join operators

Although semi-joins (⋈) are not directly specified in SQL queries, they are very useful relational operators because (i) semi-joins can be used to handle nested sub-queries with optional correlation, (ii) there exist efficient set- and tuple-oriented implementations of semi-joins, and (iii) algebraically they can be expressed as group-by clauses and joins, which enables additional optimization alternatives. To make the last item concrete, the traditional algebraic equivalence that relates semi-joins with joins and group-by clauses is as follows:

\[
A \Join B = \text{GB}(\text{key}(A), A \bowtie B)
\]

We use this equivalence to produce a canonical representation of a query that contains joins, group-by clauses and semi-joins. Suppose that the semi-join predicate in the equality above is \(A \bowtie_{a \bowtie b} B\). This equality results in the functional dependency \(\text{key}(A) \rightarrow b\) and thus in \(\text{GB}((\text{key}(A), b), A \bowtie B)\), which can then be rewritten by pushing the group-by clause down the join as in \(A \bowtie_{a \bowtie b} (\text{GB}(b, B))\). Now consider a slightly different example without an equi-join, namely \(A \bowtie_{a \bowtie b} B\). This is equivalent to the expression \(\text{GB}((\text{key}(A), a), A \bowtie B)\) but we cannot push the group-by down the join since column \(b\) in the join predicate is not a grouping column. As a more complex example, consider \(A \bowtie_{a \bowtie b} (B \bowtie_{b \bowtie 2c} C)\). In this case, the transformation results in \(\text{GB}((\text{key}(A), a), A \bowtie B \bowtie c)\), which, by a similar reasoning as in the first case above, can be pushed down through the first join into \(A \bowtie_{a \bowtie b} GB(b, B \bowtie 2c \bowtie C)\).

In general, to obtain a canonical representation for a subplan like \(R \bowtie T\) we replace the \(\bowtie\) with a \(\bowtie\) and add a group-by with \(\text{key}(R)\) as the grouping columns. We then proceed as before, by pushing the group-by as much as possible and obtaining the join graph and canonical group-by clauses. As a final example consider expression \(A \bowtie_{a \bowtie b} GB(b, B \bowtie 2c \bowtie C)\). We translate it to \(\text{GB}((\text{key}(A), a \bowtie B \bowtie C), \text{key}(A) \rightarrow a\), which is equal to \(\text{key}(B), \text{key}(B) \rightarrow b\) which is equal to \(c\), so this is actually \(\text{GB}((\text{key}(A), \text{key}(B), c), A \bowtie B \bowtie C)\), and now we can push the group-by below joins to result in \(A \bowtie B \bowtie GB(c, C)\). The canonical representation contains the join graph \(A \bowtie B \bowtie C\) and the canonical group-by clause \(\text{GB}(c, C)\).

Integrating Semi-joins

Since we transform semi-joins into joins and group-by clauses, our algorithms can handle semi-joins with no modifications. While this approach is correct, it has a drawback: group-by clauses obtained from semi-joins have sometimes special properties. For instance, often there is a single table in the right side of the semi-join, and if there are indexes, we can use efficient index-based execution plans without actually performing the group-by and join. Below we outline some suggestions to address this issue.

Leveraging that a semi-join results into a join plus a group-by, the ranking function can have certain limited look-ahead capabilities. In general, if there are no aggregate functions defined on a group-by clause over a join, we can attempt to apply the join and the corresponding group-by together using an index-based strategy when ranking the pair that contains the join. Of course, merging should be made aware of the look-ahead mechanism and implement the optimized plans whenever possible. The main challenge in this approach is that the merge operators (e.g., Pull and Push) need to be aware of new physical operators that we might introduce in specialized plans. We believe that this is not a straightforward task, and we consider this form of tighter integration of semi-joins with our Enumerate-Rank-Merge framework as part of future work.

C. Other operators

In addition to joins, semi-joins and group-by clauses, there are other operators in SQL that need to be handled. In this section we briefly comment on how we can approach such operators\(^4\). In general, these operators are handled independently of the join/semi-join/group-by components, either using pre- or post-processing, as explained next.

Anti-Joins: As far as we know, we cannot freely reorder left and right children of anti-joins as we did for joins and semi-joins. Conceptually, \(R \bowtie_{a=b} S\) is evaluated by considering each row in \(R\) and processing it according to \(S\) (in the example, it would return the tuple from \(R\) if there is no tuple in \(S\) that satisfies predicate \(a=b\)). This procedure can be seen as evaluating an expensive predicate for each row of \(R\). In fact, the cost per tuple in \(R\) depends on \(S\) and, according to cardinality estimation, only a fraction of tuples from \(R\) would pass the anti-join. To deal with queries that include anti-joins, we first obtain a heuristic execution plan for the query that omits the anti-join. We then identify the most specific point in the query plan in which the anti-join can be evaluated, and use the heuristic techniques in \(\cite{20, 21}\) to obtain the most appropriate place to position the anti-join in the tree.

Outer-Joins: We pre-process a query with outer-joins (denoted as \(\Rightarrow\)) by delaying them as much as possible (and therefore obtaining larger join components). We do this using the algebraic equivalence \(R \bowtie (S \Rightarrow T) = (R \bowtie S) \Rightarrow T\) and pull outer-joins above joins as much as possible before applying the ERM approach to the join component. A full integration of outer-join reordering is part of future work.

Other operators: We do not specifically handle other operators (such as unions or full outer-joins), and leave them as in the original query.

D. Putting it All Together

Out technique to heuristically optimize queries is as follows:

\(^4\)An alternative, more systematic approach that adapts work on anti-join and outer-join reordering (e.g., \(\cite{8, 19}\)) is part of future work.
1) Analyze the query and transform it into a set of join/semi-join/group-by canonical components (with associated anti-join expensive predicates), and connected among each other via (delayed) outer-joins, unions, and any other un-handled operators.

2) Apply ERM to each component.

3) Greedily place anti-joins in each component.

4) Use default implementations for outer-joins, unions, and any other un-handled operators.

V. EXPERIMENTAL EVALUATION

We now report an experimental evaluation of the techniques described in this paper. We implemented a C# prototype that enables us to compare different optimization variants. Each experiment is specified by three components: a query workload, an index configuration, and a set of optimizers to evaluate. We next detail each component in detail.

A. Query Workloads

A query is specified by a set of tables (with selectivity values given by single-table predicates), a join graph with associated selectivity values, and a set of group-by clauses.

Tables: We generate table cardinalities by following Zipfian distributions. The parameters of the table generation routine are number of tables NT and the Zipfian parameter Z. A value Z = 0 generates tables with the same cardinality, while a value Z = 5 generates 10 tables with the following cardinalities (for a total 100M tuples): (96440632, 3013770, 396875, 94180, 30861, 12403, 5738, 2943, 1633, 964) Alternatively, we used two additional data generation routines: random (which generates tables with random cardinality) and stratified (which was previously proposed in the literature [13] and generates tables of size 10-100 with 15% probability, size 100-1K with 30% probability, size 1K-10K with 25% probability, and size 10K-100K with 30% probability). Finally, a boolean parameter, sorted, specifies whether the tables are sorted by cardinality or are instead shuffled during query construction (this is important due to join and filter generation, explained next).

Join Graphs: These are generated using the following parameters:

- J: Number of joins. The first NT − 1 joins form a spanning tree, and remaining joins introduce cycles.
- 0 ≤ JL ≤ JR ≤ 1, which control the graph topology. The i-th join (for i < NT) is done between the i-th table (in the order produced during the table generation) and a random table between JL*(i-1) and JR*(i-1). This procedure generalizes previous approaches in the literature to generate interesting join topologies. For instance, for JL=JR=0 we obtain star topologies, for JL=JR=1 we obtain chain topologies, and for other values, such as JL=0 and JR=0.3 we obtain snowflake-like topologies. After NT − 1 joins, we randomly join pairs of tables generating cycles.
- JoinType, which is a categorical variable which determines the cardinality of the join. It could be random (which assigns a random selectivity value to the join), Min/Max/MinMax (which assigns the cardinality of the smallest/largest table, or a number in between), or N−M (which models n-m joins by picking a number of distinct values in each table and having the join selectivity be 1/max(dv1, dv2) as in [13]). We can specify a different distribution for the first NT − 1 joins and the rest.

Table filters: So far each table is joined without applying any filter on it, which is fine if no indexes are present (since we always need to read the full table). When indexes are present, tables with filters can result in big differences. We specify table filters by a pair of values (0 ≥ lo ≥ hi ≥ 1) specifying the selectivity of the filter, and a choice of tables to filter, which could be either a random fraction of the tables, the value First (which filters the fact table in a snowflake topology), or Leaves (which filters dimension tables in snowflake topologies).

Group-by clauses: Group-by clauses are generated by parameters G (the number of group-by clauses) and T (the number of random tables that are required in the group-by input signature).

![Fig. 7. Synthetically generated queries.](image)

An Example

The generation procedure described above is rather flexible, and many combinations are possible. We can produce star, linear, and snowflake topologies respecting foreign key constraints, as well as cycles simulating M-N joins. Figure 7 shows examples generated with this approach. We generated the snowflake query in Figure 7(a) using NT = 10 tables with Z = 4 tuple distribution (sorted by cardinality), a join graph with 10 joins (the first 9 using JL = 0, JR = 0.5 and MinMax selectivity values, and the last one introducing a cycle with N-M selectivity), and filter predicates on the fact table. Figure 7(b) shows a chain query generated with NT = 5 tables with Z = 0.5 tuple distribution (shuffled tables), a join graph with 4 joins using JL = JR = 1 and Max selectivity values, and a group-by clause over two tables.
B. Index Configurations

A configuration is defined a set of indexes. Each index consists of a leading column and a boolean value which specifies whether the index is covering during an index lookup. A configuration is generated by two parameters: \textit{ProbIndex} (which gives the probability that an join column or single table predicate has a backing index) and \textit{ProbCover} (which gives the probability of an index being covering).

C. Optimizers

To simplify the presentation, we encode an optimizer by a three-part name \textit{E-R-M}, where:

- \textit{E} identifies the search space, which can be \textit{L} (linear trees with no cross products) or \textit{B} (bushy trees with cross products).
- \textit{R} identifies the ranking function and can be \textit{Sel} (rank by \textit{minSel}), \textit{Size} (rank by \textit{minSize}), and \textit{ST} (rank by \textit{smallestTable}3).
- \textit{M} identifies the merging approach. It could be empty if only the trivial merge is used, + if \textit{Switch-HJ} and \textit{Switch-Ids} are used, and \textit{PP} if additionally \textit{Pull/Push} is used.

Using this notation, our experiments use subsets of the following optimizers: \textit{LSel}, \textit{LSel+}, \textit{LST}, \textit{LST+}, \textit{BSel}, \textit{BSel+}, \textit{BSize}, \textit{BSIZE}, \textit{BSIZEPP}. We additionally implemented the following exhaustive optimizers:

- \textit{LEX}: Exhaustive optimizer that considers linear trees and no cross products.
- \textit{BEX}: Exhaustive optimizer that considers bushy trees and cross products.
- \textit{LIKKBZ}, \textit{LIKKBZ+}: Optimizer that returns optimal linear trees for acyclic queries and no indexes, augmented with minimum spanning trees for cyclic queries [12].

D. Metrics

For a given workload, index configuration, and set of optimizers, we proceed as follows. We optimize each query by all optimizers. We cost each resulting execution plan using Microsoft SQL Server’s cost model. We then divide the cost of each plan by the cost of the best plan found by any optimizer in our set. The value that we assign to a given optimizer is this ratio, which goes from one (optimal) upwards, and quantifies how much worse the solution is compared to the optimal one. After repeating this procedure for all queries in a workload, we report the minimum, maximum, average, median, and the 95%-quantile of the ratio for each optimizer. These quality numbers are precise if an exhaustive optimizer is in the mix, since the optimal execution plan would be found. While we can evaluate exhaustively for small number of tables (<15), it is not possible to do so for a large number of tables. In that case we do not know what the optimal plan is, but we always include \textit{LIKKBZ} in the set of optimizers (which is guaranteed to produce optimal linear trees for acyclic queries, and thus is a baseline).3

E. Summary of Results

The sections below show a sample of experiments performed on a 3GHz dual core machine. We focused on general trends and insights, and omit several additional variants that we tried due to space constraints.

Execution times

We first report execution times for the techniques discussed in this paper. Note that our prototype is not optimized for speed (instead, we tried to reuse as much code as possible to implement all variants, which results in performance limitations). Therefore, performance numbers should be interpreted as trends rather than in absolute terms. Figure 8 shows elapsed times for all techniques (note the logarithmic scale). All heuristic implementations show the same trend and finish in less than 200 milliseconds for 60 tables, and in tens of milliseconds for up to 30 tables (query graphs were generated randomly in this experiment, as we just wanted to measure average performance). In contrast, exhaustive techniques quickly become too slow over 10 seconds for as little as 13 tables. Also note that exhaustively exploring bushy trees can be an order of magnitude slower than just exploring linear trees. The results in Figure 8 do not consider indexes. We ran the experiment again but adding 50% of relevant indexes to the physical design, and the results (while slower than the ones reported in the figure) showed the same trends.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig8}
\caption{Performance of different optimization alternatives.}
\end{figure}

Quality results

We first generated a workload with \textit{NT} = 10 tables with \textit{Z} = 4 cardinalities (not sorted by cardinality), and a chain join topology (\textit{JL = JR = 1}) where each join has a cardinality given by \textit{MinMax}. We did not consider any index. Figure 9 shows the quality metrics for all optimizers (ordered by average ratio). We make the following observations:

1) \textit{BEX} is (obviously) the best strategy in all cases.
2) With the exception of \textit{BSel}, all bushy strategies are better than linear strategies. This is a direct consequence of the join topology, which is very restrictive and does not give flexibility to recover from large intermediate results.
3) \textit{LEX} is on average 4.2 times (up to 35.2 times) worse than \textit{BEX}, which shows that bushy trees can be substantially better than optimal linear trees. It is interesting to note that most bushy heuristics result in better plans than that of the exhaustive linear tree.
4) LIKKBZ+ has almost the same quality as LEx (the slight difference comes from small variations in the cost model used by IKKBZ [12]).
5) Simple merging variants (+) generally pay off, resulting in better quality than the corresponding technique with naive merging.
6) Our technique BSizePP performs on average 6% worse than the optimal bushy strategy (and 32% worse in 95% of cases). The advanced merging operations Push and Pull did not additionally improve quality over BSize+.

<table>
<thead>
<tr>
<th>Avg</th>
<th>Min</th>
<th>Median</th>
<th>95%</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEx</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>BSizePP+</td>
<td>1.600</td>
<td>1.000</td>
<td>1.000</td>
<td>1.328</td>
</tr>
<tr>
<td>BSizePP+</td>
<td>1.600</td>
<td>1.000</td>
<td>1.000</td>
<td>1.328</td>
</tr>
<tr>
<td>BSize+</td>
<td>1.367</td>
<td>1.000</td>
<td>1.131</td>
<td>2.462</td>
</tr>
<tr>
<td>BSize+</td>
<td>3.764</td>
<td>1.000</td>
<td>1.509</td>
<td>15.184</td>
</tr>
<tr>
<td>LEx</td>
<td>4.288</td>
<td>1.000</td>
<td>1.533</td>
<td>17.345</td>
</tr>
<tr>
<td>LIKKBZ</td>
<td>4.295</td>
<td>1.000</td>
<td>1.533</td>
<td>17.345</td>
</tr>
<tr>
<td>LIKKBZ</td>
<td>4.295</td>
<td>1.000</td>
<td>1.533</td>
<td>17.345</td>
</tr>
<tr>
<td>BSel+</td>
<td>11.241</td>
<td>1.033</td>
<td>3.156</td>
<td>51.432</td>
</tr>
<tr>
<td>BSel+</td>
<td>12.074</td>
<td>1.005</td>
<td>4.169</td>
<td>43.504</td>
</tr>
<tr>
<td>LST+</td>
<td>12.331</td>
<td>1.000</td>
<td>3.797</td>
<td>50.236</td>
</tr>
<tr>
<td>LST+</td>
<td>13.614</td>
<td>1.037</td>
<td>4.779</td>
<td>50.371</td>
</tr>
<tr>
<td>LST</td>
<td>45.256</td>
<td>1.012</td>
<td>12.420</td>
<td>177.877</td>
</tr>
</tbody>
</table>

Fig. 9. 10-table queries with no indexes.

We repeated the experiment increasing the number of tables from 10 to 50. In this case, we could not obtain results for LEx or BEx. However, LIKKBZ (which for acyclic queries and no indexes is almost the same as LEx) was included in the set of optimizers. The results are shown in Figure 10. We see that results are amplified, and LIKKBZ+ (which would be almost the same as an exhaustive optimizer considering linear trees) is, on average, 62 times worse than BSizePP (and up to 477 times worse overall). We also see that BSizePP consistently returns the best quality among the heuristics.

<table>
<thead>
<tr>
<th>Avg</th>
<th>Min</th>
<th>Median</th>
<th>95%</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSizePP+</td>
<td>2.295</td>
<td>1.000</td>
<td>1.533</td>
<td>17.345</td>
</tr>
<tr>
<td>BSizePP+</td>
<td>1.004</td>
<td>1.000</td>
<td>1.000</td>
<td>1.320</td>
</tr>
<tr>
<td>LIKKBZ+</td>
<td>62.615</td>
<td>1.000</td>
<td>20.991</td>
<td>279.523</td>
</tr>
<tr>
<td>LST+</td>
<td>203.410</td>
<td>1.000</td>
<td>73.013</td>
<td>1056.145</td>
</tr>
<tr>
<td>LST+</td>
<td>218.113</td>
<td>1.000</td>
<td>74.098</td>
<td>814.333</td>
</tr>
</tbody>
</table>

Fig. 10. 50-table queries with no indexes.

**Indexes**

We repeated the previous experiments with three variations. First, we added covering indexes for 25% of all join predicates. Second, we added single-table predicates with 1% to 10% selectivity to half the tables in the query. Finally, we added 2 additional join predicates (resulting in join cycles). The results of this experiment, with NT = 10 tables, is shown in Figure 11. In this case, we observe:

1) BEx is optimal, and LEx is a close second. This shows that the presence of indexes changes the landscape of optimal plans in subtle ways.
2) In general, non-trivial merges (+) result in better quality than when using naive merges due to indexes.
3) Our proposed technique BSizePP results in 25% average degradation compared to the optimal exhaustive optimizer, and is 2.7 times worse for 95% of the queries.

<table>
<thead>
<tr>
<th>Avg</th>
<th>Min</th>
<th>Median</th>
<th>95%</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSizePP+</td>
<td>1.011</td>
<td>1.000</td>
<td>1.000</td>
<td>1.042</td>
</tr>
<tr>
<td>BSize+</td>
<td>1.913</td>
<td>1.000</td>
<td>1.071</td>
<td>6.622</td>
</tr>
<tr>
<td>LIKKBZ+</td>
<td>8.571</td>
<td>1.000</td>
<td>1.030</td>
<td>18.617</td>
</tr>
<tr>
<td>LST+</td>
<td>3.618E+11</td>
<td>1.000</td>
<td>2.121</td>
<td>6841.553</td>
</tr>
</tbody>
</table>

Fig. 11. 10-table chain with 50% indexes and join cycles.

4) Previous techniques proposed in the literature (LST, LST, and BSize can be hundred times worse (and even thousands in the worst case) than our approach.

As in the previous experiment, we increased the number of tables from 10 to 50. The results are summarized in Figure 12. The two main observations from this figure is (i) BSizePP consistently ranks the best among polynomial heuristics (including those that return optimal plans for acyclic join trees), and (ii) the quality of the other heuristics is not consistent across workloads (contrast LST+ and BSize+ in Figures 10 and 12). As additional supporting evidence for these two claims, Figure 13 shows the results of changing the join topology to snowflake (JL = 0, JR = 0.5) and introducing filter predicates (with selectivity between 1% and 50%) to all dimension tables. In this case, BSizePP is still the overall winner, while LST performs really poorly for the bottom 5-percentile (note that LST was the second-best in Figure 12).

**Group-by Clauses**

To evaluate group-by clauses we generated NT = 10 table queries (Z = 4, not sorted by cardinality) with a join cycle (JL = JR = 1 for the first 9 joins, and the last join closing the loop). Additionally, we randomly chose a connected component of size 3, and use it as the group-by clause with a selectivity value between 5% and 95%. Figure 14 shows the result of this experiment comparing LEX, BEx, BSizePP, and two new optimizers: LExNoGB and BExNoGB. These last two versions do not attempt to reorder group-by clauses. Instead, they perform exhaustive join reordering of the connected join.
components (separated by group-by clauses) in the query. This represents the best possible behavior of a heuristic that does not consider group-by clauses. \( BSizePP \) results in execution plans that are 15% worse than the optimal on average (up to 6 times worse in the worst case), which coincidentally is almost the same performance that the exhaustive \( LEx \). The alternatives that do not handle group-by clauses are not robust. While for 50% of the queries they perform very well (6% and 33% degradation with respect to the optimal plan), there are situations in the bottom 5-percentile where the techniques are tens of thousands of times worse than the optimal. The reason for this behavior is that in some situations, the 3-way join that covers the group-by tables results in a very big intermediate result. By pulling the group-by above other joins, and then reordering a larger join sub-graph, we can significantly reduce the cardinality of intermediate results. This can be seen as a group-by clause artificially restricting the search space of join orders to be forced to go through a specific intermediate result (the subset of tables in the group-by clause), which in general can be a very bad choice. This behavior for \( LExNoGB \) and \( BExNoGB \) is common for the cycle join topology that we chose, but happens in a much lest extreme way for other workloads (we omit these results due to space constraints).

<table>
<thead>
<tr>
<th></th>
<th>Avg</th>
<th>Min</th>
<th>Median</th>
<th>95%</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>( BEx )</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>( LEx )</td>
<td>1.147</td>
<td>1.00</td>
<td>1.005</td>
<td>2.083</td>
<td>5.893</td>
</tr>
<tr>
<td>( BSizePP )</td>
<td>1.155</td>
<td>1.00</td>
<td>1.004</td>
<td>1.733</td>
<td>6.021</td>
</tr>
<tr>
<td>( BExNoGB )</td>
<td>17282.120</td>
<td>1.00</td>
<td>1.064</td>
<td>16875.686</td>
<td>408364.416</td>
</tr>
<tr>
<td>( LExNoGB )</td>
<td>18290.878</td>
<td>1.00</td>
<td>1.332</td>
<td>17793.046</td>
<td>1415005.930</td>
</tr>
</tbody>
</table>

Fig. 14. 10-table join cycle with a group-by clause.

Real Data

We also experimented on real data and workloads. As an example, Figure 15 shows the estimated cost of different techniques for a data warehouse query that joins 15 tables together using a shallow snowflake schema, and two different index configurations (\( C_{lo} \) and \( C_{hi} \) correspond to lightly and heavily indexed configurations, respectively). We see that \( BSizePP \) is very close to the optimal plans returned by \( BEx \) and \( LEx \) in both cases, and improves over the basic \( BSize+ \) that does not consider \( Pull \) and \( Push \) operations. We also see that other heuristics are not robust (e.g., both \( LSel+ \) and \( LST+ \) range from close to \( BSizePP \) to being the worst alternative for some index configuration.

<table>
<thead>
<tr>
<th></th>
<th>( C_{lo} ) (opt. time)</th>
<th>( C_{hi} ) (opt. time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( BEx )</td>
<td>7.86</td>
<td>1.64</td>
</tr>
<tr>
<td>( LEx )</td>
<td>7.86</td>
<td>1.09</td>
</tr>
<tr>
<td>( BSizePP )</td>
<td>7.89</td>
<td>1.14</td>
</tr>
<tr>
<td>( LIKKIBiz+ )</td>
<td>7.91</td>
<td>3.08</td>
</tr>
<tr>
<td>( LST+ )</td>
<td>7.90</td>
<td>3.34</td>
</tr>
<tr>
<td>( LSel+ )</td>
<td>7.89</td>
<td>3.68</td>
</tr>
<tr>
<td>( LST+ )</td>
<td>908.72</td>
<td>1.58</td>
</tr>
</tbody>
</table>

Fig. 15. 15-table snowflake for using real data.

VI. CONCLUSIONS

Motivated by new requirements for query processing techniques, in this paper we introduced polynomial-time heuristics to optimize SQL queries. Our techniques generalize previous approaches in the literature in a number of aspects. First, we introduce the notion of merging, which can be seen as a corrective action that helps recovering from early mistakes in the greedy techniques. Second, we extend our techniques to handle group-by clauses, semi-joins, and the availability of indexes. Finally, we designed a realistic workload generator and validated our hypothesis on both synthetic and real data.

REFERENCES