Optimizing Iceberg Queries with Complex Joins*

Brett Walenz
Duke University
bwalenz@cs.duke.edu

Sudeepa Roy
Duke University
sudeepa@cs.duke.edu

Jun Yang
Duke University
junyang@cs.duke.edu

ABSTRACT

Iceberg queries, commonly used for decision support, find groups whose aggregate values are above or below a threshold. In practice, iceberg queries are often posed over complex joins that are expensive to evaluate. This paper proposes a framework for combining a number of techniques—a-priori, memoization, and pruning—to optimize iceberg queries with complex joins. A-priori pushes partial `GROUP BY` and `HAVING` condition before a join to reduce its input size. Memoization caches and reuses join computation results. Pruning uses cached results to infer that certain tuples cannot contribute to the final query result, and short-circuits join computation. We formally derive conditions for correctly applying these techniques. Our practical rewrite algorithm produces highly efficient SQL that can exploit combinations of optimization opportunities in ways previously not possible. We evaluate our PostgreSQL-based implementation experimentally and show that it outperforms both baseline PostgreSQL and a commercial database system.

1 Introduction

Iceberg queries, first introduced by Fang et al. [8], are an important class of queries in OLAP and data mining. These queries perform grouping of data and return only those groups who aggregate values pass a given threshold. The name "iceberg" follows from the observation that while the number of candidate groups is potentially huge (like an iceberg), the number of groups in the final answer tends to be much smaller (like the tip of the iceberg), because the threshold is typically very selective. Therefore, the strategy of first computing the entire set of candidate groups and then filtering them may not work well. Below is a simple example iceberg query from [8], which finds popular items and regions where the revenue in the region from the item is at least one million dollars:

```
SELECT partKey, region, SUM(numSales * price)
FROM LineItem
GROUP BY partKey, region
HAVING SUM(numSales * price) >= 1000000;
```

Many iceberg queries that arise in practice involve joins between two or more tables. One example that has been studied in depth is *market basket* or *frequent itemset* queries, which find subsets of items purchased together more than some number of times. Listing 1 shows one such query that finds pairs of items that appear together in at least 20 baskets. Here, table `Basket(bid, item)` stores the items contained in each basket (identified by `bid`), one item per record.

```
SELECT i1.item, i2.item
FROM Basket i1, Basket i2
WHERE i1.bid = i2.bid
GROUP BY i1.item, i2.item HAVING COUNT(*) >= 20;
```

Listing 1: A market basket query.

Other iceberg queries can involve more complex inequality joins. For example, k-*skyband* queries find objects that are *dominated* by no more than k others. These queries are a natural generalization of the popular *skyline* queries (which are equivalent to 0-skyband queries). Listing 2 shows an example k-skyband query over a table `Object(id, x, y)`, where `x` and `y` represent numerical dimensions of interest, such as price, rating, availability, etc.

```
SELECT L.id, COUNT(*)
FROM Object L, Object R
WHERE L.x<=R.x AND L.y<=R.y AND (L.x<R.x OR L.y<R.y)
GROUP BY L.id
HAVING COUNT(*) <= 50;
```

Listing 2: A k-skyband query.

Query processing techniques for iceberg queries have been studied over the last two decades, both for general iceberg queries and for specific classes like market basket or k-skyband queries. Work targeting general iceberg queries, such as [8], focuses on `GROUP BY` and `HAVING`, but not on optimizing joins before grouping. Work targeting specific classes of iceberg queries focuses on developing efficient but specialized techniques. For example, the seminal work in [3] developed the *Apriori* algorithm for association rule mining (a generalization of market baskets), based on the following observation: *if a set of items T is frequent (appears together in k baskets), then all $T' \subseteq T$ are frequent as well.* Applying this observation to the query in Listing 1, we see that we can reduce the `Basket` before the join by filtering out those items that appear fewer than 20 times by themselves. For k-skyband or skyline queries, *transitivity* of dominance relationships has often been exploited for optimization: if a dominates b and b dominates c, then a dominates c. Applying this observation to the query in Listing 2, we note that if c is dominated by b and b is dominated by more than 50 other objects, then c must be dominated by more 50 others as well, so we can safely eliminate it from further consideration.

*This work was supported in part by NSF awards IIS-1408846 and IIS-1552538, and a Google Faculty Research Award.

SIGMOD'17, May 14-19, 2017, Chicago, IL, USA.

DOI: http://dx.doi.org/10.1145/3035918.3064053

```
SELECT S1.id, S1.attr, S2.attr, COUNT(*)
FROM Product S1, Product S2, Product T1, Product T2
WHERE S1.id = S2.id AND T1.id = T2.id
AND S1.category = T1.category
AND T1.attr = S1.attr AND T2.attr = S2.attr
AND T1.val > S1.val AND T2.val > S2.val
GROUP BY S1.id, S1.attr, S2.attr
HAVING COUNT(*) >= 10;
```

Listing 3: An iceberg query for identifying unexciting products.

While there is no shortage of techniques for processing iceberg queries, existing approaches have several notable limitations. *First* and overall, there is no general framework for optimizing iceberg queries over complex joins: techniques are either highly specialized (*e.g.*, for market baskets and skylines) or fail to optimize the join part of query processing, which can be exceedingly expensive even for a small result set. *Second*, the specialized techniques are difficult to generalize to larger classes of iceberg queries, mostly due to the lack of formal procedures for verifying their applicability to general SQL queries. In practice, users often write SQL queries that are variants of those targeted by specialized techniques, and will be unable to benefit from automatic optimization. *Third*, there are interesting opportunities for generalizing and combining specialized techniques within a single iceberg query. The following examples illustrate the last two points above.

Example 1. *Consider a table* Product(id, category, attr, val) *storing information about products carried by a store. Here,* id *identifies the product,* category *denotes its category (with functional dependency* id $\rightarrow$ category*), and* (attr, val) *is a key-value pair storing the value of some relevant attribute for the product;* e.g., ('units sold', 5000) *means 5000 of this product have been sold, while* ('release year', 2015) *means this product was released in 2015. A product in general is associated with multiple attributes and therefore multiple rows in the table.*

*An analyst may wish to compare products in the same categories (*e.g.*, laptops or lawn mowers) to identify those are "overshadowed" by others and therefore can be discontinued. To this end, she poses the following question:* Is there a product that is strictly dominated by at least 10 others in the same category along two dimensions? Show such products and the pairs of dimensions. *Listing 3 shows this iceberg query in SQL, which involves a four-way self-join:* S1 *and* S2 *correspond to two attributes of a product p_1 of interest, while* T1 *and* T2 *correspond to matching attributes of a product p_2 being compared with p_1. The query counts the number of p_2's with strictly larger values than p_1 for both attributes.*

The first observation is that this query is fairly complicated; we have already simplified it for brevity of discussion, but in reality Product *may itself be a join, direction of dominance may depend on the attributes, and the query may additionally enforce uniqueness of attribute pairs (up to order), etc. Intuitively, the query has substructures resembling both market basket and k-skyband queries, but it does not fit either template exactly. Existing approaches cannot recognize the applicability of specialized techniques on this query.*

Interestingly, this query is simultaneously amenable to optimizations for both market basket and k-skyband queries. 1) If a product p is strictly dominated by fewer than 10 others on one attribute a, it must be strictly dominated by fewer than 10 others on both a and b for any other attribute b. Hence, the row associated with (p, a) can then be safely discarded from Product *upfront so that the join is performed on a smaller table (analogous to* Apriori*). 2) If a product p_1 is strictly dominated by fewer than 10 others on two attributes, and if product p_2 is no worse than p_1 on both attributes, then p_2 must be strictly dominated by fewer than 10 others on these attributes as well. Therefore, no further computation is needed for p_2. Existing approaches cannot detect and apply the combination of both optimizations.*

```
WITH pair AS
  (SELECT s1.pid AS pid1, s2.pid AS pid2,
          AVG(s1.hits) as hits1, AVG(s1.hruns) AS hruns1,
          AVG(s2.hits) as hits2, AVG(s2.hruns) AS hruns2
   FROM Score s1, Score s2
   WHERE s1.teamid = s2.teamid AND s1.year = s2.year
   AND s1.round = s2.round AND s1.pid < s2.pid
   GROUP BY s1.pid, s2.pid
   HAVING COUNT(*) >= 3)
SELECT L.id1, L.id2, COUNT(*)
FROM pair L, pair R
WHERE R.hits1 >= L.hits1 AND R.hruns1 >= L.hruns1
  AND R.hits2 >= L.hits2 AND R.hruns2 >= L.hruns2
AND (R.hits1 > L.hits1 OR R.hruns1 > L.hruns1
  OR R.hits2 > L.hits2 OR R.hruns2 > L.hruns2)
GROUP BY L.id1, L.id2
HAVING COUNT(*) <= 20;
```

Listing 4: An iceberg query for finding notable player "pairs."

Example 2. *We have been working with the athletics department at our university to help our sports teams find timely statistics, interesting facts, and noteworthy records on their websites and social media. As an example in baseball (see [1] for a concrete instance cited in media), we want to identify players who score significant runs as a pair. Listing 4 shows the query in SQL. Here, table* Score(pid, year, round, teamid, hit, hrun) *stores, for each year, round, and player (identified by* pid*, playing for* teamid*), the number of batting hits (*hit*) and home runs (*hrun*). The query finds pairs of players who played at least 3 years together, and are dominated by no more than 20 other pairs. Here, dominance (or* weak dominance*) means performing no worse on all four dimensions (two per player) and strictly better on at least one of them (note the difference from* strong dominance *in Example 1).*

This query also benefits from optimizations for both market basket and k-skyband queries, but in a different way from the query in Listing 3. Here, the query consists of two blocks, each of which is an iceberg query by itself and benefits from one of the two types of optimizations (in this sense it is easier to optimize than Listing 3).

*During a sports season, we issue a wide variety of complex iceberg queries such as the one above on one or more tables stored in a database. We have found existing database systems to be frustratingly slow for such queries, preventing any sort of interactive experience. On the other hand, while it is possible to code specialized algorithms for each type of analysis or discovery task, doing so requires a great deal of effort and expertise because of the great variety of these tasks. We have also found hand-coding specialized algorithms to be error-prone, because even slight changes in the query condition (*e.g.*, strong vs. weak dominance) can in corner cases break properties that hand-coded optimizations rely on. Thus, we need a better DBMS-based solution that can deliver acceptable performance automatically (even if it is not as fast as hand-coded solutions) for general and potentially complex iceberg queries expressed in SQL.*

Our Contributions In this paper, we develop automated techniques for optimizing iceberg queries with complex joins that exploit a suite of optimization techniques, thereby addressing the above limitations of existing approaches. In particular, we use three orthogonal and complementary techniques in our optimizations:

- **Generalized a-priori:** Motivated by *Apriori* [3], our generalized *a-priori* technique optimizes an iceberg query with a join by reducing the size of input tables by first applying the `HAVING` constraint to them (if applicable), allowing the original iceberg query to run on smaller inputs.
- **Cache-based pruning:** Motivated by the example of exploiting the properties of dominance for k-skyband query processing, we generalize the idea of pruning join computation using cached results of previous computation. We introduce a query operator called *NLJP* (*Nested-Loop Join with Pruning*) that operates in a nested loop, invoking the inner input query with join attribute values supplied by each tuple from the outer input query. NLJP caches results of the inner query by join attribute values, and for each new outer input tuple, evaluates a *pruning predicate* to determine whether the inner query evaluation can be safely skipped.
- **Memoization:** With the cache inside the NLJP operator, we also enable memoization: if multiple outer input tuples supply identical join attribute values, previously computed and cached results of the inner query can be looked up from the cache, thereby avoiding unnecessary computation.

We give formal conditions for applicability of the above techniques that can be automatically verified, and prove their correctness. We show how to apply these techniques automatically using query rewrite and code generation techniques. Notably, we do not rely on users to identify optimization opportunities or define pruning techniques. Instead, we analyze SQL queries with the knowledge of the database constraints, and we automatically deduce pruning predicates for query conditions involving linear equalities and inequalities using quantifier elimination and Fourier-Motzkin variable elimination [11] methods. We outline an optimization procedure that can identify and apply combinations of our optimization techniques. We implement our techniques in PostgreSQL, and provide experimental evaluation on representative iceberg query workloads with complex join conditions, showing how we outperform both baseline PostgreSQL and a commercial database system.

2 Related Work

Iceberg queries were first introduced in [8]. They focused on computation of the groups that satisfy the threshold in the `HAVING` clause with compact in-memory data structures, without requiring materialization of relation R on which the group-by is performed, and avoiding techniques like hashing or sorting the relation R that would require such materialization. Instead, they proposed a suite of techniques (`defer-count`, `multi-level`, `multi-stage`) that aim to reduce false positives (groups that do not satisfy the `HAVING` condition but are returned) as much as possible without returning any false negatives; the false positives are subsequently eliminated by scanning the relation R. Since these techniques do not require R to be materialized, R can be the result of a join as long as R can be generated in one pass. However, the techniques focus on computing the groups satisfying the threshold, and do not focus on optimizing the joins before these groups are computed. Our work differs by considering the entire query processing pipeline, from detection of scope of optimizations for joins, and for general `HAVING` conditions (`COUNT`, `SUM`, `MIN`, `MAX` with both $\leq$ and $\geq$), to actually implementing these techniques using a DBMS.

Iceberg queries have been extended to *data cubes* (`GROUP BY` on all possible subsets of the grouping attributes) by Beyer and Ramakrishnan [5], where the goal is to output the groups in the cube that satisfy a `HAVING` condition like `COUNT(*) >=` k. The approach in [5] exploits a bottom-up pass on the data cube, *e.g.*, if a group on (AB) does not satisfy the count threshold, then groups on (ABC) or $(ABCD)$ cannot satisfy the count threshold either (similar concept as in *Apriori* [3]). Therefore unlike the standard data cube algorithms that do top-down passes (*e.g.*, computes (ABC) from $(ABCD)$), it uses a bottom-up pass for pruning groups. Since we focus on standard group-by queries without cube, all our subsets are of the same size and these ideas do not directly apply to our problem. Shou et al. [17] extends the concept of iceberg queries to iceberg distance joins for spatial queries, returning object pairs that are within a maximum distance with one pair occurring at a high frequency. Query flocks [18] generalize associate-rule mining to a datalog-based syntax with a monotone filter. We use similar ideas borrowed from *Apriori* [3], but we additionally consider pruning and memoization techniques in the one framework where all these techniques can apply.

There are other query optimization techniques with a similar flavor to ours but none match our optimizations. The *eager aggregation and lazy aggregation* technique proposed by Yan and Larson [23] moves group-by operations up and down a query tree, including in some cases through join operations, which was described in an earlier work [22]. The extended version of [22] describes necessary and sufficient conditions for pushing the having clause through a join as well as group by [21]. The assumptions in [21] is that there are two non-empty tables R_d, R_u participating in the join where R_d contains the aggregate functions in both the `HAVING` and σ clauses, and R_u does not (the group by attributes can belong to both). The paper gives conditions under which the query can be rewritten to an equivalent query that replaces R_d by R'_d, where the `GROUP BY` and `HAVING` clauses are pushed to R_d (it also distributes selection conditions accordingly). Since [21] does not apply the `HAVING` condition after join, and since they obtain the same set of conditions irrespective of whether the `HAVING` condition is monotone or anti-monotone, their conditions are much more restricted than ours. In particular, the conditions for a-priori derived in our work differ from [21] in two ways: (i) we do not assume any constraint on the aggregate function in the σ clause, (ii) the conditions obtained in [21] apply to the join of R_d, R_u and are of the form $GA_d, GA_u \rightarrow GA_d^+$ and $GA_d^+, GA_u \rightarrow RowId(R_u)$, where GA_d, GA_u are grouping attributes from R_d, R_u in the final query, and $GA_d^+ \supseteq GA_d$ also includes additional columns from R_d needed to apply selection after join. In contrast, the conditions obtained in our work apply to individual tables, and therefore are much simpler to check given the information on functional dependencies that hold in individual relations (and therefore less restrictive as well).

The query rewriting technique called *magic sets* [4, 12] was introduced in SQL in 1985-90s. The technique generates a set of auxiliary views that have a series of bindings restricting the table (from the initial query), resulting in a view that produces no irrelevant tuples in the subsequent join. Later, [16] completed the magic sets integration by formalizing the technique in a cost-based scenario, allowing the technique to mix with other optimization strategies. The difference between magic sets and our techniques is that we *prune* tuples before reaching the join by inferring the `HAVING` condition using only partial groups; magic sets technique does not do this.

Optimizations for iceberg queries have been studied for specific applications like market basket queries and k-skyband queries as discussed in the introduction. We generalize the conditions used in the *Apriori* algorithm of [3] in our work. Sarawagi et al. [15] discussed the problem of expressing association rule mining in the form of SQL queries in a DBMS. Ng et al. [13] studied mining and pruning optimizations for constrained association rules in the con-

```
SELECT 𝔾_L, 𝔾_R, Λ
FROM L, R
WHERE Θ
GROUP BY 𝔾_L, 𝔾_R
HAVING Φ;
```

Listing 5: Generic single-block iceberg query Q.

text of exploratory association mining. They defined constrained association queries of the form $\{(S_1, S_2)|\mathcal{C}\}$, where $\mathcal{C}$ is a constraint (*e.g.*, predicates allowed in the HAVING clause) on two subsets S_1, S_2 denoting the pairs of subsets that are of interest. Using optimizations based on anti-monotonicity (like *Apriori*) and succinctness of the constraints, they give efficient algorithms for obtaining constrained frequent sets. Although we use the a-priori techniques (both for monotone and anti-monotone conditions), we focus not on outputting association rules, but instead on using these optimizations for handling iceberg queries with complex joins.

Skyline queries have been studied in the context of outputting pareto optimal objects (not dominated by any other object on all dimensions); in the context of databases, the result consists of input tuples for which there is no input tuple having better or equal values in all attributes, and a better value in at least one attribute. The concept of k-dominant skyline was studied by Chan et al. [6], which relaxes the dominance on all d dimensions to any subset of size k. The survey by Chomicki et al. [7] describes several common algorithmic techniques for skyline queries. The k-skyband query [14] generalizes skyline queries (for skyline $k = 0$). One of the techniques commonly used in skyline or skyband queries is transitivity: if t_1 dominates t_2 which in turn dominates t_3, then t_1 dominates t_3. This property is often used for pruning in skyline algorithms. We generalize such pruning techniques for arbitrary join predicates involving inequalities, by automatically inferring them and using them to optimize query execution.

Generalized pruning techniques have been used in the setting of constraint satisfaction. In a database setting, Searchlight [10] integrated constraint satisfaction with a column-store DBMS, and heavily relied on both pruning and parallelism. However, their technique considers simpler, well-defined constraints, where our techniques are more general. Perada [19] has a similar flavor but provides a flexible framework for pruning, however the pruning predicates must be manually specified.

To the best of our knowledge, we provide the first framework for evaluating general iceberg queries with complex joins that combines multiple optimization techniques of a-priori, pruning, and memoization.

3 Preliminaries

In this section we describe some concepts and notations that are used in the rest of the paper. For a relation instance R, we denote the number of tuples in R by $|R|$. For an attribute A of R, $\texttt{dom}(A)$ denotes the domain of A; $\texttt{adom}(A)$ denotes the active domain of A in the instance R; given a tuple $t \in R$, $t.A$ denotes the value of the attribute A in t. For a list of attributes Z from R, $\texttt{dom}(Z)$ denotes the product of the domains of the attributes in Z; $\texttt{adom}(Z)$ denotes the set of tuples in $\pi_Z R$; given a tuple $t \in R$, $t.Z$ denotes a tuple formed by t's values for the attributes in Z.

For simplicity, we shall present our main techniques using a single-block iceberg query Q joining two relations L and R as shown in Listing 5. Our techniques extend to subqueries of this form, and to joins involving more than two relations (discussed in Appendix D). As seen in examples earlier, many complex iceberg queries involve self-joins, so in general L and R may refer to the same underlying relation. Let $\mathbb{A}_L$ denote the attributes of L and $\mathbb{A}_R$ those of R. In case of ambiguity, we prefix attribute references with the relation (*e.g.*, $L.A$ or $R.A$). We introduce notations related to L below; those for R are analogous. Given the query Q, $\mathbb{G}_L \subseteq \mathbb{A}_L$ denotes the (possibly empty) set of GROUP-BY attributes from L. The join condition is denoted Θ. For simplicity here we do not consider selection conditions in WHERE that can be evaluated over either L or R alone; our techniques still apply simply by treating L and R as filtered versions of the respective relations with selection conditions applied upfront. Let $\mathbb{J}_L \subseteq \mathbb{A}_L$ denote L's join attributes, *i.e.*, the subset of L's attributes that are referenced in Θ. Finally, Φ denotes the HAVING condition, and Λ denotes the list of output expressions in the SELECT clause.

Example 3. *Consider the first block of the "pairs" query (Listing 4) that defines* pair*. Suppose this (sub)query is our Q. Then L would be* s1 *and R would be* s2*, both referring to the same underlying relation* Score*. Both $\mathbb{G}_L$ and $\mathbb{G}_R$ are* {pid}*, and both $\mathbb{J}_L$ and $\mathbb{J}_R$ are* {pid, teamid, round, year} *(prefixed with* s1 *or* s2 *as appropriate). Θ is the entire* WHERE *condition, Φ is* COUNT(*) >= 3*, and Λ computes four* AVG *aggregates for output.*

Now consider the second block of the "pairs" query (Listing 4) as our Q. Both L and R refer to the pair *view defined earlier. $\mathbb{G}_L$ = {id1, id2} and $\mathbb{G}_R = \varnothing$. Both $\mathbb{J}_L$ and $\mathbb{J}_R$ are* {hits1, hits2, hruns1, hruns2} *(prefixed with L or R as appropriate). Θ is the entire* WHERE *condition, Φ is* COUNT(*) <= 20*, and Λ consists of a single aggregate expression* COUNT(*)*.*

We call a tuple $\ell \in L$ an *L-tuple* and a tuple $r \in R$ *an R-tuple*. An *LR-tuple*, denoted $\langle \ell, r \rangle$, is formed by concatenating $\ell \in L$ and an $r \in R$. $\Theta(\langle \ell, r \rangle)$ evaluates to either true or false for a given LR-tuple. The set of LR-tuples in $L \times R$ that satisfy Θ is partitioned into *LR-groups* according to their values for $\mathbb{G}_L \cup \mathbb{G}_R$. Let $LR^{(u,v)}$ denote the LR-group with value u for $\mathbb{G}_L$ and value v for $\mathbb{G}_R$, where $u \in \texttt{adom}(\mathbb{G}_L)$ and $v \in \texttt{adom}(\mathbb{G}_R)$.[1] $\Phi(LR^{(u,v)})$ evaluates to either true or false for a given LR-group. For each LR-group that satisfies Φ, a result tuple is produced; given this LR-group, each expression in Λ evaluates to an atomic value for an output attribute. Conceptually, the notions of *L-groups* and *R-groups* are also useful. We partition the set of L-tuples into *L-groups* according to their values for $\mathbb{G}_L$, and use $L^{(u)}$ to denote the L-group with value u for $\mathbb{G}_L$. Finally, given values w for L's join attributes $\mathbb{J}_L$, we denote the subset of joining R-tuples by $R_{\ltimes w}$. Table 1 summarizes the notations used in this paper.

In this paper we assume that the relational operators π, σ, and $\bowtie$ are all duplicate-preserving, which is needed for the SQL semantics. We also assume that the subset/superset relationships ($\subseteq$ and $\supseteq$) have duplicate semantics as well. For notational convenience, if a set of attributes $\mathbb{A} = \varnothing$, we will interpret the (selection or join) condition $\mathbb{A} = \langle\rangle$ as simply true.

The HAVING condition Φ can be *monotone* or *anti-monotone* w.r.t. its input. Such monotonicity properties enable us to identify and apply optimizations safely. We formalize these properties below and show examples in Table 2.

Definition 1 (Monotone and anti-monotone conditions). *Let Φ be a condition that evaluates to either true or false given a set of tuples as an input. We call* Φ monotone *(resp.* anti-monotone*) if for all $T \subseteq T'$ (resp. $T \supseteq T'$) with schema compatible with Φ, $\Phi(T) \Rightarrow \Phi(T')$.*

[1] Note that in case $\mathbb{G}_L = \varnothing$ (the case for $\mathbb{G}_R$ is analogous), an LR-group can be identified simply by its value v for $\mathbb{G}_R$, and we would denote that group by $LR^{(\langle\rangle,v)}$.

Notation	Meaning
$\mathbb{J}_L$	join attributes from L
$\mathbb{G}_L$	GROUP-BY attributes from L
$L^{(u)}$	L-group identified by $u \in \texttt{adom}(\mathbb{G}_L)$; *i.e.*, $\sigma_{\mathbb{G}_L=u} L$
$LR^{(u,v)}$	LR-group identified by $u \in \texttt{adom}(\mathbb{G}_L)$ and $v \in \texttt{adom}(\mathbb{G}_R)$; *i.e.*, $\sigma_{\mathbb{G}_L=u \wedge \mathbb{G}_R=v}(L \bowtie_\Theta R)$
$R_{\ltimes w}$	R-tuples joining with $w \in \texttt{dom}(\mathbb{J}_L)$; *i.e.*, $\pi_{\mathbb{A}_R}(\{w\} \bowtie_\Theta R)$, where $\{w\}$ denotes a relation with attributes $\mathbb{J}_L$ and a single tuple specified by w

Table 1: Notations used in the paper, defined in the context of a query Q over relations L and R, defined in Listing 5. Notations for R are omitted as they are analogous to those for L.

Monotone	Anti-monotone
COUNT(*) >= c	COUNT(*) <= c
COUNT(A) >= c	COUNT(A) <= c
SUM(A) >= c (if $\texttt{dom}(A) \subseteq \mathbb{R}_{\geq 0}$)	SUM(A) <= c (if $\texttt{dom}(A) \subseteq \mathbb{R}_{\geq 0}$)
MAX(A) >= c	MAX(A) <= c
MIN(A) >= c	MIN(A) <= c
COUNT(DISTINCT A) >= c	COUNT(DISTINCT A) <= c

Table 2: Examples of monotone and anti-monotone HAVING conditions. A represents an attribute and c represents a constant.

4 Generalized A-Priori Techniques

In this section we discuss the *generalized a-priori technique*, *i.e.*, when we can reduce the cost of a join by pushing down the condition in the HAVING clause to individual relations to select a subset of tuples from either or both relations before we execute the original iceberg query. The goal of generalized a-priori technique is to achieve this reduction in the size of the input relations by safely removing tuples that are guaranteed to not to contribute to the final answers in the original iceberg query.

The a-priori technique was proposed in the seminal market basket analysis paper [3] to find frequent itemsets. The frequent itemset problem aims to find sets of elements that co-occur in a set of baskets at least τ times where τ is a threshold. The key observation is that, if a set of elements co-occur in τ baskets, then any subset of these elements must also co-occur in τ baskets, and in particular, any individual element must also appear in at least τ baskets, which helps safely discard item or itemsets that cannot belong to the final answers. This idea was revisited in the *Query Flocks* paper [18] by focusing on pair of items as the itemset instead of arbitrary subsets (see Listing 1). Basically, the HAVING clause in Listing 1 can be applied to the table Basket to safely prune the items of individual frequency lower than 20, which can never appear in frequent itemset pairs. This reduces the cost of the join since now the self-join is performed on a subset of tuples that can be much smaller than the original Basket relation.

Although the above optimization is intuitive for the specific frequent itemset problem, there has not been a study of utilizing this idea for query optimization in general. The traditional query optimizers apply the HAVING clause after performing the grouping, and typically attempt to push the selection conditions before grouping or joins after grouping, but do not attempt to prune individual tables based on the having condition. Optimization using pushing HAVING condition prior to the join was studied by Yan and Larson [23] (see Section 2). However, since they do not execute the original query again with the HAVING condition on the reduced relations, and do not exploit the monotonicity/anti-monotonicity properties of the HAVING conditions, certain constraints on the input query have to be applied; further, the constraints obtained are more restrictive and apply to the table obtained by joining L and R. Instead, our goal in this section is to derive constraints on individual relations L or R that are easier to verify. These constraints exploit the monotonicity/anti-monotonicity properties of the HAVING condition Φ and allow us to safely apply a-priori technique to individual tables for general iceberg queries as shown in Listing 5.

4.1 Safe A-Priori Technique

Consider the generic query Q in Listing 5. Suppose the HAVING condition Φ is *applicable* to L: *i.e.*, all attributes in Φ are from L, and *, like in COUNT(*), is also allowed. The (generalized) a-priori technique replaces L in Q with the following "reduced" relation $L' \subseteq L$:

$L' =$ SELECT * FROM L WHERE $\mathbb{G}_L$ IN
(SELECT $\mathbb{G}_L$ FROM L GROUP BY $\mathbb{G}_L$ HAVING Φ)

We call the subquery following IN a *reducer* for L. If Φ is applicable to R, the technique can similarly apply to obtain R'. While the HAVING condition in L' (or R') is syntactically identical to the original Φ, it is evaluated for each group of L-tuples (or R-tuples) as opposed to each group of LR-tuples. To produce the correct result, the rewritten query must still retain the final HAVING condition.

Definition 2 (Safe a-priori). *Given a query Q in Listing 5 and a database instance $\mathcal{D}$, we say that the a-priori technique is* safe *for relation L (or R) if query Q', obtained from Q by replacing L with L' (or R with R') as defined above, returns the result as Q on $\mathcal{D}$.*

Consider the candidate LR-groups produced by the original query Q. Replacing L with L' (or R with R') effectively discards a subset of these groups with particular $\mathbb{G}_L$ (or $\mathbb{G}_R$) values, while leaving other groups intact. Hence, to guarantee safety, it suffices to check that no LR-group satisfying Φ in Q is discarded by a-priori. Formally, the following holds:

Proposition 1. *It is safe to apply a-priori to L if Φ is applicable to L and for every candidate LR-group $LR^{(u,v)}$ (where $u \in \texttt{adom}(\mathbb{G}_L)$ and $v \in \texttt{adom}(\mathbb{G}_R)$), $\Phi(LR^{(u,v)}) \Rightarrow \Phi(L^{(u)})$. (A similar condition holds for R.)*

The correctness of the above proposition easily follows from the observation that L' discards only those L-groups that fail Φ, which implies that they cannot contribute to any LR-group that would eventually satisfy Φ. Next, we study how to check the condition of Proposition 1 by analyzing the query, and in particular, by exploiting the monotonicity property of its HAVING condition.

4.2 Generic Instance-Based Checks

Proposition 1 requires checking the implication $\Phi(LR^{(u,v)}) \Rightarrow \Phi(L^{(u)})$. Informally, if Φ is monotone, a sufficient condition would be that $LR^{(u,v)}$ is "contained in" $L^{(u)}$. However, LR-groups and L-groups have different schemas (even though the same Φ can be applied to both), so we need a more precise way of describing this relationship. To this end, we introduce the notion of *non-inflationary* queries (and similarly, *non-deflationary* queries for use with an anti-monotone Φ).

Definition 3 (Non-inflationary and non-deflationary queries). *Consider query Q in Listing 5 over a given database instance $\mathcal{D}$ with relations L and R. Q is* non-inflationary *(resp.* non-deflationary*) w.r.t. L in $\mathcal{D}$ if for every candidate LR-group $LR^{(u,v)}$ of Q (where $u \in \texttt{adom}(\mathbb{G}_L)$ and $v \in \texttt{adom}(\mathbb{G}_R)$),*

$$\forall \ell \in L^{(u)} : \left|\sigma_{\mathbb{A}_L = \ell.\mathbb{A}_L} LR^{(u,v)}\right| \leq 1 \quad \textit{(resp. } \geq 1\textit{)}.$$

The definition is analogous for R.

In other words, given $L^{(u)}$ and $LR^{(u,v)}$, for a non-inflationary (resp. non-deflationary) query, each tuple in $L^{(u)}$ maps to at most

(resp. at least) one tuple in $LR^{(u,v)}$. (In the other direction, every tuple in $LR^{(u,v)}$ always maps to exactly one tuple in $L^{(u)}$.) It is worth noting that the non-inflationary condition is weaker than requiring each L-tuple ℓ to join with at most one R-tuple overall—ℓ can join with multiple R-tuples as long as no two joined tuples belong to the same LR-group after grouping by $\mathbb{G}_L$ and $\mathbb{G}_R$.

Intuitively, a non-inflationary (resp. non-deflationary) Q allows us to conclude that $L^{(u)}$ is a "superset" (resp. "subset") of $LR^{(u,v)}$, and hence, in conjunction with a monotone (resp. anti-monotone) Φ, the safe a-priori condition of Proposition 1 can be satisfied.

Theorem 1 (**Instance-based safety conditions for generalized a-priori).** *Consider query Q in Listing 5. Given database instance $\mathcal{D}$, the a-priori technique is safe to apply to L if Φ is applicable to L, and*

- *Φ is monotone and Q is non-inflationary w.r.t. L in $\mathcal{D}$, or,*
- *Φ is anti-monotone and Q is non-deflationary w.r.t. L in $\mathcal{D}$.*

Similar conditions hold for R.

Proof. We prove the sufficiency of the first condition (for monotone Φ); the second one (for anti-monotone Φ) follows similarly.

Consider any candidate LR-group $LR^{(u,v)}$ where $u \in \texttt{adom}(\mathbb{G}_L)$ and $v \in \texttt{adom}(\mathbb{G}_R)$. By Proposition 1, it suffices to show that $\Phi(LR^{(u,v)}) \Rightarrow \Phi(L^{(u)})$. Since Φ is applicable to L, $\Phi(LR^{(u,v)}) \Leftrightarrow \Phi(\pi_{\mathbb{A}_L} LR^{(u,v)})$, where π is duplicate-preserving. Since Q is non-inflationary w.r.t. L, by Definition 3, we have $L^{(u)} \supseteq \pi_{\mathbb{A}_L} LR^{(u,v)}$, with duplicate semantics. As Φ is monotone, $\Phi(\pi_{\mathbb{A}_L} LR^{(u,v)})$ implies $\Phi(L^{(u)})$, completing the proof. □

Example 4. *Consider the market basket query (Listing 1). The* `HAVING` *condition* `COUNT(*) >= 20` *is monotone. The query is non-inflationary: for any* `i1`*-tuple, there can be at most one joined tuple in each candidate* `i1-i2` *group, as there can be at most one* `i2`*-tuple with a given* `item` *value in the same basket as* `i1`*. Hence, a-priori is safe for* `i1` *(and analogously for* `i2`*).*

Similarly, for the first block of the "pairs" query (Listing 4) that defines `pair`*, the* `HAVING` *condition* `COUNT(*) >= 3` *is monotone. This (sub)query is non-inflationary: each* `s1`*-tuple contributes to at most one joined tuple in each candidate* `s1-s2` *group, because for any given player identified by* `s2.pid`*, there can be at most one* `s2` *tuple with* `year` *and* `round` *matching those of* `s1` *(not to mention additional conditions on* `teamid` *and* `pid`*). Hence, a-priori is safe for* `s1` *(and analogously for* `s2`*).*

We note that the conditions in Theorem 1 are "tight" in the sense that the non-inflationary (resp. non-deflationary) property is necessary for achieving safety for queries with a monotone (resp. anti-monotone) `HAVING` condition. The following example illustrates this point.

Example 5. ***(Monotone Φ)*** *Consider a query over relations $L(\mathbb{G}_L, \mathbb{J}_L)$ and $R(\mathbb{J}_R, \mathbb{O}_R, \mathbb{G}_R)$ with join condition $\mathbb{J}_L = \mathbb{J}_R$ and a monotone* `HAVING` *condition* `COUNT(*) >= 2`*. Suppose L contains a single tuple $\langle u, w\rangle$, and R contains two tuples $\langle w, z_1, v\rangle$, $\langle w, z_2, v\rangle$. The query is inflationary since the single L-tuple contributes to two tuples in the same LR-group $LR^{(u,v)}$. Applying a-priori to L would be erroneous since* `COUNT(*) >= 2` *will discard the single tuple from L, leading to an empty result, whereas the LR-group $LR^{(u,v)}$ in fact satisfies the* `HAVING` *condition and should be reflected in the correct result.*

(Anti-monotone Φ) *Consider a query over relations $L(\mathbb{G}_L, \mathbb{J}_L)$ and $R(\mathbb{J}_R, \mathbb{G}_R)$ with join condition $\mathbb{J}_L = \mathbb{J}_R$ and an anti-monotone* `HAVING` *condition* `COUNT(*) <= 1`*. Suppose L has two tuples $\ell_1 = \langle u, w_1\rangle$ and $\ell_2 = \langle u, w_2\rangle$, while R has a single tuple $r = \langle w_1, v\rangle$. The query is deflationary: among the two tuples in the L-group $L^{(u)}$, ℓ_1 joins with r and contributes to the LR-group $LR^{(u,v)}$, but ℓ_2 does not contribute $LR^{(u,v)}$. Applying a-priori to L would be erroneous since it will discard both ℓ_1 and ℓ_2, leading to an empty result, whereas the LR-group $LR^{(u,v)}$ in fact satisfies the* `HAVING` *condition and should be reflected in the correct result.*

4.3 Schema-Based Checks

The generic safety conditions in Theorem 1 may depend on a given database instance, since non-inflationary and non-deflationary joins are defined in terms of tuples in the participating relations. In this section, we develop conditions that rely only on the schema and functional dependencies (that hold on all instances), which can be easily applied by a query optimizer.

Theorem 2 (**Schema-based safety conditions for generalized a-priori).** *Consider query Q in Listing 5. The a-priori technique is safe to apply to L if Φ is applicable to L, and*

- *Φ is monotone and $\mathbb{G}_R \cup \mathbb{J}_R^{\Theta} \rightarrow \mathbb{A}_R$ (*i.e.*, $\mathbb{G}_R \cup \mathbb{J}_R^{\Theta}$ is a superkey of R), where $\mathbb{J}_R^{\Theta} \subseteq \mathbb{J}_R$ is the subset of R's attributes involved in equality join predicates in Θ, or,*
- *Φ is anti-monotone and $\mathbb{G}_L \rightarrow \mathbb{J}_L$.*

Similar conditions hold for R.

Proof. We give the proof for L by showing that if the stated conditions hold, then the corresponding condition in Theorem 1 holds too; the proof for R is similar.

(Monotone Φ) Given $\mathbb{G}_R \cup \mathbb{J}_R^{\Theta} \rightarrow \mathbb{A}_R$, we need to argue that Q is non-inflationary w.r.t. L. Assume to the contrary that there exists a candidate LR-group $LR^{(u,v)}$, an L-tuple $\ell \in L^{(u)}$, and two R-tuples r_1, r_2 such that $\langle \ell, r_1\rangle \in LR^{(u,v)}$ and $\langle \ell, r_2\rangle \in LR^{(u,v)}$. As both $\langle \ell, r_1\rangle$ and $\langle \ell, r_2\rangle$ are in $LR^{(u,v)}$, $r_1.\mathbb{G}_R = r_2.\mathbb{G}_R = v$; furthermore, both join with ℓ, so they must agree on the equality join attributes, *i.e.*, $r_1.\mathbb{J}_R^{\Theta} = r_2.\mathbb{J}_R^{\Theta}$. From $\mathbb{G}_R \cup \mathbb{J}_R^{\Theta} \rightarrow \mathbb{A}_R$, we conclude that r_1 and r_2 are in fact the same tuple, completing the proof.

(Anti-monotone Φ) Given $\mathbb{G}_L \rightarrow \mathbb{J}_L$, we need to argue that the query is non-deflationary w.r.t. L. Consider any candidate LR-group $LR^{(u,v)}$, which is by definition non-empty. Pick any $\langle \ell_0, r_0\rangle \in LR^{(u,v)}$. For any $\ell \in L^{(u)}$, $\ell.\mathbb{G}_L = \ell_0.\mathbb{G}_L = u$, so by $\mathbb{G}_L \rightarrow \mathbb{J}_L$, $\ell.\mathbb{J}_L = \ell_0.\mathbb{J}_L$, meaning that ℓ also joins with r_0. Therefore $\langle \ell, r_0\rangle \in LR^{(u,v)}$, completing the proof.

□

Intuitively, for a monotone Φ, the functional dependency $\mathbb{G}_R \cup \mathbb{J}_R^{\Theta} \rightarrow \mathbb{A}_R$ ensures that for each tuple ℓ in L-group $L^{(u)}$, and for each LR-group $LR^{(u,v)}$ that ℓ can potentially contribute to, $\ell.\mathbb{J}_L$ and v together allow for only one LR-tuple to be associated with ℓ. (Any additional non-equality join predicates in Θ would not affect this guarantee.)

For an anti-monotone Φ, the functional dependency $\mathbb{G}_L \rightarrow \mathbb{J}_L$ ensures that for each tuple $\ell \in L^{(u)}$, the presence of a tuple in any LR-group $LR^{(u,v)}$ implies that it shares the same $\mathbb{J}_L$ values as ℓ, so ℓ must contribute to $LR^{(u,v)}$ as well.

Example 6. *Consider again the market basket query (Listing 1) with the monotone* `HAVING` *condition. Here L is* `i1`*, R is* `i2`*, $\mathbb{G}_L = \{$`i1.item`$\}$, $\mathbb{G}_R = \{$`i2.item`$\}$, $\mathbb{J}_L = \mathbb{J}_L^{\Theta} = \{$`i1.bid`$\}$, and $\mathbb{J}_R = \mathbb{J}_R^{\Theta} = \{$`i2.bid`$\}$. We have $\mathbb{G}_R \cup \mathbb{J}_R^{\Theta} = \{$`i2.item, i2.bid`$\}$, which is clearly a superkey of R. Therefore, by Theorem 2, a-priori is safe for* `i1` *(and analogously for* `i2`*).*

Note that if we instead change the query to output infrequent item pairs using an anti-monotone `HAVING` *condition* `COUNT(*) <= 20`*, a-priori would not be safe for* `i1` *(resp.* `i2`*), as the required check* $\mathbb{G}_L \rightarrow \mathbb{J}_L$ *(resp.* $\mathbb{G}_R \rightarrow \mathbb{J}_R$*), or* `item` $\rightarrow$ `bid`*, fails (*`item` *is not a key of* `Basket`*). Intuitively, we cannot discard a frequent item because it may still participate in an infrequent pair.*

Example 7. *Consider two tables* `Basket`(`bid`, `item`, `did`) *(where* `did` *identifies the particular discount under which the item was bought—a basket may contain multiple instances of the same item bought under different discounts) and* `Discount`(`did`, `rate`) *(where* `rate` *denotes the rate of the discount). The following query outputs the discount rates applied for items, for least 25 baskets:*

```
SELECT item, rate
FROM Basket L, Discount R
WHERE L.did = R.did
GROUP BY item, rate
HAVING COUNT(DISTINCT bid) >= 25;
```

Once again, we have a monotone `HAVING` *condition.* $\mathbb{J}_L = \mathbb{J}_L^{=} = \{\texttt{did}\} = \mathbb{J}_R^{=} = \mathbb{J}_R$*,* $\mathbb{G}_L = \{\texttt{item}\}$*, and* $\mathbb{G}_R = \{\texttt{rate}\}$*.*

We can safely apply a-priori to L *(*`Basket`*), because* $\mathbb{G}_R \cup \mathbb{J}_R^{=}$ *is a superkey of* R*. However, we cannot safely apply a-priori to* R *(*`Discount`*), because* $\mathbb{G}_L \cup \mathbb{J}_L^{=}$ *is not a superkey of* L *(the same item-did can appear in multiple bid-s).*

Suppose instead we change the `HAVING` *condition to be anti-monotone, say* `COUNT(DISTICT bid) <= 25`*, and additionally assume that* `item` $\rightarrow$ `did` *(*i.e.*, only one type of discount can be applied to given item). Theorem 2 still allows us to apply a-priori safely to* L*, although via a different check:* $\mathbb{G}_L \rightarrow \mathbb{J}_L$*. Intuitively, in this case, if an item appears in more than* 25 *baskets, there is no need to consider it because it would have the same discount rate in all these baskets, making it impossible to pass the final* `HAVING` *condition.*

Example 8. *We revisit the examples in Example 5 used to illustrate the tightness of the conditions in Theorem 1, now by applying Theorem 2.*

***(Monotone* $\boldsymbol{\Phi}$*)** In this example of a query over* $L(\mathbb{G}_L, \mathbb{J}_L)$ *and* $R(\mathbb{J}_R, \mathbb{O}_R, \mathbb{G}_R)$ *with a monotone* `HAVING` *condition, the functional dependency* $\mathbb{G}_R, \mathbb{J}_R \rightarrow \mathbb{A}_R$ *did not hold because there were two distinct* R*-tuples with the same values for* $\mathbb{G}_R$ *and* $\mathbb{J}_R$*. Therefore, we would not apply a-priori to* L *per Theorem 2.*

***(Anti-monotone* $\boldsymbol{\Phi}$*)** In this example of a query over* $L(\mathbb{G}_L, \mathbb{J}_L)$ *and* $R(\mathbb{J}_R, \mathbb{G}_R)$ *with an anti-monotone* `HAVING` *condition, the functional dependency* $\mathbb{G}_L \rightarrow \mathbb{J}_L$ *did not hold because there were two distinct* L*-tuples with the same* $\mathbb{G}_L$ *value but different* $\mathbb{J}_L$ *values. Therefore, we would not apply a-priori to* L *per Theorem 2.*

5 Cache-Based Pruning Techniques

This section discusses our cache-based pruning techniques. Unlike the generalized a-priori technique discussed in Section 4, which were applied by SQL rewriting, here we apply pruning at run time by extending the nested loop join (NLJ) algorithm (we describe its realization using a new NLJP operator in Section 7). Similar to a nested loop join, we consider tuples from the outer relation one at a time. The key idea is that tuples that failed to generate any final result tuple allow us to potentially "short-circuit" the processing of new tuples. Under certain conditions, we can perform a simple check between the new tuple and a previously processed tuple whose outcome has been cached, which allows us to determine whether the new tuple has any chance of contributing to the result; if not, we avoid expensive evaluation involving the inner relation.

```
C ← ∅ # cache of unpromising join values
T ← ∅ # collection of contributions of the form contrib(ℓ, u, v),
        # where ℓ ∈ L, u ∈ adom(𝔾_L), and v ∈ adom(𝔾_R)
for each ℓ in L:
  if prune(ℓ, C):
    continue
  compute contrib(ℓ, ℓ.𝔾_L, v) and add to T for each unique v ∈ R⋉ℓ.𝕁_L
  add ℓ.𝕁_L to C if ℓ.𝕁_L is unpromising
for each partition T[u, v] = {contrib(_, u, v) ∈ T} of T by u, v:
  compute and return, if applicable, a result tuple for LR^(u,v)
```

Listing 6: Algorithm template for cache-based pruning for Q in Listing 5, assuming Φ is applicable to R.

Recall from Section 1 the example on the k-skyband query (Listing 2) that illustrates this idea. Suppose we have already processed a tuple $\ell_1 = \langle i_1, 10, 10\rangle$ from the outer relation L, which resulted in a group with `COUNT(*)` greater than 50, and we need to process a tuple $\ell_2 = \langle i_2, 5, 5\rangle$ from L. From the `WHERE` condition and the observation that $\ell_2.\texttt{x} \leq \ell_1.\texttt{x} \wedge \ell_2.\texttt{y} \leq \ell_1.\texttt{y}$, we can infer that any R-tuple that joins with ℓ_1 must also join with ℓ_2, so ℓ_2 must result in a group with `COUNT(*)` greater than 50 as well, failing the `HAVING` condition. We can therefore prune ℓ_2 immediately.

In the following, we give a detailed description of the principle behind pruning and how to derive pruning conditions automatically.

5.1 Subsumption, Caching, and Safe Pruning

Consider again the generic query Q in Listing 5, with L as the outer relation and R as the inner. For pruning to apply, we assume that the `HAVING` condition Φ is applicable to R: *i.e.*, all attributes in Φ are from the inner relation, and $*$ is also allowed.

Pruning involves comparing two L-tuples—one previously processed and one to be processed—in terms of their contribution to the final query result. Since they contribute to the final result by joining with R-tuples, it is natural to compare the subsets of R-tuples that they join with respectively. To this end, we introduce the notion of *subsumption* below. Since L-tuples with identical join attribute values always join with the same subset of R-tuples, we define the subsumption relationship on the $\mathbb{J}_L$ values rather than L-tuples themselves.

Definition 4 (Subsumption). *Given a database instance* $\mathcal{D}$ *with relations* L *and* R*, and* $w, w' \in \texttt{adom}(\mathbb{J}_L)$*, we say that* w subsumes w'*, denoted* $w \succeq w'$*, if* $R_{\ltimes w} \supseteq R_{\ltimes w'}$*;* i.e.*, the set of* R*-tuples that joins with (any* L*-tuple with* $\mathbb{J}_L$ *equal to)* w *is a superset of those that joins with* w'*.*

We now clarify what information we cache during execution to enable subsequent pruning. Since we assume that Φ is applicable to R, given an L-tuple ℓ, we can test Φ using the subset of R-tuples joining with ℓ. We then record $\ell.\mathbb{J}_L$ in a cache $\mathcal{C}$ if $\ell.\mathbb{J}_L$ is *unpromising*, as defined below:

Definition 5 (Unpromising $\mathbb{J}_L$ values). *Given a database instance* $\mathcal{D}$ *with relations* L *and* R*, we say that* $w \in \texttt{dom}(\mathbb{J}_L)$ *is* unpromising *if* $\forall v \in \pi_{\mathbb{G}_R} R_{\ltimes w} : \neg\Phi(\sigma_{\mathbb{G}_R = v} R_{\ltimes w})$*;* i.e.*,* Φ *evaluates to false for every partition of joining* R*-tuples by* $\mathbb{G}_R$*. (If* $\mathbb{G}_R = \varnothing$*, the condition reduces to* $\neg\Phi(R_{\ltimes w})$*.)*

Listing 6 sketches the algorithm for cache-based pruning. Basically, for each L-tuple ℓ that cannot be pruned, it computes ℓ's contribution to each LR-group's (potential) result tuple; in this process, it also discovers whether $\ell.\mathbb{J}_L$ is promising, and if not, records it in the cache. Finally, all contributions for each LR-group are considered in computing the associated final result tuple (if any). We omit unnecessary details for now (Section 7 further describes its implementation using the NLJP operator), and focus on

$\mathsf{prune}(\ell, \mathcal{C})$. Intuitively, the fact that a particular $w' \in \mathtt{dom}(\mathbb{J}_L)$ is unpromising, in conjunction with a subsumption test, may be used to prune L-tuples. In the simplest case, if there is a one-to-one correspondence between L-tuples and the candidate LR-groups, then for each $\ell \in L$, evaluating Φ on its corresponding LR-group is equivalent to evaluating Φ on $R_{\ltimes \ell.\mathbb{J}_L}$. For a monotone (or anti-monotone) Φ, $\ell.\mathbb{J}_L \preceq w'$ (resp. $\ell.\mathbb{J}_L \succeq w'$) would allow us to conclude that ℓ joins with a subset (resp. superset) of R-tuples that join with w'; then, an unpromising w' would imply that $\ell.\mathbb{J}_L$ is also unpromising, so ℓ cannot contribute to the final query result.

Unfortunately, in general there is not a one-to-one correspondence between L-tuples and the candidate LR-groups. Each L-tuple may in fact contribute to multiple LR-groups (by joining with R-tuples with different $\mathbb{G}_R$ values), and each LR-group may include contributions from multiple L-tuples (with same or different $\mathbb{J}_L$ values). We need to be more careful when using unpromising join attribute values in inferring contribution (or lack therefore) to the final query result. Therefore, to ensure safe pruning, we develop additional checks based on the schema and functional dependencies below.

Theorem 3 (Safe pruning conditions). *Suppose Φ is applicable to R, and the algorithm in Listing 6 correctly computes Q when $\mathsf{prune}(\ell, \mathcal{C})$ always returns false. Then, the algorithm also computes Q correctly with the following definition for $\mathsf{prune}(\ell, \mathcal{C})$:*

- *If Φ is monotone and $\mathbb{G}_L \rightarrow \mathbb{A}_L$ (*i.e.*, $\mathbb{G}_L$ is a superkey of L), then $\mathsf{prune}(\ell, \mathcal{C}) \equiv \exists w' \in \mathcal{C} : \ell.\mathbb{J}_L \preceq w'$.*
- *If Φ is anti-monotone, $\mathbb{G}_L \rightarrow \mathbb{A}_L$ (*i.e.*, $\mathbb{G}_L$ is a superkey of L), and $\mathbb{G}_R = \varnothing$ (*i.e.*, no* `GROUP-BY` *attributes from R), then $\mathsf{prune}(\ell, \mathcal{C}) \equiv \exists w' \in \mathcal{C} : \ell.\mathbb{J}_L \succeq w'$.*

A similar algorithm with R in the outer loop is correct when Φ is applicable to L.

The proof of Theorem 3 is given in Appendix A.

Example 9. *Consider again the k-skyband query (Listing 2). The* `HAVING` *condition* `COUNT(*) <= 50` *is anti-monotone and applicable to R. $\mathbb{G}_L = \{L.\mathtt{id}\}$, $\mathbb{G}_R = \varnothing$, $\mathbb{J}_L = \{L.\mathtt{x}, L.\mathtt{y}\}$, and $\mathbb{J}_R = \{R.\mathtt{x}, R.\mathtt{y}\}$. Since $\mathbb{G}_L$ (*`id`*) is a key of L (*`Object`*), we can safely applying cache-based pruning for L by Theorem 3.*

Intuitively, when Φ is monotone and $\mathbb{G}_L$ is a superkey of L, every LR-group $LR^{(u,v)}$ can receive contribution from only a single L-tuple (namely the one with value u for $\mathbb{G}_L$). Therefore, the LR-groups that $\ell \in L$ contributes to essentially form a partitioning of $R_{\ltimes \ell.\mathbb{J}_L}$ by $\mathbb{G}_R$. If $\ell.\mathbb{J}_L \preceq w'$, we have $R_{\ltimes \ell.\mathbb{J}_L} \subseteq R_{\ltimes w'}$, and the subset relationship must also hold for every pair of corresponding $R_{\ltimes \ell.\mathbb{J}_L}$ and $R_{\ltimes w'}$ partitions. Since none of the $R_{\ltimes w'}$ partitions satisfies Φ, neither can any partition for ℓ because of monotonicity. Hence ℓ does not contribute to the final result.

On the other hand, when Φ is anti-monotone, $\mathbb{G}_R = \varnothing$ implies that each $\ell \in L$ contributes to single LR-group $LR^{(\ell.\mathbb{G}_L, \langle\rangle)}$; moreover, $\mathbb{G}_L$ is a superkey of L, so this single LR-group receives contribution from only ℓ, and its contents are essentially $R_{\ltimes \ell.\mathbb{J}_L}$. If $\ell.\mathbb{J}_L \succeq w'$, we have $R_{\ltimes \ell.\mathbb{J}_L} \supseteq R_{\ltimes w'}$. Since $R_{\ltimes w'}$ fails Φ, so does $R_{\ltimes \ell.\mathbb{J}_L}$ because of anti-monotonicity. Hence ℓ does not contribute to the final result.

The above arguments show that $\mathsf{prune}(\ell, \mathcal{C})$ of Theorem 3 avoids false negatives. We also argue that it avoids false positives, *i.e.*, pruning will not introduce incorrect result tuples. In general, spurious result tuples may result if multiple L-tuples contribute to the same LR-group. For example, suppose Φ is `COUNT(*) <= 50` (anti-monotone), and a particular candidate LR-group has contribution from ℓ_1 and ℓ_2 (with same $\mathbb{G}_L$ but different $\mathbb{J}_L$ values). Further suppose ℓ_1 is pruned (because we can infer from $\mathcal{C}$ that $\ell_1.\mathbb{J}_L$ would generate more than 50 tuples for the group), but ℓ_2 is processed and contributes only 40 to the count. Unless the algorithm has some way of recording that any LR-group with ℓ_1's contribution is "infeasible," it may incorrectly process the LG-group for output because it sees only ℓ_2's contribution after pruning ℓ_1. Theorem 3 avoids false positives by requiring $\mathbb{G}_L$ to be a superkey of L: in that case, every LR-group is associated with a single L-tuple, so pruning one L-tuple does not affect the computation of LR-groups associated with other L-tuples.

Note that it is possible to store additional information in $\mathcal{C}$ to enable more general pruning conditions than those of Theorem 3. For example, if we define a finer-grained notion of "unpromising" for $\mathbb{J}_L \cup \mathbb{G}_R$ values, and make $\mathcal{C}$ additionally record LR-groups already deemed infeasible, we may be able to generate more pruning opportunities. We leave such extensions as future work.

5.2 Automatic Subsumption Test Generation

The last issue remaining in applying Theorem 3 is that $\mathsf{prune}(\ell, C)$ involves testing subsumption between $w, w' \in \mathtt{adom}(\mathbb{J}_L)$. Definition 4 gives an instance-based definition of $w \succeq w'$, which is impractical to evaluate—its evaluation would have necessitated joining the L-tuple in consideration with R, rendering pruning useless. Instead, we should like to develop a test that can be evaluated by examining w and w' alone, without accessing R. In other words, we seek a predicate $p_\succeq$ involving w and w' alone, such that $p_\succeq(w, w') \Leftrightarrow (w \succeq w'$ for any database instance $\mathcal{D})$.

Example 10. *Consider again the k-skyband query (Listing 2). The instance-oblivious subsumption predicate is $p_\succeq(\langle x, y\rangle, \langle x', y'\rangle) \equiv (x \leq x') \wedge (y \leq y')$. It is easy to see that if this predicate evaluates to true, any R-tuple that joins with $\mathbb{J}_L$ values of $\langle x', y'\rangle$ in Q must also join with $\mathbb{J}_L$ values of $\langle x, y\rangle$. Thus, if $R_{\ltimes \langle x', y'\rangle}$ does not satisfy the anti-monotone* `HAVING` *condition (*`COUNT(*) <= 50`*), neither can $R_{\ltimes \langle x, y\rangle}$.*

Although the above example for k-skyband query is intuitive and $p_\succeq$ can be easily obtained by hand, for arbitrary join conditions, manual derivation can be complex and error-prone. In the following, we give a formal procedure for obtaining $p_\succeq$ automatically by analyzing the join condition Θ. Given the instance-based definition of subsumption in Definition 4, we consider the instance-oblivious version $\forall w_r \in \mathtt{dom}(\mathbb{J}_R) : \Theta(w, w_r) \Rightarrow \Theta(w', w_r)$. Our goal is to essentially eliminate w_r from this predicate, leaving only w and w'. To this end, we use the *Fourier-Motzkin elimination method* [11] (henceforth referred to as *FME*) for the common case where Θ involve only linear constraints. Our implementation uses Mathematica [20], which also handles more complex predicates.

Variable Elimination by FME FME method gives an procedure to eliminate variables from a system of linear constraints involving variables from the real domain, *e.g.*,

$$x \geq y + 500 \ \wedge \ x + 10 \leq z \ \wedge \ x \leq 5y + 100. \qquad (1)$$

FME eliminates variables one by one. Given a variable x to eliminate, FME projects the constraints involving x onto the rest of the system: i) if x is constrained to be equal to an expression involving other variables, we simply replace x the corresponding expression; ii) if x has both lower bounds $(e_1^{\mathsf{l}}, e_2^{\mathsf{l}}, \ldots, e_m^{\mathsf{l}})$ and upper bounds $(e_1^{\mathsf{u}}, e_2^{\mathsf{u}}, \ldots, e_n^{\mathsf{u}})$ in terms of other variables, we add $m \times n$ new constraints to the system as $e_i^{\mathsf{l}} \leq e_j^{\mathsf{u}}$ (or $<$ if either bound is strict) for all $i \in [1, m]$ and $j \in [1, n]$. iii) if x is *unbounded* (*i.e.*, bounded on neither side, or from one side but not the

other), we simply drop x. This projection procedure forms a new system with one variable less, but possibly with more constraints, and the new system is satisfiable if and only if the original system is satisfiable. For example, eliminating x from Eq. (1) results in $y + 500 \leq z - 10 \ \wedge \ y + 500 \leq 5y + 100$.

This FME step can be repeated as needed. Although the number of constraints may increase quadratically in each step, we have found FME to be practical in all scenarios we encountered, as typical SQL queries involve only a moderate number of variables.

Procedure for Deriving $p_{\succeq}$ Using FME Starting with $\forall w_r \in \texttt{dom}(\mathbb{J}_R) : \Theta(w', w_r) \Rightarrow \Theta(w, w_r)$, or $\forall w_r : \neg\Theta(w', w_r) \vee \Theta(w, w_r)$, we expand w_r, w, and w' into variables representing individual attribute values; variables from w_r are universally quantified. Assuming that all attributes are from the real domain, we repeatedly choose and apply one of the following three steps to eliminate all variables from w_r, while preserving those from w and w'. The order of application is unimportant; any applicable step can be chosen.

(UE) *Elimination of universal quantifiers*: Replace $\forall x\, \theta$ by $\neg\exists x\, \neg\theta$.

(DE) *Elimination of disjunction*: Replace $\exists x\, (\theta_1 \vee \theta_2)$ by $(\exists x_1\, \theta_1) \vee (\exists x_2\, \theta_2)$ by renaming variable x to x_1 in θ_1, and to x_2 in θ_2.

(EE) *Elimination of existential quantifiers*: Given a formula in prenex normal form (where all quantifiers precede a quantifier-free formula) $Q_1 x_1 \ \cdots Q_p x_p \ \exists x\ \theta$, where θ only involves conjunction over linear constraints, project it to equivalent $Q_1 x_1 \ \cdots Q_p x_p \ \theta'$, where $x \notin \theta'$, by eliminating variable x from θ using the FME method described above.

Example 11. *Consider again the k-skyband query (Listing 2). We illustrate the above algorithm for deriving $p_{\succeq}$ for a simplified join condition* `L.x<R.x AND L.y<R.y`. *The derivation for the original join condition is more tedious, and we include it for completeness in Appendix B.*

We start from $\forall x_r \forall y_r : (x' < x_r \wedge y' < y_r) \Rightarrow (x < x_r \wedge y < y_r)$, where $w_r = \langle x_r, y_r \rangle$, $w = \langle x, y \rangle$, and $w' = \langle x', y' \rangle$. We have

$$
\begin{aligned}
& \forall x_r \forall y_r \left(\neg(x' < x_r \wedge y' < y_r) \vee (x < x_r \wedge y < y_r)\right) \\
\overset{UE}{\equiv}\ & \neg\left[\exists x_r \exists y_r \left((x' < x_r \wedge y' < y_r) \wedge \neg(x < x_r \wedge y < y_r)\right)\right] \\
\equiv\ & \neg\left[\exists x_r \exists y_r \left(x' < x_r \wedge y' < y_r \wedge (x \geq x_r \vee y \geq y_r)\right)\right] \\
\overset{DE}{\equiv}\ & \neg\big[\exists x_r \big[\left(\exists y_{r1} \left(x' < x_r \wedge y' < y_{r1} \wedge x \geq x_r\right)\right) \vee \\
& \qquad \left(\exists y_{r2} \left(x' < x_r \wedge y' < y_{r2} \wedge y \geq y_{r2}\right)\right) \big]\big] \\
\overset{EE}{\equiv}\ & \neg\left[\exists x_r \left[\left(x' < x_r \wedge x \geq x_r\right) \vee \left(x' < x_r \wedge y' < y\right)\right]\right] \\
\overset{DE}{\equiv}\ & \neg\big[\left(\exists x_{r1} \left(x' < x_{r1} \wedge x \geq x_{r1}\right)\right) \vee \\
& \qquad \left(\exists x_{r2} \left(x' < x_{r2} \wedge y' < y\right)\right) \big] \\
\overset{EE}{\equiv}\ & \neg\left[(x' < x) \vee (y' < y)\right] \\
\equiv\ & (x \leq x') \ \wedge \ (y \leq y').
\end{aligned}
$$

Since Φ for this query is anti-monotone, $\mathsf{prune}(\ell, \mathcal{C}) \equiv \exists \langle x', y' \rangle \in \mathcal{C} : \langle \ell.\mathtt{x}, \ell.\mathtt{y} \rangle \succeq \langle x', y' \rangle$, *which simplifies to* $\exists \langle x', y' \rangle \in \mathcal{C} : (\ell.\mathtt{x} \leq x') \ \wedge \ (\ell.\mathtt{y} \leq y')$.

6 Memoization Techniques

Memoization is an idea that builds upon the classic semijoin-based evaluation techniques for conjunctive queries, but also considers grouping and aggregation in SQL queries. As an example, again consider the k-skyband query in Listing 2. Here, each L-tuple ℓ leads to one potential candidate LR-group. Both the `HAVING` condition Φ and `SELECT` expressions Λ for this LR-group can be evaluated over the joining R-tuples $R_{\ltimes(\ell.\mathtt{x}, \ell.\mathtt{y})}$. These results depend only on the join attribute values $(\ell.\mathtt{x}, \ell.\mathtt{y})$, and can be memoized and keyed on such values. Subsequent evaluations involving the same join attribute values can simply reuse the memoized results.

In our implementation described in Section 7, we support runtime memoization using a cache in our new NLJP operator based on nested-loop execution. Recall from Section 5 that we use a cache $\mathcal{C}$ to record join attribute values that do not contribute to the final query result, for the purpose of pruning. To support memoization, we augment $\mathcal{C}$ to record also the results (or useful intermediate results) of evaluating Φ and Λ, so that we can use them in subsequent iterations if the same join attribute values are encountered.

Currently, our implementation applies memoization when the following conditions hold: $\mathbb{G}_R = \varnothing$, Φ is applicable to R, all aggregates in Λ only involve attributes in R or `*`, and finally, all aggregates in Φ and Λ are *algebraic* [9] unless $\mathbb{G}_L \rightarrow \mathbb{A}_L$. The cache records the results of all aggregate subexpressions in Φ and Λ, keyed by $\mathbb{J}_L$ values. We note that if $\mathbb{G}_L \rightarrow \mathbb{A}_L$ does not hold, then the results of Φ and Λ over an LR-group must be computed by combining multiple cached partial aggregate results (hence the requirement on algebraic aggregates); see Appendix C for more details. Additionally, we check if $\mathbb{J}_L \rightarrow \mathbb{A}_L$; if yes, we do not enable memoization even though it is safe to apply, because in this case the $\mathbb{J}_L$ values will be distinct across L-tuples and memoization will not be beneficial.

An alternative method of applying memoization, without relying on our NLJP operator, is through static query rewrite. In Appendix C, we briefly describe this method, together with a relaxation of the above conditions for applying memoization, which does not require $\mathbb{G}_R = \varnothing$. Finally, the cost-effectiveness of memoization depends on how many L-tuples share the same join attribute values. In the future, instead we plan to investigate cost-based decisions for whether to enable memoization.

7 Execution and Optimization

In this section we discuss how to integrate the techniques presented in the previous sections into the execution and optimization of iceberg queries by a database system. First, we describe our new physical operator for memoization and pruning, which we call NLJP (for *Nested-Loop Join with Pruning*).

Conceptually, an NLJP operator computes an iceberg query Q of the form shown in Listing 5, with L and R both given generally as subqueries. We call L the *driver (or outer) input*, which forms the outer loop of the nested-loop join. The NLJP operator's behavior is specified by the following queries (or parameterized queries):

- *Binding query* Q_B executes L (with selections and projections in Q pushed down) and produces a stream of intermediate result tuples. For each tuple, the *binding*, or the values for the join attributes $\mathbb{J}_L$, is identified.

- *Inner query* $Q_R(b)$ is a select-aggregate query over R parameterized by a binding b. $Q_R(b)$ has selection condition $\Theta(b, \mathbb{J}_R)$ and computes any aggregates necessary for evaluating Φ and Λ in Q. Conceptually, during its execution, NLJP considers bindings produced by Q_B one at a time, evaluates the inner query when necessary, and builds up a cache $\mathcal{C}$ of inner query results keyed by the bindings.

- *Pruning query* $Q_{\mathcal{C}}(b')$ is a selection query over the cache C given a binding b'. Using the pruning predicate derived using the technique in Section 5.2, $Q_P(b')$ looks for the existence of

```
SELECT id, x, y FROM Object L; -- Q_B; binding is (x, y)
SELECT COUNT(*) AS payload FROM Object R -- Q_R(b)
  WHERE b.x <= R.x AND b.y <= R.y
  AND (b.x < R.x OR b.y < R.y);
SELECT * -- Q_C(b')
  FROM C(x, y, payload) -- cache
  WHERE b'.x <= x AND b'.y <= y
  AND payload > 50;
SELECT id, payload -- Q_P
  FROM T(id, x, y, payload) -- concatenated tuples
  WHERE payload <= 50;
```

Listing 7: NLJP-based plan for k-skyband.

a cached result with binding b implying that b' can be safely pruned.

- *Post-processing query* Q_P computes the final result of Q. NLJP concatenates each intermediate result tuple of Q_B with binding b (if not pruned) with the result tuple of the corresponding $Q_R(b)$, and evaluates Q_p over these concatenated tuples. In the simple case where $\mathbb{G}_L \rightarrow \mathbb{A}_L$, Q_P can be computed on a per-tuple basis; in more general cases Q_P may involve further aggregation (Appendix C).

For example, for the k-skyband query in Listing 2, the queries specifying the NLJP operator are shown in Listing 7.

We now sketch out the execution of an NLJP operator. In the following, we assume that both memoization and pruning are enabled (it is straightforward to turn off either feature if desired).

```
construct cache C
T ← ∅
for each t_B with binding b in the result of Q_B:
    if b in C:
        T.append(⟨t_B, C[b]⟩)
    else if Q_C(b) is empty: # cannot be pruned
        C[b] ← Q_R(b)
        T.append(⟨t_B, C[b]⟩)
return Q_P(T)
```

We have implemented the NLJP "operator" in PostgreSQL as a stored procedure. Our optimization procedure (discussed further in Appendix D) can be thought of as a pre-compiler that analyzes the original query Q and rewrites it as a combination of SQL and stored procedure code. For each specific instance of the NLJP operator, the procedure automatically generates the code for its queries, as well as statements for creating and maintaining the cache, which is implemented as a PostgreSQL table. One advantage of this approach is that PostgreSQL optimizer is able to prepare and optimize these statements in advance, which is especially important for parameterized statements that will be executed repeatedly at run time, such as Q_R, Q_C, and cache operations.

A number of other features of the NLJP operator are worth noting here. First, we detect simple cases when the post-processing $Q_P(T)$ can be evaluated in an incremental fashion over T, to avoid materialization of T and allow final result tuples to be returned in a non-blocking fashion. Second, the binding query Q_B can optionally include an `ORDER BY` clause to control the order of exploration in the space of all bindings. We simply leave the ordering unspecified (*i.e.*, whatever ordering PostgreSQL chooses to return the result tuples of Q_B) in this paper, but we plan to consider intelligently choosing this ordering in the future as it can have a significant impact on pruning effectiveness [19]. Finally, we can outfit the cache $\mathcal{C}$ with a replacement policy (based on the utility of its entries) to bound its size; we plan to investigate this option as future work.

In Appendix D, we give an optimization procedure to systematically look for opportunities to apply all our techniques—generalized a-priori, memoization, and pruning—in an iceberg query involving multi-way joins. There we also walk through the steps of optimizing the query in Listing 3 involving a four-way self-join, showing how we are able to identify and apply both generalized a-priori and pruning techniques to that same query. However, note that doing so requires a component of the optimization procedure that infers functional dependencies that hold in a join result by analyzing the join predicate and dependencies on the input relations. We do not yet have a full implementation of this procedure, so the experiments in Section 8 involving Listing 3 applies only the pruning technique, but not generalized a-priori. This temporary limitation is not inherent to our optimization framework for iceberg queries.

8 Experiments

All experiments are run on a quad-core 2.8GHz Intel i7 machine with 16GB memory and 2TB HD, running Microsoft Windows Server 2012 Datacenter. We compare our PostgreSQL-based implementation with basic PostgreSQL and a commercial database system, which we refer to as "Vendor A". Unless otherwise noted, all tables are given indexes applicable to each query. Parallelism was enabled by default for Vendor A and for PostgreSQL, with Vendor A using all 4 cores and PostgreSQL preferring to use 2. Our implementation does not take any specific advantage of parallelism.

We evaluated our techniques on a representative set of queries involving dataset of season statistics for Major League Baseball [2], with 3×10^5 rows, each with a large number of player performance statistics. Our queries largely follow the templates of k-skyband (Listing 2), "pairs" (Listing 4), and "unexciting products" (Listing 3) queries, but they are cast in the setting of baseball players and statistics and sometimes contain other variations. In the following, we refer to these queries as *skyband*, *pairs*, and *complex*, respectively. *Complex* uses an alternative, "unpivoted" organization of the same dataset, where each individual performance statistic is represented as a key-value pair by a row.

8.1 Relative Performance of Approaches

Figure 1 compares the performance of our approach (shown as "all") versus PostgreSQL (shown as "base") and Vendor A on 8 representative queries. For our approach, we also show the performance achieved by enabling only one of our three optimization techniques—pruning, memoization (shown as "memo"), and generalized a-priori (shown as "apriori").

Q_1, Q_2, and Q_3 are two-dimensional *skyband* queries with varying maximum thresholds (k) comparing different pairs of attributes (a); the objects of interests are all seasonal performance records of players. Q_4, Q_5, Q_6, and Q_7 are *pairs* query (Listing 4) with varying thresholds (c for the `HAVING` minimum in the `WITH` subquery, and k for the `HAVING` maximum in the main query) and different functions for aggregating statistics over time (using either `SUM` or `AVG` in the `WITH` subquery). In contrast to Q_1–Q_3, Q_8 first computes average player statistics over time before computing the skyband, so its objects of interest are players; it also uses a simpler join condition (`L.x < R.x AND L.y < R.y`).

From Figure 1, we see that our approach yields a tremendous speedup—consistently over PostgreSQL and in most cases over Vendor A—when all optimization techniques are enabled. There only exceptions were Q_7 and Q_8, where Vendor A offers slightly better performance—but keep in mind that Vendor A makes aggressive use of parallelism while our implementation is currently sequential. As a point reference, in these two cases PostgreSQL still fares far worse than our approach, which is also PostgreSQL-based.

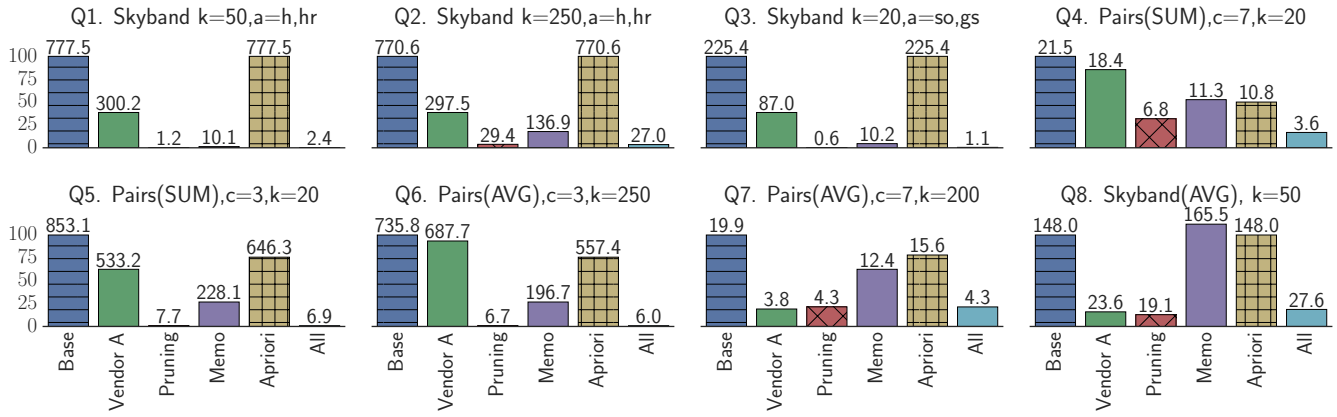

Figure 1: Performance of our approach (with one of or all three optimization techniques enabled) vs. PostgreSQL and Vendor A. Heights of the bars correspond to running times (normalized against PostgreSQL's), with actual running times in seconds also shown on top. Note that generalized a-priori does not apply to Q_1, Q_2, Q_3, and Q_8.

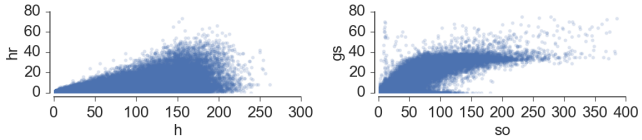

Figure 2: Data distributions for two common attribute pairings used in our experiments.

Figure 3: Cache sizes (kB) at the end of execution across eight queries.

To study the effectiveness of each our optimization techniques, we have also experimented with enabling each in isolation, but it should be noted that in some cases a particular optimization would not be applicable to a specific query. For highly selective iceberg queries, pruning can be extremely effective, with a few queries seeing more than 300× speedup. While a-priori yields the smallest isolated speedup, there appears to be a combination effect that is not entirely linear, so it can be beneficial when used in conjunction with the other techniques. Memoization, despite being the easiest optimization to apply, can sometimes provide good speedups (*e.g.*, more than 20× for Q_1, Q_2, and Q_3) even by itself; even though these speedups are not as good as pruning, memoization has the advantage of being more generally applicable.

Data Distribution It is important to note that data distributions can vary significantly across attributes and combinations thereof. Consequently, as we have observed in Figure 1, even two queries of the same template query running on the same dataset can have very different performance. For example, Figure 2 shows the data distributions over two commonly used attribute pairs in our queries. A *skyband* query with $k = 500$, running on the distribution shown on the left, would return 1.8% of all records, but the same query running on the distribution on the right would return 3.1% of all records.

Cache Size Recall that our NLJP operator uses a cache, implemented as a PostgreSQL table. Figure 3 reports the size of this cache at the end of the execution for the 8 queries above. No cache is larger than 3,000kB, and most are smaller than 500kB. Mean cache size is 571kB, with 10,371 rows on average. It should be noted, however, that for one query, Q_5, the number of rows in the cache was over 60% of the size of its input table, because of the (effectively) four-way join and a very large number of unpromising

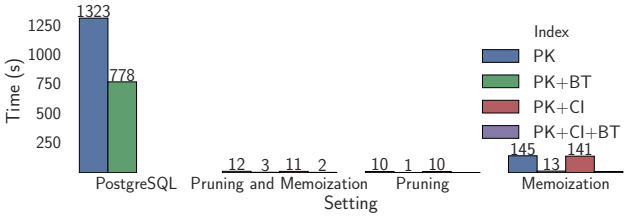

Figure 4: Comparison of query execution times for Q_1 under different index configurations.

candidates. While our cache is not constrained by memory because it is implemented as a PostgreSQL, intelligent cache replacement policies may still improve performance, and we plan to consider them as future work.

Use of Indexes Availability of indexes does influence query plans and performance. As an example, Figure 4 shows the effect of indexes had on PostgreSQL and the PostgreSQL-based implementation of our approach, with various combination of optimization techniques turned on. The query is Q_1 (recall that generalized a-priori does not apply). We tested a number of configurations, where *PK* stands for the primary-key index on the input table (always available), *BT* stands for a secondary B-tree index on attributes (`h`, `hr`) involved in comparison, and *CI* stands for a primary-key index on the cache (on the join attribute values in this case), which is applicable only to our approach. As can be seen in Figure 4, PostgreSQL is able to leverage *BT* to gain a 2× speedup from its *PK*-only execution. In comparison, with both pruning and memoization on, even the worse case for our approach, with only *PK*, offers 64× speedup over PostgreSQL *PK+BT*. With *PK+BT+CI*, we get another 6× speedup. For other queries, we also observe that *BT* and *CI* generally improve performance for all approaches; hence, results shown in Figure 1 earlier were obtained with both *BT* and *CT* in addition to *PK*.

Query Plans It is illustrative to examine the query plans of PostgreSQL and Vendor A for iceberg queries. For the relative simpler *skyband* queries such as Q_1, both PostgreSQL and Vendor A use indexed nested loop joins, followed by hash-based grouping and aggregation, and then final filtering by the `HAVING` condition (shown in Appendix E for reference). While such plans are reasonable, they are clearly inferior to our approach, as they fully evaluate the joins and apply the highly selective `HAVING` condition only in the very last step.

For *pairs* queries, which involve a larger number of joins, we see more variety in the query plans by PostgreSQL and Vendor A for

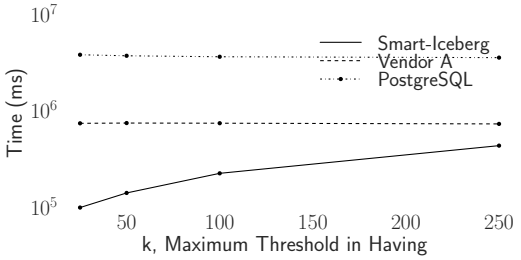

Figure 5: Running times (on a logarithmic scale) of *skyband* with varying thresholds in the `HAVING` condition.

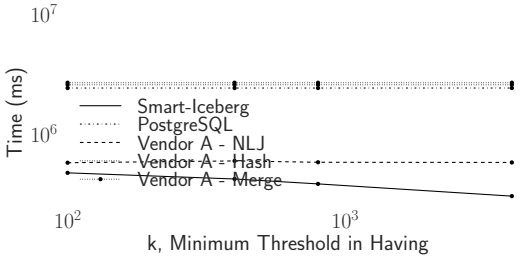

Figure 6: Running times (on a logarithmic scale) of *complex* with varying thresholds in the `HAVING` condition.

join processing, although they are still unable to take advantage of the selective `HAVING` conditions in any way. Dropping indexes (removing *BT*) also has the effect of forcing PostgreSQL and Vendor A to consider more join methods, but doing so generally leads to less efficient plans.

8.2 Effect of Workload Parameters

We further evaluate our approach when varying workload parameters, specifically the threshold used in the `HAVING` condition and the input data size. We use a *skyband* query (analogous to Listing 2) and a *complex* query (analogous to Listing 3). Generalized a-priori is not applicable to *skyband*. While *complex* benefits from simultaneous application of a-priori and pruning, as explained at the end of Section 7, we only test pruning and memoization for *complex* because of a limitation in our current implementation of the optimization procedure.

Varying the `HAVING` Threshold Figure 5 shows the running times of *skyband* on the three systems as we vary the `HAVING` threshold while keeping the input table size constant at 3×10^5 rows. Note that the y-axis uses logarithmic scale. Our system, shown as "Smart-Iceberg" performs significantly better than PostgreSQL and Vendor A. The query plans and running times of PostgreSQL and Vendor A are largely independent of the threshold, because they apply `HAVING` in the very last step of query processing. On the other hand, our system is able to exploit the highly selective nature of iceberg queries. As the `HAVING` threshold increases, the iceberg query becomes less "picky," so the advantage of our system gradually diminishes. Nonetheless, our system still beats the other systems even when the threshold is as high as 250, which returns a very large number of result tuples—a behavior that is typically not expected or desired for iceberg queries.

Figure 8 shows the *complex* query running times as we vary the threshold while fixing the input table size constant at 2×10^5 rows (recall that *complex* uses the unpivoted version of the table, and we limit its size to cap the running times of other systems). For Vendor A, we hint its optimizer to consider different join plans; indexed nested loop join performs the best. Again, our system outperforms others. The margin is smaller compared with Figure 5 because of *complex*'s four-way join (generalized a-priori would have helped). Also note that as we increase the `HAVING` threshold, *complex* becomes more "picky," so the advantage of our system increases, which is the reverse of the effect observed in Figure 5.

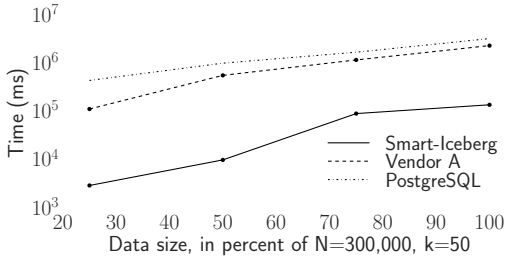

Figure 7: Running times (on a logarithmic scale) of *skyband* when the input table size varies.

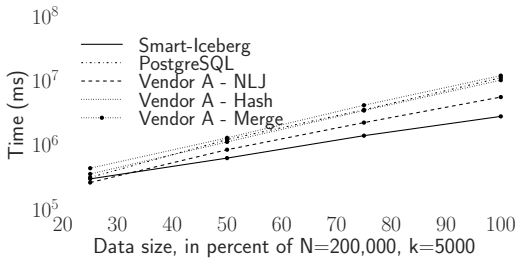

Figure 8: Running times (on a logarithmic scale) of *complex* when the input table size varies.

Varying Input Table Size Figures 7 and 8 show the effects of varying the input table size on *skyband* and *complex*, respectively. The `HAVING` thresholds are kept constant. As expected, for all systems, queries take longer to run, but our system generally performs the best. The only exception is that *complex* on Vendor A is faster for the very small input table size of 50,000 rows. Here, the setting of 5000 as the `HAVING` threshold is such that the query is not selective at all. Still, our system is only slightly slower in this case.

9 Conclusions

In this paper, we have introduced a framework with a suite of synergistic techniques—generalized a-priori, pruning, and memoization—for evaluating iceberg queries with complex joins. Given a query and knowledge of schema, we provide formal checks for safe application of these techniques. Importantly, we do not rely on users to identify optimization opportunities or specify necessary logic (such as pruning predicates) to apply them. Instead, we show that we can detect and apply optimizations automatically, thereby significantly reducing development effort and potential human errors. Experiments demonstrate that our *Smart-Iceberg* system, implemented using PostgreSQL, provides substantial performance improvements over existing database systems for complex iceberg queries, allowing quicker "time to insight" for users.

There are many promising avenues for future work. A necessary step towards greater adoption of our techniques is the integration of cost-based optimization into our framework. In addition, while we have largely considered single-block queries, there is an additional layer of complexity with nested queries that has been largely unanalyzed in research on iceberg queries. Within our framework, order of evaluation and cache replacement policies for NLJP are also worth further investigation. In conclusion, we believe that iceberg queries are a fundamental class of queries used in many applications, and there exists an intriguing possibility of consolidating and generalizing specialized techniques that have been developed for specific problem instances.

References

[1] http://www.500hrc.com/500-hrc-articles/cubs-teammates-compete-to-be-next-to-300.html.

[2] http://www.seanlahman.com/baseball-archive/statistics/. 2017.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.

[4] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *SIGMOD*, pages 1–15. ACM, 1985.

[5] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cube. In *ACM SIGMOD Record*, volume 28, pages 359–370. ACM, 1999.

[6] C.-Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. Finding k-dominant skylines in high dimensional space. In *SIGMOD*, pages 503–514, 2006.

[7] J. Chomicki, P. Ciaccia, and N. Meneghetti. Skyline queries, front and back. *SIGMOD Rec.*, 42(3):6–18, Oct. 2013.

[8] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *VLDB*. Stanford InfoLab, 1999.

[9] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, Jan. 1997.

[10] A. Kalinin, U. Cetintemel, and S. Zdonik. Searchlight: Enabling integrated search and exploration over large multidimensional data. In *VLDB*, volume 8, pages 1094–1105. VLDB Endowment, 2015.

[11] L. Khachiyan. *Fourier–Motzkin elimination method*, pages 1074–1077. Encyclopedia of Optimization, Springer US, Boston, MA, 2009.

[12] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *ACM SIGMOD Record*, volume 19, pages 247–258. ACM, 1990.

[13] R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. *SIGMOD Rec.*, 27(2):13–24, June 1998.

[14] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, Mar. 2005.

[15] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. *SIGMOD Rec.*, 27(2):343–354, June 1998.

[16] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *ACM SIGMOD Record*, volume 25, pages 435–446. ACM, 1996.

[17] Y. Shou, N. Mamoulis, H. Cao, D. Papadias, and D. W. Cheung. Evaluation of iceberg distance joins. In *SSTD*, pages 270–288. Springer, 2003.

[18] D. Tsur, J. D. Ullman, S. Abiteboul, C. Clifton, R. Motwani, S. Nestorov, and A. Rosenthal. Query flocks: A generalization of association-rule mining. In *ACM SIGMOD Record*, volume 27, pages 1–12. ACM, 1998.

[19] B. Walenz and J. Yang. Perturbation analysis of database queries. In *VLDB*, volume 9, pages 1635–1646, 2016.

[20] Wolfram Research Inc. *Mathematica Online*, 2016.

[21] W. Yan and P.-A. Larson. Interchanging the order of grouping and join. Technical report, Technical Report CS 95-09, Dept. of Computer Science, University of Waterloo, Canada, 1995.

[22] W. P. Yan and P.-A. Larson. Performing group-by before join [query processing]. In *Data Engineering, 1994. Proceedings. 10th International Conference*, pages 89–100. IEEE, 1994.

[23] W. P. Yan, P.-B. Larson, et al. Eager aggregation and lazy aggregation. In *VLDB*, volume 95, pages 345–357, 1995.

APPENDIX

A Proof of Theorem 3

Before proceeding with the proofs, we introduce some additional notation. Let $\mathfrak{A}$ denote the algorithm with prune specified by Theorem 3, and let $\mathfrak{A}_0$ denote the baseline algorithm where prune always returns false. Given a database instance, let T denote the collection of contributions computed by $\mathfrak{A}$, and let T_0 denote the collection of contributions computed by $\mathfrak{A}_0$. Clearly $T \subseteq T_0$.

Lemma 1. *Suppose $\mathbb{G}_L \to \mathbb{A}_L$. For any (non-empty) LR-group $LR^{(u,v)}$, let ℓ denote the unique L-tuple with $\ell.\mathbb{G}_L = u$. We have:*

- $\pi_{\mathbb{A}_L} LR^{(u,v)} = \{\ell\}$;
- $\pi_{\mathbb{A}_R} LR^{(u,v)} = \sigma_{\mathbb{G}_R = v} R_{\ltimes \ell.\mathbb{J}_L}$;
- $T_0[u,v]$ *consists of a single contribution* $\mathsf{contrib}(\ell, u, v)$.

Proof. Follows directly from the definitions. □

Lemma 2. *Suppose $\mathbb{G}_L \to \mathbb{A}_L$ and Φ is applicable to R. If Q produces a result tuple for $LR^{(u,v)}$, then for any L-tuple ℓ where $\ell.\mathbb{G}_L = u$ and $v \in R_{\ltimes \ell.\mathbb{J}_L}$, we have $\Phi(\sigma_{\mathbb{G}_R = v} R_{\ltimes \ell.\mathbb{J}_L})$.*

Proof. Since Φ is applicable to R, $\Phi(LR^{(u,v)}) \Leftrightarrow \Phi(\pi_{\mathbb{A}_R} LR^{(u,v)})$. Because $\mathbb{G}_L \to \mathbb{A}_L$, by Lemma 1, $\pi_{\mathbb{A}_R} LR^{(u,v)} = \sigma_{\mathbb{G}_R = v} R_{\ltimes \ell.\mathbb{J}_L}$. Therefore $\Phi(\sigma_{\mathbb{G}_R = v} R_{\ltimes \ell.\mathbb{J}_L})$. □

Proof of Theorem 3. First, we show that $\mathfrak{A}$ produces no false negatives. More precisely, suppose Q ($\mathfrak{A}_0$) produces an result tuple for an LR-group $LR^{(u,v)}$ from contributions $T_0[u,v]$. We claim that $\mathfrak{A}$ will produce the same result tuple as it will produce these contributions too: *i.e.*, $T[u,v] = T_0[u,v]$. Since $T \subseteq T_0$, it suffices to show that $T_0[u,v] \subseteq T[u,v]$. Suppose to the contrary there exists some $\mathsf{contrib}(\ell, u, v) \in T_0[u,v]$ but $\mathsf{contrib}(\ell, u, v) \notin T[u,v]$. $\mathfrak{A}$ must have pruned ℓ using some unpromising $w' \in \mathtt{dom}(\mathbb{J}_L)$.

(Monotone Φ) In this case, $\mathbb{G}_L \to \mathbb{A}_L$ and $\ell.\mathbb{J}_L \preceq w'$. By Lemma 2, $\Phi(\sigma_{\mathbb{G}_R = v} R_{\ltimes \ell.\mathbb{J}_L})$. Since $\ell.\mathbb{J}_L \preceq w'$, we have $R_{\ltimes \ell.\mathbb{J}_L} \subseteq R_{\ltimes w'}$ and therefore $\sigma_{\mathbb{G}_R = v} R_{\ltimes \ell.\mathbb{J}_L} \subseteq \sigma_{\mathbb{G}_R = v} R_{\ltimes w'}$. As Φ is monotone, $\Phi(\sigma_{\mathbb{G}_R = v} R_{\ltimes \ell.\mathbb{J}_L}) \Rightarrow \Phi(\sigma_{\mathbb{G}_R = v} R_{\ltimes w'})$, contradicting the fact that w' is unpromising.

(Anti-monotone Φ) In this case, $\mathbb{G}_L \to \mathbb{A}_L$, $\mathbb{G}_R = \varnothing$ (which implies $v = \langle\rangle$), and $\ell.\mathbb{J}_L \succeq w'$. By Lemma 2, $\Phi(R_{\ltimes \ell.\mathbb{J}_L})$. Since $\ell.\mathbb{J}_L \succeq w'$, we have $R_{\ltimes \ell.\mathbb{J}_L} \supseteq R_{\ltimes w'}$, and because Φ is anti-monotone, $\Phi(R_{\ltimes \ell.\mathbb{J}_L}) \Rightarrow \Phi(R_{\ltimes w'})$, contradicting the fact that w' is unpromising.

Next, we show that $\mathfrak{A}$ produces no false positives. More precisely, suppose Q ($\mathfrak{A}_0$) produces no result tuple for an LR-group $LR^{(u,v)}$ from contributions $T_0[u,v]$. We claim that $\mathfrak{A}$ will produced either all these contributions or none at all: *i.e.*, $T[u,v] = T_0[u,v]$ or $T[u,v] = \varnothing$. In the first case of $T[u,v] = T_0[u,v]$, $\mathfrak{A}$ would behave identically as $\mathfrak{A}_0$ and not produce a result tuple for

$LR^{(u,v)}$ (because it fails Φ). In the second case of $T[u,v] = \varnothing$, $\mathfrak{A}$ would simply ignore $LR^{(u,v)}$, which is again correct.

It remains to be shown why $T[u,v]$ is either $T_0[u,v]$ or $\varnothing$. If $\mathfrak{A}$ never prunes, obviously $T = T_0$ and $T[u,v] = T_0[u,v]$. If $\mathfrak{A}$ ever prunes, it must be the case that $\mathbb{G}_L \to \mathbb{A}_L$. By Lemma 1, $T_0[u,v]$ consists of a single contribution from a single L-tuple, say ℓ. If $\mathfrak{A}$ prunes ℓ, $T[u,v] = \varnothing$; otherwise, $T[u,v] = T_0[u,v]$. Hence the proof is complete. □

B Additional Example of Subsumption Test Generation

Example 12. *Consider again the k-skyband query (Listing 2). We now illustrate the algorithm in Section 5.2 for deriving $p_{\succeq}$ for the original join condition* `L.x <= R.x AND L.y <= R.y AND (L.x < R.x OR L.y < R.y)`. *Let $w_r = \langle x_r, y_r\rangle$, $w = \langle x, y\rangle$, and $w' = \langle x', y'\rangle$. We have:*

$$
\begin{aligned}
p_{\succeq} \equiv\ & \forall x_r \forall y_r \big[\neg\big(x' \le x_r \wedge y' \le y_r \wedge (x' < x_r \vee y' < y_r)\big) \vee \\
& \qquad (x \le x_r \wedge y \le y_r \wedge (x < x_r \vee y < y_r))\big] \\
\overset{UE}{\equiv}\ & \neg\big[\exists x_r \exists y_r \big[\big(x' \le x_r \wedge y' \le y_r \wedge (x' < x_r \vee y' < y_r)\big) \wedge \\
& \qquad \neg\,(x \le x_r \wedge y \le y_r \wedge (x < x_r \vee y < y_r))\big]\big] \\
\equiv\ & \neg\big[\exists x_r \exists y_r \big[\big(x' \le x_r \wedge y' \le y_r \wedge (x' < x_r \vee y' < y_r)\big) \wedge \\
& \qquad (x > x_r \vee y > y_r \vee (x \ge x_r \wedge y \ge y_r))\big]\big] \\
\equiv\ & \neg\big[\exists x_r \exists y_r \big[\big((x' < x_r \wedge y' \le y_r) \vee (x' \le x_r \wedge \vee y' < y_r)\big) \\
& \qquad \wedge ((x \ge x_r \vee y > y_r) \wedge (x > x_r \vee y \ge y_r))\big]\big] \\
\equiv\ & \neg\big[\exists x_r \exists y_r \big[(x' < x_r \wedge y' \le y_r) \wedge (x \ge x_r \vee y > y_r) \\
& \qquad \wedge (x > x_r \vee y \ge y_r)\big] \vee \\
& \qquad \big[(x' \le x_r \wedge y' < y_r) \wedge (x \ge x_r \vee y > y_r) \\
& \qquad \wedge (x > x_r \vee y \ge y_r)\big]\big] \\
\overset{DE}{\equiv}\ & \neg\big[\exists x_r \big[\exists y_{r1} \big[(x' < x_r \wedge y' \le y_{r1}) \wedge (x \ge x_r \vee y > y_{r1}) \\
& \qquad \wedge (x > x_r \vee y \ge y_{r1})\big] \vee \\
& \qquad \exists y_{r2} \big[(x' \le x_r \wedge y' < y_{r2}) \wedge (x \ge x_r \vee y > y_{r2}) \\
& \qquad \wedge (x > x_r \vee y \ge y_{r2})\big]\big] \\
\equiv\ & \neg[\exists x_r [A \vee B]].
\end{aligned}
$$

$$
\begin{aligned}
\textit{Now, } A \equiv\ & \exists y_{r1} \big[(x' < x_r \wedge y' \le y_{r1}) \wedge \\
& \qquad (x \ge x_r \vee y > y_{r1}) \wedge (x > x_r \vee y \ge y_{r1})\big] \\
\equiv\ & \exists y_{r1} \big[\big(x' < x_r \wedge y' \le y_{r1} \wedge x \ge x_r \wedge x > x_r\big) \vee \\
& \qquad \big(x' < x_r \wedge y' \le y_{r1} \wedge x \ge x_r \wedge y \ge y_{r1}\big) \vee \\
& \qquad \big(x' < x_r \wedge y' \le y_{r1} \wedge y > y_{r1} \wedge x > x_r\big) \vee \\
& \qquad \big(x' < x_r \wedge y' \le y_{r1} \wedge y > y_{r1} \wedge y \ge y_{r1}\big)\big] \\
\overset{DE}{\equiv}\ & \exists y_{r11} \big(x' < x_r \wedge y' \le y_{r11} \wedge x > x_r\big) \vee \\
& \exists y_{r12} \big(x' < x_r \wedge y' \le y_{r12} \wedge x \ge x_r \wedge y \ge y_{r12}\big) \vee \\
& \exists y_{r13} \big(x' < x_r \wedge y' \le y_{r13} \wedge y > y_{r13} \wedge x > x_r\big) \vee \\
& \exists y_{r14} \big(x' < x_r \wedge y' \le y_{r14} \wedge y > y_{r14}\big) \\
\overset{EE}{\equiv}\ & \big(x' < x_r \wedge x > x_r\big) \vee \big(x' < x_r \wedge y' \le y \wedge x \ge x_r\big) \vee \\
& \big(x' < x_r \wedge y' < y \wedge x > x_r\big) \vee \big(x' < x_r \wedge y' < y\big).
\end{aligned}
$$

$$
\begin{aligned}
\textit{And, } B \equiv\ & \exists y_{r2} \big[(x' < x_r \wedge y' \le y_{r2}) \wedge \\
& \qquad (x \ge x_r \vee y > y_{r2}) \wedge (x > x_r \vee y \ge y_{r2})\big] \\
\equiv\ & \exists y_{r2} \big[\big(x' < x_r \wedge y' \le y_{r2} \wedge x \ge x_r \wedge x > x_r\big) \vee \\
& \qquad \big(x' < x_r \wedge y' \le y_{r2} \wedge x \ge x_r \wedge y \ge y_{r2}\big) \vee \\
& \qquad \big(x' < x_r \wedge y' \le y_{r2} \wedge y > y_{r2} \wedge x > x_r\big) \vee \\
& \qquad \big(x' < x_r \wedge y' \le y_{r2} \wedge y > y_{r2} \wedge y \ge y_{r2}\big)\big] \\
\overset{DE}{\equiv}\ & \exists y_{r21} \big(x' < x_r \wedge y' \le y_{r21} \wedge x > x_r\big) \vee \\
& \exists y_{r22} \big(x' < x_r \wedge y' \le y_{r22} \wedge x \ge x_r \wedge y \ge y_{r22}\big) \vee \\
& \exists y_{r23} \big(x' < x_r \wedge y' \le y_{r23} \wedge y > y_{r23} \wedge x > x_r\big) \vee \\
& \exists y_{r24} \big(x' < x_r \wedge y' \le y_{r24} \wedge y > y_{r24}\big) \\
\overset{EE}{\equiv}\ & \big(x' < x_r \wedge x > x_r\big) \vee \big(x' < x_r \wedge y' \le y \wedge x \ge x_r\big) \vee \\
& \big(x' < x_r \wedge y' < y \wedge x > x_r\big) \vee \big(x' < x_r \wedge y' < y\big).
\end{aligned}
$$

Combining A and B, we have

$$
\begin{aligned}
p_{\succeq} \equiv\ & \neg[\exists x_r [A \vee B]] \\
\overset{DE}{\equiv}\ & \neg\big[\exists x_{r1} \big(x' < x_{r1} \wedge x > x_{r1}\big) \vee \\
& \qquad \exists x_{r2} \big(x' < x_{r2} \wedge y' \le y \wedge x \ge x_{r2}\big) \vee \\
& \qquad \exists x_{r3} \big(x' < x_{r3} \wedge y' < y \wedge x > x_{r3}\big) \vee \\
& \qquad \exists x_{r4} \big(x' < x_{r4} \wedge y' < y\big) \vee \\
& \qquad \exists x_{r5} \big(x' < x_{r5} \wedge x > x_{r5}\big) \vee \\
& \qquad \exists x_{r6} \big(x' < x_{r6} \wedge y' \le y \wedge x \ge x_{r6}\big) \vee \\
& \qquad \exists x_{r7} \big(x' < x_{r7} \wedge y' < y \wedge x > x_{r7}\big) \vee \\
& \qquad \exists x_{r8} \big(x' < x_{r8} \wedge y' < y\big)\big] \\
\overset{EE}{\equiv}\ & \neg\big[\big(x' < x\big) \quad \vee \quad \big(x' < x \wedge y' \le y\big) \vee \\
& \qquad \big(x' < x \wedge y' < y\big) \quad \vee \quad \big(y' < y\big) \vee \\
& \qquad \big(x' < x\big) \quad \vee \quad \big(x' < x \wedge y' \le y\big) \vee \\
& \qquad \big(x' < x \wedge y' < y\big) \quad \vee \quad \big(y' < y\big)\big] \\
\equiv\ & x \le x' \wedge y \le y'.
\end{aligned}
$$

Since Φ for this query is anti-monotone, $\mathsf{prune}(\ell, \mathcal{C}) \equiv \exists \langle x', y'\rangle \in \mathcal{C} : \langle \ell.\mathtt{x}, \ell.\mathtt{y}\rangle \succeq \langle x', y'\rangle$*, which simplifies to $\exists \langle x', y'\rangle \in \mathcal{C} : \ell.\mathtt{x} \le x' \wedge \ell.\mathtt{y} \le y'$. This check turns out to be the same as that derived in Example 11 for a simpler join condition, although in general it does not need to be.*

C Additional Details on Memoization

Here we provide additional details on handling algebraic aggregates when $\mathbb{G}_L \to \mathbb{A}_L$ does not hold on L, generalization to the case of $\mathbb{G}_R \neq \varnothing$, and how to apply memoization using static query rewrite instead of our NLJP operator.

Recall from [9] that an *algebraic* aggregate function f is one that can be characterized by a pair of aggregate functions $(f^{\mathsf{i}}, f^{\circ})$, both with bound-size output, such that given a partitioning of the input set S of tuples into $\{S_1, S_2, \ldots, S_n\}$, $f(S) = f^{\circ}(\{f^{\mathsf{i}}(S_i) \mid i = 1, 2, \ldots, n\})$. Examples of algebraic aggregates in SQL (without `DISTINCT` on input) include `SUM`, `MIN`, `MAX`, where both f^{i} and f° are the same as f itself; `COUNT`, where f^{i} is `COUNT` and f° is `SUM`; and `AVG`, where f^{i} computes and returns both `SUM` and `COUNT` for its input, while f° adds up all sums and counts respectively across its inputs and returns the ratio.

Consider again the generic query Q in Listing 5. We can apply memoization using static query rewrite as shown in Listing 8, provided that Φ is applicable to R, all aggregates in Λ only involve

```
-- The case when 𝔾_L → 𝔸_L:
WITH
  LJT AS (SELECT DISTINCT 𝕁_L FROM L),
  LJR AS (SELECT 𝕁_L, 𝔾_R,
            f_{Λ,1}(E_{Λ,1}) AS A_{Λ,1}, f_{Λ,2}(E_{Λ,2}) AS A_{Λ,2}, ...
          FROM LJT, R WHERE Θ GROUP BY 𝕁_L, 𝔾_R HAVING Φ)
SELECT 𝔾_L, 𝔾_R, Λ^a(A_{Λ,1}, A_{Λ,2}, ...)
FROM L NATURAL JOIN LJR ON 𝕁_L
GROUP BY 𝔾_L, 𝔾_R;
-- The case when 𝔾_L → 𝔸_L does not hold:
WITH
  LJT AS (SELECT DISTINCT 𝕁_L FROM L),
  LJR AS (SELECT 𝕁_L, 𝔾_R,
            f^i_{Λ,1}(E_{Λ,1}) AS A_{Λ,1}, f^i_{Λ,2}(E_{Λ,2}) AS A_{Λ,2}, ...,
            f^i_{Φ,1}(E_{Φ,1}) AS A_{Φ,1}, f^i_{Φ,2}(E_{Φ,2}) AS A_{Φ,2}, ...
          FROM LJT, R WHERE Θ GROUP BY 𝕁_L, 𝔾_R)
SELECT 𝔾_L, 𝔾_R, Λ^a(f°_{Λ,1}(A_{Λ,1}), f°_{Λ,2}(A_{Λ,2}), ...)
FROM L NATURAL JOIN LJR ON 𝕁_L
GROUP BY 𝔾_L, 𝔾_R
HAVING Φ^a(f°_{Φ,1}(A_{Φ,1}), f°_{Φ,2}(A_{Φ,2}), ...);
```

Listing 8: Memoization through static query rewriting.

attributes in R or $*$, and finally, all aggregates in Φ and Λ are algebraic unless $\mathbb{G}_L \to \mathbb{A}_L$. Note that we do not assume $\mathbb{G}_R = \varnothing$. For Listing 8, let $\Phi = \Phi^{\mathsf{a}}(f_{\Phi,1}(E_{\Phi,1}), f_{\Phi,2}(E_{\Phi,2}), \ldots)$, where each $f_{\Phi,i}(E_{\Phi,i})$ denotes an aggregate subexpression of Φ with aggregate function $f_{\Phi,i}$ and non-aggregate input expression $E_{\Phi,i}$; similarly, let $\Lambda = \Lambda^{\mathsf{a}}(f_{\Lambda,1}(E_{\Lambda,1}), f_{\Lambda,2}(E_{\Lambda,2}), \ldots)$.

The first query in Listing 8 shows the simpler case where $\mathbb{G}_L \to \mathbb{A}_L$. Here we know each LR-group comes from a single L-tuple, although one L-tuple may produce multiple LR-groups (because $\mathbb{G}_R \neq \varnothing$). Intuitively, the rewritten query first computes the set of join attribute values from L as LJT (analogous to the binding query Q_B for the caching-based implementation discussed in Section 7). Next, we join LJT with R and compute the result groups for each possible combination of join attribute values, discarding any group that does not satisfy Φ. (As for Λ, note that we can evaluate all its aggregate subexpressions at this point, but full evaluation of Λ may require $\mathbb{G}_L$ and therefore must be done later.) Finally, we join the result groups back with L to obtain the final result.

The second query in Listing 8 shows the more complex case where $\mathbb{G}_L \to \mathbb{A}_L$ does not hold, and we assume that all $f_{\Phi,1}$'s and $f_{\Lambda,1}$'s are algebraic. In this case, multiple L-tuples with possibly different $\mathbb{J}_L$ values may contribute to the same LR-group, so we cannot fully evaluate Φ or Λ when computing LJR. Instead, we use $f^{\mathsf{i}}_{\Phi,1}$'s and $f^{\mathsf{i}}_{\Lambda,1}$'s in LJR to compute partial results for aggregate subexpressions of Φ and Λ, and then use $f^{\circ}_{\Phi,1}$'s and $f^{\circ}_{\Lambda,1}$'s in the last step to combine the partial aggregate results and evaluate Φ and Λ on the complete LR-groups.

Finally, we note that while we described the details above on handling algebraic aggregates in the context of static query rewriting, they also apply to NLJP-based memoization. Note also that we can NLJP-based memoization to handle $\mathbb{G}_R \neq \varnothing$ too, simply by caching results by $\mathbb{J}_L \cup \mathbb{G}_R$ instead of only by $\mathbb{J}_L$ (as is done in LJR for static query rewriting).

D Generalization to Multiway Joins

We describe an optimization procedure that finds opportunities to apply generalized a-priori, memoization, and pruning techniques to an iceberg query involving multiway joins. Given an iceberg query Q involving a two-way join of L and R as shown in Listing 5, let gapriori(Q, L, R) return Q_L, the subquery in L' (Section 4.1) used to safely reduce L if the generalized a-priori technique applies, or error otherwise. Let memprune(Q, L, R) return

```
# consider generalized a-priori
𝒪 ← ∅ # rewrites found by gapriori
𝒯 ← 𝔗(Q) # relations to be considered by gapriori
while 𝒯 is not empty:
    try:
        ⟨T_L, Q_L⟩ ← pick_gapriori(Q, 𝒯)
        T̆_L ← subset of T_L with at least one attribute output by Q_L
        𝒪.append(⟨T̆_L, Q_L⟩)
        𝒯 ← 𝒯 \ T_L
    except: # no more opportunities for gapriori
        break
# apply memoization and pruning
try:
    ⟨T_L, Q_B, Q_R, Q_C, Q_P⟩ ← pick_memprune(Q) subject to
        ∀⟨T, _⟩ ∈ 𝒪 : T_L ⊇ T ∨ T_L ∩ T = ∅
    Q ← NLJP(Q_B, Q_R, Q_C, Q_P)
except:
    pass
# apply generalized a-priori rewrites
for ⟨T_L, Q_T⟩ in 𝒪:
    rewrite Q to replace occurrence of Q^⋈[T_L] with Q^⋈[T_L] ⋉ Q_L
return Q
```

Listing 9: Optimization procedure for iceberg query Q with a multiway join.

$\langle Q_B, Q_R, Q_C, Q_P \rangle$ return a specification of the NLJP operator if memoization and/or pruning techniques apply (Sections 5 and 6), or error otherwise. We now give a procedure that uses gapriori and memprune to optimize an iceberg query with a multiway join, such as the one in Listing 3.

Let $\mathfrak{T}(Q)$ denote the set of relation *instances* joined by Q (note that there may be multiple instances for the same relation in the case of self-joins). For brevity, we refer to relation instances simply as relations below. Suppose $T \subseteq \mathfrak{T}(Q)$ is a subset of the relations involved in Q. Let $Q^{\bowtie}[T]$ denote the query over T formed by pushing selection/join conditions and projections in Q down as much as possible.

On a high level, our strategy is simple (see Listing 9). We first search for opportunities for applying the generalized a-priori technique, by iteratively calling a subroutine pick_gapriori$(Q, \mathcal{T})$ (details omitted in Listing 9), which finds a way (if any) to safely reduce some relation(s) in a given set $\mathcal{T}$ (which starts out to all relations in Q). To consider a particular non-empty subset $T_L \subset \mathcal{T}$, pick_gapriori treats Q as an iceberg query over two relations $L = Q^{\bowtie}[T_L]$ and $R = Q^{\bowtie}[\mathfrak{T}(Q) \setminus T_L]$, and invokes gapriori. (Note that the reducer query returned by gapriori may in fact be applicable to just a subset of T_L.) After finding each a-priori optimization, we call pick_gapriori again to continue looking for opportunities in the remaining relations; the process continues until there are no more opportunities or relations to consider. At this point, we have collected a list of generalized a-priori optimizations of the form $\langle T_L, Q_L \rangle$, each corresponding to a rewrite that replaces $Q^{\bowtie}[T_L]$ with $Q^{\bowtie}[T_L] \ltimes Q_L$.

Next, we look for an opportunity for applying memoization and pruning by calling a subroutine pick_memprune over Q. Again, to consider each non-empty subset T_L of the relations, we treat Q as an iceberg query over two relations $L = Q^{\bowtie}[T_L]$ and $R = Q^{\bowtie}[\mathfrak{T}(Q) \setminus T_L]$, and invoke memprune. However, if the generalized a-priori rewriting step has determined earlier that two relations need to be joined for a reducer to apply, we would not break them apart; *i.e.*, both of them are in T_L, or neither is. If pick_memprune is successful, we rewrite Q using an NLJP operator. Note that, other than the case when generalized a-priori rewrites require an incompatible grouping of relations (which we have taken care of),

these rewrites do not affect the applicability of memoization and pruning.

The optimization procedure has a worst-case complexity that is exponential in the number of relations in Q, as each invocation of pick_$*$ needs to potentially consider a large number of possible subsets. In practice, we prioritize the search so it can quickly find an available optimization; *e.g.*, pick_memprune will first consider the minimal T_L that include all of $\mathbb{G}_L$ in Q. On the other hand, our current procedure is a best-effort algorithm that is content of finding some set of optimization opportunities; add cost-based considerations would necessitate more sophisticated search strategies, which we leave as further work.

Example 13. *We end this section by walking through the steps of optimizing the complex query involving a four-way self-join in Listing 3. We start by finding generalized a-priori optimization opportunities. Singleton candidates for T_L (*e.g.*, $\{$S1$\}$) yields no opportunity, because of the inequality join condition with the rest of the relation instances (*e.g.*, T1.val>=S1.val). However, $T_L = \{$S1, T1$\}$ reveals an opportunity with the following reducer:*

```
SELECT S1.id, S1.attr FROM Product S1, Product T1 -- Q_S1
WHERE S1.category = T1.category
AND T1.attr = S1.attr AND T1.val > S1.val
GROUP BY S1.id, S1.attr
HAVING COUNT(*) >= 10;
```

To see why this optimization applies, note that the original query Q can be seen as an iceberg query with $L = Q^{\bowtie}$[S1, T1], $R = Q^{\bowtie}$[S2, T2], $\mathbb{G}_L = \{$S1.id, S1.attr$\}$, $\mathbb{G}_R = \{$S2.attr$\}$, $\mathbb{J}_L = \{$S1.id, T1.id$\}$, and $\mathbb{J}_R = \{$S2.id, T2.id$\}$. The HAVING *condition is obviously monotone. Finally, $\mathbb{G}_R \cup \mathbb{J}_R$ is a superkey of R. This last observation is trickier: the definition of R contains the condition* T2.attr=S2.attr*; since* S2.attr *is in $\mathbb{G}_R$, the closure of $\mathbb{G}_R \cup \mathbb{J}_R$ must also contain* T2.attr*, which means that $\mathbb{G}_R \cup \mathbb{J}_R$ contains all key attributes of* S2 *and* T2*, and therefore must be a superkey of R. Hence, Theorem 2 applies.*

Note that the reducer query Q_{S1} can be applied to S1 *alone (even though it was found by considering $\{$S1, T1$\}$), so we record the rewrite $\langle\{$S1$\}, Q_{S1}\rangle$ and move on to considering the remaining relation instances $\{$S2, T2$\}$. By a process similar to the above, $T_L = \{$S2, T2$\}$ reveals a second reducer Q_{S2} that applies to* S2*, which is identical to Q_{S1} with* S1 *replaced by* S2 *and* T1 *replaced by* T2*. This process additionally requires inferring that occurrences of* S1.id *in the original query's* SELECT *and* GROUP BY *can be replaced by* S2.id *as they are equated, and that* S2.category=T2.category *follows from the dependency* id $\rightarrow$ category *and query conditions* S1.category=T1.category*,* S1.id=S2.id*, and* T1.id=T2.id*. Note that if we are unable to infer the above two properties, a less effective (but still correct) reducer will be derived.*

After identifying the generalized a-priori opportunities, we proceed to consider memoization and pruning optimization. As discussed earlier, we start by examining the minimal subset of relation instances that include the GROUP-BY *attributes—$T_L = \{$S1, S2$\}$. Interestingly, note that this pair of relation instances differs from the two pairs used by generalized a-priori earlier. Luckily, this T_L is still compatible with the grouping of relation instances required by generalized a-priori, because both reducers apply to single relation instances. With $T_L = \{$S1, S2$\}$, we have $\mathbb{G}_L = \{$S1.id, S1.attr, S2.attr$\}$ and $\mathbb{J}_L = \{$S1.category, S1.attr, S2.attr, S1.val, S2.val$\}$. It is clear that $\mathbb{G}_L \rightarrow \mathbb{A}_L$ because of the join condition* S1.id=S2.id*. Thus, Theorem 3 applies. Further applying the technique for deriving pruning technique in Section 5.2, we have the NLJP-based plan in Listing 10.*

```
SELECT S1.id, S1.category AS c, -- Q_B, binding: (c,a1,a2,v1,v2)
       S1.attr AS a1, S2.attr AS a2,
       S1.val AS v1, S2.val AS v2
  FROM Product S1, Product S2
  WHERE S1.id = S2.id;
SELECT COUNT(*) AS payload -- Q_R(b)
  FROM Product T1, Product T2
  WHERE b.c = T1.category AND T1.id = T2.id
  AND T1.attr = b.a1 AND T2.attr = b.a2
  AND T1.val > b.v1 AND T2.val > b.v2;
SELECT * -- Q_C(b')
  FROM C(c, a1, a2, v1, v2, payload) -- cache
  WHERE b'.c = c AND b'.a1 = a1 AND b'.a2 = a2
  AND b'.v1 >= v1 AND b'.v2 >= v2
  AND payload < 10;
SELECT id, a1, a2, payload -- Q_P
  FROM T(id, c, a1, a2, v1, v2, payload) -- concatenated tuples
  WHERE payload >= 10;
```

Listing 10: NLJP-based plan for the complex query in Listing 3.

```
WITH U(id, attr) AS (Q_S1)
SELECT S1.id, S1.category AS c, -- Q_B
       S1.attr AS a1, S2.attr AS a2, S1.val AS v1, S2.val AS v2
FROM
 (SELECT * FROM Product WHERE (id,attr) IN (SELECT * FROM U)) S1,
 (SELECT * FROM Product WHERE (id,attr) IN (SELECT * FROM U)) S2
WHERE S1.id = S2.id;
```

Listing 11: Modified Q_B for the NLJP-based plan in Listing 10.

Finally, going back to the two generalized a-priori opportunities identified earlier, we can modify Q_B by applying the same reducer. Listing 11 shows the reduced Q_B.

E Query Plans for Skyband

For the following *skyband* query Q_1:

```
SELECT COUNT(1)
FROM player_performance L, player_performance R
WHERE L.b_h >= R.b_h AND L.b_hr >= R.b_hr
AND (L.b_h > R.b_h OR L.b_hr > R.b_hr)
GROUP BY R.playerid, R.year, R.round
HAVING COUNT(1) < 50;
```

PostgreSQL uses the following query plan:

```
Finalize GroupAggregate
  Group Key: r.playerid, r.year, r.round
  Filter: (COUNT(1) < 50)
  ->  Sort
    Sort Key: r.playerid, r.year, r.round
    ->  Gather
          Workers Planned: 2
          ->  Partial HashAggregate
            Group Key: r.playerid, r.year, r.round
            ->  Nested Loop
              ->  Parallel Seq Scan on players
                 ->  Index Scan using b_h,b_hr on  players
                    Index Cond: ((l.b_h >= b_h)
                                AND (l.b_hr >= b_hr))
                    Filter: ((l.b_h > b_h) OR (l.b_hr > b_hr))
```

Vendor A uses the following:

```
SELECT
  Parallelism (Gather Streams)
    Filter
      Compute Scalar
       Hash Match (Aggregate)
         Parallelism (Repartition Streams)
          Stream Aggregate (Aggregate)
            Nested Loops (Inner Join)
                Index Scan (NonClustered) [players.ibx]
                Index Scan (NonClustered) [players.bhr_bh]
```