Robust Join Processing with Diamond Hardened Joins

Altan Birler
Technische Universität München
altan.birler@tum.de

Alfons Kemper
Technische Universität München
kemper@in.tum.de

Thomas Neumann
Technische Universität München
neumann@in.tum.de

ABSTRACT

Join ordering and join processing has a huge impact on query execution and can easily affect the query response time by orders of magnitude. In particular, when joins are potentially growing n:m joins, execution can be very expensive. This can be seen by examining the sizes of intermediate results: If a join query produces many redundant tuples that are later eliminated, the query is likely expensive, which is not justified by the query result. This gives the query a diamond shape, with intermediate results larger than the inputs and the output. This occurs frequently in various workloads, particularly, in graph workloads, and also in benchmarks like JOB.

We call this issue the diamond problem, and to address it, we propose the diamond hardened join framework, which splits join operators into two suboperators: Lookup & Expand. By allowing these suboperators to be freely reordered by the query optimizer, we improve the runtime of queries that exhibit the diamond problem without sacrificing performance for the rest of the queries. Past theoretical work such as worst-case optimal joins similarly try to avoid huge intermediate results. However, these approaches have significant overheads that impact all queries. We demonstrate that our approach leads to excellent performance both in queries that exhibit the diamond problem and in regular queries that can be handled by traditional binary joins. This allows for a unified approach, offering excellent performance across the board. Compared to traditional joins, queries' performance is improved by up to 500x in the CE benchmark and remains excellent in TPC-H and JOB.

PVLDB Reference Format:
Altan Birler, Alfons Kemper, and Thomas Neumann. Robust Join Processing with Diamond Hardened Joins. PVLDB, 17(11): 3215-3228, 2024.
doi:10.14778/3681954.3681995

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/umbra-db/diamond-vldb2024.

1 INTRODUCTION

The join operation is the cornerstone of relational data processing. Its efficiency is crucial for the performance of analytical workloads. In particular, when queries contain joins that are potentially growing n:m joins, execution can be very expensive. However, users seldom write queries that produce large outputs; most tuples that are produced as intermediate results in such queries are either filtered out or aggregated before the end of the query. These queries' runtimes are therefore not justified by their results: The slowness of query execution is often not intrinsic to queries themselves, but rather a result of inefficiencies in how joins are performed and optimized. We visualize this problem in Figure 1, where the intermediate results of a query have a diamond shape, with small inputs coming in, growing into large intermediate results, and then shrinking back to the actual output. Such queries occur in real life, in relational benchmarks like JOB [26] and graph benchmarks such as LDBC SNB BI [42], Graphalytics [23], and CE [9].

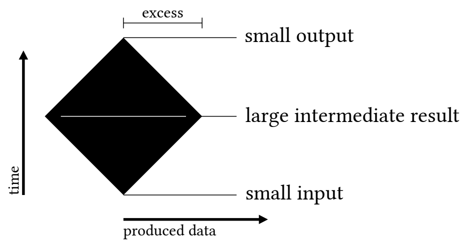

Figure 1: The diamond problem: Queries with small inputs and outputs may produce large intermediate results.

Let us give an example from healthcare informatics of a query exhibiting the diamond problem. We want to find patients who have been administered two drugs which have a known adverse interaction. To compute the query, we might first find all pairs of drugs that have been administered to the same patient, then filter the pairs that have a known interaction. Or, we might find drugs that have been administered to one patient, find all other drugs that have known interactions, and then check if the patient has also been administered the interacting drugs. Regardless of our approach, the intermediate results may be quite large, potentially quadratic in size, even though the final result is likely to be small.

Avoiding the diamond problem is intrinsically linked to tackling robustness; if we can limit the sizes of intermediate results, we can keep the query's runtime bounded and predictable. There has been much recent interest on tackling robustness by optimizing for the worst-case. However, these approaches have significant overheads that often makes them too expensive to replace traditional binary join processing. In this work, we propose diamond hardened joins. Through the decomposition of joins into two suboperators Lookup & Expand, one shrinking and one growing, and reordering these suboperators independently, we improve the runtime of queries that exhibit the diamond problem by eliminating tuples as early as possible. This approach provides great worst-case performance without sacrificing the average-case: Our framework not only provides best in class performance for classical relational workloads, it is able to handle graph workloads and cyclic queries that have historically been difficult for relational databases to optimize and process. Compared to traditional joins, the Lookup & Expand framework improves query performance by up to 500x in the CE benchmark, while still delivering excellent performance in TPC-H and JOB.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3681995

Our contributions are as follows: (1) We introduce Lookup & Expand decomposition, and show that it can be used to process α-acyclic queries optimally. (2) We prove that simple ternary joins are enough to achieve worst-case optimality for a large class of cyclic queries. We introduce a ternary Expand operator, which allows the diamond hardened join framework to tackle cyclic predicates. We empirically demonstrate that our approach is more efficient compared to generic n-ary worst-case optimal joins. (3) We discuss a low risk approach to eager aggregation, and demonstrate its effectiveness on graph pattern matching queries with `count(*)` aggregates. (4) We show how Lookup & Expand decomposition, with careful handling of `NULL` values, enables many more reorderings of outer-join plans compared to binary joins. (5) We propose a strategy for integrating Lookup & Expand plans into cost-based query optimizers. Our strategy combines search with greedy heuristics to balance cost optimization with robustness.

We present our work in five sections. In Section 2, we discuss related research in addressing robustness and tackling the diamond problem. In Section 3, we introduce Lookup & Expand (L&E) decomposition, formalize its semantics, and demonstrate how it can be used to tackle the diamond problem. We discuss how we are able to tackle acyclic predicates, cyclic predicates, aggregations, and outer-joins. In Section 4, we describe the integration of L&E into cost-based query optimizers, while avoiding regressions and maximizing the utility of L&E decomposition. Finally, in Section 5, we present the results of our experiments, showing that L&E decomposition provides excellent performance on both relational and graph queries compared to binary and worst-case optimal joins. Throughout this paper, we focus on hash joins in the in-memory setting, leaving the exploration of other settings to future work.

2 BACKGROUND AND RELATED WORK

In this section, we discuss when the diamond problem can occur, and how existing approaches try to alleviate it. The diamond problem occurs when many tuples are generated as intermediate results, only to be eliminated later in joins or aggregated together in a group-by. Tuples that are generated and then eliminated are called dangling tuples. How we can prevent dangling tuples depends on the query's predicate structure, specifically, its acyclicity or cyclicity and the presence of reordering restrictions due to outer-joins. In total, we have identified four cases of the diamond problem which we will analyze and address.

(1) Acyclic predicates: When nonredundant predicates between relations are acyclic, the diamond problem can be avoided by aggressively pushing down additional semi-join filters [4, 41, 52].

(2) Cyclic predicates: When nonredundant predicates between relations form cycles, any binary join plan, even with additional semi-join filters, is suboptimal. Worst-case optimal joins (WCOJs) replace many binary joins with a single black-box multiway join that is able to exploit per tuple runtime adaptivity to bound intermediate result sizes to the worst-case output size. As WCOJs have unique cost characteristics that are hard to estimate, integrating them into query plans remains a challenge.

(3) Queries with duplicate values: Queries may produce many duplicates that result in redundant work. Duplicates can be eliminated early with eager aggregation or delayed with factorization.

(4) Reordering restrictions: Reordering operators is essential to prevent large intermediate results, but the restrictions imposed by outer-joins can make it difficult. Compensation based reordering can allow for full reordering even with outer-joins, but at the cost of converting inner-joins to outer-joins and expensive compensation.

In the following, we further detail the issues with each case, and present how related work addresses the problems. We further discuss their limitations to motivate the alternative unified approach we propose in Section 3.

2.1 Acyclic Queries & Full Semi-Join Reduction

When nonredundant predicates between relations do not form cycles (i.e. the query is α-acyclic [14]), the diamond problem can be avoided by aggressively prefiltering relations with the domains of other relations. This filtering procedure is called full semi-join reduction and was described by Bernstein and Chiu [4]. After the prefiltering, Yannakakis describes a join order that guarantees no dangling tuples [50], leading to plans with optimal runtime up to a constant factor.

Acyclic queries are very common. All queries in JOB and almost all queries in TPC-H are α-acyclic, implying that achieving a high degree of robustness for a large set of relational queries is feasible. The following is an example for an acyclic and a cyclic query:

```sql
-- Acyclic query:
select * from R, S, T
where R.a = S.a and S.a = T.a and R.a = T.a;
-- Join tree:
-- (R join S on (R.a = S.a)) join T on (R.a = T.a)
-- One predicate per join: acyclic.

-- Cyclic query:
select * from R, S, T
where R.b = S.b and S.c = T.c and T.a = R.a;
-- Join tree:
-- (R join S on (R.b = S.b)) join T
--  on (R.a = T.a and S.c = T.c)
-- The top join has two nonredundant predicates: cyclic
```

In full semi-join reduction, the base relations are filtered by the contents of other relations in a specific order, which is given by the GYO ear removal algorithm [2, 51]. After two waves of filters, forwards and backwards, it is guaranteed that there is a join order that produces no dangling tuples, tuples that are produced and later eliminated by other joins. The second waves of filters can also be merged with the final joins, resulting in a two-phase plan, which is utilized in constant delay enumeration [22, 44]. While this filtering procedure is optimal up to a constant factor, the constant factor due to executing additional filters is significant. When running the CE benchmark [9] with full semi-join reduction according to the implementation of Gottlob et al. [20], the benchmark takes around 5x as long as running the queries without semi-join reduction. Naively executing additional semi-joins between huge tables is often significantly more costly than the optimal join order, as the optimal join order can lead to quickly shrinking intermediate results. This implies that a cost-based optimization strategy is needed to truly achieve optimal performance. In Section 3.4, we will present how the diamond hardened framework can achieve optimal performance for acyclic queries without the need for additional semi-joins.

2.2 Cyclic Queries & Worst-Case Optimality

When predicates were acyclic, we could avoid dangling tuples by applying semi-join reduction. Unfortunately, when predicates form cycles, we cannot guarantee that such plans exist. In fact, binary joins (even with additional semi-join filters and/or L&E decomposition) are suboptimal for cyclic queries, as they can produce intermediate results that are asymptotically larger in size compared to both the inputs and output [35].

Worst-case optimal joins (WCOJs) [18, 25, 34, 35, 40, 43] guarantee worst-case optimality, i.e. their runtime complexity is upper-bounded by the how large the output could possibly be given the worst possible input. While the worst-case result size can be significantly higher than the result size of most queries, WCOJs can empirically demonstrate better performance compared to binary hash joins with skewed data. However, WCOJs process joins an attribute at a time, while classical operators process joins a relation at a time. This makes the integration of WCOJs into existing optimizers difficult [18]. They are also significantly more inefficient compared to binary joins in queries with low skew [19]. When we run the TPC-H SF 10 benchmark with our research system Umbra [33], we find that WCOJs slow down overall runtime 12 times compared to binary joins. In Section 3, we will show that simple relation based ternary joins can be sufficient and more efficient. We will integrate ternary joins into the diamond hardened framework, unifying the solutions for acyclic and cyclic queries.

In Section 1, we gave an example of a cyclic query involving drugs and drug interactions, where we can demonstrate the inefficiency of binary joins. Consider the two relations R(patient, drug) describing which drugs a patient is taking and S(drug1, drug2) describing which pairs of drugs have known adverse interactions. We start with a tuple in R, with a patient and a drug. We have two options. (1) We join back with R to find all the other drugs the patient is taking, then join with S to find which of those pairs are actually interacting. (2) We join with S to find all the drugs that interact with the drug the patient is taking, then join with R to find if the patient is also taking any of those interacting drugs. Either way, the intermediate result size can be quadratic, as a patient may be taking many drugs, or a drug may be interacting with many other drugs. However, we can avoid quadratic time by adaptively choosing the join order per tuple based on the number of join partners. Starting with a tuple (p, d) in R, we can check how many drugs the patient p takes, and how many drugs d interacts with. If the p takes many drugs, we join with S first, and if the d interacts with many other drugs, we join with R first. This is the fundamental advantage that WCOJs bring. Rather than having one fixed plan, they allow for per-tuple runtime adaptivity, choosing the execution strategy based on the number of join partners a particular tuple has.

WCOJs are formally defined as attribute based joins, rather than relation based. When computing a query with result attributes (a, b, c), they first compute all the (a)s, then (a, b)s, and then (a, b, c)s, extending the result one attribute at a time rather than one relation at a time until they compute the full query result. The particular way in which WCOJs extend their result sets gives them their runtime adaptivity. Assume we have three relations R(a, b), S(b, c), and T(c, a) which we want to join naturally (on attributes with the same names). Also assume that we have available a superset of the result projected onto attributes a and b: $I(a, b) \supseteq \Pi_{a,b}(R \bowtie S \bowtie T)$. We want to efficiently extend this result to contain the attribute c. For every tuple (a, b), we query the matching c values in S and the matching c values in T. The intersection of these two sets of c gives us all possible c values corresponding to each tuple (a, b). There are many strategies for computing intersections efficiently. The key requirement is that, for worst-case optimality, the operation must take time on the order of the size of the smaller set [35]. One solution is to have hash tables on S and T with keys b and a and values as hash sets of c. We query the hash tables, iterate over the smaller hash set and query the larger hash set, giving us constant time per element in the smaller set.

```
def intersect(htR : Dict[a,c], htS : Dict[b,c], a, b):
  cR = htR[a] # Join order R, S
  cS = htS[b] # Join order S, R
  if cR.length < cS.length:
    # Join order R, S
    for c in cR:
      if cS.contains(c):
        produce(a, b, c)
  else:
    # Join order S, R
    for c in cS:
      if cR.contains(c):
        produce(a, b, c)
```

The logic in both branches is the same as hash join lookups, the difference being that we can switch the join order based on the number of elements contained in the corresponding sets. Such runtime adaptivity does not exist in classical relational processing.

2.2.1 Theoretical Power of Worst-Case Optimal Joins. WCOJs not only have empirical benefits when joining skewed data, they also have better worst-case runtime complexity than binary hash joins. Consider the extremely skewed symmetric relations $R(a, b) = S(b, c) = T(c, a) = ((1) \times [1, n]) \cup ([1, n] \times (1))$. R is a set of unidirectional edges between n nodes, the node 1 is connected to all n nodes while all the other nodes are only connected to themselves. The natural join query $Q(a, b, c) = R(a, b) \bowtie S(b, c) \bowtie T(c, a)$ has result size $O(N)$. However, binary hash joins take $O(N^2)$ time, regardless of the join order, as the size of any binary join is $O(N^2)$. WCOJs, in contrast, guarantee $O(N^{1.5})$ time for a query of this structure regardless of the content of the relations. For this particular instance of relations, many WCOJ algorithms can even compute the result in $O(N)$ time [35].

For the cyclic join of three relations, WCOJs seem to be a clear win. Nonetheless, WCOJs are not instance-optimal, their runtime complexity is only guaranteed to be on the order of the worst-case result size. Their advantages against binary hash joins weaken as the number of relations and attributes increase and the difference between actual output size and worst-case output size grows.

2.2.2 Upper Bounds for Join Result Size. A tight upper bound for (set-semantics) join result sizes was discovered by Atserias, Grohe, and Marx [1]. We will henceforth refer to this bound as the AGM bound. Since we are only dealing with equality predicates, it is more practical to refer to equivalence classes of attributes instead of individual attributes from individual relations. For example, the join query $R \bowtie_{R.a=S.a} S \bowtie_{S.a=T.a \land S.b=T.b} T$ is a join query with three relations and two attributes (attribute equivalence classes) a and

b. Given a set of attributes $\mathcal{A} = \{a_1, a_2, ..., a_m\}$, a vector of domain sizes corresponding to each attribute $v = (v_1, v_2, ..., v_m)$, relations $\mathcal{R} = \{R_1, R_2, ..., R_n\}$, their cardinalities $c = (|R_1|, |R_2|, ..., |R_n|)$, and a *fractional edge cover* weight to each relation $w = (w_1, w_2, ..., w_n)$, the AGM bound is given by the following linear program [1, 35]:

$$\begin{aligned} \text{AGM} = \underset{w}{\text{minimize}} \quad & \prod_{R_i \in \mathcal{R}} |R_i|^{w_i} \\ \text{subject to} \quad & 1 \leq \sum_{i: a_j \in \mathcal{A}(R_i)} w_i \quad \forall a_j \in \mathcal{A} \end{aligned}$$

Worst-case optimal joins guarantee, for a given query, a runtime complexity on the order of the AGM bound. Binary hash-joins, in contrast, can have runtime complexity on the order of the cross-product of the input relations, potentially much higher than the AGM bound. However, WCOJs tend to have higher constant overhead as they materialize and build complex index structures on all inputs, to enable efficient intersection operations. Nevertheless, it is hard to guarantee that the diamond problem is not present, as estimates can contain large errors. Thus, pessimistic optimizers tend to prefer WCOJs over binary hash joins.

The Free Join algorithm [47] comes closest to bridging the gap, as it can represent plans that combine worst-case optimality and binary hash-joins. Nonetheless, the algorithm is single threaded, can still involve complex index structures, and the described optimization strategy often cannot find optimal plans. Our Expand3 operator, which we will present in Section 3.5.3, is simpler, is worst-case optimal for a large class of queries, and supports bushy plans. On the first five queries of the LSQB benchmark [28] SF {0.1, 0.3, 1, 3} supported by Free Join, our approach outperforms it by over a factor of 3x even when restricted to a single thread.

The worst-case query instance can be constructed by utilizing the dual linear program of the AGM bound [1]. The dual computes domain sizes for attributes, and every relation is assigned the cross-product of the domains of its attributes. The worst-case instance does not suffer from the diamond problem and produces no dangling tuples; binary joins are worst-case optimal for the worst-case query instance [35]. Thus, counterintuitively, *worst-case optimal joins should not be used with worst-case inputs.* There are many such surprising queries where WCOJs are not beneficial compared to binary joins, which we will discuss in Section 3.5.1. Thus, similar to acyclic queries, a simpler cost-based optimization strategy is needed to truly achieve optimal performance in cyclic queries.

2.3 Queries with Duplicates & Factorization

Many queries produce intermediate results with lots of duplicate attribute values. This is inherent in how joins work; they build cross-products out of matching pairs of tuples. When we process duplicate values, we essentially do repeated work that could have been avoided. However, duplicate removal itself can be a very expensive operation. Thus, a balance needs to be struck between avoiding duplicates and not taking on too much additional cost.

There are multiple approaches to dealing with duplicate values. Factorization [36] avoids duplicate values caused by implicit cross-products. The factorized representation is denormalized; an attribute may contain a multiset of values. Such a denormalized tuple represents all tuples resulting from the cross-products of the attribute multisets. For example, we can represent the tuples $(1, 3)$, $(1, 4)$, $(2, 3)$, $(2, 4)$ with a single factorized tuple $(\{1, 2\}, \{3, 4\})$. For the result, the factorized tuples are flattened (normalized). If the query contains a group-by, the aggregation can be performed directly on the factorized representation, avoiding the flattening step.

Duplicate values are often produced as a result of n:m joins. A join hash table build factorizes its build input on the join key. However, most query execution engines do not benefit from this factorization, as they directly materialize all join partners, flattening the factorization. In Section 3, we will present how the diamond hardened framework can utilize the factorization already present in hash joins to avoid duplicate values in intermediate result.

2.3.1 *Eager Aggreation.*

Join and aggregate queries that produce intermediate results with duplicate values are amenable to eager aggregation. Eager aggregation refers to computing partial aggregates before some joins are executed to reduce the cost of all proceeding joins. For example, the query $\Gamma_{b;\text{sum}(c)}\ (R(a, b) \bowtie S(b, c))$ might be eagerly aggregated as $\Gamma_{b;\text{sum}(c')}\left(R \bowtie \Gamma_{b;\text{sum}(c):c'}(S)\right)$.

Eager aggregation can be done by applying transformative rules that push the top group-by down a join tree [48]. Alternatively, one can find the intermediate results that contain a lot of duplicates and place group-by operators at exactly those positions with corresponding keys and aggregates as proposed by Fent et al. [16].

The group-by operator is a costly operator, thus relying on cardinality estimation to decide where to place additional group-bys leads to issues. In contrast, a group-join operator merges a join and a group-by in one operator, avoiding high costs. Inspired by the group-join [16], a robust and low-risk approach to eager aggregation is to extend the hash table build of Joins to support efficient non-strict eager aggregation[1]. This avoids extra materialization costs and in turn reduces the risk of introducing an expensive group-by operator with questionable benefit.

2.3.2 *Factorization vs. Semi-Join Reduction and Eager Aggregation.*

Factorization and full semi-join reduction address the diamond problem in similar ways even though the techniques themselves are quite different. Semi-join reduction eliminates dangling tuples eagerly by executing filters. Factorization (and Lookup & Expand plans described in Section 3) delays the expansion (or flattening, in factorization terminology) of dangling tuples, allowing them to be eliminated in the plan before they lead to large intermediate results. And if the query contains a group-by on top, the join result, in many cases, does not need to be expanded, thus avoiding large intermediate results. Eager aggregation, in contrast, eagerly pushes down the aggregation, eliminating duplicate values as soon as possible. Factorization represents a lazy alternative to the eager optimizations of semi-join reduction and eager aggregation. In Section 3, we will present how the diamond hardened framework represents a practical middle ground between laziness and eagerness.

[1]Non-strict eager aggregation refers to the fact that duplicates may still remain after eager aggregation without sacrificing correctness, as the final group-by at the top of the join tree is guaranteed eliminate all duplicates in the end. Consider the query 'select k, count(*) from R group by k;' where relation R contains the values 1, 2, 2, 2, 2. A strict aggregation would produce the value-count pairs $(1, 1)$ and $(2, 4)$, while a non-strict eager aggregation would be allowed to produce $(1, 1)$, $(2, 3)$, $(2, 1)$ where the value 2 is still duplicated, but the total sum of the counts for 2 still adds up to 4. Non-strictness allows for simple and efficient eager aggregation implementations.

2.4 Reordering Restrictions & Compensation

A query optimizer for SQL needs to deal with outer-, and anti-joins as well as inner-joins. While inner-joins are both commutative and associative, this does not hold for other join types. Thus, a query optimizer needs to consider reordering restrictions when generating plans. Outer-joins are hard to reorder as they can produce null values. Outer-joins can also result in the diamond problem, as they may be growing as well as shrinking. Compensation based reordering [46] allows for full reordering, but at the cost of converting inner-joins to outer-joins and expensive compensation oeprators.

3 APPROACH

In this section, we discuss the intuition behind Lookup & Expand (L&E) decomposition and how this technique leads to more robust query plans. We start by discussing the difficulty of ordering joins and then describe how L&E decomposition addresses this difficulty.

3.1 Making Join Ordering Robust

In a first approximation, query optimizers minimize total data movement, which can be approximated by the sum of intermediate result sizes of operator (a.k.a. the C_{out} cost function [11]). If a highly selective filter is executed at the end of a query (top of the query plan), the optimizer tries to push the filter down as far as possible, so that fewer tuples are propagated all the way up the plan. In essence, the optimizer pushes down shrinking operators that produce a smaller output than their input and pushes up growing operators that produce a larger output than their input. This gives us the following rule of thumb: *An optimizer tries to push down cheap shrinking operators and push up growing operators.*

Binary hash joins build a hash table on one input and probe the hash table with the other input, producing all matching pairs[2]. The hash table build is expensive. The probe is relatively cheap. However, the probe may be growing or shrinking; each tuple may find zero, one, or many matches in the hash table. Thus, it is hard to know when to push down a join; pushing a join down might speed up the plan significantly or make it slower. The following code demonstrates a typical binary hash join in pseudocode:

```
# Hash table build
ht = {}
for k, v in buildInput:
    ht[k] = v

# Hash table probe
for k, v in probeInput:
    iterator = ht.find(k) # Lookup
    if not iterator.done():
        do: # Expand
            produce(k, *iterator)
            iterator.step()
        while not iterator.done()
```

We propose to split binary hash join probes into two suboperators, Lookup and Expand. Lookup finds the first match in the hash table. Expand iterates over the rest of the matches. Now, we have two operators that are easy to categorize. Lookups are cheap shrinking operators. Expands are growing operators. Lookups should be pushed down while Expands should be pushed up.

[2]There are, of course, many variants to the binary hash join. We focus on the build & probe variant, commonly preferred in modern relational systems for its performance.

3.2 Lookup & Expand Decomposition

In this section, we start with descriptions and pseudocode for the L&E operators. Then, we will formalize the semantics of a L&E plan by introducing a new notation.

In a join query, if a tuple is ever going to be filtered out, we want it to be filtered out as soon as possible. Thus, we propose to decompose joins in such a way that the operation of "looking for the first match if it exists" is separated into a distinct operator Lookup, which can be independently reordered. The remaining operation "iterating over all matches (given a reference to the first match)" is designated as the Expand operator. Lookup and Expand combined together make up a single binary join.

We present the operators in produce/consume [31] pseudocode:

```
def lookup_consume(produce, tuple, hashTable):
  # Look for the key in the hash table
  iterator = hashTable.find(tuple.key)
  if iterator.done():
    return
  # Return tuple and iterator over matches
  produce(tuple, iterator)

def expand_consume(produce, tuple):
  # Extract the iterator from the tuple
  currentMatch = tuple.iterator
  do: # Loop over all matches
    produce(tuple, *currentMatch)
    currentMatch.step()
  while not currentMatch.done()
```

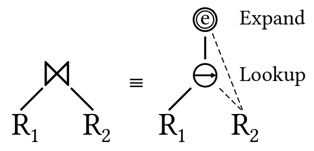

Figure 2: Binary hash join and equivalent Lookup & Expand.

The Lookup operator checks for a match and produces a single iterator for all matches. The Expand operator uses the iterator from the input to iterate over all matches. When using an in-memory hash-table, the iterator is analogous to a memory span pointing to matching hash table entries. In a pipelined query engine [32], this attribute is often not copied to main memory; it is simply passed from Lookup to Expand in a register or cache. The machine code executed in the end will be exactly equivalent to that of a binary hash join. In Figure 2 we show a binary join and the equivalent L&E decomposition. The left side of the join is designated as the probe side.

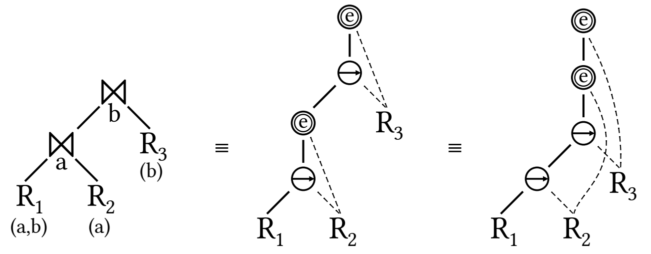

Figure 3: Threeway join, equivalent Lookup & Expand, and Lookup pushdown.

Since we now have two distinct operators, the query optimizer can reorder them independently, leading to more interesting plans. Consider the leftmost join plan in Figure 3 where we build hash tables on $R_2(a)$ and $R_3(b)$ and probe these hash tables with $R_1(a, b)$ on $R_1.a = R_2.a$ and $R_1.b = R_3.b$. L&E decomposition results in the middle plan. After the decomposition, we can freely push down the Lookups through the Expands, leading to the rightmost plan. This change can be illustrated in the following pseudocode:

```
# Join pipeline before push-down
for t in R1:
  iterator2 = ht2.find(t.a)
  if not iterator2.done():
    do:
      iterator3 = ht3.find(t.b)
      if not iterator3.done():
        do:
          produce(t, *iterator2, *iterator3)
          iterator3.step()
        while not iterator3.done()
      iterator2.step()
    while not iterator2.done()

# Join pipeline after push-down
for t in R1:
  iterator2 = ht2.find(t.a)
  if not iterator2.done():
    iterator3 = ht3.find(t.b)
    if not iterator3.done():
      do:
        do:
          produce(t, *iterator2, *iterator3)
          iterator3.step()
        while not iterator3.done()
        iterator2.step()
      while not iterator2.done()
```

After the Lookups are pushed down, it is guaranteed that the diamond problem is prevented. The execution starts with shrinking operators (Lookups) and ends with growing operators (Expands).

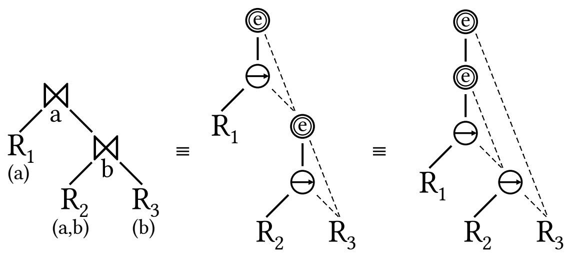

Figure 4: Threeway join, equivalent Lookup & Expand, and Expand pullup.

We want to show a second example that demonstrates the full power of L&E decomposition. Consider the leftmost join plan in Figure 4. We first join $R_2(a, b)$ and $R_3(b)$ on $R_2.b = R_3.b$, build a hash table on this result, and probe it with $R_1(a)$ on $R_1.a = R_2.a$. After decomposition, we are left with the middle plan. Here, we cannot push Lookups down as we did before, but we can pull Expands up through a hash table build! The resulting plan is the rightmost plan. R_2 probes R_3, remembers an iterator on R_3, which is stored in the hash table on R_2 aside the tuples in-place of actual tuples from R_3. Later, we probe this hash table with R_1, extract this iterator which we then Expand at the very end. Notice how we have successfully partitioned the plan, we have Lookups below and Expands above, cheap shrinking operators below, growing operators above. Thus, we have avoided a potentially very expensive hash table build on a growing intermediate result and prevented the diamond problem.

3.3 Semantics

To more easily refer to Lookup & Expand plans, we introduce a new notation. In particular, this notation needs to be able to encode the intermediary state after a Lookup has been executed. Consider the state after R_2 probes R_3 in Figure 4. We possess an iterator on R_3, but we do not have the actual tuples at hand. We need an Expand operator to actually access attributes from R_3. We denote this state as $R_2 \longrightarrow R_3$ (pronounced as R_2 looks-up R_3)[3]. We refer to R_2 as the head. We are able to access attributes of the relations in the head as they have been *expanded.* The outgoing arrows correspond to iterators. After an expansion $e_{R_3}(R_2 \longrightarrow R_3) \equiv R_2 \bowtie R_3$ we can access all attributes in R_2 and R_3. We will often omit trivial subscripts and the join sign $\bowtie$ when using our notation $e(R_2 \longrightarrow R_3) \equiv R_2 R_3$. We will also directly refer to sets of relations with joins implicit $e(\mathcal{R}_1 \longrightarrow \mathcal{R}_2) \equiv \mathcal{R}_1 \cup \mathcal{R}_2$.

An L&E expression $\mathcal{S}$ is a relation; the operations Lookup & Expand can be seen as extensions to relational algebra, they input and output relations. The grammar for L&E expressions can be stated as ($\longrightarrow$ has right-to-left precedence):

$$\mathcal{R} := \text{any set of relations including } \emptyset \tag{1}$$

$$\mathcal{S} := \mathcal{R} \mid (\mathcal{S}) \mid \mathcal{S} \longrightarrow \mathcal{S} \mid e(\mathcal{S}) \mid \mathit{head}(\mathcal{S}) \tag{2}$$

The following are result equivalences of L&E expressions:

$$\mathcal{R} \longrightarrow \emptyset \equiv \mathcal{R} \tag{3}$$

$$\mathit{head}(\mathcal{R} \longrightarrow \mathcal{S}) \equiv \mathcal{R} \tag{4}$$

$$e_{\mathcal{R}_2}((\mathcal{R}_1 \longrightarrow (\mathcal{R}_2 \longrightarrow \mathcal{S}_2)) \longrightarrow \mathcal{S}_3) \equiv ((\mathcal{R}_1 \cup \mathcal{R}_2) \longrightarrow \mathcal{S}_2) \longrightarrow \mathcal{S}_3 \tag{5}$$

$$\mathcal{S}_1 \longrightarrow \mathcal{S}_2 \longrightarrow \mathcal{S}_3 \equiv \mathcal{S}_1 \longrightarrow (\mathcal{S}_2 \longrightarrow \mathcal{S}_3) \tag{6}$$

$$(\mathcal{S}_1 \longrightarrow \mathcal{S}_2) \longrightarrow \mathcal{S}_3 \not\equiv \mathcal{S}_1 \longrightarrow (\mathcal{S}_2 \longrightarrow \mathcal{S}_3) \tag{7}$$

$$(\mathcal{S}_1 \longrightarrow \mathcal{S}_2) \longrightarrow \mathcal{S}_3 \equiv (\mathcal{S}_1 \longrightarrow \mathcal{S}_3) \longrightarrow \mathcal{S}_2 \tag{8}$$

The Equations 3-5 are definitions, Equations 6 and 7 describe operator precedence and Equation 8 demonstrates Lookup reordering. We also use the shorthand $e^*(\mathcal{S})$ to refer to expanding S entirely with a sequence of expands such that all the contained relations are in the head. The plans for Figure 3 and Figure 4 can be stated as $e^*((R_1 \longrightarrow R_2) \longrightarrow R_3)$ and $e^*(R_1 \longrightarrow R_2 \longrightarrow R_3)$ respectively.

Lookups & Expands bridge the gap between relational query processing and factorized query processing[4]. If your database does binary hash joins, you are already factorizing when you build a hash table, you simply are not benefitting from it, as the benefits come from allowing the query optimizer to reorder Lookups.

In the following, we will analyze the capabilities of L&E decomposition, demonstrating that, with simple extensions, all four cases of the diamond problem identified in Section 2 can be addressed.

[3]A lookup $\mathcal{R}_2 \longrightarrow \mathcal{R}_3$ is only valid iff. there is at least one join edge between the two sets of relations, i.e. $\mathcal{R}_2 \longrightarrow \mathcal{R}_3$ is only valid if $\mathcal{R}_2 \bowtie \mathcal{R}_3$ is not cross-product.

[4]Note that an L&E expression corresponds to a factorized d-representation [37] of the query, with the main difference being that L&E expressions only factorize the build sides on the join predicates.

3.4 Acyclic Queries

L&E decomposition can provide the same guarantees as the Yannakakis algorithm without requiring additional semi-joins. This can be achieved with two-phase plans, where the first phase of execution exclusively consists of pushed down Lookups and the second phase exclusively consists of Expands. The Lookup phase is exclusively shrinking, and the Expand phase is exclusively growing, but bounded by the query's output size. As no dangling tuples could remain after the Lookup phase, such a plan is instance optimal. Two examples of two-phase plans can be seen in Figures 3 and 4. In both examples, we were able to find a join order in which all the Lookups could be pushed down below Expands. In fact, if a query is α-acyclic, we will show that we can always find such a two-phase plan.

Similar to semi-join reduction, we can utilize the GYO ear removal algorithm to determine an order for the first phase of Lookup operators. A relation is an *ear* if all its join attributes (with remaining relations) are contained in another relation. Every time an ear is removed, we draw an edge from the containing relation to the removed relation. We repeat this process until there are no remaining relations[5]. The resulting graph is a tree, and the edges represent Lookups of an L&E expression. We follow with a sequence of Expand (e^*) operations to compute the join result. The resulting plan is two-phase, and thus, instance optimal up to a constant factor.

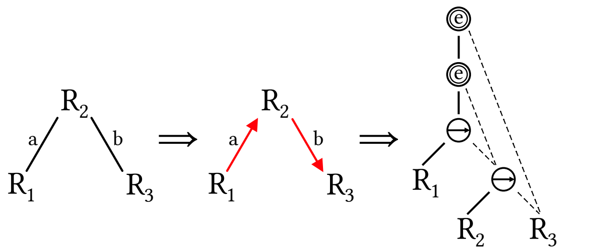

Figure 5: Relations and join attributes, possible ear removal order, and resulting two-phase L&E plan

In Figure 5, we demonstrate a possible execution of the algorithm on the query from Figure 4. We start by removing $R_3(b)$, whose join attribute b is contained in $R_2(a,b)$. We can then remove $R_2(a,b)$, as its remaining join attribute a is contained within $R_1(a)$. R_1, as the only remaining relation, becomes the head of Lookup sequence, resulting in the L&E expression $e^*(R_1 \rightarrow R_2 \rightarrow R_3)$.

The advantages of L&E over the Yannakakis algorithm are twofold. First, eliminating semi-joins can directly lead to a factor of two improvement in query performance. Second, instead of requiring semi-join filters, we extend the search space of join optimizers with two-phase plans. The optimizer can then pick the best plan among all possible plans, including two-phase and non-two-phase plans.

In this work we focus on equi-joins and do not consider inequality predicates neither from an execution perspective [39] nor from an optimization perspective [45].

3.5 Cyclic Queries & Worst-Case Optimality

When predicates were acyclic, we guarantee that we can always find an L&E plan where all Lookup were pushed down below Expands. Such two-phase plans guarantee that no dangling tuples remain after the first phase, implying that they are instance optimal. Unfortunately, when predicates form cycles, we cannot guarantee that such plans exist, and must rely on WCOJs for better runtime guarantees. However, as discussed in Section 2.2, WCOJs are inefficient and hard to integrate into existing optimizers. We also find that generic WCOJs are often not necessary. In the following, we will first demonstrate that ternary joins with 3 inputs can replace WCOJs in a large class of queries. Then, we will introduce the Expand3 operator, that extends L&E plans with per tuple adaptivity and worst-case optimality.

3.5.1 When Achieving Worst-Case Optimality is Easy. We would like to demonstrate a few examples where WCOJs produce little or no benefit over binary joins. Based on these examples, we will propose ternary joins, which, combined with binary joins, can produce worst-case optimal plans for a large class of queries.

Consider the triangle query $R(a,b) \bowtie S(b,c) \bowtie T(c,a)$. For simplicity, assume that all relations have size N. If we optimize the AGM minimization linear program, we will find that the optimal fractional edge cover is 0.5 for all relations resulting in the worst-case result size of $N^{1.5}$. Binary join, in contrast, takes $O(N^2)$ time in the worst case, as any join of two relations may have quadratic size. WCOJs are clearly beneficial.

Now consider the quadrangle query $R(a,b) \bowtie S(b,c) \bowtie T(c,d) \bowtie U(d,a)$. The optimal fractional edge cover is either 0.5 for all relations or $(1,0,1,0)$ (symmetric covers are omitted). Regardless, the worst-case result size is N^2. The runtime complexity for binary joins is also N^2; we first join two relations each $R \bowtie S$ and $T \bowtie U$. Each takes $O(N^2)$ time. Then we join these intermediate results together, where both the input and output are $O(N^2)$. Thus the total runtime is $O(N^2)$. There is a worst-case optimal binary join plan for quadrangle queries.

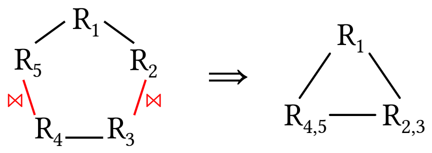

Figure 6: Pentagon query transformed into a triangle query

Finally consider the pentagon query $R(a,b) \bowtie S(b,c) \bowtie T(c,d) \bowtie U(d,e) \bowtie V(e,a)$. The worst-case result size is $N^{2.5}$. All binary join plans might take $O(N^3)$ time for a query; they are not worst-case optimal. Ternary joins can be worst-case optimal. We can join together $S \bowtie T$ and $U \bowtie V$, which leaves us with three relations R, $(S \bowtie T)$, and $(U \bowtie V)$, which we can join with a ternary join. This idea is illustrated in Figure 6. If we can assume that the ternary join is worst-case optimal, the entire query runs in $O(N^{2.5})$ time, same as a 5-way worst-case optimal join. There is a combined binary & ternary join plan that is worst-case optimal[6].

Binary joins are worst-case optimal for all n-gon queries where n is even. Binary joins combined with a final ternary join is worst-case

[5]A query is α-acyclic iff. only a single relation remains after all ears are removed [2].

[6]For the ternary join to be strictly worst-case optimal as we have described here, it needs to carefully manage duplicates in all inputs and the attributes c and e, the attributes used by the initial joins. The intermediate results can be projected onto their remaining edges $\Pi_{b,d} S \bowtie T$ and $\Pi_{d,e} S \bowtie T$ and the attributes c and e can be joined in after the ternary join is executed. It is hard to determine when additional projections are beneficial in practice; ternary joins have some overhead even without it. Our implementation of Expand3 ignores these issues.

optimal for all n-gon queries. In the upcoming section, we show that, allowing for arbitrary combinations of binary and ternary joins, a large class of queries can be answered with worst-case optimal runtime, without the need for larger multiway joins.

3.5.2 Ternary Joins are Enough for Many Queries. In the examples of the previous subsection, the fractional edge cover weight w was 0.5 for all relations. In fact, the query graphs where w is one of $\{0, 0.5, 1\}$ are exactly the query graphs where binary and ternary joins are worst-case optimal (with an additional restriction to be elaborated later). We have empirically verified all queries in the CE benchmark [9] to possess this condition. Nonetheless, this property is quite abstract. To give a better intuition of queries possessing this property, we also prove that all queries with binary edges in fact result in fractional edge cover weights in $\{0, 0.5, 1\}$.

Lemma 3.1. *If all attributes are present in at most 2 relations, there is an optimal fractional edge cover with weights in* $\{0, 0.5, 1\}$.

We give the proof for Lemma 3.1 in our technical report [5]. The underlying idea is that, whenever we have weights larger or smaller than 0.5, we can always round them to 0 or 1 without increasing the objective. Note that Lemma 3.1 is sufficient but not necessary for amenable weights. While queries tend to have transitive predicates such as $R.a = S.a = T.a$, we observe that such predicates in real queries often lead to acyclic structures and rarely to complex cyclic structures. A counterexample cyclic query where attributes occur 3 times is $R(a, b, c) \bowtie S(a, b, d) \bowtie T(a, c, d) \bowtie U(b, c, d)$. If user queries have such structures, fractional edge cover weights may be distinct from $\{0, 0.5, 1\}$, and full multiway joins may have better runtime.

Lemma 3.2. *If there is an optimal fractional edge cover with weights w in* $\{0, 0.5, 1\}$*, there is a plan with binary/ternary joins and semi-join reductions that computes the query result in worst-case optimal time, as long as the $|R_{max}| \leq |R_{min}|^2$ where R_{max} is the largest and R_{min} the smallest relations with fractional cover weights* 0.5.

We give a constructive proof for Lemma 3.2 in our technical report [5]. In summary, we recursively decompose the query into three partitions, where the sizes of the partitions do not exceed the query result size. These partitions are computed recursively and joined together using ternary joins. We essentially show that we can choose such partitions by starting with the largest relation, and iteratively growing that partition until it grows too large, at which point we decompose the query into the partition containing the largest relation, a partition containing the most recently added relations that took us over the size limit, and the remaining relations. There are certain additional cases in the full proof that result in the $|R_{\text{max}}| \leq |R_{\text{min}}|^2$ condition. It is rare for the relation sizes to be so significantly different. Additionally, it becomes less likely that the fractional cover values for R_{max} and R_{min} are both 0.5 as their difference grows. We have empirically verified that every query in the CE benchmark fulfills the conditions of Lemma 3.2.

In summary, we have shown for a large class of queries that a combination of ternary and binary joins is worst-case optimal. Queries can be constructed where ternary joins are not worst-case optimal. Still, for WCOJs to be beneficial, both the query structure and input data must be quite peculiar. We believe that only using binary and ternary joins will lead to more robust performance, as the optimizer can more easily reason about these operators. In Section 5, we show that exclusively using WCOJs can leads to orders of magnitude slowdowns, demonstrating that choosing them over ternary joins is a risky trade-off for the optimizer.

3.5.3 Ternary Expand (Expand3). We have shown that ternary joins are often good enough for worst-case optimality. They are also easier to understand and simple to implement in the framework of Lookup & Expand operators. In the following, we will describe the ternary Expand (Expand3) operator that, combined with two Lookups, implements a ternary worst-case optimal join. We base ternary-join on Algorithm-2 from Ngo et al. [35], opting to do intersections based on hashing.

Consider the cyclic query $R(a, b) \bowtie S(b, c) \bowtie T(c, a)$. The Lookup & Expand plan for this query might look like $e_T\ (e_S(R \rightarrow S) \rightarrow T)$. Note how we expand before the second lookup as both R and S share predicates with T. If we avoid the inner expansion $(R \rightarrow S) \rightarrow T$, our intermediate result contains R with two iterators, one for S and one for T. Expand3 takes these two iterators, and computes their intersection on the predicate $S.c = T.c$. The resulting plan looks like $e3_{S,T}\ ((R \rightarrow S) \rightarrow T)$.

Ngo et al. [35] show that, for this plan to be worst-case optimal, the intersection must be computed in time in the order of minimum iterator length. Expand3 picks the iterator referring to the smallest number of tuples, and makes hash-lookups into the tuples referred to by the other iterator. To facilitate these lookups, we build two hash tables on S and T each, primary hash tables on $S.b$ and $T.a$ and secondary hash tables on $S.\{b, c\}$ and $T.\{a, c\}$. The primary hash tables are used within the Lookup operators while the secondary hash tables are used within Expand3 for the intersection. The operation of Expand3 is illustrated in the following pseudocode:

```
def expand3(r, iteratorS, iteratorT,
            secondaryTableS, secondaryTableT, produce):
  if iteratorS.length < iteratorT.length:
    iteratorSmall = iteratorS
    secondaryTable = secondaryTableT
    primaryKey = r.a
  else:
    iteratorSmall = iteratorT
    secondaryTable = secondaryTableS
    primaryKey = r.b
  for v in iteratorSmall:
    secondaryKey = v.c
    for vp in secondaryTable[(primaryKey,secondaryKey)]:
      produce(v, vp)
```

The Expand3 operator (and its preceding Lookups) are not pipeline breakers. Large inputs can be streamed through without materializing tuples. For example, consider the quadrangle query $R(a, b) \bowtie S(b, c) \bowtie T(c, d) \bowtie U(d, a)$. We can join R and S and subsequently perform a ternary join: $e3_{T,U}\ ((e_S(R \rightarrow S) \rightarrow T) \rightarrow U)$. The result size of $e(R \rightarrow S)$ might be large, but since we can pipeline the result without materializing it, its impact on performance will be low. For some input data, it might be further beneficial to reduce the size of S before executing the Lookup $R \rightarrow S$. L&E decomposition allows for such an optimization without any additional semi joins: $e3_{T,U}\ (e_S(R \rightarrow (S \rightarrow T)) \rightarrow U)$. Note how $e3$ is able to use the iterator to T from the Lookup $S \rightarrow T$. Such optimizations are not possible without decomposing joins.

3.6 Queries with Duplicates

Queries may produce many duplicates in intermediate results, resulting in redundant work. In the diamond hardened join framework, we eagerly aggregate tuples when building hash tables, by evaluating available aggregations at that point in the plan as described by Fent et al. [16]. This results in no additional materializations, thus improving robustness at very little cost. Nonetheless, eager aggregation is mainly beneficial for free-connex[7] queries. If the projection contains attributes from multiple relations, but the join keys of those relations are not projected as in the query $\pi_{a,c}(R(a,b) \bowtie S(b,c))$, more advanced techniques are needed [12].

3.7 Reordering Restrictions & Compensation

Reordering restrictions are a common problem in query optimization. The worst offender for reordering restrictions are outer-joins. They are hard to reorder as they can produce null values. With L&E decomposition, Expands can be pushed above outer-joins without changing the query result, as Lookups produce exactly the matching tuples. Note that the same decomposition cannot be made with Bloom filters (probabilistic semi-joins):

$$(R \bowtie S) ⟕ T \equiv e((R \longrightarrow S) ⟕ T) \not\equiv ((R \mathbin{\widetilde{\ltimes}} S) ⟕ T) \bowtie S$$

Outer-joins can also result in the diamond problem, as they may be growing as well as shrinking. Thus, we would like to apply Lookup & Expand decomposition for outer-joins. Interestingly, if we make Lookup the null-producing operator, and keep Expand simple, this allows us to freely push Expands up join trees. Even though we do not have full reorderability for all operations[8], we are able to produce plans with many more orderings for Expands.

We define two additional parameters for Lookup, whether it "produces-nulls" and/or "produces-all". (1) A produce-null Lookup, after finding matches for all tuples in the left side, produces null tuples with iterators to unmatched tuples on the right. The hash table build reserves space in the tuple storage for markers to mark matched tuples. (2) A produce-all Lookup does not filter out unmatched tuples on the left, but instead, produces them with iterators pointing to null right-side tuples.

The following pseudocode illustrates these new parameters:

```
for left in tuplesLeft:
  it = htRight.find(left.key)
  if it.done():
    if produceAll:
      produce(left, NULL)
  else:
    produce(left, it)
  if produceNulls and not iterator->matchMarker:
    # First time, mark all matches
    it2 = it.copy()
    while not it2.done():
      it2->matchMarker = True

if produceNulls:
  for right in tuplesRight:
    if not right.matchMarker:
      produce(NULL, right)
```

[7]A query is free-connex if it is α-acyclic and remains α-acyclic after joining with an arbitrary relation that contains the projection attributes [8].

[8]For full reorderability, expensive compensation operators are needed [46].

A left-outer join is decomposed into a produce-all Lookup and an Expand. A right-outer join is decomposed into a produce-null Lookup and an Expand. A full-outer join is decomposed into a produce-null & produce-all Lookup and an Expand. The Expand needs to handle NULL iterators by producing a right-side consisting of NULL values. Under these definitions, Expands can be pushed through produce-all and/or produce-null Lookups freely, thus the optimizer can focus on reordering Lookups and execute the dangerous (for the diamond problem) Expands as late as possible (before the attributes produced by the right side of the Expand are used).

L&E decomposition significantly increases the possible reorderings compared to binary hash joins. For example, for star queries, all possible orderings for Expands can be generated. However, compensation operators [46] are still needed for full reorderability.

3.8 Summary

If the query plan is acyclic, there is an instance optimal L&E plan. If the query plan is cyclic there is (most likely) a worst-case optimal L&E plan. If the query contains duplicates, there are many factorized and/or eagerly aggregated L&E plans. If the query contains outer-joins, there are L&E plans that allow for many reorderings not possible with binary joins.

The search space for L&E plans contains many very powerful plans and all their combinations. This is the search space in which we will attempt to find the *best* plan, the plan we want to execute. The upcoming Section 4 will describe how we conduct this search.

4 COST-BASED OPTIMIZATION

In this section, we review the principles of join order optimization and propose an optimization strategy for finding an efficient ordering for Lookup, Expand, and Expand3 operations for arbitrary queries. To achieve this, we demonstrate an encoding for Lookup & Expand operator trees that makes it easy to integrate them into an existing query optimizer.

We are proponents of constructive (bottom-up) dynamic programming optimizers [29] due to their efficiency, and base our ideas on extending such optimizers with physical properties such as groupings and orderings [13]. In principle, there is nothing that prevents a top-down [15] or transformative optimizer [21] from supporting Lookup & Expand queries. Regardless, we base our following discussions on the DPhyp algorithm [30] for its efficiency and ability to handle reordering restrictions.

A query optimizer constructs many execution plans and tries to find the plan that will most efficiently compute the query. Given a cost function that determines the efficiency of a plan, an optimizer tries to find the optimal plan of minimum cost.

Relational algebra queries exhibit the Bellman principle of optimality [3]: There is an optimal plan where all its subtrees are optimal. Constructive (bottom-up) optimizers exploit the principle of optimality by first finding optimal plans for subqueries, which are gradually combined together to build up the full optimal plan that will compute the full query. We say that two plans with equivalent output have equivalent *state*. Constructive optimizers avoid generating too many plans by only storing one plan per unique *state*, the plan with the lowest cost. For binary joins, this would imply one plan per each set of relations.

The *state* of a L&E operator tree must also uniquely determine its output. This state can be encoded using L&E expressions, which were introduced in Section 3. To make sure that there is a bijection between possible states and encodings, we can restrict ourselves to only using minimized L&E expressions only consisting of sets of relations and Lookups. An arbitrary L&E expression can be minimized by combining matching Lookups and Expands.

$$e_S(R \rightarrow S) \rightarrow T \Rightarrow RS \rightarrow T$$

The resulting expression forms a tree of sets of relations. The nodes are sets of relations and the edges are the Lookups.

As we now have a definition and an encoding for state, we can easily extend a constructive optimization algorithm to support L&E. A possibility is to generate all possible L&E plans. However, this results in a huge search space with many suboptimal plans. In situations where cardinality estimates are unreliable, a large search space can work against the optimizer similar to the "fleeing from knowledge to ignorance" problem [27]; the optimizer would pick the most underestimated plan rather than the best plan. We instead only allow the optimizer to reorder Lookups, and place Expands based on two fundamental heuristics: (1) To prevent the diamond problem, Expands should happen as late as possible[9]. (2) Too many delayed Expands, especially ones with low multiplicity, lead to pointer chasing and thus bad cache utilization. Thus, we allow certain Expands to happen early based on cardinality estimates.

In the following, we start with a simple exhaustive enumeration of all possible L&E plans, and further refine it with the two heuristics. The following pseudocode generates all L&E plans by iterating over all connected component pairs of relations [30]:

```
for s1, s2 in connectedComponentPairs(query):
  for p1 in plans[s1]:
    for p2 in plans[s2]:
      lookupPlan = p1 -> p2
      plans[s1 + s2].insert(lookupPlan)
      for p in generateExpandPlans(lookupPlans):
        plans[s1 + s2].insert(p)
```

For every set of relations, we have a corresponding set of plans. For every connected pair of sets of relations (in an order corresponding to the principle of optimality), we iterate over all plan pairs, connect them via a Lookup, and then generate all possible expansions on top of those plans. Within the `insert` function, we take care to only preserve the smallest plan if a plan of the same state already exists.

We improve exhaustive enumeration by only reordering Lookups, and greedily placing expands as in the following:

```
for s1, s2 in connectedComponentPairs(query):
  for p1 in plans[s1]:
    for p2 in plans[s2]:
      req1 = computeBoundary(s1, s2)
      req2 = computeBoundary(s2, s1)
      lookupPlan = e(p1, req1) -> e(p2, req2)
      plans[s1 + s2].insert(lookupPlan)
```

Using `computeBoundary`, we compute all relations (boundary nodes) in a first set that are connected to a second set over join conditions. Before forming the `lookupPlan`, we make sure all boundary nodes are available in the heads of both sides of the Lookup. In other words, we delay expansions until the first moment they are referenced by a join predicate, at which point they are required. This generates safer plans, as the dangerous growing Expand operators are executed as late as possible while Lookups filter on all available join conditions.

To improve plan runtime, we further refine the optimizer with the ability to execute expands early before they are needed:

```
for s1, s2 in connectedComponentPairs(query):
  for p1 in plans[s1]:
    for p2 in plans[s2]:
      req1 = computeBoundary(s1, s2)
      req2 = computeBoundary(s2, s1)
      left = e(p1, req1)
      right = e(p1, req2)
      left = earlyExpand(left, |left|, |right|)
      right = earlyExpand(right, |right|, |left|)
      lookupPlan = left -> right
      plans[s1 + s2].insert(lookupPlan)
```

`earlyExpand` greedily decides whether to eagerly place Expands on subplans. This decision is based on cardinality estimates. While there are cases where cardinality estimates can contain great errors, there are also cases where a system can partially guarantee its estimates are reliable. In such cases, we can allow for certain Expands to happen early, greatly increasing average case performance.

We have discussed how to encode the state of Lookup & Expand plans, how to integrate them into an existing constructive optimizer, and discussed how we can intelligently restrict the search space of the optimizer to generate plans with better robustness by reducing the impact of cardinality estimation errors. In the following, we will further detail the cost function we utilize for our operators. Finally, we will take a look at how we support the Expand3 operator.

We have omitted implementation details such as removing redundant predicates in cases where two join predicates might imply a third. To eliminate redundant predicates, the optimizer should keep track of the equivalence class of attributes, and make sure the utilized equality predicates between equivalent attributes never form a cycle. This is essential to prevent unnecessary expansion resulting from our requirement that all relevant predicates are utilized as early as possible. Consider $R(a) \bowtie S(a) \bowtie T(a)$ with three join predicates $R.a = S.a$, $R.a = T.a$, and $S.a = T.a$. If we want to lookup from R to $S \rightarrow T$, the right side does not need to be expanded to be able to apply the predicate $R.a = T.a$. Simply doing the Lookup $R \rightarrow (S \rightarrow T)$ is enough as both $R.a = S.a$ and $S.a = T.a$ will have been applied, making $R.a = T.a$ redundant. Note that, by considering redundant predicates, we have implicitly ascertained that this query is α-acyclic and not cyclic, in contrast to the query $R(a,b) \bowtie S(b,c) \bowtie T(a,c)$, where we would need an Expand on the right side, which could be replaced by a Ternary-Expand as will be discussed in Subsection 4.2.

4.1 Sideways Information Passing

Simply reordering Lookups does not eliminate the diamond problem, unless a two-phase plan is picked as described in Section 3.4. To minimize the diamond problem in general, we can apply semi-join filters across the plan using sideways information passing [10, 24].

When constructing a hash table, we can build additional Bloom filters on different sets of attributes, which can then be used to eagerly filter other intermediate results. Instead of following the

[9]To facilitate this, the cost function needs to underestimate the potential cost of Expands. Our full cost function is given in our technical report [5].

reverse GYO [51] order as in full semi-join reduction, we order the hash table build operators by input cardinality and build filters on the inputs of smaller tables first, which are then employed to filter the inputs of larger tables, similar to how Yang et al. [49] determine the topology of a predicate transfer graph. In a sense, by ordering hash tables by size, we minimize the size of the largest hash table.

4.2 Supporting Ternary Expansion

The optimizer needs to be able to place Expand3 operators to support worst-case optimal cyclic plans. As Expand3 operators are only useful for cyclic queries, we must take care to only consider them when a Lookup closes a cycle. We can detect this by checking whether the input to the Lookup contains boundary relations that we needed to expand. Consider the triangle query $R(a,b) \bowtie S(b,c) \bowtie T(c,a)$. We have two subplans $R \longrightarrow S$ and T. If we proceed to do a Lookup from $R \longrightarrow S$ to T, we need to expand R and S to be able to apply the join predicate $R.a = T.a$, resulting in the plan $e_S(R \longrightarrow S) \longrightarrow T$. This indicates that this Lookup closes a cycle. In such cases, we can remove the expansion and introduce an Expand3 operator that expands on both the Lookups: $e3_{S,T}((R \longrightarrow S) \longrightarrow T)$. Note that $e3$ corresponds to two separate Expands; the binary plan would have been: $e_T\ (e_S(R \longrightarrow S) \longrightarrow T)$.

5 EVALUATION

In this section we evaluate the effectiveness of Lookup & Expand decomposition and show that Lookup & Expand (1) results in dramatic improvements up to 500x in n:m queries, (2) causes minimal regressions in even the most benign queries. This demonstrates that Lookup & Expand decomposition is a simple and effective approach to improving the robustness of in-memory join processing.

We have implemented Lookup & Expand decomposition in the compiling in-memory database Umbra [33] and compare it with baseline Umbra, WCOJs due to Freitag et al. [18], and DuckDB [38] v0.9.2 on a microbenchmark, the relational benchmarks TPC-H Scale Factor 10, and JOB [26], and the CE graph benchmark [9]. The TPC-H benchmark mainly consists of key-foreign key joins with little to no skew in the data, thus we do not expect our optimizations to result in significant improvements. The JOB benchmark exclusively consists of α-acyclic queries, meaning that semi-join filters and Lookup & Expand decomposition are partially useful in queries demonstrating the diamond problem, while WCOJs and Expand3 will not be. The CE benchmark was designed to stress test the cardinality estimators for graph databases and contains a wide range of complex pattern matching queries, both acyclic and cyclic. We have translated the queries to SQL to evaluate them on relational databases[10]. We only evaluate the queries from CE that have result size smaller than 10^9. The queries in the original benchmark can go up to 10^{16}, and such large result sizes are not feasible to compute without eager aggregation. We expect all our optimizations to be useful on this benchmark, as it contains many queries with complex structures and large intermediate results.

We have 4 aspects of our implementation that we individually evaluate: (1) *ht*: Using a hash table with dense collision lists (adjacency array) [6] instead of a chaining hash table (the Umbra default). Optimizing the hash table is fundamental for robustness against skew [17]. This optimization will improve most queries. (2) *lookup*: Lookup & Expand decomposition and Expand3 optimization. These optimizations target the diamond problem in acyclic and cyclic queries. (3) *SIP*: Using additional Bloom filters [7] as in Section 4.1. Bloom filters both increase the robustness of join ordering through sideway information passing. Additionally, Bloom filters are more amenable to high performance vectorized filtering and lead to constant factor improvements in selective joins. (4) *agg*: Using eager aggregation. This optimization mainly targets the CE benchmark, which exclusively consists of `count(*)` graph pattern matching queries with sometimes large result sizes. Eager aggregation also makes it feasible for Umbra to run through the entire CE benchmark, including the queries with extremely large result sizes (which we omit here to focus on the other optimizations).

All benchmarks have been evaluated on a Ryzen 5950X system with 16 cores and 32 threads with 64GB of RAM. The databases are allowed to use 50GB of RAM for queries. All queries are forced to run in-memory. We repeat all query executions 10 times (upto an hour) and show the minimum execution time. We disable index nested loop join in Umbra and its variants to avoid that some queries are dominated by index access costs instead of join costs.

5.1 Microbenchmark

L&E decomposition results in asymptotic improvements over binary joins, meaning that we can construct queries where an L&E plan is arbitrarily faster than any possible binary join plan. To demonstrate this, we have constructed one acyclic and one cyclic query where base table have sizes $\theta(N)$, L&E requires $\theta(N)$ time to execute, and binary joins require $\theta(N^2)$ time to execute. With $N = 5 \cdot 10^4$, we find that *lookup* is around 730x and 15x faster than *ht* and *WCOJ* respectively on the acyclic query. On the cyclic query, the speedup over *ht* and *WCOJ* is around 200x and 8x respectively.

$[a, b]$ is the range of integers from a to b inclusive. The acyclic query is $X(a,b) \bowtie Y(b,c) \bowtie Z(c,d)$ for the relations $X = (1,1) \cup ([1,N] \times (2))$, $Y = (1,1) \cup ((2) \times [4,N]) \cup ([3,N] \times (3))$, $Z = (1,1) \cup ((3) \times [1,N])$. The cyclic query is $R(a,b) \bowtie S(b,c) \bowtie T(c,a)$ for the relations $R = S = T = ((1) \times [1,N]) \cup ([1,N] \times (1))$.

5.2 Relational & Graph Benchmarks

In the TPC-H Scale Factor 10 benchmark, we observed the hash table with dense collision lists results in a significant improvement of around 5% over the default chaining hash table, while the rest of the optimizations do not result in significant improvements or regressions. WCOJs, in contrast, result in around a 12x slowdown on total benchmark runtime.

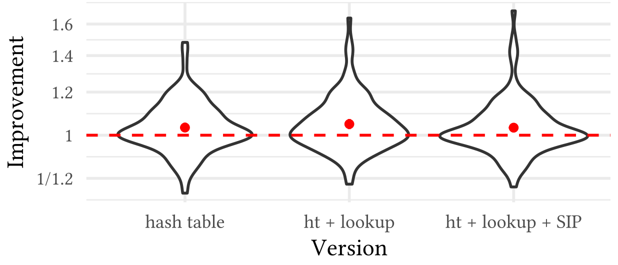

Figure 7: Runtime improvement of different optimizations over baseline Umbra on the JOB benchmark.

[10]Our reproducibility package contains all queries and datasets for all benchmarks.

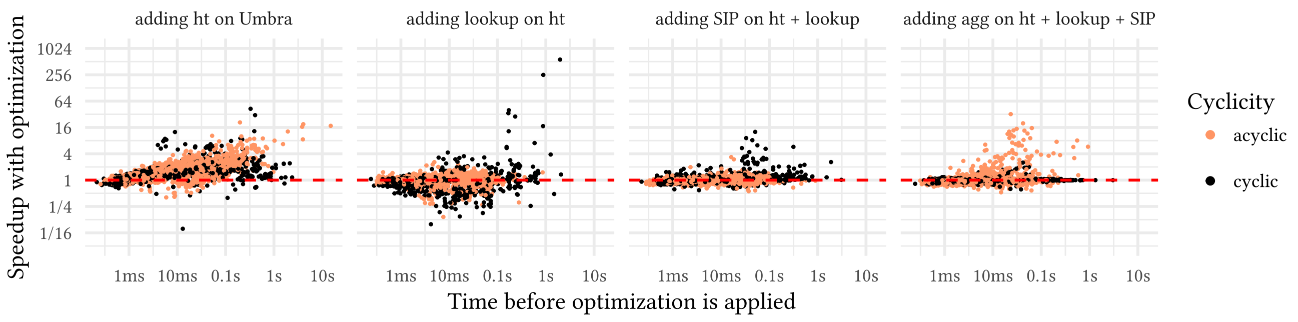

Figure 8: Runtime improvement of individual queries with different optimizations on the CE benchmark.

In Figure 7, we show the runtime improvement of each optimization over the baseline Umbra implementation on the JOB benchmark. We observed that the *ht* optimization results in noticeable improvements, while the rest of the optimizations do not result in significant improvements or regressions. We found similar results for LDBC SNB BI [42] SF 10: *ht* results in some improvements, while the performance with the rest of the optimizations remains similar. In these benchmarks, simple pushdowns of join filters suffice to achieve good performance. We also observed that WCOJs result in around 25x slowdown in JOB. As the JOB benchmark consists of α-acyclic queries, we do not expect WCOJs to be beneficial, still, the slowdown demonstrates the difficulty of integrating worst-case optimal joins into a general-purpose query optimizer.

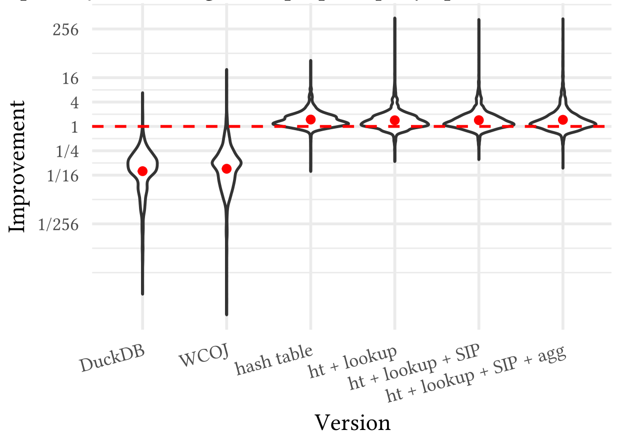

Figure 9: Runtime improvement of different optimizations over baseline in the CE benchmark.

In Figure 9, we show the runtime improvement of each optimization over the baseline Umbra implementation on the CE benchmark. In this benchmark we find that baseline Umbra is significantly faster than DuckDB and WCOJs and that our optimizations make Umbra even faster. In Figure 8, we show the runtime improvement of each optimization over the previous optimization. We find that simply replacing the hash table with a skew optimized one results in the biggest improvements overall. Lookup, Expand, and Expand3 significantly improve performance for certain pathological queries. The biggest wins are for cyclic queries, where dblp_cyclic_q9_06 improves by a factor of over 500x from 2s to 4ms. However, some queries have also slowed down, watdiv_cyclic_q10_00 went from 4ms to 40ms, a slowdown of around 10x, due to bad plan choices resulting from estimation errors. Such impacts are often more significant on queries that utilize Expand3. Even though Expand3 has a smaller constant factor overhead compared to WCOJs, it is still slower than normal Expands. We also found that the total runtimes of ht + *SIP* and *ht* + *lookup* + *SIP* differ only by around 1%, even including the 500x improved cyclic query. This implies that the techniques we utilize, while improving pathological queries, have constant overheads that are small but not insignificant. After all optimizations are applied, the total runtime of CE is around 100x as fast as DuckDB[11], around 230x as fast as WCOJs[12], and around 2.4x as fast as baseline Umbra.

The CE benchmark contains 1839 α-acyclic and 1165 cyclic queries, and we find that the cyclicity of the query impacts the effectiveness of the optimizations. *ht* improves queries across the board. *lookup* and *SIP* primarily improve cyclic queries and have little impact on acyclic queries. Finally, *agg* is primarily useful for acyclic queries, with little impact on cyclic queries.

Overall, we find that all our optimizations significantly improve the performance of some of the slowest queries, resulting in more predictable performance for all queries. Among the optimizations, Lookup presents the hardest trade-off. The theoretical strength of L&E is observable in some queries, but not all. Nevertheless, it is not hard to find or construct queries where L&E with Expand3 is exceptionally useful due to their runtime complexity. For example, only WCOJs and Expand3 are able to execute the Graphalytics [23] LCC query on the dota-league graph without running out of memory. We are also optimistic there is further room to optimize the newly proposed operators.

6 CONCLUSION

We have proposed a simple and effective approach to improving the robustness of in-memory join processing by decomposing joins into two suboperators, Lookup & Expand. In contrast to existing techniques for tackling robustness, our technique is able to improve the performance of pathological queries by many orders of magnitude while not regressing the performance of well behaved queries. We have given a theoretical foundation for our approach, by analyzing four categories of pathological queries and showing that our approach is able to tackle each category. To further support this theoretical foundation, we have demonstrated the strength of our approach on a variety of benchmarks.

[11]We exclude the 29 queries out of 3004 where DuckDB runs out of memory.
[12]We exclude the 6 queries out of 3004 where WCOJs take over an hour.

REFERENCES

[1] Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size Bounds and Query Plans for Relational Joins. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. 739–748. https://doi.org/10.1109/FOCS.2008.43

[2] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. 1983. On the Desirability of Acyclic Database Schemes. *J. ACM* 30, 3 (July 1983), 479–513. https://doi.org/10.1145/2402.322389

[3] Richard Bellman and Stuart Dreyfus. 2010. *Dynamic Programming*. Vol. 33. Princeton University Press. https://doi.org/10.2307/j.ctv1nxcw0f arXiv:j.ctv1nxcw0f

[4] Philip A. Bernstein and Dah-Ming W. Chiu. 1981. Using Semi-Joins to Solve Relational Queries. *J. ACM* 28, 1 (Jan. 1981), 25–40. https://doi.org/10.1145/322234.322238

[5] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. *Hardening Joins with Suboperator Decomposition*. Technical Report. TUM School of Computation, Information and Technology.

[6] Altan Birler, Tobias Schmidt, Philipp Fent, and Thomas Neumann. 2024. Simple, Efficient, and Robust Hash Tables for Join Processing. In *Proceedings of the 20th International Workshop on Data Management on New Hardware (DaMoN '24)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3662010.3663442

[7] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

[8] Nofar Carmeli and Markus Kröll. 2020. Enumeration Complexity of Conjunctive Queries with Functional Dependencies. *Theory of Computing Systems* 64, 5 (July 2020), 828–860. https://doi.org/10.1007/s00224-019-09937-9

[9] Jeremy Chen, Yuqing Huang, Mushi Wang, Semih Salihoglu, and Ken Salem. 2022. Accurate Summary-Based Cardinality Estimation through the Lens of Cardinality Estimation Graphs. *Proceedings of the VLDB Endowment* 15, 8 (April 2022), 1533–1545. https://doi.org/10.14778/3529337.3529339

[10] Ming-Syan Chen, Hui-I Hsiao, and Philip S. Yu. 1997. On Applying Hash Filters to Improving the Execution of Multi-Join Queries. *The VLDB Journal* 6, 2 (May 1997), 121–131. https://doi.org/10.1007/s007780050036

[11] Sophie Cluet and Guido Moerkotte. 1995. On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products. In *Database Theory — ICDT '95 (Lecture Notes in Computer Science)*, Georg Gottlob and Moshe Y. Vardi (Eds.). Springer, Berlin, Heidelberg, 54–67. https://doi.org/10.1007/3-540-58907-4_6

[12] Shaleen Deep, Xiao Hu, and Paraschos Koutris. 2021. Enumeration Algorithms for Conjunctive Queries with Projection. *LIPIcs, Volume 186, ICDT 2021* 186 (2021), 14:1–14:17. https://doi.org/10.4230/LIPICS.ICDT.2021.14

[13] Marius Eich and Guido Moerkotte. 2015. Dynamic Programming: The next Step. In *2015 IEEE 31st International Conference on Data Engineering*. 903–914. https://doi.org/10.1109/ICDE.2015.7113343

[14] Ronald Fagin. 1983. Degrees of Acyclicity for Hypergraphs and Relational Database Schemes. *J. ACM* 30, 3 (July 1983), 514–550. https://doi.org/10.1145/2402.322390

[15] Pit Fender and Guido Moerkotte. 2013. Top down Plan Generation: From Theory to Practice. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 1105–1116. https://doi.org/10.1109/ICDE.2013.6544901

[16] Philipp Fent, Altan Birler, and Thomas Neumann. 2023. Practical Planning and Execution of Groupjoin and Nested Aggregates. *The VLDB Journal* 32, 6 (Nov. 2023), 1165–1190. https://doi.org/10.1007/s00778-022-00765-x

[17] Daniel Flachs, Magnus Müller, and Guido Moerkotte. 2022. The 3D Hash Join: Building on Non-Unique Join Attributes. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org.

[18] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proceedings of the VLDB Endowment* 13, 12 (July 2020), 1891–1904. https://doi.org/10.14778/3407790.3407797

[19] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. *Combining Worst-Case Optimal and Traditional Binary Join Processing*. Technical Report TUM-I2082. Technische Universität München.

[20] Georg Gottlob, Matthias Lanzinger, Davide Mario Longo, Cem Okulmus, Reinhard Pichler, and Alexander Selzer. 2023. Structure-Guided Query Evaluation: Towards Bridging the Gap from Theory to Practice. https://doi.org/10.48550/arXiv.2303.02723 arXiv:2303.02723 [cs]

[21] G. Graefe and W.J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of IEEE 9th International Conference on Data Engineering*. 209–218. https://doi.org/10.1109/ICDE.1993.344061

[22] Muhammad Idris, Martin Ugarte, and Stijn Vansummeren. 2017. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1259–1274. https://doi.org/10.1145/3035918.3064027

[23] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *Proceedings of the VLDB Endowment* 9, 13 (Sept. 2016), 1317–1328. https://doi.org/10.14778/3007263.3007270

[24] Zachary G. Ives and Nicholas E. Taylor. 2008. Sideways Information Passing for Push-Style Query Processing. In *2008 IEEE 24th International Conference on Data Engineering*. 774–783. https://doi.org/10.1109/ICDE.2008.4497486

[25] Ahmad Khazaie and Holger Pirk. 2023. SonicJoin: Fast, Robust and Worst-case Optimal. https://doi.org/10.48786/EDBT.2023.46

[26] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query Optimization through the Looking Glass, and What We Found Running the Join Order Benchmark. *The VLDB Journal* 27, 5 (Oct. 2018), 643–668. https://doi.org/10.1007/s00778-017-0480-7

[27] V. Markl, P. J. Haas, M. Kutsch, N. Megiddo, U. Srivastava, and T. M. Tran. 2007. Consistent Selectivity Estimation via Maximum Entropy. *The VLDB Journal — The International Journal on Very Large Data Bases* 16, 1 (Jan. 2007), 55–76. https://doi.org/10.1007/s00778-006-0030-1

[28] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. 2021. LSQB: A Large-Scale Subgraph Query Benchmark. In *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA '21)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3461837.3464516

[29] Guido Moerkotte and Thomas Neumann. 2006. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB '06)*. VLDB Endowment, Seoul, Korea, 930–941.

[30] Guido Moerkotte and Thomas Neumann. 2008. Dynamic Programming Strikes Back. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 539–552. https://doi.org/10.1145/1376616.1376672

[31] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[32] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[33] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org.

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

[35] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *ACM SIGMOD Record* 42, 4 (Feb. 2014), 5–16. https://doi.org/10.1145/2590989.2590991

[36] Dan Olteanu and Jakub Závodný. 2012. Factorised Representations of Query Results: Size Bounds and Readability. In *Proceedings of the 15th International Conference on Database Theory (ICDT '12)*. Association for Computing Machinery, New York, NY, USA, 285–298. https://doi.org/10.1145/2274576.2274607

[37] Dan Olteanu and Jakub Závodný. 2015. Size Bounds for Factorised Representations of Query Results. *ACM Transactions on Database Systems* 40, 1 (March 2015), 2:1–2:44. https://doi.org/10.1145/2656335

[38] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[39] Maximilian Reif and Thomas Neumann. 2022. A Scalable and Generic Approach to Range Joins. *Proceedings of the VLDB Endowment* 15, 11 (July 2022), 3018–3030. https://doi.org/10.14778/3551793.3551849

[40] Tobias Schmidt. 2019. *Index-Structures for Worst-Case Optimal Join Algorithms*. Bachelor's Thesis. Technische Universität München.

[41] K. Stocker, D. Kossmann, R. Braumandi, and A. Kemper. 2001. Integrating Semi-Join-Reducers into State-of-the-Art Query Processors. In *Proceedings 17th International Conference on Data Engineering*. IEEE Comput. Soc, Heidelberg, Germany, 575–584. https://doi.org/10.1109/ICDE.2001.914872

[42] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proceedings of the VLDB Endowment* 16, 4 (Dec. 2022), 877–890. https://doi.org/10.14778/3574245.3574270

[43] Todd L. Veldhuizen. 2013. Leapfrog Triejoin: A Worst-Case Optimal Join Algorithm. https://doi.org/10.48550/arXiv.1210.0481 arXiv:1210.0481 [cs]

[44] Qichen Wang, Xiao Hu, Binyang Dai, and Ke Yi. 2023. Change Propagation Without Joins. *Proceedings of the VLDB Endowment* 16, 5 (Jan. 2023), 1046–1058. https://doi.org/10.14778/3579075.3579080

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

[46] TaiNing Wang, Yunpeng Niu, and Chee-Yong Chan. 2023. Complete Join Reordering for Null-Intolerant Joins. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 1734–1746. https://doi.org/10.1109/ICDE55515.2023.00136

[47] Yisu Remy Wang, Max Willsey, and Dan Suciu. 2023. Free Join: Unifying Worst-Case Optimal and Traditional Joins. *Proceedings of the ACM on Management of Data* 1, 2 (June 2023), 1–23. https://doi.org/10.1145/3589295

[48] Weipeng P. Yan and Per-Åke Larson. 1995. Eager Aggregation and Lazy Aggregation. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio (Eds.). Morgan Kaufmann, 345–357.

[49] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. 2024. Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries. In *14th Annual Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, USA, January 14-17, 2024*. www.cidrdb.org.

[50] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Very Large Data Bases, International Conference on Very Large Data Bases*.

[51] C.T. Yu and M.Z. Ozsoyoglu. 1979. An Algorithm for Tree-Query Membership of a Distributed Query. In *COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979.* 306–312. https://doi.org/10.1109/CMPSAC.1979.762509

[52] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking Ahead Makes Query Plans Robust: Making the Initial Case with in-Memory Star Schema Data Warehouse Workloads. *Proceedings of the VLDB Endowment* 10, 8 (April 2017), 889–900. https://doi.org/10.14778/3090163.3090167