

One Join Order Does Not Fit All: Reducing Intermediate Results with Per-Split Query Plans

Yujun He^{1,2*}, Hangdong Zhao³, Simon Frisk¹, Yifei Yang¹, Kevin Kristensen¹

Paraschos Koutris¹, Xiangyao Yu¹

¹University of Wisconsin-Madison ²Southern University of Science and Technology ³Microsoft Gray Systems Lab

ABSTRACT

Minimizing intermediate results is critical for efficient multi-join query processing. Although the seminal Yannakakis algorithm offers strong guarantees for acyclic queries, cyclic queries remain an open challenge. In this paper, we propose SplitJoin, a framework that introduces *split* as a first-class query operator. By partitioning input tables into heavy and light parts, SplitJoin allows different data partitions to use distinct query plans, with the goal of reducing intermediate sizes using existing binary join engines. We systematically explore the design space for split-based optimizations, including threshold selection, split strategies, and join ordering after splits. Implemented as a front-end to DuckDB and Umbra, SplitJoin achieves substantial improvements: on DuckDB, SplitJoin completes 43 social network queries (vs. 29 natively), achieving 2.1× faster runtime and 7.9× smaller intermediates on average (up to 13.6× and 74×, respectively); on Umbra, it completes 45 queries (vs. 35), achieving 1.3× speedups and 1.2× smaller intermediates on average (up to 6.1× and 2.1×, respectively).

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/hyj2003/SplitJoin>.

1 INTRODUCTION

Minimizing intermediate join results has been a crucial goal for achieving good performance for multi-join queries. For *acyclic* queries, the seminal Yannakakis algorithm [34] can provably guarantee intermediate table sizes of at most $O(N + OUT)$, where N and OUT are the sizes of query input and output, respectively. This can lead to fast and robust query plans in practical systems [28, 32, 33, 37]. For *cyclic* queries, however, the intermediate table sizes can be much larger than the Yannakakis bound, and thus are challenging to minimize in both theory and practical systems.

Prior work has extensively studied how to address the inefficiency of binary join plans on *cyclic* queries and skewed data (i.e., some values appear many times in a column). Worst-case optimal join (WCOJ) algorithms [22, 25, 27] guarantee runtimes bounded by the maximum possible output size, thereby avoiding the quadratic

intermediate blow-ups that can arise in traditional plans. For instance, in the triangle query, a binary join plan may generate $\Theta(N^2)$ intermediate tuples, while a WCOJ algorithm completes in $O(N^{1.5})$ time. Despite this theoretical elegance, WCOJ has not been widely adopted in practice: binary joins remain dominant after decades of engineering optimization, WCOJ implementations typically rely on complex indexing structures, and efficient parallelization is challenging even with recent advances such as HoneyComb [31].

Complementary to WCOJs, another line of theoretical work investigates *partition-based strategies* [8, 9, 15, 17]. These algorithms mitigate skew directly by separating heavy and light values in the join key domains and evaluating them under different subplans. By isolating values with high occurrence, partitioning provides finer control over join orders and reduces intermediate result sizes. In the triangle query, for example, partitioning can also guarantee a $O(N^{1.5})$ bound on intermediate results. Although these strategies are theoretically powerful, their adoption in real systems remains limited due to the complexity of existing approaches [17], leaving a gap between theory and practice.

Building on the above insights, we take the initial step toward making partition-based strategies practical. Our goal is to bridge the gap between theoretical advances and real-world query engines. To achieve this, we propose SplitJoin, which introduces **split** as a first-class operator in query processing. The split operator partitions an input table R into a light table (R_L) and a heavy table (R_H), which are then processed with potentially different query plans. SplitJoin can substantially reduce intermediates without requiring new data structures like in existing WCOJ algorithms. In this paper, we systematically explore how splits can be incorporated into traditional execution frameworks during query planning.

Our first contribution is a general framework (Section 3) that incorporates split as a first-class citizen in a query plan. This framework reveals a rich design space with several key dimensions, including which table(s) to split, on which attributes to split, how to determine the appropriate split thresholds, and how to decide the best join order on each part of the partition.

From a theoretical perspective, we show in Section 4 that our framework can be used to create a large class of provably worst-case optimal algorithms for queries on binary relations that all have the same cardinality. However, achieving this requires strict rules on how to split and which join orders to use. To improve practical performance, we relax these rules in favor of heuristics.

From a practical perspective, we implement an instantiation of our framework as SplitJoin (Section 5), a front-end layer that can work for any SQL database: it takes a join query as input, generates an optimized query that incorporates split operators where appropriate, and submits it to the underlying database engine for execution. This design allows SplitJoin to take advantage of the

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

*Work done while interning at the University of Wisconsin-Madison.

existing query processing infrastructure that may support only binary joins while transparently introducing split-based optimizations. We conduct experiments by placing this layer in front of two state-of-the-art databases, DuckDB [24] and Umbra [12], and evaluate on commonly used social network datasets, which often exhibit data skew and cyclic query patterns. Our experiments demonstrate that SplitJoin significantly reduces intermediate result sizes and improves query execution time, while preserving the efficiency of carefully optimized binary joins. In particular, DuckDB with SplitJoin successfully completes 43 queries compared to 29 with native DuckDB, and Umbra with SplitJoin completes 45 queries compared to 35 with native Umbra. For the subset of queries that can finish both with and without SplitJoin, SplitJoin delivers substantial improvements: on DuckDB, queries are 2.1× faster and produce 7.9× smaller intermediates on average (up to 13.6× and 74×, respectively); on Umbra, it achieves 1.3× speedups and 1.2× smaller intermediates on average (up to 6.1× and 2.1×, respectively). Notably, for all query plans generated by SplitJoin in our experiments, we have manually checked and verified that they are provably worst-case optimal, demonstrating that our proposed heuristics work well for the tested queries and data.

The remainder of this paper is organized as follows. Section 2 introduces the background and motivation of our work, highlighting the limitations of existing solutions. Section 3 presents an overview of SplitJoin’s design space. Section 4 provides the theoretical motivation underlying our design. Section 5 describes our current heuristics and optimizations in each design dimension. Section 6 reports experimental findings and Section 7 concludes the paper.

2 BACKGROUND AND RELATED WORK

We focus on the evaluation of natural join queries. As an example, given three relations $R(A, B)$, $S(B, C)$, and $T(C, A)$, our objective could be to evaluate the triangle query, expressed as follows:

$$Q(A, B, C) = R(A, B) \bowtie S(B, C) \bowtie T(C, A).$$

In SQL, this query can be written as:

```
SELECT R.A, R.B, S.C
FROM R, S, T
WHERE R.B = S.B AND R.A = T.A AND S.C = T.C;
```

In this work, we primarily focus on joins over binary relations, i.e., relations with exactly two attributes. Joins over binary relations have been extensively studied in prior work [5, 22, 26] and provide a foundation that can be extended to more general multi-attribute cases. Moreover, this setting is particularly relevant for graph workloads, where tuples typically represent edges and the data often exhibit skew, which makes binary relations a representative and practically important case to study.

For a (natural) join query Q with binary relations, we define the *query graph* as follows: for every uniquely named attribute X in the query, we create a vertex X . For a relation $R(A, B)$, we create an edge between the vertices A and B . If e is an edge of the query graph, we use R_e to refer to the relation that corresponds to it.

2.1 The Problem with Data Skew

Most database engines evaluate join queries using a *binary join plan*: this plan forms a tree of join operators, each operator computing

the join between two tables. Binary join plans are theoretically optimal when the structure of the query is *acyclic*, as shown by Yannakakis [34]. In these cases, we can find a join plan that runs in time $O(N + OUT)$, where N is the database input size and OUT is the size of the output.

However, for non-acyclic queries (as the triangle query in our example), binary join plans are provably suboptimal [4]. We explain this observation next with an example using the triangle query.

Let’s first consider the simple no-skew case where all input relations are as in Figure 1(a):

$$R = S = T = \{(1, 1), (2, 2), \dots, (N, N)\}.$$

We have three possible join orders, and for any order the intermediate relation (which is $R \bowtie S$, $R \bowtie T$, or $S \bowtie T$) has exactly N tuples. This is a good case for binary join, where the intermediate table does not explode in size.

In contrast, consider the following skewed example (Figure 1(b)):

$$R = S = T = \{(1, 1), (1, 2), (1, 3), \dots, (1, N), (2, 1), (3, 1), \dots, (N, 1)\}.$$

In this instance, the data is highly skewed, where in table R , the value $A = 1$ appears N times and the value $B = 1$ also appears N times (similarly for tables S and T). This skew leads to a disproportionately large number of intermediate tuples for binary join. Specifically, regardless of the chosen join order, we will create an intermediate table with $\Theta(N^2)$ tuples, while the output size of the entire query is actually $3N - 2$, which is linear in N .

To further illustrate why a join result can explode in size, we define the *degree* of a value: for a value a in the domain of A , its *degree* is defined as $d_R(a) = |\sigma_{A=a}R|$ (i.e., the number of tuples in relation R whose attribute A equals a), and define the *maximum degree* of attribute A in R as $\delta_R(A) = \max_{A=a} d_R(a)$ (i.e., the largest degree of any value in a given attribute).

When joining R with S on attribute B , a single tuple in S may match up to $\delta_R(B)$ tuples in R , implying that the output size of the join contributed by that tuple is at most $\delta_R(B)$. Consequently, the join output size of R and S is bounded by $\delta_R(B) \cdot N$.

In the first, no-skew example above, $\delta_R(B) = 1$ and $\delta_S(B) = 1$, so the intermediate join result grows linearly with N , and no explosion occurs. In contrast, in the skewed example, the maximum degree of any attribute in any relation is always N . In this case, a single join key value in a relation can match N tuples in the other relation, producing $\Theta(N^2)$ intermediate tuples in the worst case, even though the final output size of the query may be much smaller.

This analysis above illustrates that a fundamental reason of intermediate result explosion in binary join plans is *data skew*: high-degree values in the join attributes can lead to disproportionately large intermediate relations, significantly impacting the efficiency of binary join processing.

2.2 Worst-Case Optimal Joins

To solve the problem we illustrated above in skewed cyclic queries, a line of work has focused on developing *worst-case optimal join (WCOJ)* algorithms [1–3, 6, 7, 11, 12, 22, 25, 27, 29, 31]. These algorithms guarantee that the runtime is bounded by the worst-case size of the query output, rather than the potentially much larger intermediate results generated by traditional binary join plans. For example, in theory, all triangle queries can be executed

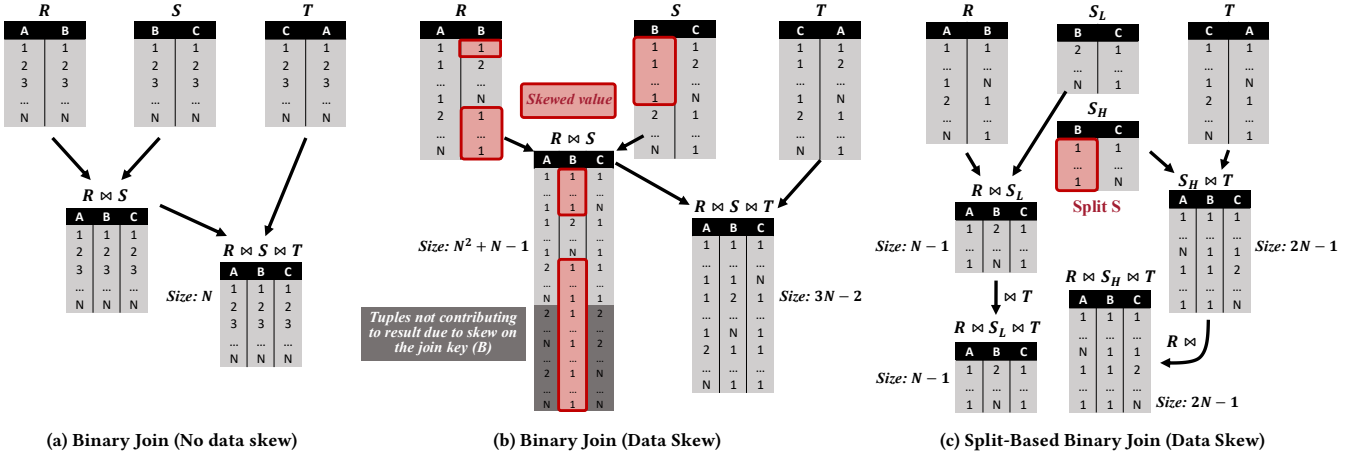


Figure 1: Motivating Example on How Split Avoids Producing Intermediate Tuples That Do Not Contribute to the Final Result.

in time $O(N^{1.5})$, instead of $O(N^2)$. Well-known algorithms such as Leapfrog Triejoin [27] have been successfully adopted in several systems [1, 2, 11, 12, 25], and they provide strong theoretical guarantees, where both query runtime and intermediate table sizes can be bounded by $O(N^{1.5})$ for triangle queries.

We identify three key limitations of existing WCOJ solutions. First, these algorithms require sophisticated indexing structures (e.g., hash trie) which are not supported by most database systems today. Second, in real-world workloads, WCOJ algorithms may not always outperform carefully optimized binary joins. Third, these special indexing structures are inherently limited in exploiting modern multi-core architectures and in taking advantage of the well-established efficiency of binary join algorithms [23]. Recent efforts, such as HoneyComb [31], have attempted to parallelize WCOJ algorithms, yet the complexity of indexing and limited scalability remain challenging.

2.3 Partitioning to the Rescue

Another line of theoretical research explores *partition-based* strategies [8, 9, 15, 17] to mitigate the data skew problem. These algorithms separate the “heavy” (high-degree) and “light” (low-degree) values in the join key domains and evaluate each partition under different subplans. By treating values with high degree independently, these approaches limit the maximum degree we mentioned before and gain finer-grained control over join orders, thus reducing the size of intermediate results.

Taking the general triangle query for example, assume there are no duplicate tuples in R , S , and T . We split S into two tables: S_H , consisting of tuples whose B -values appear more than $\sqrt{N}$ times, and S_L , containing the remaining tuples. For S_H , there can be at most $\sqrt{N}$ distinct B -values. Since no duplicate (b, c) pairs exist, each C -value can appear at most $\sqrt{N}$ times—once per distinct B -value—with at most $\sqrt{N}$ different C -values. Consequently, $\delta_{R_H}(C) \leq \sqrt{N}$, so executing $(S_H(B, C) \bowtie T(C, A)) \bowtie R(A, B)$ produces at most $O(N^{1.5})$ intermediate results. For S_L , each B -value occurs at most $\sqrt{N}$ times, meaning that $\delta_{S_L}(B) \leq \sqrt{N}$, and the total size of a

different join order $(R(A, B) \bowtie S_L(B, C)) \bowtie T(C, A)$ is also bounded by $O(N^{1.5})$.

The general triangle query illustrates how the heavy-light partitioning strategy effectively reduces the maximum degree of join keys in each sub-table, thereby mitigating intermediate result explosion caused by skew. Returning to our running example, in Figure 1(c), if we split table S into $S_H = \{(1, 1), (1, 2), (1, 3), \dots, (1, N)\}$ and $S_L = \{(2, 1), (3, 1), \dots, (N, 1)\}$ depending on the degree on attribute A , and execute $(S_H(B, C) \bowtie T(C, A)) \bowtie R(A, B)$ and $(R(A, B) \bowtie S_L(B, C)) \bowtie T(C, A)$ separately, the intermediate result size would be only $(2N - 1)$ for the former subquery and $(N - 1)$ for the latter.

Although partitioning has been well studied in theory, it has not been adopted in real systems due to the complexity of existing approaches. For example, The theoretical state-of-the-art PANDA algorithm [17] further partitions each relation into not only two, but multiple pieces; and each one is part of a different sub-plan. This gap between theoretical benefits and practical applicability motivates us to ask the following question, which drives this work: *how can we leverage **split** as a new operator in query processing to solve the data skew problem?*

2.4 Other Related Work

Free Join: Free Join [30] made a step toward reconciling the theoretical guarantees of WCOJ with the practical advantages of binary hash-join execution. It provides a unified framework that, in principle, can capture both robustness and efficiency. In practice, however, Free Join suffers from several shortcomings: the current implementation is restricted to a single-threaded setting, its reliance on specialized index structures complicates deployment, and the current optimization procedure does not always produce high-quality plans. These challenges limit its applicability in modern query engines, leaving the question open of how to systematically close the gap between theory and practice.

Lookup & Expand: A recent work aims to minimize intermediate results in join processing by decomposing a traditional hash join into two sub-operations: *Lookup* and *Expand* [5]. In this framework,

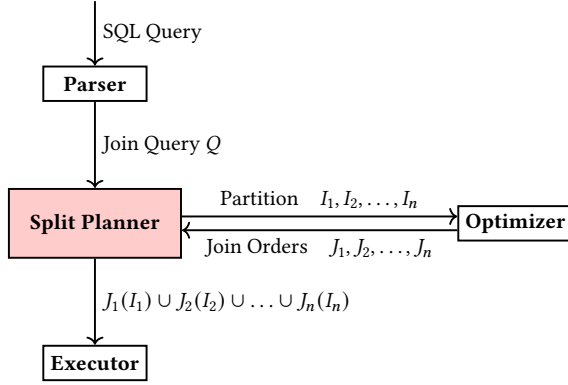


Figure 2: Overview of the Split Planner.

Lookup verifies the existence of a matching tuple in the join partner relation without immediately materializing all join partners, thereby filtering out dangling tuples early. Expand, in contrast, is invoked only when it becomes necessary to enumerate all matches, which postpones the creation of large intermediate results. This separation enables more robust join execution and opens opportunities for reordering these two operators in the query plan. The Lookup & Expand paradigm may be further enhanced when combined with SplitJoin, which further limits the intermediate result size.

3 THE SPLITTING FRAMEWORK

In this section, we present a general framework that incorporates splitting into query evaluation.

The framework extends the query optimizer with an additional component called a *split planner* (see Figure 2). The split planner consists of two phases: the *split phase* (Algorithm 1) and the *join phase* (Algorithm 2). This component is integrated into the optimizer so that query planning is not only aware of join orders and access paths, but also of possible split strategies.

The split phase takes as input the join query Q , a database instance I (i.e., the set of tables referenced by Q), and a set of splits Σ . A *split* consists of a pair $(\mathcal{R}, A)$, where $\mathcal{R}$ is a set of relations, and A is an attribute that appears in all relations $R \in \mathcal{R}$. For every split $(\mathcal{R}, A)$, we split the values of the attribute A into two subsets: light (A_L) and heavy (A_H) . Then, every relation $R \in \mathcal{R}$ is split into a light (R_L) and heavy (R_H) part according to the split of the join attribute. This creates two subinstances I_L, I_H . Each of the two subinstances is then recursively partitioned using the remaining splits in the split set Σ until Σ becomes empty. In general, we want to make sure that every relation is split at most one time in Σ (some relations may not be split at all). The end result is a partition of I in smaller subinstances $I_1, I_2, \dots$.

Algorithm 1 is parametrized by two functions. The first function is *splitAttribute*, which determines the heavy and light values of the common (join) attribute. The second is a function *chooseSplitSet*, which will select a split set Σ that will be used as the initial input. We will discuss their possible implementation in the next two sections.

The join phase takes as input the subinstances $I_1, I_2, \dots$ generated from the split phase, and then decides an appropriate join order for each subinstance. Finally, it computes the union of the results to form the final query output. One important consideration in the

Algorithm 1: Split Phase

Input: Join query Q , instance I , split set Σ

Output: a partition $\{I_1, I_2, \dots\}$

```

1 if  $\Sigma = \emptyset$  then
2   return  $\{I\}$ 
3  $(\mathcal{R}, A) \leftarrow$  a split in  $\Sigma$ ;
4  $(A_L, A_H) \leftarrow \text{splitAttribute}(\mathcal{R}, A)$ ;
5 for every  $R \in \mathcal{R}$  do
6    $R_L \leftarrow \{t \in R \mid t.A \in A_L\}$ ;
7    $R_H \leftarrow \{t \in R \mid t.A \in A_H\}$ ;
8  $I_L \leftarrow I$  with  $\mathcal{R}$  replaced by  $\{R_L \mid R \in \mathcal{R}\}$ ;
9  $I_H \leftarrow I$  with  $\mathcal{R}$  replaced by  $\{R_H \mid R \in \mathcal{R}\}$ ;
10  $\Sigma \leftarrow \Sigma \setminus \{(\mathcal{R}, A)\}$ ;
11 return  $\text{Split}(Q, I_L, \Sigma) \cup \text{Split}(Q, I_H, \Sigma)$ 

```

Algorithm 2: Join Phase

Input: Join query Q , partition $\mathcal{I} = \{I_1, I_2, \dots\}$

Output: $Q(I)$

```

1 for each part  $I_i \in \mathcal{I}$  do
2   compute  $Q(I_i)$  using a skew-aware query optimizer
3 return  $\bigcup_i Q(I_i)$ 

```

join phase is that the optimizer needs to incorporate the properties of this subinstance (e.g., what is light, what is heavy) in order to find the best plan. We will also discuss how this can be achieved in the next two sections.

4 THEORETICAL MOTIVATION

We next present an instantiation of our splitting framework for binary relations that is provably worst-case optimal. In particular, it achieves the AGM bound $\mathcal{AGM}(Q) = N^\rho$, where ρ is the minimum fractional edge cover of the query graph. We next discuss how to instantiate the split and join phase to achieve the desired bound.

Split Phase. We pick a split set Σ as follows. For every relation R , we choose either one of its attributes (say A), and add $(\{R\}, A)$ to the split set. In other words, we split every relation using one of its attributes – this means we will end up with 2^ℓ subinstances (ℓ is the number of relations). Then, we choose $A_L = \{a \mid d_R(a) \leq \sqrt{N}\}$ and $A_H = \{a \mid d_R(a) > \sqrt{N}\}$. In other words, we split the values according to their degree in R and a threshold $\tau = \sqrt{N}$. This threshold choice means that if we split a relation $R(A, B)$ into R_L, R_H using attribute A , the degree of any value of A will be light in R_L (smaller than $\sqrt{N}$), while the degree of any value of B will be light in R_H . We can depict this visually by thinking the query graph as a directed graph, where an edge is directed away from the light attribute. Since we split every relation, each subinstance corresponds to one way of making the query graph directed.

Join Phase. We now discuss how we pick a good join order for a subinstance I_i . Recall from above that we can think of this subinstance as a directed query graph G_i of Q . The key intuition we want to follow is: *try to join on light attributes as much as possible*. We define this formally via an operation called a *light join*. Given

an intermediate result $T(X_1, \dots, X_k)$ in a query plan, a light join $T(X_1, \dots, X_k) \bowtie R(X_k, Y)$ is a join where the edge (X_k, Y) is in G_i , i.e. X_k is light in relation R .

In the below pseudocode, we describe the join order. Intuitively, we do the following: pick any relation to start with (Line 2), and perform as many light joins as possible (Lines 5-7). If no light joins are possible, start over somewhere else and repeat. When there are two intermediate results that reach each other, they are merged (Lines 8-11). When we are left with one intermediate result, the query output is obtained (Line 14).

The algorithm below works on queries that are connected, i.e. do not contain cartesian products. Cartesian products are handled by first computing each connected sub-query.

Algorithm 3: Join Ordering

Data: Q : Query, I_i : Input

```

1  $C \leftarrow \{\}$ ; // no intermediate results yet
2 while  $\exists$  attribute  $X$  not in any set in  $C$  s.t. an unused relation
    $R(X, Y)$  is light in  $X$  do
3    $c \leftarrow \{X, Y\}$ ;
4    $T_c \leftarrow R(X, Y)$ ;
5   while  $\exists$  light join between  $T_c$  and an unused relation
       $R(X, Y)$  with  $X \in c$  do
6      $T_{c \cup \{Y\}} \leftarrow T_c \bowtie R(X, Y)$ ;
7      $c \leftarrow c \cup \{Y\}$ ;
8   while  $\exists c' \in C$  such that  $c \cap c' \neq \emptyset$  do
9      $T_{c \cup c'} \leftarrow T_c \bowtie T_{c'}$ ;
10     $c \leftarrow c \cup c'$ ;
11     $C \leftarrow C - \{c'\}$ ;
12   $C \leftarrow C \cup c$ 
13 //  $C = \{c\}$ , i.e. it has a single component
14 return  $T_c$ ;

```

Example 4.1. Consider the following query

$$\begin{aligned}
Q = & R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(A, C) \bowtie R_4(C, D) \bowtie \\
& R_5(A, D) \bowtie R_6(D, E) \bowtie R_7(E, K) \bowtie R_8(B, F) \bowtie \\
& R_9(F, G) \bowtie R_{10}(G, H) \bowtie R_{11}(H, I) \bowtie R_{12}(H, J) \bowtie \\
& R_{13}(I, K) \bowtie R_{14}(J, K) \bowtie R_{15}(C, I) \bowtie R_{16}(B, H)
\end{aligned}$$

The query graph is depicted in the figure below. The color/number indicates which intermediate relation a relation is first joined into by line 5-7. Suppose the algorithm start doing light joins from A . This will yield an intermediate relation with the red (1) relations, since this intermediate has no outgoing edges (light joins). Suppose that next, the algorithm starts expanding from H . It will expand an intermediate relation of the blue (2) relations. The red and blue intermediates will then be merged. Continuing, the green (3) intermediate is computed and merged with the red/blue intermediate. Finally, the same happens with the orange (4) component.

We prove the following theorem in Appendix A.

THEOREM 4.2. *Let Q be a natural join query with binary relations of at most size N . Then, the above instantiation of the splitting framework runs in time at most $\mathcal{AGM}(Q) = O(N^p)$.*

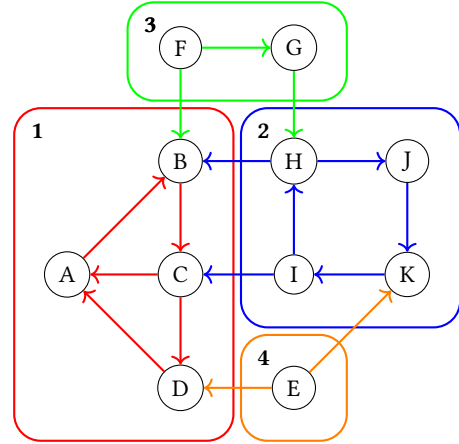


Figure 3: Example query graph and initial intermediate relations

We give some intuition behind the theorem. For any intermediate that is created only with light joins, it is never bigger than $\mathcal{AGM}(Q)$, because light joins grow the intermediate very slowly (the intermediate grows by at most a factor $\sqrt{N}$ for every light join). For queries over binary relations, this is enough to match $\mathcal{AGM}(Q)$. The trickier part is understanding why the merge step matches the AGM bound. It is because the intermediate results have a form that is close to *induced subgraph form*, which is discussed more in the appendix.

Discussion. This section has proven that there is a large class of query plans with splits that match the AGM bound. Indeed, several join orders are produced from Algorithm 3, since we are free to choose what is our starting point and which light join we do first. In practice, these theoretically optimal plans may not be desirable — they split every table, which is not necessary and expensive in practice. In the next section, we will see how we can split more carefully. Additionally, we would like to give the query optimizer more freedom in picking among join orders during the join phase. However, we want to keep the same intuition: light joins are less likely to grow an intermediate result too much and should be given preference. One alternative interpretation of our theoretical result is that by splitting, the query optimizer is much more likely to pick a good join order for each subinstance, since the join orders that satisfy the AGM bound cover a much larger space.

5 SPLITJOIN DESIGN SPACE

While the theoretical algorithm described in Section 4 ensures worst-case optimality, it may not lead to the best performance in practice. In this section, we explore the design space of SplitJoin and propose heuristics that can lead to good practical performance.

5.1 Co-split

The first design decision of SplitJoin is to pick splits of the form $(\{R, T\}, A)$, i.e., we always split two relations at once. To explain why this is beneficial compared to splitting one relation at a time, consider a query Q that has two tables $R(A, B), T(A, C)$ that join on attribute A . If we choose to split R on attribute A , then we create two subrelations R_L and R_H . Now observe that since R, T join on

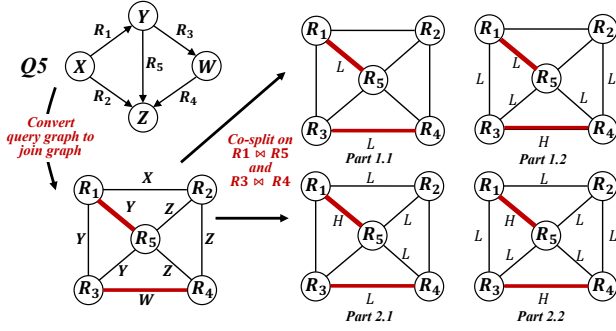


Figure 4: Split Example on Q5, L/H indicates a join is Light/Heavy.

attribute A , this means that in the subinstances I_L, I_H it suffices to keep only the parts of T that join with R_L, R_H respectively:

$$T_L(A, C) \leftarrow T \bowtie \pi_A(R_L)$$

$$T_H(A, C) \leftarrow T \bowtie \pi_A(R_H)$$

In other words, a split on R can be thought as a split on T as well; this can reduce the input size for each join by ensuring that related tuples are split consistently. The co-split strategy makes this explicit. One additional consideration is that we want to avoid splitting both R and T on attribute A independently, since this would generate four subproblems: $R_L \bowtie T_L, R_L \bowtie T_H, R_H \bowtie T_L$, and $R_H \bowtie T_H$.

We can visualize co-splits alternatively by considering the *join graph* of a query Q . This graph can be thought as the "dual" of the query graph: each vertex is a relation, and two vertices (relations) are connected via an edge only if they join on some attribute. Figure 4 shows as example the join graph of query Q_5 (defined in Section 6.2 and Figure 6). A co-split then corresponds to an edge of the join graph. For instance, the co-split $(\{R_1, R_5\}, Y)$ corresponds to the edge $\{R_1, R_5\}$ of the join graph in our running example. It will be convenient to use the shorthand $R_1 \bowtie_Y R_5$ for a co-split.

5.2 Choosing the Split Threshold

In this part, we discuss how we implement the method *splitAttribute*. We will start by presenting the method for a single split, and then extend this to a co-split.

Single Split. Suppose we want to split the values of attribute A in a relation $R(A, B)$ of size N into a heavy part A_H and a light part A_L . Recall the *degree* defined in Section 2, we pick a *split threshold* value $\tau > 0$ such that a value a with $d_R(a) > \tau$ is sent to the heavy part, and with $d_R(a) \leq \tau$ to the light part. The key question is: *how do we choose the right split threshold?*

To answer this question, we need to dive deeper into what is achieved by splitting a relation into two parts.

- **Light part (A_L):** The maximum degree of any value of attribute A is less than τ . This implies that if we join any relation with R_L using A as the join key, the intermediate result will increase by a factor of at most τ .
- **Heavy part (A_H):** The main observation is that the number of heavy values of A is at most N/τ . Thus, the maximum degree of attribute B in $R_H(A, B)$ is at most N/τ . This similarly implies that joining with R_H on B will increase the intermediate result by a factor of at most N/τ .

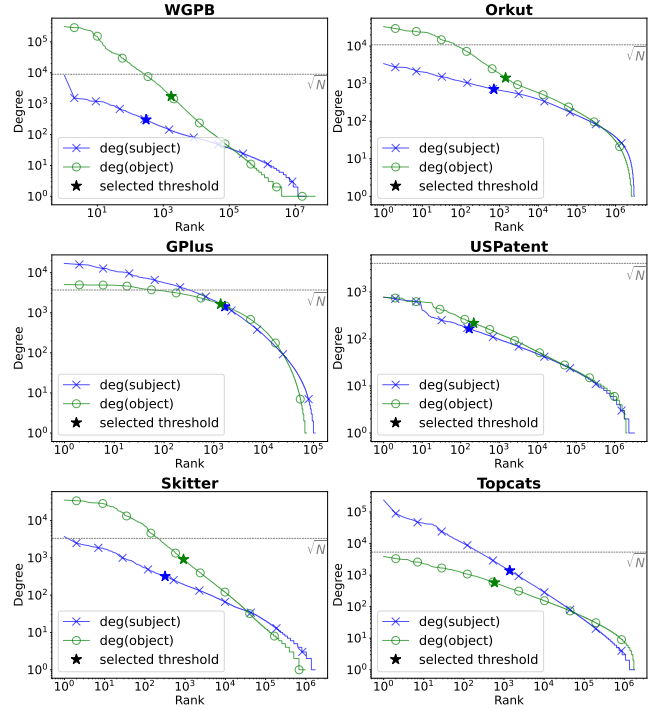


Figure 5: Degree distributions of six datasets. Nodes are ordered by descending degree, with the subject attribute and the object attribute distributions shown on a log-log scale.

Theoretically, we want to balance the two parts, so we would choose τ such that $\tau = N/\tau$, or equivalently $\tau = \sqrt{N}$ as in Section 4. However, this threshold matches only the worst-case bound. In practice, the bound of $\sqrt{N}$ for the number of heavy values can be overly conservative, as the actual degree distribution on real-world datasets often deviates from the theoretical worst-case assumptions. Figure 5 illustrates the degree distributions from several real-world network datasets, which are the datasets in Table 1. As shown in the figure, the vast majority of values have degrees much less than $\sqrt{N}$. Therefore, by replacing the $\sqrt{N}$ threshold with a smaller value K , we can reduce the overhead caused by values with degree less than $\sqrt{N}$ while preserving the worst-case optimality.

We now discuss our method to choose the threshold τ . First, we compute the *degree sequence* [16] of A in $R(A, B)$. The degree sequence is constructed by calculating the degree $d_R(a)$ of every value a in A , and sorting them in non-increasing order:

$$\deg_1 \geq \deg_2 \geq \deg_3 \geq \dots \geq \deg_m$$

Here, m is the number of distinct values in A . We want to guarantee that we pick τ such that the number of values with degree $> \tau$ are at most τ . Hence, we pick τ to be the first index $K \in \{1, \dots, m\}$ in the degree sequence such that $K \geq \deg_K$. Observe that this choice of threshold is sensitive to the specific data distribution, in contrast to the choice of $\sqrt{N}$ which is data-independent. In particular, if the degree distribution is highly skewed (say a zipfian distribution), then $\deg_K$ will be significantly smaller than $\sqrt{N}$, leading to a more precise and effective splitting. Figure 5 shows the chosen threshold of our heuristic compared to the choice of $\sqrt{N}$. Observe that the

chosen threshold is smaller than $\sqrt{N}$, which means that we are much more conservative in what we consider a "light" value.

The split is designed to handle skew by partitioning tuples into heavy and light sides, but this process is not free, since it involves multiple overheads: (i) the cost of partitioning a relation into heavy and light, and (ii) the cost of executing additional join branches for each partition. When the output size of a join is already small, the cost of these extra steps can easily outweigh any potential performance gains from splitting. To address this issue, we tweak the threshold choice in one more way. Specifically, we pick two predefined parameters Δ_1, Δ_2 , and check whether:

$$\deg_1/\Delta_1 \leq K \leq \Delta_2$$

If this condition is satisfied, then we put everything in the light part (this is equivalent to setting the threshold $\tau = +\infty$). Intuitively, the second inequality tells us that there are very few heavy values, and the first one that these few heavy values are not too skewed. In this case, partitioning is unnecessary because of the overheads so we opt to skip it. We choose $\Delta_1 = 5$ and $\Delta_2 = 240$ in our experiments.

Co-Split. Suppose now we want to choose the threshold for a co-split $R \bowtie_A T$. Instead of constructing two degree sequences, we construct a combined degree sequence by considering the combined degree $d_{R,T}(a) = \min\{d_R(a), d_T(a)\}$. We similarly choose the threshold K to be the smallest index such that $K \geq d_{R,T}(a)_K$, and also use Δ_1, Δ_2 to further optimize the threshold choice exactly as in the case of a single split.

Moreover, it is also important to understand how heaviness distributes across related joins. As shown in Figure 4, given that a join is heavy, we can infer that its adjacent joins involving different attributes are light, since the number of distinct attributes in those joins is limited. For example, in part 1.2, if $R_3 \bowtie R_4$ is identified as heavy, then $R_3 \bowtie R_1, R_3 \bowtie R_5, R_4 \bowtie R_5$, and $R_4 \bowtie R_2$ can be inferred to be light. This inference helps us reason about the broader join structure and identify which parts of the query are likely to contribute less to the overall cost.

5.3 Choosing the Split Set

We next turn to the problem of deciding the split set. Recall that our split set consists of co-splits, which can be viewed as edges in the join graph. In order to minimize partitioning, we choose the edges such that they form an *edge packing* of the graph (i.e., we cannot choose two edges that are adjacent to the same table). This guarantees that each relation will be split at most once. We will say that an edge $\{R, S\}$ in the join graph is *uncovered* by a split set Σ if R, S do not occur in Σ .

Specifically, we construct all possible co-split sets through a recursive enumeration procedure, $\text{enum}(\Sigma)$. Initially, we start with $\Sigma = \emptyset$, so we call $\text{enum}(\emptyset)$. At each recursive step, to compute $\text{enum}(\Sigma)$, we first generate all $\Sigma \cup \{R \bowtie_A S\}$ such that the new co-split $R \bowtie_A S$ satisfies two conditions: (i) R, S is an uncovered edge in the join graph w.r.t. Σ ; and (ii) there is no other uncovered edge whose two relations belong to a smaller cycle in the query graph. The function then returns the union of the split sets produced recursively by each $\text{enum}(\Sigma \cup \{R \bowtie_A S\})$. The latter condition makes sure that we prioritize edge packings that contain edges in smaller

cycles. When Σ is an edge packing (i.e., there are no more uncovered edges), we return $\text{enum}(\Sigma) = \{\Sigma\}$ and the recursion terminates.

Example 5.1. Consider again Q_5 (Figure 4). The split set construction prioritizes co-splitting joins within the sub-cycles $R_1(X, Y) \bowtie R_2(X, Z) \bowtie R_5(Z, Y)$ and $R_5(Z, Y) \bowtie R_3(Y, U) \bowtie R_4(U, Z)$, rather than on the joins $R_1 \bowtie R_3$ and $R_2 \bowtie R_4$ that occur in a cycle of length 4. The reason we want to do this is that the former joins each contain a filter-table, allowing the filter effect to be applied immediately during splitting, whereas the latter do not provide this benefit. Hence, we will obtain the following possible co-split sets from the enumeration procedure:

$$\Sigma_1 = \{R_1 \bowtie_Y R_5, R_3 \bowtie_W R_4\}$$

$$\Sigma_2 = \{R_2 \bowtie_Z R_5, R_3 \bowtie_W R_4\}$$

$$\Sigma_3 = \{R_1 \bowtie_X R_2, R_3 \bowtie_W R_4\}$$

$$\Sigma_4 = \{R_1 \bowtie_X R_2, R_3 \bowtie_Y R_5\}$$

$$\Sigma_5 = \{R_1 \bowtie_X R_2, R_4 \bowtie_Z R_5\}$$

Note that the co-splits $R_1 \bowtie_Y R_3$ and $R_2 \bowtie_Z R_4$ will never be chosen by our construction.

Given all the generated split sets, the question is now: which set do we choose for splitting? Our strategy is to prioritize splitting on *light joins*, i.e., joins whose threshold is small.

Technically, we will assign a *cost* to each edge in the join graph: the cost is the threshold k we choose for this co-split. The key idea here is that the intermediate size of the join is bounded by $k \cdot N$, where N is the size of the largest relation involved. The cost of a set of co-split is simply the maximum of the thresholds of each co-split. Therefore, choosing the joins with the smallest threshold yields the tightest upper bound on intermediate result size, and is expected to produce the most efficient plan.

Example 5.2. Consider again the running example for Q_5 , and assume that the threshold for the co-split $R_i \bowtie R_j$ is k_{ij} . Then the cost of Σ_1 would be $\max\{k_{15}, k_{34}\}$, the cost of Σ_2 would be $\max\{k_{25}, k_{34}\}$, and so on. In the case of Figure 4, the smallest cost is Σ_1 , so we would pick this co-split to partition the instance into the four subinstances than can be seen in the figure.

5.4 Split-aware Query Optimizer

Conventional query optimizers in existing database systems are generally unaware of splits. For example, when evaluating triangle queries in DuckDB or Umbra, the optimizer only considers standard join ordering and cost models, without explicitly accounting for the structural changes introduced by splits. Besides, these query optimizers do not utilize degree information. As a result, the execution strategies are often suboptimal, and the potential benefits of splitting cannot be fully exploited.

Recent work [36] incorporates degree information into cost estimation, but its approach relies on solving complex linear programs and assumes precise degree distributions, which limits its practicality for general-purpose optimizers. In particular, it incurs high estimation overhead and cannot handle nested queries, which are common in our split-based plans. Instead, we adopt a simpler yet effective heuristic that directly leverages observed maximum degrees and thresholds to guide the cost model.

Building on this idea, we design a simple split-aware optimizer that explicitly incorporates degree information into query planning. In particular, our approach estimates the cost of query plans by combining the maximum degree of attributes before splitting with the thresholds determined after splitting. This enables the optimizer to better capture the effect of splits and to generate more efficient execution plans. As an example, if we have partitioned $R(A, B)$ on attribute A with threshold K , the cost of the join $T(A, C) \bowtie R_L(A, B)$ can be estimated as $|T| \cdot K$ (since the maximum degree of A in R is at most K). Similarly, the cost of the join $S(B, C) \bowtie R_H(A, B)$ is at most $|S| \cdot K$ (since there are at most K heavy values of A).

Apart from this modification, the overall optimization procedure still follows the conventional dynamic programming (DP) framework used in query optimizers. That is, plan enumeration and cost-based pruning remain unchanged, while only the cost estimation step is extended to be aware of splits using the maximum degree and thresholds. This design ensures compatibility with existing optimizers while enabling them to effectively leverage the advantages introduced by splitting.

6 EXPERIMENTAL EVALUATION

In this section, we present experimental results to demonstrate the effectiveness and advantage of SplitJoin. We begin by briefly describing our implementation in 6.1, then state the experimental setup in Section 6.2, followed by evaluations of overall performance in Section 6.3, then demonstrate our proposed optimizations by effectiveness studies in Section 6.4.

6.1 Implementation

Preprocessing: To determine the thresholds required for splits, we first collect degree information by using aggregate clause for each table in the input query. For each column of every relation, we construct a compact summary table consisting of pairs (value, degree), where degree denotes the number of occurrences of a given attribute value. To keep the overhead manageable, we only record the at most top 100K values with the highest degrees. This summary table is typically very small, amounting to less than 0.1% of the size of the original relation, while still being sufficient to capture the skew patterns that drive effective splitting decisions.

Front-end Layer for Splits: To integrate SplitJoin into existing systems with minimal modification, we designed a lightweight front-end layer that operates independently of the underlying database engine. This layer takes a join query as input and rewrites it into an optimized SQL query that incorporates split operators before passing it to the database. In addition, this front-end layer can leverage the underlying execution engine to collect the statistical information required by preprocessing, ensuring that degree summaries are obtained efficiently without adding additional data management components. The approach is simple to implement and nonintrusive, as it does not require altering the executor internals.

6.2 Experimental Setup

Hardware Platform: We conduct all experiments on a CloudLab [10] machine with r6525 instance type with two AMD EPYC 7543 processors (32 cores per socket, 2 hardware threads per core,

totaling 128 hardware threads) running at 2.8 GHz, equipped with 251 GB DDR4 memory. The used SSD is a Dell Ent NVMe AGN MU U.2 with 1.6 TB capacity.

Tested Dataset: The datasets used in our experiments are listed in Table 1. They are the same graph datasets evaluated in [31], covering a range of sizes, sparsity levels, and degree distributions.

Name	Nodes	Edges	Features
WGPB [13, 14]	54.0M	81.4M	skew, sparse
Orkut [21]	3.07M	117M	uniform, partial dense
GPlus [19]	107K	13.6M	skew, dense
USPatent [18]	3.77M	16.5M	uniform, sparse
Skitter [18]	1.69M	11.1M	partial skew, sparse
Topcats [35]	1.79M	28.5M	skew, partial dense

Table 1: Sizes of the tested graph datasets.

For the WGPB dataset, which originally contains three columns, we discard the second column as it is not used as a join key in our queries, and remove duplicates from the remaining two columns (about 1% reduction regarding the number of rows).

Tested Queries: The queries we used are listed in Figure 6, which are cyclic queries with a number of tables less than nine from the paper [20]. Additionally, we add the 5-cycle query.

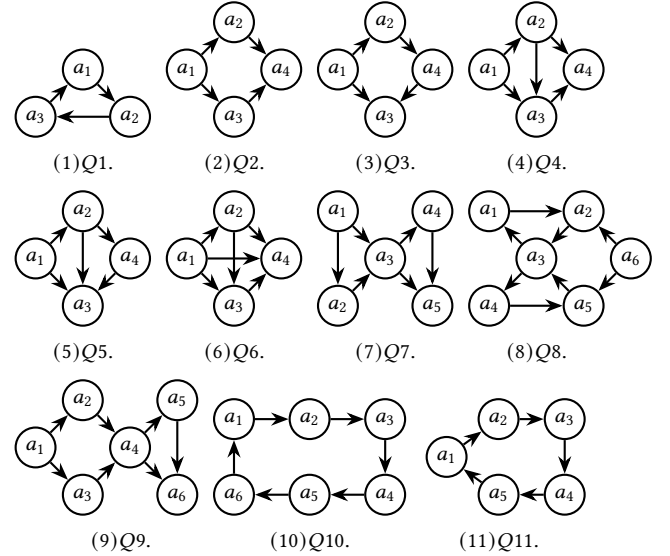


Figure 6: Tested Queries

Tested Systems: We test our method in two representative database systems: DuckDB [24] and Umbra [12].

DuckDB (v1.3.0) is an open-source in-process columnar DBMS optimized for OLAP workloads, providing easy integration into analytical applications within the same process. DuckDB primarily relies on optimized binary joins and has no built-in worse-case optimal joins. Umbra (25.07.1) is a high-performance DBMS designed for modern hardware. It uses a compact columnar format

and morsel-driven execution for parallelism. Umbra natively supports worst-case optimal joins, enabling efficient multiway join execution for dense graph queries.

Metrics: We measure the overall query execution time and intermediate table sizes for all engines. For SplitJoin, we report end-to-end execution time that includes the time for obtaining thresholds, performing table partitioning, and executing the query. For DuckDB, we use its memory mode (all tables stored in memory, no disk I/O). For Umbra, we use the default configuration, except for adjustments to query memory consumption, timeout, and the WCOJ-related options described below.

For both SplitJoin and the baselines, each query is executed four times, and we report the minimum execution time to avoid cold start effects. We set a memory limit of 220 GB and a timeout of 15 minutes for each run, and use 32 threads for query execution.

6.3 Overall Performance Evaluation

To assess the overall performance benefits introduced by the strategies proposed in the previous sections, we conduct a comprehensive comparison between SplitJoin and the baseline join algorithms within each database system.

6.3.1 DuckDB Comparison. Table 2 reports the runtime comparison between SplitJoin over DuckDB and the original DuckDB binary join, and Table 3 shows the maximum intermediate result sizes. Overall, all queries that can be executed by the original DuckDB (29 in total) can also be executed with SplitJoin. Moreover, with SplitJoin, DuckDB is able to complete 14 additional queries that the original binary join fails to finish due to timeouts or out-of-memory (43 vs. 29). For the queries that can finish in both cases, SplitJoin reduces runtime by 2.1 $\times$ on average (up to 13.6 $\times$) and cuts the maximum intermediate results by 7.9 $\times$ on average (up to 74 $\times$).

For datasets with high skew (i.e., WGPB, GPlus, and Topcats), SplitJoin achieves substantial performance gains. On WGPB and Topcats, DuckDB frequently fails due to timeouts or out-of-memory, while SplitJoin completes most queries successfully with significantly lower runtime. For GPlus, both approaches fail on the majority of queries, but for the completed query (Q1), SplitJoin can achieve a speedup of 1.38 $\times$ over the original DuckDB. These results highlight the effectiveness of our strategy in coping with highly skewed data distributions, which are prevalent in real-world social network datasets.

For the partially skewed dataset (i.e., Skitter), SplitJoin still demonstrates clear benefits. Because Skitter is relatively sparse and many joins have small thresholds, a large portion of splits can be skipped. Consequently, most of the queries degenerate to DuckDB’s original binary join plan. For the remaining finished queries (Q3, Q4), SplitJoin achieves improvements over the baseline in Q4; however, for Q3, the performance is slightly worse than DuckDB’s original plan. The slowdown primarily stems from the extra cost introduced by the split operator, which lengthens the execution pipeline and causes one of the subqueries to run slower than the baseline query, even though the total size of intermediate results is smaller than in the baseline.

For uniform datasets (i.e., Orkut and USPatent), the performance benefit largely depends on graph density. On USPatent, the gap between SplitJoin and the DuckDB baseline is relatively small,

since all joins in the query graph are lightweight, leading SplitJoin to consistently adopt the original binary join plans. In contrast, on Orkut, SplitJoin outperforms the baseline in most queries except Q7. The exception arises because Q7 involves two triangles, $(R_1 \bowtie R_2 \bowtie R_3) \bowtie (R_4 \bowtie R_5 \bowtie R_6)$, and the optimal strategy would be to compute each triangle independently and then join the intermediate results. In practice, however, our execution generates four subqueries that cannot fully reuse overlapping computations, thereby causing performance degradation on this query.

Taken together, these results confirm that SplitJoin provides consistent and often substantial runtime improvements, especially under skewed data distributions where traditional binary joins struggle the most.

6.3.2 Umbra Comparison. Table 4 presents the runtime comparison between SplitJoin and three join settings in Umbra. Table 5 reports the maximum intermediate result size when executing queries in Umbra. *Binary* indicates that Umbra is restricted to binary joins only, while *WCOJ* enforces the use of its built-in worst-case optimal join. *Default* refers to Umbra’s optimizer-chosen plan, which may combine binary joins, WCOJ, or a hybrid of the two. Overall, all queries that can be executed under Umbra’s binary join (35 in total) can also be executed with SplitJoin. Moreover, SplitJoin enables Umbra to finish 10 additional queries that the default setting cannot complete (45 vs. 35). For the queries that can run in both cases, SplitJoin achieves 1.3 $\times$ speedups on average (up to 6.1 $\times$) and reduces maximum intermediate results by 1.2 $\times$ on average (up to 2.1 $\times$).

In our analysis, we first focus on comparing SplitJoin with Umbra’s binary join (denoted as UmbraBJ below), as this aligns directly with the primary objective of our work.

Overall, the results are consistent with our observations on DuckDB: SplitJoin outperforms UmbraBJ in similar query cases. When SplitJoin underperforms the default binary join, the performance gap remains marginal and is significantly smaller than that of Umbra’s WCOJ (i.e., Umbra’s WCOJ often times out in cases where the default binary join successfully finishes), underscoring SplitJoin’s robustness and balanced efficiency across diverse workloads. For queries where SplitJoin takes longer time, Orkut Q7 shares the same reason discussed in the DuckDB experiments. The most interesting case is Skitter Q4—Umbra’s query plan, $(R_1 \bowtie R_2) \bowtie ((R_4 \bowtie R_5) \bowtie R_3)$, enables the reuse of intermediate results from $(R_1 \bowtie R_2)$ and $(R_4 \bowtie R_5)$, whereas SplitJoin currently cannot exploit such reuse. These two queries highlights a potential optimization direction for handling redundant computations during splitting. For the remaining slower queries (Orkut Q4 and Topcats Q11), the cause is similar to DuckDB’s Skitter Q3—the additional split operator increases pipeline length and introduces extra overhead, making one of the subqueries slower than the baseline plan. This suggests room for improvement in the scheduling mechanism when applying SplitJoin.

In summary, similar to the DuckDB results, the UmbraBJ experiments confirm that SplitJoin consistently delivers substantial runtime improvements. We now extend our analysis to examine Umbra’s built-in worst-case optimal join (WCOJ) and its default setting for a more comprehensive comparison.

		Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
WGPB	SplitJoin	3.824	29.879	25.095	16.074	59.876	19.824	19.824	OOM	TLE	OOM	48.775
	Default	<u>TLE</u>	59.230	OOM	<u>TLE</u>	<u>TLE</u>	OOM	<u>TLE</u>	OOM	<u>TLE</u>	OOM	666.440
Orkut	SplitJoin	15.076	OOM	OOM	57.86	56.548	101.028	567.008	128.968	TLE	OOM	OOM
	Default	19.200	OOM	OOM	OOM	OOM	<u>TLE</u>	515.240	OOM	OOM	OOM	OOM
GPlus	SplitJoin	27.156	OOM	OOM	<u>TLE</u>	<u>TLE</u>	OOM	OOM	OOM	OOM	OOM	<u>TLE</u>
	Default	34.770	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
USPatent	SplitJoin	<u>0.365</u>	<u>1.036</u>	<u>1.167</u>	<u>1.906</u>	<u>1.676</u>	<u>13.907</u>	<u>1.437</u>	<u>1.336</u>	<u>2.536</u>	<u>1.877</u>	<u>0.886</u>
	Default	<u>0.355</u>	<u>1.010</u>	<u>1.070</u>	<u>1.840</u>	<u>1.850</u>	<u>14.280</u>	<u>1.370</u>	<u>1.200</u>	<u>2.690</u>	<u>1.840</u>	<u>0.910</u>
Skitter	SplitJoin	<u>0.741</u>	<u>7.748</u>	4.732	2.753	4.632	<u>TLE</u>	<u>72.324</u>	7.094	<u>TLE</u>	OOM	29.716
	Default	<u>0.706</u>	<u>7.750</u>	4.220	4.820	4.360	<u>TLE</u>	<u>72.300</u>	6.950	<u>TLE</u>	OOM	<u>29.770</u>
Topcats	SplitJoin	4.525	24.056	22.917	7.976	9.207	11.836	47.456	671.069	<u>TLE</u>	OOM	433.366
	Default	7.800	30.660	248.680	OOM	OOM	<u>TLE</u>	384.250	OOM	OOM	OOM	803.250

Table 2: Runtime (s) in DuckDB. **TLE** indicates 900s time limit exceeded. **OOM** indicates out-of-memory. For comparison, the best performance is highlighted with a gray background, and underline indicates both systems use the same query plan.

		Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
WGPB	SplitJoin	559	559	1026	1026	2982	559	559	OOM	TLE	OOM	7345
	Default	<u>TLE</u>	936	OOM	<u>TLE</u>	<u>TLE</u>	OOM	<u>TLE</u>	OOM	<u>TLE</u>	OOM	26706
Orkut	SplitJoin	9450	OOM	OOM	11803	9450	16452	9450	9450	TLE	OOM	OOM
	Default	10913	OOM	OOM	OOM	OOM	<u>TLE</u>	57171	OOM	OOM	OOM	OOM
GPlus	SplitJoin	3286	OOM	OOM	<u>TLE</u>	<u>TLE</u>	OOM	OOM	OOM	OOM	OOM	<u>TLE</u>
	Default	6949	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM
USPatent	SplitJoin	<u>82</u>	<u>82</u>	<u>237</u>	<u>234</u>	<u>237</u>	<u>11528</u>	<u>179</u>	<u>80</u>	<u>234</u>	<u>194</u>	<u>194</u>
	Default	<u>82</u>	<u>82</u>	<u>237</u>	<u>234</u>	<u>237</u>	<u>11528</u>	<u>179</u>	<u>80</u>	<u>234</u>	<u>194</u>	<u>194</u>
Skitter	SplitJoin	<u>454</u>	<u>454</u>	454	496	<u>454</u>	<u>TLE</u>	<u>30693</u>	<u>454</u>	<u>TLE</u>	OOM	18774
	Default	<u>454</u>	<u>454</u>	454	454	<u>454</u>	<u>TLE</u>	<u>30693</u>	<u>454</u>	<u>TLE</u>	OOM	18774
Topcats	SplitJoin	1944	1944	1944	1944	1944	1994	1944	2371	<u>TLE</u>	OOM	199772
	Default	2663	2663	144725	OOM	OOM	<u>TLE</u>	144725	OOM	OOM	OOM	238984

Table 3: Max intermediate result size (in million tuples) in DuckDB. **TLE** indicates 900s time limit exceeded. **OOM** indicates out-of-memory. The best (smallest) value per dataset-query pair is highlighted with a gray background, and underline indicates both systems use the same query plan.

In general, Umbra’s built-in WCOJ occasionally outperforms SplitJoin, but is often slower than UmbraBJ. This suggests that while WCOJ can exploit specific join patterns effectively (e.g., Q1 triangle and Q6 4-clique), its benefits are not consistent across datasets and queries. For Umbra’s default setting, we observe that the optimizer does not always choose the optimal execution plan—for example, in Skitter Q6 and Topcats Q6, the selected plans result in suboptimal performance. Moreover, in Q7, which joins two triangles ($R_1 \bowtie R_2 \bowtie R_3$) $\bowtie$ ($R_4 \bowtie R_5 \bowtie R_6$), Umbra’s optimizer applies WCOJ within each triangle and then performs a binary join to combine the intermediate results. This design leverages the strengths of both strategies but also illustrates the inherent complexity and difficulty of hybrid plan selection.

Overall, while Umbra’s built-in WCOJ and its default optimizer can provide advantages in specific scenarios, SplitJoin can offer consistent and stable improvements on a wide range of queries and datasets. This highlights its effectiveness as a general-purpose strategy for accelerating join workloads.

6.4 Effectiveness Study

In this section, we use DuckDB to evaluate the effectiveness of each strategy proposed in SplitJoin.

6.4.1 Split Threshold. To evaluate the effectiveness of the split threshold proposed in Section 5.2, we conduct experiments on query Q1 (triangle query) over the GPlus and Topcats datasets, for the remaining datasets, the results are similar. Specifically, we fix the splitting attribute on a single table and vary the threshold to compare their impact on join execution time. Note that threshold=0 corresponds to the binary join plan.

In Figure 7, as the threshold increases from 0, the execution time and the maximum intermediates first decrease and then increase again. The selected thresholds in SplitJoin (highlighted with slashes) consistently achieve near-optimal performance across datasets.

6.4.2 Split Schedule. To evaluate the effectiveness of the proposed strategies, namely *co-split*, and *split-set-selection* strategies, we conduct experiments on three representative queries: Q1 (triangle), Q2

		Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
WGPB	SplitJoin	3.013	25.354	12.665	10.861	255	10.544	24.506	OOM	TLE	OOM	39.586
	Binary	5.43	156.777	TLE	OOM	OOM	OOM	OOM	OOM	OOM	TLE	TLE
	WCOJ	3.774	TLE	17.108	12.726	4.404	4.426	37.652	TLE	TLE	TLE	39.716
	Default	5.929	158.576	TLE	OOM	OOM	OOM	OOM	OOM	OOM	TLE	TLE
Orkut	SplitJoin	11.767	OOM	OOM	36.268	32.068	39.277	89.032	20.327	OOM	OOM	OOM
	Binary	13.182	OOM	OOM	30.719	36.294	TLE	58.998	OOM	OOM	OOM	OOM
	WCOJ	7.833	722.336	677.691	724.578	TLE	53.473	TLE	TLE	TLE	TLE	TLE
	Default	7.824	OOM	OOM	30.679	36.156	TLE	52.707	OOM	OOM	OOM	OOM
GPlus	SplitJoin	33.458	OOM	OOM	TLE	OOM	OOM	TLE	OOM	OOM	OOM	OOM
	Binary	41.520	OOM	OOM	OOM	OOM	OOM	TLE	OOM	OOM	OOM	OOM
	WCOJ	13.397	TLE	TLE	TLE	TLE	TLE	TLE	TLE	TLE	TLE	TLE
	Default	13.386	OOM	OOM	OOM	OOM	OOM	TLE	OOM	OOM	OOM	OOM
USPatent	SplitJoin	0.170	0.472	0.486	0.553	0.339	1.936	0.389	8.097	3.487	2.854	0.764
	Binary	0.172	0.503	0.477	0.489	0.346	1.941	0.378	7.947	3.591	2.818	0.721
	WCOJ	0.400	1.125	2.139	1.460	2.246	0.626	0.984	126.174	6.606	2.423	1.351
	Default	0.390	0.433	0.467	0.483	0.305	1.947	0.956	8.004	3.573	2.552	0.725
Skitter	SplitJoin	0.601	2.900	1.509	1.845	0.671	310	2.360	OOM	66.540	OOM	14.068
	Binary	0.425	2.865	1.992	0.664	0.676	305	2.226	OOM	70.859	OOM	14.087
	WCOJ	0.511	9.611	6.379	9.221	186.201	1.365	57.504	TLE	TLE	TLE	240.686
	Default	0.531	2.969	1.864	0.698	0.716	306	1.358	OOM	68.826	OOM	13.770
Topcats	SplitJoin	2.269	16.398	12.642	5.475	5.820	8.935	10.453	62.254	311	OOM	366
	Binary	7.499	16.886	15.466	10.782	6.650	10.592	12.431	OOM	OOM	OOM	250
	WCOJ	1.428	TLE	140.463	64.064	54.212	4.537	TLE	TLE	TLE	TLE	TLE
	Default	1.387	17.020	15.206	10.375	6.781	10.592	5.475	OOM	OOM	OOM	249.193

Table 4: Runtime (s) in Umbra. **TLE** indicates 900s time limit exceeded. **OOM** indicates out-of-memory. For comparison, the smaller (better) result between SplitJoin and Binary is highlighted with a gray background, and underline indicates SplitJoin and Binary are using the same plan. Bold indicates that Umbra’s built-in WCOJ used less time comparing to SplitJoin and Binary.

		Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
WGPB	SplitJoin	559	559	1026	1026	2982	559	559	OOM	TLE	OOM	7345
	Binary	936	936	TLE	OOM	OOM	OOM	OOM	OOM	OOM	TLE	TLE
Orkut	SplitJoin	9450	OOM	OOM	11803	9540	16452	9450	9450	OOM	OOM	OOM
	Binary	10913	OOM	OOM	12489	10913	TLE	10913	OOM	OOM	OOM	OOM
GPlus	SplitJoin	3286	OOM	OOM	TLE	OOM	OOM	TLE	OOM	OOM	OOM	OOM
	Binary	6949	OOM	OOM	OOM	OOM	OOM	TLE	OOM	OOM	OOM	OOM
USPatent	SplitJoin	82	82	362	239	82	1539	82	1381	1381	1381	362
	Binary	82	82	362	239	82	1539	82	1381	1381	1381	362
Skitter	SplitJoin	444	454	454	496	444	237177	444	OOM	3999	OOM	18786
	Binary	444	454	454	444	444	237177	444	OOM	3999	OOM	18786
Topcats	SplitJoin	1944	1944	1944	1944	1944	1944	1944	2371	74733	OOM	199772
	Binary	2663	2663	2663	2663	2663	2663	2663	OOM	OOM	OOM	238985

Table 5: Max intermediate result size(million tuples) in Umbra. **TLE** indicates 900s time limit exceeded. **OOM** indicates out-of-memory. The best (smallest) value per dataset-query pair is highlighted with a gray background, and underline indicates using the same plan.

(rectangle), and Q5 (diamond). The experiments are performed on the WGPB, Orkut, GPlus, and Topcats datasets. For the USPatent and Skitter datasets, splitting is not applied to these queries and thus they are excluded from this evaluation. We consider three configurations in total: (i) **config1**: only split on a single table each time, and in this case, we will split on the tables and attributes that config3 chooses, (ii) **config2**: *co-split* on a fixed split set, and

(iii) **config3**: config2 enhanced with *Split-Set-Selection* strategies mentioned in Section 5.3.

As shown in Table 6, our final configuration, config3, achieves better performance compared with both config1 and config2. When compared with config1, the improvement mainly comes from a reduction in the number of generated subqueries—config1 produces 4, 16, and 16 subqueries for Q1, Q2, and Q5, respectively, while config3 only produces 2, 4, and 4 subqueries. When compared with

Method	WGPB			Orkut			GPlus			Topcats		
	Q1	Q2	Q5	Q1	Q2	Q5	Q1	Q2	Q5	Q1	Q2	Q5
DuckDB Default	TLE	59.230	TLE	19.200	OOM	OOM	34.770	OOM	OOM	7.800	30.660	TLE
Single Split	8.262	74.667	107.836	17.756	OOM	181.599	46.750	OOM	OOM	6.795	70.766	36.136
Co-split	22.276	59.945	61.923	34.997	OOM	74.049	34.724	OOM	OOM	10.577	26.947	355
Co-split + Split-Set-Selection Strategies	3.824	29.879	59.876	15.076	OOM	56.548	27.156	OOM	TLE	4.525	24.056	9.207

Table 6: Runtime(s) effects on Q1, Q2, and Q5 across datasets. TLE indicates 900s time limit exceeded. OOM indicates out-of-memory.

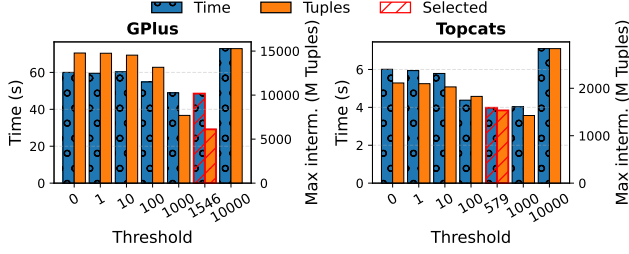
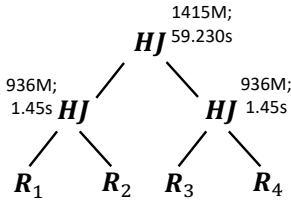


Figure 7: Threshold vs Time (s) in Q1



Max Intermediate Size = 936M; Total Time = 59.230s

Figure 8: Per-join performance breakdown of the most efficient plan on Topcats Q2. Labels in the form “x;y” indicate that a subtree outputs x million tuples and runs for y seconds.

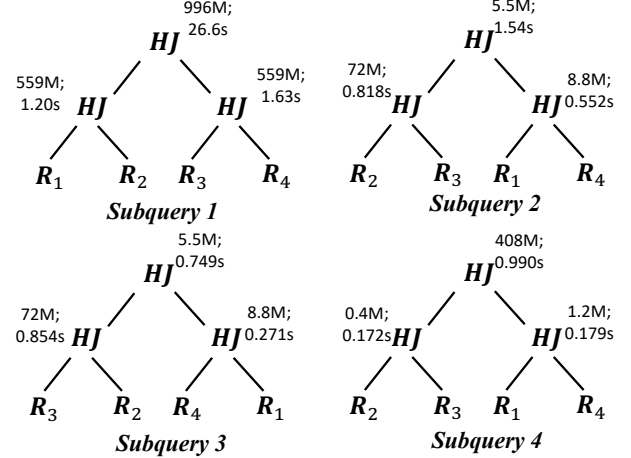
config2, the gain primarily stems from a smaller upper bound on intermediate results with appropriate co-split set selection, which helps reduce execution overhead. For example, in Q1, splitting on a lighter join rather than a larger one significantly reduces the size of intermediate results.

6.5 Case Study

To provide a deeper understanding of how our approach works, we present a case study on query Q2 in the WGPB dataset. Q2 in Figure 6 can be expressed as $R_1(X, Y) \bowtie R_2(Y, W) \bowtie R_4(X, Z) \bowtie R_3(Z, W)$ where the original query is split into four subqueries by splitting on $R_1 \bowtie R_2$ and $R_3 \bowtie R_4$.

We compare the baseline plan in DuckDB with the plan in DuckDB + SplitJoin, as it reveals the advantage of split-based execution: SplitJoin avoids generating large intermediates by isolating heavy keys, leading to faster overall runtime (i.e., overall 29.9s with SplitJoin vs. 59.2s without SplitJoin).

Figure 8 shows that the baseline plan produces a massive intermediate result when joining R_3 with R_4 and R_1 with R_2 (both of them expanding to more than 936 million tuples). This blowup significantly increases the number of hash probes in the hash table built by $(R_3 \bowtie R_4)$, and it also increases the time consumption on



Max Intermediate Size = 559M; Total Time = 29.879s

Figure 9: Per-join performance breakdown of SplitJoin on Topcats Q2. Labels in the form “x;y” indicate that a subtree outputs x million tuples and runs for y seconds.

building hash table on $(R_3 \bowtie R_4)$, ultimately becoming the bottleneck of the entire join plan, as confirmed by the time breakdown of each step.

In contrast, SplitJoin decomposes Q2 into four subqueries (Table 9), each with more balanced join sizes. The largest intermediate relation is reduced to about 559 million tuples, which makes the final join steps substantially more efficient and effectively alleviates the bottleneck observed in the baseline. As a result, SplitJoin achieves a more stable execution profile and improves overall performance for Q2. This case study highlights the performance bottlenecks of the baseline and demonstrates how SplitJoin avoids skewed joins and achieves better efficiency.

7 CONCLUSION

In this work, we presented SplitJoin, a lightweight framework that introduces the split operator to minimize intermediate join results and mitigate data skew. Implemented as a non-intrusive front-end layer, SplitJoin integrates seamlessly with existing systems and delivers substantial performance gains on real-world skewed workloads. This work takes a first step toward making split-based strategies practical, opening new opportunities for adaptive cost models and broader integration into modern query engines.

REFERENCES

- [1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4, Article 20 (Oct. 2017), 44 pages. <https://doi.org/10.1145/3129246>
- [2] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 1371–1382. <https://doi.org/10.1145/2723372.2742796>
- [3] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, Juan L. Reutter, Javier Rojas-Ledesma, and Adrián Soto. 2021. Worst-Case Optimal Graph Joins in Almost No Space. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) (SIGMOD '21). Association for Computing Machinery, New York, NY, USA, 102–114. <https://doi.org/10.1145/3448016.3457256>
- [4] Albert Atserias, Martin Grohe, and Daniel Marx. 2013. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.* 42, 4 (2013), 1737–1767.
- [5] Altan Birlir, Alfons Kemper, and Thomas Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *Proc. VLDB Endow.* 17, 11 (July 2024), 3215–3228. <https://doi.org/10.14778/3681954.3681995>
- [6] Nieves R. Brisaboa, Ana Cerdeira-Pena, Antonio Fariña, and Gonzalo Navarro. 2015. A Compact RDF Store Using Suffix Arrays. In *Proceedings of the 22nd International Symposium on String Processing and Information Retrieval - Volume 9309* (London, UK) (SPIRE 2015). Springer-Verlag, Berlin, Heidelberg, 103–115. https://doi.org/10.1007/978-3-319-23826-5_11
- [7] Shumo Chu, Magdalena Balazinska, and Dan Suciu. 2015. From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 63–78. <https://doi.org/10.1145/2723372.2750545>
- [8] Shaleen Deep, Xiao Hu, and Paraschos Koutris. 2020. Fast Join Project Query Evaluation using Matrix Multiplication. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) (SIGMOD '20). Association for Computing Machinery, New York, NY, USA, 1213–1223. <https://doi.org/10.1145/3318464.3380607>
- [9] Shaleen Deep, Hangdong Zhao, Austen Z. Fan, and Paraschos Koutris. 2024. Output-sensitive Conjunctive Query Evaluation. *Proc. ACM Manag. Data* 2, 5, Article 220 (Nov. 2024), 24 pages. <https://doi.org/10.1145/3695838>
- [10] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. <https://www.flux.utah.edu/paper/duplyakin-atc19>
- [11] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoglu. 2023. KÜZÜ- Graph Database Management System. In *Proceedings of the 13th Conference on Innovative Data Systems Research (CIDR '23)*. paper 48, 8 pages. <https://www.cidrdb.org/cidr2023/papers/p48-jin.pdf> CIDR 2023, Amsterdam, The Netherlands.
- [12] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.* 13, 12 (July 2020), 1891–1904. <https://doi.org/10.14778/3407790.3407797>
- [13] Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. 2019. Wikidata Graph Pattern Benchmark (WGPB) for RDF/SPARQL. Zenodo. <https://doi.org/10.5281/zenodo.4035223>
- [14] Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. 2019. A Worst-Case Optimal Join Algorithm for SPARQL. In *The Semantic Web – ISWC 2019*, Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtěch Svátek, Isabel Cruz, Aidan Hogan, Jie Song, Maxime Léfrancois, and Fabien Gandon (Eds.). Springer International Publishing, Cham, 258–275.
- [15] Xiao Hu. 2024. Fast Matrix Multiplication for Query Processing. *Proc. ACM Manag. Data* 2, 2, Article 98 (May 2024), 25 pages. <https://doi.org/10.1145/3651599>
- [16] Mahmoud Abo Khamis, Vasileios Nakos, Dan Olteanu, and Dan Suciu. 2024. Join Size Bounds using ℓ_p -Norms on Degree Sequences. *Proc. ACM Manag. Data* 2, 2 (2024), 96.
- [17] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2025. PANDA: Query Evaluation in Submodular Width. *TheoretCS* 4, 12 (2025), 1–42. <https://doi.org/10.46298/theoretcs.25.122>
- [18] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining* (Chicago, Illinois, USA) (KDD '05). Association for Computing Machinery, New York, NY, USA, 177–187. <https://doi.org/10.1145/1081870.1081893>
- [19] Julian McAuley and Jure Leskovec. 2012. Learning to discover social circles in ego networks. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1* (Lake Tahoe, Nevada) (NIPS'12). Curran Associates Inc., Red Hook, NY, USA, 539–547.
- [20] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.* 12, 11 (July 2019), 1692–1704. <https://doi.org/10.14778/3342263.3342643>
- [21] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement* (San Diego, California, USA) (IMC '07). Association for Computing Machinery, New York, NY, USA, 29–42. <https://doi.org/10.1145/1298306.1298311>
- [22] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case Optimal Join Algorithms. *J. ACM* 65, 3, Article 16 (March 2018), 16:1–16:40 pages. <https://doi.org/10.1145/3180143>
- [23] Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2013. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Rec.* 42, 4 (2013), 5–16. <https://doi.org/10.1145/2590989.2590991>
- [24] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science - Towards Embedded Analytics. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. <http://cidrdb.org/cidr2020/papers/p23-raasveldt-cidr20.pdf>
- [25] Semih Salihoglu. 2023. KÜZÜ: A Database Management System For "Beyond Relational" Workloads. *SIGMOD Rec.* 52, 3 (Nov. 2023), 39–40. <https://doi.org/10.1145/3631504.3631514>
- [26] Yufei Tao. 2020. A Simple Parallel Algorithm for Natural Joins on Binary Relations. In *23rd International Conference on Database Theory (ICDT 2020) (Leibniz International Proceedings in Informatics (LIPIcs))*, Carsten Lutz and Jean Christoph Jung (Eds.), Vol. 155. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:18. <https://doi.org/10.4230/LIPIcs.ICDT.2020.25>
- [27] Todd L. Veldhuizen. 2014. Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *Proceedings of the 17th International Conference on Database Theory (ICDT)*, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy (Eds.). OpenProceedings.org, Athens, Greece, 96–106. <https://doi.org/10.5441/002/icdt.2014.13>
- [28] Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. 2025. Yannakakis+: Practical Acyclic Query Evaluation with Theoretical Guarantees. *Proc. ACM Manag. Data* 3, 3, Article 235 (June 2025), 28 pages. <https://doi.org/10.1145/3725423>
- [29] Yisu Remy Wang, Max Willsey, and Dan Suciu. 2023. Free Join: Unifying Worst-Case Optimal and Traditional Joins. *Proc. ACM Manag. Data* 1, 2, Article 150 (June 2023), 23 pages. <https://doi.org/10.1145/3589295>
- [30] Yisu Remy Wang, Max Willsey, and Dan Suciu. 2023. Free Join: Unifying Worst-Case Optimal and Traditional Joins. *Proc. ACM Manag. Data* 1, 2, Article 150 (June 2023), 23 pages. <https://doi.org/10.1145/3589295>
- [31] Jiacheng Wu and Dan Suciu. 2025. HoneyComb: A Parallel Worst-Case Optimal Join on Multicores. In *SIGMOD '25: International Conference on Management of Data, Berlin, Germany, June 22 - 27, 2025*. ACM.
- [32] Yifei Yang and Xiangyao Yu. 2025. Accelerate Distributed Joins with Predicate Transfer. *Proc. ACM Manag. Data* 3, 3, Article 122 (June 2025), 27 pages. <https://doi.org/10.1145/3725259>
- [33] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. 2024. Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries. In *Proceedings of the 14th Annual Conference on Innovative Data Systems Research (CIDR '24)*. <https://www.cidrdb.org/cidr2024/papers/p22-yang.pdf>
- [34] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *VLDB*. IEEE Computer Society, 82–94.
- [35] Hao Yin, Austin R. Benson, Jure Leskovec, and David F. Gleich. 2017. Local Higher-Order Graph Clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Halifax, NS, Canada) (KDD '17). Association for Computing Machinery, New York, NY, USA, 555–564. <https://doi.org/10.1145/3097983.3098069>
- [36] Haozhe Zhang, Christoph Mayer, Mahmoud Abo Khamis, Dan Olteanu, and Dan Suciu. 2025. ℓ_p -Bound: Pessimistic Cardinality Estimation Using ℓ_p -Norms of Degree Sequences. *Proc. ACM Manag. Data* 3, 3, Article 184 (June 2025), 27 pages. <https://doi.org/10.1145/3725321>
- [37] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. 2025. Debunking the Myth of Join Ordering: Toward Robust SQL Analytics. *Proc. ACM Manag. Data* 3, 3, Article 146 (June 2025), 28 pages. <https://doi.org/10.1145/3725283>

A ANALYSIS OF ALGORITHM 3

A.1 Correctness

Throughout this section, we will assume that the query Q is connected, meaning there are no cartesian products. This limitation can easily be remedied by computing each connected sub-query individually and then taking the cartesian product.

THEOREM A.1. *Algorithm 3 terminates and returns the correct output.*

PROOF. We will show the following three statements.

- (1) The algorithm terminates
- (2) When the algorithm terminates, C has one element
- (3) The unique element in C is the correct output of the query

1) The loop on line 2 terminates because the number of iterations is bounded by the number of attributes. The loop on line 5 terminates because the number of possible light joins is bounded by the number of edges. The loop on line 8 terminates since the number of components in C is finite.

2) Suppose there are more than one component in C . One of these must have been discovered in the last iteration of the loop on line 2. Since the components were not merged, they cannot have any overlap. Since Q is connected, this means there is at least one attribute that is not in any component, because otherwise some components would have intersected. Some such attribute would have either caused another iteration of the loop on line 2, or would have been discovered by a light join from some component, so its existence is a contradiction.

3) We will show that every relation is joined into the output, which makes the output be correct. Let e be an arbitrary relation. Let x be the attribute on the light end of e . x will be integrated into some component at some point, by the same argument as in 2). This component would contain the relation e , since the algorithm exhausts light joins. Since all components are merged, the output will be a join that includes the relation e . $\square$

A.2 Bounding the Size of Intermediate Results

We will now show that Algorithm 3 matches the AGM bound. We remark that this works for join queries with binary relations, where all cardinalities are at most N . We will think of a query Q as its query graph (V, E) , which we described previously.

Definition A.2 (Fractional Vertex Packing). Let $Q = (V, E)$ be a natural join query. A fractional vertex packing u is an assignment to every vertex $x \in V$ in the graph a weight $u_x \in [0, 1]$ such that for every edge $\{x, y\}$, we have $u_x + u_y \leq 1$.

We denote by VP be the set of all possible vertex packings of Q .

THEOREM A.3 (AGM BOUND). *Let Q be a join query with binary relations. Let I be a database instance such that every relation has size at most N . Then*

$$|Q(I)| \leq \text{AGM}(Q) := \max_{u \in VP} N^{\sum_x u_x}$$

The maximum fractional vertex packing is the same as the minimum fractional edge cover, by duality of linear programs. Therefore, the the maximum $v \in VP$ has a total weight of ρ .

The goal of a worst-case optimal algorithm is to match this bound, i.e. make sure that no intermediate result has size more than $\text{AGM}(Q)$.

Definition A.4. Given a query $Q = (V, E)$, and a subset of $I \subseteq V$, define the subquery $Q_I = (I, E_I)$, where $E_I = \{\pi_I R \mid R \in E \text{ where } \exists x \in R \text{ s.t. } x \in I\}$. We will call subqueries on this form **induced sub-graph queries**.

The next lemma means that any intermediate relation that has the form of an induced subgraph query can never be too big. This is a key idea in the area of worst case optimal joins.

LEMMA A.5. *Let $Q = (V, E)$ be a query. Then, for any $I \subseteq V$*

$$\text{AGM}(Q_I) \leq \text{AGM}(Q)$$

PROOF. Consider any fractional vertex packing u_{Q_I} in Q_I . We can create a feasible fractional vertex packing u_Q for Q by taking u_{Q_I} , and extending it to Q by assigning a weight 0 to any vertex in $V - I$. These fractional vertex packings have the same weight. This means that the maximal fractional vertex packing in Q is at least as big as the maximal fractional vertex packing in Q_I . $\square$

LEMMA A.6. *Let $Q = (V, E)$ be a query where all cardinalities are at most N . Let I be an intermediate result from Q , that contains k_I unique attributes, that is obtained by starting from any relation and doing light joins. Then, $|I| \leq N^{k_I/2} \leq \text{AGM}(Q)$.*

PROOF. The first inequality follows from the fact that a light join that reaches a new attribute grows the intermediate size by at most a factor $\sqrt{N}$, and that we start with a single relation, whose size is at most $N^{2 \times 1/2}$. For the second inequality, let k be the number of vertices in Q . Clearly $N^{k_I/2} \leq N^{k/2}$. The inequality $N^{k/2} \leq \text{AGM}(Q)$ then follows from the fact that we can obtain a feasible vertex packing by assigning weight 1/2 to every vertex. $\square$

LEMMA A.7. *Let $Q = (V, E)$ be a join query and let I be a database instance. Let $U \subset V$. Suppose that*

- (1) *for every $Y \in V - U$ there exists a relation $R(X, Y) \in E$ for some $X \in U$*
- (2) *any relation $R(X, Y) \in E$ where $X \in U, Y \in V - U$ is light in X*

Let T be an intermediate relation obtained by computing Q_U and then joining with all relations $R(X, Y) \in E$ where $X \in U, Y \in V - U$. Then

$$|T| \leq \text{AGM}(Q)$$

PROOF. We will show that there exists an instance I' such that $|T| \leq Q(I')$. This proves the lemma by Theorem A.3. For some $R \in E$, denote by R_I and $R_{I'}$ the relation-instances in I and I' , respectively.

For any relation $R(X, Y)$ where $X, Y \in U$, let $R_{I'} = R_I$. For any $R(X, Y)$ where $X, Y \in V - U$, $R_{I'}$ is a product-relation $[\sqrt{N}] \times [\sqrt{N}]$. Finally, consider relations $R(X, Y)$ where $X \in U$ and $Y \in V - U$. For each $x \in \pi_X R_I$, add to $R_{I'}$ the tuples $(x, 1), \dots, (x, \deg_R(x))$.

The cardinality constraints are satisfied for I' , since for each R with at least 1 variable in U , each tuple in R_I adds one tuple in $R_{I'}$.

We now show $|T| \leq Q(I')$. Let $\Psi_I : \pi_U T \rightarrow \mathbb{Z}_{\geq 0}$, where $\Psi_I(t) = |\{t' \in T, \pi_U t' = t\}|$ (resp. let $\Psi_{I'} : \pi_U Q(I') \rightarrow \mathbb{Z}_{\geq 0}$, where $\Psi_{I'}(t) = |\{t' \in Q(I'), \pi_U t' = t\}|$). It is clear that the output sizes now can

be written as the sum $\sum_{t \in \pi_U T} \Psi_I(t)$ (resp. $\sum_{t \in \pi_U Q(I')} \Psi_{I'}(t)$). It remains to show that for all t , $\Psi_I(t) \leq \Psi_{I'}(t)$.

We claim the following

$$\Psi_{I'}(t) = \prod_{y \in V-U} \min_{R(x,y): x \in U} \deg_R(x)$$

This is because of how the relations are picked, all the relations going to some $y \in V - U$ cover the same values. Additionally, from the assumption (2), $\deg_R(x) \leq \sqrt{N}$, which means that the relations with both variables in $V - U$ do not do any filtering. Finally, by the assumption (1), there exists a light join reaching y , meaning the minimization above is well defined.

On the other hand, $\Psi_I(t)$ is upper bounded by the above. This is because these minimum degree relations are needed to cover any output tuple using t . The relations of larger degrees may then act as filters, reducing $\Psi_I(t)$. $\square$

We will finally prove the main theorem.

THEOREM A.8. *Let Q be a natural join query with binary relations of at most size N . Then, the above instantiation of the splitting framework runs in time at most $\mathcal{AGM}(Q) = O(N^\rho)$.*

PROOF. This instantiation creates 2^l subproblems, where l is the number of relations. For each subproblem, we have a join plan,

where every relation appears once. Hence, the number of binary joins performed only depends on the query. Using that a binary join can be computed in time $O(IN + OUT)$, it only remains to argue that all intermediate relations have size at most $\mathcal{AGM}(Q)$.

By Lemma A.6, the intermediate result obtained from light joins, i.e. between lines 3 and 5 in the algorithm, is bounded by AGM. Additionally, when the merging step is finished, at line 9, the intermediate result has the form of an induced subquery, which is bounded by AGM by Lemma A.5. It therefore remains to argue that the size is bounded by $\mathcal{AGM}(Q)$ also for intermediates obtained during any iteration of the loop on line 6.

Let V' be the set of attributes occurring in such an intermediate T , and let U be the set of attributes that were first included in any intermediate by T . Any attributes in $V' - U$ must have occurred in another intermediate relation that was created before T . T includes all relations in Q_U and additionally, any relation $R(X, Y)$ where $X \in U$, $Y \in V'$, where R is light in X . Furthermore, there cannot exist any $R'(X', Y')$ where $X' \in V' - U$, $Y' \in U$ that is light in X' , because then an earlier intermediate result would have had a light join to Y' .

Therefore, any such intermediate T has the form described in Lemma A.7, in the query $Q_{V'}$. The application of this Lemma finishes this proof. $\square$