



# TUM

TECHNISCHE UNIVERSITÄT MÜNCHEN  
INSTITUT FÜR INFORMATIK

## Combining Worst-Case Optimal and Traditional Binary Join Processing

Michael Freitag, Maximilian Bandle, Tobias  
Schmidt, Alfons Kemper, Thomas Neumann

TUM-I2082

# Combining Worst-Case Optimal and Traditional Binary Join Processing

Michael Freitag  
Technische Universität München  
freitagm@in.tum.de

Maximilian Bandle  
Technische Universität München  
bandle@in.tum.de

Tobias Schmidt  
Technische Universität München  
tobias.schmidt@in.tum.de

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

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

## ABSTRACT

Worst-case optimal join algorithms are attractive from a theoretical point of view, as they offer asymptotically better runtime than binary joins on certain types of queries. In particular, they avoid enumerating large intermediate results by processing multiple input relations in a single multi-way join. However, existing implementations incur a sizable overhead in practice, primarily since they rely on suitable ordered index structures on their input. Systems that support worst-case optimal joins often focus on a specific problem domain, such as read-only graph analytic queries, where extensive precomputation allows them to mask these costs.

In this paper, we present a comprehensive implementation approach for worst-case optimal joins that is practical within general-purpose relational database management systems supporting both hybrid transactional and analytical workloads. The key component of our approach is a novel hash-based worst-case optimal join algorithm that relies only on data structures that can be built efficiently during query execution. Furthermore, we implement a hybrid query optimizer that intelligently and transparently combines both binary and multi-way joins within the same query plan. We demonstrate that our approach far outperforms existing systems when worst-case optimal joins are beneficial while sacrificing no performance when they are not.

## 1 INTRODUCTION

The vast majority of traditional relational database management systems (RDBMS) relies on binary joins to process queries that involve more than one relation, since they are well-studied and straightforward to implement. Owing to decades of optimization and fine-tuning, they offer great flexibility and excellent performance on a wide range of workloads. Nevertheless, it is well-known that there are pathological cases in which any binary join plan exhibits suboptimal performance [10, 17, 28]. The main shortcoming of binary joins is the generation of intermediate results that can become much larger than the actual query result [44].

Unfortunately, this situation is generally unavoidable in complex analytical settings where joins between non-key attributes are commonplace. For instance, a conceivable query on the TPC<sub>H</sub> schema would be to look for parts within the same order that could have been delivered by the same supplier. Answering this query involves a self-join of `lineitem`

and two non-key joins between `lineitem` and `partsupp`, all of which generate large intermediate results [15]. Self-joins that incur this issue are also prevalent in graph analytic queries such as searching for triangle patterns within a graph [3]. On such queries, traditional RDBMS that employ binary join plans frequently exhibit disastrous performance or even fail to produce any result at all [2, 3, 46, 52].

Consequently, there has been a long-standing interest in *multi-way joins* that avoid enumerating any potentially exploding intermediate results [10, 17, 28]. Seminal theoretical advances recently enabled the development of *worst-case optimal* multi-way join algorithms which have runtime proportional to tight bounds on the worst-case size of the query result [9, 43, 44, 52]. As they can guarantee better asymptotic runtime complexity than binary join plans in the presence of growing intermediate results, they have the potential to greatly improve the robustness of relational database systems. However, existing implementations of worst-case optimal joins have several shortcomings which have impeded their adoption within such general-purpose systems so far.

First, they require suitable indexes on all permutations of attributes that can partake in a join which entails an enormous storage and maintenance overhead [3]. Second, a general-purpose RDBMS must support inserts and updates, whereas worst-case optimal systems like EmptyHeaded or LevelHeaded rely on specialized read-only indexes that require expensive precomputation [2, 3]. The LogicBlox system does support mutable data, but can be orders of magnitude slower than such read-optimized systems [3, 8]. Finally, multi-way joins are commonly much slower than binary joins if there are no growing intermediate results [40]. We thus argue that an implementation within a general-purpose RDBMS requires (1) an optimizer that only introduces a multi-way join if there is a tangible benefit in doing so, and (2) performant indexes structures that can be built efficiently *on-the-fly* and do not have to be persisted to disk.

In this paper, we present the first comprehensive approach for implementing worst-case optimal joins that satisfies these constraints. The first part of our proposal is a carefully engineered worst-case optimal join algorithm that is hash-based instead of comparison-based and thus does not require any precomputed ordered indexes. It relies on a novel hash trie data structure which organizes tuples in a trie based on the hash values of their key attributes. Crucially, this data

structure can be built efficiently in linear time and offers low-overhead constant-time lookup operations. As opposed to previous implementations, our join algorithm thus handles changing data transparently as any required data structures are built on-the-fly during query processing. The second part of our proposal is a heuristic extension to traditional cost-based query optimizers that intelligently generates hybrid query plans by utilizing the existing cardinality estimation framework. Finally, we implement our approach within the code-generating Umbra RDBMS developed by our group. This system constitutes the evolution of the high-performance in-memory database HyPer towards an SSD-based system [42]. Like HyPer, Umbra is explicitly designed for hybrid OLTP and OLAP (HTAP) workloads. Our experiments show that the proposed approach outperforms binary join plans and several systems employing worst-case optimal joins by up to two orders of magnitude on complex analytical workloads and graph pattern queries, without sacrificing any performance on the traditional TPC-H and JOB benchmarks where worst-case optimal joins are rarely beneficial.

The remainder of this paper is organized as follows. In Section 2 we present some background on worst-case optimal join algorithms. The hash trie index structure and associated multi-way join algorithm are described in detail in Section 3, and the hybrid query optimizer is presented in Section 4. Section 5 contains the experimental evaluation of our system, Section 6 gives an overview of related work, and conclusions are drawn in Section 7.

## 2 BACKGROUND

In the following section, we provide a brief overview of worst-case optimal joins and their key differences to traditional binary join plans. In the remainder of this paper, we consider natural join queries of the form

$$Q := R_1 \bowtie \dots \bowtie R_m, \quad (1)$$

where the  $R_j$  are relations with attributes  $v_1, \dots, v_n$ . Note that any inner join query containing only equality predicates can be transformed into this form by renaming attributes suitably. While most queries of this type can be processed efficiently by traditional binary join plans, query patterns such as joins on non-key attributes can lead to exploding intermediate results which pose a significant challenge to RDBMS which rely purely on binary join plans. Consider, for example, the query

$$Q_\Delta := R_1(v_1, v_2) \bowtie R_2(v_2, v_3) \bowtie R_3(v_3, v_1).$$

If we set  $R_1 = R_2 = R_3$  and view tuples as edges in a graph,  $Q_\Delta$  will contain all directed cycles of length 3, i.e. triangles in this graph (cf. Figure 1a). Any binary join plan for this query will first join two of these relations on a single attribute, which is equivalent to enumerating all directed paths of length 2 in the corresponding graph. This intermediate result will generally be much larger than the actual query result, since a graph with  $e$  edges contains on the order of  $O(e^2)$  paths of length 2 but only  $O(e^{1.5})$  triangles [51]. The resulting large

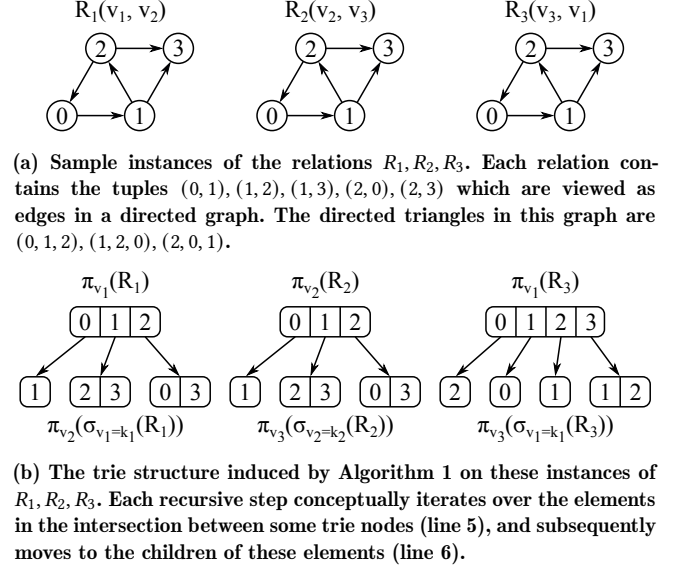


Figure 1: Algorithm 1 on the triangle query  $Q_\Delta$ .

amount of redundant work will severely impact the overall query processing performance.

Worst-case optimal join algorithms, on the other hand, avoid such exploding intermediate results [44]. Continuing our example, a worst-case optimal join conceptually performs a recursive backtracking search to find valid assignments of the join keys  $v_1$ ,  $v_2$ , and  $v_3$  before enumerating any result tuples. Specifically, we begin by iterating over the *distinct* values  $k_1$  of  $v_1$  that occur in both  $R_1$  and  $R_3$ , i.e.  $k_1 \in \{0, 1, 2\}$  in Figure 1a. For a given  $k_1$  we then recursively iterate over the distinct values  $k_2$  of  $v_2$  that occur in both  $R_2$  and the subset of  $R_1$  with  $v_1 = k_1$ , e.g.  $k_2 \in \{1\}$  for  $k_1 = 0$  in Figure 1a. Finally, we proceed analogously to find valid assignments  $k_3$  of  $v_3$ . Unlike a binary join plan, a worst-case optimal join avoids redundant intermediate work if a specific join key value occurs in multiple tuples, since only the distinct join key values need to be considered. Thus, as discussed in detail in our experimental evaluation (cf. Section 5), any relational join query in which a large fraction of tuples have multiple join partners can potentially benefit from worst-case optimal joins.

### 2.1 Worst-Case Optimal Join Algorithms

Formally, this paper builds on the generic worst-case optimal join algorithm shown in Algorithm 1 which directly implements the conceptual backtracking approach motivated above [44, 45]. It operates on the *query hypergraph*  $H_Q = (V, \mathcal{E})$  of a query  $Q$ , where the vertex set  $V$  contains the attributes  $\{v_1, \dots, v_n\}$  of  $Q$ , and the edge set  $\mathcal{E} = \{E_j \mid j = 1, \dots, m\}$  contains the attribute sets of the individual relations  $R_j$ . In case of our running example  $Q_\Delta$ , the query hypergraph is given by  $V = \{v_1, v_2, v_3\}$  and  $\mathcal{E} = \{E_1, E_2, E_3\}$  with  $E_1 = \{v_1, v_2\}$ ,  $E_2 = \{v_2, v_3\}$ ,  $E_3 = \{v_1, v_3\}$ .

**Algorithm 1:** Generic Worst-Case Optimal Join

---

**given** : A query hypergraph  $H_Q = (V, \mathcal{E})$  with attributes  $V = \{v_1, \dots, v_n\}$  and hyperedges  $\mathcal{E} = \{E_1, \dots, E_m\}$ .

**input** : The current attribute index  $i \in \{1, \dots, n+1\}$ , and a set of relations  $\mathcal{R} = \{R_1, \dots, R_m\}$ .

```

1 function enumerate( $i, \mathcal{R}$ )
2   if  $i \leq n$  then
3     // Relations participating in the current join
4      $\mathcal{R}_{join} \leftarrow \{R_j \in \mathcal{R} \mid v_i \in E_{R_j}\}$ ;
5     // Relations unaffected by the current join
6      $\mathcal{R}_{other} \leftarrow \{R_j \in \mathcal{R} \mid v_i \notin E_{R_j}\}$ ;
7     // Key values appearing in all joined relations
8     foreach  $k_i \in \bigcap_{R_j \in \mathcal{R}_{join}} \pi_{v_i}(R_j)$  do
9       // Select matching tuples
10       $\mathcal{R}_{next} \leftarrow \{\sigma_{v_i=k_i}(R_j) \mid R_j \in \mathcal{R}_{join}\}$ ;
11      // Recursively enumerate matching tuples
12      enumerate( $i+1, \mathcal{R}_{next} \cup \mathcal{R}_{other}$ );
13   else
14     // Produce result tuples
15     produce( $\bigtimes_{R_j \in \mathcal{R}_{next}} R_j$ );

```

---

Algorithm 1 consists of a recursive function which searches for valid assignments of a single join key  $v_i$  in each recursive step. The index  $i$  of the current join key is passed as a parameter to the algorithm. In later recursive steps (i.e.  $i > 1$ ), the backtracking nature of the algorithm entails that a specific assignment for the join keys  $v_1, \dots, v_{i-1}$  has already been selected in the previous recursive steps (see above). The second parameter  $\mathcal{R}$  consists of  $m$  separate sets, one for each input relation  $R_j$ , which contains all tuples from  $R_j$  that match this specific assignment of join key values. Initially,  $i$  is set to 1 and  $\mathcal{R}$  contains the full relations  $R_j$ .

Within a given recursive step  $i$ , the algorithm first determines which relations contain the join key  $v_i$  and thus have to be considered when searching for matching assignments of  $v_i$  (line 3). These relations are collected as separate elements in the set  $\mathcal{R}_{join}$ . Next, the algorithm iterates over all assignments  $k_i$  of  $v_i$  that appear in every one of these relations (line 5). In every iteration of this loop, the tuples that match the current assignment  $k_i$  of  $v_i$  are selected from the relations in  $\mathcal{R}_{join}$  (line 6) and the algorithm proceeds to the next recursive step (line 7). In the final recursive step (i.e.  $i = n+1$ ), the relations in  $\mathcal{R}$  contain only tuples that match one specific assignment of the join keys and are thus part of the query result (line 9).

When taking a closer look at a specific input relation  $R_j$ , we observe that the parameter  $\mathcal{R}$  of Algorithm 1 contains only tuples from  $R_j$  that share a common prefix of join key values. In case of the input relation  $R_1$  of the triangle query, for example,  $\mathcal{R}$  will contain the full relation  $R_1$  in the first recursive step, all tuples that match a specific value of  $v_1$  in the second step, and all tuples that match a specific value of

$(v_1, v_2)$  in the final step. Therefore, Algorithm 1 induces a trie structure on each input relation, as illustrated in Figure 1b [3]. The levels of this trie correspond to the join keys appearing in this relation, in the order in which they are processed by the join algorithm.

The theoretical foundation for the study of worst-case optimal join algorithms such as Algorithm 1 was laid down by Atserias, Grohe, and Marx, who derived a non-trivial and tight bound on the output size of  $Q$  that depends only on the size of the input relations  $R_j$  [9, 44, 45]. Given the query hypergraph  $H_Q$  of  $Q$  as defined above, we consider an arbitrary fractional edge cover  $\mathbf{x} = (x_1, \dots, x_m)$  of  $H_Q$  [45], which is defined by  $x_j > 0$  for all  $j \in \{1, \dots, m\}$  and  $\sum_{v_i \in E_j} x_j \geq 1$  for all  $v_i \in V$ . Then this bound states that

$$|Q| \leq \prod_{j=1}^m |R_j|^{x_j}, \quad (2)$$

and the worst-case output size of  $Q$  can be determined by minimizing the right-hand side of Inequality 2 [45]. A join algorithm for computing  $Q$  is defined to be *worst-case optimal* if its runtime is proportional to this worst-case output size [44, 45]. In case of our running example  $Q_\Delta$ , the right-hand side of Inequality 2 is minimal for the fractional edge cover  $\mathbf{x} = (0.5, 0.5, 0.5)$  which results in an upper bound of  $\sqrt{|R_1| \cdot |R_2| \cdot |R_3|}$  on the size of  $Q_\Delta$  [3, 45].

Central to the analysis of the runtime complexity of worst-case optimal joins is the *query decomposition lemma* proved by Ngo et al. [45]. For a given query hypergraph  $H_Q$  and a subset of join attributes  $U \subseteq V$ , we write  $\mathcal{E}_U := \{E_j \in \mathcal{E} \mid U \cap E_j \neq \emptyset\}$  to identify the set of all hyperedges that contain at least one of the join attributes in  $U$ . Then the query decomposition lemma can be stated as follows.

**LEMMA 2.1.** *Consider the query hypergraph  $H_Q = (V, \mathcal{E})$  describing the natural join query  $Q = R_1 \bowtie \dots \bowtie R_m$ . Let  $U \uplus W = V$  be an arbitrary partition of  $V$  with  $1 \leq |U| < |V|$  and  $L := \bowtie_{E_j \in \mathcal{E}_U} \pi_U(R_j)$ . Then*

$$\sum_{t \in L} \left( \prod_{E_j \in \mathcal{E}_W \cap \mathcal{E}_U} |R_j \bowtie t|^{x_j} \prod_{E_j \in \mathcal{E}_W \setminus \mathcal{E}_U} |R_j|^{x_j} \right) \leq \prod_{E_j \in \mathcal{E}} |R_j|^{x_j} \quad (3)$$

*holds for any fractional edge cover  $\mathbf{x} = (x_1, \dots, x_m)$  of  $H_Q$ .*

From their constructive proof of this lemma, they derive a generic worst-case optimal join algorithm that has runtime in  $O(nm \prod_{E_j \in \mathcal{E}} |R_j|^{x_j})$  for an arbitrary fractional edge cover  $\mathbf{x} = (x_1, \dots, x_m)$  of the query hypergraph [45]. Algorithm 1 as shown here can be obtained as a special case of this generic algorithm by setting  $U = \{v_i\}$  in Lemma 2.1. In particular, the set intersection in Algorithm 1 corresponds to the set  $L$ , and the runtime of the loop over this set intersection corresponds to the left-hand side of Inequality 3.

## 2.2 Implementation Challenges

Any implementation of Algorithm 1 has to rely on indexes that explicitly model the trie structure on the input relations in order to maintain the runtime complexity guarantees that



**Algorithm 2:** Hash Trie Join Build Phase

---

**given:** A hyperedge  $E_j \in \mathcal{E}$  and hash functions  $h_i$  for the join attributes  $v_i \in V$ .

**input:** The global index  $i \in \{1, \dots, n+1\}$  of the currently processed attribute  $v_i \in E_j$  and a linked list  $L$  of tuples.

```

1 function build( $i, L$ )
2   if  $i \leq n$  then
3     // Allocate hash table memory
4      $M \leftarrow \text{allocateHashtable}(2^{\lceil \log_2(1.25 \cdot |L|) \rceil})$ ;
5     // Build outer hash table
6     foreach tuple  $t$  in  $L$  do
7       pop  $t$  from  $L$ ;
8        $B \leftarrow \text{lookupBucket}(M, h_i(\pi_{v_i}(t)))$ ;
9       push  $t$  onto the linked list stored in  $B$ ;
10    // Build nested hash tables
11     $i_{next} \leftarrow$  index of the next attribute in  $E_j$ ;
12    foreach populated bucket  $B$  in  $M$  do
13       $L_{next} \leftarrow$  extract linked list stored in  $B$ ;
14       $M_{next} \leftarrow \text{build}(i_{next}, L_{next})$ ;
15      store  $M_{next}$  in  $B$ ;
16  return( $M$ );
17 else
18  // All attributes in  $E_j$  have been processed
19  return( $L$ );

```

---

Algorithm 1). The amount of redundant work introduced by this relaxation will generally be negligible since hash collisions are extremely rare in any decent hash function like AquaHash or MurmurHash [7, 50].

These modifications allow for a much more efficient implementation of the nested hash table structure, since no information about the actual key values is required. Thus, all hash tables share a uniform compact memory layout, and both set intersections and lookup operations can be computed without any type-specific logic by only relying on fast integer comparisons. Moreover, the modified version of Algorithm 1 does not require any actual key comparisons for tentative matches that are later rejected.

### 3.2 Join Algorithm Description

The proposed join processing approach can be split into clearly separated *build* and *probe* phases. In the build phase the input relations are materialized and the corresponding hash tries are created. In the subsequent probe phase, the worst-case optimal hash trie join algorithm utilizes these index structures to enumerate the join result.

**3.2.1 Hash Tries.** As outlined above, a hash trie represents a prefix tree on the hashed join attribute values of a relation, where the join attributes and their order are determined by a given query hypergraph. Thus, we assume in the following that there is a hash function  $h_i$  for each join attribute  $v_i$

**Table 1: The trie iterator interface used in the probe phase of our hash trie join algorithm (cf. Algorithm 3). An iterator points to a specific bucket within one of the nodes of a hash trie, and the interface functions allow navigation within the trie.**

function	description
<b>up</b>	Move the iterator to the parent bucket of the current node.
<b>down</b>	Move the iterator to the first bucket in the child node of the current bucket.
<b>next</b>	Move the iterator to next occupied bucket within the current node. Return <b>false</b> if no further occupied buckets exist.
<b>lookup</b>	Move the iterator to the bucket with specified hash. Return <b>false</b> if no such bucket exists.
<b>hash</b>	Return the hash value of the current bucket.
<b>size</b>	Return the size of the current node.
<b>tuples</b>	Return the current tuple chain (only possible after calling <b>down</b> on the last trie level).

which maps the values of  $v_i$  to some integer domain. A node within a hash trie consists of a single hash table which maps these hash values to child pointers. These point to nodes on the next trie level in case of inner nodes, and to the actual tuples associated with a full prefix in case of leaf nodes. Within a leaf node, these tuples are stored in a linked list. For example, Figure 2 illustrates a possible hash trie on the relation  $R_1(v_1, v_2)$  shown in Figure 1, containing the tuples  $(0, 1), (1, 2), (1, 3), (2, 0), (2, 3)$ . Its root hash table contains the distinct *hash values* of  $v_1$ , i.e.  $h_1(0), h_1(1)$ , and  $h_1(2)$ . The child hash table of the entry for  $h_1(1)$ , for instance, then contains the distinct *hash values* of  $v_2$  that occur in tuples with  $h_1(v_1) = h_1(1)$ , i.e.  $h_2(2)$  and  $h_2(3)$ .

**3.2.2 Build Phase.** In the build phase, such a hash trie data structure is built on each input relation  $R_j$  of the join query  $Q$ . For a given relation  $R_j$ , we first materialize all tuples in  $R_j$  in a linked list. Subsequently, this linked list is passed to Algorithm 2 which recursively constructs the hash tables comprising the hash trie from top to bottom. Its inputs are the global index  $i$  of the join attribute on which to build a hash table, and a linked list  $L$  of tuples. The algorithm first allocates space for the hash table, where the number of buckets is chosen as the next power of two larger than some fixed multiple of the number of tuples in  $L$  (line 3). Subsequently, the tuples in  $L$  are inserted into the hash table based on the hash value of the current join attribute  $v_i$ . Tuples that fall into the same bucket are collected in a linked list stored in that bucket (lines 4–7). Finally, the hash tables on the next join key attribute are built by calling Algorithm 2 recursively on these linked lists (lines 8–12). In the base case (line 15), the linked list  $L$  itself is returned unchanged as the leaf node.

**Algorithm 3:** Hash Trie Join Probe Phase

---

**given** : A query hypergraph and hash tries on the input relations with iterators  $\mathcal{I} = \{I_1, \dots, I_m\}$ .  
**input** : The current attribute index  $i \in \{1, \dots, n+1\}$ .

```

1 function enumerate( $i$ )
2   if  $i \leq n$  then
3     // Select participating iterators
4      $\mathcal{I}_{join} \leftarrow \{I_j \in \mathcal{I} \mid v_i \in E_j\}$ ;
5      $\mathcal{I}_{other} \leftarrow \{I_j \in \mathcal{I} \mid v_i \notin E_j\}$ ;
6     // Select smallest hash table
7      $I_{scan} \leftarrow \arg \min_{I_j \in \mathcal{I}_{join}} \mathbf{size}(I_j)$ ;
8     // Iterate over hashes in smallest hash table
9     repeat
10    // Find hash in remaining hash tables
11    foreach  $I_j \in \mathcal{I}_{join} \setminus \{I_{scan}\}$  do
12    | if not lookup( $I_j, \mathbf{hash}(I_{scan})$ ) then
13    | | skip current iteration of outer loop;
14    // Move to the next trie level
15    foreach  $I_j \in \mathcal{I}_{join}$  do
16    | | down( $I_j$ )
17    // Recursively enumerate matching tuples
18    enumerate( $i+1$ );
19    // Move back to the current trie level
20    foreach  $I_j \in \mathcal{I}_{join}$  do
21    | | up( $I_j$ )
22    until not next( $I_{scan}$ );
23  else
24    // All iterators now point to tuple chains
25    foreach  $\mathbf{t} \in \times_{I_j \in \mathcal{I}_{join}} \mathbf{tuples}(I_j)$  do
26    | if join condition holds for  $\mathbf{t}$  then
27    | | produce( $\mathbf{t}$ );

```

---

**3.2.3 Probe Phase.** The probe phase is responsible for actually enumerating the tuples in the join result of a query. As outlined above, we modify the generic multi-way join algorithm shown in Algorithm 1 to defer key comparisons and make use of the hash trie data structures created in the build phase. Our implementation accesses hash tries through *iterators*. A hash trie iterator points to a specific bucket within one of the nodes of a hash trie, and thus identifies a unique prefix stored within this trie. Iterators can be moved through a set of well-defined interface functions which are shown in Table 1. These functions allow horizontal navigation within the buckets of a given node (**next**, **lookup**), and vertical navigation between different nodes of the hash trie (**up**, **down**). Crucially, all functions can be implemented with amortized constant time complexity as they directly map to elementary operations on the underlying hash tables.

The resulting worst-case optimal hash trie join algorithm is shown in Algorithm 3. It exclusively interacts with iterators  $I_j \in \mathcal{I}$  on the hash tries corresponding to the input

relations  $R_j \in \mathcal{R}$ . In preparation for our subsequent analysis, we introduce some additional notation. As outlined above, an iterator  $I_j \in \mathcal{I}$  always points to a specific node within a hash trie. We write  $H(I_j)$  to identify the set of hashed join keys that is represented by this node and its children, and  $R(I_j)$  to identify the multiset of tuples stored in the leaves of the subtree rooted in this node. In the following, we will view  $H(I_j)$  as a relation with the same attribute names as the corresponding  $R(I_j)$ .

From a high-level point of view Algorithm 3 operates in exactly the same way as the generic algorithm shown in Algorithm 1, with the key difference that it initially enumerates all tuples for which the *hash values* of the join keys match. In particular, the loop in lines 6–15 iterates over the elements  $k_i$  in the set intersection  $\bigcap_{I_j \in \mathcal{I}_{join}} \pi_{v_i}(H(I_j))$ , and invoking **down** on the participating iterators is equivalent to computing  $\sigma_{v_i=k_i}(H(I_j))$  for  $I_j \in \mathcal{I}_{join}$ . Any false positives arising due to hash collisions are filtered by a final check just before passing the tuples to the output consumer of the multi-way join operator (line 18).

**3.2.4 Complexity Analysis.** In the following, we present a formal investigation of the time and space complexity of the proposed hash trie join approach, proving in particular that its runtime is indeed worst-case optimal.

**THEOREM 3.1.** *The build phase of the proposed approach has time and space complexity in*

$$O\left(n \cdot \sum_{E_j \in \mathcal{E}} |R_j|\right). \quad (4)$$

**PROOF.** As outlined above, the same operations are performed for each input relation  $R_j$  during the build phase, hence we focus on a given  $R_j$  in the following. The initial materialization of  $R_j$  in a linked list clearly requires time and space proportional to  $|R_j|$ . Moving on to Algorithm 2, we note that each tuple in the input linked list  $L$  is moved to exactly one of the linked lists that are processed recursively. That is, no additional space is required for tuple storage, and the overall set of tuples that is processed in each recursive step of Algorithm 2 is some partition of  $R_j$ . As there are at most  $n$  join attributes in a relation, we obtain a total time and space complexity of  $O(n \cdot |R_j|)$  for the build phase of a single relation  $R_j$ .  $\square$

Before delving into the runtime analysis of Algorithm 3, it is important to recall that we intend to integrate this algorithm into a general-purpose RDBMS, and thus have to adhere to the bag semantics imposed by the SQL query language. However, both the theoretical groundwork on worst-case optimal join processing as well as existing implementations only consider the case of set semantics as they are used by the Datalog query language, for example [3, 44]. We thus pursue the following line of reasoning. In the first step, we formally prove that Algorithm 3 is worst-case optimal under set semantics, where exactly one tuple is associated with

each distinct join key in the input relations  $R_j$ . Note, however, that  $R(I_j)$  as defined above can still be a multiset even under set semantics due to the possibility of hash collisions. Subsequently, we informally motivate how this worst-case optimality under set semantics translates to bag semantics.

**THEOREM 3.2.** *Consider the query hypergraph  $H_Q = (V, \mathcal{E})$  describing the natural join query  $Q = R_1 \bowtie \dots \bowtie R_m$ . Let  $\mathbf{x} = (x_1, \dots, x_m)$  be an arbitrary fractional edge cover of  $H_Q$ , and let  $\mathcal{I} = \{I_1, \dots, I_m\}$  be iterators pointing to the root nodes of hash tries on the relations  $R_j$ . Then the time complexity of Algorithm 3 is in  $O(nm \prod_{E_j \in \mathcal{E}} |H(I_j)|^{x_j})$  and its space complexity is in  $O(nm)$ .*

**PROOF.** We begin by proving the time complexity of Algorithm 3 by induction over its recursive steps  $i$ . Our approach is based on the assumption that good hash functions are used, in the sense that collisions occur only very rarely. As we impose set semantics for the purposes of this proof, we can formalize this assumption as

$$|H(I_j)| \in \Theta(|R(I_j)|) \quad (5)$$

for any hash trie iterator  $I_j$ . This formalization encompasses the intuitive formulation that hash collisions occur with a fixed small probability.

In the base case  $i = n + 1$  all hash trie iterators point to leaf nodes, i.e. by construction  $|H(I_j)| = 1$  for all iterators  $I_j \in \mathcal{I}$ . Under Assumption 5, this yields  $|R(I_j)| \in O(1)$  and thus the cross product of the  $R(I_j)$  enumerated in lines 17–19 contains  $O(1)$  elements. Actually constructing the candidate result tuple  $\mathbf{t}$  and checking the join condition on  $\mathbf{t}$  can then easily be done in  $O(nm)$  time which yields an overall runtime of  $O(nm) = O(nm \prod_{E_j \in \mathcal{E}} |H(I_j)|^{x_j})$  for the base case.

In the inductive case  $1 \leq i \leq n$  we will apply Lemma 2.1 to the sets  $H(I_j)$ . As outlined above, the loop in lines 6–15 iterates over the elements in

$$\begin{aligned} L &:= \bigcap_{I_j \in \mathcal{I}_{join}} \pi_{v_i}(H(I_j)) \\ &= \bigcap_{E_j \in \mathcal{E}_U} \pi_U(H(I_j)) \\ &= \bowtie_{E_j \in \mathcal{E}_U} \pi_U(H(I_j)) \end{aligned} \quad (6)$$

for  $U = \{v_i\}$ . By construction this set intersection is computed in time proportional to  $\text{size}(I_{scan})$  (cf. line 5), i.e. proportional to

$$\begin{aligned} |\mathcal{E}_U| \min_{E_j \in \mathcal{E}_U} |\pi_U(H(I_j))| &\leq m \left( \min_{E_j \in \mathcal{E}_U} |H(I_j)| \right)^{\sum_{E_j \in \mathcal{E}} x_j} \\ &= m \prod_{E_j \in \mathcal{E}} \left( \min_{E_j \in \mathcal{E}_U} |H(I_j)| \right)^{x_j} \\ &\leq m \prod_{E_j \in \mathcal{E}} |H(I_j)|^{x_j} \end{aligned} \quad (7)$$

since  $|H(I_j)| \geq 1$  and  $x_j > 0$ .

If a given iteration of the loop is not skipped in line 8, each iterator in  $\mathcal{I}_{join}$  points to the bucket containing a specific hash value  $k_i \in L$ . In the following, we will view these hash values

as tuples  $\mathbf{t}$  with the single attribute  $v_i$ . After invoking **down** on these iterators in lines 10–11, we have thus restricted the set of hashed join keys  $H(I_j)$  associated with these iterators to

$$\sigma_{v_i=k_i}(H(I_j)) = H(I_j) \bowtie \mathbf{t}. \quad (8)$$

Applying the inductive hypothesis then yields that the runtime of the recursive call in line 12 is proportional to

$$nm \prod_{E_j \in \mathcal{E}_U} |H(I_j) \bowtie \mathbf{t}|^{x_j} \prod_{E_j \in \mathcal{E} \setminus \mathcal{E}_U} |H(I_j)|^{x_j}. \quad (9)$$

Let  $W := V \setminus U = V \setminus \{v_i\}$ , and note that hyperedges  $E_j \in \mathcal{E}_U \setminus \mathcal{E}_W$  contain only the join key  $v_i$ . Thus,  $|H_j \bowtie \mathbf{t}| = 1$  for  $E_j \in \mathcal{E}_U \setminus \mathcal{E}_W$ . Moreover, one can easily verify that  $\mathcal{E} \setminus \mathcal{E}_U = \mathcal{E}_W \setminus \mathcal{E}_U$  since there are no empty hyperedges  $E_j$ . Thus, the runtime of the recursive call shown in (9) is equivalent to

$$nm \prod_{E_j \in \mathcal{E}_W \cap \mathcal{E}_U} |H(I_j) \bowtie \mathbf{t}|^{x_j} \prod_{E_j \in \mathcal{E}_W \setminus \mathcal{E}_U} |H(I_j)|^{x_j}. \quad (10)$$

Moreover, the loops invoking **down** and **up** on the iterators in lines 10–11 and 13–14 each have runtime in  $O(m)$ . In conjunction with (10), this allows us to state the overall runtime of the loop in lines 6–15 as proportional to

$$\sum_{\mathbf{t} \in L} \left( 2m + nm \prod_{E_j \in \mathcal{E}_W \cap \mathcal{E}_U} |H(I_j) \bowtie \mathbf{t}|^{x_j} \prod_{E_j \in \mathcal{E}_W \setminus \mathcal{E}_U} |H(I_j)|^{x_j} \right)$$

which is clearly bounded by

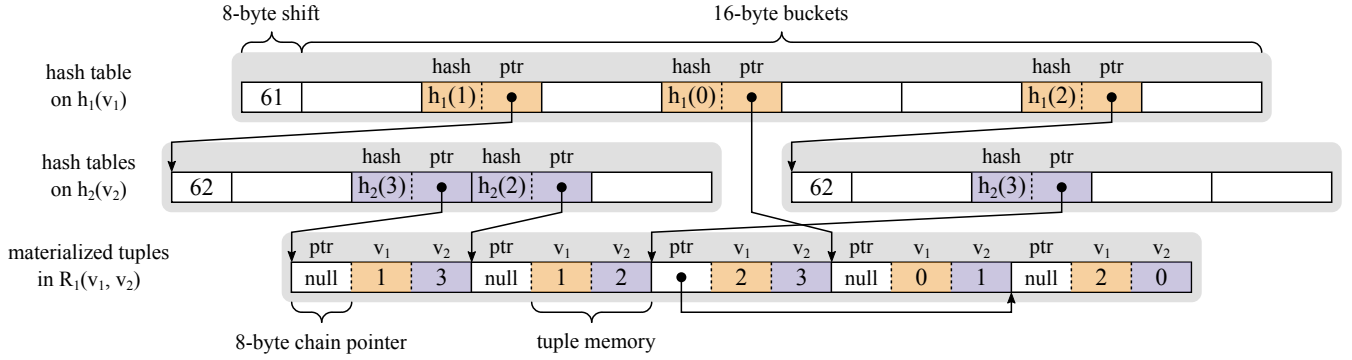
$$3nm \sum_{\mathbf{t} \in L} \left( \prod_{E_j \in \mathcal{E}_W \cap \mathcal{E}_U} |H(I_j) \bowtie \mathbf{t}|^{x_j} \prod_{E_j \in \mathcal{E}_W \setminus \mathcal{E}_U} |H(I_j)|^{x_j} \right). \quad (11)$$

Hence, the prerequisites for Lemma 2.1 are satisfied by (6) and (11), and we conclude that the runtime of this loop is in  $O(nm \prod_{E_j \in \mathcal{E}} |H(I_j)|^{x_j})$ . In combination with (7) this yields the desired time complexity for Algorithm 3.

Finally, we observe that the hash trie iterators and interface functions required by Algorithm 3 can easily be implemented using  $O(nm)$  additional space, as each iterator only needs to store the path to the current bucket.  $\square$

Taking into account that  $|H(I_j)| \leq |R(I_j)|$  clearly holds, Theorem 3.2 yields that the runtime of Algorithm 3 is indeed worst-case optimal under set semantics. Concluding our analysis, we note that under bag semantics, we can view the algorithm as performing a worst-case optimal join on the *set* of join key values, before expanding the *bag* of tuples corresponding to the join keys that are part of the join result. By construction (cf. Algorithm 2), the inner nodes of a hash trie store only the *distinct* join attribute hash values present in the respective input relation, and consequently only the leaf nodes are affected when multiple tuples can be associated with a single join key. Such duplicated tuples are simply stored in the linked list associated with the respective leaf node and enumerated as part of the cross product between tuple chains in the base case of Algorithm 3 (line 17). Crucially, this expansion occurs *after* Algorithm 3 has determined that all tuples in this cross product are part of the





**Figure 3: Memory layout of the hash trie in Figure 2. The gray boxes correspond to the individual hash tables and materialized input tuples. No nested hash table is built for the tuple (0, 1) due to singleton pruning.**

join result, except of course for potential false positives due to hash collisions.

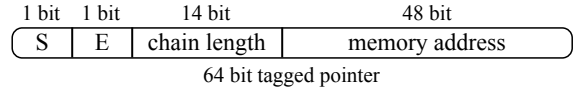
### 3.3 Implementation Details

In the following, we provide implementation details of the proposed approach, and a detailed account of its integration into a compiling query execution engine [41].

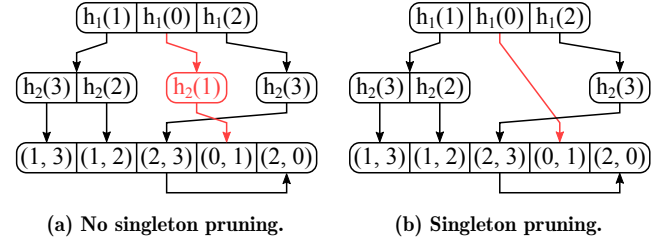
**3.3.1 Hash Trie Implementation.** Figure 3 shows the memory layout of a hash trie as it is implemented within the Umbra system [42]. We assume that the size of a hash value is 64 bits, which is sufficient even for very large data sets. As outlined above, the size of hash tables is restricted to powers of two, as this allows us to compute the bucket index for a given hash value using a fast bitwise shift instead of a slow modulo operation. Specifically, for a hash table size of  $2^p$  and a 64-bit hash value, the bucket index is computed by shifting the hash value  $64 - p$  bits to the right. Each hash table contiguously stores this shift value, i.e.  $64 - p$ , as a single 8-byte integer followed by an array of  $2^p$  16-byte buckets.

The first 8 bytes of each bucket contain the full hash value that is stored in the bucket, which is required as we use linear probing to resolve collisions within the bucket array. In comparison to other collision resolution schemes such as chaining, linear probing has the advantage that all distinct hash values are stored separately in the hash table. This allows us to store the associated child pointer directly within the remaining 8 bytes of a bucket, which would otherwise require at least one further level of indirection. The upper 16 bits of child pointers are unused on prevalent 64-bit architectures, and we encode additional information about the target of the pointer in these bits (cf. Figure 4). This plays a central role in the two main optimizations of hash tries, namely *singleton pruning* and *lazy child expansion*.

Singleton pruning is based on the observation that the size of hash tables tends to decrease drastically in the lower levels of the trie. In particular, we observed that inner nodes quite frequently represent a prefix that occurs only in a single tuple. Such singleton nodes and their descendants form a path on which each node has exactly one child, and we represent



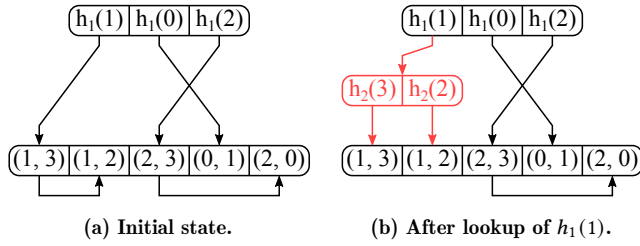
**Figure 4: Structure of tagged child node pointers. The S bit is set if the child node is a singleton tuple, and the E bit indicates whether the child node has already been expanded.**



**Figure 5: Illustration of singleton pruning. Any sub-trie that represents only a single tuple (shown red in (a)) is represented by a direct pointer to the corresponding tuple (shown red in (b)).**

such paths by a direct pointer to the corresponding singleton tuple (cf. Figure 5). Single-entry paths that are associated with multiple tuples are not pruned, as we cannot cheaply detect this case without actually building the corresponding hash tables (e.g.  $h(3)$  in Figure 5). The upper bit of a child pointer is used to distinguish between regular child pointers and singleton child pointers (cf. Figure 4). We do not apply singleton pruning to the root node, as this simplifies our code and we expect the root node to contain more than one tuple anyway.

Lazy child expansion exploits that Algorithm 3 computes the intersection of multiple hash tables before actually accessing any children thereof. Depending on the selectivity of this intersection operation, many inner nodes of the hash trie are never accessed. In order to avoid the overhead of unnecessarily creating nodes, we lazily expand child nodes when they are accessed for the first time. During the build



**Figure 6: Illustration of lazy child expansion. Initially, only the root hash table is built (a), and nested hash tables are built lazily when required (b).**

phase, only the root node is eagerly created by Algorithm 2, as it is usually accessed at least once by the join algorithm. Any recursive calls to Algorithm 2 are then deferred to the probe phase, until the corresponding child node is actually accessed. When a node is first created, all tuples that fall into a given hash bucket are collected in a linked list (cf. Algorithm 2), and a pointer to the head of this list is stored in the corresponding child pointer (cf. Figure 6). The second bit of the child pointer is then used to indicate whether the corresponding child node has already been expanded (cf. Figure 4). Upon the first access to this bucket, the tuple chain is scanned and the respective child node is built by executing the respective deferred recursive call of Algorithm 2.

The number of tuples in the chain needs to be known to choose a correct size for this child hash table. In order to avoid having to scan the tuple chain twice, we use the remaining 14 unused bits of the child pointer to track the length of the tuple chain while building the parent hash table (cf. Figure 4). Of course, we can only store chain lengths up to a certain limit this way. Specifically, the child pointer can be used to store chain lengths up to  $2^{14} - 2 = 16382$ , and the value  $2^{14} - 1$  is used as a sentinel to indicate that an overflow occurred. In the latter case, we resort to scanning the tuple chain twice in order to determine its length. Fortunately, most hash tables in the lower levels of a hash trie are small, so such long chains are only encountered very rarely.

For this reason we simply expand a child node in the first thread that accesses it. This thread atomically replaces the child pointer with a sentinel value that cannot occur during regular operation ( $2^{63} - 1$ ). Before following a child pointer, threads first check for this sentinel value and spin until the value becomes valid. We could also allow multiple threads to collaboratively expand child nodes, but our experiments show that the simple approach implemented in our system works fine in the vast majority of cases. We thus leave the exploration of alternative approaches to future work.

Our implementation of hash tries makes heavy use of pointer tagging which calls for a brief discussion of the portability of our approach. Most importantly, we note that we can maintain the overall asymptotic complexity guarantees of the data structure even without pointer tagging. We merely use it to optimize for cache performance by reducing the size of the hash buckets. If the upper 16 bits of pointers are not

available to encode additional information, we can simply store this information in an additional data field within the hash buckets.

**3.3.2 Build Phase.** As outlined in Section 3.2.2 the incoming tuples within a given input pipeline are conceptually placed in a linked list as part of the build phase. In our implementation, we materialize these incoming tuples contiguously in an in-memory buffer. They are stored using a fixed-length memory layout that is determined during query compilation time, in order to facilitate subsequent random tuple accesses. In case of variable-length data, this is achieved by materializing a fixed-length metadata entry containing a pointer to the actual variable-length data [42]. In addition to the actual tuple data, we reserve an additional 8 bytes of memory per tuple which is used later to store the tuple chain pointer required by the linked lists (cf. Figure 3). Note that we do not materialize the hash values of the join key attributes at this point, but generate functions to compute these hash values from the materialized tuple data on-demand. This reduces the storage overhead per tuple as the hash values are stored as part of the hash trie anyway.

As part of the materialization step, the tuples are partitioned based on the hash values of the first join key attribute. This ensures that tuples with similar join key hash values reside in physically close memory locations which is critical to achieve acceptable cache performance during the remainder of the build and probe phases. For this purpose, we adapt the two-pass radix partitioning scheme proposed by Balkesen et al. to the morsel-driven parallelization scheme employed by Umbra [12, 35, 59].

After the incoming tuples have been materialized and partitioned, we create the root node of the corresponding hash trie. In contrast to the lazily expanded nested hash tables, these root hash tables can routinely become quite large, depending on the number of distinct join attribute values in the corresponding input pipelines. For this reason we fully parallelize their creation within the morsel-driven parallelization framework provided by Umbra [35, 42]. Concurrent insertions into the same bucket are synchronized with lock-free atomic operations.

An artifact of the self-join patterns frequently found in graph analytic workloads is that multiple input pipelines to a worst-case optimal join may produce exactly the same hash tries. This is evident, for example, in Figure 1b where two of the three tries on the participating relations are identical. We detect this during code generation and only build the corresponding data structures once.

**3.3.3 Probe Phase.** After the build phase, the initial hash trie structure for each input pipeline is available, and the join result can be computed by Algorithm 3. Within the Umbra RDBMS, the hash trie data structure and trie iterators are implemented in plain C++, while the code that implements the build and probe phases of a multi-way join for a specific query is generated by the query compiler. At query compilation time, the query hypergraph and, in particular, the number and order of join attributes is statically

known. This allows us to fully unroll the recursion in Algorithm 3 within the generated code, resulting in a series of tightly nested loops that enumerate the tuples in the join result. This code is fully parallelized by splitting the outermost loop, i.e. the first set intersection, into morsels that can be processed independently by worker threads within the work-stealing framework provided by Umbra [35].

### 3.4 Further Considerations

An attractive way to reduce the amount of work required in the build phase is to exploit existing index structures. As the proposed join algorithm is hash-based, it is unfortunately not possible to reuse traditional comparison-based indexes like B<sup>+</sup>-trees for this purpose. However, with minor extensions to allow for insertions, the proposed hash trie data structure could also be used as a secondary index structure. Then, the build phase can be skipped for input pipelines that scan a suitably indexed relation.

Even more aggressive optimizations are possible if the data is known to be static. In this case, it is actually desirable to perform as much precomputation as possible in order to minimize the time required to answer a query. While this obviates the need for data structures that can be built efficiently on-the-fly, a hash-based approach retains the advantage that complex attribute types can be handled much more efficiently than in a comparison-based approach.

## 4 OPTIMIZING HYBRID QUERY PLANS

As discussed in Section 2, even an efficiently implemented worst-case optimal join can be much slower than a binary join plan if there are no growing binary joins that can be avoided by the worst-case optimal join [2]. Therefore, we argue that a general-purpose system cannot simply replace all binary join plans by worst-case optimal joins and consequently, its query optimizer must be able to generate hybrid plans containing both types of joins.

The main objective of our optimization approach is to avoid binary joins that perform exceptionally poorly due to exploding intermediate results. We thus propose a heuristic approach that refines an optimized binary join plan by replacing cascades of potentially growing joins with worst-case optimal joins. Although the hybrid plans generated by this approach are not necessarily globally optimal, they nevertheless avoid growing intermediate results and thus improve over the original binary plans. We identify such growing joins based on the same cardinality estimates that are used during regular join order optimization. As query optimizers depend heavily on accurate cardinality estimates, state-of-the-art systems have been subject to decades of fine-tuning to produce reasonable estimates on a wide variety of queries. Thus, although it is well-known that errors in these estimates are fundamentally unavoidable [22], we expect our approach to work well on a similarly wide range of queries.

The pseudocode of our approach is shown in Algorithm 4. We perform a recursive post-order traversal of the optimized join tree, and decide for each binary join whether to replace

---

#### Algorithm 4: Refining binary join trees

---

```

input : An optimized operator tree  $T$ 
output: A semantically equivalent operator tree  $T'$ 
        which may employ multi-way joins

1 function refineSubtree( $T$ )
2   if  $T \neq T_l \bowtie T_r$  then
3     return  $T$ ;
4    $T'_l \leftarrow$  refineSubtree( $T_l$ );
5    $T'_r \leftarrow$  refineSubtree( $T_r$ );
6   // Detect growing joins and multi-way join inputs
7   if  $|T| > \max(|T'_l|, |T'_r|) \vee T'_l \neq T_l \vee T'_r \neq T_r$  then
8     return collapseMultiwayJoin( $T'_l \bowtie T'_r$ );
9   return  $T'_l \bowtie T'_r$ ;

```

---

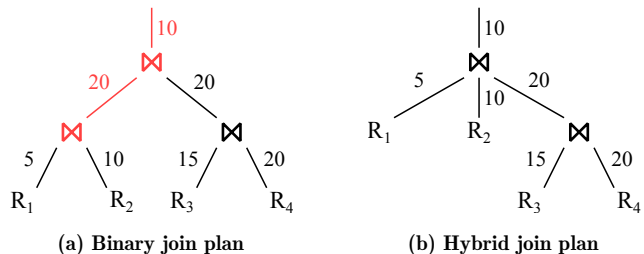


Figure 7: Illustration of the proposed join tree refinement algorithm. A growing binary join and all its ancestors (shown in red in (a)) are collapsed into a single multi-way join (shown in (b)).

it by a multi-way join. A binary join is replaced either if it is classified as a growing join, i.e. its output cardinality is greater than the maximum of its input cardinalities, or if one of its inputs has already been replaced by a multi-way join (line 6). In both cases, a single multi-way join is built from the inputs and the current join condition (cf. Figure 7). We choose to eagerly collapse the ancestors of a growing binary join into a single multi-way join, as the output of a growing join will necessarily contain duplicate key values which would cause redundant work when processed by a regular binary join. Note that the formulation in Algorithm 4 is slightly simplified, as our actual implementation contains additional checks to ensure that only inner joins with equality predicates are transformed into a multi-way join, as this is not possible for other join types in the general case. Furthermore, we do not create multi-way join nodes with only two inputs as they offer no benefit over regular binary joins.

Like many commercial and research RDMBS, Umbra employs a dynamic programming approach for cost-based join order optimization, and we could also attempt to integrate hybrid query plans into the search space of this optimizer. However, this attempts to holistically improve the quality of all query plans, whereas we only want to avoid binary joins that suffer from exploding intermediate results. Furthermore,

recent work within a specialized graph system has shown that accurate cost estimates for such plans require detailed cardinality information that cannot be computed cheaply within a general-purpose RDBMS like Umbra [40].

As the final step of our optimization process, the join attribute order of each multi-way join introduced by Algorithm 4 is optimized in isolation. For this purpose, we adopt the cost-based optimization strategy that was developed for the worst-case optimal Tributary Join algorithm [13]. We selected this particular optimization strategy over other alternatives [2, 3, 8, 40], as its cost estimates rely only on cardinality information that is already maintained within Umbra, and the generated attribute orders exhibited good performance in our preliminary experiments. We emphasize that the multi-way join optimization strategy is entirely independent of both the actual join implementation presented in the previous section and the join tree refinement algorithm presented in this section. Therefore, other multi-way join optimization approaches such as generalized hypertree decompositions could easily be integrated into our system [3, 16].

## 5 EXPERIMENTS

In the following, we present a thorough evaluation of the implementation of the proposed hybrid optimization and hash trie join approach within the Umbra RDBMS [42]. We will subsequently refer to the corresponding system configuration as Umbra<sup>OHT</sup>. For comparison purposes, we also run experiments in which all binary joins are EAGERly replaced by worst-case optimal joins, and refer to the corresponding system configuration as Umbra<sup>EAG</sup>.

### 5.1 Setup

We compare our implementation to the unmodified version of Umbra and to the well-known column-store MonetDB (v11.33.11) both of which exclusively rely on binary join plans [21, 42]. Furthermore, we run comparative experiments with a commercial database system (DBMS X) and the EmptyHeaded system, both of which implement worst-case optimal joins based on ordered index structures [1, 3]. We additionally intended to compare against LevelHeaded, an adaptation of EmptyHeaded for general-purpose queries, but were unable to obtain a copy of its source code which is not publicly available [2]. Finally, we implemented the Leapfrog Triejoin algorithm within Umbra (Umbra<sup>LFT</sup>), based on dense sorted arrays that are built during query processing using the native parallel sort operator of Umbra [52, 54]. Our preliminary experiments showed that using sorted arrays within the Umbra<sup>LFT</sup> system is consistently faster than using the B<sup>+</sup>-tree indexes available within Umbra as the former incur substantially less overhead.

For our experiments, we select the join order benchmark (JOB) which is based on the well-known IMDB data set [36], and the TPCB benchmark at scale factor 30. Furthermore, we run a set of graph-pattern queries on selected network datasets from the Stanford Large Network Dataset Collection which have been used extensively in previous work [38, 46].

**Table 2: Key statistics of the graph datasets used in our experiments.**

dataset	nodes	directed edges	undirected edges
Wiki	7.1 K	103.7 K	100.8 K
Epinions	75.9 K	508.8 K	405.7 K
Slashdot	82.2 K	948.5 K	582.5 K
Google+	0.1 M	13.7 M	12.2 M
Orkut	3.1 M	117.2 M	117.2 M
Twitter	41.7 M	1 468.4 M	1 202.5 M

For a comprehensive evaluation of our approach, we include both comparably small and extremely large data sets. In particular, we choose the Wikipedia vote network [37], as well as the Epinions and Slashdot social networks [39, 49], all of which we classify as small data sets. Finally, we select the much larger Google+ and Orkut user networks [11, 56], as well as the Twitter follower network which is one of the largest publicly available network data sets [34].

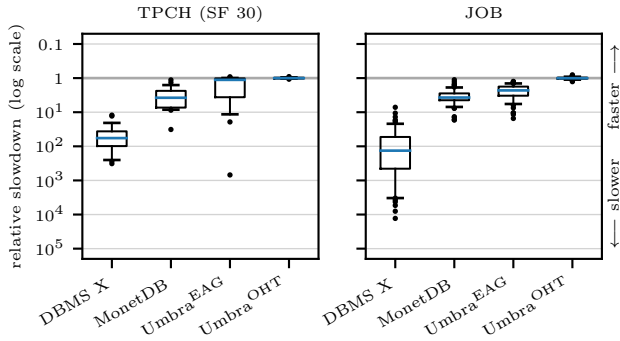
All graph data sets are in the form of edge relations in which each tuple represents a directed edge between two nodes identified by unsigned 64-bit integers. Like previous work on the subject [3, 6, 20, 40, 46], we focus on undirected clique queries on these graphs as they are a common subpattern in graph workloads [46]. In order to allow for undirected queries, the edges are preprocessed such that the source node identifier is less than or equal to the target node identifier. We then run queries that count the number of directed 3, 4, and 5-cliques in these preprocessed graphs, which is equal to the number of undirected cliques in the original graphs [51]. The queries used in our experiments are available online [15].

All experiments are run on a server system with 28 CPU cores (56 hyperthreads) on two Intel Xeon E5-2680 v4 processors and 256 GiB of main memory. Each measurement is repeated three times and we report the results of the best repetition. Our runtime measurements reflect the end-to-end query evaluation time including any time required for query optimization or compilation, and a timeout of one hour is imposed on each individual experiment repetition.

### 5.2 End-To-End Benchmarks

We first present end-to-end benchmarks which demonstrate the effectiveness of the hash trie join implementation.

**5.2.1 Traditional OLAP Workloads.** In our first experiment, we expand upon the preliminary results that were briefly discussed in Section 2. In particular, we demonstrate that a hybrid query optimization strategy is critical to achieve acceptable performance on relational workloads such as TPCB and JOB. For this purpose, we run the TPCB and JOB benchmarks on the unmodified version of Umbra as a baseline, and compare the end-to-end query execution times to DBMS X, MonetDB, and to the Umbra<sup>EAG</sup> and Umbra<sup>OHT</sup> systems. EmptyHeaded is excluded in this experiment as it does not support the complex analytical queries in these benchmarks.



**Figure 8: Relative slowdown of the different systems in comparison to binary join plans within Umbra on TPCH and JOB. The boxplots show the 5th, 25th, 50th, 75th, and 95th percentiles.**

Here, the unmodified version of Umbra that relies purely on binary join plans outperforms all other systems except for the Umbra<sup>OHT</sup> system which employs our novel hybrid optimization strategy. The relative slowdown of these systems in comparison to Umbra is shown in Figure 8. The worst-case optimal join plans of DBMS X exhibit the lowest performance by far, with a median slowdown of 57.4 $\times$  on TPCH and 134.0 $\times$  on JOB. MonetDB performs much better than DBMS X on these benchmarks, but is still outperformed by Umbra with a median slowdown of 3.7 $\times$  on TPCH and JOB, which are measurements consistent with previous work [42]. Our implementation of multi-way joins within Umbra further improves over both DBMS X and MonetDB even when eagerly replacing all binary joins with multi-way joins in the Umbra<sup>EAG</sup> configuration. However, it still incurs a median slowdown of 1.1 $\times$  on TPCH and 2.3 $\times$  on JOB in comparison to Umbra. These results constitute the key observation in this benchmark, as they demonstrate that our implementation of worst-case optimal joins is highly competitive even in comparison to mature and optimized systems such as MonetDB. However, they also show that even such a competitive implementation falls short of binary join plans if the latter do not incur any redundant work. Similar results have been obtained in previous work on the LevelHeaded system [2]. The Umbra<sup>OHT</sup> system which employs our novel hybrid query optimizer closes this gap in performance and incurs no slowdown over the unmodified version of Umbra on the TPCH and JOB benchmarks. In fact, our optimizer correctly determines that a worst-case optimal join plan is *never* beneficial on these queries as there always exists a binary join plan without growing joins (cf. Section 5.3.2).

**5.2.2 Relational Workloads with Growing Joins.** This situation changes when growing joins are unavoidable, e.g. when looking for parts within the same order that are available in the same container from the same supplier on TPCH (cf. Section 1). Our hybrid query optimizer correctly identifies the growing non-key joins in this query, and generates a plan containing both binary and multi-way joins. As a result, the

Umbra	1	0	0	0	0	0	31	0
DBMS X	8	3	16	5	0	0	0	0
MonetDB	8	1	3	10	8	2	0	0
Umbra <sup>EAG</sup>	1	0	0	7	18	6	0	0
Umbra <sup>OHT</sup>	0	0	0	0	0	1	25	5
	timeout	[10 <sup>4</sup> , 10 <sup>3</sup> )	[10 <sup>3</sup> , 10 <sup>2</sup> )	[10 <sup>2</sup> , 10)	[10, 2)	[2, 1.1)	[1.1, 0.9)	[0.9, 0)

**Figure 9: Histogram of the relative slowdown of the different systems in comparison to binary join plans within Umbra when running the JOB queries without filter predicates.**

Umbra<sup>OHT</sup> system exhibits the best overall performance, improving over Umbra by a factor of 1.9 $\times$  and over Umbra<sup>EAG</sup> by a factor of 4.2 $\times$ . In comparison to MonetDB and DBMS X, the speedup of Umbra<sup>OHT</sup> increases even further to 7.6 $\times$  and 350.0 $\times$ , respectively.

We broaden this experiment by additionally running the JOB queries without any filter predicates on the base tables. Similar to the previous query on TPCH, they contain a mix of non-growing and growing joins and are thus challenging to optimize. Query 29 is excluded in this experiment as it contains an extremely large number of joins which causes the query result to explode beyond the size that even a worst-case optimal join plan can realistically enumerate. We again measure the relative performance of the competitor systems in comparison to the unmodified version of Umbra.

Figure 9 shows the distribution of this relative performance for each system. Most importantly, we observe that although the benchmark now contains growing joins, neither DBMS X nor the Umbra<sup>EAG</sup> system are able to match the performance of the unmodified version of Umbra, by a similar margin as in the previous experiment. This indicates that pure worst-case optimal join plans are still not feasible on queries which contain a mix of growing and non-growing joins, where the non-growing joins could be processed much more efficiently by binary joins. The relative performance of MonetDB deteriorates sharply in comparison to the regular JOB benchmark, as it materializes all intermediate results which become much larger in the presence of growing joins and might even have to be spilled to disk. In contrast, the Umbra<sup>OHT</sup> system with our hybrid query optimizer matches or improves over the performance of Umbra, by identifying five queries on which a hybrid query plan containing worst-case optimal joins is superior to a traditional binary join plan. Moreover, the hybrid query plans employed by the Umbra<sup>OHT</sup> system do not incur any timeouts on this benchmark, unlike any other system that we investigate.

**5.2.3 Graph Pattern Queries.** Finally, we evaluate our hash trie join implementation on the graph pattern queries and data sets introduced above. On such queries, worst-case optimal join plans typically exhibit asymptotically better runtime complexity than binary join plans, and previous research has shown that large improvements in query processing time

**Table 3: Absolute runtime in seconds of the graph pattern queries on the small network data sets.**

		Wiki	Epinions	Slashdot
3-clique	<i>EH-Probe</i>	0.28	0.30	0.29
	EmptyHeaded	0.43	0.79	1.07
	DBMS X	0.28	0.52	1.37
	MonetDB	0.37	0.96	0.97
	Umbra	<b>0.03</b>	<b>0.06</b>	<b>0.08</b>
	Umbra <sup>LFT</sup>	0.36	0.53	0.49
	Umbra <sup>OHT</sup>	<b>0.04</b>	<b>0.07</b>	<b>0.07</b>
4-clique	<i>EH-Probe</i>	0.40	0.55	0.47
	EmptyHeaded	0.55	1.04	1.24
	DBMS X	1.66	6.53	13.95
	MonetDB	8.16	16.58	10.63
	Umbra	1.61	12.04	7.91
	Umbra <sup>LFT</sup>	3.82	7.02	4.09
	Umbra <sup>OHT</sup>	<b>0.10</b>	<b>0.23</b>	<b>0.18</b>
5-clique	<i>EH-Probe</i>	0.97	3.19	1.57
	EmptyHeaded	1.12	3.69	2.35
	DBMS X	8.98	85.21	80.93
	MonetDB	368.34	1 392.41	timeout
	Umbra	45.06	570.92	166.30
	Umbra <sup>LFT</sup>	21.47	57.13	37.48
	Umbra <sup>OHT</sup>	<b>0.42</b>	<b>1.43</b>	<b>0.90</b>

are possible [3, 46]. However, systems that are optimized for such read-only workloads require expensive precomputation of index structures in order to achieve high performance [3]. Our experiments show that the hash trie join implementation within Umbra achieves competitive end-to-end performance on such workloads, even though it computes all required data structures on-the-fly during query processing.

We first run the 3 and 4-clique queries on the small Wiki, Epinions, and Slashdot graph data sets. The absolute end-to-end query execution times of the different systems are shown in Table 3. Note that our measurements for EmptyHeaded include the time required for its precomputation step, without any disk IO that is done as part of this step. For reference, we also provide measurements for EmptyHeaded that exclude this precomputation step (*EH-Probe*). First of all, we observe that the hash trie join implementation within the Umbra<sup>OHT</sup> system consistently exhibits the best runtime across all data sets and queries, outperforming the remaining systems by up to two orders of magnitude. In general, the performance advantage of worst-case optimal join plans rapidly increases as the complexity of the graph pattern queries grows. This is to be expected, as more complex pattern queries result in more intermediate results that can explode when using a binary join plan. Interestingly, the unmodified version of Umbra matches the performance of our hash trie join implementation on the 3-clique query, and all other systems perform considerably worse. In case of EmptyHeaded, this is evidence of both a large optimization and compilation overhead that

**Table 4: Absolute runtime in seconds of the 3-clique query on the large network data sets.**

	Google+	Orkut	Twitter
<i>EH-Probe</i>	0.64	2.78	150.16
EmptyHeaded	18.67	309.14	timeout
DBMS X	59.77	311.44	timeout
Umbra	28.53	55.49	timeout
Umbra <sup>LFT</sup>	14.55	30.61	1 175.97
Umbra <sup>OHT</sup>	<b>7.70</b>	<b>15.25</b>	<b>579.07</b>

we observed to be essentially static on this benchmark, and its expensive precomputation step. The multi-way join implementations of DBMS X and Umbra<sup>LFT</sup> rely on ordered data structures which are less efficient than our optimized hash trie data structure, a finding that is also evident on the more complex graph pattern queries. Finally, we emphasize that the Umbra<sup>OHT</sup> system outperforms the highly optimized EmptyHeaded system even on the complex 4- and 5-clique queries where the static overhead incurred by EmptyHeaded does not affect its runtime as significantly.

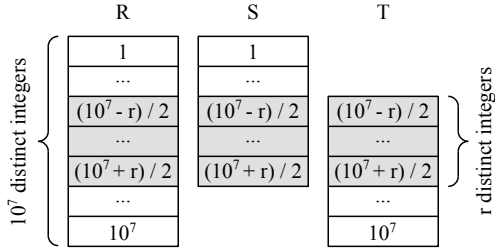
Finally, we run the 3-clique query on the much larger Google+, Orkut, and Twitter graph data sets (cf. Table 4). We do not run the more complex graph pattern queries on these data sets as their result size on large data sets quickly increases beyond the size that can be reasonably enumerated by any system. We also exclude MonetDB in this experiment, as it cannot compute the query result within the one hour time frame allocated for each experiment repetition. Once again, the Umbra<sup>OHT</sup> system consistently outperforms its competitors by a large margin. The performance of EmptyHeaded degrades in comparison to the benchmarks on the small data sets, as its precomputation step becomes excessively expensive on these larger data sets.

### 5.3 Detailed Evaluation

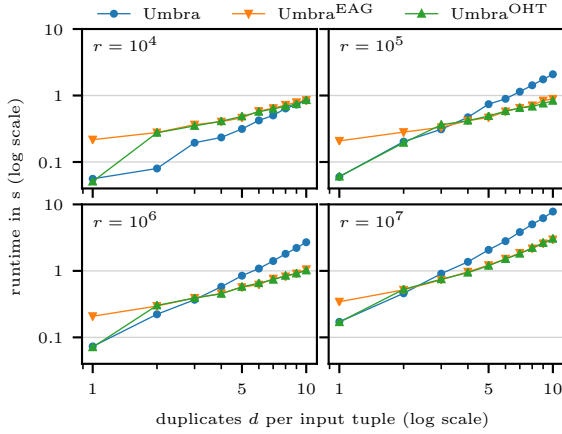
The remainder of our experiments provide a detailed evaluation of the applicability of worst-case optimal joins to relational workloads, and of the proposed optimization strategy.

**5.3.1 Applicability of Worst-Case Optimal Joins.** We expand on the end-to-end benchmarks presented above, and study the applicability of worst-case optimal joins within a general-purpose RDBMS in detail. Traditional relational workloads such as TPCCH or JOB do not produce any growing intermediate results and thus there is no benefit in introducing a worst-case optimal join. In fact, as demonstrated above, worst-case optimal joins incur a substantial overhead on such workloads, primarily since they have to materialize all their inputs in suitable index structures. However, as exemplified by the experiments in Section 5.2.2, growing intermediate results can arise, for example, due to unconstrained joins between foreign keys.

In order to study such workloads under controlled conditions, we generate an additional synthetic benchmark. In



**Figure 10: Illustration of the synthetically generated relations  $R$ ,  $S$ , and  $T$ . The tuples in each relation are duplicated  $d \in \{1, \dots, 10\}$  times, leading to the natural join  $R \bowtie S \bowtie T$  containing  $rd^3$  tuples.**



**Figure 11: Absolute query runtime on the synthetic query  $R \bowtie S \bowtie T$  as the number of distinct values  $r$  and duplicated tuples  $d$  in the query result is varied.**

particular, we choose two parameters  $r \in \{10^4, 10^5, 10^6, 10^7\}$  and  $d \in \{1, \dots, 10\}$ , and generate randomly shuffled relations  $R$ ,  $S$ , and  $T$  as follows.  $R$  simply contains the distinct integers  $1, \dots, 10^7$ , while  $S$  and  $T$  contain the distinct integers  $1, \dots, (10^7 + r)/2$  and  $(10^7 - r)/2, \dots, 10^7$  respectively (cf. Figure 10). Each distinct integer in  $R$ ,  $S$ , and  $T$  is duplicated  $d$  times. Thus the result of the natural join  $R \bowtie S \bowtie T$  contains exactly  $r$  distinct integers, each of which is duplicated  $d^3$  times for a total of  $rd^3$  tuples. While any binary join plan for this query will contain growing intermediate results for  $d > 1$ , they do not grow beyond the size of the query result if the join  $S \bowtie T$  is performed first. This differs from graph pattern queries, where usually *any* binary join plan produces an intermediate result that is larger than the query result.

Figure 11 shows the absolute runtime of the query  $R \bowtie S \bowtie T$  for different configurations of the Umbra system. As expected, we observe that as the number of duplicates in the join result is increased, the runtime of binary join plans increases much more rapidly in comparison to worst-case optimal joins. As outlined above, each distinct value in the join result is duplicated  $d^3$  times. In a binary join plan, enumerating each one of these duplicates requires at least two

**Table 5: Breakdown of the decisions made by the hybrid query optimizer on each benchmark. The table shows the total number of joins, as well as the number of introduced multi-way joins categorized into true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN).**

Benchmark	Joins	TP	TN	FP	FN
TPCH	59	0	59	0	0
JOB	864	0	859	0	5
JOB (no filters)	234	19	140	0	75
Graph	48	48	0	0	0
Synthetic	80	52	8	18	2
Total	1285	119	1066	18	82

hash table lookups. In contrast, a hash trie join determines once that the distinct value is part of the join result after which all duplicates thereof can be enumerated without any additional hash table operations.

However, we also observe that the superior scaling behavior of worst-case optimal joins does not necessarily translate to an actual runtime advantage. If there are few distinct values  $r$  or duplicates  $d$  in the query result, binary join plans still exhibit reasonable performance and commonly outperform worst-case optimal joins. In these cases, the additional time required by a hash trie join for materializing all input relations in hash tries exceeds the time saved by its more efficient join evaluation. Consequently, the break-even point at which worst-case optimal joins begin to outperform binary join plans moves towards a smaller number of duplicates  $d$  as the number of distinct values  $r$  in the query result is increased. That is, worst-case optimal joins offer the greatest benefit on join queries where most tuples from the base relations have a large number of join partners.

Finally, we note that the hybrid query optimizer employed by Umbra<sup>OHT</sup> accurately detects this break-even point for  $r > 10^4$ , resulting in good performance across the full range of possible query behavior. While the optimizer does switch to worst-case optimal joins too early for  $r = 10^4$ , we determined that this is caused by erroneous cardinality estimates. This is a common failure mode in relational databases which unfortunately cannot be avoided in the general case [22, 36]. Crucially, however, the optimizer always correctly detects the case  $d = 1$  which corresponds to a traditional relational workload without growing intermediate results.

**5.3.2 Optimizer Evaluation.** In order to gain more detailed insights into the behavior of our hybrid query optimizer, we additionally analyze every decision made by the optimizer on the benchmarks presented in this paper so far. Specifically, we collect both the estimated and true input and output cardinalities of all join operators inspected by Algorithm 4. For a given join, we subsequently check if the optimizer decided to introduce a multi-way join or not, and whether this decision matches the correct decision given the true input and output cardinalities. This allows us to categorize these

decisions into true and false positives, respectively negatives, where the correct introduction of a multi-way join is counted as a true positive.

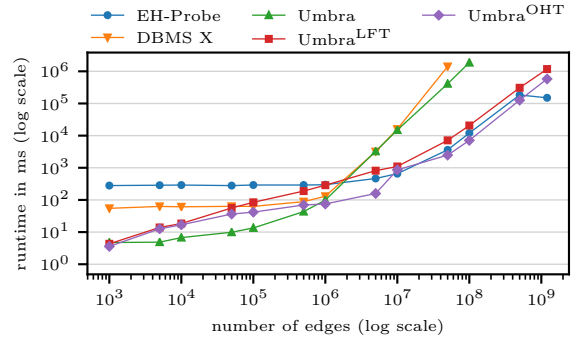
An overview of the results is shown in Table 5. As discussed previously, growing joins are exceptionally rare in traditional relational workloads like TPC-H and JOB. Out of a total of 923 joins, the optimizer incurs only 5 false negatives where a growing join is incorrectly classified as non-growing. These errors occur on two JOB queries (8c and 16b) where the initial binary join ordering produces a suboptimal plan containing mildly growing joins due to incorrect cardinality estimates. We determined that the optimal plan for these queries would not contain any growing joins. Beyond that, our results show that the proposed hybrid query optimizer is insensitive to the cardinality estimation errors that routinely occur in relational workloads [22, 36]. This is to be expected, as the optimizer relies only on the relative difference between the cardinality estimates of the input and output of join operators, and not their absolute values.

On the modified JOB queries, the optimizer correctly identifies the severely growing joins, while also incurring a comparably large number of false negatives. They occur primarily on weakly growing joins, where minor errors in the absolute cardinality estimates can already affect the decision made by the hybrid optimizer. However, we measured that these false negatives affect the absolute query runtime only on 3 of the 32 queries, on which we only miss a potential further speedup of up to 1.6 $\times$ . Nevertheless, the corresponding hybrid query plans still outperform query plans that contain only binary joins (cf. Figure 9). The join attributes in this benchmark are frequently primary keys, which generally causes Umbra to estimate lower cardinalities. A major advantage of this behavior is that it makes false positives, i.e. the incorrect introduction of multi-way joins, unlikely and in fact no false positives occur on these queries. This is critical to ensure that we do not compromise the performance of Umbra on traditional relational queries.

The graph pattern queries contain only very rapidly growing joins, all of which are correctly identified by the hybrid optimizer. As discussed above, the behavior of joins in the synthetic benchmark varies. However, as none of the join attributes are marked as primary key columns the system is much less hesitant to estimate high output cardinalities for joins. This is evident in Figure 11 for  $r = 10^4$  and results in some false positives which in comparison to the optimal plan increase the absolute query runtime by up to 3.7 $\times$  for  $d = 2$ . However, it is important to note that these false positives affect only joins on non-key columns where  $d > 1$ . In summary, our results show that the proposed hybrid query optimizer achieves high accuracy even under difficult circumstances and across a wide range of different queries.

## 5.4 Microbenchmarks

We conclude our experiments by providing an in-depth evaluation of key aspects of our implementation. These microbenchmarks are conducted using the simple 3-clique graph pattern



**Figure 12: Absolute runtime of the 3-clique query on increasingly larger random subsets of the Twitter data set.**

query as this query has been used extensively to evaluate the performance of other systems in related work [2, 46].

**5.4.1 Scaling Behavior.** First, we investigate the scaling behavior of the different systems as the data set size grows. For this purpose, we run the 3-clique query on increasingly larger randomly chosen subsets of the Twitter data and record the end-to-end query execution time, the results of which are shown in Figure 12. We exclude precomputation time for EmptyHeaded in this benchmark in order to better highlight the scaling behavior of its join implementation.

We can make several key observations on these results. First, we note that the binary join plans of the unmodified version of Umbra actually exhibit the best overall performance up to a data set size of roughly  $10^6$ , which is explained by the fact that the smaller random subgraphs are highly disconnected with only few 2-paths and 3-cliques. Therefore, a binary join plan will not have to enumerate large intermediate results, and at the same time forgoes the overhead of building the trie data structures required by the Umbra<sup>LFT</sup> and Umbra<sup>OHT</sup> systems. Similar to the experiments on synthetic data presented in Section 5.3.1, the hybrid optimizer incurs some false positives in these cases due to incorrect cardinality estimates. That is, it incorrectly introduces a worst-case optimal join although there is no runtime benefit in doing so. Analogous to the end-to-end benchmark results presented above, we observe a large static overhead on these small data sets for the EmptyHeaded system.

On the larger subgraphs with more than  $10^6$  edges, the performance of Umbra quickly degrades until query execution times out on graphs with more than  $10^8$  edges. Surprisingly, the runtimes of DBMS X exhibit virtually the same asymptotic behavior which could indicate that the system incorrectly uses a binary join plan in this benchmark. In contrast, our hybrid optimizer selects a worst-case optimal join plan for the Umbra<sup>LFT</sup> and Umbra<sup>OHT</sup> systems resulting in greatly improved runtime, and the Umbra<sup>OHT</sup> system consistently outperforms the comparison-based Umbra<sup>LFT</sup> system. In fact, the hash trie join implementation in the Umbra<sup>OHT</sup> configuration actually even matches or outperforms EmptyHeaded on all subgraph queries and only falls



**Table 6: Ablation tests using the 3-clique query on random subsets of the Twitter data. Runtime is shown in seconds, and memory consumption is shown in GiB.**

edges	metric	baseline	-LE	-SP	-RP
5 M	runtime	0.17	1.24×	2.33×	2.25×
	memory	0.35	1.08×	1.79×	1.79×
50 M	runtime	2.52	1.21×	1.49×	1.32×
	memory	3.69	1.05×	1.29×	1.29×
500 M	runtime	126.96	1.01×	1.04×	1.18×
	memory	35.48	1.02×	1.05×	1.05×
1 202 M	runtime	579.07	1.00×	1.03×	1.24×
	memory	84.03	1.01×	1.02×	1.02×

short on the full Twitter data set, although we do not measure the precomputation time required by EmptyHeaded in this experiment. EmptyHeaded heavily relies on aggressive set layout optimizations in its precomputed index structures which enable it to employ an optimized set intersection algorithm [3]. These optimizations are dependent on a suitable dense numbering of the nodes within the graph data which is present in the full Twitter data, but not in any random subgraphs thereof. In contrast, the performance of the proposed hash trie approach is entirely independent of such data set specifics, making it much more versatile in practice.

**5.4.2 Ablation Tests.** Next, we study which impact the main optimizations introduced in Section 3 have on the performance of the hash trie join algorithm. We thus disable these optimizations one-by-one within the Umbra<sup>OHT</sup> system, and record the runtime and memory consumption of the 3-clique query on selected random subgraphs of the Twitter data. Specifically, Table 6 shows the performance with all optimizations enabled (baseline), and with successively disabled lazy child expansion (-LE), singleton pruning (-SP), and radix partitioning of the input (-RP).

Overall, the experiment shows that these optimizations have a positive impact on both runtime and memory consumption in all cases, and disabling all optimizations increases runtime by up to a factor of 2.25× and memory consumption by up to a factor of 1.79×. Lazy child expansion and singleton pruning are generally more useful on the smaller random subgraphs. As mentioned above, this is to be expected since these graphs are sparse and highly disconnected, leading to many nodes that never have to be expanded or that have only a single outgoing edge. In contrast, radix partitioning has a positive impact on runtime regardless of the size of the data set. It is arguably the most important optimization of our approach, as it eliminates any runtime fluctuations due to the specific order in which data is stored in the base tables. In combination, the proposed optimizations thus enable the hash trie join algorithm to perform well on a wide variety of data sets with diverse sizes and characteristics.

**Table 7: Comparison of the absolute runtime in seconds of the 3-clique query on the Orkut data when using string keys instead of integer keys.**

	integer	string	slowdown
DBMS X	311.44	726.80	2.33×
Umbra <sup>LFT</sup>	30.61	58.53	1.91×
Umbra <sup>OHT</sup>	<b>15.25</b>	<b>17.29</b>	<b>1.13×</b>

**Table 8: Comparison of the build and probe times in seconds required for the 3-clique query on the Orkut data set.**

	1 thread		56 threads	
	build	probe	build	probe
EmptyHeaded	471.42	<b>75.85</b>	306.36	<b>2.78</b>
Umbra <sup>LFT</sup>	207.87	729.31	8.91	21.70
Umbra <sup>OHT</sup>	<b>20.84</b>	435.21	<b>1.01</b>	14.23

**5.4.3 Non-Integer Key Attributes.** Previous work has shown that non-integer key attributes are ubiquitous in real-world data sets [53]. As outlined in Section 3, the proposed hash trie join approach is for the most part not comparison-based, and therefore the actual key data types used in a multi-way join do not significantly affect the query runtime. We demonstrate this by changing the data type of the edge relation attributes from 64-bit integers to variable-length strings representing the same integers, and subsequently run the 3-clique query on this modified graph data set. We choose the Orkut data set for the remaining experiments to avoid timeouts due to excessively long runtimes in our competitors. EmptyHeaded does not support strings as join key attributes and is thus excluded from this experiment.

Table 7 shows the query execution time of DBMS X, Umbra<sup>LFT</sup> and Umbra<sup>OHT</sup> when using 64-bit integers or strings as the join key attributes. Unsurprisingly, the comparison-based approaches incur a large performance hit of 2.33× in case of DBMS X, and 1.91× in case of Umbra<sup>LFT</sup> when computing the 3-clique query on string attributes, as string comparisons are much more expensive than integer comparisons. In contrast, the performance of the Umbra<sup>OHT</sup> system is hardly affected and decreases only by a factor of 1.13×. A small performance penalty is unavoidable even in case of the Umbra<sup>OHT</sup> system, as we still have to compute hash values of string attributes and check the actual join condition before producing a result tuple (cf. Section 3).

**5.4.4 Build and Probe Times.** Finally, we investigate the tradeoff that the different systems make between the effort spent on building the required index structures and the time required for query execution. For this purpose, we separately record the build and probe times for the 3-clique query on the Orkut data set. As EmptyHeaded does not support fully multi-threaded precomputation of its index structures, we additionally run this experiment single-threaded. DBMS X

does not provide any means to separately record build and probe times and is thus excluded from this experiment.

As shown in Table 8, EmptyHeaded spends far more time on precomputation than on query execution even in the single-threaded case, by a factor of roughly 6×. EmptyHeaded builds a common dense dictionary encoding of all join attribute values during precomputation. This operation is hard to parallelize efficiently and EmptyHeaded does not provide an optimized multi-threaded implementation. Therefore, this factor increases to 110× in the multi-threaded case. While this expensive precomputation results in greatly improved query execution time, the combined runtime falls short of that required by the hash trie join approach of the Umbra<sup>OHT</sup> system. In the proposed hash trie join approach, we trade a much lower build time for a somewhat increased probe time. Crucially, however, this enables us to avoid any precomputation of persistent index structures while still offering competitive performance. This is not possible with an ordered trie join approach, as both the build and probe times of the Umbra<sup>LFT</sup> system are greatly increased in comparison to the Umbra<sup>OHT</sup> system.

## 6 RELATED WORK

As outlined in Section 1, it is well-known that binary joins exhibit suboptimal performance in some cases, and especially in the presence of growing intermediate results [10, 17, 28, 58]. Hash Teams and Eddies were early approaches that addressed some of these shortcomings by simultaneously processing multiple input relations in a single multi-way join [10, 17, 28]. However, these approaches do not specifically focus on avoiding growing intermediate results as Hash Teams are primarily concerned with avoiding redundant partitioning steps in cascades of partitioned hash joins [17, 28], and Eddies allow different operator orderings to be applied to different subsets of the base relations [10]. They still rely on binary joins internally and hence are not worst-case optimal in the general case.

Ngo et al. were among the first to propose a worst-case optimal join algorithm [43–45], which provides the foundation of most subsequent worst-case optimal join algorithms, including our proposed hash trie join algorithm (cf. Section 3). On this basis, theoretical work has since continued in a variety of directions, such as operators beyond joins [3, 23, 24, 27, 30, 55], stronger optimality guarantees [5, 31, 32, 43], and incremental maintenance of the required data structures [25, 26]. Implementations of worst-case optimal join algorithms have been proposed and investigated in a variety of settings. Veldhuizen proposed the well-known Leapfrog Triejoin algorithm that is used in the LogicBlox system and can be implemented on top of existing ordered indexes or plain sorted data [13, 52, 54]. Variants of such join algorithms have been adopted in distributed query processing [4, 6, 13, 33] graph processing [3, 6, 20, 40, 57, 60], and general-purpose query processing [2, 8].

However, such comparison-based implementations incur a number of problems, as outlined in more detail in Sections 1

and 2. Using persistent precomputed index structures is only feasible if the maximum number of indexes is bounded such as in specialized graph processing or RDF engines [3, 20, 40], whereas a general-purpose RDBMS cannot guarantee any such bounds. A more feasible approach is to sort the input data on-the-fly during query processing. This has been shown to work well in distributed query processing where communication costs far outweigh the computation costs [13], but can severely impact the performance of a single-node system. The proposed approach could also be applied to this domain, e.g. by integrating it into the approach proposed by Chu et al. [13] Here, data is sent to worker nodes in a single communication round, after which the entire query result can be computed by running the original query locally on the data sent to each node. The latter step could be performed by the proposed hybrid join processing technique, allowing different query plans to be chosen on the worker nodes depending on the local data characteristics.

Veldhuizen already suggested representing the required trie index structures through nested hash tables [52]. However, as this paper demonstrates a careful implementation of this idea is required to achieve acceptable performance, and we are not aware of any previous work addressing this practical challenge. Fekete et al. propose an alternative, radix-based algorithm that achieves the same goal, but do not evaluate an actual implementation of their approach [14].

The hash trie data structure itself is structurally similar to hash array mapped tries [47] and the data structure used in extendible hashing schemes [18]. However, while these approaches allow for optimized point lookups of individual keys, our hash trie data structure supports optimized range-lookups of key prefixes as they are required by a hash-based multi-way join algorithm. Prefix hash trees within peer-to-peer networks address a similar requirement, albeit with different optimization goals such as resilience [48].

A key contribution of this paper is a comprehensive implementation of our approach within the general-purpose Umbra RDBMS [42]. The LevelHeaded system is an evolution of the graph processing engine EmptyHeaded towards such a general-purpose system, but like EmptyHeaded it requires expensive precomputation of persistent index structures and only allows for static data [2, 3]. The most mature system that implements worst-case optimal joins is the commercial LogicBlox system which allows for fully dynamic data through incremental maintenance of the required index structures [8, 25]. However, previous work has shown that it exhibits poor performance on standard OLAP workloads [2].


Similar to our approach, LogicBlox is reported to also employ a hybrid optimization strategy [2], but no information is available on the details of this strategy. Approaches that holistically optimize hybrid join plans have been proposed for graph processing [40, 60], but as outlined in Section 4 these approaches generally rely on statistics that are prohibitively expensive to compute or maintain in a general-purpose RDBMS. An algorithm that is similar to our join tree refinement approach has been proposed for introducing multi-way joins using generalized hash teams into binary

join plans [17, 19, 28]. However, this approach greedily transforms as many binary joins as possible into a multi-way join which results in suboptimal performance according to our experiments.

## 7 CONCLUSIONS

In this paper, we presented a comprehensive approach that allows the seminal work on worst-case optimal join processing to be integrated seamlessly into general-purpose relational database management systems. We demonstrated the feasibility of this approach by implementing and evaluating it within the state-of-the-art Umbra system. Our implementation offers greatly improved runtime on complex analytical and graph pattern queries, where worst-case optimal joins have an asymptotic runtime advantage over binary joins. At the same time, it loses no performance on traditional OLAP workloads where worst-case optimal joins are rarely beneficial. We achieve this through a novel hybrid query optimizer that intelligently combines both binary and worst-case optimal joins within a single query plan, and through a novel hash-based multi-way join algorithm that does not require any expensive precomputation. Our contributions thereby allow mature relation database management systems to benefit from recent insights into worst-case optimal join algorithms, exploiting the best of both worlds.

## ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement number 725286). 

## REFERENCES

- [1] Christopher Aberger. [n.d.]. EmptyHeaded GitHub repository. <https://github.com/HazyResearch/EmptyHeaded>.
- [2] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. 2018. LevelHeaded: A Unified Engine for Business Intelligence and Linear Algebra Querying. In *ICDE*. IEEE Computer Society, 449–460.
- [3] 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 (2017), 20:1–20:44.
- [4] Foto N. Afrati and Jeffrey D. Ullman. 2011. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Trans. Knowl. Data Eng.* 23, 9 (2011), 1282–1298.
- [5] Kaleb Alway, Eric Blais, and Semih Salihoglu. 2019. Box Covers and Domain Orderings for Beyond Worst-Case Join Processing. *CoRR* abs/1909.12102 (2019).
- [6] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows. *PVLDB* 11, 6 (2018), 691–704.
- [7] Austin Appleby. [n.d.]. Murmurhash GitHub repository. <https://github.com/aappleby/smhasher>.
- [8] 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 *SIGMOD Conference*. ACM, 1371–1382.
- [9] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.* 42, 4 (2013), 1737–1767.
- [10] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *SIGMOD Conference*. ACM, 261–272.
- [11] Lars Backstrom, Daniel P. Huttenlocher, Jon M. Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *KDD*. ACM, 44–54.
- [12] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*. IEEE Computer Society, 362–373.
- [13] Shumo Chu, Magdalena Balazinska, and Dan Suciu. 2015. From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System. In *SIGMOD Conference*. ACM, 63–78.
- [14] Alan Fekete, Brody Franks, Herbert Jordan, and Bernhard Scholz. 2019. Worst-Case Optimal Radix Triejoin. *CoRR* abs/1912.12747 (2019).
- [15] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Queries used in the experimental evaluation. <https://github.com/freitmi/queries-vldb2020>.
- [16] Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. 2005. Hypertree Decompositions: Structure, Algorithms, and Applications. In *WG (Lecture Notes in Computer Science)*, Vol. 3787. Springer, 1–15.
- [17] Goetz Graefe, Ross Bunker, and Shaun Cooper. 1998. Hash Joins and Hash Teams in Microsoft SQL Server. In *VLDB*. Morgan Kaufmann, 86–97.
- [18] Sven Helmer, Robin Aly, Thomas Neumann, and Guido Moerkotte. 2007. Indexing Set-Valued Attributes with a Multi-level Extendible Hashing Scheme. In *DEXA (Lecture Notes in Computer Science)*, Vol. 4653. Springer, 98–108.
- [19] Michael Henderson and Ramon Lawrence. 2013. Are Multi-way Joins Actually Useful?. In *ICEIS (1)*. SciTePress, 13–22.
- [20] Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. 2019. A Worst-Case Optimal Join Algorithm for SPARQL. In *ISWC (1) (Lecture Notes in Computer Science)*, Vol. 11778. Springer, 258–275.
- [21] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35, 1 (2012), 40–45.
- [22] Yannis E. Ioannidis and Stavros Christodoulakis. 1991. On the Propagation of Errors in the Size of Join Results. In *SIGMOD Conference*. ACM Press, 268–277.
- [23] Manas Joglekar, Rohan Puttagunta, and Christopher Ré. 2015. Aggregations over Generalized Hypertree Decompositions. *CoRR* abs/1508.07532 (2015).
- [24] Manas R. Joglekar, Rohan Puttagunta, and Christopher Ré. 2016. AJAR: Aggregations and Joins over Annotated Relations. In *PODS*. ACM, 91–106.
- [25] Oren Kalinsky, Yoav Etsion, and Benny Kimelfeld. 2017. Flexible Caching in Trie Joins. In *EDBT*. OpenProceedings.org, 282–293.
- [26] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2019. Counting Triangles under Updates in Worst-Case Optimal Time. In *ICDT (LIPICs)*, Vol. 127. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 4:1–4:18.
- [27] Ahmet Kara and Dan Olteanu. 2018. Covers of Query Results. In *ICDT (LIPICs)*, Vol. 98. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 16:1–16:22.
- [28] Alfons Kemper, Donald Kossmann, and Christian Wiesner. 1999. Generalised Hash Teams for Join and Group-by. In *VLDB*. Morgan Kaufmann, 30–41.
- [29] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. 195–206. <https://doi.org/10.1109/ICDE.2011.5767867>
- [30] Mahmoud Abo Khamis, Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2019. On Functional Aggregate Queries with Additive Inequalities. In *PODS*. ACM, 414–431.
- [31] Mahmoud Abo Khamis, Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2016. Joins via Geometric Resolutions: Worst Case and beyond. *ACM Trans. Database Syst.* 41, 4 (2016), 22:1–22:45.
- [32] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *PODS*. ACM, 13–28.
- [33] Paraschos Koutris, Paul Beame, and Dan Suciu. 2016. Worst-Case Optimal Algorithms for Parallel Query Processing. In *ICDT (LIPICs)*, Vol. 48. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 8:1–8:18.
- [34] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon. 2010. What is Twitter, a social network or a news media?. In *WWW*. ACM, 591–600.

- [35] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*. ACM, 743–754.
- [36] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.
- [37] Jure Leskovec, Daniel P. Huttenlocher, and Jon M. Kleinberg. 2010. Signed networks in social media. In *CHI*. ACM, 1361–1370.
- [38] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. <http://snap.stanford.edu/data>.
- [39] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2009. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics* 6, 1 (2009), 29–123.
- [40] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *PVLDB* 12, 11 (2019), 1692–1704.
- [41] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550.
- [42] Thomas Neumann and Michael Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.
- [43] Hung Q. Ngo, Dung T. Nguyen, Christopher Ré, and Atri Rudra. 2014. Beyond worst-case analysis for joins with minesweeper. In *PODS*. ACM, 234–245.
- [44] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case Optimal Join Algorithms. *J. ACM* 65, 3 (2018), 16:1–16:40.
- [45] Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2013. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record* 42, 4 (2013), 5–16.
- [46] Dung T. Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2015. Join Processing for Graph Patterns: An Old Dog with New Tricks. In *GRADES@SIGMOD/PODS*. ACM, 2:1–2:8.
- [47] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. 2011. Lock-Free Resizable Concurrent Tries. In *LCPC (Lecture Notes in Computer Science)*, Vol. 7146. Springer, 156–170.
- [48] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. 2004. Brief announcement: Prefix hash tree. In *PODC*. ACM, 368.
- [49] Matthew Richardson, Rakesh Agrawal, and Pedro M. Domingos. 2003. Trust Management for the Semantic Web. In *International Semantic Web Conference (Lecture Notes in Computer Science)*, Vol. 2870. Springer, 351–368.
- [50] J. Andrew Rogers. [n.d.]. AquaHash GitHub repository. <https://github.com/jandrewrogers/AquaHash>.
- [51] Thomas Schank and Dorothea Wagner. 2005. Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study. In *WEA (Lecture Notes in Computer Science)*, Vol. 3503. Springer, 606–609.
- [52] Todd L. Veldhuizen. 2014. Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*. 96–106.
- [53] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *DBTest@SIGMOD*. ACM, 1:1–1:6.
- [54] Haicheng Wu, Daniel Zinn, Molham Aref, and Sudhakar Yalamanchili. 2014. Multipredicate Join Algorithms for Accelerating Relational Graph Processing on GPUs. In *ADMS@VLDB*. 1–12.
- [55] Konstantinos Xirogiannopoulos and Amol Deshpande. 2019. Memory-Efficient Group-by Aggregates over Multi-Way Joins. *CoRR* abs/1906.05745 (2019).
- [56] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *ICDM*. IEEE Computer Society, 745–754.
- [57] Wangda Zhang, Reynold Cheng, and Ben Kao. 2014. Evaluating multi-way joins over discounted hitting time. In *ICDE*. IEEE Computer Society, 724–735.
- [58] Xiaofei Zhang, Lei Chen, and Min Wang. 2012. Efficient Multi-way Theta-Join Processing Using MapReduce. *PVLDB* 5, 11 (2012), 1184–1195.
- [59] Zuyu Zhang, Harshad Deshmukh, and Jignesh M. Patel. 2019. Data Partitioning for In-Memory Systems: Myths, Challenges, and Opportunities. In *CIDR*.
- [60] Guanghui Zhu, Xiaoqi Wu, Liangliang Yin, Haogang Wang, Rong Gu, Chunfeng Yuan, and Yihua Huang. 2019. HyMJ: A Hybrid Structure-Aware Approach to Distributed Multi-way Join Query. In *ICDE*. IEEE, 1726–1729.