Adopting Worst-Case Optimal Joins in Relational Database Systems

Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, Thomas Neumann
Technische Universität München
{freitagm, bandle, tobias.schmidt, kemper, 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.

PVLDB Reference Format:
Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. Adopting Worst-Case Optimal Joins in Relational Database Systems. *PVLDB*, 13(11): 1891-1904, 2020.
DOI: https://doi.org/10.14778/3407790.3407797

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, 19, 30]. The main shortcoming of binary joins is the generation of intermediate results that can become much larger than the actual query result [46].

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 TPCH 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 [16]. 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, 48, 54].

Consequently, there has been a long-standing interest in *multi-way joins* that avoid enumerating any potentially exploding intermediate results [10, 19, 30]. 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, 45, 46, 54]. 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 [42]. 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.

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407797

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 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 [44]. 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 TPCH 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

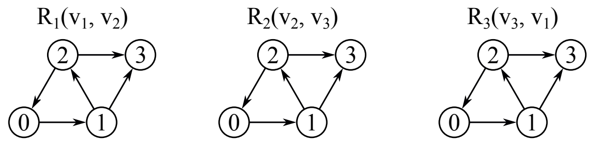
In the following section, we provide a brief overview of worst-case optimal joins and their key differences to traditional binary join plans. In the remainder of this paper, we consider natural join queries of the form

$$Q := R_1 \bowtie \cdots \bowtie R_m, \tag{1}$$

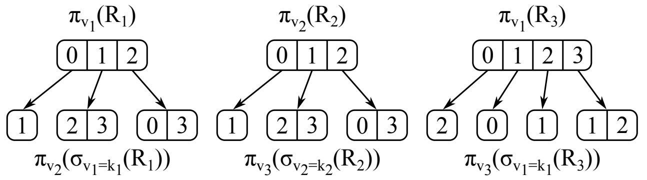
where the R_j are relations with attributes $v_1, \ldots, v_n$. Note that any inner join query containing only equality predicates can be transformed into this form by renaming attributes suitably. While most queries of this type can be processed efficiently by traditional binary join plans, query patterns 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

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

If we set $R_1 = R_2 = R_3$ and view tuples as edges in a graph, $Q_\triangle$ 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 [53]. The resulting large amount of redundant work will severely impact the overall query processing performance.



(a) Sample instances of the relations R_1, R_2, R_3. Each relation contains the tuples $(0,1), (1,2), (1,3), (2,0), (2,3)$ which are viewed as edges in a directed graph. The directed triangles in this graph are $(0,1,2), (1,2,0), (2,0,1)$.



(b) The trie structure induced by Algorithm 1 on these instances of R_1, R_2, R_3. Each recursive step conceptually iterates over the elements in the intersection between some trie nodes (line 5), and subsequently moves to the children of these elements (line 6).

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

Worst-case optimal join algorithms, on the other hand, avoid such exploding intermediate results [46]. 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 [46, 47]. It operates on the *query hypergraph* $H_Q = (V, \mathcal{E})$ of a query Q, where the vertex set V contains the attributes $\{v_1, \ldots, v_n\}$ of Q, and the edge set $\mathcal{E} = \{E_j \mid j = 1, \ldots, m\}$ contains the attribute sets of the individual relations R_j. In case of our running example $Q_\triangle$, 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 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.

Algorithm 1: Generic Worst-Case Optimal Join

```
given : A query hypergraph $H_Q = (V, \mathcal{E})$ with
        attributes $V = \{v_1, \ldots, v_n\}$ and hyperedges
        $\mathcal{E} = \{E_1, \ldots, E_m\}$.
input : The current attribute index
        $i \in \{1, \ldots, n+1\}$, and a set of relations
        $\mathcal{R} = \{R_1, \ldots, R_m\}$.
1 function enumerate($i$, $\mathcal{R}$)
2     if $i \leq n$ then
          // Relations participating in the current join
3         $\mathcal{R}_{join} \leftarrow \{R_j \in \mathcal{R} \mid v_i \in E_{R_j}\}$ ;
          // Relations unaffected by the current join
4         $\mathcal{R}_{other} \leftarrow \{R_j \in \mathcal{R} \mid v_i \notin E_{R_j}\}$ ;
          // Key values appearing in all joined relations
5         foreach $k_i \in \bigcap_{R_j \in \mathcal{R}_{join}} \pi_{v_i}(R_j)$ do
              // Select matching tuples
6             $\mathcal{R}_{next} \leftarrow \{\sigma_{v_i = k_i}(R_j) \mid R_j \in \mathcal{R}_{join}\}$ ;
              // Recursively enumerate matching tuples
7             enumerate($i + 1$, $\mathcal{R}_{next} \cup \mathcal{R}_{other}$) ;
8     else
          // Produce result tuples
9         produce($\bigtimes_{R_j \in \mathcal{R}} R_j$) ;
```

$i > 1$), the backtracking nature of the algorithm entails that a specific assignment for the join keys $v_1, \ldots, v_{i-1}$ 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 [9,46,47]. Given the query hypergraph H_Q of Q as defined above, we consider an arbitrary *fractional edge cover* $\mathbf{x} = (x_1, \ldots, x_m)$ of H_Q [47], which is defined by $x_j > 0$ for all $j \in \{1, \ldots, m\}$ and $\sum_{v_i \in E_j} x_j \geq 1$ for all $v_i \in V$. Then this bound states that

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

and the worst-case output size of Q can be determined by minimizing the right-hand size of Inequality 2 [47]. A join algorithm for computing Q is defined to be *worst-case optimal* if its runtime is proportional to this worst-case output size [46,47]. In case of our running example $Q_\triangle$, 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 $Q_\triangle$ [3,47].

Central to the analysis of the runtime complexity of worst-case optimal joins is the *query decomposition lemma* proved by Ngo et al. [47] 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 $\mathbf{x} = (x_1, \ldots, x_m)$ of the query hypergraph. Algorithm 1 as shown here is a special case of this generic algorithm [47].

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 are required for the algorithm to be worst-case optimal [46,47]. However, this requirement for index structures poses a considerable practical challenge. The order in which the join keys v_i of a query are processed heavily influences the performance of Algorithm 1 [2]. Depending on the query and its optimal join key order, indexes are required on different permutations of attributes from the input relations. The number of such permutations is usually much too large to store the corresponding indexes persistently. Therefore, they have to be built on-the-fly during query processing, precluding any expensive precomputation of the indexes themselves. Moreover, a general-purpose RDBMS and in particular an HTAP database has to support changing data. This makes it difficult to precompute data structures that could be reused across indexes. For instance, EmptyHeaded and LevelHeaded rely heavily on a suitable dense dictionary encoding of the join attribute values which is hard to maintain in the presence of changing data [2,3].

At the same time, the overall runtime of Algorithm 1 is dominated by the set intersection computation in line 5 which has to be implemented using these trie indexes [3]. While traditional B$^+$-trees or plain sorted lists are comparably cheap to build, they exhibit poor performance on this computation. The read-optimized data structures employed by EmptyHeaded and LevelHeaded can perform orders of magnitude better, but as outlined above are far too expensive to build on-the-fly [3,8,13]. For example, we measured in Section 5 that EmptyHeaded spends up to two orders of magnitude more time on precomputation than on actual join processing [3]. In contrast, our hash trie index structure proposed in Section 3 is much cheaper to build while still offering competitive join processing performance.

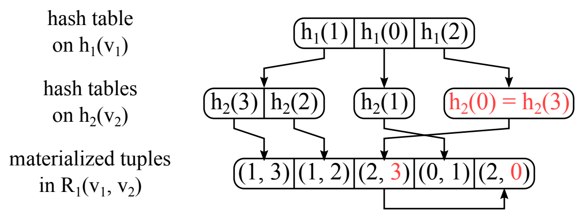


Figure 2: Illustration of a hash trie on the relation $R_1(v_1, v_2)$ shown in Figure 1. The example contains a collision between $h_2(0)$ and $h_2(3)$ (marked in red).

Finally, binary join processing has been studied and optimized for decades, leading to excellent performance on a wide range of queries. Even efficiently implemented worst-case optimal join algorithms frequently fail to achieve the same performance on queries that do not contain growing joins [2]. For instance, even when disregarding precomputation cost, the highly optimized LevelHeaded system is outperformed by HyPer by up to a factor of two on selected TPCH queries [2, 31]. Moreover, we measured that the Umbra RDBMS which employs binary join plans outperforms a commercial database system that relies on worst-case optimal joins by up to four orders of magnitude on the well-known TPCH and JOB benchmarks (cf. Section 5 and Figure 5) [38, 44]. Therefore, we propose a hybrid query optimization approach that only replaces binary joins with growing intermediate results by worst-case optimal joins, as we expect a tangible benefit in this case (cf. Section 4).

3. MULTI-WAY HASH TRIE JOINS

In this section, we present our hash-based worst-case optimal join algorithm. The workhorse of this approach is a novel *hash trie* data structure which is carefully designed to fulfill the requirements identified above.

3.1 Outline

Conceptually, the trie structure required by Algorithm 1 can be modeled easily through nested hash tables, where each level of nesting corresponds to exactly one join key attribute [54]. The path to a nested hash table then determines a unique prefix of join key values, and the nested hash table itself stores the distinct values of the corresponding join key attribute that appear in tuples with this prefix. On the last level, the hash tables store some sort of tuple identifiers that allow access to the tuple payload. The set intersections required by the worst-case optimal join algorithm can then trivially be computed in linear time, and the tuples matching a specific join key value can be selected by a single constant-time hash table lookup.

However, a straightforward implementation of this approach will suffer from suboptimal performance due to the substantial overhead incurred by each hash table lookup. Most importantly, every successful lookup into a hash table involves at least one key comparison in order to detect and eliminate hash collisions. This requires that the actual key values are accessible from the hash table buckets, and consequently, we either have to follow a pointer to the actual tuple on each hash table lookup, or the key values have to be stored within the buckets themselves. In either case, the cache performance of lookup operations will suffer considerably even if the actual key comparison function is cheap. Variable-length join keys such as strings further exacerbate this problem [55]. Finally, Algorithm 1 will generally produce many tentative matches in the upper levels of the tries that are later rejected because no corresponding matches exist on the lower levels, each of which still requires at least one key comparison.

The proposed *hash trie* data structure is based on the core insight that this key comparison can be deferred until the actual result tuples are enumerated by the join algorithm. Specifically, we modify Algorithm 1 to operate exclusively on the hash values of join keys, i.e. enumerate all tuples for which the *hash values* instead of the actual values of the join keys match. As a result, the corresponding trie structures will also be built on the hash values instead of the actual values of the join keys (cf. Figure 2). Of course, this enumeration will now include some false positives due to hash collisions, but we eliminate these false positives by verifying the actual join condition just before producing a result tuple (line 9 in 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, 52].

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 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 $R_1(v_1, v_2)$ of Q_Δ 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, this hash trie data structure is built on each input relation R_j of the join query Q. For a given

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
           // Allocate hash table memory
3          $M \leftarrow$ allocateHashtable($2^{\lceil \log_2(1.25 \cdot |L|) \rceil}$) ;
           // Build outer hash table
4          while $L$ is not empty do
5              $\mathbf{t} \leftarrow$ pop next tuple from $L$ ;
6              $B \leftarrow$ lookupBucket($M$, $h_i(\pi_{v_i}(\mathbf{t}))$) ;
7              push $\mathbf{t}$ onto the linked list stored in $B$ ;
           // Build nested hash tables
8          $i_{next} \leftarrow$ index of the next attribute in $E_j$ ;
9          foreach populated bucket $B$ in $M$ do
10             $L_{next} \leftarrow$ extract linked list stored in $B$ ;
11             $M_{next} \leftarrow$ build($i_{next}$, $L_{next}$) ;
12             store $M_{next}$ in $B$ ;
13         return($M$) ;
14     else
           // All attributes in $E_j$ have been processed
15         return($L$) ;
```

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.

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

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

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. From a high-level point of view it 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. 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 1. *The build phase of the proposed approach has time and space complexity in $O(n \cdot \sum_{E_j \in \mathcal{E}} |R_j|)$.*

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. $\square$

At this point, it is important to recall that we intend to integrate our approach 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, used for example in the Datalog query language [3, 46]. We

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

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. Subsequently, we informally motivate how this worst-case optimality under set semantics translates to bag semantics.

In preparation for our subsequent analysis, some additional notation is required. As outlined above, a hash trie iterator $I_j \in \mathcal{I}$ always points to a specific node within a hash trie. We write $R(I_j)$ to identify the set of tuples stored in the leaves of the subtrie rooted in this node, and $H(I_j)$ to identify the set of hashed join keys that are present in these tuples. For example, let I_j point to a bucket in the inner node containing $h_2(2)$ and $h_2(3)$ in Figure 2. Then $R(I_j)$ consists of the tuples $(1,2)$ and $(1,3)$, while $H(I_j)$ consists of the tuples $(h_1(1), h_2(2))$ and $(h_1(1), h_2(3))$. Clearly, $|H(I_j)| \leq |R(I_j)|$ holds for any hash trie iterator I_j. In the following, we will view $H(I_j)$ as a relation with the same attribute names as the corresponding $R(I_j)$.

Theorem 2. *Let $\mathbf{x} = (x_1, \dots, x_m)$ be an arbitrary fractional edge cover of the query hypergraph H_Q. Then Algorithm 3 has time complexity in $O(nm \prod_{E_j \in \mathcal{E}} |H(I_j)|^{x_j})$ and space complexity 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

$$|H(I_j)| \in \Theta(|R(I_j)|) \tag{3}$$

for any hash trie iterator I_j.

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 3, 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 $O(nm)$ 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.

For the inductive step, we observe that the recursive part of Algorithm 3 operates analogous to that of the generic worst-case optimal Algorithm 1. In particular, the loop in lines 6–15 runs for exactly size(I_{scan}) iterations, and its body in lines 10–14 is executed for each element k_i in the set intersection $\bigcap_{I_j \in \mathcal{I}_{join}} \pi_{v_i}(H(I_j))$. Invoking down on the participating iterators is equivalent to computing $\sigma_{v_i = k_i}(H(I_j))$ for $I_j \in \mathcal{I}_{join}$. Under these conditions, and given the base case runtime derived above, the proof provided by Ngo et al. for the runtime of Algorithm 1 yields that the runtime of Algorithm 3 is indeed in $O(nm \prod_{E_j \in \mathcal{E}} |H(I_j)|^{x_j})$ [47].

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 $|H(I_j)| \leq |R(I_j)|$ as outlined above, Theorem 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. Crucially, this expansion occurs *after* Algorithm 3 has determined that all tuples in this cross product are part of the join result, except of course for potential false positives due to hash collisions.

3.3 Implementation Details

In the following, we provide essential implementation details of the proposed approach, and a brief account of its integration into a compiling query execution engine [43].

3.3.1 Hash Trie Implementation

Figure 3 shows the memory layout of a hash trie as it is implemented within the Umbra system [44]. 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.

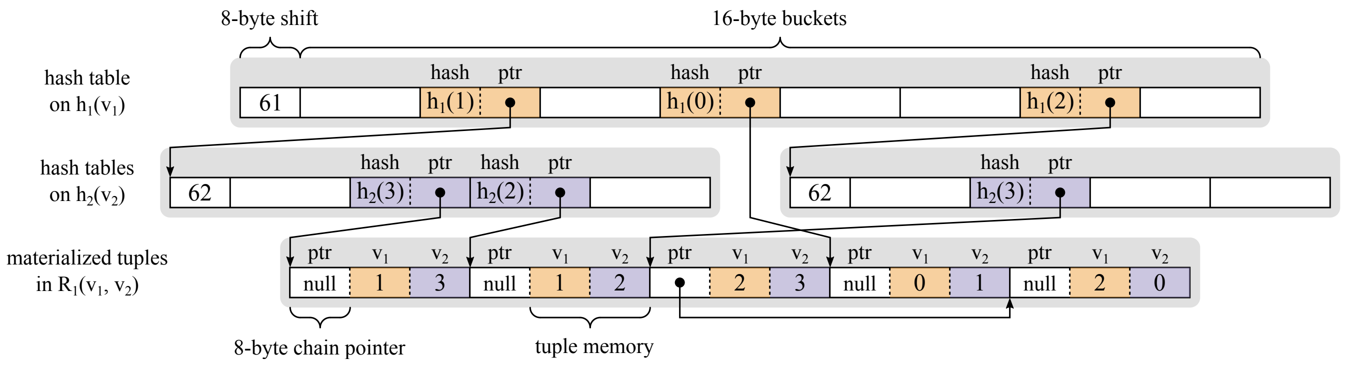


Figure 3: Memory layout of the hash trie in Figure 2. The gray boxes correspond to the individual hash tables and materialized input tuples. No nested hash table is built for the tuple $(0, 1)$ due to singleton pruning.

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.

We apply two further optimizations to the hash trie data structure, namely *singleton pruning* and *lazy child expansion*. Singleton pruning compresses paths in which every node has exactly one child, which occurs for the tuple $(0, 1)$ in Figure 3, for instance. With lazy child expansion, we only create the root nodes of the hash tries in the build phase, and create any nested hash tables on-demand when they are accessed for the first time during the probe phase. The latter optimization is especially beneficial in selective joins, where many nested hash tables are never accessed. We provide the full implementation details and a thorough evaluation of these additional optimizations in a supplemental technical report [15].

3.3.2 Build Phase

In the build phase, the incoming tuples are materialized contiguously in an in-memory buffer prior to running Algorithm 2. Tuples are stored using a fixed-length memory layout that is determined during query compilation time, in order to facilitate subsequent random tuple accesses. 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). As part of this 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, 37, 61].

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 [37].

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^+-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.

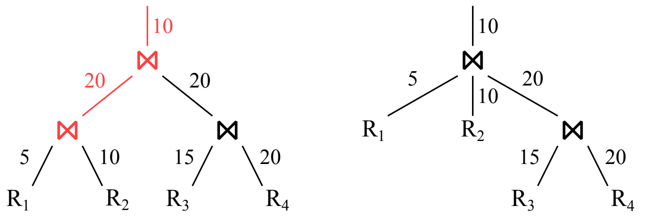
Even more aggressive optimizations are possible if the data is known to be static. In this case, it is actually desirable to perform as much precomputation as possible in order to minimize the time required to answer a query. While this obviates the need for data structures that can be built efficiently 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

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($T_r$) ;
      // Detect growing joins and multi-way join inputs
6     if $|T| > \max(|T'_l|, |T'_r|) \vee T'_l \neq T_l \vee T'_r \neq T_r$ then
7         return collapseMultiwayJoin($T'_l \bowtie T'_r$) ;
8     return $T'_l \bowtie T'_r$ ;
```



(a) Binary join plan **(b) Hybrid join plan**

Figure 4: 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)).

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 [24], 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 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 4). 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 [42].

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 strategy over other alternatives [2,3, 8,42], as its cost estimates rely only on cardinality information that is already maintained within Umbra [17], 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, 18].

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 [44]. We will subsequently refer to the corresponding system configuration as Umbra$^{\mathrm{OHT}}$. 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$^{\mathrm{EAG}}$.

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 [23, 44]. 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$^{\mathrm{LFT}}$), based on dense sorted arrays that are built during query

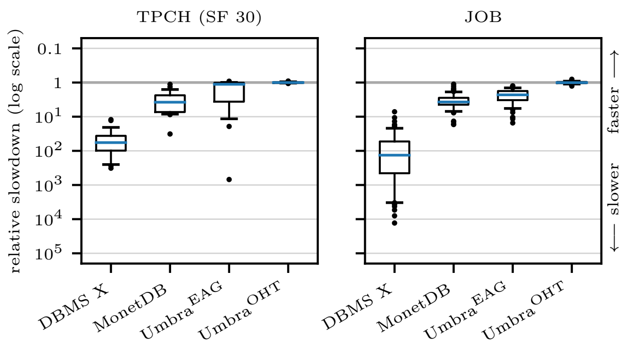


Figure 5: Relative slowdown of the different systems in comparison to binary join plans within Umbra on TPCH and JOB. The boxplots show the 5th, 25th, 50th, 75th, and 95th percentiles.

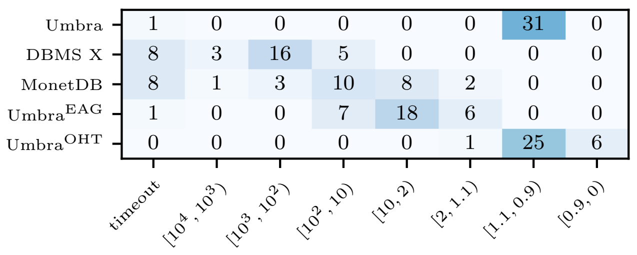


Figure 6: Histogram of the relative slowdown of the different systems in comparison to binary join plans within Umbra on JOB without filter predicates.

processing using the native parallel sort operator of Umbra [54,56]. Our preliminary experiments showed that using sorted arrays within the Umbra^{LFT} system is consistently faster than using the B$^+$-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 [38], and the TPCH benchmark at scale factor 30. Furthermore, we run a set of graph-pattern queries on selected network datasets which have been used extensively in previous work [40, 48]. In particular, we choose the Wikipedia vote network [39], the Epinions and Slashdot social networks [41, 51], the much larger Google+ and Orkut user networks [11, 58], as well as the Twitter follower network which is one of the largest publicly available network data sets [36]. 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 [3,6,22,42,48], we focus on undirected 3 and 4-clique queries on these graphs as they are a common subpattern in graph workloads [48]. The queries used in our experiments are available online [16].

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 TPCH and JOB. EmptyHeaded is excluded in this experiment as it does not support the complex analytical queries in these benchmarks.

Here, the unmodified version of Umbra that relies purely on binary join plans outperforms all other systems except for the Umbra^{OHT} system which employs our novel hybrid optimization strategy. The relative slowdown of these systems in comparison to Umbra is shown in Figure 5. The key observation in this benchmark is that the Umbra^{EAG} system which eagerly replaces all binary joins by worst-case optimal joins consistently outperforms both DBMS X and MonetDB. These results 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^{OHT} 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^{OHT} system exhibits the best overall performance, improving over Umbra by a factor of 1.9× and over Umbra^{EAG} by a factor of 4.2×.

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 6 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^{EAG} system are able to match the performance of the unmodified version of Umbra, by a similar margin as in the previous experiment. This indicates

Table 2: Absolute runtime in seconds of the graph pattern queries on the small network data sets.

		Wiki	Epinions	Slashdot
3-clique	*EH-Probe*	*0.28*	*0.30*	*0.29*
	EmptyHeaded	0.43	0.79	1.07
	DBMS X	0.28	0.52	1.37
	Umbra	**0.03**	**0.06**	**0.08**
	Umbra$^{\mathrm{LFT}}$	0.36	0.53	0.49
	Umbra$^{\mathrm{OHT}}$	**0.04**	**0.07**	**0.07**
4-clique	*EH-Probe*	*0.40*	*0.55*	*0.47*
	EmptyHeaded	0.55	1.04	1.24
	DBMS X	1.66	6.53	13.95
	Umbra	1.61	12.04	7.91
	Umbra$^{\mathrm{LFT}}$	3.82	7.02	4.09
	Umbra$^{\mathrm{OHT}}$	**0.10**	**0.23**	**0.18**

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. In contrast, the Umbra$^{\mathrm{OHT}}$ system with our hybrid query optimizer matches or improves over the performance of Umbra, by identifying six 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$^{\mathrm{OHT}}$ 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 are possible [3, 48].

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 2. 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$^{\mathrm{OHT}}$ system consistently exhibits the best runtime across all data sets and queries, outperforming the remaining systems by up to two orders of magnitude. 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 we observed to be essentially static on this benchmark, and its expensive precomputation step. The multi-way join implementations of DBMS X and Umbra$^{\mathrm{LFT}}$ 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 4-clique query.

Finally, we run the 3-clique query on the much larger Google+, Orkut, and Twitter graph data sets (cf. Table 3).

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

	Google+	Orkut	Twitter
EH-Probe	*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$^{\mathrm{LFT}}$	14.55	30.61	1 175.97
Umbra$^{\mathrm{OHT}}$	**7.70**	**15.25**	**579.07**

Once again, the Umbra$^{\mathrm{OHT}}$ 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 that were presented above, and study the applicability of worst-case optimal joins within a general-purpose RDBMS in detail. Traditional relational workloads such as TPCH 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 particular, we choose two parameters $r \in \{10^4, 10^5, 10^6, 10^7\}$ and $d \in \{1, \ldots, 10\}$, and generate randomly shuffled relations R, S, and T as follows. R simply contains the distinct integers $1, \ldots, 10^7$, while S and T contain the distinct integers $1, \ldots, (10^7 + r)/2$ and $(10^7 - r)/2, \ldots, 10^7$ respectively. 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 7 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 hash table lookups. In contrast, a hash trie join determines once that the distinct value is part of the join

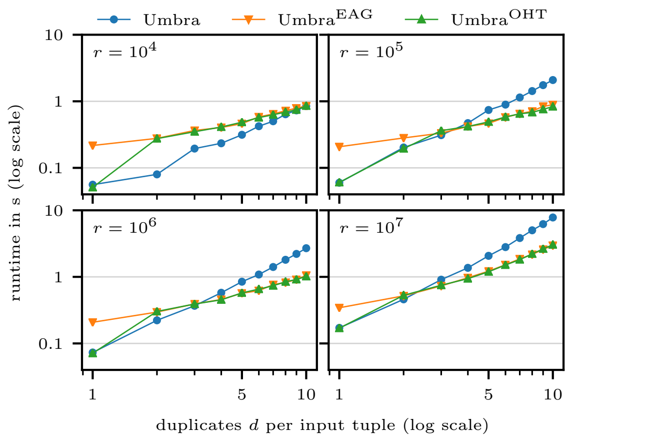


Figure 7: Absolute query runtime on the synthetic query $R \bowtie S \bowtie T$ as the number of distinct values r and duplicated tuples d in the query result is varied.

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^{OHT} 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 [24, 38]. Crucially, 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. 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 sort these decisions into true and false positives, respectively negatives, where the correct introduction of a multi-way join is counted as a true positive.

Table 4: 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	1 285	119	1 066	18	82

An overview of the results is shown in Table 4. As discussed previously, growing joins are exceptionally rare in traditional relational workloads like TPCH 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 [24, 38]. 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×. 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 7 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× 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.

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, 19, 30, 60]. 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, 19, 30]. 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 [19, 30], 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 [45, 46, 47], 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, 25, 26, 29, 32, 57], stronger optimality guarantees [5, 33, 34, 45], and incremental maintenance of the required data structures [27, 28], 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, 54, 56]. Variants of such join algorithms have been adopted in distributed query processing [4, 6, 13, 35] graph processing [3, 6, 22, 42, 59, 62], 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. Persistent precomputed index structures are only feasible in limited numbers, e.g. in specialized graph processing or RDF engines [3, 22, 42]. One could 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 techniques could also be applied to this domain, e.g. by integrating them into the approach developed 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 [54]. 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 [49] and the data structure used in extendible hashing schemes [20]. 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 [50].

A key contribution of this paper is a comprehensive implementation of our approach within the general-purpose Umbra RDBMS [44]. 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, 27]. 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 its details. Approaches that holistically optimize hybrid join plans have been proposed for graph processing [42, 62], 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 [19, 21, 30]. 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.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for critically reading the initial manuscript and suggesting detailed improvements. 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).

9. REFERENCES

[1] C. Aberger. EmptyHeaded GitHub repository. `https://github.com/HazyResearch/EmptyHeaded`.

[2] C. R. Aberger, A. Lamb, K. Olukotun, and C. Ré. LevelHeaded: A unified engine for business intelligence and linear algebra querying. In *ICDE*, pages 449–460. IEEE Computer Society, 2018.

[3] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. EmptyHeaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, 2017.

[4] F. N. Afrati and J. D. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Trans. Knowl. Data Eng.*, 23(9):1282–1298, 2011.

[5] K. Alway, E. Blais, and S. Salihoglu. Box covers and domain orderings for beyond worst-case join processing. *CoRR*, abs/1909.12102, 2019.

[6] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using worst-case optimal and low-memory dataflows. *PVLDB*, 11(6):691–704, 2018.

[7] A. Appleby. Murmurhash GitHub repository. `https://github.com/aappleby/smhasher`.

[8] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the LogicBlox system. In *SIGMOD Conference*, pages 1371–1382. ACM, 2015.

[9] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.

[10] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD Conference*, pages 261–272. ACM, 2000.

[11] L. Backstrom, D. P. Huttenlocher, J. M. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *KDD*, pages 44–54. ACM, 2006.

[12] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, pages 362–373. IEEE Computer Society, 2013.

[13] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD Conference*, pages 63–78. ACM, 2015.

[14] A. Fekete, B. Franks, H. Jordan, and B. Scholz. Worst-case optimal radix triejoin. *CoRR*, abs/1912.12747, 2019.

[15] M. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann. Combining worst-case optimal and traditional binary join processing. Technical Report TUM-I2082, Technische Universität München, 2020.

[16] M. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann. Queries used in the experimental evaluation, Jan. 2020. `https://github.com/freitmi/queries-vldb2020`.

[17] M. Freitag and T. Neumann. Every row counts: Combining sketches and sampling for accurate group-by result estimates. In *CIDR*, 2019.

[18] G. Gottlob, M. Grohe, N. Musliu, M. Samer, and F. Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In *WG*, volume 3787 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2005.

[19] G. Graefe, R. Bunker, and S. Cooper. Hash joins and hash teams in microsoft SQL server. In *VLDB*, pages 86–97. Morgan Kaufmann, 1998.

[20] S. Helmer, R. Aly, T. Neumann, and G. Moerkotte. Indexing set-valued attributes with a multi-level extendible hashing scheme. In *DEXA*, volume 4653 of *Lecture Notes in Computer Science*, pages 98–108. Springer, 2007.

[21] M. Henderson and R. Lawrence. Are multi-way joins actually useful? In *ICEIS (1)*, pages 13–22. SciTePress, 2013.

[22] A. Hogan, C. Riveros, C. Rojas, and A. Soto. A worst-case optimal join algorithm for SPARQL. In *ISWC (1)*, volume 11778 of *Lecture Notes in Computer Science*, pages 258–275. Springer, 2019.

[23] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[24] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD Conference*, pages 268–277. ACM Press, 1991.

[25] M. Joglekar, R. Puttagunta, and C. Ré. Aggregations over generalized hypertree decompositions. *CoRR*, abs/1508.07532, 2015.

[26] M. R. Joglekar, R. Puttagunta, and C. Ré. AJAR: aggregations and joins over annotated relations. In *PODS*, pages 91–106. ACM, 2016.

[27] O. Kalinsky, Y. Etsion, and B. Kimelfeld. Flexible caching in trie joins. In *EDBT*, pages 282–293. OpenProceedings.org, 2017.

[28] A. Kara, H. Q. Ngo, M. Nikolic, D. Olteanu, and H. Zhang. Counting triangles under updates in worst-case optimal time. In *ICDT*, volume 127 of *LIPIcs*, pages 4:1–4:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

[29] A. Kara and D. Olteanu. Covers of query results. In *ICDT*, volume 98 of *LIPIcs*, pages 16:1–16:22. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[30] A. Kemper, D. Kossmann, and C. Wiesner. Generalised hash teams for join and group-by. In *VLDB*, pages 30–41. Morgan Kaufmann, 1999.

[31] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[32] M. A. Khamis, R. R. Curtin, B. Moseley, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. On functional aggregate queries with additive inequalities. In *PODS*, pages 414–431. ACM, 2019.

[33] M. A. Khamis, H. Q. Ngo, C. Ré, and A. Rudra. Joins via geometric resolutions: Worst case and beyond. *ACM Trans. Database Syst.*, 41(4):22:1–22:45, 2016.

[34] M. A. Khamis, H. Q. Ngo, and A. Rudra. FAQ: questions asked frequently. In *PODS*, pages 13–28. ACM, 2016.

[35] P. Koutris, P. Beame, and D. Suciu. Worst-case optimal algorithms for parallel query processing. In

ICDT, volume 48 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[36] H. Kwak, C. Lee, H. Park, and S. B. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600. ACM, 2010.

[37] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*, pages 743–754. ACM, 2014.

[38] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[39] J. Leskovec, D. P. Huttenlocher, and J. M. Kleinberg. Signed networks in social media. In *CHI*, pages 1361–1370. ACM, 2010.

[40] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection, June 2014. `http://snap.stanford.edu/data`.

[41] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[42] A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *PVLDB*, 12(11):1692–1704, 2019.

[43] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[44] T. Neumann and M. Freitag. Umbra: A disk-based system with in-memory performance. In *CIDR*, 2020.

[45] H. Q. Ngo, D. T. Nguyen, C. Ré, and A. Rudra. Beyond worst-case analysis for joins with minesweeper. In *PODS*, pages 234–245. ACM, 2014.

[46] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3):16:1–16:40, 2018.

[47] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Record*, 42(4):5–16, 2013.

[48] D. T. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra. Join processing for graph patterns: An old dog with new tricks. In *GRADES@SIGMOD/PODS*, pages 2:1–2:8. ACM, 2015.

[49] A. Prokopec, P. Bagwell, and M. Odersky. Lock-free resizeable concurrent tries. In *LCPC*, volume 7146 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 2011.

[50] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Brief announcement: Prefix hash tree. In *PODC*, page 368. ACM, 2004.

[51] M. Richardson, R. Agrawal, and P. M. Domingos. Trust management for the semantic web. In *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 351–368. Springer, 2003.

[52] J. A. Rogers. AquaHash GitHub repository. `https://github.com/jandrewrogers/AquaHash`.

[53] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *WEA*, volume 3503 of *Lecture Notes in Computer Science*, pages 606–609. Springer, 2005.

[54] T. L. Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, pages 96–106, 2014.

[55] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Mühlbauer, T. Neumann, and M. Then. Get real: How benchmarks fail to represent the real world. In *DBTest@SIGMOD*, pages 1:1–1:6. ACM, 2018.

[56] H. Wu, D. Zinn, M. Aref, and S. Yalamanchili. Multipredicate join algorithms for accelerating relational graph processing on GPUs. In *ADMS@VLDB*, pages 1–12, 2014.

[57] K. Xirogiannopoulos and A. Deshpande. Memory-efficient group-by aggregates over multi-way joins. *CoRR*, abs/1906.05745, 2019.

[58] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *ICDM*, pages 745–754. IEEE Computer Society, 2012.

[59] W. Zhang, R. Cheng, and B. Kao. Evaluating multi-way joins over discounted hitting time. In *ICDE*, pages 724–735. IEEE Computer Society, 2014.

[60] X. Zhang, L. Chen, and M. Wang. Efficient multi-way theta-join processing using mapreduce. *PVLDB*, 5(11):1184–1195, 2012.

[61] Z. Zhang, H. Deshmukh, and J. M. Patel. Data partitioning for in-memory systems: Myths, challenges, and opportunities. In *CIDR*, 2019.

[62] G. Zhu, X. Wu, L. Yin, H. Wang, R. Gu, C. Yuan, and Y. Huang. HyMJ: A hybrid structure-aware approach to distributed multi-way join query. In *ICDE*, pages 1726–1729. IEEE, 2019.