Data Science and Engineering (2026) 11:143–154
https://doi.org/10.1007/s41019-025-00325-7

RESEARCH PAPERS

SieveJoin: Boosting Multi-way Joins by Filtering Unneeded Intermediate Results

Renrui Li[1] · **Qingzhi Ma**[1] · **Xiaomeng Shi**[2] · **An Liu**[1]

Received: 2 August 2025 / Revised: 10 October 2025 / Accepted: 28 October 2025 / Published online: 28 December 2025

Abstract

Improving the performance of data systems for join operations has long been a critical challenge. Recently, substantial attention has been focused on optimizing multi-way join performance, particularly in reducing the overhead caused by generating intermediate tuples that do not contribute to the final result. In this paper, we propose a novel algorithm called SieveJoin, which extends the established Bloomjoin approach to support multi-way joins. SieveJoin sets a new benchmark for the efficiency of join query execution. A key innovation of SieveJoin is its ability to propagate Bloom filters along the join path, allowing the system to terminate early and avoid producing superfluous intermediate results. The primary design objective of SieveJoin is to efficiently estimate join results using Bloom filters, while maintaining minimal memory overhead. We analyze the bottlenecks associated with deferred multi-way joins and detail how Bloom filters are utilized to suppress the creation of redundant intermediate tuples. To assess the effectiveness of SieveJoin, we conduct a comprehensive experimental evaluation using the TPC-H benchmark, citation datasets, and a synthetic dataset. Our results compare SieveJoin with a state-of-the-art column-store database and a worst-case optimal join algorithm, highlighting its advantages in both response time and memory usage.

Keywords Multi-way join · Worst-case optimal join · Query processing · Bloom filter

1 Introduction

The majority of database systems predominantly utilize binary join plans to process multi-way joins [1–3]. Binary join plans execute the join operation on two relations at a time and have been extensively studied and optimized over several decades [4–6]. While binary joins are flexible and robust across many scenarios, they are theoretically suboptimal [7, 8]. Recent studies [9–12] have shown that they often produce excessive intermediate tuples that do not contribute to the final result, leading to wasted computation and higher query latency.

In contrast, recent research on multi-way joins aims to circumvent the generation of useless intermediate join results by simultaneously accessing multiple tables [13–15]. These efforts have been applied or considered in commercial database systems [9–11, 16, 17]. However, some have been abandoned due to stability concerns and performance issues [16]. Theoretical advancements in database systems have led to the development of worst-case optimal join algorithms (WCOJ), such as the leapfrog algorithm [10, 18, 19], the Umbra database system [11], among others [12, 20]. Despite their potential to outperform binary joins by filtering intermediate join results, these methods exhibit notable disadvantages, limiting their adoption into database systems.

Another category of efforts to enhance join processing involves the use of Bloom filters. A Bloom filter [21–24] is a space-efficient data structure for fast membership checks, allowing false positives but prohibiting false negatives. In the context of table joins, if join results could be pruned in

✉ Qingzhi Ma
qzma@suda.edu.cn

Renrui Li
rrli@stu.suda.edu.cn

Xiaomeng Shi
Xiaomeng.Shi@xjtlu.edu.cn

An Liu
anliu@suda.edu.cn

[1] School of Computer Science and Technology, Soochow University, Suzhou, China

[2] IBSS, Xi'an Jiaotong-Liverpool University, Suzhou, China

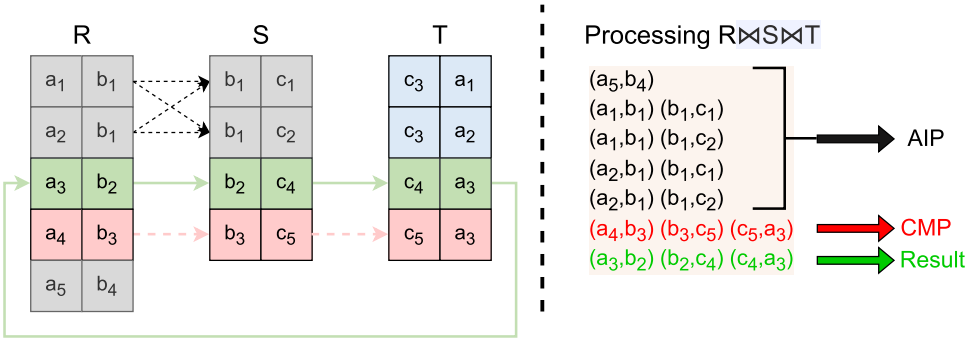
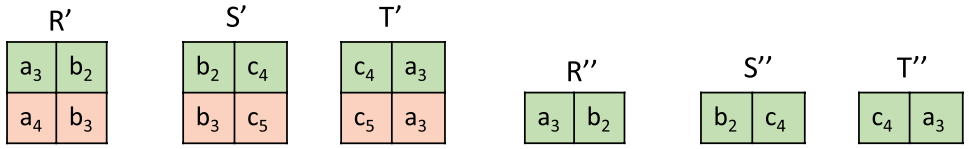


Fig. 1 Types of intermediate join results. The background colors of tuples indicate whether they make it into the final result. Gray indicates tuples involved in an incomplete join path or AIP. The red end tuple represents a cyclic mismatch path, where the join path is complete but does not meet the final join conditions in a cyclic join



(a) Pruned Relations without AIP (b) Pruned Relations without AIP and CMP

Fig. 2 Types of pruned tables

advance to filter out useless intermediate join results, the total query processing time would be significantly reduced. Previous efforts, including the Bloomjoin algorithm [25, 26], spectral bloom filters [27], bloom filter operations [28, 29], etc [30, 31], have been limited to binary joins or could only be applied to the same join attribute. These methods have shown more potential in reducing data transferred during query processing than in query optimization.

1.1 Intermediate Join Results

As mentioned earlier, binary joins are not always the ideal choice for multi-way queries as they may generate a large number of intermediate join results. Here, we will provide a detailed explanation of the different types of intermediate join results. Consider, for example, a query in the following form:

$$Q := R(a,b) \bowtie_b S(b,c) \bowtie_c T(c,a) \tag{1}$$

with the join conditions $R.b = S.b$ and $S.c = T.c$, where R, S, and T are relations with attributes a, b, or c. The results in the final join result fall into the form $(r(a_i, b_i), s(b_j, c_j), t(c_k, a_k))$, referred to as a complete join path, denoted as (r, s, t). This is termed 'complete' as it involves tuples from all joined tables. Similarly, (r, s) is an incomplete join path. In addition, this acyclic query can be extended to perform a cyclic query, with the join conditions $R.b = S.b$ and $S.c = T.c$ and $T.a = r.a$.

1.1.1 Acyclic Incomplete Path (AIP)

Figure 1 illustrates the intermediate join results during the processing of the cyclic query Q that extended to cyclic query. The majority of join paths are incomplete, like (r) or (r,s) (refer to the join paths in gray in Fig. 1). We term such join path as Acyclic Incomplete Join Path (**AIP**). Notably, 71% (5 out of 7) of the join paths are incomplete.

Acyclic incomplete paths are common across workloads and inevitable with binary join plans. In low-selectivity queries, they lead to significant overhead from processing intermediate results that never reach the final output. As illustrated in Fig. 2a, pruning such paths early by filtering table entries or indexes can greatly accelerate query execution. For example, reducing table R to R' eliminates 60% of its tuples (3 out of 5). Notably, acyclic incomplete paths arise in both cyclic and acyclic joins.

1.1.2 Cyclic Mismatch Path (CMP)

For cyclic joins, intermediate join results are more complex. Apart from AIPs, many complete join paths do not satisfy the cyclic join conditions and are discarded during the final condition checks, such as the red tuple shown in Fig. 1. Although they do not meet the join requirements, they are generated during query processing. Though ultimately invalid, these paths are still generated during processing. We term them Cyclic Mismatch Path (**CMP**).

As shown in Fig. 2b, eliminating CMPs can further reduce relation sizes. CMPs do not exist for acyclic joins but usually dominate the exploding intermediate join results for cyclic joins. To address the issues caused by AIP and CMP, this work proposes a new method, coined SieveJoin, to prune the relations based on Bloom filters.

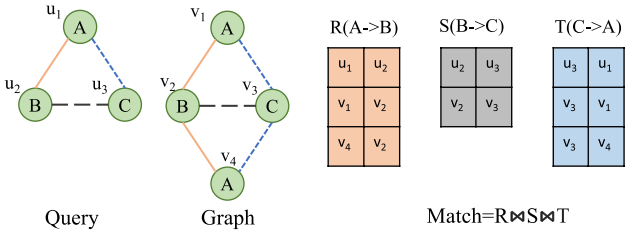


Fig. 3 The graph matching problem

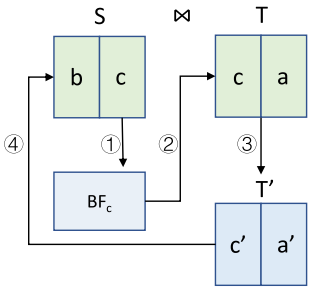


Fig. 4 The bloomjoin algorithm

1.2 Challenge for Complex Queries

Multi-way joins find applications in graph querying tasks, including the graph matching problem [20, 32–34], the clique problem [35, 36], etc. Take the graph matching problem as an example. As shown in Fig. 3, finding the matching graphs of *Query* within *Graph* could be obtained by processing the corresponding conjunctive query $R(a,b) \bowtie S(b,c) \bowtie T(c,a)$ using RDMSs [37–39], where R, S, and T are edge relations representing links AB, BC, and CA, respectively.

This method is straightforward, albeit arguably time-consuming, as the edge relations are typically very large. However, typically the number of matching graphs is relatively small, and most of the time is wasted on generating useless intermediate join results.

The situation for 3-clique/4-clique queries is similar. A {3–4}-clique query aims to find subgraphs with {3,4} nodes such that every two nodes are connected [40]. For instance, finding the 3-clique graph of relation R is equivalent to $R_1(a,b) \bowtie R_2(b,c) \bowtie R_3(c,a)$. Again, many intermediate join results are generated during query processing. Unfortunately, as we shall see, there is still much to be done with respect to improving the efficiency and space overheads when removing unnecessary intermediate results in multi-way join queries. We propose SieveJoin, a Bloom-filter-based method to address this.

1.3 Contributions

This paper presents the design and implementation of SieveJoin, a join-boosting method addressing the aforementioned needs. It is based on using prebuilt, a priori state (i.e., Bloom filters), while offering, compared to the state of the art:

- 1–3 orders of magnitude shorter query response times,
- significantly smaller memory overheads (with very small Bloom filters),
- an easy-to-implement method for existing database systems,
- support for scenarios with frequent insertions/updates.

This paper will:

- Propose a new easy-to-implement method to boost query processing by eliminating both AIP paths and CMP paths based on Bloom filters,
- Analyze the abilities of SieveJoin for pruning table sizes for multiple joins,
- Perform a comprehensive performance evaluation of SieveJoin, comparing it against state-of-the-art RDBMs(Umbra, MonetDB) using queries based on the TPC-H datasets, a synthetic dataset, and other open-source graph datasets.

The remainder of this paper is organized as follows: Sect. 2 provides background on multi-way joins and introduces the core algorithm behind SieveJoin, including Bloom filter propagation. Section 3 details the implementation, followed by experimental evaluation in Sect. 4. Section 5 reviews related work, and Sect. 6 concludes.

2 SieveJoin

2.1 The Bloomjoin Algorithm

In this section, we will recall the well-known Bloomjoin algorithm[25, 26], which inspired our work. It is designed for binary joins. Assume there are two relations S and T, with the join condition on $S.c = T.c$. Figure 4 shows the typical Bloomjoin algorithm, which consists of four steps.

1. **Generate a Bloom filter BF_c on attribute c within relation S.** The Bloom filter is a large vector of bits, which is initially set to 0. By the linear scanning and hashing of every tuple in Attribute c, the corresponding bit within BF_c is set to 1.
2. **Send BF_c to Relation T.** The size of the Bloom filter at most is 1 bit per tuple, which is significantly smaller than transferring the actual relation S.
3. **Hasing every tuple c_i of Attribute C in Relation T.** If BF_c.contains(c_i), send this tuple to Relation S as tuple

stream T'. The expected number of tuples in T', or the Bloomjoin cardinality BC of Relation T', is given by

$$BC = SC_T + bits_S(1 - e^{-\frac{\alpha D_T}{F}}) \quad (2)$$

where

$$\alpha = 1 - \frac{SC_T}{C_T} \quad (3)$$

is the fraction of non-matching tuples in T,

$$bits_S = F(1 - e^{-\frac{D_S}{F}}) \quad (4)$$

is the approximate number of bits set to '1' in BF_1 [41], SC_T is the semijoin cardinality of T, F is the size of BF_c, D_T is the number of distinct values of Attribute c in Relation T.

4. **Join the tuple stream T' to S, and return the join result to the user.**

The Bloomjoin algorithm is designed for binary(two-way) distributed equi-joins, and is exceptionally advantageous for cases where the message costs are high and any table remote from the join site is quite large. One could envisage a straightforward application of Bloomjoin for multi-way joins, as follows: Produce a binary join plan for the multi-way join and repeat the algorithm for each binary join in the plan. However, this is still a binary-join plan, suffering from all aforementioned drawbacks (e.g., AIP and CMP intermediate join paths).

2.2 The SieveJoin Algorithm

Motivated by the Bloomjoin algorithm, we propose the SieveJoin algorithm that extends the application of Bloom filters to multi-way joins. To simplify the problem, we will re-use relations R, S and T, as introduced in Fig. 1. Our target is to boost query processing of the multi-way acyclic join $Q := R \bowtie_b S \bowtie_c T$. Figure 5 shows the query execution flow of SieveJoin.

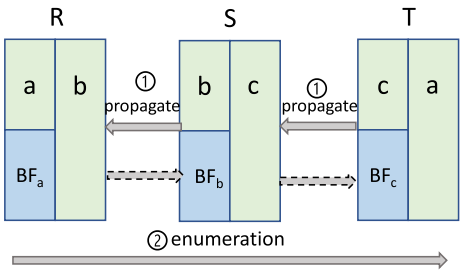


Fig. 5 SieveJoin's training flow

1. **Propagation Phase**. In this phase, Bloom filters are generated and propagated backward. Specifically, we generate a Bloom filter BF_c on $T.c$. After that, Relation S receives BF_c, which is used to generate Bloom filter BF_b on $S.b$.
   Not all tuples in S are used to generate BF_b, tuples that belong to S and occur in $S \bowtie_c T$ are used instead. Thus, Bloom filter BF_b is pruned to eliminate unnecessary intermediate join results. We repeat this propagation phase from S to R to generate Bloom filter BF_a on $R.a$.
2. **Enumeration Phase**. In this phase, pre-generated Bloom filters are employed to pre-filter all participating tables, effectively avoiding the generation of redundant intermediate results and facilitating the production of the final join results.

The propagation phase relies on pre-built Bloom filters to prune the current relation and/or Bloom filter. SieveJoin trains propagative Bloom filters via a linear scan of the target attribute while simultaneously probing relevant Bloom filters on the fly. The prune conditions are complex, and a detailed discussion is given in Sect. 2.3.

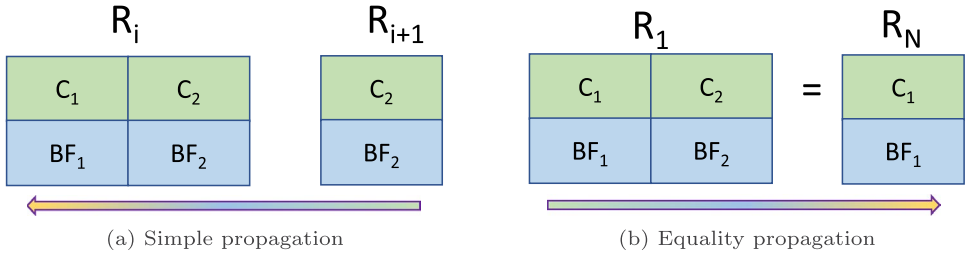
Note, SieveJoin adopts a simple one-pass backward propagation strategy. Another forward propagation phase (from leftmost to rightmost) might be applied to further prune relations on the right side, at the cost of doubling the training time. It has to be noted that the two-pass strategy produces more strict Bloom filters (and perfectly pruned tables for all), while the one-pass strategy produces less strict Bloom filters. (e.g. BF_c on the right side is not compact enough, while the BFs on the leftmost side are identical for both strategies.) Both strategies eliminate intermediate results and produce correct join results.

Suppose there are indexes built on join attributes, we could boost the propagation phase by accessing the indexes directly. In addition, for predictable workloads, such pre-built Bloom filters might be generated in advance for faster processing of such queries. SieveJoin is applicable to existing join algorithms, including nested loop join, index-based nested loop join [42, 43], hash join [44–46], other multi-way joins [9, 11], etc. In this study, we demonstrate SieveJoin based on binary join plans, assuming a given join order.

2.3 Propagation Conditions

The propagation of Bloom filters is the core component of SieveJoin for multi-way joins. It enables the elimination of useless intermediate join results, including AIPs and CMPs. In this section, we describe how to propagate Bloom filters for acyclic and cyclic queries.

Figure 6 lists the typical propagation conditions where the Bloom filters are generated for multi-way joins, which



(a) Simple propagation (b) Equality propagation

Fig. 6 Propagation conditions

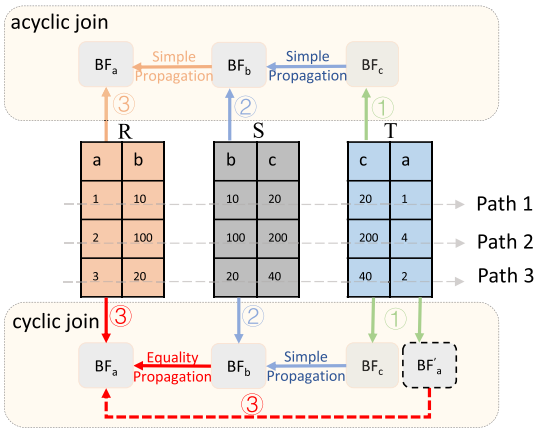


Fig. 7 Illustration of propagation of bloom filters

are simple propagation and equality propagation. Other more complex conditions are simply the combination or extension of them. For instance, a join query over a star schema on the same attribute is simply a combination of simple propagation.

2.3.1 Simple Propagation

Simple propagation represents the common condition to propagate Bloom filters for acyclic queries. In this example as shown in Fig. 6a, the query segment is $\ldots \bowtie R_i \bowtie R_{i+1} \bowtie \ldots$ on $R_i.C_2 = R_{i+1}.C_2$. The support for simple propagation is straightforward, and only a sequential scan of table R_i is needed.

Algorithm 1: Simple Propagation

Input: relation R_i, R_{i+1}, and Bloom filters $R_{i+1}.BF_2$.
Output: the propagated Bloom filter $R_i.BF_1$.

```
1: Initialize bloom filter R_i.BF_1 to '0s'
2: for each row (c_{1,j}, c_{2,j}) in R_i do
3:     if R_{i+1}.BF_2.contains(c_{2,j}) then
4:         Insert c_{1,j} to R_i.BF_1
5:     end if
6: end for
```

Line 3–4 is the 'approximate' equality check of the join condition, and tuples that contribute to the final join results are used to generate Bloom filters. In other words, tuples that do not satisfy the join conditions are removed to avoid intermediate join results. It is 'approximate' as Bloom filters allow a small portion of false positives.

2.3.2 Equality Propagation

Simple propagation is effective for acyclic queries. Take the 3-way join query $R(a,b) \bowtie S(b,c) \bowtie T(c,a)$ as an example (as shown in Fig. 1). As illustrated in Fig. 7, if it is an acyclic query, simple propagation suffices to eliminate unneeded intermediate join results. However, if the target query is cyclic, (which involves an extra equality condition on $R.a = T.a$), simple propagation does not perform well as the final equality condition is not met.

The solution is to generate another Bloom filter on T called $BF_a{}'$ to facilitate validation of the equality condition. In this way, we need to visit two Bloom filters to build the compact Bloom filter BF_a (one BF_b for propagation, and another $BF_a{}'$ for equality checks). Algorithm 2 summarises this procedure.

Algorithm 2: Equality Propagation

Input: relation R, T. Bloom filters BF_b, $BF_a{}'$.
Output: the propagated Bloom filter BF_a.

```
1: Initialize bloom filter BF_a to '0s'
2: for each row (a_j, b_j) in R do
3:     if b_j in BF_b and a_j in BF_a' then
4:         Insert a_j to BF_a
5:     end if
6: end for
```

2.3.3 Limitation of Equality Propagation

Bloom filters are designed for efficient existence checks but do not guarantee equality conditions. Thus, the equality propagation introduced previously approximates the equality condition checks by examining the tuple's existence instead. As illustrated in Fig. 7, equality propagation is able to eliminate join path 2 as it fails the existence checks. However, path 3 is unavoidable as a=2 exists in both R.a and T.a, and equality propagation is not able to handle such kind of situation.

In spite of this drawback, equality propagation still eliminates a large number of CMPs, which will be demonstrated in the experimental part. In addition, we could remove

CMPs completely, by producing actual join results and re-generating Bloom filters for equality propagation. This procedure takes a longer time but works well for scenarios with less frequent updates.

2.4 Limitations

SieveJoin is more advantageous for popular queries with expected workloads. For such queries, Bloom filters could be trained in advance, leading to highly efficient query execution. SieveJoin could also be applied to ad hoc queries, where Bloom filters are built on-the-fly. The improvement is marginal, and sometimes even worse than state-of-the-art. But the reduction in data being transferred is a crucial factor to be considered in distributed computing tasks.

In addition, as Bloom filters allow false positives and strictly prohibit false negatives, only equi-joins are supported. Queries with inequality conditions are not supported. It is possible that the query part holding inequality conditions could be materialized in advance; such an implementation is not studied in this paper. Finally, the propagation of Bloom filters is trained sequentially, and the parallel training of this procedure is not studied.

3 Implementation

SieveJoin[1] is implemented in C++. It adopts the column-oriented fashion where columns are stored in a list of vectors. B-tree indexes are generated in advance for relevant columns.

3.1 Reuse of Bloom Filters

SieveJoin is designed for popular queries with predictable workloads and might also be applied to solve ad hoc queries, where applicable. Specifically, assuming we have generated Bloom filter BFs for a popular query template Q_1. If Q_1 is a sub-query of an ad hoc query Q_2, these pre-generated Bloom filters could be reused for Q_2.

The application to acyclic queries is achieved straightforwardly. For cyclic queries, we could only reuse part of the pre-generated BFs, while the Bloom filter belonging to the final closing condition has to be re-generated on the fly. Figure 8 demonstrates the reuse strategy for cyclic joins in more detail. We reuse the majority of the pre-generated BFs (BF_1 to BF_{n-1}) and pay extra time to generate Bloom filters for R_n and R_{n+1} on the fly.

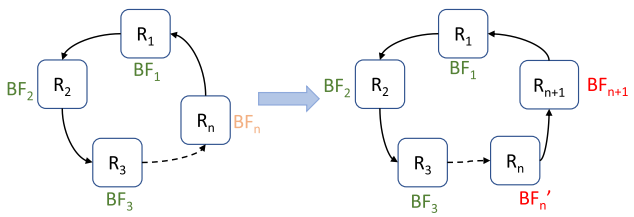


Fig. 8 Reuse of bloom filters for cyclic joins

3.2 Support for Updates

Since Bloom filters are the only additional data structures used, update performance largely depends on them. Fortunately, Bloom filters support constant-time insertions with $O(1)$ complexity. For a new tuple t_1 inserted into relation R_i, which maintains Bloom filter BF_1, we first check if $BF_1.\text{contains}(t_1)$. If true, no update is needed, costing only $O(1)$. Otherwise, we search from R_i to its adjacent relations. Assuming pre-built B-tree indexes, the total complexity is $O(n \log N)$, where n is the number of joined relations and N is the largest relation size.

Currently, SieveJoin based on traditional Bloom filters only supports frequent insertions. If counting Bloom filters, which hold a counter for each existing tuple, are applied to SieveJoin, deletions and updates will be supported straightforwardly, with the cost of slightly larger space overheads.

4 Performance Evaluation

4.1 Experimental Setup

In this section, we present a comprehensive evaluation of the SieveJoin algorithm against state-of-the-art database systems and methods. Specifically, we compare SieveJoin with the following baselines:

- **Umbra** [11]: The state-of-the-art multi-way join database, which uses a heuristic optimizer to trade-off between WCOJ and binary join plans.
- **MonetDB** [1, 47]: The well-known column-store database that exclusively relies on binary join plans.
- **Postgres** [2]: A free and open-source relational database management system (RDBMS) based on binary join plans.

All experiments are run on a Ubuntu 20.04 server with 16 CPU cores (32 threads) on an AMD 7950X processor, 128GiB RAM, and 4TB of SSD disk space. Each experiment is repeated three times, and the average values are reported. Unless specified, all systems run in parallel.

[1] code is available at https://github.com/qingzma/SieveJoin

4.2 Synthetic Workload

To demonstrate SieveJoin's performance for cases with a controlled number of intermediate join results, we conduct experiments on a synthetic benchmark, as described in [11]. Two parameters, $r \in \{10^4, 10^5, 10^6, 10^7\}$, and $d \in \{1, \ldots, 10\}$, are used to control the number of intermediate join results. Three relations (R, S, and T) are generated, with R containing integers from 1 to 10^7, S containing integers from 1 to $(10^7 + r)/2$, and T containing integers from $(10^7 - r)/2$ to 10^7. Each integer in R, S, and T is duplicated d times. For a pair of parameters r and d, the number of rows in the tables for R, S, and T are $10^7 \times d$, $\frac{(10^7+r)}{2} \times d$, and $\frac{(10^7+r)}{2} \times d$, respectively. The resulting natural join $R \bowtie S \bowtie T$ contains exactly r distinct integers, each duplicated d^3 times. Thus, the final join results contain rd^3 tuples.

Figure 9 compares the response time of the query $R \bowtie S \bowtie T$ among different systems and methods. Clearly, Postgres performs the worst among them, relying purely on binary join plans. There are even some missing values for Postgres and MonetDB as the system runs out of memory. Also, we notice that as the number of duplicates increases, the runtime of binary join plans, specifically Postgres and MonetDB, increases much more rapidly than SieveJoin and Umbra. Umbra is based on a multi-way join implementation that meets the worst-case optimal criteria and is more robust with increased duplicates. Interestingly, the performance of Umbra is slightly worse than MonetDB. On the contrary, the query response time grows much slower for SieveJoin, as it has the ability to 'stop early' and avoid the generation of exploding intermediate join results. SieveJoin is always at least 1–2 orders of magnitude faster than other systems. This demonstrates the strong abilities of SieveJoin at filtering intermediate join results.

We quantify SieveJoin's ability in pruning relations when $r = 10^4$ and $d \in \{1, \ldots, 10\}$. Relative size change (RSC), as defined below, is used to reflect the change in relation size before and after pruning.

$$\text{Relative Size Change} = \frac{\text{Size of the Pruned Relation}}{\text{Size of the Original Relation}} \tag{5}$$

Figure 10a summarizes RSC with an increased number of duplicates d. We do not include the statistics for Relation T as it is the rightmost relation, and there is no reduction for such a relation if the one-pass propagation strategy is used (refer to Sect. 2.2). As shown, the size of relation S is reduced to 0.2% of its original size. The size of the leftmost relation R is reduced even further. This is expected as the propagation phase will filter more and more tuples from the rightmost to the leftmost relation.

As shown in Fig. 10b, we represent the training time of SieveJoin, which includes the time to load data, the time to build traditional tree-based indexes, and the time to build Bloom filters. It is interesting to note that it takes less time to build Bloom filters than to build indexes. This is expected as the complexity of updating a Bloom filter is $O(1)$, while the complexity of updating a B-tree is $O(\log(N))$.

Although the training time for building indexes and Bloom filters is much longer than the query response time,

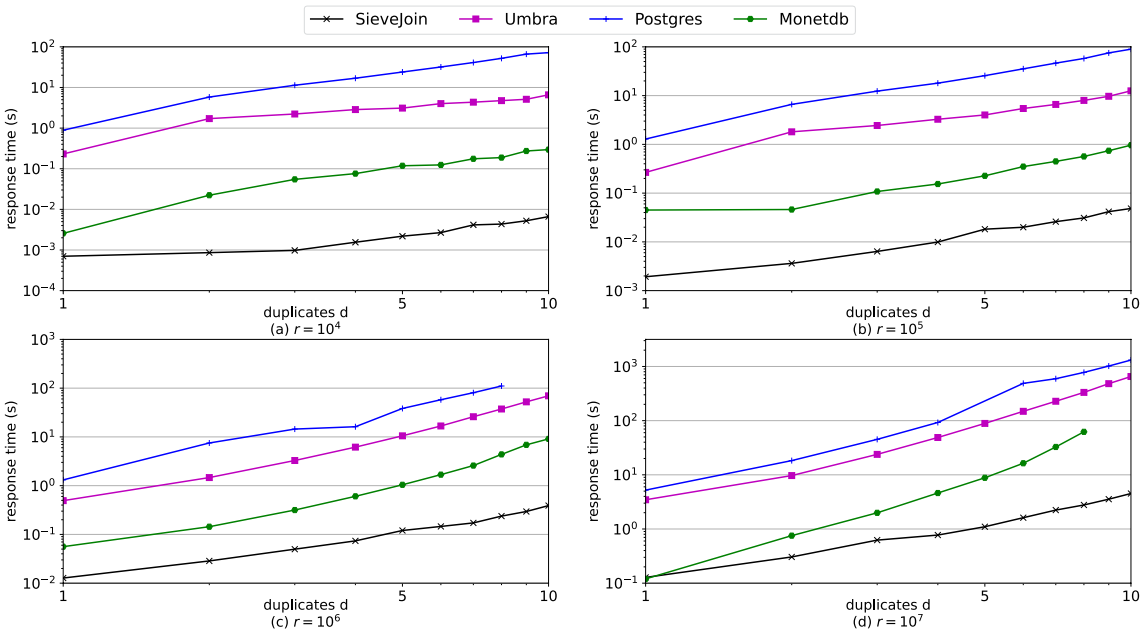
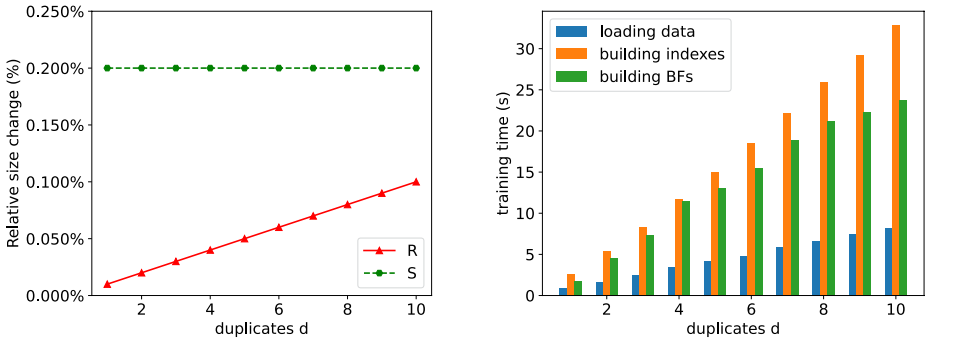


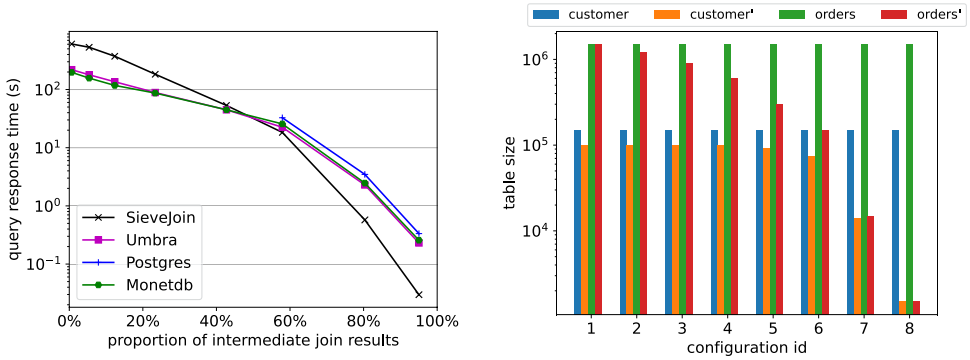
Fig. 9 Comparison of query response time for the synthetic data

Fig. 10 Performance for synthetic workload



(a) Reduction of Relation Size (b) Training Time for the Synthetic Data

Fig. 11 Performance for TPC-H



(a) Query Response Time for TPC-H (b) Table Size Reduction for TPC-H

we argue that SieveJoin targets popular queries with predictable workloads, and Bloom filters could be prepared in advance.

4.3 TPC-H Workload

We use a scale factor SF = 1, resulting in the largest table having approximately 6 million rows. (Note that we also ran experiments for SF=10 and 40, the same conclusion holds, and these results are omitted for space reasons.). We run acyclic joins of query X involving 5 tables, as described in [48]. Specifically, this query aims to find suppliers and customers in the same nations with the purchase history of the customers, which is:

```
SELECT n_nationkey, s_suppkey, c_custkey, o_orderkey,
   l_linenumber
FROM nation, supplier, customer, orders, lineitem
WHERE n_nationkey = s_nationkey
AND s_nationkey = c_nationkey
AND c_custkey = o_custkey
AND o_orderkey = l_orderkey;
```

For this experiment, we will measure the proportion of intermediate join results over the total join paths generated during query processing. To control the proportion of intermediate join results, we make 8 configurations of table `lineitem`, with samples of size $\in \{100\%, 80\%, 60\%, 40\%, 20\%, 10\%, \ 1\%, 0.1\%\}$, which generates $\approx \{0.6\%, 5\%, 10\%, 20\%, 40\%, 60\%, 80\%, 95\%\}$ of intermediate join results.

Figure 11a summarizes the query response time from different systems. As the proportion of intermediate join results increases, query response time drops. This is expected as fewer tuples are processed and generated during query processing, showcasing various database systems' abilities to avoid the generation of intermediate join results. The performance of Umbra and MonetDB is similar. Postgres performs the worst, and the system runs out of memory (128GB) in many cases.

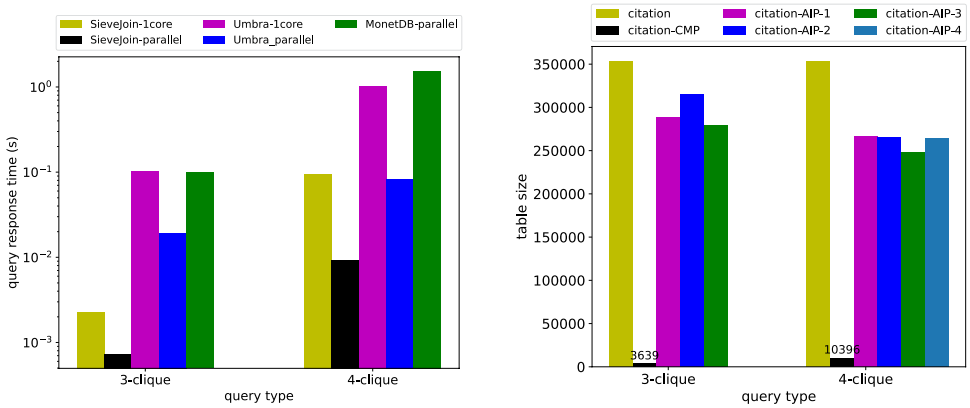


Fig. 12 Performance for citation workload

(a) Query Response Time for Citation

(b) Table Size Reduction for Citation

Figure 11b shows the reduction in table size with the participation of Bloom filters. The leftmost tables `nation` and `supplier` are small (with tens or hundreds of tuples), and pre-trained Bloom filters indicate that all tuples in these two tables are involved in the final join result.

Similar to Fig. 10a, we notice a significant reduction in table size for cases with large intermediate join results (see configuration id=8, where the proportion of intermediate join result is ≈ 95%). Specifically, `customer` enjoys two orders of magnitude reduction in table size, and `orders` enjoys three orders of magnitude reduction. For other configurations, as there are fewer intermediate join results, the reduction in table size becomes less significant.

4.4 Citation Workload

This section presents the experimental evaluation of cyclic joins. The arXiv HEP-PH citation graph [49, 50] contains 27,770 papers and 352,807 citation edges, where each row represents a citation from id1 to id2. A clique join on this dataset identifies patterns where papers cited by paper *p* also cite *p*. The 4-clique query is:

```
SELECT t1.k1, t2.k1, t3.k1, t4.k1
FROM t1, t2, t3, t4
WHERE t1.k2 = t2.k1
AND t2.k2 = t3.k1
AND t3.k2 = t4.k1
AND t4.k2= t1.k1;
```

, where table {1–4} are the same table in the dataset. We run experiments for 3-clique and 4-clique queries.

Figure 12a compares the query response time. Clearly, the parallel version of SieveJoin outperforms other systems with 1–3 orders of magnitude reduction. For the 3-clique query, even a single-threaded SieveJoin beats other systems by at least 10 times.

We further demonstrate SieveJoin's effectiveness in eliminating intermediate results from both AIPs and CMPs. In cyclic joins, both types exist; in acyclic joins, only AIPs appear. In Fig. 12b, the yellow bars show the original size of the `citation` table, while black bars indicate the size after pruning AIPs and CMPs. *citation-AIP-{1–4}* denotes the pruned sizes of 3/4 views for 3-clique and 4-clique queries. Removing only AIPs reduces table size by 20% for 3-clique and 25% for 4-clique. To further eliminate CMPs, we apply equality propagation (Sect. 2.3) in SieveJoin, leading to substantial reductions in final table size–down to 3639 and 10,396 rows–achieving one to two orders of magnitude reduction.

The `citation` dataset is sparse but generates an overwhelming number of intermediate results. Designed for such cases, SieveJoin achieves orders-of-magnitude reductions in query time, even with simple binary join plans.

5 Related Work

With respect to multi-way joins, the existing solution space mainly falls into two categories: binary joins, and worst-case optimal joins.

Binary joins target the join procedure between two relations, and are extensively studied and optimized over the past several decades [4–6]. Techniques of various formats are used to improve the execution efficiency of binary joins, including indexes [51, 52], selections on row-store [2] and column-store [1, 53, 54], design choice for on-disk and in-memory databases [3, 55]. Binary joins provide great flexibility, high reliability, and robustness for a variety of query workloads, and are applied in to mainstream RDBMSs and big data infrastructures for multi-way joins processing. Recent advances in database theory show that binary join plans are theoretically suboptimal [7, 8], a conclusion further supported by empirical studies [9–12]. The main bottleneck lies in binary joins' inability to eliminate exploding intermediate results. Join paths are often generated only to be discarded later due to missing matches, leading to unnecessary I/O and slower query processing.

Worst-case optimal joins (WCOJ) have gained attention following recent advances in database theory by Ngo et al. [7, 8, 56]. This algorithm was proposed with a tighter asymptotic runtime complexity compared with binary joins. The Leapfrog Triejoin algorithm [10], extends the idea of sort-merge join to multi-way joins, achieves great success and is adopted in the LogicBlox system [57]. Extensions or variations of the Leapfrog algorithms have been adopted in distributed query processing [58, 59] and graph processing [60, 61]. Umbra [11] uses a Hash Trie for fast tuple access and introduces a heuristic optimizer to choose between binary and WCOJ plans, achieving state-of-the-art performance. Other multi-way join efforts include [12, 62, 63], etc.

Works on applying Bloom filters for join query processing are closest to SieveJoin. Bloomjoin [25, 26] was the early study that adopted Bloom filters to reduce data transferred between sites. [26] shows that Bloomjoin consistently outperforms the basic semi-join algorithm. LIP [64] relies on Bloom filters to generate a robust and efficient query plan for multi-way joins, the work focuses on the order of applying Bloom filters and is limited to star schema. Bit-vector filtering [65] further analyzes a much broader range of decision support queries and plan search space, reducing CPU execution time significantly. Bloom filter operations [28, 29] are used for improving multi-way join performance in the distributed environment, and they introduced new bloom filter operations to support this. However, such operations could only be applied to the same join attribute.

6 Conclusion

In this paper, we introduce SieveJoin, a Bloom filter-based algorithm designed to boost multiple join operations efficiently. The distinctive feature of SieveJoin lies in its ability to 'stop early' by leveraging Bloom filters, thereby eliminating unnecessary intermediate join results. The core insight is the proactive pruning of tables involved in multiple joins, optimizing for popular workloads and maintaining information integrity for accurate results. SieveJoin delivers significant query speedups with minimal memory overhead, leveraging the compactness of Bloom filters. The introduction of acyclic incomplete paths (AIP) and cyclic mismatch paths (CMP) intermediate join results, along with the propagation of Bloom filters, opens up new avenues for join processing optimization. The paper provides a comprehensive evaluation of SieveJoin against state-of-the-art databases. The evaluation considers various aspects, including query processing speedup, data size reduction, model training time, and additional space overheads. Future work will focus on exploring equality propagation and enhancing the integration of SieveJoin with other database systems, aiming to further advance the efficiency and applicability of the proposed algorithm.

Acknowledgements This work is supported by Natural Science Foundation of China (Grant No. 62272332), and by Project Funded by Priority Academic Program Development of Jiangsu Higher Education Institutions.

Author contributions The individual contributions to this work are specified as follows: Ma proposed the core research idea and designed the experimental framework. Li and Ma conducted data preprocessing and performed experiments on Synthetic dataset and TPC-H workload. Shi and Liu performed experiments on citation dataset. The first draft of the manuscript was written by Li and Ma. Liu and Shi provided detailed optimizations for SieveJoin algorithm and offered valuable feedback to improve this revised paper. All authors participated in manuscript revision and final approval, and agree to be accountable for all aspects of the work.

Data availability The implementation code and data is available at: https://github.com/qingzma/SieveJoin

Declarations

Competing interests Peter Triantafillou, University of Warwick 1 cmQing Li, Hong Kong Polytechnic University.

References

1. Nes SIFGN, Kersten SMSMM (2012) Monetdb: two decades of research in column-oriented database architectures. Data Eng 35:40
2. Stonebraker M, Rowe LA (1986) The design of postgres. ACM SIGMOD Rec 15(2):340–355
3. Lahiri T, Neimat M-A, Folkman S (2013) Oracle timesten: an in-memory database for enterprise applications. IEEE Data Eng Bull 36(2):6–13
4. Idris M, Ugarte M, Vansummeren S (2017) The dynamic yannakakis algorithm: compact and efficient query processing under updates. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp 1259–1274
5. Yannakakis M (1981) Algorithms for acyclic database schemes. In: VLDB, vol. 81, pp 82–94
6. Goodman N, Shmueli O, Tay YC (1983) Gyo reductions, canonical connections, tree and cyclic schemas and tree projections. In: Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, pp 267–278
7. Ngo HQ, Porat E, Ré C, Rudra A (2018) Worst-case optimal join algorithms. J of the ACM (JACM) 65(3):1–40
8. Ngo HQ, Ré C, Rudra A (2014) Skew strikes back: new developments in the theory of join algorithms. ACM SIGMOD Rec 42(4):5–16
9. Graefe G, Bunker R, Cooper S (1998) Hash joins and hash teams in microsoft sql server. In: VLDB, vol. 98, pp 86–97
10. Veldhuizen TL (2014) Leapfrog triejoin: a simple, worst-case optimal join algorithm. In: Proc. International Conference on Database Theory
11. Freitag M, Bandle M, Schmidt T, Kemper A, Neumann T (2020) Adopting worst-case optimal joins in relational database systems. Proc VLDB Endow 13(12):1891–1904
12. Shanghooshabad AM, Triantafillou P (2022) Graphical join: a new physical join algorithm for rdbmss. arXiv preprint arXiv:2206.10435
13. Avnur R, Hellerstein JM (2000) Eddies: continuously adaptive query processing. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, pp 261–272
14. Aberger CR, Lamb A, Tu S, Nötzli A, Olukotun K, Ré C (2017) Emptyheaded: a relational engine for graph processing. ACM Trans Datab Syst (TODS) 42(4):1–44
15. Aberger C, Lamb A, Olukotun K, Ré C (2018) Levelheaded: a unified engine for business intelligence and linear algebra querying. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp 449–460. IEEE
16. Kemper A, Kossmann D, Wiesner C (1999) Generalised hash teams for join and group-by. In: VLDB, vol 99, pp 30–41. Citeseer
17. Zukowski M, Wiel M, Boncz P (2012) Vectorwise: a vectorized analytical dbms. In: 2012 IEEE 28th International Conference on Data Engineering, pp 1349–1350. IEEE
18. Chu S, Balazinska M, Suciu D (2015) From theory to practice: efficient join query evaluation in a parallel database system. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp 63–78
19. Wu H, Zinn D, Aref M, Yalamanchili S (2014) Multipredicate join algorithms for accelerating relational graph processing on gpus. In: International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures, vol. 10
20. Zou L, Mo J, Chen L, Özsu MT, Zhao D (2011) Gstore: answering sparql queries via subgraph matching. Proc VLDB Endow 4(8):482–493
21. Bloom BH (1970) Space/time trade-offs in hash coding with allowable errors. Commun ACM 13(7):422–426
22. Kirsch A, Mitzenmacher M (2006) Less hashing, same performance: Building a better bloom filter. In: Algorithms–ESA 2006: 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006. Proceedings 14, pp 456–467. Springer
23. Luo L, Guo D, Ma RT, Rottenstreich O, Luo X (2018) Optimizing bloom filter: challenges, solutions, and comparisons. IEEE Commun Surv Tutor 21(2):1912–1949
24. Li Y, Wang F, Yu X, Yang Y, Yang K, Yang T, Ma Z, Cui B, Uhlig S (2023) Ladderfilter: filtering infrequent items with small memory and time overhead. Proc ACM on Manag Data 1(1):1–21
25. Bratbergsengen K (1984) Hashing methods and relational algebra operations. In: Proceedings of the 10th International Conference on Very Large Data Bases, pp 323–333
26. Mackert LF, Lohman GM (1986) R* optimizer validation and performance evaluation for local queries. In: Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, pp 84–95
27. Cohen S, Matias Y (2003) Spectral bloom filters. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp 241–252
28. Michael L, Nejdl W, Papapetrou O, Siberski W (2007) Improving distributed join efficiency with extended bloom filter operations. In: 21st International Conference on Advanced Information Networking and Applications (AINA'07), pp 187–194. IEEE
29. Koutris P (2011) Bloom filters in distributed query execution. University of Washington, CSE 544
30. Lee T, Kim K, Kim H-J (2012) Join processing using bloom filter in mapreduce. In: Proceedings of the 2012 ACM Research in Applied Computation Symposium, pp 100–105
31. Ramesh S, Papapetrou O, Siberski W (2009) Optimizing distributed joins with bloom filters. In: Distributed Computing and Internet Technology: 5th International Conference, ICDCIT 2008 New Delhi, India, December 10-12, 2008. Proceedings 5, pp 145–156. Springer
32. Caetano TS, McAuley JJ, Cheng L, Le QV, Smola AJ (2009) Learning graph matching. IEEE Trans Pattern Anal Mach Intell 31(6):1048–1058
33. Bunke H (2000) Recent developments in graph matching. In: Proceedings 15th International Conference on Pattern Recognition. ICPR-2000, vol 2, pp 117–124. IEEE
34. Livi L, Rizzi A (2013) The graph matching problem. Pattern Anal Appl 16:253–283
35. Bomze IM, Budinich M, Pardalos PM, Pelillo M (1999) The maximum clique problem handbook of combinatorial optimization. Kluwer Academic Publishers, Boston
36. Gross JL, Yellen J (2003) Handbook of Graph Theory (Discrete Mathematics and Its Applications). CRC Press
37. Chandra AK, Merlin PM (1977) Optimal implementation of conjunctive queries in relational data bases. In: Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, pp 77–90
38. Chekuri C, Rajaraman A (2000) Conjunctive query containment revisited. Theoret Comput Sci 239(2):211–229
39. Chaudhuri S, Vardi MY (1993) Optimization of real conjunctive queries. In: Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp 59–70

40. Nguyen D, Aref M, Bravenboer M, Kollias G, Ngo HQ, Ré C, Rudra A (2015) Join processing for graph patterns: An old dog with new tricks. In: Proceedings of the GRADES'15, pp 1–8
41. Severance DG, Lohman GM (1976) Differential files: their application to the maintenance of large databases. ACM Trans Database Syst (TODS) 1(3):256–267
42. Patel JM, DeWitt DJ (1996) Partition based spatial-merge join. ACM SIGMOD Rec 25(2):259–270
43. DeWitt DJ, Naughton JF, Burger J (1993) Nested loops revisited. In: [1993] Proceedings of the Second International Conference on Parallel and Distributed Information Systems, pp 230–242. IEEE
44. Lo M-L, Ravishankar CV (1996) Spatial hash-joins. In: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, pp 247–258
45. Blanas S, Li Y, Patel JM (2011) Design and evaluation of main memory hash join algorithms for multi-core cpus. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pp 37–48
46. Barber R, Lohman G, Pandis I, Raman V, Sidle R, Attaluri G, Chainani N, Lightstone S, Sharpe D (2014) Memory-efficient hash joins. Proc VLDB Endow 8(4):353–364
47. Boncz PA, Zukowski M, Nes N (2005) Monetdb/x100: hyper-pipelining query execution. In: Cidr, vol 5, pp 225–237
48. Zhao Z, Christensen R, Li F, Hu X, Yi K (2018) Random sampling over joins revisited. In: Proceedings of the 2018 International Conference on Management of Data, pp 1525–1539
49. Leskovec J, Kleinberg J, Faloutsos C (2005) Graphs over time: densification laws, shrinking diameters and possible explanations. In: Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, pp 177–187
50. Gehrke J, Ginsparg P, Kleinberg J (2003) Overview of the 2003 kdd cup. ACM SIGKDD Explor Newsl 5(2):149–151
51. Graefe G et al (2011) Modern b-tree techniques. Found Trends® Datab 3(4):203–402
52. Chan C-Y, Ioannidis YE (1998) Bitmap index design and evaluation. In: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, pp 355–366
53. Chandramouli B, Goldstein J, Barnett M, DeLine R, Fisher D, Platt JC, Terwilliger JF, Wernsing J (2014) Trill: a high-performance incremental query processor for diverse analytics. Proc VLDB Endow 8(4):401–412
54. Bykov S, Geller A, Kliot G, Larus JR, Pandya R, Thelin J (2011) Orleans: cloud computing for everyone. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, pp 1–14
55. Fang J, Mulder YT, Hidders J, Lee J, Hofstee HP (2020) In-memory database acceleration on fpgas: a survey. VLDB J 29:33–59
56. Ngo HQ, Nguyen DT, Re C, Rudra A (2014) Beyond worst-case analysis for joins with minesweeper. In: Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp 234–245
57. Aref M, Cate B, Green TJ, Kimelfeld B, Olteanu D, Pasalic E, Veldhuizen TL, Washburn G (2015) Design and implementation of the logicblox system. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp 1371–1382
58. Afrati FN, Ullman JD (2011) Optimizing multiway joins in a map-reduce environment. IEEE Trans Knowl Data Eng 23(9):1282–1298
59. Ammar K, McSherry F, Salihoglu S, Joglekar M (2018) Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. Proc VLDB Endowm 11(6):1
60. Hogan A, Riveros C, Rojas C, Soto A (2019) A worst-case optimal join algorithm for sparql. In: The Semantic Web–ISWC 2019: 18th International Semantic Web Conference, Auckland, New Zealand, October 26–30, 2019, Proceedings, Part I 18. Springer, pp 258–275
61. Mhedhbi A, Salihoglu S (2019) Optimizing subgraph queries by combining binary and worst-case optimal joins. Proc VLDB Endow. https://doi.org/10.14778/3342263.3342643
62. Le-Phuoc D (2018) Adaptive optimisation for continuous multi-way joins over rdf streams. In: Companion Proceedings of the The Web Conference 2018, pp 1857–1865
63. Viglas SD, Naughton JF, Burger J (2003) Maximizing the output rate of multi-way join queries over streaming information sources. In: Proceedings 2003 VLDB Conference. Elsevier, pp 285–296
64. Zhu J, Potti N, Saurabh S, Patel JM (2017) Looking ahead makes query plans robust: making the initial case with in-memory star schema data warehouse workloads. Proc VLDB Endow 10(8):889–900
65. Ding B, Chaudhuri S, Narasayya V (2020) Bitvector-aware query optimization for decision support queries. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp 2011–2026