I Can’t Believe It’s Not Yannakakis: Pragmatic Bitmap Filters in Microsoft SQL Server

Hangdong Zhao
Gray Systems Lab, Microsoft
USA

Yuanyuan Tian
Gray Systems Lab, Microsoft
USA

Rana Alotaibi
KACST
Saudi Arabia

Bailu Ding
Microsoft Research
USA

Nicolas Bruno
Gray Systems Lab, Microsoft
USA

Jesús Camacho-Rodríguez
SQL DW, Microsoft
USA

Vassilis Papadimos
SQL Server, Microsoft
USA

Ernesto Cervantes Juárez
SQL DW, Microsoft
USA

Cesar Galindo-Legaria
SQL DW, Microsoft
USA

Carlo Curino
Gray Systems Lab, Microsoft
USA

ABSTRACT

The quest for optimal join processing has reignited interest in the Yannakakis algorithm, as researchers seek to realize its theoretical ideal in practice via bitmap filters instead of expensive semijoins. While this academic pursuit may seem distant from industrial practice, our investigation into production databases led to a startling discovery: over the last decade, Microsoft SQL Server has built an infrastructure for bitmap pre-filtering that subsumes the very spirit of Yannakakis! This is not a story of academia leading industry; but rather of industry practice, guided by pragmatic optimization, outpacing academic endeavors. This paper dissects this discovery. As a crucial contribution, we prove how SQL Server’s bitmap filters, pull-based execution, and Cascades optimizer conspire to not only consider, but often generate, instance-optimal plans, when it truly minimizes the estimated cost! Moreover, its rich plan search space reveals novel, largely overlooked pre-filtering opportunities on intermediate results, which approach strong semi-robust runtime for arbitrary join graphs. Instead of a verdict, this paper is an invitation: by exposing a system design that is long-hidden, we point our community towards a challenging yet promising research terrain.

1 INTRODUCTION

In 1981, Yannakakis proposed a seminal join algorithm [39] (see Sec. 2.1) that, in its essence, established for any acyclic join query, a set of *instance-optimal* query plans, for which no asymptotically faster alternatives can be hoped for. Despite this breakthrough, the algorithm has been largely sidelined by database engines as it bears an additional (costly) semijoin phase before any join execution.

A new renaissance has recently begun. When researchers [38] came to the realization that, similar to Bloom joins [18, 45], bitmap (or Bloom) filters can achieve near-equivalent, but much cheaper, pre-filtering as opposed to semijoins, the strong promise of Yannakakis re-gained appeal. A surge of academic pursuits (see Sec. 2.2) have since sought to retrofit its theoretical guarantees into modern databases via bitmap filters, showing they can not only accelerate joins, but also mitigate the negative impact of poor join orders [44].

Our mission was set to bridge the gap between recent academic insights and production engines. To this purpose, we dived into SQL Server, on which bitmap filters have been deployed since 2012. Our investigation led to a surprising discovery: **SQL Server, through a decade of continual refinement, has already captured Yannakakis as part of its design!** More shockingly, SQL Server unveils in a cost-based framework novel pre-filtering opportunities that were largely overlooked in the existing work.

Given the widespread adoption of Bloom filters and semijoins in industrial engines [1, 2, 22], the “Yannakakis effect" may potentially appear in other modern databases incorporating bitmap filters. This paper, however, presents a first formal analysis of this phenomenon, centered around its concrete implementation in SQL Server.

We show how SQL Server elegantly blends bitmap filters into a pull-based execution (Sec. 3). Its hash join creates bitmaps alongside hash tables and pushes them into the probe-side subplan to drop mismatches early. In its pull-based execution, chaining such hash joins naturally cascades bitmaps through multi-joins before recursive `next()` calls for hash probing. We prove such execution paradigm being instance-optimal, meaning that **SQL Server implicitly considers all Yannakakis-style plans for acyclic joins!**

Since full instance-optimality is provably impossible for arbitrary join queries, we propose a generalized notion of performance *semi-robustness* (Sec. 3.4), where runtime depends only on a small number of critical intermediates, rather than on every intermediate result. By considering bitmap filters to be built from and applied to intermediate join results, a capability largely absent in prior pre-filtering work, we show that **SQL Server’s execution model can achieve strong semi-robust guarantees even on cyclic joins**.

While recent works often treat bitmap filters as a heuristic add-on, **SQL Server considers them directly in the costing framework of its Cascades optimizer** [13, 20] (Sec. 4). During planning,

```
SELECT L_ORDERKEY,
       SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)),
       O_ORDERDATE,
       O_SHIPPRIORITY
FROM   CUSTOMER,
       ORDERS,
       LINEITEM
WHERE C_MKTSEGMENT = 'BUILDING' AND
      C_CUSTKEY = O_CUSTKEY AND
      L_ORDERKEY = O_ORDERKEY AND
      O_ORDERDATE < '1995-03-15' AND
      L_SHIPDATE > '1995-03-15'
GROUP BY L_ORDERKEY,
         O_ORDERDATE,
         O_SHIPPRIORITY
ORDER BY REVENUE DESC,
         O_ORDERDATE
```

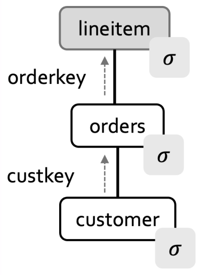

Figure 1: TPC-H Q3 in SQL (left) and its join graph (right). It is also a join tree rooted at lineitem (shaded). The dotted arrows indicate the bottom-up semijoin pass and the top-down pass is its reverse.

bitmap filter is a native cost dimension in the optimizer. After estimating the build-side selectivity, it propagates a bitmap "context" into the probe subtree and (re-)optimizes it in the new context (accounting for cascading bitmap selectivities). To avoid blowups of cost alternatives during query optimization, SQL Server uses a number of heuristics to limit re-exploration.

On TPC-H benchmark, we showed that SQL Server's cost-based search frequently selects instance-optimal plans for execution (Sec. 5). Its bitmap filter design achieves consistent speedups (up to 3.5×) and near-optimal pre-filtering—approaching the Yannakakis ideal. However, for the most demanding queries in production settings, in which complex join patterns and misestimated selectivities abound, bitmap placement/transferring can falter and erode performance. This is where the new frontier for pre-filtering optimization unfolds, opening compelling avenues for future research (Sec. 6).

2 BACKGROUND AND RELATED WORK

We revisit the Yannakakis algorithm [39] and recent advancements on pre-filtering techniques to make Yannakakis practical.

2.1 Yannakakis Algorithm

The Yannakakis algorithm introduces a semijoin phase [7, 39] prior to join execution (i.e., the join phase). It assumes an acyclic (formally, α-acyclic [39]) join graph, which can be seen as a *join tree* [39].

Example 2.1. Consider TPC-H Q3, which joins tables customer, orders, and lineitem. Each table has a local filter σ (e.g., `C_MKTSEGMENT = "BUILDING"`), and the join predicates are `c_custkey = o_custkey` and `o_orderkey = l_orderkey`. The join graph itself is a join tree (so it is acyclic, see Fig. 1). We use `C`, `O`, and `L` to denote the three tables after being reduced by their respective local filters.

The semijoin phase consists of a *bottom-up* and a *top-down passes*. The bottom-up pass traverses the join tree in post-order, filtering the table (at each node) by semijoining it with its child tables until reaching the root. The top-down pass reverses this process, starting from the root and filtering each child table by its parent. For Q3's join tree, the two semijoin passes are (also shown in Fig. 1):

// bottom-up: $O^{\perp} \leftarrow O \ltimes_{\mathtt{custkey}} C,\quad L^{\perp} \leftarrow L \ltimes_{\mathtt{orderkey}} O^{\perp}$

// top-down: $O^{\top} \leftarrow O^{\perp} \ltimes_{\mathtt{orderkey}} L^{\perp},\quad C^{\top} \leftarrow C \ltimes_{\mathtt{custkey}} O^{\top}$

Now, Yannakakis moves to the join phase and answers the query by hash joining the reduced tables along the join tree, in our case, $C^{\top} \bowtie O^{\top} \bowtie L^{\perp}$. The next two properties make the algorithm appealing: for acyclic queries, ① the semijoins eliminate all rows from base tables that do not occur in the final result in time $O(N)$ [7], N being the input table sizes[1], and in the join phase, ② hash joins on reduced tables take time $O(N + \text{OUT})$, where OUT is the query output size[2]. The algorithm is said to be *instance optimal* as N + OUT is the unavoidable cost of input scanning and emitting the output.

2.2 Making Pre-filtering Practical

Despite its strong guarantees, Yannakakis incurs significant semijoin overhead that has impeded its adoption in modern database engines. Recent work has proposed novel techniques for reducing this cost while retaining similar theoretical appeal.

Predicate Transfer. Recent work [38] proposed Predicate Transfer (PT), a technique that replaces the semijoins of Yannakakis by Bloom filters [18, 32, 45], which can efficiently propagate across join predicates. Beyond direct acceleration of execution, PT delivers robustness for acyclic queries, i.e., making optimizers less susceptible to poor join orders [44]. The Bloom filter transferring schedule is determined by heuristics, e.g., traversing from small to large tables.

Reducing the Number of Semijoins. Birler et al. [8] advocates for a novel decomposition of hash joins into two operators: Lookup and Expand. Each Lookup performs a semijoin while keeping iterators that allow a later expansion into full joins. This decoupling facilitates operator reordering, e.g., Lookup pushdown to early prune base tables. Wang et al. [35] identify conditions under which certain semijoins can be removed without compromising optimality, thereby minimizing the number of required semijoin operations.

Pre-computed Data Structures. Another approach to curb semijoin overhead is pre-computing compact data structures to pre-filter at runtime: Kandula et al. [23] caches zone maps to skip scanning non-joining rowgroups; Stoian et al. [33] pre-stores Bloom filters to streamline the second pass of Yannakakis into a single scan.

Consider Pre-filtering in Optimizers. Recently, researchers from industry [13, 40] started to explore how to make optimizers produce efficient pre-filtering plans while reducing overheads for searching in a much larger plan space. We defer those discussions to Section 4.

3 PRE-FILTERING IN QUERY EXECUTION

We show how SQL Server achieves the N+OUT instance optimality via an elegant execution design. SQL Server uses both Bloom filters and exact bit-vector filters (i.e., equivalent to semijoins), which we will collectively term bitmap filters throughout this paper.

[1]A semijoin is considered as a linear-time operator, i.e. $O(N)$.

[2]When the context is clear, we use OUT to denote the output relation or its cardinality. In the join phase, the intermediate joins on the reduced tables only monotonically increase in size (since every tuple joins) and will not grow beyond OUT.

3.1 Batch Mode Hash Join

Batch mode is SQL Server's vectorized execution model that organizes columnar data on disk into *rowgroups* (each holding ~900 rows, while at execution time, batch mode operators extract them into in-memory data *batches*, as opposed to operating on one row at a time, or row mode [25][3]. It brings great performance gains by exploiting cache locality and SIMD instructions, among others [9].

The (batch mode) hash join (or simply `HJ`) is the sole execution primitive that unlocks the potential of SQL Server's bitmap filters[4]. In its iterator execution model, `HJ.next()`[5] implements two sequential steps—① opens the build side, creates a hash table and (optionally) bitmap filters, closes it, and then ② opens the probe, pushes down bitmaps into the probe subtree, and calls `next()` to pull batches from the probe subtree to probe against the hash table and emit matching rows. Consider TPC-H Q12 that has a single join between lineitem and orders. We denote `L`, `O` as their reduced tables after filters. We will ignore local filters throughout the paper as they are assumed to be applied during table scans. Fig. 2 shows a pseudocode for this `HJ`, `L` being the build side (since `L` < `O` in size).

```
1:  for t1 in L.next():
2:    construct_hash_table(t1)
3:    // if HJ is selective (as pre-determined by the optimizer)
4:    construct_bitmap(t1.l_orderkey) // config adaptively chosen
5:  for t2 in O.next():  // scan operator of O probes bitmap, and
6:     // possibly skips non-matching rowgroups
7:    if hash_table.probe(t2):
8:      emit(t1 ⋈ t2)
```

Figure 2: SQL Server's `HJ` when calling `next()` for TPC-H Q12

Bitmap Construction (Lines 3-4). The optimizer estimates join selectivity (e.g. via histograms) and makes decisions on whether using bitmaps is cost-effective (Line 3). However, the actual bitmap implementation—whether to use bloom filters or exact bit-vectors, and how much memory to allocate for those—is postponed to runtime (Line 4). This decision exploits statistics collected during hash table construction (Line 2), including the cardinality and data distribution of distinct `l_orderkey` values. SQL Server thus adaptively selects the optimal bitmap configuration, trading off bitmap sizes against false positive rates. For example, if the ranges of `l_orderkey` values are compact, SQL Server may opt for a bit-vector for zero false positives. Beyond the bitmap itself, `construct_bitmap` maintains min/max values of that column for rowgroup skipping (see below).

Bitmap Probing (Line 5). Once created, bitmaps are heuristically pushed as deep as possible into the probe-side subplan to drop non-matching rows at the earliest opportunity. When pushed down to scan operators (as the case for `o_orderkey`, in Q12), the min/max values accompanying the bitmap achieve *rowgroup elimination*: in SQL Server, each rowgroup keeps min/max statistics of its underlying data chunk. If the min/max ranges coming from the build side does not overlap, the scan (including decoding) of that rowgroup is completely bypassed. This optimization may yield substantial I/O savings, depending on the physical data layout [23]. If the scanned tuple passes the bitmap filter, it probes into the hash table (Line 7) to pipeline outputs upstream (Line 8). In TPC-H Q12 at scale factor 100, the bitmap filter (in this case, a Bloom filter is chosen) rejects 80% out of the 15M orders rows, handing only 2.96M rows to the hash join—approaching a perfect semijoin pre-filtering (which would otherwise remain 2.92M orders rows under zero false positives).

[3]https://learn.microsoft.com/en-us/sql/relational-databases/query-processing-architecture-guide?view=sql-server-ver17
[4]More precisely, this is the batch mode parallel hash join operator in SQL Server. We omit parallelism for clarity, though all discussions apply and parallelize naturally.
[5]In SQL Server's vectorized execution, `next()` is really `next_batch()` that returns a batch of tuples, but we will use `next()` for simplicity.

```
The root HJ between C and (O ⋈ L) calls next()
1:  for t1 in C.next():
2:    construct_hash_table(t1)
3:    construct_bitmap(t1.c_custkey) // bitmap pushdown into O ⋈ L
4:  for t2 in (O ⋈ L).next(): // calls next() on the inner HJ
5:    if hash_table.probe(t2):
6:      emit(t1 ⋈ t2)

The inner HJ between O and L calls next()
7:  for t3 in O.next():  // scan of O probes bitmap(c_custkey)
8:    construct_hash_table(t3) // hash table creation on O⊥
9:    construct_bitmap(t3.o_orderkey) // bitmap creation on O⊥
10: for t4 in L.next():  // scan of L probes bitmap(l_orderkey)
11:   if hash_table.probe(t4): // t4 comes from L⊥
12:     emit(t3 ⋈ t4)
```

Figure 3: SQL Server's pull-based `HJ` execution for TPC-H Q3

3.2 Composing the Hash Joins

While each `HJ` (Fig. 2) implements nothing but extra bitmaps, the magic lies in how multiple `HJ`s compose in SQL Server's pull-based execution—the cascading pushdown of bitmaps implicitly realizes the bottom-up pass of Yannakakis (or PT [38]), attaining instance optimality with no additional machinery! To demonstrate, we revisit TPC-H Q3 (Example 2.1) and the SQL Server's chosen execution plan (see Fig. 4). The plan involves two `HJ` operators and Fig. 3 traces the execution starting from the root `HJ` between `C` and $(\mathtt{O} \bowtie \mathtt{L})$ calling `next()`. The execution unfolds as follows—① the root `HJ` builds a hash table and then a bitmap filter for `C` on `c_custkey` (Line 1-3) and the bitmap is pushed down through the join into the scan of `O`; ② then the probe side triggers `next()` on the inner `HJ` of $(\mathtt{O} \bowtie \mathtt{L})$ (Line 4). This inner `HJ` invokes the scan of `O`, but crucially, each tuple of `O` is pre-filtered by the `c_custkey` bitmap (Line 7). Only qualifying rows are inserted into the hash table and a second bitmap filter for `O` on `o_orderkey` (Line 8-9); ③ as the final pipeline, the table `L` is scanned and pre-filtered by the `o_orderkey` bitmap. Surviving tuples are then piped back through both hash table probes (`O` and `C`) to emit the final result (Line 12, then Line 6).

For the rest of this section, assume that the bitmaps have no false positives, that is, they are exact semijoins. The reader may realize that the first two stages of execution are just the bottom-up pass of Yannakakis outlined in Sec. 2.1, but also constructing hash tables on `C` and $\mathtt{O}^{\perp} = \mathtt{O} \ltimes \mathtt{C}$ along the way. In fact, since the pull-based execution model invokes `next()` recursively from the root `HJ`, each `HJ` opens the build side followed by the probe; it naturally traverses the rooted join tree bottom-up, i.e., `C->O->L` for Q3 (see Fig. 1).

A major difference from the Yannakakis algorithm and PT, however, is in the last stage, where SQL Server directly starts a backward probing pipeline of $\mathtt{L}^{\perp} = \mathtt{L} \ltimes \mathtt{O}^{\perp}$ into the already built hash tables of $\mathtt{O}^{\perp}$ and then `C` instead of another top-down pre-filtering pass. As such, SQL Server executes the query by a *single scan* over input (a

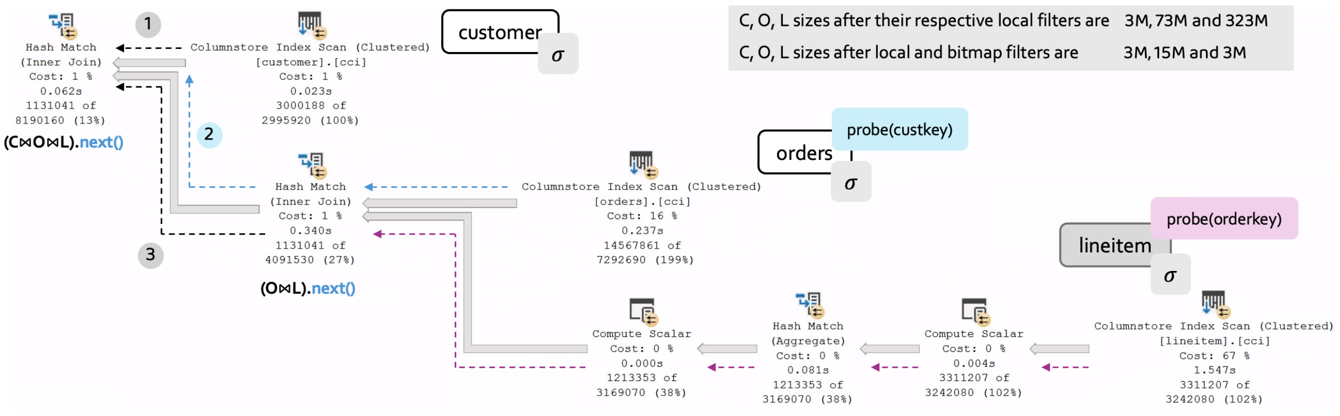

Figure 4: SQL Server's (partial) query plan for TPC-H Q3. Dotted arrows are the order of pull-based execution (cascading bitmap pushdowns and join probing). Each operator is annotated below with actual execution rows, estimated rows, and their ratio as percentage (actual/estimated).

common rule of thumb to avoid regressions caused by materialization, or possibly disk spills), in contrast to the three back and forth scans—despite possibly on smaller input—of Yannakakis and PT.

3.3 Instance-optimal Query Plan Execution

But by eliminating the top-down semijoin pass of Yannakakis, does SQL Server compromise instance optimality? Surprisingly, *no*! This follows as a corollary of Bagan et al.'s results [6] on output enumeration. We show a dedicated proof on TPC-H Q3 for this insight.

Proof of Instance Optimality for Q3's Execution. The bitmap filters (assuming no false positives) and hash tables cost $O(N)$ to build, akin to Yannakakis' bottom-up pass. We will bound the number of backward probes by $O(\mathrm{OUT})$. The insight is that when $\mathtt{L}^{\perp}$ probes $\mathtt{O}^{\perp}$, every probe is a hit (i.e., emits some join results). Because $\mathtt{L}^{\perp} = \mathtt{L} \ltimes \mathtt{O}^{\perp}$, every row in $\mathtt{L}^{\perp}$ has some matches in $\mathtt{O}^{\perp}$. As for the subsequent $(\mathtt{L}^{\perp} \bowtie \mathtt{O}^{\perp})$ probing into $\mathtt{C}$, since every row in $\mathtt{O}^{\perp}$ has some matches in $\mathtt{C}$ and $(\mathtt{L}^{\perp} \bowtie \mathtt{O}^{\perp})$ can only contain fewer distinct rows in $\mathtt{O}^{\perp}$, every probe into $\mathtt{C}$ must also be a hit. Hence, all intermediate probes (and outputs) are bounded by $O(\mathrm{OUT})$. □

Using batch mode hash join (HJ) as a fundamental physical building block, **SQL Server's plan space subsumes any Yannakakis-style instance-optimal plan for acyclic join queries**. As such, we can even rewrite Yannakakis algorithm using *only* SQL Server's HJ operator (Fig. 2, denoted as $\bowtie$) in Algorithm 1. The algorithm requires as input a post-order traversal of a join tree (i.e. leaves to root). At each step, 1 a (leaf) table becomes the build side. Using HJ, it constructs the hash table and bitmaps, pushing the latter into the probe side; 2 the remaining hash tables and bitmaps are built recursively. Once the recursion hits the root, it initiates the probing pipeline, where every backward probe *must* emit join results, ensuring that all intermediate work is bounded by the final output size. Consider a star schema with a fact table F and dimension tables $\mathtt{D}_1, \ldots, \mathtt{D}_n$. Following a post-order of $(\mathtt{D}_1, \ldots, \mathtt{D}_n, \mathtt{F})$ (F as the root), Algorithm 1 iteratively sets each $\mathtt{D}_i$ as the build side, sends each $\mathtt{D}_i$'s bitmap to F and finally, $\mathtt{F}^{\perp}$ (reduced by bitmaps) is probed into each $\mathtt{D}_i$. Similar strategies are proposed in [18, 45] on star schemas.

Algorithm 1 Yannakakis alg. expressed by chaining SQL Server's HJ

```
Input: input tables $R_1, \ldots, R_n$ in an acyclic schema, ordered by a post-order traversal of a rooted join tree
Output: query results $(R_1 \bowtie \ldots \bowtie R_n)$
1: if $n = 1$ then return $R_1$                                   ▷ base case
2: take $R_1$ as the build side to build hash table and bitmap filters;
3: let $R_p$ be the parent table of $R_1$ in the rooted join tree;
4: let $R_p^{\perp}$ be the reduced table $R_p$ pre-filtered by the bitmap filter on $R_1$;
5: return $R_1 \bowtie$ Yannakakis$(R_2, \ldots, R_p^{\perp}, \ldots, R_n)$   ▷ recursion on probe
```

Proof of Instance Optimality for Algorithm 1. We present a much simpler proof than Bagan et al. [6] for arbitrary acyclic joins. Using the same arguments for Q3, we only need to bound the number of backward join probes by $O(\mathrm{OUT})$, as constructing hash tables and performing bottom-up semijoins cost $O(N)$ (Line 2-4).

A key insight is that when joining (in a top-down order) between a parent and a child table in the join tree (Line 5), every probe into the child's hash table is a hit because the parent table has already been semijoined by that child in the bottom-up phase.

As such, starting from the root (reduced) table, the intermediate join results can only grow *monotonically* along the join tree until reaching the final output, bounding all intermediates by $O(\mathrm{OUT})$.

□

Robustness. An interesting by-product of Algorithm 1's instance optimality is robustness to join ordering, i.e. any top-down join orders along *the* join tree have intermediate sizes at most $O(\mathrm{OUT})$. In a star schema (as observed by [13]), the join order between $\mathtt{F}^{\perp}$ and $\mathtt{D}_i$'s can be arbitrary without significant performance impacts. This is a weaker robustness guarantee than Yannakakis and Zhao et al. [44], which use multiple pre-filtering passes (hence repeated scans and materialization overheads) to tolerate any join orders along almost *any* join tree. However, as SQL Server's bitmaps are baked inside a specific query plan (i.e. a fixed join tree), this tree-specific robustness suffices in production as a regress-free approach.

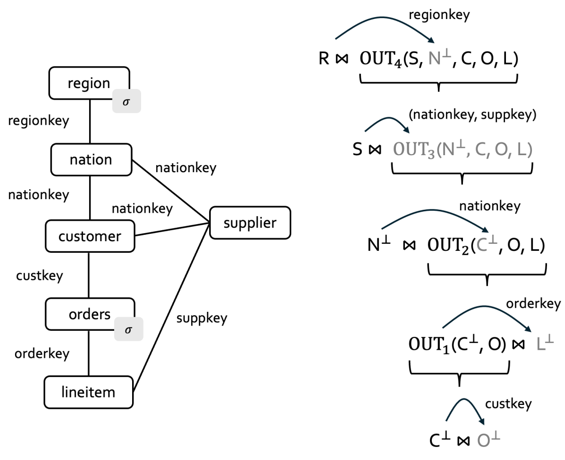

Figure 5: The cyclic join graph of TPC-H Q5 (left) and the bushy plan chosen by SQL Server (right), where the left-hand side of each hash join (⋈) is the build side. Each arrow annotates a bitmap pushdown and labels show the attributes used to construct the bitmap.

3.4 Beyond Instance-optimal Pre-filtering

TPC-H queries involve mostly acyclic joins. Yet in practice, queries may exhibit arbitrary join topologies and relational constructs, such as nested subqueries. In many cases, instance optimality may no longer be achievable. Fortunately, while Algorithm 1 is constrained to right-deep plans (i.e. always a single base table at the build side), SQL Server's plan space encompasses a much broader class of query plans. Critically, SQL Server considers bitmap filtering over *bushy plans* where the build side of a hash join (Line 5) can be intermediate results of an arbitrary subplan rather than a single table R_1.

A distinguishing feature of SQL Server, largely absent from existing pre-filtering implementations, is that **bitmap filters can be constructed from and applied to intermediate join results**[6]. This capability follows naturally from the batch mode hash join implementation (recall Fig. 2): through cost-based decisions, the build side can construct bitmap filters—even multi-column ones—from its output before initiating probe-side execution.

Consider TPC-H Q5's (cyclic) join graph in Fig. 5(left). Fig. 5(right) shows the bushy plan chosen by SQL Server, where R, N, etc., denote tables already reduced by local predicates, and $\mathsf{N}^{\perp}, \mathsf{S}^{\perp}$, etc., denote tables further reduced by pushed-down bitmaps. Let $\mathsf{OUT}_1(\mathsf{C}^{\perp}, \mathsf{O})$ be the intermediate result of $(\mathsf{C}^{\perp} \bowtie \mathsf{O})$. We define $\mathsf{OUT}_2, \mathsf{OUT}_3, \mathsf{OUT}_4$ similarly for the remaining subplans, and OUT for the final output. The second hash join takes S as the build side and creates a double-column bitmap on (`nationkey`, `suppkey`) to pre-filter the probe-side output OUT_3. Here, no pushdown is possible.[7]

The second-to-last hash join treats $(\mathsf{C} \bowtie \mathsf{O})$ as the build side, constructs a bitmap over the resulting `orderkey` values, and then pushes this bitmap to the probe-side of L. This optimizer's departure from strictly right-deep bitmap cascading, as in Algorithm 1, has its own advantages: by materializing $(\mathsf{C} \bowtie \mathsf{O})$ first, SQL Server bases its runtime bitmap decisions on the realized build-side output, thereby strengthening pre-filtering on L. Indeed, the build side is usually far selective than its underlying base tables (i.e. C and O), especially in the presence of post-join filters (e.g., `C.x < O.y`), antijoins, or aggregates (e.g. `HAVING` clause) that further reduce its cardinality.

Output-sensitive Analysis / Semi-robustness. For general join queries such as cyclic joins, instance-optimal bounds are provably unattainable [16, 17]. Consequently, much of the literature turn to worst-case (optimal) guarantees like the AGM bound [5, 19] and its refinements via tree decompositions [30, 31][8]. In practice, however, many analytic workloads expect outputs that are far smaller than their worst-case envelopes, motivating recent theoretical progress on *output-sensitive join algorithms* that extends the spirit of instance optimality to arbitrary joins [4, 12, 21]. Given input tables of total size N, suppose a query plan produces intermediate results of sizes $\mathsf{OUT}_1, \mathsf{OUT}_2, \ldots, \mathsf{OUT}_n$ and eventually OUT (i.e. the final output). A naïve runtime bound is $O(N + \sum_{i=1}^{n} \mathsf{OUT}_i + \mathsf{OUT})$. An output-sensitive plan instead avoids runtime dependence on every intermediate OUT_i, but on a carefully chosen subset of them—ideally only on OUT, as in the instance-optimal case (i.e. N + OUT). This generalization introduces an appealing *semi-robustness* argument for query optimization: performance becomes sensitive to only a few critical intermediates, rather than to all, reducing the vulnerability of cardinality estimation errors.

SQL Server's bitmap filtering mechanisms (discussed in Sec. 3.3-3.4) gives a pragmatic approach to semi-robust join execution for arbitrary queries. Intuitively, a bushy plan decomposes a complex multi-way join into smaller, often nested, acyclic components—some directly over base tables, others over intermediate results—and exploit instance-optimal plans (like those in Sec. 3.3) to each component. By composing these acyclic subplans, SQL Server achieves semi-robustness for general join queries whose runtime depends only on a small set of stitching boundary intermediates. We demonstrate this through an analysis of TPC-H Q5 (Fig. 5).

Output-sensitive Analysis on TPC-H Q5. The top two joins can be analyzed as a star schema of R, S joining into a "fact" table OUT_3, so their execution time is $O(N + \mathsf{OUT}_3 + \mathsf{OUT})$, by the same arguments in Sec. 3.3. The subplan OUT_3 is a (nested) acyclic join among $\mathsf{N}^{\perp}, \mathsf{C}, \mathsf{O}, \mathsf{L}$; since this subplan deviates from Algorithm 1 and materializes OUT_1 first, this acyclic join executes in time $O(N + \mathsf{OUT}_1 + \mathsf{OUT}_3)$. The overall execution time is $O(N + \mathsf{OUT}_1 + \mathsf{OUT}_3 + \mathsf{OUT})$, semi-robust against the other two intermediates. □

4 PRE-FILTERING IN QUERY OPTIMIZATION

Although bitmap filters can offer substantial gains, most previous efforts rely on heuristics to guide pre-filtering decisions [38, 44]. Integrating bitmap filters into optimizers is non-trivial, since they dramatically enlarge the search space—a cross product of candidate plans and filter placements. Recent work have identified such

[6]Wang et al. [35] and EmptyHeaded [3] address cyclic joins by materializing carefully chosen subjoins—typically guided by a hypertree decomposition [19]—such that the remaining join becomes acyclic and amenable to Yannakakis. SQL Server's pre-filtering operates more flexibly: it applies bitmap filtering over arbitrary join queries, whenever predicted to be cost-effective, rather than enforcing explicit cycle elimination.

[7]When estimated to be selective, SQL Server may, as a cost-based decision, build one or more bitmap filters on a single join attribute (e.g., `nationkey` or `suppkey`, instead of both) so that these bitmaps can be pushed down into scans for rowgroup skipping.

[8]For readers interested in worst-case analysis for cyclic joins: SQL Server's bushy plans and bitmap pre-filtering over intermediates can achieve an $O(N^{w} + \mathsf{OUT})$ runtime bound, where $w \geqslant 1$ denotes the so-called *integral edge cover* [19, 24]. Worst-case optimal join algorithms [3, 30, 31, 34] can further improve this theoretical bound.

challenges in production. To mitigate this, Ding et al. [13] exploit robustness across pre-filtering plans on snowflake schemas to skip budgeting plans of similar costs (see Sec. 3.3 on robustness). Zeyl et al. [40] introduced an additional phase of annotating bitmaps on plans in a bottom-up optimizer to make filtering decisions. Both put heavy emphasis on using heuristics to control optimizer overhead. This section examines how SQL Server seamlessly integrates bitmap filters into its Cascades optimizer [20] and some effective heuristics it employs in production. The reader may refer to [10, 14] for in-depth explanations of Cascades/Volcano query optimizers.

4.1 Bitmap Filters in Cascades Optimizer

SQL Server's optimizer integrates bitmaps during logical-to-physical transformations, when each logical join tranforms (via implementation rules [14]) into a hash join operator. By exploiting the top-down nature of Cascades, the optimizer keeps a context `ctx`[9] that contains bitmap statistics (e.g., size and selectivity estimates) propagated from ancestral parts of the plan when optimizing each logical (sub)-expression. This recursive optimization (Fig. 6, modified from [10]) returns an estimated optimal physical plan for a logical subplan rooted at a logical `join`; `cap` is the allocated budget that bounds exploration for early pruning, while `ctx` carries forward the context from previously optimized plan components, including bitmap filter statistics that may affect the current subplan's cost.

When exploring the implementation rule for hash join (Lines 6-20), we first optimize the build side of the `join` (Line 8, $\mathtt{child\_p}_0$), and if the join exhibits low selectivity, a new `ctx'` is created (Line 16) that incorporates the estimated selectivity of the bitmap. This new context `ctx'` is propagated into the probe side optimizations (Line 8, $\mathtt{child\_p}_1$). The cost of a hash join plan `p.cost` is the sum of the hash join (and bitmap) cost, the lowest build, and probe side costs (Line 15). If this `p.cost` is below the current `cap`, we record this plan as the current best `winner_cur` (Line 17) and lower `cap` for early stopping in future candidate plan explorations (Line 18). At any point, if the accumulated cost exceeds `cap` (Line 9) or if no valid plan is found (Line 12), we abandon this hash join exploration. Although it is one of multiple physical join implementations—among sort-merge joins and others—the optimizer explores all alternatives to identify a minimal cost plan. Upon identifying such a plan for an expression, the optimizer caches the plan in a winner table for reuse (Line 22) before returning it to its parent (Line 23).

We show how the optimizer works on Q3 (Fig. 1). Disregarding bitmap filters, it joins ($\mathtt{C} \bowtie \mathtt{O}$) first based on lower estimated cardinality. Taking them into account, the optimizer propagates `C`'s bitmap context when costing the join ($\mathtt{O} \bowtie \mathtt{L}$), i.e. now ($\mathtt{O}^{\perp} \bowtie \mathtt{L}$), where $\mathtt{O}^{\perp}$ is the reduced table, resulting in a smaller estimated join size. In a recursive costing of ($\mathtt{O}^{\perp} \bowtie \mathtt{L}$), the build side is a scan over $\mathtt{O}^{\perp}$ and the optimizer considers another bitmap from $\mathtt{O}^{\perp}$ to `L`, compounding the bitmap selectivities. Thus, the final probe side cost is a scan over $\mathtt{L}^{\perp}$, a relation pre-filtered by the cascaded bitmap contexts. Indeed, Fig. 4 shows much lower estimated row counts (e.g. 3.3M for `L`) than that after only local filtering (e.g. 323M for `L`).

[9]SQL Server uses `ctx` for a range of top-down costing decisions such as `LIMIT` k above a (non-blocking) plan tree to convey that "only k outputs are needed from below".

```
optimize(join, ctx, cap)
1:  if memoized and winner[join, ctx].cost ⩽ cap:
2:    return winner[join, ctx]
3:  else: // try each physical join op and choose the minimal cost plan
4:    winner_cur ⟵ NULL
5:    for each join implementation rule:
6:      // rule  join ⟶ batch mode hash join
7:      p.cost ⟵ hash_join_cost; ctx' ⟵ ctx
8:      for each input $i$ of join // build side, then probe side:
9:        if p.cost ⩾ cap:
10:          go back to 5 // out of bound
11:       child_p$_i$ ⟵ optimize(join.input$_i$, ctx', cap - p.cost)
12:       if child_p$_i$ is NULL:
13:          go back to 5 // no solution
14:       p.winner_p$_i$ ⟵ child_p$_i$
15:       p.cost ⟵ p.cost + child_p$_i$.cost
16:       ctx' ⟵ add_ctx_if_selective(ctx', child_p$_i$.stats)
17:     // find valid plan, update state
18:     if p.cost < cap:
19:       winner_cur ⟵ p
20:       cap ⟵ p.cost // for early stopping in future explorations
21:   ... // omit other physical join operators
22: insert (winner_cur, ctx) into winner
23: return winner_cur
```

Figure 6: SQL Server's (significantly simplified) Cascades optimizer logic emphasizing the costing of hash join and bitmap filters.

4.2 Controlling Optimizer's Overhead

We describe two effective heuristics that have been largely lacking in previous literature that SQL Server employs to prune the optimizer's space when considering bitmap filters as costing contexts.

Selectivity Discretization. "Optimize in a context" prevents memoization reuse—each new bitmap context requires re-optimizing expressions from scratch. SQL Server reduces this overhead by discretizing selectivity estimates into predefined thresholds, such as 75%, 10%, 1%, 0.1% and etc., and when optimizing in a context (say, 3% selectivity), it is truncated into the nearest threshold (e.g., 1% in this case) as the optimizer's context. This limits the context alternatives while maintaining meaningful cost distinctions.

Early Stopping. The optimizer usually prioritizes bitmap-aware hash join implementations (Line 6, Fig. 6), which are likely to yield low-cost plans under effective pre-filtering. When a promising physical plan is found, the optimizer tightens the `cap` to the cost of this plan (Line 20), allowing aggressive pruning of less promising alternatives in subsequent explorations.

5 EXPERIMENTS AND RESULTS

We show experimentally using TPC-H (SF100) the results discussed so far. All experiments were run on a Microsoft Azure VM that has an 8-core Intel Xeon Platinum 8370C processor (hyper-threading) and 64 GB of RAM. The VM hosts SQL Server 2025, having bitmap filters as default, running multi-threaded on Windows 10 Enterprise.

Runtime Analysis. Table 1 compares SQL Server performance of bitmaps (i.e., the default build) versus no bitmaps on all 22 TPC-H queries (median of 10 runs). Results show consistent speedups and 7 cases achieve over 2× speedups (marked blue). Plan analysis shows that 12 (acyclic) queries execute using instance-optimal plans. For cyclic joins (Q5), or join across nested subqueries (e.g., Q17), bushy plans where bitmaps are created on intermediate build sides yield substantial performance gains. In other acyclic cases such as Q8, Q9,

ID	Shape of join graph	Runtime speedup	Inst.-opt.?	#rows to join	SQL Srv. reads (%)	Yannak. reads (%)
1	No join	—	—	—	—	—
2	Line	**3.47×**	✓	81.1M	0.43%	0.21%
3	Line (C-O-L)	1.91×	✓	399.4M	**5.23%**	**1.25%**
4	A semijoin	1.92×	✓	385.1M	5.28%	5.14%
5▲	Cyclic join	**2.33×**	—	638.8M	**4.2%**	**1.21%**
6	No join	—	—	—	—	—
7▲	Line	**3.00×**	✓	348.4M	**4.15%**	**0.81%**
8▲	Snowflake	**2.86×**	✗	661.8M	**1.92%**	**0.16%**
9	Snowflake	1.47×	✗	832.2M	8.23%	8.22%
10	Line (N-C-O-L)	1.23×	✓	168.8M	12.51%	12.01%
11	Line	1.75×	✓	81.1M	8.01%	8.01%
12	Single join	1.21×	✓	153.2M	3.97%	3.95%
13	An outer join	—	—	—	—	—
14	Single join	1.14×	✓	27.5M	49.98%	49.98%
15	Single join	1.07×	✓	23.7M	95.78%	95.78%
16	Antijoin & join	1.03×	✓	83.0M	17.91%	17.91%
17	Correlate joins	**2.55×**	—	51.5M	1.22%	1.21%
18▲	Line (C-O-L)	1.22×	✗	1215.1M	49.39%	49.39%
19▲	Single join	1.82×	✓	25.8M	0.43%	0.39%
20	Correlate joins	**2.01×**	—	172.3M	0.2%	0.07%
21	Correlate joins	**2.36×**	—	379.4M	**21.43%**	**19.24%**
22	An antijoin	1.27×	✓	155.8M	**17.41%**	**15.94%**

Table 1: TPC-H (SF=100) results. ✓ (✗) mark instance-optimal plans (not) chosen; — are inapplicable cases (e.g. cyclic or correlated joins); ▲ denotes optimizer slowdown. Last three columns show total numebr of rows (in millions) for all input tables reduced by local filters, % after pre-filtering of SQL Server, and % after Yannakakis / PT.

and Q18, SQL Server departs from strict instance-optimality, yet its pre-filtering effects stay on par with Yannakakis. This is because the optimizer omits creating futile bitmaps for joins estimated as non-selective (e.g. pk/fk), while other impactful bitmap filters trigger a reordering of the entire join sequence. Q1, Q6, and Q13 have either no join or only outer joins, and do not benefit from bitmap filters.

Optimizer Overhead. The ▲ symbols in Table 1 reveal that bitmap costing imposes certain query planning slowdowns. Only 5 queries exhibit perceptible optimizer overheads, while others show comparable or even faster plan search (all variations are within ±5%).

Pre-filtering Gains & Gaps. The last three columns of Table 1 show the percentage of rows remain after pre-filtering of SQL Server and Yannakakis (or PT [38]). The baseline row counts (i.e. #rows to join) are the total data volume passing local predicates and fed into join execution. For the last column (i.e. Yannakakis reads), we hand-picked a strong semijoin schedule (no false positives!): following join trees for acyclic joins, and using multi-pass PT-style semijoin propagation for others. In Example 2.1 (Q3), the two percentages are $\texttt{C}+\texttt{O}^{\perp}+\texttt{L}^{\perp}$ and $\texttt{C}^{\top}+\texttt{O}^{\top}+\texttt{L}^{\perp}$, divided by the baseline C+O+L. SQL Server bitmaps attain dramatic reductions, some (Q2, Q19, and Q20) rejecting over 99% non-joining rows and many (e.g. Q9, Q11, Q17) show near-optimal pre-filtering. Gaps >1% (marked red) arise from: (1) SQL Server (at most a single-pass of bitmap pre-filtering) usually does not aggressively circulate bitmaps across the entire join graph as Yannakakis' (or PT's) multi-pass semijoins; (2) the cumulative impact of Bloom filter false positive rates—along with skipped, less effective pre-filtering—further widens the gap.

6 CONCLUSION & FUTURE OUTLOOK

This paper unveils a surprising discovery: SQL Server has subsumed Yannakakis in its design! We have proven so for acyclic joins and

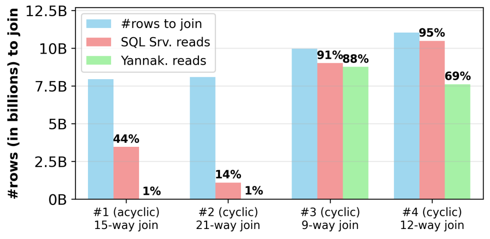

Figure 7: Pre-filtering on four adversarially selected (long-running) production queries; blue bars are total row counts. The latter two bars show #rows after pre-filtering of SQL Server (scheduled by its optimizer) and Yannakakis (hand-fixing an effective semijoin order).

semi-robust runtimes for cyclic ones. This is not only a theoretical elegance; but a battle-tested approach that has proven effective across the majority of real-world production workloads.

Inevitably, however, production contains cases of hyper-complex queries that push the boundaries of any optimizer. Our telemetry data shows that 7% of join queries involve beyond 8-way (up to 1,000-table!) joins, 15% include hybrid join types (outer, range, and anti), and far fewer pk/fk conditions were strictly enforced. In these cases, accumulating mis-estimation can make bitmap placement fragile. Fig. 7 showcases four such adversarial queries where neither SQL Server's nor Yannakakis' (or PT) bitmap choices are ideal: in query #1, #2, and #4, SQL Server's (rather conservative) costing misses profitable pre-filtering opportunities that extensive semijoin passes (à la Yannakakis) would have caught; whereas in query #3 (and partly #4), the prevalence of non-selective joins turns aggressive Yannakakis-style pre-filtering largely overhead for marginal gains. Collectively, these cases call for a new generation of grounded, reliable, and versatile pre-filtering techniques.

Bitmap-aware Cardinality Estimation. Recent advances in cardinality estimation, from learned models (e.g. [36]) to pessimistic approaches [11, 42], have markedly improved cost models and predictions [26, 27]. However, existing cardinality estimators often ignore how bitmaps reshape intermediate result sizes. Accounting for bitmap impacts into cardinality models is crucial for making reliable, cost-based pre-filtering decisions.

Pre-filtering and Robustness Beyond Acyclicity. Pre-filtering remains underexploited for non-acyclic joins, such as cyclic joins (e.g. Fig. 7 #2-#4) and non-equijoins (e.g. band joins [28]). Current systems lack first-principles counterparts to Yannakakis in these regimes. One promising direction is to bridge the gap of pre-filtering and worst-case optimal joins [30, 31], aiming for performance (semi)-robustness (see Sec. 3.4) on any join graphs; another is to explore new pre-filtering primitives for non-equijoins like range queries [15, 41], where traditional bitmaps become inadequate.

Towards New Paradigms. The core promise of cost-based bitmap filtering extends well beyond single-node relational databases. Distributed database engines, bearing high data shuffling costs [37]; graph analytics, often dominated by many-to-many joins (or even recursive joins [43]); streaming platforms, subject to volatile workloads and evolving data; or GPU-accelerated analytics, constrained by memory footprint and data transfer overheads [29], each can benefit from tailored pre-filtering methods. Systematic exploration in these domains may surface new optimization principles.

(⋆) ACKNOWLEDGMENTS

The bitmap pre-filtering infrastructure described in this paper were shipped in SQL Server 2014 and have been continuously improved in subsequent releases. We are deeply grateful to the original engineering team whose vision and expertise brought these innovations to life. Specifically, the design and implementation were made possible by the substantial contributions of: Ciprian Clinciu, Campbell Fraser, Cesar Galindo-Legaria, Milind Joshi, Michal Nowakiewicz, Vassilis Papadimos, Andrew Richardson, and Aleksandras Surna.

In addition, the authors thank the anonymous CIDR reviewers for their insightful and constructive comments.

REFERENCES

[1] 2019. Getting started with Oracle Database In-Memory Part IV - Joins In The IM Column Store. https://blogs.oracle.com/in-memory/post/getting-started-with-oracle-database-in-memory-part-iv-joins-in-the-im-column-store.

[2] 2021. Best Practices for Optimizing Your DBT and Snowflake Deployment. https://www.snowflake.com/wp-content/uploads/2021/10/Best-Practices-for-Optimizing-Your-dbt-and-Snowflake-Deployment.pdf.

[3] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4, Article 20 (Oct. 2017), 44 pages. https://doi.org/10.1145/3129246

[4] Mahmoud Abo Khamis, Ahmet Kara, Dan Olteanu, and Dan Suciu. 2025. Output-Sensitive Evaluation of Regular Path Queries. *Proc. ACM Manag. Data* 3, 2, Article 105 (June 2025), 20 pages. https://doi.org/10.1145/3725242

[5] Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size Bounds and Query Plans for Relational Joins. In *FOCS*. IEEE Computer Society, 739–748.

[6] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *CSL (Lecture Notes in Computer Science)*, Vol. 4646. Springer, 208–222.

[7] Philip A. Bernstein and Dah-Ming W. Chiu. 1981. Using Semi-Joins to Solve Relational Queries. *J. ACM* 28, 1 (1981), 25–40.

[8] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *Proc. VLDB Endow.* 17, 11 (2024), 3215–3228.

[9] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*. www.cidrdb.org, 225–237.

[10] Nicolas Bruno, César A. Galindo-Legaria, Milind Joshi, Esteban Calvo Vargas, Kabita Mahapatra, Sharon Ravindran, Guoheng Chen, Ernesto Cervantes Juárez, and Beysim Sezgin. 2024. Unified Query Optimization in the Fabric Data Warehouse. In *SIGMOD Conference Companion*. ACM, 18–30.

[11] Kyle B. Deeds, Dan Suciu, and Magdalena Balazinska. 2023. SafeBound: A Practical System for Generating Cardinality Bounds. *Proc. ACM Manag. Data* 1, 1 (2023), 53:1–53:26.

[12] Shaleen Deep, Hangdong Zhao, Austen Z. Fan, and Paraschos Koutris. 2024. Output-sensitive Conjunctive Query Evaluation. *Proc. ACM Manag. Data* 2, 5, Article 220 (2024), 24 pages. https://doi.org/10.1145/3695838

[13] Bailu Ding, Surajit Chaudhuri, and Vivek R. Narasayya. 2020. Bitvector-aware Query Optimization for Decision Support Queries. In *SIGMOD*. ACM, 2011–2026.

[14] Bailu Ding, Vivek R. Narasayya, and Surajit Chaudhuri. 2024. Extensible Query Optimizers in Practice. *Found. Trends Databases* 14, 3-4 (2024), 186–402.

[15] Navid Eslami, Ioana O. Bercea, and Niv Dayan. 2025. Diva: Dynamic Range Filter for Var-Length Keys and Queries. *Proc. VLDB Endow.* 18, 11 (2025), 3923–3936.

[16] Austen Z. Fan, Paraschos Koutris, and Hangdong Zhao. 2023. The Fine-Grained Complexity of Boolean Conjunctive Queries and Sum-Product Problems. In *ICALP (LIPIcs)*, Vol. 261. Schloss Dagstuhl, 127:1–127:20.

[17] Austen Z. Fan, Paraschos Koutris, and Hangdong Zhao. 2024. Tight Bounds of Circuits for Sum-Product Queries. *Proc. ACM Manag. Data* 2, 2 (2024), 87.

[18] Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy, and Jignesh M. Patel. 2022. SQLite: Past, Present, and Future. *Proc. VLDB Endow.* 15, 12 (2022), 3535–3547.

[19] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 1999. Hypertree decompositions and tractable queries *(PODS '99)*. ACM, New York, NY, USA, 21–32.

[20] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[21] Xiao Hu. 2025. Output-Optimal Algorithms for Join-Aggregate Queries. *Proc. ACM Manag. Data* 3, 2, Article 104 (June 2025), 27 pages. https://doi.org/10.1145/3725241

[22] James B. Rothnie Jr., Philip A. Bernstein, Stephen Fox, Nathan Goodman, Michael Hammer, Terry A. Landers, Christopher L. Reeve, David W. Shipman, and Eugene Wong. 1980. Introduction to a System for Distributed Databases (SDD-1). *ACM Trans. Database Syst.* 5, 1 (1980), 1–17.

[23] Srikanth Kandula, Laurel J. Orr, and Surajit Chaudhuri. 2022. Data-induced predicates for sideways information passing in query optimizers. *VLDB J.* 31, 6 (2022), 1263–1290.

[24] Paraschos Koutris, Shaleen Deep, Austen Z. Fan, and Hangdong Zhao. 2025. The Quest for Faster Join Algorithms (Invited Talk). In *ICDT (LIPIcs)*, Vol. 328. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:12.

[25] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. 2011. SQL server column store indexes. In *SIGMOD Conference*. ACM, 1177–1184.

[26] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.

[27] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2025. Still Asking: How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 18, 12 (2025), 5531–5536.

[28] Rundong Li, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Near-Optimal Distributed Band-Joins through Recursive Partitioning. In *SIGMOD*. 2375–2390.

[29] Yinan Li, Bailu Ding, Ziyun Wei, Lukas M. Maas, Momin Al-Ghosien, Spyros Blanas, Nicolas Bruno, Carlo Curino, Matteo Interlandi, Craig Peeper, Kaushik Rajan, Surajit Chaudhuri, and Johannes Gehrke. 2025. Scaling GPU-Accelerated Databases beyond GPU Memory Size. *Proc. VLDB Endow.* 18, 11 (2025), 4518–4531.

[30] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case Optimal Join Algorithms. *J. ACM* 65, 3, Article 16 (March 2018), 40 pages. https://doi.org/10.1145/3180143

[31] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.* 42, 4 (Feb. 2014), 5–16. https://doi.org/10.1145/2590989.2590991

[32] Sukriti Ramesh, Odysseas Papapetrou, and Wolf Siberski. 2009. Optimizing distributed joins with bloom filters. In *ICDCIT 2008*. Springer, 145–156.

[33] Mihail Stoian, Andreas Zimmerer, Skander Krid, Amadou Ngom, Jialin Ding, Tim Kraska, and Andreas Kipf. 2025. Parachute: Single-Pass Bi-Directional Information Passing. *Proc. VLDB Endow.* 18, 10 (2025), 3299–3311.

[34] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*. OpenProceedings.org, 96–106.

[35] Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. 2025. Yannakakis+: Practical Acyclic Query Evaluation with Theoretical Guarantees. *Proc. ACM Manag. Data* 3, 3 (2025), 235:1–235:28.

[36] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: A New Cardinality Estimation Framework for Join Queries. *Proc. ACM Manag. Data* 1, 1 (2023), 41:1–41:27.

[37] Yifei Yang and Xiangyao Yu. 2025. Accelerate Distributed Joins with Predicate Transfer. *Proc. ACM Manag. Data* 3, 3 (2025), 122:1–122:27.

[38] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. 2024. Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries. In *CIDR*. www.cidrdb.org.

[39] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes *(VLDB '81)*. VLDB Endowment, 82–94.

[40] Tim Zeyl, Qi Cheng, Reza Pournaghi, Jason Lam, Weicheng Wang, Calvin Wong, Chong Chen, and Per-Åke Larson. 2025. Including Bloom Filters in Bottom-up Optimization. In *SIGMOD Conference Companion*. ACM, 703–715.

[41] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *SIGMOD Conference*. ACM, 323–336.

[42] Haozhe Zhang, Christoph Mayer, Mahmoud Abo Khamis, Dan Olteanu, and Dan Suciu. 2025. LpBound: Pessimistic Cardinality Estimation Using ℓ_p-Norms of Degree Sequences. *Proc. ACM Manag. Data* 3, 3 (2025), 184:1–184:27.

[43] Hangdong Zhao, Zhenghong Yu, Srinag Rao, Simon Frisk, Zhiwei Fan, and Paraschos Koutris. 2025. FlowLog: Efficient and Extensible Datalog via Incrementality. *CoRR* abs/2511.00865 (2025).

[44] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. 2025. Debunking the Myth of Join Ordering: Toward Robust SQL Analytics. *Proc. ACM Manag. Data* 3, 3 (2025), 146:1–146:28.

[45] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking Ahead Makes Query Plans Robust: Making the Initial Case with in-Memory Star Schema Data Warehouse Workloads. *Proc. VLDB Endow.* 10, 8 (2017), 889–900.