Bitvector-aware Query Optimization for Decision Support Queries

Bailu Ding
Microsoft Research
badin@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

ABSTRACT

Bitvector filtering is an important query processing technique that can significantly reduce the cost of execution, especially for complex decision support queries with multiple joins. Despite its wide application, however, its implication to query optimization is not well understood.

In this work, we study how bitvector filters impact query optimization. We show that incorporating bitvector filters into query optimization straightforwardly can increase the plan space complexity by an exponential factor in the number of relations in the query. We analyze the plans with bitvector filters for star and snowflake queries in the plan space of right deep trees without cross products. Surprisingly, with some simplifying assumptions, we prove that, the plan of the minimal cost with bitvector filters can be found from a linear number of plans in the number of relations in the query. This greatly reduces the plan space complexity for such queries from exponential to linear.

Motivated by our analysis, we propose an algorithm that accounts for the impact of bitvector filters in query optimization. Our algorithm optimizes the join order for an arbitrary decision support query by choosing from a linear number of candidate plans in the number of relations in the query. We implement our algorithm in a commercial database DBMS-X as a transformation rule. Our evaluation on both industry standard benchmarks and customer workload shows that, compared with DBMS-X, our technique reduces the total CPU execution time by 22%-64% for the workloads, with up to two orders of magnitude reduction in CPU execution time for individual queries.

SIGMOD '20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389769

CCS CONCEPTS

• **Information systems → Query optimization**; **Query planning**.

KEYWORDS

database; query optimization; query processing; bitvector filter; Bloom filter; join order enumeration

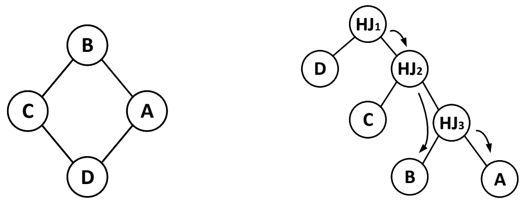
ACM Reference Format:
Bailu Ding, Surajit Chaudhuri, and Vivek Narasayya. 2020. Bitvector-aware Query Optimization for Decision Support Queries. In *2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389769

1 INTRODUCTION

Bitvector filters, including bitmap or hash filter [6, 7, 18], Bloom filter and its variants [2, 7, 15, 24, 32], perform 'probabilistic' semi-join reductions to effectively prune out rows that will not qualify join conditions early in the query execution pipeline. Because they are easy to implement and low in overhead, bitvector filters are widely used in commercial databases [13, 17, 21, 23].

Prior work on using bitvector filters has heavily focused on optimizing its effectiveness and applicability for query processing. One line of prior work has explored different schedules of bitvector filters for various types of query plan trees to optimize its effect on query execution [10–12]. Many variants of bitvector filters have also been studied that explore the trade-off between the space and accuracy [2, 7, 9, 15, 24, 32].

In query processing, bitvector filters are mostly used in hash joins [10–12]. Specifically, the commercial database DBMS-X implements the bitvector filter scheduling algorithm following [18] (Section 2). At a high level, a *single* bitvector filter is created with the equi-join columns at a hash join operator and is pushed down to the *lowest* possible level of the subplan rooted at the probe side. Figure 1 shows an example of applying bitvector filters to a query plan . Figure 1a shows the join graph of the query and Figure 1b shows its query plan, where the arrow in Figure 1b points from the operator that creates the bitvector filter to the operator where the bitvector filter is pushed down to. As shown in Figure 1b, a bitvector filter is created from the build side



(a) Join graph

(b) Plan with bitvector filters

Figure 1: Example of pushing down bitvector filters for a query plan joining relations A, B, C, D

of each hash join operator (HJ_1, HJ_2, and HJ_3). Since C only joins with B, the bitvector filter created from HJ_2 bypasses HJ_3 and is pushed down to B. Similarly, because D joins with both A and C, the bitvector filter created from HJ_1 consists of columns from both A and C. Thus, the lowest possible level to push down this bitvector filter is HJ_2. Bitvector filters can also be adapted for merge joins.

Surprisingly, despite the wide application of and decades of research on bitvector filters for query processing, the impact of bitvector filters on query optimization is not well understood. To the best of our knowledge, most state-of-the-art DBMSs add bitvector filters to the query plans produced by the query optimizer as a post-processing step.

Neglecting the impact of bitvector filters in query optimization can miss out opportunities of significant plan quality improvement. Figure 2 shows an example of such an opportunity with a query using the JOB [25] benchmark schema:

```
SELECT COUNT(*)
FROM movie_keyword mk, title t, keyword k
WHERE mk.movie_id = t.id AND mk.keyword_id = k.id
AND t.title LIKE '%(' AND k.keyword LIKE '%ge%'
```

Figure 2a shows the join graph of the query, where each edge is annotated with the join cardinality of the corresponding tables. Figure 2b shows the best query plan P_1 without using bitvector filters. Each operator is annotated with the number of tuples after filter predicates being applied and the operator cost ($tuple/cost$).

Figure 2c shows the query plan after adding bitvector filters to P_1 as a post-processing step. Although the cost of P_1 is reduced after adding bitvector filters, it still costs 3× as much as the best plan when the impact of bitvector filters is considered during query optimization (Figure 2d).

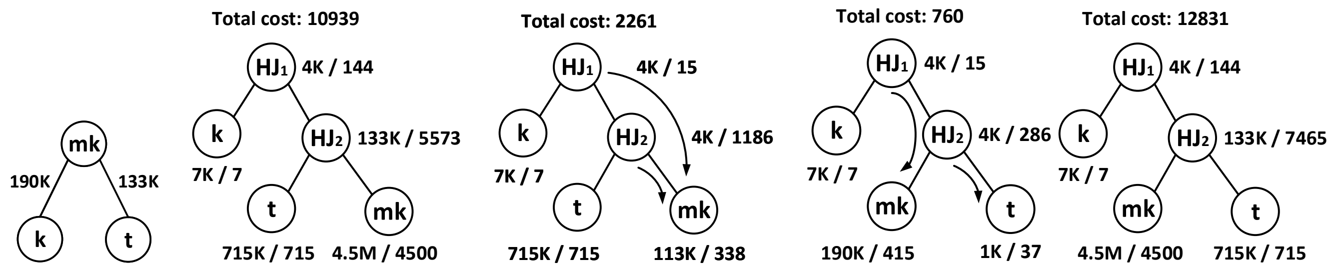
Because P_2 is more expensive than P_1 without using bitvector filters (Figure 2e), the optimizer will choose P_1 as the best plan if it neglects the impact of bitvector filters during query optimization. Therefore, the optimizer will choose a much worse plan (Figure 2c) if the bitvector filters are only considered as a post-processing step after query optimization.

Incorporating bitvector filters into query optimization is surprisingly challenging. Existing top-down or bottom-up dynamic programming (DP) based query optimization framework cannot directly integrate the bitvector filters into its optimization, because the effect of bitvector filters can violate the substructure optimality property in DP. In a DP-based query optimization framework, either top-down or bottom-up, an optimal subplan is stored for each subset $\mathcal{A}$ of relations involved in a query. With bitvector filters, however, in addition to the relations in $\mathcal{A}$, the optimal subplan also depends on what bitvector filters are pushed down to $\mathcal{A}$ and how these bitvector filters apply to the relations in $\mathcal{A}$ based on the structure of the subplan. For example, Figure 2c and Figure 2d both contain a subplan of joining $\{mk, t\}$. The cost of the two subplans, however, is more than 3× different due to the different bitvector filters pushed down to the subplan.

Incorporating bitvector filters into query optimization straightforwardly can be expensive. Similar to supporting interesting orders in query optimization [34], the number of optimal substructures can increase by an exponential factor in the number of relations to account for the impact of various combinations of bitvector filters.

Surprisingly, prior work has shown that, under limited conditions, different join orders results in similar execution cost when bitvector filters are used. LIP [38] analyzes the impact of Bloom filters for star schema with a specific type of left deep trees, where the fact table is at the bottom. They observe that, if bitvector filters created from dimension tables are pushed down to the fact table upfront, plans with different permutations of dimension tables have similar cost.

Motivated by this observation, we study the impact of bitvector filters on query optimization. We focus on an important class of queries, i.e., complex decision support queries, and the plan space of right deep trees without cross products, which is shown to be an important plan space for such queries [12, 17]. **Our first contribution** is to systematically analyze the impact of bitvector filters on optimizing the join order of star and snowflake queries with primary-key-foreign-key (PKFK) joins in the plan space of right deep trees without cross products (Section 3-5). Prior work has shown that, without bitvector filters, the number of plans for star and snowflake queries in this plan space is exponential in the number of relations in the query [31]. Intuitively, the plan space complexity should further increase with bitvector filters integrated into query optimization due to violation of substructure optimality. **Our key observation** is that, when the bitvector filters have no false positives, certain join orders can be equivalent or inferior to others with respect to the cost function C_{out} [28, 30], regardless of the query parameters or the data distribution. By exploiting this observation, we prove that, with some simplifying assumption, for star and snowflake queries with PKFK joins, the plan of the



(a) Join graph w/ cardinality **(b) Best plan P_1 without bitvector filters** **(c) Post-process P_1 by adding bitvector filters** **(d) Best plan P_2 with bitvector filters** **(e) P_2 is suboptimal without bitvector filters**

Figure 2: Example of ignoring bitvector filters in query optimization results in a suboptimal plan

minimal C_{out} with bitvector filters can be found by choosing from a **linear** number of plans in the number of relations in the query in this plan space. To the best of our knowledge, this is the first work that analyzes the interaction between bitvector filters and query optimization for a broad range of decision support queries and a wide plan search space.

While star and snowflake queries are common patterns for decision support queries, in practice, the join graphs can include multiple fact tables and non-PKFK joins. **Our second contribution** is to propose an algorithm that optimizes the join order for arbitrary decision support queries motivated by our analysis (Section 6). Our technique applies to queries with arbitrary join graphs. Since creating and applying bitvector filters adds overhead, we further optimize our algorithm by selectively adding bitvector filters based on their estimated benefit (Section 6.3). Our algorithm can be integrated into a query optimization framework as a transformation rule [19, 20]. Depending how a DBMS handles bitvector filters in query optimization, we propose three options to integrate our technique into the DBMS (Section 6.4).

We implement our algorithm in a commercial database DBMS-X (Section 7.1). We evaluate our technique on industry benchmarks TPC-DS [1] and JOB [25] as well as a customer workload (Section 7). We show that, comparing to the query plans produced by DBMS-X, our technique reduces the total CPU execution time of a workload by 22% to 64%, with up to two orders of magnitude reduction in CPU execution time for individual queries. We show that our technique is especially effective in reducing execution cost for expensive queries with low selectivity, where right deep trees is a preferable plan space [12, 17].

We discuss related work in Section 8 and conclude the work in Section 9.

2 BITVECTOR FILTER ALGORITHM

In this section, we describe the details of bitvector filters creation and push-down algorithm following [18].

At a high level, each hash join operator creates a *single* bitvector filter from the equi-join columns on the build side. This bitvector filter is then pushed down to the lowest possible level on the subtree rooted at the probe side so that it can eliminate tuples from that subtree as early as possible.

Algorithm 1 shows how to push down bitvectors given a query plan. The algorithm takes a query plan as its input. Starting from the root of the query plan, the set of bitvector pushed down to the root is initialized to be empty (line 3) and each operator is then processed recursively in a pre-order traversal. At each operator, it takes the set of bitvector filters pushed down to this operator as an input. If the operator is a hash join, a bitvector filter is created from the build side with the equi-join columns of this hash join as the keys of the bitvector filter and is added to the set of bitvector filters applied to the probe side of this hash join (line 8-10). Now consider every bitvector filter that is pushed down to this hash join operator. If one of the child operator of the join operator contains all the columns in the bitvector filter, the bitvector filter is added to the set of bitvector filters pushed down to this child operator; otherwise, the bitvector filter cannot be pushed down further, and it is added to the set of bitvector filters pushed down to this join operator (line 12 - 23). If the set of bitvector filters pushed down to this join operator is non-empty, add a filter operator on top of this join operator to apply the bitvector filters. In this case, update the root of this subplan to the filter operator (line 24-29). Recursively process the bitvector filters pushed down to the child operators and update the children accordingly (line 30 - 33). Finally, return the updated root operator of this subplan (line 34). An example of creating and pushing down bitvector filters with Algorithm 1 is shown in Figure 1.

3 OVERVIEW AND PRELIMINARIES

3.1 Overview

We start with the properties of bitvector filters and the cost function (Section 3). We then show that, with bitvector filters,

```
PlanPushDown(plan):
    Input: Query plan plan
    Output: New query plan plan′ with bitvectors
    root ← plan.GetRootOperator()
    plan′ ← plan
    root′ ← OpPushDown(op, ∅)
    plan′.SetRootOp(root′)
    return plan′

OpPushDown(op, B):
    Input: Operator op, set of bitvectors B
    Output: New operator op′ with bitvectors
    residualSet ← ∅
    pushDownMap ← ∅
    if op is Hash Join then
        b ← bitvector created from
          op.GetBuildChild()
        pushDownMap[op.GetProbeChild()] ←
          pushDownMap[op.GetProbeChild()] ∪ b
    end
    foreach bitvector b in B do
        ops ← ∅
        foreach child c of operator op do
            if b can be pushed down to c then
                ops ← ops ∪ {c}
            end
        end
        if |ops| ≠ 1 then
          residualSet ← residualSet ∪ {b}
        else
            pushDownMap[c] ←
              pushDownMap[c] ∪ {b}
        end
    end
    op′ ← op
    if residualSet ≠ ∅ then
        filterOp ← CreateFilterOp(op, residualSet)
        filterOp.AddChild(op)
        op′ ← filterOp
    end
    foreach child c of op do
        c′ ← OpPushDown(c, pushDownMap[c])
        op.UpdateChild(c, c′)
    end
    return op′
```

Algorithm 1: Push down bitvectors

the number of candidate plans of the minimal cost is linear for star and snowflake queries with PKFK joins in the plan space of right deep trees without cross products (Section 4 and Section 5). We finally describe the general bitvector-aware query optimization *algorithm* for arbitrary decision support queries and how to integrate it with a Volcano / Cascades style optimizer (Section 6). Table 1 summarizes the notations. Table 2 summarizes the results of our analysis.

Table 1: List of notations

Notation	Description
q	a query
R	a relation
$\mathcal{R}$	a set of relations
$\mathcal{T} = T(R_1, \cdots, R_n)$	a right deep tree with R_1 as the right most leaf and R_n as the left most leaf
$S(R_1, \cdots, R_n, B_1, \cdots, B_m)$	join of relations $R_1, \cdots, R_n$ after applying bitvector filters created from $B_1, B_2, \cdots, B_m$, where B_i is either a base relation or a join result. We omit $B_1, \cdots, B_m$ when they are clear from the context. We use the notation interchangeably with $\bowtie$
$\|R\|$	cardinality of a base relation or an intermediate join result after applying bitvector filters
R_1/R_2	semi join of R_1 with R_2, where $R_1/R_2 \subseteq R_1$
$R_1/(R_2, \cdots, R_n)$	semi join of R_1 with $R_2, \cdots, R_n$, where $R_1/(R_2, \cdots, R_n) \subseteq R_1$
$R_1 \rightarrow R_2$	the join columns of R_1 and R_2 is a key in R_2. If the join columns form a primary key in R_2, then $R_1 \rightarrow R_2$ is a primary-key-foreign-key join
C_{out}	cost function (See Section 3.3)
$\prod_{R_1}(R_2)$	project out all the columns in R_1 from R_2, where the columns in R_2 is a superset of that in R_1. The resulting relation has the same number of rows as R_2 but less number of columns per row

3.2 Properties of bitvector filters

We start with the properties of bitvector filters:

Property 1. ***Commutativity**: $R/(R_1, R_2) = R/(R_2, R_1)$*

Property 2. ***Reduction**: $|R/R_1| \leq |R|$*

Property 3. ***Redundancy**: $(R_1 \bowtie R_2)/R_2 = R_1 \bowtie R_2$*

Property 4. ***Associativity**: $R/(R_1, R_2) = (R/R_1)/R_2$ if there are no false positives with the bitvector filters created from (R_1, R_2), R_1, and R_2.*

Now we prove the absorption rule of bitvector filters for PKFK joins. The absorption rule says that, if R_1 joins R_2 with a key in R_2, the result of joining R_1 and R_2 is a subset of the result of semi-joining R_1 and R_2. Formally,

Table 2: Summary of the plan space complexity for star and snowflake queries with unique key joins

join graph	graph size	# of relations	original complexity	complexity w/ our analysis	candidate plans with minimal C_{out}
star	n dimension tables	$n+1$	exponential to n	$n+1$	$T(R_0, R_1, \cdots, R_n)$, $\{T(R_k, R_0, R_1, R_2, \cdots, R_{k-1}, R_{k+1}, \cdots, R_n), 1 \leq k \leq n\}$
snowflake	m branches of lengths $n_i, 1 \leq i \leq m$	$n+1, n = \sum_{i=1}^{m} n_i$	exponential to n	$n+1$	$T(R_0, R_{1,1}, \cdots, R_{1,n_1}, \cdots, R_{n,1}, \cdots, R_{n,n_m})$, $\{T(R_{i,a_1}, \cdots, R_{i,a_{n_1}}, R_0, R_{1,n_1}, \cdots, R_{i-1,1}, \cdots, R_{i-1,n_{i-1}}, R_{i+1,1}, \cdots, R_{i+1,n_{i+1}}, \cdots, R_{n,1}, \cdots, R_{n,n_m})\}$ (see Section 5 for $a_1, \cdots, a_{n_1}$)

Lemma 1. ***Absorption rule**: If $R_1 \rightarrow R_2$, then $R_1/R_2 \supseteq \prod_{R_1}(R_1 \bowtie R_2)$ and $|R_1/R_2| \geq |R_1 \bowtie R_2|$. The equality happens if the bitvector filter created from R_2 has no false positives.*

Proof. For every tuple r in R_1, it can join with a tuple in R_2 if and only if the join columns in r exist in R_2. Because $R_1 \rightarrow R_2$, there is at most one such tuple in R_2. Thus, $R_1/R_2 \subseteq \prod_{R_1}(R_1 \bowtie R_2)$. □

3.3 Cost function

Since our analysis focuses on the quality of logical join ordering, we measure the intermediate result sizes (i.e., C_{out}) as our cost function similar to prior work on join order analysis [28, 30]. In practice, C_{out} is a good approximation for comparing the actual execution cost of plans.

C_{out} measures the cost of a query plan by the sum of intermediate result sizes. Because bitvector filters also impact the cardinality of a base table, we adapt C_{out} to include the base table cardinality as well. Formally,

$$C_{out}(T) = \begin{cases} |T| & \text{if } T \text{ is a base table} \\ |T| + C_{out}(T_1) + C_{out}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases} \tag{1}$$

Note that $|T|$ has reflected the impact of bitvector filters, where $|T|$ represents the cardinality after bitvector filters being applied for both base tables and join results.

4 ANALYSIS OF STAR QUERIES WITH PKFK JOINS

We define star queries with PKFK joins as the following:

Definition 1. ***Star query with PKFK joins:** Let $\mathcal{R} = \{R_0, R_1, \cdots, R_n\}$ be a set of relations and q be a query joining relations in $\mathcal{R}$. The query q is a star query with PKFK joins if $R_0 \rightarrow R_k$ for $1 \leq k \leq n$. R_0 is called a fact table, and $R_k, 1 \leq k \leq n$, is called a dimension table.*

Figure 3 shows an example of a star query, where R_0 is the fact table and R_1, R_2, R_3 are dimension tables.

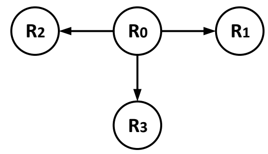
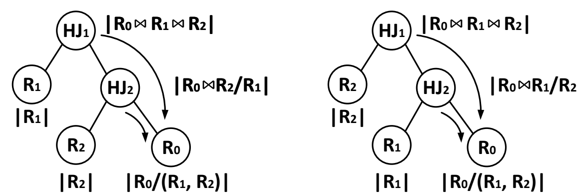


Figure 3: Star query graph with PKFK joins, where the fact table is R_0 and dimension tables are R_1, R_2, R_3



(a) Plan P_1 **(b) Plan P_2**

Figure 4: Example of two plans of a star query $\{R_0, R_1, R_2\}$ with PKFK joins using bitvector filters. Each operator is annotated with the intermediate result size. Plan P_1 and P_2 have different join orders of dimension tables but the same cost.

Now we analyze the plan space complexity for star queries with PKFK joins. We show that, in the plan space of right deep trees without cross products, we can find the query plan of the minimal cost (under the cost function from Section 3.3) from $n+1$ plans with bitvector filters if the bitvector filters have no false positives, where $n+1$ is the number of relations in the query. In contrast, the original plan space complexity for star queries in this plan space is exponential to n [31].

Our key intuition is that, in the plan space of right deep trees without cross products, the cost of plans of a star query with PKFK joins can be the same with different join orders of dimension tables. This is because all the bitvector filters for a star query will be pushed down to the fact table; and by Lemma 1, we can show the cost of many join orders is the same. Figure 4 shows an example of two plans of a star query with PKFK joins using different join orders of dimension tables but having the same cost.

Formally, **our key results** in this section are:

THEOREM 4.1. ***Minimal cost right deep trees for star query:*** *Let $\mathcal{R}$ be the set of relations of a star query as defined in Definition 1. Let $\mathcal{A} = \{T(X_0, \cdots, X_n)\}$ be the set of right deep trees without cross products for q, where $X_0, \cdots, X_n$ is a permutation of $R_0, \cdots, R_n$. If $C_{min} = min\{C_{out}(\mathcal{T}), \mathcal{T} \in \mathcal{A}\}$, then there exists a plan $\mathcal{T} \in \mathcal{A}_{candidates} = \{T(R_0, R_1, \cdots, R_n)\} \cup \{T(R_k, R_0, R_1, \cdots, R_{k-1}, R_{k+1}, \cdots, R_n), 1 \leq k \leq n\}$ such that $C_{out}(\mathcal{T}) = C_{min}$.*

THEOREM 4.2. ***Plan space complexity for star query:*** *Let $\mathcal{R}$ be the set of $n+1$ relations of a star query as defined in Definition 1. We can find the query plan with the minimal cost in the place space of right deep trees without cross products from $n+1$ candidate plans.*

We omit most of the proofs due to space limit, and they can be found in our technical report [14].

We start the analysis by understanding the plan space of right deep trees without cross products for star queries:

LEMMA 2. ***Right deep trees for star query:*** *Let $\mathcal{R}$ be the set of relations of a star query as defined in Definition 1. Let $\mathcal{T} = T(X_0, X_1, X_2, \cdots, X_n)$ be a query plan, where $X_0, \cdots, X_n$ is a permutation of $\{R_0, R_1, R_2, \cdots, R_n\}$. Then $\mathcal{T}$ is a right deep tree without cross products if and only if $X_0 = R_0$ or $X_1 = R_0$.*

By Lemma 2, we divide the plans into two cases: whether R_0 is the right most leaf or not.

We first generalize Lemma 1 to multiple relations:

LEMMA 3. ***Star query absorption rule****: Let $\mathcal{R}$ be a star query as defined in Definition 1, then $R_0/(R_1, R_2, \cdots, R_n) \supseteq \prod_{R_0}(R_0 \bowtie R_1 \bowtie \cdots \bowtie R_n)$ and $|R_0/(R_1, R_2, \cdots, R_n)| \geq |R_0 \bowtie R_1 \bowtie \cdots \bowtie R_n|$. The equality happens when the bitvector filters created from $(R_1, R_2, \cdots, R_n)$ has no false positives.*

We now show that, all the plans in this plan space where the right most leaf is R_0 has the same cost C_{out} if bitvector filters have no false positives. Formally,

LEMMA 4. ***Minimal cost right deep tree for star query with right most leaf*** R_0***:*** *Let $\mathcal{R}$ be the set of relations of a star query as defined in Definition 1. The cost of the right deep tree $C_{out}(T(R_0, X_1, X_2, \cdots, X_n))$ is the same for every permutation $X_1, X_2, \cdots, X_n$ of $R_1, R_2, \cdots, R_n$.*

PROOF. Because $R_1, R_2, \cdots, R_n$ only connects to R_0, and R_0 is the right most leaf, based on Algorithm 1, all the bitvector filters created from $R_1, R_2, \cdots, R_n$ will be pushed down to R_0. Thus, $C_{out}(X_k) = |X_k|$ for $1 \leq k \leq n$ and $C_{out}(R_0) = |R_0/(X_1, X_2, \cdots, X_n)|$. By Lemma 3, $C_{out}(R_0) = |R_0/(R_1, R_2, \cdots, R_n)|$.

Now consider the intermediate join result for $S(R_0, X_1, \cdots, X_k)$, where $1 \leq k \leq n$. By Lemma 3, $|S(R_0, X_1, \cdots, X_k)| = |S(R_0/(R_1, \cdots, R_n), X_1, \cdots, X_k)| = |S(R_0, R_1, \cdots, R_n)|$.

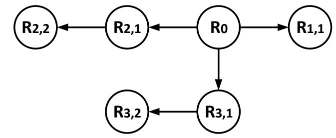


Figure 5: Snowflake query with PKFK joins, where the fact table is R_0 and the branches are $\{R_{1,1}\}, \{R_{2,1}, R_{2,2}\}, \{R_{3,1}, R_{3,2}\}$

Thus, $C_{out}(S(R_0, X_1, \cdots, X_k)) = C_{out}(S(R_0, R_1, \cdots, R_n))$ for all $1 \leq k \leq n$.

Since the total cost of the plan is $C_{out}(T(R_0, X_1, \cdots, X_{n-1})) = \sum_{i=1}^{n} |R_i| + n \cdot |S(R_0, R_1, \cdots, R_0)|$, every permutation $X_1, \cdots, X_n$ of $R_1, \cdots, R_n$ has the same cost. □

Now consider the other case where R_0 is not the right most leaf, and $X_1 = R_0$. Let $X_1 = R_k, 1 \leq k \leq n$, similarly, we show that the cost of the plans in the form of $T(R_k, R_0, X_1, X_2, \cdots, X_{n-1})$ is the same for every permutation of $R_1, R_2, \cdots, R_{k-1}, R_{k+1}, \cdots, R_n$ if bitvector filters have no false positives. Formally,

LEMMA 5. ***Minimal cost right deep tree for star query with right most leaf*** R_k***:*** *Let $\mathcal{R}$ be the set of relations of a star query as defined in Definition 1. The cost of the right deep tree $C_{out}(T(R_k, R_0, X_1, X_2, \cdots, X_{n-1})$ is the same for every permutation $X_1, X_2, \cdots, X_{n-1}$ of $R_2, R_3, \cdots, R_{k-1}, R_{k+1}, \cdots, R_n$.*

By combining Lemma 4 and Lemma 5, we can prove Theorem 4.1 and Theorem 4.2.

5 ANALYSIS OF SNOWFLAKE QUERIES WITH PKFK JOINS

We define snowflake queries with PKFK joins as below:

DEFINITION 2. ***Snowflake query with PKFK joins****: Let $\mathcal{R} = \{R_0, R_{1,1}, \cdots, R_{1,n_1}, R_{2,1}, \cdots, R_{2,n_2}, \cdots, R_{m,1}, \cdots, R_{m,n_m}\}$ be a set of relations and q be a query joining relations in $\mathcal{R}$. The query q is a snowflake query with PKFK joins if*

- *$R_0 \longrightarrow R_{i,1}$ for $1 \leq i \leq m$ and*
- *$R_{i,j-1} \longrightarrow R_{i,j}$ for $1 \leq i \leq m, 1 < j \leq n_i$.*

We call R_0 the fact table and $R_{i,1}, R_{i,2}, \cdots, R_{i,n_i}$ a branch. We denote the branch $\{R_{i,1}, R_{i,2}, \cdots, R_{i,n_i}\}$ as $\mathcal{R}_i$.

Figure 5 shows an example of a snowflake query, where R_0 is the fact table, and $\{R_{1,1}\}, \{R_{2,1}, R_{2,2}\}, \{R_{3,1}, R_{3,2}\}$ are three branches of dimension tables.

Now we analyze the plan space complexity for the snowflake query (Definition 2). We will show that, in the plan space of right deep trees without cross products, we can find the query plan of the minimal cost (under the cost function from Section 3.3) from $n+1$ query plans with bitvector filters if the bitvector filters have no false positives, where $n+1$ is the number of relations in the snowflake query. In contrast, the original plan space complexity for snowflake queries in this plan space is exponential to n.

We divide the plans into two cases: whether R_0 is the right most leaf or not. We start with the case where R_0 is the right most leaf. Then we analyze a subproblem of the plan space for a branch in a snowflake query. We finally analyze the case where R_0 is not the right most leaf.

Formally, **our key results** in this section are:

THEOREM 5.1. ***Minimal cost right deep trees for snowflake query:*** *Let $\mathcal{R}$ be the set of relations of a snowflake query q as described in Definition 2. Let $C_{min} = min\{C_{out}(T(X_0, X_1, \cdots, X_n))\}$, where $X_0, X_1, \cdots, X_n$ is a permutation of $\mathcal{R}$, and $T(X_1, X_2, \cdots, X_n)$ is a right deep tree without cross products for q. Then there exists a right deep tree $\mathcal{T}' \in \{T(R_{i,a_1}, R_{i,a_2}, \cdots, R_{i,a_{n_i}}, R_0, R_{1,1}, \cdots, R_{1,n_1}, \cdots, R_{i-1,1}, \cdots, R_{i-1,n_{i-1}}, R_{i+1,1}, \cdots, R_{i+1,n_{i+1}} \cdots, R_{n,1}, \cdots, R_{n,n_m})\} \cup \{T(R_0, R_{1,1}, R_{1,2}, \cdots, R_{n,1}, \cdots, R_{n,n_m})\}$, where $a_1, a_2, \cdots, a_{n_i}$ is a permutation of $1, 2, \cdots, n_i$, such that $C_{out}(\mathcal{T}') = C_{min}$.*

THEOREM 5.2. ***Plan space complexity for snowflake query:*** *Let $\mathcal{R}$ be the set of $n + 1$ relations of a snowflake query q as described in Definition 2. We can find the query plan with the minimal cost in the place space of right deep trees without cross products from $n + 1$ candidate plans.*

We omit most of the proofs due to space limit, and they can be found in our technical report [14].

5.1 R_0 is the right most leaf

Let's first look at the right deep trees where R_0 is the right most leaf. **Our key insight** is to extend our analysis on star queries and show that all the trees in this plan space have the same C_{out}.

We define a class of right deep trees where a relation with a PKFK join condition only appears on the right side of the relations it joins with in a snowflake query. Formally,

DEFINITION 3. ***Partially-ordered right deep tree:*** *Let $\mathcal{R}$ be the set of relations of a snowflake query q as described in Definition 2. Let $\mathcal{T} = T(R_0, X_1, \cdots, X_n)$ be a plan for q, where $X_1, \cdots, X_n$ is a permutation of $\mathcal{R} - \{R_0\}$. If for any $X_i, 1 \le i \le n$, either $X_i = R_{p,1}$ or there exists $X_j, 1 \le j < i$ such that $X_j \rightarrow X_i$, we call $\mathcal{T}$ a partially-ordered right deep tree.*

Now we show that the plans in the space of right deep trees without cross products are partially-ordered trees if R_0 is the right most leaf. Formally,

LEMMA 6. ***Right deep tree without cross products for snowflake query:*** *Let $\mathcal{R}$ be the set of relations of a snowflake query q as described in Definition 2. If $\mathcal{T} = T(R_0, X_1, X_2, \cdots, X_n)$ is a right deep tree without cross products for q, then $\mathcal{T}$ is a partially-ordered right deep tree.*

PROOF. If $\mathcal{T}$ is not partially ordered, then there exists X_i such that $X_i \notin \{R_{1,1}, R_{2,1}, \cdots, R_{n,1}\}$ and there does not exist $X_j, i < j \le n$ such that $X_j \rightarrow X_i$. Then X_i does not join with $R_0, X_n, X_{n-1}, \cdots, X_{i+1}$. So there exists a cross product. □

Now we show all the partially-ordered right deep trees have the same cost if R_0 is the right most leaf.

Follow Lemma 6 and Algorithm 1, we have

LEMMA 7. ***Bitvector filters in partially-ordered right deep tree:*** *Let $\mathcal{R}$ be the set of relations of a snowflake query q as described in Definition 2. If $\mathcal{T} = T(R_0, X_1, X_2, \cdots, X_n)$ is a right deep tree without cross products for q, then the bitvector filter created from $R_{i,j}$ will be pushed down to $R_{i,j-1}$ if $j > 1$ or R_0 if $j = 1$.*

Follow Lemma 7, we have

LEMMA 8. ***Equal cost for partially-ordered right deep tree:*** *Let $\mathcal{R}$ be the set of relations of a snowflake query q as described in Definition 2. Let $\mathcal{T} = T(R_0, X_1, X_2, \cdots, X_n)$ and $\mathcal{T}' = T(R_0, Y_1, Y_2, \cdots, Y_n)$ be two partially ordered right deep trees of q. Then $C_{out}(\mathcal{T}) = C_{out}(\mathcal{T}')$.*

5.2 Branch of a snowflake query

Before diving into the case where R_0 is not the right most leaf, we first analyze a subproblem of a branch in a snowflake query in the plan space of right deep trees without cross products. We show that the plan space complexity is linear in the number of relations in the branch. Formally, we define a branch as the following:

DEFINITION 4. ***Branch of a snowflake query:*** *Let $\mathcal{R} = \{R_0, R_1, \cdots, R_n\}$ be a set of relations and q be a query joining relations in $\mathcal{R}$. The query q is a branch if $R_{k-1} \rightarrow R_k$ for all $1 \le k \le n$.*

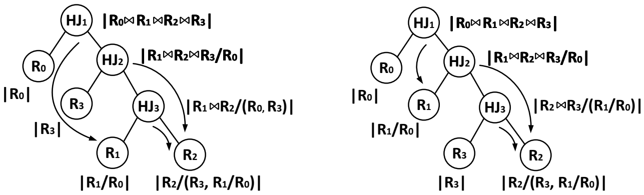
Figure 5 shows an example of a snowflake query with three branches.

We show that, in the plan space of right deep trees without cross products, we can find the query plan with minimal C_{out} from $n + 1$ plans with bitvector filters if the bitvector filters have no false positives, where $n+1$ is the number of relations in the query. In contrast, the original plan space complexity for a branch is n^2 [31].

Our key insight is that, for a plan with right most leaf R_k, where $1 \le k \le n$, if the plan has minimal cost, it must join $R_k, R_{k+1}, \cdots, R_n$ consecutively in its right subtree. Otherwise, we can reduce the plan cost by altering the join order and 'pushing down' the relations $R_n, R_{n-1}, \cdots, R_{m+1}$ into the right subtree. Figure 6 shows an example of how the plan cost can be reduced by 'pushing down' the relations.

Formally, **our key results** in this subsection are:

THEOREM 5.3. ***Minimal cost right deep trees for a branch:*** *Let $\mathcal{R}$ be the set of relations of a branch as described in Definition 4. Let $\mathcal{A} = \{T(X_0, X_1, \cdots, X_n)\}$ be the*



(a) Plan P_1 **(b) Plan P_2**

Figure 6: Example of two plans for a branch $\{R_0, R_1, R_2, R_3\}$ of a snowflake query with PKFK joins using bitvector filters. Each operator is annotated with the intermediate result size. Plan P_1 does not join R_2 and R_3 consecutively in its right subtree. Pushing down R_3 to join with R_2 consecutively results in plan P_2 with reduced cost.

set of right deep trees without cross products for q, where $X_0, X_1, \cdots, X_n$ is a permutation of $R_0, R_1, \cdots, R_n$. If $C_{min} = min\{C_{out}(T(X_0, X_1, \cdots, X_n))\}$, then there exists a plan $\mathcal{T} \in \mathcal{A}_{candidates} = \{T(R_n, R_{n-1}, \cdots, R_0)\} \cup \{T(R_k, R_{k+1}, \cdots, R_n, R_{k-1}, R_{k-2}, \cdots, R_0), 0 \leq k \leq n-1\}$ such that $C_{out}(\mathcal{T}) = C_{min}$.

Theorem 5.4. ***Plan space complexity for a branch:*** *Let $\mathcal{R}$ be the set of $n+1$ relations of a branch as described in Definition 4. We can find the query plan with the minimal cost in the place space of right deep trees without cross products from $n+1$ candidate plans.*

Consider the query plan for a branch $\{R_0, R_1, \cdots, R_n\}$ of the snowflake in the plan space of right deep trees without cross products. Let's first look at the query plans where R_n is the right most leaf. Formally,

Lemma 9. *Let $\mathcal{R}$ be the set of relations of a branch as described in Definition 4. There exists only one right deep tree without cross products such that R_n is the right most leaf, that is, $T(R_n, R_{n-1}, \cdots, R_0)$.*

This can be derived from the join graph of a branch.

Now we look at the query plans where R_n is not the right most leaf. Let $T(X_0, \cdots, X_n)$ be a right deep tree without cross products where $X_0, \cdots, X_n$ is a permutation of $R_0, \cdots, R_n$. We show that, without joining $R_n, R_{n-1}, \cdots, R_k$ consecutively, a plan cannot have the minimal cost. Formally,

Lemma 10. ***Cost reduction by pushing down R_n:*** *Let $\mathcal{R}$ be the set of relations of a branch as described in Definition 4. Let $\mathcal{T} = T(X_0, X_1, \cdots, X_n)$ be a right deep tree without cross products for $R_0, R_1, \cdots, R_n$. Assume $X_k = R_n$ for some $1 \leq k \leq n$. If $X_{k-1} \neq R_{n-1}$, then $\mathcal{T}' = T(X_0, X_1, \cdots, X_k, X_{k-1}, X_{k+1}, X_{k+2}, \cdots, X_n)$ is a right deep tree without cross products and $C_{cout}(\mathcal{T}') \leq C_{out}(\mathcal{T})$.*

Lemma 11. ***Cost reduction by pushing down $R_n, R_{n-1}, \cdots, R_{n-m}$:*** *Let $\mathcal{R}$ be the set of relations of a branch as described in Definition 4. Let $\mathcal{T} = T(X_0, X_1, \cdots, X_n)$ be a right deep tree without cross products for $R_0, R_1, \cdots, R_n$. Let $X_k = R_n, X_{k-1} = R_{n-1}, \cdots, X_{k-m} = R_{n-m}$ for some $m \leq k \leq n$. If $X_{k-m-1} \neq R_{n-m-1}$, then $\mathcal{T}' = T(X_0, X_1, \cdots, X_{k-m-2}, X_{k-m}, X_{k-m+1}, \cdots, X_k, X_{k-m-1}, X_{k+1}, \cdots, X_n)$ is a right deep tree without cross products and $C_{cout}(\mathcal{T}') \leq C_{out}(\mathcal{T})$.*

The proofs can be found in our technical report [14].

By combining Lemma 9 and Lemma 11, we can prove Theorem 5.3 and Theorem 5.4.

5.3 R_0 is not the right most leaf

Now let's look at the right deep trees where R_0 is not the right most leaf for a snowflake query with PKFK joins.

We first show that the relations appear on the left side of R_0 can only come from a single branch given the join graph of a snowflake query. Formally,

Lemma 12. ***Single branch in right most leaves:*** *Let $\mathcal{R}$ be the set of relations of a snowflake query q as described in Definition 2. Let $\mathcal{T} = T(X_0, X_1, \cdots, X_n)$ be a right deep tree without cross products for q, where $X_0, X_1, \cdots, X_n$ is a permutation of $\mathcal{R}$. If $X_k = R_0$, then $X_0, X_1, \cdots, X_{k-1}$ is a permutation of $R_{i,1}, R_{i,2}, \cdots, R_{i,k}$ for some $1 \leq i \leq m$.*

Now we show that the relations on the left side of R_0 are partially ordered. Formally,

Lemma 13. ***Partially-ordered subtree:*** *Let $\mathcal{R}$ be the set of relations of a snowflake query q as described in Definition 2. Let $\mathcal{T} = T(X_0, X_1, \cdots, X_n)$ be a right deep tree without cross products for q, where $X_0, X_1, \cdots, X_n$ is a permutation of $\mathcal{R}$. If $X_k = R_0$, then $X_{k+1}, X_{k+2}, \cdots, X_n$ is a partially ordered right deep tree of the new relation $R_0' = X_0 \bowtie X_1 \bowtie \cdots \bowtie X_k$.*

Now we show that if a subset of relations of a single branch $\mathcal{R}_i$ is on the right side of R_0, there exists a query plan with lower cost where all the relations in $\mathcal{R}_i$ are on the right side of R_0. Formally,

Lemma 14. ***Cost reduction by consolidating a single branch:*** *Let $\mathcal{R}$ be the set of relations of a snowflake query q as described in Definition 2. Let $\mathcal{T} = T(X_0, X_1, \cdots, X_{k-1}, R_0, X_{k+1}, \cdots, X_n)$ be a right deep tree without cross products for q, where $X_0, X_1, \cdots, X_{k-1}$ is a permutation of $R_{i,1}, R_{i,2}, \cdots, R_{i,k}$ for some $1 \leq i \leq m, 1 \leq k \leq n_i - 1$. Then there exists a right deep tree without cross products $\mathcal{T}' = T(X_0, X_1, \cdots, X_{k-1}, R_{i,k+1}, R_{i,k+2}, \cdots, R_{i,n_i}, R_0, Y_1, Y_2, \cdots, Y_{n-n_i-1})$ for q such that $C_{out}(\mathcal{T}') \leq C_{out}(\mathcal{T})$.*

By combining Lemma 8 and Lemma 14, we can prove Theorem 5.1, and Theorem 5.2 directly follows from Theorem 5.4 and Theorem 5.1.

6 BITVECTOR-AWARE QO FOR GENERAL SNOWFLAKE QUERIES

While star and snowflake queries with PKFK joins are important patterns in decision support queries, in practice, such queries can have more complicated join graphs. For example, a decision support query can join multiple fact tables, where the joins may not be PKFK joins. In addition, there can be join conditions between the dimension tables or branches, where the bitvector filters created from the dimension tables may not be pushed down to the fact table. Finally, there can be dimension tables or branches that are larger than the fact table after predicate filters, where the fact table should be on the build side in the plan space of right deep trees.

In this section, we first propose an algorithm to extend bitvector-aware query optimization to an arbitrary snowflake query with a single fact table. We then generalize it to arbitrary decision support queries with multiple fact tables. Our algorithm applies to queries with arbitrary join graphs. We further optimize our algorithm with cost-based bitvector filters. We also discuss options to integrate our algorithm into a Volcano / Cascades query optimizer.

6.1 Queries with a single fact table

We propose an algorithm (Algorithm 2) with simple heuristics to construct the join order for an arbitrary snowflake query with a single fact table. The key insight is to leverage the candidate plans of minimal cost analyzed in Section 5. Algorithm 2 shows how to construct the join order for a decision support query with a single fact table.

We first assign priorities to the branches based on their violations of the snowflake pattern as defined in Definition 2. We then sort the branches in descending order by their priorities (line 1). Intuitively, if the bitvector filters created from dimension tables are all pushed down to the fact table except for one, where the corresponding dimension table either joins with another dimension table or is not on the build side. Since this dimension table does not create a bitvector filter that is pushed down to the fact table, joining this dimension table early with the fact table can eliminate the unnecessary tuples that do not qualify the join condition early in the plan.

Specifically, we assign priorities to branches for snowflake queries with the following heuristics:

- Group P0: Relations that do not have join condition or PKFK joins with the fact table (line 23). This can happen when joining multiple fact tables. As a heuristic, we join these branches by descending selectivity on the fact table (line 23).
- Group P1: Branches that do not join with any other branches and have smaller cardinality than the fact table (line 24). These branches are joined with the fact table before joining the branches in group P0.
- Group P2: Branches joining with other branches (line 21). Such branches should be joined consecutively in the right deep tree to allow pushing down bitvector filters created by these branches. As a heuristic, within a set of connected branches, we join these branches with descending selectivity on the fact table (line 31); across sets of connected branches, we prioritize the sets of larger numbers of connected branches (line 21).
- Group P3: Branches that are larger than the fact table (line 25). Since it is clearly suboptimal to put these branches on the build side, we reorder the build and probe sides for them (line 12-13). Joining these branches early allows pushing down the bitvector filters created from the fact table. As a heuristic, we order the branches in this group with descending selectivity on the fact table (line 31).

Based on the analysis in Section 5, we construct the candidate plans by two cases. If R_0 is the right most leaf, we join all the branches with the fact table (line 2); otherwise, for each branch, we optimize the branch based on the analysis in Section 5.2, join the remaining branches to complete the plan, and update the best plan if the estimated cost of the new plan is lower (line 3-7).

6.2 Queries with multiple fact tables

In addition to snowflakes with a single fact table, complex decision support queries can include multiple fact tables. We further extend our algorithm to arbitrary join graphs by iteratively extracting and optimizing snowflake join graphs.

At a high level, our algorithm produces a join order for a join graph by alternating two stages iteratively as shown in Algorithm 3. In the snowflake extraction stage (line 2), we extract a snowflake subgraph from a join graph by identifying a single fact table and its related dimension tables, potentially with non-PKFK joins. In the snowflake optimization stage (line 3), we use Algorithm 2 to produce a join order for the extracted subgraph. The resulting snowflake will be marked as 'optimized' and considered as a new relation in the updated join graph (line 4-5). Our algorithm alternates the two stages until the full join graph is optimized (line 1).

Specifically, when extracting a snowflake (line 8-19), a relation is considered as a fact table if it does not join with any other table where the join predicate is an equi-join on its key columns. Among all the unoptimized fact tables in G, we find the one with the smallest cardinality and expand from this table recursively to include all related dimension relations (line 4-9). If there is only one fact table in G, we simply return the original join graph (line 11).

6.3 Cost-based Bitvector Filter

In practice, creating and applying bitvector filers has overheads. Consider a hash join with build side R and probe side

```
OptimizeSnowflake(G):
  Input: Join graph G
  Output: Query plan plan
  B ← SortedBranches(G.Branches)
  best → JoinBranches(B, G.Fact, ∅)
  foreach branch b in B do
    p ← Join(OptimizeChain(b, G.Fact), G.Fact)
    p ← JoinBranches(B \ b, G.Fact, p)
    if best.Cost > p.Cost then best ← p
  end
  return best
JoinBranches(B, f, p):
  Input: A set of branches B, fact table f, a plan p
  Output: A query plan p′
  p′ ← p
  foreach branch b in B do
    foreach table t in b do
      if t.Card > f.Card then
        p′ ← Join(p′, t)
      else p′ ← Join(t, p′)
    end
  end
  return p′
SortBranches(G):
  Input: Join graph G
  Output: Sorted branches sortedBranches
  groups ← GroupBranches(G)
  sortedG ← SortBySizeDesc(groups)
  priority ← []
  for i = 0; i < groups.Count(); i + + do
    if sortedG[i].Size > 1 then
      priority[i] ← sortedG[i].Size
    else
      if IsNonUniqueKeyJoin(g[0], f) then
        priority[i] ← 0
      if g[0].Card < f.Card then
        priority[i] ← 1
      else priority[i] ← |G| + 1
    end
  end
  sortedG ← SortByPriorityDesc(groups, priority)
  sortedBranches ← []
  foreach group in sortedG do
    branches ← SortBySelectivityDesc(group)
    foreach b in branches do
      sortedBranches.Add(b)
  end
  return sortedBranches
```

Algorithm 2: Construct a join order for a snowflake query with a single fact table

```
OptimizeJoinGraph(G):
  Input: Join graph G
  Output: Query plan plan
  while |G| > 1 do
    G′ ← ExtractSnowflake(G)
    p ← OptimizeSnowflake(G′)
    G ← UpdateJoinGraph(G, G′)
    plan ← UpdateQueryPlan(plan, p)
  end
  return plan
ExtractSnowflake(G):
  Input: Join graph G
  Output: Snowflake G′
  n ← 0
  G_sorted ← SortByCardinalityAsc(G)
  foreach g in G_sorted do
    if g is an unoptimized fact table then
      if n == 0 then
        G′ ← ExpandSnowflake(g)
      end
      n ← n + 1
    end
  end
  if n == 1 then G′ ← G
  return G′
```

Algorithm 3: Construct a join order for a decision support query with an arbitrary join graph

S. Assume the bitvector filter eliminates λ percent of the tuples from S. The ratio λ can be estimated by the optimizer the same way as an anti-semi join operator, and it can include the estimated false positive rate of the bitvector filter.

Assume the cost of a hash join consists of building the hash table g_b, probing the hash table g_p, and outputing the resulting tuples g_o. Let the cost of creating and applying a bitvector filter be h and f. The cost difference of the hash join with and without using the bitvector filter is

$$Cost_\Delta = g_p(|S|) - g_p(\lambda|S|) - f(|R|) - h(|S|)$$

Assume the cost of probing a tuple is C_p, the cost of checking a tuple against a bitvector filter is C_f, and creating a bitvector filter is relatively cheap, i.e., $f(|R|) << h(|S|)$. Then

$$Cost_\Delta = |S|((1-\lambda)C_p - C_f) - f(|R|) \sim |S|((1-\lambda)C_p - C_f)$$

Using a bitvector filter reduces the cost of a hash join if

$$Cost_\Delta < 0 \sim |S|((1-\lambda)C_p - C_f) < 0 \Leftrightarrow \lambda > 1 - C_f/C_p$$

Let $\lambda_{thresh} = 1 - C_f/C_p$. Note that λ_{thresh} is independent of R and S. We can run a micro-benchmark to profile C_f and C_p and compute λ_{thresh}. When the bitvector filter is pushed down below the root of the probe side, a more detailed analysis is needed to account for the cascading effect of tuple elimination. Empirically, choosing a threshold that is slightly smaller than $1 - C_f / C_p$ works well.

6.4 Integration

Our algorithm can transform a query plan by optimizing the join order with the underlying join graph. Thus, our algorithm can be used as a new transformation rule in a Volcano / Cascades query optimization framework upon detecting a snowflake join (sub)graph. There are three integration options depending on how the underlying optimizer accounts for the impact of bitvector filters:

- *Full integration:* When applying join order transformation to a (sub)plan, the placement of bitvector filters and their selectivity can change. If the underlying Volcano / Cascades query optimization framework can correctly account for the placement and the selectivity of bitvector filters *during* query optimization, the new transformation rule can be transparently integrated into the query optimizer the same way as any existing transformation rule.
- *Alternative-plan integration:* If the query optimizer can account for the placement and the selectivity of bitvector filters in a *final plan after* query optimization, the new transformation rule can be used to produce an alternative plan. The optimizer can then choose the plan with the cheaper estimated cost from the alternative plan and the plan produced by the original query optimization.
- *Shallow integration:* We mark a (sub)plan after it is transformed by our new transformation rule. The underlying query optimization framework works as usual, except additional join reordering on marked (sub)plans is disabled.

7 EVALUATION

7.1 Implementation

We implement Algorithm 3 in a commercial database DBMS-X as a transformation rule. DBMS-X has a cost-based, Volcano / Cascades style query optimizer. Starting from an initial query plan, the optimizer detects various patterns in the plan and fires the corresponding transformation rules. Due to the importance of decision support queries, DBMS-X has implemented heuristics to detect snowflake patterns and transform the corresponding subplans.

We leverage the snowflake detection in DBMS-X and transform the corresponding subplan as described in Algorithm 3. We implement a shallow integration (Section 6.4), where join reordering is disabled on the transformed subplan. The subplan is subject to other transformations in DBMS-X. We use the original cardinality estimator and cost modeling in DBMS-X, and the selectivity of a bitvector filter is estimated the same way as the existing semi-join operator. We implement the cost-based bitvector filter as described in Section 6.3,

Table 3: Statistics of workloads, including database size, the number of tables, queries, indexes (B+ trees and columnstores), and joins.

Statistics	TPC-DS	JOB	CUSTOMER
DB Size	100GB	7GB	700GB
Tables	25	21	475
Queries	99	113	100
B+ trees / columnstores	0 / 20	44 / 20	680 / 0
Joins avg / max	7.9 / 48	7.7 / 16	30.3 / 80

and we will discuss how we profile the elimination threshold λ_{thresh} in Section 7.3. The final plan is chosen with the existing cost-based query optimization framework.

7.2 Experimental Setup

Workload. We evaluation our technique on three workloads: TPC-DS [1] 100GB with columnstores, JOB [25] with columnstores, primary key indexes, and foreign key indexes, and a customer workload (CUSTOMER) with B+-tree indexes. Table 3 summarizes the statistics of our workloads. In particular, CUSTOMER has the highest number of average joins per query, and JOB has the most complex join graphs, including joining multiple fact tables, large dimension tables, and joins between dimension tables. Our workloads also cover the range of different physical configurations, with B+ trees (CUSTOMER), columnstores (TPC-DS), or both (JOB).

Baseline. We use the query plans produced by the original DBMS-X as our baseline. Bitvector filters are widely used in the query plans of DBMS-X. As shown in Appendix A, 97% queries in JOB, 98% queries in TPC-DS, and 100% queries in CUSTOMER have bitvector filters in their original plans. A bitvector filter can be created from a hash join operator, and it is pushed down to the lowest level on the probe side as described in Algorithm 1. The query optimizer in DBMS-X uses heuristics to selectively add bitvector filters to the query plan without fully accounting for the impact of bitvector filters during the query optimization stage. In particular, the heuristics used in its snowflake transformation rules neglect the impact of bitvector filters. We use a generous timeout for the query optimizer in DBMS-X so that it can explore a large fraction of the relevant plan search.

Overhead. Our technique adds very low overhead to query optimization. In fact, since we disable join reordering on the snowflake subplan after it is optimized by our transformation rule, the query optimization time with our transformation rule is one third of that with the original DBMS-X in average. We also measure the memory consumption for query execution. We observe some increase in memory consumption with our technique, since it favors right deep trees. The overall increase in memory consumption is not significant.

Environment. All the experiments are run on a machine with Intel Xeon CPU E5 - 2660 v3 2.6GHz, 192GB memory, a 6.5TB hard disk, and Windows Server 2012 R2. To reduce runtime variance, all the queries are running in isolation at the same level of parallelism. The query CPU time reported is an average over ten warm runs.

7.3 Overhead of bitvector filters

As discussed in Section 6.3, we can choose a tuple elimination threshold to selectively create bitvector filters. We profile the overhead of bitvector filters with a micro-benchmark by running the following query in TPC-DS:

```
SELECT COUNT(*)
FROM store_sales, customer
WHERE ss_customer_sk = c_customer_sk
AND c_customer_sk % 1000 < @P
```

The query plan joins *customer* and *store_sales* with a hash join. A bitvector filter is created from *customer* on the build side and pushed down to *store_sales* on the probe side, where tuples are eliminated before the join. We control the selectivity of the bitvector filter with the parameter @P.

Figure 7 shows the CPU time of execution of the query varying its selectivity with and without bitvector filtering, normalized by the same constant. We further break down the CPU time by the hash join operator, the probe side, and the build side. Since the CPU time for reading *customer* is very small, we omit it in Figure 7 for readability.

With selectivity 1, no tuples are eliminated by the bitvector. With bitvector filtering, the hash join operator is slightly more expensive due to creating the bitvector filter, and the probe side operator has higher execution CPU due to the overhead of checking the tuples from *store_sales* against the bitvector filter. As the selectivity increases, the bitvector filter eliminates more tuples from the probe side and the execution cost of the hash join operator reduces. The plan with bitvector filtering becomes cheaper than the other plan once the bitvector filter eliminates more than 10% of the tuples. The cost reduction can be even more with queries of multiple joins. Empirically, we find 5% to be a good threshold, and we set λ_{thresh} to 5% in our implementation.

In Appendix A, we further evaluate the effectiveness and applicability of bitvector filters as a query processing technique. As shown in Table 4, DBMS-X uses bitvector filters for 97% – 100% queries in the benchmarks, with 10% – 80% workload-level execution CPU cost reduction. This confirms that bitvector filters is a widely applicable query processing technique, and thus bitvector-aware query optimization can potentially impact a wide range of queries.

7.4 Evaluation on bitvector-aware query optimization

Figure 8 shows the total amount of CPU execution time reduction with our technique. We sum up the total CPU execution time of the plans produced by DBMS-X with our technique and divide it by that of the plans produced by the original DBMS-X. On average, the total workload execution CPU time has been reduced by 37%. We observe that workloads with more complicated decision support queries benefit more from our technique, with the highest reduction of 64% in CPU execution time for JOB. Since DBMS-X has been heavily tuned to optimize for these benchmarks, the degree of reductions in CPU execution time is very significant.

We break down the CPU execution cost by query types. We divide the queries into three groups based on their selectivity, i.e., high (*S*), moderate (*M*), low (*L*). We approximate the query selectivity by the execution CPU cost of the original query plans, with the cheapest 33.3% queries in group *S*, the 33.3% most expensive queries in group *L*, and the rest in group *M*. We showed that, our technique is especially effective in reducing CPU execution cost for expensive queries or queries with low selectivity, i.e., with execution CPU reduced by 4.8× for expensive queries in JOB benchmark. This is because that right deep trees is a preferable plan space for queries with low selectivities ([12, 17]), and our technique produces a better join order for right deep trees.

Figure 9 shows the total number of tuples output by operators in the query plans produced by the original query optimizer (*Original*) and the bitvector-aware query optimizer (*BQO*), normalized by the total number of tuples output by the original query plans in each workload. We sum up the number of tuples by the type of operators, including leaf operators, join operators, and other operators. Figure 9 sheds some insight on the amount of logical work done by operators and thus the quality of query plans. With *BQO*, both the number of tuples processed by join operators as well as leaf operators reduces. In particular, for JOB benchmark, *BQO* reduces the normalized number of tuples output by join operators from 0.50 to 0.24, i.e., a 52% reduction. This again confirms that *BQO* improves query plan quality by producing a better join order.

Figure 10 shows the normalized CPU execution time for individual queries with the plans using our technique and these from the original DBMS-X. The queries are sorted by the CPU execution time of their original query plans, and the top 60 most expensive queries are shown for readability. Note that the Y axis uses a logarithmic scale. We observe a reduction of up to two orders of magnitude in CPU execution time for individual queries. Again, Figure 10 confirms that our technique is especially effective in reducing the CPU execution time for expensive decision support queries.

Figure 7: Profile bitvector filters. A bitvector filter reduces overall cost if it eliminates > 10% tuples

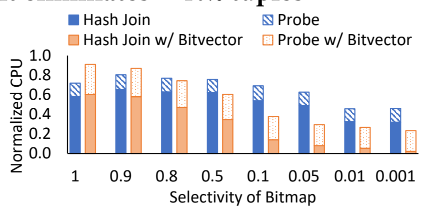


Figure 8: Total query execution CPU time for a workload, breaking down by query selectivity

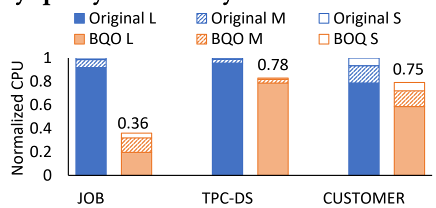


Figure 9: Total number of tuples output by operators in a workload, breaking down by operator types

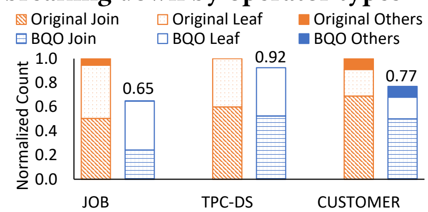
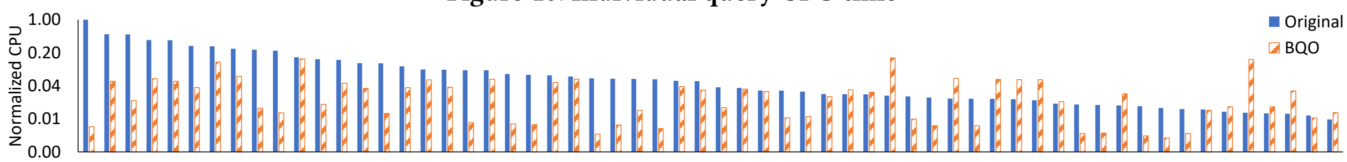
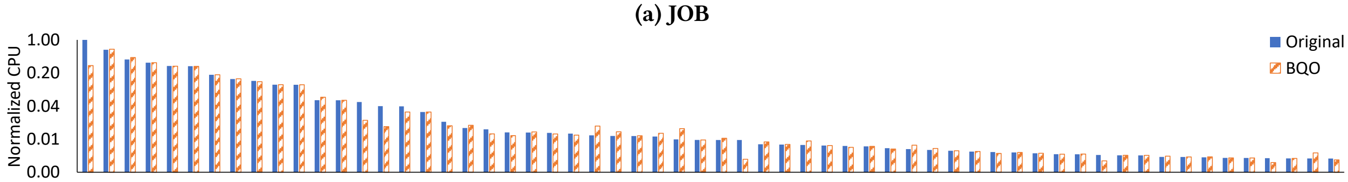


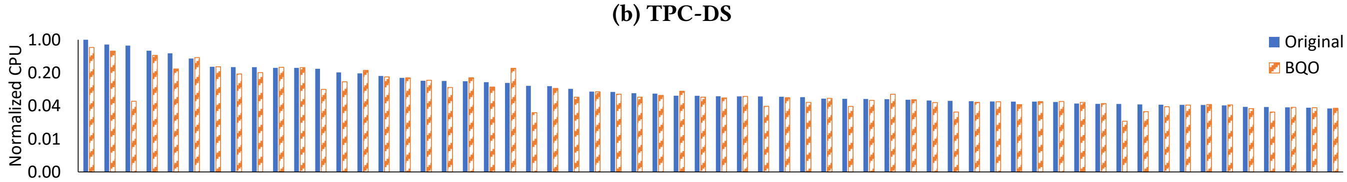
Figure 10: Individual query CPU time



(a) JOB



(b) TPC-DS



(c) CUSTOMER

Our technique can improve plan quality for two reasons. First, if a query optimizer does not fully integrate bitvector filters into query optimization, it can consider the best plan with bitvector filters as 'sub-optimal' as shown in Figure 2. Second, due to the importance of decision support queries, many commercial DBMSs have developed dedicated heuristics to identify and optimize snowflake queries [3, 17, 37]. If these heuristics do not consider the impact of bitvector filters, they can explore a different plan space which does not even contain the plans considered by our technique.

Inevitably, there are regressions compared with the original plans. We investigate such regressions and discover three major reasons. First, our cost function C_{out} does not capture the physical information of operators and can be inaccurate. Second, our technique favors right deep trees, which can become suboptimal when the query is highly selective. Finally, our algorithm uses heuristics to extend to complex decision support queries, which can be suboptimal in some cases.

8 RELATED WORK

We discuss two lines of related work: plan search and bitvector filters.

Plan search. Many query optimization (QO) frameworks in DBMSs are based on either top-down [19, 20, 35] or bottom-up [4] dynamic programming (DP). There has been a large body of prior work on join ordering and plan space complexity analysis with such QO frameworks [16, 26, 27, 29, 31].

Due to the importance of decision support queries, many commercial DBMSs have developed dedicated heuristics for optimizing complex decision support queries [3, 17, 37] based on the plan space of snowflake queries [22].

In this work, we adapt the cost function used in analyzing join order enumeration [28, 30] for our analysis. We analyze the space of right deep trees without cross products, which has been shown to be a favorable plan space for decision support queries and bitvector filters [12, 17, 38].

Bitvector filter and its variants. Semi-join is first introduced to reduce communication cost of distributed queries [6]. Efficient implementation of semi-joins have been heavily studied in the past [8, 18, 36]. Several prior work has explored different schedules of bitvector filters for various types of query plan trees [10–12]. Sideways information passing and magic sets transformation generalize the concept of bitvector filters and combines them with query rewriting [5, 33].

Many variants of bitvector filters have also been studied in the past, such as Bloom filters [7], bitvector indexes [9], cuckoo filters [15], performance-optimal filters [24] and others [2, 32]. The focus of this line of research is on the trade-off between space and accuracy, the efficiency of filter operations, and the extensions of Bloom filter.

Due to the effectiveness of bitvector filters in reducing query execution cost, several commercial DBMSs have implemented bitvector filter or its variants as query processing techniques for decision support queries [13, 17, 21, 23].

In this work, our analysis is based on the classic bitvector filter algorithm described in [18]. We mainly study the interaction between bitvector filters and query optimization, which is orthogonal to the prior work on bitvector filters as query processing techniques.

Lookahead Information Passing (LIP) [38] is the closest prior work to our work. LIP studies the star schema where Bloom filters created from dimension tables are all applied to the fact table. The focus is on the order of applying Bloom filters, and they observe such query plans are robust with different permutations of dimension tables. Compared with LIP, our work systematically analyzes a much broader range of decision support queries and plan search space. Their conclusion on plan robustness can be derived from our analysis.

9 CONCLUSION

In this work, we systematically analyze the impact of bitvector filters on query optimization. Based on our analysis, we propose an algorithm to optimize the join order for arbitrary decision support queries. Our evaluation shows that, instead of using bitvector filters only as query processing techniques, there is great potential to improve query plan quality by integrating bitvector filters into query optimization for commercial databases.

This work is the first step to understand the interaction between bitvector filters and query optimization, and it opens new opportunities for query optimization with many open challenges. Extending the analysis to additional plan space, query patterns, operators beyond hash joins, and more complex cost modeling is challenging. Efficient full integration of bitvector filters for commercial databases with various architectures remains an open problem. Since our analysis shows that bitvector filters result in more robust query plans, which is also observed in [38], understanding how bitvector filters impact robust and interleaved query optimization is also an interesting direction.

Table 4: Query plans with and without bitvector filters

Workload	CPU ratio	Ratio of queries w/ bitvector filters	Improved queries	Regressed queries
JOB	0.20	0.97	0.58	0.00
TPC-DS	0.53	0.98	0.88	0.00
CUSTOMER	0.90	1.00	0.42	0.00

A ADDITIONAL EVALUATION

We evaluate the effectiveness of bitvector filters by executing the same query plan with and without bitvector filtering. We use the original DBMS-X to produce a query plan p with bitvector filters. DBMS-X provides an option to ignore bitvector filters during query processing. For comparison, we execute the same plan p with bitvector filters ignored.

Table 4 shows the performance of the plans with and without bitvector filters for the three benchmarks. At a workload level, using bitvector filters reduces the execution CPU cost by 10% – 80% (*CPU ratio*). In addition, for 97% – 100% of the queries (*Ratio of queries w/ bitvectof filters*), the original query plan uses bitvector filters. At an individual query level, 48% – 88% of the queries has CPU execution cost reduced by more than 20% (*Improved queries*), with no regression on CPU execution cost by more than 20% (*Regressed queries*).

This confirms that bitvector filtering is a widely applicable query processing technique, and thus bitvector-aware query optimization can potentially impact a wide range of queries.

REFERENCES

[1] 2012. *TPC-DS*. http://www.tpc.org/tpcds/

[2] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable bloom filters. *Inform. Process. Lett.* 101, 6 (2007), 255–261.

[3] Lyublena Antova, Amr El-Helw, Mohamed A. Soliman, Zhongxian Gu, Michalis Petropoulos, and Florian Waas. 2014. Optimizing Queries over Partitioned Tables in MPP Systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. Association for Computing Machinery, New York, NY, USA, 373–384. https://doi.org/10.1145/2588555.2595640

[4] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, and et al. 1976. System R: Relational Approach to Database Management. *ACM Trans. Database Syst.* 1, 2, 97–137. https://doi.org/10.1145/320455.320457

[5] Catriel Beeri and Raghu Ramakrishnan. 1991. On the power of magic. *The journal of logic programming* 10, 3-4, 255–299.

[6] Philip A. Bernstein and Dah-Ming W. Chiu. 1981. Using Semi-Joins to Solve Relational Queries. *J. ACM* 28, 1, 25–40. https://doi.org/10.1145/322234.322238

[7] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7, 422–426. https://doi.org/10.1145/362686.362692

[8] Kjell Bratbergsengen. 1984. Hashing Methods and Relational Algebra Operations. In *Tenth International Conference on Very Large Data Bases, August 27-31, 1984, Singapore, Proceedings*. Morgan Kaufmann, 323–333.

[9] Chee-Yong Chan and Yannis E. Ioannidis. 1998. Bitmap Index Design and Evaluation. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*. Association for Computing Machinery, New York, NY, USA, 355–366. https://doi.org/10.1145/276304.276336

[10] M. . Chen and P. S. Yu. 1992. Interleaving a join sequence with semijoins in distributed query processing. *IEEE Transactions on Parallel and Distributed Systems* 3, 5, 611–621. https://doi.org/10.1109/71.159044

[11] Ming-Syan Chen, Hui-I Hsiao, and Philip S. Yu. 1993. Applying Hash Filters to Improving the Execution of Bushy Trees. In *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*. Morgan Kaufmann, 505–516.

[12] Ming-Syan Chen, Hui-I Hsiao, and Philip S. Yu. 1997. On Applying Hash Filters to Improving the Execution of Multi-Join Queries. *VLDB J.* 6, 2, 121–131. https://doi.org/10.1007/s007780050036

[13] Dinesh Das, Jiaqi Yan, Mohamed Zaït, Satyanarayana R. Valluri, Nirav Vyas, Ramarajan Krishnamachari, Prashant Gaharwar, Jesse Kamp, and Niloy Mukherjee. 2015. Query Optimization in Oracle 12c Database In-Memory. *PVLDB* 8, 12, 1770–1781. https://doi.org/10.14778/2824032.2824074

[14] Bailu Ding, Surajit Chaudhuri, and Vivek Narasayya. 2020. Bitvector-aware query optimization for decision support queries (extended version), MSR-TR-2020-8.

[15] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT'14)*. Association for Computing Machinery, New York, NY, USA, 75–88. https://doi.org/10.1145/2674005.2674994

[16] P. Fender, G. Moerkotte, T. Neumann, and V. Leis. 2012. Effective and Robust Pruning for Top-Down Join Enumeration Algorithms. In *2012 IEEE 28th International Conference on Data Engineering*. 414–425. https://doi.org/10.1109/ICDE.2012.27

[17] C. A. Galindo-Legaria, T. Grabs, S. Gukal, S. Herbert, A. Surna, S. Wang, W. Yu, P. Zabback, and S. Zhang. 2008. Optimizing Star Join Queries for Data Warehousing in Microsoft SQL Server. In *2008 IEEE 24th International Conference on Data Engineering*. 1190–1199. https://doi.org/10.1109/ICDE.2008.4497528

[18] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2, 73–169. https://doi.org/10.1145/152610.152611

[19] Goetz Graefe. 1995. The Cascades framework for query optimization. *IEEE Data Eng. Bull.* 18, 3, 19–29.

[20] G. Graefe and W. J. McKenna. 1993. The Volcano optimizer generator: extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*. 209–218. https://doi.org/10.1109/ICDE.1993.344061

[21] Hui-I Hsiao, Ming-Syan Chen, and Philip S. Yu. 1994. On Parallel Execution of Multiple Pipelined Hash Joins. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD'94)*. Association for Computing Machinery, New York, NY, USA, 185–196. https://doi.org/10.1145/191839.191879

[22] Nikos Karayannidis, Aris Tsois, Timos K. Sellis, Roland Pieringer, Volker Markl, Frank Ramsak, Robert Fenk, Klaus Elhardt, and Rudolf Bayer. 2002. Processing Star Queries on Hierarchically-Clustered Fact Tables. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*. Morgan Kaufmann, 730–741. https://doi.org/10.1016/B978-155860869-6/50070-6

[23] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T. Lee, J. Loaiza, N. Macnaughton, V. Marwah, N. Mukherjee, A. Mullick, S. Muthulingam, V. Raja, M. Roth, E. Soylemez, and M. Zait. 2015. Oracle Database In-Memory: A dual format in-memory database. In *2015 IEEE 31st International Conference on Data Engineering*. 1253–1258. https://doi.org/10.1109/ICDE.2015.7113373

[24] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter A. Boncz. 2019. Performance-Optimal Filtering: Bloom overtakes Cuckoo at High-Throughput. *PVLDB* 12, 5, 502–515. https://doi.org/10.14778/3303753.3303757

[25] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5, 643–668. https://doi.org/10.1007/s00778-017-0480-7

[26] Guido Moerkotte and Thomas Neumann. 2006. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. ACM, 930–941. http://dl.acm.org/citation.cfm?id=1164207

[27] Guido Moerkotte and Thomas Neumann. 2008. Dynamic Programming Strikes Back. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. Association for Computing Machinery, New York, NY, USA, 539–552. https://doi.org/10.1145/1376616.1376672

[28] Thomas Neumann. 2009. Query Simplification: Graceful Degradation for Join-Order Optimization. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD'09)*. Association for Computing Machinery, New York, NY, USA, 403–414. https://doi.org/10.1145/1559845.1559889

[29] Thomas Neumann. 2009. Query Simplification: Graceful Degradation for Join-Order Optimization. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD'09)*. Association for Computing Machinery, New York, NY, USA, 403–414. https://doi.org/10.1145/1559845.1559889

[30] Thomas Neumann and César A. Galindo-Legaria. 2013. Taking the Edge off Cardinality Estimation Errors using Incremental Execution. In *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings (LNI)*, Vol. P-214. GI, 73–92. https://dl.gi.de/20.500.12116/17356

[31] Kiyoshi Ono and Guy M. Lohman. 1990. Measuring the Complexity of Join Enumeration in Query Optimization. In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*. Morgan Kaufmann, 314–325.

[32] Felix Putze, Peter Sanders, and Johannes Singler. 2007. Cache-, hash- and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms*. Springer, 108–121.

[33] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. 1996. Cost-Based Optimization for Magic: Algebra and Implementation. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD'96)*. Association for Computing Machinery, New York, NY, USA, 435–446. https://doi.org/10.1145/233269.233360

[34] David Simmen, Eugene Shekita, and Timothy Malkemus. 1996. Fundamental Techniques for Order Optimization. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*

(SIGMOD'96). Association for Computing Machinery, New York, NY, USA, 57–67. https://doi.org/10.1145/233269.233320

[35] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, and et al. 2014. Orca: A Modular Query Optimizer Architecture for Big Data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. Association for Computing Machinery, New York, NY, USA, 337–348. https://doi.org/10.1145/2588555.2595637

[36] Patrick Valduriez and Georges Gardarin. 1984. Join and Semijoin Algorithms for a Multiprocessor Database Machine. *ACM Trans. Database Syst.* 9, 1, 133–161. https://doi.org/10.1145/348.318590

[37] Andreas Weininger. 2002. Efficient Execution of Joins in a Star Schema. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*. Association for Computing Machinery, New York, NY, USA, 542–545. https://doi.org/10.1145/564691.564754

[38] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking Ahead Makes Query Plans Robust. *PVLDB* 10, 8, 889–900. https://doi.org/10.14778/3090163.3090167