# BaCon: Efficient Batch Processing of Counting Queries

Yuxi Liu
Duke University
yuxi.liu@duke.edu

Xiao Hu
University of Waterloo
xiaohu@uwaterloo.ca

Pankaj K. Agarwal
Duke University
pankaj@cs.duke.edu

Jun Yang
Duke University
junyang@cs.duke.edu

## ABSTRACT

Counting queries are ubiquitous in database systems, serving a dual purpose in supporting user analytics and driving internal system optimization. Learned models for cardinality estimation rely heavily on large-scale training data, yet generating such data by executing massive batches of counting queries is expensive. We propose BaCon, an efficient algorithm for batch evaluation of counting queries on top of a database system, without modifying its internals. BaCon integrates the idea of factorized databases with a workload-aware domain quantization strategy, allowing it to evaluate batches of counting queries using compact data structures rather than materializing massive join results. Implemented as a client-side application compatible with an existing database system, BaCon delivers speedups between 2× and 178× over baselines and robust performance against workload variations, making training and maintenance of learned cardinality estimation models significantly more practical.

**PVLDB Reference Format:**
Yuxi Liu, Xiao Hu, Pankaj K. Agarwal, and Jun Yang. BaCon: Efficient Batch Processing of Counting Queries. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/louisja1/bacon.

## 1 INTRODUCTION

Counting queries are not only useful in their own right to database applications, but they also frequently serve the purpose of collecting basic statistics from data for monitoring and optimization. With the growing popularity of *learned query optimization* [12, 42, 43, 73] in recent years, an interesting workload has emerged: collecting training data for *learned cardinality estimation* (CE) [13, 25, 27, 36, 45, 46, 51, 53, 64]. CE is a critical component of query optimization, as its accuracy directly impacts the quality of query execution plans [33, 54]. In these workloads, queries typically involve joins and selections over base tables but report only the final counts of the result sets. These query-count pairs are subsequently used to train CE models. Such queries may be derived from past workloads or synthesized specifically for training. While the number of distinct join patterns is naturally limited by the database schema, the queries themselves additionally contain a variety of selection conditions, targeting different attributes with varying constants. Beyond simple equality comparisons, many conditions involve inequality or range predicates. Table 1 summarizes nine public query workloads widely used in CE research for training and evaluation. These workloads span three commonly studied benchmark databases — IMDB [33], STATS [17] and DSB [10] — and capture a broad range of query shapes, join structures, and predicate types.

Executing such workloads is time-consuming. Even when viewed as a one-time training cost, the overhead can be daunting; in our experience, some workloads require hours or even more than a day to execute (Section 5). Furthermore, models are rarely trained once and then forgotten. As the database state changes, they can become outdated [30, 35]: cardinalities from previously executed queries may no longer be accurate, necessitating the re-execution of queries to retrain the models. While monitoring real result counts from query execution feedback can mitigate this issue, simply observing final query result counts is insufficient [17, 27, 53]. Effective training requires (1) the size of intermediate results for subqueries within *potentially optimal* plans, not just the plan actually executed, and (2) feedback from unseen queries to prepare for potential workload shifts [35, 65, 68]. In a continuously running environment, executing these monitoring or training queries must not disrupt normal workloads. These observations underscore the need for more efficient support for executing batches of counting queries.

Given the extensive literature on query processing, many ideas are applicable to the general problem of batch counting queries — including multi-query optimization, scalable continuous query processing, and factorized databases — see Section 6 for more discussion. A reasonable starting point, leveraging multi-query optimization, is to exploit the fact that many queries share the same join pattern, differing only in their selection conditions. Instead of executing them independently, one could perform the full join first and then apply individual selection conditions to obtain per-query counts. However, while this baseline enables shared processing of joins, it leads to materializing the full join result, which can be massive for large databases.

We propose ***BaCon***, a practical algorithm for efficient ***Ba****tch processing of* ***Coun****ting queries*. A key idea, inspired by work on factorized databases [7, 8, 50], is that a counting query over a join can be evaluated without enumerating the join result. While the count operator cannot generally be pushed down through a join, if the join is processed in a "factorized" manner — grouped by joining attribute values — we can simply count the joining "factors" and


Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

**Table 1:** Summary of nine query workloads across three databases, including the workload name with the link to file, associated database, number of queries (Qs), number of tables per query (Ts), number of join patterns (JPs, formally defined in Section 2), and a short description with source in the second row for each workload.

| Query Workload | DB | # of Qs | # of Ts | # of JPs |
|---|---|---|---|---|
| **synthetic [29]** | IMDB | 5,000 | 1 to 3 | 16 |
| [27]: generated by the same generator as the training data (with a different seed) | | | | |
| **scale [28]** | IMDB | 500 | 1 to 5 | 31 |
| [27]: designed to show how the model (i.e., MSCN) generalizes to more joins. | | | | |
| **job-light [26]** | IMDB | 70 | 2 to 5 | 18 |
| [27]: derived from JOB [33] and does not contain string predicates or disjunctions. | | | | |
| **job-light-single [19]** | IMDB | 254 | 1 | 1 |
| [17]: single-table sub-plan queries extracted from job-light. | | | | |
| **job-light-join [18]** | IMDB | 696 | 2 to 5 | 27 |
| [17]: join sub-plan queries extracted from job-light. | | | | |
| **stats-ceb [23]** | STATS | 146 | 2 to 7 | 58 |
| [17]: designed to be more comprehensive, with more diverse queries and more complex join patterns on STATS, compared to job-light on IMDB. | | | | |
| **stats-ceb-single [22]** | STATS | 632 | 1 | 1 |
| [17]: single-table sub-plan queries extracted from stats-ceb. | | | | |
| **stats-ceb-join [21]** | STATS | 2,603 | 2 to 7 | 120 |
| [17]: join sub-plan queries extracted from stats-ceb. | | | | |
| **dsb-grasp-20k [37]** | DSB | 20,000 | 1 to 5 | 16 |
| A sample of the original workload [67] generated by SeConCDF [68] and GRASP [66] on DSB. From 149,828 original queries, we remove queries not applicable to our methods and randomly sample 20,000 queries (details in Section 5). | | | | |

multiply these counts, avoiding enumeration of the joined result tuples. A second key idea is workload-aware quantization of selection attribute domains. This idea allows the data required for counting queries to be compressed, naturally exploiting the overlap and sharing of selection predicates within the workload. BaCon seamlessly combines these ideas, using compact "count maps" rather than bloated join tuples to represent intermediate results; to combine these intermediate results, it uses a pair of operators $\otimes$ and $\oplus$, defined with clear semantics and amenable to optimization. Finally, to ensure practicality and ease of adoption, we implement BaCon as a client application running on top of a database system (DBMS). It leverages the DBMS for processing, but carefully balances database- and client-side execution and minimizes interfacing overhead.

The result is highly competitive for training workloads for learned CE in Table 1, achieving 2× to 178× speedups over baselines. Moreover, BaCon is robust across workload variations, providing significant speedups for expensive join patterns while remaining competitive on those where baseline approaches are already efficient.

## 2 PRELIMINARIES

*Problem Statement and Notations.* Consider a database $\mathcal{D}$ with $n$ tables $R_1, \ldots, R_n$, where each table $R_i$ has a set of attributes denoted $\text{Attrs}_{R_i}$. We are interested in processing a set of counting queries $\mathcal{Q}$ over $\mathcal{D}$. Each query counts the number of tuples returned by a single-table selection or a selection-join over a subset of the tables in $\mathcal{D}$. We assume equality joins with no cycles or self-joins, and that the selection is a conjunction of predicates comparing a single attribute with a literal using =, >, <=, etc. Extensions to other forms of selection and join predicates and self-joins are possible but not discussed in this paper. We begin by introducing some useful notations. Given each $Q \in \mathcal{Q}$:

**Listing 1:** First 4 queries in `stats_ceb.sql` [23].

```
SELECT COUNT(*) FROM badges as b, users as u -- Q1
 WHERE b.UserId = u.Id AND u.UpVotes >= 0;
SELECT COUNT(*) FROM comments as c, badges as b -- Q2
 WHERE c.UserId = b.UserId AND c.Score = 0 AND b.Date <=
       '2014-09-11 14:33:06'::timestamp;
SELECT COUNT(*) FROM comments as c, postHistory as ph -- Q3
 WHERE c.UserId = ph.UserId AND c.Score = 0 AND
       ph.PostHistoryTypeId = 1;
SELECT COUNT(*) FROM comments as c, postHistory as ph -- Q4
 WHERE c.UserId = ph.UserId AND ph.PostHistoryTypeId = 1 AND
       ph.CreationDate >= '2010-09-14 11:59:07'::timestamp;
```

- $\text{Tables}_Q$ denotes the subset of tables in $\mathcal{D}$ referenced by $Q$.
- $\text{JAttrs}_Q$ denotes the subset of attributes in $\text{Tables}_Q$ referenced by the join predicates of $Q$.
- $\text{JPreds}_Q$ denotes $Q$'s join predicates, represented as the set of equivalent classes of attributes in $\text{JAttrs}_Q$ induced by $Q$'s join predicates. Two attributes $A_1, A_2$ belong to the same equivalent classes iff $A_1 = A_2$ is logically implied by $Q$'s join predicates.
- Given disjoint subsets $\mathcal{R}$ and $\mathcal{R}'$ of tables in $\text{Tables}_Q$, $\text{JAttrs}_Q[\mathcal{R}|\mathcal{R}']$ denotes the subset of attributes of $\mathcal{R}$ that are used by $Q$ to join with $\mathcal{R}'$; i.e., $\text{JAttrs}_Q[\mathcal{R}|\mathcal{R}'] = \{A \in \bigcup_{R\in\mathcal{R}} \text{Attrs}_R \mid \exists E \in \text{JPreds}_Q : A \in E \wedge (\exists A' \in E : A' \in \bigcup_{R\in\mathcal{R}'} \text{Attrs}_R)\}$.
- $\text{SAttrs}_Q$ denotes the subset of attributes in $\text{Tables}_Q$ referenced by $Q$'s selection predicates.
- $\text{SPreds}_Q$ denotes $Q$'s selection predicates, represented as a mapping from each attribute $A \in \text{SAttrs}_Q$ to a range $G$ over the domain of $A$, such that $G$ is the widest range such that $A \in G$ is logically implied by $Q$'s selection predicates.
- Given a subsets $\mathcal{R}$ of tables in $\text{Tables}_Q$, $\text{SAttrs}_Q[\mathcal{R}]$ denotes the set of attributes of $\mathcal{R}$ referenced by $Q$'s selection conditions; i.e., $\text{SAttrs}_Q(\mathcal{R}) = \text{SAttrs}_Q \cap \bigcup_{R\in\mathcal{R}} \text{Attrs}_R$.

For example, Listing 1 shows the first four queries in the workload `stats_ceb.sql` [23] over the STATS database [17]. For $Q_4$:

- $\text{Tables}_{Q_4} = \{\text{comments}, \text{postHistory}\}$;
- $\text{JAttrs}_{Q_4} = \{\text{comments.UserId}, \text{postHistory.UserId}\}$;
- $\text{JPreds}_{Q_4} = \{\{\text{comments.UserId}, \text{postHistory.UserId}\}\}$ (both join attributes are in one equivalence class);
- $\text{JAttrs}_{Q_4}[\{\text{comments}\}|\{\text{postHistory}\}] = \{\text{comments.UserId}\}$;
- $\text{JAttrs}_{Q_4}[\{\text{postHistory}\}|\{\text{comments}\}] = \{\text{postHistory.UserId}\}$;
- $\text{SAttrs}_{Q_4} = \{\text{postHistory.PostHistoryTypeId}, \text{postHistory.CreationDate}\}$;
- $\text{SPreds}_{Q_4} = \{\text{postHistory.PostHistoryTypeId} \mapsto [1, 1],$ $\text{postHistory.CreationDate} \mapsto [\text{'2010-09-14 11:59:07'::timestamp}, \infty)\}$.

*Baseline Solution: Independent Processing (**IndProc**).* A straightforward solution, adopted by most existing work on learned CE, runs the queries in $\mathcal{Q}$ one by one, independently, using the DBMS that manages $\mathcal{D}$. We call this approach *IndProc* for short. The performance of IndProc is heavily dependent on the underlying DBMS. A capable DBMS will optimize each query by pushing down selection conditions, reordering joins, and choosing appropriate join methods, leveraging data statistics and available indices. Caching by the DBMS buffer pool may also improve execution performance across queries. On the other hand, most DBMS lack advanced methods for optimizing multiple queries simultaneously. Moreover, most

**Listing 2:** Computing $k$ queries with join template $\mathfrak{J}$ in PostFilt.

```
SELECT COUNT(CASE WHEN SPreds_Q1 THEN 1 END), ...,
    COUNT(CASE WHEN SPreds_Qk THEN 1 END)
FROM Tables_J WHERE JPreds_J;
```

of them choose to execute a counting query $Q$ by first joining all tables in $Q$ before applying the final COUNT aggregation, failing to explore opportunities for pushing COUNT below joins.

*Join Patterns.* Before presenting a more advanced baseline solution as well as our solution in Section 3, we introduce a concept used by both. The *join pattern* of a query $Q$ is characterized by $\langle \text{Tables}_Q, \text{JPreds}_Q \rangle$, i.e., the tables $Q$ joins and $Q$'s join predicates. Suppose queries in $\mathcal{Q}$ have $K$ distinct join patterns $\mathfrak{J}_1, \ldots, \mathfrak{J}_K$; these patterns partition $\mathcal{Q}$ into a disjoint union of $K$ subsets of queries, denoted $\mathcal{Q}[\mathfrak{J}_1], \ldots, \mathcal{Q}[\mathfrak{J}_K]$, where each $\mathcal{Q}[\mathfrak{J}_i]$ is the set of queries with join pattern $\mathfrak{J}_i$ (but may differ in their selection predicates).

For example, $Q_3$ and $Q_4$ in Listing 1 have the same join pattern (with tables {comments, postHistory} and join predicate comments.UserId = postHistory.UserId, despite having different selection predicates). $Q_1$ and $Q_2$ each contribute one more distinct join pattern. In practice, because all queries in $\mathcal{Q}$ come from the same underlying database $\mathcal{D}$, the number $K$ of distinct join patterns tends to be much smaller than the number of queries. For example, synthetic.sql [29] contains 5,000 queries but only 16 distinct join patterns.

*Alternative Baseline: Join and Post-Filtering (**PostFilt**).* Intuitively, this approach seeks to avoid redundant computation across queries that share identical join patterns. We pre-process the input queries into partitions $\mathcal{Q}[\mathfrak{J}_1], \ldots, \mathcal{Q}[\mathfrak{J}_K]$ according to their join patterns. This step requires only a single scan over $\mathcal{Q}$ to perform syntactical analysis. For each query partition $\mathcal{Q}[\mathfrak{J}_i]$, we only need to compute the join once, followed by a post-filtering step, where we check each join result tuple against the selection predicates of queries in $\mathcal{Q}[\mathfrak{J}_i]$ to determine which queries need to have their counters incremented. We call this baseline *PostFilt*.

There are many options for performing the post-filtering step, as discussed in Section 6, including methods that first build a data structure for $\mathcal{Q}[\mathfrak{J}_i]$ to enable efficient identification and updating of counters in time sublinear in $|\mathcal{Q}[\mathfrak{J}_i]|$ per join result tuple. While some these methods make it possible to scale to thousands or millions of queries, the typical workloads we target do not have such a large number of queries per join pattern, as evidenced in Table 1. Through our experimentation, we have found a simple strategy leveraging the underlying DBMS to be the most effective. Given a join pattern $\mathfrak{J} = \langle \text{Tables}_\mathfrak{J}, \text{JPreds}_\mathfrak{J} \rangle$ and $k$ queries $Q_1, \ldots, Q_k$ sharing this join pattern, we issue a single SQL query in Listing 2 to compute all of them. Here, SPreds, Tables, and JPreds are translated into SQL. Each CASE expression determines whether a join result tuple contributes to a query $Q_i$'s result count by checking its selection predicates (a failed check yields NULL, which is ignored by COUNT). In the end, a single $k$-component result tuple is computed, with each component supplying the result count for one query.

*Discussion.* Compared with IndProc, PostFilt eliminates redundant computation of joins among queries sharing the same join pattern. However, IndProc can still outperform PostFilt if the selection predicates of these queries have little overlap, and if database indices enable IndProc to apply selective selection predicates early in query processing. In contrast, PostFilt has no effective means to push selection predicates down because in most workloads, the disjunction of all selection predicates from multiple queries cannot be expressed as a succinct, "sargable" [57] WHERE condition without introducing many false positives. Furthermore, most database optimizers do not push aggregation below joins, let alone those with conditional expression inputs as in Listing 2. Therefore, PostFilt effectively enumerates the full join result before post-filtering and counting — a key limitation that we seek to overcome.

## 3 BASIC BACON

This section introduces basic BaCon, focusing on key ideas and the high-level algorithm. Many implementation and optimization details are also crucial to making BaCon competitive in practice, but to simplify presentation, we defer their discussion to Section 4. We start with three key ideas, along with some results and definitions based on them; we then describe the algorithm. Like PostFilt, given a set $\mathcal{Q}$ of queries, BaCon partitions $\mathcal{Q}$ into subsets according to join patterns, such that queries in each subset $\mathcal{Q}[\mathfrak{J}]$ share the same join pattern $\mathfrak{J} = \langle \text{Tables}_\mathfrak{J}, \text{JPreds}_\mathfrak{J} \rangle$. Hence, most of this section focuses on how to process one such subset of queries given $\mathfrak{J}$.

Before proceeding, we briefly present a geometric view of the problem for intuition. Each result tuple in the cross product of $\text{Tables}_\mathfrak{J}$ can be seen as a point in a high-dimensional space $\mathbb{S}$, where each dimension corresponds to an attribute in $\bigcup_{R \in \text{Tables}_\mathfrak{J}} \text{Attrs}_R$, ignoring those that are not referenced by any predicate in $\mathcal{Q}[\mathfrak{J}]$. Let $J$ denote the result of the full join of $\text{Tables}_\mathfrak{J}$ with $\text{JPreds}_\mathfrak{J}$. Points in $J$ are those cross-product points that lie on a hyperplane $\mathbb{J}$ in $\mathbb{S}$ defined by $\text{JPreds}_\mathfrak{J}$. Each query $Q \in \mathcal{Q}[\mathfrak{J}]$ corresponds to an orthogonal hyperrectangle in $\mathbb{S}$ with range predicates restricted to dimensions in $\text{SAttrs}_Q$. Our problem boils down to counting, for each $Q$, how many points in $J$ fall into $Q$'s hyperrectangle. Intuitively, these points are not positioned arbitrarily: not only do they lie on $\mathbb{J}$ because of the join predicates, but they also come from the cross product of $\text{Tables}_\mathfrak{J}$, which means that their projections onto the subspace $\mathbb{S}[\text{Attrs}_R]$ for each $R \in \text{Tables}_\mathfrak{J}$ cannot exceed $|R|$ distinct points, which is often much smaller than $|J|$. These special properties make it possible to perform our counting tasks more efficiently than simply enumerating $J$ upfront (which PostFilt does). Further efficiency can be gained by exploiting overlapping among the query hyperrectangles.

### 3.1 Conditional Orthogonality of Joins

The first idea has its roots in the well-studied problem of factorized databases and related WCO join methods (further discussed in Section 6). A simple observation is that, in order to count the number of result tuples in a cross product between two tables $R_1$ and $R_2$, we just need to calculate $|R_1| \times |R_2|$, without enumerating $R_1 \times R_2$. While the same observation no longer holds for $|R_1 \bowtie_{R_1.A=R_2.A} R_2|$, if we additionally set the join attribute value $A = v$, then the number of result tuples *conditioned on this specific setting* can still be computed directly: i.e., $|\sigma_{A=v}(R_1 \bowtie_{R_1.A=R_2.A} R_2)| = |\sigma_{A=v} R_1| \times |\sigma_{A=v} R_2|$. We can generalize this idea further to a star-shaped join as follows.

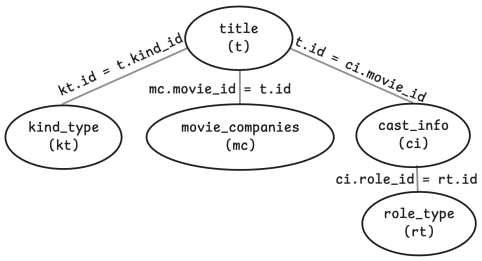


**Figure 1:** A join pattern in the IMDB schema [34].

Lemma 3.1 (Conditional Orthogonality of Joins). *Consider a star-shaped join query centered at $E_0$:*

$$E_0 \bowtie_{\theta_1} E_1 \bowtie_{\theta_2} \cdots \bowtie_{\theta_n} E_n.$$

*Here, the $E_i$'s are subqueries, and for each $i \in [1, n]$, $\theta_i$ is a conjunction of predicates equating pairs of attributes from $E_0$ and $E_i$. Note that there are no join predicates across $E_1, \ldots, E_n$. Denote by $\mathcal{A}_0$ the set of join attributes from $E_0$ referenced by $\theta_1, \ldots, \theta_n$. Let $v$ be any mapping of every attribute $A \in \mathcal{A}_0$ to a value $v(A)$ in $A$'s domain, and let $v[\![\theta_i]\!]$ denote the condition obtained by applying $v$ to $\theta_i$ (i.e., replacing each $A \in \mathcal{A}_0$ by $v(A)$ — note that the resulting condition becomes a selection over $E_i$). The following equivalence holds:*

$$\sigma_{\wedge_{A\in\mathcal{A}_0} A=v(A)} \left(E_0 \bowtie_{\theta_1} E_1 \bowtie_{\theta_2} \cdots \bowtie_{\theta_n} E_n\right)$$
$$= \left(\sigma_{\wedge_{A\in\mathcal{A}_0} A=v(A)} E_0\right) \times \left(\sigma_{v[\![\theta_1]\!]} E_1\right) \times \cdots \times \left(\sigma_{v[\![\theta_n]\!]} E_n\right).$$

*Example 3.2.* Consider the join pattern illustrated in Figure 1. Tables `title AS t`, `kind_type AS kt`, and `movie_companies AS mc` correspond to $E_0, E_1, E_2$ in Lemma 3.1, while the join between `cast_info AS ci` and `role_type AS rt` corresponds to $E_3$. In `title`, suppose we fix `t.kind_id`=$x$ (which joins `kt.id`) and `t.id`=$y$ (which joins `mc.movie_id` and `ci` $\bowtie$ `rt`). By Lemma 3.1, the number of full join result tuples with (`t.kind_id`, `t.id`) = $(x, y)$ can be computed directly as: $|\sigma_{\texttt{t.kind\_id}=x \wedge \texttt{t.id}=y}\texttt{t}| \times |\sigma_{\texttt{kt.id}=x}\texttt{kt}| \times |\sigma_{\texttt{mc.movie\_id}=y}\texttt{mc}| \times |\sigma_{\texttt{ci.movie\_id}=y}(\texttt{ci} \bowtie \texttt{rt})|$, without enumerating the full join result tuples. To count the full join result size overall, we iterate over all possible combinations of (`t.kind_id`, `t.id`) values in `t`, apply the above procedure to each combination, and tally the total.

The style of processing illustrated by the above example has been used recently for efficient computation of aggregate queries [31]. Processing the subtree $\sigma_{\texttt{ci.movie\_id}=y}(\texttt{ci} \bowtie \texttt{rt})$ can be handled by the same procedure. Later in this section, we will see how to extend the idea to computing result counts of multiple queries with different selection predicates beyond simply counting the full join result.

## 3.2 Quantization of Selection Attributes

While tables and attribute domains can be large, the number of queries in $\mathcal{Q}$ places a natural constraint on the number of constants from each domain appearing in selection predicates. In other words, from the perspective of $\mathcal{Q}$, many fine-grained differences among attribute values do not affect result counts. Our key idea is to effectively compress the attribute domains using workload-aware *quantiziation*, turning large, complex domains into a small range of integers that are efficient to work with.

Given the set $\mathcal{Q}[\mathfrak{J}]$ of queries with the same join pattern $\mathfrak{J}$, let $\textsc{SAttrs}_{\mathfrak{J}} = \bigcup_{Q\in\mathcal{Q}[\mathfrak{J}]} \textsc{SAttrs}_Q$ denote the set of selection attributes. We construct a *quantization scale* $\mathfrak{b}_A$ for each selection attribute $A \in \textsc{SAttrs}_{\mathfrak{J}}$ as follows. First, we extract from $\mathcal{Q}[\mathfrak{J}]$ the set $\Delta = \{\textsc{SPreds}_Q(A) \mid \exists Q \in \mathcal{Q}[\mathfrak{J}] : A \in \textsc{SPreds}_Q\}$ of predicate ranges associated with $A$. We sort all range endpoints, which partition the domain of $A$ into an ordered list of atomic ranges. For each atomic range contained in at least one query range in $\Delta$, we create a new *bucket* in $\mathfrak{b}_A$ and assign it a serial integer id (starting with 1). Hence, $\mathfrak{b}_A$ maps each relevant atomic range (bucket) to an integer, preserving order. A value $v$ from $A$'s domain is quantized into an integer $\mathfrak{b}_A(v)$, the id of the bucket containing $v$, or 0 if $v$ lies outside all of buckets in $\mathfrak{b}_A$. We present the detailed construction algorithm in [38], which handles additional intricacies with open/close intervals.

We denote by $\mathfrak{B}_{\mathfrak{J}} = \{\mathfrak{b}_A \mid A \in \textsc{SAttrs}_{\mathfrak{J}}\}$ the collection of all attribute quantization scales for $\mathcal{Q}[\mathfrak{J}]$, and $\mathfrak{B}_{\mathfrak{J}}[R] = \{\mathfrak{b}_A \mid A \in \textsc{SAttrs}_{\mathfrak{J}} \cap \textsc{Attrs}_R\}$ those for attributes in table $R \in \textsc{Tables}_{\mathfrak{J}}$. Returning to the geometric view introduced at the beginning of the section, $\mathfrak{B}_{\mathfrak{J}}$ induces a grid structure over the subspace $\mathbb{S}[\textsc{SAttrs}_{\mathfrak{J}}]$ consisting of the selection attribute dimensions. By construction of $\mathfrak{B}_{\mathfrak{J}}$, all query hyperrectangles perfectly align with grid boundaries. As example, Figure 2 shows the grid induced by the two quantization scales $\mathfrak{b}_{\texttt{company\_type\_id}}$ and $\mathfrak{b}_{\texttt{company\_id}}$ for table `mc`. Note that $\mathfrak{b}_{\texttt{company\_type\_id}}$ assigns bucket 0 to the five leftmost tuples, while $\mathfrak{b}_{\texttt{company\_id}}$ does not assign bucket 0 because the predicates from $Q_1, Q_2$ already cover the entire domain of `company_id`.

The following lemma formalizes the guarantee that precise evaluation of selection predicates are possible in the quantized space.

Lemma 3.3 (Quantization Preserves Selections). *Given a set of selection-join queries $\mathcal{Q}$ and quantization scales $\mathfrak{B}$ constructed from $\mathcal{Q}$, there exists a function $f(\mathfrak{b}_A, \delta)$ returning an integer range $[i_1, i_2]$ for a range $\delta$ over an attribute $A$ with a quantization scale $\mathfrak{b}_A \in \mathfrak{B}$, such that for any $Q \in \mathcal{Q}$ and every selection predicate $A \mapsto \delta$ in $\textsc{SPreds}_Q$: $A \in \delta \Leftrightarrow \mathfrak{b}_A(A) \in f(\mathfrak{b}_A, \delta)$.*

## 3.3 Quantized Selection Count Maps

Since queries in $\mathcal{Q}[\mathfrak{J}]$ ultimately only care about counts, a natural idea following selection attribute quantization is to further aggregate the points that fall into each grid cell induced by $\mathfrak{b}_{\mathfrak{J}}$ into a single count, instead of enumerating them.

Given a table $R \in \textsc{Tables}_{\mathfrak{J}}$ with selection attributes $\textsc{SAttrs}_R = \{A_1, \ldots, A_k\}$ and quantization scales $\mathfrak{B}_{\mathfrak{J}}[R] = \{\mathfrak{b}_{A_1}, \ldots, \mathfrak{b}_{A_k}\}$, we compress a subset $T$ of tuples in $R$ into a *(quantized selection) count map* $\mathfrak{M}$: each entry of $\mathfrak{M}$ maps a *grid coordinate* $\vec{b} = \langle b_1, \ldots, b_k\rangle$ (where each $b_i$ represents a bucket id of $\mathfrak{b}_{A_i}$) to $\mathfrak{M}[\vec{b}]$, the count of tuples $t \in T$ that fall within the corresponding grid cell, i.e., $\mathfrak{b}_{A_i}(t.A_i) = b_i$ for all $i \in [1, k]$. For example, Figure 2 shows the count map for the set of points.

We generalize the concept of count map $\mathfrak{M}$ over any subset of tables $\mathcal{R} \subseteq \textsc{Tables}_{\mathfrak{J}}$. Let $\textsc{SAttrs}_{\mathcal{R}} = \textsc{SAttrs}_{\mathfrak{J}} \cap \bigcup_{R\in\mathcal{R}} \textsc{Attrs}_R$ denote the set of selection attributes in $\mathcal{R}$. A grid coordinate for $\mathfrak{M}$, with one component for each attribute in $\textsc{SAttrs}_{\mathcal{R}}$, identifies a grid cell induced by quantization scales $\{\mathfrak{b}_A \mid A \in \textsc{SAttrs}_{\mathcal{R}}\}$ in subspace $\mathbb{S}[\textsc{SAttrs}_{\mathcal{R}}]$. Given a subset $T$ of tuples in the cross product of $\mathcal{R}$ corresponding to points in $\mathbb{S}[\textsc{SAttrs}_{\mathcal{R}}]$, $\mathfrak{M}$ counts the points in $T$ in each grid cell.

Our goal is to construct a count map over the entire $\textsc{Tables}_{\mathfrak{J}}$ for the full join result set $J$, but importantly, without enumerating $J$.

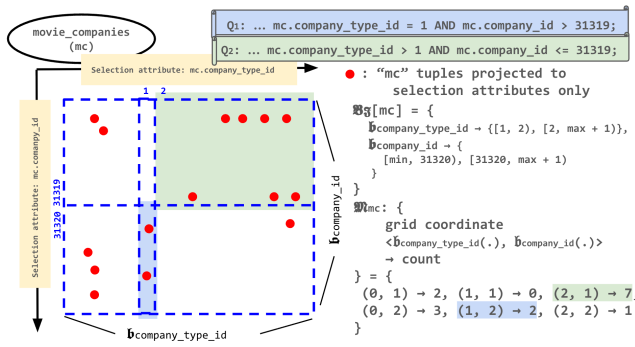


**Figure 2:** Quantization and quantized selection count map on mc for $Q_1$, $Q_2$ (displaying subqueries on mc only).

Intuitively, we would like to use count maps to represent intermediate results compactly. Obtaining a count map for a single table is easy, but combining count maps for different subsets of tables into a big one is complicated by joins, because not all points in two grid cells in orthogonal subspaces will join with each other. However, leveraging Lemma 3.1, we can process points in $J$ in groups: points in each group all share appropriate join attribute values, thereby allowing count maps to be computed for different subsets of tables and then conveniently "multiplied." Then, the product count maps across groups can be "added" to obtain the final result count map. An example will be provided in Section 3.4.

We formally define "multiply" and "add" as follows:

⊗ **(multiply)** $\mathfrak{M}_1$ and $\mathfrak{M}_2$ are count maps over disjoint subsets $\mathcal{R}_1$ and $\mathcal{R}_2$ of tables in $\mathit{Tables}_{\mathfrak{J}}$. We define $\mathfrak{M}_1 \otimes \mathfrak{M}_2$, a count map over $\mathcal{R}_1 \cup \mathcal{R}_2$, as $\left\{\vec{b}_1 \frown \vec{b}_2 \mapsto (\mathfrak{M}_1[\vec{b}_1] \cdot \mathfrak{M}_2[\vec{b}_2]) \mid \vec{b}_1 \in \mathfrak{M}_1, \vec{b}_2 \in \mathfrak{M}\right\}$, where $\frown$ concatenates grid coordinate vectors.

⊕ **(add)** $\mathfrak{M}_1$ and $\mathfrak{M}_2$ are count maps over the same subset of tables $\mathcal{R} \subseteq \mathit{Tables}_{\mathfrak{J}}$ (and hence same quantization scales). We define $\mathfrak{M}_1 \oplus \mathfrak{M}_2$, a count map over $\mathcal{R}$, as $\left\{\vec{b} \mapsto (\mathfrak{M}_1[\vec{b}] + \mathfrak{M}_2[\vec{b}]) \mid \vec{b} \in \mathfrak{M}_1\right\}$.

The following lemmas establish the correctness of using ⊗ and ⊕ for computing count maps. Formally, given $Q[\mathfrak{J}]$, quantization scales $\mathfrak{B}_{\mathfrak{J}}$, a subset of tables $\mathcal{R} \subseteq \mathit{Tables}_{\mathfrak{J}}$, and a subset $T$ of tuples in the cross product of $\mathcal{R}$, we say that a count map $\mathfrak{M}$ over $\mathcal{R}$ is *complete* with respect to $T$ if for any query $Q \in Q[\mathfrak{J}]$, the size of the intersection between $T$ and the selection-join subquery of $Q$ restricted[1] to $\mathcal{R}$ can be computed from $\mathfrak{M}$ and $\mathfrak{B}_{\mathfrak{J}}$.

Lemma 3.4 (Multiplying Count Maps). *Given $Q[\mathfrak{J}]$, consider disjoint subsets $\mathcal{R}_0, \ldots, \mathcal{R}_n$ of tables in $\mathit{Tables}_{\mathfrak{J}}$, where $\mathit{JPreds}_{\mathfrak{J}}$ implies a join condition $\theta_i$ relating $\mathcal{R}_0$ to $\mathcal{R}_i$ for each $i \in [1, n]$, but no join condition across $\mathcal{R}_1, \ldots, \mathcal{R}_n$ that is not already implied by $\bigwedge_i \theta_i$. For each $i \in [1, n]$, let $E_i$ denote the join of tables in $\mathcal{R}_i$, with condition implied by $\mathit{JPreds}_{\mathfrak{J}}$. Consider the star-shaped join query centered at $E_0$:*

$$E_0 \bowtie_{\theta_1} E_1 \bowtie_{\theta_2} \cdots \bowtie_{\theta_n} E_n,$$

*which conforms to the query structure in Lemma 3.1. Denote by $\mathcal{A}_0 = \mathit{JAttrs}_{\mathfrak{J}}[\mathcal{R}_0 \mid \cup_{i\in[1,n]} \mathcal{R}_i]$ the set of join attributes from $E_0$ referenced by $\theta_1, \ldots, \theta_n$. Let $v$ denote any mapping of every attribute $A \in \mathcal{A}$ to a value $v(A)$ in $A$'s domain, for some attribute set $\mathcal{A}$ where $\mathcal{A}_0 \subseteq \mathcal{A} \subseteq \bigcup_{R\in\mathcal{R}_0} \mathit{Attrs}_R$. Suppose:*

- *$\mathfrak{M}_0$ is a count map over $E_0$ complete w.r.t. $\sigma_{\wedge_{A\in\mathcal{A}} A=v(A)} E_0$; and*
- *$\forall i \in [1, n]$: $\mathfrak{M}_i$ is a count map over $E_i$ complete w.r.t. $\sigma_{v[\![\theta_1]\!]} E_i$.*

*Then, $\mathfrak{M}_0 \otimes \mathfrak{M}_1 \otimes \cdots \otimes \mathfrak{M}_n$ is complete with respect to*

$$\sigma_{\wedge_{A\in\mathcal{A}} A=v(A)} \left(E_0 \bowtie_{\theta_1} E_1 \bowtie_{\theta_2} \cdots \bowtie_{\theta_n} E_n\right).$$

Lemma 3.5 (Adding Count Maps). *Given $Q[\mathfrak{J}]$ and a subset of tables $\mathcal{R} \subseteq \mathit{Tables}_{\mathfrak{J}}$, suppose $\mathfrak{M}_1, \ldots, \mathfrak{M}_n$ are count maps over $\mathcal{R}$, where each $\mathfrak{M}_i$ is complete with respect to a subset $T_i$ of tuples in the cross product of $\mathcal{R}$. If $T_1, \ldots, T_n$ are disjoint, then $\mathfrak{M}_1 \oplus \cdots \oplus \mathfrak{M}_n$ is complete with respect to $\bigcup_{i\in[1,n]} T_i$.*

[1] More precisely, we remove from $Q$ any table outside $\mathcal{R}$; remove from any equivalence class in $\mathit{JPreds}_Q$ any attribute outside $\mathcal{R}$ and then remove any empty or singleton equivalence class; and remove any entry in $\mathit{SPreds}_Q$ for an attribute outside $\mathcal{R}$.

**Algorithm 1** BaCon$(Q, \mathcal{D})$

**Input:** A set $Q$ of counting selection-join queries over $\mathcal{D}$.
**Output:** Result count for each query in $Q$.

```
Scan Q and partition it into ⋃_ℑ Q[ℑ] by join pattern;
for each subset Q[ℑ] of Q do
    B_ℑ ← {QuantScales(R, Q[ℑ]) | R ∈ Tables_ℑ};
    Determine the plan tree for ℑ;
    M ← BaConRecurse(root(ℑ), ∅);
    yield from ComputeCounts(Q[ℑ], M);
end for
```

**Algorithm 2** BaConRecurse$(R, u)$

**Input:** Mapping $u$ binds attributes a subset of $R$'s attributes to specific values. Implicitly, the function also has access to $\mathcal{D}$, $\mathfrak{J}$ and its plan tree, and the quantization scales $\mathfrak{B}_{\mathfrak{J}}$.
**Output:** A count map $\mathfrak{M}$ over $\mathit{subtree}(R)$, complete w.r.t. result tuples of $Q[\mathfrak{J}]$ restricted to $\mathit{subtree}(R)$ and consistent with $u$.

```
M ← ∅;   ▷ missing entries in all maps default to count 0
for each subsequence S[v] of entries of the form ⟨v, ·, ·⟩ with the
same v, returned by ProcessTable(R, u, JAttrs_ℑ(R | children(R))) do
    M_0 ← {b⃗ ↦ c | ⟨v, b⃗, c⟩ ∈ S[v]};
    for each table R_i ∈ children(R) do
        u_i ← { A' ↦ v(A) | A ∈ JAttrs_ℑ(R|R_i) ∧ A' ∈ JAttrs_ℑ(R_i|R)
                            ∧ JPreds_ℑ ⇒ (A = A') };
        M_i ← BaConRecurse(R_i, u_i);
        M_0 ← M_0 ⊗ M_i;
    end for
    M ← M ⊕ M_0;
end for
return M;
```

## 3.4 Basic BaCon Algorithm

We are now ready to describe the overall BaCon algorithm (Algorithm 1). First, we make a pass over all queries and partition them into subset where each contains queries with the same join pattern. For each subset $Q[\mathfrak{J}]$ with join pattern $\mathfrak{J}$, we construct the quantization scales for them, as discussed in Section 3.2. We then determine an "execution plan" for $\mathfrak{J}$ as an ordered tree, akin to the notion of "(paths in) f-tree" in factorized database literature [8, 50]. Nodes in this tree correspond to tables of $\mathit{Tables}_{\mathfrak{J}}$, and the edges connecting the nodes represent equijoin conditions among them. Assuming

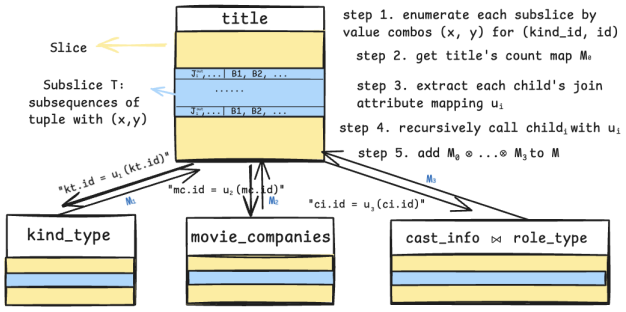


**Figure 3:** Illustration of Algorithm 2, continuing Figure 1.

that queries are acyclic and contain no cross products, one such tree always exists to capture $\text{JPreds}_{\mathfrak{J}}$ precisely. Indeed, there can be multiple alternative plan trees; we defer the discussion of how to choose one to Section 4. At a high level, we process $Q[\mathfrak{J}]$ using an in-order traversal of the plan tree starting from its root, by calling its main workhorse, the recursive procedure BaConRecurse (Line 5), to compute a count map for $Q[\mathfrak{J}]$. This count map is then used to calculate the result counts for queries — naively by summing up the counts of grid coordinates that lie in each query hyperrectangle — but we describe an optimized implementation in Section 4.

BaConRecurse (Algorithm 2, illustrated in Figure 3) is called on a plan tree node (table) $R$ and a filter specified by a mapping $u$ that binds some attributes of $R$ to specific values, which defines a "slice," or the subset $T$ of result tuples, of the join of $\text{subtree}(R)$ (set of tables in the subtree rooted at $R$) satisfying $u$. The goal of BaConRecurse is to compute a count map for this slice $T$ (the call to $\text{root}(\mathfrak{J})$ has $u = \emptyset$, so its slice in fact contains all results tuples in the full join of $\text{Tables}_{\mathfrak{J}}$). As discussed in Section 3.3, processing the entirety of $T$ efficiently is hard; instead, BaConRecurse partitions the slice into "subslices," where each slice $T_v$ binds a particular combination $v$ of values for $R$'s attributes that join with its children, i.e., $\text{JAttrs}_{\mathfrak{J}}(R \mid \text{children}(R))$. To this end, BaConRecurse calls ProcessTable (Listing 3) to enumerate all possible such bindings (Line 2). Each iteration of the loop (Lines 2–10) computes a count map for the subslice $T_v$ defined by a particular $v$. By Lemma 3.4, this count map can be computed by projecting $T_v$ into "projected subslices," one for $R$ and each of $R$'s child subqueries, and multiplying the count maps of these projected subslices. Conveniently, the call to ProcessTable simultaneously computes the coordinate-count pairs to populate the count map for each projected subslice for $R$. We recursively call BaConRecurse on each child of $R$ to compute the count map for its projected subslice. Line 7, using $\otimes$, combines these count maps into a single count map for $T_v$. Finally, applying Lemma 3.5, Line 9 uses $\oplus$ to accumulate the count maps for all $T_v$'s into the final count map for $T$ to be returned.

*Example 3.6.* Following Figure 3, at the root `title`, BaCon iterates over all possible bindings for $(\texttt{t.kind\_id}, \texttt{t.id})$ in turn. For example, the blue subslice fixes $\texttt{t.kind\_id} = 1, \texttt{t.id} = 1$. For this subslice, suppose $\mathfrak{M}_0 = \mathfrak{M}_{\texttt{t}} = \{(9) \mapsto 2, \dots\}$. We recursively call each child with the corresponding binding, as indicated by the arrows pointing to each child. For the "projected subslices" on children, we further assume that $\mathfrak{M}_2 = \mathfrak{M}_{\texttt{mc}}$ is the one shown in Figure 2. We omit $\mathfrak{M}_1$ ($\mathfrak{M}_{\texttt{kt}}$) and $\mathfrak{M}_3$ ($\mathfrak{M}_{\texttt{ci}\bowtie\texttt{rt}}$) for simplicity. The contribution of this subslice to the global count map $\mathfrak{M}$ is: $\mathfrak{M} \leftarrow \mathfrak{M} \oplus (\mathfrak{M}_0 \otimes \cdots \otimes \mathfrak{M}_3) = \mathfrak{M} \oplus \{\dots, (*, 9, 0, 1, *) \mapsto 4p, \dots, (*, 9, 2, 1, *) \mapsto 14q, \dots\}$, where $*$ denotes a grid coordinate from $\mathfrak{M}_1$ or $\mathfrak{M}_3$, and $p, q$ are products of a count from $\mathfrak{M}_1$ and a count from $\mathfrak{M}_3$. Concretely, the middle entries in $(*, 9, 2, 1, *)$ result from concatenating $(9)$ from $\mathfrak{M}_0$ and $(2, 1)$ from $\mathfrak{M}_2$, yielding a product of counts is $14 = 7 \times 2$. After processing this subslice, BaConRecurse proceeds to the next $(\texttt{t.kind\_id}, \texttt{t.id})$ combination, following the ordered order of `t` tuple.

**Listing 3:** SQL code for $\text{ProcessTable}(R, u, \mathcal{A}^{\text{out}})$.

**Input:** Mapping $u$ binds attributes $J_1^{\text{in}}, J_2^{\text{in}}, \dots$ to specific values, and $\mathcal{A}^{\text{out}} = \{J_1^{\text{out}}, J_2^{\text{out}}, \dots\}$ specifies the set of attributes for partitioning result entries. The quantization scales are $\vec{\mathfrak{B}}_{\mathfrak{J}}[R] = \{\mathfrak{b}_{S_1}, \mathfrak{b}_{S_2}, \dots\}$.
**Output:** Result entries have the form $\langle v, \vec{b}, c \rangle$, sorted by $v$, where $v$ is a mapping from $\mathcal{A}^{\text{out}}$ to values, $\vec{b}$ is a grid coordinate for $\vec{\mathfrak{B}}_{\mathfrak{J}}[R]$, and $c$ is the associated count.

```
SELECT J1^out, J2^out, ..., B1, B2, ..., COUNT(*)
FROM (
    SELECT J1^out, J2^out, ...,
        quantize(S1, b_S1) AS B1, quantize(S2, b_S2) AS B2, ...
    FROM R
    WHERE J1^in = u(J1^in) AND J2^in = u(J2^in) AND ...
) AS TMP
GROUP BY J1^out, J2^out, ..., B1, B2, ...
ORDER BY J1^out, J2^out, ...;
```

Delving into ProcessTable (Listing 3), we see how a single SQL query over a table $R$ performs "subslicing" (by $\mathcal{A}^{\text{out}}$) and computes the count maps of the projected subslices together. In the `FROM` subquery that defines `TMP`, the `WHERE` condition applies the attribute binding $u$ that defines $T$, the slice of interest; the `SELECT` clause uses a user-defined function `quantize` to check each selection attribute value against a quantization scale and returns its grid coordinate. The outer query groups the `TMP` tuples by $\mathcal{A}^{\text{out}}$ to perform subslicing, and then by their grid coordinates to compute tuple count per grid cell for each projected subslice. Finally, the `ORDER BY` clause ensures that entries for the same projected subslice, i.e., those with the same binding $v$ for $\mathcal{A}^{\text{out}}$, appear consecutively in the output. This ordering allows the caller, BaConRecurse, to detect when a group of entries with a new $v$ starts, so it can start processing a new subslice.

## 4 ADDITIONAL OPTIMIZATION AND IMPLEMENTATION DETAILS

This section fills in some of the details about BaCon not covered by Section 3 and describes additional optimizations needed to make BaCon practical for query workloads for training learned CE models. When actual workloads deviate from what we expect from typical scenarios (e.g., when predicates in training queries have little overlap or when full join size is small to begin with), BaCon may not outperform baselines IndProc or PostFilt. Therefore, at the end of this section, we also present an approach that serves as an optimizer for picking the appropriate method to use among the three for a given workload. However, as discussed later in Section 5.4, this approach offers no consistent improvement over BaCon on the query workloads found in the learned CE literature. This observation speaks to the robustness of BaCon for its intended use.

Before delving into details, we note that BaCon takes the high-level approach of leveraging DBMS for processing, not only because it houses the data, but also because it already provides versatile support for indexing and querying. To make BaCon easy to adopt,

we do not modify any DBMS internals but instead implement, in a client application, parts of the algorithm that are inefficient for the DBMS to handle. Many optimizations thus involve balancing database- and client-side processing options and mitigating the overhead of interfacing them. While BaCon is currently implemented on top of PostgreSQL, we believe that ideas in this section generalize to other DBMSs, although some details may differ.

## 4.1 Choosing Execution Plan for a Join Pattern

Give a join pattern $\mathfrak{J}$, the cost of BaConRecurse to process $Q[\mathfrak{J}]$ depends on the choice of a tree-based execution plan, introduced in Section 3.4. A cost-based decision requires fine-grained knowledge of the data distribution, which itself is expensive to acquire. Instead, we follow some heuristics below to construct a tree-based plan.

The first heuristic is to pick the root of the tree to be the table $R$ with the highest degree in $\mathfrak{J}$'s join graph: i.e., the root joins with the largest number of other tables in $\mathsf{Tables}_{\mathfrak{J}}$ according to $\mathsf{JPreds}_{\mathfrak{J}}$. The rationale is to maximize the saving opportunities identified by Lemma 3.4: a root with a large in-degree can potentially avoid enumerating more cross products. Once we pick the root, the rest of the tree structure follows naturally from $\mathsf{JPreds}_{\mathfrak{J}}$. Since we assume queries to be acyclic, no additional join conditions are needed beyond those corresponding to the tree edges. For example, in Figure 1, even though `mc` and `ci` can join on `mc.movie_id = ci.movie_id`, there is no edge between them in the tree, because the two join conditions from root `t` to `mc` and `ci`, involving `t.id`, together imply `mc.movie_id = ci.movie_id`.

The next optimization then marks certain tree edges that correspond to joining a foreign key (FK) of the parent table $R$ to a primary key (PK) of the child table $R'$. For such a marked FK-PK edge, when BaConRecurse processes $R$, it will simultaneously process $R'$: on Line 2, ProcessTable will additionally query $R'$ and compute its count map along with that for $R$ (with appropriate changes to Listing 3); and on Line 4, children of $R'$ will be directly included, instead of $R'$ itself. The justification behind this optimization is that, because of the FK-PK join, each $v$ would yield only one joining tuple from $R'$, meaning there is no cross product to avoid in the first place. Therefore, we should "short-cut" this edge to eliminate the associated overhead.[2] For simplicity, our implementation currently restricts this optimization to edges incident to the root, but applying it to the entire tree is possible as future work.

The last heuristic dynamically reorders subtrees under each node. We start with an arbitrary ordering. Given a node $R$, once we complete some iterations of the loop on Lines 2–10 of Algorithm 2, we count, for each child $R_i$, how many settings of $v$ yield no tuples from $R_i$. Then, we reorder the children so that those with more settings that yield no tuples come first. Intuitively, they are more likely to produce an empty count map, with which we can "short-circuit" the sequence of multiplies.

## 4.2 Mitigating SQL Querying Overhead

*Server- vs. Client-Side Cursors.* When BaCon runs a SQL query on the underlying database, we have the choice of using either a server-side cursor [61] or a client-side one [59]. A client-side cursor simply fetches the entire query result set into client memory. A server-side cursor, in contrast, allows the client to step through the result set row-at-a-time, without having to fetch and materialize a potentially massive result set. While server-side cursors offer more flexibility and make client applications scalable, they are considerably slower than client-side cursors. Since they require state to be maintained on the database server, a large number of concurrent open server-side cursors can also cause performance issues. Therefore, BaCon uses a server-side cursor for ProcessTable on the root table, but switches to client-side cursors for all other ProcessTable calls. The rationale is that the call on the root table (from Line 5 of Algorithm 1) has no bound attributes ($u = \emptyset$) and may generate a large result set. In contrast, other calls access much fewer tuples that are relevant to each specific subslice; even with the batching optimization below, the result set sizes remain manageable for client-side cursors.

*Batching of ProcessTable.* During the course of executing BaConRecurse recurisvely over the plan tree, the overhead of issuing many SQL queries through ProcessTable can quickly add up, especially when many of these queries, with specific attribute bindings, return small result sets. To mitigate this overhead, we combine multiple invocations of ProcessTable for the same table $R$, with a collection of attribute bindings $U$ rather than a single binding $u$, into a single SQL query. Specifying the combined condition precisely in the `WHERE` clause of Listing 3 is infeasible in general, as the query would have to enumerate all bindings in $U$. Instead, we compute the minimum and maximum elements[3] of $U$ and use $[\min(U), \max(U)]$ as a safe but possibly imprecise range bound in `WHERE`, i.e., `WHERE` $(J_1^{\mathsf{in}}, J_2^{\mathsf{in}}, \ldots)$ `BETWEEN` $\min(U)$ `AND` $\max(U)$. We then extend the query to additionally group by $J_1^{\mathsf{in}}, J_2^{\mathsf{in}}, \ldots$ and return them, such that ProcessTable can, using these attribute values, filter out result tuples that do not belong to $U$.

In BaConRecurse, we implement this optimization by collecting $\beta$ consecutive settings of $v$ into a set $V$ for processing as a batch, analogous to a $\beta$-fold unrolling of the loop on Lines 2–10 of Algorithm 2. For each child table $R_i$, we project $V$ to obtain the collection $U_i$ of unique bindings for $R_i$'s join attributes with $R$ (Line 5), and then invoke ProcessTable on $R_i$ with the entire $U_i$. The detailed algorithm is presented in [38].

We now provide some further analysis of this batching optimization. First, besides reducing the overhead of issuing many queries, batching also promotes reuse of ProcessTable calls on the same table and attribute binding. Without batching, consider the settings of $v$ in consecutive iterations of the loop on Lines 2–10 of Algorithm 2: it is likely that they agree on most components since they are sorted. Therefore, when $v$ is projected to obtain some $u_i$, the same $u_i$ setting in the previous iteration may be used again to call ProcessTable on the same table $R_i$, leading to waste unless we cache the results from previous calls. With batching, however, such reuse occurs automatically. For this reason, we have not implemented additional caching mechanisms for BaCon, although further exploration will be a good direction of future work.

Second, it may seem that compressing a set of bindings into a single range can introduce too many false positives. Indeed, it is always possible for $R_i$ to contain some tuple whose join attribute

[2] Note that the direction is important: a PK-FK edge, where the parent table has the primary key, would be a bad candidate to "short-cut," with an opposite argument.

[3] If each binding involves multiple attributes, we order the bindings lexicographically, attribute by attribute.

values lie in $[\min(U_i), \max(U_i)]$ but are not contained in $U_i$, because $U_i$ is generated from $V$ and subject to all other constraints on the slice. Nonetheless, the concern with false positives is at least partly alleviated by the fact that BaConRecurse processes settings of $v$ in order, meaning that it will call ProcessTable on $R_i$ with "clustered" settings of $u$ in $U$.

*Implementation of* `quantize`. Since the user-defined function `quantize` in Listing 3 is invoked many times, an efficient implementation is crucial to performance. We implement `quantize` as C-extensions in PostgreSQL [1], with one variant for each SQL data type. The quantization scale is implemented as an array of sorted boundary points, and a binary search is used to determine the correct bucket id. These C-extensions run as compiled code natively inside the server process, greatly outperforming SQL-based user-defined functions. BaCon creates these extensions at the beginning of its execution and drops them at the end.

## 4.3 Implementation of ComputeCounts

ComputeCounts is invoked by BaCon to compute the result counts of queries in $\mathcal{Q}[\mathfrak{J}]$ given the final count map $\mathfrak{M}$ computed by BaConRecurse. Conceptually, it performs a "join" between a set of query hyperrectangles ($\mathcal{Q}[\mathfrak{J}]$) and a set of weighted points ($\mathfrak{M}$), and reports the total weights within each hyperrectangle. Many processing strategies are possible, and the best depends on query and data distributions. For our target of query workloads for training learned CE models, we observe that $|\mathcal{Q}[\mathfrak{J}]|$ is moderate (in hundreds or thousands) and $\mathfrak{M}$ tends to be sparse (32% on average across join patterns in `synthetic.sql`). After experimenting with several methods, we have found a simple method based on nested loops to be the most effective for this setting. Basically, for each $Q \in \mathcal{Q}[\mathfrak{J}]$ and for each non-zero entry $\vec{b} \mapsto c$ of $\mathfrak{M}$, we check whether, for every dimension of $Q$'s hyperrectangle with a range bound, the corresponding coordinate in $\vec{b}$ falls within this bound. If yes, we add the entry's count $c$ to $Q$'s running total. While BaCon is implemented in Python, we specifically use Numba's just-in-time compilation [2] for ComputeCounts. With the help from Numba's compiler optimizations, this simple implementation was able to beat more sophisticated methods.

## 4.4 Choosing among BaCon/IndProc/PostFilt

We introduce an approach called *Hybrid* for picking the appropriate method to use among the three for a given query pattern. As we will see in Section 5.4, BaCon performs well across our target workloads, so Hybrid only serves to validate the robustness of BaCon. Nonetheless, we briefly describe Hybrid here for completeness.

We train Hybrid from sample workloads as a classifier that predicts, given a join pattern, one of three classes $\{0, 1, 2\}$ representing BaCon, IndProc, and PostFilt respectively. We use a lightweight model based on `RandomForestClassifier` [56] over a suite of features — the most important of which are illustrated in Figure 4. Features with prefixes `sum_`, `max_`, `mean_`, and `prod_` aggregate a collection of values: for example, `prod_number_of_bucket` is the product of the number of buckets across all quantization scales for a join pattern; `mean_query_coverage_ratio` is the average, taken per query, of the ratio between the number of grid cells queried and `prod_number_of_bucket`. Features containing `intermediate_join_size` are AGM bounds [6, 16]

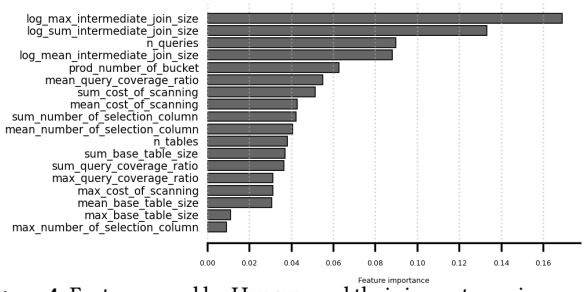


**Figure 4:** Features used by Hybrid, and their importance in a model trained as described in Section 5.4.

(i.e., overestimation) of intermediate result sizes of join subplans chosen by PostgreSQL for joining all tables in the join pattern. Features with `cost_of_scanning` denote PostgresSQL's estimated cost for enumerating a table ordered by join attributes that connects it to its descendant tables (if any), reflecting a key component of BaCon. In general, we include as features per-join-pattern statistics that are inexpensive to collect at runtime. However, we exclude more advanced selectivity estimates provided by PostgreSQL, because we have found them to be unreliable and systematically underestimating the costs of baselines.

To ensure a lightweight model and fast inference, we train our model with hyperparameters `train_val_split=0.25`, `n_estimators=100`, `min_sasmples_leaf=2`, and `class_weight={"0": 1, "1": 2, "2": 2}`. The higher weights for classes 1 and 2 penalize mistakenly picking baseline approaches, because such mistakes cost much higher. To obtain class labels for training data, we measure and compare the running time for all three methods. We deem a baseline method *safe* if its running time is either (i) at least 1.5× faster than BaCon, or (ii) no less than 10 seconds faster than BaCon. We assign class 1 only when IndProc is safe and PostFilt is either unsafe or slower; class 2 is assigned analogously; all remaining cases are labeled as class 0. For inference, rather than selecting the class with the highest posterior probability, we require the predicted probability of choosing IndProc or PostFilt to exceed a precision threshold (0.9 by default), calibrated to guarantee minimum precision on held-out validation data. If no probability threshold can guarantee the required precision, the one with highest precision is selected. This robust thresholding strategy deliberately trades recall for safety, selecting baselines only under high confidence.

# 5 EXPERIMENTS

We compare IndProc, PostFilt, and BaCon. All algorithms are implemented in Python3, using `psycopg2` [60] to connect to PostgreSQL V16.8 with default configuration. All experiments are conducted on a Linux server with an Intel(R) Xeon(R) Gold 5215 CPU (40 logical cores, 2.50GHz) and 1TB of disk storage. We focus on single-threaded execution: parallelism is disabled in PostgreSQL via `SET max_parallel_workers_per_gather = 0`, and all client-side code is likewise restricted to a single thread. We set $\beta$ to 50,000 for BaCon for all experiments and present results for different $\beta$ in [38]. The codes are available in [39]

We test three popular datasets: IMDB [33], STATS [17], and DSB [10] (of scaling factor 2). They are loaded and initialized using

**Table 2:** End-to-end running times (in seconds, rounded) and relative speedups. +: lower-bound values, as IndProc is capped at 900s per query and PostFilt at 3,600s per join pattern.

| Query Workload | IndProc | PostFilt | BaCon | Speedup vs. IndProc |
|---|---|---|---|---|
| synthetic | 19,663 | 11,329+ | 1,574 | ↑ 12.49 × |
| scale | 21,100+ | 26,342+ | 3,475 | ↑ 6.07+ × |
| job-light | 1,620+ | 13,662+ | 728 | ↑ 2.22+ × |
| job-light-single | 249 | 13 | 11 | ↑ 22.50 × |
| job-light-join | 8,601+ | 18,353+ | 1,541 | ↑ 5.58+ × |
| stats-ceb | 4,274+ | 47,353+ | 58 | ↑ 73.62+ × |
| stats-ceb-single | 18 | 2 | 2 | ↑ 8.70 × |
| stats-ceb-join | 35,292+ | 53,363+ | 198 | ↑ 178.14+ × |
| dsb-grasp-20k | 19,746 | 4,211 | 928 | ↑ 21.28 × |

**Table 3:** Number of timed-out cases. Q is query for short and JP is join pattern. In $(a, b)$, $a$ is the number of timed-out queries, and $b$ is the number of join patterns that contain any timed-out queries.

| Query Workload | # of Qs | # of JPs | IndProc | PostFilt |
|---|---|---|---|---|
| synthetic | 5,000 | 16 | (0, 0) | (251, 1) |
| scale | 500 | 31 | (20, 7) | (84, 6) |
| job-light | 70 | 18 | (1, 1) | (8, 2) |
| job-light-single | 254 | 1 | (0, 0) | (0, 0) |
| job-light-join | 696 | 27 | (3, 1) | (32, 3) |
| stats-ceb | 146 | 58 | (3, 2) | (30, 11) |
| stats-ceb-single | 632 | 1 | (0, 0) | (0, 0) |
| stats-ceb-join | 2,603 | 120 | (22, 7) | (166, 13) |
| dsb-grasp-20k | 20,000 | 16 | (0, 0) | (0, 0) |

the scripts in [11, 20, 32], and the algorithms use exactly the indices provided therein without creating any additional ones.

We evaluate the algorithms on nine publicly available workloads in Table 1. These workloads span a wide range of sizes, query templates, join structures, predicate types, and result sizes, and are widely used for training and evaluating cardinality estimation techniques [17]. Among the workloads, only `dsb-grasp-20k` is post-processed (script in [40]) from the source file in [67]. Specifically, CE work, SeConCDF [68] and GRASP [66], have numerical/categorical definitions for several attributes that are originally defined as `char(...)` in DSB (e.g., `cd.cd_education_status`). As these transformations are not clearly specified, we remove the queries involving such attributes, and then randomly sample 20k queries to ensure that baselines complete within a reasonable time. We additionally construct three new workloads based on `job-light.sql`, used exclusively in Section 5.3.

Running time is our main performance metric. For IndProc, end-to-end running time is the sum over all queries. For PostFilt and BaCon, it includes pre-processing and time for each join pattern. To prevent baselines from running for too long, we cap IndProc at 900s per query, and PostFilt at 3,600s per join pattern; therefore, some baseline numbers are only lower bounds.

## 5.1 End-to-End Running Time

Table 2 reports the end-to-end running times of all three approaches across all nine real workloads. Lower-bound values (i.e., those followed by +) are reported if a baseline runs out of time. We additionally show the relative speedups of BaCon over IndProc, which range from approximately 2× to 178×. Speedups over PostFilt are omitted, as IndProc outperforms PostFilt on most long-running workloads. Overall, BaCon consistently outperforms both baselines and never "times out": the worst per-join-pattern overhead of BaCon across all workloads is below 380s, far below the thresholds for baselines. The observed performance trends between IndProc and PostFilt align with our analysis in Section 2. In particular, PostFilt is advantageous for join patterns with many queries — where predicates tend to cover larger regions — and without intermediate join blow-ups (e.g., `synthetic.sql` and `dsb-grasp-20k.sql`). In the remaining cases, IndProc performs substantially better due to selection push-down in each query. BaCon is able to effectively combine the strengths of both IndProc and PostFilt.

Since both PostFilt and BaCon are designed around the notion of join patterns, we further analyze the results on a per-join-pattern basis in Section 5.2. Before doing so, we briefly summarize the other overheads — which are negligible compared to per-join-pattern running time — and omit them from ensuing discussions. IndProc incurs no such overheads. Both PostFilt and BaCon require loading and parsing SQL queries, extracting join patterns, and grouping queries accordingly, with complexity linear in the number of queries, tables, and selection attributes. This pre-processing step happens at the beginning before processing each join pattern, and typically completes within a few seconds; the only exception is `dsb-grasp-20k.sql`, where it reaches approximately 20s due to the large number of queries (but is still small relative to the end-to-end running time).

## 5.2 Per-Join-Pattern Result

In this section, we analyze the results on a per-join-pattern basis. Although IndProc does not operate on join patterns, we group its results accordingly to facilitate direct comparison with the others.

We first report the number of timed-out cases in Table 3. BaCon has no timed-out cases. Overall, PostFilt times out more frequently: since it always materializes the unfiltered join, if that times out, all queries associated with the pattern time out. Additionally, workloads on STATS exhibit more time-outs, primarily due to larger intermediate joins than those on IMDB, as noted in [17].

Next, we analyze the per-join-pattern overhead across all workloads. For clarity, consistent with the pre-defined time-out thresholds, we assign 3,600s to timed-out join patterns in PostFilt and 900s to timed-out queries in IndProc; actual running times will be higher. Figure 5 reports the running times of all join patterns, ordered by IndProc's times. Before index 120, all join patterns under IndProc complete within 10s; beyond this point, running times increase markedly. The inset zooms in the region before index 200, where BaCon closely tracks IndProc, sometimes faster and sometimes slower within a bounded margin, while PostFilt already times out on several patterns. Between indices 160 and 200, BaCon exhibits a small number of clear regressions relative to IndProc, with the largest gap (98s) at index 165; we analyze these cases in detail later. Outside the zoomed region, where join patterns run substantially longer, BaCon consistently and significantly outperforms IndProc and PostFilt.

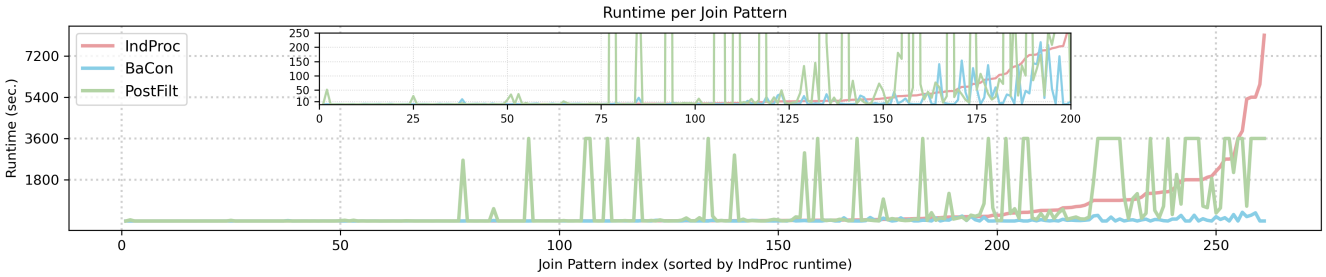


**Figure 5:** Running time (sec.) per join pattern across 283 join patterns from 9 workloads. Join patterns are ordered by IndProc's times. The inset is the zoom-in of the first 200 indices.

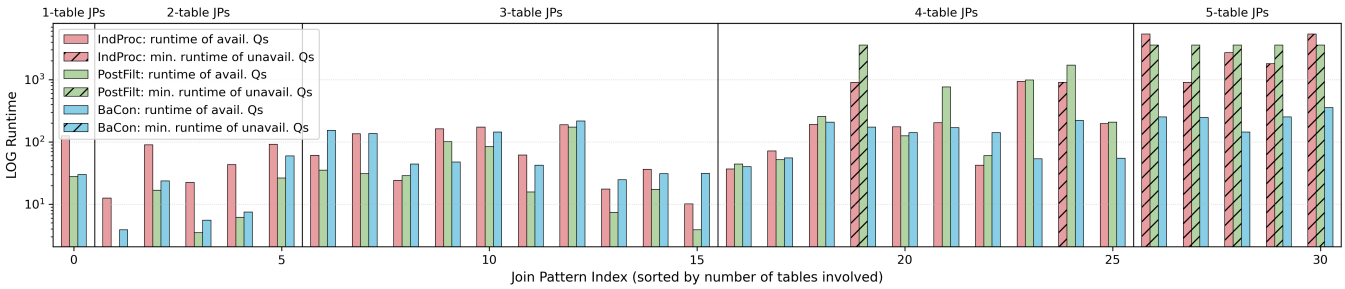


**Figure 6:** Log-scale running time per join pattern in `scale.sql`. Join patterns are ordered by the number of tables involved, and the figure is partitioned accordingly (labels shown above). Hatched bars (baselines only) indicate the lower bounds for timed-out cases.

Additionally, we present a detailed per-join-pattern analysis for the `scale.sql` and `job-light.sql` workloads; results for the remaining workloads are in [38]. As shown in Figure 6, `scale.sql` contains join patterns involving 1 to 5 tables, with more tables generally run longer. Timed-out queries arise mainly from patterns with many tables. For most long-running join patterns, BaCon consistently outperforms the baselines. For fast patterns, BaCon remains competitive, as the time gaps are small. However, there are several patterns where BaCon is noticeably slower than either baseline. Taking join pattern 6 as an example, IndProc, PostFilt, and BaCon take 61s, 35s, and 153s, respectively. This pattern has conditions `ci.movie_id = t.id AND mc.movie_id = t.id` and only five queries. A breakdown of BaCon's running time shows 118s (76%) for executing SQLs, 5s (3%) for aggregating rows by join combos and coordinates, and 18s (15%) for merging count maps from descendants; thus, SQL execution dominates and already exceeds the running times of the baselines. This regression stems from two factors. First, algorithmically, enumerating tuples from each table and quantizing them can be less efficient than IndProc when predicates are selective and overlaps are limited. Second, even though BaCon perform less computation than PostFilt, BaCon's overhead in repeated cursor invocations and row fetching/parsing can outweigh PostFilt's server-side processing cost.

We have similar observations from Figure 7. As discussed in [17], the `job-light.sql` is a classical but relatively simple workload with 1 to 8 queries for each join pattern. The true cardinality range is also a magnitude smaller than that of `stats-ceb.sql`. Therefore, IndProc consistently performs well on all join patterns, except

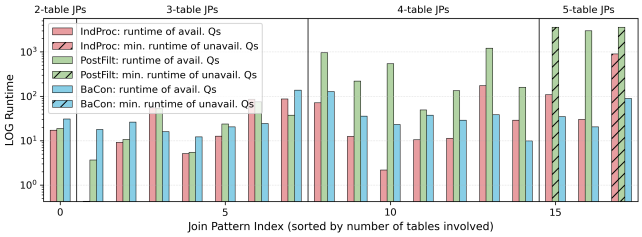


**Figure 7:** Log-scale running time per join pattern in `job-light.sql`, with same format as Figure 6.

one severe time-out for join pattern 17. However, BaCon remains competitive and robust (never timing out) across all join patterns, with limited regressions[4] compared to both baselines.

Overall, consistent with the global results in Section 5.1, BaCon performs robustly across all join patterns: it significantly outperforms the baselines on long-running join patterns and remains acceptable on fast ones.

## 5.3 Scalability

In this section, we evaluate the scalability of the three approaches using three synthetic workloads, denoted `job-light-*.sql`. We choose `job-light.sql` as the reference because, as shown in Figure 7, IndProc performs consistently well, except for a single join

[4]In the worst case, BaCon incurs an additional 55s compared to IndProc on join pattern 8, and 100s compared to PostFilt on join pattern 7.

**Table 4:** `job-light-*.sql`: end-to-end running times (in seconds, rounded) and relative speedups. +: lower-bound values, as IndProc is capped at 900s per query and PostFilt at 3,600s per join pattern.

| Query Workload | IndProc | PostFilt | BaCon | Speedup vs. IndProc |
|---|---|---|---|---|
| job-light-1k | 6,208 | 22,490+ | 1,020 | ↑ 6.09 × |
| job-light-2k | 13,818 | 26,923+ | 1,051 | ↑ 13.15 × |
| job-light-4k | 23,617+ | 33,981+ | 1,109 | ↑ 21.29+ × |

pattern that times out. Specifically, on `jog-light.sql`, BaCon outperforms IndProc in only 7 out of the 18 (39%) join patterns. Earlier, we have made the observation that the baselines — especially IndProc — can outperform BaCon on simple join patterns, e.g., when the number of queries is small and intermediate joins do not blow up. Here, we further investigate how each approach scales as the workload size increases while preserving the original distribution.

We construct workloads with 1,000, 2,000, and 4,000 queries each. For each workload, queries are generated independently as follows: (i) randomly select a query from `job-light.sql` and reuse its template (and thus its join pattern); (ii) randomly modify predicate constants according to rules (iii)-(iv); (iii) for equality and inequality predicates, replace the constant with a random value drawn from the active domain of the corresponding attribute; (iv) for range predicates, randomize the left boundary and adjust the right accordingly, to keep the range length the same.

Table 4 shows end-to-end running times and relative speedups. As expected, IndProc's running time grows roughly linearly with workload size, since queries are processed independently. PostFilt grows more slowly because its dominant cost — computing the full join — is amortized across queries with the same join pattern; nevertheless, it still times out frequently. BaCon scales better for two reasons. First, quantization generally benefits larger workloads more. Second, the costs of enumerating join attribute bindings and quantization remain mostly stable: as the size of each quantization scale is bounded by the workload size $|Q[\mathfrak{J}]|$, the cost of quantizing an attribute is only $O(\log(|Q[\mathfrak{J}]|))$ thanks to binary search. Per-join-pattern results are provided in [38]. While the number of join patterns remains 18, the subset where BaCon outperforms IndProc increases to 14 (78%) in `job-light-1k.sql`, to 17 (94%) in `job-light-2k.sql`, and finally to 18 (100%) in `job-light-4k.sql`. Overall, BaCon demonstrates superior scalability compared to both IndProc and PostFilt.

## 5.4 Validation against Hybrid

As shown in Section 5.2, there exist join patterns for which a baseline, IndProc or PostFilt, outperforms BaCon. These cases typically involve few queries and small intermediate join sizes. As discussed in Section 4.4, we can adopt a hybrid approach that uses a classifier to pick which method to use given a join pattern. Here, we train Hybrid using observations from `job_light_join.sql` and `stats_ceb_join.sql`, and evaluate it on the remaining seven workloads. Table 5 reports the E2E running times of Hybrid (using BaCon as the reference) on the seven test workloads. Overall, Hybrid selects a baseline for 9 join patterns in `scale.sql` and 4 join patterns in `stats_ceb.sql`, while selecting BaCon for all

**Table 5:** End-to-end (E2E) running times for BaCon and Hybrid, with the number of join patterns (JPs) handled by each algorithm. I/P/B denotes IndProc/PostFilt/BaCon.

| Query Workload | BaCon E2E running time | Hybrid E2E running time | # of JPs using I/P/B |
|---|---|---|---|
| scale | 3,475 | 3,319 | 0/9/22 |
| stats-ceb | 58 | 195 | 3/1/54 |
| other 5 | - | - | 0/0/all |

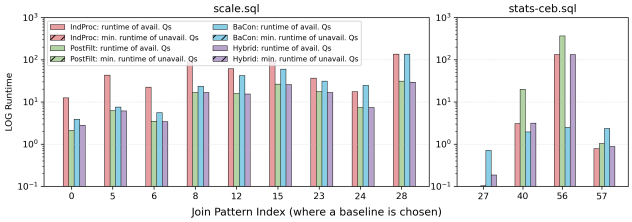


**Figure 8:** Log-scale running time per join pattern (where a baseline is chosen) in `scale.sql` and `stats_ceb.sql`, including Hybrid's.

other join patterns and workloads. Even with Hybrid's conservative design, end-to-end running times show that mispredictions can occur, sometimes catastrophically. To take a closer look, Figure 8 presents a per-join-pattern analysis, focusing on patterns where Hybrid chooses a baseline. As shown in Figure 8, for `scale.sql`, all 9 switches to PostFilt are beneficial, yielding a total saving of 156s. In `stats_ceb.sql`, Hybrid gains minor savings by choosing PostFilt on pattern 27 and IndProc on pattern 57, but incorrectly selecting IndProc on patterns 40 and 56 causes slowdowns of 1s and 129s, respectively. Overall, these results confirm that BaCon remains a safe bet for practical workloads. With additional data statistics or training, a better Hybrid might be possible as future work, but the additional overhead for statistics collection and training may outweigh its benefit.

# 6 RELATED WORK

*Select-Join-Aggregate Query Processing.* Counting the join results (with selection predicate on the base tables) is a special class of select-join-aggregate queries (intuitively, applying an aggregate function on top of the select-join queries) with the output size as 1. Processing such queries usually first push down the selection predicate to the base table, and ends up with processing join-count queries, which is the focus below.

The previous works have achieved two flavors of results: worst-case optimal and output-sensitive. Namely, worst-case optimal algorithms work only on pathological instances with huge outputs, which are rare in practice. In contrast, output-sensitive algorithms express the runtime as a function of the input size and output size, which are more practically meaningful, especially for queries where the aggregation (such as count) may significantly reduce the output size. The classical Yannakakis algorithm [70] can compute *free-connex* join-count queries in $O(N)$ time, where $N$ is the total number of tuples in the database. Very recently, [24] exploit the hybrid strategy of Yannakakis algorithms, which can compute acyclic

join-count queries in $O(N \cdot \text{OUT}^{1-\frac{1}{\text{fnfhtw}}} + \text{OUT})$ time, where OUT is the size of the projection of the join results onto the counting columns, and fnfhtw is the free-connex fractional hypertree width of the query. For general join-count queries, the state-of-the-art approach is to convert the query into a free-connex one using the tree decomposition technique and the worst-case optimal join algorithm [47–49, 62], and then run the hybrid Yannakakis algorithm on the tree decomposition.

Our work is also inspired by some ideas in these works. The instantiations of worst-case optimal join algorithms [3, 14, 44, 63] motivate BaCon to decompose computation via child recursions and combine multiple coordinate-to-count maps during backtracking. Cost-based optimizers in these systems, such as Free Join's [63], together with the AGM bound [6, 16], further inform our initial exploration of cost-based hybrid mechanism. However, achieving more accurate cost estimates for BaCon may conversely require more accurate (pessimistic) cardinality estimation at a higher cost, such as [71], and/or specialized data storages and execution engines, such as Free Join's COLT [63] and its integration to DuckDB [52].

*Aggregation Push-down.* Early works explore aggregation push-down in query plans to reduce intermediate result sizes. Eager aggregation and lazy aggregation [69] introduce rules for early or deferred partial aggregation based on algebraic properties, while [9] integrates `GROUP BY` into cost-based optimization; both focus on intra-query optimization via query rewriting. More recently, GuAo [31] identifies aggregate queries that can be evaluated without materializing any joins and introduces a corresponding physical operator for SparkSQL [5]. GuAo targets single-query execution and assumes selection push-down at table scans. For the case without `GROUP BY`, its propagation of frequencies grouped by join attributes resembles BaCon's bucketization when reduced to a single-query join pattern. Unlike BaCon, however, GuAo merges frequencies via standard binary joins in SparkSQL rather than supporting multi-way combination during count-map merging.

*Factorizations.* Factorized databases (FDBs) propose the factorized representations of query results [8] to eliminate tuple-level redundancy and boost the relational processing performance [50], supporting aggregations beyond `COUNT` [7]. These works focus on intra-query redundancy in query output and are different but complementary to our approach. Moreover, they assume storage layout different from traditional relational databases, like PostgreSQL.

*Optimized Batch Processing of Aggregate Queries.* LMFAO is a framework proposed by [55] to efficiently compute a batch of group-by aggregates over shared joins without fully materializing intermediate results. It decomposes queries into views, groups them according to a join tree, and evaluates each view group using a multi-output plan. Similar to PostFilt, LMFAO supports selection predicates by encoding them as conditional expressions inside aggregate functions. In contrast to BaCon, LMFAO does not analyze or pre-process overlaps among cross-query predicates (e.g., via bucket construction), but instead focuses on sharing computation at the level of join tree traversal and attribute-ordered evaluation, reusing identical conditional expressions and partial products within a view group. Moreover, LMFAO is implemented as a standalone engine, rather than being integrated into a traditional DBMS.

*Multi-Query Optimization & Shared Workload Optimization.* Multi-query optimization (MQO) [58] identifies shared or similar subexpressions across queries and performs inter-query optimization by executing the common subexpressions jointly or by rewriting query plans with a boarder subquery. Along this line, Zhou et al. [72] propose a practical common subexpression manager integrated into Microsoft SQL Server and its cost-based query optimizer, demonstrating improvements on workloads of a few tens of queries. As the search space for identifying common subplans grow rapidly with the number of queries, Shared Workload Optimization (SWO) [15] optimizes an entire workload – often comprising hundreds or thousands of concurrent queries – by identifying shared operators instead. This typically requires specialized or substantially modified execution engines (e.g., shared work systems) to support shared physical operators and coordinated execution. Both lines of work target more general queries and require accurate cost estimates, which motivates future extensions of BaCon toward tighter, cost-based integration with DBMSs.

*Continuous Query Processing.* Related works on continuous query processing [4, 41] in stream systems explore shared computations by maintaining data summaries or predicate indices over evolving streams. These approaches are designed for workloads with large numbers of continuous queries or filters (e.g., on the order of 100k), usually exceeding the workload sizes considered in our setting, and rely on specialized structures to efficiently support updates. Nevertheless, they offer complementary insights that could inform strategies for handling workloads at different scales as future work.

## 7 CONCLUSION

In this paper, we presented BaCon, a method for efficient batch processing of counting queries. BaCon brings together multiple optimization ideas, focusing particularly on developing compact, alternative representations of intermediate results that enable fine-grained sharing of computation. BaCon combines lightweight SQL execution with client-side processing, incorporating a suite of optimizations to ensure practical performance on our target workloads.

A strength of BaCon is its practicality: it requires no modifications to DBMS internals or specialized physical designs, relying primarily on query rewriting, and can be deployed directly on top of existing DBMSs. Our results demonstrate significant performance gains on real workloads and robustness across diverse workload characteristics. This level of improvement is empowering: shorter (re)training times make learned CE more practical and allow the use of larger, more comprehensive training workloads.

BaCon opens up several promising directions for future work. Extending it to support richer join structures, more expressive selection predicates, and additional aggregate functions beyond `COUNT` would further broaden its applicability. Beyond functional extensions, there are opportunities for optimization, such as sharing computation across join patterns; delayed evaluation of $\oplus$ and $\otimes$ in count map expressions (and adapting ComputeCounts to take advantage); more intelligent selection of tree-based execution plans; aggressive "short-cutting" of all FK-PK edges; caching and parallelization. Finally, new applications and additional optimizations will likely require models beyond the current Hybrid to enable cost-based selection of processing methods.

# REFERENCES


[1] PostgreSQL 17. 2025. Extending SQL: C-Language Functions. https://www.postgresql.org/docs/current/xfunc-c.html

[2] Inc. 2012-2020, Anaconda et al. 2025. Numba - a just-in-time compiler for Python that works best on code that uses NumPy arrays and functions, and loops. https://numba.pydata.org/numba-doc/dev/index.html#

[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] Pankaj K. Agarwal, Junyi Xie, Jun Yang, and Hai Yu. 2006. Scalable continuous query processing by tracking hotspots. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 31–42.

[5] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

[6] Albert Atserias, Martin Grohe, and Dániel Marx. 2017. Size bounds and query plans for relational joins. arXiv:1711.03860 [cs.DB] https://arxiv.org/abs/1711.03860

[7] Nurzhan Bakibayev, Tomáš Kočiský, Dan Olteanu, and Jakub Závodný. 2013. Aggregation and ordering in factorised databases. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1990–2001. https://doi.org/10.14778/2556549.2556579

[8] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. 2012. FDB: a query engine for factorised relational databases. *Proc. VLDB Endow.* 5, 11 (July 2012), 1232–1243. https://doi.org/10.14778/2350229.2350242

[9] Surajit Chaudhuri and Kyuseok Shim. 1994. Including Group-By in Query Optimization. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 354–366.

[10] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: a decision support benchmark for workload-driven and traditional database systems. *Proc. VLDB Endow.* 14, 13 (Sept. 2021), 3376–3388. https://doi.org/10.14778/3484224.3484234

[11] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB Initialization Files and Scripts. https://github.com/microsoft/dsb/tree/main/scripts

[12] Lyric Doshi, Vincent Zhuang, Gaurav Jain, Ryan Marcus, Haoyu Huang, Deniz Altinbüken, Eugene Brevdo, and Campbell Fraser. 2023. Kepler: Robust Learning for Parametric Query Optimization. *Proc. ACM Manag. Data* 1, 1, Article 109 (May 2023), 25 pages. https://doi.org/10.1145/3588963

[13] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *Proc. VLDB Endow.* 12, 9 (May 2019), 1044–1057. https://doi.org/10.14778/3329772.3329780

[14] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.* 13, 12 (July 2020), 1891–1904. https://doi.org/10.14778/3407790.3407797

[15] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. 2014. Shared workload optimization. *Proc. VLDB Endow.* 7, 6 (Feb. 2014), 429–440. https://doi.org/10.14778/2732279.2732280

[16] Georg Gottlob, Stephanie Tien Lee, Gregory Valiant, and Paul Valiant. 2012. Size and Treewidth Bounds for Conjunctive Queries. *J. ACM* 59, 3, Article 16 (June 2012), 35 pages. https://doi.org/10.1145/2220357.2220363

[17] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality estimation in DBMS: a comprehensive benchmark evaluation. *Proc. VLDB Endow.* 15, 4 (Dec. 2021), 752–765. https://doi.org/10.14778/3503585.3503586

[18] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2023. job-light-join.sql. https://github.com/Nathaniel-Han/End-to-End-CardEst-Benchmark/blob/master/workloads/job-light/sub_plan_queries/job_light_sub_query.sql

[19] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2023. job-light-single.sql. https://github.com/Nathaniel-Han/End-to-End-CardEst-Benchmark/blob/master/workloads/job-light/sub_plan_queries/job_light_single_table_sub_query.sql

[20] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2023. STATS Initialization Files and Scripts. https://github.com/Nathaniel-Han/End-to-End-CardEst-Benchmark/tree/master/scripts/sql

[21] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2023. stats_ceb_join.sql. https://github.com/Nathaniel-Han/End-to-End-CardEst-Benchmark/blob/master/workloads/stats_CEB/sub_plan_queries/stats_CEB_sub_queries.sql

[22] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2023. stats_ceb_single.sql. https://github.com/Nathaniel-Han/End-to-End-CardEst-Benchmark/blob/master/workloads/stats_CEB/sub_plan_queries/stats_CEB_single_table_sub_query.sql

[23] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2023. stats_ceb.sql. https://github.com/Nathaniel-Han/End-to-End-CardEst-Benchmark/blob/master/workloads/stats_CEB/stats_CEB.sql

[24] 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

[25] Xiao Hu, Yuxi Liu, Haibo Xiu, Pankaj K. Agarwal, Debmalya Panigrahi, Sudeepa Roy, and Jun Yang. 2022. Selectivity Functions of Range Queries are Learnable. In *SIGMOD* (Philadelphia, PA, USA) *(SIGMOD '22)*. 959–972.

[26] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. job-light.sql. https://github.com/andreaskipf/learnedcardinalities/blob/master/workloads/job-light.sql

[27] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[28] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. scale.sql. https://github.com/andreaskipf/learnedcardinalities/blob/master/workloads/scale.sql

[29] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. synthetic.sql. https://github.com/andreaskipf/learnedcardinalities/blob/master/workloads/synthetic.sql

[30] Meghdad Kurmanji and Peter Triantafillou. 2023. Detect, Distill and Update: Learned DB Systems Facing Out of Distribution Data. *Proc. ACM Manag. Data* 1, 1, Article 33 (May 2023), 27 pages. https://doi.org/10.1145/3588713

[31] Matthias Lanzinger, Reinhard Pichler, and Alexander Selzer. 2025. Avoiding Materialisation for Guarded Aggregate Queries. *Proc. VLDB Endow.* 18, 5 (Jan. 2025), 1398–1411. https://doi.org/10.14778/3718057.3718068

[32] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2013-2019. IMDB Initialization Files and Scripts. https://event.cwi.nl/da/job/

[33] 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 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[34] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. IMDB Relational Schema by JOB. https://event.cwi.nl/da/job/

[35] Beibin Li, Yao Lu, and Srikanth Kandula. 2022. Warper: Efficiently Adapting Learned Cardinality Estimators to Data and Workload Drifts. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1920–1933. https://doi.org/10.1145/3514221.3526179

[36] Jie Liu, Wenqian Dong, Qingqing Zhou, and Dong Li. 2021. Fauce: fast and accurate deep ensembles with uncertainty for cardinality estimation. *Proc. VLDB Endow.* 14, 11 (July 2021), 1950–1963. https://doi.org/10.14778/3476249.3476254

[37] Yuxi Liu, Xiao Hu, Pankaj Agarwal, and Jun Yang. 2026. dsb_grasp_20k.sql. https://github.com/louisja1/bacon/blob/main/workload/dsb_grasp_20k.sql

[38] Yuxi Liu, Xiao Hu, Pankaj Agarwal, and Jun Yang. 2026. [Full Version] BaCon: Efficient Batch Processing of Counting Queries. https://github.com/louisja1/bacon/blob/main/fullversion.pdf

[39] Yuxi Liu, Xiao Hu, Pankaj Agarwal, and Jun Yang. 2026. Github Repository of BaCon. https://github.com/louisja1/bacon

[40] Yuxi Liu, Xiao Hu, Pankaj Agarwal, and Jun Yang. 2026. Script for generating dsb_grasp_20k.sql. https://github.com/louisja1/bacon/blob/main/workload/raw/dsb_grasp_csv_to_sql.py

[41] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. 2002. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/564691.564698

[42] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1275–1288. https://doi.org/10.1145/3448016.3452838

[43] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: a learned query optimizer. *Proc. VLDB Endow.* 12, 11 (July 2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[44] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.* 12, 11 (July 2019), 1692–1704. https://doi.org/10.14778/3342263.3342643

[45] Magnus Müller, Lucas Woltmann, and Wolfgang Lehner. 2023. Enhanced Featurization of Queries with Mixed Combinations of Predicates for ML-based Cardinality Estimation. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, Julia Stoyanovich, Jens Teubner, Nikos Mamoulis, Evaggelia Pitoura, Jan Mühlig, Katja Hose, Sourav S. Bhowmick, and Matteo Lissandrini (Eds.). OpenProceedings.org, 273–284. https://doi.org/10.48786/EDBT.2023.22

[46] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-loss: learning cardinality estimates that matter. *Proc. VLDB Endow.* 14, 11 (July 2021), 2019–2032. https://doi.org/10.14778/3476249.3476259

[47] Hung Q. Ngo. 2018. Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Houston, TX, USA) *(PODS '18)*. Association for Computing Machinery, New York, NY, USA, 111–124. https://doi.org/10.1145/3196959.3196990

[48] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case Optimal Join Algorithms. arXiv:1203.1952 [cs.DB] https://arxiv.org/abs/1203.1952

[49] 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

[50] Dan Olteanu and Jakub Zavodny. 2012. Factorised representations of query results: size bounds and readability. In *15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012*, Alin Deutsch (Ed.). ACM, 285–298. https://doi.org/10.1145/2274576.2274607

[51] Y. Park, S. Zhong, and B. Mozafari. 2020. Quicksel: Quick selectivity learning with mixture models. In *Proc. 39th ACM SIGMOD Int. Conf. Management Data,*. 1017–1033.

[52] Mark Raasveldt. 2022. DuckDB - A Modern Modular and Extensible Database System. In *CDMS@VLDB*. https://api.semanticscholar.org/CorpusID:252384081

[53] Silvan Reiner and Michael Grossniklaus. 2023. Sample-Efficient Cardinality Estimation Using Geometric Deep Learning. *Proc. VLDB Endow.* 17, 4 (Dec. 2023), 740–752. https://doi.org/10.14778/3636218.3636229

[54] Wolfgang Scheufele and Guido Moerkotte. 1997. On the complexity of generating optimal plans with cross products. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. 238–248.

[55] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. 2019. A Layered Aggregate Engine for Analytics Workloads. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1642–1659. https://doi.org/10.1145/3299869.3324961

[56] scikit-learn developers. 2007-2025. scikit-learn – RandomForestClassifier. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#randomforestclassifier

[57] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, Philip A. Bernstein (Ed.). ACM, 23–34. https://doi.org/10.1145/582095.582099

[58] Timos K. Sellis. 1988. Multiple-query optimization. *ACM Trans. Database Syst.* 13, 1 (March 1988), 23–52. https://doi.org/10.1145/42201.42203

[59] The Psycopg Team. 2001-2021. Psycopg – Client-side Cursors. https://www.psycopg.org/docs/cursor.html

[60] The Psycopg Team. 2001-2021. Psycopg – PostgreSQL database adapter for Python. https://www.psycopg.org/docs/#

[61] The Psycopg Team. 2001-2021. Psycopg – Server-side Cursors. https://www.psycopg.org/docs/usage.html#server-side-cursors

[62] Todd L. Veldhuizen. 2013. Leapfrog Triejoin: a worst-case optimal join algorithm. arXiv:1210.0481 [cs.DB] https://arxiv.org/abs/1210.0481

[63] Yisu Remy Wang, Max Willsey, and Dan Suciu. 2023. Free Join: Unifying Worst-Case Optimal and Traditional Joins. *Proc. ACM Manag. Data* 1, 2, Article 150 (June 2023), 23 pages. https://doi.org/10.1145/3589295

[64] Peizhi Wu and Gao Cong. 2021. A unified deep model of learning from both data and queries for cardinality estimation. In *Proceedings of the 2021 International Conference on Management of Data*. 2009–2022.

[65] Peizhi Wu and Zachary G. Ives. 2024. Modeling Shifting Workloads for Learned Database Systems. *Proc. ACM Manag. Data* 2, 1, Article 38 (March 2024), 27 pages. https://doi.org/10.1145/3639293

[66] Peizhi Wu, Rong Kang, Tieying Zhang, Jianjun Chen, Ryan Marcus, and Zachary G. Ives. 2025. Data-Agnostic Cardinality Learning from Imperfect Workloads. *Proc. VLDB Endow.* 18, 8 (April 2025), 2519–2532. https://doi.org/10.14778/3742728.3742745

[67] Peizhi Wu, Rong Kang, Tieying Zhang, Jianjun Chen, Ryan Marcus, and Zachary G. Ives. 2025. Original query workload of GRASP. https://github.com/shoupzwu/GRASP/blob/master/queries/dsb.csv

[68] Peizhi Wu, Haoshu Xu, Ryan Marcus, and Zachary G. Ives. 2025. A Practical Theory of Generalization in Selectivity Learning. *Proc. VLDB Endow.* 18, 6 (Feb. 2025), 1811–1824. https://doi.org/10.14778/3725688.3725708

[69] Weipeng P. Yan and Per-Åke Larson. 1995. Eager Aggregation and Lazy Aggregation. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 345–357.

[70] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7* (Cannes, France) *(VLDB '81)*. VLDB Endowment, 82–94.

[71] Haozhe Zhang, Christoph Mayer, Mahmoud Abo Khamis, Dan Olteanu, and Dan Suciu. 2025. LpBound: Pessimistic Cardinality Estimation Using $\ell_p$-Norms of Degree Sequences. *Proc. ACM Manag. Data* 3, 3 (2025), 184:1–184:27. https://doi.org/10.1145/3725321

[72] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. 2007. Efficient exploitation of similar subexpressions for query processing. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (Beijing, China) *(SIGMOD '07)*. Association for Computing Machinery, New York, NY, USA, 533–544. https://doi.org/10.1145/1247480.1247540

[73] Rong Zhu, Lianggui Weng, Bolin Ding, and Jingren Zhou. 2024. Learned Query Optimizer: What is New and What is Next. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 561–569. https://doi.org/10.1145/3626246.3654692

# A MORE DETAILS OF BASIC BACON

We present the detailed construction algorithm of $\mathfrak{b}_A$ for a selection attribute $A$ in Appendix A.1.

## A.1 Construction of Buckets

Continuing from Section 3.2, we extract from $\mathcal{Q}[\mathfrak{J}]$ the set $\Delta = \{\mathsf{SPreds}_Q(A) \mid \exists Q \in \mathcal{Q}[\mathfrak{J}] : A \in \mathsf{SPreds}_Q\}$ of predicate ranges associated with $A$. We transform each range $G$ specified by $\mathsf{SPreds}_Q(A)$ to a left-closed, right-open interval. Concretely, a left-open boundary $(l)$ is converted to a left-closed boundary $\mathsf{next}(l)$, and a right-closed boundary $(r)$ is converted to a right-open boundary $\mathsf{next}(r)$, where $\mathsf{next}(\cdot)$ advances the value by one unit at the attribute's resolution [5] . Next, we sort the boundaries in ascending order and remove duplicates. The $i$-th boundary and the $(i+1)$-th one (if exists), where there are more left boundaries than right ones among the first $i$ boundaries, form a left-closed-right-open interval in the domain of $A$ that is of interest to at least one query. We refer to each of these intervals as a bucket, and all these buckets form $\mathfrak{b}_A$. As detailed in Section 3.2, a value $v$ is quantized into an integer, i.e., the id of the bucket containing $v$, or 0 if $v$ lies outside all of the buckets in $\mathfrak{b}_A$. Note that if a query in $\mathfrak{J}$ has no selection predicates on $A$, it still satisfies Lemma 3.3 by setting $i_1 = 0$ and $i_2 = |\mathfrak{b}_A|$.

We present an example of quantization in Example A.1.

*Example A.1.* Continuing from Listing 1, we have

$$\mathfrak{b}_{\texttt{c.Score}} = \{[0, 1)\}$$

,

$$\mathfrak{b}_{\texttt{ph.PostHistoryTypeId}} = \{[1, 2)\}$$

, and

$$\mathfrak{b}_{\texttt{ph.CreationDate}} = \{[\texttt{2010-09-14 11:59:07}, \infty)\}$$

.

By Lemma 3.3, the integer range associated with $Q_3$ is $[1, 1]$ for `c.Score`, $[1, 1]$ for `ph.PostHistoryTypeId`, and $[0, 1]$ for `ph.CreationDate`. Similarly, the integer range associated with $Q_4$ is $[0, 1]$ for `c.Score`, $[1, 1]$ for `ph.PostHistoryTypeId`, and $[1, 1]$ for `ph.CreationDate`.

[5]For example, if $\alpha$ is an integer, $\mathrm{next}(\alpha) = \alpha + 1$; if $\alpha$ is a `timestamp without time zone`, $\mathrm{next}(\alpha) = \alpha + 1$ microsecond; and so on.

**Listing 4:** SQL code for $\beta$-ProcessTable$(R, U, \mathcal{A}^{\text{in \& out}})$.

**Input:** Set $U$ of mappings, with each binds attributes $J_1^{\text{in}}, J_2^{\text{in}}, \ldots$ to specific values, and $\mathcal{A}^{\text{in \& out}} = \{J_1^{\text{in}}, J_2^{\text{in}}, \ldots, J_1^{\text{out}}, J_2^{\text{out}}, \ldots\}$ specifies the set of attributes for partitioning result entries. The quantization scales are $\mathfrak{B}_{\mathfrak{J}}[R] = \{\mathfrak{b}_{S_1}, \mathfrak{b}_{S_2}, \ldots\}$.

**Output:** Result entries have the form $\langle v, \vec{b}, c\rangle$, sorted by $v$, where $v$ is a mapping from $\mathcal{A}^{\text{in \& out}}$ to values, $\vec{b}$ is a grid coordinate for $\mathfrak{B}_{\mathfrak{J}}[R]$, and $c$ is the associated count.

```
SELECT J1^in, J2^in, ..., J1^out, J2^out, ..., B1, B2, ..., COUNT(*)
FROM (
    SELECT J1^in, J2^in, ..., J1^out, J2^out, ...,
        quantize(S1, b_S1) AS B1, quantize(S2, b_S2) AS B2, ...
    FROM R
    WHERE J1^in BETWEEN min(U(J1^in)) AND max(U(J1^in))
        J2^in BETWEEN min(U(J2^in)) AND max(U(J2^in)) AND ...
) AS TMP
GROUP BY J1^in, J2^in, ..., J1^out, J2^out, ..., B1, B2, ...
ORDER BY J1^in, J2^in, ..., J1^out, J2^out, ...;
```

# B MORE DETAILS OF BACON IMPLEMENTATION

We present the details of $\beta$-ProcessTable and $\beta$-BaConRecurse in Appendix B.1.

## B.1 Details of $\beta$-ProcessTable and $\beta$-BaConRecurse

In $\beta$-ProcessTable (Listing 4), the second and third arguments differ from those of ProcessTable. Specifically, $\beta$-ProcessTable takes a set $U$ of bindings rather than a single binding $u$, and it is passed both $J_1^{\text{in}}, J_2^{\text{in}}, \ldots$ and $J_1^{\text{out}}, J_2^{\text{out}}, \ldots$ to enable partition of result entries. Moreover, in the inner block, $J_1^{\text{in}}, J_2^{\text{in}}, \ldots$ are filtered using safe but potentially imprecise range bounds. Concretely, $\min U(J_i^{\text{in}})$ (resp. $\max U(J_i^{\text{in}})$) denotes the minimum (resp. maximum) value in the set $\{u(J_i^{\text{in}}) \mid \forall u \in U\}$. Finally, $\beta$-ProcessTable also SELECTs $J_1^{\text{in}}, J_2^{\text{in}}, \ldots$ and partitions (and orders) the result entries by these attributes, allowing the receiver, $\beta$-BaConRecurse, to discard tuples that do not correspond to any binding in $U$.

We propose $\beta$-BaConRecurse (Algorithm 3) by incorporating batching into BaConRecurse. Instead of producing a single count map, $\beta$-BaConRecurse outputs a dictionary that maps each binding $u \in U$ to a corresponding count map, enabling batch processing of $\beta$ consecutive bindings. Lines 3-29 enumerate subsequences of result entries. Unlike BaConRecurse, these subsequences are not processed immediately; computation is deferred until either $\beta$ subsequences have been accumulated or the end of the returned entries is reached (Line 9). As discussed earlier, returned tuples may not belong to $U$. Therefore, a post-check filters tuples by testing whether $v$'s projection onto $J_1^{\text{in}}, J_2^{\text{in}}, \ldots$ is contained in $U$ (Lines 4-7). During batch processing (Lines 9-28), we first collect the set $U_i$ of distinct bindings on the join attributes between the child table $R_i$ and $R$ (Lines 12-16). We then recursively invoke $\beta$-BaConRecurse for each pair $(R_i, U_i)$. Finally, Lines 19-26 update the dictionary $\mathbf{M}$ by aggregating the contribution of each subsequence. The procedure for merging count maps largely follows BaConRecurse, with two key differences: $R_i$'s count map is indexed by $u_i$ and obtained from the returned dictionary; and the merged count maps are accumulated into the dictionary entry indexed by $\omega^{\text{in}}$. Note that $\omega^{\text{in}}$ (and similarly $v^{\text{in}}$ in Line 4) is empty for the root table, which has no $J^{\text{in}}$ attributes.

**Algorithm 3** $\beta$-BaConRecurse$(R, U)$

**Input:** Set $U$ of mappings, with each binds a subset of $R$'s attributes to specific values. Implicitly, the function also has access to $\mathcal{D}$, $\mathfrak{J}$ and its plan tree, and the quantization scales $\mathfrak{B}_{\mathfrak{J}}$.

**Output:** Dictionary $\mathbf{M}$ maps each binding $u \in U$ to a count map $\mathfrak{M}$ over subtree$(R)$; formally, $\mathbf{M} := \{u \mapsto \mathfrak{M} \mid u \in U\}$. Each $\mathfrak{M}$ (or equivalently $\mathbf{M}[u]$) is complete w.r.t. result tuples of $\mathcal{Q}[\mathfrak{J}]$ restricted to subtree$(R)$ and consistent with $u$.

1: $\mathbf{M} \leftarrow \{u \mapsto$ an empty count map $\mid \forall u \in U\}$;
2: $n_{\text{subsequence}} \leftarrow 0$; ▷ counter for the number of subsequences to be processed
3: **for** each $\mathbf{S}[v]$ of entries of the form $\langle v, \cdot, \cdot\rangle$ with the same $v$, returned by $\beta$-ProcessTable$\big(R, U, \text{JAttrs}_{\mathfrak{J}}(\text{parent}(R) \mid R) \cup \text{JAttrs}_{\mathfrak{J}}(R \mid \text{children}(R))\big)$ **do**
4: $\quad v^{\text{in}} \leftarrow \langle v[J_1^{\text{in}}], \vec{[}J_2^{\text{in}}], \ldots\rangle$; $v^{\text{out}} \leftarrow \langle v[J_1^{\text{out}}], \vec{[}J_2^{\text{out}}], \ldots\rangle$;
5: $\quad$ **if** $v^{\text{in}} \notin U$ **then** ▷ the false positive case mentioned in Section 4.2: join attribute values lie in $[\min(U), \max(U)]$ but are not contained in $U$
6: $\quad\quad$ **continue**;
7: $\quad$ **end if**
8: $\quad n_{\text{subsequence}} \leftarrow n_{\text{subsequence}} + 1$;
9: $\quad$ **if** $n_{\text{subsequence}}{=}\beta$ or $\mathbf{S}[v]$ is the last subsequence returned by $\beta$-ProcessTable **then** ▷ batch processing of $\beta$ subsequences; equivalently, batch processing a set $V$ of consecutive values of $v$
10: $\quad\quad$ **for** each table $R_i \in \text{children}(R)$ **do**
11: $\quad\quad\quad U_i \leftarrow \emptyset$;
12: $\quad\quad\quad$ **for** each subsequence $\mathbf{S}[\omega]$ in the batch **do** ▷ each subsequence in the batch ended with $\mathcal{S}[v]$
13: $\quad\quad\quad\quad \omega^{\text{out}} \leftarrow \langle \omega[J_1^{\text{out}}], \omega[J_2^{\text{out}}], \ldots\rangle$;
14: $\quad\quad\quad\quad u_i \leftarrow \left\{ A' \mapsto \omega^{\text{out}}(A) \;\middle|\; \begin{array}{l} A \in \text{JAttrs}_{\mathfrak{J}}(R|R_i) \\ \wedge\, A' \in \text{JAttrs}_{\mathfrak{J}}(R_i|R) \\ \wedge\, \text{JPreds}_{\mathfrak{J}} \Rightarrow (A = A') \end{array} \right\}$;
15: $\quad\quad\quad\quad U_i \leftarrow U_i \cup u_i$;
16: $\quad\quad\quad$ **end for**
17: $\quad\quad\quad \mathbf{M}_i \leftarrow \beta$-BaConRecurse$(R_i, U_i)$;
18: $\quad\quad$ **end for**
19: $\quad\quad$ **for** each subsequence $\mathbf{S}[\omega]$ in the batch **do**
20: $\quad\quad\quad \omega^{\text{in}} \leftarrow \langle \omega[J_1^{\text{in}}], \omega[J_2^{\text{in}}], \ldots\rangle$; $\omega^{\text{out}} \leftarrow \langle \omega[J_1^{\text{out}}], \omega[J_2^{\text{out}}], \ldots\rangle$;
21: $\quad\quad\quad \mathfrak{M}_0 \leftarrow \{\vec{b} \mapsto c \mid \langle \omega, \vec{b}, c\rangle \in \mathbf{S}[\omega]\}$;
22: $\quad\quad\quad$ **for** each table $R_i \in \text{children}(R)$ **do**
23: $\quad\quad\quad\quad \mathfrak{M}_0 \leftarrow \mathfrak{M}_0 \otimes \mathbf{M}_i[u_i]$; ▷ $u_i$ as defined in Line 14
24: $\quad\quad\quad$ **end for**
25: $\quad\quad\quad \mathbf{M}[\omega^{\text{in}}] \leftarrow \mathbf{M}[\omega^{\text{in}}] \oplus \mathfrak{M}_0$;
26: $\quad\quad$ **end for**
27: $\quad\quad n_{\text{subsequence}} \leftarrow 0$;
28: $\quad$ **end if**
29: **end for**
30: **return** $\mathbf{M}$;

# C MORE DETAILS OF EXPERIMENTS

We present the results of varying $\beta$ in Appendix C.1. We present the per-join-pattern results of both the remaining real workloads and scalability workloads in Appendix C.2. Additionally, the most detailed result of BaCon and PostFilt can be found in the log files of the form `results/*.print` in [39].

## C.1 Experiments of Various $\beta$

We report results for $\beta$ = 5,000, 50,000 and 500,000 respectively, with $\beta$= 50,000 as the default setting used throughout Section 5. The corresponding end-to-end running times are shown in Table 6. Overall, increasing $\beta$ reduces the end-to-end running time, at the cost of higher space overhead (e.g., larger $\mathbf{M}_i$ in Line 17 of Algorithm 3). As indicated by the last two columns of Table 6, the primary source of performance differences lie in the number of SQLs issued to the underlying database and the time spent fetching their results. With larger $\beta$s, BaCon issues fewer SQLs and incurs lower result-fetching overhead, resulting in more savings on the end-to-end running times.

**Table 6:** `job-light.sql`: end-to-end running times (in seconds, rounded), together with the total number of SQLs executed on the underlying database and the total time spent fetching their results, for different values of $\beta$. The default configuration in Section 5 ($\beta$ = 50,000) is shown in bold.

| $\beta$ | E2E running time | Total # of SQLs | Total result-fetching overhead |
|---|---|---|---|
| 5,000 | 922 | 19,427 | 639 |
| **50,000** | **728** | **1,974** | **497** |
| 500,000 | 584 | 243 | 288 |

## C.2 More Per-Join-Pattern Results

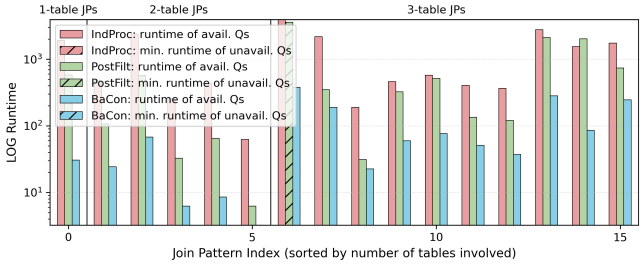


**Figure 9:** Log-scale running time per join pattern in `synthetic.sql`.

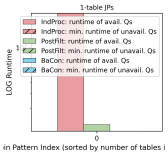

**Figure 10:** Log-scale running time per join pattern in `job_light_single.sql`.

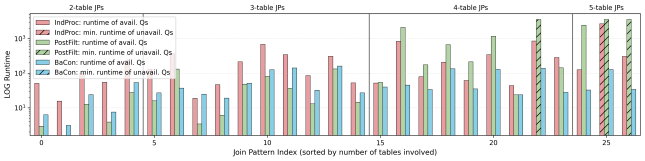


**Figure 11:** Log-scale running time per join pattern in `job_light_join.sql`.

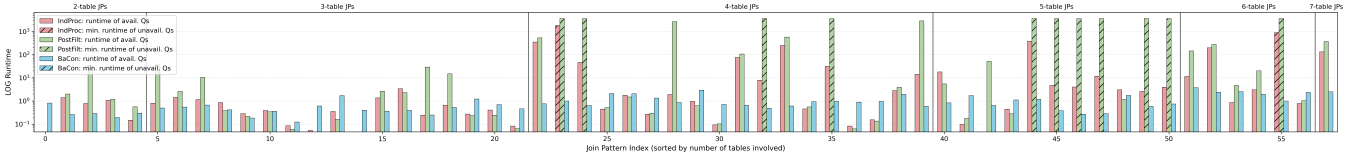


**Figure 12:** Log-scale running time per join pattern in `stats_ceb.sql`.

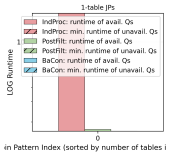

**Figure 13:** Log-scale running time per join pattern in `stats_ceb_single.sql`.

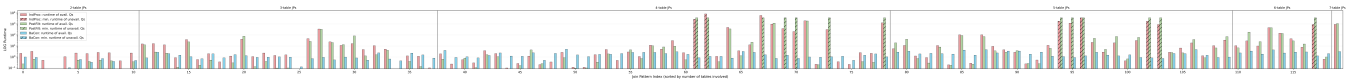

**Figure 14:** Log-scale running time per join pattern in `stats_ceb_join.sql`.

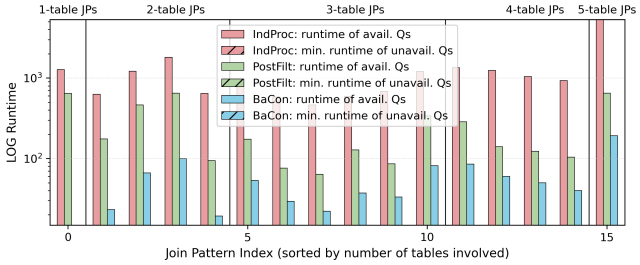


**Figure 15:** Log-scale running time per join pattern in `dsb_grasp_20k.sql`.

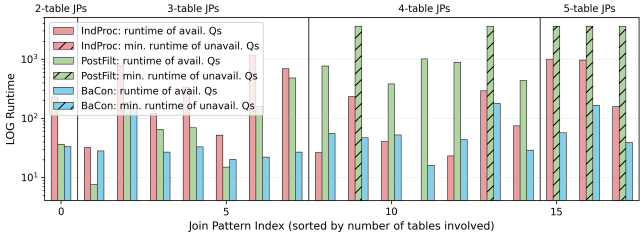


**Figure 16:** Log-scale running time per join pattern in `job_light_1k.sql`.

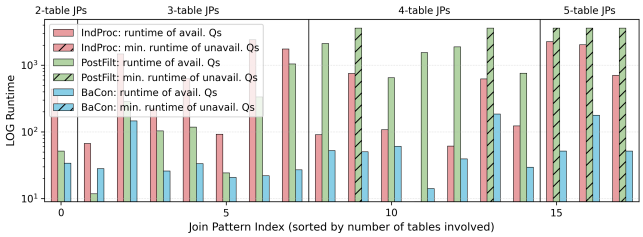


**Figure 17:** Log-scale running time per join pattern in `job_light_2k.sql`.

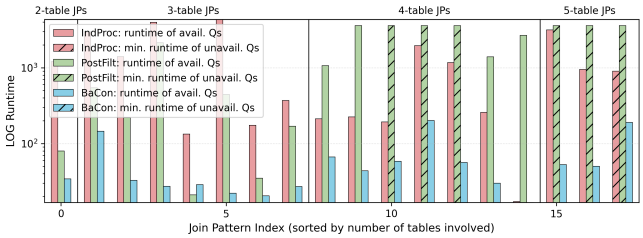


**Figure 18:** Log-scale running time per join pattern in `job_light_4k.sql`.