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, particularly for 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. BaCon's design is compatible with most database management system, and we have implemented it as a client-side application on PostgreSQL with a lightweight C-language UDF (user-defined function). This implementation delivers speedups between 2× and 178× over baselines and good performance across various workloads, 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, 19(9): 2508-2521, 2026.
doi:10.14778/3819518.3819567

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

1 INTRODUCTION

Batches of counting queries are not only useful in their own right for database applications, but also frequently serve to collect basic statistics from data for monitoring and optimization. With the growing popularity of *learned query optimization* [11, 44, 45, 73] in recent years, an interesting workload has emerged: collecting training data for *learned cardinality estimation* (CE) [12, 26, 28, 39, 48, 49, 53, 55, 64]. CE is a critical component of query optimization, as its accuracy directly impacts the quality of query execution plans [34, 56]. 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 [34], STATS [17] and DSB [9]—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 [31, 36]: 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, 28, 55]. 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 [36, 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. While this baseline enables shared join processing, it leads to materializing the full join result, which can be massive for large databases.

We propose ***BaCon***, a practical, scalable algorithm for efficient ***Ba**tch processing of **Coun**ting queries*. A key idea, inspired by work on factorized databases [6, 7, 51], 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

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 19, No. 9 ISSN 2150-8097.
doi:10.14778/3819518.3819567

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 (formally defined in Section 2), and a short description with source in the second row for each workload.

Query Workload	DB	# Qs	# Ts	# join patterns
synthetic [30]	IMDB	5,000	1 to 3	16
Kipf et al. [28]: generated by the training data generator with a different seed				
scale [29]	IMDB	500	1 to 5	31
Kipf et al. [28]: designed to show how MSCN generalizes to more joins.				
job-light [27]	IMDB	70	2 to 5	18
Kipf et al. [28]: derived from JOB [34], excluding string predicates and disjunctions.				
job-light-single [19]	IMDB	254	1	1
Han et al. [17]: single-table sub-plan queries extracted from job-light.				
job-light-join [18]	IMDB	696	2 to 5	27
Han et al. [17]: join sub-plan queries extracted from job-light.				
stats-ceb [23]	STATS	146	2 to 7	58
Han et al. [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
Han et al. [17]: single-table sub-plan queries extracted from stats-ceb.				
stats-ceb-join [21]	STATS	2,603	2 to 7	120
Han et al. [17]: join sub-plan queries extracted from stats-ceb.				
dsb-grasp-20k [40]	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 on top of a database system (DBMS) without modifying its internals. BaCon leverages the DBMS for processing, utilizes user-defined functions (UDFs), and carefully balances database- and client-side execution to minimize interface overhead.

Our experimental results show that BaCon is highly competitive for training workloads for learned CE in Table 1, achieving 2× to 178× speedups over baselines. Moreover, BaCon performs well across diverse workloads, 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 $\mathbf{Q}$ over $\mathcal{D}$.[1] Each query counts the number of tuples returned by a

[1]For simplicity, we discuss only counting queries here, but our methods easily generalize to all *distributive* and *algebraic* SQL aggregates [16] including also SUM, MIN, MAX, AVG, and STDDEV, where each input expression is a single attribute or computable over attributes from the same table.

Listing 1: First 4 queries in `stats_ceb` [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;
```

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. We begin by introducing some useful notations (a table of notations is presented in the full version [41]). Given each query $Q \in \mathbf{Q}$:

- Tables_Q denotes the subset of tables in $\mathcal{D}$ referenced by Q.
- $\text{Attrs}_Q^{\bowtie}$ denotes the subset of attributes in Tables_Q referenced by the join predicates of Q.
- $\text{Preds}_Q^{\bowtie}$ denotes Q's join predicates, represented as the set of equivalent classes of attributes in $\text{Attrs}_Q^{\bowtie}$ 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 a pair of disjoint subsets $\mathcal{R}$ and $\mathcal{R}'$ of tables in Tables_Q, $\text{Attrs}_Q^{\bowtie}[\mathcal{R}|\mathcal{R}']$ denotes the subset of attributes of $\mathcal{R}$ used by Q to join with $\mathcal{R}'$; i.e., $\text{Attrs}_Q^{\bowtie}[\mathcal{R}|\mathcal{R}'] = \{A \in \bigcup_{R\in\mathcal{R}} \text{Attrs}[R] \mid \exists \mathcal{E} \in \text{Preds}_Q^{\bowtie} : A \in \mathcal{E} \wedge (\exists A' \in \mathcal{E} : A' \in \bigcup_{R\in\mathcal{R}'} \text{Attrs}[R])\}$.
- Attrs_Q^{σ} denotes the subset of attributes in Tables_Q referenced by Q's selection predicates.
- Preds_Q^{σ} denotes Q's selection predicates, represented as a mapping from each attribute $A \in \text{Attrs}_Q^{\sigma}$ 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 Tables_Q, $\text{Attrs}_Q^{\sigma}[\mathcal{R}]$ denotes the set of attributes of $\mathcal{R}$ referenced by Q's selection conditions; i.e., $\text{Attrs}_Q^{\sigma}[\mathcal{R}] = \text{Attrs}_Q^{\sigma} \cap \bigcup_{R\in\mathcal{R}} \text{Attrs}[R]$.

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

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

The First Baseline: Independent Processing (IndProc). A straightforward solution, adopted by most existing work on learned CE, runs the queries in $\mathbf{Q}$ 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 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 $\langle \mathsf{Tables}_Q, \mathsf{Preds}^{\bowtie}_Q \rangle$, i.e., the tables that Q joins, and Q's join predicates. Suppose queries in $\mathbf{Q}$ have q distinct join patterns $\mathfrak{J}_1, \ldots, \mathfrak{J}_q$; these patterns partition $\mathbf{Q}$ into a disjoint union of q subsets of queries, denoted $\mathbf{Q}[\mathfrak{J}_1], \ldots, \mathbf{Q}[\mathfrak{J}_q]$, where each $\mathbf{Q}[\mathfrak{J}_i]$ 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 a distinct join pattern. In practice, because all queries in $\mathbf{Q}$ come from the same underlying database $\mathcal{D}$, the number of distinct join patterns tends to be much smaller than the number of queries. For example, JOB's `synthetic` [30] contains 5,000 queries but only 16 distinct join patterns.

The Second Baseline: Join and Post-Filtering (PostFilt). This approach seeks to avoid redundant computation across queries with identical join patterns. We pre-process the queries into partitions $\mathbf{Q}[\mathfrak{J}_1], \ldots, \mathbf{Q}[\mathfrak{J}_q]$ according to their join patterns. This step requires only a scan over $\mathbf{Q}$ to perform syntactical analysis. For each partition $\mathbf{Q}[\mathfrak{J}_i]$, we compute the join only once, followed by a post-filtering step, which checks each join result tuple against the selection predicates of queries in $\mathbf{Q}[\mathfrak{J}_i]$ to determine which queries' counts to increment. We call this baseline PostFilt.

There are many options for implementing the post-filtering step, as discussed in Section 6, including methods that first build a data structure for $\mathbf{Q}[\mathfrak{J}_i]$ for efficient identification and updating of counters in time sublinear to $|\mathbf{Q}[\mathfrak{J}_i]|$ per join result tuple. While these methods can scale to thousands or millions of queries, the typical workloads we target do not have so many queries per join pattern, as evidenced in Table 1. Through our experiments, we have found a simple strategy leveraging the underlying DBMS to be the most effective. Given a join pattern $\mathfrak{J} = \langle \mathsf{Tables}_{\mathfrak{J}}, \mathsf{Preds}^{\bowtie}_{\mathfrak{J}} \rangle$ and k queries $Q_1, \ldots, Q_k$ sharing this pattern, we issue a single SQL query in Listing 2 to compute all of them. Here, Preds^{σ}, Tables, and $\mathsf{Preds}^{\bowtie}$ are translated into SQL. Each CASE expression determines whether a join result tuple contributes to a query Q_i 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 holding 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" [58] 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.

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

```
SELECT COUNT(CASE WHEN Preds^σ_Q1 THEN 1 END), ...,
    COUNT(CASE WHEN Preds^σ_Qk THEN 1 END)
FROM Tables_J WHERE Preds^⋈_J;
```

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 them 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 $\mathbf{Q}$ of queries, BaCon partitions $\mathbf{Q}$ into subsets according to join patterns, such that queries in each subset $\mathbf{Q}[\mathfrak{J}]$ share the same join pattern $\mathfrak{J} = \langle \mathsf{Tables}_{\mathfrak{J}}, \mathsf{Preds}^{\bowtie}_{\mathfrak{J}} \rangle$. 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 $\mathsf{Tables}_{\mathfrak{J}}$ can be seen as a point in a high-dimensional space $\mathbb{S}$, where each dimension corresponds to an attribute in $\bigcup_{R \in \mathsf{Tables}_{\mathfrak{J}}} \mathsf{Attrs}[R]$, ignoring those not referenced by any predicate in $\mathbf{Q}[\mathfrak{J}]$. Let J denote the result of the full join of $\mathsf{Tables}_{\mathfrak{J}}$ using $\mathsf{Preds}^{\bowtie}_{\mathfrak{J}}$. Points in J are those cross-product points that lie on a hyperplane $\mathbb{J}$ in $\mathbb{S}$ defined by $\mathsf{Preds}^{\bowtie}_{\mathfrak{J}}$. Each query $Q \in \mathbf{Q}[\mathfrak{J}]$ corresponds to an orthogonal hyperrectangle in $\mathbb{S}$ with range predicates restricted to dimensions in $\mathsf{Attrs}^{\sigma}_{Q}$. 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 $\mathsf{Tables}_{\mathfrak{J}}$, meaning that their projections onto the subspace $\mathbb{S}[\mathsf{Attrs}[R]]$ for each $R \in \mathsf{Tables}_{\mathfrak{J}}$ cannot exceed $|R|$ distinct points, which is often much smaller than $|J|$. In fact, if we consider the subset of J restricted to any particular combination of values for all join attributes, their projections onto $\mathbb{S}[\bigcup_{R \in \mathsf{Tables}_{\mathfrak{J}}} \mathsf{Attrs}[R]]$ will form a perfect Cartesian product of their projections onto each of $\mathbb{S}[\mathsf{Attrs}[R]]$ for $R \in \mathsf{Tables}_{\mathfrak{J}}$ (which we shall substantiate in Section 3.1). This property, along with the fact the query hyperrectangles may overlap significantly, makes it possible to perform counting tasks more efficiently than simply enumerating J upfront (which PostFilt does).

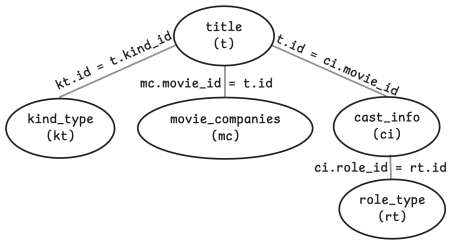

Figure 1: A join pattern in the IMDB schema [35].

To further exploit the overlaps among the query hyperrectangles, consider a subset of J that forms a Cartesian product described above. We can partition its projection onto to each subspace $\mathbb{S}[\mathsf{Attrs}[R]]$ into a coarse grid, using the endpoints of the query range predicates (a process we call "quantization" later in Section 3.2). For example, in Figure 2, this projection consists of a collection of red points, partitioned by a 4×4 grid. A key property of this grid is that all points within a cell lie in the same subset of query hyperrectangles. Therefore, we can compress the red points in each cell to a single "weighted" point, yielding a compressed representation of J (which we refer to as a "count map" later in Section 3.3). This compressed representation is constructed recursively for each $\mathfrak{J}$, without having to enumerate J first (Section 3.4).

3.1 Conditional Orthogonality of Joins

The first idea has its roots in the well-studied problem of factorized databases and related worst-case optimal join algorithms (further discussed in Section 6). A simple observation is that, in order to count the number of 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 $|R_1 \bowtie_{R_1.A=R_2.A} R_2|$ when join predicate exists, if we additionally set the join attribute value $A = x$, then the number of result tuples *conditioned on this specific setting* can still be computed directly: i.e., $|\sigma_{A=x}(R_1 \bowtie_{R_1.A=R_2.A} R_2)| = |\sigma_{A=x}R_1| \times |\sigma_{A=x}R_2|$. We can generalize this idea further to a star-shaped join as follows.

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 $i = 1, \ldots, n$, θ_i is a conjunctive predicate 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 θ_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:

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

Example 3.1. *Consider the join pattern in Figure 1. Tables* title AS t, kind_type AS kt, *and* movie_companies AS mc *correspond to E_0, E_1, E_2, 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:* $|\sigma_{\text{t.kind_id}=x \wedge \text{t.id}=y}\mathsf{t}| \times |\sigma_{\text{kt.id}=x}\mathsf{kt}| \times |\sigma_{\text{mc.movie_id}=y}\mathsf{mc}| \times |\sigma_{\text{ci.movie_id}=y}(\mathsf{ci} \bowtie \mathsf{rt})|$*, without enumerating the full join result tuples. To compute 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 [32]. The subexpression $|\sigma_{\text{ci.movie_id}=y}(\mathsf{ci} \bowtie \mathsf{rt})|$ can be processed 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 $\mathfrak{Q}$ places a natural constraint on the number of constants from each domain appearing in selection predicates. In other words, from the perspective of $\mathfrak{Q}$, fine-grained differences among attribute values may not affect result counts. Our key idea is to 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 a join pattern $\mathfrak{J}$, let $\mathsf{Attrs}^{\sigma}_{\mathfrak{J}} = \bigcup_{Q\in\mathfrak{Q}[\mathfrak{J}]} \mathsf{Attrs}^{\sigma}_{Q}$ denote the set of selection attributes in all queries of pattern $\mathfrak{J}$. We construct a *quantization scale* $\mathfrak{b}_A$ for each selection attribute $A \in \mathsf{Attrs}^{\sigma}_{\mathfrak{J}}$ as follows. First, we extract from $\mathfrak{Q}[\mathfrak{J}]$ the set $\mathsf{Preds}^{\sigma}_{\mathfrak{J}}(A) = \{\mathsf{Preds}^{\sigma}_{Q}(A) \mid \exists Q \in \mathfrak{Q}[\mathfrak{J}] : A \in \mathsf{Preds}^{\sigma}_{Q}\}$ 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 $\mathsf{Preds}^{\sigma}_{\mathfrak{J}}(A)$, 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 x from A's domain is quantized into an integer $\mathfrak{b}_A(x)$, the id of the bucket containing x, or 0 if x lies outside all of buckets in $\mathfrak{b}_A$. We present the detailed construction algorithm in the full version [41], which handles additional intricacies with open or close intervals.

Let $\mathfrak{B}_{\mathfrak{J}} = \{\mathfrak{b}_A \mid A \in \mathsf{Attrs}^{\sigma}_{\mathfrak{J}}\}$ denote the collection of all attribute quantization scales for queries in $\mathfrak{Q}[\mathfrak{J}]$, and $\mathfrak{B}_{\mathfrak{J}}[R] = \{\mathfrak{b}_A \mid A \in \mathsf{Attrs}^{\sigma}_{\mathfrak{J}} \cap \mathsf{Attrs}[R]\}$ denote those for attributes in table $R \in \mathsf{Tables}_{\mathfrak{J}}$. Returning to the geometric view introduced at the beginning of the section, $\mathfrak{B}_{\mathfrak{J}}[R]$ induces a grid over the subspace of $\mathbb{S}$ spanning the selection attribute dimensions of R. This grid allows us to map each tuple in R, by its selection attribute values $\langle x_1, \ldots, x_k\rangle$, to a *grid coordinate* $\vec{b} = (b_1, \ldots, b_k)$, where each $b_i = \mathfrak{b}_{A_i}(x_i)$ is the bucket id for value x_i in the quantization scale for attribute A_i.

Continuing with the geometric intuition, collectively, quantization scales $\mathfrak{B}_{\mathfrak{J}}$ induces a grid over the subspace $\mathbb{S}[\mathsf{Attrs}^{\sigma}_{\mathfrak{J}}]$ consisting of the selection attribute dimensions across all tables in $\mathsf{Tables}_{\mathfrak{J}}$. By construction of $\mathfrak{B}_{\mathfrak{J}}$, all query hyperrectangles in $\mathfrak{Q}[\mathfrak{J}]$ perfectly align with grid boundaries. Hence, all selection predicates can be quantized using the same scales. Lemma 3.2 below formalizes the

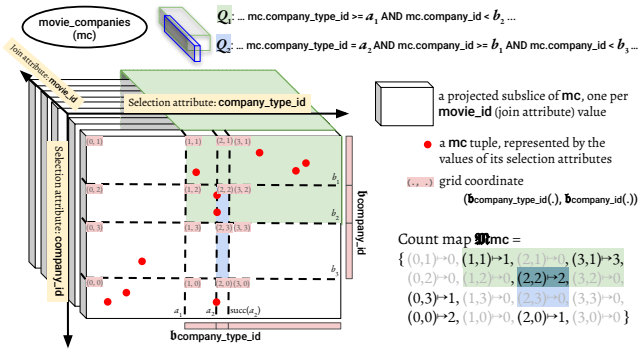

Figure 2: Quantized selection count map for a projected subslice of table mc. The quantization scales for mc are induced by queries Q_1 and Q_2 (only selection predicates on mc are shown).

guarantee that precise evaluation of selection predicates is possible in the quantized space.

Lemma 3.2 (Quantization Preserves Selections). *Given a set of selection-join queries $\mathfrak{Q}[\mathfrak{J}]$ and quantization scales $\mathfrak{B}_{\mathfrak{J}}$ constructed from $\mathfrak{Q}[\mathfrak{J}]$, there exists a function $f(\mathfrak{b}_A, \delta)$ returning an integer range $[i_1, i_2]$ for a range δ over an attribute A with quantization scale $\mathfrak{b}_A \in \mathfrak{B}_{\mathfrak{J}}$, such that for any $Q \in \mathfrak{Q}[\mathfrak{J}]$ and every selection predicate $A \mapsto \delta$ in $\text{Preds}^{\sigma}_{Q}$: $x \in \delta \Leftrightarrow \mathfrak{b}_A(x) \in f(\mathfrak{b}_A, \delta)$.*

As an example, Figure 2 shows the grid over table mc induced by two queries. The quantization scales $\mathfrak{b}_{\text{company_type_id}}$ and $\mathfrak{b}_{\text{company_id}}$ respectively correspond to the two mc attributes referenced by the queries' selection predicates. (For now, ignore the mention of "projected subslice," which will be formally introduced in Section 3.4.) The buckets for $\mathfrak{b}_{\text{company_type_id}}$, numbered 1 through 3, are $[a_1, a_2)$, $[a_2, \text{succ}(a_2))$, and $[\text{succ}(a_2), \infty)$, where $\text{succ}(a_2)$ denotes the successor value of a_2 in the domain; any value in $(-\infty, a_1)$ will be quantized to special bucket id 0 because this range is not contained in any selection predicate. The buckets for $\mathfrak{b}_{\text{company_id}}$ are similarly induced by selection predicates involving company_id.

3.3 Quantized Selection Count Maps

Since queries in $\mathfrak{Q}[\mathfrak{J}]$ 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 $R \in \text{Tables}_{\mathfrak{J}}$ with selection attributes $\{A_1, \ldots, A_k\}$ and quantization scales $\mathfrak{B}_{\mathfrak{J}}[R] = \{\mathfrak{b}_{A_1}, \ldots, \mathfrak{b}_{A_k}\}$, we compress a subset of tuples in R into a *(quantized selection) count map* $\mathfrak{M}$: each entry of $\mathfrak{M}$ maps a grid coordinate $\vec{b} = (b_1, \ldots, b_k)$ to $\mathfrak{M}[\vec{b}]$, the count of tuples within the grid cell—i.e., any tuple t satisfying $\mathfrak{b}_{A_i}(t.A_i) = b_i$ for $i = 1, \ldots, k$. For example, Figure 2 shows the count map for the set of tuples (2D points in the geometric view).

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

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

- $\otimes$ **(multiply)** $\mathfrak{M}_1$ and $\mathfrak{M}_2$ are count maps over disjoint subsets $\mathcal{R}_1$ and $\mathcal{R}_2$ of tables in $\text{Tables}_{\mathfrak{J}}$. We define $\mathfrak{M}_1 \otimes \mathfrak{M}_2$, a count map over $\mathcal{R}_1 \cup \mathcal{R}_2$, as $\{\vec{b}_1 \smile \vec{b}_2 \mapsto (\mathfrak{M}_1[\vec{b}_1] \cdot \mathfrak{M}_2[\vec{b}_2]) \mid \vec{b}_1 \in \mathfrak{M}_1, \vec{b}_2 \in \mathfrak{M}\}$, where $\smile$ concatenates grid coordinate vectors.
- $\oplus$ **(add)** $\mathfrak{M}_1$ and $\mathfrak{M}_2$ are count maps over the same subset of tables $\mathcal{R} \subseteq \text{Tables}_{\mathfrak{J}}$ (and thus same quantization scales). We define $\mathfrak{M}_1 \oplus \mathfrak{M}_2$, a count map over $\mathcal{R}$, as $\{\vec{b} \mapsto (\mathfrak{M}_1[\vec{b}] + \mathfrak{M}_2[\vec{b}]) \mid \vec{b} \in \mathfrak{M}_1\}$.

The following lemmas establish the correctness of using $\otimes$ and $\oplus$ for computing count maps. Formally, given $\mathfrak{Q}[\mathfrak{J}]$, quantization scales $\mathfrak{B}_{\mathfrak{J}}$, a subset of tables $\mathcal{R} \subseteq \text{Tables}_{\mathfrak{J}}$, 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 $Q \in \mathfrak{Q}[\mathfrak{J}]$, the size of the intersection between T and the selection-join subquery of Q restricted[2] to $\mathcal{R}$ can be computed from $\mathfrak{M}$ and $\mathfrak{B}_{\mathfrak{J}}$.

Lemma 3.3 (Multiplying Count Maps). *Given $\mathfrak{Q}[\mathfrak{J}]$, consider disjoint subsets $\mathcal{R}_0, \ldots, \mathcal{R}_n$ of tables in $\text{Tables}_{\mathfrak{J}}$, where $\text{Preds}^{\bowtie}_{\mathfrak{J}}$ implies a join condition θ_i relating $\mathcal{R}_0$ to $\mathcal{R}_i$ for each $i = 1, \ldots, n$, but there is no join condition across $\mathcal{R}_1, \ldots, \mathcal{R}_n$ that is not already implied by $\bigwedge_i \theta_i$. For each $i = 1, \ldots, n$, let E_i denote the join subquery of tables in $\mathcal{R}_i$, with condition implied by $\text{Preds}^{\bowtie}_{\mathfrak{J}}$. 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 $\mathcal{A}_0 = \text{Attrs}^{\bowtie}_{\mathfrak{J}}[\mathcal{R}_0 \mid \cup_{i \in [1,n]} \mathcal{R}_i]$ 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 $\mathcal{A}_0 \subseteq \mathcal{A} \subseteq \bigcup_{R \in \mathcal{R}_0} \text{Attrs}[R]$. 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*
- *$\forall i = 1, \ldots, n$: $\mathfrak{M}_i$ 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 3.4 (Adding Count Maps). *Given $\mathfrak{Q}[\mathfrak{J}]$ and a subset of tables $\mathcal{R} \subseteq \text{Tables}_{\mathfrak{J}}$, suppose $\mathfrak{M}_1, \ldots, \mathfrak{M}_n$ are count maps over $\mathcal{R}$,*

[2] More precisely, we remove from Q any table outside $\mathcal{R}$; remove from any equivalent class in $\text{Preds}^{\bowtie}_{Q}$ any attribute outside $\mathcal{R}$ and then remove any empty or singleton equivalent class; and remove any entry in $\text{Preds}^{\sigma}_{Q}$ for an attribute outside $\mathcal{R}$.

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

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

1: Scan $\mathfrak{Q}$ and partition it into $\bigcup_{\mathfrak{J}} \mathfrak{Q}[\mathfrak{J}]$ by join pattern;
2: **for** each subset $\mathfrak{Q}[\mathfrak{J}]$ of $\mathfrak{Q}$ **do**
3: $\quad \mathfrak{B}_{\mathfrak{J}} \leftarrow \{\textsc{QuantScales}(R, \mathfrak{Q}[\mathfrak{J}]) \mid R \in \mathsf{Tables}_{\mathfrak{J}}\}$;
4: $\quad$ Determine the plan tree for $\mathfrak{J}$;
5: $\quad \mathfrak{M} \leftarrow \textsc{BaConRecurse}(\mathsf{root}(\mathfrak{J}), \emptyset)$;
6: $\quad$ **yield from** $\textsc{ComputeCounts}(\mathfrak{Q}[\mathfrak{J}], \mathfrak{M})$;
7: **end for**

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 $\bigcup_{i \in [1,n]} T_i$.

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 $\mathfrak{Q}[\mathfrak{J}]$ 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 [7, 51]. Nodes in this tree correspond to the tables of $\mathsf{Tables}_{\mathfrak{J}}$, and edges connecting the nodes represent equijoin conditions among them. Assuming that queries are acyclic and contain no cross products, one such tree always exists to capture $\mathsf{Preds}^{\bowtie}_{\mathfrak{J}}$ 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 $\mathfrak{Q}[\mathfrak{J}]$ 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 $\mathfrak{Q}[\mathfrak{J}]$. This count map is then used to calculate the result counts for queries, which can be done naively by summing up the counts at grid coordinates in each query hyperrectangle; 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 $\mathsf{subtree}(R)$ (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 $\mathsf{root}(\mathfrak{J})$ has $u = \emptyset$, so its slice in fact contains all result tuples in the full join of $\mathsf{Tables}_{\mathfrak{J}}$). As discussed in Section 3.3, processing the entirety of T efficiently is hard; instead, BaConRecurse partitions the slice into "subslices," where each subslice T_v binds a particular combination v of values for R's attributes that join with its children, i.e., $\mathsf{Attrs}^{\bowtie}_{\mathfrak{J}}[R \mid \mathsf{children}(R)]$. 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 3.3, 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 computes, for each projected subslice of R, the

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 $\mathsf{subtree}(R)$, complete w.r.t. result tuples of $\mathfrak{Q}[\mathfrak{J}]$ restricted to $\mathsf{subtree}(R)$ and consistent with u.

1: $\mathfrak{M} \leftarrow \emptyset$; ▹ missing entries in all maps default to count 0
2: **for** each subsequence $\mathsf{S}[v]$ of entries $\langle v, \cdot, \cdot \rangle$ with the same v, returned by $\textsc{ProcessTable}\big(R, u, \mathsf{Attrs}^{\bowtie}_{\mathfrak{J}}[R \mid \mathsf{children}(R)]\big)$ **do**
3: $\quad \mathfrak{M}_0 \leftarrow \{\vec{b} \mapsto c \mid \langle v, \vec{b}, c \rangle \in \mathsf{S}[v]\}$;
4: $\quad$ **for** each table $R_i \in \mathsf{children}(R)$ **do**
5: $\quad\quad u_i \leftarrow \left\{ A' \mapsto v(A) \;\middle|\; \begin{array}{l} A \in \mathsf{Attrs}^{\bowtie}_{\mathfrak{J}}[R|R_i] \wedge A' \in \mathsf{Attrs}^{\bowtie}_{\mathfrak{J}}[R_i|R] \\ \wedge\, \mathsf{Preds}^{\bowtie}_{\mathfrak{J}} \Rightarrow (A = A') \end{array} \right\}$;
6: $\quad\quad \mathfrak{M}_i \leftarrow \textsc{BaConRecurse}(R_i, u_i)$;
7: $\quad\quad \mathfrak{M}_0 \leftarrow \mathfrak{M}_0 \otimes \mathfrak{M}_i$;
8: $\quad$ **end for**
9: $\quad \mathfrak{M} \leftarrow \mathfrak{M} \oplus \mathfrak{M}_0$;
10: **end for**
11: **return** $\mathfrak{M}$;

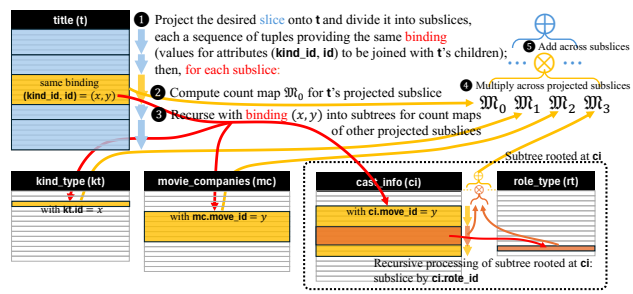

Figure 3: Illustration of Algorithm 2, continuing Figure 1.

coordinate-count pairs to populate the count map. 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 3.4, 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 3.2. *Following Figure 3, at the root* title, *BaCon iterates over all possible bindings for* (t.kind_id, t.id) *in turn. Suppose the blue subslice fixes* t.kind_id $= 1$, t.id $= 1$. *For the subslice projected to* t, *suppose* $\mathfrak{M}_0 = \mathfrak{M}\mathrm{t} = \{(9) \mapsto 2, \ldots\}$. *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* $\mathfrak{M}_2 = \mathfrak{M}_{\mathsf{mc}}$ *is the one shown in Figure 2. From the geometric view,* $\mathfrak{M}_2$ *is computed by the flattened hyperrectangle (in Figure 2) with* mc.movie_id = t.id = 1. *For illustration, let us say that* $\mathfrak{M}_1$ $(\mathfrak{M}_{\mathsf{kt}}) = \{\ldots, (1) \mapsto 1, \ldots\}$ *and* $\mathfrak{M}_3$ $(\mathfrak{M}_{\mathsf{ci}\bowtie\mathsf{rt}}) = \{\ldots, (3,3,3) \mapsto 3, \ldots\}$.

This subslice contributes to the global count map $\mathfrak{M}$ *as follows:* $\mathfrak{M} \leftarrow \mathfrak{M} \oplus (\mathfrak{M}_0 \otimes \cdots \otimes \mathfrak{M}_3)$. *We zoom in on one example—how the global count map entry at grid coordinate* $(9, 1, 2, 2, 3, 3, 3)$ *is updated. Concretely, this global grid coordinate is the concatenation of* (9) *from* $\mathfrak{M}_0$, (1) *from* $\mathfrak{M}_1$, $(2, 2)$ *from* $\mathfrak{M}_2$, *and* $(3, 3, 3)$ *from* $\mathfrak{M}_3$. *Recall that the counts in* $\mathfrak{M}_0, \ldots, \mathfrak{M}_3$ *at these coordinates are* 2, 1, 2, 3, *respectively; therefore, the contribution to the global count map entry*

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

Input: Mapping u binds attributes $J_1^{\text{in}}, J_2^{\text{in}}, \ldots$ to specific values, and $\mathcal{A}^{\texttt{out}} = \{J_1^{\text{out}}, J_2^{\text{out}}, \ldots\}$ 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 $\mathcal{A}^{\texttt{out}}$ to values, $\vec{b}$ is a grid coordinate for $\mathfrak{B}_{\mathfrak{J}}[R]$, and c is the associated count.

```
SELECT J_1^out, J_2^out, ..., B1, B2, ..., COUNT(*)
FROM (
    SELECT J_1^out, J_2^out, ...,
        quantize(S_1, b_S1) AS B1, quantize(S_2, b_S2) AS B2, ...
    FROM R
    WHERE J_1^in = u(J_1^in) AND J_2^in = u(J_2^in) AND ...
) AS TMP
GROUP BY J_1^out, J_2^out, ..., B1, B2, ...
ORDER BY J_1^out, J_2^out, ...;
```

is $2 \times 1 \times 2 \times 3 = 12$. *After processing this subslice,* BaConRecurse *proceeds to the next* (t.kind_id, t.id) *combination in order.*

Delving into ProcessTable (Listing 3), we see how a single SQL query over a table R performs "subslicing" (by $\mathcal{A}^{\texttt{out}}$) and computes the count maps of all projected subslices. 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 $\mathcal{A}^{\texttt{out}}$ to perform subslicing, and then by their grid coordinates to compute tuple count per grid cell for each projected subslice. Finally, ORDER BY ensures that entries for the same projected subslice, i.e., those with the same binding v for $\mathcal{A}^{\texttt{out}}$, are consecutive in the output. This ordering allows the caller BaConRecurse to detect where a sequence 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. 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. Many optimizations thus involve balancing database- and client-side processing options and mitigating the overhead of interfacing them. Overall, we perform ProcessTable (Listing 3) inside the DBMS, using cursors to fetched join values, quantization grid coordinates, and associated counts. The remainder of BaConRecurse (i.e., merging of count maps along the tree structure) and the remainder of BaCon (i.e., ComputeCounts) are performed on the client side. The associated optimizations are presented below. While BaCon is currently implemented on top of PostgreSQL, the ideas in this section generalize to other DBMSs, although details may differ.

Note that BaCon is not optimized for join patterns characterized by queries exhibiting little opportunity for shared processing (e.g., when there are too few of them, or their predicates overlap very little), or an underlying full join that is small and cheap to compute to begin with. In such cases, BaCon may not outperform baselines IndProc or PostFilt. Therefore, we also have developed a method called Hybrid, which acts as an optimizer to choose the appropriate method to use among IndProc, PostFilt, and BaCon for a given join pattern. In our experimental comparison of these methods in Section 5, we empirically analyze the cases where BaCon is not optimal (Section 5.2): they follow our characterizations above and play only a small role in the overall performance of query workloads found in the learned CE literature. Furthermore, despite Hybrid's ability to make on per-pattern decisions, it offers no consistent improvement over BaCon overall. This observation speaks to the effectiveness of BaCon, even without Hybrid, for handling practical CE training workloads. More details on Hybrid can be found in the full version [41].

4.1 Choosing Execution Plan for a Join Pattern

Give a join pattern $\mathfrak{J}$, the cost of BaConRecurse to process $\mathfrak{Q}[\mathfrak{J}]$ 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 tree root 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 $\text{Tables}_{\mathfrak{J}}$ according to $\text{Preds}^{\bowtie}_{\mathfrak{J}}$. The rationale is to maximize the saving opportunities identified by Lemma 3.3: a root with a large in-degree can potentially avoid enumerating more cross products. Geometrically, this heuristic ensures that, during recursion, the grid coordinates in each descendant lie in low-dimensional subspaces. Once we pick the root, the rest of the tree structure follows naturally from $\text{Preds}^{\bowtie}_{\mathfrak{J}}$. 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.[3] 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. For a node R, after some iterations of the loop on Lines 2–10 of Algorithm 2, we count, for

[3]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.

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 such cases 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 can use either a server-side cursor [62] or a client-side one [60]. A client-side cursor fetches the entire query result set into client memory. A server-side cursor, in contrast, allows the client to iterate row-by-row without materializing a potentially massive result set. While server-side cursors offer more flexibility and improve scalability, they are considerably slower and require state to be maintained on the database server; a large number of concurrent server-side cursors can cause performance issues. Therefore, BaCon uses a server-side cursor for ProcessTable on the root, but switches to client-side cursors for all other 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 per subslice, and even with the batching optimization below, their result sizes remain manageable for client-side cursors.

Batching of ProcessTable. During recursive execution of BaConRecurse over the plan tree, the overhead of issuing many SQL queries via ProcessTable can accumulate, especially when many of these queries, with specific attribute bindings, return small result sets. To mitigate this overhead, we batch multiple invocations of ProcessTable for the same table R, using a set of bindings U rather than a single binding u, into one SQL query. Specifying the combined condition precisely in the WHERE clause of Listing 3 is generally infeasible, as it would have to enumerate all bindings in U. Instead, we compute the minimum and maximum elements[4] of U and use $[\min(U), \max(U)]$ as a safe but possibly imprecise range bound in WHERE, i.e., WHERE $(J_1^{\text{in}}, J_2^{\text{in}}, \ldots)$ BETWEEN $\min(U)$ AND $\max(U)$. We extend the query to additionally group by $J_1^{\text{in}}, J_2^{\text{in}}, \ldots$ 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 β consecutive settings of v into a set V for processing as a batch, analogous to a β-fold loop unrolling on Lines 2–10 of Algorithm 2. For each child R_i, we project V to obtain the set U_i of unique bindings on its join attributes with R (Line 5), and invoke ProcessTable on R_i with U_i. The detailed algorithm is presented in the full version [41].

We now provide further analysis of this batching optimization. First, beyond reducing the overhead of issuing many queries, batching promotes reuse of ProcessTable calls on the same table and attribute binding. Without batching, consecutive settings of v in 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 naturally. So we have not implemented additional caching mechanisms for BaCon, although further exploration will be a good future work.

Second, compressing a set of bindings into a single range may introduce false positives. Indeed, R_i may always contain some tuples whose join 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 UDF quantize in Listing 3 is invoked many times, efficient implementation is crucial for performance. We implement quantize as C-language UDFs 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-language UDFs run as compiled code natively inside the server process, greatly outperforming SQL-based UDFs. BaCon creates these extensions at the beginning of its execution and drops them at the end. More generally, quantize can be implemented in other DBMSs using available extensions, including CLR (common language runtime) integration for Microsoft SQL Server [47] or external procedures for Oracle Database [52].

4.3 Implementation of ComputeCounts

ComputeCounts is invoked by BaCon to compute the result counts of queries in $\mathfrak{Q}[\mathfrak{J}]$ given the final count map $\mathfrak{M}$ computed by BaConRecurse. Conceptually, it performs a "join" between a set of query hyperrectangles ($\mathfrak{Q}[\mathfrak{J}]$) 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 $|\mathfrak{Q}[\mathfrak{J}]|$ is moderate (in hundreds or thousands) and $\mathfrak{M}$ tends to be sparse (32% on average across join patterns in JOB's `synthetic`). 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 $Q \in \mathfrak{Q}[\mathfrak{J}]$ 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 [3] for ComputeCounts. With the help from Numba's compiler optimizations, this simple implementation was able to beat more sophisticated methods.

5 EXPERIMENTS

We compare IndProc, PostFilt,[5] and BaCon. All algorithms are implemented in Python3, using `psycopg2` [61] 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

[4] If each binding involves multiple attributes, we order the bindings lexicographically.

[5] We have also evaluated an alternative implementation of PostFilt based on materialized views. The details and results, reported in the full version [41], show that materialization introduces additional storage costs and preparation-time overhead, without providing overall performance benefits. Hence, Section 5 focuses on the current implementation of PostFilt.

(40 logical cores, 2.50GHz), 256GB of main memory, and 1TB of disk storage. We set the batching parameter β (Section 4.2) to 50,000 for BaCon for all experiments and present results for different β's in the full version [41]. Code is available in the GitHub repository [42].

We test three popular datasets: IMDB [34], STATS [17], and DSB [9] (of scaling factor 2). They are loaded and initialized using the provided scripts [10, 20, 33], 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 [43] from the source file [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`, 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.

In terms of client-side memory usage, because BaCon processes each join pattern separately, memory peaks at the single most complex join pattern in the workload. Across all nine workloads, the maximum memory footprint of BaCon occurs for a join pattern in `synthetic`, at 1.7GB. We do not report memory usage for PostFilt and IndProc, as they executed entirely inside the database server.

Single-Threaded vs. Parallel Execution. Results below focus on single-threaded execution to isolate the algorithmic differences of BaCon compared with the baselines, without the potentially confounding factor of parallelization. We disable parallelism in PostgreSQL by setting max_parallel_workers_per_gather to 0, and restrict all client-side code to a single thread. Besides this single-threaded setting, we have also evaluated all methods in a parallel setting. Under the parallel setting, IndProc and PostFilt fully leverage PostgreSQL's built-in parallelism; for BaCon, we use a simple multi-process design in which each process handles a subset of join patterns (more sophisticated parallelization strategies for BaCon are possible as future work). Briefly, the results show that, even with this simple strategy, BaCon achieves end-to-end speedups of 2× to 598× compared with IndProc, consistent with the single-threaded results shown below in Section 5.1. Additional details on the parallel setting can be found in the full version [41].

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 (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

Table 2: End-to-end running times (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 ×

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` and `dsb-grasp-20k`). 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 with 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 when processing each join pattern, and typically completes within a few seconds; the only exception is `dsb-grasp-20k`, where it reaches ≈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 never times out. 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 by Han et al. [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 4 reports the running times of all join patterns,

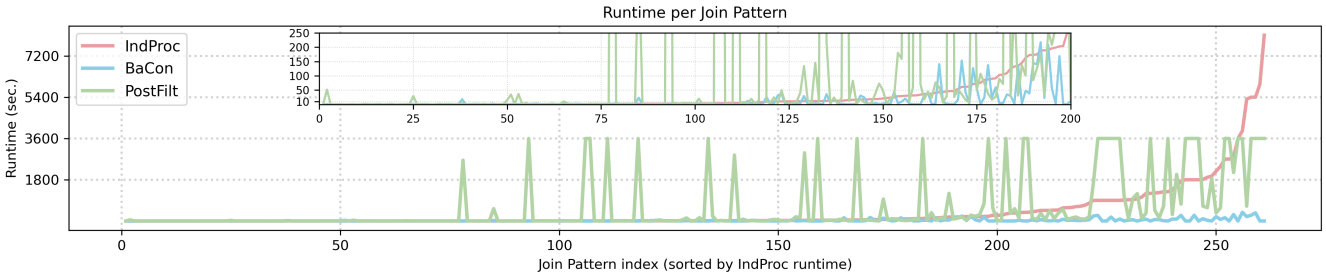

Figure 4: Running time (seconds) per join pattern across 283 join patterns from 9 workloads. Join patterns are ordered by IndProc's times. The inset is the zoom-in of the first 200 indices.

Table 3: Number of timed-out cases. 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	# queries	# join patterns	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)

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.

Additionally, we present a detailed per-join-pattern analysis for `scale` and `job-light` here; results for the remaining workloads are in the full version [41]. As shown in Figure 5, `scale` 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 attribute bindings and coordinates, and 18s (15%) for merging count maps; 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, although BaCon performs less work than PostFilt, BaCon's cursor invocations and row fetching/parsing incur more overhead than PostFilt's server-side processing.

We have similar observations from Figure 6. As discussed by Han et al. [17], `job-light` 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`. Therefore, IndProc consistently performs well on all join patterns, except one severe time-out for join pattern 17. However, BaCon remains competitive across join patterns and never times out. Regression compared with baselines is limited: in the worst case, BaCon is 55s slower compared with IndProc on join pattern 8, and 100s compared with PostFilt on join pattern 7.

Overall, consistent with the results in Section 5.1, BaCon does well across join patterns: it significantly outperforms the baselines on expensive patterns and remains acceptable on cheap ones.

5.3 Scalability

In this section, we evaluate the scalability of the three approaches using three synthetic workloads, denoted `job-light-*`. We choose `job-light` as the reference because, as shown in Figure 6, IndProc performs consistently well, except for a single join pattern that times out. Specifically, on `jog-light`, 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., where the number of queries is small and intermediate join sizes remain moderate. In particular, if the number of queries is small, there is less opportunity for sharing, and IndProc's simple approach of optimizing and evaluating queries one by one works fine. 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. Queries are generated randomly and independently. We randomly select a query from `job-light`, reuse its template (and thus its join pattern), but modify its predicate constants: 1) for a constant in equality and inequality predicates, we replace it with a random

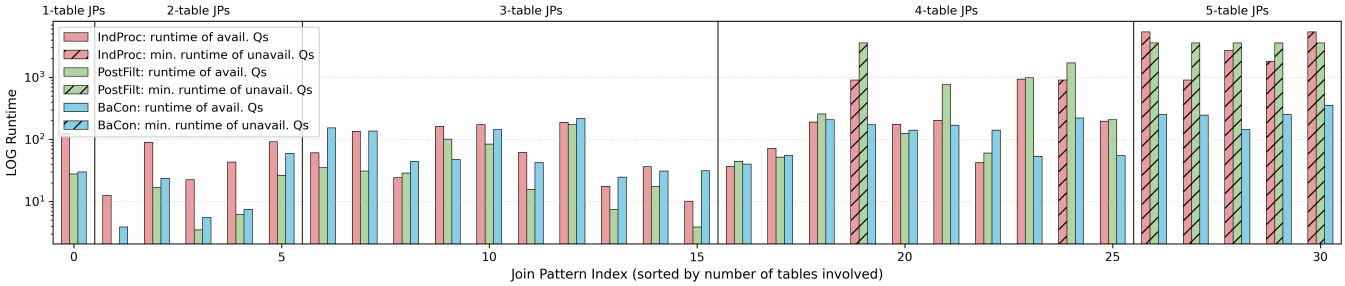

Figure 5: Log-scale running time per join pattern in `scale`. Join patterns are ordered by the number of tables involved, and the figure is partitioned accordingly (labels shown above). Hatched bars (baselines only) indicate the lower bounds for timed-out cases.

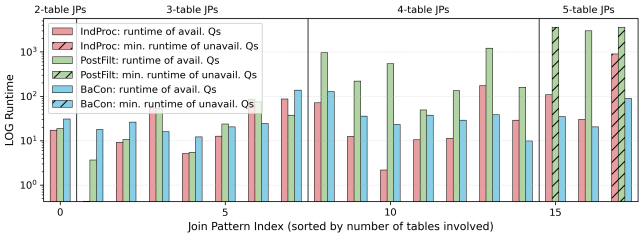

Figure 6: Log-scale running time per join pattern in `job-light`, with same format as Figure 5.

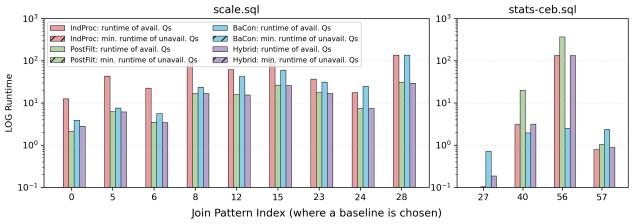

Figure 7: Log-scale running time per join pattern (where a baseline is chosen) in `scale` and `stats_ceb`, including Hybrid's.

Table 4: End-to-end running times (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+ ×

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

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

value from the active domain of the corresponding attribute; 2) for range predicates, we randomize the left endpoint and adjust the right accordingly to preserve the range length.

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 $|\mathfrak{Q}[\mathfrak{J}]|$, the cost of quantizing an attribute is only $O(\log(|\mathfrak{Q}[\mathfrak{J}]|))$ thanks to binary search. Per-join-pattern results are provided in the full version [41]. While the number of join patterns remains 18, the number of them where BaCon outperforms IndProc increases to 14 (78%) in `job-light-1k`, 17 (94%) in `job-light-2k`, and finally 18 (100%) in `job-light-4k`. Overall, BaCon demonstrates superior scalability compared with 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 the full version [41], 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` and `stats_ceb_join`, 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` and 4 join patterns in `stats_ceb`, while selecting BaCon for all 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 7 presents a per-join-pattern analysis, focusing on patterns where Hybrid chooses a baseline. As shown in Figure 7, for `scale`, all 9 switches to PostFilt are beneficial, yielding a total saving of 156s. In `stats_ceb`, 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 overhead for statistics collection and training may outweigh its benefit.

6 RELATED WORK

Select-Join-Aggregate Query Processing. Counting join results (with selection predicates on base tables) is a special form of select-join-aggregate queries. Standard processing typically involves pushing down selection predicates before evaluating the join and then aggregate. The classic Yannakakis algorithm [70] computes free-connex join-count queries in $O(N)$ time, where N is the number of tuples. Recent advancements [25] exploit a hybrid strategy of Yannakakis algorithm that handles acyclic join-count queries in $O(N + N \cdot \mathsf{OUT}^{1-\frac{1}{w}})$ time. BaCon draws inspiration from these techniques, specifically the recursive computation and merging of coordinate-to-count maps during backtracking from [2, 13, 46, 63]. Cost-based optimizers in these systems, such as Free Join's [63], together with the AGM bound [5, 15], further inform our initial exploration of cost-based hybrid mechanism detailed in the full version [41]. However, more accurate cost estimates may conversely require more expensive estimation methods, such as LpBound [71], and/or specialized data storages and execution engines, such as Free Join's COLT [63] and its integration to DuckDB [54].

Aggregation Push-down and Factorizations. Aggregation push-down reduces intermediate results via early or deferred aggregation. Classical approaches include eager and lazy aggregation [69] and integrating GROUP BY into cost-based optimization [8]. GuAo [32] identifies cases where join materialization can be avoided entirely and introduces a corresponding physical operator for SparkSQL [4]. Its propagation of frequencies grouped by join attributes resembles BaCon's bucketization in single-query settings. However, it relies on binary joins, whereas our approach support multi-way merging of count maps. Factorized databases [6, 7, 51] eliminate tuple-level redundancy in query results through compact representations that can support efficient aggregation beyond COUNT. These works, focusing on eliminating intra-query redundancy, are complementary to our approach, which also exploits inter-query sharing. Furthermore, these works typically employ non-traditional storage layout different from traditional relational databases.

Optimized Batch Processing of Aggregate Queries. LMFAO [57] evaluates batches of ground-by aggregates over shared joins without materializing intermediates by decomposing queries into views, organizing them via a join tree, and executing a multi-output plan. Like PostFilt, it encodes selection predicates as conditional expressions within aggregates. Unlike BaCon, LMFAO does not preprocess overlaps among cross-query predicates, but instead focuses on sharing computation of join tree traversal and attribute-ordered evaluation, and is implemented as a standalone execution engine.

Multi-Query & Shared Workload Optimization. Multi-query optimization (MQO) [59] identifies shared/similar subexpressions across queries and evaluate them jointly or via plan rewriting. Zhou et al. [72] propose a common subexpression manager integrated into Microsoft SQL Server, demonstrating improvements on workloads of tens of queries. Shared Workload Optimization (SWO) [14] extends this idea to larger workloads by sharing operators, often requiring specialized execution engines. These approaches target general workloads and require accurate cost estimates.

Learned Cardinality Estimation. Query-aware cardinality estimation (CE) models [12, 26, 28, 36, 38, 39, 48, 49, 53, 64] learn from query-count pairs. Training is typically performed either in a single-shot manner, where a model is (re-)trained from scratch on a batch of queries [12, 26, 36, 37, 39], or in multiple rounds, where the model is incrementally updated as new query-count pairs become available [24, 38, 53]. Some hybrid CE models [28, 38] additionally require data statistics, e.g., samples/histograms. Recent work studies robustness, training efficiency, and adaptivity to data updates and workload shifts [24, 31, 36, 38, 50, 55, 65, 68], increasing the demand for efficiently supporting counting queries.

Other Related Work. Our work is also related to continuous query processing. More details are provided in the full version [41].

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 can be deployed without modifying DBMS internals or physical designs. Components of BaCon can be accelerated using efficient UDF implementations (such as C-extensions in PostgreSQL), if they are supported by the DBMS. Our results demonstrate significant performance gains across real workloads with diverse characteristics. This level of improvement is empowering: shorter (re)training times make learned CE more practical and allow 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 more complex aggregates (complex expression inputs or non-algebraic aggregate functions) would further broaden its applicability. Beyond functional extensions, there are opportunities for more 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 more sophisticated parallelization. Finally, new applications and additional optimizations will likely require models beyond the current Hybrid to enable cost-based selection of processing methods.

ACKNOWLEDGMENTS

This work of J.Y., P.A., and Y.L. was partially supported by NSF Grant IIS-2402823. P.A. was also supported by NSF Grant CCF-2223870 and a US-Israel Binational Science Foundation Grant 2022131. This work of X.H. was supported by the Natural Sciences and Engineering Research Council of Canada Discovery Grant.

REFERENCES

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

[2] 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

[3] Inc. 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#

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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.

[9] 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

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

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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

[16] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Min. Knowl. Discov.* 1, 1 (Jan. 1997), 29–53. https://doi.org/10.1023/A:1009726021843

[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] Mike Heddes, Igor Nunes, Tony Givargis, and Alex Nicolau. 2024. Convolution and Cross-Correlation of Count Sketches Enables Fast Cardinality Estimation of Multi-Join Queries. *Proc. ACM Manag. Data* 2, 3, Article 129 (May 2024), 26 pages. https://doi.org/10.1145/3654932

[25] 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

[26] 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.

[27] 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

[28] 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).

[29] 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

[30] 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

[31] 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

[32] 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

[33] 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/

[34] 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

[35] 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/

[36] 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

[37] Beibin Li, Yao Lu, Chi Wang, and Srikanth Kandula. 2021. Cardinality Estimation: Is Machine Learning a Silver Bullet?. In *AIDB*. https://www.microsoft.com/en-us/research/publication/cardinality-estimation-is-machine-learning-a-silver-bullet/

[38] Pengfei Li, Wenqing Wei, Rong Zhu, Bolin Ding, Jingren Zhou, and Hua Lu. 2023. ALECE: An Attention-based Learned Cardinality Estimator for SPJ Queries on Dynamic Workloads. *Proc. VLDB Endow.* 17, 2 (Oct. 2023), 197–210. https://doi.org/10.14778/3626292.3626302

[39] 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

[40] 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

[41] 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

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

[43] 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

[44] 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

[45] 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

[46] 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

[47] Microsoft. [n.d.]. Microsoft SQL Server: Common Language Runtime (CLR) Integration. https://learn.microsoft.com/en-us/sql/relational-databases/clr-integration/common-language-runtime-integration-overview?view=sql-server-ver17

[48] 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

[49] 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

[50] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1520–1533. https://doi.org/10.14778/3583140.3583164

[51] 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

[52] Oracle. [n.d.]. Oracle Database: External Procedures. https://docs.oracle.com/en/database/oracle/oracle-database/19/ntqrf/external-procedures-overview.html

[53] 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.

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

[55] 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

[56] 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.

[57] 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

[58] 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

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

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

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

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

[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 ℓ_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