A Backend-Agnostic Compiler for Approximate Query Processing with Probabilistic Tensor Algebra

JINGWEN PAN, The University of Edinburgh, United Kingdom
JAMES CHENEY, The University of Edinburgh, United Kingdom
AMIR SHAIKHHA, The University of Edinburgh, United Kingdom

Despite decades of research, the practical adoption of Approximate Query Processing (AQP) remains limited. Sampling-based systems such as VerdictDB often yield slow or inaccurate results for complex analytical queries, as their effectiveness depends heavily on sample quality. Model-based systems such as DeepDB perform well on simple queries but incur high runtime latency for multiway joins and group-by operations, as each group is evaluated as an independent query with instantiated attribute values, preventing global optimization across groups. Both approaches are tied to fixed hardware backends, limiting flexibility across CPU and GPU platforms.

This paper presents BayesAQP, a backend-agnostic AQP framework that compiles Bayesian Network (BN) inference into tensor algebra (TA) programs. During training, BayesAQP models per-table statistics using BNs and integrates join histograms for efficient multi-table queries. At runtime, SQL queries are compiled into TA programs in BTL++, a tensor-centric probabilistic programming language, and optimized into tensor contractions for execution on tensor processing frameworks across CPUs and GPUs. Experimental results show that BayesAQP achieves an average 37× speed-up over existing AQP systems on three real-world datasets while maintaining comparable accuracy and significantly improving performance for complex multiway join and group-by queries.

CCS Concepts: • **Information systems** → **Query optimization**.

Additional Key Words and Phrases: Approximate Query Processing; Query Compilation; Probabilistic Programming; Bayesian Networks; Tensor Algebra

ACM Reference Format:
Jingwen Pan, James Cheney, and Amir Shaikhha. 2026. A Backend-Agnostic Compiler for Approximate Query Processing with Probabilistic Tensor Algebra. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 126 (June 2026), 25 pages. https://doi.org/10.1145/3802003

1 Introduction

The exponential growth of data has made exact query processing increasingly expensive. Approximate Query Processing (AQP) mitigates this by trading accuracy for efficiency, enabling near-real-time analytics over large datasets, with state-of-the-art approaches falling into two main categories: sampling-based and machine-learning-based (ML-based) systems.

Sampling-based systems such as VerdictDB [28] and WanderJoin [21] approximate query results by operating on partial data. While they support a broader range of query types and integrate easily with existing DBMSs, their runtime performance degrades on analytical queries involving multiple `GROUP BY` attributes or joins, and their accuracy suffers when maintaining sufficiently representative samples becomes infeasible.

Authors' Contact Information: Jingwen Pan, J.Pan-15@sms.ed.ac.uk, The University of Edinburgh, United Kingdom; James Cheney, jcheney@inf.ed.ac.uk, The University of Edinburgh, United Kingdom; Amir Shaikhha, amir.shaikhha@ed.ac.uk, The University of Edinburgh, United Kingdom.

ACM 2836-6573/2026/6-ART126
https://doi.org/10.1145/3802003

ML-based systems [14, 24, 25] instead learn predictive models offline to estimate query results. These models are typically more compact than sampling-based approaches but face the same limitations in efficiently processing group-by queries. DeepDB [14], for instance, decomposes each group-by query into many equality-constrained subqueries executed independently at runtime, similar to WanderJoin. This design prevents shared computation and global optimization across groups, leading to poor scalability as the number of grouping attributes or the domain cardinality increases. Moreover, existing AQP systems are primarily CPU-bound and cannot benefit from the parallelism offered by modern GPU or tensor-based computation frameworks, leaving the efficient handling of complex analytical queries an open challenge.

Recent advances in model-based ML [5, 13, 29, 41] have renewed interest in probabilistic models for data analysis. Bayesian Networks (BNs), a class of probabilistic graphical models (PGMs), capture the intrinsic data dependency and offer compactness, interpretability, and robustness to data changes [10], with successful applications across biomedicine [3], astrophysics [31], and geoscience [39].

However, several challenges hinder the use of BNs for AQP. Exact inference over discrete distributions is computationally expensive and remains an active research problem [7, 15]. Existing database applications of BNs, such as BayesCard [42] and FactorJoin [43], are confined to cardinality estimation (CardEst), a restricted form of AQP. To date, there is no well-developed architecture for employing BNs for more general forms of AQP due to the NP-hard model construction and the computational cost of exact inference [7, 15].

In this paper, we propose BayesAQP, a backend-agnostic framework for AQP that supports fast approximation over large-scale data. BayesAQP combines the advantages of recent advances in probabilistic programming and high-performance computing by using a probabilistic programming language (PPL) to represent the probabilistic queries compiled from SQL queries and leveraging advanced tensor processing frameworks (e.g., TensorFlow [1] and PyTorch [30]) to solve the resulting inference problems. BayesAQP learns relational data as discrete BNs and translates SQL aggregations into probabilistic queries. By compiling BN inference into tensor computations within tensor-processing backends, BayesAQP leverages vectorization and parallelization for efficient observation incorporation and variable elimination during inference. In addition, we propose BTL++, a tensor-centric PPL built upon BTL [27], both designed for efficient exact inference over discrete distributions. While BTL focuses on general Bayesian Network inference, BTL++ extends its expressiveness by supporting additional tensor constructs, such as expectation and outer product, that enable the representation of database aggregations and join operations. Specifically, BTL++ compiles SQL queries with aggregations (e.g., `COUNT`, `AVG`, `SUM`) into tensor algebra (TA) programs, which are symbolic representations of multi-dimensional tensor operations. By leveraging these constructs, BTL++ expresses relational inner joins and SQL aggregations as tensor computations. To further enhance runtime performance, BayesAQP applies compilation-level optimizations on the resulting TA programs, enabling efficient execution on NumPy CPU or PyTorch GPU backends. As our experimental results show, BayesAQP offers significant advantages both for accuracy and performance compared to state-of-the-art systems such as DeepDB and VerdictDB, specifically for queries with multiway joins and multiple grouped attributes over large attribute domains.

Contributions. Our main contributions are as follows:

- We formulate AQP queries into probabilistic queries using conditional distributions of data attributes, including single-table and multi-table scenarios (Sections 2–3).
- We propose BayesAQP, a backend-agnostic compiler prototype that supports fast approximation of SQL aggregations over large-scale data by leveraging BTL++, an extension of BTL [27],

a tensor-centric PPL for fast exact inference over discrete distributions, and advanced tensor processing frameworks through a compilation pipeline (Section 4).

- We demonstrate how BayesAQP optimizes translated TA programs from SQL queries through multiple compilation phases that enable efficient execution on tensor processing backends (Section 5).
- We show that BayesAQP outperforms existing AQP systems, achieving an average 37× speed-up on real-world datasets while maintaining high accuracy, particularly for complex multiway joins and group-by queries (Section 6).

2 Background: Probabilistic Programming

This section briefly reviews two foundations of BayesAQP: probabilistic graphical models and probabilistic programming languages.

2.1 Probabilistic Graphical Models

Probabilistic graphical models (PGMs) capture statistical dependencies among random variables using graphs in which nodes represent variables and edges encode conditional relationships. PGMs are broadly categorized into directed models, such as BNs, and undirected models, such as Markov Random Fields (MRFs). BayesAQP focuses on BNs because it compiles queries into BTL++ programs that enable efficient exact inference for directed models, whereas no comparably fast inference backend exists for undirected models.

Bayesian Networks. A Bayesian Network is a directed acyclic graph (DAG) whose nodes represent random variables associated with conditional probability distributions (CPDs) conditioned on their parents. Given a BN with n variables $\mathbf{x} = \{x_1, \ldots, x_n\}$ and evidence $\mathbf{e} = \{e_1, \ldots, e_k\}$, the likelihood of $\mathbf{e}$ is computed via variable elimination (VE) [44]:

$$\mathbb{P}(\mathbf{e}) = \sum_{\mathbf{x} \setminus \mathbf{x_e}} \prod_{i=1}^{n} \mathbb{P}(x_i \mid \mathbf{pa_i}), \tag{1}$$

where $\mathbf{pa_i}$ denotes the parents of x_i, and $\mathbf{x_e}$ the variables instantiated by $\mathbf{e}$. The marginal distribution of a variable x_k given $\mathbf{e}$ is:

$$\mathbb{P}(x_k \mid \mathbf{e}) = \frac{1}{Z} \sum_{\mathbf{x} \setminus (\{x_k\} \cup \mathbf{e})} \prod_{i=1}^{n} \mathbb{P}(x_i \mid \mathbf{pa_i}), \tag{2}$$

where $Z = \mathbb{P}(\mathbf{e})$ normalizes the distribution. Section 3 formulates AQP queries as inference over BNs.

2.2 Probabilistic Programming Languages

Probabilistic programming languages provide high-level abstractions for specifying PGMs and automating inference under uncertainty, typically supporting posterior inference in Bayesian models as well as inference tasks beyond marginal queries over PGMs [40]. Analogous to automatic differentiation in deep learning, PPLs reduce the cost of implementing and experimenting with inference algorithms. Existing PPLs span a wide design space, from generic inference-oriented languages [2, 15, 33, 35] to database-oriented systems [4, 16] that emphasize probabilistic query semantics and efficient execution in database settings. BayesAQP lies between these extremes, adopting a system-oriented design that leverages probabilistic programming as a compilation target rather than a user-facing language, enabling efficient execution of inference-driven AQP workloads.

$$\begin{array}{lll} \mathsf{q} & ::= & \text{SELECT } \mathbf{e} \\ & & \text{FROM } t_1, t_2, \ldots, t_n \\ & & \text{WHERE } c \\ & & \text{GROUP BY } \mathbf{a}; \end{array} \qquad \begin{array}{lll} c & ::= & t.A = u.B \mid t.A \in [x, y] \mid t.A \in S \mid c \wedge c' \\ e & ::= & t.A \mid \text{COUNT}(*) \mid \text{AVG}(t.A) \mid \text{SUM}(t.A) \\ & \mid & \text{SUM}(t.A \cdot t.B) \mid \text{SUM}(t.A) - \text{SUM}(t.B) \end{array}$$

Fig. 1. SQL query template supported by BayesAQP, where e, $\mathbf{a}$, and c denote aggregations, group-by attributes, and query predicates (including join conditions), respectively. S is a finite set of domain values, and x and y are constants.

In particular, BayesAQP builds on BTL [27], a tensor-centric PPL designed for efficient exact inference over discrete probabilistic programs, including BNs, making it a strong foundation for BayesAQP. Since database aggregations and joins are not natively supported in BTL, we extend it to BTL++, a tensor-centric PPL specialized for AQP workloads.

3 Modeling AQP with Discrete Distributions

This section formalizes how BayesAQP models AQP using BNs for single-table and multi-table workloads from a statistical perspective.

3.1 Supported Queries

Figure 1 shows the SQL template supported by BayesAQP, including aggregations such as `COUNT`, `AVG`, `SUM`, and `GROUP BY`. `AVG` and `SUM` operate on continuous attributes, while `GROUP BY` applies to categorical attributes. BayesAQP supports equi-joins and selection predicates, including `IN` clauses, equality conditions (i.e., =), range filters (e.g., <, >, ≤, ≥), and conjunctions of such predicates.

3.2 Problem Formulation

Single-table CardEst. Given a database with relations $A, B, \ldots, Z$ under schema $\mathcal{J}$ and a query Q, the cardinality estimation on relation A is defined as

$$|Q| = sel(Q) * |A|,$$

where $sel(Q)$ is the query selectivity and $|A|$ is the table cardinality. In BayesAQP, $sel(Q)$ is computed as $\mathbb{P}_A(Q)$, the joint probability of the evidence Q inferred from the BN of relation A.

Case 1: Single-table CardEst with `GROUP BY`. For relation A with attributes $col(A) = \{aid, a_2, \ldots, a_n\}$ and a query Q grouping by attributes $\mathbf{a} \subseteq col(A)$,

$$\mathbf{r} \approx \mathbb{P}_A(\mathbf{a} \mid Q) * |A|, \tag{3}$$

where the tensor order of $\mathbf{r}$ corresponds to $|\mathbf{a}|$, and $\mathbb{P}_A(\mathbf{a} \mid Q)$ is the unnormalized marginal distribution of the group-by attributes inferred from the BN of relation A.

To efficiently handle large-scale and continuous data, BayesAQP discretizes each attribute $a \in col(A)$ into equi-depth histogram buckets $\varphi_a = \{\alpha_1, \ldots, \alpha_m\}$ over its domain $\mathcal{D}_a$. The group-by result in Equation 3 becomes

$$\begin{aligned} \mathbf{r}' &= \mathbb{P}_A(\mathbf{a}' \mid Q)^{(*)} * |A|, \\ \mathbf{r} &\approx \gamma(\mathbf{r}'), \end{aligned} \tag{4}$$

where $\mathbf{a}'$ denotes the discretized form of $\mathbf{a}$, each variable's states corresponding to its histogram buckets. $\mathbb{P}_A^{(*)}$ represents the marginal distribution of the discretized group-by attributes, weighted by query-dependent vectors when predicates partially overlap with histogram buckets of relation A. Inference in BayesAQP operates in this discretized space, producing tensors indexed by histogram buckets rather than raw values. To obtain meaningful query results, the reverse-mapping function

Algorithm 1 Offline Pairwise Correlation Matrix Construction

Require: Tables A and B joined on $A.v = B.v$; equi-depth histograms $\varphi_{A.v}, \varphi_{B.v}$; domains $\mathcal{D}_{A.v}, \mathcal{D}_{B.v}$
Ensure: Correlation matrix $\rho_{AB} \in \mathbb{R}^{n \times k}$

$U \leftarrow \mathcal{D}_{A.v} \cup \mathcal{D}_{B.v}$; let $U = (u_1, \ldots, u_m)$
$n \leftarrow |\varphi_{A.v}|, \quad k \leftarrow |\varphi_{B.v}|$
Initialize $\mathbf{W}_{A.v} \in \mathbb{R}^{n \times m}, \quad \mathbf{W}_{B.v} \in \mathbb{R}^{k \times m}$
for $i = 1$ to n **do**
  for $j = 1$ to m **do**
    $\mathbf{W}_{A.v}[i, j] \leftarrow \dfrac{\left|\{\, t \in \varphi_{A.v}[i] : t = u_j \,\}\right|}{\left|\varphi_{A.v}[i]\right|}$
  end for
end for
Construct $\mathbf{W}_{B.v}$ analogously using $\varphi_{B.v}$
$\rho_{AB} \leftarrow \mathbf{W}_{A.v} \times \mathbf{W}_{B.v}^{\top}$
return ρ_{AB}

γ expands each bucket into the domain values contained within it using precomputed mappings, restoring outputs to the original value space.

Case 2: CardEst with Pairwise Inner Join. The correlation matrix ρ models the overlap between histogram buckets of different relations, thereby estimating join selectivity between attributes (cf. Algorithm 1). Given a query Q joining A and B on $A.aid = B.bid$, with base predicates $Q(A)$ and $Q(B)$, the estimated join cardinality is

$$|A \bowtie B| \approx \mathbb{P}_A(aid \mid Q(A))^{(*)} * |A| * \rho_{AB} * (\mathbb{P}_B(bid \mid Q(B))^{(*)})^{\top} * |B|, \tag{5}$$

where $\mathbb{P}_A(aid \mid Q(A))^{(*)}$ and $\mathbb{P}_B(bid \mid Q(B))^{(*)}$ are the marginals of the join attributes under their respective predicates, represented over the discretized domain of the corresponding attributes. This formulation generalizes naturally to multi-way joins and group-by queries: star or chain joins can be composed through successive pairwise correlations, and group-by joins extend by inferring joint marginals over the union of group-by and join attributes. In practice, BayesAQP currently supports star schemas and can be extended to chain joins via composed correlation matrices.

AVG queries. An average aggregation in SQL corresponds to computing the expected value of an attribute conditioned on the query predicates. For a query Q with $\texttt{AVG}(V)$ on relation T:

$$\mathbb{E}(V \mid Q) \approx \mathbb{P}_T(V \mid Q)^{(*)}_{\text{norm}} * \text{Vec}_V, \tag{6}$$

where $\mathbb{P}_T(V \mid Q)^{(*)}_{\text{norm}}$ is the normalized marginal distribution of V under Q, and Vec_V enumerates the possible values (or bucket means if discretized).

When V is an arithmetic expression over multiple attributes (e.g., $\texttt{AVG}(T.v_1 * T.v_2)$), inference yields a joint marginal:

$$\mathbb{E}(v_1 * v_2 \mid Q) \approx \mathbb{P}_T(v_1, v_2 \mid Q)^{(*)}_{\text{norm}_{v_1,v_2}} * (\text{Vec}_{v_1} \otimes \text{Vec}_{v_2}), \tag{7}$$

where $\otimes$ denotes the outer product, and normalization applies over the v_1, v_2 dimensions.

For queries with group-by attributes $\mathbf{g} \subseteq col(T)$ and aggregated attributes $\mathbf{v} = v_1, \ldots, v_n$:

$$\begin{aligned} \mathbf{r}' &\approx \mathbb{P}_T(\mathbf{g} \cup \mathbf{v} \mid Q)^{(*)}_{\text{norm}_{\mathbf{v}}} *_{(\mathbf{v})} (\text{Vec}_{v_1} \otimes \cdots \otimes \text{Vec}_{v_n}), \\ \mathbf{r} &\approx \gamma(\mathbf{r}'). \end{aligned} \tag{8}$$

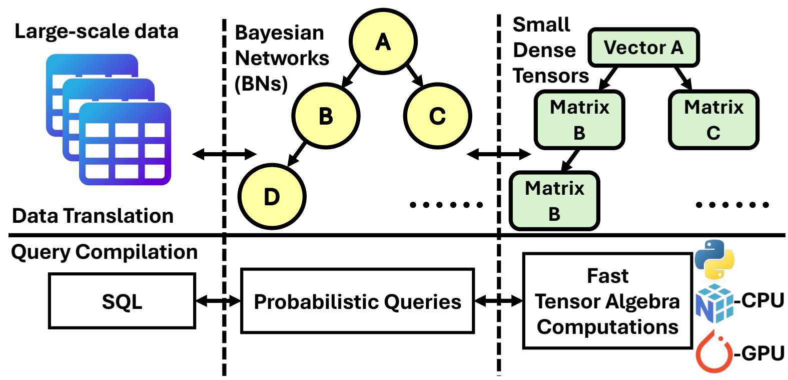


Fig. 2. The high-level architecture of BayesAQP.

```
1   SELECT [u1.A1,...,um.Am], AGG
2   FROM t1,...,tn
3   WHERE t1.B1 = t2.C1
4     AND ...
5     AND tn-1.Bn-1 = tn.Cn-1
6     AND c
7   [GROUP BY u1.A1,...,um.Am];
```

(a) The SQL template, where lines 3-7 and group-by attributes in [...] are optional.

$$
\begin{aligned}
f_{a_i} &:= \text{select } a_i\ [\overline{r}]\\
a_{i_{\text{weight}}} &:= \text{vector } [\overline{r}]\\
w_{a_i} &:= \text{weight } a_i\ a_{i_{\text{weight}}}\\
o_{a_i} &:= \text{observe } a\ [\overline{r}]\\
dist_{t_j} &:= \text{query } [\overline{a_i}]\ [\overline{f_{a_i}}]\ [\overline{o_{a_i}}]\ [\overline{w_{a_i}}]\\
result\_dist_j &:= dist_{t_j}
\end{aligned}
$$

(b) Base relation actions translated in BTL++.

(JOIN, optional)

$FOR\ j \in 1 \ldots n-1$

$$
\begin{aligned}
prep_j &:= \text{outer } result\_dist_j\ dist_{j+1}\\
result\_dist_{j+1} &:= \text{expectation } [\overline{a_i}]\ prep_j\ \rho_j\ [x, y]
\end{aligned}
$$

(w/o optional joins)

$$dist := result\_dist_n$$

(c) Translated pairwise joins depend on the star schema in BTL++.

(COUNT)

$$count := dist * nrows$$

(AVG)

$$
\begin{aligned}
m &:= \text{loadTensor}(m_{name}, \overline{m_d})\\
norm_m &:= \text{normalize } dist\ [\overline{r}]\\
avg &:= \text{expectation } [\overline{a_i}]\ norm_m\ m\ [\overline{r}]
\end{aligned}
$$

(SUM)

$$sum := \text{mul}(count, avg)$$

(d) Translated aggregations in BTL++.

Fig. 3. Transformation mappings from SQL to BTL++, including base relation actions, pairwise join for star schema, and different aggregations. Group-by attributes are tracked by the first input of both the query and the expectation constructs.

Here, $*_{(\mathbf{v})}$ denotes a contraction summing over the aggregated dimensions $\mathbf{v}$. As inference operates in a discretized space, $\mathbf{r}'$ is mapped back to domain values via γ.

SUM queries. A `SUM` query is evaluated as the element-wise product of its corresponding `COUNT` and `AVG` queries that share identical predicates.

4 System Overview

This section presents the overall architecture of BayesAQP (Section 4.1) and introduces its probabilistic query compilation workflow in Sections 4.2-4.3.

Example 1. Throughout this section, we use a simplified version of the Star Schema Benchmark (SSB) [26], named *MiniSSB*, along with three representative SQL queries, to illustrate BayesAQP's compilation pipeline (Sections 4.2-4.3) as shown in Figures 5-7. Figure 5 shows how to perform a cardinality estimation query, while Figure 6 presents how BayesAQP can answer a group-by average query with a two-way join. The group-by sum query with a join of 4 tables defined in Figure 7 shows how effective the compilation phases can be, reflecting optimizations on performance.

The MiniSSB dataset follows the star schema of SSB (cf. Figure 7a), with `lineorder` as the fact table joined to four dimension tables: `dwdate`, `supplier`, `customer`, and `part`. It is a reduced-scale variant of SSB used for illustration, where categorical attributes have at most two distinct values and continuous attributes up to four. MiniSSB is used solely for demonstration; in practice, BayesAQP scales to real-world datasets (Section 6.3).

4.1 System Architecture

The high-level architecture of BayesAQP is illustrated in Figure 2. BayesAQP translates data in relational form into BNs during the offline training phase. In the online inference phase, BayesAQP approximates the input SQL queries by compiling them into probabilistic queries in BTL++ using BNs. The translated BTL++ programs are then used to generate optimized TA programs, enabling efficient execution within advanced tensor processing frameworks.

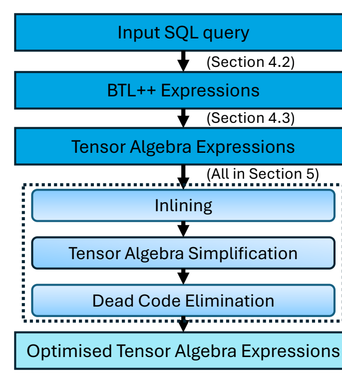


Fig. 4. An overview of transformations from the input SQL query to an optimized tensor algebra program in BayesAQP.

Data Translation. For each relation, BayesAQP constructs a BN to capture dependencies among attributes, addressing the independence assumption inherent in traditional histogram construction. In BayesAQP, each node represents the conditional distribution of a data attribute, materialized as a dense tensor. During offline training, BayesAQP learns discrete BNs using Chow–Liu tree structures [8] via `pgmpy` [2]. BayesAQP leverages `pgmpy` as an off-the-shelf library for learning discrete BNs. Specifically, we use `pgmpy`'s `TreeSearch` estimator to learn a Chow–Liu tree structure, followed by maximum-likelihood estimation to derive the corresponding conditional distributions. To support efficient inference for join queries, BayesAQP enforces the primary key of each table as the root of the learned tree. Our contribution does not lie in BN structure or parameter learning; instead, it lies in how these learned models are oriented and integrated into a tensor compiler to support efficient inference for AQP during runtime.

Model Expressiveness and Inference Tractability. Chow–Liu trees provide a maximum-likelihood tree approximation that preserves all pairwise mutual information, yielding a principled trade-off between modeling accuracy and inference complexity. By construction, this restricts expressiveness to pairwise dependencies and excludes higher-order interactions that would otherwise induce high treewidth and intractable inference. In BayesAQP, this restriction is intentional: it captures the dominant correlations affecting selectivity estimation while ensuring linear-time marginal inference and avoiding high-order tensor materialization.

At runtime, inference is further optimized through graph reduction guided by the BN topology encoded in compiled BTL++ programs. While exact inference in general BNs is #P-hard and exponential in treewidth, BayesAQP avoids this blow-up by combining tree-structured intra-table models with explicit inter-table dependencies handled via pairwise correlation matrices

(Algorithm 1). Although BTL++ may denote higher-order tensor expressions (e.g., from multi-attribute group-by queries), these tensors are never materialized: the compiler applies compiler optimizations (Section 5) to rewrite them into sequences of low-order tensor contractions, ensuring that all executed tensors have order at most two. As a result, inference cost is dominated by tensor operations whose complexity depends on join and group-by dimensionality, which we evaluate empirically in Section 6. A formal complexity analysis of the compiled tensor programs is left as future work.

Probabilistic Query Compilation. Figure 4 overviews the transformation pipeline in BayesAQP. BayesAQP translates the input SQL query into a probabilistic query in BTL++ over trained BNs, which is then compiled into a TA program. To enhance performance, BayesAQP applies compiler optimizations, such as inlining and algebraic simplification, producing an efficient TA program for execution on NumPy (CPU) or PyTorch (GPU) backends. Sections 4.2–4.3 illustrate this workflow using two representative SQL queries (Figures 5–6), covering the transformations from SQL to BTL++ and from BTL++ to TA.

4.2 SQL to BTL++

The front end of BayesAQP translates SQL queries into probabilistic programs in BTL++, which extends BTL [27] with probabilistic operators and tensor constructs (e.g., expectation and outer product) to support SQL aggregations and joins. Key BTL++ constructs for expressing SQL queries are shown in Figure 3, where the components of a generated BTL++ program follow the fixed order shown in Figures 3b–3d.

BTL++ VS. BTL. BTL++ extends BTL from a PPL into a compiler-based framework for AQP over relational data. While BTL was originally designed for exact inference over discrete distributions, it is limited to independent inference tasks and lacks constructs to express relational analytical semantics such as joins, aggregations, and group bys.

BTL++ introduces three key extensions over BTL. First, it generalizes the supported AQP query class from single-table cardinality estimation to relational analytical queries, including multi-table joins and a broader range of aggregate operators such as sum, average, and group bys. Second, BTL++ redesigns the intermediate representation in BTL, called TIL, into a tensor-algebra IR (cf. Figure 7c) that enables compiler optimizations essential for efficient AQP query execution, including inlining, algebraic simplification, and dead-code elimination (Section 5). Third, BTL++ extends the execution backend to use sparse tensor representations and alternative execution backends (e.g., CPU and GPU), enabling better scalability and memory efficiency for large conditional distributions. These extensions transform BTL from a probabilistic inference language into an efficient AQP compiler.

Example 2: `COUNT` on a Single Table without `GROUP BY`. Figure 5 illustrates an example of single-table cardinality estimation by compiling an SQL query (Figure 5a) into an equivalent BTL++ program (Figure 5b).

BTL++ provides the `select`, `observe`, `vector`, and `weight` constructs to encode SQL selection predicates. The `select` construct performs unnormalized filtering over histogram buckets, while `observe` applies conditioning with automatic normalization. The `vector` and `weight` constructs define bucket-level correction weights (e.g., $\mathbf{W}_A$ in Figure 5c), precomputed to handle partial predicate overlaps based on bin coverage, and integrate them into BN distributions; weight generation is elided when all weights equal one. At runtime, BN distributions are accessed via `loadTensor` (cf. Figure 3d) and weight vectors are instantiated via `vector`, both omitted in Figure 5b for clarity.

In Figure 5b, lines 1–2 apply unnormalized conditioning and weighting correction to `lo_ordertotalprice`. Line 3 computes the joint distribution over the BN of the lineorder table using the `query` construct, where an empty marginalization list yields the joint distribution. Finally, line 4

```sql
SELECT COUNT(*)
FROM lineorder
WHERE lo_ordertotalprice>3000000;
```

(a) The SQL representation of the example query.

```
let filter1 = select lo_ordertotalprice 0:3 in
let weight1 = weight lo_ordertotalprice
      lo_ordertotalprice_weight in
let prob = query [] [filter1] [] [weight1] in
count = prob * nrows
```

(b) Translated BTL++ program.

$$Count := A(a=[0,1,2,3],r) * W_A(a=[0,1,2,3]) * R(r,l) * L(l) * nrows$$

(c) Equivalent tensor algebra expression based on VE.

```python
prob = contract('ar,a,rl,l,->',
      lo_ordertotalprice[[0,1,2,3]],
      lo_ordertotalprice_weight[[0,1,2,3]],
      lo_revenue,lo_orderkey)
count = prob * nrows
```

(d) Generated Python code from the tensor algebra expression.

Fig. 5. Compilation pipeline for a count query over the lineorder table from the MiniSSB dataset in BayesAQP.

```sql
SELECT c_city, AVG(lo_ordertotalprice)
FROM lineorder, customer
WHERE lo_custkey = c_custkey
GROUP BY c_city;
```

(a) The SQL representation of the example query.

```
let customer = query [c_custkey, c_city] [] []
      [] in
let lineorder = query [lo_custkey,
      lo_ordertotalprice] [] [] [] in
let prep1 = outer lineorder customer in
let join1 = expectation [lo_ordertotalprice,
      c_city] prep1 rho1 [0, 2] in
let Norm = normalize join1 [1] in
avgGB = expectation [c_city] Norm
      lo_ordertotalprice_mean [0]
```

(b) Translated BTL++ program for SQL query in Figure 6a.

$$\begin{aligned}
customer(c,t) &:= T(t,e) * E(e,c) * C(c)\\
lineorder(k,a) &:= K(k,a) * A(a,r) * R(r,l) * L(l)\\
prep1(k,a,c,t) &:= lineorder(k,a) * customer(c,t)\\
join1(a,t) &:= prep1(k,a,c,t) * rho1(k,c)\\
Joint(t) &:= prep1(k,a,c,t) * rho1(k,c)\\
Norm(a,t) &:= join1(a,t)/Joint(t)\\
avgGB(t) &:= Norm(a,t) * A_{mean}(a)
\end{aligned}$$

(c) Equivalent tensor algebra expression based on VE.

```python
customer = contract('te,ec,c->ct',c_city,c_phone
      ,c_custkey)
lineorder = contract('ka,ar,rl,l->ka',lo_custkey
      ,lo_ordertotalprice,lo_revenue,lo_orderkey)
prep1 = contract('ct,ka->ctka',customer,
      lineorder)
join1 = contract('ctka,kc->at',prep1,rho1)
joint = contract('kact,kc->t',prep1,rho1)
norm = join1/joint
avgGB = contract('at,a->t',norm,
      lo_ordertotalprice_mean)
```

(d) Generated Python code from the tensor algebra expression.

Fig. 6. Compilation pipeline for an average query with an inner join between two tables of the MiniSSB dataset in BayesAQP. The structures of the models are shown in Figure 7a.

multiplies the inferred joint probability by `nrows`, the table cardinality, to produce the final `count` result.

Example 3: `GROUP-BY AVG` with inner join of 2 tables. An SQL `AVG` query is translated into a probabilistic query that computes expectations over aggregated attributes, while the `GROUP BY` clause corresponds to marginalization over group-by attributes. In BayesAQP, pairwise inner joins are optional and restricted to star schemas; when present, joins are executed as a sequence of pairwise inner joins (Figure 3c). Joins and aggregations are expressed uniformly using the **expectation** construct in BTL++, together with the **outer** construct to form outer products.

For a BN node X with n states, the expectation is defined as $\mathbb{E}[X] = \sum_{i=1}^{n} x_i p_i$, where p_i denotes the conditional probability of the ith state and x_i is the mean value of the corresponding histogram bin. In BTL++, this is expressed as **expectation** $[\overline{a_i}]$ $prep_j$ ρ_j `[x,y]`, which performs element-wise multiplication between the x-th dimension of $prep_j$ and the y-th dimension of ρ_j to aggregate join-key axes, while retaining group-by attributes $\overline{a_i}$ when specified. When the fourth input is empty, **expectation** reduces to a conventional expectation over the input distribution.

Figure 6 illustrates the compilation of an SQL `AVG` query with a `GROUP BY` clause and an inner join between the `lineorder` and `customer` tables. In Figure 6b, Lines 1–2 perform base relation inference, and line 3 computes the outer product of intermediate results, leveraging tensor operations and Einstein summation notation to streamline contractions. Line 4 applies the inner join via **expectation** while preserving the group-by attribute. Line 5 normalizes the inferred joint distribution over the group-by dimension `c_city`, and line 6 computes the expected value using `lo_ordertotalprice_mean`, a vector of per-bucket mean values precomputed offline from the histogram of `lo_ordertotalprice`.

4.3 BTL++ to Tensor Algebra

To leverage optimized tensor-processing backends and enable compiler optimizations, BayesAQP lowers BTL++ programs into equivalent TA programs. The BTL++ definition of a BN (i.e., its CDTs) can be viewed as a collection of tensors. Probabilistic queries are evaluated via variable elimination, which naturally translates into tensor contraction (Einstein summation) expressions. In practice, these contractions are executed efficiently using opt_einsum [38] across CPU and GPU backends.

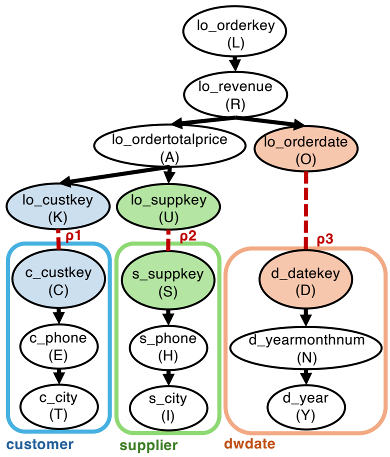
In each TA expression, tensors connected by the asterisk ($*$) participate in the contraction. Dimension symbols associated with each term specify tensor indices, while a slash (/) denotes element-wise division. When multiple tensors share the same dimension symbol, element-wise multiplication is performed along that dimension, followed by summation over the contracted dimensions as dictated by the expression; division is applied along shared dimensions between numerator and denominator tensors. Query selections over data attributes are encoded as constraints within tensors representing the corresponding conditional distributions.

Based on TA expressions, BayesAQP emits executable Python code targeting the selected backend. Following NumPy's convention, symbols on the right-hand side of the arrow specify the index order of input tensors (comma-separated), while the symbols after the arrow define the index order of the resulting tensor.

Example 2 (cont.). Figure 5c shows the equivalent tensor algebra expression of our count without group by example. Based on this tensor algebra expression, BayesAQP generates the corresponding tensor computation in the NumPy backend, presented in Figure 5d. In Figure 5d, line 1 expresses the tensor contraction, where the symbol `rl` at the left side of the arrow represents the order of the conditional distribution of `lo_revenue`. Query selections over `lo_ordertotalprice`, `lo_ordertotalprice_weight` (i.e., the weight of `lo_ordertotalprice`) are tensor slicing (e.g., `lo_ordertotalprice[[0, 1, 2, 3]]`) at lines 1-2.

5 Compiler Optimizations

This section presents the compiler optimizations that BayesAQP applies across multiple phases to improve the efficiency of TA programs generated from SQL queries. All examples refer to executions on the NumPy CPU backend; running on PyTorch requires minor configuration adjustments.




```sql
SELECT c_city, s_city, d_year, SUM(
    lo_revenue)
FROM lineorder, dwdate, supplier,
    customer
WHERE lo_orderdate = d_datekey AND
    lo_suppkey = s_suppkey AND
    lo_custkey = c_custkey
GROUP BY c_city, s_city, d_year;
```

(a) The SQL representation of a group-by sum query using the MiniSSB dataset.

```
let customer = query [c_custkey, c_city] [] [] []
    in
let supplier = query [s_suppkey, s_city] [] [] []
    in
let dwdate = query [d_datekey, d_year] [] [] [] in
let lineorder = query [lo_custkey, lo_suppkey,
    lo_orderdate, lo_revenue] [] [] [] in
let prep1 = outer lineorder customer in
let join1 = expectation [lo_suppkey, lo_orderdate,
     lo_revenue, c_city] prep1 rho1 [0, 4] in
let prep2 = outer join1 supplier in
let join2 = expectation [lo_orderdate, lo_revenue,
     c_city, s_city] prep2 rho2 [0, 4] in
let prep3 = outer join2 dwdate in
let join3 = expectation [lo_revenue, c_city,
    s_city, d_year] prep3 rho3 [0, 4] in
let join3Norm = normalize join3 [1, 2, 3] in
let avgGB = expectation [c_city, s_city, d_year]
    join3Norm lo_revenue_mean [0] in
let ones = unitMat join3 in
let countProb = expectation [c_city, s_city,
    d_year] join3 ones [0, 1, 2, 3] in
let countGB = countProb * nrows in
let sumGB = avgGB * countGB in
```

(b) The translated BTL++ program of the example in Figure 7a.

$$
\begin{aligned}
customer(c,t) &:= T(t,e) * E(e,c) * C(c)\\
supplier(s,i) &:= I(i,h) * H(h,s) * S(s)\\
dwdate(d,y) &:= Y(y,n) * N(n,d) * D(d)\\
lineorder(k,u,o,r) &:= K(k,a) * U(u,a) * A(a,r)\\
&\quad * O(o,r) * R(r,l) * L(l)\\
Prep1(k,u,o,r,c,t) &:= lineorder(k,u,o,r)\\
&\quad * customer(c,t)\\
join1(u,o,r,t) &:= prep1(k,u,o,r,c,t) * rho1(k,c)\\
prep2(u,o,r,t,s,i) &:= join1(u,o,r,t) * supplier(s,i)\\
join2(o,r,t,i) &:= prep2(u,o,r,t,s,i) * rho2(u,s)\\
prep3(o,r,t,i,d,y) &:= join2(o,r,t,i) * dwdate(d,y)\\
join3(r,t,i,y) &:= prep3(o,r,t,i,d,y) * rho3(o,d)\\
Joint(t,i,y) &:= prep3(o,r,t,i,d,y) * rho3(o,d)\\
Norm(r,t,i,y) &:= join3(r,t,i,y) / Joint(t,i,y)\\
avgGB(t,i,y) &:= Norm(r,t,i,y) * R_{mean}(r)\\
Ones(r,t,i,y) &:= UnitMat(r,t,i,y)\\
countProb(t,i,y) &:= Joint(r,t,i,y) * Ones(r,t,i,y)\\
countGB(t,i,y) &:= countProb(t,i,y) * nrows\\
sumGB(t,i,y) &:= avgGB(t,i,y) * countGB(t,i,y)
\end{aligned}
$$

(c) Equivalent TA expressions based on VE (red terms indicate where inlining occurs).

After O1:

$$
\begin{aligned}
join3(r,t,i,y) &:= K(k,a) * U(u,a) * A(a,r) * O(o,r)\\
&\quad * R(r,l) * L(l) * T(t,e) * E(e,c)\\
&\quad * C(c) * I(i,h) * H(h,s) * S(s)\\
&\quad * Y(y,n) * N(n,d) * D(d) * rho1(k,c)\\
&\quad * rho2(u,s) * rho3(o,d)\\
Joint(t,i,y) &:= K(k,a) * U(u,a) * A(a,r) * O(o,r)\\
&\quad * R(r,l) * L(l) * T(t,e) * E(e,c)\\
&\quad * C(c) * I(i,h) * H(h,s) * S(s)\\
&\quad * Y(y,n) * N(n,d) * D(d) * rho1(k,c)\\
&\quad * rho2(u,s) * rho3(o,d)\\
Norm(r,t,i,y) &:= join3(r,t,i,y) / Joint(t,i,y)\\
sumGB(t,i,y) &:= Norm(r,t,i,y) * R_{mean}(r)\\
&\quad * Joint(t,i,y) * Ones(r,t,i,y) * nrows
\end{aligned}
$$

After O2 and O3:

$$
\begin{aligned}
join3(r,t,i,y) &:= K(k,a) * U(u,a) * A(a,r) * O(o,r)\\
&\quad * R(r,l) * L(l) * T(t,e) * E(e,c)\\
&\quad * C(c) * I(i,h) * H(h,s) * S(s)\\
&\quad * Y(y,n) * N(n,d) * D(d) * rho1(k,c)\\
&\quad * rho2(u,s) * rho3(o,d)\\
sumGB(t,i,y) &:= join3(r,t,i,y) * R_{mean}(r) * nrows
\end{aligned}
$$

(d) Optimized TA program (O1: inlining, O2: algebra simplification, O3: dead code elimination).

Fig. 7. Effect of compiler optimizations on a group-by sum query over MiniSSB in BayesAQP (red terms indicate optimized TA expressions; reverse mapping is not treated as a TA operation).

Expressing SQL aggregations, joins, and group-by clauses in BTL++, and subsequently in TA, can introduce redundant contractions and high-dimensional intermediate tensors. To address this, BayesAQP performs three key compilation optimizations at the TA level: inlining, algebraic simplification, and dead code elimination. These optimizations reduce both tensor dimensionality and computational cost before code generation.

Inlining (O1). Inlining replaces symbolic tensor variables or function calls with their concrete definitions at the call site, avoiding the materialization of intermediate tensors and exposing opportunities for contraction fusion. BayesAQP applies inlining within TA expressions to eliminate high-order intermediates that can otherwise dominate inference cost. While performing joins, each pair of join keys introduces a tensor contraction between the inference results of the participating tables and their correlation matrix. By inlining, BayesAQP fuses these contractions into a single composite operation, enabling tensor backends to optimize contraction ordering and execution more effectively.

Example 4: `GROUP-BY SUM` with inner join of 4 tables. Figure 7a presents an example of a group-by sum query with three joins between the relations `lineorder`, `dwdate`, `customer`, and `supplier` in the MiniSSB dataset, and the BNs related to this query, where each join key pair is connected by a red dashed line. For instance, ρ_1 refers to the correlation matrix of `lineorder` and `customer`. For demonstration purposes, we omit the details of defining the BNs, loading the correlation matrices (e.g., ρ in Figure 7b, lines 6, 8, and 10), and the 1-dimensional tensor of possible values for each aggregated attribute (e.g., *lo_revenue_mean* in Figure 7b, line 12). The operator *unitMat* (Figure 7b, line 13) constructs an all-one matrix with the same shape as the tensor *join*1. Figure 7c shows the TA program translated from the BTL++ program in Figure 7b, where terms highlighted in red indicate where inlining occurs. Producing intermediate tensors such as $Prep_1$, a 6-dimensional tensor with two dimensions later summed out, substantially increases computational complexity and memory footprint. Through inlining, BayesAQP replaces $Prep_1$ with its defining contraction, which has a lower tensor order and can be directly fused with subsequent contractions. This transformation eliminates the need to materialize high-dimensional intermediates and enables advanced tensor processing frameworks to plan and execute contraction sequences more efficiently (cf. Figure 7d).

Tensor Algebra Simplification (O2). Translating SQL aggregations into TA expressions may yield redundant contractions and normalizations. BayesAQP mitigates this by applying three algebraic simplification rules.

In BayesAQP, an average query corresponds to computing the expected value of the aggregated attributes. This often yields expressions where a tensor (or contraction) is multiplied by a division: the division term represents the normalized joint marginal distribution of the aggregated attributes, while the remaining terms encode their possible values. Computing such an average typically requires at least three tensor contractions: two to obtain the normalized marginal distribution and one to compute the expectation. Since the marginal distribution is eventually summed after elementwise multiplication with the attribute values, BayesAQP merges the final contraction into the numerator of the division. This transformation eliminates high-dimensional intermediates and allows tensor processing frameworks to optimize long einsum expressions more effectively.

In SQL, a `SUM` query can be expressed as the product of a `COUNT` query and an `AVG` query that share the same predicates and aggregated attributes. During inference, both the aggregated and group-by attributes must be preserved, which can create large intermediate tensors and redundant computations. The `AVG` component introduces a normalization term for the joint marginal distribution of the aggregated attributes that is equivalent to the probability term already computed by the `COUNT` query. BayesAQP cancels this redundant normalization with the `COUNT` query, eliminating unnecessary tensor contractions and reducing intermediate dimensionality. This optimization enables further algebraic simplifications and produces more efficient tensor programs for SQL aggregations in tensor processing backends.

Example 4 (cont.). Based on the TA program after inlining in Figure 7d, BayesAQP can detect the normalization of the marginal distribution introduced by the average query, which can be canceled by the probability of the count query, referring to the tensor variables $Joint$ in the bodies of $Norm$ and *sumGB*. As a result, the tensor variables $Norm$ and $Joint$ in the body of *sumGB* will be replaced by the numerator of $Norm$, which is $Unnorm$. In addition, the all-one matrix *Ones* does not change the contraction result, which is a redundant computation that can be removed.

Dead Code Elimination (O3). After applying inlining and algebraic simplification, some TA expressions no longer contribute to the final aggregation result. BayesAQP performs dead code elimination to remove such expressions, producing a lightweight TA program for execution.

Example 4 (cont.). After applying inlining and algebra simplification, tensor variables like $Norm$ and $Joint$ do not contribute to the computation of the SQL aggregations anymore, introducing redundant tensor contractions (cf. Figure 7d). BayesAQP detects and removes these expressions via dead code elimination, yielding a more efficient TA program.

6 Experiments

This section aims to answer the following research questions:

- How efficient and accurate is BayesAQP for AQP including cardinality estimation compared to state-of-the-art systems (Sections 6.2 and 6.3)?
- How do the optimizations employed by BayesAQP impact the performance of inference (Section 6.4)?
- How do different data distributions affect the inference performance of BayesAQP in AQP (Section 6.5)?
- What is the offline model construction cost of BayesAQP in terms of training time and storage overhead (Section 6.6)?
- What is the compilation overhead of BayesAQP to achieve efficient tensor algebra computations (Section 6.7)?

6.1 Setup

All experiments are conducted on an Apple Silicon M4 Pro chip equipped with a 14-core CPU, a 20-core GPU, 48 GB of unified memory, and a 1 TB SSD. BayesAQP is implemented in Scala 2.12.11 and compiled ahead of time using GraalVM Native Image 21.0.2. For runtime tensor processing, we use Python 3.12.11 with opt_einsum 3.4.0, NumPy 2.2.6 and PyTorch 2.8.0.

Datasets and Queries. We evaluate the AQP performance of all systems on both single-table and multi-table benchmarks:

(1) **Flights**: This real-world dataset contains information on U.S. domestic flights. We scale it up to 1 billion tuples using the IDEBench data generator [9] and reuse all query templates from DeepDB [14], covering diverse aggregation and group-by patterns with selectivities ranging from 0.01% to 5%. The Flights dataset is a widely adopted benchmark for single-table AQP evaluation [12, 14, 17, 19, 24].
(2) **Star Schema Benchmark** (SSB) [26]: This synthetic dataset is designed for evaluating multi-table query processing with a star schema derived from the TPC-H dataset. We use a 50 GB scale factor due to the available hardware resources and evaluate the standard queries with joins of up to 5 tables.

Systems	Median	90th	95th	Max	Latency (ms)
BayesAQP (numpy)	2.20	11.87	89.65	1668.95	243.85
BayesCard	1.29	3.54	4.84	19.14	**2.25**
FactorJoin	3.99	21.41	36.04	102.63	3.13
DeepDB	1.23	4.10	4.47	58.51	4.23
PilotDB ('sample' mode)	1.04	1.67	13.55	215.57	557.00
PilotDB ('aqp' mode)	**1.00**	**1.00**	**1.00**	**1.00**	3820.70

Table 1. CPU-based performance comparison on the IMDB Job-Light benchmark, evaluated using q-error and average per-query inference latency (ms).

(3) **IMDB**: The IMDB dataset is a real-world, multi-table benchmark containing worldwide information about movies, video games, and related entities. We use the data snapshot provided by [20] and evaluate cardinality estimation using the *JOB-light* workload, a variant of the original benchmark. JOB-light consists of 6 tables organized in a star schema with 5 join conditions and includes 70 queries, each involving 3–6 tables and 1–4 selection predicates.

Competitors. We compare BayesAQP with two BN-based cardinality estimation approaches, BayesCard [42] and FactorJoin [43], as well as several representative AQP systems: DeepDB [14], VerdictDB [28], and PilotDB [45]. DeepDB learns relational sum–product networks (RSPNs) to model data distributions for approximate query answering. VerdictDB is a middleware-based AQP system that rewrites queries to execute over variational subsamples within existing DBMSs; we follow the configuration recommended in [28], using a 1% sampling rate on the fact table. PilotDB is an online AQP system that provides error guarantees using block-based sampling and multi-pass query rewriting over the underlying DBMS. For PilotDB, we use the default configuration specified in [45]. We also attempted to reproduce DBEst++ [24], but were unable to load our datasets; this issue has been reported to the authors.

Metrics. We evaluate AQP performance using two standard metrics: the inference latency and approximation accuracy. Inference latency is measured in milliseconds (ms), referring to the runtime used solely for query approximation, excluding SQL parsing and other system-specific preprocessing. For fairness, we exclude system-dependent per-query overheads such as the compilation time in BayesAQP, query rewriting and sample indexing in VerdictDB, and SQL parsing for all systems. For all systems, the inference latency is reported using the `perf_counter()` function in Python, averaged over five runs with four warm-up runs. Accuracy is evaluated using the average relative error (%) following the standard AQP evaluation practice [14, 22, 28, 37]. For experiments of cardinality estimation, we use the q-error to assess accuracy [14, 42, 43].

6.2 Cardinality Estimation

Cardinality estimation is a fundamental component of the query optimizer. More accurate cardinality estimates enable better resource allocation decisions and lead to higher execution efficiency in DBMSs during query execution. This section compares the performance of different systems on the IMDB JOB-light workload, as summarized in Table 1. Although VerdictDB generates samples for this benchmark, it falls back to executing queries on the raw data in PostgreSQL; therefore, its results are omitted from the table.

Overall, BayesCard achieves the best balance between estimation accuracy and inference latency. While PilotDB attains highly accurate estimates in its AQP mode, its runtime performance approaches that of PostgreSQL and is significantly slower than other systems as well as its own

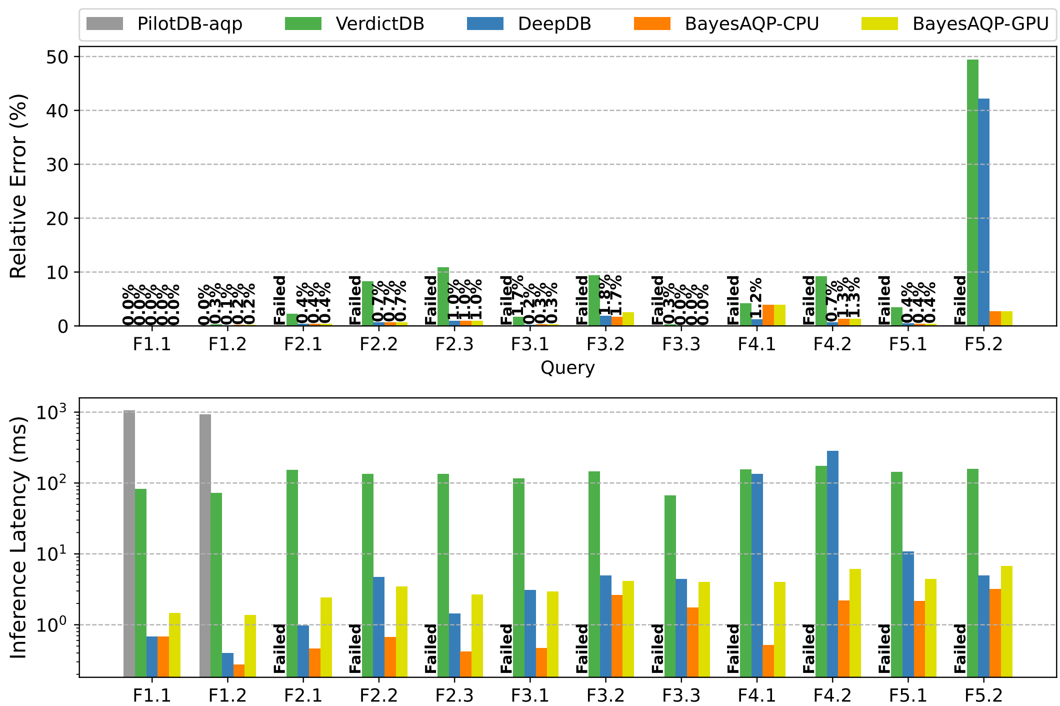


Fig. 8. AQP performance comparison among PilotDB, VerdictDB, DeepDB and BayesAQP on Flights (sf=1 billion) dataset. 'Failed' indicates that PilotDB falls back to exact execution in PostgreSQL.

sample mode. The CPU backend of BayesAQP performs poorly on this workload, as the IMDB JOB-light queries do not trigger any compilation optimization phases in BayesAQP, and several outlier queries (e.g., Q58, Q67, and Q68) exhibit substantially degraded accuracy. Although GPU execution of BayesAQP is 143.4x faster than its CPU execution, we exclude it to ensure a fair comparison against existing CPU-based approaches.

6.3 Performance of AQP

This section evaluates the efficiency of BayesAQP for AQP and compares its performance (using both CPU and GPU execution backends) against DeepDB and VerdictDB on single-table and multi-table workloads, shown in Figures 8 and 9, respectively.

Single table. Figure 8 compares the performance of BayesAQP, DeepDB, VerdictDB, and PilotDB on the Flights dataset scaled to 1 billion tuples. BayesAQP achieves accuracy comparable to DeepDB, with an average relative error of 1.09%, and consistently higher accuracy than VerdictDB across all queries, while outperforming DeepDB significantly on query F5.2. In terms of inference latency, the CPU backend of BayesAQP outperforms both DeepDB and VerdictDB on all queries, achieving average speed-ups of 34.9x and 172.2x, respectively. Although the GPU backend of BayesAQP is slightly slower than its CPU backend for all queries and slower than DeepDB on roughly half of them, it delivers average speed-ups of 7.5x over DeepDB and 39.5x over VerdictDB. PilotDB achieves high accuracy in its aqp mode at the cost of substantially higher runtime latency. For this scale, PilotDB is unable to resolve sampling due to insufficient sample sizes and therefore falls back to exact execution in Postgres by default.

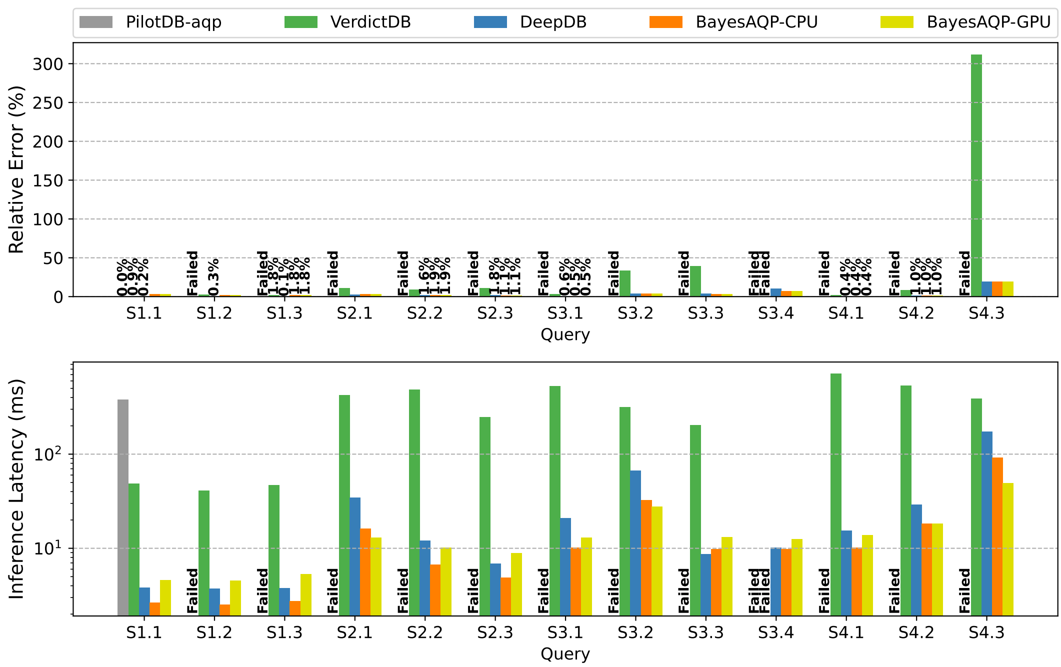


Fig. 9. AQP performance comparison among PilotDB, VerdictDB, DeepDB and BayesAQP on SSB dataset (sf=50). 'Failed' indicates that PilotDB falls back to exact execution in PostgreSQL.

Multi-table. Figure 9 presents a performance comparison for executing SSB queries across VerdictDB, DeepDB, PilotDB, and BayesAQP. As in the Flights workload, PilotDB attains high accuracy in its aqp mode but incurs substantially higher inference latency. At this scale, limited sample sizes prevent effective sampling, causing PilotDB to frequently fall back to exact execution in PostgreSQL. Overall, BayesAQP and DeepDB consistently outperform VerdictDB in inference latency. For query S3.4, VerdictDB fails to return a result within the 10-minute timeout threshold, whereas both backends of BayesAQP and DeepDB complete execution in 13 milliseconds. The CPU backend of BayesAQP is faster than DeepDB on all queries except query S3.3, achieving an average speed up of 1.6x while the GPU backend is faster than DeepDB on approximately half of the queries with an average speed up of 1.4x. In terms of accuracy, BayesAQP and DeepDB yield comparable results, differing by less than 1% for most queries and by at most 3% for query S3.4. On average, the CPU and GPU backends of BayesAQP reduce inference latency by 35.3% and 10.8%, respectively, compared to DeepDB, and significantly outperform VerdictDB on all queries.

In summary, for both single-table and multi-table workloads, BayesAQP and DeepDB consistently outperform VerdictDB in terms of both accuracy and inference latency. DeepDB tends to infer faster for queries yielding low-dimensional results, determined by a smaller number of group-by attributes, and achieves higher accuracy for query patterns like `SUM(A*B)` without any group bys (e.g., S1.1–S1.3). The CPU backend of BayesAQP attains larger speed-ups on low-dimensional intermediate computations, similar to DeepDB, while its GPU backend performs notably better on queries involving higher-dimensional computations caused by a larger number of group-by attributes, due to its hardware architecture. Overall, BayesAQP provides a balanced trade-off between accuracy and inference latency compared to both VerdictDB and DeepDB.

Furthermore, when the dataset size increases substantially (e.g., by 10x larger) but its data statistics remain stable, BayesAQP's inference performance can remain consistent, since it can be

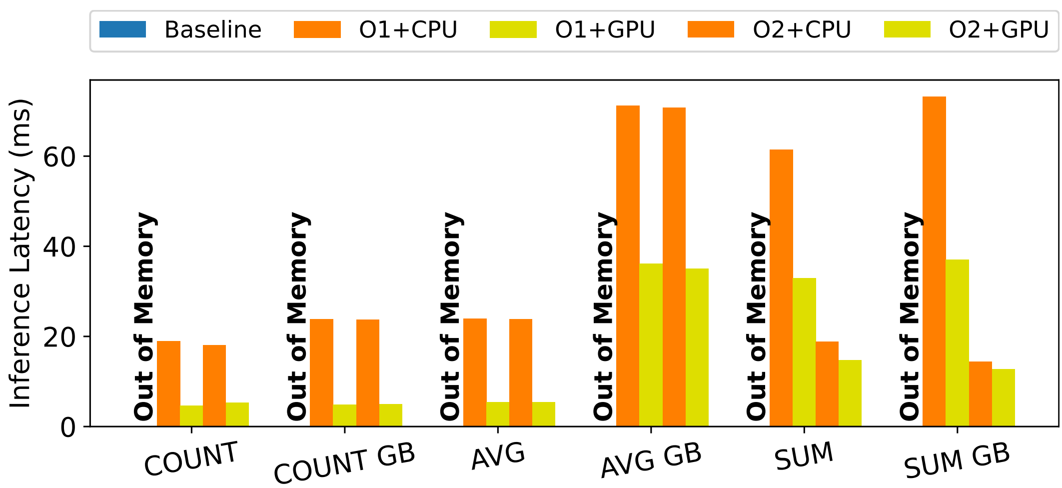


Fig. 10. Impact of different optimization phases in inference performance of BayesAQP.

trained effectively on representative samples, similar to DeepDB, maintaining model sizes that are relatively small compared to the original data. However, incremental updates that alter data distributions require recomputation of affected statistics. In particular, if updates modify nodes in the BNs with many parent or child dependencies which introduces higher computational complexity, retraining models from scratch is preferable to incremental updates to preserve inference efficiency.

6.4 Optimizations in Compilation Phases

This section evaluates the impact of compiler optimizations in BayesAQP across different aggregation types using a query template adapted from SSB query S2.1 (cf. Figure 10):

```
SELECT [d_year, p_brand1], AGG(lo_revenue)
FROM lineorder, dwdate, part, supplier
WHERE lo_orderdate = d_datekey AND lo_partkey = p_partkey AND lo_suppkey = s_suppkey
AND p_category = 'MFGR#12' AND s_region = 'AMERICA'
GROUP BY [d_year, p_brand1];
```

In this template, `AGG` denotes a placeholder for aggregation functions `COUNT`, `AVG`, and `SUM`, which correspond to the x-axis labels in Figure 10. The `COUNT` label represents a query without a group-by clause, while `COUNT GB` refers to the same query with grouping enabled. All variants group by both `d_year` and `p_brand1`.

Figure 10 reports the effect of these optimizations on inference latency. `Baseline` indicates compilation without optimizations, `O1` applies inlining, and `O2` performs algebraic simplifications on top of inlining. Without the proposed optimizations, BayesAQP fails to efficiently process queries involving high-dimensional computations, leading to termination due to out-of-memory errors within two minutes of execution. Applying inlining mitigates this issue by eliminating unnecessary tensor intermediates, improving execution efficiency across all aggregation types, and exposing opportunities for subsequent algebraic simplifications. Although inlining introduces limited benefit for `COUNT` and `AVG` queries and their grouped variants, it significantly accelerates `SUM` queries (Figure 10). This behavior arises because the current implementation applies these optimizations (Section 5) only when the order of the output tensor exceeds two for each contraction in the tensor algebra program. For this query template, which involves two group-by attributes, this design choice yields minimal gains for `COUNT` and `AVG` queries and may slightly degrade inference performance due to additional contraction overhead.

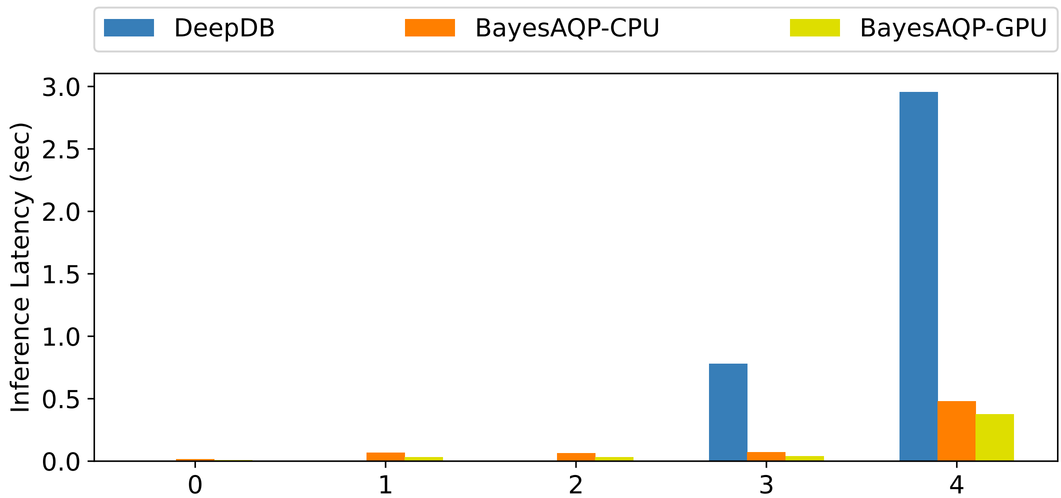


Fig. 11. Impact of group-by dimensionality on inference latency, comparing BayesAQP and DeepDB.

For SUM and its group-by queries, the normalization coefficient in the corresponding AVG computation equals the joint distribution derived from the COUNT query, which can be cancelled out. BayesAQP detects this pattern through inlining and removes the redundant normalization step, yielding performance improvements of 3.3x for SUM and 5.1x for group-by SUM queries on the CPU backend, and 2.2x and 2.9x on the GPU backend, respectively. Overall, the GPU backend achieves 3.2x and 2.9x greater performance gains than the CPU backend for inlining and for inlining combined with algebraic simplifications, respectively, by parallelizing group-by computations in accordance with its architectural characteristics.

6.5 Scalability with Group-By Complexity

This section evaluates how query complexity, defined by the number and domain size of group-by attributes, affects inference performance.

Number of Distinct Group-By Attributes. The number of group-by attributes in a query directly influences computational complexity, as each additional attribute increases the dimensionality of the result. Since both BayesAQP and DeepDB consistently outperform VerdictDB on single-table and multi-table workloads, we focus this analysis on BayesAQP and DeepDB. The experiment is based on the following query template:

```
SELECT [p_mfgr, p_category, p_type, p_brand1, p_name], COUNT(*)
FROM lineorder, dwdate, part, supplier
WHERE lo_orderdate = d_datekey AND lo_partkey AND p_partkey AND lo_suppkey = s_suppkey
AND s_region = 'AMERICA' AND d_weeknuminyear = 6 AND lo_quantity>=5
GROUP BY [p_mfgr, p_category, p_type, p_brand1, p_name];
```

Figure 11 shows how varying the number of group-by attributes (from zero to four) affects inference latency. DeepDB handles group-by queries by decomposing them into multiple non-group-by queries, one for each unique combination of group-by attribute values, all sharing the same selection predicates. This approach is efficient when the number of groups is small but scales poorly as dimensionality increases, since the number of distinct combinations grows rapidly and each must be evaluated independently.

In contrast, BayesAQP represents the entire group-by aggregation as a single tensor algebra expression, which enables global optimization and efficient execution through tensor contraction

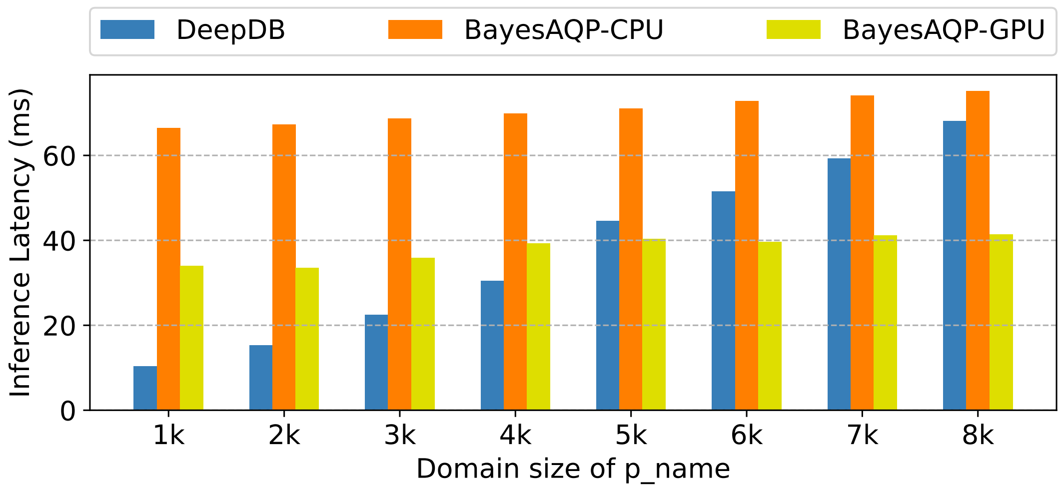


Fig. 12. Impact of the domain size of p_name on inference latency, comparing BayesAQP and DeepDB on a synthetic SSB-based dataset.

planning. As a result, for queries with three and four group-by attributes, BayesAQP achieves 3.5× and 5.7× speed-ups on CPU and GPU backends, respectively (cf. Figure 11). By optimizing the entire query jointly rather than decomposing it into subqueries, BayesAQP scales effectively for high-dimensional analytical workloads that challenge existing AQP systems.

Domain Sizes of Group-By Attributes. We further evaluate the impact of increasing domain sizes of group-by attributes on inference latency (Figure 12). We modify the template above to group by `p_name`, a high-cardinality attribute, and synthetically vary its domain size from 1,000 to 8,000. DeepDB performs well for small domains, where the number of subqueries is limited, but its runtime grows rapidly with domain size due to query decomposition overhead. BayesAQP, on the other hand, maintains stable scaling by executing the entire aggregation as a single high-dimensional tensor contraction, achieving up to 1.14× lower latency on large domains with its GPU backend. This improvement arises from the tensor-based execution in BayesAQP, which optimizes over the entire query expression rather than iterating through per-group subqueries, leading to better scalability for complex analytical queries.

Skewness in Group-By Attributes. Beyond varying domain sizes, we study how skewness in group-by attributes affects accuracy and inference latency. We reuse the same group-by query evaluated in Figure 11 and vary the skewness of the group-by attribute `p_name` in the `part` table, while keeping the schema, table cardinalities, all other attributes in the `part` table, and all other relations unchanged. For each skew parameter $\alpha \in \{0.0, 0.5, 1.0, 1.5, 2.0\}$ (from uniform to heavy-tailed), values of `p_name` are resampled i.i.d. from a bounded Zipfian distribution over the observed domain, i.e., $\Pr(r) \propto 1/r^{\alpha}$ for rank r, using `scipy.stats.zipfian`. This preserves the attribute domain and isolates the effect of value-frequency skew.

As shown in Figure 13, BayesAQP consistently achieves lower or comparable relative error than DeepDB across all skew levels. Although both systems exhibit increasing error as skew grows, BayesAQP better captures heavy-tailed distributions (e.g., $\alpha = 2.0$), with accuracy recovering at $\alpha = 1.5$. In terms of inference latency, BayesAQP with GPU execution achieves the lowest latency across all skew levels. Overall, BayesAQP using the GPU backend is two orders of magnitude faster than using its CPU backend and provides a 2.69× speedup over DeepDB on average.

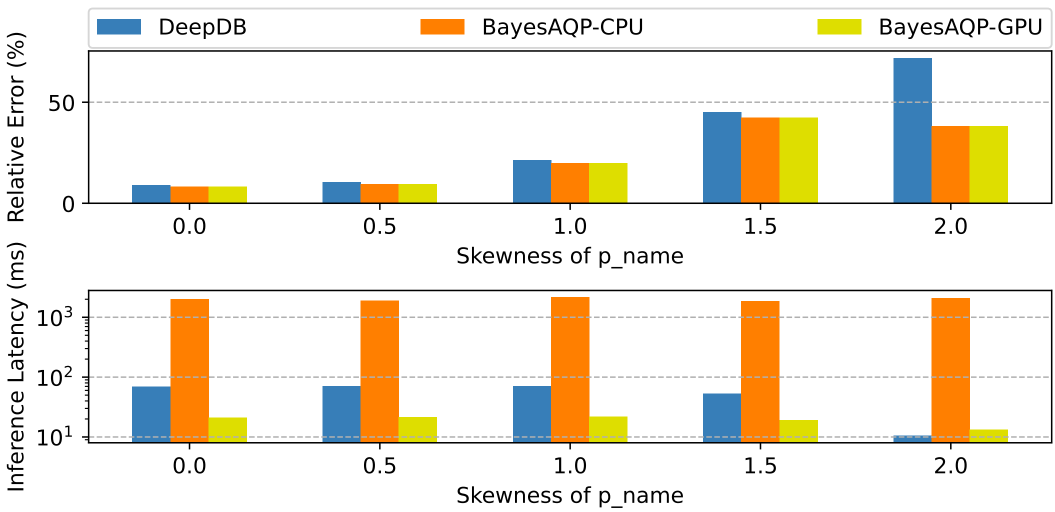


Fig. 13. Impact of skewness in the group-by attribute p_name on runtime performance, comparing BayesAQP and DeepDB.

6.6 Offline Training

Figure 14 compares training time and storage cost across systems as the SSB scale factor increases from 10 to 50. Results for BayesAQP are highlighted in red, since the same trained parameters are used for both CPU and GPU execution.

Compared to VerdictDB, both BayesAQP and DeepDB incur lower model training time. Across scale factors 10–50, BayesAQP consistently exhibits lower training overhead than DeepDB; however, its training time increases gradually with scale, whereas the training time of DeepDB converges toward an upper bound, indicating better scalability at larger scale factors. In terms of storage, BayesAQP requires significantly less space than the original data by storing model parameters in sparse tensor format (i.e., coordinate format, COO), but remains less compact than DeepDB; improving storage efficiency is left as future work.

6.7 Compilation Overhead

This section evaluates the compilation overhead of BayesAQP when handling AQP workloads. BayesAQP generates optimized TA programs from input SQL queries through its compilation pipeline. However, the current prototype introduces non-negligible overhead that affects end-to-end performance. Two main factors contribute to this overhead: (1) the compilation process that translates SQL queries into tensor algebra expressions and applies optimizations, and (2) the cost of invoking external tensor algebra libraries from Python, which adds runtime latency when called from Scala.

Figure 15 presents the end-to-end performance of BayesAQP on the SSB dataset (sf=50) in milliseconds, compared with VerdictDB, as both systems follow a similar architectural pattern in which query processing is delegated to an external execution backend. The overhead of each compilation phase in BayesAQP is detailed in Figure 15: `IR Translation`, which converts SQL queries into a sequence of intermediate representations (some redundant in the current prototype); `O1` and `O2`, introduced in Section 5, refers to inlining and algebraic simplifications (the latter on top of inlining), respectively; `Codegen`, which generates executable Python code from the optimized tensor algebra (TA) program; and `Python Overhead`, which measures the latency of invoking

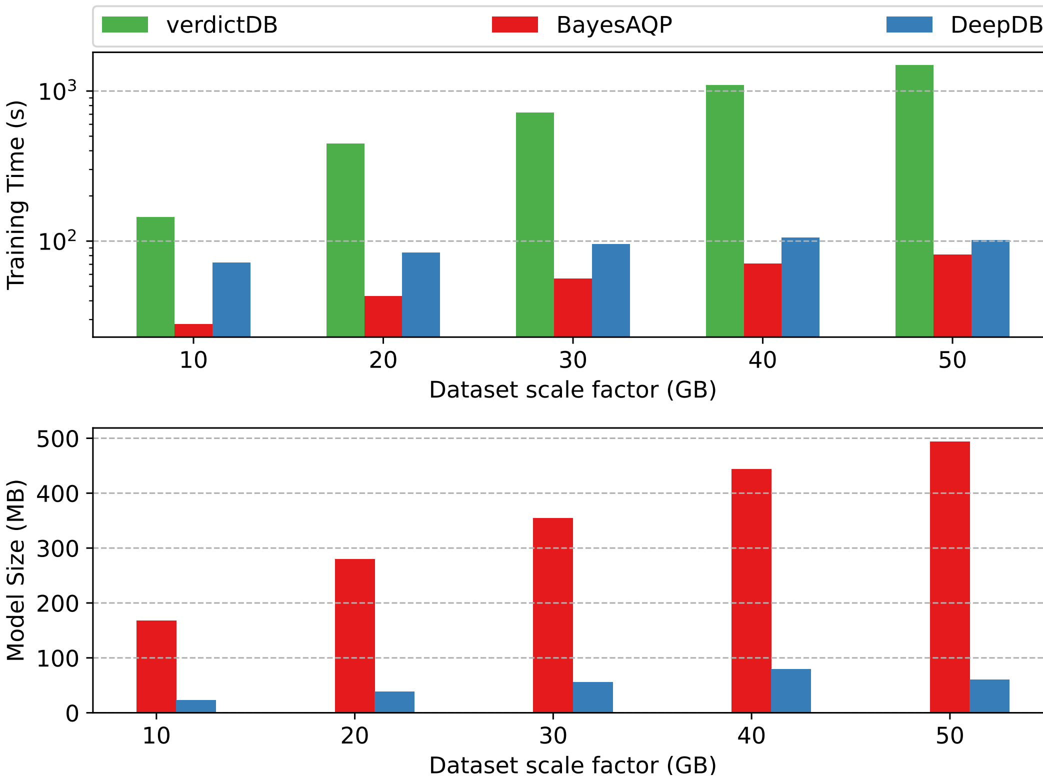


Fig. 14. Model construction cost on SSB dataset (sf10-50).

Python code from Scala. Among these, the compilation and code generation phases dominate total overhead, while Python invocation adds a smaller but nontrivial constant cost.

The compilation time remains nearly constant across dataset scales, as long as the query pattern remains consistent. However, more complex SQL queries produce longer translated tensor algebra programs, leading to larger code generation overhead. In the current prototype, compilation is performed for every query invocation, resulting in a fixed overhead of roughly 100 ms that dominates latency for trivial queries. This overhead is independent of data size and query complexity and thus does not represent query-time behavior. For this reason, we report inference latency separately and quantify the compilation cost in isolation. In realistic deployments where compiled tensor programs are cached, end-to-end performance would converge toward the reported inference times.

7 Future Directions

Query Patterns. Section 3 illustrated pairwise joins, and BayesAQP can also support acyclic join patterns by composing correlation matrices (e.g. $\rho_{AB}^{\top} \cdot \rho_{AC}^{\top} \cdot \rho_{AD}^{\top} \cdots$ for a star join based at A, or $\rho_{AB}^{\top} \cdots \rho_{BC}^{\top} \cdots \rho_{CD}^{\top} \cdots$ for a chain join). However, the results of BayesAQP can differ depending on the join order, since for example $\rho_{AB}^{\top} \cdot \rho_{AC}^{\top}$ is not guaranteed to be equal to $\rho_{AB}^{\top} \cdot \rho_{BC}^{\top}$. Addressing this requires updating pairwise correlation matrices during runtime inference, enabling adaptive refinement of join statistics and more accurate approximations for complex join patterns. Supporting self-joins requires modifying BayesAQP to be alias-aware, but we believe our approach can be adapted to this case.

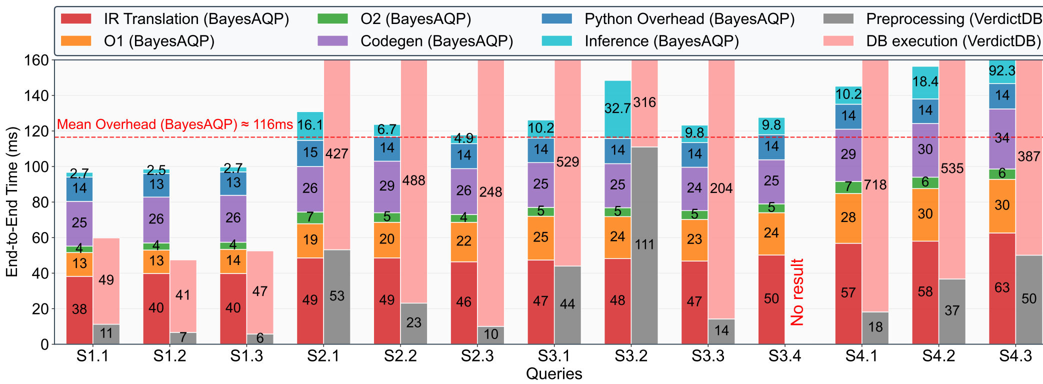


Fig. 15. Compilation overhead comparison of the end-to-end performance for AQP between BayesAQP and VerdictDB using SSB queries.

BayesAQP currently restricts inference to tree-structured BNs, so does not support queries whose join structure induces loopy inference. Factor graphs [18] could be used instead enabling inference over loopy graphical models, as in FactorJoin [43].

Extreme statistics (e.g., `MIN` and `MAX`) are not yet supported, but appear possible to handle for continuous attributes by locating the smallest and largest domain intervals with non-zero probability mass in the inferred marginal distributions. General disjunctive predicates could be supported by extending the inference backend with integer arithmetic, following recent advances in probabilistic programming [6].

Compilation Overhead. In BayesAQP, probabilistic queries are lowered through multiple intermediate representations as in BTL [27]. This design repeatedly traverses intermediate representations to eventually generate tensor expressions, which contributes to the currently high overhead of compilation. We plan to decrease this overhead by removing redundant transformation stages and streamline the intermediate representations, which will also make it easier to target both dense and sparse tensor backends.

As future work, we plan to reduce this overhead by shifting tensor execution from Python to C++ backends, thereby trading Python invocation and interpretation overhead for faster C++ compilation. We also intend to optimize the compilation pipeline by eliminating redundant intermediate translations and exploring more efficient compiler optimization passes.

8 Related Work

Approximate query processing provides efficient yet approximate query answers in database systems. Existing AQP techniques are primarily sampling-based or learning-based, whereas BayesAQP adopts a probabilistic programming approach using BNs.

Sampling-based AQP. These approaches [21, 28, 45] approximate query results by operating on partial data. VerdictDB [28] rewrites queries to execute over variationally subsampled tables within existing DBMSs, providing compatibility with modern database engines but relying heavily on sample representativeness for accuracy. Because execution is bound to the host DBMS, its optimization scope is limited to query rewriting, whereas BayesAQP enables compiler-level optimization over tensor backends. WanderJoin [21] employs a Monte Carlo algorithm that samples random join paths to estimate aggregates; its efficiency depends on join conditions known in advance and requires evaluating each group-by value as an independent query, preventing global optimization over large grouped domains. PilotDB [45], similarly to VerdictDB, relies on DBMS execution, which makes it robust to diverse query patterns; however, despite achieving high accuracy in its aqp mode, PilotDB is relatively slow compared to other systems (Table 1, Section 6), even without accounting for execution overhead.

ML-based AQP. These approaches [14, 24, 25] train predictive models offline to approximate query results. DeepDB [14] uses relational sum–product networks (RSPNs) to model data distributions, but decomposes group-by queries into equality-constrained subqueries executed independently at runtime, preventing shared computation across groups. As shown in Section 6.5, this design incurs significant runtime overhead for multiway joins and large grouped domains. BayesAQP addresses this limitation through a tensor–based compilation pipeline that enables algebraic optimization and efficient execution on both CPUs and GPUs. DBEst++ [24] replaces RSPNs with mixture density networks and word embeddings but faces similar challenges in interpretability and scalability; we were unable to reproduce its reported results due to implementation issues.

BN-based Cardinality Estimation. As shown in Table 1 (Section 6), BN-based approaches such as BayesCard [42] and FactorJoin [43] are more effective on cardinality estimation (a subproblem of AQP). Relative to these approaches, BayesAQP supports a broader range of analytical queries, including group bys, averages, summations, and arithmetic compositions. Some accuracy outliers observed in our evaluation may be mitigated by adopting stronger correlation measures, such as the Randomized Dependence Coefficient [23] adopted by BayesCard and DeepDB.

Probabilistic Relational Models. Probabilistic relational models (PRMs) describe statistical models over abstract relational domains and can be specialized to PGMs (e.g., BNs). While PRMs provide greater modeling flexibility, exact inference is #P-hard [32], with complexity exponential in the treewidth of the grounded graph, which may grow with database size [11]. In contrast, BayesAQP adopts a BN-based approach that handles intra-table and inter-table dependencies separately, using Chow–Liu trees and pairwise correlation matrices. This design trades some semantic expressiveness for improved scalability and tighter control over inference complexity.

9 Conclusions

We presented BayesAQP, a system and framework for efficient AQP leveraging recent advances in probabilistic programming. The core contribution of BayesAQP is a compilation pipeline that applies compiler optimizations on the translated TA programs from the input SQL queries to generate efficient executable code for tensor processing backends. This is enabled by supporting BTL++, a probabilistic programming language that supports fast exact inference over discrete distributions and introduces probabilistic computation constructs (e.g., expectation) to express SQL aggregations.

Our experiments on three real-world datasets show that BayesAQP achieves an average 37x speed-up over existing AQP systems while maintaining high accuracy. BayesAQP can target multiple backends, demonstrated on CPU-based NumPy and GPU-based PyTorch, where GPU execution benefits queries with complex tensor contractions. Future work includes offering approximate query answers with accompanying error bounds similar to DeepDB, optimizing the training phase through structured histograms [43], enhanced parameter and structure learning [36], integrating sparse tensor backends [34], and extending the pipeline to other probabilistic graphical models.

Acknowledgments

This work was partially funded by an unrestricted gift from Google and RelationalAI.

References

[1] Martín Abadi et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org.

[2] Ankur Ankan and Johannes Textor. 2024. pgmpy: A Python Toolkit for Bayesian Networks. *Journal of Machine Learning Research* 25, 265 (2024), 1–8. http://jmlr.org/papers/v25/23-0487.html

[3] Paul Arora, Devon Boyne, Justin J. Slater, Alind Gupta, Darren R. Brenner, and Marek J. Druzdzel. 2019. Bayesian Networks for Risk Prediction Using Real-World Data: A Tool for Precision Medicine. *Value in Health* 22, 4 (2019), 439–445. doi:10.1016/j.jval.2019.01.006

[4] Vince Bárány, Balder Ten Cate, Benny Kimelfeld, Dan Olteanu, and Zografoula Vagena. 2017. Declarative Probabilistic Programming with Datalog. *ACM Trans. Database Syst.* 42, 4, Article 22 (Oct. 2017), 35 pages. doi:10.1145/3132700

[5] Concha Bielza and Pedro Larranaga. 2014. Discrete Bayesian Network Classifiers: A Survey. *Comput. Surveys* 47 (07 2014), 1–43. doi:10.1145/2576868

[6] William X. Cao, Poorva Garg, Ryan Tjoa, Steven Holtzen, Todd Millstein, and Guy Van den Broeck. 2023. Scaling Integer Arithmetic in Probabilistic Programs. arXiv:2307.13837 [cs.AI] https://arxiv.org/abs/2307.13837

[7] Mark Chavira and Adnan Darwiche. 2008. On probabilistic inference by weighted model counting. *Artificial Intelligence* 172, 6 (2008), 772–799. doi:10.1016/j.artint.2007.11.002

[8] C. K. Chow and C. N. Liu. 1968. Approximating discrete probability distributions with dependence trees. *IEEE Trans. Inf. Theory* 14, 3 (1968), 462–467. doi:10.1109/TIT.1968.1054142

[9] Philipp Eichmann, Carsten Binnig, Tim Kraska, and Emanuel Zgraggen. 2018. IDEBench: A Benchmark for Interactive Data Exploration. arXiv:arXiv:1804.02593

[10] Nir Friedman and Moises Goldszmidt. 1997. Sequential update of Bayesian network structure. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence* (Providence, Rhode Island) *(UAI'97)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 165–174.

[11] Lise Getoor and Ben Taskar. 2007. *Introduction to statistical relational learning*. MIT press.

[12] Rong Gu, Han Li, Haipeng Dai, Wenjie Huang, Jie Xue, Meng Li, Jiaqi Zheng, Haoran Cai, Yihua Huang, and Guihai Chen. 2023. ShadowAQP: Efficient Approximate Group-by and Join Query via Attribute-Oriented Sample Size Allocation and Data Generation. *Proc. VLDB Endow.* 16, 13 (Sept. 2023), 4216–4229. doi:10.14778/3625054.3625059

[13] Haipeng Guo and William Hsu. 2002. A Survey of Algorithms for Real-Time Bayesian Network Inference. (10 2002).

[14] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2019. DeepDB: Learn from Data, not from Queries. arXiv:1909.00607 [cs.DB] https://arxiv.org/abs/1909.00607

[15] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *Proc. ACM Program. Lang. (OOPSLA)* (2020). doi:10.1145/342820

[16] Mathieu Huot, Matin Ghavami, Alexander K. Lew, Ulrich Schaechtle, Cameron E. Freer, Zane Shelby, Martin C. Rinard, Feras A. Saad, and Vikash K. Mansinghka. 2024. GenSQL: A Probabilistic Programming System for Querying Generative Models of Database Tables. *Proceedings of the ACM on Programming Languages* 8, PLDI (June 2024), 790–815. doi:10.1145/3656409

[17] Aaron Hurst, Daniel E. Lucani, and Qi Zhang. 2024. PairwiseHist: Fast, Accurate and Space-Efficient Approximate Query Processing with Data Compression. *Proc. VLDB Endow.* 17, 6 (Feb. 2024), 1432–1445. doi:10.14778/3648160.3648181

[18] Frank R. Kschischang, Brendan J. Frey, and Hans-Andrea Loeliger. 2001. Factor graphs and the sum-product algorithm. *IEEE Trans. Inf. Theory* 47, 2 (2001), 498–519. doi:10.1109/18.910572

[19] Moritz Kulessa, Alejandro Molina, Carsten Binnig, Benjamin Hilprecht, and Kristian Kersting. 2018. Model-based Approximate Query Processing. arXiv:1811.06224 [cs.DB] https://arxiv.org/abs/1811.06224

[20] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. doi:10.14778/2850583.2850594

[21] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 615–629. doi:10.1145/2882903.2915235

[22] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2017. Wander Join and XDB: Online Aggregation via Random Walks. *SIGMOD Rec.* 46, 1 (may 2017), 33–40. doi:10.1145/3093754.3093763

[23] David Lopez-Paz, Philipp Hennig, and Bernhard Schölkopf. 2013. The Randomized Dependence Coefficient. arXiv:1304.7717 [stat.ML] https://arxiv.org/abs/1304.7717

[24] Qingzhi Ma et al. 2021. Learned Approximate Query Processing: Make it Light, Accurate and Fast. In *Conference on Innovative Data Systems Research.*

[25] Qingzhi Ma and Peter Triantafillou. 2019. DBEst: Revisiting Approximate Query Processing Engines with Machine Learning Models. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1553–1570. doi:10.1145/3299869.3324958

[26] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5895)*, Raghunath Othayoth Nambiar and Meikel Poess (Eds.). Springer, 237–252. doi:10.1007/978-3-642-10424-4_17

[27] Jingwen Pan and Amir Shaikhha. 2023. Compiling Discrete Probabilistic Programs for Vectorized Exact Inference. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction* (Montréal, QC, Canada) *(CC 2023)*. Association for Computing Machinery, New York, NY, USA, 13–24. doi:10.1145/3578360.3580258

[28] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2018. VerdictDB: Universalizing Approximate Query Processing. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD/PODS '18)*. ACM. doi:10.1145/3183713.3196905

[29] Iago París, Raquel Sánchez-Cauce, and Francisco Javier Díez. 2020. Sum-product networks: A survey. arXiv:2004.01167 [cs.LG] https://arxiv.org/abs/2004.01167

[30] Adam Paszke et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG]

[31] Christina Peters, Aaron Higuera, Shixiao Liang, Venkat Roy, Waheed U. Bajwa, Hagit Shatkay, and Christopher D. Tunnell. 2022. A Method for Quantifying Position Reconstruction Uncertainty in Astroparticle Physics using Bayesian Networks. arXiv:2205.10305 [astro-ph.IM]

[32] Dan Roth. 1996. On the hardness of approximate reasoning. *Artificial Intelligence* 82, 1 (1996), 273–302. doi:10.1016/0004-3702(94)00092-1

[33] Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. 2021. SPPL: probabilistic programming with fast exact symbolic inference. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*. ACM, 804–819. doi:10.1145/3453483.3454078

[34] Maximilian Schleich et al. 2023. Optimizing Tensor Programs on Flexible Storage. *Proc. ACM Manag. Data* 1, 1, Article 37 (may 2023), 27 pages.

[35] Jacob Schreiber. 2017. Pomegranate: fast and flexible probabilistic modeling in python. *J. Mach. Learn. Res.* 18, 1 (Jan. 2017), 5992–5997.

[36] Amir Shaikhha et al. 2022. Functional collection programming with semi-ring dictionaries. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 89 (apr 2022), 33 pages.

[37] Nikhil Sheoran, Supawit Chockchowwat, Arav Chheda, Suwen Wang, Riya Verma, and Yongjoo Park. 2023. A Step Toward Deep Online Aggregation. *Proc. ACM Manag. Data* 1, 2, Article 124 (jun 2023), 28 pages. doi:10.1145/3589269

[38] Daniel G. a. Smith and Johnnie Gray. 2018. opt_einsum - A Python package for optimizing contraction order for einsum-like expressions. *Journal of Open Source Software* 3, 26 (2018), 753. doi:10.21105/joss.00753

[39] Yiquan Song, Jianhua Gong, Sheng Gao, Dongchuan Wang, Tiejun Cui, Yi Li, and Baoquan Wei. 2012. Susceptibility assessment of earthquake-induced landslides using Bayesian network: A case study in Beichuan, China. *Computers & Geosciences* 42 (2012), 189–199. doi:10.1016/j.cageo.2011.09.011

[40] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2021. An Introduction to Probabilistic Programming. arXiv:1809.10756 [stat.ML] https://arxiv.org/abs/1809.10756

[41] John Winn. 2023. *Model-based machine learning*. CRC Press.

[42] Ziniu Wu et al. 2021. BayesCard: Revitilizing Bayesian Frameworks for Cardinality Estimation. arXiv:2012.14743 [cs.DB]

[43] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: A New Cardinality Estimation Framework for Join Queries. *Proc. ACM Manag. Data* 1, 1, Article 41 (may 2023), 27 pages. doi:10.1145/3588721

[44] Nevin L Zhang and David Poole. 1994. A simple approach to Bayesian network computations. In *Proc. of the Tenth Canadian Conference on Artificial Intelligence*.

[45] Yuxuan Zhu, Tengjun Jin, Stefanos Baziotis, Chengsong Zhang, Charith Mendis, and Daniel Kang. 2025. PilotDB: Database-Agnostic Online Approximate Query Processing with A Priori Error Guarantees. *Proc. ACM Manag. Data* 3, 3, Article 198 (June 2025), 28 pages. doi:10.1145/3725335

Received October 2025; revised January 2026; accepted February 2026