

Lightweight Materialization for Fast Dashboards Over Joins

ZEZHOU HUANG, Columbia University, USA

EUGENE WU, Columbia University, USA

Dashboards are vital in modern business intelligence tools, providing non-technical users with an interface to access comprehensive business data. With the rise of cloud technology, there is an increased number of data sources to provide enriched contexts for various analytical tasks, leading to a demand for interactive dashboards over a large number of joins. Nevertheless, joins are among the most expensive operations in DBMSes, making the support of interactive dashboards over joins challenging.

In this paper, we present Treant, a dashboard accelerator for queries over large joins. Treant uses factorized query execution to handle aggregation queries over large joins, which alone is still insufficient for interactive speeds. To address this, we exploit the incremental nature of user interactions using Calibrated Junction Hypertree (CJT), a novel data structure that applies lightweight materialization of the intermediates during factorized execution. CJT ensures that the work needed to compute a query is proportional to *how different it is from the previous query*, rather than the overall complexity. Treant manages CJTs to share work between queries and performs materialization offline or during user "think-times." Implemented as a middleware that rewrites SQL, Treant is portable to any SQL-based DBMS. Our experiments on single node and cloud DBMSes show that Treant improves dashboard interactions by two orders of magnitude, and provides 10× improvement for ML augmentation compared to SOTA factorized ML system.

CCS Concepts: • **Information systems** → **Query optimization; Database views; Online analytical processing engines; Data access methods.**

Additional Key Words and Phrases: Dashboard; Data Exploration; Materialized Views; Factorized Query Execution; Message Passing; Junction Trees; Machine Learning; Interactive Visualization Systems

ACM Reference Format:

Ze Zhou Huang and Eugene Wu. 2023. Lightweight Materialization for Fast Dashboards Over Joins. *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 248 (December 2023), 27 pages. <https://doi.org/10.1145/3626735>

1 INTRODUCTION

Dashboards are at the heart of modern BI tools (e.g., PowerBI [25], Looker [1], Sigma Computing [27]) and provide a comprehensive view of a business within a single interface. Modern organizations store data across dozens or hundreds of tables in data warehouses, so dashboard creation consists of two stages. Offline, data engineers pre-define join relationships between relevant tables so that they can be queried like a denormalized "wide table", and create dashboard visualizations. Online, domain users interact with the dashboards and make business decisions. For example, let's consider a hypothetical scenario based on Sigma Computing:

EXAMPLE 1 (SIGMA COMPUTING DASHBOARD). *Anna, a sales manager, is responsible for driving revenue growth. To gain a better understanding of the current state of the business, she asks Shannon, a data engineer, to build a dashboard to display sales pipelines for potential revenue opportunities.*

Authors' addresses: Zezhou Huang, Columbia University, USA, zh2408@columbia.edu; Eugene Wu, Columbia University, USA, ewu@cs.columbia.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

2836-6573/2023/12-ART248 \$15.00

<https://doi.org/10.1145/3626735>

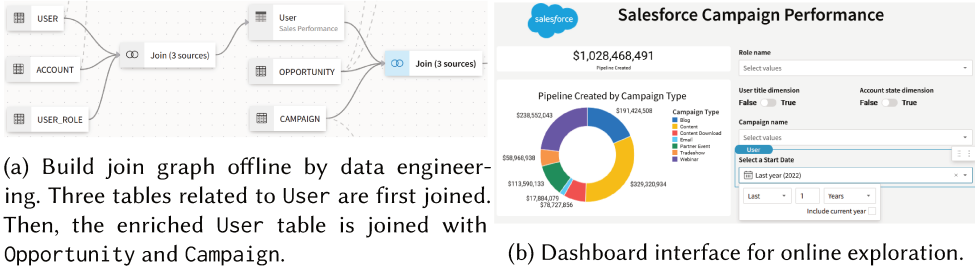


Fig. 1. Screenshots for Sigma Computing dashboards.

Shannon collects relevant tables from Salesforce (e.g., Opportunities, Campaigns, Users) for deals from various sales representatives, and creates the **join graph** $R_{\bowtie}$ shown in Figure 1a.

Shannon designs the dashboard in Figure 1b, where each chart is generated from an initial **dashboard query**, and interactions change parts of those queries to update the charts. The total "Pipeline Created" is computed by $Q_1 = \gamma_{\text{SUM}(\text{amount})} R_{\bowtie}$ while the pie chart displaying "Pipeline Created by Campaign Type" is computed by $Q_2 = \gamma_{\text{SUM}(\text{amount}), \text{Campaign_Type}} R_{\bowtie}$. Shannon adds interactions (e.g., dropdown boxes and switches) to filter or change the grouping attributes. For example, selecting "Sales Associate" in the "Role Name" dropdown triggers an **interaction query** that filters $R_{\bowtie}$ by the role before aggregating the data for Q_1, Q_2 , while toggling the "User Title" switch adds the attribute to the group by. However, every interaction translates to queries over the join graph that take many seconds to complete, causing Anna to stop using the dashboard.

The pattern in the above example is common. Data engineers build a dashboard over a complex acyclic join graph. When a domain user loads the dashboard, it first executes the **dashboard queries** to load the initial view, and then executes many **interaction queries** in response to user manipulations. Each of the *interaction queries* is similar to the *dashboard queries*, but may add/change a filter, grouping attribute, or add/remove a relation. The key challenge is that users expect fast response times [46], yet joins are notoriously expensive to execute [9, 43, 50]. Traditional techniques, like cubing [28] and indexing [48], are designed for a single table, but poorly handle joins because the denormalized table size can be exponential to the number of relations: $O(n \times f^r)$, where f is the fanout along join graph edges with r relations, each of size n .

Recent factorized query execution techniques [4, 54] speed up queries over large joins by pushing down aggregation through the joins, in the spirit of projection pushdown. This reduces the space overhead (for acyclic joins) to a linear scale: $O(rn)$, making it promising for developing interactive dashboards. However, naive factorized query execution of *interaction queries* online still results in high latency. The process requires scanning, joining, and aggregating all the relations as part of the factorized query execution, which prevents achieving interactive speeds. Recent work [65] has proposed optimizing a pre-determined batch of factorized queries offline. However, *interaction queries* are only determined by the combination of user interactions online. Batching all possible *interaction queries* offline would lead to a combinatorially large overhead.

In this paper, we present Treant, a dashboard accelerator for *interaction queries* over large joins. Offline, the engineering team connects Treant to a DBMS, specifies the join graph (tables and join conditions), and defines Selection-Projection-Join-Aggregation (SPJA) queries (potentially from a BI dashboarding tool) as *dashboard queries* to create visualizations. Treant precomputes compact data structures and stores them as tables in the DBMS; this incurs a constant factor runtime cost relative to running the *dashboard queries*. Online, Treant supports a wide range of *interaction queries* that

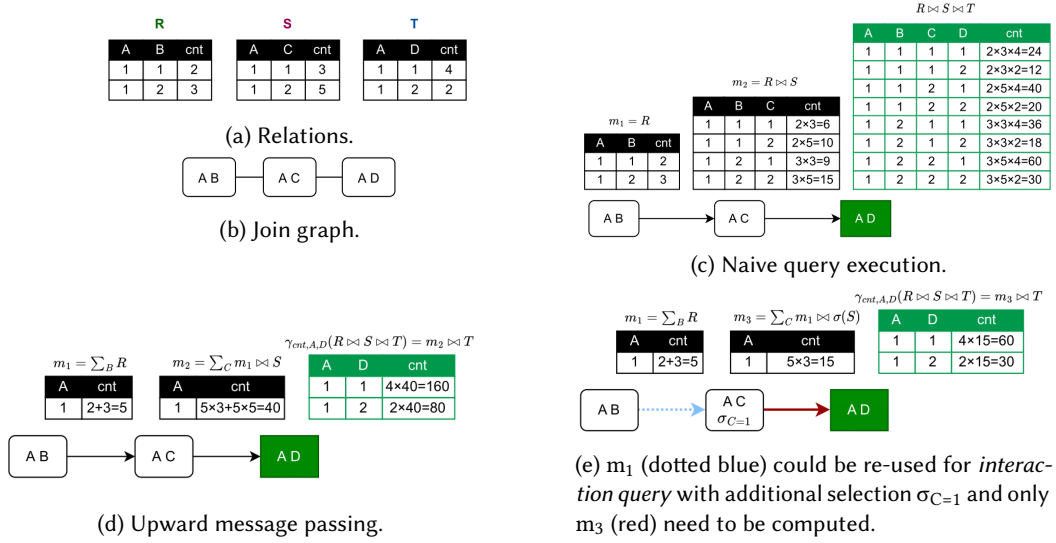


Fig. 2. Example database with three relations, join graph, naive query execution for the total count, and factorized query execution by upward message passing. The final result is in green.

modify select/group clauses, update or remove tables, or join new tables to the *dashboard queries*, all at interactive speeds.

To support efficient aggregation queries over joins, we observe that *interaction queries* differ from the *dashboard query* by keeping the query structure but modifying a subset of the SPJA operators. To this end, we introduce the novel Calibrated Junction Hypertree (CJT) data structure to support work sharing and ensure that the work needed to compute an *interaction query* is proportional to how different it is from its corresponding *dashboard query* (or *interaction query*), rather than its overall complexity. Our design builds on the observation by Abo et al. [5] that factorized query execution can be modeled as message passing in Probabilistic Graphical Models (PGM) [41], as described in the following example:

EXAMPLE 2. Figure 2(a,b) list example relations (duplicates are tracked with a cnt “annotation”) and the join graph, respectively. Consider a dashboard query that computes the total count over the full join result: $Q = \gamma_{cnt}(R \bowtie S \bowtie T)$. Figure 2c naively executes the query, which computes the full join in order $(R \bowtie S) \bowtie T$ before summing the counts, and requires exponential space.

In contrast, factorized query execution distributes the summation through joins, so that each node first sums out (marginalizes) attributes irrelevant downstream, and then emits a smaller message. Any sequence of messages from leaves to root in the join graph results in the correct result. Figure 2d chooses T as the root, then passes messages along $R \rightarrow S \rightarrow T$. AB marginalizes out B and AC marginalizes out C. Therefore, we only sum 2 tuples to compute the final query result.

The benefit of viewing query execution as message passing is that work-sharing opportunities become self-evident. Consider the *interaction query* in Figure 2e, which adds a predicate $C=1$ over $S[AC]$. Factorized query execution would re-pass messages along $R \rightarrow S \rightarrow T$, but misses the opportunity to reuse m_1 . A partial solution is to cache the messages when executing *dashboard query*. However, message contents are sensitive to the message-passing order: if we executed the *dashboard query* along $T \rightarrow S \rightarrow R$, then the message between R and S would differ from m_1 . Thus, *interaction queries* would be forced to use the same (possibly suboptimal) message passing order, or

sacrifice message reuse. An alternative is to enumerate all orders and store their messages, but has compute and storage costs quadratic in the number of relations.

Our algorithmic contribution is to show that after executing the *dashboard query*, sending messages in reverse from root to leaves (e.g., $R \leftarrow S \leftarrow T$) is sufficient to support any execution order over the join graph. We, therefore, use CJT to proactively materialize *dashboard query* messages for all possible orders. Such a process is called "calibration" and is first brought up in PGM [68], which is used to similarly share computation between queries on posterior distributions. We are the first to apply it to SPJA queries.

CJTs allow work sharing between *dashboard queries* and *interaction queries*, and we further extend work sharing between *interaction queries*. The key idea is that, given *dashboard query* q_0 , after a user performs an interaction (e.g., filter by Role as q_1), they may *add* another interaction (e.g., further add a filter by Campaign as q_2). In this case, the difference between the successive *interaction queries* (q_1 and q_2) is smaller than between q_2 and q_0 . Since q_1 is not known offline, we calibrate it online during "think-time" between interactions [22]. Such calibration doesn't need to be fully completed, and the user's next interaction preempts calibration. The next query can benefit from any messages that are newly materialized.

We implement Treant as a Python dashboard accelerator library. Treant acts as a middleware that transparently rewrites queries from dashboards to benefit from factorized execution and work sharing from CJTs. The generated queries are simple and easily ported to different DBMSes or data frame systems [3, 56]. Furthermore, Treant accelerates advanced dashboard features like interactive *Data Augmentation* [15] for analytics or machine learning.

To summarize, our contributions are as follows:

- We design the novel CJT data structure, which improves the efficiency of *interaction queries* over join by reusing messages and applying calibration. The cost of materializing the data structure is within a constant factor of the *dashboard query* execution, but accelerates *interaction queries* by multiple orders of magnitude.
- We build Treant, which manages CJT to transparently accelerate dashboards. It builds CJTs based on initial *dashboard queries*, and uses think time to further calibrate *interaction queries*. Its simple rewrite-based design is easily portable to any SQL-based DBMS.
- We evaluate the effectiveness of Treant on both local and cloud DBMSes using real-world dashboards and TPC benchmarks. Our results show that Treant accelerates most *interaction queries* by $> 100\times$, and speeds up ML augmentation by $10\times$.

2 BACKGROUND

This section provides a brief overview of annotated relations, and variable elimination to accelerate join-aggregation queries, and the junction hypertree for join representation.

Data Model. Let uppercase symbol A be an attribute, $\text{dom}(A)$ is its domain, and lowercase symbol $a \in \text{dom}(A)$ be a valid attribute value. For the purpose of analytical simplicity, we assume categorical attributes with a fixed domain size.¹ Given relation R , its schema S_R is a set of attributes, and its domain $\text{dom}(R) = \times_{A \in S_R} \text{dom}(A)$ is the Cartesian product of its attribute domains. An attribute is incident of R if $A \in S_R$. Given tuple t , let $t[A]$ be its value of attribute A .

Annotated Relations. Since relational algebra (first-order logic) does not support aggregation, it has been extended with the use of commutative structures to support aggregation. The main idea is that tuples are annotated with values from a semi-ring, and when relational operators (e.g., join,

¹However, the system, Treant, doesn't rely on the fixed attribute domain sizes and can trivially support numerical attributes. Treant simply issues SPJA queries to DBMSes (Section 4), which can be executed over relations with numerical attributes.

project, group-by) concatenate or combine tuples, they also multiply or add their annotations, such that the final annotations correspond to the aggregation results.

A commutative semi-ring $(D, +, \times, 0, 1)$ consists of a set D and binary operators $+$ and $\times$ that are commutative and closed over D , along with the zero 0 and unit 1 elements. We focus on semi-rings with elements in D of constant size for efficiency. The semi-ring structure accommodates nearly all standard aggregates such as count, sum, min, max, etc [5]. More complex aggregates and even ML model can be constructed from semi-ring: variance can be derived from count, and sum ($\text{var}(A) = \text{sum}(A^2)/\text{count}(A) - \text{sum}(A)^2/\text{count}(A)^2$), and linear regression can be trained based on the sum of pairwise products among features and target variable. There are two commonly used classes of aggregates not supported: percentile-based (e.g., median), and distinct-based (e.g., distinct count) aggregates. They require the tracking of a distribution or a unique value set, and cannot be represented by a set D of constant-sized elements. These can be approximated for future work [17]. For simplicity, the text will be based on *COUNT* queries and the natural numbers semi-ring $(\mathbb{N}, +, \times, 0, 1)$, which operates as in grade school math. Each relation R annotates each of its tuples $t \in \text{dom}(R)$ with a natural number, and $R(t)$ refers to this annotation for tuple t [29, 36, 52]. We will use the terms *relation* and *annotated relation* interchangeably.

Semi-ring Aggregation Query. Aggregation queries are defined over annotated relations, and the relational operators are extended to add or multiple tuple annotations together, so that the output tuples' annotations are the desired aggregated values².

Consider an example query $\gamma_{A, \text{COUNT}}(R_1 \bowtie R_2 \dots \bowtie R_n)$ that joins n relations, groups by a set of attributes A , and computes the *COUNT*. The operators that combine annotations are joins and groupbys and they compute the output tuple annotations as follows:

$$(R \bowtie T)(t) = R(\pi_{S_R}(t)) \times T(\pi_{S_T}(t)) \quad (1)$$

$$\left(\sum_A R\right)(t) = \sum \{R(t_1) \mid t_1 \in S_R, t = \pi_{S_R \setminus \{A\}}(t_1)\} \quad (2)$$

(1) states that given a join output tuple t , its annotation is the product of counts from the contribution pair of input tuples. (2) defines the count for output tuple t , and $\sum_A R$ denotes that we *marginalize* over A and remove it from the output schema. This corresponds to summing the counts for all input tuples in the same group as t . In this paper, we assume natural joins with identical names for join keys for clarity³. To summarize, join and groupby correspond to $\times$ and $+$, respectively. Let the schema of $R_1 \bowtie R_2 \dots \bowtie R_n$ be S . q can be rewritten as $\sum_{A \in S-A} (R_1 \bowtie R_2 \dots \bowtie R_n)$.

Early Marginalization. In simple algebra (as well as semi-rings), multiply distributes over addition, and can allow us to push marginalization through joins, in the spirit of projection push down [32].

Consider Figure 2, which computes $\gamma_{A, \text{COUNT}}(R \bowtie S \bowtie T)$. We can rewrite it as marginalizing B , C , and D from the full join result $\sum_B \sum_C \sum_D R[A, B] \bowtie S[A, C] \bowtie T[A, D]$. Although the naive cost is $O(n^3)$ where n is the relation size, we can push down marginalizations: $\sum_D (\sum_C ((\sum_B R[A, B]) \bowtie S[A, C]) \bowtie T[A, D])$ where the largest intermediate result, and thus the join cost, is $O(n)$.

Join Ordering and Variable Elimination. Early marginalization is applied to a given join order. Thus we may also reorder the joins to cluster relations that involve a given attribute, so that it can be safely marginalized. Consider the query $\sum_A R[A, B] \bowtie S[B, D] \bowtie T[A, C]$. We can reorder the joins so that A can be marginalized out earlier: $S[B, D] \bowtie \sum_A (R[A, B] \bowtie T[A, C])$. The above procedure, where for each marginalized attribute A , we first cluster and join relations incident to A ,

²Note that this means different aggregation functions are defined over different semi-ring structures, and our examples will focus on *COUNT* queries.

³For the system, different names can be used as long as join conditions are specified. Our approach can be easily adapted to theta joins and outer joins, by multiplying the annotations of matching tuples (non-existing tuples are annotated by zero)

and then marginalize A, is called variable elimination [18]. Variable Elimination is widely used for PGM inference [41] and factorized execution [5]. Variable Elimination reduces the optimization to: (a) identifying the order in which attribute(s) are marginalized out (by clustering and joining the incident relations), and (b) determining the best arrangement of relations within the cluster. (b) is the traditional join ordering problem [69]. DBMSes that employ binary join use information such as relation cardinality to optimize the ordering. Prior works [5, 65] also apply worst-case-optimal-join (WCOJ [51]) to simultaneously join these tables for asymptotic improvement. (a) is called the *variable elimination order* and its complexity is dominated by the intermediate join result size of the clustered relations (using WCOJ). It is well known that finding the optimal order (with the minimum intermediate size) is NP-hard [26]. However, common DBMS queries are over the acyclic join, whose optimal order could be found efficiently by GYO-elimination procedure [5, 76]. The central idea is to repeatedly: (1) eliminate attributes that are present in only one relation, and (2) join relations R with S if the schema of R is a subset of S's schema.

EXAMPLE 3. Consider $\sum_{ABCD} R[A, B] \bowtie S[B, C] \bowtie T[C, D]$, which is acyclic and we apply GYO-elimination: we first eliminate A, D because they each only appear in one relation R, T respectively (the order could also be D, A). This results in the intermediates $M_1[B]$, $M_2[C]$, whose schemas are subsets of S. So we further join M_1, M_2 , with S and eliminate B, C. The final variable elimination order is ADBC:

$$\begin{aligned} & \sum_{DBC} S[B, C] \bowtie T[C, D] \bowtie \sum_A(R[A, B]) \\ &= \sum_{BC} S[B, C] \bowtie M_1[B] \bowtie \sum_D(T[C, D]) \\ &= \sum_C M_2[C] \bowtie \sum_B(S[B, C] \bowtie M_1[B]) = \sum_C(M_3[C] \bowtie M_2[C]) \end{aligned}$$

Each elimination step is highlighted in red.

Junction Hypertree. The Junction Hypertree⁴ (JT) is a representation of a join query that is amenable to complexity analysis [5, 37] and semi-ring aggregation query optimization [4]. Given a join graph $R_1 \bowtie \dots \bowtie R_n$, JT is a pair (E, V) , where each vertex $v \in V$ is a subset of attributes in the join graph, and the undirected edges form a tree that spans the vertices. The join graph may be explicitly defined by a query, or induced by the foreign key relationships in a database schema. Following prior work [5], a JT vertex is also called a *bag*. A JT must satisfy three properties:

- **Vertex Coverage:** The union of all bags must be equal to the set of attributes in the join graph.
- **Edge Coverage:** For every R in the join graph, there exists at least 1 bag that's a superset of S_R .
- **Running intersection:** For any attribute in the join graph, the bags containing the attribute must form a connected subtree.

The last property is important because JTs are related to variable elimination and are used for query execution. Given an elimination ordering, let each join cluster be a bag in the JT, and adjacent clusters be connected by an edge. In this context, executing the variable elimination order corresponds to traversing the tree (path); when execution moves beyond an attribute's connected subtree, then it can be safely marginalized out. Note that since the JT is undirected, it can induce many variable elimination orders from a given JT, all with the same runtime complexity.

Finally, there are many valid JT for a given join graph, and the complexity (query execution cost) of a JT is dominated by the largest bag (the join size of the relations covered by the bag). Although finding the optimal JT for an arbitrary join graph is NP-hard [26], we can trivially create the optimal JT for an acyclic join graph by creating one bag for each relation (e.g., the JT is simply

⁴JT is also called Hypertree Decomposition [5, 37], Join Tree, Join Forest [35, 65] in databases and Clique Hypertree in PGM [41].

the join graph) and the size of each bag is bounded by its corresponding relation size. We refer readers to FAQ [5] for a complete description.

Message Passing for Query Execution. Message Passing was first introduced by Judea Pearl in 1982 [57] (known as belief propagation) in order to efficiently perform inference (compute marginal probability) over probabilistic graphical models. In database terms, each probability table corresponds to a relation, the probabilistic graphical model corresponds to the full join graph in a database (as expressed by a JT), the joint probability over the model corresponds to the full join result, and marginal probabilities correspond to grouping over different sets of attributes. To further support semi-ring aggregation, Abo et al. [5] established the equivalence between factorized query execution, and (upward) message passing. Below, we illustrate how message passing over JT is used for query execution.

The procedure first determines a traversal order over the JT—since the JT is undirected, we can arbitrarily choose any bag as the root and create directed edges that point towards the root—and then traverses from leaves to root. We first compute the initial contents of each bag by joining the necessary relations based on the bag’s attributes. When we traverse an outgoing edge from a bag l to its parent p , we marginalize out all attributes that are not in their intersection—the result is the *Message* between l and p . The parent bag then joins the message with its contents. Each bag waits until it has received messages from all incoming edges before it emits along its outgoing edge. Once the root has received all incoming messages, its updated contents correspond to the query result.

EXAMPLE 4 (MESSAGE PASSING). Consider the relations in Figure 2a, and the JT in Figure 2b where each bag is a base relation. We wish to execute $\sum_{ABCD} R(A, B) \bowtie S(A, C) \bowtie T(A, D)$ by traversing along the path $R \rightarrow S \rightarrow T$ (Figure 2d). We first marginalize out B from AB , so the message to AC is a single row with count 5. The bag AC joins the row with its contents, and thus multiplies each of its counts by 5. It then marginalizes out C , so its message to AD is a single row with count $(3+5) \times 5$. Finally, bag AD absorbs the message (Figure 2d) and marginalizes out A and D to finalize the computation.

3 CALIBRATED JUNCTION HYPERTREE

While message passing over JT exploits early marginalization to accelerate query execution, it has traditionally been limited to single-query execution. This section introduces Calibrated Junction Tree (CJT) to enable work-sharing for interactive dashboards on large joins. The idea is to materialize messages over the JT for *dashboard query*, and reuse a subset of its messages for *interaction queries*. This section will focus on the basis for the CJT data structure and how it is used to execute *interaction queries*. The next section will describe how Treant applies CJT to build an interactive dashboard.

Our novelty is (1) to use JTs as a concrete data structure to support message reuse, and (2) to borrow *calibration* [68] from PGM to materialize messages for any message passing order. Although CJT is widely used across engineering [60, 78], ML [13, 20], and medicine [42, 58], it was used only for probabilistic inference (sum over probability); we are the first to extend it to general SPJA queries with semi-ring aggregations in DBMS for work sharing.

3.1 Motivating Example

We illustrate the work sharing between a *dashboard query* $Q_1 = \sum_{ABCD} R(A, B) \bowtie S(A, C) \bowtie T(A, D)$, and an *interaction query* $Q_2 = \sum_{ABCD} R(A, B) \bowtie \sigma_{C=1}(S(A, C)) \bowtie T(A, D)$ with additional predicate $C=1$, to motivate CJT.

EXAMPLE 5. Consider the JTs in Figure 3a which assign AD as the root for Q_1, Q_2 and traverse along the path $R \rightarrow S \rightarrow T$. Although the message $R \rightarrow S$ will be identical (blue edges), the additional filter over S means that its outgoing message (and all subsequent messages) will differ from Q_1 ’s and

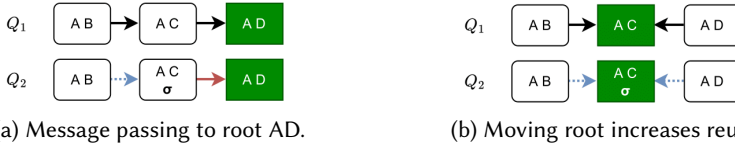


Fig. 3. Work sharing between queries Q_1 (total count query) and Q_2 (additional selection to S). Dotted blue edges are reusable messages and solid red edges are non-reusable.

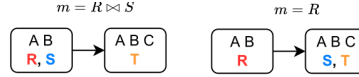


Fig. 4. The same JT over relations $R(A,B)$, $S(A)$, $T(B,C)$ can have different mappings (χ) and each results in different messages (m). For each bag, its attributes are at the top and mapped relations are at the bottom.

cannot be reused (red edges). In contrast, Figure 3b uses S as the root, so both messages can be reused and the S bag simply applies the filter after joining its incoming messages.

This example shows that message reuse depends on how the root bag is chosen for *dashboard query* (Q_1), and for different *interaction queries*, we may wish to choose different roots. Since we don't know the exact join, grouping, and filter criteria of future *interaction queries*, the naive solution is to (costly) materialize messages for all possible roots. We next present CJT, a novel data structure for query execution and message reuse, and address these limitations.

3.2 Junction Hypertree as Data Structure

A naive approach to re-use messages is to execute a query over JT, and cache the messages; when a future query traverses an edge in the JT, it reuses the cached message. Unfortunately, this is 1) inaccurate, because messages generated along an edge are not symmetric and depend on the specific traversal order during message passing, 2) insufficient, because it cannot directly express filter-group-by queries, and 3) leaves performance on the table. To do so, we extend JT as follows:

Directed Edges. To support arbitrary traversal orders, we replace each undirected edge with two directed edges, and use $\mathcal{J}(i \rightarrow j)$ to refer to the cached message for the directed edge $i \rightarrow j$.

Relation Mapping. $\chi(R)$ maps each base relation R to exactly one bag containing R 's schema. Although different mappings can lead to different messages (Figure 4), acyclic join graphs have a good default mapping where each bag is mapped by a single relation. Relations mapped to the same bag are joined during message passing.

Empty Bags. To avoid large paths during message passing, it's beneficial to add custom *empty* bags to create "short cuts". *Empty* bags are not mapped from any relations and are simply a mechanism to materialize custom views for work sharing. They join incoming messages, marginalize using standard rules, and materialize the outgoing messages. Empty bags are a novel addition in this work: previous works [4, 5, 74] focus on non-redundant JT without empty bags. This is because they are in the context of single query optimization, where empty bags offer no advantage.

EXAMPLE 6 (EMPTY BAG). Consider the simplified TPC-DS JT in Figure 5a. *Store Sales* is a large fact table (2.68M rows at $SF=1$), while the rest are much smaller. To accelerate a query that aggregates sales grouped by (Store, Time), we can create the empty bag *Time Stores* between *Store_Sales*, *Time* and *Stores* (Figure 5b). The message from *Store_Sales* to the empty bag is sufficient for the query and is $17.3\times$ smaller (154K rows) than the fact table.

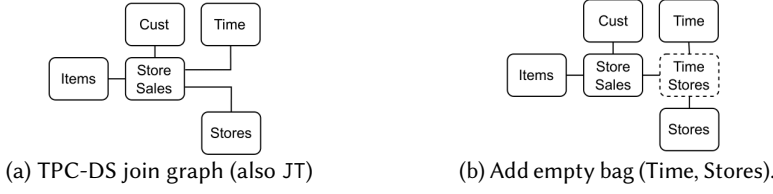


Fig. 5. Simplified Join graph of TPC-DS. Adding an empty bag can accelerate queries group-by Time and Stores.

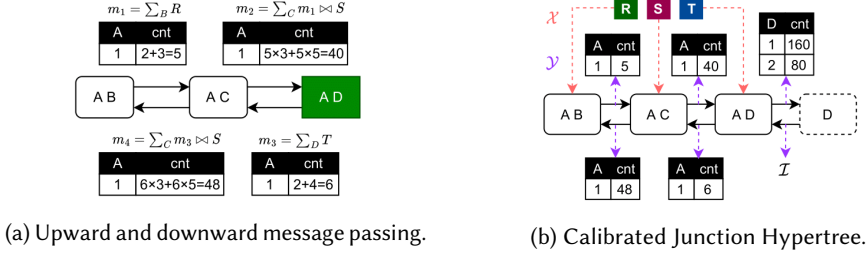
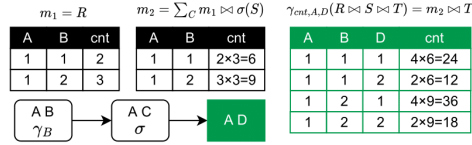


Fig. 6. Message Passing and Calibration. Green rectangle is the root. Dotted one is the empty bag. $\mathcal{I}$ is identity relation. $\mathcal{X}$ maps relations to bags, and $\mathcal{Y}$ maps edges to messages.



Note that leaf empty bag may result in an empty output message; we avoid this by mapping the identity relation $\mathcal{I}$ ⁵ to it, such that $R \bowtie \mathcal{I} = R$ for any relation R . Essentially, the empty bag is “pass-through” and doesn’t change the join results nor the query result. When the bag is a leaf node, its message is simply $\mathcal{I}$. We do not materialize the identity relation, as it’s evident from the JT.

EXAMPLE 7 (JT DATA STRUCTURE). Figure 6b illustrates the JT data structure for the example in Figure 2. Each relation maps to exactly one bag (orange dotted arrows), and each directed edge between bags (black arrows) stores its corresponding message (purple dashed arrows). Bag D (dotted rectangle) is an empty bag and materializes the view of “count group by D”. $\mathcal{I}$ is the identity relation.

3.3 Message Passing Over Annotated Bags

We now describe support for general SPJA queries over JT. Although each query JT has the same structure, we annotate the bags based on the query’s SPJA operations. We then modify message passing rules to accommodate the bag annotations. These annotations will come in handy when determining work-sharing opportunities for a new interactive query given a *dashboard query*.

Given $\mathbf{R} = \{R_1, R_2, \dots, R_n\}$ and $\text{JT} = ((E, V), \mathcal{X}, \mathcal{Y})$, we focus on SPJA queries of the following form:

SELECT $\mathcal{G}$, COUNT(*) FROM $\mathcal{J}$ WHERE [JOIN COND] AND $\mathcal{P}$ GROUP BY $\mathcal{G}$

where $\mathcal{G}$ is the grouping attributes, $\mathcal{J} \subseteq \mathbf{R}$ is the set of relations joined in the FROM clause, and $\mathcal{P}$ is the set of predicates referencing attributes in one bag. Query execution is based on message passing (Section 2). However, the processing of each bag differs based on annotations. We propose 4 annotation types, summarized in Table 1:

⁵The schema is the same as the bag and all tuples in its domain are annotated with 1 element in the semiring.

Anno Effect**Applicability**

γ_A	Prevent A from being marginalized out for all downstream bags.	Any bag containing A.
$\sum_A$	Marginalize out A. "Cancels" γ_A for downstream bags.	Any bag containing A.
$\bar{R}$	Exclude relation R from the bag during message passing.	The bag $\mathcal{X}(R)$.
$R_{\text{ver.}}^*$	Update relation R in the bag to the specified version during message passing.	The bag $\mathcal{X}(R)$.
σ_{id}	Apply selection uniquely identified by id to relations during message passing.	Any bag covers atts.

Table 1. Table of annotations, their effects and applicability.

- **GROUP BY $\mathcal{G}$** . For each attribute $A \in \mathcal{G}$, we annotate exactly one bag u that contains this attribute with γ_A . Messages emitted by the annotated bag and all downstream bags do not marginalize out A. Since all bags containing A form a connected subtree (running intersection), which bag we annotate does not affect correctness. However, we will later discuss the performance implications of different choices when we use CJT to queries.
- **Joined Relations $\mathcal{J}$** . The query may not join all relations in the join graph, or the joined relations are updated. For each relation R *not* included (resp. updated) in the query, we annotate the corresponding bag $u = \mathcal{X}(R)$ with $\bar{R}$ (resp. $R_{\text{ver.}}^*$). When computing messages from this bag, R will be excluded from $\mathcal{X}^{-1}(u)$ ⁶ (resp. R will be updated in $\mathcal{X}^{-1}(u)$). We allow only the exclusions of relations that don't violate JT properties.
- **PREDICATES $\mathcal{P}$** . Let predicate $\sigma \in \mathcal{P}$ be over attribute **A**. We choose a bag u such that $\mathbf{A} \subseteq u$, and annotate it with σ_{id} —the effect is that the predicate filters all messages emitted by u . The choice of bag to annotate is important—for a single query, we want to pick a bag far from the root in the spirit of selection push down, whereas to maximize message re-usability, we want to pick the bag near the root. We discuss this trade-off in Section 3.3.4.

3.3.1 Message Passing. We now modify how message passing, generation, and absorption work to take the annotations into account.

Upward Message Passing. Traditional message passing chooses a root bag and traverses edges from leaves to the root. Since JT uses bidirectional edges, we call this "upward message passing". The message $\mathcal{Y}(b \rightarrow p)$ from bag b to parent p is defined as follows: Let $M(b, p) = \{\mathcal{Y}(i \rightarrow b) \mid i \rightarrow b \in E \wedge i \neq p\}$ be the set of incoming messages (except from p). We join between all relations (updated to the specified versions) in $M(b)$ and $\mathcal{X}^{-1}(b)$, and marginalize out all attributes not in p . Given annotations, we exclude relations in $\bar{R}$ from the join, apply predicates σ (with appropriate push-down), and exclude attributes in γ : $\mathcal{Y}(b \rightarrow p) = \sum_{b - (p \cap b) - \gamma} \sigma(\bowtie (M(b, p) \cup \mathcal{X}^{-1}(b) - \bar{R}))$. b 's message to p is ready iff all its messages from child bags are received. During message passing, if b contains group-by annotation γ , we temporarily annotate all its downstream bags also with γ .

Absorption. Absorption is when the root bag r consumes *all* incoming messages. It is identical to the join and filter during message generation: $\text{Absorption}(r) = \sigma(\bowtie (M(r, \emptyset) \cup \mathcal{X}^{-1}(r) - \bar{R}))$. To generate the final results, we marginalize away all attributes not in $\mathcal{G}$.

EXAMPLE 8. Consider database and JT in Figure 2. Suppose we want to query the total count filter by $C = 1$ and group by B . This requires us to annotate bag AB with γ_B and bag AC with σ (*id* is omitted). Figure 7 shows the upward message passing over the annotated JT to root AD , where attribute B is not marginalized out and the predicate $C=1$ is applied to S . After upward message passing, bag AD performs absorption and marginalizes out AD to answer the query.

3.3.2 Runtime Complexity. The main purpose of this runtime complexity analysis is to serve as a baseline for analyzing the runtime benefits of the CJT in Section 3.4.3. Our analysis largely follows

⁶Rigorously, $\mathcal{X}$ doesn't have an inverse function. We define $\mathcal{X}^{-1}(u)$ to be $\{i \mid \mathcal{X}(i) = u\}$.

FAQ [5], with extensions to consider the effects of selection (by selectivity) and group-by (by attribute domain size) annotations. While FAQ also supports group-by, it's by modifying variable elimination to construct a new JT, and bounding JT by relation sizes. We first use an example to contrast the difference:

EXAMPLE 9. Consider relations $R(A, B), S(A, C)$ with a JT of $(AB) - (AC)$; each relation and attribute domain size is $O(n)$ and $O(d)$ respectively. For the query $\gamma_{B,C,\text{count}(*)}(R \bowtie S)$, Treant annotates (AB) with γ_B and (AC) with γ_C , leading to a JT of $(AB\gamma_B) - (AC\gamma_C)$. If (AC) is the root, the absorption of (AC) is bounded by $O(nd)$ (increased by $O(d)$ for γ_B). But in FAQ, this is viewed as variable elimination of only A , leading to JT of a single bag (ABC) with absorption size $O(n^2)$.

Our extension offers two benefits: (1) it improves message reuse as compared to running variable elimination to construct a new JT, and (2) dashboards typically have few group-by attributes with domain sizes much smaller than relation sizes, which improve the bounds⁷. Next, we lay out the setting for our analysis:

- **Measure:** We measure complexity in terms of both the query and database sizes using the standard RAM model of computation.
- **Query:** We focus on a SPJA query over an acyclic⁸ natural join of n relations $R_1, \dots, R_n$ with attribute domains of size $O(d)$. The selections and group-bys have been annotated in JT (below).
- **Annotated JT:** Let the annotated JT($E, V, \mathcal{X}, \mathcal{Y}$) over n bags be the SPJA query. Each bag maps to exactly one relation, whose schema is the same as bag attributes. To simplify notation, we denote the relation of bag u as $R_u = \mathcal{X}^{-1}(u)$. For annotations:
 - **Selection:** $s(u)$ denotes the combined selectivity of all σ on u .
 - **Group-by:** $g(u, v)$ denotes the number of attributes to group-by from the upstream bags to $u \rightarrow v$ (excluding the group-bys in u). Note that the $\sum$ annotations can cancel out the corresponding group-bys and are not counted in $g(u, v)$.
 - **Update:** We assume R_u references the latest relation.
 - **Exclusion:** While relation exclusion is common for multi-relation bag (e.g., for graph analytics [62]), it's rare for single-relation bag, and may lead to disconnected join graph, Cartesian products, and size blowup (except for leaf bag). For simplicity, we assume no relation exclusion.
- **Query Execution:** Following FAQ [5], we analyze the WCO LeapFrog Triejoin [71]. Given a natural join of n relations, m unique attributes, maximum input relation size N , and the fractional edge cover bound [11] of join size ρ (based on the join graph and relation sizes), the runtime complexity is $O(mn \cdot \rho \cdot \log N)$. For SPJA, we pre-apply selection, join, and use standard hash-based aggregation in $O(\rho)$; join dominates the aggregation cost.

PROPOSITION 1 (RUNTIME COMPLEXITY). Executing SPJA query naively takes $O(|V| \cdot |\cup_{v \in V} v| \cdot \rho \log(\max_{v \in V} s(v) \cdot |R_v|) + \sum_{v \in V} |R_v|)$, where ρ is the fractional edge cover bound of join size over selected relations. For message passing over JT, given root $r \in V$, define $\text{Tra}(r, \text{JT})$ as the set of pairs (u, v) for all directed edges $u \rightarrow v$ on the path to r in E . The complexity is then $O(\sum_{u,v \in \text{Tra}(r, \text{JT}) \cup \{(r, \emptyset)\}} \mathcal{M}(u, v) + |R_u|)$, where $\mathcal{M}(u, v) = (|U(u, v)| + 1)(|u| + g(u, v))S(u, v) \log \max(\max_{x \in U(u, v)} S(x, u), s(u)|R_u|)$ is the message passing $(u \rightarrow v)$ cost, $U(b, p) = \{i | i \rightarrow b \in E \wedge i \neq p\}$ is the set of b 's neighbour bags (except p) and $S(u, v) = d^{g(u, v)} s(u)|R_u|$ is the size bound of message from u to v .

PROOF SKETCH. The naive SPJA query execution runtime complexity is the sum of leapfrog triejoin and selection cost (in $O(\sum_{v \in V} |R_v|)$). Message passing analysis is similar with one difference: messages

⁷If there are many group-by attributes with large domains, the bound can be tightened by fractional edge cover based on the relation sizes and functional dependency [6].

⁸Or cycles has been pre-joined, following standard hypertree decomposition [5, 37].

are aggregated results. This allows us to bound the join output sizes by the domain size of group-by attributes. For each message $u \rightarrow v$, we join u with all its incoming messages and aggregate; the cost is dominated by the join. Without group-by annotation, the join size is bounded by $O(s(u)|R_u|)^9$. With group-by annotations, the join (and message) size increases by a factor of the group-by attributes' domain size $O(d^{g(u,v)})$. Absorption for r follows the same analysis. The final complexity sums selection, message passing, and absorption.

Beside the polynomial difference in query complexity, join sizes are the core difference in data complexity: naive query execution materializes joins ($O(p)$), which can be prohibitive. In contrast, factorized execution restricts the join size to $d^{g(u,v)}s(u)|R_u|$ for each message passing $u \rightarrow v$, which in practice is much smaller.

3.3.3 Single-query Optimization. For a given SPJA query, we can choose different bags to annotate, and different roots for upward message passing. We make these choices based on heuristics that minimize the runtime complexity. Since relation removal and update annotations $\bar{R}, R_{ver}^*$ can be only placed on $X(R)$, and the placement of group-by don't affect the message passing, the only factor is the choice of root bag and selection annotations. We enumerate every possible root bag, greedily push down selections, and choose the root with the smallest complexity; the total time complexity to find the root is polynomial in the number of bags.

3.3.4 Message Reuse Across Queries. Messages reuse between queries requires that the message along edge $u \rightarrow v$ only depends on the annotated sub-tree rooted at u . Thus, a new query can reuse materialized messages in CJT that have the same subtree (and annotations).

PROPOSITION 2 (MESSAGE REUSABILITY). *Given a JT and annotations for two queries, consider the directed edge $u \rightarrow v$ present in both queries. Let T_u be the subtree rooted at u . If the annotations for T_u are the same for both queries, then the message along $u \rightarrow v$ will be identical irrespective of the traversal order nor choice of the root.*

This proposition is well established in PGM [68], and follows for message passing over JT. The proof sketch is as follows: leaf nodes send messages that only depend on outgoing edges, base relations and annotations, while a given bag's outgoing message only depends on its mapped relations (X), annotations and incoming messages. None of these depend on the traversal order nor the root. Proposition 2 implies that an annotation can "block" reuse along all of its downstream messages. For group-by annotation, we greedily push down it to the leaf of the connected subtree closest to the root to maximize reusability.

3.4 Calibration

We saw above that message reuse depends on choosing a good root for message passing, however upward message passing only materializes messages for a single root. *Calibration* materializes messages for all roots, letting future queries pick arbitrary roots.

3.4.1 Calibration. Given an edge $u \rightarrow v$, u and v are calibrated iff their marginal absorption results are the same in both directions: $\sum_{u-(v \cap u)} \text{Absorption}(u) = \sum_{v-(v \cap u)} \text{Absorption}(v)$. The JT is calibrated if all pairs of adjacent bags are calibrated. We call this a *Calibrated Junction Hypertree* (CJT), which is achieved by Downward Message Passing discussed next.

Downward Message Passing. Upward message passing computes messages along half of the edges from leaves to root. Downward Message Passing simply reverses the edges and passes messages from the root (now the leaf) to the leaves (now all roots). Now, all directed edges store messages.

⁹The effects of selections from upstream bags are not accounted for simplicity; standard cost estimation [67] can approximate the combined selectivity for optimization.

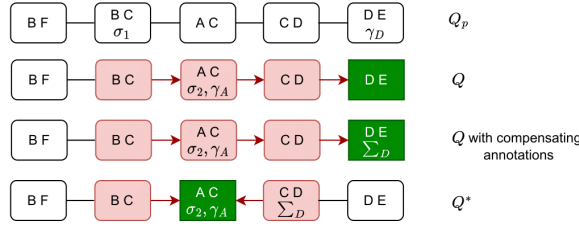


Fig. 8. Given CJT of Q_p , the Steiner tree to execute Q is highlighted (green is root and red is non-root nodes). Compensating annotation $\sum_D$ is introduced to compensate γ_D , and can be moved for better plan. Q^* is the optimal query with minimum Steiner tree size and runtime complexity.

EXAMPLE 10. Consider the example in Figure 6a. During upward message passing, root AD receives the message from leaf AB. After that, we send messages back from AD to AB. We can verify that the JT is calibrated by checking the equality between the absorptions.

Calibration means all bags are ready for absorption. This immediately accelerates the class of queries that further adds one grouping or filtering over attribute A. We simply pick a bag containing A and apply the filter/group-by to its absorption result.

3.4.2 Query Execution Over a CJT. How do we execute a new interactive query Q over the CJT of a dashboard query Q_p ? Since they share the same JT structure, they only differ in their annotations. The main idea is that query execution is limited to the subtree where the annotated bags differ between the two queries, while we can reuse messages for all other bags in the CJT.

Let $\mathbf{A}_p$ and $\mathbf{A}$ be the set of annotations for Q_p and Q , respectively; note that the annotations in $\mathbf{A}_p$ are bound to specific bags in the CJT, while the annotations in $\mathbf{A}$ are not yet bound. Further, let $\mathbf{B}_D$ be the subset of bags whose annotations differ between the two queries. The minimal **Steiner tree** T is the subtree in the CJT that connects all bags in $\mathbf{B}_D$ using the least number of bags. Owing to the simplicity of tree structure, the minimal Steiner tree for a specified set of $\mathbf{B}_D$ can be identified efficiently. From Proposition 2, edges that cross into T have the same messages as in the CJT and can be reused. Thus, we can only perform upward message passing inside of T . Let us first start with an illustrative example:

EXAMPLE 11 (STEINER TREE). In Figure 8, the dashboard query Q_p groups by D and filters by $B = 1$, and so its annotations are $\mathbf{A}_p = \{\sigma_1, \gamma_D\}$. Suppose query Q (row 2) instead groups by A and filters by $C = 1$ ($\mathbf{A} = \{\sigma_2, \gamma_A\}$), and we place its annotations σ_2 and γ_A on AC. The two queries differ in bags $\mathbf{B}_D = \{BC, AC, DE\}$, and we have colored their Steiner tree. Therefore, we can reuse the message $BF \rightarrow BC$, but otherwise re-run the upward message passing along the Steiner tree.

Although the example allows us to reuse one message, it's sub-optimal because $\mathbf{B}_D$ can be further compressed, and the root is poorly chosen. Instead, we use a greedy procedure: we arbitrarily place the annotations on valid bags to create an initial Steiner tree, and then greedily shrink it. Given the minimal Steiner tree over shrunked the $\mathbf{B}_D$, we find the optimal root following Section 3.3.

Initialization. For annotations only in $\mathbf{A}$, they are added to Q 's JT based on the single-query optimization rules in Section 3.3. For annotations only in $\mathbf{A}_p$, we need to compensate for their effects. For σ_p and $\bar{R}$, we remove the annotation, while for γ_D , we introduce the *compensating annotation* $\sum_D$, which marginalizes out D, and place it on the same bag. A unique property of $\sum_D$ is that we can freely place it on any bag that contains D. For all of the above annotations, we add their bags to $\mathbf{B}_D$. This defines the initial Steiner tree:

EXAMPLE 12. The third row in Figure 8 adds the compensating annotation $\sum_D$ to DE. Its execution is as follows: BC doesn't apply $B = 1$, AC applies $C = 1$, groups by A, and DE marginalizes out D, E.

Shrinking. Given the leaves of the Steiner tree, we try to move the differed annotations of Q toward the interior of the tree. Recall that σ , γ , and $\sum$ can be placed on any bag containing the annotation's attribute. We greedily choose the bag with the largest underlying relation and move its annotations first to reduce the Steiner tree.

EXAMPLE 13. Q^* in Figure 8 shows the optimal execution plan over the minimal Steiner tree for Q. It has moved $\sum_D$ to CD, and made AC the root. CD will marginalize out D, and AC performs the filter and group-by. In this way, we also reuse the message $DE \rightarrow CD$.

3.4.3 Runtime Benefit. After optimization, the query execution runtime complexity with CJT is primarily dictated by the difference between the current query and the CJT, as measured by the size of the Steiner tree, and is never worse than naive execution without the CJT. The proof sketch is as follows: Consider naive query execution over $JT(E, V, \mathcal{X}, \mathcal{Y})$ with root r . With CJT, we can share the message outside the Steiner tree $ST(E' \subseteq E, V' \subseteq V)$. We analyze the simple case of message passing over CJT with the original annotation placements *without shrinking* and the same root choice r , which is sufficient to show the runtime benefit; better root choices and shrinking can further improve the time complexity. Let $r' = r$ if $r \in V'$; otherwise, r' is the bag $\in V'$ closest to r . The runtime complexity for query execution with CJT with root r becomes:

$$O(\sum_{u,v \in (\text{Tra}(r, JT) \cup \{(r, \emptyset)\}) \setminus (\text{Tra}(r', JT) \setminus \text{Tra}(r', ST))} \mathcal{M}(u, v) + |R_u|)$$

The core benefit of CJT is to enable the sharing of messages highlighted in **red**. These messages have all upstream bags outside of ST, whose annotations are thus the same. They can therefore be shared according to Proposition 2. As we can see, similar queries require fewer annotations, and thus a smaller ST that requires the same or fewer messages. The benefits are particularly pronounced for imbalanced relation sizes, such as a snowflake schema, where changes are in the small dimension tables. Messages from the fact table can be shared to avoid joining and aggregating the fact table.

Applying CJT to Dashboard and Challenge. CJT is highly suitable for the interactive dashboard: Offline, we build CJT for *dashboard query*. Online, given *interaction query*, CJT ensures that computation needed is proportional to the difference between it and dashboard query (Steiner tree); such a difference is generally small, thanks to the incremental nature of the dashboard interactions. However, one challenge arises when users submit multiple *interaction queries*, each building upon the previous one. As the number of iterations increases, the Steiner tree is likely to expand due to the growing differences. Ideally, we would want to calibrate not only *dashboard query* but also *interaction queries*. However, *interaction queries* are only available online; calibrating them will slow down user interactions. In the next section, we present an optimization to hide such a slowdown. The insight is that users typically have "think-times" [22] during interactions; we leverage it to calibrate *interaction queries* in the background without affecting users.

4 SYSTEM OVERVIEW

In this section, we provide the overview of Treant, a dashboard accelerator that manages CJT to support interactive queries over join. We discuss Treant's usage, architecture, and optimizations.

4.1 Usage Walkthrough

We describe the detailed process of using Treant to build and use an interactive dashboard. Treant has both offline and online stages.

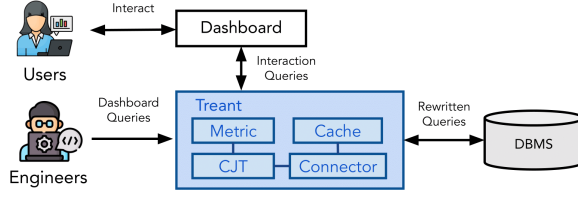


Fig. 9. Treant architecture.

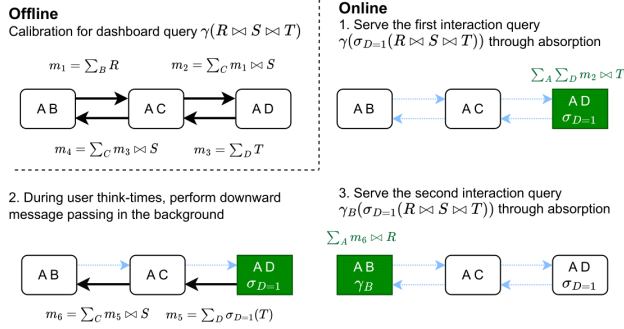


Fig. 10. Example CJT management by Treant.

4.1.1 Offline Stage. The engineering team gathers data and constructs the dashboard offline through the following steps:

Define Metrics. Different domain users have different metrics of interest. For example, the sales department is interested in revenue, while the marketing department is interested in return on investment (ROI). The engineering team defines them as semi-ring aggregation (Section 2) which express a wide range of aggregations.

Construct Join Graph. In enterprise data warehouses, there are typically a large number of tables for metrics and enrichment dimensions (join). The engineering team constructs a join graph that specifies these tables and the join conditions.

Build Visualization. The engineering team specifies the visualizations in the dashboard. Each visualization encodes data from a *dashboard query*. For instance, a bar chart encodes a *dashboard query* with one group-by, while a heatmap encodes one with two group-bys. Treant provides basic visualizations for the dashboard, but is also compatible with any external visualization system.

Finally, Treant takes as input the *dashboard queries* (with metrics as semi-ring aggregations and join graphs in join clauses), connects to DBMS and pre-processes them for dashboard interactions.

4.1.2 Online Stage. Domain users navigate dashboard to analyze metrics of interest. They interact with the dashboard through widgets, which in turn trigger *interaction queries* that modify the initial *dashboard query*. For example, a drop-down menu can modify the group-by attribute, while a slider can change the selected month of the *dashboard query*. Treant supports *interaction queries* that:

- modify select/group clause of the *dashboard query* (Section 3.3)
- update or remove table in the join clause (Section 3.3)
- join with new table that create/affect one bag (Section 4.3)

4.2 Architecture

We first describe how Treant manages CJT for the interactive dashboard, and then delve into the internal components.

4.2.1 Management of CJTs. Treant manages CJTs for work sharing between *dashboard queries* and *interaction queries*. Treant first builds CJTs for *dashboard queries* offline to accelerate *interaction query* online. However, users are likely to engage in more interactions that incrementally modify the previous *interaction query*. Treant further calibrates *interaction query* online: for each visualization (*dashboard query*) and user session, Treant builds CJT for the latest *interaction query* during users' "think-times" [22] in the background. Note that the calibration *doesn't need to be complete* and is halted on receiving the next *interaction query* to not degrade interactivity. Treant can use the partially finished CJT and take advantage of the finished messages to speed up *interaction query*.

EXAMPLE 14. We illustrate the CJT management using the example join graph (Figure 2b) in Figure 10. Given a visualization of a single number with dashboard query $Q_1 = \gamma(R \bowtie S \bowtie T)$, Treant builds its CJT offline. During online phase, user interact with the dashboard with interaction query $Q_2 = \gamma(\sigma_{D_1}(R \bowtie S \bowtie T))$, and Treant uses Q_1 's CJT to share messages. During user think-times, Treant calibrates Q_2 in the background. User performs the next interaction with interaction query $Q_3 = \gamma_B(\sigma_{D_1}(R \bowtie S \bowtie T))$, and Treant uses Q_2 's CJT.

4.2.2 Internal Components. Treant internals are shown in Figure 9. In contrast to previous factorized systems [40, 65] that use custom engines, Treant is a middleware that sits between the dashboard and users' DBMSes. It takes a *dashboard query* or *interaction query* as input, applies *pure query rewriting*, uses CJT to determine the necessary messages to be computed, and computes messages by issuing SPJA queries to DBMSes. This makes Treant portable to any DBMS that executes SPJA queries. The contents of messages (SPJA query results) are stored as tables in the DBMS and Treant stores the pointers (table names) to these messages.

Offline, Treant establishes a connection to DBMS through *connectors*. For each visualization, the data engineer specifies its *dashboard query* (with semi-ring aggregation) and Treant stores it in the *metric* component. Then, Treant re-writes *dashboard query* for message passing and builds CJT to pre-compute messages.

Online, Treant takes an *interaction query* as input, finds the corresponding CJT of previous *interaction query* (Section 4.2.1) based on the user session and the visualization, if available, or uses the *dashboard query* otherwise. Then Treant uses that CJT to pass messages only within the Steiner tree (Section 3.4.2), performs absorption, retrieves the results and sends them to the dashboard for visualizations. The created messages are similarly stored within the DBMS, and only the pointers are returned. In the background, Treant further calibrates *interaction query* (Section 4.2.1)

Finally, while the focus of Treant is to share messages from the most recent CJT (the current dashboard state) for single-user session, there can be work-sharing opportunities from (1) prior partially calibrated JT and (2) CJT across user sessions. At present, Treant takes advantage of these opportunities through a message-level *cache*: For each message, Treant encodes its query definitions and the upstream sub-tree of the messages as the cache key (adequate to uniquely identify the message as per Proposition 2), with the message pointer as value. Before sending a message query to the DBMS, Treant checks if the message is cached. The cache utilizes an LRU replacement policy by default, but refrains from removing messages if they are referenced by CJTs from *dashboard query* or active user sessions. Upon removal of a message from the cache, Treant submits a deletion query for the message to the DBMS.



Fig. 11. Augmenting the join with DE. The Steiner tree is $AD \rightarrow DE$ with root DE and requires 1 message (red).



Fig. 12. Database schema. Red relation is the largest.

4.3 Augmentation Optimization

Simple ML models like linear regression are widely used to examine attribute relationships in the dashboard [49]. Data and feature augmentation [15] further identify datasets to join with an existing training corpus in order to provide more informative features, and is a promising application on top of data warehouses and markets [15, 23, 24]. However, the major bottleneck is the cost of joining each augmentation dataset and then retraining the ML model.

The SOTA factorized ML [19, 52, 65, 66] avoids join materialization when training models over join graphs. First, it designs semi-ring structures for common models (linear regression [66], factorization machines [64], k-means [19]), and then performs upward message passing through the join graph. If we augment with relation r , then factorized learning approaches execute the message passing through the whole augmented join graph again.

In contrast, CJT allows us to choose any bag b that contains the join keys, construct an edge $b \rightarrow r$, and perform message passing using r as the root. In this setting, the Steiner tree is exactly 2 bags, and the rest of the messages in the CJT can be reused. For instance, Figure 11 shows a join graph $AB \rightarrow AC \rightarrow AD$ that we augment with DE. The Steiner tree is simply AD and DE, and we only need to send one message to compute the updated ML model.

4.4 Limitations and Future Works

The focus of this paper is on the message sharing opportunities, demonstrated through a simplified class of SPJA queries. There are limitations in the types of queries currently supported, and we aim to enhance expressiveness in future work to broaden applicability.

Limitations. The current implementation uses parameterized SPJA based on exact matches, as per the format in Section 3.3 and does not parse general SQL strings that can be rewritten to match this format. In terms of expressiveness, Treant supports predicates over only attributes in a single bag for selection; for a predicate clause that references attributes from different bags (e.g., $T.a = R.b$), Treant treats it as a post-processing of group-bys, which could be inefficient. Selection over aggregate query results is unsupported. For aggregation, Treant only supports semi-ring aggregations; while semi-ring can express almost all commonly used aggregations, percentile-based (e.g., median), and distinct-based (e.g., distinct count) aggregates are not supported. For augmentation, Treant currently allows changes that create or affect a single bag; augmentations with attributes spanning multiple bags are not permitted. Beyond SPJA queries, Treant supports ORDER BY and LIMIT, but as a naive post-processing over the SPJA query result that's not optimized.

Future works. We aim to support semantically equivalent rewrites using existing methods like SPES [77]. For selections that reference attributes spanning multiple bags, recent optimization [39] can be applied to build range searching data structure for inequality predicates. For predicate that references results from other SPJA queries, a hierarchical CJT [73] that references other CJTs is a

prospective direction. For percentile-based aggregations, we plan to apply semi-ring approximations [17]. Dealing with augmentations where join keys span multiple bags is another challenge we hope to address. For ORDER BY and LIMIT, we plan to integrate TOP-N optimizations [21] and approximations from MAP [16, 72] with Treant.

5 EXPERIMENTS

Can Treant support *interaction queries* at interactive speeds? What is the overhead? How well can Treant handle more complex applications like ML augmentation? We conducted experiments on both a single node DBMS (DuckDB [59]) and a cloud DBMS (Redshift).

5.1 Single-node DBMS Experiments

Our single-node DBMS experiments are designed to emulate the settings of local processing on a laptop. We evaluate Treant on DuckDB [59] due to its popularity: DuckDB is an OLAP DBMS with superior single-node performance and seamless integration with Python/R data analytics libraries.

Setup. We use two datasets for dashboards: **Salesforce**[12], a public dataset provided by Sigma Computing for CRM and marketing analysis with 36 numerical and 229 categorical attributes; and **Flight**[53], a real-world dataset with 27 numerical and 5 categorical attributes, commonly used in interactive data exploration [22]. However, **Salesforce** is a demo dataset of only 27MB, while modern laptop can process > 10GB data with DuckDB. To address this discrepancy, we employ the data scaler in IDEBench [7] to scale both datasets. The scaling process involves denormalization, estimating the distribution, sampling rows from the distribution, and finally normalizing the table through vertical partitioning. We scale **Salesforce** to 50M rows for 13.7GB, and **Flight** for 300M rows for 15GB. The final normalized schemas are in Figure 12.

For ML augmentation, we use the **Favorita** [2] dataset of purchasing and sales forecasts, widely used in prior factorized ML [65, 66] (see Figure 17 for the schema). **Sales** is the largest relation (241MB), while the others are < 2MB. All experiments were run on GCP c2d-standard-4 (4 vCPUs, 16 GB RAM).

5.1.1 Salesforce Dashboard. We conduct experiments on the Salesforce dashboard [12] built by Sigma Computing.

Workloads. The Salesforce dashboard tracks various metrics such as pipeline, and productivity, which are all computed using the sum aggregations. However, we find that the specific metric chosen has little impact on the query performance. Therefore, we randomly choose the total pipeline amounts as the metric.

We consider two types of visualizations from the Salesforce dashboard illustrated in Figure 1b: a single value visualization, which corresponds to a *dashboard query* without group-by or selection, and a pie chart that is grouped by Campaign (**Camp**) type. The dashboard includes drop-down lists for selecting the user name, title, campaign start date, role name, and group-by user title or account state; we experiment with all of these interactions. We further conduct tests of interactions that modify the **Camp** relation (by random cell value perturbations) and remove the **Acc** relation.

Baselines. We consider the following baselines

- **Naive:** Execute the naive SPJA queries translated from dashboard interactions in the DBMS without implementing factorized query execution nor work-sharing from pre-computed data structures.
- **Factorized:** Rewrite the naive queries into message passing for factorized execution, but still doesn't exploit work-sharing.

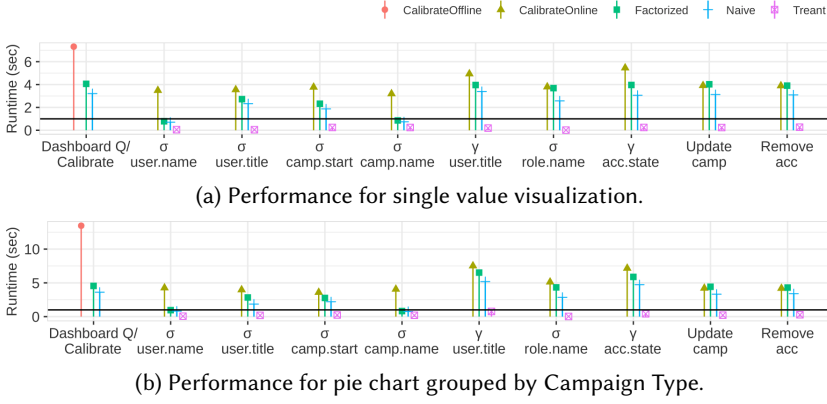


Fig. 13. **Salesforce** Dashboard Performance. Horizontal line shows the 1s interactive response threshold.

- Treant consists of multiple stages for online *interaction query* execution (directly experienced by the users) and the offline/background calibration overheads. We report them separately:
 - **Treant**: online *interaction query* execution using CJs (from prior *interaction query* or *dashboard query*) for acceleration.
 - **CalibrateOffline**: offline calibration overhead to build CJs for the initial *dashboard queries*.
 - **CalibrateOnline**: background calibration overhead to build CJT for the current *interaction query* (to proactively expedite future queries), which doesn't need full completion.

Results. Figure 13 shows the results. The performance of both **Factorized** and **Naive** varies depending on the type of interaction, with faster performance for selection due to smaller data sizes to process. However, they still take $> 2s$ for most *interaction queries* and can be as slow as 7s. We find that factorized execution (**Factorized**) alone results in even slower performance than **Naive**. This is because **Factorized** is optimized for many-to-many joins, which are not present in **Salesforce** workloads. Additionally, **Factorized** introduces extra aggregations for each join edge.

In contrast, **Treant** is able to execute various types of *interaction queries*, from selection, group-by, to relation update and removal, within 100ms by reusing messages, offering two orders of magnitude improvements; the offline overhead (**CalibrationOffline**) is only $\sim 2\times$ the cost of executing the *dashboard query* by **Factorized**. To ensure quick responses in future interactions, Treant calibrates *interaction query* (**CalibrationOnline**) whose time is at the same scale as **Factorized** as it only requires downward message passing (Section 3.4), and is well within user think-times ($< 10s$ [22]). Furthermore, **CalibrationOnline** is in the background during user think-times, and doesn't require full completion. Regarding storage, the intermediate messages occupy 363MB ($< 3\%$ of DB size).

5.1.2 Flight Dashboard. We next experiment with the Flight dataset [53].

Workloads. IDEBench [22] produces a random workload of a collection of visualizations (*dashboard queries*) and, for each visualization, a series of *interaction queries* that progressively incorporate selections. We use the default workload¹⁰, which contains 8 total *interaction queries* across 5 visualizations. We use the same baselines as in Section 5.1.1. However, to demonstrate the advantage of online calibration, we evaluate **Tre+Offline**, which only uses CJs created offline. Based on IDEBench recommendation, we use a think-time of 10s by default, and study the sensitivity later.

Results. The results are displayed in Figure 14. Both **Factorized** and **Naive** take 3–6s for most queries. For Treant, offline calibration is $\sim 3\times$ **Factorized** due to the larger message size during the

¹⁰independent in <https://github.com/IDEBench/IDEBench-public/tree/master/data>

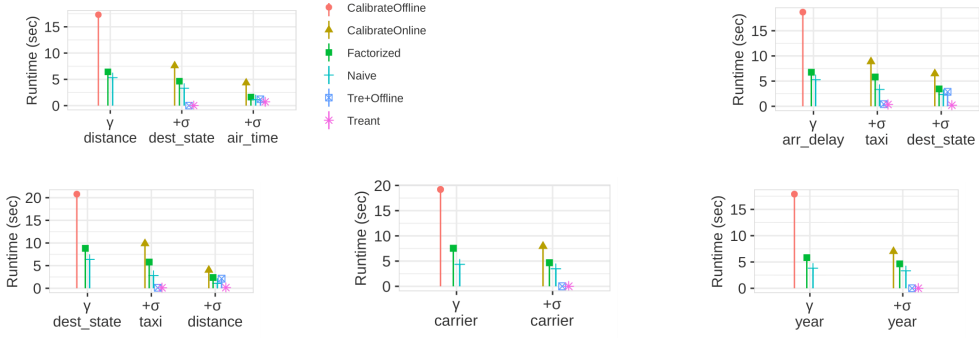


Fig. 14. **Flights** Dashboard Performance. The dashboard features five visualizations. For each visualization, the leftmost is *dashboard query*. *interaction queries* progressively adds (+) selection or group-by element to the previous one.

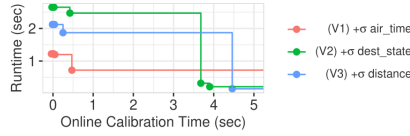


Fig. 15. Runtime for the 2nd *interaction query* over the first three visualizations, with varying online calibration.

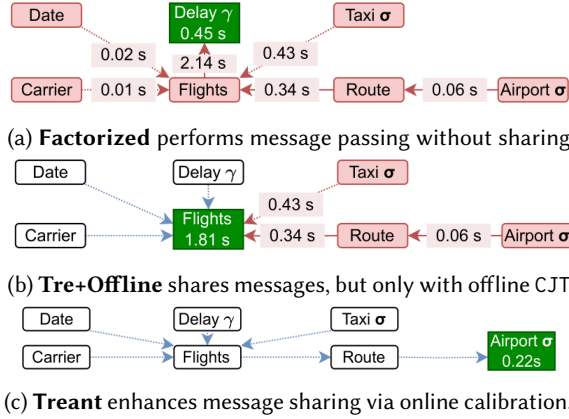


Fig. 16. Case study for the second *interaction query* over the second visualization. Blue dotted lines represent shared messages, while red lines indicate computed messages with respective runtimes noted on the edges. The absorption runtimes are displayed in the green rectangle bag.

group-bys, but reduces **Treant** to <200ms. Online calibration takes 4–9s, well within the think-time of 10s. For the second *interaction queries* of the 2nd and 3rd visualizations, using only offline-created CJT (**Tre+Offline**) takes > 2s due to the larger Steiner tree size, and online calibration reduces this time by >10×. In terms of storage, the intermediate messages take up just 89MB (< 1% of DB size).

Online Calibration Sensitivity. For online calibration, think-time may not always be possible to leverage. For instance, in multi-tenant DBMS, other users may execute their queries concurrently while one user is thinking. To examine this, we study the second *interaction query* in the first three visualizations, whose runtime depends on the online calibration. Figure 15 presents the *interaction query* runtime with varying online calibration time. The plot shows a stepped pattern, as reductions in *interaction query* runtime only occur upon completion of sharable message computations.

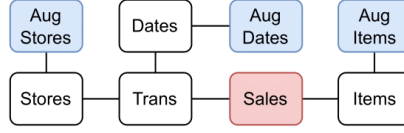


Fig. 17. **Favorita** schema. **Sales** is the largest relation. **Aug Stores**, **Aug Dates** and **Aug Items** are augmentation relations.

Consequently, each step corresponds to a completed sharable message. Without online calibration, the runtime equals that of **Tre+Offline**. As online calibration time increases, the runtime decreases accordingly due to partial calibration, and reaches <1s after only <5s.

Case Study. To understand where the performance improvements stem from, we conduct a case study on the second *interaction query* over the second visualization, because it shows the most significant improvement. Figure 16 shows the detailed runtime for each message and absorption for different baselines. **Factorized** performs message passing without sharing, while **Tre+Offline** shares messages but only with the offline CJT. They are slowed by the costly message passing or absorption over the large fact table (>1.8s). In contrast, **Treant** greatly improves message sharing using online calibration. Once the online calibration is completed, the JT is calibrated. For the interaction that selects Airport attribute, this only requires an absorption over Airport (small dimension table) in 0.22s.

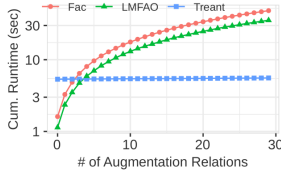
5.1.3 ML augmentation. We next evaluate the benefit of Treant for ML augmentation using the **Favorita** dataset.

Workloads. We train linear regression using (Sales.unit_sales, Stores.type, Items.perishable) as features, and Trans.transactions (number of transactions per store, date) as the target variable Y.

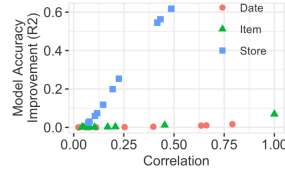
To simulate a data warehouse with augmentation data of varying effectiveness, we generate synthetic data to augment (join) with Dates, Stores, and Items. For each of these three relations, we first generate a predictive feature $\hat{Y}$ as the average of Y grouped by primary key. We then create 10 augmentation relations with schema (k, v), where k is the primary key and v varies in correlation $\hat{Y}$ [38]: The correlation coefficient ϕ is drawn from the inverse exponential distribution $\min(1, 1/\text{Exp}(10))$, and the values are the weighed average between $\hat{Y}$ and a random variable weighed by ϕ . We individually evaluate the model accuracy (R2) for each of the 30 augmentation relations, and measure the cumulative runtimes.

In addition to **Treant**, we also compare the training time of **Fac** that applies factorized ML but trains each model independently without work sharing, and **LMFAO** [65], the SOTA factorized ML system that is algorithmically similar to **Fac** but implemented with a custom engine in C++ and not portable to user DBMSes. To ensure a fair comparison, we exclude the time required to read files from disk and the compilation time for **LMFAO**, but include all the time needed to build data structures and run queries.

Results. Figure 18a reports the cumulative runtime to augment and retrain the model. **Fac** takes >1.3 min, while **Treant** takes ~6s: calibration dominates the cost, and is ~2× the cost of training a single model because of the downward message passing. However, after calibration, **Treant** evaluates all 30 augmentations in <1s. **LMFAO** takes ~1.3× less time than **Fac** for model training due to implementation difference, but even when including the offline calibration cost, **Treant** is ~13× faster than **LMFAO** after 30 augmentations. Figure 18b reports the accuracy improvement above the baseline (0.031) after each augmentation, and we see a wide discrepancy between good and bad augmentations (+0 to +0.61).



(a) Cumulative runtime (log)



(b) Accuracy (R2) improvements.

Fig. 18. Augmentation run time and model performance.

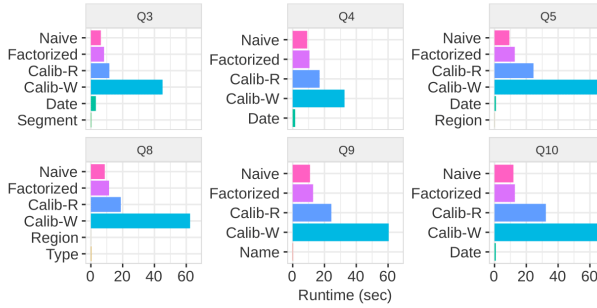


Fig. 19. Run time for TPC-H dashboard. **Naive** executes queries without message passing. **Factorized** executes queries with message passing. **Calib-R** computes messages for calibration without materializing them, and **Calib-W** materializes them. The remaining bars are for *interaction queries* that vary the values of the parameters (labels).

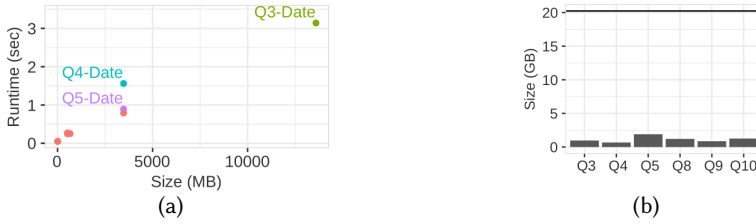


Fig. 20. (a) Size of annotated bag (by predicate) vs query runtime. (b) Total message size overhead of Treant. Horizontal line is the TPC-H database size (~ 20 GB).

5.2 Cloud DBMS Experiments

We now evaluate Treant on the cloud DBMS (AWS Redshift).

Setup. We use TPC-H (SF=50) for dashboard and TPC-DS (SF=50) for empty bag optimizations. We used dc2.large node (2 vCPU, 15GB memory, 0.16TB SSD, 0.60 GB/s I/O). All experiments warm the cache by pre-executing queries until the runtime stabilizes.

5.2.1 Interactive Dashboard. We evaluate Treant on TPC-H queries.

Workloads. We build an interactive dashboard based on a subset of the TPC-H queries (Q3-5,8-10) that can be rewritten as SPJA queries. These TPC-H queries are parameterized, so we construct a *dashboard query* for each using random parameter values and then create *interaction queries* that vary each parameter.

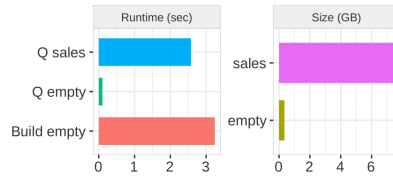


Fig. 21. Runtime to build and query the empty bag and **Store_Sales** table, and their storage sizes.

We compared the runtime of the *interaction query* using different approaches: **Naive** simply executes the query on Redshift; **Factorized** rewrites the query as message passing for factorized query execution; and Treant. For Treant, we reported calibration execution cost (**Calib-R**) separately from the calibration materialization cost (**Calib-W**), since writes on Redshift are particularly expensive.

Results. Figure 19 shows the run time. Calibration (**Calib-W**) takes $4\sim 7\times$ longer than **Naive**. As expected, upward and downward message passing alone is $\sim 2\times$ slower (**Calib-R**), and the rest is dominated by high write overheads; Q8 groups by 2 attributes, so its message sizes are $\sim 2\times$ larger, and $4\times$ slower overall. As in Section 5.1.1, **Factorized** is slower than **Naive** because there is no many-to-many join and it has additional aggregation overheads.

In contrast, Treant accelerates TPC-H queries by nearly $1000\times$ over **Naive** for parameters including Segment, Region and Type. Naturally, the speedup depends linearly on the size of the bag that contains the parameterized attribute (Figure 20a). Q3 Date incurs a higher cost as it includes the fact table in Steiner tree, which could be optimized by creating an empty bag for Date (Section 3.2). We note that the space overhead for calibration is only $< 2\text{GB}$ compared to the original database size $\sim 20\text{GB}$ (Figure 20b). This is because messages are aggregated results from these relations.

5.2.2 Empty Bag Optimization. Empty bags are a novel extension to materialize custom views. We evaluate the costs and benefits of empty bags using TPC-DS. We create an empty bag (**Store,Time**) as illustrated in Figure 5b. Then, we query the maximum count of sales for all stores and times: $Q = Y_{\text{MAX}}(\text{COUNT})(Y_{\text{COUNT}}(\cdot, \text{Store,Time}(\otimes)))$ in two unique ways: (1). Without Empty Bag, Q is executed by first aggregating the count over the absorption result of **Store_Sales**, since **Store_Sales** is the only bag contains both **Store** and **Time** dimensions, then computing the max sales. (2). With Empty Bag, Q is executed directly over the absorption result of the empty bag, which is sufficient to answer aggregation queries over (**Store,Time**).

Figure 21 shows the runtimes and sizes. Empty bag takes $\sim 3\text{s}$ to build, and accelerates Q by $\sim 25\times$. Additionally, the storage space required for the empty bag is $21\times$ smaller than that of **Store_Sales**.

6 RELATED WORK

Interactive Queries. Previous works use indexing and data cubes [28] to support interactive queries. Some studies have improved them to Nanocubes [45] for spatiotemporal data and Hashed-cubes [55] with additional optimizations, or use sophisticated materialization techniques [47, 48]. However, these approaches often have high preprocessing overhead (e.g., taking hours for 200MB data [45]), and require denormalization for large joins. In contrast, CJT has been shown to be exponentially more efficient for *interactive queries* over large joins than data cubes with a constant factor overhead.

Early Marginalization. Early Marginalization was first introduced by Gupta et al. [32] as a generalized projection for simple e.g., count, sum, max queries. It was extended by factorized databases to compactly store relational tables [54] and quickly execute semi-ring aggregation

queries [36, 65]. Abo et al. [5] generalize early marginalization and establish the equivalence between early marginalization and variable elimination in Probabilistic Graphical Models [41]. However, prior works [65] only share work within a query batch but not between batches for interactive queries.

Calibrated Junction Tree. Calibration Junction Tree was first proposed by Shafer and Shenoy [68] to compute inference over probabilistic graphical models. While it has found extensive applications across fields such as engineering [60, 78], ML [13, 20], and medicine [42, 58], its use has been limited to probabilistic tables (for sum of probabilities). The calibration process is reminiscent of Yannakakis's two-pass semi-join reduction [75]. However, Yannakakis's algorithm mainly aims to eliminate redundant tuples in individual relations as a pre-processing step for single query execution. Furthermore, it does not materialize messages and is restricted to the 0/1 semi-ring. In contrast, CJT materializes messages for future reuse across queries. In this work, we broaden the scope of CJT to general semi-ring aggregation for SPJA queries.

Semantic Caching. To accelerate online queries, semantic caching [8, 10, 14, 33, 61] caches previous SPJA query results and reuses them for later SPJA queries, by taking row/column containment, predicate overlaps [31, 70], and constraints into account [30, 33, 61]. The analysis for identifying reuse opportunities has also been applied to identify materialized views to reuse [44]. In contrast, Treant specifically exploits the semiring properties of the aggregation functions to 1) leverage factorized query execution via message passing, 2) identify partial aggregates (messages) to cache during query execution, and 3) identify the best plan that reuses the cached partial aggregates. Like caching, Treant executes and caches online. In addition, Treant proactively populates the cache in anticipation of incremental changes to the most recently executed query, leveraging user think-time. Treant currently only shares "messages" with identical query definitions and leave other types of sharing (e.g., row/column containment, predicate overlaps) as future works.

Multi-query optimization. Multi-query optimization (MQO) [34, 63, 65] shares the state and computation of subexpressions across queries (e.g., sharing scans across queries). However, it centers on batches of queries known a priori. The optimization of interactive dashboards can be considered as an online version of multi-query optimization. Treant approaches this by applying a practical heuristic that interactive queries are incremental, and uses "messages" as the core unit to reuse. These ideas could be integrated into MQO.

7 CONCLUSIONS

We present Treant, a dashboard accelerator over joins. Treant uses factorized query execution for aggregation queries over large joins, and proactively materializes messages, the core intermediates during factorized query execution, that can be shared across *interaction queries*. To effectively manage and reuse messages, we introduced the novel Calibrated Junction Hypertree (CJT) data structure. CJT uses annotations to support SPJA queries, applies calibration to materialize messages in both directions and computes the Steiner tree to assess the reusability of messages. We implement Treant to manage CJT as middleware between DBMSes and dashboards. Our experiments evaluate Treant on a range of datasets on both single node and cloud DBMSes, and we find that Treant accelerates dashboard interactions by two orders of magnitude.

ACKNOWLEDGMENTS

This research received support from the NSF through Grant Numbers 1845638, 2008295, 2106197, 2103794, 2312991, the Columbia Data Science Institute's Avanesian PhD Fellowship, and additional backing from Amazon and Adobe. Thanks to anonymous reviewers for their insights.

REFERENCES

- [1] 2016. Looker. <https://www.looker.com/>.
- [2] 2017. Corporación Favorita Grocery Sales Forecasting. <https://www.kaggle.com/c/favorita-grocery-sales-forecasting>.
- [3] 2018. Modin: Scale your pandas workflows by changing one line of code. <https://github.com/modin-project/modin>.
- [4] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)* 42, 4 (2017), 1–44.
- [5] Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. 2016. FAQ: questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 13–28.
- [6] Mahmoud Abo Khamis, Hung Q Ngo, and Dan Suciu. 2016. Computing join queries with functional dependencies. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 327–342.
- [7] Manar Abourezq and Abdellah Idrissi. 2016. Database-as-a-service for big data: An overview. *International Journal of Advanced Computer Science and Applications* 7, 1 (2016).
- [8] Munir Ahmad, Muhammad Abdul Qadir, Atta Rahman, Rachid Zagrouba, Fahd Alhaidari, Tariq Ali, and Farzana Zahid. 2020. Enhanced query processing over semantic cache for cloud based relational databases. *Journal of Ambient Intelligence and Humanized Computing* (2020), 1–19.
- [9] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively parallel sort-merge joins in main memory multi-core database systems. *arXiv preprint arXiv:1207.0145* (2012).
- [10] Zohreh Asgharzadeh Talebi, Rada Chirkova, and Yahya Fathi. 2009. Exact and inexact methods for solving the problem of view selection for aggregate queries. *International Journal of Business Intelligence and Data Mining* 4, 3-4 (2009), 391–415.
- [11] Albert Atserias, Martin Grohe, and Daniel Marx. 2008. Size bounds and query plans for relational joins. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 739–748.
- [12] Oscar Bashaw. 2022. Drive Revenue by Using Sigma with Salesforce. <https://www.sigmacomputing.com/blog/drive-revenue-by-using-sigma-with-salesforce>.
- [13] Tanya Braum and Ralf Möller. 2016. Lifted junction tree algorithm. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*. Springer, 30–42.
- [14] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. 1995. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering*. IEEE, 190–200.
- [15] Nadiia Chepurko, Ryan Marcus, Emanuel Zraggen, Raul Castro Fernandez, Tim Kraska, and David Karger. 2020. ARDA: automatic relational data augmentation for machine learning. *arXiv preprint arXiv:2003.09758* (2020).
- [16] Diarmaid Conaty, Denis D Mauá, and Cassio P De Campos. 2017. Approximation complexity of maximum a posteriori inference in sum-product networks. *arXiv preprint arXiv:1703.06045* (2017).
- [17] Graham Cormode. 2011. Sketch techniques for approximate query processing. *Foundations and Trends in Databases*. NOW publishers (2011), 15.
- [18] Fabio Gagliardi Cozman et al. 2000. Generalizing variable elimination in Bayesian networks. In *Workshop on probabilistic reasoning in artificial intelligence*. Citeseer, 27–32.
- [19] Ryan Curtin, Benjamin Moseley, Hung Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Rkmeans: Fast clustering for relational data. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 2742–2752.
- [20] Jia Deng, Nan Ding, Yangqing Jia, Andrea Frome, Kevin Murphy, Samy Bengio, Yuan Li, Hartmut Neven, and Hartwig Adam. 2014. Large-scale object classification using label relation graphs. In *European conference on computer vision*. Springer, 48–64.
- [21] Donko Donjerkovic and Raghu Ramakrishnan. 1999. *Probabilistic optimization of top N queries*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.
- [22] Philipp Eichmann, Emanuel Zraggen, Carsten Binnig, and Tim Kraska. 2020. Idebench: A benchmark for interactive data exploration. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1555–1569.
- [23] Raul Castro Fernandez, Ziawasch Abedjan, Famiem Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A data discovery system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1001–1012.
- [24] Raul Castro Fernandez, Pranav Subramaniam, and Michael J Franklin. 2020. Data market platforms: Trading data assets to solve data problems. *arXiv preprint arXiv:2002.01047* (2020).
- [25] Alberto Ferrari and Marco Russo. 2016. *Introducing Microsoft Power BI*. Microsoft Press.
- [26] Wolfgang Fischl, Georg Gottlob, and Reinhard Pichler. 2018. General and fractional hypertree decompositions: Hard and easy cases. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 17–32.

- [27] James Gale, Max Seiden, Gretchen Atwood, Jason Frantz, Rob Woollen, and Çağatay Demiralp. 2020. Sigma Worksheet: Interactive Construction of OLAP Queries. *arXiv preprint arXiv:2012.00697* (2020).
- [28] 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 mining and knowledge discovery* 1, 1 (1997), 29–53.
- [29] Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 31–40.
- [30] Jarek Gryz. 1998. Query folding with inclusion dependencies. In *Proceedings 14th International Conference on Data Engineering*. IEEE, 126–133.
- [31] Sha Guo, Wei Sun, and Mark Allen Weiss. 1996. On satisfiability, equivalence, and implication problems involving conjunctive queries in database systems. *IEEE Transactions on Knowledge and Data Engineering* 8, 4 (1996), 604–616.
- [32] Ashish Gupta, Venky Harinarayan, and Dallan Quass. 1995. Aggregate-query processing in data warehousing environments. (1995).
- [33] Ashish Gupta, Inderpal S Mumick, and Kenneth A Ross. 1995. Adapting materialized views after redefinitions. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. 211–222.
- [34] Mingsheng Hong, Mirek Riedewald, Christoph Koch, Johannes Gehrke, and Alan Demers. 2009. Rule-based multi-query optimization. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. 120–131.
- [35] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1259–1274.
- [36] Manas Joglekar, Rohan Puttagunta, and Christopher Ré. 2015. Aggregations over generalized hypertree decompositions. *arXiv preprint arXiv:1508.07532* (2015).
- [37] Manas R Joglekar, Rohan Puttagunta, and Christopher Ré. 2016. Ajar: Aggregations and joins over annotated relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 91–106.
- [38] Henry F Kaiser and Kern Dickman. 1962. Sample and population score matrices and sample correlation matrices from an arbitrary population correlation matrix. *Psychometrika* 27, 2 (1962), 179–182.
- [39] Mahmoud Abo Khamis, Ryan R Curtin, Benjamin Moseley, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Functional Aggregate Queries with Additive Inequalities. *ACM Transactions on Database Systems (TODS)* 45, 4 (2020), 1–41.
- [40] Mahmoud Abo Khamis, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. AC/DC: in-database learning thunderstruck. In *Proceedings of the second workshop on data management for end-to-end machine learning*. 1–10.
- [41] Daphne Koller and Nir Friedman. 2009. *Probabilistic graphical models: principles and techniques*. MIT press.
- [42] Steffen L Lauritzen and Nuala A Sheehan. 2003. Graphical models for genetic analyses. *Statist. Sci.* (2003), 489–514.
- [43] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal* 27, 5 (2018), 643–668.
- [44] Jingni Li, Zohreh Asgharzadeh Talebi, Rada Chirkova, and Yahya Fathi. 2005. A formal model for the problem of view selection for aggregate queries. In *East European Conference on Advances in Databases and Information Systems*. Springer, 125–138.
- [45] Lauro Lins, James T Klosowski, and Carlos Scheidegger. 2013. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Transactions on Visualization and Computer Graphics* 19, 12 (2013), 2456–2465.
- [46] Zhicheng Liu and Jeffrey Heer. 2014. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2122–2131.
- [47] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. 2013. imMens: Real-time visual querying of big data. In *Computer Graphics Forum*, Vol. 32. Wiley Online Library, 421–430.
- [48] Dominik Moritz, Bill Howe, and Jeffrey Heer. 2019. Falcon: Balancing interactive latency and resolution sensitivity for scalable linked visualizations. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–11.
- [49] Emily G Morton-Owens and Karen L Hanson. 2012. Trends at a glance: A management dashboard of library statistics. *Information Technology and Libraries* 31, 3 (2012), 36–51.
- [50] Thomas Neumann and Bernhard Radke. 2018. Adaptive optimization of very large join queries. In *Proceedings of the 2018 International Conference on Management of Data*. 677–692.
- [51] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case optimal join algorithms. *Journal of the ACM (JACM)* 65, 3 (2018), 1–40.
- [52] Milos Nikolic and Dan Olteanu. 2018. Incremental view maintenance with triple lock factorization benefits. In *Proceedings of the 2018 International Conference on Management of Data*. 365–380.

- [53] B. of Transportation Statisticsw. 2017. Bureau of transportation statistics. <http://www.transtats.bts.gov>.
- [54] Dan Olteanu and Jakub Závodný. 2015. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)* 40, 1 (2015), 1–44.
- [55] Cicero AL Pahins, Sean A Stephens, Carlos Scheidegger, and Joao LD Comba. 2016. Hashedcubes: Simple, low memory, real-time visual exploration of big data. *IEEE transactions on visualization and computer graphics* 23, 1 (2016), 671–680.
- [56] The pandas development team. 2020. *pandas-dev/pandas: Pandas*. <https://doi.org/10.5281/zenodo.3509134>
- [57] Judea Pearl. 1982. *Reverend Bayes on inference engines: A distributed hierarchical approach*. Cognitive Systems Laboratory, School of Engineering and Applied Science
- [58] Arturo Lopez Pineda and Vanathi Gopalakrishnan. 2015. Novel application of junction trees to the interpretation of epigenetic differences among lung cancer subtypes. *AMIA Summits on Translational Science Proceedings* (2015), 31.
- [59] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.
- [60] Julio C Ramirez, Guillermina Munoz, and Ludivina Gutierrez. 2009. Fault diagnosis in an industrial process using Bayesian Networks: Application of the junction tree algorithm. In *2009 Electronics, Robotics and Automotive Mechanics Conference (CERMA)*. IEEE, 301–306.
- [61] Qun Ren, Margaret H Dunham, and Vijay Kumar. 2003. Semantic caching and query processing. *IEEE transactions on knowledge and data engineering* 15, 1 (2003), 192–210.
- [62] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph databases: new opportunities for connected data*. " O'Reilly Media, Inc".
- [63] Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. 2000. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 249–260.
- [64] Maximilian Schleich. 2021. Structure-aware machine learning over multi-relational databases. In *Proceedings of the 2021 International Conference on Management of Data*. 6–7.
- [65] 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*. 1642–1659.
- [66] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*. 3–18.
- [67] P Griffiths 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*. 23–34.
- [68] Glenn R Shafer and Prakash P Shenoy. 1990. Probability propagation. *Annals of mathematics and Artificial Intelligence* 2, 1 (1990), 327–351.
- [69] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. 1997. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal* 6 (1997), 191–208.
- [70] Dimitri Theodoratos and Wugang Xu. 2004. Constructing search spaces for materialized view selection. In *Proceedings of the 7th ACM international workshop on Data warehousing and OLAP*. 112–121.
- [71] Todd L Veldhuizen. 2014. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*.
- [72] Martin J Wainwright, Tommi S Jaakkola, and Alan S Willsky. 2005. MAP estimation via agreement on trees: message-passing and linear programming. *IEEE transactions on information theory* 51, 11 (2005), 3697–3717.
- [73] Dan Wu and Libing Wu. 2007. Hierarchical Junction Trees as the Secondary Structure for Inference in Bayesian Networks. In *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*, Vol. 3. IEEE, 706–712.
- [74] Konstantinos Xirogiannopoulos and Amol Deshpande. 2019. Memory-efficient group-by aggregates over multi-way joins. *arXiv preprint arXiv:1906.05745* (2019).
- [75] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *VLDB*, Vol. 81. 82–94.
- [76] Clement Tak Yu and Meral Z Ozsoyoglu. 1979. An algorithm for tree-query membership of a distributed query. In *COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979*. IEEE, 306–312.
- [77] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. 2019. Automated verification of query equivalence using satisfiability modulo theories. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1276–1288.
- [78] Feng Zhu, HM Abdul Aziz, Xinwu Qian, and Satish V Ukkusuri. 2015. A junction-tree based learning algorithm to optimize network wide traffic control: A coordinated multi-agent framework. *Transportation Research Part C: Emerging Technologies* 58 (2015), 487–501.

Received April 2023; revised July 2023; accepted August 2023