Lightweight Materialization for Fast Dashboards Over Joins

ABSTRACT

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.

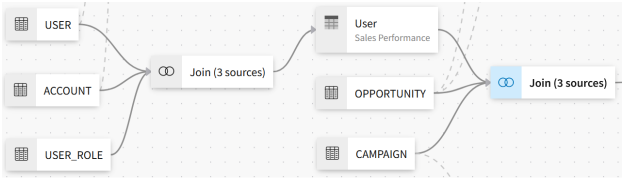
In this paper, we present `Treant`, a dashboard accelerator for queries over large joins. `Treant` uses factorized query execution to handle aggregation queries over large joins, which alone is still insufficient for interactive speeds. To address this, we exploit the incremental nature of user interactions 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 `CJT`s 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.

ACM Reference Format:
. 2023. Lightweight Materialization for Fast Dashboards Over Joins. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

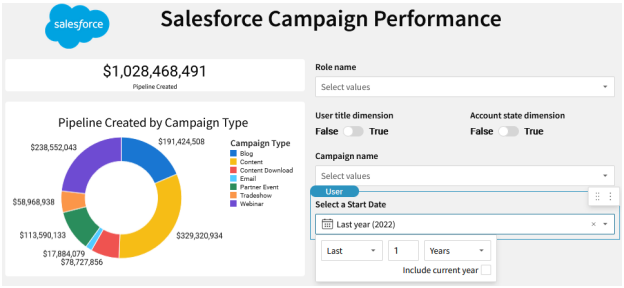
1 INTRODUCTION

Dashboards are at the heart of modern BI tools (e.g., PowerBI [23], Looker [1], Sigma Computing [26]) 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. 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.*



(a) Build join graph offline by data engineering. Three tables related to `User` are first joined. Then, the enriched `User` table is joined with `Opportunity` and `Campaign`.



(b) Dashboard interface for online exploration.

Figure 1: Screenshots for Sigma Computing dashboards.

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* $Q_1 = \gamma_{SUM(amount)} R_{\bowtie}$ *while the pie chart displaying "Pipeline Created by Campaign Type" is computed by* $Q_2 = \gamma_{SUM(amount),Campaign\_Type} R_{\bowtie}$. *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 [44], yet joins are notoriously expensive to execute [9, 40, 49]. Traditional techniques, such as data cubes [27] and indexing [47], are designed for a single denormalized table, but poorly handle many joins because the denormalized table size can be exponential: $O(n \times f^r)$, where f is the fanout along join graph edge with r relations, each of size n.

Recent factorized query execution techniques [5, 53] 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

Conference'17, July 2017, Washington, DC, USA
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [61] 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`[1], 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 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 a *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. [6] that factorized query execution can be modeled as message passing in Probabilistic Graphical Models (PGM) [37], as described in the following example:

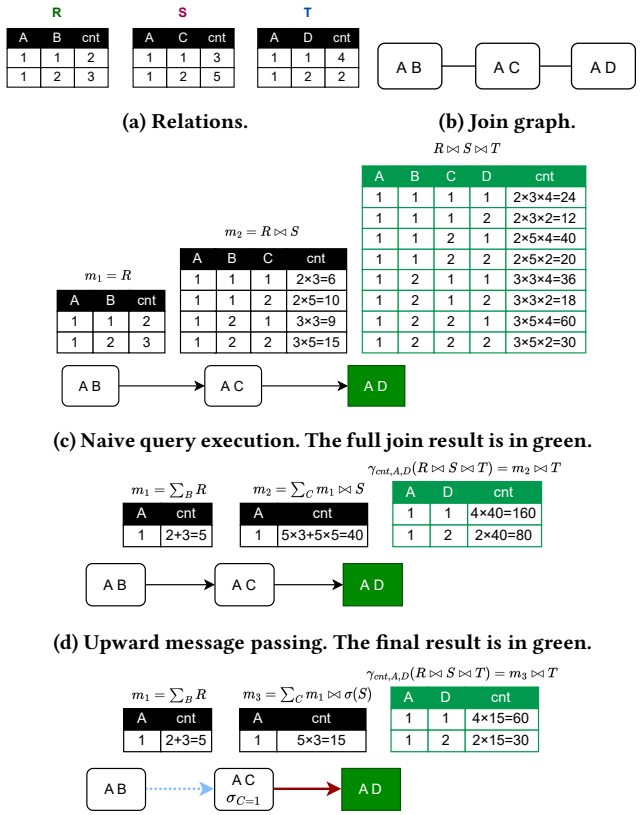
Example 2. *Figure 2(a,b) list example relations (duplicates are tracked with a* `cnt` *"annotation") and the join graph, respectively. Consider a dashboard query that computes the total count over the full join result:* $Q = \gamma_{cnt}(R \bowtie S \bowtie T)$. *Figure 2c naively executes the query, 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→S→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→S→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→S→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←S←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 [64], which is used to similarly share computation between queries on posterior distributions. We are the first to apply it to SPJA queries.

`CJT`s 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 [20]. 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.



(a) Relations. **(b) Join graph.**

(c) Naive query execution. The full join result is in green.

(d) Upward message passing. The final result is in green.

(e) m_1 (dotted blue) could be re-used for *interaction query* with additional selection $\sigma_{C=1}$ and only m_3 (red) need to be computed.

Figure 2: Example database with three relations, its join graph (also JT), naive query execution for the total count, and factorized query execution by upward message passing.

[1] `Treant` efficiently manages tree-like data structures, akin to the tree-like characters from "The Lord of the Rings".

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, 55]. Furthermore, Treant accelerates advanced dashboard features like interactive *Data Augmentation* [14] 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×, and speeds up ML augmentation by 10×.

Note: The paper is self-contained. Any references to the Appendix can be ignored or found in the technical report [4].

2 BACKGROUND

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

Data Model. Let uppercase symbol A be an attribute, dom(A) is its domain, and lowercase symbol a ∈ dom(A) be a valid attribute value. By default, we assume categorical attributes. Numerical attributes are usually part of the semi-ring annotation discussed below. However, we can easily support numerical attributes by introducing a domain with infinite size. Given relation R, its schema S_R is a set of attributes, and its domain $dom(R) = \times_{A \in S} dom(A)$ 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, 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 $(D, +, \times, 0, 1)$ defines a set D, binary operators + and × closed over D where both are commutative, and the zero 0 and unit 1 elements. 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. However, our work extends to arbitrary commutative semi-ring structures that support aggregation queries containing common statistical functions (mean, min, max, std), as well as machine learning models (e.g., linear regression, regression trees, etc). Each relation R annotates each of its tuples t ∈ dom(R) with a natural number, and R(t) refers to this annotation for tuple t [28, 32, 51]. 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 values[2].

Consider an example query $q=\gamma_{\mathbf{A},COUNT}(R_1 \bowtie R_2 ... \bowtie R_n)$ that joins n relations, groups by a set of attributes **A**, and computes the *COUNT*. The operators that combine multiple tuples are joins and groupbys (projection under set semantics corresponds to groupby), and they compute the output tuple annotations as follows:

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

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

The first statement states that given a join output tuple t, its annotation is defined by multiplying the annotations of the contribution pair of input tuples. The second 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 annotations for all input tuples that are in the same group as t.

To summarize, join and groupby correspond to × and +, respectively. Let the schema of $R_1 \bowtie R_2 ... \bowtie R_n$ be **S**. q can be rewritten as $\sum_{A \in \mathbf{S}-\mathbf{A}}(R_1 \bowtie R_2 ... \bowtie R_n)$. This lets us distribute summations across multiplications as in simple algebra, as we discuss next.

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 [29].

Consider Figure 2, which computes $\gamma_{A;COUNT}(R \bowtie S \bowtie T)$. 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 cardinality of relations, we can push down marginalizations to derive the following, where the largest intermediate result, and thus the join cost, is O(n):

$$\sum_D (\sum_C ((\sum_B R[A,B]) \bowtie S[A,C]) \bowtie T[A,D])$$

Join Ordering and Variable Elimination. Variable elimination is a class of query execution plans that combines early marginalization with join ordering. 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, and then marginalize A, is called variable elimination [15] and is widely used for inference in PGM [37]. The order in which attribute(s) are marginalized out (by clustering and joining the incident relations) is called the

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

variable elimination order. Note that a given order is simply an execution plan. The complexity of variable elimination is dominated by the intermediate join result size of the clustered relations (using worst-case optimal join [50]). It is well known that finding the optimal order (with the minimum intermediate size) is NP-hard [24]. However, common database queries are over acyclic join graph, whose optimal order could be found efficiently as discussed below.

Junction Hypertree. The Junction Hypertree[3] (JT) is a representation of a join query that is amenable to complexity analysis [6, 33] and semi-ring aggregation query optimization [5]. In the next section, we will show how to materialize and maintain query intermediates based on JT to provide work-sharing and optimization opportunities for join-aggregation queries.

Given a join graph $R_1 \bowtie \ldots \bowtie R_n$ using natural joins for simplicity, a Junction Hypertree 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 [6], a JT vertex is also called a *bag*. A JT must satisfy three properties:

- **Vertex Coverage:** The union of all bags in the tree must be equal to the set of attributes in the join graph.
- **Edge Coverage:** For every relation R in the join graph, there exists at least one bag that is a superset of R's attributes.
- **Running intersection:** For any attribute in the join graph, the bags containing the attribute must form a connected subtree. In other words, if two bags both contain attribute A, all bags along the path between them should also contain A.

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 (execution plans) 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 [24], 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 the join graph) and the size of each bag is bounded by its corresponding relation size. We refer readers to FAQ [6] for a complete description.

Message Passing for Query Execution. Message Passing was first introduced by Judea Pearl in 1982 [56] (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. [6] established the equivalence between factorized query execution, and (upward) message passing. The full algorithm can be found in Appendix A; below, we illustrate how message passing over JT is used for query execution, and the next section leverages the message reusability across queries.

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 3 (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→S→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 compute the final result.*

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.

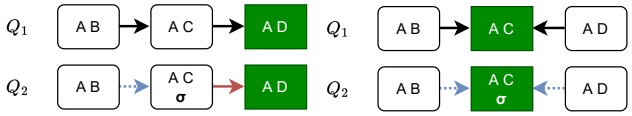
Our novelty is (1) to use JTs as a concrete data structure to support message reuse, and (2) to borrow *calibration* [64] from PGM to materialize messages for any message passing order. Although CJT is widely used across engineering [59, 73], ML [13, 18], and medicine [39, 57], we are the first to introduce CJT in the context of query execution and generalize it to semi-ring SPJA queries.

3.1 Motivating Example

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

Example 4. *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 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.*

[3] JT is also called Hypertree Decomposition [6, 33], Join Tree, Join Forest [31, 61] in databases and Clique Hypertree in probalistic graphical models [37].



(a) Message passing to root AD. (b) Moving root increases reuse.

Figure 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.

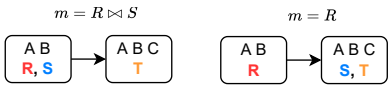


Figure 4: The same JT over relations R(A,B), S(A), T(B,C) can have different relation mappings ($\mathcal{X}$) and each mapping results in different messages (m). For each bag, its attributes are at the top and mapped relations are at the bottom.

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 addresses these limitations.

3.2 Junction Hypertree as Data Structure

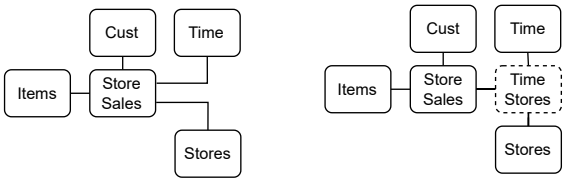
A naive approach to re-use messages is to execute an aggregation query over JT, and cache the messages; when a future query traverses an edge in the JT, it reuses the corresponding message. Unfortunately, this is 1) inaccurate, because messages generated along an edge are not symmetric and depend on 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 $\mathcal{Y}$(i → j) to refer to the cached message for the directed edge i → j.

Relation Mapping. $\mathcal{X}$(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 relation maps to a single bag[4]. Relations mapped to the same bag are joined during message passing.

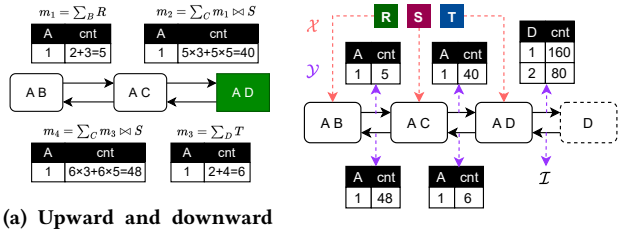
Empty Bags. To avoid large paths during message passing, it's beneficial to add custom *empty* bags to create "short cuts". *Empty* bags are not mapped from any relations and are simply a mechanism to materialize custom views for work sharing. They join incoming messages, marginalize using standard rules, and materialize the outgoing messages. Empty bags are a novel addition in this work: previous works [5, 6, 69] 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. However, empty bags aid in work sharing for multiple queries.

[4]Heuristics for the general case have been studied by the greedy variable elimination in Probabilistic Graphical Model [37].



(a) TPC-DS join graph (also JT) (b) Add empty bag (Time, Stores).

Figure 5: Simplified Join graph (JT) of TPC-DS. Adding an empty bag can accelerate queries group-by `Time` and `Stores`.



(a) Upward and downward message passing. **(b) Calibrated Junction Hypertree.**

Figure 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.

Example 5 (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× *smaller (154K rows) than the fact table.*

Note that leaf empty bag may result in an empty output message; we avoid this special case by mapping the identity relation $\mathcal{I}$[5] to it, such that R ⋈ $\mathcal{I}$ = 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 6 (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 the database **R** = {$R_1, R_2, ..., R_n$} and JT = ((E, V), $\mathcal{X}$, $\mathcal{Y}$), we focus on SPJA queries of the following form, where any semi-ring

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

Annotation	Effect	Applicability	Section
γ_A	Prevent A from being marginalized out for all downstream messages.	Any bag containing A.	Section 3.3
$\sum_A$	Marginalize out A. "Cancels" γ_A for downstream messages.	Any bag containing A.	Section 3.4.2
$\overline{R}$	Exclude relation R from the bag during message passing.	The bag $\mathcal{X}(R)$.	Section 3.3
$R^*_{ver.}$	Update relation R in the bag to the specified version during message passing.	The bag $\mathcal{X}(R)$.	Section 3.3
σ_{id}	Apply selection uniquely identified by id to relations during message passing.	Any bag σ_{id} is applied to.	Section 3.3

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

aggregation is acceptable:

```
SELECT 𝒢, COUNT(*) FROM 𝒥
WHERE [JOIN COND] AND 𝒫 GROUP BY 𝒢
```

where $\mathcal{G}$ is the grouping attributes, $\mathcal{J} \subseteq \mathcal{R}$ is the set of relations joined in the FROM clause, and $\mathcal{P}$ is the set of single-attribute predicates[6]. Query execution is based on message passing as in Section 2. However, the processing of each bag differs based on its annotations. We propose 4 annotation types, summarized in Table 1:

- 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, 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 $\overline{R}$ (resp. $R^*_{ver.}$). When computing messages from this bag, R will be excluded from $\mathcal{X}^{-1}$(u)[7] (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. If A is not in any relation in $\mathcal{J}$, we can skip it. Otherwise, we choose a bag u that contains A, 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 the next section.

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", as it materializes messages along edges that point towards the root.

Message Generation $\mathcal{Y}$(b→p)**.** The message $\mathcal{Y}$(b→p) from bag b to parent p is defined as follows. Let M(b) = $\{\mathcal{Y}(i{\to}b) | i{\to}b \in E \wedge i \neq p\}$ 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 $\overline{R}$ from the join, apply predicates σ (with appropriate push-down), and exclude attributes in γ.

$$\mathcal{Y}(b \to p) = \sum_{b-(p\cap b)-\gamma} \sigma\left(\bowtie (M(b) \cup \mathcal{X}^{-1}(b) - \overline{R})\right)$$

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 parent 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: Absorption(r) = $\sigma\left(\bowtie (M(r) \cup \mathcal{X}^{-1}(r) - \overline{R})\right)$ where M(r) = $\{\mathcal{Y}(i{\to}r) | i{\to}r \in E\}$. To generate the final query results, we marginalize away all attributes not in the grouping conditions $\mathcal{G}$.

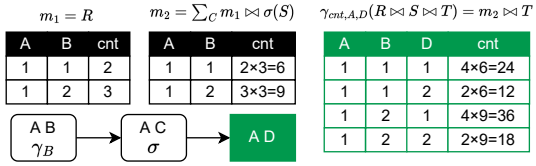


Figure 7: Filter-group-by query with annotated JT.

Example 7. *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.*

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 query complexity (Appendix B). Since relation removal and update annotations $\overline{R}, R^*_{ver.}$ can be only placed on $\mathcal{X}$(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.

3.3.2 Message Reuse Across Queries. Messages reuse between queries requires that the message along edge u → 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 1 (Message Reusability). *Given a* JT *and annotations for two queries, consider the directed edge* u → v *present in both queries. Let* T_u *be the subtree rooted at* u. *If the annotations for*

[6]Multi-attributes predicates have interesting optimization opportunities [35] but is not our focus. We rewrite them into group-by those attributes followed by the predicate.
[7]Rigorously, $\mathcal{X}$ doesn't have an inverse function. We define $\mathcal{X}^{-1}$ to be a mapping from one bag to a set of base relations such that $\mathcal{X}^{-1}$(u) = {i|$\mathcal{X}$(i) = u}.

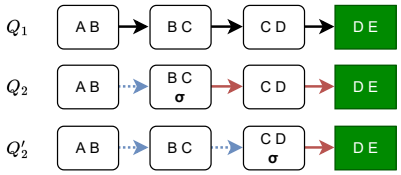


Figure 8: Message size vs reuse trade-off. Given total count query Q_1, Q_2 adds a predicate to C. Pushing down selection as in Q_2 may reduce message size but hinders reuse as compared to Q_2'. Dotted blue edges are reusable messages and solid red edges are non-reusable edges.

T_u *are the same for both queries, then the message along* $u \to v$ *will be identical irrespective of the traversal order nor choice of the root.*

This proposition is well established in PGM [64], 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 ($\mathcal{X}$), annotations and incoming messages. None of these depend on the traversal order nor the root.

Proposition 1 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. However, pushing selections down trades-offs potentially smaller message sizes for limited reusability:

Example 8. *Suppose we have materialized messages for* Q_1 *in Figure 8, and want to execute* Q_2*, which has an additional predicate over C. If we annotated bag* BC *with* σ*, this may reduce the message size but we cannot reuse the message in* BC → CD*. If we annotate* CD *(Q_2'), we can reuse the message but risk larger message sizes.*

In practice, we prioritize reuse by pulling annotations close to the root—reuse helps avoid scan, join, and aggregation costs, whereas larger message sizes simply increase scan sizes.

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 $u \to v$, 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 materialized messages. Our algorithm extends Shafer–Shenoy inference algorithm [64] in PGM to semiring aggregation, and is fully described in the technical report [4]. If the semi-ring is also a semi-field, the Hugin algorithm [41] can further avoid redundant joins. This benefits common aggregations such as sum, stddev, and even linear regression models [62].

Example 9. *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 marginalized absorptions.*

Calibration means all bags are ready for absorption. This immediately accelerates the class of queries that furthers adds one grouping or filtering over attribute A[8]. 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 **steiner tree** T is the minimal subtree in the CJT that connects all bags in $\mathbf{B_D}$. From Proposition 1, edges that cross into T have the same messages as in the CJT and can be reused. Thus, we only need to perform upward message passing inside of T. Let us first start with an illustrative example:

Example 10 (Steiner Tree). *In Figure 9, 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 → BC*, but otherwise re-run the upward message passing along the steiner tree.*

Although the example allows us to reuse one message, it's suboptimal because the steiner tree isn't minimal, and the root is poorly chosen. Instead, we use a greedy procedure to find the minimal steiner tree: we arbitrarily place the annotations on valid bags to create an initial steiner tree, and then greedily shrink it. Given the minimal steiner tree, 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 $\overline{R}$, 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 11. *The third row in Figure 9 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 *maginalizes out* D, E.

[8]These queries correspond to marginal posterior probability (group-by) and incremental update (filter) in PGM [37].

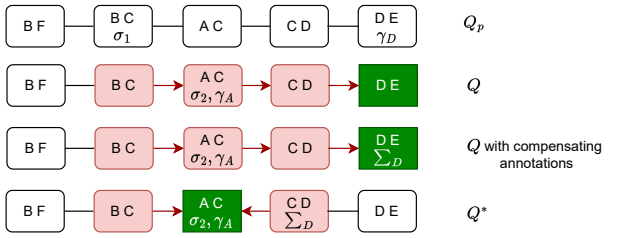


Figure 9: Given CJT with *dashboard query* Q_p, the Steiner Tree to execute Q is highlighted (green is Steiner Tree root and red is non-root nodes). Compensating annotation $\sum_D$ is introduced to compensate γ_D, and can be moved for better query plan. Q^* is the optimal query with minimum steiner tree size and runtime complexity.

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 ∑ 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 12. Q^* *in Figure 9 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 ⟶ CD.

After the optimization procedure above, the execution plan over the CJT will always be as efficient (or in many cases much more efficient than) executing the query without the CJT. Our proof sketch analyzes two scenarios. If the optimal root without the CJT is within the steiner tree, we can reuse messages outside the steiner tree. If the optimal root is outside the steiner tree, we can still move the root to the closest bag within the steiner tree. In both cases, all messages within the steiner tree are the same. Calibration is the key mechanism that allows us to freely move the root.

Applying CJT to Dashboard and Challenge. CJT is highly suitable for the interactive dashboard: We build CJT for *dashboard query* offline. Given an *interaction query* online, CJT ensures that computation needed is proportional to *the difference between it and* dashboard query (steiner tree), rather than its overall complexity; 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, diminishing the benefits. 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" [20] 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.

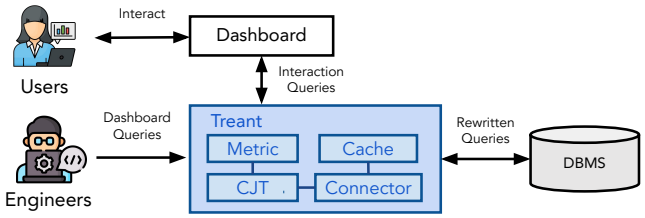


Figure 10: Treant architecture.

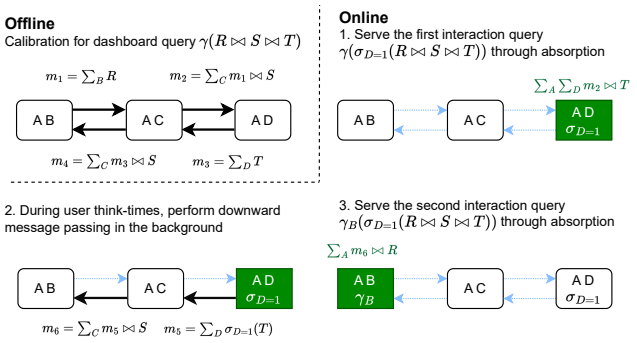


Figure 11: Example CJT management by Treant.

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.

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 (e.g., as sum, average, min, and even ML [61, 62]).

Construct Join Graph. In enterprise data warehouses, there are typically a large number of tables containing the desired metrics and interesting dimensions for enrichment (join). The engineering team constructs a join graph that specifies these tables and the join conditions between them, but doesn't materialize join.

Build Visualization. The engineering team specifies the visualizations to be included 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 that can encode data from *dashboard query*.

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 future 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 clauses of the *dashboard query* (Section 3.3)
- update or remove tables in the join clause (Section 3.3)
- join with new tables (Section 4.3)

We note that Treant doesn't require the dashboard widgets to be pre-defined offline, but rather allows for on-demand interactions based on the user's preferences, providing greater flexibility.

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" [20] 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 13. *We illustrate the* CJT *management using the example join graph (Figure 2b) in Figure 11. 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 10. In contrast to previous factorized systems [36, 61] 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* for message passing, 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, which are SPJA query results, are not returned to Treant but are instead stored as tables in the DBMS.

During the offline stage, Treant establishes a connection to DBMS through the *connector* component. For each dashboard 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: Treant issues rewritten queries to DBMS, stores messages as tables, and uses CJT to keep track of the pointers (table names) to these messages.

During the online stage, Treant takes an *interaction query* as input, finds the corresponding CJT of previous *interaction query* (Section 4.2.1) from based on the user session and the visualization, if available, or uses the CJT of the *dashboard query* otherwise. Then

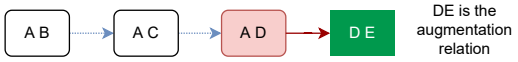
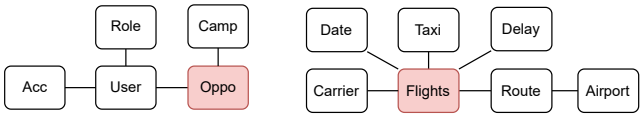


Figure 12: Augmenting the join graph with DE. The steiner tree is AD → DE with root DE and requires one message (red).



(a) Salesforce **(b) Flight**

Figure 13: Database schema. Red relation is the largest.

Treant uses that CJT to pass messages only within the steiner tree (Section 3.4.2), performing absorption as a query to DBMS, retrieves the results from DBMS and returns 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, Treant stores all the messages in its *cache* to further share work among user sessions. 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 1), with the message pointer as value. 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. When a message is removed from the cache, Treant issues a deletion query for the message to the DBMS.

4.3 Augmentation Optimization

Simple ML models like linear regression are widely used to examine attribute relationships in the dashboard [48]. Data and feature augmentation [14] 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 [14, 21, 22]. However, the major bottleneck is the cost of joining each augmentation dataset and then retraining the ML model.

The SOTA factorized ML [16, 51, 61, 62] avoids join materialization when training models over join graphs. First, it designs semi-ring structures for common models (linear regression [62], factorization machines [60], k-means [16]), 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 → 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 12 shows a join graph AB → AC → 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.

Although the above is likely the common case, the augmentation relation may have join keys that span multiple bags in the CJT. In these cases, the steiner tree spans the bags containing the join keys as well as the new relation. We discuss details in Appendix C.

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 [58]) and a cloud DBMS (Redshift).

5.1 Single-node DBMS Experiments

We first evaluate Treant on DuckDB [58].

Setup. We use two datasets for dashboards: **Salesforce** [12] is a public dataset by Sigma Computing for CRM and marketing analysis, and **Flight** [52] is a real-world dataset used by interactive data exploration [20]. However, **Salesforce** is a demo dataset of only 27MB, while real-world enterprise datasets are typically at the GB level [7]. To address this discrepancy, we employ the data scaler in IDEBench [7] to scale both datasets to 10M rows. The scaling process involves denormalization, estimating the distribution, sampling rows from the distribution, and finally normalizing the table through vertical partitioning. The final normalized schemas are in Figure 13, with **Salesforce** sized at 2.7GB and **Flight** at 0.5GB.

For ML augmentation, we use the **Favorita** [2] dataset of purchasing and sales forecasts, widely used in prior factorized ML [61, 62] (see Figure 16 for the schema). **Sales** is the largest relation (241MB), while the others are < 2MB.

All experiments were run on a GCP n1-standard-1 VM with 3.75GB RAM. All experiments fit and ran in memory.

5.1.1 *Salesforce Dashboard.*

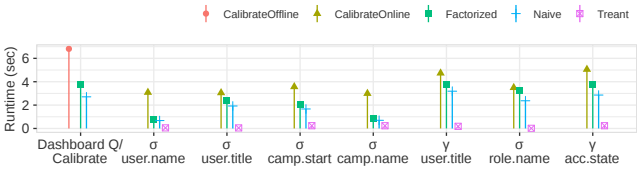
We conduct experiments on the Salesforce dashboard [12] built by Sigma Computing.

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.

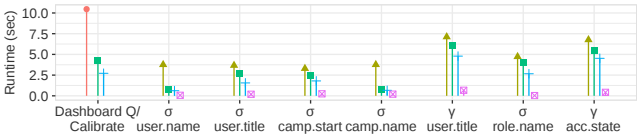
We consider two types of visualizations from the Salesforce dashboard illustrated in Figure 1b: a single value visualization, which corresponds to a *dashboard query* without group-by or selection, and a pie chart that is grouped by Campaign (**Camp**) type. The dashboard includes drop-down lists for 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.

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.
- **CalibrateOffline**: Treant builds and calibrates CJTs for *dashboard queries* during offline phase.
- **Treant**: Treant uses CJTs (from previous *interaction query* or *dashboard query*) to execute and accelerate *interaction query*.
- **CalibrateOnline**: Treant calibrates the CJT for the current *interaction query* in the background, to proactively speed up future queries. Note that this process doesn't need to be fully completed.



(a) Performance for single value visualization.



(b) Performance for pie chart grouped by Campaign Type.

Figure 14: Salesforce Dashboard Performance.

Results. Figure 14 presents 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 additional aggregations for each join edge.

In contrast, **Treant** is able to execute *interaction queries* within 100ms by reusing messages, offering two orders of magnitude improvements; the offline overhead (**CalibrationOffline**) is only ~2× 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 [20]). Furthermore, **CalibrationOnline** is in the background during user think-times, and doesn't require full completion.

5.1.2 *Flight Dashboard.*

We next experiment with the Flight dataset [52].

Workloads. IDEBench [20] produces a random workload consisting of a collection of visualizations (*dashboard queries*) and, for each visualization, a series of *interaction queries* that progressively incorporate selections. We use the default workload[9], 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 CJTs created offline. IDEBench recommends a think-time of 3–10s. Our plots use a think-time of 10s; we also test when the think time is 3s, and Treant can still attain interactive speeds by leveraging partially calibrated CJT. Only for the 2nd interaction ($+\sigma_{\text{distance}}$) of the 3rd visualization is the time >0.5s because the required message computations happen to be unfinished during the think-time.

Results. The results are displayed in Figure 15. Both **Factorized** and **Naive** take 3–6s for most queries. For Treant, offline calibration is ~3× **Factorized** due to the larger message size during the 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

[9] independent in https://github.com/IDEBench/IDEBench-public/data

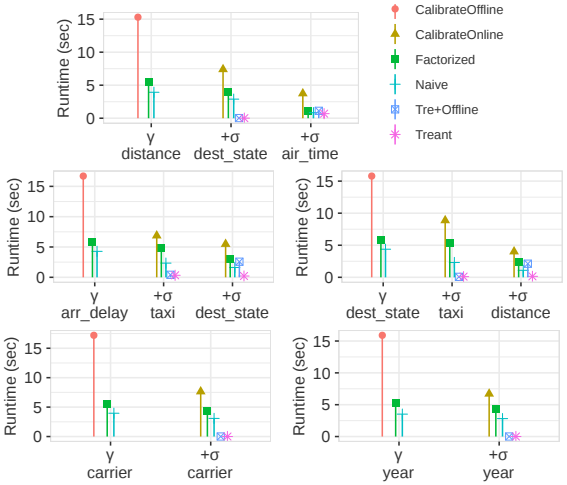


Figure 15: Flights Dashboard Performance. The dashboard features five visualizations. For each visualization, the left-most is *dashboard query*. *interaction queries* progressively adds (+) selection or group-by element to the previous one.

`CJT` (**Tre+Offline**) takes > 2s due to the larger steiner tree size, and online calibration reduces this time by >10×.

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 $\hat{Y}$ [34]: 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** [61], 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 17a 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.

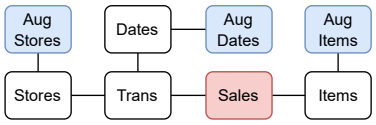
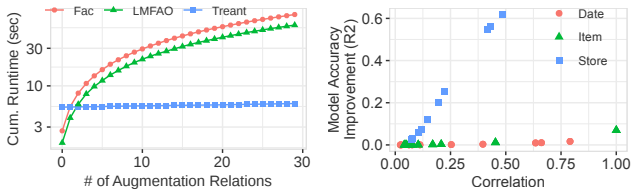


Figure 16: Favorita schema. Sales is the largest relation. Aug Stores, Aug Dates and Aug Items are augmentation relations.



(a) Cumulative runtime (log) (b) Accuracy (R2) improvements.

Figure 17: Augmentation run time and model performance.

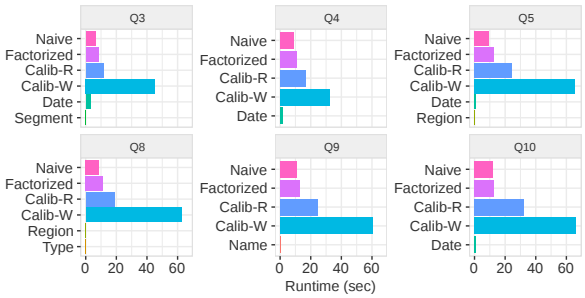


Figure 18: Run time for TPC-H dashboard. Naive executes queries without message passing. Factorized executes queries with message passing. Calib-R computes messages for calibration without materializing them, and Calib-W materializes them. The remaining bars are for *interaction queries* that vary the values of the parameters (labels).

Figure 17b 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).

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=10) 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 (see Appendix F). 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.

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

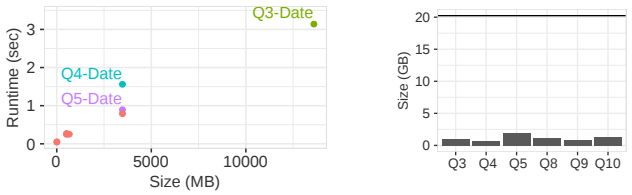


Figure 19: (a) Size of annotated bag (by predicate) vs query runtime. (b) Total message size overhead of Treant. Horizontal line is the TPC-H database size (~20 GB).

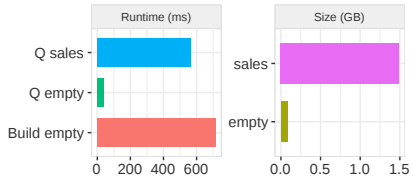


Figure 20: Runtime to build and query the empty bag and Store_Sales table, and their storage sizes.

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 18 shows the run time. Calibration (**Calib-W**) takes 4~7× longer than **Naive**. As expected, upward and downward message passing alone is ~2× slower (**Calib-R**), and the rest is dominated by high write overheads; Q8 groups by 2 attributes, so its message sizes are ~2× larger, and 4× 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× 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 19a). 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 <2GB compared to the original database size ~20GB (Figure 19b). 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: $\mathbf{Q} = \gamma_{\text{MAX(COUNT)}}(\gamma_{\text{COUNT}(\cdot),\text{Store},\text{Time}}(\bowtie))$ 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 20 shows the runtimes and sizes. Empty bag takes <800ms to build, and accelerates **Q** by ~8×. Additionally, the storage space required for the empty bag is 15× smaller than that of **Store_Sales**.

6 RELATED WORK

Interactive Queries. Previous works use indexing and data cubes [27] to support interactive queries. Some studies have improved them to Nanocubes [43] for spatiotemporal data and Hashedcubes [54] with additional optimizations, or use sophisticated materialization techniques [45, 47]. However, these approaches often have high preprocessing overhead (e.g., taking hours for 200MB data [43]), and require denormalization for large joins. In contrast, CJT has been shown to be exponentially more efficient for *interactive queries* over large joins (Appendix D) than data cubes with a constant factor overhead. Other approaches utilize approximation [25, 46] and prefetching [19], which are complementary to Treant.

Early Marginalization. Early Marginalization was first introduced by Gupta et al. [29] as a generalized projection for simple e.g., count, sum, max queries. It was extended by factorized databases to compactly store relational tables [53] and quickly execute semi-ring aggregation queries [32, 61]. Abo et al. [6] generalize early marginalization and establish the equivalence between early marginalization and variable elimination in Probabilistic Graphical Models [37]. However, prior works [61] only share work within a query batch but not between batches for interactive queries.

Calibrated Junction Tree. Calibration Junction Tree is first proposed by Shafer and Shenoy [64] to compute inference over probabilistic graphical models. Calibration Junction Tree has been widely used across engineering [59, 73], ML [13, 18], and medicine [39, 57], but they are limited to probabilistic tables. Yannakakis's algorithm [70] applies two-pass semi-join reduction to relations, but is limited to 0/1 semi-ring. We generalize the Calibration Junction Tree to semi-ring aggregation and extend it to support SPJA queries. There have been hardware optimizations [71, 72] that utilize GPU and SIMD to accelerate message passing for probabilistic inference; these are applicable to Treant and we leave them as future works.

Materialized View. Previous materialized view works [10, 42] focus on the optimization of view selections given workloads; these works require historical query logs, have significant overhead and rely on heuristics to solve the intractable optimization. CJT is lightweight with the same complexity of the *dashboard query*. Recent works on IVM [8, 51] are orthogonal to Treant: IVM reduces the message sizes by only including the changes, while Treant allows messages to be passed through only a subset of bags.

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.

REFERENCES

[1] Looker. https://www.looker.com/.

[2] Corporación favorita grocery sales forecasting. https://www.kaggle.com/c/favorita-grocery-sales-forecasting, 10 2017.

[3] Modin: Scale your pandas workflows by changing one line of code. https://github.com/modin-project/modin, 2018.

[4] Lightweight materialization for fast dashboards over joins (technical report). https://anonymous.4open.science/r/WideTable-627D/tech/tech.pdf, 2022.

[5] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):1–44, 2017.

[6] M. Abo Khamis, H. Q. Ngo, and A. Rudra. Faq: questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 13–28, 2016.

[7] M. Abourezq and A. Idrissi. Database-as-a-service for big data: An overview. *International Journal of Advanced Computer Science and Applications*, 7(1), 2016.

[8] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *arXiv preprint arXiv:1207.0137*, 2012.

[9] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *arXiv preprint arXiv:1207.0145*, 2012.

[10] Z. Asgharzadeh Talebi, R. Chirkova, and Y. Fathi. 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):391–415, 2009.

[11] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 739–748. IEEE, 2008.

[12] O. Bashaw. Drive revenue by using sigma with salesforce. https://www.sigmacomputing.com/blog/drive-revenue-by-using-sigma-with-salesforce, 2022.

[13] T. Braun and R. Möller. Lifted junction tree algorithm. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 30–42. Springer, 2016.

[14] N. Chepurko, R. Marcus, E. Zgraggen, R. C. Fernandez, T. Kraska, and D. Karger. Arda: automatic relational data augmentation for machine learning. *arXiv preprint arXiv:2003.09758*, 2020.

[15] F. G. Cozman et al. Generalizing variable elimination in bayesian networks. In *Workshop on probabilistic reasoning in artificial intelligence*, pages 27–32. Citeseer, 2000.

[16] R. Curtin, B. Moseley, H. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. Rk-means: Fast clustering for relational data. In *International Conference on Artificial Intelligence and Statistics*, pages 2742–2752. PMLR, 2020.

[17] F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing the data cube. *Distributed and Parallel Databases*, 11(2):181–201, 2002.

[18] J. Deng, N. Ding, Y. Jia, A. Frome, K. Murphy, S. Bengio, Y. Li, H. Neven, and H. Adam. Large-scale object classification using label relation graphs. In *European conference on computer vision*, pages 48–64. Springer, 2014.

[19] P. R. Doshi, E. A. Rundensteiner, M. O. Ward, and I. Stroe. Prefetching for visual data exploration. *Database Systems for Advanced Applications (DASFAA)*, 2002.

[20] P. Eichmann, E. Zgraggen, C. Binnig, and T. Kraska. Idebench: A benchmark for interactive data exploration. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1555–1569, 2020.

[21] R. C. Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker. Aurum: A data discovery system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1001–1012. IEEE, 2018.

[22] R. C. Fernandez, P. Subramaniam, and M. J. Franklin. Data market platforms: Trading data assets to solve data problems. *arXiv preprint arXiv:2002.01047*, 2020.

[23] A. Ferrari and M. Russo. *Introducing Microsoft Power BI*. Microsoft Press, 2016.

[24] W. Fischl, G. Gottlob, and R. Pichler. General and fractional hypertree decompositions: Hard and easy cases. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 17–32, 2018.

[25] D. Fisher, I. Popov, S. Drucker, and M. Schraefel. Trust me, i'm partially right: incremental visualization lets analysts explore large datasets faster. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1673–1682, 2012.

[26] J. Gale, M. Seiden, G. Atwood, J. Frantz, R. Woollen, and Ç. Demiralp. Sigma worksheet: Interactive construction of olap queries. *arXiv preprint arXiv:2012.00697*, 2020.

[27] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery*, 1(1):29–53, 1997.

[28] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40, 2007.

[29] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. 1995.

[30] J. Han, N. Stefanovic, and K. Koperski. Selective materialization: An efficient method for spatial data cube construction. In *Pacific-Asia conference on knowledge discovery and data mining*, pages 144–158. Springer, 1998.

[31] M. Idris, M. Ugarte, and S. Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1259–1274, 2017.

[32] M. Joglekar, R. Puttagunta, and C. Ré. Aggregations over generalized hypertree decompositions. *arXiv preprint arXiv:1508.07532*, 2015.

[33] M. R. Joglekar, R. Puttagunta, and C. Ré. Ajar: Aggregations and joins over annotated relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 91–106, 2016.

[34] H. F. Kaiser and K. Dickman. Sample and population score matrices and sample correlation matrices from an arbitrary population correlation matrix. *Psychometrika*, 27(2):179–182, 1962.

[35] M. A. Khamis, R. R. Curtin, B. Moseley, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. Functional aggregate queries with additive inequalities. *ACM Transactions on Database Systems (TODS)*, 45(4):1–41, 2020.

[36] M. A. Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. Ac/dc: In-database learning thunderstruck. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, pages 1–10, 2018.

[37] D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[38] N. Kotsis and D. R. McGregor. Elimination of redundant views in multidimensional aggregates. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 146–161. Springer, 2000.

[39] S. L. Lauritzen and N. A. Sheehan. Graphical models for genetic analyses. *Statistical Science*, pages 489–514, 2003.

[40] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal*, 27(5):643–668, 2018.

[41] V. Lepar and P. P. Shenoy. A comparison of lauritzen-spiegelhalter, hugin, and shenoy-shafer architectures for computing marginals of probability distributions. *arXiv preprint arXiv:1301.7394*, 2013.

[42] J. Li, Z. A. Talebi, R. Chirkova, and Y. Fathi. A formal model for the problem of view selection for aggregate queries. In *East European Conference on Advances in Databases and Information Systems*, pages 125–138. Springer, 2005.

[43] L. Lins, J. T. Klosowski, and C. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2456–2465, 2013.

[44] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics*, 20(12):2122–2131, 2014.

[45] Z. Liu, B. Jiang, and J. Heer. immens: Real-time visual querying of big data. In *Computer Graphics Forum*, volume 32, pages 421–430. Wiley Online Library, 2013.

[46] D. Moritz, D. Fisher, B. Ding, and C. Wang. Trust, but verify: Optimistic visualizations of approximate queries for exploring big data. In *Proceedings of the 2017 CHI conference on human factors in computing systems*, pages 2904–2915, 2017.

[47] D. Moritz, B. Howe, and J. Heer. Falcon: Balancing interactive latency and resolution sensitivity for scalable linked visualizations. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–11, 2019.

[48] E. G. Morton-Owens and K. L. Hanson. Trends at a glance: A management dashboard of library statistics. *Information Technology and Libraries*, 31(3):36–51, 2012.

[49] T. Neumann and B. Radke. Adaptive optimization of very large join queries. In *Proceedings of the 2018 International Conference on Management of Data*, pages 677–692, 2018.

[50] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):1–40, 2018.

[51] M. Nikolic and D. Olteanu. Incremental view maintenance with triple lock factorization benefits. In *Proceedings of the 2018 International Conference on Management of Data*, pages 365–380, 2018.

[52] B. of Transportation Statisticsw. Bureau of transportation statistics. http://www.transtats.bts.gov, 2017.

[53] D. Olteanu and J. Závodnỳ. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)*, 40(1):1–44, 2015.

[54] C. A. Pahins, S. A. Stephens, C. Scheidegger, and J. L. Comba. Hashedcubes: Simple, low memory, real-time visual exploration of big data. *IEEE transactions on visualization and computer graphics*, 23(1):671–680, 2016.

[55] T. pandas development team. pandas-dev/pandas: Pandas, Feb. 2020.

[56] J. Pearl. *Reverend Bayes on inference engines: A distributed hierarchical approach*. Cognitive Systems Laboratory, School of Engineering and Applied Science …, 1982.

[57] A. L. Pineda and V. Gopalakrishnan. Novel application of junction trees to the interpretation of epigenetic differences among lung cancer subtypes. *AMIA Summits on Translational Science Proceedings*, 2015:31, 2015.

[58] M. Raasveldt and H. Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, 2019.

[59] J. C. Ramirez, G. Munoz, and L. Gutierrez. Fault diagnosis in an industrial process using bayesian networks: Application of the junction tree algorithm. In *2009 Electronics, Robotics and Automotive Mechanics Conference (CERMA)*, pages 301–306. IEEE, 2009.

[60] M. Schleich. Structure-aware machine learning over multi-relational databases. In *Proceedings of the 2021 International Conference on Management of Data*, pages 6–7, 2021.

[61] M. Schleich, D. Olteanu, M. Abo Khamis, H. Q. Ngo, and X. Nguyen. A layered aggregate engine for analytics workloads. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1642–1659, 2019.

[62] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*, pages 3–18, 2016.

[63] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, 1979.

[64] G. R. Shafer and P. P. Shenoy. Probability propagation. *Annals of mathematics and Artificial Intelligence*, 2(1):327–351, 1990.

[65] D. Taniar and R. B.-N. Tan. Parallel processing of multi-join expansion-aggregate data cube query in high performance database systems. In *Proceedings International Symposium on Parallel Architectures, Algorithms and Networks. I-SPAN'02*, pages 51–56. IEEE, 2002.

[66] J. S. Vitter, M. Wang, and B. Iyer. Data cube approximation and histograms via wavelets. In *Proceedings of the seventh international conference on Information and knowledge management*, pages 96–104, 1998.

[67] W. Wang, J. Feng, H. Lu, and J. X. Yu. Condensed cube: An effective approach to reducing data cube size. In *Proceedings 18th International Conference on Data Engineering*, pages 155–165. IEEE, 2002.

[68] Z. Wang, Y. Chu, K.-L. Tan, D. Agrawal, A. E. Abbadi, and X. Xu. Scalable data cube analysis over big data. *arXiv preprint arXiv:1311.5663*, 2013.

[69] K. Xirogiannopoulos and A. Deshpande. Memory-efficient group-by aggregates over multi-way joins. *arXiv preprint arXiv:1906.05745*, 2019.

[70] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, volume 81, pages 82–94, 1981.

[71] L. Zheng and O. Mengshoel. Optimizing parallel belief propagation in junction treesusing regression. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 757–765, 2013.

[72] L. Zheng, O. Mengshoel, and J. Chong. Belief propagation by message passing in junction trees: Computing each message faster using gpu parallelization. *arXiv preprint arXiv:1202.3777*, 2012.

[73] F. Zhu, H. A. Aziz, X. Qian, and S. V. Ukkusuri. A junction-tree based learning algorithm to optimize network wide traffic control: A coordinated multi-agent framework. *Transportation Research Part C: Emerging Technologies*, 58:487–501, 2015.

A MESSAGE PASSING ALGORITHM

Algorithm 1 Message Passing and Calibration Algorithm

```
// Pass Message from bag u to v where u, v ∈ V
function PassMessage(((E, V), 𝒳, 𝒴), u, v)
    // All the neighbours
    N(u) = {c|c → u ∈ ℰ}
    // All incoming messages from in-neighbours except v
    M(u) = {𝒴(i → u)|i ∈ N(u) ∧ i ≠ v}
    // Compute and store message from u to v
    𝒴(u → v) = ∑_{u−v∩u} ⋈ (M(u) ∪ 𝒳^{-1}(u))
end function

// Upward Message Passing to root r ∈ V
function Upward(((E, V), 𝒳, 𝒴), r)
    for all Bag c ∈ V − r from leaves to root r bottom up do
        p = parent of c
        PassMessage(((E, V), 𝒳, 𝒴), c, p)
    end for
end function

// Downward Message Passing from root r ∈ V
function Downward(((E, V), 𝒳, 𝒴), r)
    for all Bag p ∈ V from root r to leaves top down do
        for all child bag c of p do
            PassMessage(((E, V), 𝒳, 𝒴), p, c)
        end for
    end for
end function

// Calibrate Junction Hypertree
function Calibration(((E, V), 𝒳, 𝒴))
    // choose a random bag as root
    r ∈ V
    Upward(((E, V), 𝒳, 𝒴), r)
    Downward(((E, V), 𝒳, 𝒴), r)
end function
```

Algorithm 2 Join Algorithms

```
function Worst Case Optimal Join implements Join(R)
    // Apply worst-case optimal join algorithm
    return ⋈_{R∈R}
end function

function Indicator Projection implements Join(R, R_db)
    // Find all attributes in join result
    U = ⋃_{R∈R} S_R
    // Find all relations whose schema intersect with U and
    build indicator projection for them
    R_ind = {π^ind_{S_R∩U}(R)|R ∈ R_db}
    return Worst Case Optimal Join(R ∪ R_ind)
end function
```

B QUERY COMPLEXITY

In this section, we provide the background of *Fractional Hypertree Width*, and extend it for SPJA query [6, 32, 53].

B.1 Fractional Hypertree Width

We start from *Fractional Edge Cover*, which takes join graph and the size of each relation as input and outputs the complexity of join result. Quantifying the complexity of final join result is not directly useful because, with Early Marginalization, we can avoid the full join result. We then discuss *Fractional Hypertree Width*, which outputs the complexity of intermediate join result given the Early Marginalization optimization opportunity.

Hypergraph: Hypergraph is graph $\mathcal{G}$ with vertices $\mathcal{V}$ and hyperedges $\mathcal{E}$, where each hyperedge connects non-empty subset of $\mathcal{V}$. Hypergraph has been widely used to represent join graph, where each vertex represents attribute and hyperedge represents relation.

Given a set of annotated relations $\mathbf{R} = \{R_1, R_2..., R_n\}$, we assume that attributes to join share the same name and consider natural join $R_1 \bowtie R_2... \bowtie R_n$. We then build hypergraph $(\mathcal{V}, \mathcal{E})$ for the join, where $\mathcal{V}$ is the set of all attributes $S_{R_1} \cup S_{R_2}... \cup S_{R_n}$ and $\mathcal{E}$ are the schemas of relations $S_{R_1}, S_{R_2}, ..., S_{R_n}$.

By default, we assume that the set of annotated relations $\mathbf{R} = \{R_1, R_2..., R_n\}$ will result in a connected hypergraph. It's possible that user wants to compute aggregation queries over relations, where there aren't join keys connecting them, a set of hypergraph will be generated and Cartesian Products have to be computed. Our data structure and applications could be easily extended to support this special case. However, this situation is rare for real-world use case, and we don't discuss this special case here.

Fractional Edge Cover. (aka AGM bound [11]). Given Hypergraph $\mathcal{G} = (\mathcal{E}, \mathcal{V})$ where each hyperedge $e \in \mathcal{E}$ is associated with the size of corresponding relation $|R_e|$, the *Fractional Edge Cover* number ρ^* is the cost of an optimal solution of the following linear program:

$$\begin{aligned} \min \sum_{E \in \mathcal{E}} \log_2(|R_E|) x_E \\ \text{s.t.} \sum_{E:V \in E} x_E \geq 1, \forall V \in \mathcal{V} \\ x_E \geq 0, \forall E \in \mathcal{E} \end{aligned} \tag{3}$$

Fractional Edge Cover ρ^* has been proved to be tight output size bound of join result $O(|R_1 \bowtie R_2... \bowtie R_n|) = O(2^{\rho^*})$.

Example 14 (*Fractional Edge Cover). Consider the triangle query* $\sum_{A,B,C}(R(A, B) \bowtie S(B, C) \bowtie T(A, C))$. *Suppose that the size of each relation is* $O(n)$*, Its Fractional Edge Cover number* $\rho^* = \log_2(n)1.5$ *and the join size of all three relations is bounded by* $O(n^{1.5})$.

Worst Case Optimal Join. Even if the join result is bounded by *Fractional Edge Cover*, traditional binary join may result in intermediate result asymptomatically larger. Consider the triangle query $\sum_{A,B,C}(R(A, B) \bowtie S(B, C) \bowtie T(A, C))$ whose join size is is bounded by $O(n^{1.5})$. However, joining any pair of relations will result in intermediate result with size $O(n^2)$. To bound the intermediate result size during the execution of join, worst case optimal join [50] has been proposed to guarantee that the time complexity of evaluating join is proportional to the worst-case output size $O(2^{\rho^*})$. The basic idea of to join multiple relations together and carefully skip tuples impossible to appear in join result.

Fractional Hypertree Width. For semiring aggregation query over join graph, we don't need to fully compute the join result, as

we can apply early marginalization to eliminate attributes. Given Junction Hypertree $(\mathcal{T}, \mathcal{X}, \mathcal{Y}, \mathcal{Z})$ of hypergraph $(\mathcal{E}, \mathcal{V})$, we only need to join relations in each bag during Message Passing. *Fractional Hypertree Width* [6, 32] of a given Junction Hypertree is just computing the maximum *Fractional Edge Cover* placed on bags instead of the whole hypergraph. The *Fractional Hypertree Width* of Junction Hypertree fhtw$((\mathcal{T}, \mathcal{X}, \mathcal{Y}, \mathcal{Z}))$ is $\max_{t \in \mathcal{V}_\mathcal{T}} \rho^*_t$ where ρ^*_t is the *Fractional Edge Cover* of each bag:

$$\begin{aligned} \min \sum_{E \in \mathcal{E}} \log_2(|R_E|) x_E \\ \text{s.t.} \sum_{E: V \in E} x_E \geq 1, \forall V \in \mathcal{X}(t) \\ x_E \geq 0, \forall E \in \mathcal{E} \end{aligned} \tag{4}$$

Given hypergraph $(\mathcal{E}, \mathcal{V})$, *Fractional Hypertree Width* of hypergraph fhtw$((\mathcal{E}, \mathcal{V}))$ is the minimum of the *Fractional Hypertree Width*s over all possible Junction Hypertrees. Finding the minimum *Fractional Hypertree Width* is known to be NP-hard [24].

Notice that, to achieve the time complexity of *Fractional Hypertree Width* for semiring aggregation, using worst case optimal join alone during message passing is not enough.

Example 15 (Wost Case Optimal Join insufficient for *Fractional Hypertree Width*). *Consider the Junction Hypertree in the right of* **??** *for the triangle query* $\sum_{A,B,C}(R(A,B) \bowtie S(B,C) \bowtie T(A,C))$. *While the Fractional Hypertree Width is* $O(n^{1.5})$*, message from bag ABC to any other bag is* $O(n^2)$.

B.2 SPJA Query

To support SPJA queries over CJT, we previously use annotations to highlight the changes necessary for the message computations. These changes, however will affect the complexity of the queries:

Annotations in SPJA queries. We consider the effects of all annotations for a given JT:

- γ_A **:** Group-by will increase the complexity as the attributes can't be early marginalized out. Naturally, if its incoming messages contain additional attributes not contained by this bag, the join size of this bag naturally grows by the product of the domain sizes for attributes. This bound considers the worst case when the cartesian product of group-by attributes has to be computed. However, for group-by attributes from the same relation, their join size could be bounded by the relation size, which could be smaller than the product of domain sizes.
- $\sum_A$**:** In contrast to γ_A, canceling out the group-by decrease the complexity as attributes are early marginalized out.
- $\overline{R}$**:** Exclusion of relations may decrease (e.g., for bag containing two relations whose schema are different and share join key, removing one of them reduces the bag size from $O(n^2)$ to $O(n)$) or increase the complexity (e.g., for bag containing three relations with triangle join, removing one of them increases the bag size from $O(n^{1.5})$ to $O(n^2)$).
- R^*_{ver}**:** Updating relations will change the size of relation $|R|$.
- σ_{id}**:** Similar to update, selection will reduce the size of relation. We apply the standard estimation [63] to compute the selectivity.

To provide a bound of query complexity over annotated relations, we modify the JT based on the annotations, and compute the fractional hypertree width of the updated JT: for γ_A, we add attribute A to all bags, and for $\overline{R}$ we remove the relation R from JT. For R^*_{ver} and σ_{id}, we update the relation size.

C OPTIMIZE FEATURE AUGMENTATION WITH CJT.

While feature augmentations over single join key are efficient, those over multiples multiple join keys are complex. We need to query the CJT group-by all join keys, which might result in a large Steiner Tree and we need to re-design the JT after augmentation.

Feature Augmentation over Multiple Bag. For Feature Augmentation over multiple, we want to query the aggregation group-by the join keys from CJT. This could be considered as a SPJA query with group-by annotations, and can be computed through Upward Message Passing in the Steiner tree.

Connect Augmentation Relation to JT. To connect augmentation relation to JT where the join key is distributed over multiple bag, we have to add all the join key to the bags of Steiner tree, create an augmentation bag containing augmentation relation, and connect the augmentation bag to any of the bag in Steiner tree. Notice that the JT with added attributes in the bags can be inefficient, and we may redesign JT to find a better one.

Optimize CJT design. To optimize CJT for feature augmentation, we create empty bags for common join keys. Consider TPC-DS as an example, whose (simplified) join graph (also JT) is shown in Figure 5a. We can cluster time and stores in an empty bag shown in Figure 5b to support efficient augmentation of spatio-temporal features.

D DATA CUBE

OLAP data cubes [27] materialize a lattice of data cuboids parameterized by the set of attributes that future queries will filter/group by. Traditionally, the data structure is built bottom-up in order to share computation—each cuboid is built by marginalizing out irrelevant attribute(s) from a descendant cuboid. If the cube is over a join graph, then there is the additional cost of first materializing the (potentially very large) join result to compute the bottom cuboid. Although prior work explored many optimizations (parallelization [17, 65, 68], approximation [66], partial materialization [30, 67], early projection [38]), neither early marginalization nor work-sharing based on CJTs have been explored.

CJTs are a particularly good fit for building data cubes because, in practice, they are restricted to a small number of attributes in order to avoid exponentially large cuboids. In this setting, we can build CJTs for a carefully selected set of pivot queries to accelerate cube construction by 1) not materializing the full join graph when building the cuboids, and 2) aggressively reuse messages to answer OLAP queries not directly materialized by a cuboid.

D.1 Complexity Analysis

Let us first analyze the complexity of using CJTs to answer OLAP queries. This will provide the tools to trade-off between OLAP query performance and space requirements for materialization.

Let the database contain r relations each with O(n) rows, the domain of each attribute is O(d), and the join graph contains m

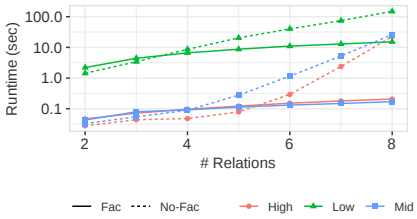


Figure 21: Run time of total count query with (JT)/without message passing (No-JT) in seconds (log scale). High, Mid, and Low are for different fanouts.

unique attributes. Suppose we have calibrated each cuboid with k group-by attributes (the pivot queries). Calibration costs $O(rnd^k)$ for each pivot query where the cost per bag is $O(nd^k)$ (cross product between relation size n and incoming messages). Since the output message size is also bound by $O(nd^k)$ due to marginalization, we incur this cost for each of r bags in the CJT. Thus the total cost is $O(rn(dm)^k)$ to calibrate all $\binom{k}{m} = O(m^k)$ pivot queries.

To simplify our analysis, let us also materialize the absorption results (join result of all incoming messages and relations mapped to a given bag) for each bag during calibration (Section 3.3.1). This does not change the worst-case runtime complexity, and increases the storage cost by at most the size of the base relations.

Notice that these absorption results can be directly used to answer OLAP queries with k + 1 attributes with no cost in complexity (Section 3.3.2). Thus, materializing cuboids of up to k + 1 attributes only requires the cost to calibrate cuboids with k attributes.

More generally, given an OLAP query that groups by h attributes (so it contains h group-by annotations $\mathbf{A_h}$), it is executed over a CJT with k attributes by finding the annotations that differ between the pivot and new query $\mathbf{A_{h-k}}$, and performing message passing over the associated steiner tree (Section 3.4.2). Further, since the calibrated pivot queries span all combinations of k attributes, we simply need to find the pivot query that results in the steiner tree that spans the fewest bags. The optimal pivot query that minimizes the steiner tree could be found in time polynomial in r through dynamic programming, and the algorithm is in Appendix E.

To summarize, calibration of all pivot queries with k attributes costs $O(rn(dm)^k)$, and cost to execute an OLAP query with h > k attributes is $O(s(\mathbf{A_{h-k}}) \times \varphi)$, where $s(\mathbf{A_{h-k}})$ is the number of bags in the steiner tree spanning $\mathbf{A_{h-k}}$, and φ is the size of the absorption result in $O(nd^{h-1})$, which upper bounds the message size.

D.2 Experiments

Dataset. Following prior JT work [69], we created synthetic dataset that contains $r \in [2, 8]$ relations with a chain schema:

$$R(A_1, A_2), R(A_2, A_3), \ldots, R(A_r, A_{r+1}).$$

We vary the fanout f between adjacent relations (low=2, mid=5, high=10), and the attribute domain size d. For each value of A_i in $R(A_i, A_{i+1})$, we assign f unique values to A_{i+1} with fanout f being implemented by, for each value in A_i, assigning f sequential values to A_{i+1}, such that the n^{th} value is n%d. Thus, the fanout f is in both directions. We vary the fanout (and domain) and keep the total join size $d \times f^8$ fixed to be 10^9. The domain sizes d for different fanouts are $d_{low} = 3906250$, $d_{mid} = 2560$, and $d_{high} = 10$.

D.2.1 Message Passing Costs. We first evaluate the benefits of message passing (but not calibration) in cloud settings. The compiler generates CREATE VIEW statements, so that messages are *not* materialized. We execute the total count query as a large join-aggregation query (No-JT) or as an upward message passing (JT).

Figure 21 varies the number of relations (x-axis) and fanout (line marker). Message passing reduces the runtimes from exponential to linear due to early marginalization, but incurs a small overhead to perform marginalization when there are few relations. Low fan-out has the largest runtime because we fix the total join size and hence the low fan-out has the largest domain size. Note that the x-axis is also interpretable as the steiner tree size.

For message passing, we can also interpret number of relations as the Steiner tree size, and the run time grows linearly in the Steiner tree size when the bag size is constant.

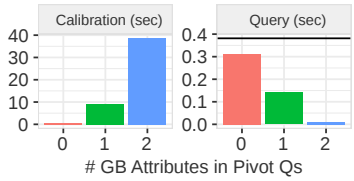


Figure 22: We vary the dimensionality of the calibrated pivot queries k ∈ {0, 1, 2} and measure calibration runtime and impact on 4-attribute OLAP queries. Horizontal line represents the average runtime with JT.

D.2.2 Cubes in the Cloud. CJTs help developers build data cubes to explicitly trade-off build costs and query performance. To evaluate this, we use the synthetic dataset with f = 10 (high) fanout and r = 8 relations, and calibrate all cuboids with $k \in [1, 3]$ grouping attributes. For each k, we use the cuboids to execute 100 random OLAP queries with 4 grouping attributes.

The results are in Figure 22. Although calibration cost increases exponentially (as expected), message passing is still significantly faster than naive query execution: computing *all* 2-attribute cuboids (through calibration of all 1 group-by attribute Pivot Qs in 8.8s) is substantially faster than naively computing a *single* 0-attribute cuboid (22.6s for No-JT in Figure 21). At the same time, increasing the dimensionality of the cuboids (k) significantly reduces the query runtimes (2.71× speedup for k = 1, and 33.73× speedup for k = 2) due to the smaller steiner tree.

The total Redshift table sizes created during calibration is exponential in k, and consistent with the analysis in Appendix D.1. Redshift appears to pad small tables to >6MB, hence the large sizes (~4GB for k = 2). Unfortunately. the tables cannot be naively compacted (by unioning into a single table) because their schemas are different. Thus we report the actual *Data Size* by adding tuple size times cardinality across the tables. The overhead is only 0.17× for k = 0, 5× for k = 1 and 127.73× for k = 2. Given the significant query performance improvement, the space-time trade-off may be worthwhile.

Takeaways. For CJT, the calibration time grows exponentially in the number of group-by attributes, as there are exponential more pivot queries to calibrate, each takes longer time for larger messages. CJT is much more efficient than data cube: CJT can compute all data

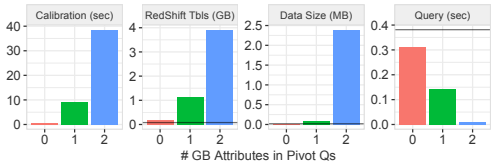


Figure 23: RedShift pads tables to be at least 6MB, which penalizes many small tables. So we report the total RedShift table size, and actual data size. Horizontal lines represent, from left-to-right: base DB table size, base DB actual data size, and average runtime with JT

cuboids up to two group-by attributes in 8.8s, while data cube with naive join takes 22.6s just to compute apex cuboid (Figure 21). The larger number of group-by attributes improves the future OLAP queries performance significantly by reducing Steiner tree size. The reduction of Steiner tree size is especially significant for pivot queries of 2 group-by attributes: given 4 random attributes, there is a high chance (~ 90%) that two of them are in the same relation so only a scan over absorption is needed.

The total Redshift table sizes created during calibration is exponential in k, and consistent with the analysis in Appendix D.1. Redshift appears to pad small tables to >6MB, hence the large sizes (~4GB for k = 2). Unfortunately. the tables cannot be naively compacted (by unioning into a single table) because their schemas are different. Thus we report the actual *Data Size* by adding tuple size times cardinality across the tables. The overhead is only 0.17× for k = 0, 5× for k = 1 and 127.73× for k = 2. Given the significant query performance improvement, the space-time trade-off may be worthwhile.

E MINIMIZE STEINER TREE

In this section, we present the algorithm that, given the JT with r bags and h group-by annotations, find the minimum Steiner tree of k annotations in time polynomial to r. We simplify the problem by assuming that each bag has the same size, so the problem is to identify the minimum Steiner tree in terms of the number of bags; our algorithm can be easily extended to the general case.

In JT, each group-by annotation could be applied to a set of bags containing group-by attributes. We first solve the problem where each annotation could be placed to exactly one bag, then generalizes the algorithm.

Single bag annotation. If each annotation could only be placed on single bag, the problem reduced to: given JT with r bags and h annotated bags, find the minimum Steiner tree of k annotated bags in terms of the number of bags. We solve the problem by recursion and dynamic programming. We make the edges in JT bidirectional. For each directed e, it keeps track of x[e][n] defined as "In the sub-tree this edge directs to, what is the minimum number of bags in the Steiner tree that contains the target bag of this edge and n annotated bags (including the target bag if it's annotated)", where n is from 0 to k. This can be computed as follows:

- **Base Case:** For edge e whose target bag is leaf bag, then x[e][1] = 1 and x[e][n] = Inf for n > 1. For all edges, x[e][0] = 0
- **Recursive Case:** Given edge e and target bag b, let E be the set of edges from b but is not e. One naive way to compute x[e][n] is to consider all possible assignment of the number of annotated bags in to E. This is inefficient as the number of assignments is exponentially large. Instead we combine edges in E one-by-one into one edge. Given two directed edges e_1 and e_2 in E, we combine them into one directed edge e^* as follows:
  $x[e^*][n] = \min(x[e_1][m] + x[e_2][n - m]$ for m = 0...n)
  Given the final e^* that combines all edges in E, we add the target bag. If the target bag is annotated, $x[e][n] = x[e^*][n - 1] + 1$ for n = 1...k. Otherwise, $x[e][n] = x[e^*][n] + 1$.

After we compute the x[e][n] for all directed edges and n from 0 to h, we can find the minimum Steiner tree size by iterating over all edge and compute the minimum Steiner tree containing any end node of this edge. For edge whose two directed edges are e_1 and e_2, the minimum Steiner tree has size $\min(x[e_1][m] + x[e_1][j - m]$ for m = 0...n).

We analyze the time complexity of the algorithm. We assume both h and k is O(r). For each recursion, each combine takes time $O(r^2)$ (as each $x[e^*][n]$ takes O(r) and there are O(r) n to consider) and there are O(r) combines so $O(r^3)$ in total. There are O(r) directed edges, so $O(r^4)$ to compute all x[e][n]. To find the minimum Steiner tree with x, we iterate O(r) edges and each iteration takes O(r). Therefore, the algorithm takes $O(r^4)$ in total.

Multiple bags annotation. In general, each annotation could be placed on multiple bags. One naive solution is to consider all possible placements, which is exponential in r. Instead, we consider the case where each bag is the root such are all annotations are greedily placed to the bags closest to the root in $O(r^2)$. There are O(r) possible roots, each could be solved using the previous algorithm in $O(r^4)$, so the final algorithm takes $O(r^5)$.

F TPC-H DETAILS

We discussed how we rewrite TPC-H queries into semi-ring SPJA queries.

Query 3. We remove top and order-by. We also remove L_ORDERKEY group-by because otherwise the result has too many groups.

```
SELECT SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE,
O_ORDERDATE, O_SHIPPRIORITY
FROM CUSTOMER, ORDERS, LINEITEM
WHERE C_MKTSEGMENT = 'FURNITURE' AND
C_CUSTKEY = O_CUSTKEY AND L_ORDERKEY = O_ORDERKEY AND
O_ORDERDATE < '1995-03-28' AND L_SHIPDATE > '1995-03-28'
GROUP BY O_ORDERDATE, O_SHIPPRIORITY;
```

Query 4. We rewrite the nested query and remove order-by and distinct.

```
SELECT O_ORDERPRIORITY, count(distinct  O_ORDERKEY)
FROM LINEITEM, ORDERS
WHERE O_ORDERDATE >= '1997-04-01' AND
O_ORDERDATE < cast (date '1997-04-01' + interval '3 months' as date)
AND L_ORDERKEY = O_ORDERKEY AND L_COMMITDATE < L_RECEIPTDATE
GROUP BY O_ORDERPRIORITY;
```

Query 5. For query 5, we break cycle with additional optimization.

```
SELECT N_NATIONKEY,
SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS REVENUE
FROM CUSTOMER, ORDERS, LINEITEM, SUPPLIER, NATION, REGION
```

```
WHERE C_CUSTKEY = O_CUSTKEY AND L_ORDERKEY = O_ORDERKEY AND
L_SUPPKEY = S_SUPPKEY AND C_NATIONKEY = S_NATIONKEY AND
S_NATIONKEY = N_NATIONKEY AND N_REGIONKEY = R_REGIONKEY AND
R_NAME = 'MIDDLE EAST' AND  o_orderdate >= date '1994-01-01' AND
o_orderdate < cast (date '1994-01-01' + interval '1 year' as date)
GROUP BY N_NATIONKEY;
```

Break cycle for Q_5. Q_5 joins customer and supplier by nation, which makes the join graph cyclic and JT expensive. Luckily, Q_5 also group-by nation. We discuss the technique to break cycle: rewrite join + group-by as a set of selections.

Consider the cyclic join of R(A,B), S(A,C), T(B,C). If we know that all future queries will group-by attribute A, we can break the cycle through query rewriting. The original join query is:

```
SELECT A, COUNT(*)
FROM R(A,B), S(A,C), T(B,C)
WHERE R.A = S.A AND R.B = T.B AND S.C = T.C
GROUP BY R.A
```

The group by query could be considered as a set of smaller queries, each select a value of A in its domain dom(A). Therefore, for each a ∈ dom(A), we query

```
SELECT COUNT(*)
FROM R(A,B), S(A,C), T(B,C)
WHERE R.A = S.A AND R.B = T.B AND S.C = T.C AND R.A
= a AND S.A = a
```

The rewritten query has acyclic join graph. This optimization is closely related to conditioning in Probabilistic graphical model [37].

Query 8. We only consider the inner query, as outer query is cheap to compute.

```
SELECT extract(year from o_orderdate) as o_year,
SUM(L_EXTENDEDPRICE * (1-L_DISCOUNT)), N2.N_NATIONKEY
FROM PART, SUPPLIER, LINEITEM, ORDERS,
CUSTOMER, NATION N1, NATION N2, REGION
WHERE P_PARTKEY = L_PARTKEY AND S_SUPPKEY = L_SUPPKEY AND
L_ORDERKEY = O_ORDERKEY AND O_CUSTKEY = C_CUSTKEY AND
C_NATIONKEY = N1.N_NATIONKEY AND N1.N_REGIONKEY = R_REGIONKEY AND
R_NAME = 'ASIA' AND S_NATIONKEY = N2.N_NATIONKEY AND
O_ORDERDATE BETWEEN '1995-01-01' AND '1996-12-31' AND
P_TYPE = 'MEDIUM ANODIZED COPPER'
GROUP BY N2.N_NATIONKEY, o_year;
```

Query 9. We remove the order-by.

```
SELECT N_NAMEAS NATION,
extract(year from o_orderdate) as o_year,
SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)-PS_SUPPLYCOST*L_QUANTITY)
FROM PART, SUPPLIER, LINEITEM, PARTSUPP, ORDERS, NATION
WHERE S_SUPPKEY = L_SUPPKEY AND PS_SUPPKEY = L_SUPPKEY
AND PS_PARTKEY = L_PARTKEY AND P_PARTKEY = L_PARTKEY
AND O_ORDERKEY = L_ORDERKEY AND S_NATIONKEY = N_NATIONKEY
AND P_NAME LIKE '%green%'
GROUP BY NATION, O_YEAR
```

Query 10. We remove the limit, order-by and modify the interval to 1 day because otherwise, the result size is too large to return without materialization.

```
SELECT C_CUSTKEY, C_NAME, SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT))
AS REVENUE, C_ACCTBAL, N_NAME, C_ADDRESS, C_PHONE, C_COMMENT
FROM CUSTOMER, ORDERS, LINEITEM, NATION
WHERE C_CUSTKEY = O_CUSTKEY  AND L_ORDERKEY = O_ORDERKEY
AND O_ORDERDATE >= '1994-01-01'
AND O_ORDERDATE < cast (date '1994-01-01' + interval '1 days'
as date) AND L_RETURNFLAG = 'R' AND C_NATIONKEY = N_NATIONKEY
GROUP BY C_CUSTKEY, C_NAME, C_ACCTBAL,
C_PHONE, N_NAME, C_ADDRESS, C_COMMENT
```