Calibration: A Simple Trick for Fast Interactive Join Analytics

ABSTRACT

Data analytics over normalized databases typically requires computing and materializing expensive joins. Factorized query execution models execution as message passing between relations in the join graph and pushes aggregations through joins to reduce intermediate result sizes. Although this accelerates query execution, it only optimizes a single query. In contrast, analytics is iterative, and offers novel work-sharing opportunities between queries.

This work shows that carefully materializing messages during query execution can accelerate subsequent aggregation queries by $>10^5\times$ as compared to factorized execution, and only incurs a constant factor overhead. The key challenge is that messages are sensitive to the message passing ordering (akin to join ordering). To address this challenge, we borrow the concept of calibration in probabilistic graphical models to materialize sufficient messages to support any ordering. We manifest these ideas in the novel *Calibrated Junction Hypertree* (CJT) data structure, which is fast to build, aggressively re-uses messages to accelerate future queries, and is incrementally maintainable under updates. We further show how CJTs benefit applications such as OLAP, query explanation, streaming data, and data augmentation for ML. Our experiments evaluate three versions of the CJT that run in a single-threaded custom engine, on cloud DBs, and in Pandas, and show $30\times - 10^5\times$ improvements over state-of-the-art factorized execution algorithms on the above applications.

ACM Reference Format:

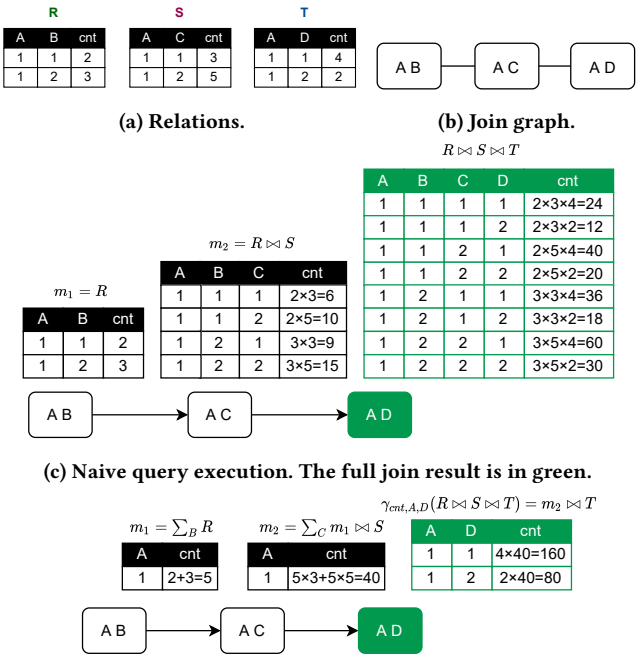
. 2022. Calibration: A Simple Trick for Fast Interactive Join Analytics. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Schema normalization is a foundational concept in databases, and is used to minimize redundancy, potential data inconsistencies, and storage costs. It is widely used in practice and taught in nearly every database course. Unfortunately, normalized schemas present a number of usability challenges in modern data analytics. First, analyses often access data from disparate tables that necessitate joining across the normalized schema (called *join graph*). These massive joins are difficult to optimize [9, 49, 59], expensive to materialize, and dominate analytics costs. Second, joins are notoriously confusing to students and programmers [17, 30, 41, 56].

As such, there is increasing advocacy for a wide-table abstraction [15, 58, 58], where users directly perform analytics over a fully denormalized schema. Unfortunately, materializing a denormalized schema incurs exponential space overhead $O(n \times f^r)$, where f is the fanout along edges in a join graph with r relations each of size n.

To address this challenge, factorized query execution [6, 70] accelerates SPJA queries over a large join graph using early marginalization. In the spirit of projection pushdown, early marginalization pushes down aggregation through the joins to reduce intermediate result sizes. Abo et al. [7] established the equivalence between early marginalization and message passing in Probabilistic Graphical Model [43] (PGM). Factorized query execution can then be modeled as passing messages between relations in the join graph. The messages are of size O(n), so the space overhead (for acyclic join) is only linear: O(rn).



(a) Relations. (b) Join graph.

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

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

Figure 1: Example database with three relations, its join graph (also JT), naive query execution for total count, and factorized query execution by upward message passing.

Example 1. *Figure 1(a,b) list example relations (duplicates are tracked with a* `cnt` *"annotation") and the join graph, respectively. Figure 1c naively computes the total count over the full join result (wide table) using message passing, where each message is the intermediate result so far. For instance,* AB *sends itself to* AC*, which sends the join result to* AD*, which computes the full join before summing the counts. This clearly requires exponential space. In contrast, factorized query execution distributes the summation through the joins, so that each*

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

node first sums out (marginalizes) any attributes irrelevant downstream, so that the node emits a smaller message. In Figure 1d, AB *marginalizes out* B *and* AC *marginalizes out* C*, so that the final join result is only 2 tuples.*

This concept has been extended to handle arbitrary join graph structures [62]; identify good message passing orders [70] (akin to join ordering); and support complex aggregations over semi-ring structures such as factorized learning [20, 71], where the aggregation function trains a ML model.

These properties make factorized execution promising for wide-table analytics. However, data analytics is iterative [27, 73]: the user incrementally builds and changes the query, and expects interactive response times. Although factorized execution reduces individual query latencies, it does not exploit work-sharing across iterations (either between similar queries, or between pre-computed data structures and queries). Our experiments show that queries can be $>10^5\times$ slower than need be.

Our core question is: **given an analytics workload over a wide-table (join graph), what intermediates should be materialized?** We decompose this into several technical challenges. First, which messages can be shared between two wide-table queries? Iterative analyses primarily change the selection and grouping clauses [27, 73] and could aggressively re-use messages. To this end, we introduce an efficient method to check the equivalence of intermediate messages.

Second, given an initial query, which of its messages are sufficient to support re-use for a future query workload? The key challenge is that simply caching the messages emitted when executing the initial query is insufficient because message re-usability is sensitive to the message-passing order (i.e., join order). This would force future queries to either use the same ordering or forgo message re-use altogether. To address this, we observe that query execution passes messages across all edges in the join graph along a single direction, and show that sending messages in reverse is sufficient to support arbitrary orderings in future queries. We relate this to *calibration* in probabilistic graphical models [72], which is similarly used to share computation between posterior distributions.

Third, we bring these ideas together in the **Calibrated Junction HyperTree** (CJT). Given a pivot query, the novel data structure manages message materialization and re-use. Building the data structure only takes up to twice the time as executing a single query, but supports re-use for arbitrary message passing orders.

Finally, we apply CJT to three important classes of wide-table applications. CJT accelerates **Data Cube** construction by re-using messages from low-dimensional cuboids to answer higher dimensional OLAP queries, and avoiding the exponential cost of constructing high dimensional cuboids directly. CJT enables interactive **Data Augmentation for ML** by reducing the time to add a new relation (features) to the join graph and update an ML model by $> 10^2\times$ faster than prior approaches. CJT can directly use factorized IVM [8, 61] to accelerate **Data Explanation and Streaming** applications, and we further reduce IVM maintenance overheads by $>92\times$ by lazily maintaining materialized messages.

To summarize, our contributions are as follows:

- Conceptually, we expand the connection between factorized queries and PGM by drawing on the idea of calibration.
- Practically, we design the novel CJT data structure, which uses calibration to enable work-sharing for wide-table analytics. The cost of materializing the data structure is within a constant factor of a single query execution, but accelerates future queries by multiple orders of magnitude.
- We apply CJT to data cube, data augmentation for ML, streaming and explanation applications, and describe additional application-specific optimizations.
- To illustrate the algorithmic benefits of our ideas, we implement and evaluate three versions of CJT: a custom single threaded query engine and middleware compilers to SQL and Pandas data frame operations. Our custom engine out-performs the state-of-the-art LMFAO factorized engine by $\sim 30\times$ on OLAP queries; compared to factorized execution algorithms, our SQL experiments on AWS Redshift reduce execution by up to $10^3\times$ on TPC-H queries, while Pandas experiments accelerate data augmentation for ML by $>100\times$.

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 join representation. Our goal in this paper is to keep the content accessible. To this end, we avoid technical concepts (e.g., hypergraphs) that are needed for deriving bounds but not needed for developing intuition, and limit the discussion to *COUNT* queries. However, our work generalizes to any commutative semi-ring aggregation query [33], and the full technical details can be found in the technical report [4].

Data Model. Let uppercase symbol A be an attribute, dom(A) is its domain, and lowercase symbol $a \in \text{dom(A)}$ 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 $\text{dom(R)} = \times_{A \in S}\text{dom(A)}$ is the Cartesian product of its attribute domains. An attribute is incident of R if $A \in S_R$. Given tuple t, let t[A] be its value of attribute A.

Annotated Relations. Since relational algebra (first-order logic) does not support aggregation, it has been extended with the use of commutative structures to support aggregation. The main idea is that tuples are annotated with values from a semi-ring, and when relational operators (e.g., join, project, group-by) concatenate or combine tuples, they also multiply or add the tuple annotations, such that the final annotations correspond to the desired aggregation results.

A commutative semi-ring $(D, +, \times, 0, 1)$ defines a set D, binary operators + and $\times$ 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). Our applications and experiments illustrate these use cases, and the technical report presents a full treatment [4]. 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 [33, 37, 61]. 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[1].

Consider the query $\varphi_{\mathbf{S'}} = \gamma_{\mathbf{S}-\{A\},COUNT}(R_1 \bowtie R_2 ... \bowtie R_n)$ that joins n relations, groups by all attributes except A: **S'** = **S** − {A}, and computes the *COUNT*. The operators that combine multiple tuples are join and groupby (projection under set semantics corresponds to groupby), and they compute the output tuple annotations as follows:

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

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

The first statement states that given a join output tuple t, its annotation is defined by multiplying the annotations of the pair of input tuples, where S_R and S_T are R and T's schemas. 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, and the query can be rewritten as $\varphi_{\mathbf{S'}} = \sum_{A \in \mathbf{S'}}(R_1 \bowtie R_2 ... \bowtie R_n)$. 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 [34].

Consider Figure 1, 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 the 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 [19] and is widely used for inference in Probabilistic Graphic Models [43]. The order in which which attribute(s) are marginalized out (by clustering and joining the incident relations) is called the *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 [60]). It is well known that finding the optimal order (with the minimum intermediate join size) is NP-hard [26]. Prior work [7] has shown that the intermediate result size of optimal order is bound by the fractional hypertree width of the join graph, which we introduce in Appendix A. However, common database queries are over acyclic join graph, whose optimal order could be found efficiently as discussed below.

Junction Hypertree. The Junction Hypertree[2] (JT) is a representation of a join query that is amenable to complexity analysis [7, 38] and semi-ring aggregation query optimization [6]. In the next section, we will show how JT can be materialized and maintained to provide work sharing and optimization opportunities for join-aggregation queries, and how to extend it to balance storage costs and query benefits. For now, we simply define the structure.

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 ∈ 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 [7], a JT vertex is also called a *bag*. A JT must satisfy three properties:

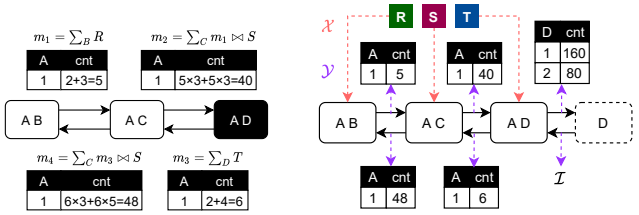
- **Vertex Coverage:** the union of all bags in the tree must be equal to the set of attributes in the join graph.
- **Edge Coverage:** for every relation R in the join graph, there exists at least one bag that is a superset of R's attributes.
- **Running intersection:** for any attribute in the join 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 [26], we can trivially create the optimal JT for an acyclic join graph by creating one bag for each relation (e.g., the JT is simply the join graph) and the size of each bag is bounded by its corresponding relation size. We refer readers to FAQ [7] for a complete description; in this work, we assume that a good JT has been determined.

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

[2] JT is also called Hypertree Decomposition in databases [7, 38] and Clique Hypertree in probalistic graphical models [43].



(a) Upward and downward message passing.

(b) Calibrated Junction Hypertree with empty bag (dotted). $\mathcal{I}$ is the identity relation.

Figure 2: Message Passing and Calibration

Message Passing for Query Execution. Message Passing was first introduced by Judea Pearl in 1982 [63] (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. Abo et al. [7] established the equivalence between variable elimination (query execution) and message passing. Below, we illustrate how message passing over a JT is used for query execution, and the next section leverages the ability to reuse messages 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 tables 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 edges, and once the root has received all incoming messages, its updated contents correspond to the query result.

Example 2 (Message Passing). *Consider the relations in Figure 1a, and the* JT *in Figure 1b where each bag is a base relation. We wish to execute* $\sum_{ABCD}$ R(A, B) ⋈ S(A, C) ⋈ T(A, D) *by traversing along the path* AB → AC → AD *(Figure 1d). 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 1d) and marginalizes out* A *and* D *to compute the final result.*

Scope. Following prior factorized query execution work [61, 62, 70], we use the acyclic join graph as the JT, or assume a good JT has already been determined. Although CJT supports any commutative semi-ring, theta joins with aribtrary conditions, outder joins, and any factorized machine learning model[3], we try keep the ideas accessible to a general database audience and base our examples on natural numbers (a semi-ring), *COUNT* queries, with natural joins, and the linear regression model.

3 CALIBRATED JUNCTION HYPERTREE

While message passing over JT exploits early marginalization to accelerate query execution, it has traditionally been limited to use in complexity analysis and single-query execution. This section introduces the Calibrated Junction Tree (CJT) to enable work-sharing for fast analytic queries over the full join graph. The idea is to materialize messages over the JT for a representative *Pivot Query*, and reuse a subset of its messages for future queries. This section will focus on the basis for the CJT data structure and how it is used to execute new queries. The next section will describe how to customize CJT for a range of useful application domains.

Our novelty is 1) to use JTs as a concrete data structure to support message reuse across queries, and 2) to borrow *Calibration* [72] from probabilistic graphical models to ensure that sufficient messages are materialized to efficiently support arbitrary SPJA queries over the full join graph. Although CJT is widely used across engineering [66, 86], ML [14, 22], and medicine [47, 64], we are the first to introduce CJT in the context of query execution and generalize it to semi-ring aggregation queries.

3.1 Motivating Example

We first illustrate work sharing examples between a pivot query $Q_1 = \sum_{ABCD}$ AB ⋈ AC ⋈ AD (whose messages are materialized) and $Q_2 = \sum_{ABCD}$ AB ⋈ $\sigma_{c=1}$(AC) ⋈ AD. Q_1 computes the total count, and Q_2 applies an additional predicate C=1.

Example 3. *The* JT*s in Figure 3a both assign* AD *as the root and traverse along the path* AB → AC → AD*. Although the message* AB → AC *will be identical (blue edges), the additional filter over* AC *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* AC *as the root, so both messages can be reused and the* AC *bag simply applies the filter after joining its incoming messages.*

This example shows that message reuse depends on how the root bag is chosen in the pivot query, and for different queries, we may wish to choose different roots. Since we may not know the exact join, grouping, and filter criteria of future queries, it is not effective to simply materialize messages for a single root. The CJT data structure addresses these limitations, and the following text describes the CJT data structure, message reuse, and query execution given a CJT.

3.2 Junction Hypertree as Data Structure

A naive approach to re-use messages is to execute an aggregation query over a JT, and store the messages; when a future query traverses an edge in the JT, it simply reuses corresponding message. Unfortunately, this is 1) inaccurate, because messages generated along an edge are not symmetric, so that message contents depend

[3]Including ridge regression, classification tree, regression tree [70], k-means (RK-means), support vector machine [42] and factorization machine [70].

(a) Message passing to root AD. **(b) Changing root increases reusability of messages.**

Figure 3: Work sharing opportunities between queries Q_1 (total count query) and Q_2 with additional predicate applied to S(A,C). Blue edges are reusable messages and red edges are non-reusable edges.

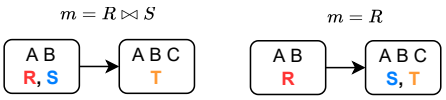


Figure 4: The same JT over relations R(A,B), S(A), T(B,C) can have different relation mappings ($\mathcal{X}$) and each mapping results in different messages (m). For each bag, its attributes are at the top and mapped relations are at the bottom.

on the specific traversal order during message passing, 2) insufficient, because it cannot directly express all filter-group-by queries over the JT, and 3) leaves performance on the table. To do so, we extend the JT data structure as follows:

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

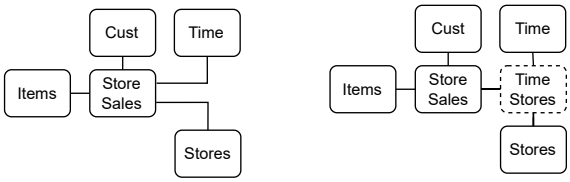
Relation Mapping. $\mathcal{X}(R)$ maps each base relation R to exactly one bag containing R's schema. Although different mappings can lead to different messages (Figure 4), acyclic join graphs have a unique mapping where each relation maps to a single bag[4]. Relations mapped to the same bag are joined during message passing.

Empty Bags. To avoid long paths during message passing it can help to add custom *Empty* bags to create "short cuts". *Empty* bags are not mapped from any relations and 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 [6, 7, 81] focus on non-redundant JT without empty bags in the context of single query optimization.

Example 4 (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.*

Example 5 (JT Data Structure). *Figure 2b illustrates the* JT *data structure for the example in Figure 1. 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 identity relation.*

[4]Heuristics for the general case have been studied by the greedy variable elimination in Probabilistic Graphical Model [43].



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

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

3.3 Message Passing Over Annotated Bags

We now describe support for general SPJA queries over the same join graph. 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 accomodate the bag annotations. These annotations will come in handy when deterimining work sharing opportunities for a new query given a pivot query.

Given the database $\mathbf{R} = \{R_1, R_2, ..., R_n\}$ and JT $= ((E, V), \mathcal{X}, \mathcal{Y})$, we focus on queries of the following form, where any semi-ring 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[5]. Query execution is based on message passing as in Section 2, however the processing at 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 performance implications of different choices when we use the annotated JT to execute new queries.
- `Joined Relations` $\mathcal{J}$. The query may not join all relations in the join graph. For each relation R *not* included in the query, we annotate the corresponding bag $u = \mathcal{X}(R)$ with $\overline{R}$. When computing messages from this bag, R will be excluded from $\mathcal{X}^{-1}(u)$[6].
- `PREDICATES` $\mathcal{P}$. Let predicate $\sigma \in \mathcal{P}$ be over attribute A. If A is not in any relation in $\mathcal{J}$, then 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

[5]Multi-attributes predicates have interesting optimization opportunities [42] but is not our focus. We rewrite them into group-by those attributes followed by the predicate.
[6]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 $\mathcal{X}^{-1}(u) = \{i | \mathcal{X}(i) = u\}$.

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
σ_{id}	Apply selection specified 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.

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 the JT data structure uses bidirectional edges, we call this procedure 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 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 it all messages from child bags are received. During message passing, we temporarily annotate all parent bags 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) = $\sigma\left(\bowtie (M(r) \cup \mathcal{X}^{-1}(r) - \{\overline{R}\})\right)$. To generate the final query results, we marginalize away all attributes not in the query's grouping conditions $\mathcal{G}$.

Example 6. *Consider database and* JT *in Figure 1. 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 witg* σ *(id is omitted). Figure 6 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 worst-case complexity of the message sizes. Since the placement of annotations does not affect message size complexity, the only factor is the choice of root bag. We enumerate every possible root bag, and choose the root with smallest message complexity; the total time complexity is polynomial in the number of bags.

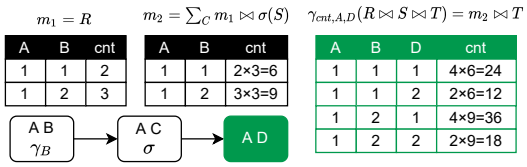


Figure 6: Filter-group-by query with annotated JT.

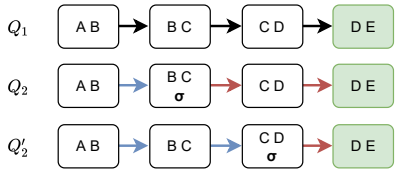


Figure 7: Trade-off between message size and re-usability. Given total count query Q_1, Q_2 further applies predicate to C. Pushing down selection as in Q_2 may reduce message size but hinders re-usability compared to Q_2'.

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 the pivot query's JT that have the same subtree (and annotations).

Proposition 1 (Message Reusability). *Given a* JT *and it annotations for two queries, consider a directed edge* u → v *present in both queries. Let* T_u *be the subtree rooted at* u. *If the bags in* T_u *for both queries have the same annotations, then the message along* u → v *will be identical irrespective of the traversal order nor choice of root.*

This proposition is well established in probabilistic graphical models [72], and follows for message passing over JT. The proof sketch is as follows: leaf nodes send messages that only depend on its outgoing edges, base relations and annotations, while a given bag's outgoing message only depends on its mapped relations in $\mathcal{X}$, annotations and its incoming messages. None of these messages 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 connected subtree closest to root to maximize re-usability. However, pushing selections down trades-offs potentially smaller message sizes for limited reusability:

Example 7. *Suppose we have materialized messages for* Q_1 *in Figure 7, 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 bag. *Calibration* materializes messages along edges in the opposite direction, and thus lets future queries pick arbitrary roots.

3.4.1 Calibration. Given an edge u → 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.

Downward Message Passing. Upward message passing computes messages along half of the edges (from leaves to root). Calibration simply reverses the edges and and runs upward message passing from root (now the leaf) to leaves (now all roots). Now, all edges store materialized messages. Our algorithm extends Shafer–Shenoy inference algorithm [72] 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 [50] can further avoid redundant multiplications. This benefits common aggregation functions such as sum, stddev, and even gram matrix computation for training linear models [71].

Example 8. *Consider the example in Figure 2a. During upward message passing, AD is the root and has received all incoming 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 marginal absorptions.*

Calibration means all bags are ready for absorption. This immediately accelerates the class of queries that adds one grouping or filtering attribute A to the pivot query[7]. We simply pick a bag containing A and apply the filter/group-by to its absorption result.

Generalization of Prior Work. Calibration generalizes the two-pass semi-join reduction in Yannakakis's algorithm [82], which is simply calibration over a 0/1 semi-ring. Similarly, the upward-downward passes to compute local sensitivities [76] is calibration for COUNT over an acyclic join graph.

3.4.2 Query Execution Over a CJT. How do we execute a new SPJA query Q over the CJT of a pivot 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.

[7]These queries correspond to marginal posterior probability (group-by) and incremental update (filter) in PGM [43].

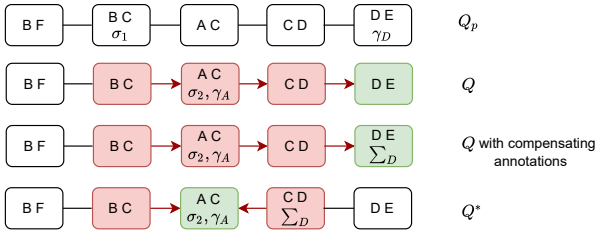


Figure 8: Given CJT with pivot query Q_p, the Steiner Tree to execute Q is highlighted (green is Steiner Tree root and red is Steiner Tree non-root nodes.). Q^* is the optimal query whose Steiner Tree size and run time complexity is minimum.

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 re-used. Thus, we only need to perform upward message passing inside of T. Let us first start with an illustrative example:

Example 9 (Steiner Tree). *In Figure 8, the pivot query Q_p groups by* D *and filters by* B = 1, *and so its annotations are $A_p = \{\sigma_1, \gamma_D\}$. Suppose query* Q *(row 2) instead groups by A and filters by* C = 1, *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. Naively, we can reuse the message* BF → BC, *but otherwise re-run upward message passing along the steiner tree.*

Although the example allows us to re-use one message, it is a sub-optimal execution plan because the steiner tree is not 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 $\mathbf{A}$, 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 key 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 (Q in Figure 8).

Example 10. *The third row in Figure 8 adds the compensating annotation $\sum_D$ to* DE. *Its execution is as follows:* BC *does not apply* B = 1, AC *applies* C = 1 *and groups by* A, *and* DE *maginalizes out* D *and* E.

Shrinking. Given the leaves of the steiner tree, we try to move their annotations towards 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 try to move its annotatinos first. Once a leaf bag does not have any

annotations that differ from the pivot CJT, it is removed from the steiner tree.

Example 11. Q^* *in Figure 8 shows the optimal execution plan over the minimal steiner tree for* Q*. It has moved* $\sum_D$ *to* CD*, and made* AC *the root.* CD *will marginalize out* D*, and* AC *performs the filter and group-by. In this way, we also reuse the message* DE → 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.

4 APPLICATIONS

Semi-ring structures are highly expressive, and also support a wide range of use case. We now describe how to choose good[8] pivot queries and leverage CJT for four popular classes of applications: OLAP, Explanation, Streamning and Data Augmentation for ML. We note that CJT also applies to differential privacy [76], web table analysis [68] and what-if analysis [46], and leave elaborations to future work.

4.1 OLAP Data Cubes

OLAP data cubes [32] 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 [21, 75, 79], approximation [77], partial materialization [35, 78], early projection [44]), 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.

4.1.1 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 unique attributes. Suppose we have calibrated each cuboids with k group-by attributes (the pivot queries). Calibration costs $O(d^k rn)$ 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 $\mathbf{A_h}$, it is executed over a CJT by finding the annotations that differ between the pivot and new query, 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 than 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 C.

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. Note that when $\mathbf{A_{h-k}} \leq 1$, the query complexity is O(1).

4.1.2 OLAP Construction Procedure. Suppose we wish to materialize all cuboids with up to h attributes. Our complexity analysis shows that there is a space-time tradeoff. To minimize the time complexity, we calibrate pivot queries with h − 1 attributes, so that materializing cubodis with h attributes is O(1). However, calibrating pivot queries with fewer attributes reduces build sizes at the expense of larger steiner trees during cuboid computation.

4.1.3 Comparison With Data Cubes. From an algebraic perspective, data cubes exploit the addition operator of the aggregation semiring to marginalize out unwanted attributes. CJT further exploits the multiply operator to join relations, as well as its distributive properties for early marginalization. In terms of complexity, join materialization alone costs $O(n^r)$ under traditional cube construction, which dominates total costs of using CJTs.

4.2 Machine Learning Augmentation

Data and feature augmentation [16] is used to 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 lakes and markets [16, 24, 25]. However, the major bottleneck is the cost of joining each augmentation dataset and then retraining the ML model.

The state-of-the-art approach uses factorized learning [20, 61, 70, 71] to avoid join materialization when training models over join graphs. First, it designs semi-ring structures for common ML models (linear regression [71], factorization machines [69], k-means [20]),

[8] In general, finding the optimal pivot query and JT structure is NP-hard.

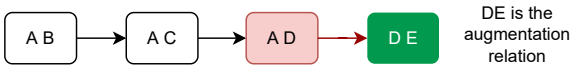


Figure 9: Augmenting the join graph with relation DE only requires one message from AD to DE.

and then performs early marginalization by pushing the aggregation (training) through the join. This is equivalent to upward message passing through the join graph. If we augment with relation r, then factorized learning approaches executes message passing through the augmented join graph again.

In contrast, the CJT allows us to choose any bag b that contains the join keys, construct an edge b → r, and perform calibration 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 9 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 B.

4.3 Additional Optimizations

A benefit of the CJT data structure is that it composes with other optimizations in query processing. Below we outline two extensions as examples: incremental maintenance and lazy calibration.

Incremental Maintenance. We may wish to maintain the CJT data structure as the base relations are updated. For instance, stream processing regularly appends records to base relations [5], while query explanation intervenes on aggregation queries by deletes records from base relations [67, 80]. We can directly apply Factorized-IVM [61] to maintain the CJT—a base relation accumulates delta records, and we periodically execute upward and downward message passing using the delta records to update the messages. If the delta records are deletions (as in query explanation), then the aggregation must be a ring that supports the minus operator.

Lazy Calibration. Calibration is the dominant cost of using CJTs, particularly if we wish to maintain CJTs under updates, because each base relation update will invalidate half of the messages and they need to be re-calibrated. In the spirit of lazy view maintenance [18, 85], we implement lazy calibration. When a relation is updated, its corresponding bag sends an invalitation message. A query over the CJT checks validity of the bags in its steiner tree, and recalibrates the invalid bags by sending messages from the updated base relation to the invalid bag. There is an interesting trade-off between the frequency of eager calibration, and recalibration during query processing. The experiment in Section 5.2.4 shows that lazy calibration can improve performance by 2000x for write-heavy workloads.

There are other optimizations that are related to the design of the underlying JT structure for cyclic join graph. First, while previous JT algorithm breaks cycle by clustering relations into single bag [7], we propose to alternatively create empty bag to break cycle. This alternative may increase space complexity but can reduce update latency through Factorized-IVM, which is critical for Stream Query Processing [5]. We have successfully applied this optimization to improves latency by > 100x as discussed in Appendix D. Second, for queries with group-by over join keys, we can rewrite them as selections on each join key values to break cycle. We apply this optimization to eliminate cycle for TPC-H Q5 as discussed in Appendix F.

5 EXPERIMENTS

Does CJT really accelerate OLAP, intervention, and ML augmentation applications? What are the trade-offs between CJT size and query latency benefits for OLAP queries? Is lazy calibration beneficial? We have implemented three versions of CJT: as a custom C++ query engine that uses worst-case optimal join [31], as a middleware compiler for cloud databases (Redshift), a compiler to Pandas data frame operations. We use them to study the above questions.

5.1 Single-node Custom-engine Experiments

We first study the benefits of worksharing and IVM by comparing our custom CJT engine with LMFAO [70], the state-of-the-art factorized query compiler using the applications in Section 4 (OLAP, intervention, ML augmentation).

Setup. We compare CJT, LMFAO [70], and a variation of our implementation that doesn't perform calibration (JT-IVM). The latter uses a similar query execution algorithm as LMFAO (message passing) but also applies incremental view maintenance.

We use the IMDB [1] movie dataset (Figure 10), a popular benchmark dataset in query optimization [48, 53, 54]. Following prior work [6, 65], we preprocess the dataset to dictionary encode strings into 32-bit unsigned integers. The resulting data is 1.2GB over 11 relations, and the `Cast Info` relation dominates (~1GB). For space reasons, results on 3 additional datasets (Favorita [2], Lego [3], and TPC-DS [57]) can be found in [4].

We run the systems single threaded on a GCP n1-standard-16 VM, running Debian 10, Xeon 2.20GHz CPU, and 60GB RAM. We exclude the time of reading files from disk, and exclude compilation time for LMFAO. All experiments fit and run in memory.

Workloads. Our goal is to study the algorithmic benefits of work sharing, so count aggregation queries are sufficient. Our experiment will calibrate the total count query (count with no grouping attributes) as the pivot query, and execute representative queries for each application class. For OLAP, we execute two group-by queries

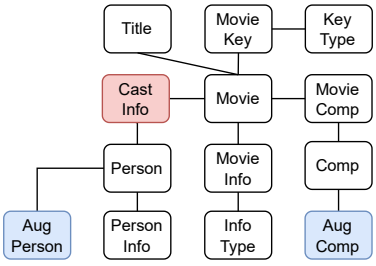


Figure 10: IMDB schema. `CastInfo` is the largest relation. `Aug Person` and `Aug Comp` are augmented relations.

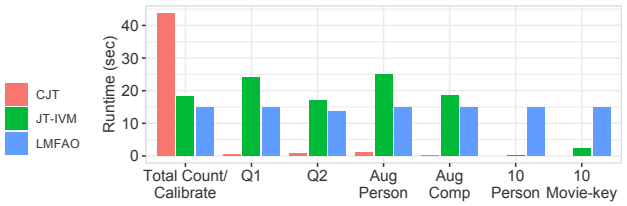


Figure 11: Run time of different workloads on IMDB dataset (log scale).

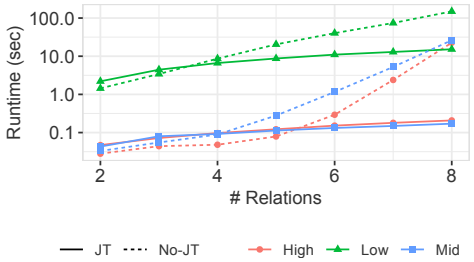


Figure 12: Run time of total count query with/without message passing over JT in seconds (log scale). High, Mid, and Low are for different fanouts. Msg is message passing.

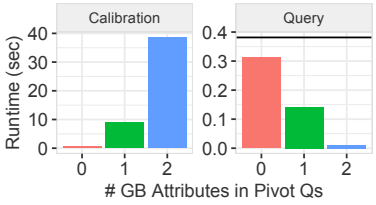


Figure 13: (Left) Calibration time for all pivot queries. (Right) Average query time for 4-attr OLAP query given the number of group-by attributes in calibrated pivot queries (x-axis), horizontal line is cost without any cuboids.

$\mathbf{Q_1} = \gamma_{\text{COUNT}(\cdot),\text{Person}}(\bowtie)$, and $\mathbf{Q_2} = \gamma_{\text{COUNT}(\cdot),\text{Movie}}\sigma_{\text{Company}}(\bowtie)$ over the join graph $\bowtie$. For interventions, we remove 10 records from the `Person` (`Person-10`) or `MovieKey` (`MovieKey-10`) relations and refresh the pivot query result. For augmentation, we create two-attribute augmentation relations where the first attribute is the primary key of `Person` (`Aug Person`) or `Company` (`Aug Comp`) and the second is a random integer, and refresh the pivot query result.

Results. Figure 11 reports the calibration time along with the query times. As expected, calibration is >2× LMFAO due to the downward message pass and write costs, however is run once offline.

`CJT` executes the OLAP queries ~30× faster than LMFAO because it avoids messages that span the large `Cast Info` relation. Similarly, augmentation ~1s vs LMFAO's ~15s due to message reuse. Augmenting `Person` takes ~4× longer than augmenting `Company` because the `Person` bag is 3.1× larger (18MB vs 5.8MB). `JT-IVM` is 1.2~1.7× slower than LMFAO, we suspect due to implementation differences since they are algorithmically identical.

`CJT` accelerates intervention queries over LMFAO by $>10^5\times$ because the steiner tree is simply the intervened relation/bag, and no messages are passed. `JT-IVM` naturally out-performs LMFAO because it applies factorized IVM, however it still needs to send messages across the join graph, and is dominated by message sizes (`Persons-10` is ~128× faster than `MovieKeys-10`). This highlights the fact that IVM relies on low fan-out paths to achieve performance, but does not improve the worst-case complexity.

5.2 SQL Compiler Cloud Experiments

We now evaluate `CJT` (the compiler middleware) on AWS Redshift using OLAP and intervention queries. `CJT` is initialized with the join graph, and translates the pivot query into a set of `CREATE TABLE` statements—after picking a random root, each message during upward and downward message passing instead generates a query with the corresponding join, filter, aggregation logic. Each query over the `CJT` is similarly translated.

Setup. We use three datasets: TPC-H (SF=10), TPC-DS (SF=1), and `synthetic`. Following prior `JT` work [81], `synthetic` 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 is 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.

We used a AWS redshift cluster (one dc2.large node, 2 vCPU, 15 GB memory, 0.16TB SSD, 0.60 GB/s I/O). All experiments warm the cache by pre-executing queries until runtime stablizes.

5.2.1 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 no messages are materialized. We execute the total count query as a large join-aggregation query (`-Naive`), and using message passing (`-JT`).

Figure 12 varies the number of relations (x-axis) and fanout (line style). 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. Note that the x-axis is also interpretable as the steiner tree size.

5.2.2 Cubes in the Cloud. `CJT`s help developers build data cubes to explicitly trade-off build costs and query performance. To evaluate this, we use the synthetic dataset and with f = 2 (low) 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.

Figure 13 (left) reports the calibration cost, which increases exponentially. However, 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 12). At the same time, Figure 13 (right) shows

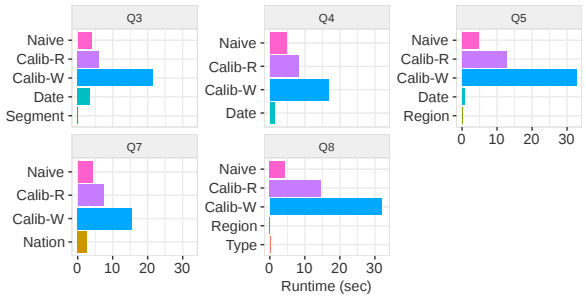


Figure 14: Run time for TPC-H queries. `Naive` is for original queries without message passing. `Calib-R` is for computing all messages without materializing them. `Calib-W` is for computing and materializing messages.

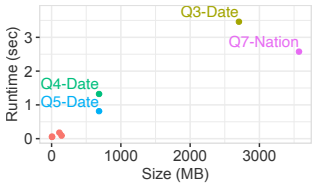


Figure 15: Size of annotated bag (by predicate) vs query runtime.

that increasing k significantly reduces the OLAP query execution costs due to the smaller steiner tree.

5.2.3 TPC-H. TPC-H queries contain parameterized queries that can benefit from `CJT`. We use a subset of TPC-H queries (Q3-5,8) that can be re-written as acyclic queries (see Appendix F). For each query, we calibrate the query with random parameter values, and then vary the parameters one at a time.

Figure 14 reports the calibration and query execution costs. `Naive` simply runs the query on Redshift. For `CJT`, we report calibration cost (`Calib-R`) separately from the write cost (`Calib-W`), since writes on Redshift are particularly expensive. The remaining bars correspond to varying their corresponding predicate's parameter value.

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.

Note that prior factorized databases work [62, 70] is equivalent to upward message passing, and expected to be 2× faster than calibration alone (`Calib-R`). However, we can see that factorized execution alone does not execute the TPC-H queries tangibly faster than `Naive`. This is because factorization is optimized for many-to-many joins, however TPC-H is dominated by one-to-many joins, so the benefits are minimal.

In contrast, `CJT` accelerates TPC-H queries by nearly 10^3× over `Naive`. Naturally, the speedup depends linearly on the size of the bag that contains the parameterized attribute (Figure 15).

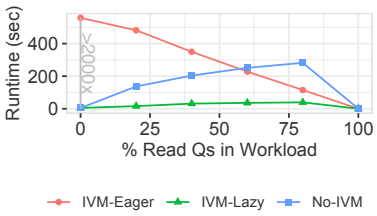


Figure 16: Runtime for read-write workload (100 Qs).

5.2.4 Lazy Calibration. We now use TPC-DS (SF=1) on a read-write workload (100 queries) to showcase the trade-offs of lazy calibration (`Lazy-IVM`) compared to lazy re-evaluation (`No-IVM`) or eager maintenance (`Eager-IVM`). The database contains 9 relations each write query inserts a random row into the `Store Sales` relation (largest relation, 181MB), and each read query randomly chooses one of the other relations (1 – 42MB) and computes count grouped by its primary key.

Figure 16 reports total workload cost with varying read-write mixtures. `Eager-IVM` linearly improves as it contains fewer writes, while both lazy approaches increase in cost with read %, however `No-IVM` is more expensive due to full message sizes rather than deltas. The costs are near-zero at the extremes since there are no reads (no message passing) or no writes (no maintenance needed). Eager calibration without IVM is not plotted because it exceeds 1hr at 20% writes.

5.3 Pandas Dataframe Compiler

Python and Pandas is one of the dominant programming environments for ML. Thus we evaluate the Pandas [55] dataframe compiler for an ML data augmentation task using linear regression [71]. The compiler takes the join graph and relations (as Pandas dataframes) as input, and then translates calibration, model training, model evaluation and data augmentation into Pandas merge (join), group-by aggregation and array operations.

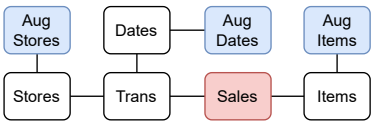


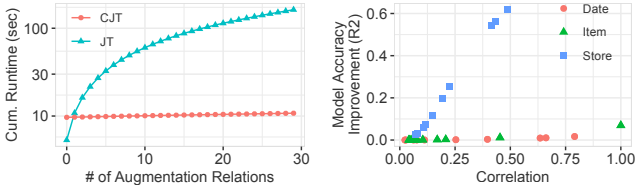
Figure 17: Favorita schema. `Sales` is the largest relation. `Aug Stores`, `Aug Dates` and `Aug Items` are augmented relations.

Setup. We use the Favorita [2] dataset of purchasing and sales forecasts, used in prior factorized learning work [70, 71] (schema in Figure 17). `Sales` is the largest relation (241MB), and the others are < 2MB. The model uses (`Sales`.unit_sales, `Stores`.type, `Items`.perishable) as features, and `Trans`.transactions (number of transactions for each store, date) as the target Y variable.

To simulate a data lake with augmentation data of varying effectiveness, we generate synthetic data to join with (augment) `Dates`, `Stores`, and `Items`. For each relation R (e.g., `Dates`), we first generate a predictive feature $\hat{Y}$ as the average of Y grouped by R's primary key. Then we create 10 augmentation relations with schema

(key, val), where key is the join attribute and val varies in correlation $\hat{Y}$ [39]. The correlation coefficient φ is drawn from the inverse exponential distribution: min(1, 1/Exp(10)), and the values are the weighed average between $\hat{Y}$ and a random variable, weighed by φ.

We individually update the model for each of the 30 augmentation relation, and measure the cumulative runtimes and model accuracy (R2). We used the GCP machine in Section 5.1.



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

Figure 18: Augmentation run time and model performance.

Results. Figure 18a reports the cumulative costs to augment and retrain the the model. Using factorized learning (JT) takes >1.5 min whereas CJT takes ~10sec—the cost is dominated by calibration, which is about twice the cost of training a single model (because join costs dominate). Figure 18b reports the model accuracy improvement above the baseline accuracy of 0.031 after each augmentation. There is a wide discreptancy in accuracy improvements—from ~0 to 0.61. CJT accelerates the cost of search the corpus of datasets by over 2 orders of magnitude; after calibration, CJTs evaluated all 30 augmentation relations in less than 1 sec.

6 RELATED WORK

Wide Table and Factorized Representation. The wide table in-memory columnar engine [52] shows that denormalized query execution can speed TPC-H queries by up to 10×. However, it is limited to in-memory relations, and denormalized relations can be too large to fit in the memory. Kumar et al. [45] use user-defined aggregate functions to push ML computations over join and show improvement in run time and storage. However, their algorithm only improves the space but not time complexity, and performs *worse* than the naive approach when the data fits in the memory.

Early Marginalization. Early Marginalization is first introduced by Gupta et al. [34] as a generalized projection that supports popular queries like count, sum, max, etc. This algorithmic optimization has been further exploited by factorized databases to store relational tables compactly [62] and execute semi-ring aggregation queries efficiently [37, 70]. Abo et al. [7] generalize early marginalization and establish the eqivalence between early marginalization and variable elimination in Probabilistic Grpahical Model [43]. CJT is based on early marginalization and further exploits the work sharing opportunities. Queries over CJT has a better run time complexity compared to these works as analyzed in Appendix A.

Calibrated Junction Tree. Calibration Junction Tree is first proposed by Shafer and Shenoy [72] to efficient compute inference over probabilistic graphical models. Calibration Junction Tree has been widely used across different areas engineering [66, 86], ML [14, 22], and medicine [47, 64], but they are limited to probabilistic tables. Yannakakis's algorithm [82] applies two-pass semi-join reduction to relations, but is limited to 0/1 semi-ring. We generalize the idea to semi-ring aggregation and extend it to support SPJA queries. There have been many hard-ware optimizations [83, 84] that utilize GPU and SIMD to accelerate message passing for probabilistic inference; these optimizations also apply to CJT and we leave them as future works.

Materialized View. Previous materialized view works [12, 51] focus on the optimization of view selections given workloads; these works have significant overhead and rely on heuristics to solve the intractable optimization. CJT is lightweight and has the same complexity of pivot query. CJT is closely related to and generalizes data cube [32] as it materializes views of the full join result; as shown in Section 5.2, CJT is much more efficient than data cube. Recent higher-order IVM [8] and factorized IVM [61] exploit the semi-ring structure to optimize IVM. CJT benefits from IVM as shown in Section 5.1.

Machine Learning Systems. Traditional machine learning systems either only accept a single relation as input [29, 36] and require the materialization of full join result. Structure-aware machine learning systems [70, 71] apply early marginalization to train model over join result efficiently. CJT contributes to Machine Learning Systems in two ways: (1). CJT supports efficient machine learning augmentation. (2). CJT shares computations between tree-based models like Random Forest by re-using messages from low-dimensional cuboids discussed in Section 4.1.

7 CONCLUSIONS

This work presented CJT, a novel data structure based on calibration, that materializes and manages messages in a join graph. New queries are able to re-use messages in the CJT to accelerate execution costs by orders of magnitude as compared to factorized query execution. To do so, we developed an annotation-based analysis to identify the messages that can be reused, and the optimal execution order for the new query. We showed how CJTs accelerates OLAP workloads, query explanation, streaming data, and data augmentation for ML; and benefits from incremental view maintenance and lazy calibration.

We evaluated three implementations of CJT—a single-node query engine and middleware compilers to SQL and Pandas dataframe operations—on local and cloud databases. Our single-node engine accelerates OLAP queries by ~30× and intervention queries by over 10^5× over the state-of-the-art factorized engine LMFAO; our cloud compiler accelerates TPC-H queries on Redshift by up to 10^3× compared to factorized execution; and our Pandas compiler accelerates data augmentation for ML by >100×.

REFERENCES

[1] Imdb. https://www.imdb.com/interfaces/, 05 2013.
[2] Corporación favorita grocery sales forecasting. https://www.kaggle.com/c/favorita-grocery-sales-forecasting, 10 2017.
[3] Lego database. https://www.kaggle.com/rtatman/lego-database, 07 2017.
[4] Calibration: A simple trick for fast analytics over joins (technical report). 2021.
[5] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[6] 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.

[7] 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.

[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] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

[11] A. S. Arun, S. V. M. Jayaraman, C. Ré, and A. Rudra. Hypertree decompositions revisited for pgms. *arXiv preprint arXiv:1804.01640*, 2018.

[12] 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.

[13] 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.

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

[15] C. Carroll. Modeling marketing attribution. https://blog.getdbt.com/modeling-marketing-attribution/, 2020.

[16] 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.

[17] @codinglay. Is it just me or are joins the most confusing concept to understand in sql??? https://twitter.com/codinglay/status/1501594773685166083, 2022.

[18] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 469–480, 1996.

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

[20] 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.

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

[22] 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.

[23] T. Eden, A. Levi, D. Ron, and C. Seshadhri. Approximately counting triangles in sublinear time. *SIAM Journal on Computing*, 46(5):1603–1646, 2017.

[24] 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.

[25] 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.

[26] 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.

[27] D. Fisher, S. M. Drucker, and A. C. König. Exploratory visualization involving incremental, approximate database queries and uncertainty. *IEEE computer graphics and applications*, 32(4):55–62, 2012.

[28] M. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann. Adopting worst-case optimal joins in relational database systems. *Proceedings of the VLDB Endowment*, 13(12):1891–1904, 2020.

[29] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *2011 IEEE 27th International Conference on Data Engineering*, pages 231–242. IEEE, 2011.

[30] H. Gonzalez, A. Y. Halevy, C. S. Jensen, A. Langen, J. Madhavan, R. Shapley, W. Shen, and J. Goldberg-Kidon. Google fusion tables: web-centered data management and collaboration. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1061–1066, 2010.

[31] G. Gottlob, Z. Miklós, and T. Schwentick. Generalized hypertree decompositions: Np-hardness and tractable variants. *Journal of the ACM (JACM)*, 56(6):1–32, 2009.

[32] 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.

[33] 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.

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

[35] 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.

[36] J. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The madlib analytics library or mad skills, the sql. *arXiv preprint arXiv:1208.4165*, 2012.

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

[38] 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.

[39] 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.

[40] A. Kara, H. Q. Ngo, M. Nikolic, D. Olteanu, and H. Zhang. Counting triangles under updates in worst-case optimal time. *arXiv preprint arXiv:1804.02780*, 2018.

[41] R. Kearns, S. Shead, and A. Fekete. A teaching system for sql. In *Proceedings of the 2nd Australasian conference on Computer science education*, pages 224–231, 1997.

[42] 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.

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

[44] 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.

[45] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1969–1984, 2015.

[46] L. V. Lakshmanan, A. Russakovsky, and V. Sashikanth. What-if olap queries with changing dimensions. In *2008 IEEE 24th International Conference on Data Engineering*, pages 1334–1336. IEEE, 2008.

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

[48] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[49] 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.

[50] 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.

[51] 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.

[52] Y. Li and J. M. Patel. Widetable: An accelerator for analytical data processing. *Proceedings of the VLDB Endowment*, 7(10):907–918, 2014.

[53] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Learning to steer query optimizers. *arXiv preprint arXiv:2004.03814*, 2020.

[54] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711*, 2019.

[55] W. McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[56] D. Miedema, E. Aivaloglou, and G. Fletcher. Identifying sql misconceptions of novices: findings from a think-aloud study. *ACM Inroads*, 13(1):52–65, 2022.

[57] R. O. Nambiar and M. Poess. The making of tpc-ds. In *VLDB*, volume 6, pages 1049–1058, 2006.

[58] P. Navid. Modern data warehouse modelling: The definitive guide - part 2. https://hightouch.io/blog/data-warehouse-modelling-part-2/, 2021.

[59] 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.

[60] 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.

[61] 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.

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

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

1982.

[64] 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.

[65] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, et al. Db2 with blu acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment*, 6(11):1080–1091, 2013.

[66] 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.

[67] S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1579–1590, 2014.

[68] A. Santos, A. Bessa, F. Chirigati, C. Musco, and J. Freire. Correlation sketches for approximate join-correlation queries. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1531–1544, 2021.

[69] 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.

[70] 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.

[71] 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.

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

[73] M. Staniak and P. Biecek. The landscape of r packages for automated exploratory data analysis. *arXiv preprint arXiv:1904.02101*, 2019.

[74] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614, 2011.

[75] 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.

[76] Y. Tao, X. He, A. Machanavajjhala, and S. Roy. Computing local sensitivities of counting queries with joins. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 479–494, 2020.

[77] 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.

[78] 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.

[79] 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.

[80] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. 2013.

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

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

[83] 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.

[84] 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.

[85] J. Zhou, P.-A. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *Proceedings of the 33rd international conference on Very large data bases*, pages 231–242, 2007.

[86] 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.

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
```

A COMPLEXITY OF CALIBRATED JUNCTION HYPERTREE

In this section, we study the complexity of Calibrated Junction Hypertree. We start with the background of Fractional Hypertree Width, which is the complexity of semiring aggregation query and factorized representation [7, 37, 62]. Then, we study the work sharing opportunities in Calibrated Junction Hypertree and propose fractional sub-hyertree width as the complexity of queries over Calibrated Junction Hypertree.

A.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 [13]). 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 12 (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 [60] 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 [7, 37] 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 $\text{fhtw}((\mathcal{T}, \mathcal{X}, \mathcal{Y}, \mathcal{Z}))$ 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 $\text{fhtw}((\mathcal{E}, \mathcal{V}))$ is the minimum of the fractional hypertree widths over all possible Junction Hypertrees. Finding the minimum fractional hypertree width is known to be NP-hard [26].

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 13 (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* $O(n^2)$.

Indicator Projection. Indicator projection has been introduced [7] to remove redundant tuples inside message and achieve Fractional Hypertree Width for semiring aggregation. The detailed algorithm of Indicator Projection is in Algorithm 2. Basically, besides relations needed to join for message, we need to further add relations whose schema intersects with join result's to remove redundant tuples. For worst case optimal join, adding more relations to join will make the linear program result smaller as long as their shcema has been covered by other relations.

While Indicator Projection is effective to guarantee that semiring aggregation is bounded by Fractional Hypertree Width, it aggressively eliminates all dangling tuples which might be useful for future queries and reduces the reusability of messages. We introduce Static Calibrated Junction Hypertree, which imposes constraints on the use cases to ensure that Indicator Projection could be safely Indicator Projection while ensure the work sharing opportunities of messages. For use case goes beyond Static Calibrated Junction Hypertree, Indicator Projection is not recommended.

Static Calibrated Junction Hypertree. Calibrated Junction Hypertree is static if the set of relations $\mathbf{R} = \{R_1, R_2 ..., R_n\}$ stored is unchanged, tuples inside relations won't be updated, and queries over only a subset of relations is disallowed. Therefore, all queries over Static Calibrated Junction Hypertree are over the join of a fixed set of relations $R_1 \bowtie R_2 ... \bowtie R_n$, and dangling tuples eliminated by Indicator Projection will never be used

Besides limiting work sharing opportunities of messages, Indicator Projection also doesn't improve performance empirically because of the large overhead of projection [11]. Because of these limitations, we will only use Indicator Projection for theoretic analysis and we assume that we use worst case optimal join without Indicator Projection to compute message in this paper unless otherwise specified.

A.2 Fractional Sub-Hypertree Width

We start with Junction Hypertree whose bags are dangling tuple free. This assumption is made to ensure that message passing inside Junction Hypertree is always bounded by its Fractional Hypertree Width even without Indicator Projection, so that we can keep focus on the work sharing opportunities inside of Junction Hypertree. We will relax this assumption and discuss general cases in next section.

Dangling Tuple Free. Calibrated Junction Hypertree $(\mathcal{T}, \mathcal{X}, \mathcal{Y}, \mathcal{Z})$ over a set of relations $\mathbf{R} = \{R_1, R_2 ..., R_n\}$ is dangling tuple free if it doens't contain empty bag and, for each bag $v \in \mathcal{V}_\mathcal{T}$, the join of relations in v doesn't contain dangling tuples: $\bowtie_{R \in \mathcal{Y}^{-1}(v)} R = \pi_{\mathcal{X}(v)}(R_1 \bowtie R_2 ... \bowtie R_n)$ under 0/1 semiring. When Calibrated Junction Hypertree is dangling tuple free, we don't have to apply Indicator Projection and can still achieve Fractional Hypertree Width for calibration.

Next, we study the work sharing opportunities of Calibrated Junction Hypertree. Note that we are discussing more general cases than before: in Section 3 we discuss message re-use between different queries in the same CJT; here we discuss message re-use between two different queries from different CJT. Given Calibrated Junction Hypertree φ^*, and aggregation query Q over the same semiring, we first consider (not calibrated) Junction Hypertree φ in which we can execute query Q through message passing to root r. We assume that both φ^* and φ are dangling tuple free. Without φ^*, the message passing algorithm takes $\text{ftw}(\varphi)$, whose bottleneck is the maximum of the fractional edge cover of bags in φ: $\max_{V \in \mathcal{V}_\mathcal{T}} \rho^*(\mathcal{Y}^{-1}(v))$.

Now let's consider how we can reuse messages from φ^* for φ. Unlike previous works of junction tree [43] where the underlying probabilistic graphical model is assumed to be fixed, we consider cases when we have different join graphs but share some join relations. At a high level, we want to find part of sub-Hypertrees of φ^* and φ that are equivalent, such that their messages are the same and we can directly reuse messages. We start by defining the equivalence of Junction Hypertree, and study when the messages can be reused.

Definition 1 (Equivalence of bags.). *Bag* u *and* v *(may from different Junction Hypertree) are equivalent iff* $\mathcal{Y}^{-1}(u) = \mathcal{Y}^{-1}(v)$. *That is, the set of relations mapped to them are the same.*

We use the equivalence of bags to define isomorphism of Junction Hypertree.

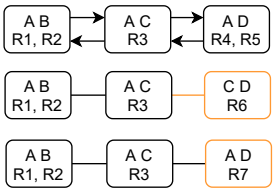


Figure 19: Message Reuse Example

Definition 2 (Isomorphism of Junction Hypertree). *Given two Junction Hypertrees* $(\mathcal{T}_1, \mathcal{X}_1, \mathcal{Y}_1, \mathcal{Z}_1)$ *and* $(\mathcal{T}_2, \mathcal{X}_2, \mathcal{Y}_2, \mathcal{Z}_2)$*, they are isomorphic if there exists a bijection between the vertex* $f : \mathcal{V}_{\mathcal{T}_1} \to \mathcal{V}_{\mathcal{T}_2}$ *such that bags are equivalent:* $\forall u \in \mathcal{V}_{\mathcal{T}_1}, u = f(u)$*, the edges are equivalent:* $\forall u, v \in \mathcal{V}_{\mathcal{T}_1}, [u, v] \in \mathcal{E}_{\mathcal{T}_1} \Leftrightarrow [f(u), f(v)] \in \mathcal{E}_{\mathcal{T}_2}$*, and the attributes of bags are equivalent* $\mathcal{X}_1 = \mathcal{X}_2 \circ f$.

Definition 3 (Junction sub-Hypertree.). *Given Junction Hypertree* $\varphi_1 = (\mathcal{T}_1, \mathcal{X}_1, \mathcal{Y}_1, \mathcal{Z}_1)$ *which is symmetric directed and unrooted and* $\varphi_2 = (\mathcal{T}_2, \mathcal{X}_2, \mathcal{Y}_2, \mathcal{Z}_2)$ *with root* $r_2 \in \mathcal{V}_{\mathcal{T}_2}$*,* φ_2 *is Junction sub-Hypertree of* φ_1 *if there exists root in* φ_1 *such that* $\mathcal{T}_2$ *rooted in* r_2 *is a subtree of* $\mathcal{T}_1$ *rooted in* r_1 *(*r_2 *is a node in* r_1 *and* $\mathcal{T}_2$ *contains all* r_2*'s descendants in* r_1*), and* $\mathcal{X}_2, \mathcal{Y}_2, \mathcal{Z}_2$ *are equivalent to their counterparts but their domains are restricted to bags and edges in* $\mathcal{T}_2$.

We can use the isomorphism of Junction Hypertree to share the messages between φ^* and φ:

Proposition 2. *Given Calibrated Junction Hypertree* φ^**, and Junction Hypertree* φ *which we want to perform message passing to root r, let u and v be bags in* φ *where u is the parent of v. Then, the message from v to u could be reused from* φ^* *iff:*

(1). the Junction Sub-Hypertree φ_{sub} *of* φ *rooted in v is isomorphic to some Junction Sub-Hypertree* φ^*_{sub} *of* φ^* *rooted in* r^*.

(2). the intersection of schema between v and u is equivalent to the intersection of schema between r^* *and parent* p^* *of* r^* *in* φ^**:* $\mathcal{X}(v) \cap \mathcal{X}(u) = \mathcal{X}(r^*) \cap \mathcal{X}(p^*)$

The first condition ensures that the join result is the same, and the second condition ensures that the marginalization attributes are the same. Therefore, the message between u and v is the same as the message between r^* and p^*: $\mathcal{Z}([v, u]) = \mathcal{Z}([r^*, p^*])$.

Example 14 (Message Reuse). *Consider three Junction Hypertrees in Figure 19, where the first one is calibrated and the rest two are not. For all three Junction Hypertree, the subtrees AB-AC are isomorphic. For the third Junction Hypertree, the message from AC to AD can be reused. For the second Junction Hypertree, the message from AC to CD can't be reused as the second condition is not satisfied (first Junction Hypertree has intersection {A}, while second has intersection {C}).*

Fractional Sub-Hypertree Width Finally, we study the time complexity of message passing in φ given the message sharing opportunities. Because of the simplicity of tree structure, we can check which message could be reused from φ^* efficiently and there are only messages linear to the number of relations to check. We assume that the number of relations is much smaller than the number of rows in relations. Then, the bottleneck of message passing in φ is the largest bag which can't share computations with φ^*.

Let $\texttt{SharedMessages}(\varphi^*, \varphi, r)$ be the set of messages from leaves to root r in φ that could be reused from φ^* and let $\texttt{SubHypertree}(\varphi^*, \varphi, r)$ be the set of bags in φ where there exists some message from this bag not in $\texttt{SharedMessages}(\varphi^*, \varphi, r)$ plus the root of message passing for absorption. The bags in $\texttt{SubHypertree}(\varphi^*, \varphi, r)$ form a connected tree because if message could be shared between node v to u, so can messages between any descendants of v. We need to perform join and marginalization to bags in $\texttt{SubHypertree}(\varphi^*, \varphi, r)$ because some of their messages can't reused. The Fractional Sub-Hypertree Width of φ rooted in r with respect to φ^* is then the maximum of Fractional Edge Cover placed on these bags: $\text{fhstw}_{\varphi^*}(\varphi, r) = \max_{t \in \texttt{SubHypertree}(\varphi^*, \varphi, r)} \rho^*_t$.

Example 15 (Fractional Sub-Hypertree Width). *Continue with the example in Figure 19. Given the first Calibrated Junction Hypertree* φ^**, for the second Junction Hypertree, its Fractional Sub-Hypertree Width* $\text{fhstw}_{\varphi^*}(\varphi, r)$ *is the maximum of Fractional Edge Cover placed on bag AC and CD since the message from AC to CD can't be reused and we have to join all relations in AC. For the third Junction Hypertree, its Fractional Sub-Hypertree Width* $\text{fhstw}_{\varphi^*}(\varphi, r)$ *is only the Fractional Edge Cover placed on bag AD.*

We note that $\text{fhstw}_{\varphi^*}(\varphi, r) \leq \text{fhtw}_{\varphi}(\varphi)$ because $\texttt{SubHypertree}(\varphi^*, \varphi, r)$ is a subset of bags in φ. The gap between $\text{fhstw}_{\varphi^*}(\varphi, r)$ and $\text{fhtw}_{\varphi}(\varphi)$ could be as large as the Fractional Edge Cover of bags in φ.

A.3 Fractional Sub-Hypergraph Sub-Hypertree Width

We consider the general case where Junction Hypertree is not dangling tuple free and messages between bags are computed without Indicator Projection. We first define Fractional Sub-Hypergraph Hypertree Width, which quantifies the time complexity of calibration for Junction Hypertree without Indicator Projection. Then, we define Fractional Sub-Hypergraph Sub-Hypertree Width, which further exploits the work sharing opportunities.

Fractional Sub-Hypergraph Hypertree Width. Previously, we use Fractional Hypertree Width to study the time complexity of calibration for Junction Hypertree, where Indicator Projection is used to quantify the effect of relations in whole hypergraph. As discussed before, Indicator Projection is not practical and limits the ability of work sharing. Therefore, we consider the time complexity of calibration without Indicator Projection. The main extensions are: (1). consider the join inside a sub-hypergraph instead of whole hypergraph (2). consider the join for each message instead of bag.

Fractional Hypertree Width is defined as the maximum Fractional Edge Cover of bags because join is the most expensive part and all messages sent from each bag has size smaller than the join result (as they are marginalization over join result). However, without marginalization, different messages from the same bag may incur different join sizes, and we need to consider the join separately for different messages.

Given Calibrated Junction Hypertree $(\mathcal{T}, \mathcal{X}, \mathcal{Y}, \mathcal{Z})$ of hypergraph $(\mathcal{E}, \mathcal{V})$ and directed edge $[v, u] \in \mathcal{E}_{\mathcal{T}}$, we define $\texttt{SubHypergraph}(\varphi, [v, u])$ to be the hypergraph of relations in the bags of subtree $\mathcal{T}_{sub}$ of $\mathcal{T}$ where $\mathcal{T}$ has root u and $\mathcal{T}_{sub}$ has root v. $\texttt{SubHypergraph}(\varphi, [v, u])$ is the hypergraph of relations involved with the join when computing messages from v to u. Then, Fractional Sub-Hypergraph

Hypertree Width of Junction Hypertree is computing the maximum Fractional Edge Cover of edges over its SubHypergraph: $\mathrm{fsghtw}(\mathcal{T},\mathcal{X},\mathcal{Y},\mathcal{Z}) = \max_{e\in\mathcal{E}_\mathcal{T}} \rho^*_e$ where ρ^*_e is the Fractional Edge Cover of each bag over $(\mathcal{E}_{\mathrm{sub}},\mathcal{V}_{\mathrm{sub}}) = \mathtt{SubHypergraph}(\varphi, e)$:

$$\begin{aligned} \min \sum_{E\in\mathcal{E}_{\mathrm{sub}}} & \log_2(|R_E|)x_E \\ \text{s.t.} \sum_{E:V\in E} & x_E \geq 1, \forall V \in \mathcal{X}(u) \\ & x_E \geq 0, \forall E \in \mathcal{E}_{\mathrm{sub}} \end{aligned} \tag{5}$$

Given Hypergraph $(\mathcal{E},\mathcal{V})$, Fractional Sub-Hypergraph Sub-Hypertree Width $\mathrm{fsghtw}((\mathcal{E},\mathcal{V}))$ is the minimum of the Sub-Hypergraph Sub-Hypertree Width over all possible Junction Hypertrees.

Fractional Sub-Hypergraph Sub-Hypertree Width. Equipped with Fractional Sub-Hypergraph Hypertree Width, we study the time complexity of message passing in $\varphi = (\mathcal{T},\mathcal{X},\mathcal{Y},\mathcal{Z})$ to root r given Calibrated Junction Hypertree φ^*. Let $\mathcal{E}^r_\mathcal{T}$ be a subset of directed edges in φ when the root is $r \in \mathcal{V}_\mathcal{T}$. The messages we need to compute for message passing to r are: $\mathcal{E}^r_\mathcal{T}$–$\mathtt{SharedMessages}(\varphi^*,\varphi,r)$. Then, the Fractional Sub-Hypergraph Sub-Hypertree Width of φ rooted in r with respect to φ^* is the maximum Fractional Edge Cover placed on these bags over sub-hypergraph for all edges e plus the Fractional Edge Cover of root
$\mathrm{fsghstw}_{\varphi^*}(\varphi, r) = \max(\max_{e\in\mathcal{E}^r_\mathcal{T}-\mathtt{SharedMessages}(\varphi^*,\varphi,r)}\rho^*_e, \rho^*_{\mathrm{root}})$.

Finally, given a semiring aggregation query Q, the Fractional Sub-Hypergraph Sub-Hypertree Width of Q with respect to φ^* is the minimum of the Fractional Sub-Hypergraph Sub-Hypertree Width of all pairs of Junction Hypertree and root capable of executing this query. In general, finding the minimum of Fractional Sub-Hypergraph Sub-Hypertree Width for given query is intractable because there are exponential pairs of Junction Hypertree and root. We can prove that finding the minimum of Fractional Sub-Hypergraph Sub-Hypertree Width is NP-hard: finding the minimum fractional hypertree width is known to be NP-hard [26] and we can reduce this problem to finding the minimum of Fractional Sub-Hypergraph Sub-Hypertree Width by considering an empty φ^*. However, for most applications below where there are interactive queries over the original join graph, we can directly reuse the same Junction Hypertree. When Junction Hypertree and root are fixed, Fractional Sub-Hypergraph Sub-Hypertree Width could be found efficiently.

B 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.

C MINIMIZE STEINER TREE

In this section, we discuss 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:
  $x[e^*][n] = \min(x[e_1][m] + x[e_2][n-m] \text{ for } m = 0...n)$
  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 $\min(x[e_1][m] + x[e_1][j-m] \text{ for } m = 0...n)$.

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 $O(r^5)$.

D OPTIMIZE LATENCY WITH CJT

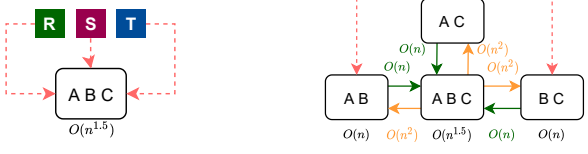
Optimize latency with CJT. F-IVM provides a trade-off between the size of CJT and update latency for cyclic join graph. The key insight is that, when there are multiple relations mapped to the same bag, we can create a new bag for individual relation and connect them with an empty bag, which might increase space but reduces latency. This trade-off is critical for Stream Query Processing, where latency is an important metric [5]. Consider the counting triangles under update [10, 23, 40]:

Example 16 (Count Triangle Under Update). *Given the triangle relations R(A,B), S(B,C), T(A,C) each with size* O(n)*, we want to count the total triangle count under update (one tuple is added to one relation). The* CJT *with the minimum size is shown in Figure 20a, where the bag size is* $O(n^{1.5})$*. We call this* CJT *the non-redundant design. The alternative design is to place relations in different bags and connect them with an empty bag as shown in Figure 20b. We call this* CJT *the redundant design, which has a larger time and space complexity as the messages from the empty bag are* $O(n^2)$*.*

While redundant design has a larger space complexity, it has a smaller update latency. Table 2 demonstrates the trade-off. Redundant design has an initial overhead of $O(n^2)$ *to build* CJT*, but after that, each update takes* O(1) *latency to see the updated query result and the* O(n) *calibration could take place in the background. On the contrary, for non-redundant design, calibration and update query result happens together, and users have to wait* O(n) *for the updated result.*

While previous works [7, 61, 70] focus on non-redundant design to optimize single query, the combination of CJT and F-IVM presents a novel trade-off. As a rule of thumb, when users are interested in the live streaming result, we should place frequently updated relations in different bags for low latency.

Experiment setup. We compare two Junction Hypertree designs for counting triangle problem $\gamma_{\text{COUNT}}(R[A,B] \bowtie S[B,C] \bowtie T[A,C])$



(a) Non-redundant design. **(b) Redundant design.**

Figure 20: Different design for count triangle.

	N	N-IVM	R	R-IVM
Space	$n^{1.5}$	$n^{1.5}$	n^2	n^2
Latency	$n^{1.5}$	n	1	1
Calibration	n	n	n^2	n^2 initial, n update

Table 2: Design trade-off. R stands for redundant and N stands for non-redundant.

	Task	Reduced	Redundant
B	Calibration + $\gamma_{\text{COUNT}(\cdot)}(\bowtie)$	126.18	621.65
	Update BC (1)	5.27	0.02
	Update BC (100)	6.45	0.05
	Augment BC (10000)	154.66	6.41
U	Calibration + $\gamma_{\text{COUNT}(\cdot)}(\bowtie)$	11.24	28332.29
	Update BC (1)	5.28	0.01
	Update BC (100)	6.14	0.05
	Augment BC (10000)	13.56	6.57

Table 3: Dataset result in millisecond. B for balanced. U for unbalanced.

on local C++ implemented query engine. There are two designs for cyclic join graph. The first design Reduced uses one unified tree node for all relations to retain the cycle. The second design Redundant breaks the cycle using one tree node for each relation, which are connected by an empty node in the middle. These two designs have different complexity for different tasks.

We consider two synthetic datasets: Balanced vs unbalanced. For balanced workload, each relation is cartesian product $[100] \times [100]$, where n is a unary relation with tuples from 1 to n. Each relation has size 10000 and the join size $10000^{1.5}$ which reaches the worst case bound. For unbalanced dataset, we generates $[1] * [10000]$ for AB and AC where A is highly biased, and $[100] \times [100]$ for BC. The join result is then only 10000. However, the message from empty tree node ABC to BC is 10000^2. We consider three tasks: Calibration for count, update count when 1 or 100 tuples in BC is updated, and augement BC. We choose BC is because BC is same in two datasets, and we want to fix the relation to study how other datasets affect. The task over other tables will scale proportional to their sizes.

Takeaways. The experiment result is in Table 3. For calibration, we find that unbalanced dataset is much larger for redundant, which is expected as it has larger theoretic bound. The calibration time for reduced is much smaller because small join size as expected. However, redundant design significantly reduces update latency by $> 100\times$. If calibration could be computed offline, it is worthwhile to place relation that is frequently updated and queried into a single bag.

E EXPLORATORY QUERY

We first study the performance improvement of queries over calibrated junction hypertree.

Datasets. We concisder four datasets: (1). Favorita [2] is a public dataset for purchasing and sales forecasting. Favorita has also been used by LMFAO [70] for factorized learning. (2). Lego [3] is a

public dataset for the inventories of every official LEGO set. (3). IMDB [1] is a public dataset containing information about movies and related facts about actors, production companies, etc. IMDB has been widely used as a benchmark for query optimizer [48, 53, 54]. (4). TPC-DS [57] (scale factor 1) is a synthetic dataset for decision support benchmark with star schema.

Similar to other works [6, 65], we apply dictionary encoding as preprocess to map original data values to 32-bit unsigned integers. Dictionary encoding eliminates fields with large strings and simplifies the operator design.

Competitors. We consider two competitors to Calibrated Junction Tree: (1). LMFAO [70] which is the state-of-art system that use junction tree to optimize for semi-ring aggregation queries. LMFAO generate efficient codes to execute queries, but it destroys all the intermediate results after query execution and can't apply IVM to maintain them. (2). Junction Hypertree-IVM is a variant of Calibrated Junction Hypertree, which doesn't calibrate the junction hypertree and, for each query, it passes message across the whole hypertee. However, it applies factorisied incremental view maintainance [61] to pass delta messages for what-if queries. Similar to LMFAO, we need to choose root for the junction tree to which all messages converge. For the experiment, we try all possible roots and report the one with the smallest run time.

Workloads. For all queries, we use COUNT as the semi-ring aggregation for simplicity. More complex aggregation has a large constant overhead, which won't affect the relative performance. Each query involves dimension tables, and we choose dimension tables popular in related queries for IMDB and TPC-DS, or in Kaggle notebook for Favorita and Lego. We consider three types of queries as workloads for experiment: (1). OLAP queries: we query the total count of join result, and add group by and dummy selection (selection doesn't exclude any tuple but force message passing) over the primary keys of dimension tables. (2). Data Explanation: we query the total count of join result after removing a small number of tuples in dimension tables. We choose 10 as the number of tuples to remove, and report the percentage of 10 tuples in the original dimension table. (3). Augmentation: we choose dimension table for augmentation and query the total count after augmentation. We generate an augmented table which maps primary key to random generated numbers.

Takeaways. The experiment result is shown in Table 4. Calibrated Junction Hypertree has the overhead of calibration, but it's orders of magnitudes faster for all three workloads by aggressively reusing the messages. We note that the calibration overhead of TPC-DS dataset is large compared (×20 compared to single query in LMFAO). This is because TPC-DS has star schema, where store_sales, the large fact table, is connected with many dimension tables, and all messages from store_sales are expensive. However, the expensive cost of calibration is one-time, and IVM could be applied to maintain the calibration for future updates to fact tables, as analyzed in the next section.

LMFAO outperforms Junction Hypertree-IVM for most OLAP and Augementation workloads. This is because of the implementation difference: We implement hash-based worst case optimal join [28], which has the overhead of iterating through trie, while LMFAO uses naive sort-merge join. LMFAO requires all the cyclic join to be pre-computed and only consider acyclic join graph as junction hypertree. LMFAO doesn't work for the common count and maintain triangles problem [74]. Besides, LMFAO is heavily optimized with efficient code generations.

Even if LFMAO is heavily optimized, Junction Hypertree-IVM outperforms Lego for some OLAP and Augmentation queries. This is mainly because LMFAO makes poor root choice in junction hypertree. The Inventory relation in Lego dataset is sparse, which causes the messages size asymmetric. LMFAO chooses root based on simple heuristics, which doesn't consider the sparsity of relation.

Finally, for what-if query, factorized IVM has a significant performance improvement especially when the ratio of tuples in original relation is small and calibration further improves the performance on top of factorized IVM. While factorized IVM allows the messages to be smaller, calibration further avoids unnecessary passing of messages.

	Task	LMFAO	JT-IVM	CJT
F	Calibration			2108.32
	$\gamma_{\text{COUNT}(\cdot)}(\bowtie)$	179.88	943.12	0.07
	$\gamma_{\text{COUNT}(\cdot),\text{Store}}(\bowtie)$	180.20	1151.70	0.06
	$\gamma_{\text{COUNT}(\cdot),\text{Oil}}\sigma_{\text{Store}}(\bowtie)$	179.71	1211.48	1.54
	Remove 10 Items (0.24%)	176.25	2.76	0.05
	Remove 10 Stores (18.86%)	177.33	24.21	0.03
	Remove 10 Transactions (0.01%)	179.24	5.63	0.02
	Augment Items (4100)	181.80	973.51	3.12
	Augment Holiday (1734)	180.74	1197.44	0.27
L	Calibration			393.64
	$\gamma_{\text{COUNT}(\cdot)}(\bowtie)$	54.15	13.84	0.09
	$\gamma_{\text{COUNT}(\cdot),\text{Set}}(\bowtie)$	53.31	369.42	0.15
	$\gamma_{\text{COUNT}(\cdot),\text{Part}}\sigma_{97.10\%\text{Color}}(\bowtie)$	56.39	14.38	0.65
	Remove 10 Color (7.46%)	54.61	127.75	0.02
	Remove 10 Themes (1.63%)	52.15	5.47	0.03
	Augment Color (135)	55.91	30.74	0.09
I	Calibration			43796.63
	$\gamma_{\text{COUNT}(\cdot)}(\bowtie)$	14722.10	18273.66	0.05
	$\gamma_{\text{COUNT}(\cdot),\text{Person}}(\bowtie)$	14854.87	24173.64	552.02
	$\gamma_{\text{COUNT}(\cdot),\text{Movie}}\sigma_{\text{Company}}(\bowtie)$	13599.41	17039.82	682.48
	Remove 10 Person (0.0002 %)	14753.22	18.11	0.12
	Remove 10 Movie-key (0.007%)	14833.76	2315.08	0.15
	Augment Person (4167491)	14953.03	24932.62	932.25
	Augment Company (234997)	14784.58	18567.45	275.66
T	Calibration			9402.95
	$\gamma_{\text{COUNT}(\cdot)}(\bowtie)$	482.53	653.70	0.06
	$\gamma_{\text{COUNT}(\cdot),\text{Store}}(\bowtie)$	483.13	1737.36	0.12
	$\gamma_{\text{COUNT}(\cdot),\text{Year}}\sigma_{\text{Store}}(\bowtie)$	478.75	1236.53	638.61
	Remove 10 Item (0.06%)	480.83	22.01	0.15
	Remove 10 Customer (0.1%)	484.98	18.25	0.05
	Augment Date (201)	484.61	652.39	0.12
	Augment Address (1920800)	498.53	1931.30	305.53

Table 4: Exploratory query. In millisecond.

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 [43].

Query 7. We rewrite the nested query and remove order-by.

```
SELECT N1.N_NAME AS SUPP_NATION,
N2.N_NAME AS CUST_NATION,
extract(year from L_SHIPDATE) as L_YEAR,
SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS VOLUME
FROM SUPPLIER, LINEITEM, ORDERS, CUSTOMER, NATION N1, NATION N2
WHERE S_SUPPKEY = L_SUPPKEY AND
O_ORDERKEY = L_ORDERKEY AND C_CUSTKEY = O_CUSTKEY AND
S_NATIONKEY = N1.N_NATIONKEY AND C_NATIONKEY = N2.N_NATIONKEY AND
( (N1.N_NAME = 'UNITED STATES' AND N2.N_NAME = 'JAPAN') OR
(N1.N_NAME = 'JAPAN' AND N2.N_NAME = 'UNITED STATES')
) AND L_SHIPDATE BETWEEN '1995-01-01' AND '1996-12-31'
GROUP BY SUPP_NATION, CUST_NATION, L_YEAR
```

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;
```