Change Propagation Without Joins

ABSTRACT

We revisit the classical change propagation framework for query evaluation under updates. The standard frameworks take a query plan and materialize the intermediate views, which incurs high polynomial costs in both space and time, and the culprit is the join operator. In this paper, we propose a new change propagation framework without using joins, thus naturally avoiding this polynomial blowup. Meanwhile, we show that the new framework still supports constant-delay enumeration of both the deltas and the full query results, the same as in the standard framework. Furthermore, we provide a quantitative analysis of its update cost, which not only recovers many recent theoretical results on the problem, but also yields an effective approach to optimizing the query plan. The new framework is also easy to integrate into an existing streaming database system. Experimental results show that our system prototype, implemented using Flink DataStream API, significantly outperforms other systems in terms of space, time, and latency.

ACM Reference Format:

. 2018. Change Propagation Without Joins. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

We study the problem of *query evaluation under under updates*, a.k.a. *incremental view maintenance*. Given a query Q, a database D, and a sequence of updates, where each update is either the insertion or deletion of a tuple, the goal is to maintain the query results $Q(D)$ continuously. More precisely, there are two modes to return the updated $Q(D)$ to the user (an end user or an upper-level application): *full result enumeration* and *delta enumeration*. The former is pull-based, i.e., we return $Q(D)$ passively upon the request of the user; while in the latter case, we push the delta $\Delta Q(D, t)$, i.e., the change to $Q(D)$ caused by the insertion/deletion of t, to the user after each update t. These two modes are applicable to different scenarios: Full result enumeration cannot be done too frequently if $Q(D)$ is large, and it may miss some ephemeral events in between two requests. Delta enumeration offers real-time responses with low latency, but it requires the user also to have the ability to "consume" the deltas in a timely fashion. It can be considered as a stream-in-stream-out operator, where the input is a stream of updates while the output is a stream of deltas. If the user wishes always to have a complete and accurate $Q(D)$, it has to maintain $Q(D)$ and update it with the deltas as they are received. If some approximation is acceptable, some more space- and time-efficient streaming algorithms can be used instead.

Change propagation. Change propagation [13, 26, 32] is a widely used framework in database systems for solving this problem. It can be instantiated with any query plan, which is a tree where the leaves are the input relations and each internal node is a relational operator. At each internal node, it maintains the results of the sub-query corresponding to the subtree at this internal node, which is often called a *materialized view*. Figure 1(a) shows a particular query plan for the query

$$Q := \pi_{x_1,x_2,x_3,x_4} R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie R_3(x_3, x_4) \bowtie R_4(x_4, x_5).$$

Under change propagation, we maintain four materialized views $V_1, V_2, V_3, V_4 = Q$ (if only delta enumeration is needed, then V_4 need not be maintained). When a tuple t is inserted or deleted in a relation, say R_1, it follows the leaf-to-root path to propagate the deltas to the root. More precisely, it first computes $\Delta V_2 = \Delta R_1 \bowtie V_1 = t \bowtie V_1$, then computes $\Delta Q = \Delta V_4 = \Delta V_2 \bowtie V_3$. Note that with the help of the materialized views, it avoids re-computing some of the sub-queries during updates.

However, the penalty is space: both V_1 and V_2 can have quadratic size in the worst case according to the AGM bound [6]. To void this space blowup, one can use a different query plan, say, the one shown in Figure 1(b). This query plan does not have any materialized views (except $V_1 = \pi_{x_4} R_4$, which has at most linear size), but it has to compute a multi-way join, e.g., $R_1 \bowtie R_2 \bowtie R_3 \bowtie t$ upon each update in R_4, which could take quadratic time. Making things worse, this quadratic blowup in either space or time exacerbates for queries involving more relations, which more precisely depends on the fractional edge cover number of the query [6].

Prior work has designed advanced techniques to address this space or time blowup. The Dynamic Yannakakis algorithm [21] has linear space and linear update time while supporting constant-delay enumeration for free-connex queries[1]; for q-hierarchical queries, the update time further reduces to $O(1)$ amortized[2]. Concurrently, Berkholz et al. [9] designed a different algorithm for the q-hierarchical case with the same space/time guarantees. However, these algorithms have not been integrated into any full-fledged database or data warehouse products, possibly due to the complications of the techniques and the use of non-standard operations not routinely found in existing database systems.

Change propagation without joins. The main contribution of this paper is to achieve (and further improvements for certain classes of queries and/or special update sequences) the results above, but still under the standard change propagation framework. Our observation is that the only relational operator that may cause a super-linear blowup is join. Thus, if the query plan has no joins, then both space and update time will be at most linear. To avoid joins, our high-level strategy is to replace each join in the query plan by a semi-join plus a projection. However, not every query plan is amenable to this replacement strategy. The key technical

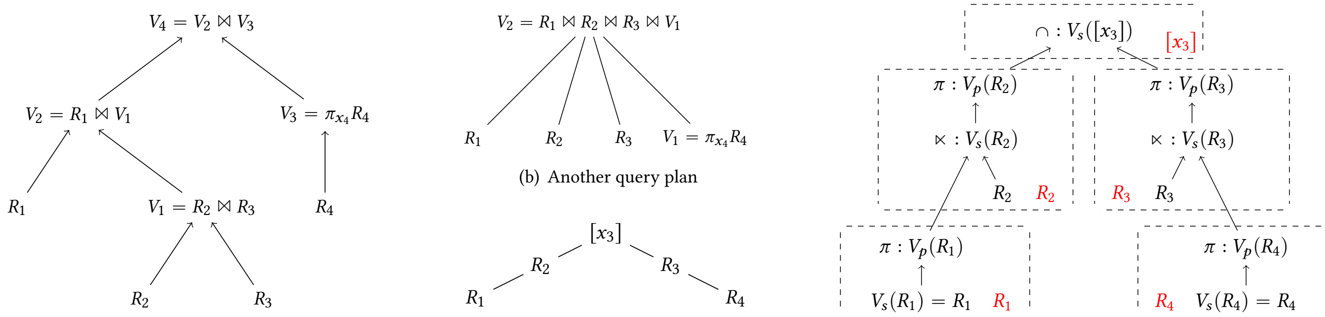
[1] All technical terms in the introduction are formally defined in Section 3.

[2] All update time bounds are amortized in this paper.

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00

https://doi.org/10.1145/1122445.1122456



(a) A query plan for change propagation

(b) Another query plan

(c) A generalized join tree

(d) Query plan under our framework

Figure 1: For the query $Q := \pi_{x_1,x_2,x_3,x_4} R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie R_3(x_3, x_4) \bowtie R_4(x_4, x_5)$, 1(a) and 1(b) are two traditional query plans for Q for change propagation and 1(d) is our new query plan induced by the free-connex generalized join tree in 1(c).

contribution of this paper, therefore, is the construction of such a query plan for every free-connex conjunctive query. For example, such a join-free query plan for the earlier query is shown in Figure 1(d), which will be elaborated in Section 4.

Since our query plan has no joins, linear space and linear update time follow straightforwardly. Still, two technical challenges remain: (1) how to support constant-delay enumeration, and (2) how to achieve a lower update time. (1) is trivial under a traditional query plan where the root corresponds to the query results $Q(D)$. Since our query plan is join-free, no node in the plan corresponds to $Q(D)$. Instead, our query plan can be considered as a compact, linear-size representation of a polynomially sized $Q(D)$. By borrowing ideas from constant-delay enumeration in the static case [7], we can enumerate $Q(D)$ with constant delay, by appropriately traversing the compact representation. Supporting constant-delay enumeration of the delta $\Delta Q(D, t)$, on the other hand, is quite different from the static case, and we need new techniques which exploit some important properties of our query plan.

To address issue (2), Wang and Yi [34] introduced the notion of *enclosureness* λ of an update sequence, which captures the hardness of the update sequence. It can be as large as the length of the update sequence itself in the worst case, but is often a constant in many common cases, such as any first-in-first-out (FIFO) update sequence. They also designed an algorithm with update cost $O(\lambda)$ for foreign-key acyclic queries (i.e., each join is between a primary key and a foreign key, and the PK-FK references form a DAG with only one root). Such queries are relatively easy to handle since these restrictions ensure that $|Q(D)|$ is at most linear, so they are immune to the polynomial blowup problem caused by non-key joins, such as free-connex queries. Indeed, we show (c.f. Theorem 15) that there is a full conjunctive query (a special case of a free-connex query) for which it is impossible to achieve $O(|D|^{1/2-\varepsilon})$ update time even over FIFO update sequences, which implies that the previous definition of λ is too aggressive to capture the hardness of update sequences for free-connex queries. Nevertheless, we show that, after a simple relaxation of the definition, λ is still an appropriate measure of the update complexity; in particular, we show that change propagation under our query plan achieves $O(\lambda)$ update time for every free-connex query. To illustrate the usefulness of our new definition of λ, we show that for certain queries (such as q-hierarchical queries) and/or update sequences (such as FIFO or insertion-only), λ is indeed a small constant. For general queries, λ also provides guidance on what would constitute a good query plan for change propagation.

Our results. Specifically, this paper achieves the following results:

(1) We show how to construct a change propagation query plan without joins for any free-connex conjunctive query, such that the space needed by the query plan is linear and the update time is $O(\lambda)$, for an appropriately defined notion of enclosureness λ of the update sequence.
(2) We show how to support constant-delay enumeration of both full query results and each delta in our query plan.
(3) We show that λ is a constant for certain classes of conjunctive queries (such as q-hierarchical queries) and/or special update sequences (such as FIFO or insertion-only). These results not only recover the prior known result of [9, 21] on q-hierarchical queries, but also extend it to cover many other cases commonly encountered in practice.
(4) We show how our framework can handle various extensions such as selections, aggregations, and non-free-connex queries.
(5) We demonstrate the practicality of our new framework by implementing it on top of Flink and comparing it with state-of-the-art view maintenance and SQL-over-stream systems.

2 RELATED WORK

Our new change propagation framework is inspired by several lines of research. In the static case, the classical Yannakakis algorithm [35] has run-time $O(|D| + |Q(D)|)$ for every free-connex query. It consists of two stages. The first stage uses a series of semi-join to remove all the dangling tuples in $O(|D|)$ time, and the second stage performs pairwise joins to compute $Q(D)$ in $O(|Q(D)|)$ time. The Dynamic Yannakakis algorithm [21] extends the algorithm to the dynamic case, but it deviates from the change propagation framework, making it harder to integrate into existing database systems. Our algorithm, in some sense, can be considered as a dynamic version of the Yannakakis algorithm in the standard change propagation framework. Furthermore, Dynamic Yannakakis can achieve $O(1)$ update time only for q-hierarchical queries, while our algorithm, thanks to the introduction of enclosureness λ, has $O(1)$ update time also for certain non-q-hierarchical if the update sequences possess special properties such as FIFO or insertion-only.

Bagan et al. [7] observe that, in the static case, the second stage of the Yannakakis algorithm can be enhanced to support constant-delay enumeration. We adapt their ideas to support enumeration in the dynamic case for our query plan. However, as there is no notion of delta in the static case, we need some new ideas to support delta enumeration with constant delay, which non-trivially rely on some nice features of our query plan.

Kara et al. [25] show that, for hierarchical (but non-q-hierarchical) queries, it is possible to increase the enumeration delay in exchange for faster update time. We have not considered this trade-off, as we believe the constant delay is important, and our update cost λ is low enough for most queries and update sequences already. Furthermore, their trade-off only applies to full enumeration, not delta enumeration. Nevertheless, for cases where λ is high, it would be an interesting direction to explore such a trade-off.

In the standard change propagation framework, a single update to a base relation may incur many changes in the intermediate views. Higher-Order Incremental View Maintenance (HIVM) [5] has been proposed to remedy this problem. It takes the changes to a view as another query (delta query) and maintains this delta query recursively. HIVM improves upon IVM for many complex queries in practice, and it can also extend to accelerate several machine learning tasks [30, 31], but there is no theoretical guarantee on its update time. Furthermore, HIVM still uses super-linear space.

The problem is also related to stream joins. In particular, a cash-register stream corresponds to an insertion-only update sequence, while a turnstile stream is an update sequence with arbitrary insertions and deletions. The sliding-window stream model is a special case of a FIFO update sequence. Most stream processing systems like Flink [10] and Trill [12] use standard change propagation for maintaining multi-way stream joins, which we will compare against in Section 8. There are also some specialized systems designed for binary stream joins [15, 18, 23, 28, 33], but they do not extend to multi-way joins.

3 PRELIMINARIES

3.1 Problem Definition

Conjunctive queries. We focus on *conjunctive queries (CQ)* of the following form:

$$Q := \pi_{\mathbf{y}} \left(R_1(e_1) \bowtie R_2(e_2) \bowtie \cdots \bowtie R_n(e_n) \right), \quad (1)$$

where each R_i is a relation with a set of attributes/variables e_i, $i = 1, \ldots, n$. Each tuple $t \in R_i$ assigns a value to each attribute in e_i. For any $x \in e_i$, $t[x] = \pi_x t$ denotes the value of t on attribute x. Similarly, for a subset of attributes $e \subseteq e_i$, $t[e] = \pi_e t$ denotes the tuple formed by the values of t on the attributes in e.

Let $\mathcal{V} = e_1 \cup \cdots \cup e_n$ be the set of all attributes in the query. We call $\mathbf{y} \subseteq \mathcal{V}$ the *output attributes*, while $\bar{\mathbf{y}} = \mathcal{V} - \mathbf{y}$ are the *non-output attributes*, also known as the *existential variables*. If $\mathbf{y} = \mathcal{V}$, such a query is known as a *full query*; otherwise, it is said to be *non-full*. For simplicity, we assume that each R_i in Q is distinct, i.e., the query does not have self-joins. Nevertheless, self-joins can be taken care of easily: Suppose a relation R appears twice in the query (with different attribute renamings). Then we consider them as two identical copies of R, and for any update to R, we apply the update to both copies of R.

Given a database D, we write $Q(D)$ for the query results of Q on D. We use $Q(D \ltimes t)$ to denote the query results that depend on a given tuple t, and call $Q(D \ltimes t)$ the query results *witnessed* by t. Such a *witness query* will be frequently used in this paper. Given a query Q in the form of (1) and a tuple $t \in R_i$, it is clear that

$$Q(D \ltimes t) = \pi_{\mathbf{y}} \left(R_1 \bowtie \cdots \bowtie R_{i-1} \bowtie \{t\} \bowtie R_{i+1} \bowtie \cdots \bowtie R_n \right).$$

Note that for a full CQ, we have $Q(D \ltimes t) = Q(D + t) \ltimes t$; for non-full queries, t itself may not appear in $Q(D \ltimes t)$ due to the projection on $\mathbf{y}$. When analyzing the costs of algorithms, we adopt the notion of *data complexity*, i.e., the size of the query Q is taken as a constant while $|D|$ is an asymptotic parameter.

Updates and update sequences. An *update* to a database D is either the insertion or deletion of a tuple t in some relation R_i of D. In this paper, we adopt set semantics. In particular, this means that if R_i already contains t, then inserting t into R_i will not change R_i; if R_i does not contain t, deleting t from R_i has no effect, either. We ignore these non-effective updates.

Given an update sequence S_D, the *lifespan* of tuple t is an interval $I(t) = [t^+, t^-]$, where t^+ denotes the timestamp when t is inserted into D and t^- denotes the timestamp when t is deleted from D. We set $t^+ = -\infty$ to indicate that t exists in the initial D and $t^- = +\infty$ indicates that t still exists in D after the update sequence. Note that if a tuple is repeatedly inserted and deleted, it will be treated as multiple tuples, which have the same values but disjoint lifespans.

Although our algorithms will be able to handle arbitrary update sequences, their performance can be better if the update sequences possess some nice properties. In particular, the following two restrictive classes of update sequences are of practical importance:

- **First-in-first-out (FIFO)**. A update sequence S_D is FIFO if for any two tuples $t_1, t_2 \in S_D$, $t_1^+ < t_2^+$ implies $t_1^- < t_2^-$. FIFO sequences are commonly used in practice, such as sliding-window or tumbling-window model over streaming data.
- **Insertion-only or deletion-only.** A update sequence S_D is insertion-only (w.r.t. deletion-only) if for any tuple $t \in S_D$, $t^- = +\infty$ (w.r.t. $t^+ = -\infty$). The two cases are symmetric, so we will only discuss the insertion-only case in this paper.

Deltas. The *delta* of an update to Q is defined as

$$\Delta Q(D, t) = Q(D + t) - Q(D)$$

in case of the insertion of t and

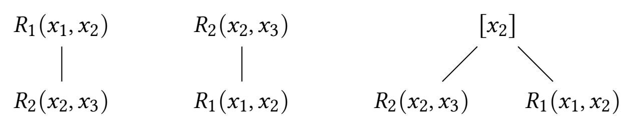
$$\Delta Q(D, t) = Q(D) - Q(D - t)$$

in the case of a deletion. For a full query, we have $\Delta Q(D, t) = Q(D \ltimes t)$. For non-full queries, we have $\Delta Q(D, t) \subseteq Q(D \ltimes t)$. In particular, it is possible to have $\Delta Q(D, t) = \emptyset$ even if $Q(D \ltimes t) \neq \emptyset$.

We guarantee *constant-delay enumeration* [7] for both full result enumeration and delta enumeration, i.e., the time between the start of the enumeration process to the first tuple in $Q(D)$ (or $\Delta Q(D, t)$), the time between any consecutive pair of tuples, and the time between the last tuple and the termination of the enumeration process should all be bounded by a constant.

3.2 Classification of CQs

Acyclic queries. The acyclicity of a CQ Q is defined by the acyclicity of the hypergraph $(\mathcal{V}, \{e_1, \ldots, e_n\})$. More precisely, Q is acyclic



(a) $\mathcal{T}_1$ (b) $\mathcal{T}_2$ (c) $\mathcal{T}_3$

Figure 2: An illustration of (generalized) join trees for the query $Q_1 := R_1(x_1, x_2) \bowtie R_2(x_2, x_3)$. In Figure 2(c), node $[x_2]$ represents a generalized relation, which contains one attribute x_2. The height of $\mathcal{T}_1, \mathcal{T}_2$ is 2 and that of $\mathcal{T}_3$ is 1.

if there exists a join tree $\mathcal{T}$, whose nodes are the relations in Q such that, for each attribute $x \in \mathcal{V}$, all nodes of $\mathcal{T}$ containing x form a connected component of $\mathcal{T}$. For example, Figure 2(a) and 2(b) are two possible join trees for the query $Q_1 := R_1(x_1, x_2) \bowtie R_2(x_2, x_3)$. We will often not distinguish between a node in $\mathcal{T}$ and the relation it represents, or its set of attributes.

In this paper, we use an equivalent definition using *generalized relations* [14, 21]. A *generalized relation* is a virtual relation whose attributes can be any proper subset of any e_i. We can show that the following is an equivalent definition of acyclic queries[3]:

Definition 1 (Acyclic queries). A CQ Q is acyclic if there exists a rooted join tree $\mathcal{T}$ satisfying the following properties:

(1) each input relation in Q correspond to a unique node in $\mathcal{T}$, each leaf of $\mathcal{T}$ corresponds to an input relation, and each internal node in $\mathcal{T}$ corresponds to either an input relation in Q or one of its generalized relations (some generalized relations may not appear in $\mathcal{T}$);
(2) for each attribute x, all nodes of $\mathcal{T}$ containing x form a connected component of $\mathcal{T}$;
(3) a node corresponding to a generalized relation must appear above any node corresponding to an input relation; and
(4) if e is the parent of e' in $\mathcal{T}$ and e is a generalized relation, $e \subseteq e'$.

An example of a generalized join tree is given in Figure 2(c). In a generalized join tree $\mathcal{T}$, we use r to denote its root node, and $\mathcal{T}_e$ for the subtree rooted at node e. For a node $e \in \mathcal{T}$, C_e denotes the set of children of e and $p(e)$ the parent of e. Let $\mathsf{key}(e) = e \cap p(e)$ be the *join key* of node e. We define the height of $\mathcal{T}$ as the maximum number of relations on any leaf to root path, without counting the generalized relations. Please see Figure 2 for some examples.

Free-connex queries. A CQ Q is *free-connex* if the hypergraphs $(\mathcal{V}, \{e_1, \ldots, e_n\})$ and $(\mathcal{V}, \{e_1, \ldots, e_n, \mathbf{y}\})$ are both acyclic. By definition, any free-connex query must by acyclic, and an acyclic full query must be free-connex. Free-connex CQs have played an important role in query enumeration. In particular, [7] showed that constant-delay enumeration after a linear-time preprocessing step is possible if and only if Q is free-connex. For our development, we need the following equivalent definition of free-connex queries:

Definition 2 (Free-connex queries). A CQ Q is free-connex if it has a generalized join tree $\mathcal{T}$, such that $r \subseteq \mathbf{y}$, and for every $x_1 \in \mathbf{y}$ and every $x_2 \in \mathcal{V} - \mathbf{y}$, $\mathsf{top}(x_2)$ is not an ancestor of $\mathsf{top}(x_1)$ in $\mathcal{T}$, where $\mathsf{top}(x)$ is the highest node in $\mathcal{T}$ that contains x. Such a $\mathcal{T}$ is called a free-connex join tree.

[3]Proof of equivalence is given in the full version [1] of the paper.

For example, for the query $Q_1' := \pi_{x_2} R_1(x_1, x_2) \bowtie R_2(x_2, x_3)$, all three join trees in Figure 2 are valid free-connex join trees. If the output attribute is x_1, then only Figure 2(a) is a valid free-connex join tree (so it does not have a height-1 free-connex join tree). If the output attributes are (x_1, x_3), then the query is not free-connex.

Q-hierarchical queries. While free-connex queries can be evaluated efficiently in the static case, efficient updates are possible only for a more restricted class of queries. Berkholz et al. [9] show that constant-delay enumeration is possible from a data structure that is updated in constant time if and only if Q is *q-hierarchical*.

Definition 3 (q-hierarchical queries). Let $\mathcal{E}_x$ denote the set of relations containing attribute x. A CQ Q is *q-hierarchical* if (1) for every pair of attributes x_1, x_2, either $\mathcal{E}_{x_1} \subseteq \mathcal{E}_{x_2}$ or $\mathcal{E}_{x_2} \subseteq \mathcal{E}_{x_1}$ or $\mathcal{E}_{x_1} \cap \mathcal{E}_{x_2} = \emptyset$; and (2) for every pair of attributes x_1, x_2, if $x_1 \in \mathbf{y}$ and $\mathcal{E}_{x_1} \subsetneq \mathcal{E}_{x_2}$, then $x_2 \in \mathbf{y}$.

4 CHANGE PROPAGATION WITHOUT JOINS

4.1 A New Query Plan

Given a free-connex query Q, our new query plan is guided by a free-connex (generalized) join tree $\mathcal{T}$ of Q. We illustrate the construction using the query in Figure 1 with the join tree in Figure 1(c) (note that the join tree is not unique). A normal query plan following this join tree would compute a series of joins $(R_1 \bowtie R_2) \bowtie (R_3 \bowtie \pi_{x_4} R_4)$. In our new query plan, we replace each join with a semi-join followed by a projection. More precisely, we maintain two views for each node $e \in \mathcal{T}$, a semi-join view $V_s(R_e)$ and a projection view $V_p(R_e)$, defined recursively as follows.

Every non-root node $e \in \mathcal{T}$ has a *projection view*

$$V_p(R_e) := \pi_{\mathsf{key}(e)} V_s(R_e). \tag{2}$$

To define the *semi-join view* $V_s(R_e)$, we distinguish three cases.

(i) If e is a leaf, R_e is an input relation, and $V_s(R_e) := R(e)$.
(ii) If e is an internal node and R_e is an input relation, then

$$V_s(R_e) := R_e \ltimes V_p(R_{e_1}) \ltimes \cdots \ltimes V_p(R_{e_k}), \tag{3}$$

where $C_e = \{e_1, \ldots, e_k\}$ are the children of e.
(iii) If e is an internal node that corresponds to a generalized virtual relation R_e, since all the $V_p(R_{e_i})$'s have the same attributes $\mathsf{key}(e_i) = e_i \cap e = e$ for every i (by the last property in Definition 1), (3) simplifies to an intersection:

$$V_s(R_e) := V_p(R_{e_1}) \cap \cdots \cap V_p(R_{e_k}). \tag{4}$$

Our query plan simply connects these views together using the formulas above. Figure 1(d) shows our query plan induced by the join tree in Figure 1(c). Note that R_2 and R_4 fall into case (ii) above, while the root node $[x_3]$ is under case (iii). Since neither projection nor semi-join (including intersection as a special case) enlarges the input relations, the following is straightforward:

Lemma 4. *All views in our query plan have size $O(|D|)$.*

4.2 Change propagation

Change propagation using our new query plan can be done using standard (actually, even simpler for certain operators) propagation formulas [13]. For completeness, we briefly describe them below,

Algorithm 1: S-Update(e, t)

Input : An update t from $V_s(R_e)$;
Output : Updated $V_p(R_e)$;

$t' \leftarrow t[\mathsf{key}(e)]$;
if t *is an insertion into* $V_s(R_e)$ **then**
  if $t' \in V_p(R_e)$ **then** $\mathsf{count}[t'] \leftarrow \mathsf{count}[t'] + 1$;
  $V_p(R_e) \leftarrow V_p(R_e) \cup \{t'\}$, $\mathsf{count}[t'] \leftarrow 1$, P-Update$(p(e), t')$;
else
  if *count*$[t'] = 1$ **then**
    $V_p(R_e) \leftarrow V_p(R_e) - \{t'\}$, P-Update$(p(e), t')$;
  else $\mathsf{count}[t'] \leftarrow \mathsf{count}[t'] - 1$;

Algorithm 2: P-Update(e, t)

Input : An update t from $V_p(R_{e_i})$ for some $e_i \in C_e$;
Output : Updated $V_s(R_e)$;

if t *is an insertion into* $V_p(R_{e_i})$ **then**
  foreach $t' \in R_e$ *with* $t'[key(e_i)] = t$ **do**
    $\mathsf{count}[t'] \leftarrow \mathsf{count}[t'] + 1$;
    if *count*$[t'] = |C_e|$ **then**
      $V_s(R_e) \leftarrow V_s(R_e) \cup \{t'\}$, S-Update$(e, t')$;
else
  foreach $t' \in R_e$ *with* $t'[key(e_i)] = t$ **do**
    $\mathsf{count}[t'] \leftarrow \mathsf{count}[t'] - 1$;
    if *count*$[t'] = |C_e| - 1$ **then**
      $V_s(R_e) \leftarrow V_s(R_e) - \{t'\}$, S-Update$(e, t')$;

Algorithm 3: R-Update(e, t)

Input : An update t from an input relation R_e;
Output : Updated $V_s(R_e)$;

if t *is an insertion into* R_e **then**
  $\mathsf{count}[t] \leftarrow 0$;
  foreach $e_i \in C_e$ **do**
    if $t[key(e_i)] \in V_p(e_i)$ **then** $\mathsf{count}[t] \leftarrow \mathsf{count}[t] + 1$;
  if *count*$[t] = |C_e|$ **then**
    $V_s(R_e) \leftarrow V_s(R_e) \cup \{t\}$, S-Update$(e, t)$;
else
  if *count*$[t] = |C_e|$ **then**
    $V_s(R_e) \leftarrow V_s(R_e) - \{t\}$, S-Update$(e, t)$;

which are also needed to understand the enumeration algorithms in Section 5. A running example is given in Section 5.3.

S-Update When there is an update to $V_s(R_e)$ for some e, we use an S-Update to update $V_p(R_e)$ by formula (2). This can be done in $O(1)$ time by *derivation counting* [13], a standard technique to propagate changes through a projection. Specifically, we associate a counter $\mathsf{count}[t']$ for each tuple $t' \in V_p(R_e)$ that stores the number of tuples $t \in V_s(R_e)$ such that $t[\mathsf{key}(e)] = t'$. The detailed process, which needs to distinguish between an insertion and a deletion, is given in Algorithm 1. Note that for the algorithm to run in $O(1)$ time, we need a hash index on $V_p(R_e)$.

P-Update Let e_i be a child of e. When there is an update to some $V_p(R_{e_i})$, we use a P-Update to update $V_s(R_e)$ by formula (3) in the case where e is an input relation or (4) in case e is a generalized relation. We consider the former case first; the latter case is similar.

The standard change propagation formula for a semi-join [19] rewrites it as a join followed by a projection, e.g., $R_e \ltimes R_{e_i} := \pi_e(R_e \bowtie R_{e_i})$. This defeats the whole purpose of avoiding joins. However, observe that in our query plan, R_{e_i} has already been projected onto $\mathsf{key}(e_i) = e_i \cap e \subseteq e$ before the semi-join, thus this allows a very simple and efficient way to maintain the whole multi-way semi-join (3) as one operator, which can also be considered as a "horizontal" version of derivation counting. More precisely, we maintain a counter $\mathsf{count}[t']$ for every tuple t' in R_e, storing the number of child nodes $e_i \in C_e$ such that $t'[\mathsf{key}(e_i)] \in V_p(R_{e_i})$. A tuple t' appears in $V_s(R_e)$ if and only if $\mathsf{count}[t'] = |C_e|$. The algorithm is then immediate, as shown in Algorithm 2. We also need a hash index[4] on R_e so that each counter change can be done in $O(1)$ time. However, unlike the S-Update, a P-Update may take more than constant time since multiple tuples may change their counters. In fact, this is the only place where the update time blows up during change propagation in our query plan.

When e is a generalized relation and $V_s(R_e)$ is defined by (4), the process is almost the same, except that R_e is virtual. Thus, we define $R_e := V_p(R_{e_1}) \cup \cdots \cup V_p(R_{e_k})$ and Algorithm 2 applies verbatim.

R-Update. The last case is when there is an update in an input relation R_e, we also need to update $V_s(R_e)$ by formula (3). We call this an R-Update. The detailed procedure, given in Algorithm 3, simply maintains the counters in R_e, and then $V_s(R_e)$, in a straightforward manner. It is obvious that an R-Update takes $O(1)$ time (also using the hash index on $V_p(R_{e_i})$).

Lemma 5. *All views in query plan can be updated in $O(|D|)$ time.*

5 ENUMERATION

5.1 Full Result Enumeration

We first consider how to perform constant-delay enumeration of $Q(D)$ from our query plan. We need the following lemma:

Lemma 6. *For any node e, $V_s(R_e) = \pi_e(\bowtie_{e' \in \mathcal{T}_e} R_{e'})$.*

In plain language, the semi-join view of node e is essentially the projection of the join results of relations in the subtree rooted at e, to attributes in e. An immediate corollary is

Corollary 7. $V_s(R_r) = \pi_r Q(D)$.

This means that the semi-join view at the root r (recall that r does not have a projection view) contains precisely all the query results projected onto r. Using the notion of a witness query, this leads to the following useful fact for full result enumeration, where $\uplus$ denotes disjoint union:

Lemma 8. $Q(D) = \biguplus_{t \in V_s(R_r)} Q(D \ltimes t)$.

Lemma 6, Corollary 7, and Lemma 8 allow us to use essentially the same algorithm from Bagan et al. [7] to achieve constant-delay enumeration of $Q(D)$; see Algorithm 4, which takes as input a node $e \in \mathcal{T}$ and a tuple $t \in R_e$, and yields the query results over $\mathcal{T}_e$ that can be joined with t. To enumerate $Q(D)$, we simply invoke FullEnum$(\mathcal{T}, r, t)$ for every tuple $t \in V_s(R_r)$.

[4]This hash index needs to support $e \cap e_i$ as the key for each $e_i \in C_e$.

Lemma 9. *Algorithm 4 enumerates $Q(D)$ with constant delay.*

5.2 Delta Enumeration

Delta enumeration is straightforward in a standard query plan, as the root node corresponds to $Q(D)$, so all changes propagated to the root are precisely $\Delta Q(D,t)$. However, it becomes tricky in our new query plan, as no node corresponds to $Q(D)$, which is necessarily the case if a linear-size representation of $Q(D)$ is desired. In our query plan, one cannot just inspect the root, because not every change propagates to the root, and many propagations stop midway, which is actually a main reason why our query plan is not only space-efficient but also time-efficient. Recall that the full result enumeration algorithm relies on Lemma 8. Then the key question is, can we have an analogy of Lemma 8 for the delta $\Delta Q(D,t)$? In other words, can we identify a set of witness tuples t' for t such that the delta $\Delta Q(D,t)$ is the disjoint union of $Q(D \ltimes t')$? Fortunately, the answer is yes, but the answer is not as simple as Lemma 8.

Let's first consider the insertion case. When we insert t into some R_e, the propagation follows the path from e to r, by (possibly) applying an R-Update first, then an S-Update, P-Update, S-Update, P-Update, ... Recall that both S-update and R-update only propagate a single change upward (see line 8, 12 in Algorithm 1 and Algorithm 3), but P-update may propagate multiple changes upward (see line 6, 12 in Algorithm 1. Hence, there could be multiple propagation paths starting from t. To be more precise, we denote the nodes lying on the path from e to r as $e_0 = e, e_1, e_2, \cdots, e_k = r$. Every propagation path inserts a tuple into each of the views on the path, and we denote the inserted tuples on such a path as $(t, t_0^s, t_0^p, t_1^s, t_1^p, \cdots,)$, where $t_i^s \in V_s(R_i)$ and $t_i^p \in V_p(R_i)$ for $i \in \{0,1,2,\cdots,k\}$.

Now, we distinguish three possible cases with respect to the ending tuple of a propagation path: (1) the ending tuple is t; (2) the ending tuple is t_j^p for some $j \in \{0,1,2,\cdots,k\}$; (3) the ending tuple is t_i^s for some $i \in \{0,1,2,\cdots,k\}$.

Case (1) happens when the first update is an R-Update and does not propagate any further change. This means that in Algorithm 3, there exists some child node e' of e such that $t[\text{key}(e')] \notin V_p(R_{e'})$, i.e., t cannot join with $\mathcal{T}_{e'}$. In this case, t will not produce any change to $Q(D)$, thus can be ignored.

Case (2) happens when P-Update(e_j, t_j^p) does not propagate any further change. Putting it into Algorithm 2, this means that either there exists no tuple $t' \in R_{p(e_j)}$ that can join with t_j^p, or if such a tuple exists, but it cannot join with any query result over $\mathcal{T}'_e$ for some child node e' of $p(e_j)$, since its counter is smaller than $|C_{p(e_j)}|$. In either case, this propagation path will not cause any change to $Q(D)$, thus can also be ignored.

Case (3) happens when S-update(e_i, t_i^s) does not propagate any further change. Putting it into Algorithm 1, this means that either we have reached the root, or there exists some other tuple $t' \in V_p(R_i)$ such that $t' \neq t_i^s$ and $t_i^s[\text{key}(e_i)] = t'[\text{key}(e_i)]$. This is the only case where changes to $Q(D)$ can *possibly* happen. We will give a more detailed characterization of this case later.

Live views. To support constant-delay delta enumeration, we maintain a *live view* for each node e such that $e \cap \mathbf{y} \neq \emptyset$:

$$V_l(R_e) := \pi_e Q(D),$$

Algorithm 4: FullEnum$(\mathcal{T}, e, t)$

Input: A free-connex generalized join tree $\mathcal{T}$, a node $e \in \mathcal{T}$ and a tuple $t \in R_e$;
Output: Query results over $\mathcal{T}_e$ that can be joined with t;

if $e - \mathbf{y} \neq \emptyset$ **then**
  if $e \cap \mathbf{y} - p(e) = \emptyset$ **then** Yield $\langle\rangle$;
  else Yield $\pi_{\mathbf{y}\cap e}(R_e \ltimes t)$;
else
  Let $C_e = \{e_1, e_2, \cdots, e_k\}$;
  foreach $t_1 \in$ FullEnum$(\mathcal{T}, e_1, t[key(e_1)])$ **do**
    foreach $t_2 \in$ FullEnum$(\mathcal{T}, e_2, t[key(e_2)]$ **do**
      $\cdots$
      foreach $t_k \in$ FullEnum$(\mathcal{T}, e_k, t[key(e_k)])$ **do**
        Yield $t \bowtie t_1 \bowtie t_2 \bowtie \cdots \bowtie t_k$;

Algorithm 5: DeltaEnum$(\mathcal{T}, t)$

Input: A free-connex generalized join tree $\mathcal{T}$; an updated tuple t.
Output: Delta results induced by t.

Let $e_0, e_1, \cdots, e_k = r$ be the nodes on t's propagation path;
foreach *witness tuple t' of t* **do**
  Let e_i be the node such that $t' \in \pi_{\mathbf{y}} \Delta V_s(R_{e_i}, t)$;
  $S \leftarrow t' \bowtie V_l(R_{e_{i+1}}) \bowtie \cdots \bowtie V_l(R_{e_k})$;
  foreach $q \in S$ **do**
    $S_i \leftarrow$FullEnum$(\mathcal{T}_{e_i}, e_i, q[e_i])$;
    $S_j \leftarrow$FullEnum$(\mathcal{T}_{e_j} - \mathcal{T}_{e_{j-1}}, e_j, q[e_j]), j \in [i+1, k]$;
    Yield $S_i \bowtie S_{i+1} \bowtie \cdots \bowtie S_k$;

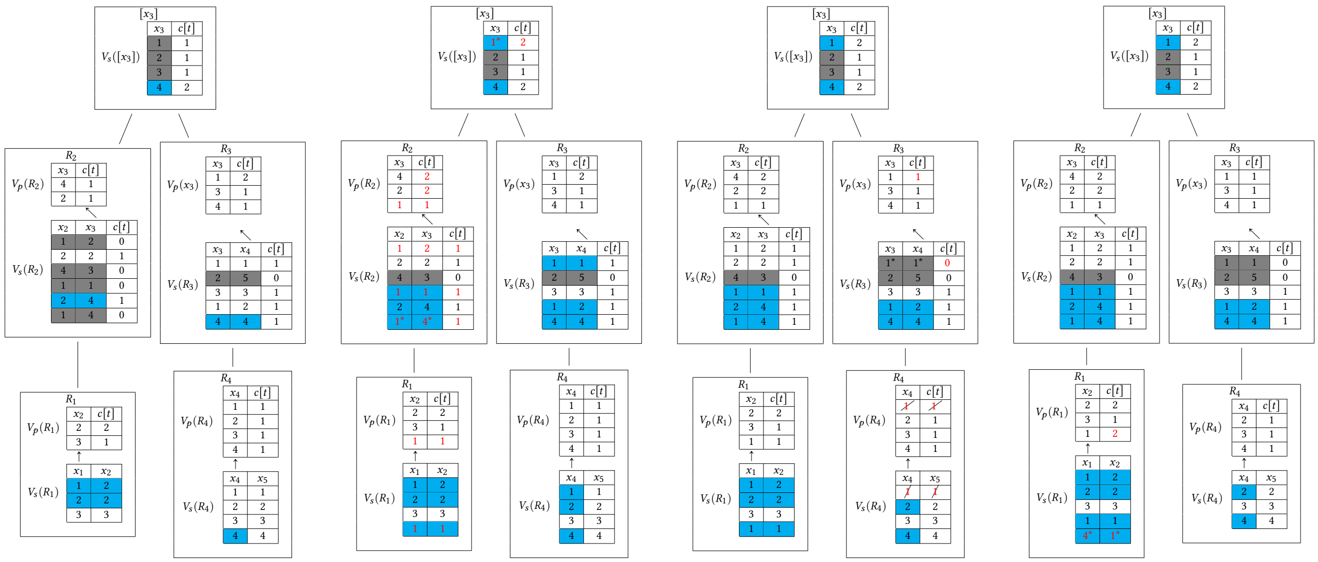
which are the "live" tuples (i.e., appearing in the query results) projected onto e. Note that $V_l(R_e) \subseteq \pi_{\mathbf{y}} V_s(R_e)$, which means for $e \subseteq \mathbf{y}$, it can be implemented by simply adding an extra bit in $V_s(R_e)$, indicating if the corresponding tuple is in $V_l(R_e)$.

For the root r, there is no need to maintain $V_l(R_r)$ separately since $V_l(R_r) = V_s(R_r)$ by Corollary 7. For the leaf nodes, their live views need not be maintained, either, since they will not be needed by delta enumeration. The other live views can be maintained by the following observation:

Lemma 10. *For any non-root node e such that $e \cap \mathbf{y} \neq \emptyset$ and any tuple $t \in \pi_{\mathbf{y}} V_s(R_e)$, $t \in V_l(R_e)$ if and only if $t \bowtie V_l(R_{p(e)}) \neq \emptyset$.*

Based on the Lemma 10, the maintenance of $V_l(R_e)$ can piggyback on the delta enumeration: After enumerating a result $t' \in \Delta Q(D,t)$, we update the live views in a top-down fashion. For every non-root e such that $e \cap \mathbf{y} \neq \emptyset$, if the update is insertion, then we always add $t'[e]$ to $V_l(R_e)$; if the update is deletion, then we delete $t'[e]$ from $V_l(R_e)$ if $t'[e]$ cannot join with $V_l(R_{p(e)})$, which can be done in $O(1)$ time with a hash index on $V_l(R_{p(e)})$ (which is physically the same hash index on $V_s(R_{p(e)})$ for $e \subseteq \mathbf{y}$). This only adds another constant to the delay of delta enumeration.

Witness tuples. We now are ready to give a more precise characterization of the ending tuples falling into Case (3) that actually cause changes to $Q(D)$, called *witness tuples*:



(a) Initialization. (b) After the insertion of $(1,1)$ into R_1. (c) After the deletion of $(1,1)$ from R_4. (d) After the insertion of $(4,1)$ into R_1.

Figure 3: A running instance for the query in Figure 1 using the query plan in Figure 1(d). Tuples in white are those in $V_s(R)$, in grey are those in $R \backslash V_s(R)$, in cyan are those in $V_l(R)$ (live views for leaf nodes are not needed, but we still show them for clarity), with star symbols are the witness tuples. Changes in each step are marked in red.

Definition 11 (Witness tuple). Suppose t is inserted into or deleted from D. A tuple t' is a witness of t if

$$t' \in \Delta V_s(R_r, t), \tag{5}$$

or

$$t' \in \pi_{\mathbf{y}} \Delta V_s(R_e, t) \ltimes V_l(R_{p(e)}) \tag{6}$$

for some non-root e such that $e \cap \mathbf{y} \neq \emptyset$.

Here $\Delta V_s(R_e, t)$ denotes the tuples to be inserted into (or deleted from) $V_s(R_e)$ due to t and $V_l(R_{p(e)})$ is the live view before the update. We give some intuition behind Definition 11. First, (5) is the counterpart of Corollary 7 for delta enumeration and such a t' is guaranteed to generate changes to $Q(D)$. (6) is specific for delta enumeration, addressing the situation mentioned earlier, where the propagation stops mid-way yet still causing changes to $Q(D)$. Note that in this case, the attributes of t' are $e \cap \mathbf{y}$. Then (6) implies that $t' \in \pi_{\mathbf{y}} \Delta V_s(R_e, t)$ and $t'[\text{key}(e)] \in \pi_{\text{key}(e)} V_l(R_{p(e)})$. Since $t'[\text{key}(e)] \in \pi_{\text{key}(e)} V_l(R_{p(e)})$, it must have $t'[\text{key}(e)] \in V_p(R_e)$, i.e. $t'[\text{key}(e)] \notin \Delta V_p(R_e, t)$, which means that the propagation stops at node e under case (3). In addition, each witness tuple t' should (i) contribute to the delta over $\mathcal{T}_e$ induced by t, and (ii) join with tuples from the remaining relations in $\mathcal{T} - \mathcal{T}_e$. For (i), it suffices to require $t' \in \Delta\left(\pi_{\mathbf{y}} V_s(R_e)\right) = \pi_{\mathbf{y}} \Delta V_s(R_e, t)$, since $\Delta\left(\pi_{e \cap \mathbf{y}}(\Join_{e' \in \mathcal{T}_e} R_{e'})\right) = \Delta\left(\pi_{\mathbf{y}} V_s(R_e)\right)$. For (ii), it suffices to require $t' \ltimes V_l(R_{p(e)}) \neq \emptyset$, and this is exactly the reason we introduced $V_l(R_e)$ in the first place.

Lemma 12. $\Delta Q(D,t) = \biguplus_{t':a \text{ witness of } t} Q(D \ltimes t')$.

We are now ready to state the counterpart of Lemma 8 for delta enumeration, in Lemma 12. Unlike Lemma 8, the proof of Lemma 12 is nontrivial, and the details are given in the full version [1].

The algorithm. To perform delta enumeration using Lemma 12, we still need to address two issues: (1) how to find all witness tuples t', and (2) how to enumerate $Q(D \ltimes t')$ with constant delay.

To find all the witness tuples, we consider the two cases in Definition 11: (5) can be computed easily after updating $V_s(R_r)$; for (6), just an extra check with $V_l(R_{p(e)})$ is needed, which can be done in $O(1)$ time using the hash index on $V_l(R_{p(e)})$. These steps only increase the update cost by a constant factor.

It remains to describe how to enumerate $Q(D \ltimes t')$ for each witness t'. As before, let $e_0, e_1, \ldots, e_k = r$ be the nodes on the propagation path, and suppose we are given a witness tuple $t' \in \pi_{\mathbf{y}} \Delta V_s(R_{e_i}, t)$ for some i. We first enumerate the query results participated by t' together with relations on the path from e_{i+1} to the root r, denoted as S. This can be done by joining t with the live views associated with these nodes. For each such result $q \in S$, we enumerate the query results that participated by q. This enumeration is done by partitioning the whole generalized join tree into disjoint subtrees $\mathcal{T}_{e_i}, \mathcal{T}_{e_{i+1}} - \mathcal{T}_{e_i}, \cdots, \mathcal{T}_{e_k} - \mathcal{T}_{e_{k-1}}$, and invoking FullEnum for each subtree separately. Finally, we join these subtrees together. The detailed process is given in Algorithm 5. Note that, as written, the algorithm does not achieve constant-delay enumeration. However, this can be easily fixed. First, the join in line 4 can be enumerated with constant delay using (a variant of) FullEnum starting from t'. Then we interleave the two enumeration processes: After enumerating each $q \in S$, we immediately call line 6–8. Finally, line 6–8 can be rewritten into nested loops so as to enumerate the join $S_i \Join \cdots \Join S_k$ with constant delay. In fact, this join is more like a cross product (common attributes must have the same value, the same as those in q), and a total of $\Pi_{j=i}^{k} |S_j|$ results will be yielded.

Lemma 13. *Algorithm 5 enumerates $\Delta Q(D,t)$ with constant delay.*

We have now closed the loop: While enumerating $\Delta Q(D, t)$, we update the live views as described earlier, which are needed for enumerating the next delta.

5.3 A Running Example in Figure 3

Initialization. Figure 3(a) shows the initial state of our index. For R_1 and R_4, their semi-join views are themselves and their projection views are simply the projections with counts. $V_s(R_2)$ consists of tuples in R_2 that can join with $V_p(R_1)$, which include $(2, 2)$ and $(2, 4)$.Similarly, $V_s(R_3)$ include 4 tuples from R_3. For the generalized node $[x_3]$, we define the virtual relation $R([x_3]) = V_p(R_2) \cup V_p(R_3)$. Only tuple (4) belongs to $V_s([x_3])$, since every other tuple in $R([x_3])$ fails to join with both $V_p(R_2)$ and $V_p(R_3)$: their counters need to be 2. Tuple $(4) \in V_s([x_3])$ is a live tuple. There are two query results initially: $(1, 2, 4, 4)$ and $(2, 2, 4, 4)$, which make $(1, 2), (2, 2) \in R_1$, $(2, 4) \in R_2$, $(4, 4) \in R_3$, $(4) \in \pi_{x_4} R_4$ as live tuples.

An insertion. Figure 3(b) shows the index after inserting $(1, 1)$ into R_1. This new tuple first triggers an insertion to $V_p(R_1)$, which further increments counters of the three tuples in $V_s(R_2)$ with $x_2 = 1$, which are then brought into $V_s(R_2)$. From here, the propagation diverges into three paths. Tuple $(1, 2) \in R_2$ increments the counter of $(1) \in V_p(R_2)$ but this propagation path stops here. As $(1, 2) \in R_2$ cannot join with any tuple in $V_l([x_3])$, it is not a witness tuple and no delta is produced. Tuple $(1, 1) \in V_s(R_2)$ first inserts a new tuple (1) to $V_p(R_2)$, which then further increments the counter of tuple (1) in the root, bringing it to $V_s([x_3])$. Tuple $(1) \in [x_3]$ is then a witness, which triggers delta enumeration. For a witness in the root, DeltaEnum simply degenerates to FullEnum$(\mathcal{T}_r, r, (1))$, which outputs $\{(1, 1, 1, 1), (1, 1, 1, 2)\}$. Finally, tuple $(1, 4) \in R_2$ increments the counter of $(4) \in V_p(R_2)$ and the propagation stops. As $(1, 4) \in R_2$ can join with $V_l([x_3])$, it is a witness. For this witness, DeltaEnum finds $S = (1, 4) \bowtie V_l([x_3]) = \{(1, 4)\}$. For $(1, 4) \in S$, it invokes FullEnum$(\mathcal{T}_r - \mathcal{T}_{R_2}, r, (4))$, which returns $\{(4, 4)\}$, and FullEnum$(\mathcal{T}_{R_2}, R_2, (1, 4))$, which returns $\{(1, 1, 4)\}$. Joining them yields the delta $\{(1, 1, 4, 4)\}$. Finally, as each new result is enumerated, we update the live views.

A deletion. Figure 3(c) shows the index after deleting $(1, 1)$ from R_4. This deletion first decrements the counter of tuple $(1) \in V_p(R_4)$, removing it from $V_p(R_4)$, and further decrements the counter of $(1, 1) \in V_s(R_3)$, removing it from $V_s(R_3)$ as well. Finally, the counter of $(1) \in V_p(R_3)$ decreases from 2 to 1, and the propagation stops here. For this deletion, $(1, 1) \in \Delta V_s(R_3)$ is a witness, as it joins with $V_l([x_3])$. For this witness, DeltaEnum first finds $S = \{(1, 1)\}$, and invokes FullEnum$(\mathcal{T}_r - \mathcal{T}_{R_3}, r, (1))$, which returns $\{(1, 1, 1)\}$, and FullEnum$(\mathcal{T}_{R_3}, R_3, (1, 1))$, which returns $\{(1, 1, 4)\}$ (note the delta enumeration upon a deletion is done on the database before the deletion so as to find the delta to be deleted). Joining them yields the delta $\{(1, 1, 1, 1)\}$. Finally, we update live views with the delta.

Another insertion. Figure 3(d) shows the index after inserting $(4, 1)$ into R_1. This insertion stops immediately at $V_s(R_1)$. As $(4, 1)$ can join with $V_l(R_2)$, it is a witness. DeltaEnum first computes $S = (4, 1) \bowtie V_l(R_2) \bowtie V_l([x_3]) = \{(4, 1, 1), (4, 1, 4)\}$. For $q = (4, 1, 1)$, it invokes FullEnum$(\mathcal{T}_{R_1}, R_1, (4, 1)) = \{(4, 1)\}$, FullEnum$(\mathcal{T}_{R_2} - \mathcal{T}_{R_1}, R_2, (1, 1)) = \{(1, 1)\}$, and FullEnum$(\mathcal{T}_r - \mathcal{T}_{R_2}, r, (1)) = \{(1, 2)\}$. Joining them yields the delta $\{(4, 1, 1, 2)\}$. For $q = (4, 1, 4)$, is invokes FullEnum$(\mathcal{T}_{R_1}, R_1, (4, 1)) = \{(4, 1)\}$, FullEnum$(\mathcal{T}_{R_2} - \mathcal{T}_{R_1}, R_2, (1, 4)) = \{(1, 4)\}$, and FullEnum$(\mathcal{T}_r - \mathcal{T}_{R_2}, r, (4)) = \{(4, 4)\}$. Joining them yields the delta $\{(4, 1, 4, 4)\}$.

6 UPDATE COST ANALYSIS

We have shown that the enumeration delay of both full query results and deltas is a constant, and this holds for the query plan defined by any free-connex join tree as in Section 4.1. On the other hand, the update cost differs for different query plans and can be as large as $O(|D|)$ in the worst case. This is caused by P-Update, which may trigger an S-Update to every tuple in its parent node. However, such a worst-case behavior only happens on contrived update sequences, and the actual update cost can be much better. Characterizing the update cost will be important for constructing a good query plan, as there can be many free-connex join trees for a given free-connex query. As we will see, the height of the join tree is an important parameter, and this is precisely the reason why we make our framework applicable to any generalized join tree, as the height of a generalized join tree can be lower than that of any standard join tree. For example, the query in Figure 2 has a generalized join tree of height 1 while the two standard join trees have height 2; the query in Figure 1 has a generalized join tree of height 2 while any standard join tree has height as least 3.

6.1 Enclosureness

The notion of *enclosureness* was first introduced in [34] to give an instance-specific characterization of the hardness of the update sequence, which we briefly review next.

Definition 14 (Enclosureness). Given an update sequence S_D, the *enclosureness* of a tuple $t \in S_D$ is

$$\lambda(t) := \max_{\substack{\mathcal{J} \subseteq S_D \\ \forall t_1 \in \mathcal{J}, I(t_1) \subset I(t) \\ \forall t_2, t_3 \in \mathcal{J}, I(t_2) \cap I(t_3) = \varnothing}} |\mathcal{J}|, \tag{7}$$

i.e., the largest number of disjoint lifespans in S_D contained in $I(t)$. Then the enclosureness of the update sequence is the average enclosureness of all the tuples (but at least 1), i.e.,

$$\lambda(S_D) := \max\left(\frac{\sum_{t \in S_D} \lambda(t)}{|S_D|}, 1\right). \tag{8}$$

We often omit S_D and simply write $\lambda := \lambda(S_D)$ for the enclosureness of an update sequence.

Then, they give an algorithm that can update any foreign-key acyclic query in $O(\lambda)$ time for any S_D while supporting $O(1)$-delay enumeration. This is appealing, since while λ can be as large as $O(|S_D|)$ in the worst case, it is often a small constant for many common update sequences, including FIFO, FILO (first-in-last-out), and insertion-/deletion-only sequences. The worst-case situation only happens when there are many tuples with long lifespans joining with many tuples with short lifespans, something that is uncommon in practice (i.e., many big but ephemeral changes to the query).

However, their analysis crucially relies on the nice property of foreign-key acyclic queries, that their result size is at most linear, which is not the case for non-key joins. In fact, we show below that the $O(\lambda)$ update time is unachievable for free-connex queries, which follows from the negative result that we prove below:

Theorem 15. *Consider the query* $Q = R_1(x_1) \bowtie R_2(x_1, x_2) \bowtie R_3(x_2, x_3) \bowtie R_4(x_3, x_4) \bowtie R_5(x_4)$ *over a FIFO update sequence. If there is an algorithm for* Q *with update time* $O(|D|^{1/2-\epsilon})$ *while supporting* $O(|D|^{1-\epsilon})$*-delay enumeration of full results for any constant* $\epsilon > 0$*, then the OuMv conjecture*[5] *fails.*

Note that this theorem separates the difficulty of (at least one of) free-connex queries from foreign-key acyclic queries, for which $O(1)$ update time is possible for FIFO sequences [34].

6.2 Join-tree-specific Enclosureness

Hope is not all lost despite the negative result above. First, Theorem 15 only holds for a particular free-connex query; other queries may still be updates in $O(1)$ time. Secondly, the definition of enclosureness in [34] only considers the time dimension while ignores the structure dimension, i.e., which relation each update is applied to. These observations motivate a more refined definition of enclosureness that also depends on the join tree (which nodes the updates are applied to). As we will see, a hard query like the one in Theorem 15 can still be solved efficiently, when information from both the structure dimension and the time dimension is taken into account.

Definition 16 (Effective lifespan). Given a free-connex query Q, a free-connex generalized join tree $\mathcal{T}$ of Q, a database D, and an update sequence S_D, the two *effective lifespans* of a tuple $t_1 \in R_e$ with lifespan $I(t_1) = [t_1^+, t_1^-]$ are

$$\widehat{I}(t_1) = \left[t_1^+, \min\left(t_1^-, \min_{t_2 \in R_{e'}: e' \in \mathcal{T}_e - \{e\}, t_2^- > t_1^+} t_2^- \right) \right];$$

$$\check{I}(t_1) = \left[\max\left(t_1^+, \max_{t_2 \in R_{e'}: e' \in \mathcal{T}_e - \{e\}, t_2^+ < t_1^-} t_2^+ \right), t_1^- \right].$$

In plain language, $\widehat{I}(t_1)$ is obtained from $I(t_1)$ by moving forward its ending time to the first deletion of a tuple from any descendent of e, while to obtain $\check{I}(t_1)$, we move its starting to the last insertion from any descendent of e.

We can now define the join-tree-specific enclosureness of a tuple:

Definition 17. Given a free-connex query Q, a free-connex generalized join tree $\mathcal{T}$ of Q, a database D, and an update sequence S_D, for a node $e \in \mathcal{T}$ and a tuple $t \in R_e$, its *enclosureness* is

$$\lambda_{\mathcal{T}}(t) = \max_{\substack{\mathcal{J} \subseteq \mathcal{T}_e - \{e\} \\ \forall t_1 \in \mathcal{J}, \dot{I}(t_1) \subseteq I(t) \\ \forall t_2, t_3 \in \mathcal{J}, \dot{I}(t_2) \cap \dot{I}(t_3) = \emptyset}} |\mathcal{J}|, \tag{9}$$

where each $\dot{I}$ is either $\widehat{I}$ or $\check{I}$, i.e., the largest number of disjoint effective lifespans of tuples in the descendants of e, which are contained in the lifespan of t. Then the enclosureness of the update sequence is still the average:

$$\lambda_{\mathcal{T}}(S_D) := \max\left(\frac{\sum_{t \in S_D} \lambda_{\mathcal{T}}(t)}{|S_D|}, 1 \right).$$

We often write $\lambda_{\mathcal{T}} := \lambda_{\mathcal{T}}(S_D)$ for the enclosureness of an update sequence with respect to $\mathcal{T}$.

[5]The OuMv conjecture [20] is that the following problem cannot be solved in $O(n^{3-\epsilon})$ time for any constant $\epsilon > 0$: Given an $n \times n$ matrix M and a sequence of n-dimensional vectors $u_1, v_1, u_2, v_2, \cdots, u_n, v_n$, compute $u_i M v_i$ for each i over the Boolean semiring. The algorithm must return $u_i M v_i$ before u_{i+1}, v_{i+1} are revealed.

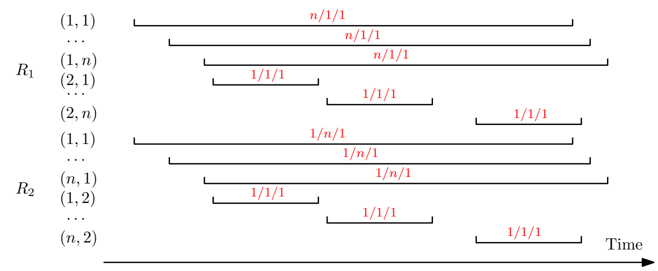


Figure 4: As update sequence in Example 18. Each interval is the lifespan of a tuple, and three numbers above each interval is its enclosureness over $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$ in Figure 2.

Example 18. Consider the query $Q := R_1(x_1, x_2) \bowtie R_2(x_2, x_3)$ in Figure 2 with $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$. For the update sequence in Figure 4, $\lambda_{\mathcal{T}_1} = \lambda_{\mathcal{T}_2} = n$ and $\lambda_{\mathcal{T}_3} = 1$. In fact, $\lambda_{\mathcal{T}_3} = 1$ for any update sequence.

The main analytical result of this paper is the following theorem, whose proof is quite technical (given in the full version [1]):

Theorem 19. *For any free-connex query* Q*, the update cost of the query plan in Section 4 induced by any given free-connex generalized join tree* $\mathcal{T}$ *of* Q *is* $O(\lambda_{\mathcal{T}})$ *under any update sequence.*

This result is complemented with a matching lower bound, for at least one particular query: $Q = \pi_{x_1}(R_1(x_1, x_2) \bowtie R_2(x_2))$, which has one join tree as shown in Figure 2(a) (one could add a generalized relation $[x_1]$ at the top but it does not change the enclosureness). Thus, for this query, $\lambda_{\mathcal{T}}$ does not really depend on $\mathcal{T}$.

Theorem 20. *Suppose there is an algorithm for the query* $Q = \pi_{x_1}(R_1(x_1, x_2) \bowtie R_2(x_2))$ *with update time* $O(\lambda^{1-\epsilon})$ *while supporting* $O(\lambda^{1-\epsilon})$*-delay enumeration of full results for any constant* $\epsilon > 0$*, then the OMv conjecture*[6] *fails.*

6.3 Implications of Enclosureness

We present some implications of our join-tree-specific enclosureness and Theorem 19, exhibiting an interesting trade-off between the hardness of update sequences and the complexity of queries.

Arbitrary update sequences. For arbitrary update sequences, prior work [9, 21] has shown how to achieve $O(1)$ update time while supporting $O(1)$-delay enumeration for any q-hierarchical query. It turns out that this is an easy consequence of Theorem 19, plus the following structural property of q-hierarchical queries, as well as the simple fact that $\lambda_{\mathcal{T}} = 1$ if the height of $\mathcal{T}$ is 1:

Lemma 21. *Every q-hierarchical query has a free-connex generalized join tree of height* 1.

For arbitrary update sequences, q-hierarchical queries are precisely the class of queries for which $O(1)$ update time is possible [9]. Thus, for queries outside this class, we must restrict the update sequence in order to achieve $O(1)$ update time. We consider the following two classes of update sequences.

FIFO sequences. The update time is shown to be $O(1)$ for foreign-key acyclic joins over FIFO sequences [34], but nothing is known

[6]The OMv conjecture is similar to the OuMv conjecture, except that the algorithm needs to compute Mv_i for every v_i.

for non-key joins (except for q-hierarchical queries, which do not rely on the FIFO property). We present the first extension in this direction:

Lemma 22. *For any free-connex query Q with a free-connex generalized join tree $\mathcal{T}$ of height at most 2, $\lambda_{\mathcal{T}} = 1$ for any FIFO update sequence.*

Note that the height limit of 2 is the best one can hope for, since the query in Theorem 15 has a join tree of height 3 and the theorem shows that it cannot be updated in $O(1)$ time over FIFO sequences. Although the height-2 limitation restricts the class of queries, this already includes some fairly complex queries, such as the one in Figure 1; more examples can be found in Section 8.

Example 23. Consider an insertion-only update sequence for the query in Figure 1: (1) tuples $(i, j) \in [n] \times [n]$ are inserted into R_2, R_3 and R_4 initially; (2) tuples $(i, j) \in [n] \times [n]$ are inserted into R_1 later. Standard change propagation or HIVM needs to materialize $\Delta(R_1 \bowtie R_2 \bowtie R_3)$, hence incurs amortized $O(n^3)$ cost; Dynamic Yannakakis [21] needs to scan all tuples $(i', j') \in R_2$ for $i' = j$, once (i, j) is inserted into R_1, hence incurs $O(n)$ cost; and our framework incurs amortized $O(1)$ cost, since the update sequence has $\lambda = 1$.

Insertion-only sequences. As we restrict the update sequence further, we can handle more queries in $O(1)$ time. For simplicity, the following result only considers insertion-only sequences, but the same result holds for deletion-only or FILO sequences as well.

Lemma 24. *For any free-connex query Q and any join tree $\mathcal{T}$, $\lambda_{\mathcal{T}} = 1$ for any insertion-only update sequence.*

Note that Lemma 24 incorporates the static result [7] as a special case. Given a static database D, we can simply insert every tuple from D into our query plan. By Lemma 24, this builds a data structure in $O(|D|)$ time that supports $O(1)$-delay enumeration of $Q(D)$. Also, the dichotomy result of [7] states that $O(|D|)$-time preprocessing and $O(1)$-delay enumeration are possible only for free-connex queries, thus Lemma 24 cannot be extended to beyond free-connex queries, either, even over insertion-only sequences.

Query plan optimization. If the given query and/or the update sequence do not fall into any of the three cases above where $O(1)$ update time can be guaranteed, our enclosureness analysis still yields an effective heuristic for choosing a good $\mathcal{T}$, which in turn determines the query plan. First, it is clear that a join tree with a smaller height is always preferred. Furthermore, Definition 17 suggests that we should put nodes with more updates higher in $\mathcal{T}$, as a tuple in a node might increase the enclosureness of tuples in its ancestors. Thus, in our implementation, we construct all join trees and use the one that minimizes $\sum_{e \in \mathcal{T}} d(e)N(e)$, where $d(e)$ is the depth of e in $\mathcal{T}$ (not counting generalized relations and itself) and $N(e)$ is the number of updates to e. If $N(e)$ is unavailable, we estimate it by observing (and buffering) the first few updates, noting that our target only depends on the relative update frequencies.

7 EXTENSIONS

The query plan in Section 4.1 works for CQs with joins and projections, but it can be equipped with other operators easily, such as selection, union, set difference, and aggregations. Due to the space constraints, we leave this part in the full version [1].

Beyond non-free-connex queries, constant-delay enumeration is not possible even in the static case [7]. Nevertheless, they can still be supported in our framework with linear space, $O(\lambda)$ update time, although the delay is not guaranteed. For an acyclic but non-free-connex query, e.g., $\pi_{x_1,x_3} R_1(x_1, x_2) \bowtie R_2(x_2, x_3)$, we simply add x_2 as another output attribute to turn it into a free-connex query, and then do a projection π_{x_1,x_3} during enumeration, or just don't do anything if duplicates are allowed (note that in SQL, duplicates are not removed unless using the `DISTINCT` keyword explicitly). When the query is cyclic, the problem is wide open; see [24] for some initial theoretical results on the triangle query, which shows that even this simplest cyclic query requires $O(\sqrt{|D|})$ update time, and the presented algorithm does not seem to be practical.

8 EXPERIMENTS

8.1 Setup

Prototype implementation. We have implemented our algorithms and built a system prototype called CROWN (**C**hange p**RO**pagation **W**ithout joi**N**s) [1] on top of Flink DataStream API. All of our algorithms are implemented as DataStream functions, which take as input an update stream. Each tuple in the update stream is associated with a flag indicating whether the update is an insertion or deletion, as well as the name of the updated relation. After processing an update, the DataStream function outputs the deltas triggered by this update. Enumeration of full query results can be invoked upon the user's request. Implementing the prototype over Flink allows us to inherit all the benefits of Flink, such as fault-tolerance and the ability to work with a variety of data sources and sinks. To dispatch tuples in a load-balanced fashion, we borrow a similar idea from massively parallel algorithms, such as HyperCube [4, 8, 34].

We have evaluated our algorithms in both centralized and distributed settings. The centralized version runs on a single machine with a single thread, where we disable certain Flink features such as false tolerance, serialization, and dispatching. This is for a fair comparison with other centralized systems (DBToaster and Trill) that do not support these features. The distributed version has all these features enabled. It runs over two machines, each equipped with two Intel Xeon 2.1GHz processors with 48 cores and 416 GB memory. The machine runs Linux, with Scala 2.11.12, dotnet 5.0.403, Flink 1.13.5 and Spark 2.2.3. Each query is evaluated 10 times on each engine and we report the average running time. We set a 4-hour time limit for each run.

Query processing engines compared. We compare our algorithms with (1) DBToaster [5], the best HIVM engine that supports multi-way joins over arbitrary update streams in centralized setting; (2) DBToaster Spark [29], which can support IVM with batch update in a distributed/parallel setting; (3) Trill [12], a continuous query evaluation system over streaming data using the standard change propagation framework; and (4) the native Flink SQL engine over streaming data.

Table 1 summarizes various features of these systems. Note that only CROWN supports both full enumeration and delta enumeration. Flink can support insertion-only update streams or window

	CROWN	Flink	DBToaster	DBToaster Spark	Trill
Distributed	✓	✓		✓	
Full enumeration	✓	✓	✓	✓	
Delta enumeration	✓				✓
Updates	Arbitrary	FIFO	Arbitrary	Batch	Arbitrary
Internal	This paper	Standard change propagation	HIVM	HIVM	Standard change propagation

Table 1: Comparison of different query processing engines.

streams, but not arbitrary update streams. We run every experiment twice: one for delta enumeration, and the other for full enumeration. For full enumeration, we request the full query results after processing every 10% of the update sequence. As Trill does not support full enumeration, we ask Trill to report the entire delta stream for full enumeration.

Queries and updates. We evaluate all systems over two classes of queries. The first class contains graph pattern queries over the SNAP dataset (Stanford Network Analysis Project) [27], such as 3-Hop, 4-Hop, and star. One example of the 3-Hop query is given below, where we use a filter over to control the output size.

```
SELECT G1.src as A, G2.src as B, G3.src as C, G3.dst as D
FROM G G1, G G2, G G3
WHERE G1.dst = G2.src AND G2.dst = G3.src
AND FILTER OVER (G3.dst)
```

The second class includes more complex analytical queries over the LDBC Social Network Benchmark (LDBC-SNB) [16], which access the neighborhood of a given node in the graph with continuous updates. The following shows one example, which finds the number of distinct messages associated with a particular tag ID, while satisfying the filter conditions:

```
SELECT t_name, t_tagid, COUNT(DISTINCT m_messageid)
FROM tag, message, message_tag, knows
WHERE m_messageid = mt_messageid AND mt_tagid = t_tagid
  AND m_creatorid = k_person2id AND m_c_replyof IS NULL
  AND FILTER OVER (k_person1id)
GROUP BY t_name, t_tag_ids
```

Figure 5 shows the join hypergraphs of all queries. Except for SNB Q3 and Q4, they have a free-connex generalized join tree of height at most 2. In addition, the star query (figure 5(c)) has a height-1 free-connex generalized join tree, so it is q-hierarchical. Note that 4-Hop query (figure 5(b)) and SNB Q4 query (figure 5(e)) have the same hypergraph structure but different output attributes, thus the 4-Hop query has a free-connex generalized join tree with height 2, while SNB Q4 query does not.

We create FIFO streams for queries with a parameter w. For graph queries, we assign a distinct integer t_e to each edge e in the graph, where e has its lifespan $[t_e, t_e + w]$. For LDBC-SNB queries, each tuple t in the benchmark already has an insertion timestamp t^+, and we set its deletion time as w days after its insertion, i.e., $t^- = t^+ + w$. Note that the sliding window for graph queries is count-based, i.e., the window always contains the same number of tuples. On the other hand, the window for LDBC-SNB queries are time-based, so that the number of tuples in a window fluctuates over time.

8.2 Experiment Results

Run-time comparison. Figure 6 shows the total running time of evaluating each graph queries over a mid-sized graph *Epinions* in the centralized setting. We set a filter condition that only keeps 10% of the designated endpoints for all full join queries. A missing bar in the figure indicates that the corresponding system did not finish within the 4-hour limit or aborted with an error (mostly out-of-memory errors and garbage collection timeout). Only CROWN is able to finish all queries successfully. Trill can only handle a few graph queries. One possible explanation is that graph queries tend to generate a large number of deltas. On the other hand, Flink ran out of memory when evaluating SNB Q2, Q3, and Q4. For those queries where the systems can finish, we see that CROWN provides a speedup from 2x to 50x compared with Flink, 1.2x to 88x compared with DBToaster, and 2.4x to 196x compared with Trill. Moreover, in handling non-full queries, CROWN requires much less time than handling full queries, while Flink requires more time. In addition, CROWN performs well for both full and delta enumeration, where different modes of output do not affect the overall performance of CROWN.

Selectivity. Figure 7 shows the run-time when varying the selectivity of join conditions. For standard change propagation and HIVM, the maintenance cost does not only depend on the input and output size, but also the size of the intermediate views. For the 3-Hop query $G_1(A,B) \bowtie G_2(B,C) \bowtie G_3(C,D)$ for $G_1 = G_2 = G$ and $G_3 = \textsc{Filter}(G)$, the maintenance cost will be bounded by the size of the view $G_1 \bowtie G_2$ even when G_3 is empty. In the meantime, the maintenance cost of CROWN only depends on the input and output size. To better show such a property, we adjust the filter condition in the 3-Hop query, which only changes $|G_3|$ instead of $|G_1 \bowtie G_2|$. Trill is omitted here as it exceeded the 4-hour limit for all data points except for the first one. When $|G_3| \geqslant 0.5\%|G|$, the output size exceeds the input size; and when $|G_3| \geqslant 20\%|G|$, the output size exceeds the intermediate join size $|G_1 \bowtie G_2|$. From the results, we can see the run-time of CROWN scales almost linearly as $|G| + |Q|$, which is as expected since the update sequence has $\lambda = 1$. On the other hand, the run-time of the DBToaster and Flink scales proportionally to $|G_1 \bowtie G_2| + |Q|$, which leads to poor performance when $|G_3| \leqslant 20\%|G|$. A larger gap can be observed in Figure 8 when evaluating the 4-Hop query with projection, where the intermediate join size exceeds the size of the query results, even without any filter conditions. The run-time of Flink and DBToaster on the 4-Hop query exceeds the 3-Hop query, even with a small output size. Meanwhile, the run-time of CROWN is much smaller, which only depends on the input and output size.

Enclosureness. To test the influences of enclosureness, we create multiple update sequences with different λ, over different graphs from SNAP dataset. We disable the output to see how the update cost would change with different λ. The experiment results are shown in Figure 9. From the results, we can see the maintenance cost of CROWN increases almost linear as λ increases.

Distributed processing. To test CROWN's performance in a distributed setting, we built a small cluster with 32 task slots and compared the performance amount DBToaster Spark, Flink, and CROWN. We tested 4-Hop and SNB Q3 query, on which DBToaster

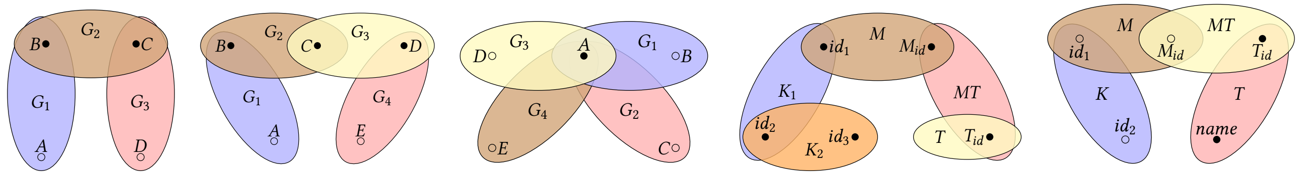


Figure 5: The relational hypergraphs of queries. The solid dots are output attributes for non-full and aggregation queries.

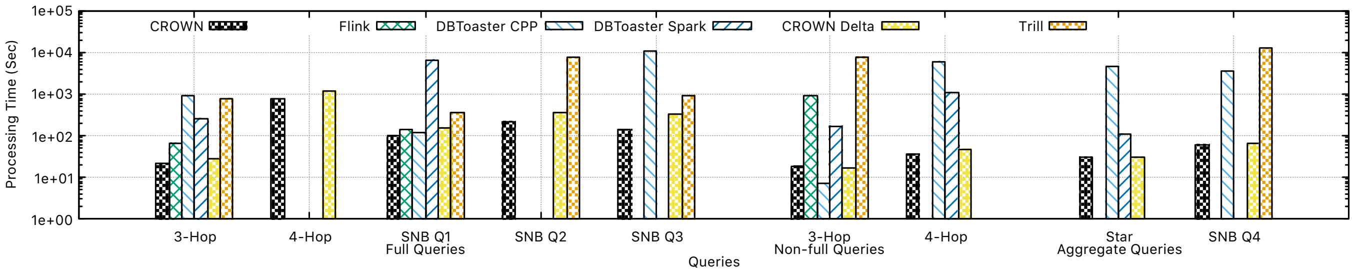


Figure 6: Processing times of CROWN, Flink, DBToaster and Trill

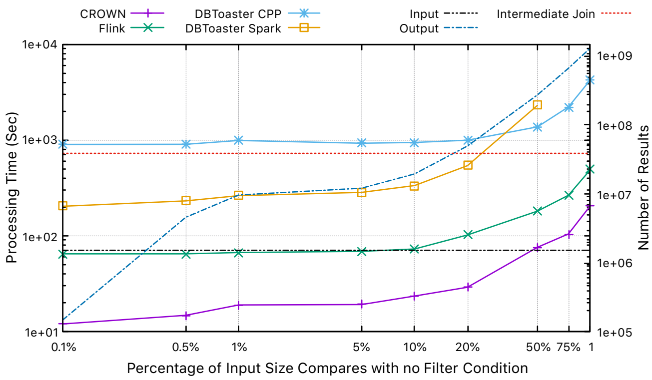


Figure 7: Run-time v.s. selectivity on 3-Hop query.

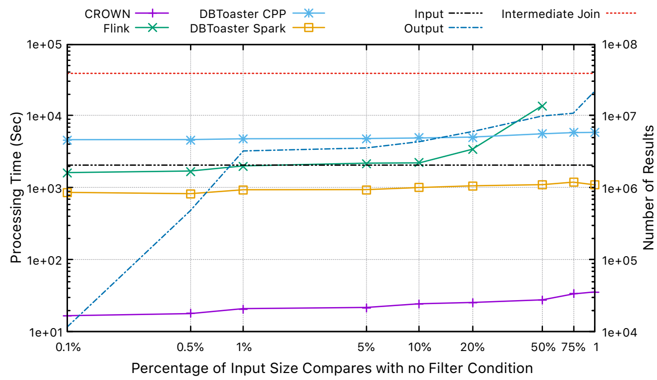


Figure 8: Run-time v.s. selectivity on 4-Hop query.

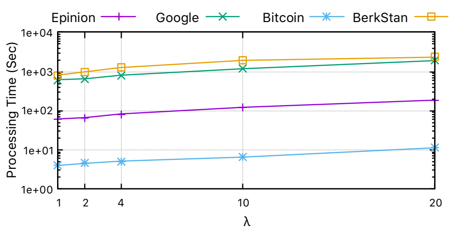


Figure 9: Run-time v.s. enclosureness λ.

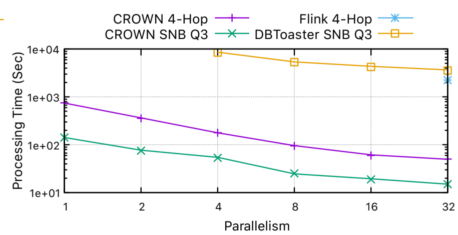


Figure 10: Run-time v.s. parallelism p.

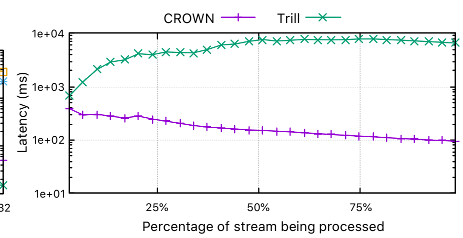


Figure 11: Average latency.

and Flink cannot finish in a centralized setting. Figure 10 shows the results; missing data points or lines indicate the system cannot finish within the time limit. Although we adopt the HyperCube algorithm to dispatch all tuples, CROWN can still obtain linear speedup with $p < 16$, where p is the number of workers. When more workers are available, the margin gain becomes smaller. This is as expected, since (1) speedup becomes sublinear when adding more workers implied by HyperCube; (2) the processing time is already short, causing the system's overhead to dominate the entire run-time. For all finished data points, CROWN can provide a speedup from 45x to 245x.

As Flink and DBToaster cannot finish all experiments with 128GB memory, so we increase the memory usage for these two systems to 500GB, where these two systems still only complete a tiny portion of the experiments. On the other hand, CROWN can finish all experiments with only 128GB memory. If we further limit the memory usage of CROWN to 16GB, i.e., 500MB per worker, CROWN still work well without much change in its performance.

Latency. Finally, We tested the latency of delta enumeration, i.e., the time between an update is received and the its deltas are outputted. The result is shown in Figure 11. The average latency of CROWN is less than 100ms, while the average latency of Trill is more than 6s. In addition, the average latency is stable for CROWN, but it keeps growing for Trill, making it infeasible to process streams for long periods.

REFERENCES

[1] [n.d.]. Change Propagation without Joins: codes and full version. https://anonymous.4open.science/r/SIGMOD2023-1D56/.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases.* Addison-Wesley Longman Publishing Co., Inc.

[3] Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. 2016. FAQ: questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems.* 13–28.

[4] Foto N. Afrati and Jeffrey D. Ullman. 2011. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE Transactions on Knowledge and Data Engineering* 23, 9 (2011), 1282–1298.

[5] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *Proceedings of the VLDB Endowment* 5, 10 (2012), 968–979.

[6] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size bounds and query plans for relational joins. *SIAM J. Comput.* 42, 4 (2013), 1737–1767.

[7] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic*. Springer, 208–222.

[8] Paul Beame, Paraschos Koutris, and Dan Suciu. 2013. Communication Steps for Parallel Query Processing. In *Proc. ACM Symposium on Principles of Database Systems.*

[9] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. 2017. Answering Conjunctive Queries under Updates. In *Proc. ACM Symposium on Principles of Database Systems.*

[10] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* 38, 4 (2015), 28–38.

[11] Nofar Carmeli and Markus Kröll. 2019. On the Enumeration Complexity of Unions of Conjunctive Queries. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems.* ACM, 134–148.

[12] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. 2014. Trill: A high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment* 8, 4 (2014), 401–412.

[13] Rada Chirkova and Jun Yang. 2012. Materialized views. *Foundations and Trends® in Databases* 4, 4 (2012), 295–405.

[14] Arnaud Durand. 2020. Fine-Grained Complexity Analysis of Queries: From Decision to Counting and Enumeration. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems.* 331–346.

[15] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. 2014. Scalable and adaptive online joins. VLDB.

[16] Orri Erling, A. Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, A. Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of ACM SIGMOD International Conference on Management of Data 2015 (SIGMOD'15).*

[17] R. Fagin. 1983. Degrees of acyclicity for hypergraphs and relational database schemes. *JACM* 30, 3 (1983), 514–550.

[18] Buğra Gedik, Rajesh R Bordawekar, and Philip S Yu. 2009. CellJoin: a parallel stream join operator for the cell processor. *The VLDB journal* 18, 2 (2009), 501–519.

[19] Timothy Griffin and Bharat Kumar. 1998. Algebraic Change Propagation for Semijoin and Outerjoin Queries. *SIGMOD Rec.* 27, 3 (Sept. 1998), 22–27. https://doi.org/10.1145/290593.290597

[20] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. 2015. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing.* 21–30.

[21] Muhammad Idris, Martin Ugarte, and Stijn Vansummeren. 2017. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *Proc. ACM SIGMOD International Conference on Management of Data.*

[22] Manas R. Joglekar, Rohan Puttagunta, and Christopher Ré. 2016. AJAR: Aggregations and Joins over Annotated Relations. In *Proc. ACM Symposium on Principles of Database Systems.*

[23] Jaewoo Kang, Jeffrey F Naughton, and Stratis D Viglas. 2003. Evaluating window joins over unbounded streams. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405).* IEEE, 341–352.

[24] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2020. Maintaining Triangle Queries under Updates. *ACM Trans. Database Syst.* 45, 3, Article 11 (aug 2020), 46 pages. https://doi.org/10.1145/3396375

[25] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2020. Trade-offs in static and dynamic evaluation of hierarchical queries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems.* 375–392.

[26] Ki Yong Lee, Jin Hyun Son, and Myoung Ho Kim. 2001. Efficient Incremental View Maintenance in Data Warehouses. In *Proceedings of the Tenth International Conference on Information and Knowledge Management* (Atlanta, Georgia, USA) *(CIKM '01).* Association for Computing Machinery, New York, NY, USA, 349–356. https://doi.org/10.1145/502585.502644

[27] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[28] Qian Lin, Beng Chin Ooi, Zhengkui Wang, and Cui Yu. 2015. Scalable distributed stream join processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* 811–825.

[29] Milos Nikolic, Mohammad Dashti, and Christoph Koch. 2016. How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates. In *Proc. ACM SIGMOD International Conference on Management of Data.* ACM, 511–526.

[30] Milos Nikolic and Dan Olteanu. 2018. Incremental view maintenance with triple lock factorization benefits. In *Proc. ACM SIGMOD International Conference on Management of Data.* ACM, 365–380.

[31] Milos Nikolic, Haozhe Zhang, Ahmet Kara, and Dan Olteanu. 2020. F-IVM: learning over fast-evolving relational data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 2773–2776.

[32] Kenneth A Ross, Divesh Srivastava, and S Sudarshan. 1996. Materialized view maintenance and integrity constraint checking: Trading space for time. In *ACM SIGMOD Record*, Vol. 25. ACM, 447–458.

[33] Pratanu Roy, Jens Teubner, and Rainer Gemulla. 2014. Low-latency handshake join. *Proceedings of the VLDB Endowment* 7, 9 (2014), 709–720.

[34] Qichen Wang and Ke Yi. 2020. Maintaining Acyclic Foreign-Key Joins under Updates. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 1225–1239.

[35] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *Proc. International Conference on Very Large Data Bases.* 82–94.

A MISSING PROOFS IN SECTION 3

Lemma 25 ([2, 17]). *A query is α-acyclic if it has a join tree.*

Lemma 26. *Given a query with a generalized join tree $\mathcal{T}$, there exists a join tree $\mathcal{T}'$ without generalized relation for the query, i.e., the query is α-acyclic query.*

Proof. It suffices to show that any generalized join tree $\mathcal{T}$ can be transformed into a tree $\mathcal{T}'$ satisfying the following properties: (i) there is a one-to-one mapping between input relations and nodes in $\mathcal{T}'$; (ii) for each attribute x, all nodes of $\mathcal{T}'$ containing x form a connected component of $\mathcal{T}'$. If $\mathcal{T}$ does not contain any generalized relations, then we are done, since (1)-(2) in Definition 1 already implies (i)-(ii).

We start with the base case that $\mathcal{T}$ only contains one generalized relation, say $\hat{R}$. Implied by (3) in Definition 1, $\hat{R}$ must be the root of $\mathcal{T}$. In this case, we can arbitrary pick one of its child relations to be the root, say R_c, and make all R_c's siblings to be the child node of R_c. It is clear that the resulted tree $\mathcal{T}'$ is a join tree, since both (i) and (ii) are preserved automatically.

In general, we choose a subtree $\mathcal{T}''$ that only contains one generalized relation which must also be the root of $\mathcal{T}''$, and perform local transformation on $\mathcal{T}''$ as we have done in the base case. Then, there is no generalized relation in $\mathcal{T}''$. We repeat this procedure until all generalized relations are removed. It can be easily checked that (ii) is always preserved in the whole procedure. After all generalized relations are removed, the resulted tree must be a join tree, as both (i) and (ii) are satisfied. □

B MISSING PROOFS IN SECTION 5

Proof of Lemma 6. We prove it by the induction on the height of generalized join tree $\mathcal{T}$. First, it holds for any leaf node e, since $V_s(R_e) = R_e$. We next consider an arbitrary internal node e. Let $C_e = \{e_1, e_2, \cdots, e_k\}$ be the set of children of node e. By hypothesis, we assume this lemma holds for every $e_i \in C_e$, i.e.

$$V_s(R_{e_i}) = \pi_{e_i}(\Join_{e' \in \mathcal{T}_{e_i}} R_{e'})$$

Then, we can rewrite $V_s(R_e)$ as follows:

$$\Leftrightarrow R_e \ltimes V_p(R_{e_1}) \ltimes \cdots \ltimes V_p(R_{e_k})$$
$$\Leftrightarrow R_e \Join V_p(R_{e_1}) \Join \cdots \Join V_p(R_{e_k})$$
$$\Leftrightarrow R_e \Join \left(\pi_{\mathsf{key}(e_1)} V_s(R_{e_1})\right) \Join \cdots \Join \left(\pi_{\mathsf{key}(e_k)} V_s(R_{e_k})\right)$$
$$\Leftrightarrow R_e \Join \left(\pi_{\mathsf{key}(e_1)} (\Join_{e' \in \mathcal{T}_{e_1}} R_{e'})\right) \Join \cdots \Join \left(\pi_{\mathsf{key}(e_k)} (\Join_{e' \in \mathcal{T}_{e_n}} R_{e'})\right)$$
$$\Leftrightarrow \pi_e(R \Join_{e' \in \mathcal{T}_e} R_{e'}))$$

where the first equation follows the definition of semi-join views, the second equation follows the fact that $\mathsf{key}(e_i) = e_i \cap e \subseteq e$, the third equation follows the definition of projection views, the fourth equation follows the hypothesis, and the last equation follows the facts that $\mathcal{T}_e = \{e\} \cup \mathcal{T}_{e_1} \cup \mathcal{T}_{e_2} \cup \cdots \cup \mathcal{T}_{e_k}$ and $\mathsf{key}(e_i)$ is exactly the set of join attributes shared by R_e and $(\Join_{e' \in \mathcal{T}_{e_i}} R_{e'})$. □

Proof of Lemma 9. We prove it by induction on the height of $\mathcal{T}$. The algorithm will stop if the root contains no output attributes, for ease of expression, we assume all nodes e with $e \cap \mathbf{y} = \emptyset$ are removed from $\mathcal{T}$. We first establish a based case, in which $\mathcal{T}$ contains only one node. The algorithm returns $\pi_{\mathbf{y} \cap e} V_s(R_e)$ in $O(|\pi_{\mathbf{y} \cap e} V_s(R_e)|)$ time, since all the tuples in $\pi_{\mathbf{y} \cap e} V_s(R_e)$ can be enumerated in $O(1)$ delay. Hence, this base case can be handled with with $O(1)$ delay.

In general, we have the hypothesis holds on all child nodes e_i of e (line 7-10): Algorithm 4 can enumerate all join results that agree with values $\pi_{\mathsf{key}(e_i)} t'$ over attributes $\mathsf{key}(e_i)$ in the subtree $\mathcal{T}_{e_i}$. Let t_i be a join result returned from $\mathcal{T}_{e_i}$. From the properties of join tree, line 10 will return a valid join result. Emitting every combination of join results over all subtrees of e_1 just takes $O(1)$ time. □

Proof of Lemma 12. Without loss of generality, we assume that t is inserted. The case that t is deleted follows the same argument.

Direction $\supseteq$. We show that each result in $Q(D \ltimes t')$ also appears in $\Delta Q(D, t)$, for every witness tuple t' of t. Wlog, consider a query result $q \in Q(D \ltimes t')$ for some witness tuple t', where either $t' \in R_e$ for $e \subseteq \mathbf{y}$ or $t' \in \pi_{\mathbf{y}} R_e$ for $e \cap \mathbf{y} - p(e) \neq \emptyset$.

First, $Q(D \ltimes t') \subseteq Q(D + t)$ since we have $t' \in \Delta\left(\pi_{\mathbf{y}} V_s(R_e)\right)$ and $\pi_{\mathsf{key}(e)} t' \in V_p(R_e)$ after the insertion of t. Hence, all results witnessed by t' appear in $Q(D + t)$ after the insertion of t, i.e., $q \in Q(D + t)$. We next show $q \notin Q(D)$. Now let's go back to the timestamp before the insertion of t. Implied by the definition of witness tuple, $t' \notin \pi_{\mathbf{y}} V_s(R_e)$ then. We distinguish two more cases.

- Case 1: $e \subseteq \mathbf{y}$, $q \notin Q(D)$ since $\pi_e q = t'$ but $t' \notin \pi_e Q(D)$ before the insertion of t. This further indicates $q \notin Q$.
- Case 2: $e - \mathbf{y} \neq \emptyset$ and $e \cap \mathbf{y} - p(e) \neq \emptyset$, $t' \notin \pi_{\mathbf{y}} V_s(R_e)$ before the insertion of t'. This way, $t' \notin \pi_{\mathbf{y}} Q(D)$, thus $q \notin Q$.

Combining the analysis above, we have $q \in Q(D+t)$, and $q \notin Q(D)$ i.e., $q \in \Delta Q(D, t)$. Thus, $\biguplus_{t': \text{a witness of } t} Q(D \ltimes t') \subseteq \Delta Q(D \ltimes t)$.

Direction $\subseteq$. We next show that every result in $\Delta Q(D, t)$ belongs to $Q(D+t) \ltimes t'$ for some witness tuple t' of t. Consider an arbitrary query result $q \in Q(D + t) - Q(D)$.

It suffices to show that there exists at least one node $e \in \mathcal{T}$ such that tuple $t' = \pi_e q$ if $e \subseteq \mathbf{y}$, or tuple $t' \in \pi_{\mathbf{y}} R_e$ with $t' = \pi_{e \cap \mathbf{y}} q$ if $e \cap \mathbf{y} - p(e) \neq \emptyset$, must be a witness. An important observation is that t' now belongs to $\Delta \pi_{\mathbf{y}} V_s(R_e)$; otherwise, $q \in Q(D)$, coming to a contradiction. Now consider the highest node e_1 such that $t_1 = \pi_{e_1 \cap \mathbf{y}} q$ and $t_1 \in \Delta\left(\pi_{\mathbf{y}} V_s(R_{e_1})\right)$. If e_1 is the root, t_1 must be a witness of t, implied by the Definition 11. Otherwise, e_1 is not the root. Consider $t_2 = \pi_{p(e_1) \cap \mathbf{y}} q$. As $t_2 \notin \Delta V_s(R_{e_2})$, t_2 must in $V_s(R_{e_2})$ and $t_2 \in \pi_{e_2} Q(D)$ before the insertion of t, which indicates the $\pi_{\mathsf{key}(e_1)} t_1 = \pi_{\mathsf{key}(e_1)} t_2 \in V_p(R_{e_1})$, and $\pi_{\mathsf{key}(e_1)} t_1 \in \pi_{\mathsf{key}(e_1)} Q(D)$. In this way, t_1 is a witness of t by definition.

Critical Property: $\Delta Q(D, t_1) \cap \Delta Q(D, t_2) = \emptyset$ holds for any pair of witness tuples t_1, t_2. It remains to show that there is no duplicate results in $\bigcup_{t': \text{a witness of } t} \Delta Q(D \ltimes t')$. By contradiction, assume that there exists a query result q with at least two witness tuples. Wlog, let t_1, t_2 be two distinct witness tuples in q, where $t_1 \in R_{e_1}$ for some $e_1 \subseteq \mathbf{y}$ or $e_1 \cap \mathbf{y} - p(e_1) \neq \emptyset$, and some $e_2 \subseteq \mathbf{y}$ or $e_2 \cap \mathbf{y} - p(e_2) \neq \emptyset$. First, $e_1 \neq e_2$, as q contains at most one tuple in each relation. Note that the insertion of $t \in R_e$ can only change the status of tuples in the ancestors of e. Without loss of generality, let e_1 be the ancestor of e_2. Let e_3 be parent node of e_2 (it could be the case that $e_1 = e_3$). Let $t_3 = \pi_{e_3 \cap \mathbf{y}} q$. Implied by the definition of witness tuples, $t_3 \in \pi_{\mathbf{y}} V_s(R_{e_3})$ before the insertion of t. Implied by $t_3 \notin \Delta \pi_{\mathbf{y}} V_s(R_{e_3})$, $t_1 \in \pi_{\mathbf{y}} V_s(R_{e_1})$ before the insertion, contradicting the fact that t_1 is a witness tuple. This way, each result in $\Delta Q(D, t)$ corresponds to one witness tuple, thus there is no duplicates across the extended query results over different witness tuples. $\square$

Proof of Lemma 13. We first show the correctness of Algorithm 5. Consider an arbitrary witness tuple $t' \in R_{e_1}$. Denote the nodes lying on the path from e_1 to r as $e_1, e_2, \cdots, e_k(r)$ sequentially. We can first expand $Q(D \ltimes t')$ as follows:

$$t' \bowtie \left(\bowtie_{i=1}^{k} V_r(e_i)\right) \bowtie Q_{\mathcal{T}_{e_1}} \bowtie \left(\bowtie_{i=2}^{k} Q_{\mathcal{T}_{e_i} - \mathcal{T}_{e_{i-1}}}\right) \quad (10)$$

where $Q_{\mathcal{T}}$ represents the query defined over relations in $\mathcal{T}$. Implied by the join operator and the properties of generalized join tree, we can further rewrite (10) =:

$$\bigcup_{S \in t' \bowtie (\bowtie_{i=1}^{k} V_r(e_i))} S \bowtie Q_{\mathcal{T}_{e_1}} \bowtie \left(\bowtie_{i=2}^{k} Q_{\mathcal{T}_{e_i} - \mathcal{T}_{e_{i-1}}}\right)$$

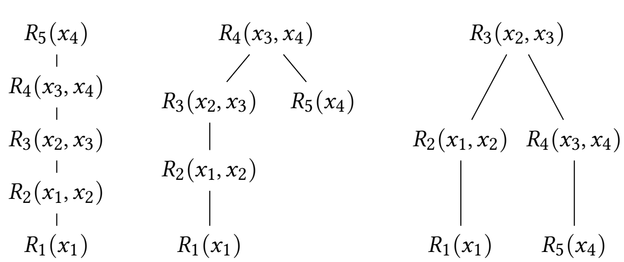
$$= \bigcup_{S \in t' \bowtie (\bowtie_{i=1}^{k} V_r(e_i))} S \times (Q_{\mathcal{T}_{e_1}} \ltimes \{S\}) \times \left(\bowtie_{i=2}^{k} (Q_{\mathcal{T}_{e_i} - \mathcal{T}_{e_{i-1}}} \ltimes \{S\})\right)$$

which is exactly followed by Algorithm 5. Together with Lemma 12, all results of $\Delta Q(D, t)$ are enumerated without duplication.

We next analyze the time complexity. As all witness tuples can be stored in a data structure (e.g., a linked list) supporting constant-delay enumeration, every t' (line 1) can be retrieved in $O(1)$ delay. It then suffices to show that $Q(D \ltimes t')$ can be enumerated with $O(1)$ delay for every t'. Note that subquery $t' \bowtie \left(\bowtie_{i=1}^{k} V_r(e_i)\right)$ (line 4) can be done in $O(1)$ delay with our hashing index. For the remaining subquery $Q_{\mathcal{T}_{e_1}} \ltimes \{S\}$ or $Q_{\mathcal{T}_{e_i} - \mathcal{T}_{e_{i-1}}} \ltimes \{S\}$, we invoke the procedure FullEnum (line 6-8) and all query results can be enumerated with $O(1)$ delay, proved by Lemma 9. Combing those subqueries in a form of Cartesian product can yield query results with $O(1)$ delay, thus completing the whole proof. $\square$

C MISSING MATERIALS IN SECTION 6

Proof of Theorem 15. Given an instance of OuMv, we encode the matrix by R_3 and vectors (v_i, u_i) by R_2 and R_4 separately. We construct an update sequence S for Q as follows: (1) we add a tuple



(a) $\mathcal{T}_1$ (b) $\mathcal{T}_2$ (c) $\mathcal{T}_3$

Figure 12: Join trees for $Q = R_1(x_1) \bowtie R_2(x_1, x_2) \bowtie R_3(x_2, x_3) \bowtie R_4(x_3, x_3) \bowtie R_5(x_4)$**.**

$t = (i, j)$ with lifespan $I(t) = [-k, k]$, for each pair $(i, j) \in [n] \times [n]$ if $M_{ij} \neq 0$; (2) we add a tuple $t = (i)$ with lifespan $I(t) = [i - 2k, i]$ into R_1 and R_5; (3) for each pair of vectors (v_i, u_i), we add a tuple $t = (i, v_{ij})$ with lifespan $I(t) = [i, i + 2k]$ to R_2 if $v_{ij} \neq 0$, and add a tuple $t = (u_{ij}, i)$ with lifespan $I(t) = [i, i + 2k]$ to R_4 if $u_{ij} \neq 0$; (4) if a query result is enumerated, we output true for $v_i^T M u_i$, and false otherwise; (5) we repeat (3)-(4) for the next pair (v_{i+1}, u_{i+1}), until n pairs of vectors are all processed. Each tuple in S has the same lifespan as $2k$, thus it is a FIFO sequence.

We note that in any generalized join tree $\mathcal{T}$ of Q, there always exists a subtree in which either $R_1 - R_2 - R_3$ or $R_5 - R_4 - R_3$ is a leaf-to-root path. Wlog, assume $R_1 - R_2 - R_3$ is a leaf-to-root path. First, for each tuple $t \in R_1$, $\lambda(t) = 1$ as R_1 is a leaf node. For $t = (i, v_{ij}) \in R_2$, we observe that $\tilde{I}(t) = [i, i]$ as $I(t) = [i, i + 2k]$ and $I(t') = [i - 2k, i]$ for some tuple $t' \in R_1$. But in this case, $\lambda(t) = 1$ still holds, as there exists no tuple $t' \in R_1$ with $\tilde{I}(t') \subseteq [i, i]$. However, for each tuple $t \in R_3$, $\lambda(t) = n$ as there exists a tuple $t' \in R_2$ such that $\tilde{I}(t') = [i, i]$ for every $i \in [n]$. Hence, the enclosureness of S on every generalized join tree is $\lambda = \frac{n^2 \cdot n + n^2 \cdot 1}{n^2} = n$.

The correctness of this simulation is obvious. This way, if there is a data structure that can be updated in $O(\lambda^{1-\epsilon})$ time while supporting $O(\lambda^{2-\epsilon})$-delay enumeration for Q over any FIFO sequence, then the OuMv problem can be solved in $O(n^2 \cdot \lambda^{1-\epsilon} + n \cdot \lambda^{2-\epsilon}) = O(n^{3-\epsilon})$ time. Note that the construction above requires a database of size at least $n^2 = \lambda^2$, thus $\lambda \leqslant \sqrt{|D|}$. $\square$

Proof of Theorem 19. We next turn to the update cost of our indexes. As mentioned in the beginning of Section 6, the total update cost of the entire sequence is asymptotically dominated by that of P-Update, which is further bounded by the number of times all the counters count$[t]$ can change. The following lemma connects this quantity with the enclosureness of the update sequence.

Lemma 27. *For any tuple t, count$[t]$ changes $O(\lambda(t))$ times.*

Proof. The status change of tuple $t \in R_e$ falls into one of the following two cases: (1) tuple t is being inserted or deleted; (2) some tuple $t' \in R_{e'}$ for $e' \in \mathcal{T}_e$ is inserted or deleted, and this update propagates to t. Note that tuple t can be inserted and deleted once in its lifespan, thus bounded by $O(1)$ and the cost is reflected in R-Update. Then, we will focus on the second case.

We start with the case that e has one child node in $\mathcal{T}$. In this case, count$[t]$ has its value changed between 0 and 1. Note that if an insertion changes t from $R_e/V_s(R_e)$ to $V_s(R_e)$, subsequent insertions won't change the status of t unless a deletion occurs. Consider a set of k disjoint intervals $\tilde{I}_1, \tilde{I}_2, \cdots, \tilde{I}_k$ in ordering, such that $\tilde{I}_j \in \tilde{\mathcal{I}}_e$, $\tilde{I}_j \subseteq I(t)$ for each $j \in [k]$, and there exists no additional interval $\tilde{I}$ such that $\tilde{I} \subsetneq \tilde{I}_j$ or $\tilde{I}' \subseteq [\tilde{I}_j^+, \tilde{I}_{j+1}^-]$ for any $j \in \{1, 2, \cdots, k\}$. Each of the k intervals can change the status of t at most twice, so they together can change the status of t at most $O(k)$ times. The effective lifespan of t exactly captures such a quantity.

We next consider a case when e has two child nodes $e_1, e_2 \in \mathcal{T}$. Similarly, consider a set of k disjoint intervals $\tilde{I}_1, \tilde{I}_2, \cdots, \tilde{I}_k$ in ordering, such that $\tilde{I}_j \in \tilde{\mathcal{I}}_e$, $\tilde{I}_j \subseteq I(t)$ for each $j \in [k]$, and there exists no additional interval $\tilde{I}$ such that $\tilde{I} \subsetneq \tilde{I}_j$ or $\tilde{I} \subseteq [\tilde{I}_j^+, \tilde{I}_{j+1}^-]$ for any $j \in \{1, 2, \cdots, k\}$. We can make the following two observations:

(1) For any $\tilde{I}_j$, count$[t]$ can change at most 2 times within $\tilde{I}$.
(2) For any two adjacent intervals $\tilde{I}_j$ and $\tilde{I}_{j+1}$, count$[t]$ can change at most 4 times in their gap.

Together, we can conclude that count$[t]$ can change at most $6 \cdot k = O(k)$ times when there are two child nodes. We next go into details of (1) and (2) separately.

For (1), we assume $\tilde{I}_j \in \widetilde{\mathcal{I}}_{e_1}$ without loss of generality. By the definition of effective lifespan, there cannot be any insertion or deletion in any node of $\mathcal{T}_{e_1}$ within $\tilde{I}_j$. Nevertheless, updates may still exist within $\tilde{I}_j$ on some node of $\mathcal{T}_{e_2}$, which might further change count$[t]$. We distinguish two more cases. If count$[t]$ changes from 1 to 2, due to an insertion from $\mathcal{T}_{e_2}$, a deletion must not exist within $\tilde{I}_j$ on any node of $\mathcal{T}_{e_2}$, implied by the fact that there exists no $\tilde{I}$ such that $\tilde{I} \subsetneq \tilde{I}_j$. Hence, count$[t]$ can change at most once in $\tilde{I}_j$ for this case. Otherwise, count$[t]$ changes from 2 to 1, after a deletion from $\mathcal{T}_{e_2}$. We then go into the first case and count$[t]$ can change at most one more time. In total, count$[t]$ can change at most twice.

For (2), it is clear that at the right endpoint of $\tilde{I}_j$ and the left endpoint of $\tilde{I}_{j+1}$, count$[t]$ can change once as the deletion and insertion of an effective lifespan. In the meantime, there does not exist another effective lifespan within their gap, so for any $e_i \in \{e_1, e_2\}$, there exists no deletion on $\mathcal{T}_{e_i}$ in the gap following an insertion in $\mathcal{T}_{e_i}$. This way, count$[t]$ can change at most four times (i.e. $2 \to 1 \to 0 \to 1 \to 2$) within their gap.

At last, we consider the general case when e has multiple child nodes in $\mathcal{T}$. In this case, count$[t]$ has its value changed among $0, 1, \cdots, j$, where j is the number of child nodes of e. By extending the previous two observations, we conclude that count$[t]$ can change at most $3j \cdot k$ times, where j can be considered as a constant. With respect to all possible choices of k, we observe that

$$k \leqslant \max_{\substack{\mathcal{J} \subseteq \widetilde{\mathcal{I}}_e \\ \forall I(t_1) \in \mathcal{J}, I(t_1) \subseteq I(t) \\ \forall I(t_2), I(t_3) \in \mathcal{J}, I(t_2) \cap I(t_3) = \emptyset}} 1 + |\mathcal{J}| = \lambda(t),$$

thus count$[t]$ can change at most $O(\lambda(t))$ times. □

The time cost of Algorithm 2 is determined by the number of iterations of for-loop (line 2 or 8). One can easily observe that count$[t]$ will be changed for some tuple t once in each iteration, therefore the running time can be bounded by the number of changes to count$[t]$ over all tuples t. Now consider an update sequence S with enclosureness λ. Implied by Lemma 27 the total update cost is $O\left(\sum_{t \in \mathcal{I}} \lambda(t)\right)$, which is $O\left(\frac{\sum_{t \in \mathcal{I}} \lambda(t)}{|\mathcal{I}|}\right) = O(\lambda)$ amortized. Putting everything together, we have completed the proof for Theorem 19. □

D EXTENSIONS IN SECTION 7

D.1 Selection, union, and set difference

The query plan in Section 4.1 works for CQs with joins and projections, but it can be equipped with other operators easily.

- If there is a selection σ_ϕ on an input relation R_e where ϕ is a predicate on e, then for an update with tuple $t \in R_e$, we simply check if $\phi(t)$ is true, and discard this update if not. This only adds $O(1)$ time to the update cost.
- For the union of CQs $Q = Q_1 \cup \cdots \cup Q_k$, we just maintain each Q_i separately. Full result enumeration can be supported with $O(1)$ delay using the technique in [11]. We note that [11] assumes that the data structure on each $Q_i(D)$ can check if $t \in Q(D_i)$ in $O(1)$ time for any given t, which is indeed supported by our query plan. For delta enumeration, we can use the same technique to enumerate $\Delta Q_1(D) \cup \cdots \cup \Delta Q_k(D)$. However, this is not the same as $\Delta(Q_1 \cup \cdots \cup Q_k)$, and we need to check, say, if some new result $t \in \Delta(Q_1)$ already exists in $Q_2(D)$. Thus, while the technique of [11] is still correct for delta enumeration, the delay is not bounded by a constant. How to support $O(1)$-delay delta enumeration for UCQs remains an interesting open problem.
- For a query like $Q = Q_1 - Q_2$, we can as above maintain Q_1 and Q_2 separately. For enumeration, we enumerate every $t \in Q_1(D)$ and check if $t \in Q_2(D)$, although this does not guarantee constant delay. In fact, even in the static case, it is an open question whether $Q_1(D) - Q_2(D)$ can be enumerated in $O(1)$ delay after linear-time preprocessing where Q_1 and Q_2 are both free-connex.

D.2 Aggregations

Standard relational algebra can be extended to support aggregations, and we adopt the following formalism [3, 22]. Let $(S, \oplus, \otimes)$ be a commutative ring[7]. Every tuple $t \in R_e$ has an annotation $v(t) \in S$. For a full CQ Q in the form of (1), the annotation for any join result $t \in Q(D)$ is defined as $v(t) := \bigotimes_{e \in Q} v(t[e])$. For a non-full query $\pi_{\mathbf{y}} Q$, the projection becomes `GROUP BY` $\mathbf{y}$, and the annotation for each result $t \in \pi_{\mathbf{y}} Q(D)$ (i.e., the aggregate of each group) is $v(t) := \bigoplus_{t' \in Q(D): \pi_{\mathbf{y}} t' = t} v(t')$.

Our new change propagation framework can support aggregations easily. For any relation e, let $v(t)$ be the annotation for $t \in R_e$, $v_s(t)$ be the annotation for $t \in V_s(R_e)$ and $v_p(t)$ be the annotation for $t \in V_p(R_e)$. Following the definitions of semi-join view V_s and projection view V_p (see Section 4), the annotation of $t \in V_s(R_e)$ can be written as

$$v_s(t) := \left(\bigotimes_{e_i \in \{e_1, \cdots, e_k\}: e_i \cap \mathbf{y} = \emptyset} v_p(\pi_{e_i} t) \right) \otimes \begin{cases} v(t) & e \cap \mathbf{y} = \emptyset \\ 1 & e \cap \mathbf{y} \neq \emptyset \end{cases} \tag{11}$$

where $\{e_1, e_2, \cdots, e_k\}$ are the children nodes of e. The annotation of $t \in V_p(R_e)$ can be written as

$$v_p(t) := \bigoplus_{t' \in V_s(R_e): \pi_{\mathrm{key}(e)} t' = t} v_s(t'). \tag{12}$$

Now, we can rewrite $v(t)$ for each query result $t \in Q$ as:

$$v(t) := \bigotimes_{e \in Q} \left(v(\pi_e t) \otimes v_s(\pi_e t) \right). \tag{13}$$

We store these annotations alongside their counters in the query plan. Then, the change propagation and enumeration procedures should be modified according to the formulas above. More precisely, whenever the counter of a tuple is updated, we also update its annotation by (11) or (12). One technical difference is that, during an S-Update, we need to keep propagating the change upwards whenever the counter changes, not just when the counter changes from 0 to 1 or vice versa as in Algorithm 1. Finally, after enumerating a result, we compute its annotation by (13).

[7] In the static case, $(S, \oplus, \otimes)$ is only required to be a semi-ring, but we need additive inverses to support deletions.