RESULTS reproduced

Artifacts Available acm V1.1

Change Propagation Without Joins

Qichen Wang
Hong Kong Baptist University
qcwang@hkbu.edu.hk

Xiao Hu
University of Waterloo
xiaohu@uwaterloo.ca

Binyang Dai
Hong Kong University of Science and Technology
bdaiab@ust.hk

Ke Yi
Hong Kong University of Science and Technology
yike@ust.hk

ABSTRACT

We revisit the classical change propagation framework for query evaluation under updates. The standard framework takes a query plan and materializes the intermediate views, which incurs high polynomial costs in both space and time, with the join operator being the culprit. In this paper, we propose a new change propagation framework without 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 be integrated 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.

PVLDB Reference Format:
Qichen Wang, Xiao Hu, Binyang Dai, and Ke Yi. Change Propagation Without Joins. PVLDB, 16(5): 1046 - 1058, 2023.
doi:10.14778/3579075.3579080

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/hkustDB/CROWN.

1 INTRODUCTION

We study the problem of *query evaluation 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 enumeration* and *delta enumeration*. The former is pull-based, i.e., the system returns $Q(D)$ passively upon 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 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 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 to the base tables, while the output is a stream of updates to the query result (i.e., a stream of deltas). If the user wishes to always have a complete and accurate $Q(D)$, it has to maintain $Q(D)$ and update it with the deltas as they are received. If approximation is acceptable, some more efficient streaming algorithms can be used instead.

Change propagation. Change propagation [9, 23, 30] 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 base relations and each internal node is a relational operator. At each internal node, it maintains the results of the subquery 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 4-Hop query from benchmark [26]

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

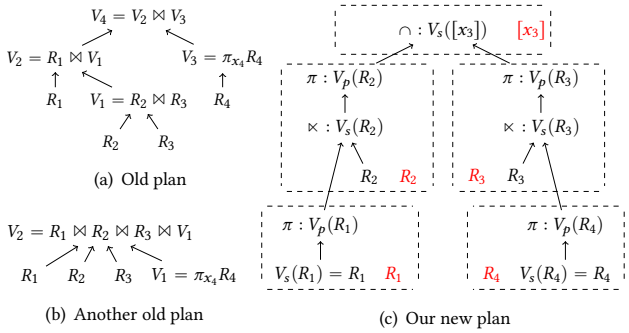
Under the standard change propagation framework, we maintain four materialized views $V_1, V_2, V_3, V_4 = Q$ (if only delta enumeration is needed, then V_4 need not be maintained). When a tuple t is inserted or deleted in a relation, say R_1, it follows the leaf-to-root path to propagate the deltas 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 [3]. To avoid 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 exacerbates for queries involving more relations [3].

Prior work has designed advanced techniques to address this space or time blowup. The Dynamic Yannakakis algorithm [17–19] has linear space and linear update time while supporting constant-delay enumeration for *free-connex queries*[1]; the update time further reduces to $O(1)$ amortized[2] for *q-hierarchical queries*. Concurrently, Berkholz et al. [6] 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

Proceedings of the VLDB Endowment, Vol. 16, No. 5 ISSN 2150-8097.
doi:10.14778/3579075.3579080

[1] All technical terms in the introduction are formally defined in Section 3.
[2] All update time bounds are amortized in this paper.



(a) Old plan

(b) Another old plan

(c) Our new plan

Figure 1: For $Q = \pi_{x_1,x_2,x_3,x_4} R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie R_3(x_3, x_4) \bowtie R_4(x_4, x_5)$, 1(a) and 1(b) are two plans under the standard change propagation framework and 1(c) is our new plan.

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 improve for certain classes of queries and/or 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 (or an intersection) plus a projection. However, not every query plan is amenable to this replacement strategy. The key technical 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(c), 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 an update time better than linear. (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 the static case [4], we show how to enumerate $Q(D)$ with constant delay, by appropriately traversing this 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 the issue (2), Wang and Yi [33] introduced the notion of *enclosureness* λ of an update sequence, which captures the hardness of the update sequence. It is linear 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*. Such queries are relatively easy to handle since their result size 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 6.2) that there is a simple 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 not achievable 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 under the new definition. To further 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 [6, 17] 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 [34] has runtime $O(|D| + |Q(D)|)$ for every free-connex query. It consists of two stages. The first stage uses a series of semi-joins 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 [17] 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 can also be viewed as a dynamic version of the Yannakakis algorithm, but it strictly follows the standard change propagation framework while achieving a better runtime. The Dynamic Yannakakis algorithm has an update cost of $O(|D|)$ for free-connex queries, while our algorithm achieves $O(\lambda)$ update time, where λ is the *enclosureness* of the update sequence. We have $\lambda \leqslant |D|$ for all update sequences, while the former is usually much smaller on real-world update sequences. Furthermore, Dynamic Yannakakis achieves $O(1)$ update time only for q-hierarchical queries, while our algorithm also achieves $O(1)$ update time for non-q-hierarchical queries if the update sequences enjoy some special properties, such as first-in-first-out or insertion-only (formally defined in Section 6.1). The gap between Dynamic Yannakakis and

our algorithm can be as large as $\Theta(|D|)$ on some non-q-hierarchical queries (see Example 6.11).

Bagan et al. [4] 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. [22] show that it is possible to increase the enumeration delay in exchange for faster update time, on hierarchical (but non-q-hierarchical) queries. 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) [2] 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 [28, 29], 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 [7] and Trill [8] use standard change propagation for multi-way stream joins, which we will compare against in Section 8. Some specialized systems are designed for two-way stream joins [11, 13, 20, 25, 31], 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:

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

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 join query*; otherwise, it is said to be *join-project query*. 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 join CQ, we have $Q(D \ltimes t) = Q(D + t) \ltimes t$; for join-project 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.

Semi-joins. The semi-join $R_i(x_i) \ltimes R_j(x_j)$ is defined as

$$R_i(x_i) \ltimes R_j(x_j) = \{t | t \in \pi_{x_i} R_i \bowtie R_j\}.$$

Updates and Deltas. 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. We denote $D + t$ as the database after inserting t and $D - t$ as the database after deleting t. 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.

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 deletion. For a full join query, $\Delta Q(D, t) = Q(D \ltimes t)$. For join-project queries, $\Delta Q(D, t) \subseteq Q(D \ltimes t)$. In particular, it is possible to have $\Delta Q(D, t) = \varnothing$ even if $Q(D \ltimes t) \neq \varnothing$.

We target *constant delay* [4] for both full 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

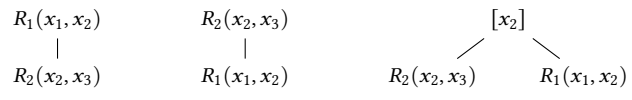
Acyclic queries. The acyclicity of a CQ Q is defined by the acyclicity of the hypergraph $(\mathcal{V}, \{e_1, \ldots, e_n\})$. More precisely, Q is acyclic if there exists a join tree $\mathcal{T}$, whose nodes are the relations in Q such that, for each attribute $x \in \mathcal{V}$, all nodes of $\mathcal{T}$ containing x form a connected component of $\mathcal{T}$. For example, Figures 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 based on *generalized relations* [10, 17]. Different from previous definition of *generalized relation*, it now can be a proper subset of any e_i. We can show that the following is an equivalent definition of acyclic queries[3]:

Definition 3.1 (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 corresponds 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 $\mathcal{T}$);

[3] Proof of equivalence is given in the full version [32] of the paper.



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

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

(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 is given in Figure 2(c). In a generalized join tree $\mathcal{T}$, we use r to denote the root, and $\mathcal{T}_e$ for the subtree rooted at node e, C_e for the set of children of node e and $p(e)$ for the parent of node e. Let $\text{key}(e) = e \cap p(e)$ be the *join key* between node e and $p(e)$. The height of $\mathcal{T}$ is defined as the maximum number of relations on any leaf-to-root path, without counting generalized relations.

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 be acyclic, and an acyclic full join query must be free-connex. For our development, we need the following equivalent definition of free-connex queries:

Definition 3.2 (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 $x_2 \in \mathcal{V} - \mathbf{y}$, $\text{top}(x_2)$ is not an ancestor of $\text{top}(x_1)$ in $\mathcal{T}$, where $\text{top}(x)$ is the highest node in $\mathcal{T}$ that contains x. Such a $\mathcal{T}$ is called a free-connex join tree.

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. A more restricted subclass of free-connex queries is *q-hierarchical query*. Let $\mathcal{E}_x$ denote the set of relations containing attribute x.

Definition 3.3 (Q-hierarchical queries). 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 $x_2 \in \mathbf{y}$.

These classifications capture the hardness of evaluation or enumeration for a CQ. Firstly, a full join query can be evaluated in linear time in terms of input and output size if and only if it is acyclic; for join-project CQs, this complete class extends to free-connex query. Furthermore, free-connex and q-hierarchical CQs have played important roles in query enumeration. [4] showed that in static settings, constant-delay enumeration after a linear-time preprocessing step is possible for a CQ if and only if it is free-connex. Berkholz et al. [6] showed that in dynamic settings, constant-delay enumeration is possible for a CQ from a data structure that can be updated in constant time if and only if it is q-hierarchical.

Algorithm 1: PLANGENERATION(Q, T)

Input : A generalized join tree T for query Q;
Output : A new query plan T for Q;
1 **foreach** *node e in a postorder traversal of T* **do**
2 Replace node e with $V_s(R_e)$ in T;
3 **if** *e is not the root of T* **then**
4 Add $V_p(R_e)$ between $V_s(R_e)$ and the parent of e;
5 **return** T;

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 highlighted in red (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*

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

Noted that the root node does not have a projection view.

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 $\text{key}(e_i) = e_i \cap e = e$ for every i (by the last property in Definition 3.1), (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 formulae above. Algorithm 1 takes as input a generalized join tree, and outputs a new query plan under our framework. Figure 1(c) shows the new query plan for the example query. Note that R_2 and R_4 fall into case (ii), while the root node $[x_3]$ is under case (iii). As neither projection nor semi-join (including intersection as a special case) enlarges the input relations, the following is straightforward:

LEMMA 4.1. *All views in our query plan have size $O(|D|)$.*

Example 4.2. Figure 3(a) shows the initial index built for the query in Figure 1. For R_1 and R_4, both semi-join and projection views are defined as themselves. $V_s(R_2)$ contains tuples in R_2 that can join with $V_p(R_1)$, which include $(2, 2)$ and $(2, 4)$. $V_s(R_3)$ is defined similarly including 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 $R([x_3])$, since every other tuple in $R([x_3])$ fails to join with $V_p(R_2)$ and $V_p(R_3)$: their counters need to be 2.

Algorithm 2: S-UPDATE(e, t)

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

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

Algorithm 3: 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**
    $\text{count}[t'] \leftarrow \text{count}[t'] + 1$;
    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**
    $\text{count}[t'] \leftarrow \text{count}[t'] - 1$;
    if *count*$[t'] = |C_e| - 1$ **then**
      $V_s(R_e) \leftarrow V_s(R_e) - \{t'\}$, S-UPDATE$(e, t')$;

4.2 Change Propagation

Change propagation using our new query plan can be done using standard (actually, even simpler for certain operators) propagation formulae [9]. For completeness, we briefly describe them below, which are also needed to understand the algorithms in Section 5.

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* [9], a standard technique to propagate changes through a projection. Specifically, we associate a counter $\text{count}[t']$ for each tuple $t' \in V_p(R_e)$ that stores the number of tuples $t \in V_s(R_e)$ such that $t[\text{key}(e)] = t'$. The detailed process, which needs to distinguish between an insertion and a deletion, is given in Algorithm 2. 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 [15] 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 $\text{key}(e_i) = e_i \cap e \subseteq e$ 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 $\text{count}[t']$ for every tuple t' in R_e, storing the number of child nodes $e_i \in C_e$ such that $t'[\text{key}(e_i)] \in V_p(R_{e_i})$. A tuple t' appears in $V_s(R_e)$ if and only if $\text{count}[t'] = |C_e|$. The algorithm is then immediate, as shown in Algorithm 3. We also need a hash index (that needs to support $e \cap e_i$ as the key for each $e_i \in C_e$) 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 3 applies verbatim.

Algorithm 4: R-UPDATE(e, t)

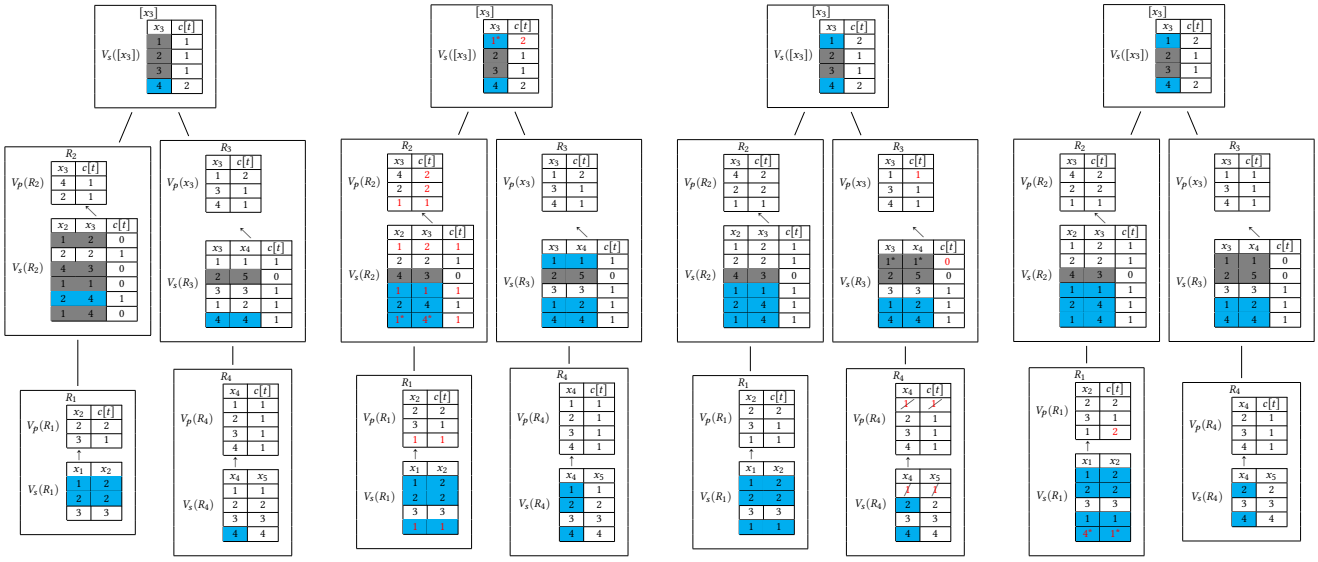
Input : An update t from an input relation R_e;
Output: Updated $V_s(R_e)$;

if *t is an insertion into R_e* **then**
  $\text{count}[t] \leftarrow 0$;
  foreach $e_i \in C_e$ **do**
    if $t[key(e_i)] \in V_p(e_i)$ **then**
      $\text{count}[t] \leftarrow \text{count}[t] + 1$;
  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$(e, t)$;

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 4, 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 $V_p(R_{e_i})$).

Example 4.3. Figure 3(b) shows the index after inserting $(1, 1)$ into R_1. This insertion first triggers an insertion to $V_p(R_1)$, which further increments counters of 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 stops here. Tuple $(1, 1) \in V_s(R_2)$ inserts a new tuple (1) to $V_p(R_2)$, which further increments the counter of (1) in the root, bringing it to $V_s([x_3])$. Tuple $(1, 4) \in R_2$ increments the counter of $(4) \in V_p(R_2)$ and the propagation stops. 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 $V_s(R_3)$. Finally, the counter of $(1) \in V_p(R_3)$ decreases from 2 to 1, and the propagation stops here.

LEMMA 4.4. *All projection and semi-join views in our query plan can be updated in $O(|D|)$ time.*



(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 query in Figure 1 using the plan in Figure 1(c). Tuples in white are in $V_s(R)$, in grey are in $R \backslash V_s(R)$, in cyan are 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. c[t] is short for count[t].

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 5.1. *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 5.2. $V_s(R_r) = \pi_r Q(D)$.

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 enumeration, where $\biguplus$ denotes disjoint union:

LEMMA 5.3. $Q(D) = \biguplus_{t \in V_s(R_r)} Q(D \ltimes t)$.

Lemma 5.1, Corollary 5.2, and Lemma 5.3 allow us to use essentially the same algorithm from Bagan et al. [4] to achieve constant-delay enumeration of $Q(D)$; see Algorithm 5, 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 $t \in V_s(R_r)$.

LEMMA 5.4. *Algorithm 5 enumerates $Q(D)$ with $O(1)$ 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 the main reason why our query plan is not only space-efficient but also time-efficient. Recall that the full enumeration algorithm relies on Lemma 5.3. Then the key question is, can we have an analogy of Lemma 5.3 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 5.3.

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 2 and Algorithm 4), but P-update may propagate multiple changes upward (see line 6, 12 in Algorithm 2. 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 cases of a propagation path with respect to its ending tuple: (1) t; (2) t_j^p for some $j \in \{0, 1, \cdots, k\}$; (3) t_i^s for some $i \in \{0, 1, \cdots, k\}$.

Case (1) happens when the first update is an R-UPDATE and does not propagate any further change. This means that in Algorithm 4, there exists some child node e' of e such that $t[\text{key}(e')] \notin V_p(R_{e'})$,

Algorithm 5: 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 − y ≠ ∅ then
    if e ∩ y − p(e) = ∅ then Yield ⟨⟩;
    else Yield π_{y∩e}(R_e ⋉ t);
else
    Let C_e = {e_1, e_2, ··· , e_k};
    foreach t_1 ∈ FullEnum(T, e_1, t[key(e_1)]) do
        foreach t_2 ∈ FullEnum(T, e_2, t[key(e_2)]) do
            ···
            foreach t_k ∈ FullEnum(T, e_k, t[key(e_k)]) do
                Yield t ⋈ t_1 ⋈ t_2 ⋈ ··· ⋈ t_k;
```

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 3, 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 2, 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 $O(1)$-delay delta enumeration, we maintain a *live view* for each node e with $e \cap \mathbf{y} \neq \emptyset$: $V_l(R_e) := \pi_e Q(D)$, 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 5.2. 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 5.5. *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 $t \ltimes V_l(R_{p(e)}) \neq \emptyset$.*

Based on Lemma 5.5, 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.

Algorithm 6: 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, ··· , 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' ∈ π_y ΔV_s(R_{e_i}, t);
    S ← t' ⋈ V_l(R_{e_{i+1}}) ⋈ ··· ⋈ V_l(R_{e_k});
    foreach q ∈ S do
        S_i ← FullEnum(T_{e_i}, e_i, q[e_i]);
        S_j ← FullEnum(T_{e_j} − T_{e_{j−1}}, e_j, q[e_j]), j ∈ [i + 1, k];
        Yield S_i ⋈ S_{i+1} ⋈ ··· ⋈ S_k;
```

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*:

Definition 5.6 (Witness tuple). Suppose t is inserted into or deleted from D. A tuple t' is a witness of t if

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

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

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 5.6. First, (5) is the counterpart of Corollary 5.2 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 causes 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 $\Delta\left(\pi_{e \cap \mathbf{y}}(\bowtie_{e' \in \mathcal{T}_e} R_{e'})\right) = \Delta\left(\pi_{\mathbf{y}} V_s(R_e)\right)$. 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 5.7. $\Delta Q(D, t) = \biguplus_{t' : \text{a witness of } t} Q(D \ltimes t')$.

We are now ready to state the counterpart of Lemma 5.3 for delta enumeration, in Lemma 5.7. Unlike Lemma 5.3, the proof of Lemma 5.7 is nontrivial, and the details are given in the full version [32].

The algorithm. To perform delta enumeration using Lemma 5.7, 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 5.6: (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 6. 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 $S_i \bowtie \cdots \bowtie S_k$ 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.

Example 5.8. In figure 3(a), there are two query results $(1, 2, 4, 4)$ and $(2, 2, 4, 4)$. In figure 3(b), we have the following observations when propagation stops. $(1, 2) \in R_2$ is not a witness as it cannot join with any tuple in $V_l([x_3])$, thus no delta is produced. $(1) \in [x_3]$ is a witness, which triggers delta enumeration. For a witness in the root, DeltaEnum simply degenerates to FullEnum$(\mathcal{T}_r, r, (1))$, which outputs $\{(1, 1, 1, 1), (1, 1, 1, 2)\}$. $(1, 4) \in R_2$ is a witness, which triggers delta enumeration. DeltaEnum finds $S = (1, 4) \bowtie V_l([x_3]) = \{(1, 4)\}$. For $(1, 4) \in S$, it invokes FullEnum$(\mathcal{T}_r - \mathcal{T}_{R_2}, r, (4))$ with $\{(4, 4)\}$ returned and FullEnum$(\mathcal{T}_{R_2}, R_2, (1, 4))$ with $\{(1, 1, 4)\}$ returned. Joining them yields the delta $\{(1, 1, 4, 4)\}$. Finally, as each new result is enumerated, we update the live views.

In figure 3(c), $(1, 1) \in \Delta V_s(R_3)$ is a witness. DeltaEnum first finds $S = \{(1, 1)\}$. For $(1, 1) \in S$, it invokes FullEnum$(\mathcal{T}_r - \mathcal{T}_{R_3}, r, (1))$ with $\{(1, 1, 1)\}$ returned, and FullEnum$(\mathcal{T}_{R_3}, R_3, (1, 1))$ with $\{(1, 1, 4)\}$ returned (delta enumeration upon a deletion is done before the tuple deletion so as to find the delta). Joining them yields the delta $\{(1, 1, 1, 1)\}$. Finally, we update live views with the delta.

Lemma 5.9. *Algorithm 6 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.

6 UPDATE COST ANALYSIS

We have shown that the delay of both full and delta enumeration is a constant, and this holds for the query plan defined by any free-connex join tree in Section 4.1. In contrast, 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 update cost of most real-world update sequences can be much better. Characterizing the update cost will be important for finding a good free-connex join tree. 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 generalized join trees can have a smaller height than standard join trees. For example, the query in Figure 2 has a generalized join tree of height 1 while the two standard join trees have height 2.

6.1 Enclosureness

Update sequences and lifespans. 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 models 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.

The notion of *enclosureness* was first introduced in [33] to give an instance-specific characterization of the hardness of the update sequence, which we briefly review next.

Definition 6.1 (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., $\lambda(S_D) := \max\left(\frac{\sum_{t \in S_D} \lambda(t)}{|S_D|}, 1\right)$. 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 6.2. *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 index 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[4] 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 [33].

6.2 Join-tree-specific Enclosureness

Hope is not all lost despite the negative result above. First, Theorem 6.2 only holds for a particular free-connex query; other queries may still be updated in $O(1)$ time. Secondly, the definition of enclosureness in [33] only considers the time dimension while ignoring 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 6.2 can still be solved efficiently, when information from both the structural dimension and the time dimension is taken into account.

Definition 6.3 (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 6.4. 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

$$\lambda_{\mathcal{T}}(t) = \max_{\substack{\forall t' \in \mathcal{J}, \exists e' \in \mathcal{T}_e - \{e\}, t' \in R_{e'} \\ \forall t_1 \in \mathcal{J}, \dot{I}(t_1) \subseteq I(t) \\ \forall t_2, t_3 \in \mathcal{J}, \dot{I}(t_2) \cap \dot{I}(t_3) = \varnothing}} |\mathcal{J}|, \tag{8}$$

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 $\mathcal{T}$.

The main analytical result of this paper is the following theorem, whose proof is quite technical (given in the full version [32]):

[4]The OuMv conjecture [16] 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.

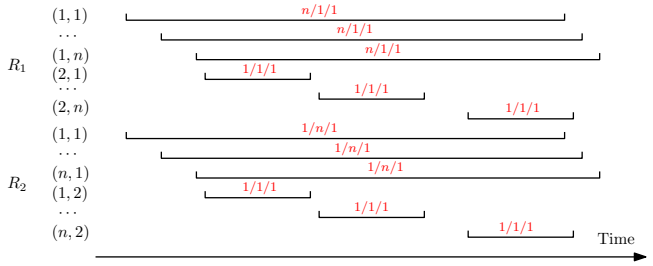


Figure 4: An example of update sequence for $Q := R_1(x_1, x_2) \bowtie R_2(x_2, x_3)$ in Figure 2 with $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$. Each interval is the lifespan of a tuple, and three numbers above each interval is its enclosureness over $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$. $\lambda_{\mathcal{T}_1} = \lambda_{\mathcal{T}_2} = n$ and $\lambda_{\mathcal{T}_3} = 1$.

Theorem 6.5. *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 6.6. *[33] 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[5] fails.*

6.3 Implications of Enclosureness

We present some implications of our join-tree-specific enclosureness and Theorem 6.5, 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 [6, 17] 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 6.5, 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:

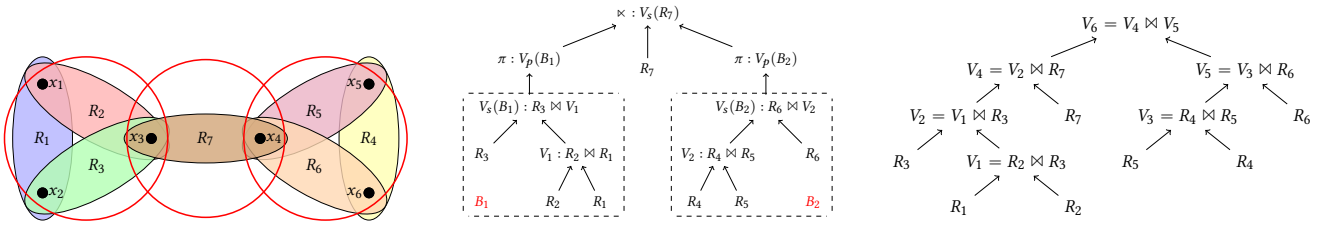
Lemma 6.7. *Every q-hierarchical query has a free-connex generalized join tree of height* 1.

For arbitrary update sequences, q-hierarchical queries are precisely the class of queries for which $O(1)$ update time is possible [6]. 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 [33], but nothing is known for non-key joins (except for q-hierarchical queries which do not rely on FIFO). We present the first extension in this direction:

Lemma 6.8. *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 sequence.*

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



(a) Hypergraph and a GHD (b) Query plan under new change propagation (c) Query plan under standard change propagation

Figure 5: 5(a) is the hypergraph of the "dumbbell" query $Q = R_1(x_1, x_2) \bowtie R_2(x_1, x_3) \bowtie R_3(x_2, x_3) \bowtie R_4(x_5, x_6) \bowtie R_5(x_4, x_5) \bowtie R_6(x_4, x_6) \bowtie R_7(x_3, x_4)$, with GHD illustrated in red circle. 5(c),5(b) are query plans under the standard, new change propagation framework respectively. In 5(c), B_1, B_2 are treated as two basic relations, on which projection and semi-join views are constructed.

Note that the height limit of 2 is the best one can hope for, since the query in Theorem 6.2 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.

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

Theorem 6.10. *For a free-connex query Q, there is an index that can be updated in $O(1)$ amortized time under any insertion-only update sequence, while supporting $O(1)$-delay enumeration.*

Note that Lemma 6.9 incorporates static case [4] as a special case. Given a static database D, we can simply insert every tuple of D into our index. By Lemma 6.9, this index can be built in $O(|D|)$ time while supporting $O(1)$-delay enumeration of $Q(D)$. Also, the dichotomy result of [4] states that with $O(|D|)$ preprocessing time, $O(1)$-delay enumeration is only possible for free-connex queries, thus Lemma 6.9 cannot be extended to beyond free-connex queries, either, even over insertion-only sequences.

Example 6.11. Consider an insertion-only update sequence for the query in Figure 1: (1) $(i, j) \in [n] \times [n]$ are inserted into R_2, R_3 and R_4 initially; (2) $(i, j) \in [n] \times [n]$ are inserted into R_1 later. Standard change propagation or HIVM needs to materialize $\Delta(R_1 \bowtie R_2 \bowtie R_3)$, incurring $O(n^3)$ cost; the Dynamic Yannakakis algorithm [17] needs to scan all tuples $(j, *) \in R_2$ once (i, j) is inserted into R_1, hence incurs $\Theta(n)$ cost; and our framework only incurs $O(1)$ cost.

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 $\mathcal{T}$ with a smaller height is always preferred. Furthermore, Definition 6.4 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 can estimate it by observing (and buffering) the first few updates.

Table 1: Comparison of different query processing engines.

	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

7 EXTENSIONS TO GENERAL QUERIES

Acyclic but non-free-connex queries. Consider such a query $\pi_{x_1,x_3} R_1(x_1, x_2) \bowtie R_2(x_2, x_3)$. We simply add x_2 as an output attribute to turn it into a free-connex query, and then do a projection over x_1, x_3 during enumeration. Note that enumeration may contain duplicates. Thus, if a `DISTINCT` keyword is declared explicitly, duplicates need to be removed, hence making the delay more than constant, but this is inevitable due to the lower bound [4].

Cyclic queries. Cyclic queries can also be easily handled in our framework by resorting to *Generalized Hypertree Decomposition* (GHD) [14]. More specifically, by grouping several relations into a *bag*, an arbitrary CQ can be converted into a free-connex one. For example, Figure 5(a) shows a GHD for the "dumbbell" query with 3 bags. We can use standard change propagation within each bag, and apply our framework across the bags. This results in the query plan in Figure 5(b), which has $O(N^2)$ space and $O(N)$ update time while supporting constant-delay enumeration. On the other hand, the standard change propagation framework would use a query plan like the one in Figure 5(c), which has $O(N^3)$ space and update time. Of course, all these are worst-case bounds; on realistic inputs, the costs are lower, but our new query plan is still order-of-magnitude better than the old plan, as shown in Section 8.

If one is interested in further improving the theoretical bounds, the algorithm for maintaining the query results inside each bag can be replaced by a better algorithm. For example, Kara et al. [21] present an algorithm for maintaining the triangle join. However, not many results are known beyond the triangle join. This is still

an actively researched problem; any improvement here will also improve general CQs when plugged into our framework.

Selection, union, set difference, aggregation. In the full version [32], we show how to support these operators in our framework.

8 EXPERIMENTS

8.1 Setup

Implementation. We have implemented our algorithms and built a system prototype called CROWN (**C**hange p**RO**pagation **W**ithout joi**N**s) [32] 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 [1, 5, 33].

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 for fair comparison (since DBToaster and Trill 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 runtime. We set a 4-hour time limit for each run.

Query processing engines compared. We compare CROWN with (1) DBToaster [2], the best HIVM engine that supports multi-way joins over arbitrary update streams in centralized settings; (2) DBToaster Spark [27], which can support IVM with batch updates in a distributed/parallel setting; (3) Trill [8], 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 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 from the benchmark by Nguyen et al. [26], over the SNAP dataset (Stanford Network Analysis Project) [24]. Such a benchmark evaluates the performance of each system for join queries over static data, and we modify it to adapt to the dynamic scenario. We test all acyclic queries from the benchmark, such as hop (path) queries, star queries and comb queries. We also test the dumbbell query, which is a variant of the lollipop query. The detailed query definition is given in the full version and 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) [12], which accesses 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 6 shows the join hypergraphs of all queries. Except for SNB Q3 and Q4, they have a height-2 free-connex generalized join tree. The star query (figure 6(d)) has a height-1 free-connex generalized join tree, so it is q-hierarchical. The 4-Hop query (figure 6(c)) and SNB Q4 query (figure 6(f)) have the same hypergraph structure but different output attributes, and the 4-Hop query has a height-2 free-connex generalized join trees while SNB Q4 query does not.

We create FIFO streams 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 is time-based, so the number of tuples in a window fluctuates over time.

8.2 Experiment Results

Runtime. Figure 10 shows the total runtime of evaluating each graph query over a mid-sized graph *Epinions* and each SNB query in the centralized setting. The graph contains approximately 500K edges and 76K vertices, as well as 3.7B 3-Hop paths and 378B 4-Hop paths. On the other hand, we use the default scale factor of 1 for all SNB queries. Under the scale factor, the total size of raw data is $1.5GB$, and the largest relation contains 15 attributes. We set a filter condition that only keeps 10% of the designated endpoints for all 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 can finish all queries successfully. Trill only handles 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 67x compared with Flink, 1.8x to 234x compared with DBToaster, and 2.7x to 523x compared with

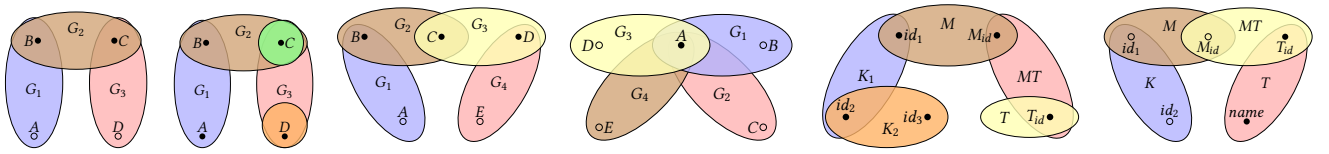


Figure 6: The relational hypergraphs of queries. The solid dots are output attributes for join-project and aggregation queries.

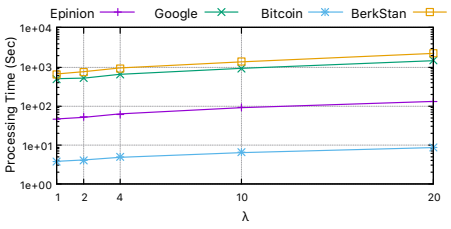


Figure 7: Runtime v.s. enclosureness λ.

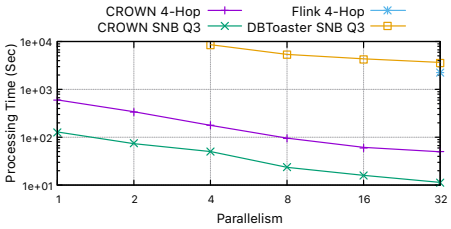


Figure 8: Runtime v.s. parallelism p.

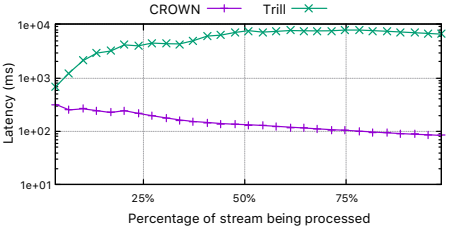


Figure 9: Average latency.

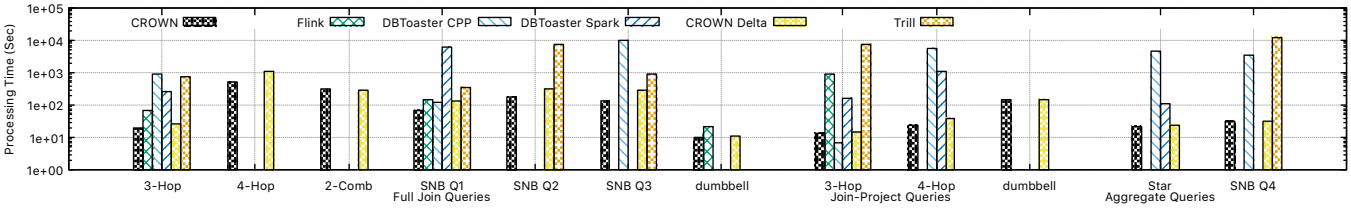


Figure 10: Processing times of CROWN, Flink, DBToaster, and Trill

Trill. Moreover, in handling join-project queries, CROWN requires much less time than handling the corresponding full join queries, while Flink requires more time. In addition, CROWN performs well for both full and delta enumeration, and different modes of output do not affect the overall performance of CROWN.

Enclosureness. To test the influences of enclosureness, we create multiple update sequences with different λ, over different graphs from the SNAP dataset. We disable the output to see how the update cost would change with different λ. The experiment results are shown in Figure 7. From the results, we can see the maintenance cost of CROWN increases almost linear as λ increases.

Distributed processing. To compare CROWN with DBToaster Spark and Flink in a distributed setting, we built a small cluster with 32 task slots, and tested 4-Hop as well as SNB Q3 query, on which DBToaster and Flink cannot finish in a centralized setting. Figure 8 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 with more workers implied by HyperCube; (2) processing time is already small, and system overhead dominates the entire runtime. For all finished data points, CROWN can provide a speedup from 45x to 324x.

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 of memory. If we further limit the memory usage of CROWN to 16GB, i.e., 500MB per worker, CROWN still works 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 its deltas are outputted. Figure 9 shows the result. The average latency of CROWN is less than 90ms, while that 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.

Scalability. To test the scalability of different platforms, we change the scale factor of the SNB benchmark and compare the average update cost between different platforms. Due to the space constraint, the results are given in [32]. The results show that the average processing time of CROWN is stable under different data sizes. In contrast, the data size will affect the average processing time of other platforms, suggesting CROWN has better scalability.

Selectivity. We also adjust the filter condition to test the performance with different output sizes. Due to the space constraint, A detailed analysis of the results of this part is given in the full version [32]. To summarize, the runtime of Flink and DBToaster depends on input, output and intermediate join size. Meanwhile, the runtime of CROWN only depends on the input and output size. It makes CROWN highly efficient, especially when the selectivity is small.

ACKNOWLEDGMENTS

This work has been supported by HKRGC under grants 16201318, 16201819, and 16205420. Qichen Wang conducted this research work when he studied at HKUST.

REFERENCES

[1] 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.

[2] 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.

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

[4] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *Computer Science Logic*. Springer Berlin Heidelberg, Berlin, Heidelberg, 208–222.

[5] Paul Beame, Paraschos Koutris, and Dan Suciu. 2017. Communication Steps for Parallel Query Processing. *J. ACM* 64, 6, Article 40 (oct 2017), 58 pages. https://doi.org/10.1145/3125644

[6] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. 2017. Answering Conjunctive Queries under Updates. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 303–318. https://doi.org/10.1145/3034786.3034789

[7] 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.

[8] 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.

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

[10] 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* (Portland, OR, USA) *(PODS'20)*. Association for Computing Machinery, New York, NY, USA, 331–346. https://doi.org/10.1145/3375395.3389130

[11] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. 2014. Scalable and Adaptive Online Joins. *Proc. VLDB Endow.* 7, 6 (feb 2014), 441–452. https://doi.org/10.14778/2732279.2732281

[12] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2723372.2742786

[13] 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.

[14] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2002. Hypertree decompositions and tractable queries. *J. Comput. System Sci.* 64, 3 (2002), 579–627.

[15] 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

[16] 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* (Portland, Oregon, USA) *(STOC '15)*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/2746539.2746609

[17] Muhammad Idris, Martin Ugarte, and Stijn Vansummeren. 2017. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1259–1274. https://doi.org/10.1145/3035918.3064027

[18] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2019. Efficient query processing for dynamically changing datasets. *ACM SIGMOD Record* 48, 1 (2019), 33–40.

[19] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2020. General dynamic Yannakakis: conjunctive queries with theta joins under updates. *The VLDB Journal* 29, 2 (2020), 619–653.

[20] 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.

[21] 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

[22] 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.

[23] 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

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

[25] 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.

[26] Dung Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q Ngo, Christopher Ré, and Atri Rudra. 2015. Join processing for graph patterns: An old dog with new tricks. In *Proceedings of the GRADES'15*. 1–8.

[27] 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.

[28] 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.

[29] 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.

[30] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. 1996. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (Montreal, Quebec, Canada) *(SIGMOD '96)*. Association for Computing Machinery, New York, NY, USA, 447–458. https://doi.org/10.1145/233269.233361

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

[32] Qichen Wang, Xiao Hu, Binyang Dai, and Ke Yi. 2023. Change Propagation Without Joins. *arXiv preprint arXiv:2301.04003* (2023). https://doi.org/10.48550/arXiv.2301.04003

[33] 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.

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