Maintaining Acyclic Foreign-Key Joins under Updates

Qichen Wang Ke Yi
Hong Kong University of Science and Technology
Hong Kong, China
(qwangbp,yike)@cse.ust.hk

ABSTRACT

A large number of analytical queries (e.g., all the 22 queries in the TPC-H benchmark) are based on acyclic foreign-key joins. In this paper, we study the problem of incrementally maintaining the query results of these joins under updates, i.e., insertion and deletion of tuples to any of the relations. Prior work has shown that this problem is inherently hard, requiring at least $\Omega(|db|^{\frac{1}{2}-\epsilon})$ time per update, where $|db|$ is the size of the database, and $\epsilon > 0$ can be any small constant. However, this negative result holds only on adversarially constructed update sequences; on the other hand, most real-world update sequences are "nice", nowhere near these worst-case scenarios.

We introduce a measure λ, which we call the *enclosureness* of the update sequence, to more precisely characterize its intrinsic difficulty. We present an algorithm to maintain the query results of any acyclic foreign-key join in $O(\lambda)$ time amortized, on any update sequence whose enclosureness is λ. This is complemented with a lower bound of $\Omega(\lambda^{1-\epsilon})$, showing that our algorithm is essentially optimal with respect to λ. Next, using this algorithm as the core component, we show how all the 22 queries in the TPC-H benchmark can be supported in $\tilde{O}(\lambda)$ time. Finally, based on the algorithms developed, we built a continuous query processing system on top of Flink, and experimental results show that our system outperforms previous ones significantly.

CCS CONCEPTS

• **Information systems** → **Stream management**; **Database views**; **Join algorithms**.

SIGMOD '20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3380586

KEYWORDS

Query evaluation under updates; incremental view maintenance; acyclic joins; sliding windows

ACM Reference Format:
Qichen Wang Ke Yi. 2020. Maintaining Acyclic Foreign-Key Joins under Updates. In *2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3380586

1 INTRODUCTION

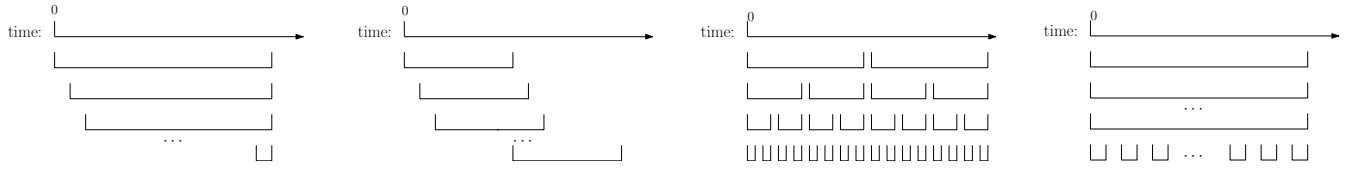
Query evaluation on a static database has been well studied. In this paper, we focus on the problem of continuous query evaluation on a database, which is a central problem in many real-time analytical systems.

Let db be the contents of the current database and let $Q(db)$ denote the results of evaluating query Q on db. For an update u to db, where u can be either the insertion or deletion of a tuple, we write $db + u$ as the new database instance after applying u. We say that the query Q can be *updated in $f(|db|)$ time*, if there is a data structure on db, denoted $\mathcal{D}(db)$, such that the following operations can be supported:

- *Update*: Given any update u, compute $\mathcal{D}(db + u)$ from $\mathcal{D}(db)$ in $f(|db|)$ time.
- *Delta enumeration*: After computing $\mathcal{D}(db + u)$ from $\mathcal{D}(db)$, one can, if needed, enumerate the delta ΔQ, i.e., all differences between $Q(db)$ and $Q(db + u)$, with constant delay.
- *Full results enumeration*: Whenever needed, all query results in $Q(db)$ can be enumerated from $\mathcal{D}(db)$ with constant delay.

In the last two operations above, one is required to enumerate a set of tuples with *constant delay* [5]. More precisely, this means that the time between the start of the enumeration process to the first tuple, the time between any two consecutive tuples, and the time between the last tuple and the termination of the enumeration process, should all be constant.

This problem is generally known as *continuous query evaluation under updates* or *incremental view maintenance*. As the convention in the literature, we treat the size of the query as



(a) Insertion-only: $\lambda = 1$ (b) First-in-first-out: $\lambda = 1$ (c) Hierarchical: $\lambda = O(\log N)$ (d) Worst-case: $\lambda = O(N)$

Figure 1: Example update sequences with different λ.

a constant, which is the number relations and attributes involved in the query, plus the number of relational operators such as selection, project, aggregation, join, etc.

Previous work. One standard approach to the problem of continuous query evaluation under updates is the *change propagation* framework [11, 13]. Given a query/view to be maintained, we take an operator tree (or a DAG), where each leaf of the DAG corresponds to a base table and an internal node corresponds to a relational operator. At each internal node v, we maintain a materialized view of the sub-query below v. When any of the base tables receives an update, we compute changes to each of these materialized views in a bottom-up fashion, eventually updating the view at the root of the operator tree, which corresponds to the results of the full query. One issue with this approach is that a single update to a base table may potentially incur a lot of changes in these views, which can be $O(|Q(db)|)$ in the worst case. *Higher-order IVM* (HIVM) [4, 17–19] has been proposed to remedy this problem, which 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, but there is still no theoretical guarantee on its update time.

In 2017, two papers [8, 14] simultaneously studied the worst-case complexity of the problem. Berkholz et al. [8] showed that constant update time can be achieved for the class of *q-hierarchical* queries, while for non-q-hierarchical queries, the update time must be at least $\Omega(|db|^{\frac{1}{2}-\epsilon})$ for any small constant $\epsilon > 0$, if constant-delay enumeration is needed. This result is mostly negative, because q-hierarchical queries are a small class of queries. Very simple queries such as $\psi_1 := \pi_x(R_1(x, y) \bowtie R_2(y))$ and $\psi_2 := R_1(x) \bowtie R_2(y) \bowtie R_3(x, y)$ are already non-q-hierarchical. Idris et al. [14] designed the Dynamic Yannakakis algorithm, a practical algorithm that supports all *α-acyclic queries* [7, 22], a superset of q-hierarchical queries, but the constant update time guarantee only holds for q-hierarchical queries. On non-q-hierarchical but α-acyclic queries, their algorithm may spend $O(|db|)$ time on an update. Recently, Kara et al. [15, 16] designed algorithms for certain non-q-hierarchical queries with update time $O(\sqrt{|db|})$.

Our contributions. The starting point of our work is the observation that the $\Omega(|db|^{\frac{1}{2}-\epsilon})$ lower bound holds only in the worst case, where the update sequence can be constructed adversarially. However, most real-world update sequences are nowhere near the worst-case scenario. Can we do better on these non-worst-case update sequences? Can we quantitatively separate "easy" update sequences with the hard ones?

The first contribution of this paper is the introduction of a measure, called *enclosureness*, which exactly captures the inherent difficulty of an update sequence. Given an update sequence, the lifespan of a tuple t can be considered as an time interval $i(t) = (i_s, i_e)$, where i_s is the start time (when t is inserted) and i_e is the end time (when t is deleted). The start time can be $-\infty$, if the tuple exists in the initial database, and the end time can be $+\infty$ if the tuple still exists in the final database. Note that if a tuple is deleted and inserted again later, it will be logically treated as two different tuples, which have the same attributes but different lifespans. Given an update sequence, all its lifespans form a set of intervals $\mathcal{I}$. For each interval $i \in \mathcal{I}$, define its *enclosureness* $\lambda(i)$ as

$$\lambda(i) = \max_{\substack{\mathcal{J} \subseteq \mathcal{I} \\ \forall j \in \mathcal{J}, j \subseteq i \\ \forall j_1, j_2 \in \mathcal{J}, j_1 \cap j_2 = \emptyset}} |\mathcal{J}|,$$

i.e., the largest number of *disjoint* intervals contained in i. The enclosureness of the whole update sequence is the average enclosureness of all its intervals, i.e.,

$$\lambda(\mathcal{I}) = \frac{\sum_{i \in \mathcal{I}} \lambda(i)}{|\mathcal{I}|}.$$

We will often just write λ instead of $\lambda(\mathcal{I})$ when the context is clear. Figure 1 shows some example update sequences with different λ; we see that, in many practically relevant scenarios, λ is small. However, λ can be $O(N)$ in the worst case, where N is the number of intervals.

We present both upper and lower bounds to substantiate the claim that λ captures the inherent difficulty of update sequences. In Section 3, we design an algorithm that can update a *foreign-key acyclic* join in time $O(\lambda)$ amortized on any update sequence with enclosureness λ. Note that λ is only used in the analysis; our algorithm does not need to

know the value of λ. Intuitively, a join is foreign-key acyclic if the join conditions are all on primary-foreign key relationships and the graph formed by the primary-foreign key relationships contains no directed cycles; a more formal definition is given in Section 2. Foreign-key acyclic joins are very common in analytical query processing. For example, all the 22 queries in the TPC-H benchmark are foreign-key acyclic joins, combined with other relational operators such as selection, projection, and aggregation (possibly with group-by).

On the lower bound side, we show in Section 4 that no algorithm can achieve $O(\lambda^{1-\epsilon})$ amortized update time, subject to the condition $\lambda \leq \sqrt{|db|}$. Note that when $\lambda > \sqrt{|db|}$, we can just invoke the lower bound of $\Omega(|db|^{\frac{1}{2}-\epsilon})$ [8].

In Section 5, we extend our basic acyclic join algorithm to more general forms, so that *all* 22 queries in the TPC-H benchmark can be supported in our framework. The majority of the 22 TPC-H queries can be updated in $O(\lambda)$ time, except for a few (Q2, Q11, Q15, Q17, Q22), for which the update time has an additional $O(\log |db|)$ factor.

Finally, based on the newly developed algorithms, we built *AJU (Acyclic Join under Updates)*, a continuous query evaluation system, on top of Flink [9]. AJU supports all the 22 TPC-H queries under arbitrary updates. AJU can be considered as a query compiler that takes in a SQL query (our current prototype takes in a manually written query plan instead of a SQL query) and produces Flink code, which is then executed by the Flink engine. We evaluated AJU against Dynamic Yannakakis [14], DBToaster [4], as well as Trill [10]. All these systems are centralized. On the other hand, by leveraging on the power of Flink, AJU easily scales to many nodes with excellent scalability and fault tolerance. It is thus the first fault-tolerant, distributed system for answering SQL queries under updates. Experimental results show that AJU, when running in single-thread mode, already offers 2x to 80x improvements over the existing systems, and adding more workers lead to even higher speedups.

2 PRELIMINARIES

2.1 Foreign-key Acyclic Schemas

Let $R_1(\bar{x}_1), R_2(\bar{x}_1), \ldots$ be the relations in the database, where $\bar{x}_i = \{x_{i1}, x_{i2}, \ldots\}$ denotes the set of attributes of relation R_i. Let $PK(R)$ be the primary key of R, and we also use the notation $\bar{x}_i = \{\underline{x_{i1}}, x_{i2}, \cdots\}$ to indicate that $PK(R) = x_{i1}$. At any time instance, all tuples in a relation must have unique values on the primary key. For a composite primary key, we (conceptually) create a combined primary key by concatenating the attributes in the composite key, while the original attributes in the composite key are treated as regular, non-key attributes. For example, in the TPC-H schema, the relation `partsupp` with a composite primary key (ps_partkey, ps_suppkey) is rewritten into partsupp(ps_partsuppkey, ps_partkey, ps_suppkey, ...).

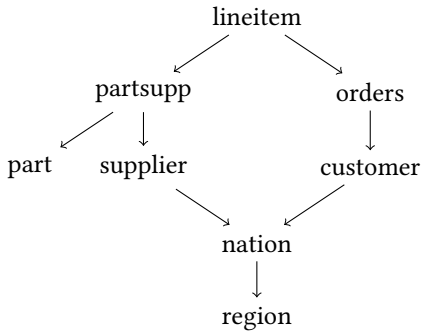


Figure 2: The foreign-key graph of TPC-H schema.

An attribute x_{ik} of a relation R_i is a *foreign key* referencing the primary key $\underline{x_{j1}}$ of relation R_j, if the x_{ik} values taken by tuples in R_i must appear in $\underline{x_{j1}}$ [21]. If a relation has a composite foreign key, we similarly create a combined attribute referencing the composite primary key. For example in the TPC-H schema, the relation `lineitem` has a composite foreign key `(l_partkey,l_suppkey)`. We create a new attribute `l_partsuppkey` referencing `ps_partsuppkey`.

We can model all the primary-foreign key relationships as a directed graph: Each vertex represents a relation, and there is a directed edge from R_i to R_j if R_i has a foreign key referencing the primary key of R_j. We call this graph the *foreign key graph*. We say that the schema is *foreign key acyclic*, if its foreign key graph is an acyclic directed graph (DAG). For example, the TPC-H schema, shown in Figure 2, is foreign key acyclic. Foreign key acyclic schemas are very common in schema design [2]; circular foreign key dependencies may cause various problems in database maintenance, and should always be avoided if possible [1].

2.2 Foreign-key Acyclic Joins

We start by considering multi-way natural joins[1] of the form

$$\sigma_{\phi_1} R_1(\bar{x}_1) \bowtie \cdots \bowtie \sigma_{\phi_n} R_n(\bar{x}_n), \tag{1}$$

where ϕ_i is a selection condition on $\bar{x}_i$.

[1] A multi-way equi-join can be written as a natural join by properly renaming the attributes. Specifically, we first find the equivalence class of attributes induced by the equalities, and then replace all attributes in the same equivalence class by the attribute name. Meanwhile, equality join conditions on composite primary keys only need to be applied on the combined attribute, not the individual attributes in the composite key. For example, the join

```
SELECT * FROM lineitem, partsupp, part
WHERE l_partkey = ps_partkey AND l_suppkey = ps_suppkey
    AND ps_partkey = p_partkey
```

We require the query to be a *foreign key join*, i.e., for any attribute x appearing in more than one relation in (1), x must be the primary key of at least one of them. We can similarly define the foreign key graph of a query: Introduce a vertex for each R_i, and build a directed edge from R_i to R_j if $PK(R_j)$ appears as an attribute of R_i. We say that the query is *foreign-key acyclic*, if this graph is a DAG with one vertex (relation) having in-degree 0. This relation is called the *root* of the query. Note that this definition implies that every relation in the multi-way join (1) must have a primary key, which is joined with another relation, except for the root relation. In the rest of the paper, the term "acyclic" always means "foreign key acyclic" unless stated otherwise.

We observe that *all* multi-way joins in the TPC-H queries satisfy these requirements. In Section 5, we will consider other relational operators such as selection, projection, and aggregations (with or without group-by).

Note that the query's foreign key graph is not necessarily a subgraph of the schema's foreign key graph, because a query may involve multiple logical copies of the same relation. For example, the multi-way join in TPC-H Q7, written as a natural join, is

$$\begin{aligned}&\texttt{lineitem}(\texttt{orderkey}, \texttt{suppkey}, \texttt{shipdate}, \dots)\\ \bowtie\ &\texttt{supplier}(\underline{\texttt{suppkey}}, \texttt{nationkey1}) \bowtie \texttt{nation}(\underline{\texttt{nationkey1}})\\ \bowtie\ &\texttt{orders}(\underline{\texttt{orderkey}}, \texttt{custkey})\\ \bowtie\ &\texttt{customer}(\underline{\texttt{custkey}}, \texttt{nationkey2}) \bowtie \texttt{nation}(\underline{\texttt{nationkey2}})\end{aligned}$$

The foreign key graph of this query is shown in Figure 3, which is not a subgraph of Figure 2. However, it is easy to see that the foreign key graph of any *foreign key join* on an acyclic schema must be acyclic.

Our notion of acyclicity differs from α-acyclicity [14]. For example, TPC-H Q5, which is the same as Q7 above except that there is only one copy of `nation` (and `nationkey1` and `nationkey2` are both just `nationkey`), is still acyclic by our definition, but is not α-acyclic. On the other hand, α-acyclic queries do not need any key constraints, so some α-acyclic queries are not acyclic by our notion, either.

2.3 Updates and Deltas

An *update* to a relation R_i is either the insertion or deletion of a tuple t in R_i. We adopt the set semantics for relations, which 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.

As a simplification, when considering updates to a query Q in the form of (1), we just discard the update if it does not pass ϕ_i. Subsequently, we will ignore all the selection operators, since it just takes $O(1)$ time to decide if t passes the condition ϕ_i or not. As a result, the database which we operate upon consists of only a subset of the original tuples, which means that the foreign-key constraint may not hold on the subset. For example, TPC-H Q7 actually has a filter condition on `nation`, and not all customers come from some nation in the filtered `nation` relation.

When inserting or deleting a tuple t in R_i, the *delta* of a query Q in the form of (1) is

$$\Delta Q = R_1 \bowtie \cdots R_{i-1} \bowtie \{t\} \bowtie R_{i+1} \bowtie \cdots \bowtie R_n.$$

If we are inserting a tuple that does not exist in R_i, the query results will be updated as $Q(db + t) := Q(db) + \Delta Q$; similarly, deleting an existing tuple from R_i will update the query results as $Q(db - t) := Q(db) - \Delta Q$. Our goal is to design a data structure and algorithms so that as to be able to enumerate both $Q(db)$ and ΔQ with constant delay.

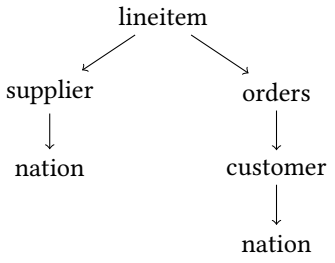


Figure 3: The foreign-key graph of TPC-H Q7.

2.4 Index Support

An important building block of our algorithms is an index structure, built on a relation R_i using one of its attributes, say x_{ik}, as the key. The index should be able to:

- enumerate all tuples in R_i in constant time per tuple;
- given any value v, enumerate all tuples whose value on x_{ik} is v with constant delay, or report that there is none;
- insert or delete a tuple in constant time; and
- use $O(|R_i|)$ memory.

A standard implementation of such an index is a hash table on all the distinct key values of x_{ik}. Each slot of the hash table stores a pointer to the list of all tuples whose value on x_{ik} is the given value. Using universal hash functions, all the above operations can be done in expected $O(1)$ time [12].

---

is written as

$$\begin{aligned}&\texttt{lineitem}(\texttt{partsuppkey}, \texttt{l\_partkey}, \texttt{l\_suppkey}, \dots)\\ \bowtie\ &\texttt{partsupp}(\underline{\texttt{partsuppkey}}, \texttt{partkey}, \texttt{ps\_suppkey}, \dots)\\ \bowtie\ &\texttt{part}(\underline{\texttt{partkey}}, \dots)\end{aligned}$$

Note that the condition `l_partkey = partkey` is not explicitly enforced by the natural join, but implicitly due to the composite key `partsuppkey`.

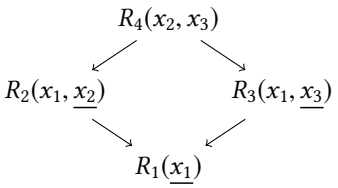


Figure 4: A simple foreign-key graph. Note that each edge $R_i \rightarrow R_j$ represents a join condition between a foreign key of R_i and the primary key of R_j.

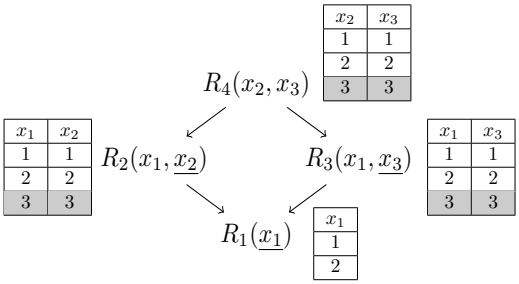


Figure 5: An instance of the query in Figure 4.

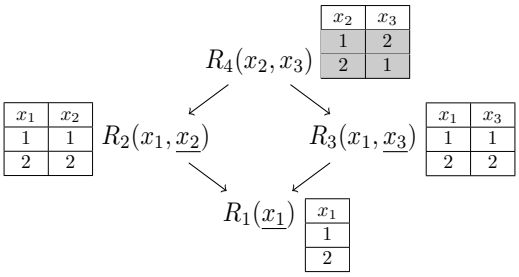


Figure 6: A counter example.

3 ALGORITHMS

3.1 Live Tuples

Given a query in the form of (1), we first construct its foreign-key DAG. A leaf in the DAG is a node with out-degree 0, while the root has in-degree 0. We say that R_j is a child of R_i if there is an edge from R_i to R_j, and R_i is a parent of R_j. We say that R_j is a descendant of R_i if there is a directed path from R_i to R_j. One important observation for acyclic joins is that every join result must include a distinct tuple in the root relation, because any tuple in the root relation can join with at most one tuple in every other relation. Thus, the key idea in maintaining the join results efficiently is to keep track of the tuples in the root relations that *can* join with one tuple in every other relation. We say that these tuples are *alive*. Once all the live tuples can be maintained, we just need constant time to retrieve each full join result.

For a relation R, let $\mathcal{C}(R)$ be the set of its child relations, $\mathcal{P}(R)$ be the set of its parent relations, and $\mathcal{D}(R)$ be the set of its descendant relations.

Definition 1. For a leaf relation R, all its tuples are *alive*. For a non-leaf relation R, a tuple t in R is *alive* if it can join with one tuple in every relation in $\mathcal{D}(R)$, i.e.

$$t \bowtie (\bowtie_{R_d \in \mathcal{D}(R)} R_d) \neq \emptyset. \tag{2}$$

However, using this definition directly to maintain the set of live tuples is expensive. We introduce the notion of *alive on* a child, which recursively depends on a child tuple being *alive*.

Definition 2. Let R be a non-leaf relation in $\mathcal{T}$, and $R_c \in \mathcal{C}(R)$ a child relation of R. A t tuple in R is *alive on* R_c if there is an *alive* tuple t_c in R_c such that $\pi_{PK(R_c)} t = \pi_{PK(R_c)} t_c$.

An example using the query in Figure 4 is given in Figure 5, where all live tuples are marked in white and non-live tuples in gray.

If whether a tuple in a relation R is alive could be determined by only checking whether it is alive on its child relations $\mathcal{C}(R)$, we would be able to close the recursion and arrive at an efficient maintenance algorithm. However, being alive on all $R_c \in \mathcal{C}(R)$ is only a necessary condition for a tuple to be alive. Figure 6 gives an example showing that this is not a sufficient condition. In this example, both tuples in R_4 are alive on both children, however, neither can produce a join result because they do not join on a common tuple in R_1. To fix the problem, before declaring a t tuple in R_1 to be alive, we need to find the tuples that join with t in its two child relations R_2 and R_3, and make sure that they have the same value on x_1, the primary key of R_1. Below, we generalize this idea and present the concept of assertion keys.

3.2 Assertion Keys

First, we observe that the mismatching problem demonstrated in Figure 6 may arise between two relations $R_i, R_j \in \mathcal{T}$ only if there exist two or more node-disjoint paths from R_j to R_i in $\mathcal{T}$. We say that such a pair of relations R_i, R_j are *unconstrained*. For example, R_1 and R_4 in Figure 4 are an unconstrained pair. A more complex example is given in Figure 7, which has 4 unconstrained pairs: $(R_1, R_3), (R_1, R_6), (R_1, R_7), (R_2, R_6)$. Note that (R_2, R_7) is not an unconstrained pair because both paths from R_7 to R_2 pass through R_6. If all paths from R_j to R_i pass a common relation, say R_k, then intuitively (a more formal statement of correctness will be given later), as long as the mismatching issue is resolved at R_k, there will be no problem at R_i, since any tuple in R_k can only join one tuple in R_i, if any at all.

For each unconstrained pair of relations (R_i, R_j), we will need to make sure that for each tuple $t_j \in R_j$ to be considered

alive, there is a unique tuple $t_i \in R_i$ that joins with t_i no matter which path is taken. Checking every possible path would be inefficient. Instead, we take the following approach. We call the primary key of R_i an *assertion key* of R_j, and add it to R_j as an additional attribute [2] In the example of Figure 7, x_1 is an assertion key of R_3, R_6, R_7, and x_2 is an assertion key of R_6. These assertion keys are extra attributes added to the relations, even though they may already exist in the relations, such as x_1 in R_3 in the example.

The value of the assertion key for each tuple $t_j \in R_j$ is defined as follows:

Definition 3. Let (R_i, R_j) be an unconstrained pair of relations so that $x = PK(R_i)$ is an assertion key at R_j. For each tuple $t_j \in R_j$, $\pi_x t_j$ is:

(1) NULL, if there is no tuple in R_i that can join with t_j;
(2) $\pi_x t_i$, if there is a unique tuple $t_i \in R_i$ that can join with t_j no matter which path is taken; or
(3) a special value $\perp$, if there are more than one tuple in R_i that can join with t_j through any path.

The following lemma gives a sufficient and necessary condition for a tuple to be alive:

Lemma 4. *For a leaf relation R, all its tuples are* alive. *For a non-leaf relation R, a tuple t in R is* alive *if and only if it is* alive *on every $R_c \in C(R)$, and for every assertion key x of R, $\pi_x t \neq \perp$.*

Proof. The "only if" direction is trivial. The "if" direction can be proved by induction on the height of R in $\mathcal{T}$. Consider any tuple $t \in R$ that is alive on every child $R_c \in C(R)$. This means that for every $R_c \in C(R)$, there is a live tuple $t_c \in R_c$ such that $\pi_{PK(R_c)} t = \pi_{PK(R_c)} t_c$. Then the key is to show that for every two children R_c and R'_c, if they have any common descendant S, t_c and t'_c should join with the same tuple in S, no matter which path is taken. This can be proved by exploiting the properties of the assertion keys. □

Note that if t is alive on at least one child of R, the assertion key of t cannot be NULL. So the condition $\pi_x t \neq \perp$ in Lemma 4 is equivalent to saying that it must fall under case (2) in Definition 3.

The remaining issue is how to maintain the assertion keys efficiently when the database is being updated. For the example in Figure 4, this is trivial since the assertion key in R_4, x_1, already exists in its two child relations R_2 and R_3. This may not be the case in general. Given a general DAG $\mathcal{T}$, we will need to add the assertion keys to certain relations as *auxiliary attributes*. More precisely, auxiliary attributes must be added so that the following condition holds:

[2] We add attributes to the relations for notational convenience; in the actual implementation, we create another relation to manage the extra attributes. The additional relation uses the same primary key as original relation, and contains all extra attributes.

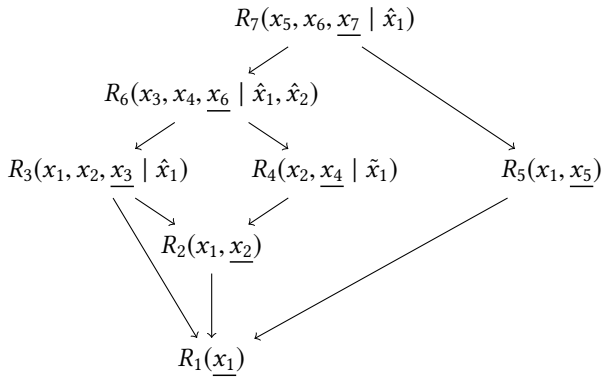


Figure 7: Assertion keys (denoted with hats) and auxiliary attributes (denoted with tilde). Attributes after the bar are extra attributes added to each relation.

Condition 5. *For each unconstrained pair of relations (R_i, R_j), and for every $R_c \in C(R_j)$, there is at least one path from R_i to R_c on which all relations have $PK(R_i)$ as an original attribute, an assertion key, or an auxiliary attribute.*

For the example of Figure 7, after adding x_1 as an auxiliary attribute of R_4 (denoted with tilde), one can verify that the condition above is satisfied. In general, finding the minimum number of auxiliary attributes to add is an NP-hard problem with respect to the size of the DAG. But since we regard the query size as a constant, we can just do a brute-force search to find the optimal solution. Furthermore, this is only done once, when the query is first registered in the system, but not during updates.

3.3 The Algorithms

Data structures. Let $L(R)$ be the set of live tuples in R and $N(R)$ the set of non-live tuples in R. The data structure $\mathcal{D}(db)$ maintained by our algorithm consists of the following indexes for each relation $R(\underline{x_1}, x_2, \dots)$.

- $I(L(R))$: An index on $L(R)$, using x_1 as the key.
- $I(N(R))$, for non-leaf R: An index on $N(R)$, using x_1 as the key.
- For a non-leaf R, a counter $s(t)$ for each tuple $t \in R$ that is equal to the number of children of R on which t is alive. This counter is stored together with the tuple in $I(N(R))$.
- $I(R, R_c)$, for non-leaf R and each child $R_c \in C(R)$: An index on $\pi_{x_1, PK(R_c)} R$ using $PK(R_c)$ as the key.

It can be easily shown that all the indexes take linear space.

Lemma 6. $|\mathcal{D}(db)| = O(|db|)$.

Algorithms. The algorithm to insert a tuple t into R is shown in Algorithm 1. We assume that t does not exist in R,

Algorithm 1: Insert(t, R)

Input: $\mathcal{D}(db)$
Output: $\mathcal{D}(db + t)$ and ΔQ

$\Delta Q \leftarrow \emptyset$
if *R is not a leaf* **then**
  $s \leftarrow 0$
  foreach $R_c \in C(R)$ **do**
    $I(R, R_c) \leftarrow I(R, R_c) + (\pi_{PK(R_c)} t \rightarrow \pi_{PK(R), PK(R_c)} t)$
    if $\pi_{PK(R_c)} t \in I(R_c)$ **then** $s(t) \leftarrow s(t) + 1$
  if $s(t) = |C(R)|$ **then**
    foreach $R_c \in C(R)$ **do**
      $t_c \leftarrow$ look up $I(R_c)$ with key $\pi_{PK(R_c)} t$
      foreach *assertion key x of R* **do**
        if $x \in R_c$ **then**
          if $\pi_x t =$ NULL **then** $\pi_x t \leftarrow \pi_x t_c$
          else if $\pi_x t \neq \pi_x t_c$ *or* $\pi_x t = \perp$ **then** $\pi_x t \leftarrow \perp$
if *R is a leaf or ($s(t) = |C(R)|$ and all assertion keys are not $\perp$)* **then**
  Insert-Update(t, R, t)
else
  $I(N(R)) \leftarrow I(N(R)) + (\pi_{PK(R)} t \rightarrow t)$

Algorithm 2: Insert-Update(t, R, *join_result*)

Input: $t \in R$ is becoming alive, *join_result* is the current new join result produced by t
Output: Update all affected tuples and ΔQ

$I(L(R)) \leftarrow I(L(R)) + (\pi_{PK(R)} t \rightarrow t)$
if *R is the root of $\mathcal{T}$* **then**
  $\Delta Q \leftarrow \Delta Q \cup \{join\_result\}$
else
  $P \leftarrow$ look up $I(R_p, R)$ with key $\pi_{PK(R)} t$
  foreach $t_p \in P$ **do**
    $s(t_p) \leftarrow s(t_p) + 1$
    if $s(t_p) = |C(R_p)|$ **then**
      foreach $R_c \in C(R_p)$ **do**
        $t_c \leftarrow$ look up $I(R_c)$ with key $\pi_{PK(R_c)} t_p$
        foreach *assertion key x of R_p* **do**
          if $x \in R_c$ **then**
            if $\pi_x t =$ NULL **then** $\pi_x t_p \leftarrow \pi_x t_c$
            else if $\pi_x t_p \neq \pi_x t_c$ *or* $\pi_x t_p = \perp$ **then** $\pi_x t_p \leftarrow \perp$
      if *$\pi_x t_p \neq \perp$ for every assertion key x* **then**
        $I(N(R_p)) \leftarrow I(N(R_p)) - (\pi_{PK(R_p)} t_p \rightarrow t_p)$
        Insert-Update(t_p, R_p, *join_result* $\bowtie t_p$)

which can be checked in $O(1)$ time. The algorithm updates the data structure $\mathcal{D}(db)$ and computes the delta ΔQ. When a new tuple t is inserted into relation R, we need to find out whether t is alive or not. If R is a leaf relation, t is alive by definition. Otherwise, we need to compute $s(t)$. To do so, for each child $R_c \in C(R)$, we look up $I(L(R_c))$ with key $\pi_{PK(R_c)} t$. If there is a tuple in $I(L(R_c))$ with key $\pi_{PK(R_c)} t$, that means t is alive on R_c. In the meantime, we maintain the assertion keys and auxiliary keys. After computing $s(t)$, we check if $s(t) = |C(R)|$ and all assertion keys not equal to $\perp$. If so, t is alive and we add it to $I(L(R))$; otherwise we add it to $I(N(R))$. Meanwhile, we also need to update the index $I(R, R_c)$ for each $R_c \in C(R)$.

If t is not alive, then it will not affect other tuples' status nor the join result. If it is alive, then it will increase the $s(t_p)$ value for each tuple $t_p \in R_p$ that joins with t. These tuples can be found by probing the index $I(R, R_p)$. This may make these tuples alive, which in turn may trigger further updates in a bottom-up fashion. When this process reaches the root of $\mathcal{T}$, a new join result will be included in ΔQ. This process is described in Algorithm 2.

For example, suppose a new tuple (3) is inserted into relation R_1 on the instance shown in Figure 5. First, because R_1 is a leaf, the tuple is alive by definition. Then the tuple (3, 3) in R_3 and the tuple (3, 3) in R_2 become alive, which further turns the tuple (3, 3) in R_4 to alive.

Deletions of tuples can be handled in a similar fashion, but slightly simpler. To delete a tuple t from R, we first check if it is alive. If it is non-alive, then we simply delete from $I(N(R))$. Otherwise, it will affect the status of tuples in R_p. More precisely, it will decrement the $s(t_p)$ value for every $t_p \in R_p$ that joins with t. If a t_p is currently alive, this will make it non-alive, which may trigger further updates recursively in a bottom-up fashion. When a live tuple in the root relation becomes non-alive or deleted, its corresponding join result will be deleted (included in ΔQ). Finally, we also need to update the index $I(R_p, R)$. The deletion process is described in Algorithm 3 and 4.

After computing ΔQ, we can obviously enumerate tuples in ΔQ with constant delay. To enumerate the full join results, we simply retrieve all tuples in $L(R)$ at the root relation. For each tuple t, we find its joining tuple in every other relation in a top-down manner. This takes $O(1)$ time per tuple so we can enumerate all join results with constant delay.

Algorithm 3: Delete(t, R)

Input: $\mathcal{D}(db)$
Output: $\mathcal{D}(db - t)$ and ΔQ

$\Delta Q \leftarrow \emptyset$
if $t \in L(R)$ **then**
  Delete-Update(t, R, t)
else
  $I(N(R)) \leftarrow I(N(R)) - (\pi_{PK(R)} t \rightarrow t)$
if *R is not the root of $\mathcal{T}$* **then**
  $I(R_p, R) \leftarrow I(R_p, R) - (\pi_{PK(R)} t \rightarrow \pi_{PK(R_p), PK(R)} t)$

Algorithm 4: Delete-Update(t, R, *join_result*)

Input: $t \in R$ is becoming non-alive, *join_result* is the current join result involving t
Output: Update all affected tuples and ΔQ

$I(L(R)) \leftarrow I(L(R)) - (\pi_{PK(R)} t \rightarrow t)$
if *R is the root of $\mathcal{T}$* **then**
  $\Delta Q \leftarrow \Delta Q \cup \{join\_result\}$
else
  $P \leftarrow$ look up $I(R_p, R)$ with key $\pi_{PK(R)} t$
  foreach $t_p \in P$ **do**
    if $t_p \in N(R_p)$ **then**
      $s(t_p) \leftarrow s(t_p) - 1$
    else
      $s(t_p) \leftarrow |C(R)| - 1$
      $I(N(R_p)) \leftarrow I(N(R_p)) + (\pi_{PK(R_p)} t \rightarrow t)$
      Delete-Update($t_p, R_p, join\_result \bowtie t_p$)

3.4 Update Time Analysis

The major concern in the update time is that a single update may trigger the status change of a lot of tuples. For the query in Figure 4, for example, it may happen that a single deletion in R_1 may make many tuples in R_3 and R_2 non-alive, which in turn may render a lot of join results invalid. However, we show below that the amortized update time is $O(\lambda)$, where λ is the enclosureness of the update sequence. Intuitively, the update cost will be high if a tuple affecting a lot of other tuples is repeatedly deleted and inserted back. However, such update sequences must have high enclosureness; see e.g. Figure 1(d). The analysis below formalizes this intuition.

The total update cost of the entire update sequence is asymptotically bounded by the number of times all the $s(t)$ counters can change. The following lemma connects this quantity with the enclosureness of the update sequence.

Lemma 7. *For any tuple $t \in R$ with lifespan $i = (i_s, i_e)$, $s(t)$ changes at most $O(\lambda(i))$ times.*

Proof Sketch. The proof of this lemma is lengthy, so we just give a sketch here. Each tuple that can join with t affects $s(t)$ at most twice, once upon insertion and once upon deletion, so it suffices to bound the number of such tuples. A child relation $R' \in C(R)$ contains at most $\lambda(i)$ such tuples, since they must have the same primary key for them to join with t. Then the bound follows from the primary key constraint that no two tuples with same primary key can be alive at the same time.

However, things get more complicated when we go further down. Consider a child relation $R'' \in C(R')$. Its tuples that can join with t may not necessarily have the same primary key, because they can join with different tuples in R', which in turn join with t. An easy but loose bound on the number of tuples in R'' that can join with t is thus $O(\lambda(i)^2)$, and continuing this analysis would lead to a final bound of $O(\lambda(i)^h)$, where h is the height of $\mathcal{T}$.

To tighten the bound, we make the key observation that, although tuples in R'' that can join with t may not have the same primary key, many of them must have disjoint lifespans, since they need to join with tuples in R' with disjoint lifespans. There are at most $\lambda(i)$ such tuples. Some other tuples in R'' may indeed have overlapping lifespans, but each tuple in R' will have at most two such tuples in R'', so the number of such tuples is also $O(\lambda(i))$. By a careful induction proof, we can show that the $O(\lambda(i))$ bound holds for any descendant relation, although the hidden constant grows exponentially in the height of $\mathcal{T}$. □

Now suppose the update sequence is $\mathcal{I}$, then by Lemma 7 the total update cost is $O\left(\sum_{i \in \mathcal{I}} \lambda(i)\right)$, which is $O\left(\frac{\sum_{i \in \mathcal{I}} \lambda(i)}{|\mathcal{I}|}\right) = O(\lambda)$ amortized.

Theorem 8. *An acyclic join Q can be updated in $O(\lambda)$ time amortized on any update sequence with enclosureness λ.*

An easy but important corollary is that the amortized size of each ΔQ is also bounded by $O(\lambda)$, because our update algorithms construct ΔQ during the update process, so its size cannot be more than the update time.

Corollary 9. *For an acyclic join Q, the amortized size of ΔQ is $O(\lambda)$ over any update sequence with enclosureness λ.*

3.5 Implication to the Change Propagation Framework

The standard change propagation framework [11, 20] can also be applied to maintain the results of a multi-way join Q incrementally. Here, one arranges the relations in a tree (see e.g. Figure 8), where each leaf corresponds to a relation, and each internal node corresponds to an intermediate join, also called a *view*. When a tuple t is inserted or deleted in a relation, say R_1, we follow the leaf-to-root path to compute

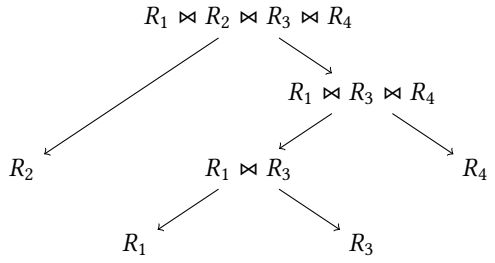


Figure 8: A change propagation join tree that guarantees $O(\lambda)$ update time for the query in Figure 4.

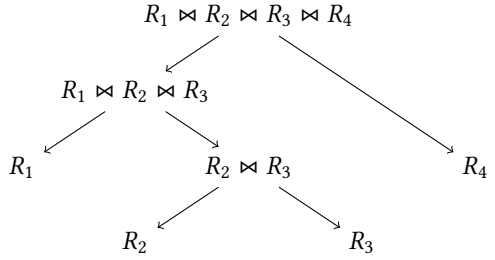


Figure 9: A change propagation join tree with linear-time update cost for the query in Figure 4.

ΔQ, i.e., we first compute $\Delta(R_1 \bowtie R_3) = \{t\} \bowtie R_3$, then $\Delta(R_1 \bowtie R_3 \bowtie R_4) = \Delta(R_1 \bowtie R_3) \bowtie R_4$, and finally $\Delta Q = \Delta(R_1 \bowtie R_3 \bowtie R_4) \bowtie R_2$.

However, the generic change propagation framework does not specify what join tree should be used, and as matter of fact, different join trees may lead to significantly different update costs. For example, Figure 8 and 9 are two possible join trees for the multi-way join in Figure 4. As we will show shortly, the one in Figure 8 also achieves $O(\lambda)$ update time while the update cost using Figure 9 is linear.

In fact, the $O(\lambda)$ guarantee of the join tree in Figure 8 follows from its equivalence to our algorithm. Specifically, we can prove the following correspondence result. Recall that we use $L(R)$ to denote the set of live tuples in R, and $\mathcal{D}(R)$ the set of descendant relations of R.

Lemma 10. *Given an acyclic multi-way join, for any relation R, we have*

$$L(R) = R \ltimes (\bowtie_{R_d \in \mathcal{D}(R)} R_d).$$

Proof. By the definition of live tuples. □

Following Theorem 8 and 10, we can show that there is always a join tree under the change propagation framework that guarantees $O(\lambda)$ update time over acyclic joins.

Theorem 11. *For any acyclic join, there exists a join tree under the change propagation framework that achieves an amortized update cost of $O(\lambda)$ on any update sequence with enclosureness λ.*

Proof. The join tree can be constructed as follows. Traverse the foreign-key DAG bottom-up. For a leaf relation R, let $V(R) := R$. For a non-leaf relation R, introduce a view $V(R) := R \bowtie (\bowtie_{R_d \in \mathcal{D}(R)} R_d)$. If R has more than one child, compute $V(R)$ using a left-deep (or right-deep) join tree in which every join is a foreign-key join. Then invoke Theorem 8 and 10. □

Now, we see that the join tree in Figure 8 is constructed following the proof of Theorem 11, thus achieving $O(\lambda)$ update time. On the other hand, the join tree in Figure 9 will incur a linear update cost, and the intermediate views $R_2 \bowtie R_3$ and $R_1 \bowtie R_2 \bowtie R_3$ may have quadratic size.

Note that unlike query optimization on static databases, finding the optimal query plan over streaming data at present is still done on an ad hoc basis. In fact, to perform complex analytic queries in most streaming systems (including Trill, Spark Streaming, and Flink) under the change propagation framework, the user has to manually write the query plan. Thus, Corollary 11 actually provides a guidance to writing the query plan for a multi-way acyclic join that achieves good update time. In Section 6, we experimentally evaluate the performance of different plans under the change propagation framework, and will see that the plan indicated by Corollary 11 can be order-of-magnitude faster than sub-optimal plans.

Furthermore, although we have shown that theoretically, there is a join tree under the change propagation framework that can also achieves $O(\lambda)$ update, the hidden constant can be much larger than that in our algorithm. It incurs a lot of overhead as many intermediate views need to be maintained. On the other hand, our algorithm does not have any intermediate views, and the only overhead is the assertions keys and the indexes. Note that in the change propagation framework, indexes are also needed on the intermediate views in order to update them efficiently.

Finally, it should be pointed out that we have assumed that changes may happen to any of the relations, and we aim at providing a performance guarantee on any update sequence. If changes can only happen to a particular relation, then certain join trees can be more efficient. For example, although the join tree in Figure 9 is bad for updates to arbitrary relations, it will be ideal if updates only happen to R_4.

4 LOWER BOUND

In this section, we show that the $O(\lambda)$ update time is essentially the best possible for dealing with update sequences with enclosureness λ, thus establishing the optimality of Theorem 8.

We consider the simple query $\psi_1 := \pi_x(R_1(x,y) \bowtie R_2(\underline{y}))$, which is clearly foreign-key acyclic, and show that it is not possible to achieve $O(\lambda^{1-\epsilon})$ update time and $O(\lambda^{1-\epsilon})$ enumeration delay simultaneously, unless the *online matrix-vector multiplication conjecture (OMv)* fails. Note that the lower bounds of [8] rely on the same conjecture.

Conjecture 12 (OMv). *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 $v_1, \ldots, v_n$, compute Mv_i for each i, where the matrix-vector multiplication is performed over the Boolean semiring. The algorithm is required to output Mv_i before v_{i+1} is revealed, for $i = 1, \ldots, n-1$.*

Theorem 13. *Suppose there exists an algorithm with amortized $O(\lambda^{1-\epsilon})$ update time that can enumerate $\psi_1 := \pi_x(R_1(x,y) \bowtie R_2(\underline{y}))$ with delay $O(\lambda^{1-\epsilon})$ over any update sequence with enclosureness $\lambda \le \sqrt{|db|}$, then there is an algorithm that can solve the OMv problem in time $O(n^{3-\epsilon})$.*

Proof Sketch. We encode the matrix by $R_1(x,y)$, and use $R_2(y)$ to encode the vectors. Given an algorithm with amortized $O(\lambda^{1-\epsilon})$ update time that can enumerate ψ_1 with delay $O(\lambda^{1-\epsilon})$, we construct an update sequence $\mathcal{I}$ as follow: (1) let $n = \lambda$ and the matrix M has size $n \cdot n$, we first encode the matrix M into $R_1(x,y)$; (2) for each vector v_i, we encode it into $R_2(y)$ and output the query result. After that, we reset $R_2(y)$ and receive vector v_{i+1} until n vectors are all processed. For each record $t \in R_1(x,y)$, its lifespan is $(1, +\infty)$. For the i-th vector, each record $t \in R_2(y)$ has lifespan $(n \cdot (i-1), n \cdot i)$ as the vector is inserted at time $n \cdot (i-1)$ and deleted at time $n \cdot i$.

By the definition of enclosureness, $\forall i(t) = (i_s, i_e), t \in R_1(x,y), \lambda(i) = n$ and $\forall i(t) = (i_s, i_e), t \in R_2(y), \lambda(i) = 1$, then the enclosurenessof $\mathcal{I}$ is $\frac{n \cdot n^2 + 1 \cdot n^2}{n^2} = O(n) = O(\lambda)$. Thus, if there exists an algorithm can achieve amortized $O(\lambda^{1-\epsilon})$ update time and $O(\lambda^{1-\epsilon})$ enumeration delay, the online matrix-vector multiplication problem can be solved in $O(n^2 \cdot \lambda^{1-\epsilon}) = O(n^{3-\epsilon})$ time. Note that the construction above requires a database of size at least $n^2 = \lambda^2$. □

5 GENERAL SQL QUERIES

Using Theorem 8, Corollary 11 as the base case, we can extend the class of queries supported recursively, using standard change propagation algorithms. The following results are straightforward (all time bounds can be amortized):

Theorem 14. *Suppose Q (resp. Q') can be updated in $f(|db|)$ (resp. $f'(|db|)$) time and has delta size $g(|db|)$ (resp. $g'(|db|)$), then*

- *$\sigma(Q), \pi(Q)$, $\mathcal{G}_{\mathsf{SUM}}(Q)$, and $\mathcal{G}_{\mathsf{COUNT}}(Q)$ (possibly with group-by) can be updated in $O(f(|db|) + g(|db|))$ time and has delta[3] size $O(g(|db|))$;*
- *$\mathcal{G}_{\mathsf{MAX}}(Q)$ and $\mathcal{G}_{\mathsf{MIN}}(Q)$ (possibly with group-by) can be updated in $O(f(|db|) + g(|db|)\log|db|)$ time and has delta size $O(g(|db|))$;*
- *$Q \cup Q'$ and $Q - Q'$ can be supported in $O(f(|db|) + f'(|db|))$ time and has delta size $O(g(|db|) + g'(|db|))$.*

For example, let Q be an acyclic multi-way join in the form of (1), so it has $f(|db|) = O(\lambda)$ and $g(|db|) = O(\lambda)$. Then the query

```
SELECT x, SUM(z') FROM
    (SELECT x, y, MAX(z) AS z' FROM Q GROUP BY x, y)
GROUP BY x
```

can be updated in $O(\lambda \log |db|)$ time with delta size $O(\lambda)$.

Importantly, however, the join operator is missing in Theorem 14. Indeed, $Q \bowtie Q'$ has no provable update time, if we are only given the fact that Q and Q' can each be updated. Nevertheless, most TPC-H queries can be rewritten without using $Q \bowtie Q'$, i.e., all the joins can be pushed into a number of acyclic multi-way joins in the form of (1), which are then connected by operators supported in Theorem 14.

Still, some TPC-H queries cannot be rewritten, as they do a join between sub-queries that perform aggregations after joins. Fortunately, we observe that all such TPC-H queries follow a particular pattern, which is captured by the two composite operators we introduce below.

Join-select. A *join-select* takes the form of

$$\sigma_{x_2 ? x_3}(Q(\underline{x_1}, x_2, \ldots) \bowtie Q'(x_1, x_3, \ldots)), \tag{3}$$

where ? can be any of $<, \le, >, \ge, =$. Here Q and Q' could be queries after recursively applying Theorem 14 and the join key x_1 is the primary key of Q. Note that handling the join and the selection separately may be costly, even if both Q and Q' have bounded deltas, because a single change in Q may affect many join results.

Below, we present an algorithm to process this join-select as one operator. For each value $v \in \mathrm{dom}(x_1)$, we maintain $Q'(v, x_3, \ldots)$ in a binary tree by the order of x_3. In addition, we keep a pointer in the binary tree to the predecessor of $\pi_{x_2} Q(v, x_2, \ldots)$. Note that this pointer allows us to enumerate the full query results with constant delay.

An update may happen in either Q or Q'. To process an update in Q', we just update the corresponding binary tree in $O(\log |db|)$ time. In this case, the delta of the join-select may only contain the updated tuple, if it satisfies the condition $x_2 ? x_3$. Next, consider an update to a tuple $t = (x_1 = a_1, x_2 = a_2, \ldots) \in Q$, changing its x_2 attribute to a_2'. In this case, we

[3] For aggregation queries, the delta will consist of changes in the aggregates.

need to update the pointer to the new location corresponding to a_2'. The delta thus consists of all tuples between the old and new locations of the pointer, which can be enumerated with constant delay. Note that although we can enumerate the delta with constant delay, there is no bound on the size of the delta, so we cannot further apply Theorem 14 after a join-select.

Join-select-aggregate. A variant of the above join-select operator is *join-select-aggregate*:

$$x_1\mathcal{G}_{\mathsf{AGG}}(\sigma_{x_2?x_3}(Q(\underline{x}_1, x_2, \dots) \bowtie Q'(x_1, x_3, \dots))), \qquad (4)$$

namely, after a join-select, an aggregate is computed for each group of join results sharing the same x_1 value (i.e., `GROUP BY` x_1). The aggregation function `AGG` can be `SUM`, `COUNT`, `MAX` or `MIN`. This composite operator appears in, e.g., TPC-H Q17.

We can use the same algorithm described above to handle this composite operator in $O(\log |db|)$ time. The only difference is that, in the binary tree on $Q'(v, x_3, \dots)$, we also annotate each internal node of the binary tree with the partial aggregate of all tuples in its subtree. For an update in Q, we can use the binary tree to recompute the new aggregate in $O(\log |db|)$ time. For an update in Q', we can also update the binary tree in $O(\log |db|)$ time and find the delta. Note that for this operator, the delta has constant size for each update in either Q or Q'.

Theorem 15. *Suppose Q (resp. Q') can be updated in $f(|db|)$ (resp. $f'(|db|)$) time and has $|\Delta Q| = g(|db|)$ (resp. $g'(|db|)$), then*

- *query (3) can be updated in $O(f(|db|) + f'(|db|) + (g(|db|) + g'(|db|)) \log |db|)$ time;*
- *query (4) can be updated in $O(f(|db|) + f'(|db|) + (g(|db|) + g'(|db|)) \log |db|)$ time and has delta size $O(g(|db|) + g'(|db|))$.*

TPC-H queries. By combining Theorem 8, 14, 15, and Corollary 11, we are able to handle all the 22 TPC-H queries with the performance guarantees that depend on λ. In particular, 5 queries (Q2, Q11, Q15, Q17, Q22) can be updated in $O(\lambda \log |db|)$ time amortized, while the rest in $O(\lambda)$ time. Both full query results and deltas can be enumerated with constant delay, although the delta size may not be bounded if a join-select is used.

6 EXPERIMENTAL EVALUATION

6.1 System Design

We have implemented our algorithms and built a system prototype called AJU (Acyclic Join under Updates). We built AJU on top of Flink, and the update sequence is fed to our system as a data stream. Each record in the stream specifies an update, including (1) a flag indicating whether it is an insertion or deletion; (2) which relation this update is applied to, and (3) the attributes of the tuple to be inserted, or just the primary key in case of a deletion.

AJU can be regarded as a query compiler that takes in a SQL query and produces Flink code that can be executed by the Flink engine. Its system architecture is shown in Figure 10. This design allows us to inherent all the benefits of Flink: good scalability through distributed stream processing, exactly-once semantics, fault-tolerance, and the ability to work with a variety of data sources and sinks.

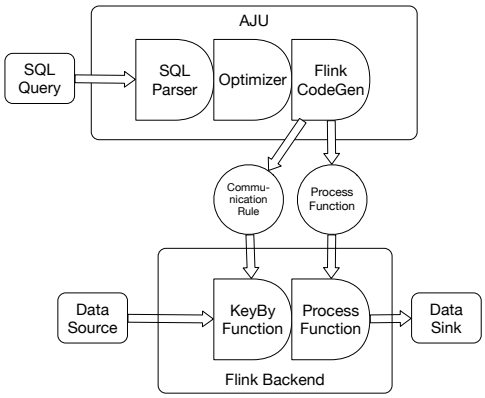


Figure 10: System architecture of AJU

More precisely, AJU generates two Flink functions: a `KeyBy` function and a `Process` function. The `KeyBy` function instructs Flink on how to dispatch tuples to workers while the `Process` function implements the actual query evaluation algorithms. The data structures needed by the algorithms are stored as *keyed state* managed by Flink in a distributed and fault-tolerant fashion.

To dispatch tuples to workers in a load-balanced fashion, we adopt ideas from the HyperCube algorithm [3, 6]. The worker are arranged as a multi-dimensional grid, where each dimension corresponds to a join key. The dimensions of the grids are found by solving an optimization problem [3, 6]. Each tuple will be sent to a subset of workers that correspond to a subspace of the grid, and the hypercube arrangement guarantees that each join result will be reported by exactly one worker.

For each relation, AJU will implement the index mentioned in Section 3.3 with standard hash map. In addition, if the relation needs an extra relation, AJU will keep the extra information in a hash map, where the key is the primary key of the extra relation.

At present, the SQL parser and optimizer of AJU are still under development. Our current prototype takes a manually written query plan as input and produces Flink code as output.

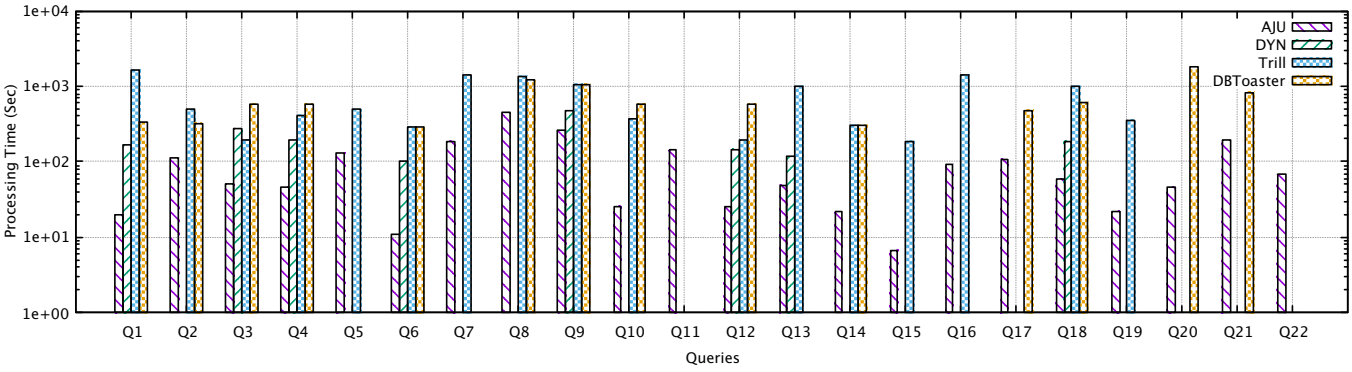


Figure 11: Running time of AJU, DYN, Trill and DBToaster.

6.2 Experimental Setup

Queries and updates. We tested all the 22 TPC-H queries in our experiments, and used the TPC-H data generator to generate test data, using a scale factor of 5. We generated two types of update sequences. The first is a FIFO sequence, where we first insert 20% of the whole data set. Next, after inserting each new tuple to the database, we immediately delete the oldest one from the database. FIFO update sequences have $\lambda = 1$, so AJU can handle each update in $O(1)$ time amortized. The other update sequence is more general, where we adjust the order of the insertions and deletions to obtain different λ.

Query processing engines compared. We compared AJU with three continuous query evaluation / incremental view maintenance systems in our experimental study: (1) the Dynamic Yannakakis algorithm (DYN) [14], which is currently the best algorithm for supporting α-acyclic queries under arbitrary updates; (2) DBToaster [4], which is currently the best Higher-order Incremental View Maintenance engine, and (3) Trill [10], which is a continuous query evaluation system over streaming data using the standard change propagation framework.

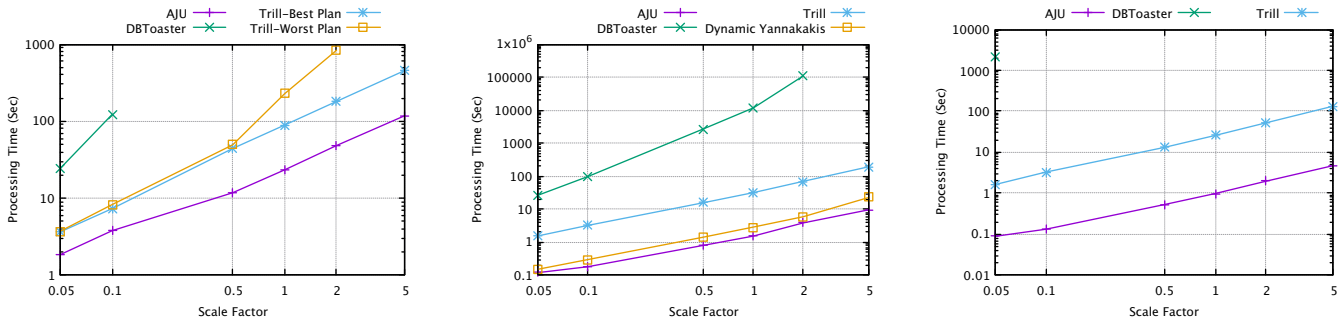
Currently, DYN is not open-source. We obtained its hand-written code for some of the TPC-H queries. DBToaster is a general query processing engine that accepts SQL queries directly. Trill requires the user to provide query plans under the change propagation framework, using basic relational operators like join, filter and user-defined aggregation function. As mentioned in Section 3.5, different queries plans will affect the performance. During the experiments, we chose multiple query plans for each query and report the best one. Trill does not support all relational operators, and as a result, we were not able to write Trill plans for Q11, Q17, and Q20-22.

Experimental environment. All experiments were performed on a machine equipped with an 8-core Intel Xeon E5-2682v4 2.5 GHz processor. Since DYN, DBToaster, and Trill are all centralized, to obtain a fair comparison, we also ran AJU in single-thread standalone mode. In addition, when testing AJU in the distributed setting, we used a small cluster of 3 such machines. All machines are running Linux, with Scala 2.11.8, JVM 1.8.0, .NET Core 2.1.504 and Flink 1.6.1. Each query was evaluated 10 times with each engine and we report the average wall-clock time. We allocated 32 GB of main memory to the JVM and forced garbage collection before each test. We ran each experiment with a two-hour timeout, not including I/O.

6.3 Experiment Results

Running time comparison. Figure 11 shows the running times of the four systems on the FIFO update sequence for all 22 TPC-H queries. Missing results are due to the query not supported, no code (in the case of DYN), or exceeding the two-hour time limit.

From the results, we can draw the following observations. (1) On α-acyclic join queries, AJU provides a speedup of 2x to 10x compared with DYN, 2x to 80x compared with Trill and 5x to 800x compared with DBToaster. Meanwhile, recall that among these queries, DYN provides constant update time guarantee only on Q1, while AJU provides this guarantee on all of them, though this constant depends on the query. (2) On queries that DYN cannot handle, AJU is 5x to 30x faster than Trill, while DBToaster is 2 to 4 orders of magnitude slower than AJU and Trill. (3) The experiment results also suggest AJU performs better at those queries with complex aggregation functions. For example, TPC-H Q1 does not require any join between tables but contains 8 aggregation functions, where Trill performs much worse than AJU or DYN. One possible reason is that Trill handles these aggregations separately, while AJU and DYN consider them as one aggregation.



(a) Q5 (b) Q13 (c) Q15

Figure 12: Processing time for Q5, Q13, Q15 under different scale factor

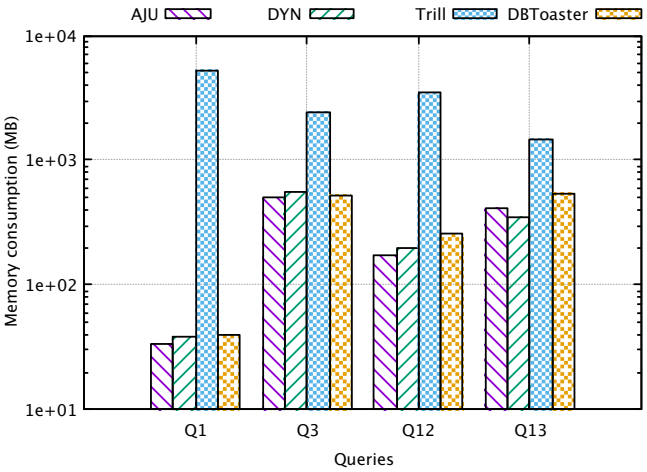


Figure 13: Memory consumption of AJU, Trill, DYN and DBToaster

Scalability. To test the scalability of AJU, we varied the data size with the scale factor ranging from 0.05 to 5. The update sequences are still FIFO. The experimental results are shown in Figure 12, running on Q5, Q13 and Q15, respectively. First, from the results we see that the running time of AJU scales linearly as the number of updates (please note that both x-axis and y-axis use logarithmic scale). This is to be expected, since AJU spends constant time processing each update on FIFO update sequences. Although the theoretical update time is $O(\log |db|)$ on Q15, the `MAX` aggregate can be maintained in $O(1)$ time in practice by the technique of [23].

Comparison between query plans. As mentioned in Section 3.5, under the standard change propagation framework, one has to choose an appropriate query plan. Theorem 11 suggests that the plan corresponding to our algorithm is optimal, while a sub-optimal plan might have a much worse performance. To verify this claim in practice, we used two query plans on Trill to run Q5, one following Theorem 11 and one does not. The experiment result is shown in Figure 12(a). As we can see, the running time of optimal query plan scales linearly as the number of updates, but the worst query plan scales exponentially. In fact, with the worst plan, Trill broke down after running 1 hour on the 5 GB data set.

Memory consumption. Figure 13 shows the memory consumption of the four systems. Compared to Trill, AJU and DYN only require 1% to 30% memory consumption. This is expected, as AJU and DYN does not materialize any auxiliary views, but Trill needs to materialize some auxiliary views following the change propagation framework. Figure 14 shows the memory consumption of AJU and Trill on Q5 with different scale factors ranging from 0.1 to 5. As we can see, the memory consumption of AJU scales linearly as the instance size, and consistently uses less memory than Trill, even with best query plan.

Comparison on different λ. By our analysis, the update cost of AJU closely depends on λ, the enclosureness of the update sequence. To verify this claim, we tested Q5 and Q13 on different update sequences that has the same length but different λ. The experiment result is shown in Figure 15 (note that both x-axis and y-axis use logarithmic scale). For Q5, we see that indeed, the processing time of AJU increases as λ, which is expected. On the other hand, Trill cannot finish the test point with $\lambda = O(|db|)$.

For Q13, on the other hand, we see that the running time of AJU does not increase as λ. This is because Q13 is a simple two-way join plus two group-by aggregations. In fact, it can be shown that our algorithm can achieve constant-time update on arbitrary update sequences. Note that this does not contradict our lower bound, which only says that for *certain* queries, one cannot achieve $O(\lambda^{1-\epsilon})$ time. On the other hand, DYN failed to finish the test point with $\lambda = O(|db|)$ after

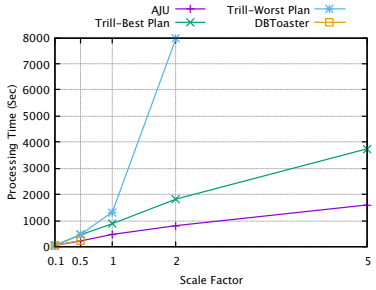
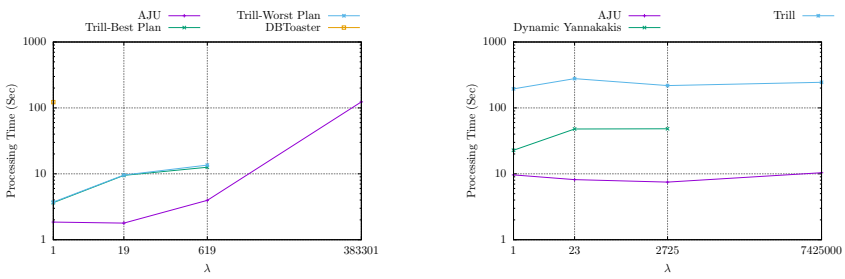
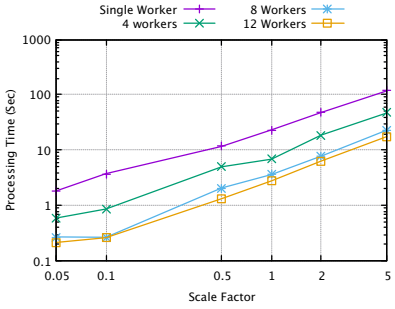


Figure 14: Memory consumption for Q5 under different scale factor

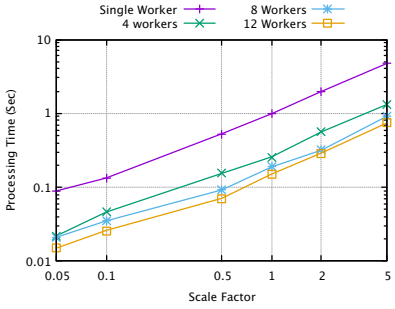


(a) Q5 (b) Q13

Figure 15: Processing time for Q5 and Q13 under different λ



(a) Q5



(b) Q15

Figure 16: Processing time for Q5 and Q13 under different parallelism

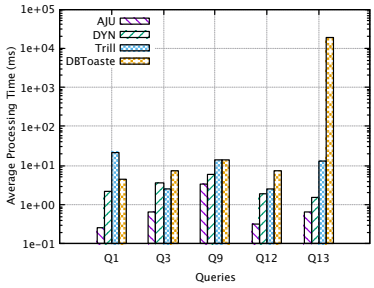


Figure 17: Enumeration latency.

Latency in delta enumeration. We also measured the latency in delta enumeration, i.e., the time from the insertion/deletion of a tuple to the output of its corresponding delta in the query result. The results are shown in Figure 17. As expected, due to the fast update time, AJU yields lower latency than the other systems.

Distributed query processing. Finally, to test the performance of AJU in a distributed environment, we built a small cluster of 3 machines, and varied the parallelism from 1 to 12. We could not perform this experiments with the other three systems as they do not support distributed query processing. The result is shown in Figure 16(a) and 16(b) for query 5 and query 15, using FIFO update sequences. As we can see, we obtain further speedups as we increase the parallelism. However, the marginal gain gets smaller as we add more workers. This is in fact to be expected, since by the HyperCube analysis, the speedup of using p workers to process a k-way join is on the order of $p^{1/k}$, i.e., sublinear in p. In addition, Flink has more scheduling overhead when managing more worker nodes.

7 CONCLUSIONS

In this paper, we have presented a new algorithm for evaluating acyclic multi-way joins under updates, whose cost depends on the enclosureness, a measure we introduce to capture the inherent hardness of the update sequence. For many practically relevant scenarios, such as insertion-only, first-in-first-out, the enclosureness is a small constant, but can be high in the worst case. We also present a lower bound showing that the update sequences with high enclosureness are inherently difficult to handle. Based on this core algorithm, as well as a number of other relational operators, we built AJU, a continuous query evaluation system on top of Flink. AJU is able to handle all the 22 TPC-H queries with provable performance guarantees.

ACKNOWLEDGMENTS

This work has been supported by HKRGC under grants 16202317, 16201318, and 16201819, as well as a grant from Alibaba Group. We would also like to thank the anonymous reviewers who have made valuable suggestions on improving the presentation of the paper.

REFERENCES

[1] [n.d.]. https://dba.stackexchange.com/questions/102903/is-it-acceptable-to-have-circular-foreign-key-references-how-to-avoid-them.

[2] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. Join synopses for approximate query answering. In *Proc. ACM SIGMOD International Conference on Management of Data*.

[3] 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.

[4] 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.

[5] 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.

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

[7] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. 1983. On the Desirability of Acyclic Database Schemes. *J. ACM* 30, 3 (1983), 479–513.

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

[9] 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.

[10] 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.

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

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press.

[13] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. 1993. Maintaining views incrementally. *ACM SIGMOD Record* 22, 2, 157–166.

[14] 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*.

[15] Ahmet Kara, Hung Q Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2019. Counting Triangles under Updates in Worst-Case Optimal Time. In *Proc. International Conference on Database Theory*.

[16] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2019. Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries. *arXiv preprint arXiv:1907.01988* (2019).

[17] Christoph Koch. 2010. Incremental query evaluation in a ring of databases. In *Proc. ACM SIGMOD International Conference on Management of Data*. ACM, 87–98.

[18] 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.

[19] 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.

[20] 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.

[21] Abraham Silberschatz, Henry F Korth, Shashank Sudarshan, et al. 1997. *Database system concepts*. Vol. 4. McGraw-Hill New York.

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

[23] Ke Yi, Hai Yu, Jun Yang, Gangqiang Xia, and Yuguo Chen. 2003. Efficient Maintenance of Materialized Top-k Views. In *Proc. IEEE International Conference on Data Engineering*.