# Keep Your Distributed Data Warehouse Consistent at a Minimal Cost

ZHICHEN XU, Google LLC, USA
YING GAO, Google LLC, USA
ANDREW DAVIDSON, Google LLC, USA

Large data warehouses store interdependent tables that are updated independently in response to business logic changes or late arrival of critical data. To keep the warehouse consistent, changes to upstream tables need to be propagated to downstream tables in a timely fashion. However, a naive change propagation algorithm can cause many unnecessary updates or recalculations of downstream tables, which drives up the cost of data warehouse management.

In this paper, we describe our solution that can ensure the eventual consistency of the data warehouse while avoiding unnecessary table updates. We also show that the optimal trade-off between computational cost reduction and meeting data freshness constraints can be found by solving a dynamic programming problem.

The proposed solution is currently in production to manage the YouTube Data Warehouse and has reduced update requests by 25% by eliminating non-trivial duplicates. These requests would have been carried out by large batch jobs over big data. Eliminating them has led to a proportionate reduction in computing resources.

One key advantage of our approach is that it can be used in a heterogeneous, distributed data warehouse environment where the operator software may not have complete control over the query processors. This is because our approach only relies on having dependency information for tables and can operate on the post-state of data sources.

CCS Concepts: • **Information systems → Data management systems**.

Additional Key Words and Phrases: Data warehouse, consistency management

**ACM Reference Format:**
Zhichen Xu, Ying Gao, and Andrew Davidson. 2023. Keep Your Distributed Data Warehouse Consistent at a Minimal Cost. *Proc. ACM Manag. Data* 1, 2, Article 190 (June 2023), 25 pages. https://doi.org/10.1145/3589770

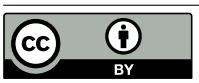
## 1 INTRODUCTION

Large-scale data warehouses contain many interdependent tables. The base tables are maintained by data engineering teams who extract data from source systems, transform it, and load it into the data warehouse. Downstream tables can be owned by different product areas to store data that is specific to their area. These are materialized views defined over the base tables by applying business logic, expressed as programs written in SQL or other programming paradigms for big data processing [6]. These programs, referred to as jobs, form a large dependency graph through data dependencies. The term "materialized view" is used broadly here to refer to any downstream data that can be derived from one or multiple upstream data sources.

Keeping the data warehouses consistent and fresh is crucial. Inconsistent or stale data can lead to suboptimal user experience, misinformed business decisions, and regulatory compliance risks. To ensure the data warehouse's consistency, changes in the upstream tables must be propagated



Authors' addresses: Zhichen Xu, Google LLC, Mountain View, California, USA; Ying Gao, Google LLC, Mountain View, California, USA; Andrew Davidson, Google LLC, Mountain View, California, USA.




2836-6573/2023/6-ART190
https://doi.org/10.1145/3589770

to downstream tables according to consistency policies and freshness requirements defined for the tables. When an upstream table is created or updated, the jobs that encode business logic for creating the initial version of the downstream views or refreshing them are scheduled to run.

Updates to multiple upstream tables may occur at any time in response to spam corrections, logic changes, or the late arrival of crucial data. These independent updates, together with the complex topology of the dependency graph, may result in unnecessary updates or recalculations of materialized views downstream when a naïve approach of immediately triggering updates to downstream tables is used. Unnecessary updates drive up the cost of warehouse management, as they can be enacted by large batch jobs over big data, consuming significant resources when run.

### 1.1 The Problem

To illustrate the problems and our solutions, we use Figure 1 as a running example. Nodes $A$, $B$, $C$, and $D$ represent jobs. Job $A$ creates or updates base table $a$. Jobs $B$ and $C$ depend on table $a$ and maintain materialized views $b$ and $c$, respectively. Job $D$ depends on both $b$ and $c$, and maintains materialized view $d$.

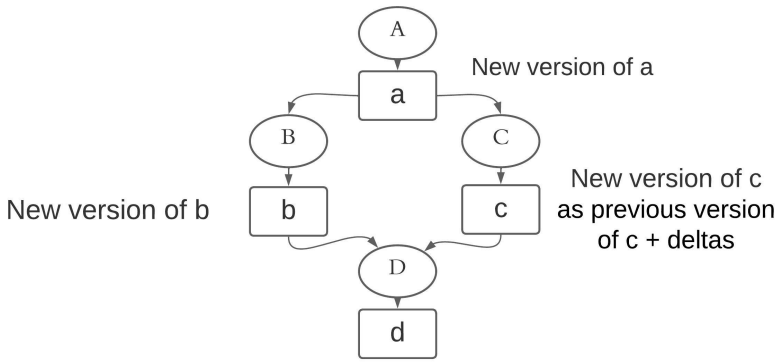



Fig. 1. When table $a$ is updated once, a naive algorithm will update materialized view $d$ twice.

Unneeded computations can arise from two sources:

(1) The shape of the dependency graph. Due to the diamond structure of the dependency graph in Figure 1, for each new version of table $a$, view $d$ will be recomputed twice if the update requests are scheduled *immediately*: once in response to a new version of $b$, and a second time for a new version of $c$. If job $A$ has $N$ direct descendants instead of two, table $d$ and all of its descendants will be updated $N$ times.
(2) Independent updates to multiple base tables. Similarly, when tables $b$ and $c$ are updated independently, $d$ could be updated twice. A more cost effective strategy is to have job $D$ wait for both changes to $b$ and $c$ and update the view $d$ once, if the freshness constraints can still be satisfied.

Updates to base tables and views can be made in a few different ways: by restating the previous version of a table with a newer version, by amending the previous version of a table with deltas (changed rows between two versions, e.g., table $c$ in Figure 1), or by a sequence of DML operations. Streaming can be regarded as a form of append-only amendments. Amendments are employed in data warehouses like Mesa [14].

It is becoming common practice for updates to be carried out by custom batch jobs owned by different teams that employ big data processing techniques [6]. Data warehouses also commonly support both relational and non-relational data models [21].

For the sake of brevity, we use the term "table" to refer to either a base table or a materialized view. We refer to the invocation of a job for refreshing a table as an "update request" or a "request."

### 1.2 Our Results

Our solution has the following properties:

(1) Any single change to an upstream table will be propagated to a downstream table at *most once*, regardless of the shape of the dependency graph. For example, in Figure 1, each update to table $a$ will be propagated to views $b$, $c$, and $d$ at most once.
(2) Independent updates to upstream tables will be combined as a single update to their common descendants, provided that the data warehouse management system is aware of these requests and can satisfy the freshness constraints defined on the tables.

For independent updates, we provide solutions for two scenarios:

(1) We show that minimizing computation cost while meeting data freshness requirements can be formulated as a dynamic programming problem.
(2) When the freshness of data is not a concern, global knowledge, such as the cadence of updates, can be used to increase the rate of deduplication. This global knowledge can also be used to make an optimal trade-off in the above scenario.

Our solution works as long as changes can be tracked by version numbers. It has been deployed in production for the YouTube Data Warehouse and is eliminating approximately 25% of updates as "non-trivial duplicates", which could not have been detected with exact matches. The data warehouse allows YouTube to store and analyze massive amounts of data, trillions of new data items per day, based on billions of videos, billions of views, and billions of watch time hours [7]. These eliminated updates would have been processed by large batch jobs over very big data. Eliminating them amounts to savings in computing resources at the same or higher proportion. For each duplicate, we avoid updating the subtree rooted at the job. More details about realized gains are discussed in Section 7.1.

We also provide a quantitative analysis of cost savings through synthetic workloads in Section 7.2 to remove possible bias.

### 1.3 Our Contributions

(1) We identify three types of consistency for ensuring whole-warehouse consistency: vertical, horizontal, and replication consistency: vertical and horizontal consistency for reasoning about the consistency among table-partitions on the dependency graph, and replication consistency for measuring consistency among replicas (Section 2.3).
(2) We introduce *Invariants* on versions that must be enforced so that the data warehouse will be eventually consistent. These invariants can be guaranteed if update requests are *strict monotonic functions* with respect to versions of their inputs and outputs (Section 3.7).
(3) We introduce a lattice model (Section 3.4) over the update request space, and a `Merge` operation for combining compatible requests. The merge operation sets input version requirements of the combined request so that the combined request will be queued until inputs meeting the version requirements become available. It also sets the version on the outputs to enforce the invariants for eventual consistency.
(4) When combining requests, there is a trade-off between the request queuing time (which impacts freshness of the tables) and unneeded computation elimination (which affects the cost).

We introduced a formulation that can make optimal trade-off between cost saving and meeting data freshness constraints. We show our formulation is solvable via dynamic programming (Section 4.2). When delaying backfills is acceptable (true for many applications in practice), we show how global knowledge can be leveraged to minimize unneeded computations (Section 4.3).

(5) The `Merge` operation needs to account for the relative positions of the originating requests on the dependency graph. We introduce a compact representation of the topology of the dependency graph, and an O(1) time and space algorithm to maintain the states in requests (Section 6.1).

### 1.4 Comparison with Related Work

We are motivated by the need to minimize unnecessary updates for enforcing whole-warehouse consistency (Section 2.3) in large-scale distributed data warehouses. Tables are typically stored in a columnar format [15], which makes them efficient to query but expensive to update [1]. Data warehouse systems need query processing software that supports both relational and non-relational data models [21], and support data parallel processing of very large, structured and unstructured enterprise data [6]. The internal logic of query processing can be opaque to the operator software.

Materialized view maintenance in database systems has been studied extensively [2, 5, 9, 13, 16–18, 22, 24, 28]. Most algorithms in database literature perform lazy updates-combining based on some variations of relational algebra and pre-state of the base tables. They require that the data warehouse know the exact query logic since they ensure consistency by evaluating the effects of updates. These algorithms are difficult to be extended to non-relational model and hard to apply across views.

Data pipeline systems are often used to orchestrate distributed data warehouses. Systems like Cloud Dataflow [12] and Apache Airflow [26] are blind to data updates and usually schedule their jobs periodically. Data pipeline frameworks, such as Snowpipe [23] and AWS code pipeline [3] can trigger a new run when new data arrives, which is similar to our *lazy triggering* policy (Section 3.3), but they do not provide solution for eliminating unnecessary runs.

Several projects use versions to make computation incremental. Differential dataflow [19] is introduced for data parallel analysis tasks, where the output of an operation on a new version of an input can be computed from the previous version of the output and the difference between the two input versions. Timely dataflow [20] is a computational model on directed graphs with stateful vertices that send and receive time stamped messages. Neither of them use versions to reason about the mergeability of competing updates.

We use versioning to track changes (Section 3.1), a lattice model to determine update compatibility (Section 3.4), and relies on strict monotonicity of updates (defined over versioning) for enforcing eventual consistency. This allows our technique to operate as a framework for updates deduplication, and permits various representations of updates (restatements, or amendments as deltas or a sequence of DML operations). It can be used in conjunction with traditional view maintenance methods, or supports incremental processing (Section 6.2). It is easy to integrate into a heterogeneous, distributed data warehouse where the operator software might not have full control over the internals of query processors.

To the best of our knowledge, no prior art has used a lattice model for update compatibility, and has relied on strict monotonicity of updates to ensure eventual consistency of data warehouses.

Minimizing unneeded updates can be achieved by identifying and combining compatible updates, and by manipulating request queuing time to maximize the pool of compatible updates without compromising data freshness. We ensure theoretical optimality for the former by incorporating as much new information in each single update as possible. This is done by identifying all compatible

updates that are observable by the system. For the latter, we describe strategies to find the best trade-off between cost reduction and data freshness guarantees. We also leverage global knowledge to increase the likelihood of having more compatible updates (Section 4).

Although this paper does not focus on applying the framework at the intra-table level, we offer an alternative to traditional view maintenance algorithms when the internal logic of updates can be represented as dependency graphs [19, 20].

The paper is organized as follows. Section 2 provides background information. Section 3 describes how compatible requests can be combined while ensuring eventual consistency of the data warehouse. Section 4 describe strategies to minimize unneeded computation. Section 5 discusses how versioning supports replication consistency. Section 6 discusses optimizations and error handling, section 7 presents realized gains for the YouTube Data Warehouse, and experimental studies. Comparison with additional related work is given in Section 8.

## 2 BACKGROUND

This section provides background information. The main symbols used in this paper and their descriptions are listed in Table 1.

Table 1. Main symbols

| Symbols | Descriptions |
|---|---|
| $A, B, C, D, J, R, G$ | Jobs. |
| $a, b, c, d, r, g, g_1, r_1$ | Tables or table partitions. |
| $a[i]$ | Partition $i$ of table $a$. |
| $a[i](V)$ | Version $V$ of partition $i$ of table $a$. |
| $a(V)$ | Version $V$ of table partition $a$. |
| $R[input_i \geq V_i] \times O \longrightarrow V_j$ | An update request to run job $R$. $R$ has inputs: $input_i$, backfill has origin $O$, output should have version $V_j$. |
| $R[input_i \geq V_i]$, $R[...]$, $A, B$ | Abbreviated form of a request. |
| $CD$ | Covered datasets. |
| $\leq$ | Partial order for requests. |
| $Merge(A, B)$ | Merge operation, $A$ and $B$ are requests. |

### 2.1 Data Model

Data warehouse tables are typically partitioned to improve performance. For example, a table that stores user activity could be partitioned into partitions of minute, hour, day, month, or other granularities, depending on the application. Each partition stores user activity for a given time period. A table can also have a singleton partition, e.g., a dimension table for storing video metadata.

Each partition of a table is versioned independently. New versions of a table-partition are produced by jobs that are scheduled in response to updates to its inputs, or changes to the business logic. A new version can be a complete restatement, or amendments to the previous version [14]. The dependencies are defined over table-partitions.

### 2.2 Rich Dependency Types

For the YouTube Data Warehouse, dependencies between upstream base tables and downstream materialized views are captured explicitly through configurations. We allow arbitrary dependency types through an algebra. Static analysis checks the validity of the dependency at configuration

creation and modification time. If the dependency graph is not explicitly captured, techniques described in [8] could be used to reverse engineer it.

Table 2 illustrates how the algebra can be used to express dependencies via partition selection.

Table 2. Dependency algebra and examples.

| Example | Descriptions |
|---|---|
| Job(“moving_average_calculator”, partition = DAY,<br>deps = [table(“day_stats”, partition = DAY-1),<br>table(“day_stats”, partition = DAY-2),<br>table(“day_stats”, partition = DAY-3)],<br>outputs = ["three_day_moving_average"]) | Range dependency. |
| Job("accounting", partition = WEEK,<br>deps = [table("daily_spending", partition = WEEK.days())],<br>outputs = ["weekly_spending"]) | Weekly job rolls up a daily upstream. |

The selection of partitions is based on abstract partition variables with time arithmetic. For example, for the second row in Table 2, the upstream is table “day_stats” with day partition. "DAY" is a partition variable. Job “moving_average_calculator” describes updates to a downstream view "three_day_moving_average". Each “day” partition of “three_day_moving_average” is computed from 3 partitions of “day_stats” for the past 3 days. {YYYYMMDD} partition of “three_day_moving_average” depends on the {YYYYMMDD-1}, {YYYYMMDD-2} and {YYYYMMDD-3} partitions of “day_stats”. Similarly, for row 3, "WEEK" is a partition variable, and "WEEK.days()" returns the abstract DAYs in the WEEK. Job "accounting" computes "weekly_spending" from "daily_spending" for the days in each week.

We show that the proposed approach is applicable to arbitrary dependency types. A full description of the algebra is beyond the scope of this paper.

### 2.3 Three Categories of Consistency

In a streaming data warehouse, updates to table-partitions can arrive incrementally, and sections of the warehouse may be updated with fresher data for insights and business analytics. These updates must be consistent *vertically*, *horizontally* and *across replicas* in order to be usable within the warehouse.

*Definition 2.1.* We define a version of a table-partition to be *vertically consistent* with an upstream table if it is derived from the same particular version as the upstream table. A downstream table-partition is vertically consistent if all changes to its ancestors (on the dependency graph) have been propagated to this table partition according to the policy defined for the downstream tables.

This policy gives table owners and data warehouse custodians control over the different types of changes that should be propagated. Vertical consistency can be attached to updates. For example, if a materialized view is created for performing a one-time analysis, there is no need for the table to reflect any new changes in the upstreams. However, if an upstream is updated for regulatory compliance, then the updates need to be propagated within a service-level objective (SLO) [31]. This paper focuses on updates requiring vertical consistency.

*Definition 2.2.* We define two table-partitions to have *horizontal consistency* if they are derived from the same versions of common upstreams. Horizontal consistency measures the consistency among siblings on a dependency graph.

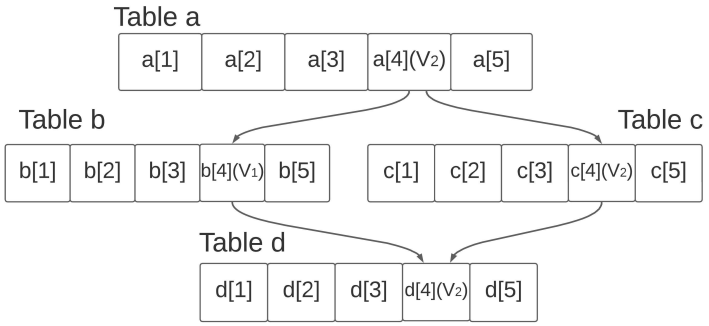



Fig. 2. Vertical and horizontal Consistency.

It is important to make sure that artifacts derived from the same sources are consistent with each other. Imagine a store provides multiple reports based on the same base table. The base table stores revenue from each item. One report is revenue made by each department, and another report is revenue for each category of goods. The two reports should be consistent with each other, e.g., the total revenue should match.

Vertical consistency and horizontal consistency are similar to the source consistency and multi-view consistency discussed in [33] in that consistency is defined in terms of whether a view has the same content of a base table with regard to a series of state transitions.

*Definition 2.3.* We define *replication consistency* as the consistency in the traditional sense of a distributed system, where new versions of a table partition should be propagated to all replicas.

Figure 2 shows examples of vertical and horizontal consistency. Tables $a$, $b$, $c$, and $d$ are partitioned (5 partitions are shown in the Figure). Let us denote partition $i$ of table $a$ as $a[i]$, and version $v$ of $a[i]$ as $a[i](v)$. The table partition $a[4]$ has two versions $V_1$ and $V_2$. $c[4](V_2)$ was computed from $a[4](V_2)$.

- $c[4](V_2)$ is consistent with $a[4](V_2)$.
- If $V_2$ is the latest version of a[4], then c[4] is vertically consistent.
- Because $b[4](V_1)$ and $c[4](V_2)$ are both derived from $a[4](V_2)$, they are horizontally consistent with each other.

*Table* and *table-partition* will be used interchangeably unless otherwise specified.

### 2.4 Frontfill, Backfill, and Dependent Backfill

The term *data backfilling* refers to the processes that make corrections to historical data [10]. Backfills are necessary in data warehouses for multiple reasons: Some business-critical data might arrive late in the Origin (the log source that the basetables derive from), and some jobs might have introduced a change in the business logic.

We refer to a job invocation creating the first version of a table partition as a *frontfill*, and the actions for refreshing subsequent versions, *backfills*. When an upstream table has a new version, this triggers the updates to all descendant tables in the subgraph that depend on this upstream. We call this a *dependent backfill*.

## 3 MERGING COMPATIBLE REQUESTS AND EVENTUAL CONSISTENCY

Our solution includes the following components:

- *Invariants* defined on versions to ensure the eventual consistency of the data warehouse.
- A *lattice model* over the request space for identifying compatible requests that are mergeable.
- A `Merge` operation for combining compatible requests.
- An update propagation algorithm for creating downstream update requests in response to upstream changes.
- Strategies for minimizing unneeded recomputation (section 4).

### 3.1 Invariants on Versions for Consistency

Our solution ensures versions are unique and that higher versions imply fresher content. To enforce this, no higher version of a descendant table can be computed from an ancestor table of a lesser version. More precisely,

Let table $a$ be an ancestor of table $d$ on the dependency graph.
Let $d(V_{d_i})$ be computed from $a(V_{a_i})$.
Let $d(V_{d_j})$ be computed from $a(V_{a_j})$.
If $V_{d_j} > V_{d_i}$ then $V_{a_j} \geq V_{a_i}$ must hold.

These properties are the foundation for all three categories of consistency. We will show that enforcing the above properties on versions amounts to ensuring the monotonicity of each update request with respect to versions of its inputs and outputs (Section 3.6).

### 3.2 Anatomy of a Request

We can assume that a request originates from a dependent backfill (Section 2.4) to an ancestor table on the dependency graph.

*Definition 3.1.* We define the *origin* as the job that is at the top of the subgraph of a dependent backfill. The origin, together with the dependency graph, determines the subgraph to be backfilled.

*Definition 3.2.* We define *covered datasets* as the datasets that should be updated when an update is carried out on a dependency graph.

Using Figure 1 as an example, an update request for job $B$ covers $b$ and $d$. The covered datasets will be set to $b$ and $d$ in the request. Assuming running this update produces a new version of $b$, $b(v2)$, $b(v2)$ will trigger a request to run job $D$. $B$ will be the origin of this new request for $D$. Job $D$ has two inputs $b$ and $c$. Input $b$ is covered and input $c$ is not covered, so the request to run job $D$ requires input $b$ to have a version that is greater or equal to $v2$, but it has no version requirement for $c$.

*Definition 3.3.* We define *triggering input* as the input that has a new version and causes the job invocation to be scheduled for a given job in the subgraph.

Continue with the previous example, the triggering input for the request to execute job $D$ is $b(v2)$.

Given the above definitions, we use a function to denote a request:

$$R[input_i \geq V_i, for\ i = 1, 2, ..., n] \times O \longrightarrow V_j$$

Which states that the job $R$ takes $n$ table partitions {$input_i$, i=1, 2, ..., n} as inputs, and version for $input_i$ must be greater or equal to version $V_i$, and the origin of the request is job $O$ on the dependency graph. The $covered\ datasets$ should be assigned version $V_j$. The *covered datasets* can be computed from the origin $O$ and the dependency graph. For conciseness, we will omit $\{for\ i = 1, 2, ..., n\}$ in our notation for request.

### 3.3 Propagate Updates

The pseudo code in Table 3 shows how to create update requests in response to a new version of an upstream table partition.

Table 3. Triggering (direct) downstream jobs in pseudocode.

| |
|---|
| Let $r = (R[...] \times O \longrightarrow V)$ be the original update request. |
| Let $CD$ be the covered datasets, and $d(V)$ be the triggering input. |
| For each job $J$ that (directly) depends on $d$, |
| Create: $J[input_i > V, input_j] \times O \longrightarrow V$, where: $Input_i \in CD$ |

The system supports two update strategies.

*Definition 3.4. Lazy Triggering*: For each update, the system will create update requests for its direct downstreams. Jobs indirectly depending on the original update will listen to the update of its direct upstream and eventually be updated.

*Definition 3.5. Eager Triggering*: For each update, the system will create update requests for all downstreams. The triggering input and the origin will be the same node.

### 3.4 A Lattice Model on Requests

When the system receives a new request, it is compared with existing requests for the same job according to a partial order $\leq$.

*Definition 3.6.* The partial order $\leq$ has the following property:

- Let $A$ and $B$ be two requests, if $A \leq B$, then inputs that can satisfy the "input version requirements" of $B$ will also satisfy the "input version requirements" of $A$.

Using Figure 1 as an example, imagine that there are two independent updates to tables $b$ and $c$, which created $b(V_i)$ and $c(V_j)$, respectively. These, in turn, result two update requests to $d$:

- $r_1 = (D[b \geq V_i, c] \times B \longrightarrow V_i)$ and
- $r_2 = (D[b, c \geq V_j] \times C \longrightarrow V_j)$

Assume the owner of table $d$, issued a third request:

- $r_3 = (D[b \geq V_i, c \geq V_j] \times D \longrightarrow V_k)$

It is clear that $r_1 \leq r_3$ and $r_2 \leq r_3$, since inputs that can satisfy $r_3$ can satisfy both $r_1$ and $r_2$.

If a new request $A$ arrives, and there is already an existing request $B$ that satisfies $A \leq B$, the system simply discards request $A$ because $A$ does not bring any new information. Otherwise, the system merges $A$ with existing compatible requests via a `Merge` operation.

### 3.5 Merge Operation

Let two update requests be

$$r_1 = (R[input_i \geq V_{i1}] \times O_1 \longrightarrow V_1) \tag{1}$$

$$r_2 = (R[input_i \geq V_{i2}] \times O_2 \longrightarrow V_2) \tag{2}$$

*Definition 3.7.*

$$\texttt{Merge}(r_1, r_2) = (R[input_i \geq max((V_{i1}, V_{i2})] \times O \longrightarrow max(V_1, V_2) + \delta), \delta >= 0$$

$O$ is set to the uppermost common descendent of $O_1$ and $O_2$. The reason for picking the uppermost common descendent will be explained further in Section 3.8. $\delta$ is set to a positive constant if any update request has been scheduled, and is set to 0 otherwise.

The term, $max(V_i, V_j) + \delta$, ensures that the output version of a new request is always greater than the output versions of the previous successfully executed requests for the same table. This is essential for enforcing eventual consistency. It will be proven in Section 3.6.

In practice, we can use the timestamp at which a dependent backfill is triggered as the version for the covered datasets.

When merging is successful, for any actual inputs that can satisfy the version requirements for $Merge(A, B)$, they will also satisfy those for both $A$ and $B$. That is, $A \leq \texttt{Merge}(A, B)$ and $B \leq \texttt{Merge}(A, B)$.

### 3.6 Proof of Eventual Consistency

Our approach enforces a total order over the versions of any given table partition with the following properties:

(1) **Property 1**: All versions are unique for any given table partition, and increase monotonically.
(2) **Property 2**: For two versions of a table partition $a$, $a(V_i)$ and $a(V_j)$: if $V_j > V_i$, then any input $x$ (direct or indirect upstream on the dependency graph) used to produce $a(V_j)$ has a version greater or equal to the version of $x$ that was used to produce of $a(V_i)$.

We show that as long as the following invariants can be enforced, *property 1* and *property 2* will hold.

(1) **Invariant 1**: At the origin of any dependent backfill, the output versions are unique and monotonically increasing.
(2) **Invariant 2**: Let $r_1$ and $r_2$ be as defined as in (1)(2). If $R[input_i \geq V_{i1}] \succ R[input_i \geq V_{i2}]$, then $V_1 > V_2$.

Invariant 1 can be enforced by ensuring that the output version of a new update request is greater than the output versions of the previous successfully executed requests for the same table.

Invariant 2 basically requires function $(R[input_i \geq V_{i1}] \times O \longrightarrow V)$ be a *strict monotonic function* [30] with respect to input $R[input_i \geq V_{i1}]$ and output $V$. Property 2 holds because composition of strict monotonic functions is still a strict monotonic function.

The `Merge` operation satisfies invariant 2 because the output version is set to $max(V_1, V_2) + \delta$.

In Section 3.7, we show how to ensure a single change to an upstream table is propagated to a downstream table "at most once", regardless of the shape of the dependency graph. In Section 3.8 and Section 3.9, we will show how independent updates to tables can be combined as a single update to their common descendants.

### 3.7 Single Dependent Backfill

This section shows how the merge operation can be used to prevent single updates from being propagated to a descendent table more than once.

Using the dependency graph in Figure 1 as an example, we assume that job $A$ is triggered at time $t$, producing a new version $a(V_t)$. This makes job $A$ the origin. For jobs $B$ and $C$, table $a(V_t)$ is the triggering input. Nodes $a$, $b$, $c$, and $d$ are covered by the subgraph originating from job $A$. The steps are shown in Table 4.

- For jobs $B$ and $C$, upon seeing the triggering input $a(V_t)$, requests $B$ and $C$ (as illustrated in the second row of Table 4) will be produced. We have omitted the origin $A$ and output version $V_t$.

Table 4. Merge duplicated requests. Illustration of how only one request for job $D$ is produced for the example in Figure 1.

| Tables a | b | c | d | Requests | Covered datasets |
|---|---|---|---|---|---|
| $V_t$ | | | | $B[\mathbf{a} \geq V_t], C[\mathbf{a} \geq V_t]$ | $a, b, c, d$ |
| | $V_t$ | | | $D[\mathbf{b} \geq V_t, c \geq V_t]$ | |
| | | $V_t$ | | $D[b \geq V_t, \mathbf{c} \geq V_t]$ | |

- When job $B$ finishes, it produces a new version of $b$ with the version $V_t$ (since $b$ is in the covered datasets and the output version is $V_t$). The system produces the request $D[\mathbf{b} \geq V_t, c \geq V_t]$ in response to the new version of $b$. Table partitions $b$ and $c$ have version constraints $V_t$ because both of them are in the covered datasets. We bold $b$, the triggering input.
- Similarly, when job $C$ finishes, $D[b \geq V_t, \mathbf{c} \geq V_t]$ will be created in response to the new version of $c$, which is an exact duplicate. The system will drop the exact duplicate request.

Even though $D$ is triggered twice—once due to a new version of $b$ and again for a new version of $c$—only one request for $D$ is produced.

### 3.8 Multiple Dependent Backfills from Independent Origins

Figure 3 shows two independent backfills, originating at $G$ and $R$, which are not ancestors of one another. The green cone, delineated by dashed lines, and the red cone indicate their respective subgraphs. The gray area indicates the intersection of the two subgraphs. Dotted lines represent ancestor-descendant relationships, while solid arrows represent input-to-job relationships.

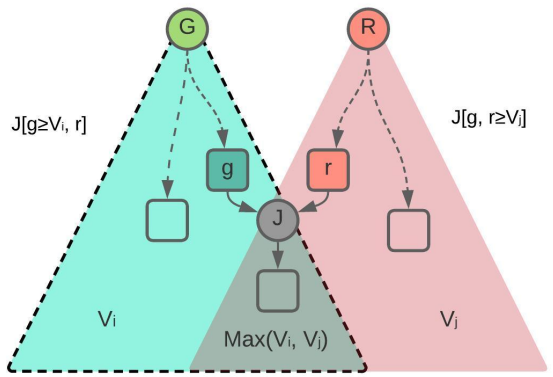



Fig. 3. Competing backfills from independent origins.

Without loss of generality, we assume that the two subgraphs intersect at job $J$, which has input $g$ from the green subgraph and input $r$ from the red subgraph. We also assume that the green origin has version $V_i$ and the red origin has version $V_j$.

The scheduler creates requests on-demand in lazy scheduling.

- In response to the new version of input $g$, the scheduler will create the request $J[g \geq V_i, r] \times G \longrightarrow V_i$.

- In response to the new version of input $r$, the scheduler will create the request $J[g, r \geq V_j] \times R \longrightarrow V_j$.

The two requests are `Merged` as

$$J[g \geq V_i, r \geq V_j] \times J \longrightarrow max(V_i, V_j) + \delta.$$

As a result, the new origin is $J$, and $J$'s descendants have version $max(V_i, V_j) + \delta$. $J$ is the uppermost common descendant of the two competing backfills.

Intuitively, descendants of $J$ have version $max(V_i, V_j) + \delta$ because they incorporate information from both $g(V_i)$ and $r(V_j)$.

*3.8.1 Monotonicity of Merge Operation.* Let us show that *invariant 2* holds. Without loss of generality, let us assume $V_j > V_i$. When $(J[g, r \geq V_j] \times R \longrightarrow V_j)$ is scheduled, there are two possibilities:

(1) $(J[g \geq V_i, r] \times G \longrightarrow V_i)$ is waiting for input data. The two requests are `Merged`, and there is only a single version for $J$'s output with version $max(V_i, V_j) + \delta$. *Invariant 2* holds and $\delta$ is set to zero.

(2) $(J[g \geq V_i, r] \times G \longrightarrow V_i)$ has been scheduled with input $r(V_k)$. $J$'s outputs have two versions:

$$V_i : computed\ from : g(V_i), r(V_k)$$
$$max(V_i, V_j) + \delta : computed\ from : g(V_i), r(V_j)$$

For case 2, $\delta$ is set to a constant positive number. We know that $V_k < V_j$ must be true, otherwise, the request to produce $r(V_j)$ would have been discarded by our algorithm. Moreover, $max(V_i, V_j) + \delta > V_i$, regardless of which is larger.

### 3.9 Multiple Dependent Backfills from Dependent Origins

In Figure 4, the red origin $R$ is a descendant of the green origin $G$ on the dependency graph.

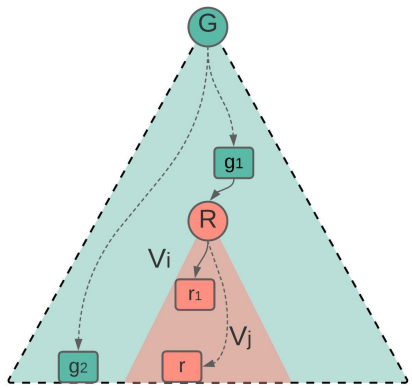



Fig. 4. Competing backfills from dependent origins.

Let $V_i$ and $V_j$ be the times when $G$ and $R$ are triggered for a dependent backfill, respectively, and assume that $V_j > V_i$. Let $g_1$ be an input of $R$. Without loss of generality, assume that $R$ is triggered by $g_1(V_i)$, as a result of the dependent backfill from the green origin at $V_i$. In this scenario, the request is $(R[g_1 > V_i] \times G \longrightarrow V_i)$.

Similarly, when the red origin is backfilled with $R$ as origin at $V_j$, the request is $(R[g_1] \times R \longrightarrow V_j)$.

There are three cases:

(1) Both requests are pending. They will be `Merged` into $R[g_1 \geq V_i] \times R \longrightarrow max(V_i, V_j]) + \delta$. As before, the new origin is set to the uppermost common descendant of the two origins. $\delta$ is set to zero and the monotonicity of the combined request holds.
(2) The red origin is triggered after its green input $g_1$ is ready. In this case, $R[g_1 \geq V_i]$ can already satisfy the requirements of $R[g_1]$; the second request can be dropped.
(3) The red origin is triggered before its input $g_1(V_i)$ is ready.

For case 3, $R$ has two invocations, and we assume that $r_1$ is one of the output tables of $R$ (See Table 5).

Table 5. The red origin is triggered before its input $g_1(V_i)$ is ready.

| Invocations | Input version | Output version |
|---|---|---|
| $(R[g_1 \geq V_k] \times R \longrightarrow V_j)$ | $V_k$ | $V_j$ |
| $(R[g_1 \geq V_i] \times G \longrightarrow V_i)$ | $V_i$ | $max(V_i, V_j) + \delta$ |

*3.9.1 Monotonicity of Merge Operation.* It is clear to see that $V_k < V_i$ by induction, and $V_j < max(V_i < V_j) + \delta$.

## 4 MINIMIZING UNNEEDED COMPUTATIONS

There is a trade-off between how long to keep a request in a queued state so that it can be merged with compatible requests in the system or those that will arrive in the future. We motivate this trade-off through a simulation study that shows how queuing times can affect the rate of requests deduplication.

For applications where meeting freshness constraints is important, we describe a principled strategy to optimize the trade-off between cost reduction and meeting freshness constraints.

For many practical applications, there are no temporal constraints on the propagation of dependent backfills to downstream tables, provided that eventual consistency is satisfied. We describe how to minimize unneeded computation by leveraging global knowledge about the time-based cadence of the backfills or active requests in the system. Such global knowledge can also be incorporated for making optimal trade-offs between cost reduction and meeting freshness requirements.

### 4.1 Queuing Time versus Deduplication Rate

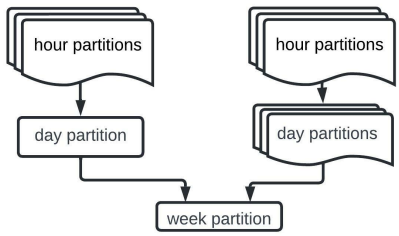



Fig. 5. Roll ups

To illustrate the trade-off between cost reduction and meeting freshness requirements, we constructed a dependency graph depicted in Figure 5. This topology illustrates cases when a downstream table is an aggregate of some upstream tables.

We experimented with a range backfill of 24 partitions of the hourly table that span 1 day. We assume that the runtime of the job for the hourly table follows a normal distribution. The mean is set to 300 seconds, and standard deviation is set to 100 seconds. We vary the request queuing time for the downstream job for the daily table from 0 to 3 standard deviations. The results are summarized in Table 6.

Table 6. Range backfills of roll up topology. Optimal number of updates to the downstream daily table is 1.

| Queuing time (# stdv) | Downstream reqs (# fresh inputs in each update) | % uneeded |
|---|---|---|
| no waiting | 24 | 96% |
| 1 stdv. | 4 (6, 15, 21, 24) | 75% |
| 2 stdv. | 2 (15, 24) | 50% |
| 3 stdv. | 2 (21, 24) | 50% |

When the queuing time is 0, the downstream daily table is updated 24 times, and each update incorporates one additional fresh input. When the queuing time is 1 standard deviation, the downstream daily table is only updated 4 times, and 6, 15, 21, and 24 fresh inputs are incorporated, respectively. When the queuing time is 2 standard deviations, the downstream job is only invoked twice, and the first request can already incorporate 15 fresh inputs. Likewise when the queuing time is 3 standard deviations, and the first request can already include 21 fresh inputs. We ran the simulation multiple times and observed the same behavior each time.

This shows that we can choose proper request queuing time based on historical runtimes and distribution of upstream backfills.

### 4.2 Trade-off between Cost and Freshness

This section describes a strategy to find the optimal balance between cost reduction and meeting freshness constraints.

For the sake of simplicity, let us assume the following scenario:

- Let $r$ be an active request in the system, and assume $r$ is a request to invoke job $R$, and is estimated to start at $T(r)$.
- Let $g$ be an active request to invoke job $G$, and $G$ is an ancestor of $R$ on the dependency graph.

Executing $g$ will eventually trigger a request $r'$ to run job $R$, and is estimated to start at $T(r')$. We need to decide whether it is beneficial to delay $r$ by waiting for the changes due to executing $g$.

If we merge $r$ and $r'$, the new start time of $r$ will be $T(r')$, and all descendants of $r$ will have an added delay of ($T(r')$ - $T(r)$) approximately. We use the start delay to estimate the finish delay, which simplifies the discussion.

- The reward is the avoidance of recomputing the entire sub-graph rooted at the $R$.
- The penalty is the total penalty of delaying $R$ and its descendants.

We use $cost(R)$ to represent the cost to run job $R$, and $penalty(R, \Delta_t)$ to represent the penalty of delaying job $R$ by time interval $\Delta_t$.

An optimal strategy to trade-off cost and freshness guarantees can be solved with the formulation below.

(1) Let $Subgraph(R)$ represent $R$ plus all its descendants, and $Children(R)$ represent direct downstream jobs of $R$. Let $t$ be $T(r') - T(r)$. We can compare the rewards and penalties of two cases:
   (a) Delaying $r$: the reward will be the cost of computing the sub graph rooted at $R$, and the penalty is the sum of penalties delaying $R$ and its descendants.
      - $reward_{merge}(R,t) = \Sigma_{S \in Subgraph(R)} cost(S)$, and
      - $penalty_{merge}(R,t) = \Sigma_{S \in Subgraph(R)} penalty(S,t)$.
   (b) Not delaying $r$: we make independent optimal decisions for each direct downstream $C$ of $R$, and have $reward_{opt}(C,t)$ and $penalty_{opt}(C,t)$, respectively. Total reward and penalty not waiting for $r'$ are:
      - $reward_{nomerge}(R,t) = \Sigma_{C \in Children(R)} reward_{opt}(C,t)$, and
      - $penalty_{nomerge}(R,t) = \Sigma_{C \in Children(R)} penalty_{opt}(C,t)$.

We can make a decision by choosing the option which will provide the largest reward and penalty ratio. This can be solved efficiently through dynamic programming, and the complexity of the algorithm is $O(|Subgraph(R)|)$.

When the goal is to ensure updates are propagated within freshness constraints, a simple formation is to set the penalty to a very large number when an update is projected to miss freshness agreement with a high probability and to zero when the update is propagated on time.

### 4.3 Leverage Global Knowledge

Often, it is a prudent practice to merge compatible requests when the opportunity arises.

*4.3.1 Knowledge about the cadence of the backfills.* Backfills of the base tables often follow a time-based schedule. Let us assume that $a$, $b$ are two base tables that are backfilled at 1 a.m. every Monday, and $c$ is a descendant of both $a$ and $b$ (similar to Figure 3). Ideally, we would like to have $c$ wait for the new versions from both $a$ and $b$ before being scheduled. With lazy scheduling, the requests triggered by the backfilling of $a$ and $b$ happen asynchronously. It is difficult to control the timing of the two duplicate requests. We can encode such global knowledge about nearby time-based schedules leveraging the covered datasets construct.

Figure 6 illustrates our approach with an example. In the figure, jobs $A$ and $B$ have a weekly backfill schedule. Every Monday, $A$ is backfilled at 1 a.m. and $B$ is backfilled at 2 a.m. Even with our duplication mechanism, job $C$ will be triggered twice unless the timing works in our favor.

Let us assume that $A$ is backfilled at $t_a$, and $B$ is backfilled at $t_b$. When we schedule $A$, we set the origin to $A$, and covered datasets to $a \geq t_a, b \geq t_b$. When we schedule $B$, we set the origin to $B$, and covered datasets to $a \geq t_a, b \geq t_b$. Regardless of whether $C$ is triggered by $a$ or $b$, it will have version constraints $C[a \geq t_a, b \geq t_b]$.

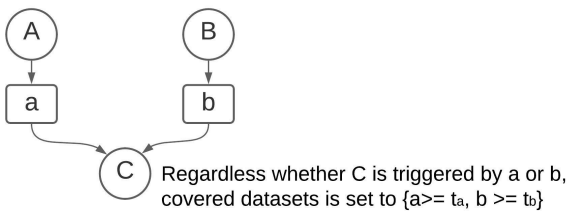



Fig. 6. Encode global knowledge about time-based schedules.

Table 7. Pros and cons of two solutions.

| Solutions | Pros | Cons |
|---|---|---|
| Walk tree backwards | Can handle a dynamic dependency graph. | Implementation complexity. |
| Eager requests generations | Simplicity. | Difficult to incorporate new changes to dependency once the requests are materialized. |

*4.3.2 Knowledge about the relative positions of current requests on the dependency graph.* We can improve the probability of compatible requests by examining the relative positions of known requests on the dependency graph. We can employ several techniques:

(1) Traverse the dependency graph backwards and search for requests received by the system that can cover any input of the current request. Using the same example, when $b$ has a new version $t_b$, the request $C[a, b \geq t_b]$ is created. Walking the dependency graph backwards by tracing input $a$, we can discover that the current backfill of $(A[...], A, t_a)$ covers the input $a$ of $C[a, b \geq t_b]$. As a result, we set its input version constraints to $C[a \geq t_a, b \geq t_b]$.
(2) Materialize requests for the entire subgraph eagerly for each dependent backfill. Eager rendering of the requests has its own caveats, which makes it hard to respond to dependency changes.

Table 7 summarizes the pros and cons of the two techniques.

## 5 REPLICATION CONSISTENCY

Versioning is essential for ensuring that all replicas are consistent and that the data warehouse can be queried as new versions of table partitions are created and replicated asynchronously. Because table partitions are versioned, it is easy to ensure that all replicas are consistent. Different versions of the same partition can be replicated separately, which can improve efficiency.

Below are several complications that can arise when replicating table partitions:

- Prioritization among different versions. Datasets can be associated with SLOs, and usually newer versions are more important than older versions. Under resource constraints, the system should give priority to newer versions over older versions.
- Multiple primary replicas. In some cases, it is desirable to have multiple instances of the same Job to run currently in different data centers for improved reliability or performance. This results in multiple primary replicas if we allow multiple instances of the job to finish and publish data. This can be addressed with a global replication plan, and race resolution uses the version numbers when the global plan can not be maintained due to network partition.
- Retention policy. For a data warehouse that supports retention policy, e.g., keep $K$ latest versions, or keep the data for $N$ days since creation. We can increase the overall efficiency by not replicating old versions out of retention.

## 6 OPTIMIZATIONS AND ERROR HANDLING

We discuss optimizations and error handling in this section.

### 6.1 Handling Large Dependency Graphs

The dependency graph can be large. A naive approach to store the covered datasets in each of the requests can take a lot of memory, and copying the requests can have long latency.


(a) R1 triggered.

(b) R2 is triggered.

Fig. 7. Build and maintain covered datasets in O(1) time and space.

We introduced an algorithm where we only store the covered datasets produced by the "frontier" of the dependency graph with the current request as one of the "leaves". The frontier consists of the outputs of the parent jobs. It is O(1) in practice since a job usually takes a small number of input tables. We extend the covered datasets by walking the dependency graph backwards.

Figure 7 illustrates this with an example. $O$ is the origin of the backfill, and $R_1$ and $R_2$ are two requests for two jobs on the dependency graph. The covered datasets we need to store for $R_1$ are $b$, $c$, $d$. When $R_2$ is triggered by dataset $c$, the origin is $O$ and the covered datasets are $b$, $c$, $d$. Dataset $e$ is added to the covered datasets because $e$ is the triggering input, and $b$ and $d$ are removed because they are no longer relevant to request $R_2$ and its descendants. Similarly, when $R_2$ is triggered by dataset $c$, the origin is $O$ and the covered datasets are again $b$, $c$, $d$. Our algorithm walks the dependency graph backwards, to discover $e$, and replace $b$, $d$.

As a result, each request only needs to store covered datasets for the part of the subgraph that is relevant to the current request. Updating the covered datasets takes O(1) time on average. The worst case is to traverse the depth of the dependency graph, which can be optimized using a standard caching mechanism.

### 6.2 Support Different Types of Updates

Updates can can take different forms as described in Section 1.1. We describe several additional optimizations.

*6.2.1 Incremental View Maintenance via Streaming.* When the business logic of a job is *associative*, we have the option to make the computation incremental.

When a job $R$ is invoked by input $a(v_j)$, and the previous version of $R$'s output $r(V_i)$ is produced from $a(v_i)$, where $(i < j)$.

$r(v_j)$ can be computed from

$$r(V_j) = Aggregate(r(V_i), R(a(V_j) - a(V_i)))$$

This strategy is beneficial for streaming data warehouses where $a(V_j) - a(V_i)$ is readily available or can be computed efficiently. For example, data warehouse systems like Mesa [14] store changed rows between two versions as delta $[V_1, V_2]$.

*6.2.2 Incremental View Maintenance with Knowledge about Query Logic.* Our approach can be used in conjunction with traditional methods as a framework for determining compatible updates across tables. In this setting, updates use version numbers to track changes and query logic to specify how the view should be updated. Traditional techniques, such as those mentioned in [2, 5, 22], can leverage query logic to avoid applying irrelevant updates to the views. We can also use techniques like Differential dataflow [19] to reduce update latency, or apply the updates combining algorithms proposed in this paper to the dependency graph representing update logic.

### 6.3 Error and Exception Recovery

Jobs can fail and the network can partition. The lattice model allows failed jobs to be subsumed by a new request with fresher inputs, besides retrying with the same input versions.

In practice, new requests can arrive while an existing compatible request is still in progress. We have introduced sophisticated "overrun policies" to allow users to choose the desirable behavior at table level. For example, if consistency is more important than latency, a newer request can be configured to preempt an existing request with "stale" inputs. In another case, the newer request can be skipped, especially when the job is resource intensive, and inconsistent data is acceptable.

The YouTube data warehouse management system is designed to be resilient to network partitions. This is achieved by persisting important state of requests and metadata about tables. The details of this process are out of the scope of this paper.

## 7 EXPERIMENTAL STUDY

We present the results of our production deployment of our solution in the YouTube Data Warehouse, as well as experimental studies utilizing synthetic workloads to eliminate bias from the production workload. Experimentation with the production workload is prohibitively expensive.

### 7.1 Gains for the YouTube Data Warehouse

The data warehouse allows YouTube to store and analyze large amounts of data, trillions of new data items per day, based on billions of videos, billions of views, and billions of watch time hours [7]. The sizes of the table partitions vary, as some need to handle trillions of data items received per day, whereas others need to store data for hundreds of millions of creators and fans. The dependency graph can be quite deep due to a diverse set of use cases to support.

Our solution is in production for the YouTube Data Warehouse and is eliminating 25% of duplicate updates that could not have been identified through trivial exact matches. This translates directly to a reduction in computing resources at the same or higher proportion.

Table 8 shows the distribution of duplicate requests detected by our algorithm. Approximately 26% of requests are detected as exact duplicates. About 25% of all requests are compatible requests that cannot be deduplicated via naive exact match. For the 25% non-exact duplicates, 17% are from merging queued requests, and 8% are from rejecting newer requests that can be subsumed by requests already processed.

Table 8. The distribution of duplicate requests detected by our algorithm.

| Exact duplicates | Not exact duplicates | |
|---|---|---|
| | Both pending | Satisfied by existing requests |
| 26% | 17% | 8% |

The distribution of relationships between the origins of completing dependent backfills is summarized in Table 9. The majority of the merged updates, 79%, are independent updates from different base tables, 19% from the same ancestor, and only 1.7% were due to the single dependent backfill (same origin). This demonstrates that our algorithm can handle dependency graphs with complex topologies and independent backfill patterns.

Table 9. The distribution of relationships between the origins of completing dependent backfills.

| Independent origins | Same ancestor | Same origin |
|---|---|---|
| 79% | 19% | 1.7% |

## 7.2 Synthetic Graphs and Update Patterns

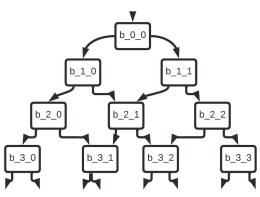
Incremental view maintenance algorithms from the literature are complementary to our techniques. They are difficult to compare directly against our approach, as most of them require knowing the internal update logic and are difficult to apply across views. Instead, we aim to provide evidence that our approach can achieve theoretical optimality by defining optimality as being able to identify and combine all compatible updates observable.

We constructed several common synthetic dependency graphs (see Figure 8) and ran experiments with synthetic workloads using the testing environment of the YouTube data warehouse system. Each job took approximately 20 seconds to 1 minute to run and was subject to the delays imposed by the actual testing environment.

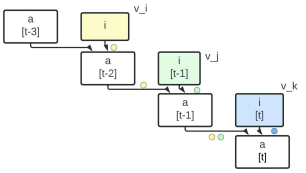
*7.2.1 Materialized Views over Binomial Tree Topology.* The binomial tree topology (Figure 8a) models typical materialized view dependency where a downstream table is computed by joining multiple upstream tables. The topology includes many diamond structures. With a naive approach, when $b_{0_0}$ is backfilled, two versions of $b_{2_1}$ can be produced which will cause 4 versions of $b_{4_2}$.

We constructed a binomial tree structure with 50 levels (1275 nodes), and randomly picked K nodes for dependent backfills. We compare three strategies:

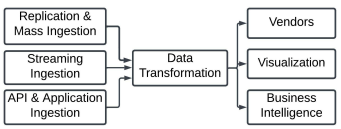
(1) Lazy scheduling, which sets the proper origin and builds the covered datasets for the frontier incrementally (techniques described in Section 6.1. We call this method "lazy" in subsequent discussions).
(2) Lazy scheduling with grouping, which also groups backfills with similar time-based cadence (referred as "grouping").
(3) Eager scheduling, which produces requests for all dependent jobs eagerly in advance (referred to as "eager").




(a) Binomial.




(b) Life-time tables & self dependency.




(c) Data pipeline patterns

Fig. 8. Three common dependency structures

The results are summarized in Table 10. Eager scheduling should produce the optimal result because all $K$ backfills are issued together. The data warehouse system has knowledge of all requests and can therefore make the best decisions about how to combine them. All variations of our algorithms either achieved the theoretical optimal result or came very close to it, eliminating 19% to 26% non-trivial duplicates, inline with the gains from the YouTube Data Warehouse. The worst case performance of lazy without grouping only results in less than 0.5% unnecessary computation.

Table 10. Duplicates detection under different strategies

| Set Up | | Statistics | | | |
|---|---|---|---|---|---|
| Backfills | Method | Initial requests | Processed requests | Unnecessary updates | Not-exact duplicates |
| 10 | Lazy | 10 | 1094 | 5 | - |
| | Group | 10 | 1089 | 0 | - |
| | Eager | 4279 | 1089 | 0 | 251 |
| 20 | Lazy | 20 | 1172 | 3 | - |
| | Group | 20 | 1169 | 0 | - |
| | Eager | 7418 | 1169 | 0 | 418 |
| 50 | Lazy | 50 | 1254 | 1 | - |
| | Group | 50 | 1253 | 0 | - |
| | Eager | 7029 | 1253 | 0 | 443 |

We assume that updating an upstream base table costs more than updating a downstream view, so we can offer a realistic analysis of the cost savings. We assume that the costs follow a normal distribution, with a mean of $max\_cost - level$, where $level$ is the level of the table on the binomial tree. We vary $max\_cost$ from 60 to 100, and fix the standard deviation to 10. When $max\_cost$ is 60, updating a table at level 0 is 6 times more expensive than updating a view at the leaf level. The average savings from eliminating non-exact duplicated updates are summarized in Table 11.

Table 11. Percentage of average cost savings.

| Maximum cost | 60 | 70 | 80 | 90 | 100 |
|---|---|---|---|---|---|
| Percentage of average cost savings | 26.7% | 26.9% | 26.4% | 26.5% | 26.5% |

There are no statistically significant differences in the average savings realized when varying $max\_cost$. This is because deduplication happens at all levels.

*7.2.2 Life-time Roll Ups via Self-Dependency.* Life-time roll up via self dependency (Figure 8b) is a pattern often used in conjunction with a time-series database to store and query data over time. The accumulative table can be used to generate summary statistics, such as the total sales for a given product over time, whereas the incremental tables can be used to store the raw data for each day. When a partition of the incremental table is backfilled, the updates will trickle down to multiple partitions of the accumulative table.

Let there be a backfill of $i[t-2](V_i)$, $i[t-1](V_j)$, and backfill of $i[t](V_k)$. If $V_k > V_j > V_i$, and take advantage of knowing all requests in the system, the version constraints for $a[t]$ would be $[i[t-1] \geq V_j, i[t] \geq V_k]$. That is, we will only see requests for $a[t-2](V_i)$, $a[t-1](V_j)$, and

$a[t](V_k)$. Without global knowledge, we could see $a[t]$ computed three times, once for $V_k$, once for $V_j$, and once for $V_i$, and $a[t-1]$ backfilled twice. A total of 6 requests, versus 3 requests total.

In our experiments, we varied the number of partitions of the incremental table to backfill and compared our algorithm with both naive and optimal strategies. We picked a 30-day period and backfilled the incremental table every N days, where N was 8, 4, and 2, respectively.

The results are summarized in Table 12. As we can observe, our algorithm has achieved optimal results, reducing unneeded computation from 53% to 85%. This translates directly to savings in computing resource at the same or higher proportion.

Table 12. Performance of self-dependency topology.

| | Statistics | | |
|---|---|---|---|
| Backfills (every N days) | Naive | Our algorithm | Optimal |
| 3 (every 8 days) | 60 | 28 | 28 |
| 7 (every 4 days) | 147 | 35 | 35 |
| 14 (every 2 days) | 280 | 43 | 43 |

*7.2.3 Updates in a Common Pipeline Topology.* In the final experiment, we used a common, simple topology (Figure 8c) from [29] to show that our approach is effective for very simple data pipelines. We also explored the trade-off between cost savings and data freshness constraints as shown in Section 4.2.

Three ingestion jobs extract data from different sources, then one transformation job pre-processes the data, which is then used for three business purposes.

We simulated three scenarios. For each scenario, ten backfills from random ingestion jobs were triggered with a random delay of 1 - 30 seconds in between to simulate a real-world setting. This prevented the system from predicting the incoming updates. Each run propagated through the pipeline. The minimal update count is 5 - 7, and the optimal depends on what the system could observe. In the first simulation, each update has a higher version and should not be rejected. In the second simulation, each update is labeled with a random version, and in the third simulation, each update is labeled with a lower version.

Table 13. Performance on a common data pipeline pattern.

| Ordering | Eager | | Eager w queuing | | Lazy | Lazy w queuing |
|---|---|---|---|---|---|---|
| | # updates | # dups | # updates | # dups | # updates | # updates |
| Ascending | 14 | (36) | **7** | (43) | 18 | **7** |
| Random | 10 | (20) | **7** | (23) | 14 | 8 |
| Descending | **7** | (4) | **7** | (4) | **7** | **7** |

The results are shown in Table 13. Columns "# updates" and "# dups" display the number of updates performed and the number of non-trivial duplicates identified, respectively. Outdated updates were rejected directly, resulting in fewer duplicate updates in the later two cases. For the columns with “queuing,” we assumed that the final stage has a 10 minute freshness constraint, computed the proper queuing budget for each stage, and held the requests in the queue. This increased the pool of compatible updates and resulted in fewer updates that had to be done. The

bolded numbers are the theoretical optima. The experiment was performed multiple times, and the results were consistent. We intend to further experiment with the optimal trade-off in future work.

In all our studies, the deduplication rates were found to be either at the theoretical optimum or very close to it, regardless of the size and complexity of the dependency graphs used.

## 8 RELATED WORK

### 8.1 Consistency Models

Our work is motivated by the desire to achieve whole-warehouse consistency, which requires vertical, horizontal, and replica consistency. We believe that our work is the first to use strict monotonicity of updates to ensure eventual consistency of data warehouses. Additionally, no prior work has discussed all three types of consistency in the data warehouse setting.

Ling and Sze [27] proposed a view maintenance model using version numbers to order updates to support vertical consistency.

Mesa [14] also supports versions, and changed rows between two versions $V_1$, $V_2$ are represented by delta $[V_1, V_2]$. The Mesa paper describes a consistent update mechanism built on top of Paxos to enforce consistency among replicas in multiple data centers.

Golab and Johnson [11] discussed consistency models in a stream data warehouse with table partitions. They introduced the notion that a base table can be open, closed, and completed. Update consistency where a data warehouse administrator can specify the desired update consistency, e.g., only update the materialized views when the base table partition is closed. We eliminate unnecessary updates backfills by merging compatible update requests.

Zhuge *et al.* [33] categorized consistency in a data warehouse into three layers: base, single view, and multi-view consistency. The latter two roughly map to our vertical and horizontal consistency. One key difference is that we define consistency over versioned table partitions. Our approach optimizes change propagation when any table partitions can be updated independently at any time.

Barga *et al.* [4] introduced a temporal model for an event streaming system to unify and enrich query language features. They formally defined a spectrum of consistency levels to deal with latency or out-of-order delivery.

Elberzhager *et al.* [25] propose to use a domain-driven design approach for consistency in distributed data-intensive systems to partition the system into different domains, each of which is responsible for a subset of the data. Our technique can be used to deduplicate updates required in their design.

### 8.2 View Maintenance & Updates-Combining

There are two main approaches to view maintenance: recomputation and incremental view maintenance. Incremental view maintenance is based on relational algebra and can be performed on the pre-state or after-state of the updates to the base table. View maintenance can also be performed eagerly or lazily [16].

The 2VNL algorithm, introduced by Quass and Widom [22], minimizes the number of updates that need to be processed by combining updates that can be applied to the same view.

Agrawal *et al.* [2] developed the SWEEP algorithm which handles updates with linear message complexity and the Nested SWEEP algorithm that can accumulate multiple updates. Blakeley *et al.* [5] developed a method to detect irrelevant and relevant updates in order to reduce the number of re-evaluations. They identified necessary and sufficient conditions for detecting updates that are irrelevant to the views. Ceri and Widom [24] proposed a mechanism to generate incremental maintenance rules and propagate updates to views. Yang, Golab and Ozsu [32] extend incremental view maintenance to handle new challenges in stream analytics via View Delta Functions (ViewDFs)

that specify how to update views when a batch of new data arrives. They describe techniques to auto translate queries into ViewDFs, and take advantage of temporal locality of the data.

However, these algorithms are not always applicable. The data warehouse may not have access to the information necessary to run these algorithms. The business logic could also be too complex for these algorithms to handle.

Our solution operates as a framework for updates deduplication, allowing for various representations of updates (restatements, or amendments). It can be used in conjunction with traditional view maintenance methods, or supports incremental processing. It is easy to integrate into a heterogeneous, distributed data warehouse where the operator software might not have full control over the internals of query processors.

The framework can be used to deduplicate updates of any size or duration. It minimizes unneeded updates by combining updates that are compatible and maximizing the pool of compatible updates by manipulating request queuing time and taking advantage of global knowledge. In our experimental studies, update requests in our synthetic workload took 20 seconds to a minute to execute, and we consistently achieved significant deduplication rates.

## 9 CONCLUSIONS

We have proposed a novel approach for ensuring the eventual consistency of large-scale data warehouses. This approach is easy to incorporate into other distributed data warehouses, and it can significantly reduce the number of unnecessary table updates, which can lead to significant resource savings.

## ACKNOWLEDGMENTS

We would like to thank Aaron Shon, Vaishnavi Sashikanth, Bin Liu, Ryan Voong, Tianli Ding, Fred Faber, Huijun Xiong, John Newlin and Jonathan Diaz for their valuable suggestions to the paper.

We would like to thank the anonymous reviewers for their insightful comments and suggestions. Their feedback was invaluable in improving the quality of this paper.

## REFERENCES

[1] Ahmed Abadi. 2009. Column-stores vs. row-stores: How different are they really? *SIGMOD Record* 38, 1 (2009), 1–18.

[2] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. 1997. Efficient View Maintenance at Data Warehouses. *SIGMOD Record (ACM Special Interest Group on Management of Data)* 26, 2 (June 1997), 417–427.

[3] Amazon Web Services (AWS). 2023. *AWS CodePipeline.* Amazon.com, Inc. https://aws.amazon.com/codepipeline/ [Accessed: November 29, 2022].

[4] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. 2007. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *CIDR.* Association for Computing Machinery, Asilomar Conference Grounds, United States, 363–374. https://doi.org/10.1145/1206348.1206388

[5] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. 1986. Efficiently updating materialized views. *SIGMOD Record (ACM Special Interest Group on Management of Data)* 15, 2 (June 1986), 61–71.

[6] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert Henry, and Robert Bradshaw. 2010. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* Association for Computing Machinery, 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 363–375. http://dl.acm.org/citation.cfm?id=1806638

[7] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew McCormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Aharon Ebenstein, Nikita Mikhaylin, Hung ching Lee, Xiaoyan Zhao, Guanzhong Xu, Luis Antonio Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Vera Lychagina, and Brett Elliott. 2019. Procella: Unifying serving and analytical data at YouTube. *PVLDB* 12(12) (2019), 2022–2034. https://dl.acm.org/citation.cfm?id=3360438

[8] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R. Ganger. 2020. Unearthing inter-job dependencies for better cluster scheduling. In *USENIX Symposium on Operating Systems Design and Implementation.* USENIX Association, Carlsbad, CA, 689–705. https://doi.org/10.5555/3434127.3437541

[9] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. 1996. Algorithms for deferred view maintenance. *SIGMOD Record (ACM Special Interest Group on Management of Data)* 25, 2 (June 1996), 469–480.

[10] Start data engineering. 2021. *How to backfill a SQL query using Apache Airflow.* Start Data Engineering. Retrieved Jan 6, 2021 from http://www.startdataengineering.com

[11] Lukasz Golab and Theodore Johnson. 2011. Consistency in a Stream Warehouse. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR '11)*. ACM, Association for Computing Machinery (ACM), Asilomar, CA, 291–302. https://doi.org/10.1145/1920841.1920889

[12] Google LLC. 2023. *Google Cloud Dataflow Documentation: Data Pipelines.* Google Cloud. [Accessed: November 29, 2022].

[13] Timothy Griffin and Leonid Libkin. 1995. Incremental maintenance of views with duplicates. *SIGMOD Record (ACM Special Interest Group on Management of Data)* 24, 2 (May 1995), 328–339.

[14] Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan, Kevin Lai, Shuo Wu, Sandeep Dhoot, Abhilash Kumar, Ankur Agiwal, Sanjay Bhansali, Mingsheng Hong, Jamie Cameron, Masood Siddiqi, David Jones, Jeff Shute, Andrey Gubarev, Shivakumar Venkataraman, and Divyakant Agrawal. 2014. Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing. In *VLDB*. VLDB Endowment, Hangzhou, China, 1285–1296.

[15] Stratos Idreos and Dimitris Papadias. 2010. The Design and Implementation of Modern Column-Oriented Database Systems. *Transactions on Database Systems* 35, 4 (2010), 1–58.

[16] Hicham G. Elmongui Jinren Zhou, Per-Ake Larson. 2007. Lazy maintenance of materialized views. *Proceedings of the 33rd international conference on Very large data bases.* 1, 1 (2007), 231–242.

[17] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record* 34, 1 (2005), 39–44.

[18] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. 2008. Out-of-order processing: a new architecture for high-performance stream systems. *Proc. VLDB Endow.* 1, 1 (2008), 274–288. https://doi.org/10.14778/1453856.1453890

[19] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, Asilomar Conference Grounds, Pacific Grove, CA, USA, 3–14. https://doi.org/10.1145/2452369.2452372

[20] Derek Gordon Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martín Abadi. 2016. Incremental, iterative data processing with timely dataflow. *Commun. ACM* 59, 10 (2016), 75–83. https://doi.org/10.1145/2983551

[21] Jayesh Patel. 2019. An Effective and Scalable Data Modeling for Enterprise Big Data Platform. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, IEEE, Los Angeles, CA, USA, 2691–2697. https://doi.org/10.1109/BigData47090.2019.9005614

[22] Dallan Quass and Jennifer Widom. 1997. On-line Warehouse View Maintenance. *SIGMOD Record (ACM Special Interest Group on Management of Data)* 26, 2 (June 1997), 393–404.

[23] Snowflake Inc. 2023. *Snowflake Data Load Auto-Ingest with Snowpipe.* Snowflake Inc. Accessed: November 29, 2022.

[24] Jennifer Widom Stefano Ceri. 1991. Deriving Production Rules for Incremental View Maintenance. *Proceedings of the 17th International Conference on Very Large Data Bases.* 1, 1 (1991), 577–589.

[25] Frank Elberzhager Susanne Braun, Annette Bieniusa. 2021. Advanced Domain-Driven Design for Consistency in Distributed Data-Intensive Systems. In *8th Workshop on Principles and Practice of Consistency for Distributed Data (PapoC's21)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/3447865.3457969

[26] The Apache Software Foundation. 2023. *Apache Airflow Documentation: Scheduler.* The Apache Software Foundation. Accessed: November 29, 2022.

[27] Eng Kong Sze Tok Wang Ling. 1999. Materialized View Maintenance Using Version Numbers. In *Proceedings of the Sixth International Conference on Database Systems for Advanced Applications (DASFAA '99)*. IEEE Computer Society, Florence, Italy, 160–173. https://doi.org/10.1109/DASFAA.1999.753869

[28] Chao Tian Wenfei Fan, Chunming Hu. 2017. Incremental Graph Computations: Doable and Undoable.. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, Chicago, IL, USA, 155–169. https://doi.org/10.1145/3035918.3035944

[29] Data Pipeline Wiki. 2022. *Data Pipeline.* Data Pipeline Community. https://dataengineering.wiki/Concepts/Data+Pipeline [Accessed: June 5, 2022].

[30] Wikipedia. 2021. *Monotonic function.* Wikipedia Foundation, Inc. Retrieved June 28, 2021 from https://en.wikipedia.org/wiki/Monotonic_function

[31] Wikipedia. 2021. *Service-level objective.* Wikipedia Foundation, Inc. Retrieved December 20, 2021 from https://en.wikipedia.org/wiki/Service-level_objective

[32] Yuke Yang, Lukasz Golab, and M. Özsu. 2017. ViewDF: Declarative incremental view maintenance for streaming data. *Inf. Syst.* 71 (2017), 55–67.

[33] Yue Zhuge, J.L. Wiener, and H. Garcia-Molina. 1997. Multiple view consistency for data warehousing. In *Proceedings 13th International Conference on Data Engineering*. IEEE Computer Society, Birmingham, AL, USA, 289–300.

Received November 2022; revised February 2023; accepted March 2023