What's the Difference? Incremental Processing with Change Queries in Snowflake

TYLER AKIDAU, Snowflake, Inc., USA
PAUL BARBIER, Snowflake, Inc., Germany
ISTVAN CSERI, Snowflake, Inc., USA
FABIAN HUESKE, Snowflake, Inc., Germany
TYLER JONES, Snowflake, Inc., USA
SASHA LIONHEART, Snowflake, Inc., USA
DANIEL MILLS, Snowflake, Inc., USA
DZMITRY PAULIUKEVICH, Snowflake, Inc., Germany
LUKAS PROBST, Snowflake, Inc., Germany
NIKLAS SEMMLER, Snowflake, Inc., Germany
DAN SOTOLONGO, Snowflake, Inc., USA
BOYUAN ZHANG, Snowflake, Inc., USA

Incremental algorithms are the heart and soul of stream processing. Low latency results depend on the ability to react to the subset of changes in a dataset over time rather than reprocessing the entirety of a dataset as it evolves. But while the SQL language is well suited for representing streams of changes (via tables) and their application to tables over time (via DML), it entirely lacks a method to query the changes to a table or view in the first place.

In this paper, we present `CHANGES` queries and `STREAM` objects, Snowflake's primitives for querying and consuming incremental changes to table objects over time. `CHANGES` queries and `STREAM`s have been in use within Snowflake for three years, and see broad adoption across our customers. We describe the semantics of these primitives, discuss the implementation challenges, present an analysis of their usage at Snowflake, and contrast with other offerings.

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

Additional Key Words and Phrases: changelogs, CDC, IVM, query differentiation, streaming SQL

ACM Reference Format:
Tyler Akidau, Paul Barbier, Istvan Cseri, Fabian Hueske, Tyler Jones, Sasha Lionheart, Daniel Mills, Dzmitry Pauliukevich, Lukas Probst, Niklas Semmler, Dan Sotolongo, and Boyuan Zhang. 2023. What's the Difference? Incremental Processing with Change Queries in Snowflake. *Proc. ACM Manag. Data* 1, 2, Article 196 (June 2023), 27 pages. https://doi.org/10.1145/3589776

Authors' addresses: Tyler Akidau, tyler.akidau@snowflake.com, Snowflake, Inc., Bellevue, Washington, USA; Paul Barbier, paul.barbier@snowflake.com, Snowflake, Inc., Berlin, Germany; Istvan Cseri, istvan.cseri@snowflake.com, Snowflake, Inc., Bellevue, Washington, USA; Fabian Hueske, fabian.hueske@snowflake.com, Snowflake, Inc., Berlin, Germany; Tyler Jones, tyler.jones@snowflake.com, Snowflake, Inc., San Mateo, California, USA; Sasha Lionheart, sasha.lionheart@snowflake.com, Snowflake, Inc., Bellevue, Washington, USA; Daniel Mills, daniel.mills@snowflake.com, Snowflake, Inc., Bellevue, Washington, USA; Dzmitry Pauliukevich, dzmitry.pauliukevich@snowflake.com, Snowflake, Inc., Berlin, Germany; Lukas Probst, lukas.probst@snowflake.com, Snowflake, Inc., Berlin, Germany; Niklas Semmler, niklas.semmler@snowflake.com, Snowflake, Inc., Berlin, Germany; Dan Sotolongo, dan.sotolongo@snowflake.com, Snowflake, Inc., San Mateo, California, USA; Boyuan Zhang, boyuan.zhang@snowflake.com, Snowflake, Inc., Bellevue, Washington, USA.

2836-6573/2023/6-ART196
https://doi.org/10.1145/3589776

1 INTRODUCTION

Batch vs. streaming: the perennial debate. Many opinions. Much controversy. Even so, the generally accepted distinction between the two approaches remains steadfast: batch systems process input datasets in their entirety to produce new output datasets in their entirety; streaming systems process the *changes* to input datasets over time to *incrementally* evolve their corresponding outputs over time.

This incrementalism, the ability to perform a computation in separable pieces over time, is the foundation of what most people consider to be stream processing. And indeed, in most modern streaming systems, incrementalism is front and center. Actor patterns [3] abound in stream processing APIs, presenting a programming interface inherently tied to record-by-record processing. NoSQL-style architectures pervade large scale streaming system design, fueling a broad suite of key-partitioned incremental use cases where computation *across* keys happens in parallel, but processing for any *single* key happens serially, record by record [4, 22, 61, 64].

Within the modern streaming ecosystem has then sprung the idea of table and stream duality [5, 38, 55]. Tables represent the complete state of a dataset at each point in time; effectively equivalent to a table in a relational database. Streams, on the other hand, capture the *changes* to a dataset *over time*; if only one row changes within a time interval, the stream captures that single change. Tables and streams are thus intimately related, representing the same information over time, but in different ways.

As is the case with so much in streaming, the relational database community has pondered and solved many of these same questions over the decades, to varying degrees. Streams are, in effect, just insert-only tables, and are utilized as such frequently. DML performs stream to table conversions, with `MERGE` [59] being the Swiss Army Knife of stream to table conversion, resolving a stream of changes as applied to a table. However, standard SQL has insufficient features to extract the changes that were applied to a table. Research efforts from the early 2000s proposed means for codifying table to stream conversions within language primitives [18], but most modern SQL implementations do not address the challenges that users face when extracting and processing changelogs.

Even so, querying changes directly remains an important capability for a number of streaming oriented use cases:

- **Event queuing**: Message queues underpin a broad swath of event processing solutions. Purpose built systems like Apache Kafka allow events to be conveniently enqueued and dequeued while maintaining some amount of order. Within SQL, `INSERT`-only tables provide a reasonable facsimile for message queues from an enqueue and storage perspective, but the SQL language lacks a convenient transactional dequeue mechanism.
- **Notifications**: Use cases where action must be taken based on specific changes in the data, such as notifications, lend themselves poorly to a table-oriented interface: incrementally consuming a stream of such changes is typically far cheaper than repeatedly polling an entire tabular dataset looking for changes.
- **Incremental View Maintenance**: Incrementally maintaining derived views of data sees broad applicability. Realtime dashboards often rely on some form of incremental view maintenance (IVM), whether it be within the underlying database, or a direct feed of incremental changes into the dashboard itself, in order to continuously render an up-to-date view of derived data with low latency and reasonable cost. Many databases provide some sort of materialized view primitive that utilizes IVM under the covers. But the supported operations and the efficiency of their implementation vary greatly, oftentimes leaving users to implement their own IVM when the database falls short.

- **Flexible change transformation**: There are many cases where being able to transform changes with SQL greatly reduces the overall complexity of a solution. For example, in scenarios that extract change data from one database and load it into another (CDC/ETL scenarios), one may only be interested in a subset of changes, or may want to interpret values in specific ways. Performing those transformations in the database with SQL greatly simplifies the story. Moreover, extracting changes via the usual database query interface (like JDBC) is a lot easier than configuring and connecting to a separate CDC endpoint.

In all these cases, it is desirable for the system to conveniently and efficiently provide user access to changes within the database itself.

In this paper, we present Snowflake's primitives for table-to-stream conversions: `CHANGES` queries and `STREAM` objects. In Section 2 we discuss semantics, both generally and for Snowflake's primitives specifically. This is followed by an overview of Snowflake's implementation in Section 3, and evaluations of usage and performance in Section 4. In Section 5, we survey related work. We then wrap things up with a short discussion of future work in Section 6, followed by a summary in Section 7.

2 SEMANTICS

To begin with, it's best to discuss the semantics of tables and streams in the abstract before diving into Snowflake's `CHANGES` queries and `STREAM` objects concretely. Abstractly, a table object represents a mutable set of rows over time. We sometimes refer to these as time-varying relations (or TVRs) [20], because they represent the state of a relation as it evolves over time. Changes occur as rows are added or removed from the relation, with each change yielding a new snapshot within the overall TVR.

An alternative way of representing the state of a relation over time is by directly encoding the stream of changes to it, rather than the sequence of point-in-time relations themselves [18, 55]. This is directly analogous to the redo logs prevalent in many database systems. Redo logs capture all changes to a TVR over time as a sequence of `INSERT`, `UPDATE`, and `DELETE` operations.

Concretely, redo logs may be represented in a number of ways. For example, some may bundle `UPDATE` operations together as a single row with both old and new values for columns that changed, while others may treat them as independent matching `DELETE` and `INSERT` operations for the old and new row respectively. Each of these change formats offer different tradeoffs that system implementations can consider when settling on an approach.

As an example, consider the following table in a hypothetical database. Notice that statements are associated with timestamps, which are consistent throughout this paper.

```
12:00> SELECT * FROM people;
+----+-------+
| id | name  |
+----+-------+
| 1  | Jeff  |
| 2  | Donny |
+----+-------+
```

Listing 1. Example table with two rows

If we were to apply a sequence of DML changes to that table, it would yield a new relation:

```
12:01> INSERT INTO people VALUES
         (3, 'Walter'),
         (4, 'Maud'),
         (5, 'Uli');
```

```
12:02> UPDATE people SET name = 'Jeffrey' WHERE id = 1;
12:03> UPDATE people SET name = 'Maude' WHERE id = 4;
12:04> DELETE FROM people WHERE id in (2, 5);
12:05> SELECT * FROM people;
+----+---------+
| id | name    |
+----+---------+
| 1  | Jeffrey |
| 3  | Walter  |
| 4  | Maude   |
+----+---------+
```

Listing 2. Mutations to the table from Listing 1

If this hypothetical database supported change queries with a redo log format, the redo log for these changes might look something like this:

```
12:06> SELECT * FROM people CHANGES AT (TS => 12:00);
+----+---------+---------+-----------+----------+
| id | name    | $ACTION | $ISUPDATE | $ROW_ID  |
+----+---------+---------+-----------+----------+
| 3  | Walter  | INSERT  | FALSE     | 5ca38031 |
| 4  | Maud    | INSERT  | FALSE     | e4d0387c |
| 5  | Uli     | INSERT  | FALSE     | 16d193d9 |
| 1  | Jeff    | DELETE  | TRUE      | f70c423b |
| 1  | Jeffrey | INSERT  | TRUE      | f70c423b |
| 4  | Maud    | DELETE  | TRUE      | e4d0387c |
| 4  | Maude   | INSERT  | TRUE      | e4d0387c |
| 2  | Donny   | DELETE  | FALSE     | f0452fd5 |
| 5  | Uli     | DELETE  | FALSE     | 16d193d9 |
+----+---------+---------+-----------+----------+
```

Listing 3. Hypothetical redo log capturing Listing 2 mutations

We will discuss the extra metadata columns in this example shortly, but for now they should be relatively self evident. The redo log encodes the full set of DML changes made to the table as a sequence of `INSERT`s and `DELETE`s, with `UPDATE` operations represented as labeled `INSERT` and `DELETE` pairs with matching row IDs.

Although this specific example shows changes to a persistent table, it's important to highlight that the concepts generalize to any abstract table object, including views defined via complex `SELECT` statements on one or more persistent tables and views. This generalization is an important motivation for the dynamic approach of change stream rendering taken in the Snowflake approach, as we'll discuss later.

Now that we understand the abstract concept of tables and change streams, we can dive more specifically into the primitives supported in Snowflake.

2.1 `CHANGES` queries

A `CHANGES` query [56] is a special type of Snowflake query that can be used to observe the changes to a table object between two points in time. The time window can be specified via an interval passed to the `CHANGES` query using `AT` and an optional `END` clause which define the start and end of the interval, respectively. If unspecified, `END` defaults to the current time. These clauses accept a Query UUID, a wall-clock timestamp, an offset from the current time, or a `STREAM` object (discussed in more detail below).

Such a query renders the changes that occurred in the source during the interval. In addition to the columns in the source, CHANGES queries define several "metadata" columns:

- **$ACTION**: Indicates the DML operation that resulted in that row, which can be one of INSERT or DELETE.
- **$ISUPDATE**: Updates to the table are modeled as a pair of rows with actions of DELETE and INSERT respectively, both of which must have $ISUPDATE marked true.
- **$ROW_ID**: Specifies a unique and immutable ID for the row which can be used to track row changes over time, including identifying matched DELETE and INSERT pairs for UPDATEs.

In addition to the time interval, CHANGES queries accept a parameter, INFORMATION, describing the change format to use. Snowflake today supports two types of change formats: *append-only* and *minimum-delta*.[1] Both are in contrast to the more traditional *redo log* style of change formats described above.

Append-only changes are those which occur when a row is first inserted into a table. An insert can happen via INSERT, MERGE, or COPY statements. It can also be driven by Snowpipe or Snowpipe Streaming, which are Snowflake's file- and row-based data ingestion features.

Append-only changes are useful for efficiently seeing new rows from one table, which can then be used in transformations and written to other tables. This matches the needs of the insert-only, event-driven types of use cases that many streaming systems focus on. Though append-only changes are conceptually just a filter[2] on a redo log, it's possible to render append-only streams far more efficiently given Snowflake's implementation.

A common pattern is to request append-only changes on a table populated via Snowpipe or Snowpipe Streaming. Append-only changes will only fetch rows that have been inserted into the table. Consumers of those changes can transform the raw ingestion data as needed and insert it into one or more tables, which is easier than doing it before ingestion. In order to keep storage costs low users typically then issue DELETE or TRUNCATE statements against the initial table. By design, these deletions are ignored by append-only changes.

Consider again the mutations from Listing 2. An append-only rendering of those changes would look something like this:

```
12:06> SELECT * FROM people
         CHANGES(INFORMATION => APPEND_ONLY)
         AT (TS => 12:00);
+----+---------+---------+-----------+----------+
| id | name    | $ACTION | $ISUPDATE | $ROW_ID  |
+----+---------+---------+-----------+----------+
| 3  | Walter  | INSERT  | FALSE     | 5ca38031 |
| 4  | Maud    | INSERT  | FALSE     | e4d0387c |
| 5  | Uli     | INSERT  | FALSE     | 16d193d9 |
+----+---------+---------+-----------+----------+
```

Listing 4. Append-only rendering of Listing 2 mutations

Compared to the redo log rendering from Listing 3, only the strict INSERT mutations are included, with the UPDATE and DELETE operations being effectively filtered out.

Minimum-delta changes comprise the smallest set of INSERT, UPDATE, and DELETE modifications that account for the difference between two points in time. In other words, they consolidate any redundant modifications that may have taken place during that interval. Min-delta changes

[1] Note that DEFAULT is the syntax keyword used for the minimum-delta change format.
[2] WHERE $ACTION = INSERT AND $UPDATE = FALSE

always capture the full row, even if only a single column is changed, because partial updates are often painful and costly to deal with.

Minimum-delta changes are commonly used in analytical scenarios where the net result of a changelog is required. Most incremental algorithms operate just as effectively (and sometimes more efficiently) using net changes rather than the fine-grained level of detail in redo logs. And in some cases these consolidated changes are also simpler to work with, such as CDC applications where the contents of a table object are being replicated into another database via `MERGE` statements, since `MERGE` behavior is undefined when the source of the `MERGE` has multiple rows with the same primary key.

To see what min-delta changes look like in practice, compare the redo log changes from Listing 3 to this minimum-delta rendering of those same mutations:

```
12:06> SELECT * FROM people
         CHANGES(INFORMATION => DEFAULT)
         AT (TS => 12:00);
+----+---------+---------+-----------+----------+
| id | name    | $ACTION | $ISUPDATE | $ROW_ID  |
+----+---------+---------+-----------+----------+
| 3  | Walter  | INSERT  | FALSE     | 5ca38031 |
| 1  | Jeff    | DELETE  | TRUE      | f70c423b |
| 1  | Jeffrey | INSERT  | TRUE      | f70c423b |
| 4  | Maude   | INSERT  | FALSE     | e4d0387c |
| 2  | Donny   | DELETE  | FALSE     | f0452fd5 |
+----+---------+---------+-----------+----------+
```

Listing 5. Delta rendering of Listing 2 mutations

As before, the `INSERT` for *Walter* is present, as is the `DELETE` for *Donny* and the `DELETE+INSERT` pair for the `UPDATE` to *Jeffrey*. However, the `INSERT` of *Maud* followed by the `UPDATE` to *Maude* have been coalesced into a single non-`UPDATE` `INSERT`. And the `INSERT` plus `DELETE` of *Uli* does not show up at all, because there was no net effect of those changes across the specified time interval. Though more compact than the redo log from Listing 3, applying the changes in this delta log to the initial table from Listing 1 still results in the same table shown at the end of Listing 2.

The key use case where minimum-deltas fall short is auditing scenarios. In those situations, it is critical to capture all changes to the database, regardless of whether they were later undone. For that reason, we will likely expose redo logs in Snowflake `CHANGES` queries in the future. But for now, append-only and delta queries have proven to effectively cover the vast majority of our customers' needs.

2.2 `CHANGES` queries on Views

Before we move on to talking about `STREAM` objects, it's worth looking a little bit closer at `CHANGES` queries on views. The semantics of change queries on views has a simple definition, but several subtle behaviors manifest out of this definition in practice. We discuss a few examples here.

2.2.1 Append-only `CHANGES` *on views.* Defining the append-only change format leads to ambiguities that must be resolved at the discretion of the implementer.

- **Monotonicity**: Computing append-only changes over non-monotonic queries (where inserts in the sources can lead to deletes in the result, e.g. anti-joins) is effectively the same as computing a redo log, then filtering out deletes. For these queries, the performance benefit of the append-only format is lost. This violates programmers' expectations that "append-only changes are cheap". To abide by the principle of least astonishment, we restrict append-only

streams to monotonic queries, favoring consistent performance over complete coverage of query classes.
- **Repeated Inserts**: Some queries can result in a row being inserted, then deleted, and then inserted again at a later time. For an append-only stream, it's not obvious whether such repeated inserts should be excluded or not. We chose to exclude them, as it leads to a simpler implementation.

2.2.2 Minimum-delta `CHANGES` *on views.* Secondly, minimum-delta `CHANGES` on view queries can yield surprising results:

- **Excluded columns**: For a view that selects a subset of columns, updates to excluded columns do not yield any changes.
- **`UPDATE` coercion**: For a view that filters out rows, updates that change a row from being excluded to included become `INSERT`s. Conversely, updates that change a row from included to excluded become `DELETE`s.

To see this in action, consider the following tables.

```
12:10> SELECT *
    FROM people;
+----+---------+
| id | name    |
+----+---------+
| 1  | Jeffrey |
| 2  | Donny   |
| 3  | Walter  |
| 4  | Maude   |
+----+---------+
```

```
12:11> SELECT * FROM items;
+----+-----+--------------+---------+
| id | oid | item         | desc    |
+----+-----+--------------+---------+
| 11 | 2   | Ball         | Bowling |
| 12 | 2   | Surfboard    | Yater   |
| 13 | 1   | Car          | 1973    |
| 14 | 1   | Rug          | Classic |
| 15 | 4   | Autobahn LP  |         |
+----+-----+--------------+---------+
```

Listing 6. People and items tables

The *people* table is as before, while the *items* table contains items, their descriptions, and a foreign key for the owner of the item (*oid*).

Now imagine we create a view that joins those two tables on the owner IDs, containing only the *name* column from the *people* table and the *item* column from the *items* table.

```
12:12> CREATE VIEW owner_and_items AS SELECT
  name, item FROM people JOIN items ON people.id = oid;

12:13> SELECT * FROM owner_and_items;
+---------+--------------+
| name    | item         |
+---------+--------------+
| Donny   | Ball         |
| Donny   | Surfboard    |
| Jeffrey | Car          |
| Jeffrey | Rug          |
| Maude   | Autobahn LP  |
+---------+--------------+
```

Listing 7. Inner join view, *owner_and_items*

Even though this is a user constructed view with a dynamically specified query, it's possible to now get change stream information in Snowflake. To see this in action, imagine we make the following modifications to the base tables, and then inspect the min-delta changes on the view:

```
12:14> UPDATE items SET item = 'Ford' WHERE id = 13;
       UPDATE items SET oid = 4 WHERE id = 14;
       UPDATE items SET desc = 'Techno' WHERE id = 15;
       DELETE FROM people WHERE id = 2;

12:15> SELECT * FROM owner_and_items;
+---------+--------------+
| name    | item         |
+---------+--------------+
| Jeffrey | Ford         |
| Maude   | Rug          |
| Maude   | Autobahn LP  |
+---------+--------------+

12:16> SELECT * FROM owner_and_items
         CHANGES(Information => DEFAULT)
         AT(TS => 12:13);
+---------+-----------+---------+-----------+----------+
| name    | item      | $ACTION | $ISUPDATE | $ROW_ID  |
+---------+-----------+---------+-----------+----------+
| Donny   | Ball      | DELETE  | FALSE     | a438feb7 |
| Donny   | Surfboard | DELETE  | FALSE     | 82fc9cd3 |
| Jeffrey | Car       | DELETE  | TRUE      | 34dce5de |
| Jeffrey | Ford      | INSERT  | TRUE      | 34dce5de |
| Jeffrey | Rug       | DELETE  | FALSE     | 0d5eebca |
| Maude   | Rug       | INSERT  | FALSE     | 08e24602 |
+---------+-----------+---------+-----------+----------+
```

Listing 8. CHANGES on an inner join view

We see a number of interesting things here:

- When we rename the *Car* item to *Ford*, this shows up as a matching DELETE+INSERT pair both marked as updates, and with matching row IDs, as you'd expect.
- When we change the *description* for the *Autobahn LP* item to *Techno*, this change is not reflected in the CHANGES output, since the description column isn't involved in the view in any way. This is the "excluded columns" case described above.
- From the base table's perspective, when we change the *Rug* owner to *Maude*, we simply updated the row. But from the view's perspective, the old row has been filtered out, and a new row has been added. As a result, the DELETE and INSERT rows in the CHANGES output are not marked as updates, and they do not share a row ID. This is the "UPDATE coercion" case described above.
- When we delete *Donny* from the *people* table, we see two deletes for the two items that row previously matched against. The noteworthy thing here is that the CHANGES system reflects the correct meaning of an inner join, producing two DELETE rows in the output. This is unlike most streaming systems, where a change to a dimension table in a join is typically only reflected going forward with new join rows (sometimes referred to as a stream-static join.)

2.3 STREAM Objects

CHANGES queries are a powerful tool for incremental processing, but they leave a key problem unsolved: tracking the progress of processing over time. This introduces several sub-problems: how

to accurately represent progress through changes, where to store this state, and how to tolerate errors and faults. To solve all of these issues, Snowflake has an object called a STREAM.

A STREAM [57] is a schema-level catalog object. It is created on a table (persistent or view), called its *source table*. A STREAM's state, called its *frontier*, represents a point in time before which all changes to its source have been consumed. When queried, a STREAM produces the changes to its source over the interval from the frontier to the present. When queried from within a DML statement, the STREAM is modified as part of its enclosing transaction; when that transaction commits, the STREAM's frontier is moved to the end of the change interval. We call this action *consuming* the STREAM . When queried from within a multi-statement transaction, it always returns the same set of changes throughout. This makes it easy to transactionally consume the STREAM, even when using those changes across multiple destinations. In this way, complex incremental data pipelines can be constructed by periodically consuming STREAMs and applying their changes to downstream tables. Each STREAM is expected to be consumed by a single reader, and multiple STREAMs can be efficiently created on the same table.

STREAMs also support a feature called *show initial rows*, which causes the first consumption to include the current state of the table in addition to any changes. This feature makes backfills easier by encapsulating it as part of progress tracking. Once the first consumption is committed, the STREAM returns changes as usual.

The following example builds on the data in Listings 1 and 2 to demonstrate the use of STREAMs. In this example, the STREAM is consumed 3 times. The first returns the current state of the people table, the second returns the first batch of inserts, and the third returns the updates made at 12:03. Notice that, even though the INSERT occurs at 12:06, it excludes the DELETE of Donny and Uli at 12:04 because the transaction started at 12:03.

```
12:00> CREATE STREAM people_stream ON TABLE people
           SHOW_INITIAL_ROWS=true;
12:00> CREATE TABLE people_changes(
      name varchar, action varchar, isUpdate varchar);
12:00> INSERT INTO people_changes
      SELECT name, $ACTION, $ISUPDATE FROM people_stream;
12:00> SELECT * FROM people_changes;
+-------+--------+----------+
| name  | action | isUpdate |
+-------+--------+----------+
| Jeff  | INSERT | FALSE    |
| Donny | INSERT | FALSE    |
+-------+--------+----------+
12:01> TRUNCATE TABLE people_changes;

12:01> INSERT INTO people_changes
      SELECT name, $ACTION, $ISUPDATE FROM people_stream;
12:01> SELECT * FROM people_changes;
+--------+--------+----------+
| name   | action | isUpdate |
+--------+--------+----------+
| Walter | INSERT | FALSE    |
| Maud   | INSERT | FALSE    |
| Uli    | INSERT | FALSE    |
+--------+--------+----------+
12:01> TRUNCATE TABLE people_changes;
```

```
12:03> BEGIN;
12:06> INSERT INTO people_changes
      SELECT name, $ACTION, $ISUPDATE FROM people_stream;
12:09> COMMIT;
12:10> SELECT * FROM people_changes;
+---------+--------+----------+
| name    | action | isUpdate |
+---------+--------+----------+
| Jeff    | DELETE | TRUE     |
| Jeffrey | INSERT | TRUE     |
| Maud    | DELETE | TRUE     |
| Maude   | INSERT | TRUE     |
+---------+--------+----------+
```

Listing 9. Transactional Stream Consumption

A STREAM imposes only negligible storage overhead, namely that of storing its frontier in the catalog. As a result, many STREAMs can be created on the same source table without creating concerns about cost, and they can advance independently from one another. One disadvantage of directly relying on table storage is that it couples the lifetime of a STREAM with the retention policy of its source table. If a STREAM is not consumed within its source table's retention period, it can become *stale*, which means its underlying data has expired. To mitigate this problem, Snowflake automatically extends table retention to prevent STREAMs from going stale, up to a configurable maximum. This feature, which is only possible because of the separation of storage and compute at the foundation of Snowflake, ensures that staleness is not a problem in practice.

3 IMPLEMENTATION

Change queries were added to Snowflake after its core functionality was already fully implemented. In this context, the design of change queries sought to satisfy several criteria:

- Reuse existing components as much as possible.
- Integrate cleanly with current and future features.
- Minimize barriers to incremental adoption by customers.

We achieved these goals by breaking down the problem into highly-targeted augmentations to the existing system:

- Add metadata in the storage layer to track changes at row granularity.
- Implement a query differentiation framework to rewrite queries to produce changes.
- Integrate STREAMs with the transaction processing engine to enable transactional consumption of changes.

The result of this design is that STREAMs and CHANGES queries are deeply integrated with the rest of Snowflake. They leverage the existing query optimizer and execution engine. They interoperate cleanly with other Snowflake features (e.g. governance, sharing, and replication). Finally, customers are able to adopt them gradually, leveraging them to improve the components of their architecture that can most benefit from them.

This section is an overview of this implementation. We begin with some high-level background on Snowflake's existing architecture. Then, we proceed to describe the change tracking metadata and query differentiation framework we use to compute changes. Finally, we describe how STREAMs work with transactions to provide a simple, yet powerful primitive for imperative, incremental data processing.

3.1 Table Metadata

Before we talk about the implementation of change queries, we need a basic understanding of Snowflake table metadata. Tables in Snowflake are comprised of a set of immutable, columnar data files, called *micro-partitions*. Micro-partitions are stored in a blobstore such as Amazon S3, Azure Blob Storage, or Google Cloud Storage, while the metadata tracking them is stored in a FoundationDB [65] cluster.

At any given point in time, the state of a table is captured in what we call a *table version*, which comprises:

- A **system timestamp** denoting the time at which the table version becomes valid.
- The set of **micro-partitions** containing the data for the table at that version.
- **Partition-level statistics** for each micro-partition, capturing various dimensions used during query optimization, such as min/max values for columns, null counts, etc.

The state of a table over time is thus a sequence of these table versions: `INSERT` operations add new micro-partitions, and other DML operations replace existing micro-partitions with new ones via copy-on-write, or remove them via metadata operations. The metadata system tracks these table versions, coordinating transactions across them to ensure snapshot isolation in the presence of concurrent operations, and ultimately expiring them once they exceed the configured data retention horizon.

This sequence of table versions is then directly analogous to the sequence of snapshot relations comprising a time-varying relation, as discussed in Section 2. Time travel queries, which allow for observing the state of a table at a previous point in time, fall naturally out of this scheme: the metadata system simply resolves the proper table version for the requested point in time and executes the query across the corresponding set of micro-partitions. Change queries, on the other hand, require more effort.

3.2 Query Differentiation

This subsection describes the extensible framework we implemented for translating `CHANGES` and `STREAM` queries into executable query plans. To avoid overloading the word "changes" too heavily, we use terminology inspired by calculus: given that we want the `CHANGES` of a query Q over an interval I, we say we *differentiate* Q to obtain the *derivative* of Q, $\Delta_I Q$, which varies over I. The framework is implemented in terms of syntactic rewrite rules which match the derivative operator and plan beneath it, and produce an equivalent expression in terms of derivatives of its internal terms. This process eliminates all derivatives, resulting in a plan that contains only executable operators like scan, project, filter, join, etc. After being rewritten, the plan is optimized and executed like any other.

There are several key aspects to this framework: the relational equivalences which justify the rewrite rules, handling different change formats, and how to define change metadata columns. We discuss each in turn.

3.2.1 Relational Equivalences. There is substantial prior work in the area of incremental view maintenance [34, 42], which gave us an excellent starting point. Unfortunately, our survey of the academic literature concluded that many important topics were left for future work. This includes support for common operators (e.g. outer, lateral, semi-, and anti-joins, and some families of aggregations) and studies of the many performance trade-offs we encountered. Our own study of the topic is just beginning, but we relay some of what we have learned in this section.

Our goal is to find algebraic equivalences to facilitate the computation of changes over an interval. Of particular interest are those equivalences which express the derivative of an operator in terms

of derivatives of the arguments of that operator, much like the familiar chain rule in calculus[3]. By repeatedly substituting across these equivalences, the derivative operator progressively moves to the leaves of the syntax tree, and is eventually eliminated.

To find these equivalences, we followed a simple procedure:

(1) For each operator op, let $Q = op(R, ...)$, where R is a relation and ... represents additional arguments for n-ary operators.
(2) Assume some arbitrary change ∂R on each argument R.
(3) Distribute the expression $op(R + \partial R, ...)$ over $+$.
(4) Rearrange terms into the form $Q + \partial Q$, where ∂Q names terms other than Q.
(5) Define $\Delta_I Q = \partial Q[\partial R := \Delta_I R, ...]$, where $[...]$ denotes substitution of terms.

Applying this approach for a number of operators yields many useful equivalences. Some examples are shown below, starting with a set of definitions, followed by the actual equivalences.

σ	is the filter operator.
π	is the projection operator.
$+$	is union-all.
$\bowtie$	is the inner join operator.
γ_k	is an aggregation over keys k.
$\ltimes_k$	is a semijoin on k.
Δ_I	is the derivative operator over interval I.
$Q\|_t$	is the time-varying relation Q at time t.
I_0, I_1	are the start and end of I.
π_-, π_+	denote deletes and inserts.

$$
\begin{aligned}
\Delta_I(\sigma(Q)) &\implies \sigma(\Delta_I(Q)) \\
\Delta_I(\pi(Q)) &\implies \pi(\Delta_I(Q)) \\
\Delta_I(Q \bowtie R) &\implies Q|_{I_0} \bowtie \Delta_I R + \Delta_I Q \bowtie \Delta_I R + \Delta_I Q \bowtie R|_{I_0} \\
&\implies Q|_{I_0} \bowtie \Delta_I R + \Delta_I Q \bowtie R|_{I_1} \\
\Delta_I(\gamma_k(Q)) &\implies \pi_-(\gamma_k(Q|_{I_0} \ltimes_k \Delta_I Q)) + \\
&\quad\ \pi_+(\gamma_k(Q|_{I_1} \ltimes_k \Delta_I Q))
\end{aligned}
$$

Many such equivalences obtain, each with different performance characteristics. In some cases, it is immediately apparent which plan would perform better. For example, consider the two inner-join derivatives above. Consolidating three joins into two confers a clear advantage (assuming the query processor supports queries over tables at different times). However in other cases, the relative performance of different equivalences seems strongly dependent on the underlying data. For example, the group-by derivative above can be substantially *more expensive* than the undifferentiated query if most keys are modified during the change interval. We have encountered such ambiguities for operators including aggregations, semi-/anti-joins, outer joins, and window (AKA analytical) functions. We expect that cost-based optimization [33][62] would choose good derivatives in the face of these ambiguities and plan to address that challenge in future work.

3.2.2 Change Tracking. Query differentiation eventually pushes the derivative operator down to the scan operators of the query plan. At this point, we need a way to compute the changes that occurred on a persistent table. In Snowflake, this functionality is supported by per-row metadata stored in hidden *change tracking* columns. Change tracking columns make it easy to determine two important properties for each row:

[3]That is, $(f \circ g)' = (f' \circ g) \cdot g'$.

(1) A unique identifier for the row, which is stable across updates. This identifier can be used to consolidate redundancies for min-delta changes.
(2) Whether the row was inserted into the current micro-partition or was copied in from another. This information makes it trivial to produce append-only changes.

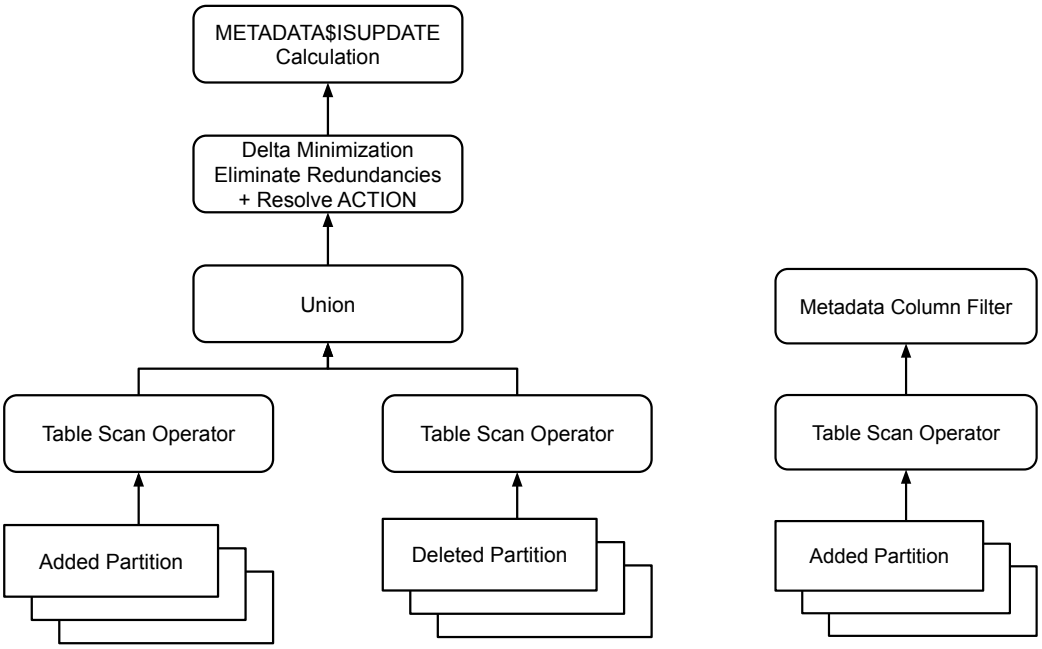
Change tracking columns represent each row's identity as its location when it was first inserted into the table. The columns are `NULL` when a row is first inserted into a table. When a copy-on-write operation takes place, some rows are moved from their previous location to a new location. So, during copies-on-write, the original location of each row is written to the new micro-partition.

Crucially, change tracking columns have very low overhead, both in processing and storage cost. For processing cost, `INSERT` DMLs have zero penalty. `UPDATE` and `DELETE` operations cost marginally more, as they must write and compress these columns. Fortunately, the location information is highly redundant, making them extremely compressible. The resulting processing cost is negligible in all but pathological cases. For storage cost, change tracking columns are extremely compact. Most rows are `NULL`, which consume effectively zero storage. Due to the aforementioned compressibility, rows which have been copied-on-write consume very little storage. Consequently, change tracking has no material impact on existing use cases, which enables its use on all Snowflake tables.

3.2.3 Change Formats. As described in Section 2.1, Snowflake change queries support two change formats: min-delta and append-only. For the **min-delta** format, the key implementation challenge is eliminating redundancies — that is, pairs of changes that cancel out. Many operations can introduce redundancies. For example, Listing 8 shows how excluding columns can make updates redundant. As a more involved example, the plan which produces delta changes on persistent table scans, shown in Figure 1a as the sub-plan below the union, produces a delta with redundancies. It works by iterating over the table versions during the change interval to find all micro-partitions that were added to or removed from a table during the change interval. These micro-partitions are scanned, with rows in added partitions given `INSERT` as their `ACTION` and rows in removed partitions given `DELETE`. Due to the copy-on-write semantics of Snowflake DMLs, many rows can appear in both added and removed partitions, creating redundancies. To account for this fact, our query differentiator has a **redundant-delta** change format to represent deltas that need to be minimized. The query rewriter considers the change format when determining which rule to apply. Any rule which can introduce redundancies only matches the redundant-delta change format. We have an additional rule which converts a min-delta into a redundant-delta when needed. Treating redundant- and minimum-deltas separately makes it easy to track when a consolidation operation is needed.

Delta minimization is a costly operation that requires repartitioning its input. But the majority of change queries only have inserts or deletes during their change interval, in which case no minimization is necessary. To take advantage of this fact, we implemented an optimization which elides the minimization in such cases. This gives rise to several plan shapes: the `MINIMIZE` shape is the default, and the `ADDED_ONLY` and `REMOVED_ONLY` shapes omit delta minimization and one branch of the union all for each base table. As discussed in Section 4.2, this confers substantial performance gains.

Append-only changes are easier to compute. Similarly to delta changes, we iterate over the table versions during the change interval. But this time, we only find added micro-partitions, and only rows with `NULL` change tracking columns are selected, as shown in Figure 1b. The key challenge to append-only changes is ensuring that the performance of the resulting derivative matches users' expectations. The raison d'être for append-only changes is their performance advantage over min-delta changes. Unfortunately, as we mentioned in Section 2.2.1, it is expensive to calculate append-only changes over non-monotonic queries. The reason for this is intuitive: when a row is



(a) Minimum-Delta (b) Append-Only

Fig. 1. Query Plans

inserted into the input of a non-monotonic operator, that change may result in a deletion from its output. Conversely, a deletion from the input may result in an insertion into its output. In general, this means that computing all the inserts in the output of such a query requires determining the consequences of both inserts and deletes in its input. That's the same problem as computing delta or redo changes, which means we've lost the performance benefit. So, to provide consistent performance, we only support append-only changes over monotonic queries.

3.2.4 Metadata Columns. Recall that Snowflake change queries produce three metadata columns.

`$ACTION` denotes whether a change is an `INSERT` or a `DELETE`. It is therefore defined by the algebraic equivalences described above.

`$ISUPDATE` only needs to be computed for min-delta changes (it's `false` for append-only). As discussed in Section 2.2.2, various operations can invalidate whether or not a change is an update. So, we defer computing this column as the last operation atop a min-delta result (see Figure 1a). To do this, we make use of distinction between min-delta and redundant-delta mentioned above. The rule that rewrites the operator to compute `ISUPDATE` only matches min-delta derivatives, ensuring the operation happens in the right place.

When implementing **`$ROW_ID`**, we encountered a number of trade-offs in their design. First is how to format the ID: should the ID contain meaningful information or be opaque? Meaningful IDs risk creating an unintended API contract, which restricts evolvability of the system. But opaque IDs tend to have poor locality, which can lead to worse performance when pruning or shuffling. We chose to make our IDs opaque to give our system more flexibility. Our row IDs are a cryptographic hash of the change tracking columns, which ensures uniqueness to a very high probability.

The second significant trade-off is how to propagate IDs through various operators. For select and project, we can simply leave the IDs unchanged. For joins, we chose to concatenate the IDs

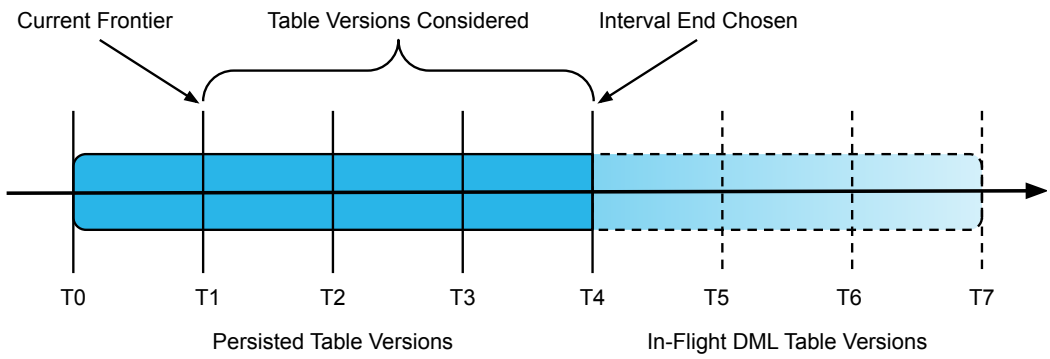


Fig. 2. Stream Interval Selection

of the matched rows and hash the resulting string. For group by, which produces at most one row per key, we hash the concatenation of the keys. Scalar aggregations produce a constant hash. Guaranteeing the uniqueness of IDs out of union-all was surprisingly challenging. For example, consider a union-all of two different filters on the same table. To guarantee uniqueness, one must either prove that the filters are mutually exclusive or tag the IDs with a branch of the union. A similar situation occurs when considering unions of joins, or unions of joins of unions, etc. Thankfully, we have not found any real use cases that depend on these edge cases, allowing us to sidestep the issue. Accordingly, our implementation detects and rejects requests for changes over queries with possibly-overlapping ID domains. In each of the above decisions, we chose uniformity, compactness, and opaqueness of the IDs at the expense of potential performance. In practice, this performance impact has not been an issue because most customer data has primary keys, and these can be used in place of our row IDs.

3.3 Stream Transactions

STREAMs offer a transactional interface for consuming changes exactly-once. As described in Section 2.3, a STREAM's position in history is represented by its frontier. When a STREAM is queried, it returns the changes over the interval from the frontier to the present. A new frontier is computed that guarantees all and only currently-committed table versions are included. This means any ongoing transactions will be shown in future STREAM reads, and past DML operations will never be shown again once consumed. Figure 2 diagrams this selection for a single table.

Our implementation of transactional consumption has to deal with several technical complications. The first challenge is that versions are totally ordered per-table, but may commit out of order across tables. Consequently, it is impossible to represent the frontier of a STREAM with multiple base tables as a single timestamp. Instead, a frontier tracks the table version of each base table separately, ensuring no versions are accidentally skipped or consumed twice.

However, we wish to maintain the abstraction that a STREAM represents a point in the past. This is useful for 2 use cases: time traveling to a STREAM 's frontier and creating a new STREAM at the current position of another STREAM. To support this functionality, we also store a *last updated* timestamp inside each frontier, representing the time AS OF when the STREAM was consumed. When using a frontier as a point in time, we automatically substitute this last updated timestamp. Note that this timestamp does not exactly correspond to the frontier because versions take a short time to propagate throughout a cluster, but the snapshot isolation it provides suffices in practice.

Another challenge is correctly handling concurrent consumption of the same STREAM. STREAM consumption is serializable in order to provide intuitive semantics. Using a combination of optimistic

Statement type	Append-only STREAM	Min-delta STREAM
INSERT	54.5%	15.0%
MERGE	29.4%	63.2%
DELETE	0.4%	0.4%
UPDATE	0.0%	0.3%
CREATE TABLE AS	2.1%	11.3%
SELECT	13.6%	9.9%

Table 1. Distribution of statement types referencing Streams.

and pessimistic concurrency control, depending on whether the STREAM is consumed in a single-statement or multi-statement transaction respectively, Snowflake ensures that only one consumer can advance the STREAM at a time.

Finally, the challenge of staleness (discussed in Section 2.3) is handled by automatic retention extension. While conceptually simple, it is in practice more challenging, requiring for each table the determination of the oldest frontier not older than the maximum extension. This information is then incorporated into our background data expiration process, which delays expiring data until this oldest frontier is consumed or exceeds the maximum.

Without STREAMs, users would have to understand and solve each of these complications. By encapsulating them, Snowflake is able to simplify the process of transactionally consuming incremental changes. Implementing that encapsulation requires making invasive changes throughout the system, e.g. in the transaction processing engine, version resolution algorithm, and background expiration process, which is facilitated by Snowflake's integrated, single-system approach.

4 USAGE AND PERFORMANCE ANALYSIS

STREAMs and the CHANGES clause have been a part of Snowflake's offering for more than three years. Since their release, their usage steadily increased and they have become an indispensable building block for many of our customers to build data pipelines. At Snowflake, we collect logs of our systems and have access to catalog metadata and detailed query statistics to be able to debug customer issues, analyze the usage of features, and make data-driven decisions. Based on these logs, we investigated how customers use change queries and how performance varies across that usage. We report our findings in this section.

4.1 Usage of Streams at Snowflake

Many of our customers are heavily using STREAMs, and their popularity is steadily growing. At the time of writing this paper, 48% of all actively used STREAMs were append-only STREAMs and 52% were min-delta STREAMs. Although their semantics narrow their applicability, append-only STREAMs are frequently used due to their better and more predictable performance (see Section 4.2 for details).

STREAMs can be used in queries and DMLs like regular tables. Table 1 shows the distribution of statement types for statements that reference a STREAM. Append-only STREAMs are mostly queried in INSERT statements, which is not surprising given that they only emit rows that were inserted into their base relation. Almost $^1/_3$ of all statements referencing append-only STREAMs are MERGE statements. There are different scenarios in which consuming an append-only STREAM with a MERGE statement is useful. One of them is to update a table with a primary key with UPSERT semantics, i.e., insert a row if its key is missing and update the row otherwise. If the STREAM's base table is a

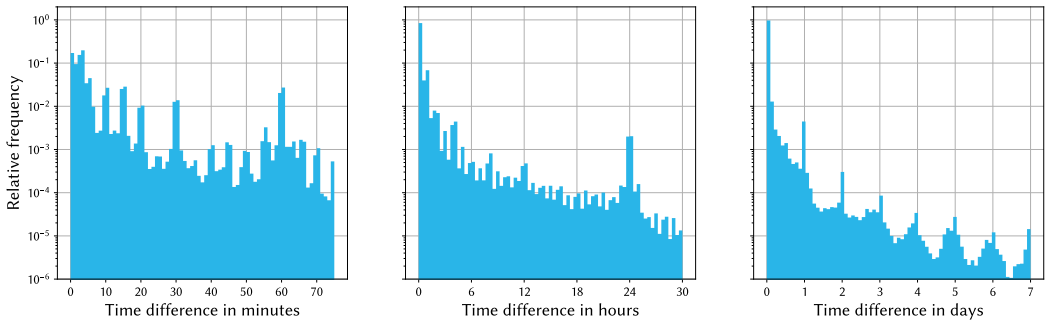


Fig. 3. Distribution of duration between two consecutive STREAM reads.

persisted changelog, it can be consistently consumed using the append-only STREAM and applied to a sink table via MERGE.

Almost $^2/_3$ of all statements that read a min-delta STREAM are MERGE statements. Since MERGE is the only statement type that can insert, update, and delete rows, and min-delta STREAMs return all types of changes on a base table, this is not unexpected. INSERT and CREATE TABLE AS (CTAS) statements are commonly used to (temporarily) persist the output of a min-delta STREAM. This can be useful if the STREAM would need to be read multiple times, which can be an expensive operation (see Section 4.2 for details). Once persisted, the min-delta STREAM's result can be consumed multiple times using more efficient append-only STREAMs. For both types of STREAMs we see around 10% of reads by SELECT queries, which do not advance the STREAM frontier. These may simply be ad-hoc queries issued by users.

DML statements that consume STREAMs are typically invoked from TASKs. A Snowflake TASK is a database object that periodically executes statements. It is triggered either by a CRON-like schedule or by the completion of another TASK [58]. As of now, the shortest possible interval to trigger a TASK is one minute. Snowflake's elastic virtual warehouses, which can be suspended and resumed quickly, ensure pipelines only incur costs when actively processing data.

Figure 3 shows the distribution of the duration of time intervals between two consecutive reads of the same STREAM. The figure is split into three charts to visualize shorter intervals with higher resolution. The left chart shows intervals shorter than one hour. The chart in the middle shows intervals up to one day. The right chart shows intervals up to one week. We can see that many STREAMs are very frequently read, i.e., in intervals of ten minutes or less. We can also identify clusters around multiples of five minutes, ten minutes, one hour, and one day. The patterns suggests that most STREAM reads are triggered by periodic TASKs. Note that a TASK can be configured to skip executions if no changes were applied to a STREAM's base tables. This behavior explains why Figure 3 also shows clusters at multiples of common TASK intervals.

From Figure 3 we can see that Snowflake's STREAMs are used for a large segment of the latency spectrum, ranging from one minute to multiple days. Users choose the interval at which to consume a STREAM depending on their business requirements for data freshness, their budget, and the frequency and size of the updates on the STREAM's source relation. Less frequent STREAM consumption results in fewer, more resource-intensive queries, but overall less cost due to the amortization of overheads.

4.2 Analyzing Change Query Performance

The execution time of a STREAM query is heavily affected by the numbers of added and removed partitions that are scanned. These numbers depend on the frequency of DML statements that are

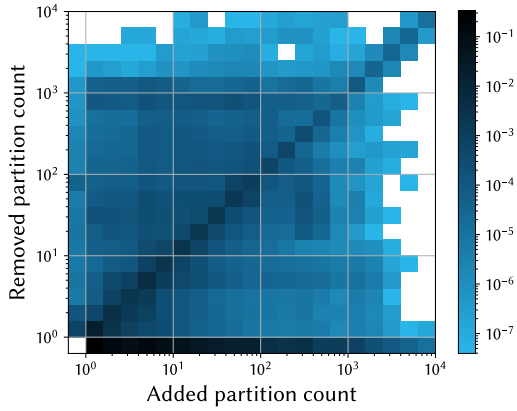


Fig. 4. Normalized number of min-delta STREAM queries per count of added and removed partitions.

Plan shape	Frequency
ADDED_ONLY	90.8%
MINIMIZE	7.1%
INITIAL_ROWS	2.0%
REMOVED_ONLY	0.1%

Table 2. Distribution of plans shapes for min-delta STREAMs.

applied on a STREAM's base table(s), the number of rows that are inserted into, updated in, and deleted from these tables, as well as the interval at which the STREAM is consumed.

Figure 4 shows the distribution of min-delta STREAM queries with respect to the numbers of added and removed partitions they scanned. Note that the x-axis, y-axis, and color-bar all have a logarithmic scale. The plot shows that a large majority of min-delta STREAM queries do not scan any deleted partitions, and most queries scan 100 or fewer added partitions. However, there are also STREAM queries that read up to 10,000 added and 10,000 removed partitions. The correlation between the number of added and removed partitions, which manifests in the figure as a dark diagonal line, is a consequence of Snowflake's copy-on-write DML mechanism, which often adds and removes partitions in similar quantities.

As discussed in Section 3.2.3, min-delta STREAMs are computed using different plan shapes depending on whether the changes interval contains added and/or removed partitions. Append-only STREAMs are always computed using the same ADDED_ONLY plan.

Table 2 shows the distribution of the plan shapes for the queries visualized in Figure 4. The queries with no deleted partitions, represented by the dark line along the bottom of the figure, make up 90.8% of all min-delta queries and are translated into efficient ADDED_ONLY plans. STREAM queries that have at least one added and one removed partition are compiled into more expensive MINIMIZE plans (7.1%). STREAM queries that only read deleted partitions are very rare (0.1%) and are executed with a REMOVED_ONLY plan that has the same performance characteristics as the ADDED_ONLY plan.

The INITIAL_ROWS plan that is used for 2% of min-delta STREAM queries is a special case. As described in Section 2.3, STREAMs can be created with a property to return the full content of their base table until their first consumption. A query on such a STREAM is translated into an INITIAL_ROWS plan, which is a time-travel query that reads the full content of the STREAM's source

		Normalized exec time					Plan
#part. rem.	**1000**	326.8x	324.7x	350.3x	578.7x	3628.0x	MINIMIZE
	100	39.1x	45.6x	72.1x	347.7x	3064.3x	MINIMIZE
	10	28.7x	34.0x	43.1x	105.9x	778.6x	MINIMIZE
	1	12.9x	14.3x	23.0x	87.3x	758.7x	MINIMIZE
	0	1.0x	1.6x	2.1x	5.3x	44.9x	ADDED_ONLY
		1	**10**	**100**	**1000**	**10000**	
		#partitions added					

Table 3. Normalized execution time for min-delta STREAM queries per count of added and removed partitions.

Exec Time	500ms	1s	10s	30s	1m	10m
Faster queries	47.8%	63.9%	87.5%	93.5%	96.1%	99.7%

Table 4. Distribution of execution times for STREAM queries.

table and returns all rows as inserts. This feature is typically used to bootstrap a sink table at the beginning of a continuous ingestion process.

In order to evaluate the performance of the different plans, we created a table and a min-delta STREAM on it, and then ran a workload of DML statements on the table and SELECT queries on the STREAM such that we precisely controlled the number of added and removed partitions scanned by the STREAM queries. Table 3 shows the normalized execution times of STREAM queries run with fixed compute resources for different numbers of added and removed partitions. As we can see from the sub-linear scaling behavior for STREAM queries with up to 100 scanned partitions, the compute resources were not fully utilized. However, the execution time noticeably increases when 1000 or more partitions are read.

Moreover, there is a 9x to 16x performance difference between STREAM queries that read zero and one deleted partition. This difference must be accounted to the different plans being used, i.e., the ADDED_ONLY plan and the MINIMIZE plan. The ADDED_ONLY plan is used for all append-only CHANGES queries and for all min-delta CHANGES queries without removed partitions in the changes interval. As soon as the changes interval of a min-delta CHANGES query includes at least one added and one removed partition, the query is executed with a less efficient MINIMIZE plan. Note that the reported performance numbers can be easily matched to the heatmap plot in Figure 4 that shows the frequency of min-delta STREAM reads for varying numbers of added and removed partitions.

We also looked at the execution time of STREAM queries that are executed in production workloads. Table 4 shows the distribution of execution time for a sample of STREAM queries executed by Snowflake users for all change formats and statement types. We see that almost 50% of all queries complete within 500 milliseconds, most queries finish within one minute, and the vast majority (99.7%) in 10 or less minutes. While there is also a long tail of queries that take significantly longer to execute (max observed execution time was 34 hours), we want to emphasize that the reported execution times not only include the time to read the STREAMs but also any additional operations specified by the user queries such as DMLs or joins. Overall, this data shows that STREAMs are efficiently powering the data pipelines of our users down into relatively low latencies.

To summarize, append-only STREAMs serve the common use case of extracting all rows added to a table, at very low costs. While min-delta STREAMs can be more expensive, they are an indispensable

feature to extract all changes that were applied on a table. Our analysis showed that most min-delta STREAM queries (90%) are cheap to compute; queries that need to scan many added and removed partitions are much less common. Snowflake users can trade between freshness and cost by varying the frequency at which STREAMs are queried. They do so across a broad spectrum, suggesting that pipeline technologies should work across this entire range.

5 RELATED WORK

5.1 Early Academic Approaches

In the 2000s there was a first big push in academia towards defining stream processing logic declaratively. Cugola and Margara survey and discuss the approaches from this decade in great detail in [25], as an attempt to unite the two research communities which developed them: on the one hand, the database community that proposed Data Stream Management Systems (DSMSs) such as Aurora [1] and STREAM [17] (with CQL [18]) which enable users to define static queries on non-static data (in contrast to DBMSs which support issuing ad-hoc queries on static data snapshots); on the other hand, the event-based systems community which proposed Complex Event Processing (CEP) systems such as Amit [2], Cayuga [31], and T-REX [26] (with TESLA [24]). These systems interpret the input data as a stream of basic events and enable the user to define logic to derive more high-level events from the input events.

All these approaches present interesting concepts for defining stream processing logic declaratively. However, CQL is unique in that it regards streams and tables separately and proposes a set of conversion operators between them.

5.2 Stream Processing Frameworks

In the 2010s, a multitude of open-source stream processing frameworks emerged and made stream processing accessible to a wider audience. Prominent representatives are Storm [60], Spark Streaming [64], Flink [22], Samza [50], Beam [21], and Kafka Streams [61], all projects governed by the Apache Software Foundation. Originating from the Hadoop ecosystem [10], the initial programming models for all of these systems required users to manually assemble data flow graphs from user-defined operators that processed individual stream elements or small batches of elements.

Many of these projects later added declarative APIs to define stream processing logic. Spark introduced Structured Streaming [19] which is built on top of Spark SQL. Confluent published KSQL [37], a wrapper around Kafka Streams that enables users to express logic in a SQL dialect. Storm, Flink, Samza, and Beam added SQL parsers and compilers based on Calcite [21] to translate queries written in their SQL dialects into data flows [8, 9, 15, 16].

Members of some of these communities published an approach to define SQL queries over streaming and static data with common semantics [20]. Snowflake's change queries are effectively a practical manifestation of the EMIT STREAM clause proposed therein.

These frameworks largely preferred low-latency and scale over ease-of-use and efficiency. In contrast, Snowflake's current approach targets higher latencies (~1 minute), ease of use, and resource efficiency. This facilitates adoption by users of batch systems who are accustomed to using DML statements over large batches of data, but leaves near-real-time use cases unaddressed.

5.3 Database Management Systems

5.3.1 Query Support. There are contemporary DBMSs which, like Snowflake, treat time-travel and CDC as first-class citizens by enabling users to query past versions or changes over time.

Many DBMSs support temporal tables as standardized in SQL:2011 [39], including Oracle Database [52], Microsoft SQL Server [47], IBM DB2 [35], SAP HANA [54], and MariaDB [40]. This SQL

extension adds the `AS OF...` clause and the `SYSTEM_TIME BETWEEN...AND...` clause[4] to retrieve all row versions that were valid at a certain point in time or valid within a time interval, respectively. In addition, support for application-time period tables allows the same in terms of user-specified columns. Although the `SYSTEM_TIME BETWEEN` functionality is similar to the change queries proposed in this paper, it has a key difference: only the upper bound of a row's validity interval changes when the row is deleted, rather than returning an explicit delete record. Thus, the rows resulting from a `SYSTEM_TIME BETWEEN` query do not comprise a changelog, which grows monotonically without updates to preceding rows. As a result, the stream processing idioms which build upon changelogs are challenging to support using temporal extensions. Furthermore, to our knowledge, none of these products feature a database object like Snowflake's `STREAM` to keep track of already read changes across client sessions.

Materialize is a streaming database built on top of differential dataflow [42, 49]. It supports retrieving the changelog of a database object using the `SUBSCRIBE` statement [41]. Although, a `SUBSCRIBE` cursor looks similar to a Snowflake `STREAM` there are differences: a `SUBSCRIBE` cursor is a client object which can be used to periodically fetch new changes from the database system. In contrast, a Snowflake `STREAM` is a database object itself and can thus be used across multiple client sessions. Moreover, Materialize does not support retrieving a past version of a database object or retrieving changes for a past time interval from within a query expression.

Google BigQuery recently launched a preview version of change history support, implemented via an `APPENDS` table-valued function [32]. `APPENDS` operates very similarly to Snowflake's append-only `CHANGES` queries, accepting a table and optional start and end timestamps, and returning all `INSERT`s that occurred within the table during that time interval, limited to the time travel window. There is currently no support for extracting `UPDATE` and `DELETE` records, and offset management must be handled manually due to the lack of a `STREAM` object analog.

5.3.2 Exported Changes. There are several DBMSs that do not support querying changes, but still expose changes via log files or other external mechanisms. Consuming and transforming these changes is deferred to a downstream processor or data warehouse.

Amazon's DynamoDB supports publishing all changes (i.e., inserts, updates, and deletes) on a table in transaction order into a DynamoDB Stream [7] for the past 24 hours. These changes can be read using the Amazon Kinesis [6] Client Library.

Azure Cosmos DB [46] maintains per container a persisted change feed which records for each document the most recent change (insert or update) as long as it is not deleted.

PostgreSQL supports Write-Ahead Logging (WAL) [53], MySQL supports a Binary Log [51], and MongoDB supports exposing an operations log [48]. All three can be read by various tools such as Meroxa [43–45] and Debezium [28–30].

5.4 Table Formats

Today, structured data is often stored in immutable files on cloud storage using table formats such as Delta Lake, Apache Iceberg, and Apache Hudi. Similar to Snowflake, these formats support reading past versions of a table [12, 13, 63]. Moreover, they have varying support for returning the changes that were applied on a table.

Delta Lake change data feed [27] gives access to changes by persisting all row-level changes, except for those which can be represented as partition-level changes like full-partition additions and deletes. In contrast, Snowflake maintains change tracking metadata alongside the actual table data, which adds negligible storage and processing overhead.

[4]In some systems the syntax is `VERSIONS BETWEEN`

Iceberg distinguishes between append, override, and delete file changes [14], which is sufficient for engines such as Spark to extract append-only-like changes from Iceberg tables. This is similar to Snowflake's append-only change format.

Hudi follows a similar approach as Snowflake of maintaining row-level metadata in system columns to enable the extraction of all types of row changes [11, 23]. Using these columns, data processing engines can retrieve changes [12].

6 FUTURE WORK

Beyond relatively obvious future directions, such as supporting more differentiable operators, improving performance, and implementing redo logs and commit timestamps, there are a few avenues we are exploring that are worth mentioning.

Dynamic Tables: As mentioned in Section 1, incremental view maintenance is one of the key use cases for change queries. We have built our own general IVM feature, called *Dynamic Tables*, using change queries as the underlying basis. Although change queries solve a key piece of the problem, there are still a number of interesting challenges around scheduling, transaction isolation, query evolution, and the surrounding development experience.

Cost-based Optimization: Incremental queries can be computed using different execution plans just like regular queries. Cost-based optimization is a common approach for choosing a plan to execute from a set of semantically equivalent plans, and as a technique has been intensively studied by the database research community. In the context of incremental computation, cost-based optimization faces a few new challenges but also opens up some new opportunities, a couple of which we enumerate here.

Firstly, incremental queries are typically static and repeatedly executed while the data they process may vary significantly depending on the actual changes that are applied to the queries' base tables. Periodic schedules (day-night, workday-weekend, quarterly report cycles, etc.) often affect the amount and characteristics of the data being processed. Knowledge about such patterns could be used to improve cardinality estimates or to proactively adjust execution plans.

Secondly, some incremental execution strategies are based on state that is persisted between subsequent incremental computations. For different execution strategies and data characteristics, this state differs in structure and size. Once a stateful execution strategy is chosen for an incremental query, its state can typically only be maintained and used for the next execution of the query if it is executed with the same strategy. Changing input data might cause another execution plan to become superior but in order to migrate to the new plan, new state needs to be computed from scratch. Bootstrapping such state can be a non-trivial investment, that a more efficient plan needs to pay off before becoming actually beneficial.

The combination of fixed queries, stateful, incremental execution strategies, and (periodically) changing characteristics of input data calls for the development new cost models and optimization techniques.

Spanning the latency spectrum: Though we began the paper alluding to the longstanding debate of batch vs. streaming, we at Snowflake firmly believe the premise of that discussion to be misguided: the real endgame is not one vs. the other, but instead the seamless blending of the two. Streaming systems that excel at low latency processing quite commonly either fail to deliver on their latency promises at large scale, or else cost such an astronomical amount that they become impractical. When that happens, users fall back to batch systems, giving up many of the semantic gains the streaming systems were built to provide.

Regardless of whether your use case requires processing megabytes within seconds or petabytes within hours, incrementalism remains key to minimizing both latency *and* cost. Having experienced the shortcomings of many existing streaming systems confronted with real world use cases, we

seek to make Snowflake naturally traverse the entire latency spectrum, from days down to seconds, all within a seamless experience that is intuitive and accessible.

Even though most STREAM queries today complete within one second, we've thus far focused primarily on computation models and user experience; this is because ease of use is paramount, and the vast majority of analytical use cases do not require sub-second latencies. With those falling into place, most of our future endeavors at the system level are focused on reaching ~1 second latencies everywhere practicable, as this is latency floor of most top-tier streaming systems today when persistence and consistency features are enabled.

SQL Standards: Lastly, we are participating in an ongoing SQL Standards Expert Group [36], collaborating on a proposal for streaming extensions to the SQL Standard. We believe change queries are integral to making stream processing a first-class citizen in SQL, and have enjoyed collaborating with other vendors to produce a better solution than any individual vendor would have in isolation.

7 SUMMARY

Stream processing is built on incrementalism: processing the sequence of changes to a dataset over time in discrete chunks, rather than processing and reprocessing the entirety of the dataset over and over. Yet despite the database and streaming communities both deriving the idea of table/stream duality, the SQL language of today only natively supports table/stream conversion in one direction — from streams to tables via DML or aggregations. A native mechanism for extracting change streams back out of tables and manipulating them in SQL remains painfully absent. Such functionality is critical for event processing, notification, IVM, ETL, and extraction scenarios. Without it, users must extract those changes into a separate system for processing, at increased complexity and cost.

In this paper, we presented Snowflake's approach to first-class SQL support for incrementalism: CHANGES *queries* and STREAM *objects*. **CHANGES queries** extract the set of changes made to a persistent table or view over an interval of time. Though most change tracking systems have historically presented changes in a fine-grained redo log format, the approach taken in Snowflake affords two change formats: *append-only* and *minimum-delta*.

Append-only changes capture the additive changes to a table or view over time: all of the INSERTs, with none of the UPDATEs or DELETEs. Focusing on INSERT mutations only yields efficiency gains while still serving a number of practical event processing scenarios such as event queueing and ETL.

Minimum-delta changes present the minimal set of INSERT, UPDATE, and DELETE changes to a table or view over a time interval. For most incremental algorithms and CDC scenarios, the minimum-delta provides a more concise alternative to a full fidelity redo log, plus the benefit of not having to deal with multiple updates to the same row. The main use case where redo logs remain a requirement are audit log scenarios.

Our change query infrastructure utilizes storage- and compute-efficient **metadata columns** to track change information at row granularity, enabling change tracking on all tables. Our extensible **query differentiation framework** applies relational equivalences to rewrite change queries into executable plans. So far, we have implemented CHANGES functionality for filters, projections, aggregations, window functions, and inner, outer, semi, and anti joins.

STREAM objects, meanwhile, provide a simple mechanism for transactionally consuming a stream of changes over time. In Snowflake, a STREAM comprises a **frontier** which tracks the STREAM 's progress across its base tables. Frontiers are updated transactionally whenever a STREAM is consumed as part of a DML operation.

Over the course of the last three years, we've seen change query usage continue to grow across a large swath of Snowflake customers. Append-only and minimum-delta formats see roughly equal

usage, suggesting that the two formats provide **meaningful tradeoffs** for a differing set of use cases within the incremental processing domain. Meanwhile, only a few customers have cited the lack of a full fidelity redo log format as a blocking shortcoming.

Additionally, we see broad usage across the latency spectrum. Although the majority of streams are queried in minute intervals with sub-minute runtimes, a substantial number of users run very large change queries with multi-hour durations, lending credence to the idea that there is real value addressing the **full breadth of the latency spectrum**, rather than focusing strictly on low latency scenarios. By generalizing streaming to include both the low *and* high ends of the latency spectrum, we can move past the tired argument of batch vs. streaming and into a future of simpler and easier data processing.

REFERENCES

[1] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. 2003. Aurora: a new model and architecture for data stream management. *The VLDB Journal* 12, 2 (2003), 120–139. https://doi.org/10.1007/s00778-003-0095-z

[2] Asaf Adi and Opher Etzion. 2004. Amit – the situation manager. *The VLDB Journal* 13, 2 (2004), 177–203. https://doi.org/10.1007/s00778-003-0108-y

[3] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

[4] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proceedings of the VLDB Endowment* 6, 11 (aug 2013), 1033–1044. https://doi.org/10.14778/2536222.2536229

[5] Tyler Akidau, Slava Chernyak, and Reuven Lax. 2018. *Streaming systems: the what, where, when, and how of large-scale data processing.* O'Reilly Media, Inc.

[6] Amazon Web Services, Inc. 2022. *Amazon Kinsesis.* https://aws.amazon.com/kinesis/ (accessed: November 2022).

[7] Amazon Web Services, Inc. 2022. *Change data capture for DynamoDB Streams.* https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html (accessed: November 2022).

[8] Apache Beam. 2022. *Beam SQL overview.* https://beam.apache.org/documentation/dsls/sql/overview/ (accessed: November 2022).

[9] Apache Flink. 2022. *Flink – SQL.* https://nightlies.apache.org/flink/flink-docs-release-1.16/docs/dev/table/sql/overview/ (accessed: November 2022).

[10] Apache Hadoop. 2022. *Hadoop.* https://hadoop.apache.org (accessed: November 2022).

[11] Apache Hudi. 2022. *Apache Hudi Technical Specification – Data Model.* https://hudi.apache.org/tech-specs/#data-model (accessed: November 2022).

[12] Apache Hudi. 2022. *Hudi – Spark Guide.* https://hudi.apache.org/docs/quick-start-guide/ (accessed: November 2022).

[13] Apache Iceberg. 2022. *Iceberg – Spark Queries.* https://iceberg.apache.org/docs/1.0.0/spark-queries/ (accessed: November 2022).

[14] Apache Iceberg. 2022. *Iceberg Java API – Update Operations.* https://iceberg.apache.org/docs/1.0.0/api/#update-operations (accessed: November 2022).

[15] Apache Samza. 2022. *Samza SQL.* https://samza.apache.org/learn/documentation/1.6.0/api/samza-sql.html (accessed: November 2022).

[16] Apache Storm. 2022. *Storm SQL integration.* https://storm.apache.org/releases/2.4.0/storm-sql.html (accessed: November 2022).

[17] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. 2003. STREAM: The Stanford Stream Data Manager. In *Proceedings of the 2003 International Conference on Management of Data, SIGMOD 2003, San Diego, California, USA, June 9-12, 2003*, Alon Y. Halevy, Zachary G. Ives, and AnHai Doan (Eds.). ACM, 665. https://doi.org/10.1145/872757.872854

[18] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 15, 2 (2006), 121–142. https://doi.org/10.1007/s00778-004-0147-z

[19] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 601–613. https://doi.org/10.1145/3183713.3190664

[20] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth L. Knowles. 2019. One SQL to Rule Them All - an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables. In

Proceedings of the 2019 International Conference on Management of Data, SIGMOD 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1757–1772. https://doi.org/10.1145/3299869.3314040

[21] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 221–230. https://doi.org/10.1145/3183713.3190662

[22] 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 Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

[23] Vinoth Chandar. 2020. *Apache Hudi – Design And Architecture.* https://cwiki.apache.org/confluence/display/HUDI/Design+And+Architecture (accessed: November 2022).

[24] Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010*, Jean Bacon, Peter R. Pietzuch, Joe Sventek, and Ugur Çetintemel (Eds.). ACM, 50–61. https://doi.org/10.1145/1827418.1827427

[25] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *Comput. Surveys* 44, 3 (2012), 15:1–15:62. https://doi.org/10.1145/2187671.2187677

[26] Gianpaolo Cugola and Alessandro Margara. 2013. Deployment strategies for distributed complex event processing. *Computing* 95, 2 (2013), 129–156. https://doi.org/10.1007/s00607-012-0217-9

[27] Databricks, Inc. 2022. *Use Delta Lake change data feed on Databricks.* https://docs.databricks.com/delta/delta-change-data-feed.html (accessed: November 2022).

[28] Debezium Community. 2022. *Debezium connector for MongoDB.* https://debezium.io/documentation/reference/2.0/connectors/mongodb.html (accessed: November 2022).

[29] Debezium Community. 2022. *Debezium connector for MySQL.* https://debezium.io/documentation/reference/2.0/connectors/mysql.html (accessed: November 2022).

[30] Debezium Community. 2022. *Debezium connector for PostgreSQL.* https://debezium.io/documentation/reference/2.0/connectors/postgresql.html (accessed: November 2022).

[31] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. 2007. Cayuga: A General Purpose Event Monitoring System. In *Proceedings of the Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007.* www.cidrdb.org, 412–422. http://cidrdb.org/cidr2007/papers/cidr07p47.pdf

[32] Google LLC. 2022. *Work with change history.* https://cloud.google.com/bigquery/docs/change-history (accessed: November 2022).

[33] Goetz Graefe. 1995. The cascades framework for query optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[34] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. 1993. Maintaining views incrementally. *ACM SIGMOD Record* 22, 2 (1993), 157–166. https://doi.org/10.1145/170035.170066

[35] IBM. 2022. *Temporal tables and data versioning.* https://www.ibm.com/docs/en/db2-for-zos/11?topic=tables-temporal-data-versioning (accessed: November 2022).

[36] InterNational Committee for Information Technology Standards. 2022. *DM32.2 Task Group on Database.* http://www.incits.org/committees/dm32 (accessed: November 2022).

[37] Hojjat Jafarpour and Rohan Desai. 2019. KSQL: Streaming SQL Engine for Apache Kafka. In *Proceedings of the 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi (Eds.). OpenProceedings.org, 524–533. https://doi.org/10.5441/002/edbt.2019.48

[38] Jay Kreps. 2016. *Introducing Kafka Streams: Stream Processing Made Simple.* https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/ (accessed: November 2022).

[39] Krishna G. Kulkarni and Jan-Eike Michels. 2012. Temporal features in SQL: 2011. *SIGMOD Rec.* 41, 3 (2012), 34–43. https://doi.org/10.1145/2380776.2380786

[40] MariaDB Foundation. 2022. *MariaDB – Temporal Tables.* https://mariadb.com/kb/en/temporal-tables/ (accessed: November 2022).

[41] Materialize, Inc. 2022. *SUBSCRIBE.* https://materialize.com/docs/sql/subscribe/ (accessed: November 2022).

[42] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *Proceedings of the Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013.* www.cidrdb.org. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf

[43] Meroxa, Inc. 2022. *Meroxa – MongoDB.* https://docs.meroxa.com/platform/resources/mongodb (accessed: November 2022).

[44] Meroxa, Inc. 2022. *Meroxa – MySQL*. https://docs.meroxa.com/platform/resources/mysql/setup/ (accessed: November 2022).
[45] Meroxa, Inc. 2022. *Meroxa – PostgreSQL – Logical Replication*. https://docs.meroxa.com/platform/resources/postgresql/connection-types/logical-replication (accessed: November 2022).
[46] Microsoft, Inc. 2022. *Change feed in Azure Cosmos DB*. https://learn.microsoft.com/en-us/azure/cosmos-db/change-feed (accessed: November 2022).
[47] Microsoft, Inc. 2022. *Temporal tables*. https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables?view=sql-server-ver16 (accessed: November 2022).
[48] MongoDB, Inc. 2022. *Replica Set Oplog*. https://www.mongodb.com/docs/manual/core/replica-set-oplog/ (accessed: November 2022).
[49] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles, SOSP 2013, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 439–455. https://doi.org/10.1145/2517349.2522738
[50] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. 2017. Stateful Scalable Stream Processing at LinkedIn. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1634–1645. https://doi.org/10.14778/3137765.3137770
[51] Oracle. 2022. *MySQL :: MySQL 8.0 Reference Manual :: 5.4.4 The Binary Log*. https://dev.mysql.com/doc/refman/8.0/en/binary-log.html (accessed: November 2022).
[52] Oracle, Inc. 2022. *Using Oracle Flashback Technology*. https://docs.oracle.com/cd/E11882_01/appdev.112/e41502/adfns_flashback.htm#ADFNS1008 (accessed: November 2022).
[53] PostgreSQLs Global Development Group. 2022. *Write-Ahead Logging (WAL)*. https://www.postgresql.org/docs/current/wal-intro.html (accessed: November 2022).
[54] SAP. 2022. *Temporal Tables*. https://help.sap.com/docs/HANA_SERVICE_CF/6a504812672d48ba865f4f4b268a881e/cf3523ab01834f5e84a32164c1fd597a.html?q=temporal (accessed: November 2022).
[55] Matthias J. Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. 2018. Streams and Tables: Two Sides of the Same Coin. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics, BIRTE 2018, Rio de Janeiro, Brazil, August 27, 2018*. Association for Computing Machinery, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/3242153.3242155
[56] Snowflake Computing, Inc. 2022. *CHANGES*. https://docs.snowflake.com/en/sql-reference/constructs/changes.html (accessed: November 2022).
[57] Snowflake Computing, Inc. 2022. *Introduction to Streams*. https://docs.snowflake.com/en/user-guide/streams-intro.html (accessed: November 2022).
[58] Snowflake Computing, Inc. 2022. *Introduction To Tasks*. https://docs.snowflake.com/en/user-guide/tasks-intro.html (accessed: November 2022).
[59] Snowflake Computing, Inc. 2023. *MERGE*. https://docs.snowflake.com/en/sql-reference/sql/merge (accessed: March 2023).
[60] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy. 2014. Storm@twitter. In *Proceedings of the 2014 International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 147–156. https://doi.org/10.1145/2588555.2595641
[61] Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J. Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, Varun Madan, and Jun Rao. 2021. Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka. In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD 2021, Virtual Event, China, June 20-25, 2021*. Association for Computing Machinery, New York, NY, USA, 2602–2613. https://doi.org/10.1145/3448016.3457556
[62] Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, et al. 2020. Tempura: A General Cost Based Optimizer Framework for Incremental Data Processing (Extended Version). *arXiv preprint arXiv:2009.13631* (2020).
[63] Burak Yavuz and Prakash Chockalingam. 2019. *Introducing Delta Time Travel for Large Scale Data Lakes*. https://www.databricks.com/blog/2019/02/04/introducing-delta-time-travel-for-large-scale-data-lakes.html (accessed: November 2022).
[64] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles, SOSP 2013, Farmington, PA, USA, November 3-6, 2013*. Association for Computing Machinery, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737

[65] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD 2021, Virtual Event, China, June 20-25, 2021*. Association for Computing Machinery, New York, NY, USA, 2653–2666. https://doi.org/10.1145/3448016.3457559

Received November 2022; revised February 2023; accepted March 2023