Row Lineage Proposal

Authors: Russell Spitzer, Nileema Shingte , attila-peter.toth@snowflake.com

Pull Request : https://github.com/apache/iceberg/pull/11130

Motivation:

Tracking changes for rows in a table as they are modified is key for a variety of applications like CDC streams and Materialized View maintenance. The current spec makes it very difficult to determine the evolution of a single row over time because we have no way of identifying when the row was added to the table, and we are unable to determine if a row was recently added or if it was just changed. To better support these use cases, we would like to add a marker to every row in an Iceberg table to indicate its origin.

Example Use Cases

- Tracking row modifications over time (CDC)
- Incremental Data Processing
- Audit Logs

Background

This concept is already used in several other systems like

System	Identifier
Microsoft SQL Server	Rowversion - Unique 8 byte Binary Number
Apache HBase	Version - Long Integer (Typically epoch millis)
Apache Cassandra	WRITETIME - Long Integer (Millis)
Delta Lake	Row ID - Globally Unique identifier Row Version - Sequence number of last commit
Postgres	Tableoid, xmin, cmin, xmax, cmax, ctid - Table, transaction, and physical location information

Snowflake (SIGMOD '23, §3.2.2)	Original file name, original row position

Most systems keep the row-versioning information as an engine internal information that is used for consistency, cdc or other purposes. Some of the systems above automatically provide these values for every row while others allow a writer/user to opt in adding these values. Some (Cassandra, Hbase) even allow users to fill in their own values for these columns, overriding engine defaults.

Requirements for tracking Row Lineage

To track a row's lineage we assign an immutable, unique identifier to it that is cheap to store and maintain. A monotonic row version is maintained in tandem with the row identifier indicating the number of times the row has been modified.

Implementation

Table level opt in:

We can define a field within the Table Metadata indicating whether or not writers and rewriters must update row lineage information. Due to the enhanced requirements for writing, this should be disabled by default.

Row-lineage: enabled | disabled

We can track individual rows then with two properties

- **_row_identifier**: uniquely identifies every row in the table.
- **_row_version**: identifies the last modification of the row

These properties are only modifiable by the engine and cannot be written manually by the end user. This means whenever files are added directly into the Table they must be NULL.

Identifier

Global Identifier

Instead of using the Snapshot local identifiers above we could attempt to assign row identifiers globally. **_row_identifier** becomes a global sparse long (we do not guarantee that every value for _row_identifier is used) which we apply to every single row as it is added to the table. New rows in branches follow the same constraints and use the same global numbering scheme. All identifiers are still **NULL** at write time except for the **last-row-id** in the Metadata itself.

Pro:
All identifiers are unique without an additional field
Almost all values are set via inheritance

Con:
More complicated inheritance
Origin Snapshot for a row can only be calculated in conjunction with Snapshot Metadata

Table Metadata Modification

- **last-row-id : Long -** 0 for new tables or the greatest possibly used id from the last committed or staged commit. This value can be calculated as the sum of all **added_rows_count** from all manifests in the commit + the table's previous **last-row-id**.

Snapshot / Manifest List Modification

Snapshots have one additional field which **must** be populated by writers.

- **first_row_id** : Long - The **last-row-id** from table metadata

Manifest Modification

Manifests an identical field but it is always **NULL** when first added

- **first_row_id :** Long - calculated as the **first_row_id** of the snapshot + the **added_rows_count** of all previous newly added data manifests in the manifest list.

Datafiles Modification

Datafiles get a new global identifier fields but they are always **NULL** when newly added

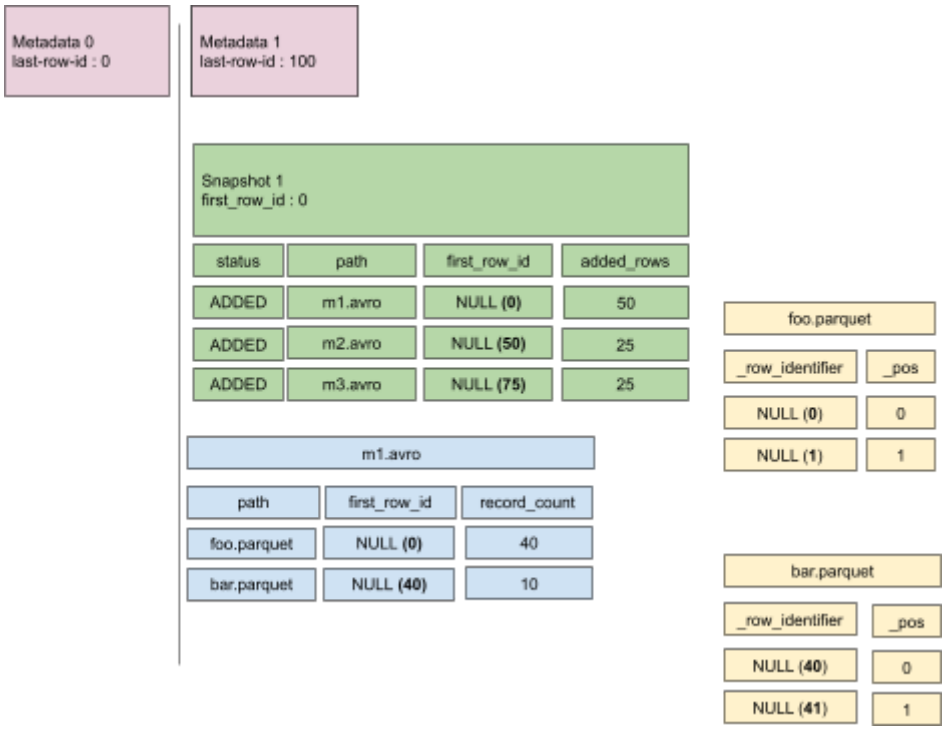
- **first_row_id:** Long - Calculated as the **_first_row_id** of the manifest + **record_count** of all previously listed datafiles with the ADDED state.

Then any particular row has an implicit metadata column which is **NULL** for new rows

_row_identifier: Long - Calculated as **first_row_id** of the datafile + **_pos**

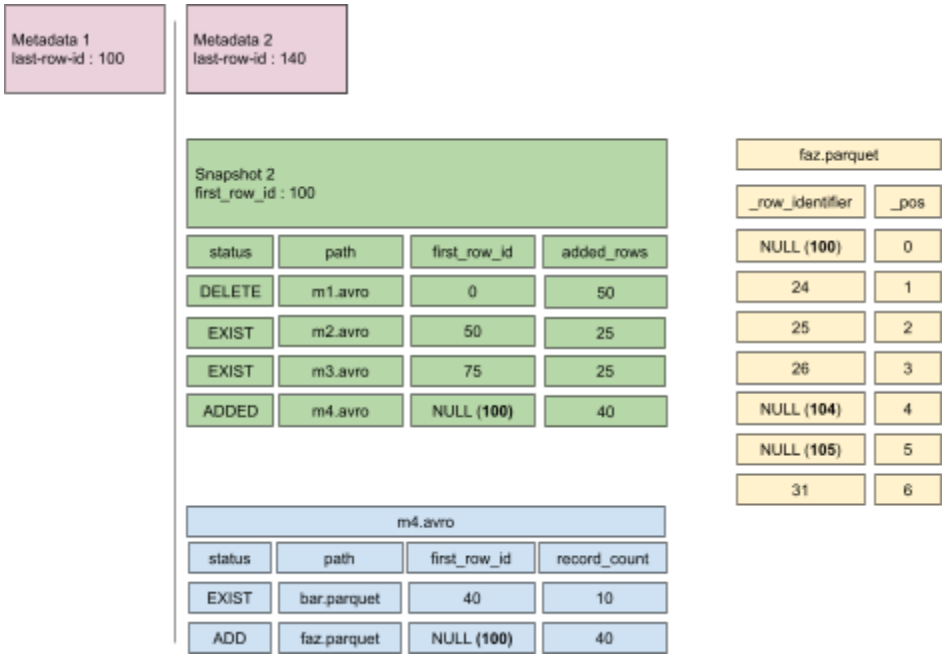
New Write to an Empty Table

(**Bold) Values are Computed**



Result of an operation which modifies an existing table

M1 is replaced with M4 which contains one new file *faz.parquet* which contains some new rows (null _row_identifier) and some existing rows. Although the new **last-row-id** is 140, the actual used ids will be less than this because some rows with *faz.parquet* are being carried over from an earlier data file and have **_row_identifier** alredy assigned.



Row Version Options

Last Updated Sequence Number

We track the **_row_version** by storing the last sequence number which modified the given row. When written the value will be NULL and inherited from the snapshot.

Copying a row without changing it into a new file would materialize the **_row_version** of the original snapshot.

Updating a row would set **_row_version** back to null so that it would inherit the sequence number of the new commit.

Requirements for Engines when Row Lineage is Enabled

Engines **must** fail any write operations (including rewrite operations) when row lineage is enabled if they cannot handle writing or copying row_lineage information.

Engines should have no read behavior changes if row_lineage is enabled.

If a file is read and there is no **_row_identifier** or **_row_version column** present it will be assumed that the file was written without row_lineage on. Any rows copied or modified from such a file should be treated as if they are being created for the first time in their new locations; The columns **_row_identifier** and **_row_version** are added but the value for both is null.

Handling Merge on Read

Engines performing merge on read based modifications can have one of two behaviors. If possible, as in the case with Position Deletes, updates can be treated as in the copy-on-write method and given a new **_row_version** for newly modified rows. This should be possible because detecting the rows to remove with position deletes makes it clear which rows are updates vs which are removed.

Engines using Equality Deletes (or those which cannot track row modification when using position deletes), should treat every new row as being added with no past history. Both **_row_identifier** and **_row_version** should be null for these rows.

Representation of Lineage Fields

An engine can decide what the best way to expose (or if to expose) row lineage fields.

Spark could expose these fields as Metadata Columns which exist on demand within the schema but are not actually writable fields from the table.

Other engines which don't have the Metadata Column option can expose these fields as actual table columns or allow promotion of the metadata_fields to actual table columns as long as they are not writable. This could be accomplished by adding a new column to the table schema which would map to the field id's of the internal row_lineage fields.

Discarded Approaches

Storing Row Lineage Information

- as optional properties, similar to identifier-filed-ids, as JSON list of field-ids that.
    - PRO:
        - Allows multiple implementations for row locator maintenance.
        - If the customer's data naturally satisfies the requirements, then no storage overhead and compute overhead. E.g. a PK in the data can serve as the [Original Row Locator].

        - Turning on/off row lineage truly resets the row tracking, whereas reserved field ids can't be un-populated.
    - CON:
        - Part of the user-visible schema, so present in "*" expansion.
        - If a column is specified to be part of the row-lineage column set, it should not be modifiable via regular DDLs.

In general we want to keep these fields as internal, not allowing users to modify them.

Identifier Options

Snapshot Scoped Identifier

The identifier column is made up of an unique number within the commit coupled with the sequence number of that commit.

Pros:
Inheritance is only required from the Snapshot
Engines not correctly following the spec will not disrupt other engines since each identifier is only dependent on its own work.
Age of any row can be determined without looking at Snapshot Metadata (the original sequence number is embedded in the row)

Cons:
Requires multiple fields for uniqueness
Manifests need an additional field which must be materialized on write

Datafile modifications

When enabled, metadata columns will be added automatically to all rows written to the table. They will not be a part of the table schema directly. Each will be assigned a metadata field ID from our reserved pool in the Iceberg Spec.

1. **_row_identifier:** a unique struct for every row containing
    a. **original_sequence_number**: A numeric representing the sequence number of that file which added the row.
    b. **original_row_position**: A numeric representing the position of the row in the snapshot which created **original_sequence_number**. Every row added in a single snapshot needs a unique value here.

Newly written Datafiles must write null for these fields When null is read, the values would be imputed from the metadata. **original _sequence_number** would come from sequence-number and **original_row_position** from _pos (the purely virtual intra file ordinal) + **first_row_position (**a newly added manifest field).

Whenever a row is copied or modified, the **row_identifier** struct would be materialized into the new file with non-null values. This struct is immutable and can not change for the life of the row.

Manifest Modifications

Manifests entries for datafiles now would include a single additional field

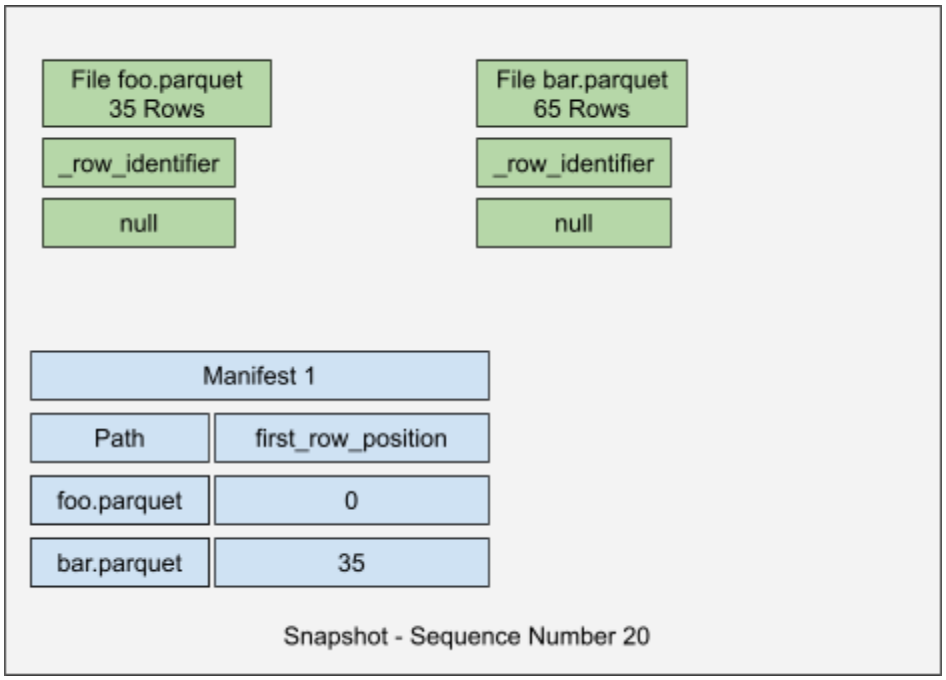
1. **first_row_position**: A numeric representing the snapshot row number of the first row within the file. This orders all datafiles within the snapshot even if they appear in different manifests.

Entries for newly written files will calculate first_row_position based on the number of rows in the files being committed.Optionally an engine can add only the number of null entries of _row_identifier to determine this value but this would requires keeping track of non-null values when reading so this may be less than ideal.

Snapshot Modification

Snapshots should track whether or not row lineage was enabled when they were created. This should allow for faster determination of whether or not a series of snapshots have accurate lineage tracking or not.

Example of a New Write

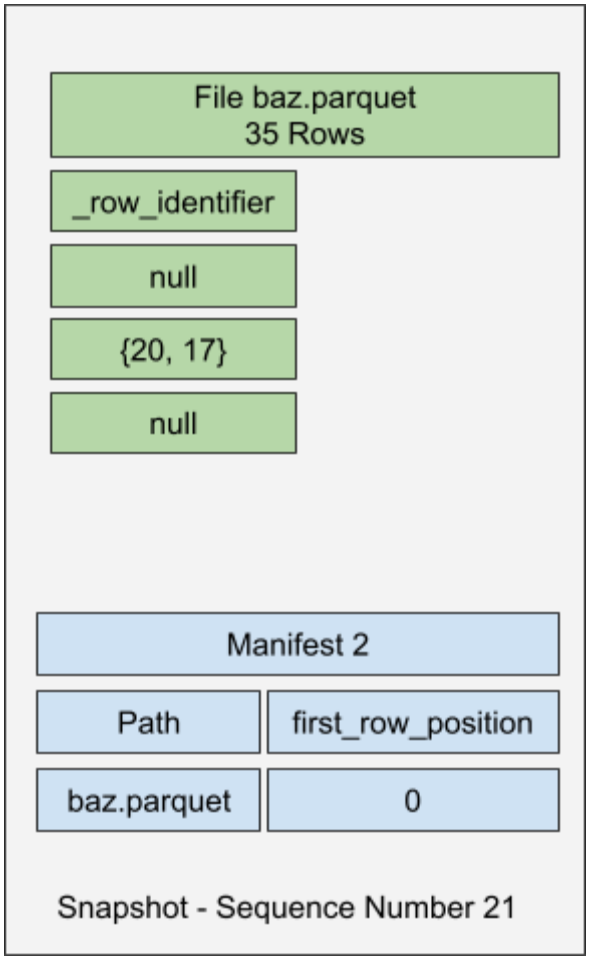


A writer adds two files to the table. Each file contains two newly added columns with all values null. This marks both fields to be imputed from the Manifest. The manifest is written with one additional field **first_row_position.** Because *foo.parquet* is added first, its size is used as the first_row_position for *bar.parquet*. This means the first row of *bar.parquet* would be read as having

1. **_row_identifier**
    - **_original_sequence_number = 20** (from the Snapshot)
    - **_original_row_position = 35** (first_row_position + _pos)

Example of Rewriting a Row

Rewriting a datafile via a row-level operation or compaction would materialize the original_row_position and original_sequence_number into the new datafile while still allowing some rows to optionally be null.



This means the reading the first rows from *baz.parquet* would be

1. **First Row:**
    a. **_row_identifier**
        - **_original_sequence_number = 21** (from the Snapshot)
        - **_original_row_position = 1** (_first_row_position + _pos)
2. **Second Row**
    - **_row_identifier**
        - **_original_sequence_number = 20 (**Existing Row**)**
        - **_original_row_position = 17 (**Existing Row**)**
3. **Third Row**
    - **_row_identifier**
        - **_original_sequence_number = 21**
        - **_original_row_position = 3 (** _first_row_position + _pos**)**

Row Version Options

Origin and Last Updated Sequence Number

We track the **_row_version** by storing both the first sequence number which created the row and the last sequence number to have modified the row. Both fields will be NULL on first write and inherited from the Snapshot.

Copying a row without changing it into a new file would materialize both fields of **_row_version** with the value from the original snapshot.

Updating a row would set **_row_version** back to null so that it would inherit the sequence number of the new commit.

Update Count and Last Updated Sequence Number

We track **_row_version** by storing the number of times the row has been updated as well as the last sequence number modifying the row. The initial **_update_count** would be 0 and the original last updated sequence number would be **NULL** as above.

Copying a row without changing it into a new file would materialize the **_last_updated** field with the value from the original snapshot.

Updating a row would increment **_update_count** and set **_last_updated** back to null so that it would inherit the sequence number of the new commit.

Pro / Con

Option	Pro	Con
Last Update Sequence	Easy to inherit Can quickly determine if row has been changed in this snapshot	Cannot determine if row is an update without additional field/ snapshot metadata
Origin + Last Update	Can quickly determine if a row	More Space Required

	has been changed in this snapshot and whether it was an update or copy without additional metadata	More difficult to implement Full row lineage may be easier to store elsewhere
Update Count + Last Update	Can quickly determine if a row has been changed in this snapshot Can determine how many times a row has been changed over time	More Space Required More difficult to implement