Deletion Vectors High Level Design

Table of Contents

Part I: Design sketch

Motivation

During Delta DML operations, both the semantics of cloud file systems as well as our guarantees about transaction history in Delta prevent us from performing any in-place updates to files. When updates are small compared to the total file size (e.g., a single row per file), this leads to an enormous performance burden of having to rewrite the entire file for a small change (aka Copy-on-write). A large fraction of DML statements that update anything, update a very small % of all the rows in the files they touch. *Deletion Vectors (DVs)* are a mechanism to deal with the case where updates are stored more efficiently, by avoiding the expensive rewrite of the unmodified data.

Requirements

Functional requirements

MUST:

- Must be able to correctly read all Delta tables with DVs.
- Must be able to VACUUM tables with DVs correctly
- Must be able to OPTIMIZE tables with DVs correctly, including compacting the DVs with their associated data files to improve scan performance
- Delta CDF Readers must be able to read CDF-enabled tables with DVs.

SHOULD:

- Should have a mechanism to expose tables with DVs via manually created External Manifests
- Should be able to return a Delta table with DVs into a state where pre-DV reader implementations can read it ("downgrade").

Non-functional requirements

- Writing the deletion vectors should be at least as fast as rewriting the parquet file 99% of the time.
- Writing deletion vectors should not produce more PUT requests to cloud storage than rewriting the parquet file 99% of the time.
- Overhead of reading files with DVs compared to the same file without a DV should be bounded relative to the size of the underlying file (e.g. no more than 2x overhead).
- Number of GET requests to cloud storage for scans should not increase by more than 1 for

each file with a DV, compared to fully compacted files.

Proposal sketch

The solution proposed in this document is to augment the parquet files of a Delta table with separate "Deletion Vector" (DV) files, instead of rewriting them immediately. **A DV is an optimized bitmap that represents a set of rows (of a particular parquet file) that are no longer valid ("deleted") in a particular version of a Delta Table. The n-th row of a Parquet file has row index n, and is associated with the n-th bit of the DV.**

DVs will be created by DML commands that delete or update existing data. As the name implies, DVs are a natural representation for deleted rows. For updated rows, DVs are more of an "invalid" marker for the old content of the row, while the current state is kept in a separate parquet file; usually together with other updates and perhaps even inserts that happened during the same command.

When reading a Delta table at a version that contains DVs, care must be taken to "ignore" (filter out) the invalid rows during scans. As reading files with DVs is going to be somewhat slower than reading a fully compacted file without DVs, the mechanism itself is a tradeoff between write and read performance. When some fraction of invalid rows is reached, it likely becomes beneficial to compact out the latest DV by rewriting the remaining valid rows into a new file, as subsequent accesses (both in read-only and DML operations) will benefit from the reduced scan overhead. Thus DVs are fundamentally a temporary construct that allows us to amortize rewrite costs.

The Life and Death of a Deletion Vector

To give an intuition of how Deletion Vectors would work in Delta, the following illustrates a high-level "happy path" of a DV from initial creation to being eliminated by a completely compacted file.

Step 1 - Initial Creation by a DML Command

Let's say we start a Delta table at version `v1` with 2 files: `file_a.parquet` and `file_b.parquet`. Each file has 1000 rows.
A simplified Delta Log might look something like this:

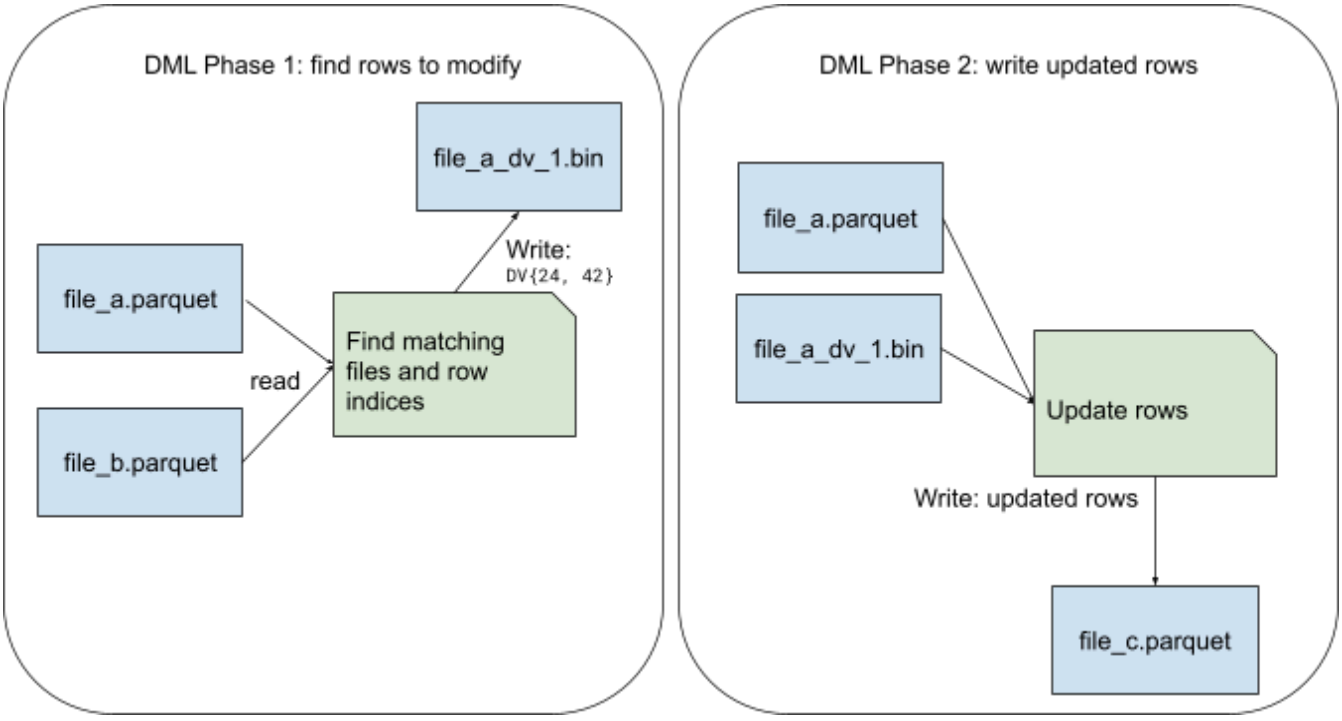
Delta Log Snapshot at Version 1		
Type	**Path**	**DV Path★**
`AddFile`	`file_a.parquet`	`NULL`
`AddFile`	`file_b.parquet`	`NULL`

★ For simplicity, we only include the DV path in the examples. However, depending on the chosen solution, this might be replaced by an inlined DV, or (DV Path, Offset, Length) tuple for storing multiple DVs in a single file. See Protocol Changes.

At this initial state, we then execute a DML command (say UPDATE) that updates 2 rows, both in

`file_a.parquet` at indices 24 and 42. The DML commands work in two phases

1. Find rows to modify: Read the table (after data skipping) to find out which files and specific rows need to be updated. Based on that, generate the deletion vector file.
2. Write update rows: Read the rows to be modified and write out updated rows.



At the end of this command, assuming the transactions succeeds, the new file `file_c.parquet` will be added to the delta log as a new version in which the original entry for `file_a.parquet` is a tombstone and the new entry for the remaining legal rows from it associates the DV `file_a_dv_1.bin` with the parquet file.

COMMIT 2		
Type	**Path**	**DV Path**
RemoveFile	file_a.parquet	NULL
AddFile	file_c.parquet	NULL
AddFile	file_a.parquet	file_a_dv_1.bin

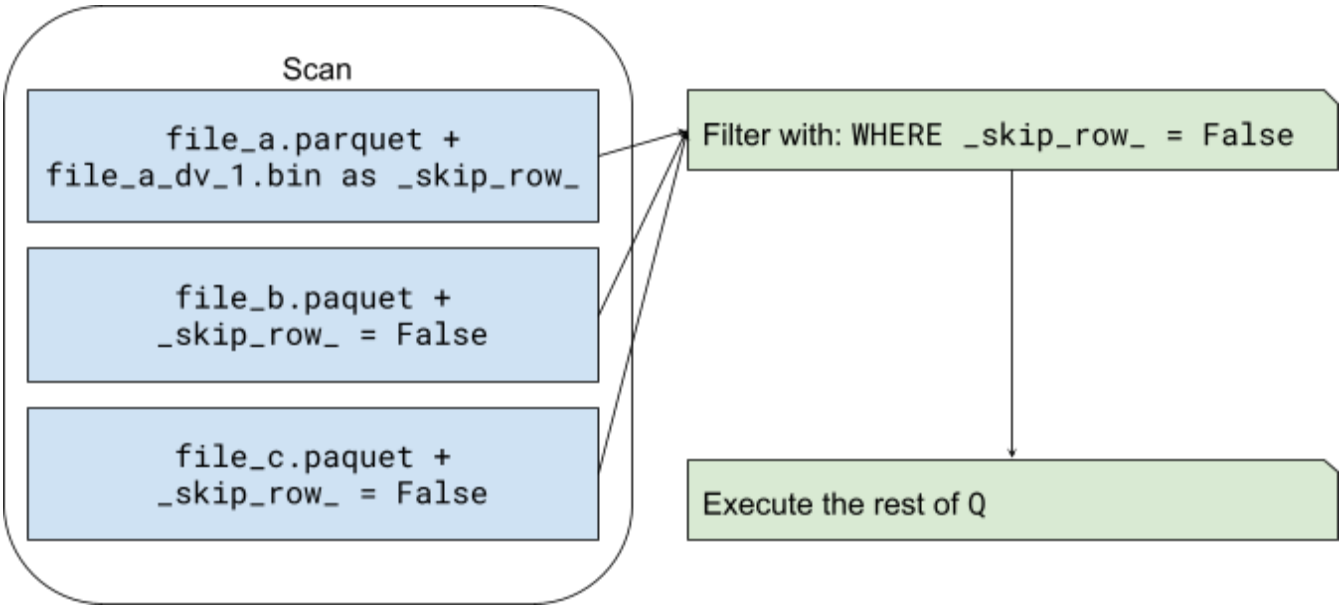
Delta Log Snapshot at Version 2		
Type	**Path**	**DV Path**
Tombstones		
RemoveFile	file_a.parquet	NULL
Current State		
AddFile	file_b.parquet	NULL
AddFile	file_c.parquet	NULL

Delta Log Snapshot at Version 2		
Type	**Path**	**DV Path**
`AddFile`	`file_a.parquet`	`file_a_dv_1.bin`

In the current Delta protocol, it is not legal for actions with the same file name to appear twice in the same commit. With Deletion Vectors, we modify the protocol so that the DV path becomes part of the key. So then the modified rule is that the same *(Path, DV Path)* combination cannot appear twice in the same commit.

Step 2 - Reading the Current State

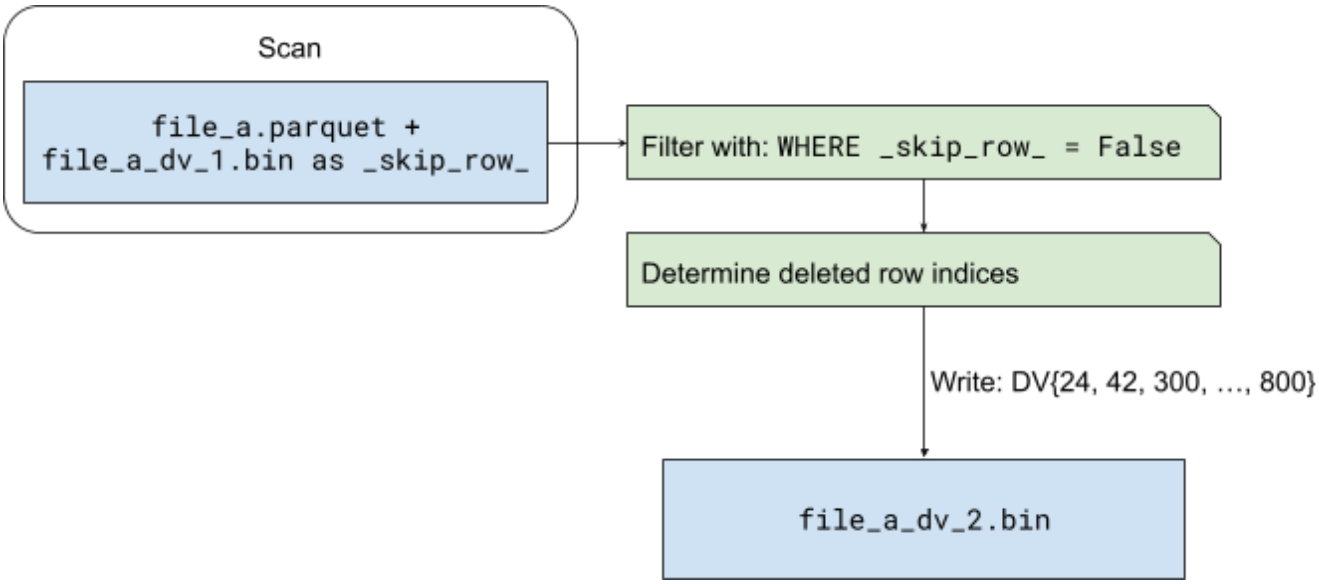
Say we want to perform some query Q now that requires a full table scan. We can read the rows in `file_b.parquet` and `file_c.parquet` as normal, but in order to read the correct rows of `file_a.parquet` we must also read the current DV `file_a_dv_1.bin` and then remove the rows which have their indices in the DV before any further processing of Q.



Step 3 - DML with existing DVs

When a file that already has an associated DV gets updated again, we must ultimately produce a merged DV between the already invalid rows from before the DML command and the newly invalidated rows from the command itself.

Let's say we somehow delete rows 300–800 from `file_a.parquet` and all other files are untouched by the operation. The high level flow might look something like this:



As can be seen in the figure, the new DV `file_a_dv_2.bin` contains all entries from the previous DV `file_a_dv_1.bin` plus the new entries (rows 300-800). The exact mechanism of how the combined DV is produced is still subject to a more detailed design, and likely going to be specific to the particular DML operation.

Whatever the mechanism, at the end of the operation we must commit a new version of the Delta table that adds a new entry for `file_a.parquet` using `file_a_dv_2.bin` and also marks the entry for `file_a.parquet` with `file_a_dv_1.bin` as removed, since it is no longer valid to read with that DV.

COMMIT 3		
Type	**Path**	**DV Path**
RemoveFile	file_a.parquet	file_a_dv_1.bin
AddFile	file_a.parquet	file_a_dv_2.bin

Delta Log Snapshot at Version 3		
Type	**Path**	**DV Path**
Tombstones		
RemoveFile	file_a.parquet	NULL
RemoveFile	file_a.parquet	file_a_dv_1.bin
Current State		
AddFile	file_b.parquet	NULL
AddFile	file_c.parquet	NULL
AddFile	file_a.parquet	file_a_dv_2.bin

Step 4 - Full Compaction

At this point there are 503 (~50%) invalid rows in `file_a.parquet`. That means every time we scan the file we read twice as much data as we really need to. This is probably more than we are willing to pay, so we should compact out the DV and rewrite the remaining valid rows into a new file. This could happen as part of an OPTIMIZE (Z-ORDER BY), or simply as part of the DML operation that pushes the file over the "invalid row"-threshold. Whichever the mechanism used, the resulting Delta Log should look something like this, where `file_d.parquet` contains the rewritten rows from `file_a.parquet` minus the invalid rows in `file_a_dv_2.bin`:

COMMIT 4		
Type	**Path**	**DV Path**
RemoveFile	file_a.parquet	file_a_dv_2.bin
AddFile	file_d.parquet	NULL

Delta Log Snapshot at Version 4		
Type	**Path**	**DV Path**
Tombstones		
RemoveFile	file_a.parquet	NULL
RemoveFile	file_a.parquet	file_a_dv_1.bin
RemoveFile	file_a.parquet	file_a_dv_2.bin
Current State		
AddFile	file_b.parquet	NULL
AddFile	file_c.parquet	NULL
AddFile	file_d.parquet	NULL

High-level Anatomy of a Deletion Vector

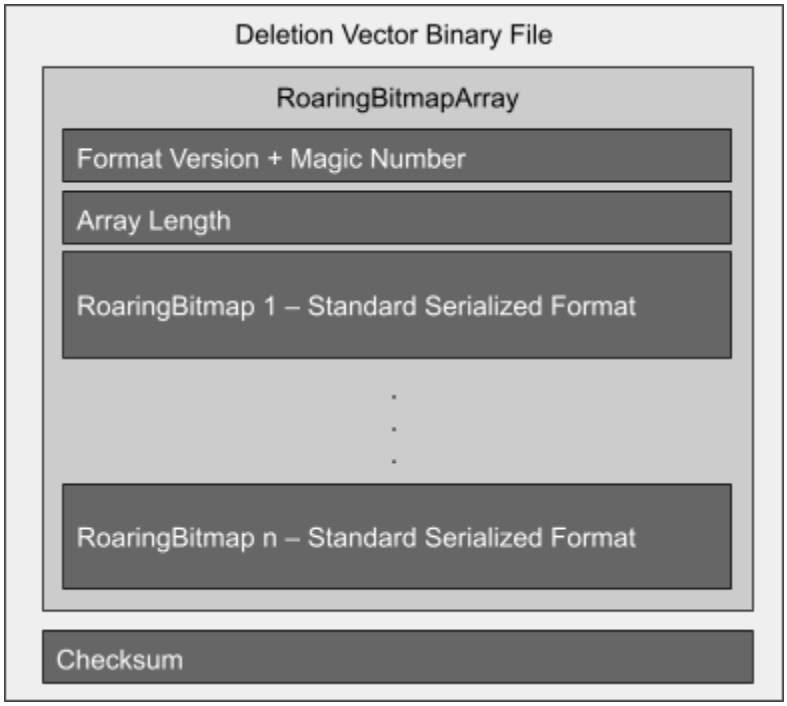
As mentioned before, Deletion Vectors are in essence sets of row indices, that is 64-bit integers. As storing a literal set of 8 byte numbers becomes quite large more quickly than would be useful for us, we must store these sets in a compressed format. The fundamental building block of the proposed implementation (identical to what is currently used in Low-Shuffle Merge) is the open source RoaringBitmap library. RoaringBitmap is a flexible format for storing 32-bit integers that automatically switches between three different encodings at the granularity of a 16-bit block:

1. Simple integer array when the number of values in the block is small.
2. Bitmap-compressed when the number of values in the block is large and sparse.
3. Run-length encoded when the number of values in the block is large, but clustered.

The serialization format is standardized and a C/C++ implementation is available as well, which will be convenient for non-Java based Delta connectors (e.g. delta-rust).

Since RoaringBitmap only covers 32-bit integers (but tables can have files with more rows than this), we extended the format for low-shuffle merge by keeping an array of 32-bit RoaringBitmaps. This is feasible because we use them to store row indices, which by their very nature are not arbitrarily sparse but are confined to a certain prefix of the 64-bit space starting at 0.

The proposed format for storing DVs in cloud storage is one (or more) of these RoaringBitmapArrays per file, together with a checksum for each DV:



See the protocol details in the Appendix.

Part II: Overview of changes needed in Delta

Protocol Changes

We are going to store the following information for each data file with an associated DV:

- Path (absolute or relative) to the DV file + offset within the file + length (to allow fetching exact ranges)
- OR: An inline serialized DV
- AND: Number of records removed by the DV

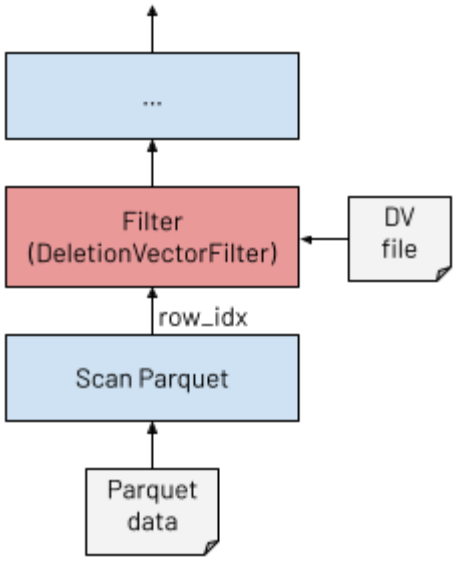
This gives us an array of options for how to store DVs:

(A) As separate files mapping 1:1 to data files
(B) As separate files mapping 1:N to data files – reducing the number of additional files created compared to (A)
(C) Inlined into the Delta logs – optimal solution for the smallest DVs

[The final proposed protocol is in the appendix.]

Scan Changes

We will modify each query plan by inserting a Filter node containing DeletionVectorFilter on top of Scan. DeletionVectorFilter is going to be an expression containing a map of data file to DV file. The map is going to be distributed as a broadcast variable.



Using the map, input_file_name() and row indexes generated by the Scan operator, the Filter operator will eliminate the deleted rows from the Scan's output.

File and Column Statistics

The following statistics are currently stored in the Delta log:

- Per Parquet file: number of records.
- Per column in Parquet file: number of nulls, min, max.

The latter are not obligatory (can be omitted, e.g., if there's too many columns) and the min/max values are not guaranteed to be present in the data (e.g., string data is truncated, and protocol spec does not require this).

With the introduction of the DVs, the status will be as follows:

- For each DV file, we're going to store the number of removed rows.
    - The number of records for a Parquet file with DV can be calculated by subtracting this value from the Parquet file stats.
    - We store row counts for Parquet file and DV file separately to be able to compute the ratio of deleted rows down the line.
- The per-column min and max values remain unchanged.
    - A min/max range that is valid for a given set of rows will always be valid for a subset of them.
    - Finding a narrower min/max range would be expensive (requiring us to reread and aggregate the content of the Parquet file) and unlikely to provide substantial improvements in uncontrived scenarios.
- The per-column null count will have to be treated as an upper bound for Parquet files with associated DVs.

DML Commands

MERGE/UPDATE/DELETE: We will have to rewrite all the DMLs such that instead of COW, it will do the following

- Scan the table for files that have matching data and need to updated (same as COW)
    - This assumes that if there are existing DVs on those files, then we will be scanning with them as detailed earlier.
- For lines in the touched files to be updated or deleted, write new deletion vectors to mark those lines as "deleted".
- Write the updated rows (as well as unmatched rows to be inserted), write them into new files.

The detailed design of this will be elaborated later as the project progresses.

INSERT: No impact.

Table Utility Commands

Optimize

The Optimize command is responsible, among other things, for combining small data files into

larger ones. With the introduction of DV, it will additionally be responsible for compacting DVs with their associated data files. Without making absolutely any changes to Optimize, it will still work "correctly" (as in not produce incorrect data) as long as we use DVs to scan the files chosen to be optimized (as detailed earlier). However, files that are more than the desired size but have a significant amount of rows marked as deleted (that is, real valid data is < desired size) will not be considered for compaction. So we will have to update Optimize to consider additional files based on the fraction of rows not deleted by DVs.

Vacuum

VACUUM command has to be amended to correctly identify which DV files need to be removed and when.

Convert to Delta

This command is only valid for a directory containing only Parquet data. In such a case, no change to the command is required.

Describe History

The list of metrics available for each of the commands should be extended to account for the DVs.

Generate

External manifests produced with GENERATE are lists of Parquet files belonging to a specific version of a Delta table. They are used to provide a method for Presto, Athena, Snowflake and Redshift Spectrum to read from a Delta table. They can be generated either on demand (via GENERATE command), or automatically (on each commit).

Since it's not possible to include DVs in an external manifest, we will only allow generating them for snapshots that contain no DVs. The user is going to be responsible for executing Full Compaction either prior to the GENERATE command, or whenever they want a new version of the external manifest to be automatically created.

RESTORE TABLE: The command must work across the "upgrade boundary", but it will not downgrade the protocol version.

Appendix - Delta Protocol Changes

In order to be able to associate a DV with a parquet file we need to make changes to the Delta protocol. On the one hand, those are obvious extensions to the storage format to add the new fields we require. On the other hand, we also need to adjust the action reconciliation rules to make sure we still produce correct snapshots, as a DV field cannot simply be treated like yet another metadata column. This is because reading a file with a DV is a correctness requirement, not a performance improvement. Ignoring the DV would cause us to read an outdated and inconsistent view of the data.
Given these constraints, the protocol changes proposed below will require a bump in both reader and writer protocol versions.

Here is a short list of necessary changes:

- Add a new struct field to `add` and `remove` actions.
- Change the action uniqueness semantics from operating on path alone, to operating on a `(path, deletionVector.uniqueId)` tuple.
- Adjust the duplicate file criteria to clarify which possible actions are legal with respect to the tuple uniqueness.
- Add a new field to stats to mark explicitly whether they are currently accurate or upper/lower bounds.

Log Format Change

We need to add a DV field to both the `add` and `remove` actions.

New fields are highlighted in green below.

New Add Action Format

Field Name	Data Type	Description
`path`	`String`	A relative path to a file from the root of the table or an absolute path to a file that should be added to the table. The path is a URI as specified by RFC 2396 URI Generic Syntax, which needs to be decoded to get the file path.
`partitionValues`	`Map[String, String]`	A map from partition column to value for this file.
`size`	`Long`	The size of this file in bytes
`modificationTime`	`Long`	The time this file was created, as milliseconds since the epoch
`dataChange`	`Boolean`	When `false` the file must already be present in

		the table or the records in the added file must be contained in one or more `remove` actions in the same version.
`stats`	Statistics Struct	Contains statistics (e.g., count, min/max values for columns) about the data in this file
`tags`	`Map[String, String]`	Map containing metadata about this file
`deletionVector`	`DeletionVector Descriptor Struct`	Either `null` (or absent in JSON) when no DV is associated with this file, or a struct (described below) that contains necessary information about the DV that is associated with this file.

New Remove Action Format

Field Name	Data Type	Description
`path`	`String`	A relative path to a file from the root of the table or an absolute path to a file that should be removed from the table. The path is a URI as specified by RFC 2396 URI Generic Syntax, which needs to be decoded to get the file path.
`deletionTimestamp`	`Option[Long]`	The time the deletion occurred, represented as milliseconds since the epoch
`dataChange`	`Boolean`	When `false` the records in the removed file must be contained in one or more `add` file actions in the same version
`extendedFileMetadata`	`Boolean`	When `true` the fields `partitionValues`, `size`, and `tags` are present
`partitionValues`	`Map[String, String]`	A map from partition column to value for this file.
`size`	`Long`	The size of this file in bytes
`tags`	`Map[String, String]`	Map containing metadata about this file
`deletionVector`	`DeletionVector Descriptor Struct`	Either `null` (or absent in JSON) when no DV is associated with this file, or a struct (described below) that contains necessary information about the DV that is associated with this file.

DeletionVectorDescriptor Format

Field Name	Data Type	Description
`storageType`	`String`	A single character to indicate how to access the DV. (See below.)
`pathOrInlineDv`	`String`	Three format options are currently proposed: • If `storageType = 'u'` then `<random prefix - optional><base85 encoded uuid>`: The deletion vector is stored in a file with a path relative to the data directory of this Delta table, and the file name can be reconstructed from the UUID. See Derived Fields for how to reconstruct the file name. The random prefix is recovered as the extra characters before the (20 characters fixed length) `uuid`. • If `storageType = 'i'` then `<base85 encoded bytes>`: The deletion vector is stored inline in the log. The format used is the same as when the DV is stored on disk (See Deletion Vector Format). • If `storageType = 'p'` then `<absolute path>`: The DV is stored in a file with an absolute path given by this path, which has the same format as the `path` field in the `add/remove` actions.
`offset`	`Option[Int]`	Start of the data for this DV in number of bytes from the beginning of the file it is stored in. Always `None` (absent in JSON) when `storageType = 'i'`.
`sizeInBytes`	`Int`	Size of the serialized DV in bytes (raw data size, i.e. before base85 encoding, if inline).
`cardinality`	`Long`	Number of rows the given DV logically removes from the file.

The Base85 variant proposed is Z85, because it is JSON-friendly.

Derived Fields

Some fields that are necessary to use the DV are not stored explicitly but can be derived in code from the stored fields.

Field Name	Data Type	Description	Calculated As
`uniqueId`	`String`	Uniquely identifies a DV for a given file. This is used for snapshot	If `offset` is `None` then `<storageType><pathOrInlineDv>`. Otherwise `<storageType><pathOrInlineD`

		reconstruction to differentiate the same file with different DVs in successive versions.	v>@<offset>
absolutePath	String/URI/Path	The absolute path to access the DV under. Can be calculated for relative path DVs by providing a parent directory path.	If storageType='p', just use the already absolute path. If storageType='u', the DV is stored at <parent path>/<random prefix>/deletion_vector_<uuid in canonical textual representation>.bin. This is not a legal field if storageType='i', as an inline DV has no absolute path.

JSON Example 1 — On Disk with Relative Path (with Random Prefix)

```
{
      "storageType" : "u",
      "pathOrInlineDv" : "ab^-aqEH.-t@S}K{vb[*k^",
      "offset" : 4,
      "sizeInBytes" : 40
      "cardinality" : 6
}
```

Assuming that this DV is stored relative to an `s3://mytable/` directory, the absolute path to be resolved here would be:
`s3://mytable/ab/deletion_vector_d2c639aa-8816-431a-aaf6-d3fe2512ff61.bin`

JSON Example 2 — On Disk with Absolute Path

```
{
      "storageType" : "p",
      "pathOrInlineDv" :
"s3://mytable/deletion_vector_d2c639aa-8816-431a-aaf6-d3fe2512ff61.bin",
      "offset" : 4,
      "sizeInBytes" : 40
      "cardinality" : 6
}
```

JSON Example 3 — Inline

```
{
      "storageType" : "i",
      "pathOrInlineDv" : "^Bg9^0rr910000000000iXQKl0rr91000f55c8Xg0@@D72lkbi5=-{L",
      "sizeInBytes" : 44,
      "cardinality" : 6

}
```

The row indices encoded in this DV are: `3, 4, 7, 11, 18, 29`

Action Reconciliation Changes

These are the new action reconciliation rules, with changes highlighted in green and original text crossed out and highlighted in red:

A given snapshot of the table can be computed by replaying the events committed to the table in ascending order by commit version. A given snapshot of a Delta table consists of:

- A single `protocol` action
- A single `metaData` action
- A map from `appId` to `transaction` version
- A collection of `add` actions with unique ~~paths~~ `(path, deletionVector.uniqueId)` tuples.
- A collection of `remove` actions with unique ~~paths~~ `(path, deletionVector.uniqueId)` tuples. The intersection of the ~~paths~~ `(path, deletionVector.uniqueId)` tuples in the `add` collection and `remove` collection must be empty. That means a file with a particular DV associated cannot exist in both the `remove` and `add` collections; however, the same file can exist with **different** DVs, as logically they represent **different** content. The `remove` actions act as tombstones, and only exist for the benefit of the VACUUM command. Snapshot reads only return add actions on the read path.

To achieve the requirements above, related actions from different delta files need to be reconciled with each other:

- The latest `protocol` action seen wins
- The latest `metaData` action seen wins
- For transaction identifiers, the latest `version` seen for a given `appId` wins
- All `add` actions for different ~~paths~~ `(path, deletionVector.uniqueId)` tuples need to be accumulated as a list. The latest `add` action (from a more recent delta file) observed for a given ~~path~~ `(path, deletionVector.uniqueId)` tuple wins.
- All `remove` actions for different ~~paths~~ `(path, deletionVector.uniqueId)` tuples need to be accumulated as a list. If a `remove` action is received later (from a more recent delta file) for the same ~~paths~~ `(path, deletionVector.uniqueId)` tuple as an `add` operation, the corresponding `add` action should be removed from the `add` collection and the file and DV combination need~~s~~ to be tracked as part of the `remove` collection.
- If an `add` action is received later (from a more recent delta file) for the same `(path, deletionVector.uniqueId)` tuple as a `remove` operation, the corresponding `remove` action should be removed from the `remove` collection and the file and DV combination need~~s~~ to be tracked as part of the `add` collection.

Duplicate Actions

The current definition of a duplication action reads as follows:
"Since actions within a given Delta file are not guaranteed to be applied in order, it is not valid for multiple file operations with the same path to exist in a single version."

We propose the following addition to clarify what constitutes a legal table in the presence of `(path, deletionVector.uniqueId)` tuples as unit of uniqueness (green highlights **legal** situations, red highlights **illegal** situations):
For any commit...

- it is **legal** for the same parquet file to occur in an `add` action and a `remove` action, but with two different DVs (or one not having a DV at all).
- it is **legal** for the same parquet file to be added and/or removed and *also* occur in an `AddCDCFile` action.
- it is **illegal** for the same parquet file to be added twice or removed twice, once without a DV and once with a DV
- it is **illegal** for the same parquet file to be added twice or removed twice with two different DVs.

For an elaboration on this subject, see this document.

Tombstones

In order clarify the lazy deletion behaviour with tombstones in the presence of DVs, we propose the following reformulation of the particular paragraph from the specification (changes highlighted in green and original text crossed out and highlighted in red):
"The `remove` action includes a timestamp that indicates when the removal occurred. Physical deletion of the ~~file~~ parquet and DV files can happen lazily after some user-specified expiration time threshold. This delay allows concurrent readers to continue to execute against a stale snapshot of the data. A `remove` action should remain in the state of the table as a tombstone until it has expired. A tombstone expires when the creation timestamp of the latest delta commit file exceeds the expiration threshold added to the `remove` action timestamp."

File Statistics

Number of Records

The `stats.numRecords` field now indicates only the **physical** number of rows in the file, while before the physical and logical number of rows were identical. For files with associated DVs the current **logical** number of rows can be calculated as `stats.numRecords - deletionVector.cardinality`.

Column Statistics

Currently the Delta specification gives the following (somewhat impossible) definition of its column statistics:

Name	Description
`nullCount`	The number of null values for this column
`minValues`	A value smaller than all values present in the file for this column.
`maxValues`	A value larger than all values present in the file for this column

In the presence of DVs, we want to give ourselves the option to have the statistics somewhat outdated, i.e. not reflecting deleted rows, yet. But we also want to be able to, for example, lazily update them to be accurate again. To make it clear which of those two states the stats for all columns are currently in, we add a `boolean` flag `stats.tightBounds` to clarify whether the bounds are **tight** (i.e. the `min/maxValue` exists in the valid state of the file) or **wide** bounds (i.e. the `minValue` is <= all valid values in the file, and the `maxValue` >= all valid values in the file). These upper/lower bounds are [sufficient information] for data skipping, but cannot be used to calculate aggregations such as `max(column)` from metadata alone.

The new description for the column stats with that scheme would be (changes highlighted in green):

Name	Description (`stats.tightBounds=true`)	Description (`stats.tightBounds=false`)
`nullCount`	The number of `null` values for this column	If the `nullCount` for a column equals the **physical** number of records (`stats.numRecords`) then **all** valid rows for this column must have `null` values (the reverse is not necessarily true). If the `nullCount` for a column equals 0 then all valid rows are non-`null` in this column (the reverse is not necessarily true). If the `nullCount` for a column is any value other than these two special cases, the value carries no information and should be treated

		as if absent.
`minValues`	A value that is equal to the smallest valid value* present in the file for this column. If all valid rows are null, this carries no information.	A value that is less than or equal to all valid values* present in this file for this column. If all valid rows are null, this carries no information.
`maxValues`	A value that is equal to the largest valid value* present in the file for this column. If all valid rows are null, this carries no information.	A value that is greater than or equal to all valid values* present in this file for this column. If all valid rows are null, this carries no information.

*String columns are cut off at a fixed prefix length, timestamp columns are truncated down to milliseconds.

Deletion Vector Format

Deletion Vectors are basically sets of row indices, that is 64-bit integers. As storing a literal set of 8 byte numbers becomes quite large more quickly than would be useful, we must store these sets in a compressed format. The fundamental building block for this is the open source RoaringBitmap library. `RoaringBitmap` is a flexible format for storing 32-bit integers that automatically switches between three different encodings at the granularity of a 16-bit block:

1. Simple integer array, when the number of values in the block is small.
2. Bitmap-compressed, when the number of values in the block is large and sparse.
3. Run-length encoded, when the number of values in the block is large, but clustered.

The serialisation format is standardised and both Java and C/C++ implementations are available (among others).

Since `RoaringBitmap` only covers 32-bit integers, but Delta tables can have files with more than 2.1 billion rows, we extend the format in a simple manner by keeping an array of 32-bit `RoaringBitmap`s and using the upper 32-bits to index into the array. This is feasible because we use them to store *row indices*, which by their very nature are not arbitrarily sparse but are confined to a certain prefix of the 64-bit space starting at 0.

The serialization format for such a `RoaringBitmapArray` is as follows:

Bytes	Name	Description
0 – 3	`magicNumber`	1681511377; Indicates that the following bytes are serialized in this exact format. Future alternative—but related—formats *must* have a different magic number, for example by

		incrementing this one.
4 — end	`bitmap`	A serialized 64 bit bitmap in the portable standard format as defined in the RoaringBitmaps Specification.

The proposed format for storing DVs in cloud storage is one (or more) of these RoaringBitmapArrays per file, together with a checksum for each DV:

Bytes	**Name**	**Description**
0 — 1	`version`	The format version of this file: 1 for the format described here.
`repeat for each DV i`		For each DV
<start of i> — <start of i>+3	`dataSize`	Size of this DV's data (without the checksum)
<start of i>+4 — <start of i>+4+ `dataSize` - 1	`bitmapData`	One `RoaringBitmapArray` serialized as described above.
<start of i>+4+ `dataSize` — <start of i>+4+ `dataSize` + 3	`checksum`	CRC-32 checksum of `bitmapData`