

Iceberg V4 Adaptive Metadata Tree

Amogh Jahagirdar amoghj@apache.org

Ryan Blue blue@apache.org

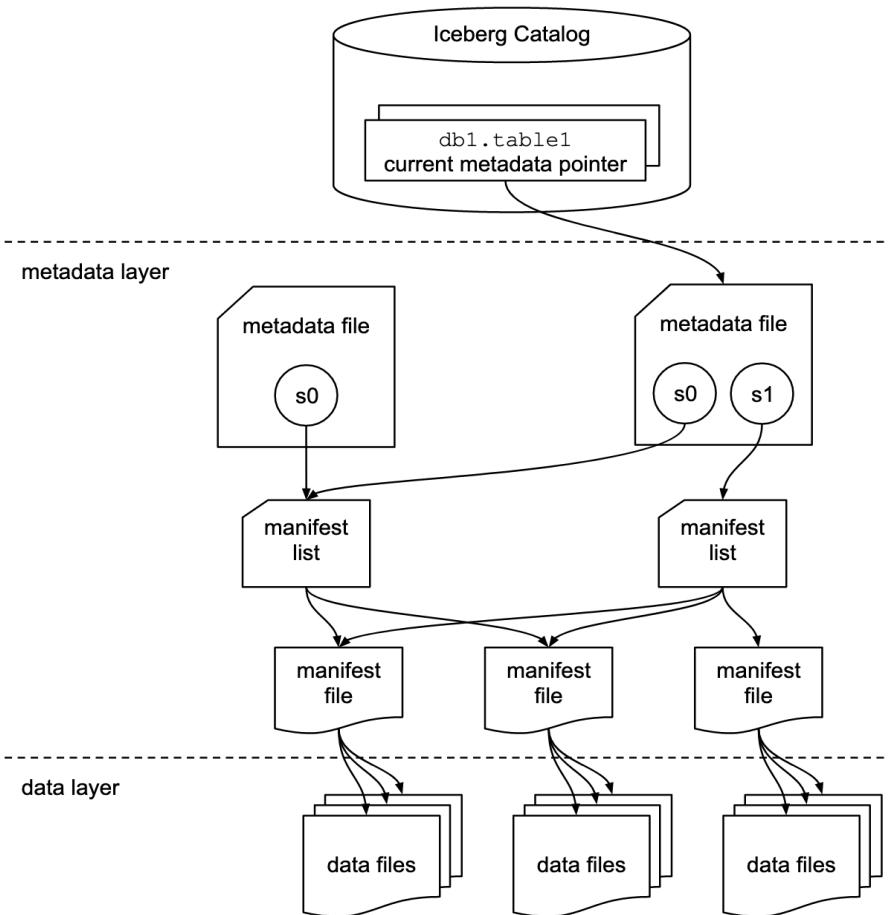
Anoop Johnson anoop@apache.org

Daniel Weeks dweeks@apache.org

*⚠ This document has been merged into this [proposal](#) .
Please refer to this other proposal and comment on that.*

Background

The existing metadata structure in Iceberg consists of a manifest list as an intermediate layer between the snapshot entry and manifest files. This manifest list provides structure and information about the contained manifests, improving scan planning by enabling pruning based on partition summaries (lower/upper bounds, contains null/nan). Over time, additional relevant fields have been added to track information such as sequence numbers and row IDs.



There are a few challenges with the current metadata tree structure:

1. High write latency since every write would need to produce new data files, new manifest, a new manifest list containing the new manifest and produce a root level metadata file which is atomically updated in the catalog. All of this is currently done serially. The high write latency is most noticeable for single file commits and small tables.
2. High metadata storage footprint: the manifest lists and manifests are immutable and rewritten when modified, and need to be retained during the time travel window.
3. High maintenance overhead: small writes produce small manifest files that need to be compacted.
4. Column upper and lower bounds currently only exist at the manifest level, but do not exist at higher levels in the tree for pruning.

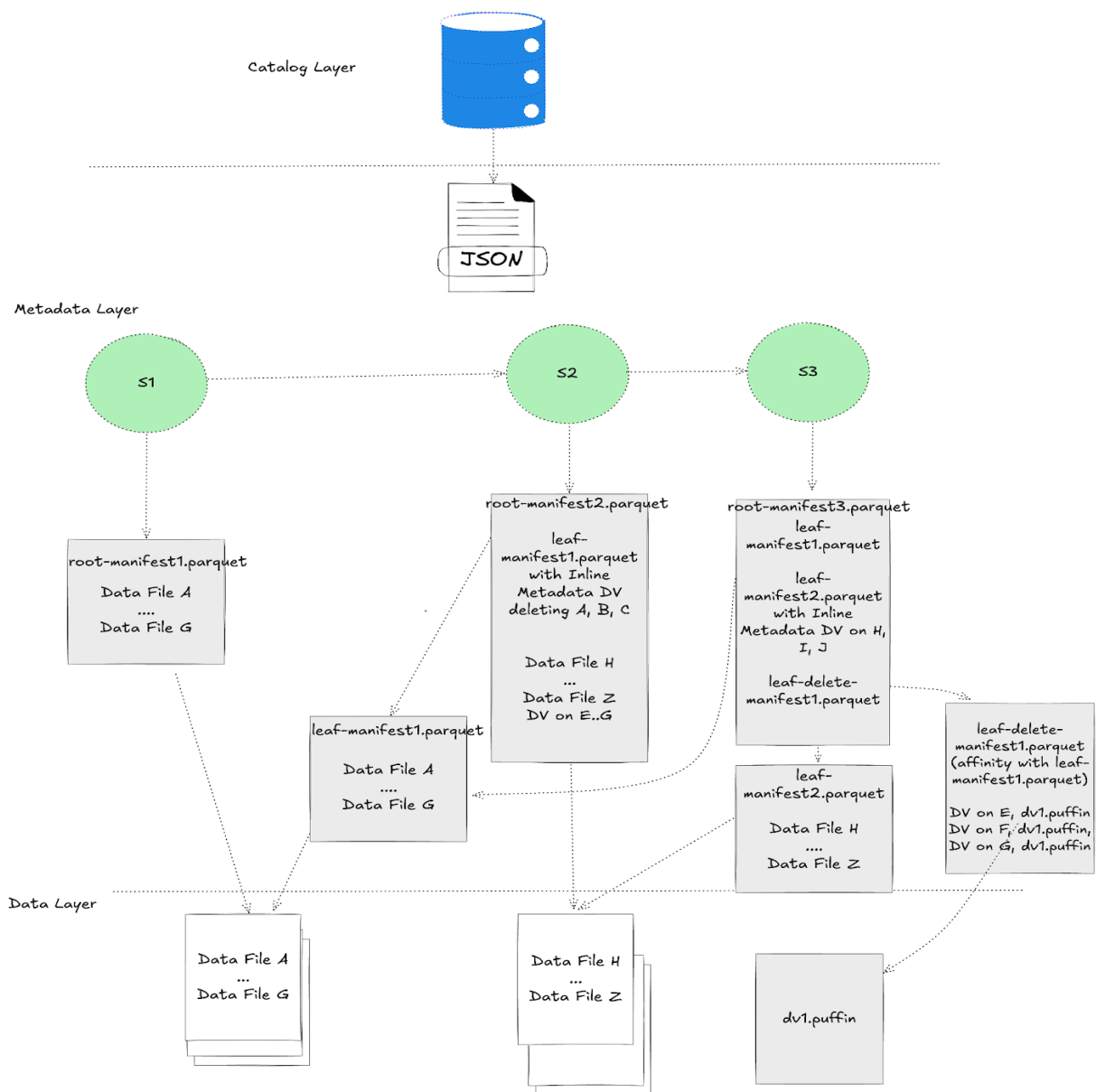
Goals

- 1.) Reduce metadata write latency for small commits by introducing an adaptive metadata tree structure which enables fast single file commits and a simple structure for small

tables, while being able to adapt and scale simply as the table grows so that Iceberg's planning performance is still retained at scale.

- 2.) Enable effective pruning at all levels in the tree by including aggregate column statistics at all levels, considering new data types such as geospatial.
- 3.) Reduce the need to compact manifests
- 4.) Further improve planning performance by reading fewer small manifests and avoid 2 phase planning between data files and deletion vector files

Proposed Structure



Key Decisions and other Considerations

The following are key structural changes being proposed for the adaptive metadata tree in V4:

1. A single manifest structure will be used throughout the tree. There is a single root manifest and there can be leaf manifests. Manifests can contain a limited set of contents depending on if it's a root or a leaf manifest (this set will be elaborated below). This will be a 2 level tree structure.
2. Leaf Manifest Deletion Vectors (DVs) will be added to reduce metadata write amplification involved in rewriting manifests; these leaf manifest DVs can only exist in root manifests and they express which positions in a leaf manifest are deleted. The leaf manifest DVs may be stored inline or in a separate file.
3. Remove partition struct metadata for manifests and data files in favor of columnar stats in manifest entries.

Each of these points are elaborated below.

Single Manifest Structure

A single manifest structure will be used in the proposed metadata tree structure where there are 2 levels, and there can be a single root manifest and leaf manifest. We still maintain separate manifests for data files and DVs at the leaf level; the rationale of this organization is explained later on. In this model, there's no completely separate "manifest list" structure; the root manifest is logically acting as that.

Manifest Type	Allowed Content
Root	<ul style="list-style-type: none">• Leaf Manifests• Leaf Manifest DVs• Data Files• DVs on Data Files• Equality Deletes
Leaf data manifest	<ul style="list-style-type: none">• Data Files
Leaf delete manifest	<ul style="list-style-type: none">• DVs• Equality Deletes

Why a common manifest structure throughout the tree?

The primary advantage of having a common manifest structure is around simply code reuse at different levels of the tree. Implementations of the Iceberg V4 spec don't need to have completely separate manifest list readers vs manifest readers/writers. There is the additional

complexity of managing the fact that certain types of content are allowed at different levels in the tree, but writers can be differentiated between root/leaf and that additional complexity should not be nearly as much as the separation of manifest and manifest list that exist today.

Why limit tree depth to 2?

The reason we propose not to have unbounded hierarchies is to prevent writers from doing things that seem performant for writers in the short term but lead to complicated reads and maintenance.

- The primary issue with not bounding the levels is that writers could keep writing a top level manifest which references the previous top level to keep having fast writes. However, this quickly leads to a skewed tree structure, which at scale leads to tables becoming unreadable without a flattening of manifests.
- With a skewed tree, parallelism on reading metadata is essentially eliminated; manifests would have to be read in a hierarchical order. This would be a step backwards compared to Iceberg's planning capabilities today
- Lastly, the depth of the tree can be scaled up in the future if it really ends up being required. With clear recursive implementations, we should be always able to increase this, but for now it seems better to start with the 2-level tree.

Leaf Manifest DVs

In this new structure, we propose adding the concept of a leaf manifest DVs which expresses positions in a leaf manifest which are deleted. We also propose that this is inlined in binary format since these are fairly small structures. Due to the requirement that this is a 2-level content metadata tree, **only** the root manifest can have these leaf manifest DVs. There can be at most 1 leaf manifest DV for a given leaf manifest.

New data file/DV writes will target the root manifest. Beyond a threshold for a large write, entries in root manifest would be flushed out to leaf manifests as part of the commit. The details of this maintenance and the scaling dynamics are elaborated more in later sections since those are inextricably linked to how we propose data and delete manifests should be laid out with affinity.

How do we track replaced or removed leaf entries with Leaf Manifest DVs?

Here we outline how inline leaf manifest DVs will be used in different write operations to track removed entries in leaf manifests.

Replacing DVs and Data Files

Replacing DVs on data files would first involve determining which manifests contain those DVs and which position in those manifests that need to be marked to be deleted. The same principle

applies for replacing data files. This reading of manifests is a cost that's already incurred for write operations in general so there's no additional work being done here compared to the current state.

Once the positions to replace in the affected manifests are determined:

1. The new data files and DVs will be written to the root manifest
2. Leaf manifest DVs containing the bitmap of the manifest positions which should be marked as removed, would be produced and written to the root.

Removing Data Files

Removing a data file contained in a leaf manifest requires removing any corresponding DV to prevent orphans. First, the position in the leaf data manifest to mark as removed will be determined. Then if there's a referenced DV file for the data file, then any associated delete manifests will need to be read to determine which position in the delete manifest will be removed. In this case, 2 leaf manifest DVs will be produced, 1 for the data manifest and 1 for the delete manifest. These leaf manifest DVs will be merged with any existing leaf manifest DV for those manifests and placed in the root manifest.

Why not have inline data DVs?

Data DVs are typically much larger than leaf manifest DVs. If the bitmap is dense, the data DVs can reach several megabytes. Since this can cause metadata bloat and out of memory issues, they are not supported.

Affinity between Data and Delete Manifests

Iceberg currently has separate manifests for data and row-level deletes. This is a flexible writing pattern, but the downside is that readers need to join the data manifests against delete manifests to match the data file with the DV. To reduce the cost of the join, we propose an affinity between data and delete manifests: a delete manifest can be *affiliated* to exactly one data manifest. A data manifest can have more than one affiliated delete manifest.

Flushing the data files or DVs from the root to leaf manifests will require leaf delete manifests to be rewritten. To reduce the write amplification, we can maintain a small number of unaffiliated delete manifests. These unaffiliated delete manifests can be read once and broadcast.

Pros:

- Single-pass planning: readers can do a parallel colocated join of data and delete manifests.
- Statistics-based pruning works on delete manifests: we only need to open the delete manifests of unpruned data manifests.
- Low write amplification: we only need to update delete manifests while updating DVs.

Cons:

- Large-scale deletions with low data locality (e.g. MERGE using a UUID field) can produce large unaffiliated delete manifests or rewrite of a large number of affiliated delete manifests.

A variant of this approach is to do *physical colocation* of the data files and the DVs as separate rows in the same leaf manifest file. The advantage is fewer manifest files and simplified planning, as leaf manifests are self-contained. We discarded it because of the high write amplification to replace DVs, as the data files and the statistics need to be rewritten as well.

How does planning work?

1. Read the root manifest to determine any leaf manifests to read as well as any applicable data/delete files in the root manifest.
2. Load leaf manifest DVs for the leaf manifests to read
3. Leaf data manifests along with their associated delete manifests, along with any additional unaffiliated delete manifests will be read, filtering out any manifest entries referenced in the leaf manifest DVs or any manifest entries marked as deleted. Note that for any leaf data and delete manifests with affinity, both manifests can be read in parallel.

Alternate approach: No affinity between data and delete manifests (Existing behavior)

Pros:

- Lowest write amplification.

Cons:

- Expensive join operation during reads.
- No statistics-based pruning for delete manifests.

Alternate approach: Unified Manifests: Manifest entry contains Data File, DV Pair

In this approach, we do not have the separation between data and delete manifests. Each manifest entry has the data file and its DV.

Pros:

- Fast single-pass planning.

Cons:

- Read amplification: changing a DV requires reading the associated statistics for the data files so that they can be copied.
- Write amplification: changing a DV requires copying the data files and associated statistics.

Metadata Tree Maintenance

The data files and DVs in the root manifest will be flushed to new leaf manifests. The flushing will be based on configurable thresholds on the maximum number of data files and DVs that can be present in the root node: we propose separate thresholds for data files and DVs, as the storage footprint is different. Ideally the time to do I/O on the root node should be close to the round trip latency of cloud storage systems. Past this point, the CoW at the root level will be so expensive relative to the size of the write and compromise any future small writes; it makes sense to flush to a leaf manifest at this point.

If there are many small leaf manifests, periodic metadata maintenance can coalesce them to optimize scan performance.

Proposed V4 Manifest Entry Fields

Field ID	Name	Type	Required or Optional	Description
0	status	int with meaning: 0: EXISTING 1: ADDED 2: DELETE	required	Carried over from current format: Used to track additions and deletions of any manifest entries including leaf manifests in the root. Deleted entries are required when the snapshot has a non-null parent-id. Deletes are not used in scans.
1	snapshot_id	long	optional	Carried over from current format: Snapshot ID where the file was added, or deleted if status is 2. Inherited when null.
134	content_type	int	required int with meaning: 0: DATA 1: DELETION VECTOR 2: EQUALITY DELETE 3: DATA_MANIFEST 4: DELETE_MANIFEST	Type of content stored by the data file: data, equality deletes, or position deletes (all v1 files are data files). Content types 3, 4 manifest can only be defined in the root manifest.
146	dv_content	binary	optional	Serialized roaring bitmap of positions in a manifest which are deleted. Can only be defined for content 3 or 4 in the root manifest,

				else must be null.
2	content_entry	Content entry struct outlined below	required	File location with stats, etc. Details here .
3	sequence_number	long	optional	Carried over from current format: Data sequence number of the file. Inherited when null and status is 1 (added)


Proposed V4 Manifest Key Value Metadata

Name	Type	Required or Optional	Description
format-version	string	required	Iceberg Table format version used when writing the manifest
content	string	required	Content being tracked by manifest. Must be data, delete, or root

Note, as seen in the table we are proposing to remove the serialized schema and spec from key/value metadata in V4 since those fields can add significant overhead without much value considering we can always determine those from their corresponding IDs.

Proposed V4 Content Entry Struct Fields

Field ID	Name	Type	Required or Optional	Description
143	referenced_file	string	optional	Location of data file that a DV references if <code>content_type</code> is 1. Location of affiliated data manifest if <code>content_type</code> is 4 or null if delete manifest is unaffiliated.
147	partition_spec_id	int	optional	ID of partition spec used to write manifest or data/delete files.
100	location	string	required	Location of the file.
101	file_format	string	optional	File format. Must be defined if location is defined
103	record_count	long	required	Number of records in this file, or the cardinality of a DV

104	file_size_in_bytes	long	optional	Total file size in bytes. Must be defined if location is defined
10000 (individual fields in column_stats struct will have their own IDs)	column_stats	struct	optional	Stats struct  Column Stats Improvements
516	min_sequence_number	long	optional	Carry over from current format: The minimum data sequence number of all live data or delete files in the manifest; use 0 when reading v1 manifest lists. Must be set if content_type is 3 or 4, else null
521	manifest_stats	struct	optional	Manifest stats struct containing added_files_count (504), existing_files_count (505), deleted_files_count (506), added_rows_count (512), existing_rows_count (513), deleted_rows_count (514) Can only be set if content_type is 3 or 4
131	key_metadata	binary	optional	Implementation-specific key metadata for encryption
132	split_offsets	list<133: long>	optional	Split offsets for the data file. For example, all row group offsets in a Parquet file. Must be sorted ascending
135	equality_ids	list<136: int>	optional	Field ids used to determine row equality in equality delete files. Required when content=2 and should be null otherwise. Fields with ids listed in this column must be present in the delete file
140	sort_order_id	int	optional	ID representing sort order for this file
142	first_row_id	long	optional	The _row_id for the first row in the data file if content is 0. If content is 3, this is the starting _row_id to assign to rows added by ADDED data files. See First Row ID Inheritance

144	content_offset	long	optional	The offset in the file where the content starts. Only applicable for DVs
145	content_size_in_bytes	long	optional	The length of a referenced content stored in the file; required if content_offset is present.

How existing manifest List Fields map to Proposed V4 fields

Manifest list field	v4 field	Rationale or description
manifest_path	location	Shared with data_file.file_path
manifest_length	file_size_in_bytes	Shared with data_file.file_size_in_bytes
partition_spec_id	manifest_entry.content_entry.partition_spec_id	Moved to content_entry field; when filtering based on predicates we will need to use the partition spec when evaluating if a given entry needs to be read or not since we will be storing partition transform results as derived values in stats. E.g. if a filter on ts = "07-06-2025T12:00:00.000UTC" is specified, and there are some entries referencing a partition spec with a days(ts) transform, we will need the spec ID to resolve the transform, apply it on the predicate and see if the entry might match.
content	manifest_entry.content	Content can be data files, DVs, eq deletes, manifest references, or manifest DV references. Manifest and manifest DV references are only allowed in the root manifest. DVs/eq deletes can exist in root manifest or delete manifests. Data files can exist in root manifest or data manifests.
sequence_number	manifest_entry.sequence_number	Shared with sequence_number
added_snapshot_id	manifest_entry.snapshot_id	Shared with snapshot_id
min_sequence_number	manifest_entry.min_sequence_number	Moved to manifest_entry since manifest entries can refer to other manifests in the root
added_files_count	manifest_entry.added_files_count	Moved to manifest_entry since manifest

		entries can refer to other manifests in the root
existing_files_count	manifest_entry.existing_files_count	Moved to manifest_entry since manifest entries can refer to other manifests in the root
deleted_files_count	manifest_entry.deleted_files_count	Moved to manifest_entry since manifest entries can refer to other manifests in the root
added_rows_count	manifest_entry.added_rows_count	Moved to manifest_entry since manifest entries can refer to other manifests in the root
existing_rows_count	manifest_entry.existing_rows_count	Moved to manifest_entry since manifest entries can refer to other manifests in the root
deleted_rows_count	manifest_entry.deleted_rows_count	Moved to manifest_entry since manifest entries can refer to other manifests in the root
partitions	REMOVED	Relocated info to column stats. General data filtering will be performed rather than specific partition filters lower_bound -> lower_bound upper_bound -> upper_bound contains_null -> null_count contains_nan -> nan_count
key_metadata	key_metadata	Shared with data_file.key_metadata
first_row_id	manifest_entry.first_row_id	First row ID is now set on manifest entry so it can be shared across entries for data files and entries which are data manifests

How existing manifest fields map to Proposed V4 fields

Manifest field	v4 field	Rationale or description
status	manifest_entry.status	
snapshot_id	manifest_entry.snapshot_id	
sequence_number	manifest_entry.sequence_number	(data sequence number)

file_sequence_number	manifest_entry.file_sequence_number	
manifest_entry.data_file	manifest_entry.content_entry	Renaming field to content_entry: since it's more general now. Can be data/delete files or data/delete manifests or leaf manifest DVs
data_file.content	manifest_entry.content	Moved to manifest_entry.content
data_file.file_path	location	Renamed, same ID
data_file.file_format	file_format	(Parquet, Avro, ORC, Puffin)
data_file.partition	REMOVED	Represented in column stats (need to support translation for equality deletes)Represented in column stats (need to support translation for equality deletes)
	Partition_spec_id with new ID of 147	Added to reconstruct partition tuple
data_file.record_count	record_count	
data_file.file_size_in_bytes	file_size_in_bytes	
	metadata_size_in_bytes	For estimating Puffin file overhead
data_file.column_sizes	REMOVED	Replaced by column stats (avg/max uncompressed size)
data_file.value_counts	REMOVED	Replaced by column stats value_count
data_file.null_value_counts	REMOVED	Replaced by column stats null_count
data_file.nan_value_counts	REMOVED	Replaced by column stats nan_count (optional)
data_file.lower_bounds	REMOVED	Replaced by column stats lower_bound
data_file.upper_bounds	REMOVED	Replaced by column stats upper_bound
data_file.key_metadata	key_metadata	
data_file.split_offsets	split_offsets	
data_file.sort_order_id	sort_order_id	
data_file.referenced_data_file	dv.referenced_file	
data_file.content_offset	dv.content_offset	Still needed for DVs

data_file.content_size_in_bytes	dv.content_size_in_bytes	Still needed for DVs
data_file.equality_ids	equality_deletes.ids	Carried over since we still need to be able to express in metadata which field IDs are stored in the delete file.

How do we remove partition stats and represent them in columnar stats?

Most partition transforms in Iceberg, such as time-based and identity transforms, are monotonically increasing—as the underlying column value increases, so does the partition value. This property enables effective pruning using lower and upper bound statistics for the original field, instead of using lower and upper bounds for partition values. Pruning via column stats can occur at the root of the tree, where these bounds represent aggregates over the manifest's contents, or at any layer, including data files and DVs. As a result, column statistics-based pruning is now possible at the top level, with root manifests holding aggregated lower and upper bounds for their referenced leaf manifests.

The notable exception to monotonically increasing transforms are bucket transformations. Bucket transforms are non-monotonic since they are the result of a hash function modulo buckets.

To handle non-monotonic functions, stats for derived values need to be stored to be able to achieve the same level of pruning that exists in the current manifest list partitions field.

Another important point to preserve the pruning capabilities of identity based transforms on strings/binary is that identity transform values stored in stats **must not be truncated**.

There are 2 high level approaches to representing partition values in the proposed columnar stats representation. They can either be stored as separate top level derived column stats structure or they can be stored as special fields within the column stats of the source column of the transform.

Let's take the example partition spec (identity(event_type), date(event_ts)):

a.) (**Preferred**) store new top level derived column stats structs for all partition transforms except for identity transforms since identity transforms are just the columns themselves. Note, in this model, the stats struct for data file/delete file entries may just keep the derived partition value in the lower_bound since there's no need to duplicate the same value in the upper_bound. For manifests, both upper and lower bounds can be defined since a given manifest can reference a range of partition values, and bounds can be used for pruning there.

As part of this approach, we propose using a global ID space for both field and partition field IDs. This update not only streamlines the ID system but also gives us the chance to improve metadata handling for expressions, particularly as it relates to virtual columns.

None

```
1 -> event_type field id
101 -> date(event_ts) expression field id for partition transform

1: {
    derived_value string; // for identity partitioning, never
truncated
    lower_bound string;
    upper_bound string; (upper_bound will be null for data/delete
file entries)
    value_count long;
    null_count long;
    average_uncompressed_length int;
    max_uncompressed_length int;
}

101: {
    lower_bound date;
    upper_bound date; (upper_bound will be null for data/delete
file entries)
    null_count long;
}
```

Pros:

- Given a global ID space across partition field IDs and schema field IDs, we can easily look up the stats struct for any partition field or regular field.
- For data file stats, writers can just leave the upper bound as null for columns which have an identity partition. If both lower and upper bounds are null, then the original column must be null.
- Should just work for multi argument transforms since the ID is just a partition field representing the output of the transform and the stats values are the transform value.

b.) Store the partition value as a field in the stats struct for all transforms which reference that field. In this approach for data files only a singular partition value will be stored in a naming scheme like `partition_field_id_transform`. Manifests would have lower/upper bounds for this partition value.

None

```
1 -> event_type field id
2 -> event_ts field id

1: {
    lower_bound string; // if identity partitioned, this is used to
construct the partition tuple
    upper_bound string;
    value_count long;
    null_count long;
    average_uncompressed_length int; // generated for variable
length types
    max_uncompressed_length int;
}

2: {
    lower_bound timestampz;
    upper_bound timestampz;
    partition_1001_ts_day date; // defined for data files
    partition_1001_ts_day_lower_bound date; // defined for
manifests
    partition_1001_ts_day_upper_bound date;
    value_count long;
    null_count long;
}
```

Pros:

- Encoding the partition field information in the source field's stats means that we do not need to worry about handling any collisions for IDs

Cons:

- It's an open question if and how this model would work for multi-argument transforms since in this approach the transformed value is associated with a single source field; this representation is at odds with a multi-arg transform.
- Writing stats is a bit more complicated since we are differentiating between fields to write for manifests vs data file stats

Why and how do we address having a global field ID space in V4?

Historically in the project, we've hit [quite a few issues](#) when it comes to partition field ID and schema field ID overlap. Generally in implementations, partition fields start at 1000 and schema fields start at 1. Combine this with the inherent assumption in many places where partition fields and schema fields are different, after 1000 fields there are collisions.

It's also important to consider ongoing work for V4 for addressing virtual columns and generated expressions where additional expressions based on column inputs will also need to be stored in metadata with IDs. Fundamentally, partition transforms are expressions on columns.

What we propose is introducing a new *expressions* field in table metadata, each with IDs that are also part of the table field ID space. Partition specs will be made of transforms, where each transform is associated with an expression. **Having this shared field ID space will allow us to consistently store stats for derived columns, including derived column for transforms or any general virtual column function; the stats structs can now be keyed by these IDs.**

Take the following example where the table is partitioned on day(ts) and bucket16(a, b). Expressions will be defined for both of these transforms. The below example also demonstrates how expressions could store

JSON

```
"schema": [{9, "ts", timestamp}, {11, "str", string}, {2, "a", int}, {3, "b", int}]
```

Partition Spec Before V4

```
"partition-spec": [{"field-id": 1000, "source-id": 9, "transform": "day", "name": "ts_day"}, {"field-id": 1001, "source-ids": [2, 3], "transform": "bucket[16]", "name": "bucket_a_b"}]
```

Partition Spec After V4

```
"partition-spec": [{"expr-id": 101}, {"expr-id": 104}]
```

```
"expressions": [
```

```

    {"expr-id": 101, "expr": {"source-id": 9, "transform": "day", "name":
"ts_day"}, "partition-field-id": 1000},
    {"expr-id": 102, "expr": {"source-id": 11, "transform": "lower", "name":
"lower_str"}},
    {"expr-id": 104, "expr": {"source-ids": [2, 3], "transform": "bucket[16]",
"name": "bucket_a_b", "partition-field-id": 1001}}
]

```

On upgrade of a table from V3 to V4, new expressions for existing transforms for the current partition specs must be defined (with IDs starting from $\max(\text{schema field IDs} + 1)$); partition specs will also be updated for each transform to have a link to its associated expression.

How do we match equality deletes to data files without partition tuples?

In the long run, if we have an effective way which allows us to remove equality deletes, then all of the following is moot, but for now we will propose a solution under the assumption that we will be preserving the ability to write equality deletes in V4.

Even though we propose to remove the explicitly materialized partition tuples, readers can still derive the partition tuple from the partition spec and the columnar stats which contain the actual values. The same indexing logic that exists today should work with modifications to derive the partition struct from the spec + the stats stored in the equality delete.

Equality delete entry stats for the partition transform derived column will be guaranteed to have a `lower_bound` for the partition value. For example, let's take a table partitioned on `identity(event_type)` and `date(event_ts)` and there is a file where the partition is ("`commit`", `06-20-2025T10:00:00.123`).

We can prove that given the equality delete stats for the transform columns and the spec itself, we can reconstitute the partition struct back into ("`commit`", `06-20-2025T10:00:00.123`).

JSON

```

Schema: <event_type 1: string, event_ts 2: timestamp>
Partitioning: (identity(1), date(event_ts))

```

```

// Column stats for event_type
event_type {

```

```

    lower_bound string= "commit";
    ...
}

// Derived Column stats for date transform on event_ts
date_event_ts {
    lower_bound date = 06-20-2025
    ...
}

```

If there's an equality delete on some records where partition is equal to "commit" and the day transform is 06-20-2025, presuming the sequence number is greater than a given data file(s), we will be able to determine that the equality delete must be indeed be applied as we already do.

V4 Upgrade Path

After upgrading to V4, older style manifest lists/manifests will co-exist with the new proposed structure.

We should be able to support an upgrade which does not require rewriting older manifest lists/manifests.

On upgrade, a new table metadata json would be written out including all the above proposed changes for modeling partition transforms as expressions. On any subsequent write, a new root manifest would be produced with whatever new data/delete files produced from the write and the older style manifests would be referenced as leaf manifests. Over time, the older manifests can age out and would be cleaned up as part of snapshot expiration. Users that want to eagerly move older manifests into the newer structure to get the benefits could run a rewrite manifest operation to produce columnar manifests with the new representation.

Questions

1. Do we need to allow non-inline leaf manifest DVs in the root manifest? Assuming 5% density, a one-million entry bitmap would require a 100KB roaring bitmap. If we only allow inline leaf manifest DVs these roaring bitmaps will need to be copied around for every commit.
 - a. Related to this, are inline leaf manifest DVs going to be required to be compressed? Will need to run some tests to figure out the metadata storage size vs compression/decompression overhead tradeoffs.

Discarded Alternatives

- 1.) Buffering changes to the metadata.json itself. Instead of a root level manifest, writers would write new file references to metadata.json in some field. This was discarded because having potentially unbounded content in the metadata.json is risky for the following reasons:

- a.) Metadata.json would essentially grow as the table data grows. Of course this can always be flushed and cleaned, but this is additional table maintenance burden that we want to move away from
- b.) Catalog load table latency would be variable depending on the size of the file which we've generally strayed away from

All in all, it seems the best to keep the root level metadata.json independent of the underlying table metadata/data size.

Next Steps

1. If there's general agreement, start working on a prototype and collecting numbers around when we should flush to leaf manifests so that we have a sane set of defaults from the beginning.

Appendix

References

- [Column Stats Improvements](#)