Iceberg Single File Commits

Iceberg Single File Commits

Authors: Russell Spitzer, Yi Fang, Steven Wu
Authors from Adaptive Metadata Tree Proposal: Amogh Jahagirdar amoghj@apache.org, Ryan Blue blue@apache.org, Anoop Johnson anoop@apache.org, Daniel Weeks dweeks@apache.org

With discussion with many members of the community.

Motivation

Currently, an Iceberg commit requires writing at least 3 files: 1 metadata.json, 1 manifest list and 1 manifest file, with metadata.json potentially generated on the Catalog (for Iceberg Rest Catalogs.) For a small write, the metadata authoring can be the longest step of the commit. The difference between ManifestLists and Manifests also complicates the code base and makes it impossible to surface non-partition column metrics to a higher level of scan planning.

In addition, the current method for deleting a file requires locating the file in an existing manifest file from the base snapshot's metadata, and then rewriting the whole manifest file in a copy-on-write fashion. In the worst case, an operation will need to rewrite all existing manifest files for deleted files, making the metadata changes proportional to the size of the table rather than the set of changes. For the same reason, the current design makes caching of manifests difficult since files are often being replaced even if their contents are not significantly changed.

To alleviate these issues we propose a new metadata structure for V4 which abandons the manifest list and replaces it with a special type of manifest, a Root Manifest, and allows for the application of delete vectors to manifests.

Goals

1. **Reducing metadata write amplification for smaller writes**
    a. Smaller operations only require writing a single new metadata file on the client
    b. Reduce metadata write amplification in removals/replacements in manifest entries
2. **Generating Manifests Proportional to the Size of the Operation**
    a. When generating a new snapshot, the number of manifests written should ideally be proportional to the size of the operation (i.e., the number of files added and deleted).
    b. When loading metadata for a new snapshot, if the metadata of a previous snapshot has been read and cached by an engine, the number of manifests read to build the new snapshot's metadata should also be proportional to the DML operation's size. This would improve the cachability of manifests.
3. **Aggregate Metrics**

    a. Unlike manifest lists, the new top level manifest should have aggregate metrics for all child manifests

4. **The Root Manifest reflects the last change to the Table**
    a. The changes from any operation should be expressed in only the changes made to the Root manifest

Non-Goals

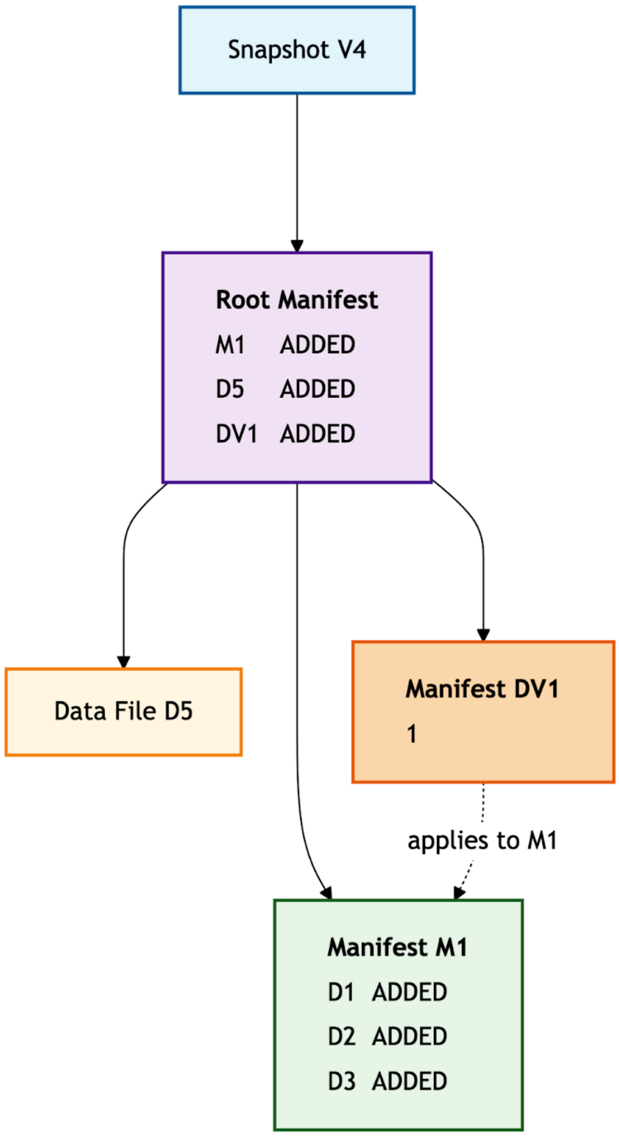
1. **Removal of Metadata.json**
    a. While this could help in latency, this document focuses on client behaviors

Proposal

The basic layout for any new snapshot starts from a “Root Manifest” which can have entries which map to Data Manifests, Delete Manifests, Data Files, Data DVs, and Manifest DVs. New writers are always required to create a new Root Manifest which replaces the previous Root Manifest and encodes the changes made by the operation.

Manifest Lists are no longer used or written.

Data manifest and delete manifests are **leaf** manifest files. They can’t link to other data or delete manifest files. This enforces a hierarchy with at most two levels.



Root Manifests

1. Root Manifest files will have the following entry types:
    a. Data Manifest
        i. Reference data files
    b. Delete Manifest
        i. Reference Data Deletion Vectors or Equality Deletes
    c. Data File
        i. A file containing either new or existing data for the table
    d. Data Delete Vector
        i. Removes rows from an existing Data File

        ii. It points to a data DV blob in a Puffin files (As in V3, no in-lined delete vectors)
    e. Manifest Delete Vector
        i. A vector deleting rows from a leaf Data or Delete Manifest
        ii. These are only allowed at the Root Manifest level
        iii. It can be an inline DV blob or a pointer to a DV blob in a Puffin file
    f. Equality Deletes
2. Every snapshot has exactly one Root Manifest in the metadata tree
3. All stats are present for both data file entries and manifest file entries. A manifest file entry gets aggregated stats of its data file contents.
4. Any changes to underlying data are marked as ADDED and anything which does not change the actual data of the table is marked as EXISTING.

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 V4 it seems better to start with the 2-level tree. With larger leaf manifests, the 2 level tree scales well into the billion file scale.

Commit Procedure

Any new commit starts by using the information in the previous snapshot's Root manifest file, this file is then modified based on the changes being applied by the new commit.

Adding data files / delete files

A new Root Manifest is created with either an ADDED Data File entry or a new ADDED Data Manifest Entry mapping to a new leaf Data Manifest with the ADDED Data File entry. The engine can determine the optimal layout for the given operation. All previously existing entries are marked as EXISTING. The same procedure is used for Equality Deletes or Delete Vectors.

Removing a data file / delete file or adding a Delete Vector for a Root Manifest Data File

A new Root Manifest is created with either an ADDED Delete Vector which removes the data or delete file from a child manifest. We do not allow the copy on write behavior (Merging Snapshot Producer) of removing and adding a child manifest with a net change to the data (including completely deleting a leaf manifest.) Whenever a change is made to leaf manifests, it should be marked by newly added Manifest Delete Vectors. Manifests (or delete vectors) which have not changed in the commit can be compacted or removed.

Manifest Compaction

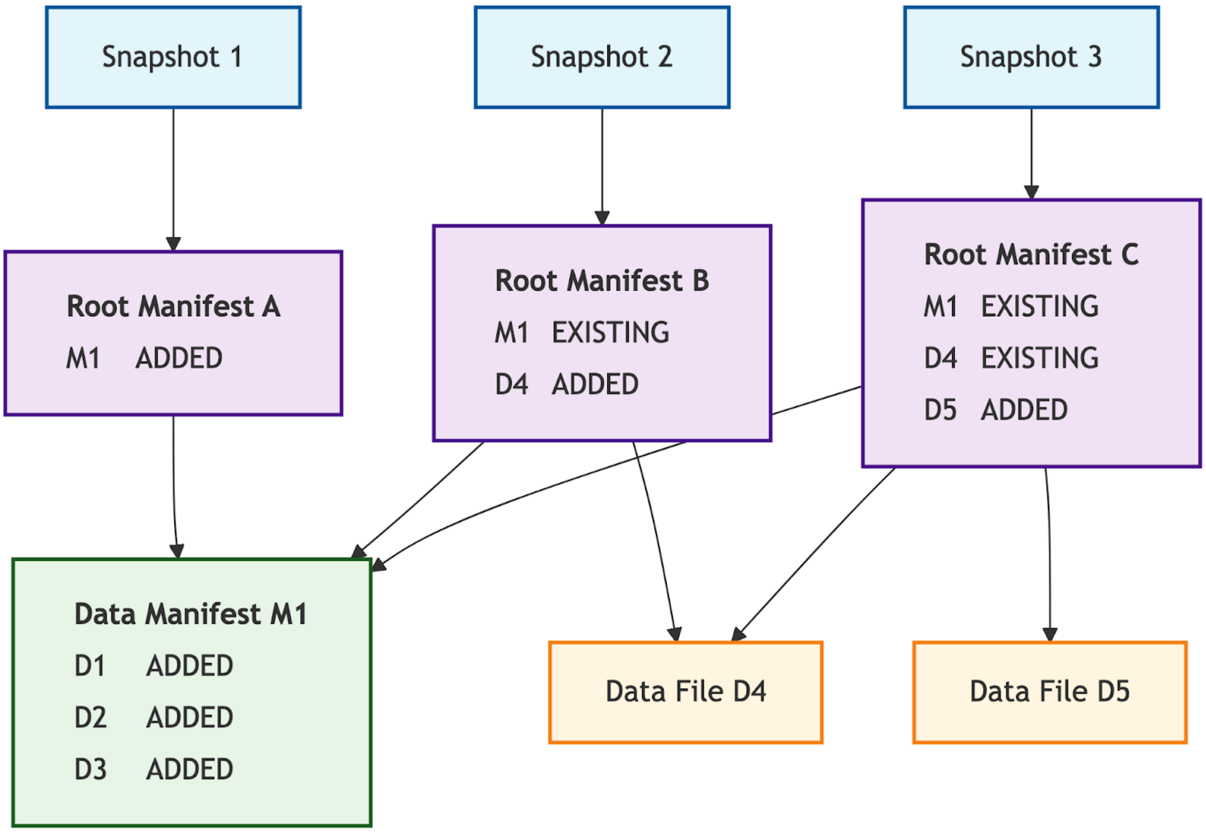
During any commit, existing Data Files, Manifests and Manifest Delete Vectors can be combined to reduce the total number of children under the root manifest. For example, two existing Data Manifests can be combined into a single Data Manifest which would also be marked as existing. Any compactions performed must not create files which cause a net change to table state. Combining an existing delete vector with its target is allowed, but combining entries for new deletes produced or new data files created in the snapshot with an existing manifest to create a new manifest is not allowed.

Examples

Append Files

Appends To the Root Manifest

Appends can be done by rewriting only the top level manifest without adding any additional child manifests.



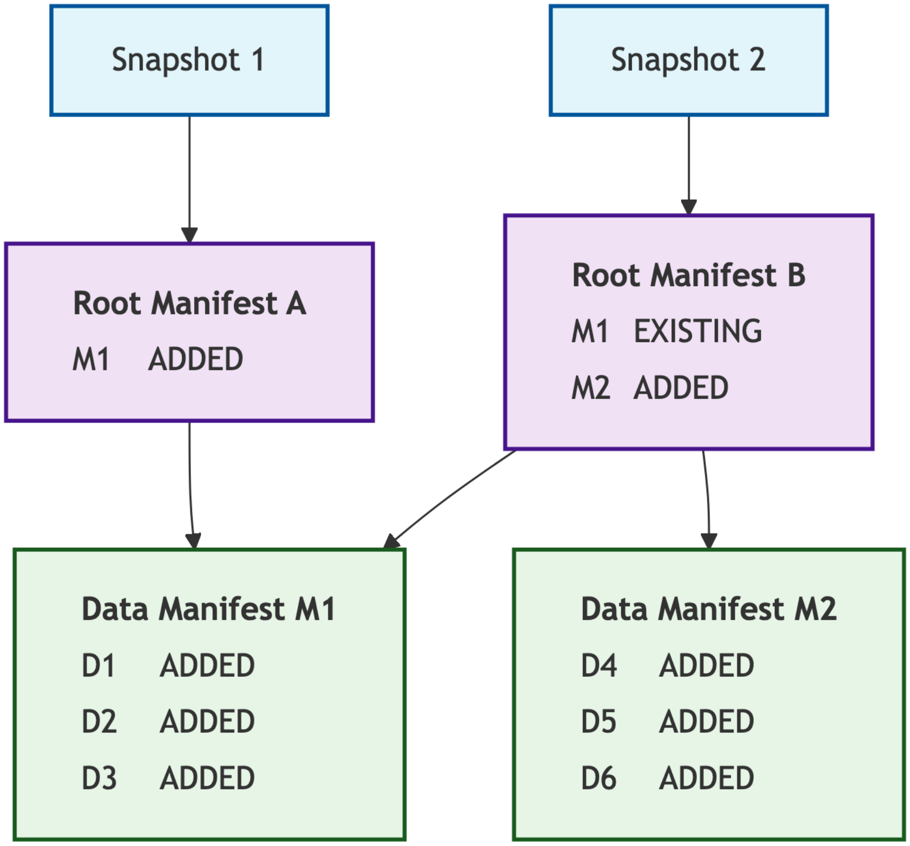
The original state of the table is shown in Snapshot 1, there are 3 data files present.

In Snapshot 2, data file D4 is added by writing a new Root Manifest which contains an entry for the existing data manifest and adding a new root level data file.

In Snapshot 3, data file D5 is added to the table by again rewriting the previous Root Manifest. The new Root Manifest C contains existing entries for the content of Manifest B as well as a new added manifest entry for data file D5.

Appends to the Root Manifest using new Data Manifests

Larger appends can still be done by adding new child manifests directly. This is basically the same as the behavior in previous versions of Iceberg where the Root manifest takes the place of the Manifest List.



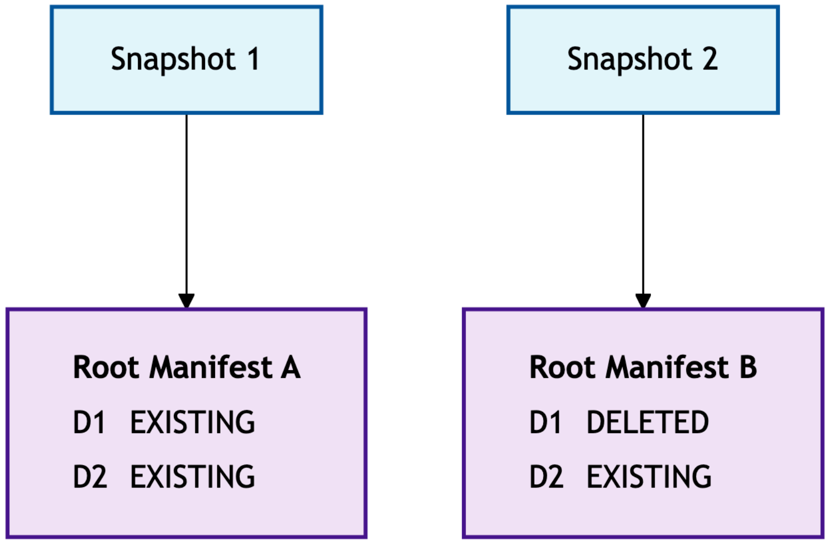
Snapshot 1 is linked to Root Manifest A which has a single ADDED Data Manifest Entry M1. Leaf Data Manifest M1 has three rows (D1, D2, D3) each representing a Data File.

Snapshot 2 is linked to Root Manifest B which has preserved the entry for M1 but also adds a new top level ADDED Data Manifest Entry M2 which contains three rows (D4, D5, D6)

Remove Files

Removing a File From the Root Manifest

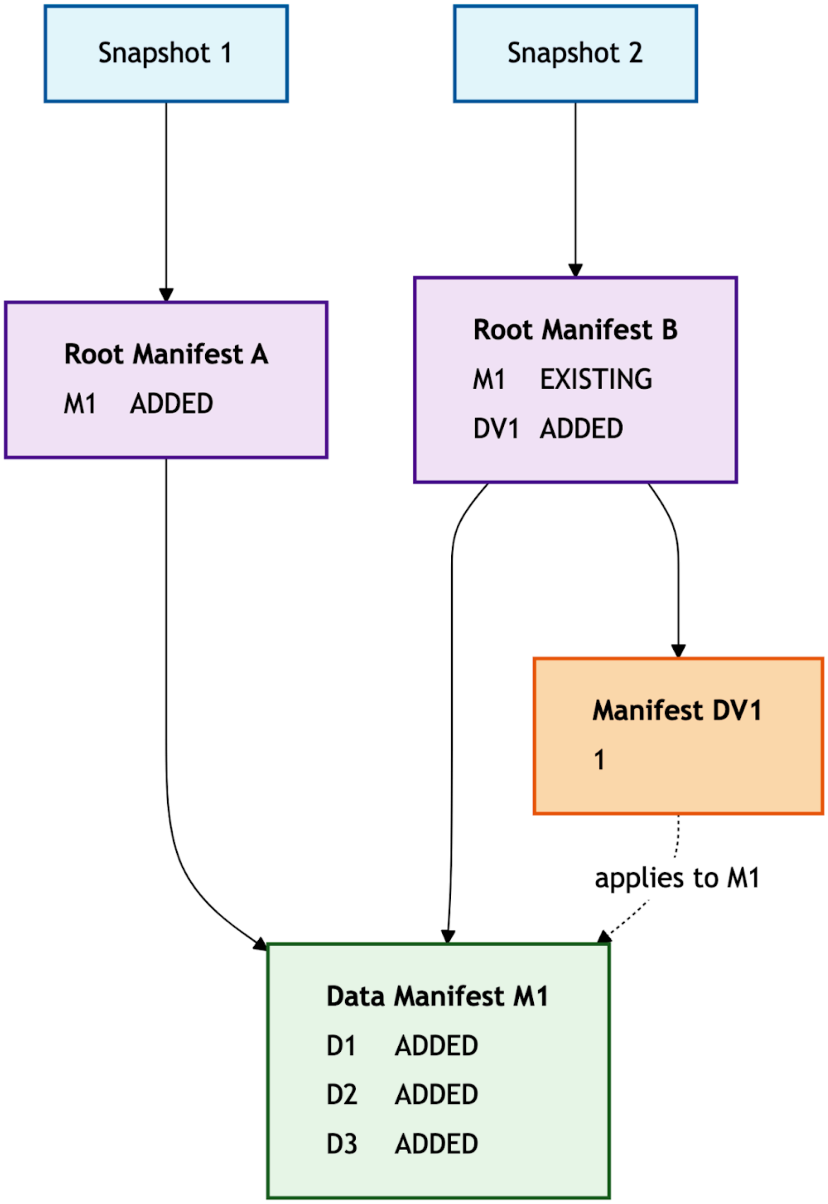
A file in the Root Manifest can be removed by producing a new Root Manifest where the file is marked as deleted.



Snapshot 1 begins with existing files D1 and D2 at the Root Manifest level. In Snapshot 2, D1 is removed by replacing it with a DELETED entry. D2 Remains unchanged.

Removing a File From a Leaf Manifest

Removing a file from a leaf manifest requires creating a Manifest Delete Vector which marks the row in the Data Manifest containing that file as deleted.



Snapshot 1 is linked to Root Manifest A which has a single ADDED Data Manifest Entry M1 which in turn links to Data Manifest M1 which has three rows (D1, D2, D3) each representing a Data File.

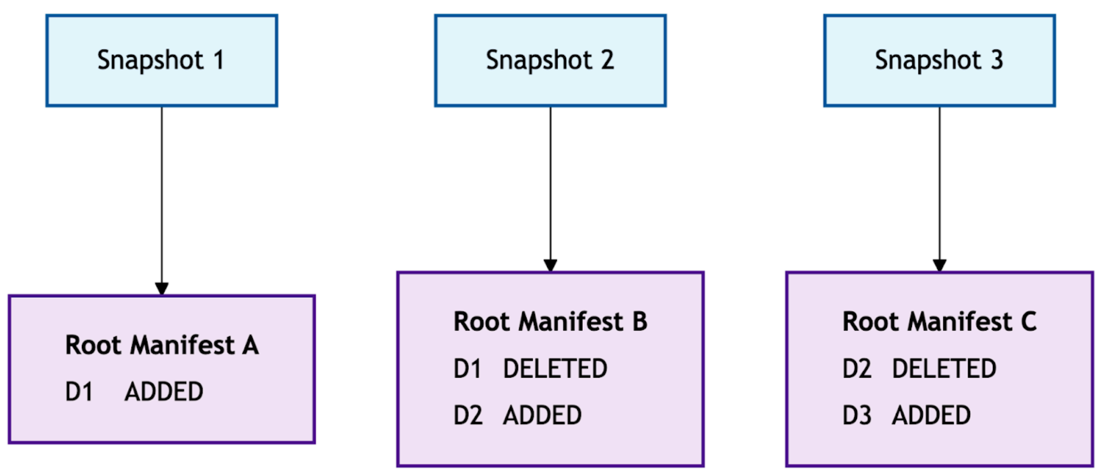
Snapshot 2 removes a file D2 by preserving the entry for M1 and then adding a new entry for a

Manifest Delete Vector (DV1) which applies to M1. The contents of DV1 mark the row at ordinal 1 (D2) as being removed.

Overwrite Files

Overwrite Files in Root Manifest

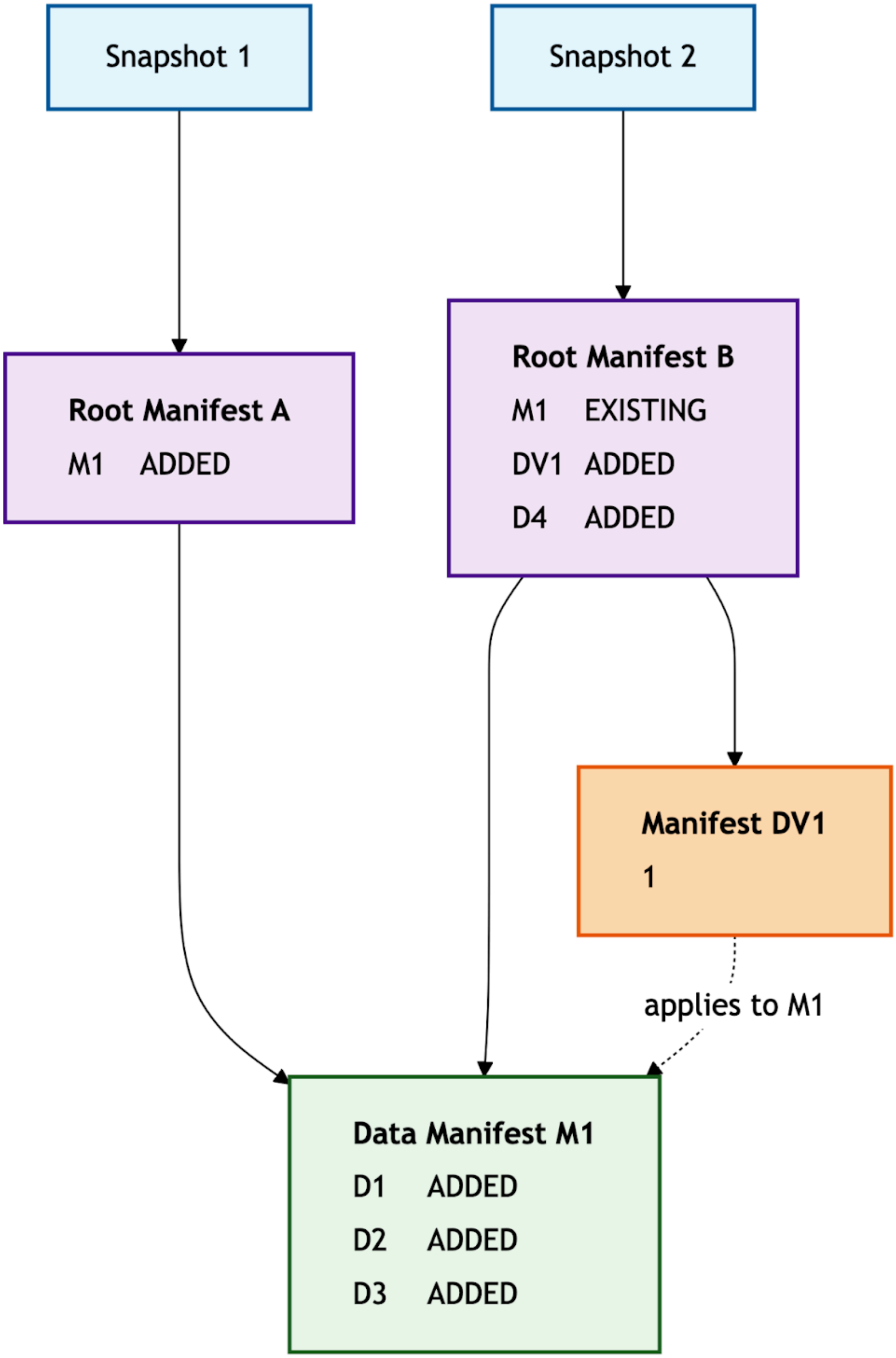
An Overwrite (Adding Files and Removing Files) can be done at the Root Manifest level without any additional metadata files by changing the entries of the Root Manifest.



Snapshot 1 begins with a single file recorded in the root manifest. Snapshot 2 replaces this file by creating a new root manifest that removes the original data file D1 and adds the rewritten data file D2
Snapshot 3 shows another replacement, removing the file added in the second snapshot with another brand new file.

Overwrite Files in in Leaf Manifest

An overwrite can also use a combination of adding files at the Root Manifest and within Manifests to represent the replacement of files within the table.

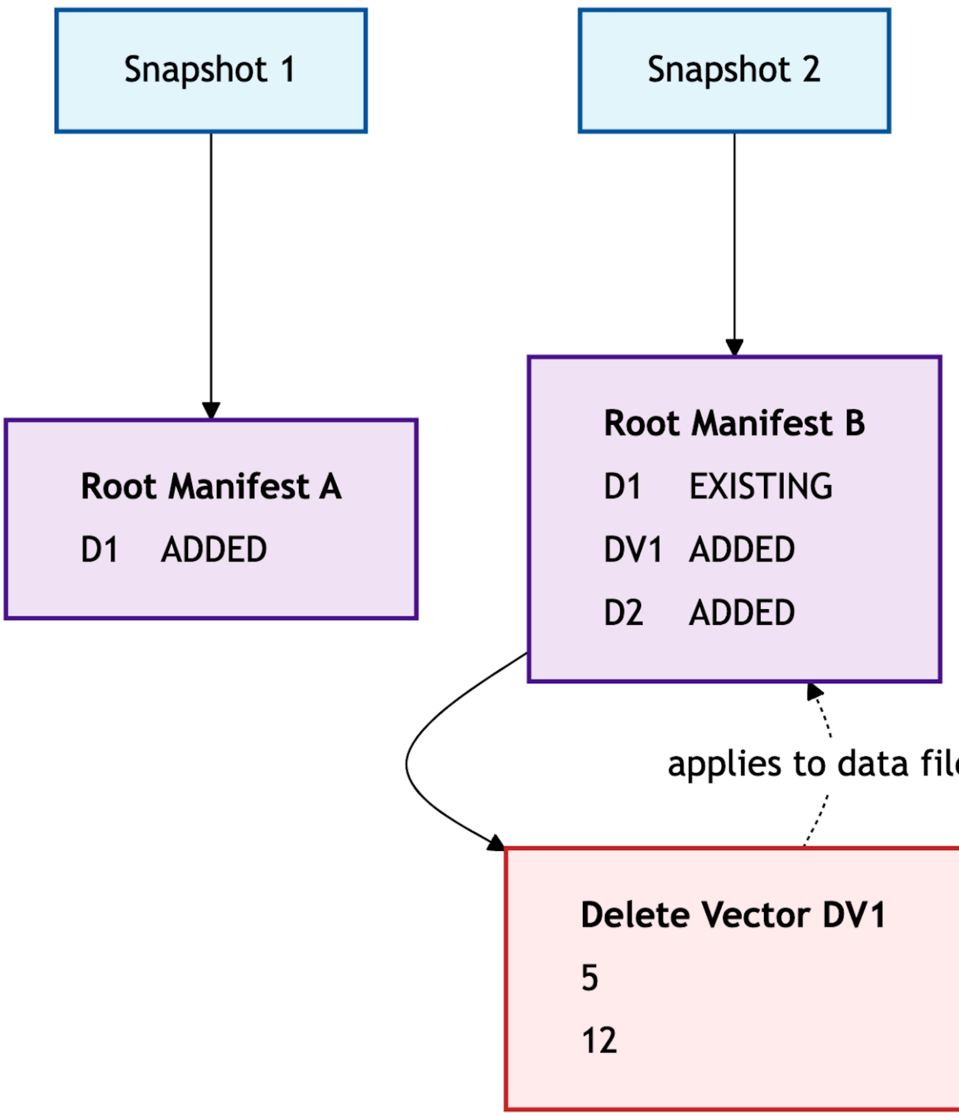

Snapshot 1
Snapshot 2
Root Manifest A
M1 ADDED
Root Manifest B
M1 EXISTING
DV1 ADDED
D4 ADDED
Manifest DV1
1
applies to M1
Data Manifest M1
D1 ADDED
D2 ADDED
D3 ADDED

Snapshot 1 shows the root manifest with a data manifest containing 3 data files (D1, D2, D3). Snapshot 2 replaces the D2 by adding a manifest delete vector (DV1) for the data manifest (M1) which removes D2. D2's replacement D4 is added at the Root Manifest level.

Row Delta

Root Manifest Only Row Delta

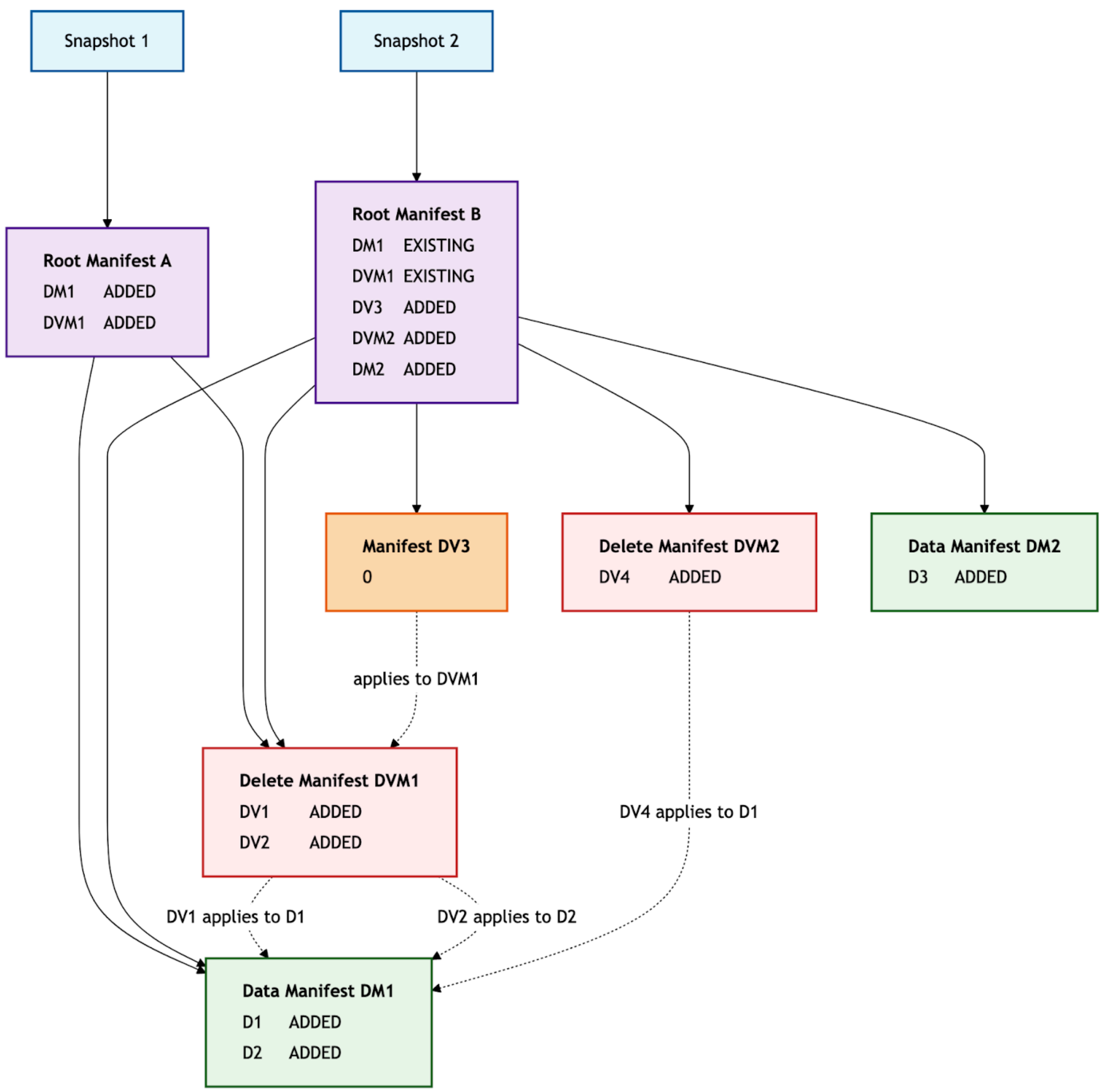
A Row Delta can be performed at the Root Manifest level by adding a root level file delete vector and adding a new root level file.



Snapshot 1 shows a Root Manifest with a single data file D1. In Snapshot 2, D1 is modified by data file delete vector DV1 and new rows are added in D2.

A Complex Row Delta with Modified Delete Vectors

A Row Delta can also be applied to a snapshot with existing delete vectors in delete manifests as well as data manifests. All delete files are removed with delete vectors applied to delete manifests, new delete vectors are created in new delete manifests (or at the Root.) Updated data files can be created in new data manifests (or at the Root.)



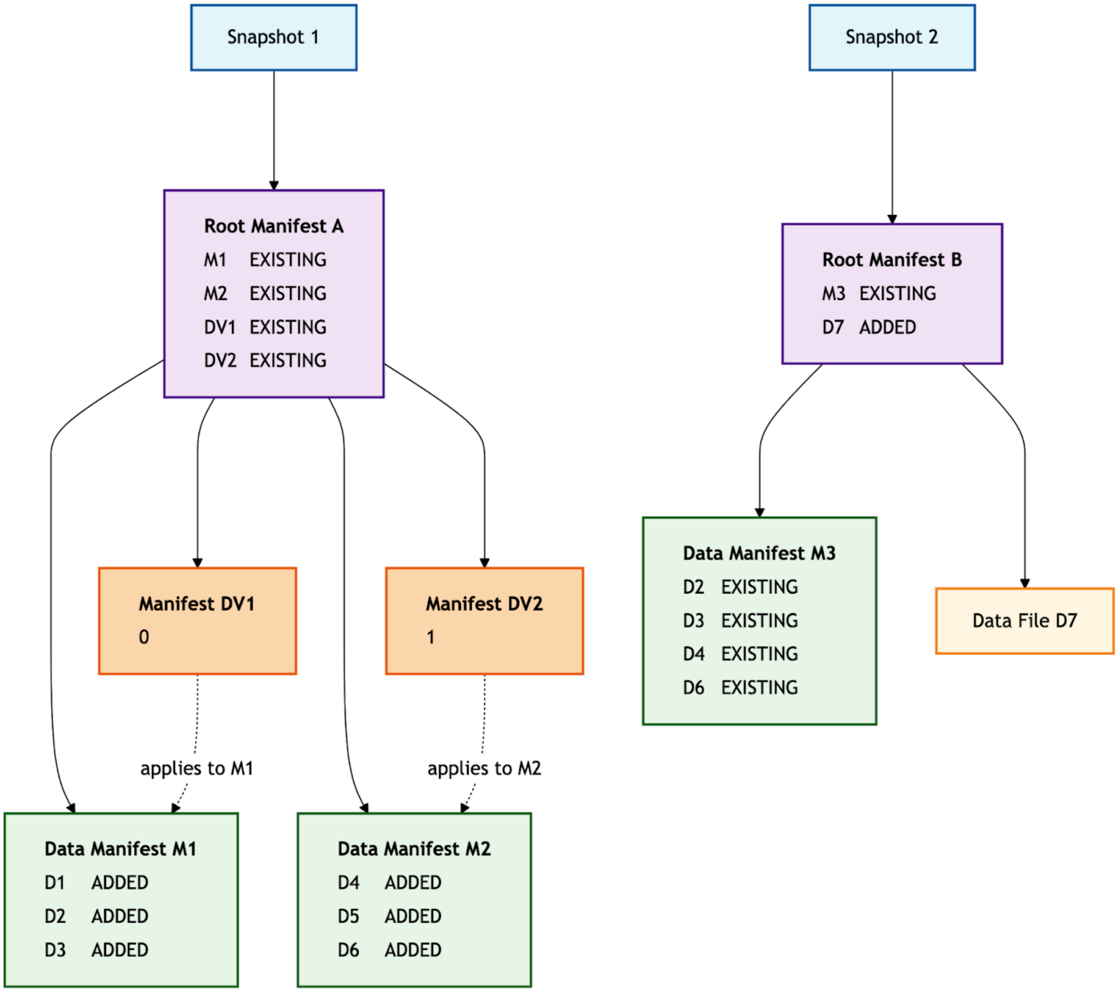
Snapshot 1 Shows a Root Manifest with a data manifest DM1 and a delete manifest DVM1. DM1 contains a data file D1 and D2. D1 has been affected by the delete vector DV1 and D2 by DV2 in DVM1.

Snapshot 2 updated additional rows in D1 by adding a new manifest delete vector DV3 at the root level which deletes the delete vector DV1. A new delete manifest DVM2 adds DV4 which applies to the data file D1. This contains the deletes from DV1 and the new deletes in this operation. The updated rows are added to a new data files D3 which is added as part of a new data manifest DM2.

Manifest Compaction

Manifest Compaction During an Append

Delete vectors can be applied and manifests can be compacted as long as their results are marked as existing. The following example shows this occurring during an append operation but it can happen for any new snapshot operation.



Snapshot 1 represents a table which has accumulated several manifest delete vectors modifying existing manifest files. Snapshot 1 points to Root Manifest A which contains several existing data manifests M1 and M2 as well as existing manifest delete vectors DV1 and DV2. M1 contains data files D1,D2,D3 and M2 contains D4, D5, D6. DV1 applies to M1 and removes the row at ordinal 0. DV2 applies to M2 and removes the row at ordinal 1.

In Snapshot 2 a new file is added to the Root manifest while simultaneously compacting the existing manifests. Snapshot 2 points to a new Root Manifest B which contains M3 (Existing), and added data file D7. M3 contains entries for existing data files D2,D3,D4 and D6.

Optional 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 perform a 2-phase planning to match data files against their DV to apply. To reduce the cost of the two phase planning, we propose an optional 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, a small number of unaffiliated delete manifests can be maintained. This means that during planning, given a set of data manifests to read both the affiliated delete manifests and the unaffiliated delete manifests whose entries may match based on stats must be read.

Pros:

- Single-pass planning: readers can do a parallel colocated planning of data and delete manifests.
- Statistics-based pruning works on delete manifests: we only need to open the delete manifests of unpruned data manifests.
- Preserving separation of data/delete manifests still enables low write amplification when replacing 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. However, this behavior is no worse than the current state.

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. This approach was discarded because of the high write amplification to replace DVs, since the data files and the statistics need to be rewritten as well.

Proposed Manifest Structure

Summary of Changes

1. Every entry in a manifest is now referred to as a content entry.
2. Added a new type, MANIFEST_DV. MANIFEST_DV can only be defined in the root manifest.
3. Defined a *deletion_vector* struct nested in the entry which encapsulates both inline and out-of line content. This struct must be defined for types 1 (position deletes) and 5 (manifest dvs)
4. Added a *version_info* struct nested in the entry which encapsulates the snapshot, status and sequence number information for the entry. This enables being able to access the fields for the entry (instead of more complicated inheritance strategies that are used

today) AND gives an isolated structure that can be modified (e.g. changing state) as desired.

5. Added *manifest_stats* struct nested in the entry for encapsulating added/removed/existing files/row counts AND min_sequence_number for manifest. This must be defined when the type is a manifest, and null otherwise

V4 content entry structure

Field ID	Name	Type	Required or Optional	Description
134	content_type	int	`required int` with meaning: 0: DATA 1: POSITION_DELETE 2: EQUALITY_DELETE 3: DATA_MANIFEST 4: DELETE_MANIFEST 5: MANIFEST_DV	Type of content stored by the data file: data, equality deletes, or position deletes (all v1 files are data files). Content types 3, 4 and 5, can only be defined in the root manifest.
100	location	string	required	Location of the file.
101	file_format	string	required	String file format name, `avro`, `orc`, `parquet`, or `puffin`
5	version_info	struct	optional	See version_info struct below. Groups information like status, and snapshot, sequence number
147	deletion_vector	struct	optional	See deletion_vector struct below. Must be defined if content type is 1 or 5. Must be null for all other types.
148	partition_spec_id	int	required	ID of partition spec used to write manifest or data/delete files.
140	sort_order_id	int	optional	ID representing sort order for this file. Can only be set if content_type is 0.
103	record_count	long	required	Number of records in this file, or the cardinality of a deletion vector

104	file_size_in_bytes	long	optional	Total file size in bytes. Must be defined if location is defined
149 (individual fields in content_stats struct will have their own IDs)	content_stats	struct	optional	Stats struct Column Stats Improv…
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), min_sequence_number(516) Must be set if content_type is 3 or 4, otherwise must be null
143	referenced_file	string	optional	Location of data file that a DV references if `content_type` is 1 or 5. Location of affiliated data manifest if `content_type` is 4 or null if delete manifest is unaffiliated.
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 must be null otherwise. Fields with ids listed in this column must be present in the delete file
142	first_row_id	long	optional	The `_row_id` for the first

				row in the data file if content_type is 0. If content_type is 3, this is the starting `_row_id` to assign to rows added by `ADDED` data files. See First Row ID Inheritance

V4 content entry version_info struct

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 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.
3	sequence_number	long	optional	Carried over from current format: Data sequence number of the file. Inherited in when null and status is 1 (added). Must be equal to file_sequence_number if type is 3 or 4.
4	file_sequence_number	long	optional	File sequence number indicating when the file was added. Inherited when null and status is added. Must be equal to sequence_number if type is 3 or 4.

V4 content entry deletion_vector struct

144	offset	long	optional	The offset in the file where the content starts.
145	size_in_bytes	long	optional	The length of a referenced content stored in the file; required if `content_offset` is present.
146	inline_content	binary	optional	Serialized bitmap for inline DVs.

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.

How existing manifest list fields map to Proposed V4 content entry fields

Manifest list field	V4 content entry field	Rationale or description
manifest_path	location	Shared location field for data files, DVs, equality deletes and manifests.
manifest_length	file_size_in_bytes	Shared file size field for data files, DVs, Equality deletes, and manifests
partition_spec_id	partition_spec_id	Still require partition spec for equality delete matching
content	type	Renamed to type since this is just an enumeration, and the

		original name of content was misleading that it contained the actual content
sequence_number	version_info.sequence_number	Moved to version_info structure
added_snapshot_id	version_info.snapshot_id	Shared with snapshot_id
min_sequence_number	manifest_stats.min_sequence_number	Moved to manifest_stats struct in content entry
added_files_count	manifest_stats.added_files_count	Moved to manifest_stats struct in content entry
existing_files_count	manifest_stats.existing_files_count	Moved to manifest_stats struct in content entry
deleted_files_count	manifest_stats.deleted_files_count	Moved to manifest_stats struct in content entry.
added_rows_count	manifest_stats.added_rows_count	Moved to manifest_stats struct in content entry.
existing_rows_count	manifest_stats.existing_rows_count	Moved to manifest_stats struct in content entry.
deleted_rows_count	manifest_stats.deleted_rows_counts	Moved to manifest_stats struct in content entry
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 content_entry.key_metadata
first_row_id	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 content entry fields

Manifest Entry field	V4 content entry field	Rationale or description
data_file	No separate struct element, the entire record for an entry in a V4 manifest will now be referred to as a content entry.	Manifest entries now have more generic content than prior versions. An entry can either be a data file, DV, equality delete or a data/delete manifest. As a result, the entire structure of a record in a manifest has been renamed to content entry.
status	version_info.status	
snapshot_id	version_info.snapshot_id	
sequence_number	version_info.sequence_number	(data sequence number)
file_sequence_number	version_info.file_sequence_number	
data_file.file_path	location	Renamed, same ID
data_file.file_format	file_format	(Parquet, Avro, ORC, Puffin)
data_file.record_count	record_count	
data_file.file_size_in_bytes	file_size_in_bytes	
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.partition	REMOVED	Represented in column stats (need to support translation for equality deletes)Represented in column stats (need to support translation for equality deletes)
data_file.key_metadata	key_metadata	Carried over, still needed for encryption
data_file.split_offsets	split_offsets	Carried over
data_file.sort_order_id	sort_order_id	Carried over
data_file.referenced_data_file	referenced_file	Renamed to referenced_file to be able to support delete manifests expressing affinity to a data manifest.
data_file.content_offset	deletion_vector.offset	Grouping offset and content size required for DVs into a deletion_vector struct. Renaming to just offset because *content* is redundant in the context of being grouped under deletion_vector.
data_file.content_size_in_bytes	deletion_vector.size_in_bytes	Grouping offset and content size required for DVs into a deletion_vector struct. Renaming to just size_in_bytes because *content* is redundant in the context of being grouped under deletion_vector.
data_file.equality_ids	equality_ids	Carried over since we still need to be able to express in metadata which field IDs are stored in the delete file.

Upgrade

Migrating from a V3 Manifest List to a V4 Root Manifest requires just propagating every Manifest List Entry into a V4 RootManifest entry. The first commit to a V4 table must perform this migration. A V4 Root Manifest may have manifest delete vectors which apply to V3 manifests.

Open Questions:

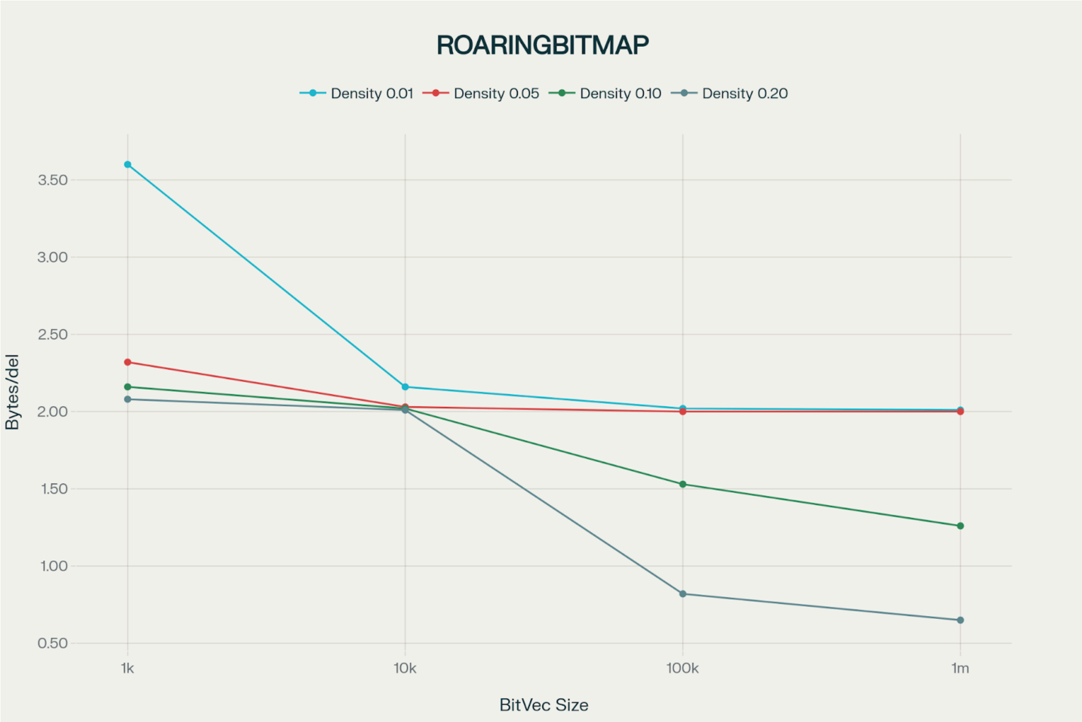
1. Are Manifest Delete Vectors always stored inline, Puffin, or possibly both?
   a. Delete Vectors in the Root manifest could potentially be stored in the Root Manifest itself either inlined with Manifest Entries or as independent Puffin Delete Vectors. While inlining would increase the size of the Root Manifest, it would allow for faster writes when doing small MOR operations. Serialized roaring bitmaps yield 2 bytes per deleted position in the manifest DV which does give buffer in the root to allow more writes to target in the root manifest. In a columnar structure, multiple runs of manifest DVs can compress even better. At a certain point, arguably it's better to just rewrite the manifest rather than accumulating manifest DVs in the root.
2. Optional Manifest Affinity?
   a. In the current proposal, an optional affinity between data and delete manifests was proposed. In this model, a delete manifest may be associated with a data manifest. Affinity removes the overhead from a 2-phase planning at the cost of more flexible writes. An optional affinity allows for writers to write unaffiliated delete manifests, but at planning those will need to be reconciled.
3. Change Detection at the Root
   a. One desired property in this new metadata tree is the ability to do incremental change detection in the root manifest for a given snapshot. Given the presence of manifest DVs, there are multiple potential ways to encode certain delete or rewrite scenarios. One possible approach is to have a DV which also encodes the *diff* in positions from the previous DV. Another open question here is whether this DV should be a separate entry in the manifest or should it be attached to the manifest entry itself as a separate column.
4. Should we remove the partition tuple from file entries and the partition field summaries from the root manifest?
   a. In the current format, the manifest list and data files include partition tuples in their entries. In the proposed V4 structure, arguably neither the root manifest does not need an explicitly materialized partition field summary in the entry nor do data file entries need an explicit materialized partition tuple. The primary benefit of removing the partition tuple is that with the presence of columnar statistics is that it is redundant metadata and given an effective way to correctly rebuild the tuple when needed (e.g. equality delete matching), column stats is sufficient. Nearly every transform is monotonic, with the exception of bucketing. What this means is that the stats can be used just as effectively at the root level for pruning (for the bucketing case the bucket transformed values would be materialized in the stats struct as stats on a derived column). There are nuances (exact bounds, truncation, how derived column stats are stored); this is elaborated in the adaptive metadata tree proposal but before that, it'd be good to establish if removing the partition tuple is a good idea.

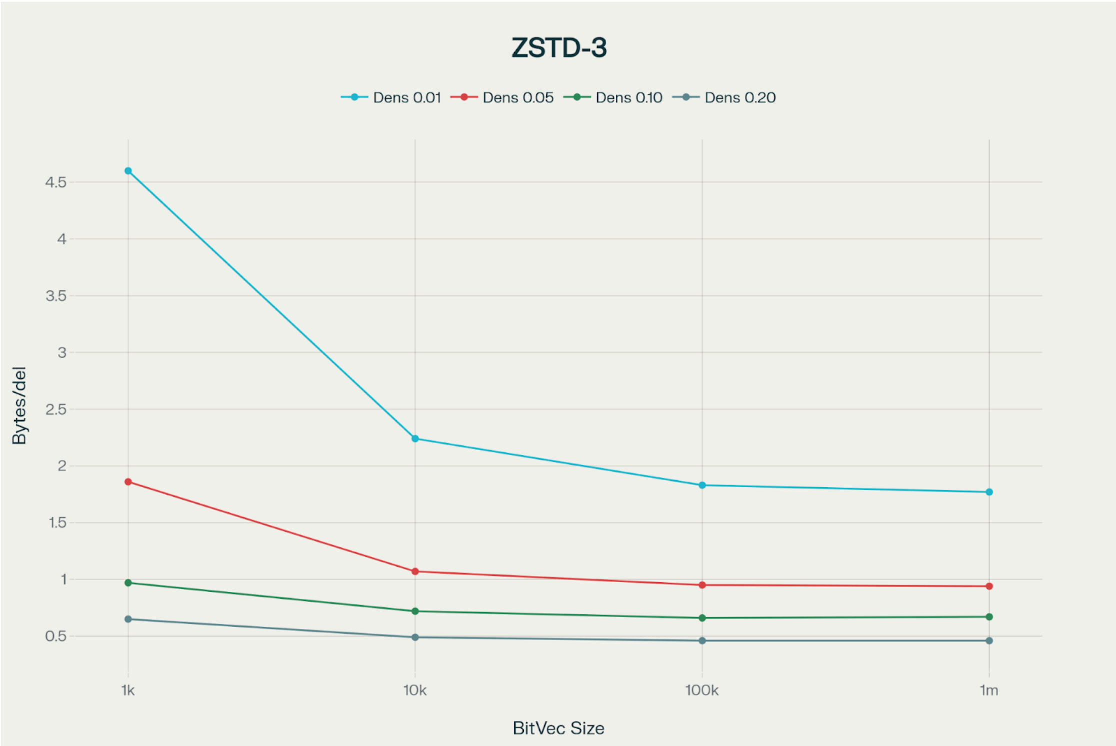
Appendix

Compressed Sizes (bytes) of Bit Vectors by Algorithm, Density, and Scale for Random Positions

In parens is bytes/delete

Method	Density	1,000	10,000	100,000	1,000,000
LZ4	0.01	57 (5.70)	383 (3.83)	3,545 (3.54)	32,713 (3.27)
	0.05	98 (1.96)	920 (1.84)	8,480 (1.70)	75,451 (1.51)
	0.1	130 (1.30)	1,155 (1.16)	10,600 (1.06)	108,987 (1.09)
	0.2	135 (0.68)	1,264 (0.63)	12,541 (0.63)	125,493 (0.63)
Roaring Bitmap	0.01	36 (3.60)	216 (2.16)	2,024 (2.02)	20,136 (2.01)
	0.05	116 (2.32)	1,016 (2.03)	10,024 (2.00)	100,136 (2.00)
	0.1	216 (2.16)	2,016 (2.02)	15,340 (1.53)	126,440 (1.26)
	0.2	416 (2.08)	4,016 (2.01)	16,408 (0.82)	129,918 (0.65)
ZSTD-3	0.01	46 (4.60)	224 (2.24)	1,826 (1.83)	17,690 (1.77)
	0.05	93 (1.86)	533 (1.07)	4,727 (0.95)	46,987 (0.94)
	0.1	97 (0.97)	715 (0.72)	6,599 (0.66)	67,170 (0.67)
	0.2	129 (0.65)	981 (0.49)	9,227 (0.46)	91,923 (0.46)
Raw Bitset (dense encoding)	0.01	123 (12.3)	1,247 (12.47)	12,485 (12.48)	124,980 (12.50)
	0.05	123 (2.46)	1,247 (2.49)	12,494 (2.50)	124,985 (2.50)
	0.1	123 (1.23)	1,247 (1.25)	12,500 (1.25)	125,000 (1.25)
	0.2	125 (0.63)	1,250 (0.63)	12,500 (0.63)	125,000 (0.63)


ROARINGBITMAP
Density 0.01
Density 0.05
Density 0.10
Density 0.20
Bytes/del
3.50
3.00
2.50
2.00
1.50
1.00
0.50
1k
10k
100k
1m
BitVec Size

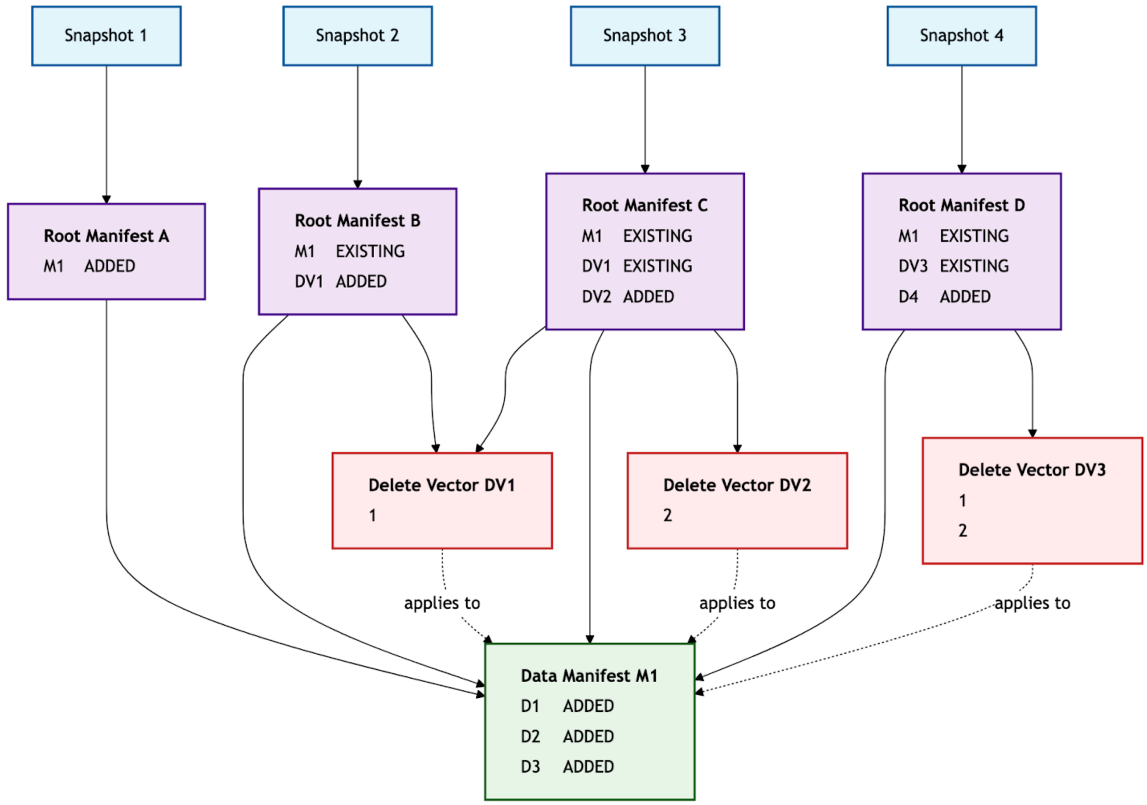

ZSTD-3
Dens 0.01
Dens 0.05
Dens 0.10
Dens 0.20
Bytes/del
4.5
4
3.5
3
2.5
2
1.5
1
0.5
1k
10k
100k
1m
BitVec Size

Discarded Approaches

Using Multiple Delete Vectors to Keep Changes Distinct

To make it easier to track differences between two different snapshots we considered storing multiple delete vectors, so each snapshot could be mapped to its changes without looking at a previous snapshot. This was rejected because the added complexity didn't outweigh the benefits which only applied to MOR writes.

By allowing for both an ADDED and EXISTING delete vector we can preserve the ability to map exactly what has occurred in the snapshot to changes in the Root Manifest. In the following example Snapshot 3 and 4 both contain two delete vectors that apply to M1, the ADDED DVs show the changes made in the respective snapshots.

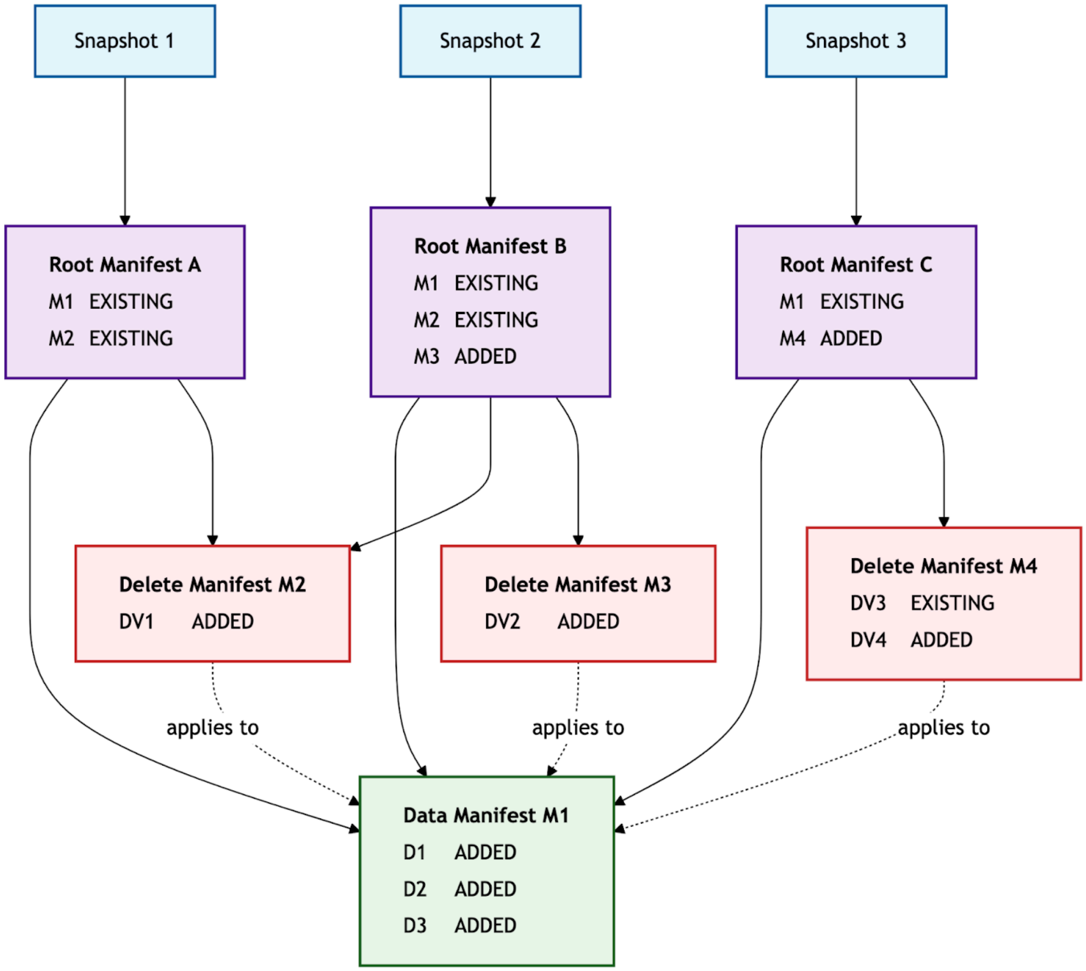


Snapshot 3 removes a file D3 by preserving the entry for M1 and the Delete Vector DV1 which are both now marked as existing. An additional DeleteVector DV2 is added which also applies to M1 and removes the row at ordinal 2.

In Snapshot 4 a new file is added to the table (D4) in the Root manifest and DV1 and DV2 are removed. They are replaced with DV3 (Existing) which removes both ordinal 1 and 2 from M1.

Modifying an existing Delete Vector in a Delete Manifest

To modify an existing Delete Vector in a Delete Manifest we need to go through an extra step to preserve the ability to differentiate changes from existing state. We start by adding a new additional delete vector but subsequent new deletes must remove and compact previous delete vectors.



In Snapshot 1 we have a Root Manifest A with existing Data Manifest M1 and existing Delete Manifest M2. M1 contains entries for D1, D2, and D3. M2 contains an entry for a Delete Vector DV1 removing rows in D2.

In Snapshot 2 we delete more rows from D2 by adding a new Delete Vector in a new Delete Manifest M3.

In Snapshot 3 we delete even more rows from D2 by adding two Delete Vectors to Delete Manifest M4. One representing the deletes present in Snapshot 1 and 2 and one representing the deletes added in snapshot 3. Delete Manifests M2 and M3 are removed.

Diagram Source

```
graph TD
    S1["Snapshot 1"] --> RMA
    S2["Snapshot 2"] --> RMB
    S3["Snapshot 3"] --> RMC

    RMA["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
A</b></td></tr><tr><td style='text-align:left'>M1</td><td
style='text-align:left'>ADDED</td></tr></table>"]
    RMB["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
B</b></td></tr><tr><td style='text-align:left'>M1</td><td
style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>D4</td><td
style='text-align:left'>ADDED</td></tr></table>"]
    RMC["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
C</b></td></tr><tr><td style='text-align:left'>M1</td><td
style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>D4</td><td
style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>D5</td><td
style='text-align:left'>ADDED</td></tr></table>"]

    RMA --> DM1["<table><tr><td colspan='2' style='text-align:left'><b>Data Manifest
M1</b></td></tr><tr><td style='text-align:left'>D1</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>D2</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>D3</td><td
style='text-align:left'>ADDED</td></tr></table>"]
    RMB --> DM1
    RMB --> D4["Data File D4"]
    RMC --> DM1
    RMC --> D4
    RMC --> D5["Data File D5"]

    classDef snapshot fill:#e1f5fe,stroke:#01579b,stroke-width:2px
    classDef rootManifest fill:#f3e5f5,stroke:#4a148c,stroke-width:2px
    classDef dataManifest fill:#e8f5e8,stroke:#1b5e20,stroke-width:2px
    classDef dataFile fill:#fff8e1,stroke:#f57f17,stroke-width:2px

    class S1,S2,S3 snapshot
    class RMA,RMB,RMC rootManifest
    class DM1 dataManifest
    class D4,D5 dataFile
graph TD
    S1["Snapshot 1"] --> RMA
    S2["Snapshot 2"] --> RMB
```

```
    RMA["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
A</b></td></tr><tr><td style='text-align:left'>M1</td><td
style='text-align:left'>ADDED</td></tr></table>"]
    RMB["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
B</b></td></tr><tr><td style='text-align:left'>M1</td><td
style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>M2</td><td
style='text-align:left'>ADDED</td></tr></table>"]

    RMA --> DM1["<table><tr><td colspan='2' style='text-align:left'><b>Data Manifest
M1</b></td></tr><tr><td style='text-align:left'>D1</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>D2</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>D3</td><td
style='text-align:left'>ADDED</td></tr></table>"]
    RMB --> DM1
    RMB --> DM2["<table><tr><td colspan='2' style='text-align:left'><b>Data Manifest
M2</b></td></tr><tr><td style='text-align:left'>D4</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>D5</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>D6</td><td
style='text-align:left'>ADDED</td></tr></table>"]

    classDef snapshot fill:#e1f5fe,stroke:#01579b,stroke-width:2px
    classDef rootManifest fill:#f3e5f5,stroke:#4a148c,stroke-width:2px
    classDef dataManifest fill:#e8f5e8,stroke:#1b5e20,stroke-width:2px
    classDef dataFile fill:#fff8e1,stroke:#f57f17,stroke-width:2px

    class S1,S2 snapshot
    class RMA,RMB rootManifest
    class DM1,DM2 dataManifest
%%{init: {"flowchart": {"htmlLabels": true}, "theme": "base", "themeVariables": {"primaryColor":
"#ffffff", "primaryTextColor": "#000000", "primaryBorderColor": "#000000", "lineColor":
"#000000", "sectionBkColor": "transparent", "altSectionBkColor": "transparent", "gridColor":
"#000000", "secondaryColor": "#ffffff", "tertiaryColor": "#ffffff", "clusterBkg": "transparent",
"clusterBorder": "transparent"}}}%%
graph TD
    subgraph " "
        S1["Snapshot 1"]
        S2["Snapshot 2"]
    end

    subgraph "  "
        RMA["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
A</b></td></tr><tr><td style='text-align:left'>DM1</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>DVM1</td><td
style='text-align:left'>ADDED</td></tr></table>"]
```

```
        RMB["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
B</b></td></tr><tr><td style='text-align:left'>DM1</td><td
style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>DVM1</td><td
style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>DV3</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>DVM2</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>DM2</td><td
style='text-align:left'>ADDED</td></tr></table>"]
    end

    subgraph rma_children ["  "]
        DM1["<table><tr><td colspan='2' style='text-align:left'><b>Data Manifest
DM1</b></td></tr><tr><td style='text-align:left'>D1</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>D2</td><td
style='text-align:left'>ADDED</td></tr></table>"]
        DVM1["<table><tr><td colspan='2' style='text-align:left'><b>Delete Manifest
DVM1</b></td></tr><tr><td style='text-align:left'>DV1</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>DV2</td><td
style='text-align:left'>ADDED</td></tr></table>"]
    end

    subgraph rmb_children ["  "]
        DV3["<table><tr><td colspan='2' style='text-align:left'><b>Manifest
DV3</b></td></tr><tr><td style='text-align:left'>0</td><td
style='text-align:left'></td></tr></table>"]
        DVM2["<table><tr><td colspan='2' style='text-align:left'><b>Delete Manifest
DVM2</b></td></tr><tr><td style='text-align:left'>DV4</td><td
style='text-align:left'>ADDED</td></tr></table>"]
        DM2["<table><tr><td colspan='2' style='text-align:left'><b>Data Manifest
DM2</b></td></tr><tr><td style='text-align:left'>D3</td><td
style='text-align:left'>ADDED</td></tr></table>"]
    end

    S1 --> RMA
    S2 --> RMB
    RMA --> DM1
    RMA --> DVM1
    RMB --> DM1
    RMB --> DVM1
    RMB --> DV3
    RMB --> DVM2
    RMB --> DM2
    DVM1 -.->|DV1 applies to D1| DM1
    DVM1 -.->|DV2 applies to D2| DM1
    DV3 -.->|applies to DVM1| DVM1
```

```
    DVM2 -.->|DV4 applies to D1| DM1

    classDef snapshot fill:#e1f5fe,stroke:#01579b,stroke-width:2px
    classDef rootManifest fill:#f3e5f5,stroke:#4a148c,stroke-width:2px
    classDef dataManifest fill:#e8f5e8,stroke:#1b5e20,stroke-width:2px
    classDef deleteManifest fill:#ffebee,stroke:#c62828,stroke-width:2px
    classDef manifestDeleteVector fill:#fed7aa,stroke:#ea580c,stroke-width:2px

    class S1,S2 snapshot
    class RMA,RMB rootManifest
    class DM1,DM2 dataManifest
    class DVM1,DVM2 deleteManifest
    class DV3 manifestDeleteVector# Diagram of snapshot relationships
```
Exported on 7/3/2025 at 12:23:29 CDT from Cursor (1.1.6)

User

Snapshot 1 is linked to Root Manifest A which has a single ADDED Data Manifest Entry M1 which in turn links to Data Manifest M1 which has three rows (D1, D2, D3) each representing a Data File.

Snapshot 2 is linked to Root Manifest B which has preserved the entry for M1 now marked as existing but adds a new top level ADDED Data File Entry referring to D4.

Snapshot 3 is linked to Root Manifest C which has preserved the entry for M1 and D4 and added a new entry D5.

Generate a diagram showing the above threee snapshots

Cursor

I'll create a diagram showing the three snapshots and their relationships to root manifests, data manifest entries, and data files.

I've created a diagram that visualizes the three snapshots and their relationships to root manifests, data manifest entries, and data files. The diagram shows:

Snapshot 1: Links to Root Manifest A with a single ADDED Data Manifest Entry M1, which points to Data Manifest M1 containing three data files (D1, D2, D3).

Snapshot 2: Links to Root Manifest B, which preserves the M1 entry (now marked as EXISTING) and adds a new ADDED Data File Entry for D4.

Snapshot 3: Links to Root Manifest C, which preserves both existing entries (M1 and D4) and adds a new entry for D5.

The diagram uses different colors to distinguish between the various components:
- Light blue for snapshots
- Light purple for root manifests
- Light orange for manifest entries
- Light green for data manifests
- Light yellow for data files

This structure clearly illustrates how Iceberg snapshots evolve, with each new snapshot building upon previous ones by preserving existing entries and adding new ones as needed.

User

Remove the labels ADDED, EXISTING from the nodes and instead add them to the edges

---

Cursor

I've updated the diagram to move the ADDED/EXISTING labels from the nodes to the edges. Now the diagram shows:

- Clean node labels with just the component names
- Edge labels indicating whether each entry is ADDED or EXISTING in that particular root manifest

This makes it clearer that the status refers to the relationship between the root manifest and its entries, rather than being an intrinsic property of the components themselves. The diagram now shows how each snapshot's root manifest references its entries with the appropriate status.

User

There should only be a single Data Manifest Entry M1 Node

Cursor

Perfect! Now the diagram correctly shows a single Data Manifest Entry M1 node that is referenced by all three root manifests. This better represents how Iceberg actually works:

- **Root Manifest A** references M1 as ADDED (first time it appears)
- **Root Manifest B** references the same M1 as EXISTING (reused from the previous snapshot)
- **Root Manifest C** references the same M1 as EXISTING (still reused)

This structure demonstrates the efficiency of Iceberg's snapshot system, where manifest entries are reused across snapshots rather than being duplicated, with only the reference status changing from ADDED to EXISTING as snapshots evolve.

User

There should only be a single Data File Entry D4

Cursor

Excellent! Now the diagram correctly shows a single Data File Entry D4 node that is referenced by both Root Manifest B and Root Manifest C. This accurately represents how Iceberg works:

- **Root Manifest B** references D4 Entry as ADDED (first time it appears)
- **Root Manifest C** references the same D4 Entry as EXISTING (reused from the previous snapshot)

The diagram now properly illustrates the efficiency of Iceberg's snapshot system where both manifest entries and data file entries are reused across snapshots rather than being duplicated. Only the reference status changes as snapshots evolve, minimizing storage overhead and maintaining referential integrity across the snapshot history.

User

Can we instead have all the Manifest Nodes represented as tables, each with a row for each child element

---

Cursor

I've created a diagram that represents the manifest nodes as tables. Each root manifest is shown as a table with:

- **Root Manifest A**: One row showing M1 as ADDED
- **Root Manifest B**: Two rows showing M1 as EXISTING and D4 Entry as ADDED
- **Root Manifest C**: Three rows showing M1 as EXISTING, D4 Entry as EXISTING, and D5 Entry as ADDED

The **Data Manifest M1** is also shown as a table with three rows listing its data files (D1, D2, D3).

This tabular representation makes it much clearer how each manifest contains multiple entries and shows their status (ADDED/EXISTING) in a structured format, providing a more detailed view of the manifest contents at each snapshot level.

User

now remove the entry nodes

Cursor

Perfect! I've removed the intermediate entry nodes and now the root manifests connect directly to their targets:

- **Root Manifest A** connects directly to **Data Manifest M1**
- **Root Manifest B** connects directly to **Data Manifest M1** and **Data File D4**
- **Root Manifest C** connects directly to **Data Manifest M1**, **Data File D4**, and **Data File D5**

The tabular representation of the manifests still shows the entry names and their status (ADDED/EXISTING), but the diagram structure is now cleaner and more direct, showing the actual references without the intermediate entry node abstractions.

User

Please Left Justify the text in the tables in the Manifest Nodes, and add status "ADDED" to all rows in the Data Manifest M1 node

---

Cursor

Perfect! I've updated the diagram with the requested changes:

1. **Left-justified all text** in the manifest table nodes using `style='text-align:left'` for better readability
2. **Added "ADDED" status** to all rows in the Data Manifest M1 node, so it now shows:
   - D1 | ADDED
   - D2 | ADDED
   - D3 | ADDED

Now the Data Manifest M1 is consistent with the root manifests in showing both the entry name and its status, and all table content is properly left-aligned for a cleaner, more professional appearance.

User

Remove the Word Entry from D4 Entry and D5 Entry

Cursor

Perfect! I've cleaned up the diagram by removing the word "Entry" from the root manifest tables. Now the entries are simply labeled as:

- **Root Manifest B**: M1 | EXISTING and D4 | ADDED
- **Root Manifest C**: M1 | EXISTING, D4 | EXISTING, and D5 | ADDED

This makes the diagram cleaner and more concise while maintaining clarity about the different types of entries (manifest references vs direct data file references) in each root manifest.

User

Using the same style and instructions as above, create the digram for this description

Snapshot 1 is linked to Root Manifest A which has a single ADDED Data Manifest Entry M1 which in turn links to Data Manifest M1 which has three rows (D1, D2, D3) each representing a Data File.

Snapshot 2 is linked to Root Manifest B which has preserved the entry for M1 but adds a new top level ADDED Data Manifest Entry M2 which contains three rows (D4, D5, D6)

Cursor

I've created a diagram that shows the new scenario with two snapshots and two data manifests. The diagram illustrates:

Snapshot 1: Links to Root Manifest A with a single entry M1 (ADDED), which connects to Data Manifest M1 containing D1, D2, and D3 (all ADDED).

Snapshot 2: Links to Root Manifest B which preserves M1 (now EXISTING) and adds a new entry M2 (ADDED), which connects to Data Manifest M2 containing D4, D5, and D6 (all ADDED).

This structure demonstrates how Iceberg efficiently manages data organization by:
- Preserving existing manifest references across snapshots (M1 stays as EXISTING in Root Manifest B)
- Adding new manifests as data grows (M2 as ADDED in Root Manifest B)
- Maintaining separate data manifests for different sets of data files

The diagram uses the same styling with left-aligned table text and consistent color coding for different component types.

User

In the same style,

Snapshot 1 is linked to Root Manifest A which has a single ADDED Data Manifest Entry M1 which in turn links to Data Manifest M1 which has three rows (D1, D2, D3) each representing a Data File.

Snapshot 2 removes a file D2 by preserving the entry for M1 and then adding a new entry for a Delete Vector DV1 which applies to M1. The contents of DV1 mark the row at ordinal 1 as being removed.

Cursor

I've created a diagram showing the delete vector scenario. The diagram illustrates:

Snapshot 1: Links to Root Manifest A with M1 (ADDED), connecting to Data Manifest M1 containing D1, D2, D3 (all ADDED).

Snapshot 2: Links to Root Manifest B which:
- Preserves M1 (EXISTING) - efficiently reusing the existing data manifest
- Adds DV1 (ADDED) - the delete vector for handling the deletion

The **Delete Vector DV1** shows:
- "Applies to M1" indicating which manifest it targets
- "Ordinal 1 | REMOVED" marking the row at position 1 (D2) as logically deleted

The dotted line from DV1 to Data Manifest M1 illustrates that the delete vector applies to that specific manifest.

This demonstrates Iceberg's efficient deletion approach using delete vectors rather than rewriting data files - the original data files (D1, D2, D3) remain unchanged, but D2 is logically removed through the delete vector, providing both performance benefits and the ability to time-travel to see data before the deletion.

User

Is there a way to lay it out so that Root manifest nodes are at the same height and DV1 and M1 are at the same height

---

Cursor

I've reorganized the diagram using subgraphs to control the layout. Now:

1. **Snapshots** (S1, S2) are grouped at the top level
2. **Root Manifests** (RMA, RMB) are grouped at the same height on the second level

3. **Data Manifest M1 and Delete Vector DV1** are grouped at the same height on the third level
4. **Data Files** (D1, D2, D3) are grouped at the bottom level

The subgraphs help ensure that nodes at each level are positioned at the same height, creating a more organized and visually balanced layout. The connections still flow logically from snapshots to root manifests to their respective entries, while the dotted line shows the "applies to" relationship between the delete vector and the data manifest.

User

Data Manifest M1 and Delete Vector DV1 should be at the same height

---

Cursor

I've simplified the diagram structure to ensure Data Manifest M1 and Delete Vector DV1 are at the same height. By removing the subgraphs and having both nodes connected at the same level from Root Manifest B, Mermaid should naturally position them at the same horizontal level. This creates a cleaner layout where:

- Snapshots are at the top level
- Root Manifests are at the second level
- Data Manifest M1 and Delete Vector DV1 are at the same third level
- Data Files are at the bottom level

The diagram maintains all the relationships while ensuring the alignment you requested.

User

Can you make the subgroup boxes transparent and without a border

---

Cursor

Perfect! I've created the diagram with transparent subgroup boxes. The subgroups are still there to control the layout and ensure that:

- Snapshots are at the top level
- Root Manifests are at the same height on the second level
- Data Manifest M1 and Delete Vector DV1 are at the same height on the third level
- Data Files are at the bottom level

But now the subgroup containers are invisible - they provide the structural layout without the visual clutter of borders and background colors. This gives you the clean positioning you wanted while maintaining the hierarchical organization of the diagram elements.

User

Remove the text "Applies to M1" from DV1 and the text "Ordinal" from Ordinal 1

Cursor

Perfect! I've cleaned up the Delete Vector DV1 table by:

1. **Removing** the "Applies to M1" row entirely
2. **Simplifying** "Ordinal 1" to just "1"

Now the Delete Vector DV1 table is much cleaner, showing only:
- **Header**: Delete Vector DV1
- **One row**: 1 | REMOVED

The relationship to Data Manifest M1 is still clear through the dotted "applies to" connection line, making the table content more concise while maintaining the visual relationship in the diagram.

User

remove the text "REMOVED" from DV1

Cursor

Perfect! I've removed the "REMOVED" text from the Delete Vector DV1 table. Now the Delete Vector DV1 shows:

- **Header**: Delete Vector DV1
- **One row**: 1 | (empty)

This creates a very clean and minimal representation where the delete vector simply lists the ordinal position "1" that is affected, with the understanding that being present in a delete vector implies removal. The dotted line to Data Manifest M1 still clearly shows the relationship.

User

please remove the borders on the subgroups

Cursor

Perfect! I've updated the diagram to remove the subgroup borders by adding `clusterBkg: "transparent"` and `clusterBorder: "transparent"` to the theme variables. Now the subgroups are completely invisible - they still provide the structural layout to keep nodes at the same height levels, but without any visual borders or backgrounds. This gives you a clean diagram with proper positioning but no distracting subgroup boundaries.

User

Using this stame style create two graphs depicting the old Iceberg metadata with a manifest list and the new layout using a Root Manifest

---

Cursor

I'll create two diagrams showing the evolution from the old Iceberg metadata structure with manifest lists to the new structure using root manifests.

I've created two diagrams showing the evolution of Iceberg's metadata structure:

First diagram - Old Structure (Manifest List):
- Snapshots pointed to a **Manifest List**
- The Manifest List could only contain references to Data Manifests
- Simple, linear structure: Snapshot → Manifest List → Data Manifests → Data Files
- Limited flexibility in what could be referenced

Second diagram - New Structure (Root Manifest):
- Snapshots point to a **Root Manifest**
- The Root Manifest can contain multiple types of entries:
  - **Data Manifest references** (M1, M2) - like before
  - **Direct Data File references** (D5) - new capability
  - **Delete Vector references** (DV1) - for efficient deletions
- More flexible structure enabling advanced features

Key improvements in the new structure:
1. **Mixed entry types** - Can reference manifests, individual files, and delete vectors all from the same root manifest
2. **Delete vectors** - Enable efficient deletions without rewriting data files
3. **Direct file references** - Allow for more granular control and optimization
4. **Better performance** - More efficient metadata operations and reduced I/O

This evolution maintains backward compatibility while enabling more sophisticated metadata operations that improve both performance and functionality.

User

Create me another diagram in this style for the following situation

Snapshot 1 represents a table which has accumulated several delete vectors modifying existing metadata files. Snapshot 1 points to Root Manifest A which contains several existing data manifests M1 and M2 as well as existing delete vectors DV1 and DV2. D1 and D2 contain data files D1,D2,D3 and D4, D5, D6 respectively. DV1 applies to M1 and removes the row at ordinal 0. DV2 applies to M2 and removes the row at ordinal 1. In Snapshot 2 a new file is added to the Root manifest while simultaneously compacting the existing manifests. Snapshot 2 points to a new Root Manifest B which contains M3, and added data file D7. M3 contains entries for existing data files D2,D3,D5 and D6.

Cursor

I've created a diagram showing the compaction scenario you described. This illustrates a common Iceberg operation where accumulated delete vectors are consolidated during compaction:

Snapshot 1 - Before Compaction:
- **Root Manifest A** contains fragmented metadata:

  - M1 and M2 (EXISTING data manifests)
  - DV1 and DV2 (EXISTING delete vectors)
- **Data Manifest M1** contains D1, D2, D3
- **Data Manifest M2** contains D4, D5, D6
- **DV1** applies to M1 and removes ordinal 0 (D1)
- **DV2** applies to M2 and removes ordinal 1 (D5)

Snapshot 2 - After Compaction:
- **Root Manifest B** contains cleaned metadata:
  - M3 (ADDED - new consolidated manifest)
  - D7 (ADDED - new data file)
- **Data Manifest M3** contains only the remaining valid data files: D2, D3, D5, D6

Key benefits of this compaction:
1. **Metadata consolidation** - Multiple manifests + delete vectors → Single manifest
2. **Performance improvement** - Faster query planning with fewer metadata files
3. **Storage optimization** - Removes references to deleted data
4. **Simplified structure** - Cleaner metadata hierarchy

This process demonstrates how Iceberg efficiently manages metadata growth over time, applying delete vectors during compaction to create a cleaner, more performant metadata structure while simultaneously allowing new data (D7) to be added.

User

Again for,

Snapshot 1 represents a table which has accumulated several delete vectors modifying existing metadata files. Snapshot 1 points to Root Manifest A which contains several existing data manifests M1 and M2 as well as existing delete vectors DV1 and DV2. D1 and D2 contain data files D1,D2,D3 and D4, D5, D6 respectively. DV1 applies to M1 and removes the row at ordinal 0. DV2 applies to M2 and removes the row at ordinal 1. In Snapshot 2 a new file is added to the Root manifest while simultaneously compacting the existing manifests. Snapshot 2 points to a new Root Manifest B which contains M3, and added data file D7. M3 contains entries for existing data files D2,D3,D4 and D6.

Cursor

I've created the corrected diagram showing the compaction scenario. The key difference from the previous version is in the final consolidated manifest:

Snapshot 1 - Before Compaction:
- **Root Manifest A** contains fragmented metadata:
  - M1 and M2 (EXISTING data manifests)
  - DV1 and DV2 (EXISTING delete vectors)
- **Data Manifest M1** contains D1, D2, D3
- **Data Manifest M2** contains D4, D5, D6
- **DV1** applies to M1 and removes ordinal 0 (D1)
- **DV2** applies to M2 and removes ordinal 1 (D5)

Snapshot 2 - After Compaction:
- **Root Manifest B** contains cleaned metadata:
  - M3 (ADDED - new consolidated manifest)
  - D7 (ADDED - new data file)
- **Data Manifest M3** contains the remaining valid data files: D2, D3, D4, D6

Effective result of the delete vectors:
- From M1: D1 is removed (ordinal 0), D2 and D3 remain
- From M2: D5 is removed (ordinal 1), D4 and D6 remain
- M3 consolidates all remaining files: D2, D3, D4, D6

This demonstrates how compaction efficiently applies accumulated delete vectors to create a clean, consolidated metadata structure while simultaneously adding new data.

User

If possible can we arrange this so all the children of the Root Manifests are on the same vertical level? Then we can order it
M1 DV1 M2 DV2 M3 then D7 in that layer

Cursor

Perfect! I've rearranged the diagram so that all the children of the Root Manifests are positioned at the same vertical level in the third subgroup. The ordering is now:

Third level (same height): M1, DV1, M2, DV2, M3, D7

This creates a much cleaner visual organization where:

1. **Snapshots** are at the top level

2. **Root Manifests** (A and B) are at the second level
3. **All manifest children** are aligned horizontally at the third level in your requested order
4. **Individual data files** are at the bottom level

The logical relationships are maintained:
- Root Manifest A connects to M1, M2, DV1, DV2
- Root Manifest B connects to M3, D7
- Delete vectors still show their "applies to" relationships with dotted lines
- Data manifests connect down to their respective data files

This layout makes it much easier to see the structural relationships and compare the before/after states of the compaction process.

User

Visually I want all of the nodes that are children of the Root manifests to be on the same vertical place, I also want to avoid edges crossing each other

---

Cursor

Perfect! I've rearranged the diagram to minimize edge crossings while keeping all children of the Root Manifests at the same vertical level. The key improvements:

Positioning to avoid crossings:
- **Left side**: Children of Root Manifest A (M1, DV1, M2, DV2) are positioned under their parent
- **Right side**: Children of Root Manifest B (M3, D7) are positioned under their parent
- **Strategic ordering**: DV1 is placed next to M1, and DV2 next to M2 to minimize the length of "applies to" dotted lines

Clean edge flow:
- Root Manifest A's edges flow down to its children on the left
- Root Manifest B's edges flow down to its children on the right
- No crossing between the solid connection lines
- Dotted "applies to" lines are short and don't interfere with other connections

This layout creates a much cleaner visual flow while maintaining all the logical relationships and keeping the children of both root manifests at exactly the same vertical level as requested.

User

Put all the children of root manifests in a transparent subgraph with no border

---

Cursor

Perfect! I've created a transparent subgraph that contains all the children of the root manifests. Now all the elements that are direct children of Root Manifest A and Root Manifest B (M1, DV1, M2, DV2, M3, D7) are grouped together in the same invisible container:

Grouped in transparent subgraph:
- Data Manifest M1
- Delete Vector DV1
- Data Manifest M2
- Delete Vector DV2
- Data Manifest M3
- Data File D7

This ensures they all appear at exactly the same vertical level while maintaining the clean visual appearance with no visible borders around the grouping. The transparent subgraph provides the structural constraint to keep them aligned without adding any visual clutter to the diagram.

User

Remove D1,D2,D3,D4,D5,D6

Cursor

Perfect! I've removed all the individual data file nodes (D1, D2, D3, D4, D5, D6) from the diagram. This creates a much cleaner view that focuses on the metadata structure:

What remains:
- Snapshots and Root Manifests at the top levels
- All children of root manifests (M1, DV1, M2, DV2, M3, D7) in the transparent subgraph at the same level
- Data manifests showing their file contents in table format
- Delete vectors with their ordinal positions

- D7 as a direct data file reference

What was removed:
- Individual data file nodes for D1-D6
- All the connections from data manifests to individual files
- The bottom subgraph layer

The diagram now provides a cleaner view of the metadata relationships while still showing the file contents within the manifest tables themselves. This focuses attention on the structure and relationships rather than the individual file nodes.

User

All the entries of M3 should be marked as Existing

Cursor

Perfect! I've updated Data Manifest M3 to mark all its entries as "EXISTING" instead of "ADDED". This accurately reflects that M3 is a compacted manifest that consolidates existing data files:

Data Manifest M3 now shows:
- D2 | EXISTING (from M1, survived deletion)
- D3 | EXISTING (from M1, survived deletion)
- D4 | EXISTING (from M2, survived deletion)
- D6 | EXISTING (from M2, survived deletion)

This makes logical sense because during compaction:
1. The delete vectors (DV1, DV2) are applied to the original manifests
2. M3 consolidates the remaining valid files that already existed in the system
3. Only D7 is truly new data being added

The diagram now correctly shows that M3 contains references to existing data files that have been reorganized and consolidated, rather than newly created files.

User

The entry for M3 should also be EXISTING

Cursor

Perfect! I've updated Root Manifest B to mark the M3 entry as "EXISTING" instead of "ADDED".

Root Manifest B now shows:
- M3 | EXISTING
- D7 | ADDED

This indicates that while M3 is a consolidated manifest created during compaction, it's being treated as an existing manifest reference in the context of the new root manifest, with only D7 representing truly new data being added to the table.

User

In the same style add 2 graphs

Snapshot V3 pointing to Manifest List which points to Manifest which points to Data Files

Snapshot V4 pointing to a Root Manifest which points to a Manifest, Data File and a Delete Vector. The Manifest has. a child data file.

Cursor

I've created two diagrams in the same style showing the evolution from the old to new Iceberg metadata structure:

First diagram - Snapshot V3 (Old Structure):
- Simple linear hierarchy: Snapshot V3 → Manifest List → Manifest M1 → Data Files
- Manifest List contains only one entry (M1)
- Manifest M1 contains D1, D2, D3
- Clean but limited structure

Second diagram - Snapshot V4 (New Structure):
- More flexible hierarchy: Snapshot V4 → Root Manifest with multiple child types
- Root Manifest contains:
  - M1 (manifest reference)
  - D5 (direct data file reference)

  - DV1 (delete vector reference)
- Manifest M1 has a child data file D4
- Delete Vector DV1 applies to M1
- All children of the Root Manifest are positioned at the same level in a transparent subgroup

Both diagrams maintain the same styling with transparent subgroups, left-aligned table text, and consistent color coding. This clearly demonstrates how the new Root Manifest structure enables more sophisticated metadata operations while maintaining the core functionality of the original Manifest List approach.

User

place both diagrams in the same graph and remove the Data Files

---

Cursor

Perfect! I've combined both diagrams into a single graph showing the comparison between the old and new Iceberg metadata structures, and removed all the individual data file nodes (D1, D2, D3, D4).

The combined diagram shows:

Left side - Snapshot V3 (Old Structure):
- Simple hierarchy: Snapshot V3 → Manifest List → Manifest M1
- Manifest List can only contain manifest references

Right side - Snapshot V4 (New Structure):
- Flexible hierarchy: Snapshot V4 → Root Manifest with multiple child types
- Root Manifest contains M1, D5 (direct data file), and DV1 (delete vector)
- Both snapshots reference the same Manifest M1

Key elements:
- Both structures share the same Manifest M1 (showing efficient reuse)
- Manifest M1 shows its contents in table format (D1, D2, D3)
- Delete Vector DV1 applies to M1 with dotted line
- All children of the manifests are positioned at the same level in a transparent subgroup
- Individual data file nodes are removed, focusing on metadata structure

This side-by-side comparison clearly demonstrates the evolution from the limited Manifest List to the more powerful Root Manifest approach while maintaining clean visual organization.

User

Never mind keep them separate

Cursor

Perfect! I've created two separate diagrams in the same style, without the individual data file nodes:

First diagram - Snapshot V3 (Old Structure):
- Simple hierarchy: Snapshot V3 → Manifest List → Manifest M1
- Clean, linear structure with limited capabilities
- Manifest M1 shows its data file contents in table format

Second diagram - Snapshot V4 (New Structure):
- Flexible hierarchy: Snapshot V4 → Root Manifest with multiple child types
- Root Manifest contains:
  - M1 (manifest reference)
  - D5 (direct data file reference)
  - DV1 (delete vector reference)
- All children positioned at the same level in a transparent subgroup
- Delete Vector DV1 applies to Manifest M1
- Demonstrates the enhanced capabilities of the Root Manifest approach

Both diagrams maintain consistent styling and clearly show the evolution from the restrictive Manifest List to the more powerful Root Manifest structure.

User

A new Diagram in the same style, do not make explicit nodes for data files

In Snapshot 1 we have a Root Manifest A with existing Data Manifest M1 and existing Delete Manifest M2. M1 contains entries for D1, D2, and D3. M2 contains an entry for a Delete Vector removing rows in D2.

In Snapshot 2 we delete more rows from D2 by adding a new Delete Vector in a new Delete Manifest M3.

In Snapshot 3 we delete even more rows from D2 by adding two Delete Vectors to Delete Manifest M4. One representing the deletes present in Snapshot 1 and 2 and one representing the deletes added in snapshot 3. Delete Manifests M2 and M3 are removed.

Cursor

I've created a diagram showing the evolution of delete manifests across three snapshots. This demonstrates how Iceberg manages and consolidates delete metadata over time:

Snapshot 1:
- Root Manifest A contains Data Manifest M1 and Delete Manifest M2
- M1 contains data files D1, D2, D3
- M2 contains DV1 (delete vector targeting D2)

Snapshot 2:
- Root Manifest B preserves M1 and M2, adds new Delete Manifest M3
- M3 contains DV2 (additional delete vector for D2)
- Shows accumulation of delete metadata

Snapshot 3:
- Root Manifest C keeps M1 but consolidates delete metadata
- New Delete Manifest M4 contains:
  - DV3 (representing consolidated deletes from Snapshots 1&2)
  - DV4 (new deletes added in Snapshot 3)
- M2 and M3 are removed, showing cleanup of fragmented delete metadata

Key features:
- All delete manifests apply to the same Data Manifest M1 (shown with dotted lines)
- Demonstrates metadata consolidation - multiple delete manifests merged into one
- Shows how Iceberg optimizes delete metadata over time
- All manifest children positioned at the same level in transparent subgroups
- No explicit data file nodes, keeping focus on metadata structure

This pattern shows how Iceberg can efficiently manage growing delete metadata by periodically consolidating multiple delete manifests into fewer, more comprehensive ones.

User

DV3 is Existing

Cursor

Perfect! I've updated Delete Manifest M4 to show DV3 as "EXISTING" instead of "ADDED". This accurately reflects that:

Delete Manifest M4 now shows:
- DV3 | EXISTING (consolidated delete vectors from Snapshots 1 & 2)
- DV4 | ADDED (new delete vectors added in Snapshot 3)

This makes logical sense because DV3 represents the consolidation of existing delete information from the previous snapshots, while only DV4 represents truly new delete operations introduced in Snapshot 3. The diagram now correctly distinguishes between reorganized existing data and newly added data in the consolidation process.

```
%%{init: {"flowchart": {"htmlLabels": true}, "theme": "base", "themeVariables": {"primaryColor": "#ffffff", "primaryTextColor": "#000000", "primaryBorderColor": "#000000", "lineColor": "#000000", "sectionBkColor": "transparent", "altSectionBkColor": "transparent", "gridColor": "#000000", "secondaryColor": "#ffffff", "tertiaryColor": "#ffffff", "clusterBkg": "transparent", "clusterBorder": "transparent"}}}%%
graph TD
    subgraph " "
        S1["Snapshot 1"]
        S2["Snapshot 2"]
        S3["Snapshot 3"]
        S4["Snapshot 4"]
    end

    subgraph "  "
        RMA["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest A</b></td></tr><tr><td style='text-align:left'>M1</td><td style='text-align:left'>ADDED</td></tr></table>"]
        RMB["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest B</b></td></tr><tr><td style='text-align:left'>M1</td><td style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>DV1</td><td style='text-align:left'>ADDED</td></tr></table>"]
        RMC["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest C</b></td></tr><tr><td style='text-align:left'>M1</td><td style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>DV1</td><td style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>DV2</td><td style='text-align:left'>ADDED</td></tr></table>"]
```

```
        RMD["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
D</b></td></tr><tr><td style='text-align:left'>M1</td><td
style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>DV3</td><td
style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>D4</td><td
style='text-align:left'>ADDED</td></tr></table>"]
    end

    subgraph "   "
        DM1["<table><tr><td colspan='2' style='text-align:left'><b>Data Manifest
M1</b></td></tr><tr><td style='text-align:left'>D1</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>D2</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>D3</td><td
style='text-align:left'>ADDED</td></tr></table>"]
        DV1["<table><tr><td colspan='2' style='text-align:left'><b>Delete Vector
DV1</b></td></tr><tr><td style='text-align:left'>1</td><td
style='text-align:left'></td></tr></table>"]
        DV2["<table><tr><td colspan='2' style='text-align:left'><b>Delete Vector
DV2</b></td></tr><tr><td style='text-align:left'>2</td><td
style='text-align:left'></td></tr></table>"]
        DV3["<table><tr><td colspan='2' style='text-align:left'><b>Delete Vector
DV3</b></td></tr><tr><td style='text-align:left'>1</td><td style='text-align:left'></td></tr><tr><td
style='text-align:left'>2</td><td style='text-align:left'></td></tr></table>"]
    end

    S1 --> RMA
    S2 --> RMB
    S3 --> RMC
    S4 --> RMD
    RMA --> DM1
    RMB --> DM1
    RMB --> DV1
    RMC --> DV1
    RMC --> DM1
    RMC --> DV2
    RMD --> DM1
    RMD --> DV3
    DV1 -.->|applies to| DM1
    DV2 -.->|applies to| DM1
    DV3 -.->|applies to| DM1

    classDef snapshot fill:#e1f5fe,stroke:#01579b,stroke-width:2px
    classDef rootManifest fill:#f3e5f5,stroke:#4a148c,stroke-width:2px
    classDef dataManifest fill:#e8f5e8,stroke:#1b5e20,stroke-width:2px
    classDef dataFile fill:#fff8e1,stroke:#f57f17,stroke-width:2px
```

```
    classDef deleteVector fill:#ffebee,stroke:#c62828,stroke-width:2px

    class S1,S2,S3,S4 snapshot
    class RMA,RMB,RMC,RMD rootManifest
    class DM1 dataManifest
    class DV1,DV2,DV3 deleteVector
%%{init: {"flowchart": {"htmlLabels": true}, "theme": "base", "themeVariables": {"primaryColor":
"#ffffff", "primaryTextColor": "#000000", "primaryBorderColor": "#000000", "lineColor":
"#000000", "sectionBkColor": "transparent", "altSectionBkColor": "transparent", "gridColor":
"#000000", "secondaryColor": "#ffffff", "tertiaryColor": "#ffffff", "clusterBkg": "transparent",
"clusterBorder": "transparent"}}}%%
graph TD
    subgraph " "
        S1["Snapshot 1"]
        S2["Snapshot 2"]
    end

    subgraph "  "
        RMA["<table><tr><td colspan='2' style='text-align:left; vertical-align: middle;'><b>Root
Manifest A</b></td></tr><tr><td style='text-align:left; vertical-align: middle;'>M1</td><td
style='text-align:left; vertical-align: middle;'>EXISTING</td></tr><tr><td style='text-align:left;
vertical-align: middle;'>M2</td><td style='text-align:left; vertical-align:
middle;'>EXISTING</td></tr><tr><td style='text-align:left; vertical-align: middle;'>DV1</td><td
style='text-align:left; vertical-align: middle;'>EXISTING</td></tr><tr><td style='text-align:left;
vertical-align: middle;'>DV2</td><td style='text-align:left; vertical-align:
middle;'>EXISTING</td></tr></table>"]
        RMB["<table><tr><td colspan='2' style='text-align:left; vertical-align: middle;'><b>Root
Manifest B</b></td></tr><tr><td style='text-align:left; vertical-align: middle;'>M3</td><td
style='text-align:left; vertical-align: middle;'>EXISTING</td></tr><tr><td style='text-align:left;
vertical-align: middle;'>D7</td><td style='text-align:left; vertical-align:
middle;'>ADDED</td></tr></table>"]
    end

    subgraph manifest_children ["   "]
        DM1["<table><tr><td colspan='2' style='text-align:left; vertical-align: middle;'><b>Data
Manifest M1</b></td></tr><tr><td style='text-align:left; vertical-align: middle;'>D1</td><td
style='text-align:left; vertical-align: middle;'>ADDED</td></tr><tr><td style='text-align:left;
vertical-align: middle;'>D2</td><td style='text-align:left; vertical-align:
middle;'>ADDED</td></tr><tr><td style='text-align:left; vertical-align: middle;'>D3</td><td
style='text-align:left; vertical-align: middle;'>ADDED</td></tr></table>"]
        DV1["<table><tr><td colspan='2' style='text-align:left; vertical-align: middle;'><b>Manifest
DV1</b></td></tr><tr><td style='text-align:left; vertical-align: middle;'>0</td><td
style='text-align:left; vertical-align: middle;'></td></tr></table>"]
```

```
        DM2["<table><tr><td colspan='2' style='text-align:left; vertical-align: middle;'><b>Data Manifest M2</b></td></tr><tr><td style='text-align:left; vertical-align: middle;'>D4</td><td style='text-align:left; vertical-align: middle;'>ADDED</td></tr><tr><td style='text-align:left; vertical-align: middle;'>D5</td><td style='text-align:left; vertical-align: middle;'>ADDED</td></tr><tr><td style='text-align:left; vertical-align: middle;'>D6</td><td style='text-align:left; vertical-align: middle;'>ADDED</td></tr></table>"]
        DV2["<table><tr><td colspan='2' style='text-align:left; vertical-align: middle;'><b>Manifest DV2</b></td></tr><tr><td style='text-align:left; vertical-align: middle;'>1</td><td style='text-align:left; vertical-align: middle;'></td></tr></table>"]
        DM3["<table><tr><td colspan='2' style='text-align:left; vertical-align: middle;'><b>Data Manifest M3</b></td></tr><tr><td style='text-align:left; vertical-align: middle;'>D2</td><td style='text-align:left; vertical-align: middle;'>EXISTING</td></tr><tr><td style='text-align:left; vertical-align: middle;'>D3</td><td style='text-align:left; vertical-align: middle;'>EXISTING</td></tr><tr><td style='text-align:left; vertical-align: middle;'>D4</td><td style='text-align:left; vertical-align: middle;'>EXISTING</td></tr><tr><td style='text-align:left; vertical-align: middle;'>D6</td><td style='text-align:left; vertical-align: middle;'>EXISTING</td></tr></table>"]
        D7["Data File D7"]
    end

    S1 --> RMA
    S2 --> RMB
    RMA --> DM1
    RMA --> DV1
    RMA --> DM2
    RMA --> DV2
    RMB --> DM3
    RMB --> D7
    DV1 -.->|applies to M1| DM1
    DV2 -.->|applies to M2| DM2

    classDef snapshot fill:#e1f5fe,stroke:#01579b,stroke-width:2px
    classDef rootManifest fill:#f3e5f5,stroke:#4a148c,stroke-width:2px
    classDef dataManifest fill:#e8f5e8,stroke:#1b5e20,stroke-width:2px
    classDef dataFile fill:#fff8e1,stroke:#f57f17,stroke-width:2px
    classDef manifestDeleteVector fill:#fed7aa,stroke:#ea580c,stroke-width:2px

    class S1,S2 snapshot
    class RMA,RMB rootManifest
    class DM1,DM2,DM3 dataManifest
    class D7 dataFile
    class DV1,DV2 manifestDeleteVector%%{init: {"flowchart": {"htmlLabels": true}, "theme": "base", "themeVariables": {"primaryColor": "#ffffff", "primaryTextColor": "#000000", "primaryBorderColor": "#000000", "lineColor": "#000000", "sectionBkColor": "transparent",
```

```
"altSectionBkColor": "transparent", "gridColor": "#000000", "secondaryColor": "#ffffff",
"tertiaryColor": "#ffffff", "clusterBkg": "transparent", "clusterBorder": "transparent"}}}%%
graph TD
    subgraph " "
        S1["Snapshot 1"]
        S2["Snapshot 2"]
        S3["Snapshot 3"]
    end

    subgraph "  "
        RMA["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
A</b></td></tr><tr><td style='text-align:left'>M1</td><td
style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>M2</td><td
style='text-align:left'>EXISTING</td></tr></table>"]
        RMB["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
B</b></td></tr><tr><td style='text-align:left'>M1</td><td
style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>M2</td><td
style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>M3</td><td
style='text-align:left'>ADDED</td></tr></table>"]
        RMC["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
C</b></td></tr><tr><td style='text-align:left'>M1</td><td
style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>M4</td><td
style='text-align:left'>ADDED</td></tr></table>"]
    end

    subgraph manifest_children ["   "]
        DM1["<table><tr><td colspan='2' style='text-align:left'><b>Data Manifest
M1</b></td></tr><tr><td style='text-align:left'>D1</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>D2</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>D3</td><td
style='text-align:left'>ADDED</td></tr></table>"]
        DLM2["<table><tr><td colspan='2' style='text-align:left'><b>Delete Manifest
M2</b></td></tr><tr><td style='text-align:left'>DV1</td><td
style='text-align:left'>ADDED</td></tr></table>"]
        DLM3["<table><tr><td colspan='2' style='text-align:left'><b>Delete Manifest
M3</b></td></tr><tr><td style='text-align:left'>DV2</td><td
style='text-align:left'>ADDED</td></tr></table>"]
        DLM4["<table><tr><td colspan='2' style='text-align:left'><b>Delete Manifest
M4</b></td></tr><tr><td style='text-align:left'>DV3</td><td
style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>DV4</td><td
style='text-align:left'>ADDED</td></tr></table>"]
    end

    S1 --> RMA
```

```
    S2 --> RMB
    S3 --> RMC
    RMA --> DM1
    RMA --> DLM2
    RMB --> DM1
    RMB --> DLM2
    RMB --> DLM3
    RMC --> DM1
    RMC --> DLM4
    DLM2 -.->|applies to| DM1
    DLM3 -.->|applies to| DM1
    DLM4 -.->|applies to| DM1

    classDef snapshot fill:#e1f5fe,stroke:#01579b,stroke-width:2px
    classDef rootManifest fill:#f3e5f5,stroke:#4a148c,stroke-width:2px
    classDef dataManifest fill:#e8f5e8,stroke:#1b5e20,stroke-width:2px
    classDef deleteManifest fill:#ffebee,stroke:#c62828,stroke-width:2px

    class S1,S2,S3 snapshot
    class RMA,RMB,RMC rootManifest
    class DM1 dataManifest
    class DLM2,DLM3,DLM4 deleteManifest%%{init: {"flowchart": {"htmlLabels": true}, "theme":
"base", "themeVariables": {"primaryColor": "#ffffff", "primaryTextColor": "#000000",
"primaryBorderColor": "#000000", "lineColor": "#000000", "sectionBkColor": "transparent",
"altSectionBkColor": "transparent", "gridColor": "#000000", "secondaryColor": "#ffffff",
"tertiaryColor": "#ffffff", "clusterBkg": "transparent", "clusterBorder": "transparent"}}}%%
graph TD
    subgraph " "
        S4["Snapshot V4"]
    end

    subgraph "  "
        RM["<table><tr><td colspan='2' style='text-align:left; vertical-align: middle;'><b>Root
Manifest</b></td></tr><tr><td style='text-align:left; vertical-align: middle;'>M1</td><td
style='text-align:left; vertical-align: middle;'>ADDED</td></tr><tr><td style='text-align:left;
vertical-align: middle;'>D5</td><td style='text-align:left; vertical-align:
middle;'>ADDED</td></tr><tr><td style='text-align:left; vertical-align: middle;'>DV1</td><td
style='text-align:left; vertical-align: middle;'>ADDED</td></tr></table>"]
    end

    subgraph manifest_children ["   "]
        DM1["<table><tr><td colspan='2' style='text-align:left; vertical-align: middle;'><b>Manifest
M1</b></td></tr><tr><td style='text-align:left; vertical-align: middle;'>D1</td><td
style='text-align:left; vertical-align: middle;'>ADDED</td></tr><tr><td style='text-align:left;
```

```
vertical-align: middle;'>D2</td><td style='text-align:left; vertical-align:
middle;'>ADDED</td></tr><tr><td style='text-align:left; vertical-align: middle;'>D3</td><td
style='text-align:left; vertical-align: middle;'>ADDED</td></tr></table>"]
        D5["Data File D5"]
        DV1["<table><tr><td colspan='2' style='text-align:left; vertical-align: middle;'><b>Manifest
DV1</b></td></tr><tr><td style='text-align:left; vertical-align: middle;'>1</td><td
style='text-align:left; vertical-align: middle;'></td></tr></table>"]
    end

    S4 --> RM
    RM --> DM1
    RM --> D5
    RM --> DV1
    DV1 -.->|applies to M1| DM1

    classDef snapshot fill:#e1f5fe,stroke:#01579b,stroke-width:2px
    classDef rootManifest fill:#f3e5f5,stroke:#4a148c,stroke-width:2px
    classDef dataManifest fill:#e8f5e8,stroke:#1b5e20,stroke-width:2px
    classDef dataFile fill:#fff8e1,stroke:#f57f17,stroke-width:2px
    classDef manifestDeleteVector fill:#fed7aa,stroke:#ea580c,stroke-width:2px

    class S4 snapshot
    class RM rootManifest
    class DM1 dataManifest
    class D5 dataFile
    class DV1 manifestDeleteVector%%{init: {"flowchart": {"htmlLabels": true}, "theme": "base",
"themeVariables": {"primaryColor": "#ffffff", "primaryTextColor": "#000000", "primaryBorderColor":
"#000000", "lineColor": "#000000", "sectionBkColor": "transparent", "altSectionBkColor":
"transparent", "gridColor": "#000000", "secondaryColor": "#ffffff", "tertiaryColor": "#ffffff",
"clusterBkg": "transparent", "clusterBorder": "transparent"}}}%%
graph TD
    subgraph " "
        S3["Snapshot V3"]
    end

    subgraph "  "
        ML["<table><tr><td colspan='2' style='text-align:left'><b>Manifest List</b></td></tr><tr><td
style='text-align:left'>M1</td><td style='text-align:left'>ADDED</td></tr></table>"]
    end

    subgraph "   "
        DM1["<table><tr><td colspan='2' style='text-align:left'><b>Manifest
M1</b></td></tr><tr><td style='text-align:left'>D1</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>D2</td><td
```

```
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>D3</td><td
style='text-align:left'>ADDED</td></tr></table>"]
  end

  S3 --> ML
  ML --> DM1

  classDef snapshot fill:#e1f5fe,stroke:#01579b,stroke-width:2px
  classDef manifestList fill:#fff3e0,stroke:#e65100,stroke-width:2px
  classDef dataManifest fill:#e8f5e8,stroke:#1b5e20,stroke-width:2px

  class S3 snapshot
  class ML manifestList
  class DM1 dataManifest%%{init: {"flowchart": {"htmlLabels": true}, "theme": "base",
"themeVariables": {"primaryColor": "#ffffff", "primaryTextColor": "#000000", "primaryBorderColor":
"#000000", "lineColor": "#000000", "sectionBkColor": "transparent", "altSectionBkColor":
"transparent", "gridColor": "#000000", "secondaryColor": "#ffffff", "tertiaryColor": "#ffffff",
"clusterBkg": "transparent", "clusterBorder": "transparent"}}}%%
graph TD
  subgraph " "
    S1["Snapshot 1"]
    S2["Snapshot 2"]
  end

  subgraph "  "
    RMA["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
A</b></td></tr><tr><td style='text-align:left'>M1</td><td
style='text-align:left'>ADDED</td></tr></table>"]
    RMB["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
B</b></td></tr><tr><td style='text-align:left'>M1</td><td
style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>DV1</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>D4</td><td
style='text-align:left'>ADDED</td></tr></table>"]
  end

  subgraph manifest_children ["   "]
    DM1["<table><tr><td colspan='2' style='text-align:left'><b>Data Manifest
M1</b></td></tr><tr><td style='text-align:left'>D1</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>D2</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>D3</td><td
style='text-align:left'>ADDED</td></tr></table>"]
    DV1["<table><tr><td colspan='2' style='text-align:left'><b>Manifest
DV1</b></td></tr><tr><td style='text-align:left'>1</td><td
style='text-align:left'></td></tr></table>"]
```

```
    end

    S1 --> RMA
    S2 --> RMB
    RMA --> DM1
    RMB --> DM1
    RMB --> DV1
    DV1 -.->|applies to M1| DM1

    classDef snapshot fill:#e1f5fe,stroke:#01579b,stroke-width:2px
    classDef rootManifest fill:#f3e5f5,stroke:#4a148c,stroke-width:2px
    classDef dataManifest fill:#e8f5e8,stroke:#1b5e20,stroke-width:2px
    classDef manifestDeleteVector fill:#fed7aa,stroke:#ea580c,stroke-width:2px

    class S1,S2 snapshot
    class RMA,RMB rootManifest
    class DM1 dataManifest
    class DV1 manifestDeleteVector%%{init: {"flowchart": {"htmlLabels": true}, "theme": "base",
"themeVariables": {"primaryColor": "#ffffff", "primaryTextColor": "#000000", "primaryBorderColor":
"#000000", "lineColor": "#000000", "sectionBkColor": "transparent", "altSectionBkColor":
"transparent", "gridColor": "#000000", "secondaryColor": "#ffffff", "tertiaryColor": "#ffffff",
"clusterBkg": "transparent", "clusterBorder": "transparent"}}}%%
graph TD
    subgraph " "
        S1["Snapshot 1"]
        S2["Snapshot 2"]
    end

    subgraph "  "
        RMA["<table><tr><td colspan='2' style='text-align:left; vertical-align: middle;'><b>Root
Manifest A</b></td></tr><tr><td style='text-align:left; vertical-align: middle;'>M1</td><td
style='text-align:left; vertical-align: middle;'>ADDED</td></tr></table>"]
        RMB["<table><tr><td colspan='2' style='text-align:left; vertical-align: middle;'><b>Root
Manifest B</b></td></tr><tr><td style='text-align:left; vertical-align: middle;'>M1</td><td
style='text-align:left; vertical-align: middle;'>EXISTING</td></tr><tr><td style='text-align:left;
vertical-align: middle;'>DV1</td><td style='text-align:left; vertical-align:
middle;'>ADDED</td></tr></table>"]
    end

    subgraph manifest_children ["   "]
        DM1["<table><tr><td colspan='2' style='text-align:left; vertical-align: middle;'><b>Data
Manifest M1</b></td></tr><tr><td style='text-align:left; vertical-align: middle;'>D1</td><td
style='text-align:left; vertical-align: middle;'>ADDED</td></tr><tr><td style='text-align:left;
vertical-align: middle;'>D2</td><td style='text-align:left; vertical-align:
```

```
middle;'>ADDED</td></tr><tr><td style='text-align:left; vertical-align: middle;'>D3</td><td
style='text-align:left; vertical-align: middle;'>ADDED</td></tr></table>"]
        DV1["<table><tr><td colspan='2' style='text-align:left; vertical-align: middle;'><b>Manifest
DV1</b></td></tr><tr><td style='text-align:left; vertical-align: middle;'>1</td><td
style='text-align:left; vertical-align: middle;'></td></tr></table>"]
    end

    S1 --> RMA
    S2 --> RMB
    RMA --> DM1
    RMB --> DM1
    RMB --> DV1
    DV1 -.->|applies to M1| DM1

    classDef snapshot fill:#e1f5fe,stroke:#01579b,stroke-width:2px
    classDef rootManifest fill:#f3e5f5,stroke:#4a148c,stroke-width:2px
    classDef dataManifest fill:#e8f5e8,stroke:#1b5e20,stroke-width:2px
    classDef manifestDeleteVector fill:#fed7aa,stroke:#ea580c,stroke-width:2px

    class S1,S2 snapshot
    class RMA,RMB rootManifest
    class DM1 dataManifest
    class DV1 manifestDeleteVector%%{init: {"flowchart": {"htmlLabels": true}, "theme": "base",
"themeVariables": {"primaryColor": "#ffffff", "primaryTextColor": "#000000", "primaryBorderColor":
"#000000", "lineColor": "#000000", "sectionBkColor": "transparent", "altSectionBkColor":
"transparent", "gridColor": "#000000", "secondaryColor": "#ffffff", "tertiaryColor": "#ffffff",
"clusterBkg": "transparent", "clusterBorder": "transparent"}}}%%
graph TD
    subgraph " "
        S1["Snapshot 1"]
        S2["Snapshot 2"]
    end

    subgraph "  "
        RMA["<table><tr><td colspan='2' style='text-align:left; vertical-align: middle;'><b>Root
Manifest A</b></td></tr><tr><td style='text-align:left; vertical-align: middle;'>D1</td><td
style='text-align:left; vertical-align: middle;'>EXISTING</td></tr><tr><td style='text-align:left;
vertical-align: middle;'>D2</td><td style='text-align:left; vertical-align:
middle;'>EXISTING</td></tr></table>"]
        RMB["<table><tr><td colspan='2' style='text-align:left; vertical-align: middle;'><b>Root
Manifest B</b></td></tr><tr><td style='text-align:left; vertical-align: middle;'>D1</td><td
style='text-align:left; vertical-align: middle;'>DELETED</td></tr><tr><td style='text-align:left;
vertical-align: middle;'>D2</td><td style='text-align:left; vertical-align:
middle;'>EXISTING</td></tr></table>"]
```

```
    end

    S1 --> RMA
    S2 --> RMB

    classDef snapshot fill:#e1f5fe,stroke:#01579b,stroke-width:2px
    classDef rootManifest fill:#f3e5f5,stroke:#4a148c,stroke-width:2px

    class S1,S2 snapshot
    class RMA,RMB rootManifest%%{init: {"flowchart": {"htmlLabels": true}, "theme": "base",
"themeVariables": {"primaryColor": "#ffffff", "primaryTextColor": "#000000", "primaryBorderColor":
"#000000", "lineColor": "#000000", "sectionBkColor": "transparent", "altSectionBkColor":
"transparent", "gridColor": "#000000", "secondaryColor": "#ffffff", "tertiaryColor": "#ffffff",
"clusterBkg": "transparent", "clusterBorder": "transparent"}}}%%
graph TD
    subgraph " "
        S1["Snapshot 1"]
        S2["Snapshot 2"]
    end

    subgraph "  "
        RMA["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
A</b></td></tr><tr><td style='text-align:left'>D1</td><td
style='text-align:left'>ADDED</td></tr></table>"]
        RMB["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
B</b></td></tr><tr><td style='text-align:left'>D1</td><td
style='text-align:left'>EXISTING</td></tr><tr><td style='text-align:left'>DV1</td><td
style='text-align:left'>ADDED</td></tr><tr><td style='text-align:left'>D2</td><td
style='text-align:left'>ADDED</td></tr></table>"]
        DV1["<table><tr><td colspan='2' style='text-align:left'><b>Delete Vector
DV1</b></td></tr><tr><td style='text-align:left'>5</td><td style='text-align:left'></td></tr><tr><td
style='text-align:left'>12</td><td style='text-align:left'></td></tr></table>"]
    end

    S1 --> RMA
    S2 --> RMB
    RMB --> DV1
    DV1 -.->|applies to data file D1| RMB

    classDef snapshot fill:#e1f5fe,stroke:#01579b,stroke-width:2px
    classDef rootManifest fill:#f3e5f5,stroke:#4a148c,stroke-width:2px
    classDef deleteVector fill:#ffebee,stroke:#c62828,stroke-width:2px

    class S1,S2 snapshot
```

```
    class RMA,RMB rootManifest
    class DV1 deleteVector
%%{init: {"flowchart": {"htmlLabels": true}, "theme": "base", "themeVariables": {"primaryColor":
"#ffffff", "primaryTextColor": "#000000", "primaryBorderColor": "#000000", "lineColor":
"#000000", "sectionBkColor": "transparent", "altSectionBkColor": "transparent", "gridColor":
"#000000", "secondaryColor": "#ffffff", "tertiaryColor": "#ffffff", "clusterBkg": "transparent",
"clusterBorder": "transparent"}}}%%
graph TD
    subgraph " "
        S1["Snapshot 1"]
        S2["Snapshot 2"]
        S3["Snapshot 3"]
    end

    subgraph "  "
        RMA["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
A</b></td></tr><tr><td style='text-align:left'>D1</td><td
style='text-align:left'>ADDED</td></tr></table>"]
        RMB["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
B</b></td></tr><tr><td style='text-align:left'>D1</td><td
style='text-align:left'>DELETED</td></tr><tr><td style='text-align:left'>D2</td><td
style='text-align:left'>ADDED</td></tr></table>"]
        RMC["<table><tr><td colspan='2' style='text-align:left'><b>Root Manifest
C</b></td></tr><tr><td style='text-align:left'>D2</td><td
style='text-align:left'>DELETED</td></tr><tr><td style='text-align:left'>D3</td><td
style='text-align:left'>ADDED</td></tr></table>"]
    end

    S1 --> RMA
    S2 --> RMB
    S3 --> RMC

    classDef snapshot fill:#e1f5fe,stroke:#01579b,stroke-width:2px
    classDef rootManifest fill:#f3e5f5,stroke:#4a148c,stroke-width:2px

    class S1,S2,S3 snapshot
    class RMA,RMB,RMC rootManifestgraph TD
    subgraph "Current"
        subgraph "Snapshot A"
            ML1["Manifest List"]
            DM1["Data Manifest 1"]
            DM2["Data Manifest 2"]
            DelM1["Delete Manifest 1"]
            DF1["Data Files"]
```

```
        DF2["Data Files"]
        DelF1["Delete Files"]

        ML1 --> DM1
        ML1 --> DM2
        ML1 --> DelM1
        DM1 --> DF1
        DM2 --> DF2
        DelM1 --> DelF1
    end
end

subgraph "Proposed"
    subgraph "Snapshot A"
        RM1["Root Manifest<br/>(Entry Point)"]

        subgraph "Direct Entries"
            DF3["Data File 1<br/>"]
            DV1["Delete Vector 1<br/>"]
        end

        subgraph "Manifest Entries"
            DM3["Data Manifest<br/>"]
            DelM2["Delete Manifest<br/>"]
            DF4["Data Files"]
            DelF2["Delete Files"]
        end

        RM1 --> DF3
        RM1 --> DV1
        RM1 --> DM3
        RM1 --> DelM2
        DM3 --> DF4
        DelM2 --> DelF2
    end
end


classDef current fill:#ffebee,stroke:#c62828,stroke-width:2px
classDef proposed fill:#e8f5e8,stroke:#1b5e20,stroke-width:2px
classDef difference fill:#fff3e0,stroke:#ef6c00,stroke-width:2px
classDef highlight fill:#e1f5fe,stroke:#01579b,stroke-width:3px

class ML1,DM1,DM2,DelM1,DF1,DF2,DelF1 current
```

class RM1,DM3,DelM2,DF3,DF4,DV1,DelF2 proposed
class RM1 highlight

(WIP Merge) Adaptive Metadata Tree

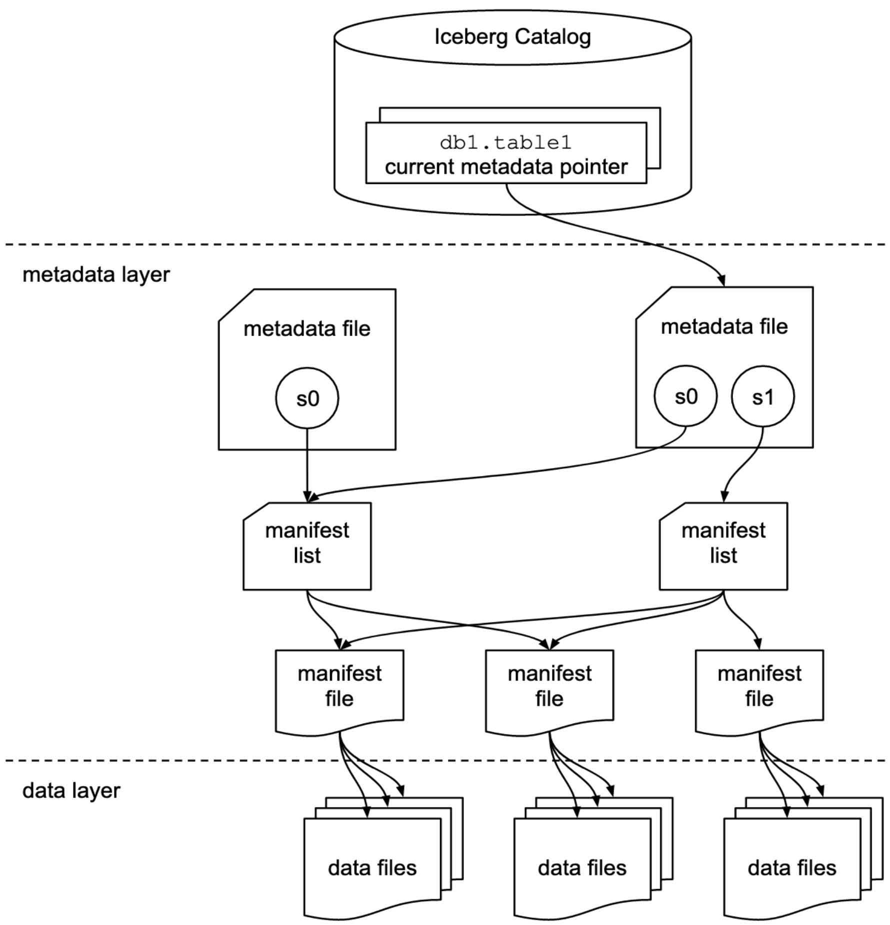
(Merging into Single File Commit Proposal)

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

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.



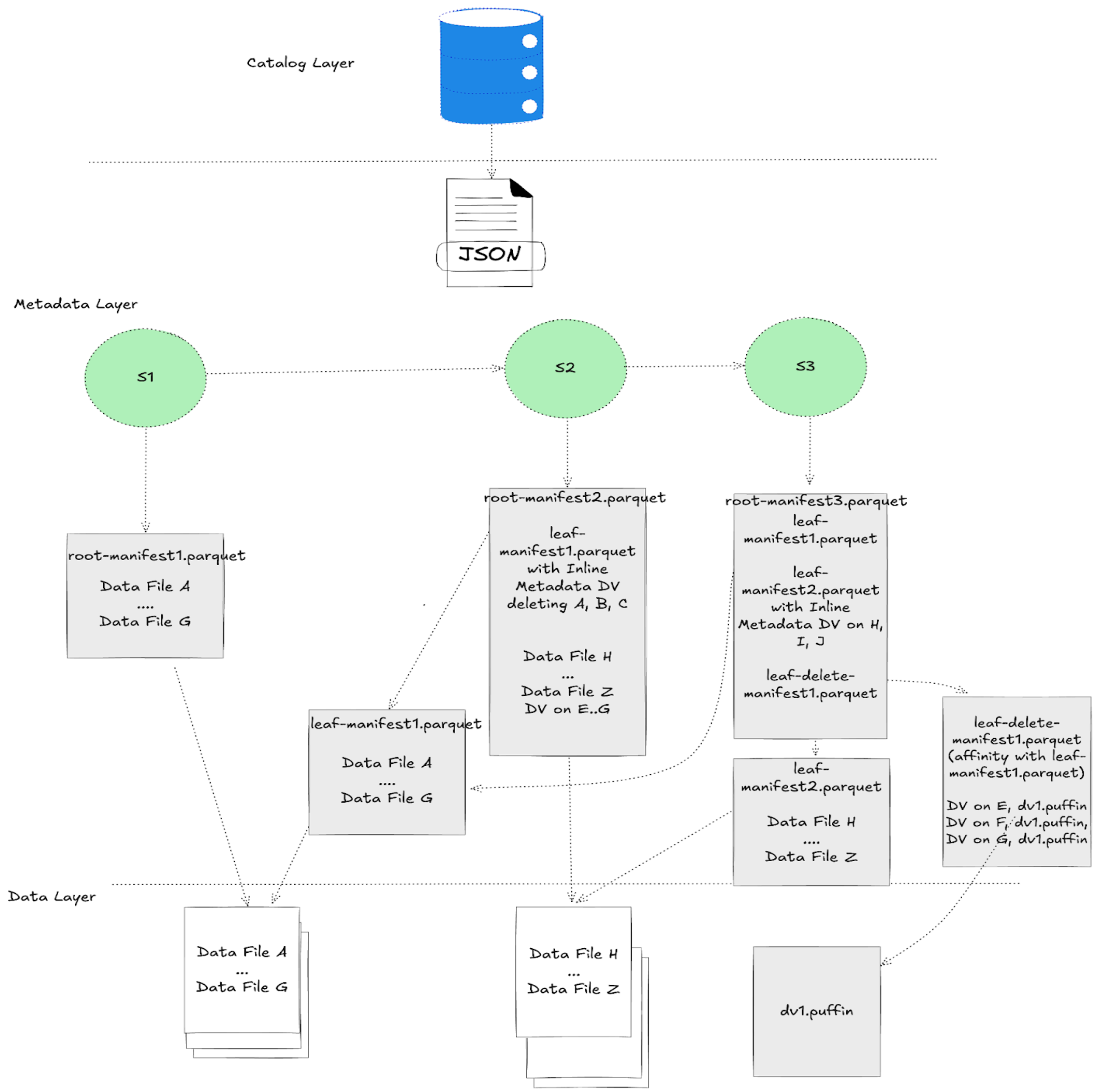
There are a few challenges with the current metadata tree structure:

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	• Leaf Data Manifests • Leaf Delete Manifests • Leaf Manifest DVs • Data Files • DVs on Data Files • Equality Deletes
Leaf data manifest	• Data Files
Leaf delete manifest	• 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 two phase planning, 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 association 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. This option was discarded because of higher write amplification to replace DVs, as the data files 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 deleted positions 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.

V4 content entry structure

Field ID	Name	Type	Required or Optional	Description
134	type	int	`required int` with meaning: 0: DATA 1: POSITION_DELETE 2: EQUALITY_DELETE 3: DATA_MANIFEST 4: DELETE_MANIFEST 5: MANIFEST_DV	Type of content stored by the data file: data, equality deletes, or position deletes (all v1 files are data files). Content types 3, 4 and 5, can only be defined in the root manifest.
100	location	string	required	Location of the file.
5	version_info	struct	optional	See version_info struct below. Groups information like status, and snapshot, sequence number
147	deletion_vector	struct	optional	See deletion_vector struct below. Must be defined if content type is 1 or 5. Must be null for all other types.
147	partition_spec_id	int	required	ID of partition spec used to write manifest or data/delete files.
140	sort_order_id	int	optional	ID representing sort order for this file. Can only be set if content_type is 0.
103	record_count	long	required	Number of records in this file, or the cardinality of a deletion vector
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	optional	Stats struct Column Stats Improv…

column_stats struct will have their own IDs)				
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), min_sequence_number(516) Must be set if content_type is 3 or 4, otherwise must be null
143	referenced_file	string	optional	Location of data file that a DV references if `content_type` is 1 or 5. Location of affiliated data manifest if `content_type` is 4 or null if delete manifest is unaffiliated.
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 must be null otherwise. Fields with ids listed in this column must be present in the delete 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

V4 content entry version_info struct

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 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.
3	sequence_number	long	optional	Carried over from current format: Data sequence number of the file. Inherited in when null and status is 1 (added). Must be equal to file_sequence_number if type is 3 or 4.
4	file_sequence_number	long	optional	File sequence number indicating when the file was added. Inherited when null and status is added. Must be equal to sequence_number if type is 3 or 4.

V4 content entry deletion_vector struct

144	offset	long	optional	The offset in the file where the content starts.
145	size_in_bytes	long	optional	The length of a referenced content stored in the file; required if `content_offset`

				is present.
146	inline_content	binary	optional	Serialized bitmap for inline DVs.

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.

How existing manifest list Fields map to Proposed V4 content entry fields

Manifest list field	V4 content entry field	Rationale or description
manifest_path	location	Shared location field for data files, DVs, equality deletes and manifests.
manifest_length	file_size_in_bytes	Shared file size field for data files, DVs, Equality deletes, and manifests
partition_spec_id	partition_spec_id	Still require partition spec for equality delete matching
content	type	Renamed to type since this is just an enumeration, and the original name of content was misleading that it contained the actual content
sequence_number	version_info.sequence_number	Moved to version_info structure
added_snapshot_id	version_info.snapshot_id	Shared with snapshot_id

min_sequence_number	manifest_stats.min_sequence_number	Moved to manifest_stats struct in content entry
added_files_count	manifest_stats.added_files_count	Moved to manifest_stats struct in content entry
existing_files_count	manifest_stats.existing_files_count	Moved to manifest_stats struct in content entry
deleted_files_count	manifest_stats.deleted_files_count	Moved to manifest_stats struct in content entry.
added_rows_count	manifest_stats.added_rows_count	Moved to manifest_stats struct in content entry.
existing_rows_count	manifest_stats.existing_rows_count	Moved to manifest_stats struct in content entry.
deleted_rows_count	manifest_stats.deleted_rows_counts	Moved to manifest_stats struct in content entry
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 content_entry.key_metadata
first_row_id	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 content entry fields

Manifest Entry field	V4 content entry field	Rationale or description
data_file	No separate struct element, the entire record for an entry in a	Manifest entries now have more generic content than prior

	V4 manifest will now be referred to as a content entry.	versions. An entry can either be a data file, DV, equality delete or a data/delete manifest. As a result, the entire structure of a record in a manifest has been renamed to content entry.
status	version_info.status	
snapshot_id	version_info.snapshot_id	
sequence_number	version_info.sequence_number	(data sequence number)
file_sequence_number	version_info.file_sequence_number	
data_file.file_path	location	Renamed, same ID
data_file.file_format	file_format	(Parquet, Avro, ORC, Puffin)
data_file.record_count	record_count	
data_file.file_size_in_bytes	file_size_in_bytes	
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.partition	REMOVED	Represented in column stats (need to support translation for equality deletes)Represented in column stats (need to support translation for equality deletes)

data_file.key_metadata	key_metadata	Carried over, still needed for encryption
data_file.split_offsets	split_offsets	Carried over
data_file.sort_order_id	sort_order_id	Carried over
data_file.referenced_data_file	referenced_file	Renamed to referenced_file to be able to support delete manifests expressing affinity to a data manifest.
data_file.content_offset	deletion_vector.offset	Grouping offset and content size required for DVs into a deletion_vector struct. Renaming to just offset because *content* is redundant in the context of being grouped under deletion_vector.
data_file.content_size_in_bytes	deletion_vector.size_in_bytes	Grouping offset and content size required for DVs into a deletion_vector struct. Renaming to just size_in_bytes because *content* is redundant in the context of being grouped under deletion_vector.
data_file.equality_ids	equality_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 tuple 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 timestamptz;
  upper_bound timestamptz;
  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(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: timestamptz>
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 manifestDVs 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

Bitvector compression benchmarking

https://github.com/amogh-jahagirdar/iceberg/tree/bitset-benchmarking

Implementation in
https://github.com/amogh-jahagirdar/iceberg/blob/bitset-benchmarking/core/src/jmh/java/org/apache/iceberg/BitsetCompressionBenchmark.java

```
None
./gradlew :iceberg-core:jmh -PjmhIncludeRegex=BitsetCompressionBenchmark
-PjmhOutputPath=benchmark/bitsetbenchmark.txt

grep "numBytes" core/benchmark/bitsetbenchmark.txt
```

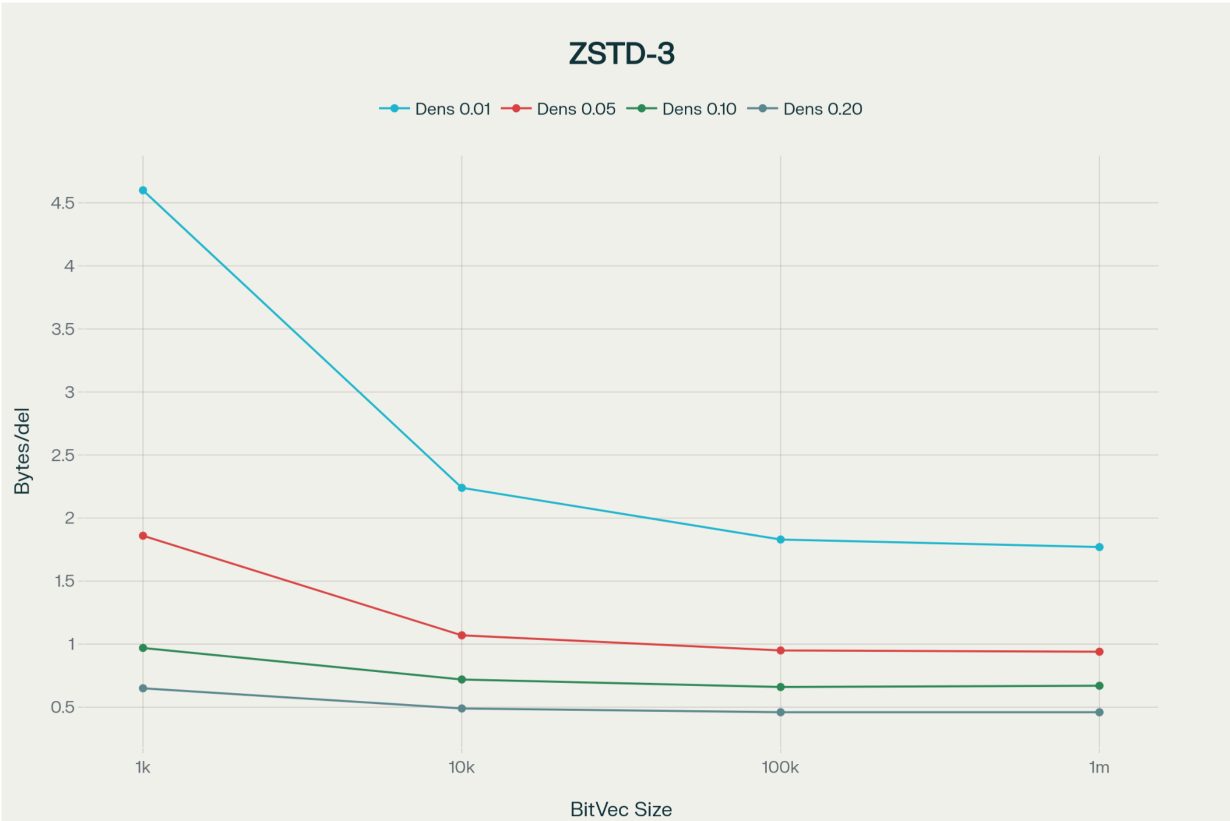
Compressed Sizes (bytes) of Bit Vectors by Algorithm, Density, and Scale for Random Positions

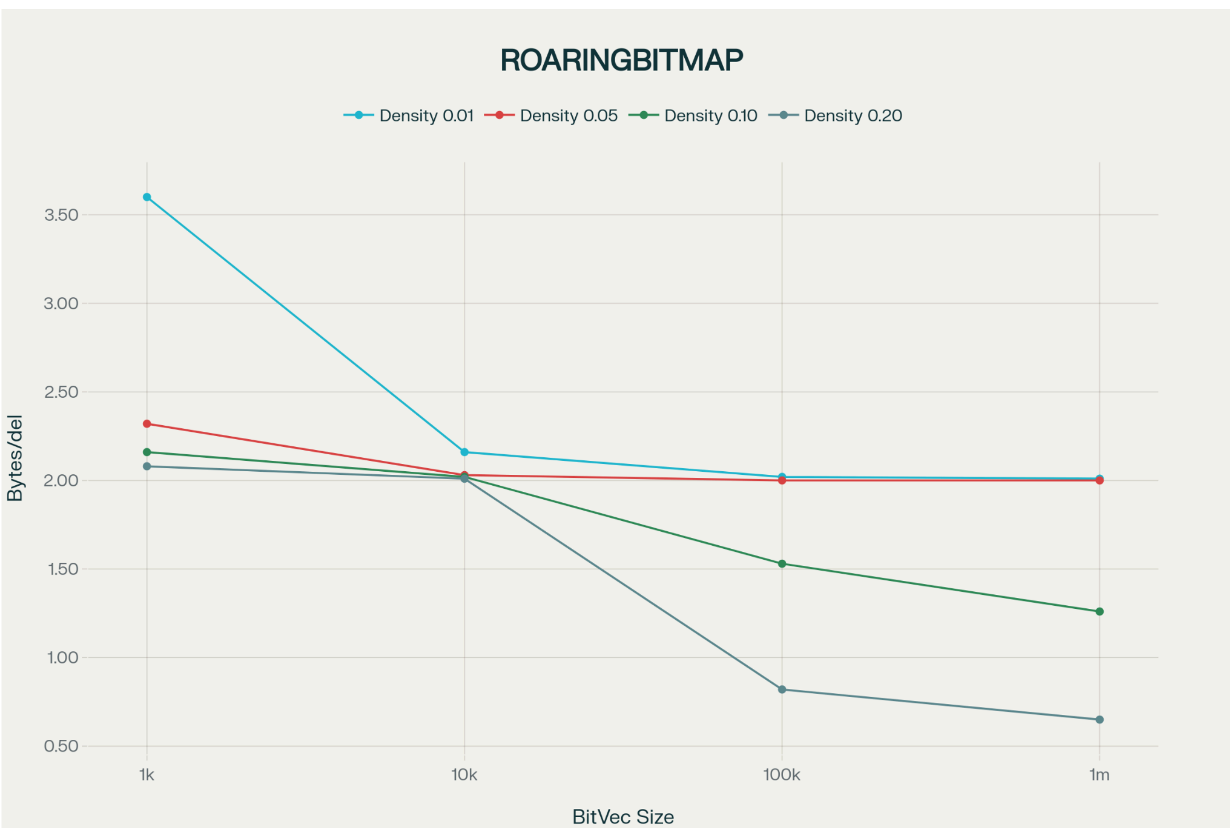
In parens, bytes/delete

Method	Density	1,000	10,000	100,000	1,000,000
LZ4	0.01	57 (5.70)	383 (3.83)	3,545 (3.54)	32,713 (3.27)
	0.05	98 (1.96)	920 (1.84)	8,480 (1.70)	75,451 (1.51)
	0.1	130 (1.30)	1,155 (1.16)	10,600 (1.06)	108,987 (1.09)
	0.2	135 (0.68)	1,264 (0.63)	12,541 (0.63)	125,493 (0.63)
Roaring Bitmap	0.01	36 (3.60)	216 (2.16)	2,024 (2.02)	20,136 (2.01)
	0.05	116 (2.32)	1,016 (2.03)	10,024 (2.00)	100,136 (2.00)
	0.1	216 (2.16)	2,016 (2.02)	15,340 (1.53)	126,440 (1.26)

	0.2	416 (2.08)	4,016 (2.01)	16,408 (0.82)	129,918 (0.65)
ZSTD-3	0.01	46 (4.60)	224 (2.24)	1,826 (1.83)	17,690 (1.77)
	0.05	93 (1.86)	533 (1.07)	4,727 (0.95)	46,987 (0.94)
	0.1	97 (0.97)	715 (0.72)	6,599 (0.66)	67,170 (0.67)
	0.2	129 (0.65)	981 (0.49)	9,227 (0.46)	91,923 (0.46)
Raw Bitset (dense encoding)	0.01	123 (12.3)	1,247 (12.47)	12,485 (12.48)	124,980 (12.50)
	0.05	123 (2.46)	1,247 (2.49)	12,494 (2.50)	124,985 (2.50)
	0.1	123 (1.23)	1,247 (1.25)	12,500 (1.25)	125,000 (1.25)
	0.2	125 (0.63)	1,250 (0.63)	12,500 (0.63)	125,000 (0.63)

Method	Density	1,000	10,000	100,000	1,000,000
LZ4	0.01	57	383	3,545	32,713
	0.05	98	920	8,480	75,451
	0.1	130	1,155	10,600	108,987
	0.2	135	1,264	12,541	125,493
Roaring Bitmap	0.01	36	216	2,024	20,136
	0.05	116	1,016	10,024	100,136
	0.1	216	2,016	15,340	126,440
	0.2	416	4,016	16,408	129,918
ZSTD-3	0.01	46	224	1,826	17,690
	0.05	93	533	4,727	46,987
	0.1	97	715	6,599	67,170
	0.2	129	981	9,227	91,923
Raw Bitset (dense encoding)	0.01	123	1,247	12,485	124,980
	0.05	123	1,247	12,494	124,985
	0.1	123	1,247	12,500	125,000
	0.2	125	1,250	12,500	125,000


ZSTD-3
Dens 0.01
Dens 0.05
Dens 0.10
Dens 0.20
Bytes/del
4.5
4
3.5
3
2.5
2
1.5
1
0.5
1k
10k
100k
1m
BitVec Size


ROARINGBITMAP
Density 0.01
Density 0.05
Density 0.10
Density 0.20
Bytes/del
3.50
3.00
2.50
2.00
1.50
1.00
0.50
1k
10k
100k
1m
BitVec Size

Brainstorming about detecting changes

1.) Diff between roots (Hold no previous state)
    a.) RootManifest(Added DV_1)
    b.) RootManifest(Added DV_2)
    c.) RootManifest(Added DV_3)
2.) Store DV Deltas Separately - Existing DV (Stored inline or Externally) + Change DV stored in root manifest
    a.) External DV lives in Puffin File
    b.) Perform a commit that adds more deletes via a Change DV in the root manifest
        i.) RootManifest (ChangeDV_1) + Puffin (StateDV)
    c.) Now I want to add another set of deletes
        i.) Rootmanifest (ChangeDV_2) + New State DV?
3.) Store Previously Removed DVs - Removed DV and New DV stored in Root Manifest
    a.) RootManifest (Existing DV_1)
    b.) RootManifest (Added DV_2, Removed DV_2)
    c.) RootManifest (Added DV_3, Removed DV_2)

References

- Column Stats Improvements

(tmp) change detection

Scenario 1: delete remaining files from a manifest file

A manifest file m1 has 10 entries. 7 of them are already deleted in a manifest DV m1_dv. The new commit needs to delete the remaining 3 entries. This could be the data TTL case of purging old partitions.

There are two options to encode this.

1) The new root manifest file trackThe new root manifest file tracks both manifest file m1 and manifest DV m1_dv as DELETED. The diff can be computed from applying m1_dv on m1 and the remaining 3 entries are the newly deleted data files.
2) s manifest file m1 as DELETED and a new **diff** DV m1_dv2 (with 3 entries) as ADDED. The diff can be computed as the 3 entries marked by the m1_dv2 are the newly deleted data files.

Both options can work and will likely have similar storage sizes.

Option 1 probably has the more initiative DV semantics. After applying m1_dv with m1, the remaining entries are the live data files before this commit. Now those remaining entries are deleted in this commit.

Scenario 2: rewrite manifest files

A manifest file m1 has 10 entries. 7 of them are already deleted in a manifest DV m1_dv.

The new commit produced a new manifest file m2 with the remaining 3 entries carried over from m1 with status as EXISTING.

The new root manifest file only tracks m2 as ADDED.

Scenario 3: combine manifest rewrite and DML operation

A manifest file m1 has 10 entries. 7 of them are already deleted in a manifest DV m1_dv.

The new commit needs to remove 1 file from m1 and add 5 new data files.

A new manifest file m2 is created with the following entries.

- 3 entries carried over from m1 with 2 in EXISTING state and 1 in DELETED state
- 5 new data files added with ADDED status

The new root manifest file tracks m1 and m1_dv as DELETED and m2 as ADDED.