Iceberg Table Spec

This is a specification for the Iceberg table format that is designed to manage a large, slow-changing collection of files in a distributed file system or key-value store as a table.

Format Versioning

Versions 1, 2 and 3 of the Iceberg spec are complete and adopted by the community.

Version 4 is under active development and has not been formally adopted.

The format version number is incremented when new features are added that will break forward-compatibility---that is, when older readers would not read newer table features correctly. Tables may continue to be written with an older version of the spec to ensure compatibility by not using features that are not yet implemented by processing engines.

Version 1: Analytic Data Tables

Version 1 of the Iceberg spec defines how to manage large analytic tables using immutable file formats: Parquet, Avro, and ORC.

All version 1 data and metadata files are valid after upgrading a table to version 2. Appendix E documents how to default version 2 fields when reading version 1 metadata.

Version 2: Row-level Deletes

Version 2 of the Iceberg spec adds row-level updates and deletes for analytic tables with immutable files.

The primary change in version 2 adds delete files to encode rows that are deleted in existing data files. This version can be used to delete or replace individual rows in immutable data files without rewriting the files.

In addition to row-level deletes, version 2 makes some requirements stricter for writers. The full set of changes are listed in Appendix E.

Version 3: Extended Types and Capabilities

Version 3 of the Iceberg spec extends data types and existing metadata structures to add new capabilities:

- New data types: nanosecond timestamp(tz), unknown, variant, geometry, geography
- Default value support for columns
- Multi-argument transforms for partitioning and sorting
- Row Lineage tracking
- Binary deletion vectors
- Table encryption keys

The full set of changes are listed in Appendix E.

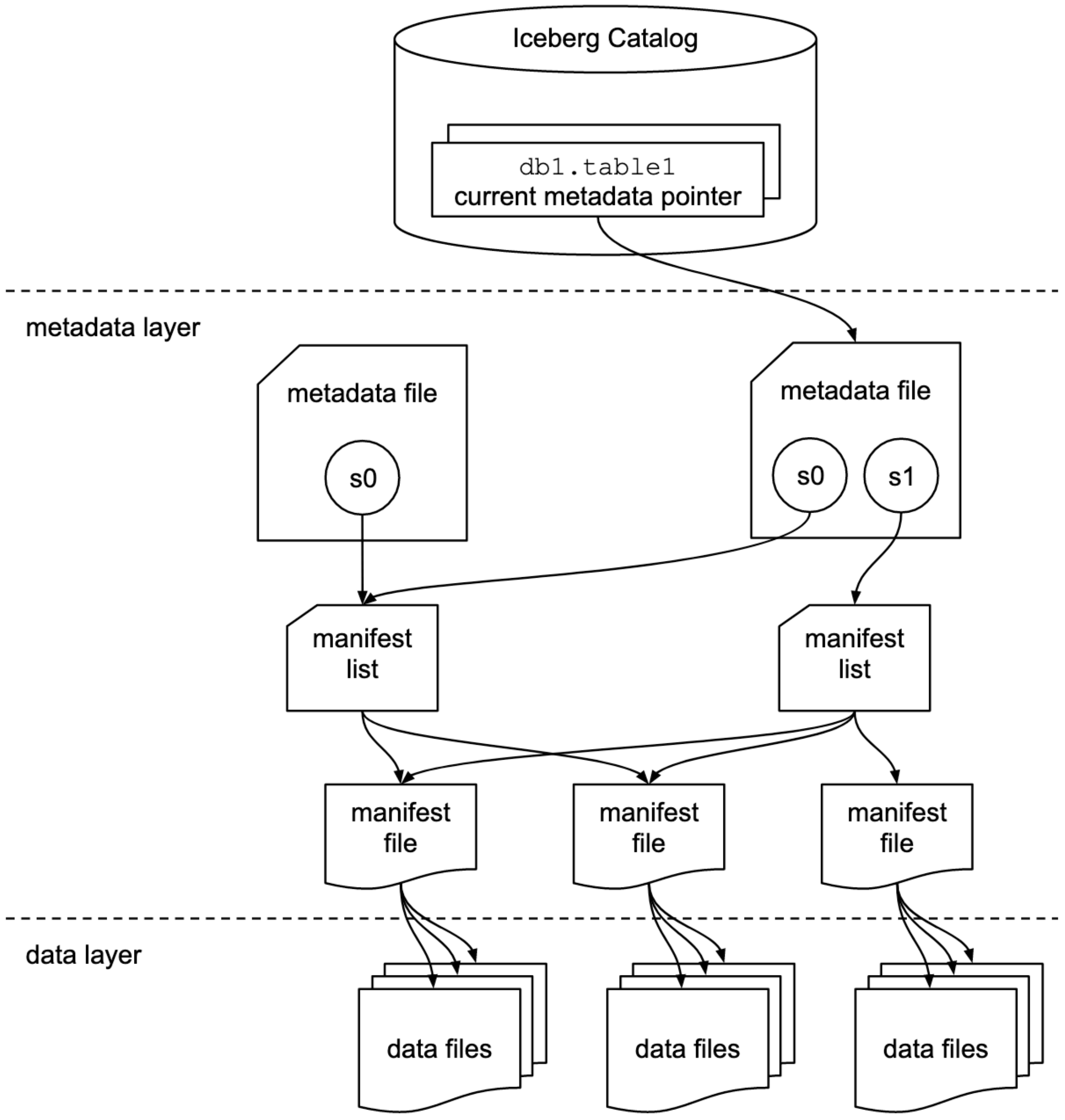
Goals

- **Serializable isolation** -- Reads will be isolated from concurrent writes and always use a committed snapshot of a table’s data. Writes will support removing and adding files in a single operation and are never partially visible. Readers will not acquire locks.
- **Speed** -- Operations will use O(1) remote calls to plan the files for a scan and not O(n) where n grows with the size of the table, like the number of partitions or files.
- **Scale** -- Job planning will be handled primarily by clients and not bottleneck on a central metadata store. Metadata will include

information needed for cost-based optimization.

- **Evolution** -- Tables will support full schema and partition spec evolution. Schema evolution supports safe column add, drop, reorder and rename, including in nested structures.
- **Dependable types** -- Tables will provide well-defined and dependable support for a core set of types.
- **Storage separation** -- Partitioning will be table configuration. Reads will be planned using predicates on data values, not partition values. Tables will support evolving partition schemes.
- **Formats** -- Underlying data file formats will support identical schema evolution rules and types. Both read-optimized and write-optimized formats will be available.

Overview



This table format tracks individual data files in a table instead of directories. This allows writers to create data files in-place and only adds files to the table in an explicit commit.

Table state is maintained in metadata files. All changes to table state create a new metadata file and replace the old metadata with an atomic swap. The table metadata file tracks the table schema, partitioning config, custom properties, and snapshots of the table contents. A snapshot represents the state of a table at some time and is used to access the complete set of data files in the table.

Data files in snapshots are tracked by one or more manifest files that contain a row for each data file in the table, the file's partition data, and its metrics. The data in a snapshot is the union of all files in its manifests. Manifest files are reused across snapshots to avoid rewriting metadata that is slow-changing. Manifests can track data files with any subset of a table and are not associated with partitions.

The manifests that make up a snapshot are stored in a manifest list file. Each manifest list stores metadata about manifests, including partition stats and data file counts. These stats are used to avoid reading manifests that are not required for an operation.

Optimistic Concurrency

An atomic swap of one table metadata file for another provides the basis for serializable isolation. Readers use the snapshot that was current when they load the table metadata and are not affected by changes until they refresh and pick up a new metadata location.

Writers create table metadata files optimistically, assuming that the current version will not be changed before the writer's commit. Once a writer has created an update, it commits by swapping the table’s metadata file pointer from the base version to the new version.

If the snapshot on which an update is based is no longer current, the writer must retry the update based on the new current version. Some operations support retry by re-applying metadata changes and committing, under well-defined conditions. For example, a change that rewrites files can be applied to a new table snapshot if all of the rewritten files are still in the table.

The conditions required by a write to successfully commit determines the isolation level. Writers can select what to validate and can make different isolation guarantees.

Sequence Numbers

The relative age of data and delete files relies on a sequence number that is assigned to every successful commit. When a snapshot is created for a commit, it is optimistically assigned the next sequence number, and it is written into the snapshot's metadata. If the commit fails and must be retried, the sequence number is reassigned and written into new snapshot metadata.

All manifests, data files, and delete files created for a snapshot inherit the snapshot's sequence number. Manifest file metadata in the manifest list stores a manifest's sequence number. New data and metadata file entries are written with `null` in place of a sequence number, which is replaced with the manifest's sequence number at read time. When a data or delete file is written to a new manifest (as "existing"), the inherited sequence number is written to ensure it does not change after it is first inherited.

Inheriting the sequence number from manifest metadata allows writing a new manifest once and reusing it in commit retries. To change a sequence number for a retry, only the manifest list must be rewritten -- which would be rewritten anyway with the latest set of manifests.

Row-level Deletes

Row-level deletes are stored in delete files.

There are two types of row-level deletes:

- **Position deletes** -- Mark a row deleted by data file path and the row position in the data file. Position deletes are encoded in a *position delete file* (V2) or *deletion vector* (V3 or above).
- **Equality deletes** -- Mark a row deleted by one or more column values, like id = 5. Equality deletes are encoded in *equality delete file*.

Like data files, delete files are tracked by partition. In general, a delete file must be applied to older data files with the same partition; see Scan Planning for details. Column metrics can be used to determine whether a delete file's rows overlap the contents of a data file or a scan range.

File System Operations

Iceberg only requires that file systems support the following operations:

- **In-place write** -- Files are not moved or altered once they are written.
- **Seekable reads** -- Data file formats require seek support.
- **Deletes** -- Tables delete files that are no longer used.

These requirements are compatible with object stores, like S3.

Tables do not require random-access writes. Once written, data and metadata files are immutable until they are deleted.

Tables do not require rename, except for tables that use atomic rename to implement the commit operation for new metadata files.

Specification

Terms

- **Schema** -- Names and types of fields in a table.
- **Partition spec** -- A definition of how partition values are derived from data fields.
- **Snapshot** -- The state of a table at some point in time, including the set of all data files.
- **Manifest list** -- A file that lists manifest files; one per snapshot.
- **Manifest** -- A file that lists data or delete files; a subset of a snapshot.
- **Data file** -- A file that contains rows of a table.
- **Delete file** -- A file that encodes rows of a table that are deleted by position or data values.

Writer requirements

Some tables in this spec have columns that specify requirements for tables by version. These requirements are intended for writers when adding metadata files (including manifests files and manifest lists) to a table with the given version.

Requirement	Write behavior
(blank)	The field should be omitted

Requirement	Write behavior
optional	The field can be written or omitted
required	The field must be written

Readers should be more permissive because v1 metadata files are allowed in v2 tables (or later) so that tables can be upgraded to without rewriting the metadata tree. For manifest list and manifest files, this table shows the expected read behavior for later versions:

v1	v2	v2+ read behavior
	optional	Read the field as *optional*
	required	Read the field as *optional*; it may be missing in v1 files
optional		Ignore the field
optional	*optional*	Read the field as *optional*
optional	*required*	Read the field as *optional*; it may be missing in v1 files
required		Ignore the field
required	*optional*	Read the field as *optional*
required	*required*	Fill in a default or throw an exception if the field is missing

If a later version is not shown, the requirement for a version is not changed from the most recent version shown. For example, v3 uses the same requirements as v2 if a table shows only v1 and v2 requirements.

Readers may be more strict for metadata JSON files because the JSON files are not reused and will always match the table version. Required fields that were not present in or were optional in prior versions may be handled as required fields. For example, a v2 table that is missing `last-sequence-number` can throw an exception.

Writing data files

All columns must be written to data files even if they introduce redundancy with metadata stored in manifest files (e.g. columns with identity partition transforms). Writing all columns provides a backup in case of corruption or bugs in the metadata layer.

Writers are not allowed to commit files with a partition spec that contains a field with an unknown transform.

Schemas and Data Types

A table's **schema** is a list of named columns. All data types are either primitives or nested types, which are maps, lists, or structs. A table schema is also a struct type.

For the representations of these types in Avro, ORC, and Parquet file formats, see Appendix A.

Nested Types

A **`struct`** is a tuple of typed values. Each field in the tuple is named and has an integer id that is unique in the table schema. Each field can be either optional or required, meaning that values can (or cannot) be null. Fields may be any type. Fields may have an optional comment or doc string. Fields can have default values.

A **`list`** is a collection of values with some element type. The element field has an integer id that is unique in the table schema. Elements can be either optional or required. Element types may be any type.

A **`map`** is a collection of key-value pairs with a key type and a value type. Both the key field and value field each have an integer id that is unique in the table schema. Map keys are required and map values can be either optional or required. Both map keys and map values may be any type, including nested types.

Semi-structured Types

A `variant` is a value that stores semi-structured data. The structure and data types in a variant are not necessarily consistent across rows in a table or data file. The variant type and binary encoding are defined in the Parquet project, with support currently available for V1. Support for Variant is added in Iceberg v3.

Variants are similar to JSON with a wider set of primitive values including date, timestamp, timestamptz, binary, and decimals.

Variant values may contain nested types:

1. An array is an ordered collection of variant values.
2. An object is a collection of fields that are a string key and a variant value.

As a semi-structured type, there are important differences between variant and Iceberg's other types:

1. Variant arrays are similar to lists, but may contain any variant value rather than a fixed element type.
2. Variant objects are similar to structs, but may contain variable fields identified by name and field values may be any variant value rather than a fixed field type.

Primitive Types

Supported primitive types are defined in the table below. Primitive types added after v1 have an "added by" version that is the first spec version in which the type is allowed. For example, nanosecond-precision timestamps are part of the v3 spec; using v3 types in v1 or v2 tables can break forward compatibility.

Added by version	Primitive type	Description	Requirements
v3	`unknown`	Default / null column type used when a more specific type is not known	Must be optional with `null` defaults; not stored in data files
	`boolean`	True or false	
	`int`	32-bit signed integers	Can promote to `long`
	`long`	64-bit signed integers	
	`float`	32-bit IEEE 754 floating point	Can promote to double
	`double`	64-bit IEEE 754 floating point	
	`decimal(P, S)`	Fixed-point decimal; precision P, scale S	Scale is fixed, precision must be 38 or less
	`date`	Calendar date without timezone or time	
	`time`	Time of day, microsecond precision, without date, timezone	

Added by version	Primitive type	Description	Requirements
	`timestamp`	Timestamp, microsecond precision, without timezone	[1]
	`timestamptz`	Timestamp, microsecond precision, with timezone	[2]
v3	`timestamp_ns`	Timestamp, nanosecond precision, without timezone	[1]
v3	`timestamptz_ns`	Timestamp, nanosecond precision, with timezone	[2]
	`string`	Arbitrary-length character sequences	Encoded with UTF-8 [3]
	`uuid`	Universally unique identifiers	Should use 16-byte fixed
	`fixed(L)`	Fixed-length byte array of length L	
	`binary`	Arbitrary-length byte array	

Added by version	Primitive type	Description	Requirements
v3	**`geometry(C)`**	Geospatial features from OGC – Simple feature access. Edge-interpolation is always linear/planar. See Appendix G. Parameterized by CRS C. If not specified, C is `OGC:CRS84`.	
v3	**`geography(C, A)`**	Geospatial features from OGC – Simple feature access. See Appendix G. Parameterized by CRS C and edge-interpolation algorithm A. If not specified, C is `OGC:CRS84` and A is `spherical`.	

Notes:

1. Timestamp values *without time zone* represent a date and time of day regardless of zone: the time value is independent of zone adjustments (`2017-11-16 17:10:34` is always retrieved as `2017-11-16 17:10:34`).
2. Timestamp values *with time zone* represent a point in time: values are stored as UTC and do not retain a source time zone (`2017-11-16 17:10:34 PST` is stored/retrieved as `2017-11-17 01:10:34 UTC` and these values are considered identical).
3. Character strings must be stored as UTF-8 encoded byte arrays.

For details on how to serialize a schema to JSON, see Appendix C.

CRS

For `geometry` and `geography` types, the parameter C refers to the CRS (coordinate reference system), a mapping of how coordinates refer to locations on Earth.

For `geometry` type, the CRS does not affect geometric calculations, which are always Cartesian.

The default CRS value `OGC:CRS84` means that the objects must be stored in longitude, latitude based on the WGS84 datum.

Custom CRS values can be specified by a string of the format `type:identifier`, where `type` is one of the following values:

- `srid`: Spatial reference identifier, `identifier` is the SRID itself.
- `projjson`: PROJJSON, `identifier` is the name of a table property where the projjson string is stored.

For `geography` types, the custom CRS must be geographic, with longitudes bound by [-180, 180] and latitudes bound by [-90, 90].

EDGE-INTERPOLATION ALGORITHM

For `geography` types, an additional parameter A specifies an algorithm for interpolating edges, and is one of the following values:

- `spherical`: edges are interpolated as geodesics on a sphere.
- `vincenty`: https://en.wikipedia.org/wiki/Vincenty%27s_formulae
- `thomas`: Thomas, Paul D. Spheroidal geodesics, reference systems, & local geometry. US Naval Oceanographic Office, 1970.
- `andoyer`: Thomas, Paul D. Mathematical models for navigation systems. US Naval Oceanographic Office, 1965.
- `karney`: Karney, Charles FF. "Algorithms for geodesics." Journal of Geodesy 87 (2013): 43-55, and GeographicLib

Default values

Default values can be tracked for struct fields (both nested structs and the top-level schema's struct). There can be two defaults with a field:

- `initial-default` is used to populate the field's value for all records that were written before the field was added to the schema
- `write-default` is used to populate the field's value for any records written after the field was added to the schema, if the writer does not supply the field's value

The `initial-default` is set only when a field is added to an existing schema. The `write-default` is initially set to the same value as `initial-default` and can be changed through schema evolution. If either default is not set for an optional field, then the default value is null for compatibility with older spec versions.

The `initial-default` and `write-default` produce SQL default value behavior, without rewriting data files. SQL default value behavior when a field is added handles all existing rows as though the rows were written with the new field's default value. Default value changes may only affect future records and all known fields are written into data files. Omitting a known field when writing a data file is never allowed. The write default for a field must be written if a field is not supplied to a write. If the write default for a required field is not set, the writer must fail.

All columns of `unknown`, `variant`, `geometry`, and `geography` types must default to null. Non-null values for `initial-default` or `write-default` are invalid.

Default values for the fields of a struct are tracked as `initial-default` and `write-default` at the field level. Default values for fields that are nested structs must not contain default values for the struct's fields (sub-fields). Sub-field defaults are tracked in sub-field's metadata. As a result, the default stored for a nested struct may be either null or a non-null struct with no field values. The effective default value is produced by setting each fields' default in a new struct.

For example, a struct column `point` with fields `x` (default 0) and `y` (default 0) can be defaulted to `{"x": 0, "y": 0}` or `null`. A non-null default is stored by setting `initial-default` or `write-default` to an empty struct (`{}`) that will use field values set from each field's `initial-default` or `write-default`, respectively.

`point` default	`point.x` default	`point.y` default	Data value	Result value
`null`	`0`	`0`	(missing)	`null`
`null`	`0`	`0`	`{"x": 3}`	`{"x": 3, "y": 0}`
`{}`	`0`	`0`	(missing)	`{"x": 0, "y": 0}`
`{}`	`0`	`0`	`{"y": -1}`	`{"x": 0, "y": -1}`

Default values are attributes of fields in schemas and serialized with fields in the JSON format. See Appendix C.

Schema Evolution

Schemas may be evolved by type promotion or adding, deleting, renaming, or reordering fields in structs (both nested structs and the top-level schema's struct).

Evolution applies changes to the table's current schema to produce a new schema that is identified by a unique schema ID, is added to the table's list of schemas, and is set as the table's current schema.

Valid primitive type promotions are:

Primitive type	v1, v2 valid type promotions	v3+ valid type promotions	Requirements
`unknown`		*any type*	
`int`	`long`	`long`	
`date`		`timestamp`, `timestamp_ns`	Promotion to `timestamptz` or `timestamptz_ns` is **not** allowed; values outside the promoted type's range must result in a runtime failure
`float`	`double`	`double`	
`decimal(P, S)`	`decimal(P', S)` if `P' > P`	`decimal(P', S)` if `P' > P`	Widen precision only

Iceberg's Avro manifest format does not store the type of lower and upper bounds, and type promotion does not rewrite existing bounds. For example, when a `float` is promoted to `double`, existing data file bounds are encoded as 4 little-endian bytes rather than 8 little-endian bytes for `double`. To correctly decode the value, the original type at the time the file was written must be inferred according to the following table:

Current type	Length of bounds	Inferred type at write time
`long`	4 bytes	`int`
`long`	8 bytes	`long`

Current type	Length of bounds	Inferred type at write time
`double`	4 bytes	`float`
`double`	8 bytes	`double`
`timestamp`	4 bytes	`date`
`timestamp`	8 bytes	`timestamp`
`timestamp_ns`	4 bytes	`date`
`timestamp_ns`	8 bytes	`timestamp_ns`
`decimal(P, S)`	*any*	`decimal(P', S)`; `P' <= P`

Type promotion is not allowed for a field that is referenced by `source-id` or `source-ids` of a partition field if the partition transform would produce a different value after promoting the type. For example, `bucket[N]` produces different hash values for `34` and `"34"` (2017239379 != -427558391) but the same value for `34` and `34L`; when an `int` field is the source for a bucket partition field, it may be promoted to `long` but not to `string`. This may happen for the following type promotion cases:

- `date` to `timestamp` or `timestamp_ns`

Any struct, including a top-level schema, can evolve through deleting fields, adding new fields, renaming existing fields, reordering existing fields, or promoting a primitive using the valid type promotions. Adding a new field assigns a new ID for that field and for any nested fields. Renaming an existing field must change the name, but not the field ID. Deleting a field removes it from the current schema. Field deletion cannot be rolled back unless the field was nullable or if the current snapshot has not changed.

Grouping a subset of a struct's fields into a nested struct is **not** allowed, nor is moving fields from a nested struct into its immediate parent struct (`struct<a,`

`b, c> ↔ struct<a, struct<b, c>>`). Evolving primitive types to structs is **not** allowed, nor is evolving a single-field struct to a primitive (`map<string, int> ↔ map<string, struct<int>>`).

Struct evolution requires the following rules for default values:

- The `initial-default` must be set when a field is added and cannot change
- The `write-default` must be set when a field is added and may change
- When a required field is added, both defaults must be set to a non-null value
- When an optional field is added, the defaults may be null and should be explicitly set
- When a field that is a struct type is added, its default may only be null or a non-null struct with no field values. Default values for fields must be stored in field metadata.
- If a field value is missing from a struct's `initial-default`, the field's `initial-default` must be used for the field
- If a field value is missing from a struct's `write-default`, the field's `write-default` must be used for the field

COLUMN PROJECTION

Columns in Iceberg data files are selected by field id. The table schema's column names and order may change after a data file is written, and projection must be done using field ids.

Values for field ids which are not present in a data file must be resolved according the following rules:

- Return the value from partition metadata if an Identity Transform exists for the field and the partition value is present in the `partition` struct on `data_file` object in the manifest. This allows for metadata only migrations of Hive tables.

- Use `schema.name-mapping.default` metadata to map field id to columns without field id as described below and use the column if it is present.
- Return the default value if it has a defined `initial-default` (See Default values section for more details).
- Return `null` in all other cases.

For example, a file may be written with schema `1: a int, 2: b string, 3: c double` and read using projection schema `3: measurement, 2: name, 4: a`. This must select file columns `c` (renamed to `measurement`), `b` (now called `name`), and a column of `null` values called `a`; in that order.

Tables may also define a property `schema.name-mapping.default` with a JSON name mapping containing a list of field mapping objects. These mappings provide fallback field ids to be used when a data file does not contain field id information. Each object should contain

- `names`: A required list of 0 or more names for a field.
- `field-id`: An optional Iceberg field ID used when a field's name is present in `names`
- `fields`: An optional list of field mappings for child field of structs, maps, and lists.

Field mapping fields are constrained by the following rules:

- A name may contain `.` but this refers to a literal name, not a nested field. For example, `a.b` refers to a field named `a.b`, not child field `b` of field `a`.
- Each child field should be defined with their own field mapping under `fields`.

- Multiple values for `names` may be mapped to a single field ID to support cases where a field may have different names in different data files. For example, all Avro field aliases should be listed in `names`.
- Fields which exist only in the Iceberg schema and not in imported data files may use an empty `names` list.
- Fields that exist in imported files but not in the Iceberg schema may omit `field-id`.
- List types should contain a mapping in `fields` for `element`.
- Map types should contain mappings in `fields` for `key` and `value`.
- Struct types should contain mappings in `fields` for their child fields.

For details on serialization, see Appendix C.

Identifier Field IDs

A schema can optionally track the set of primitive fields that identify rows in a table, using the property `identifier-field-ids` (see JSON encoding in Appendix C).

Two rows are the "same"---that is, the rows represent the same entity---if the identifier fields are equal. However, uniqueness of rows by this identifier is not guaranteed or required by Iceberg and it is the responsibility of processing engines or data providers to enforce.

Identifier fields may be nested in structs but cannot be nested within maps or lists. Float, double, and optional fields cannot be used as identifier fields and a nested field cannot be used as an identifier field if it is nested in an optional struct, to avoid null values in identifiers.

Reserved Field IDs

Iceberg tables must not use field ids greater than 2147483447 (`Integer.MAX_VALUE - 200`). This id range is reserved for metadata columns that can be used in user data schemas, like the `_file` column that holds the file path in which a row was stored.

The set of metadata columns is:

Field id, name	Type	Description
`2147483646 _file`	`string`	Path of the file in which a row is stored
`2147483645 _pos`	`long`	Ordinal position of a row in the source data file, starting at `0`
`2147483644 _deleted`	`boolean`	Whether the row has been deleted
`2147483643 _spec_id`	`int`	Spec ID used to track the file containing a row
`2147483642 _partition`	`struct`	Partition to which a row belongs
`2147483546 file_path`	`string`	Path of a file, used in position-based delete files
`2147483545 pos`	`long`	Ordinal position of a row, used in position-based delete files
`2147483544 row`	`struct<...>`	Deleted row values, used in position-based delete files
`2147483543 _change_type`	`string`	The record type in the changelog (INSERT, DELETE, UPDATE_BEFORE, or UPDATE_AFTER)
`2147483542 _change_ordinal`	`int`	The order of the change

Field id, name	Type	Description
`2147483541 _commit_snapshot_id`	`long`	The snapshot ID in which the change occurred
`2147483540 _row_id`	`long`	A unique long assigned for row lineage, see Row Lineage
`2147483539 _last_updated_sequence_number`	`long`	The sequence number which last updated this row, see Row Lineage

Row Lineage

In v3 and later, an Iceberg table must track row lineage fields for all newly created rows. Engines must maintain the `next-row-id` table field and the following row-level fields when writing data files:

- `_row_id` a unique long identifier for every row within the table. The value is assigned via inheritance when a row is first added to the table.
- `_last_updated_sequence_number` the sequence number of the commit that last updated a row. The value is inherited when a row is first added or modified.

These fields are assigned and updated by inheritance because the commit sequence number and starting row ID are not assigned until the snapshot is successfully committed. Inheritance is used to allow writing data and manifest files before values are known so that it is not necessary to rewrite data and manifest files when an optimistic commit is retried.

Row lineage does not track lineage for rows updated via Equality Deletes, because engines using equality deletes avoid reading existing data before writing changes and can't provide the original row ID for the new rows. These updates are always treated as if the existing row was completely removed and a unique new row was added.

ROW LINEAGE ASSIGNMENT

When a row is added or modified, the `_last_updated_sequence_number` field is set to `null` so that it is inherited when reading. Similarly, the `_row_id` field for an added row is set to `null` and assigned when reading.

A data file with only new rows for the table may omit the `_last_updated_sequence_number` and `_row_id`. If the columns are missing, readers should treat both columns as if they exist and are set to null for all rows.

On read, if `_last_updated_sequence_number` is `null` it is assigned the `sequence_number` of the data file's manifest entry. The data sequence number of a data file is documented in Sequence Number Inheritance.

When `null`, a row's `_row_id` field is assigned to the `first_row_id` from its containing data file plus the row position in that data file (`_pos`). A data file's `first_row_id` field is assigned using inheritance and is documented in First Row ID Inheritance. A manifest's `first_row_id` is assigned when writing the manifest list for a snapshot and is documented in First Row ID Assignment. A snapshot's `first-row-id` is set to the table's `next-row-id` and is documented in Snapshot Row IDs.

When an existing row is moved to a different data file for any reason, writers should write `_row_id` and `_last_updated_sequence_number` according to the following rules:

1. The row's existing non-null `_row_id` must be copied into the new data file
2. If the write has modified the row, the `_last_updated_sequence_number` field must be set to `null` (so that the modification's sequence number replaces the current value)
3. If the write has not modified the row, the existing non-null `_last_updated_sequence_number` value must be copied to the new data file

Engines may model operations as deleting/inserting rows or as modifications to rows that preserve row ids.

ROW LINEAGE EXAMPLE

This example demonstrates how `_row_id` and `_last_updated_sequence_number` are assigned for a snapshot. This starts with a table with a `next-row-id` of 1000.

Writing a new append snapshot creates snapshot metadata with `first-row-id` assigned to the table's `next-row-id`:

```
{
  "operation": "append",
  "first-row-id": 1000,
  ...
}
```

The snapshot's manifest list will contain existing manifests, plus new manifests that are each assigned a `first_row_id` based on the `added_rows_count` and `existing_rows_count` of preceding new manifests:

`manifest_path`	`added_rows_count`	`existing_rows_count`	`first_row_id`
...	...	...	...
existing	75	0	925
added1	100	25	1000
added2	0	100	1125
added3	125	25	1225

The existing manifests are written with the `first_row_id` assigned when the manifests were added to the table.

The first added manifest, `added1`, is assigned the same `first_row_id` as the snapshot and each of the remaining added manifests are assigned a `first_row_id` based on the number of rows in preceding manifests that were assigned a `first_row_id`.

Note that the second file, `added2`, changes the `first_row_id` of the next manifest even though it contains no added data files because any data file without a `first_row_id` could be assigned one, even if it has existing status. This is optional if the writer knows that existing data files in the manifest have assigned `first_row_id` values.

Within `added1`, the first added manifest, each data file's `first_row_id` follows a similar pattern:

`status`	`file_path`	`record_count`	`first_row_id`
EXISTING	data1	25	800
ADDED	data2	50	null (1000)
ADDED	data3	50	null (1050)

The `first_row_id` of the EXISTING file `data1` was already assigned, so the file metadata was copied into manifest `added1`.

Files `data2` and `data3` are written with `null` for `first_row_id` and are assigned `first_row_id` at read time based on the manifest's `first_row_id` and the `record_count` of previous files without `first_row_id` in this manifest: (1,000 + 0) and (1,000 + 50).

The snapshot then populates the total number of `added-rows` based on the sum of all added rows in the manifests: 100 (50 + 50)

When the new snapshot is committed, the table's `next-row-id` must also be updated (even if the new snapshot is not in the main branch). Because 375 rows were in data files in manifests that were assigned a `first_row_id` (`added1` 100+25, `added2` 0+100, `added3` 125+25) the new value is 1,000 + 375 = 1,375.

ROW LINEAGE FOR UPGRADED TABLES

When a table is upgraded to v3, its `next-row-id` is initialized to 0 and existing snapshots are not modified (that is, `first-row-id` remains unset or null). For such snapshots without `first-row-id`, `first_row_id` values for data files and data manifests are null, and values for `_row_id` are read as null for all rows. When `first_row_id` is null, inherited row ID values are also null.

Snapshots that are created after upgrading to v3 must set the snapshot's `first-row-id` and assign row IDs to existing and added files in the snapshot. When writing the manifest list, all data manifests must be assigned a `first_row_id`, which assigns a `first_row_id` to all data files via inheritance.

Note that:

- Snapshots created before upgrading to v3 do not have row IDs.
- After upgrading, new snapshots in different branches will assign disjoint ID ranges to existing data files, based on the table's `next-row-id` when the snapshot is committed. For a data file in multiple branches, a writer may write the `first_row_id` from another branch or may assign a new `first_row_id` to the data file (to avoid large metadata rewrites).
- Existing rows will inherit `_last_updated_sequence_number` from their containing data file.

Partitioning

Data files are stored in manifests with a tuple of partition values that are used in scans to filter out files that cannot contain records that match the scan's filter predicate. Partition values for a data file must be the same for all records stored in the data file. (Manifests store data files from any partition, as long as the partition spec is the same for the data files.)

Tables are configured with a **partition spec** that defines how to produce a tuple of partition values from a record. A partition spec has a list of fields that consist of:

- A **source column id** or a list of **source column ids** from the table's schema
- A **partition field id** that is used to identify a partition field and is unique within a partition spec. In v2 table metadata, it is unique across all partition specs.
- A **transform** that is applied to the source column(s) to produce a partition value
- A **partition name**

The source columns, selected by ids, must be a primitive type and cannot be contained in a map or list, but may be nested in a struct. For details on how to serialize a partition spec to JSON, see Appendix C.

Partition specs capture the transform from table data to partition values. This is used to transform predicates to partition predicates, in addition to transforming data values. Deriving partition predicates from column predicates on the table data is used to separate the logical queries from physical storage: the partitioning can change and the correct partition filters are always derived from column predicates. This simplifies queries because users don't have to supply both logical predicates and partition predicates. For more information, see Scan Planning below.

Partition fields that use an unknown transform can be read by ignoring the partition field for the purpose of filtering data files during scan planning. In v1 and v2, readers should ignore fields with unknown transforms while reading; this behavior is required in v3. Writers are not allowed to commit data using a partition spec that contains a field with an unknown transform.

Two partition specs are considered equivalent with each other if they have the same number of fields and for each corresponding field, the fields have the same source column IDs, transform definition and partition name. Writers must not create a new partition spec if there already exists a compatible partition spec defined in the table.

Partition field IDs must be reused if an existing partition spec contains an equivalent field.

Partition Transforms

Transform name	Description	Source types	Result type
`identity`	Source value, unmodified	Any except for `geometry`, `geography`, and `variant`	Source type
`bucket[N]`	Hash of value, mod `N` (see below)	`int`, `long`, `decimal`, `date`, `time`, `timestamp`, `timestamptz`, `timestamp_ns`, `timestamptz_ns`, `string`, `uuid`, `fixed`, `binary`	`int`
`truncate[W]`	Value truncated to width `W` (see below)	`int`, `long`, `decimal`, `string`, `binary`	Source type
`year`	Extract a date or timestamp year, as years from 1970	`date`, `timestamp`, `timestamptz`, `timestamp_ns`, `timestamptz_ns`	`int`
`month`	Extract a date or timestamp month, as months from 1970-01-01	`date`, `timestamp`, `timestamptz`, `timestamp_ns`, `timestamptz_ns`	`int`
`day`	Extract a date or timestamp day, as days from 1970-01-01	`date`, `timestamp`, `timestamptz`, `timestamp_ns`, `timestamptz_ns`	`int`

Transform name	Description	Source types	Result type
`hour`	Extract a timestamp hour, as hours from 1970-01-01 00:00:00	`timestamp`, `timestamptz`, `timestamp_ns`, `timestamptz_ns`	`int`
`void`	Always produces `null`	Any	Source type or `int`

All transforms must return `null` for a `null` input value.

The `void` transform may be used to replace the transform in an existing partition field so that the field is effectively dropped in v1 tables. See partition evolution below.

Bucket Transform Details

Bucket partition transforms use a 32-bit hash of the source value. The 32-bit hash implementation is the 32-bit Murmur3 hash, x86 variant, seeded with 0.

Transforms are parameterized by a number of buckets [1], `N`. The hash mod `N` must produce a positive value by first discarding the sign bit of the hash value. In pseudo-code, the function is:

```
  def bucket_N(x) = (murmur3_x86_32_hash(x) &
Integer.MAX_VALUE) % N
```

Notes:

1. Changing the number of buckets as a table grows is possible by evolving the partition spec.

For hash function details by type, see Appendix B.

Truncate Transform Details

Type	Config	Truncate specification	Examples
`int`	`W`, width	`v - (v % W)` remainders must be positive [1]	`W=10`: `1` → `0`, `-1` → `-10`
`long`	`W`, width	`v - (v % W)` remainders must be positive [1]	`W=10`: `1` → `0`, `-1` → `-10`
`decimal`	`W`, width (no scale)	`scaled_W = decimal(W, scale(v))` `v - (v % scaled_W)` [1, 2]	`W=50`, `s=2`: `10.65` → `10.50`
`string`	`L`, length	Substring of length `L`: `v.substring(0, L)` [3]	`L=3`: `iceberg` → `ice`
`binary`	`L`, length	Sub array of length `L`: `v.subarray(0, L)` [4]	`L=3`: `\x01\x02\x03\x04\x05` → `\x01\x02\x03`

Notes:

1. The remainder, `v % W`, must be positive. For languages where `%` can produce negative values, the correct truncate function is: `v - (((v % W) + W) % W)`
2. The width, `W`, used to truncate decimal values is applied using the scale of the decimal column to avoid additional (and potentially conflicting) parameters.
3. Strings are truncated to a valid UTF-8 string with no more than `L` code points.

4. In contrast to strings, binary values do not have an assumed encoding and are truncated to `L` bytes.

Partition Evolution

Table partitioning can be evolved by adding, removing, renaming, or reordering partition spec fields.

Changing a partition spec produces a new spec identified by a unique spec ID that is added to the table's list of partition specs and may be set as the table's default spec.

When evolving a spec, changes should not cause partition field IDs to change because the partition field IDs are used as the partition tuple field IDs in manifest files.

In v2, partition field IDs must be explicitly tracked for each partition field. New IDs are assigned based on the last assigned partition ID in table metadata.

In v1, partition field IDs were not tracked, but were assigned sequentially starting at 1000 in the reference implementation. This assignment caused problems when reading metadata tables based on manifest files from multiple specs because partition fields with the same ID may contain different data types. For compatibility with old versions, the following rules are recommended for partition evolution in v1 tables:

1. Do not reorder partition fields
2. Do not drop partition fields; instead replace the field's transform with the `void` transform
3. Only add partition fields at the end of the previous partition spec

Sorting

Users can sort their data within partitions by columns to gain performance. The information on how the data is sorted can be declared per data or delete file, by a **sort order**.

A sort order is defined by a sort order id and a list of sort fields. The order of the sort fields within the list defines the order in which the sort is applied to the data. Each sort field consists of:

- A **source column id** or a list of **source column ids** from the table's schema
- A **transform** that is used to produce values to be sorted on from the source column(s). This is the same transform as described in partition transforms.
- A **sort direction**, that can only be either `asc` or `desc`
- A **null order** that describes the order of null values when sorted. Can only be either `nulls-first` or `nulls-last`

For details on how to serialize a sort order to JSON, see Appendix C.

Order id `0` is reserved for the unsorted order.

Sorting floating-point numbers should produce the following behavior: `-NaN` < `-Infinity` < `-value` < `-0` < `0` < `value` < `Infinity` < `NaN`. This aligns with the implementation of Java floating-point types comparisons.

A data or delete file is associated with a sort order by the sort order's id within a manifest. Therefore, the table must declare all the sort orders for lookup. A table could also be configured with a default sort order id, indicating how the new data should be sorted by default. Writers should use this default sort order to sort the data on write, but are not required to if the default order is prohibitively expensive, as it would be for streaming writes.

Manifests

A manifest is an immutable Avro file that lists data files or delete files, along with each file’s partition data tuple, metrics, and tracking information. One or more manifest files are used to store a snapshot, which tracks all of the files in a table at some point in time. Manifests are tracked by a manifest list for each table snapshot.

A manifest is a valid Iceberg data file: files must use valid Iceberg formats, schemas, and column projection.

A manifest may store either data files or delete files, but not both because manifests that contain delete files are scanned first during job planning. Whether a manifest is a data manifest or a delete manifest is stored in manifest metadata.

A manifest stores files for a single partition spec. When a table's partition spec changes, old files remain in the older manifest and newer files are written to a new manifest. This is required because a manifest file's schema is based on its partition spec (see below). The partition spec of each manifest is also used to transform predicates on the table's data rows into predicates on partition values that are used during job planning to select files from a manifest.

A manifest file must store the partition spec and other metadata as properties in the Avro file's key-value metadata:

v1	v2	Key	Value
required	*required*	`schema`	JSON representation of the table schema at the time the manifest was written
optional	*required*	`schema-id`	ID of the schema used to write the manifest as a string
required	*required*	`partition-spec`	JSON representation of only the partition fields array of the partition spec used to write the manifest. See Appendix C
optional	*required*	`partition-spec-id`	ID of the partition spec used to write the manifest as a string

v1	v2	Key	Value
optional	*required*	`format-version`	Table format version number of the manifest as a string
	required	`content`	Type of content files tracked by the manifest: "data" or "deletes"

The schema of a manifest file is defined by the `manifest_entry` struct, described in the following section.

Manifest Entry Fields

The `manifest_entry` struct consists of the following fields:

v1	v2	Field id, name	Type	Description
required	*required*	**`0 status`**	`int` with meaning: `0: EXISTING` `1: ADDED` `2: DELETED`	Used to track additions and deletions. Deletes are informational only and not used in scans.
required	*optional*	**`1 snapshot_id`**	`long`	Snapshot id where the file was added, or deleted if status is 2. Inherited when null.

v1	v2	Field id, name	Type	Description
	optional	`3 sequence_number`	`long`	Data sequence number of the file. Inherited when null and status is 1 (added).
	optional	`4 file_sequence_number`	`long`	File sequence number indicating when the file was added. Inherited when null and status is 1 (added).
required	*required*	`2 data_file`	`data_file struct` (see below)	File path, partition tuple, metrics, ...

The manifest entry fields are used to keep track of the snapshot in which files were added or logically deleted. The `data_file` struct, defined below, is nested inside the manifest entry so that it can be easily passed to job planning without the manifest entry fields.

When a file is added to the dataset, its manifest entry should store the snapshot ID in which the file was added and set status to 1 (added).

When a file is replaced or deleted from the dataset, its manifest entry fields store the snapshot ID in which the file was deleted and status 2 (deleted). The file may be deleted from the file system when the snapshot in which it was deleted is garbage collected, assuming that older snapshots have also been garbage collected [1].

Iceberg v2 adds data and file sequence numbers to the entry and makes the snapshot ID optional. Values for these fields are inherited from manifest metadata when `null`. That is, if the field is `null` for an entry, then the entry must inherit its value from the manifest file's metadata, stored in the manifest list. The `sequence_number` field represents the data sequence number and must never change after a file is added to the dataset. The data sequence number represents a relative age of the file content and should be used for planning which delete files apply to a data file. The `file_sequence_number` field represents the sequence number of the snapshot that added the file and must also remain unchanged upon assigning at commit. The file sequence number can't be used for pruning delete files as the data within the file may have an older data sequence number. The data and file sequence numbers are inherited only if the entry status is 1 (added). If the entry status is 0 (existing) or 2 (deleted), the entry must include both sequence numbers explicitly.

Notes:

1. Technically, data files can be deleted when the last snapshot that contains the file as “live” data is garbage collected. But this is harder to detect and requires finding the diff of multiple snapshots. It is easier to track what files are deleted in a snapshot and delete them when that snapshot expires. It is not recommended to add a deleted file back to a table. Adding a deleted file can lead to edge cases where incremental deletes can break table snapshots.
2. Manifest list files are required in v2, so that the `sequence_number` and `snapshot_id` to inherit are always available.

DATA FILE FIELDS

The `data_file` struct consists of the following fields:

v1	v2	v3	Field id, name	Type	Description
	required	*required*	**`134 content`**	`int` with meaning: `0: DATA`, `1: POSITION DELETES`, `2: EQUALITY DELETES`	Type of conte stored by the data file: data equality deletes, or position dele (all v1 files a data files)
required	*required*	*required*	**`100 file_path`**	`string`	Full URI for t file with FS scheme
required	*required*	*required*	**`101 file_format`**	`string`	String file format name `avro`, `orc`, `parquet`, or `puffin`

v1	v2	v3	Field id, name	Type	Description
required	*required*	*required*	**`102 partition`**	`struct<...>`	Partition data tuple, schem based on the partition spec output using partition field ids for the st field ids
required	*required*	*required*	**`103 record_count`**	`long`	Number of records in thi file, or the cardinality of deletion vect
required	*required*	*required*	**`104 file_size_in_bytes`**	`long`	Total file size bytes
required			**~~`105 block_size_in_bytes`~~**	`long`	**Deprecated. Always writ default in v1 Do not write v2 or v3.**
optional			**~~`106 file_ordinal`~~**	`int`	**Deprecated. Do not write**

v1	v2	v3	Field id, name	Type	Description
optional			~~`107 sort_columns`~~	`list<112: int>`	**Deprecated. Do not write**
optional	*optional*	*optional*	**`108 column_sizes`**	`map<117: int, 118: long>`	Map from column id to total size on disk of all regions that store the column. Doe not include bytes necessary to read other columns, like footers. Leav null for row-oriented formats (Avr
optional	*optional*	*optional*	**`109 value_counts`**	`map<119: int, 120: long>`	Map from column id to number of values in the column (including nu

v1	v2	v3	Field id, name	Type	Description
					and NaN values)
optional	*optional*	*optional*	`110 null_value_counts`	`map<121: int, 122: long>`	Map from column id to number of nu values in the column
optional	*optional*	*optional*	`137 nan_value_counts`	`map<138: int, 139: long>`	Map from column id to number of N values in the column
optional	*optional*		~~`111 distinct_counts`~~	`map<123: int, 124: long>`	**Deprecated. Do not write**
optional	*optional*	*optional*	`125 lower_bounds`	`map<126: int, 127: binary>`	Map from column id to lower bound the column serialized as binary [1]. Ea value must b less than or equal to all n null, non-Nal

v1	v2	v3	Field id, name	Type	Description
					values in the column for th file [2]
optional	*optional*	*optional*	`128 upper_bounds`	`map<129: int, 130: binary>`	Map from column id to upper bound the column serialized as binary [1]. Ea value must b greater than equal to all n null, non-Nal values in the column for th file [2]
optional	*optional*	*optional*	`131 key_metadata`	`binary`	Implementati specific key metadata for encryption
optional	*optional*	*optional*	`132 split_offsets`	`list<133: long>`	Split offsets f the data file. For example row group offsets in a

v1	v2	v3	Field id, name	Type	Description
					Parquet file. Must be sort ascending
	optional	*optional*	**`135 equality_ids`**	`list<136: int>`	Field ids use to determine row equality equality dele files. Require when `content=2` a should be nu otherwise. Fields with ic listed in this column must present in th delete file
optional	*optional*	*optional*	**`140 sort_order_id`**	`int`	ID represent sort order fo this file [3].
		optional	**`142 first_row_id`**	`long`	The `_row_ic` for the first ro in the data fil See First Ro ID Inheritanc

v1	v2	v3	Field id, name	Type	Description
	optional	*optional*	**`143 referenced_data_file`**	`string`	Fully qualifie location (UR with FS scheme) of a data file that deletes reference [4]
		optional	**`144 content_offset`**	`long`	The offset in file where the content starts [5]
		optional	**`145 content_size_in_bytes`**	`long`	The length o referenced content store in the file; required if `content_offt` is present

The `partition` struct stores the tuple of partition values for each file. Its type is derived from the partition fields of the partition spec used to write the manifest file. In v2, the partition struct's field ids must match the ids from the partition spec.

The column metrics maps are used when filtering to select both data and delete files. For delete files, the metrics must store bounds and counts for all deleted rows, or must be omitted. Storing metrics for deleted rows ensures that the values can be used during job planning to find delete files that must be merged during a scan.

Notes:

1. Single-value serialization for lower and upper bounds is detailed in Appendix D.
2. For `float` and `double`, the value `-0.0` must precede `+0.0`, as in the IEEE 754 `totalOrder` predicate. NaNs are not permitted as lower or upper bounds.
3. If sort order ID is missing or unknown, then the order is assumed to be unsorted. Only data files and equality delete files should be written with a non-null order id. Position deletes are required to be sorted by file and position, not a table order, and should set sort order id to null. Readers must ignore sort order id for position delete files.
4. Position delete metadata can use `referenced_data_file` when all deletes tracked by the entry are in a single data file. Setting the referenced file is required for deletion vectors.
5. The `content_offset` and `content_size_in_bytes` fields are used to reference a specific blob for direct access to a deletion vector. For deletion vectors, these values are required and must exactly match the `offset` and `length` stored in the Puffin footer for the deletion vector blob.
6. The following field ids are reserved on `data_file`: 141.

Bounds for Variant, Geometry, and Geography

For Variant, values in the `lower_bounds` and `upper_bounds` maps store serialized Variant objects that contain lower or upper bounds respectively. The object keys for the bound-variants are normalized JSON path expressions that uniquely identify a field. The object values are primitive Variant representations of the lower or upper bound for that field. Including bounds for any field is optional and upper and lower bounds must have the same Variant type.

Bounds for a field must be accurate for all non-null values of the field in a data file. Bounds for values within arrays must be accurate for all values in the array. Bounds must not be written to describe values with mixed Variant types (other than null). For example, a **measurement** field that contains int64 and null values may have bounds, but if the field also contained a string value such as **n/a** or **0** then the field may not have bounds.

The Variant bounds objects are serialized by concatenating the Variant encoding of the metadata (containing the normalized field paths) and the bounds object. Field paths follow the JSON path format to use normalized path, such as `$['location']['latitude']` or `$['user.name']`. The special path `$` represents bounds for the variant root, indicating that the variant data consists of uniform primitive types, such as strings.

Examples of valid field paths using normalized JSON path format are:

- `$` -- the root of the Variant value
- `$['event_type']` -- the `event_type` field in a Variant object
- `$['user.name']` -- the `"user.name"` field in a Variant object
- `$['location']['latitude']` -- the `latitude` field nested within a `location` object
- `$['tags']` -- the `tags` array
- `$['addresses']['zip']` -- the `zip` field in an `addresses` array that contains objects

For `geometry` and `geography` types, `lower_bounds` and `upper_bounds` are both points of the following coordinates X, Y, Z, and M (see Appendix G) which are the lower / upper bound of all objects in the file.

For `geography` only, xmin (X value of `lower_bounds`) may be greater than xmax (X value of `upper_bounds`), in which case an object in this bounding box may match if it contains an X such that x >= xmin OR x <= xmax. In geographic terminology, the concepts of xmin, xmax, ymin, and ymax are also known as westernmost, easternmost, southernmost and northernmost, respectively. These points are further restricted to the canonical ranges of [-180..180] for X and [-90..90] for Y.

When calculating upper and lower bounds for `geometry` and `geography`, null or NaN values in a coordinate dimension are skipped; for example, POINT (1 NaN) contributes a value to X but no values to Y, Z, or M dimension bounds. If a dimension has only null or NaN values, that dimension is omitted from the bounding box. If either the X or Y dimension is missing then the bounding box itself is not produced.

Sequence Number Inheritance

Manifests track the sequence number when a data or delete file was added to the table.

When adding a new file, its data and file sequence numbers are set to `null` because the snapshot's sequence number is not assigned until the snapshot is successfully committed. When reading, sequence numbers are inherited by replacing `null` with the manifest's sequence number from the manifest list. It is also possible to add a new file with data that logically belongs to an older sequence number. In that case, the data sequence number must be provided explicitly and not inherited. However, the file sequence number must be always assigned when the snapshot is successfully committed.

When writing an existing file to a new manifest or marking an existing file as deleted, the data and file sequence numbers must be non-null and set to the original values that were either inherited or provided at the commit time.

Inheriting sequence numbers through the metadata tree allows writing a new manifest without a known sequence number, so that a manifest can be written once and reused in commit retries. To change a sequence number for a retry, only the manifest list must be rewritten.

When reading v1 manifests with no sequence number column, sequence numbers for all files must default to 0.

First Row ID Inheritance

When adding a new data file, its `first_row_id` field is set to `null` because it is not assigned until the snapshot is successfully committed.

When reading, the `first_row_id` is assigned by replacing `null` with the manifest's `first_row_id` plus the sum of `record_count` for all data files that preceded the file in the manifest that also had a null `first_row_id`.

The inherited value of `first_row_id` must be written into data file metadata when creating existing and deleted entries. The value of `first_row_id` for delete files is always `null`.

Any null (unassigned) `first_row_id` must be assigned via inheritance, even if the data file is existing. This ensures that row IDs are assigned to existing data files in upgraded tables in the first commit after upgrading to v3.

Snapshots

A snapshot consists of the following fields:

v1	v2	v3	Field	Description
required	*required*	*required*	**`snapshot-id`**	A unique long ID
optional	*optional*	*optional*	**`parent-snapshot-id`**	The snapshot ID of the snapshot's parent. Omitted for any snapshot with no parent
	required	*required*	**`sequence-number`**	A monotonically increasing long that tracks the order of changes to a table
required	*required*	*required*	**`timestamp-ms`**	A timestamp when the snapshot was created, used for garbage collection and table inspection

v1	v2	v3	Field	Description
optional	*required*	*required*	**`manifest-list`**	The location of a manifest list for this snapshot that tracks manifest files with additional metadata
optional			**`manifests`**	A list of manifest file locations. Must be omitted if `manifest-list` is present
optional	*required*	*required*	**`summary`**	A string map that summarizes the snapshot changes, including `operation` as a *required* field (see below)
optional	*optional*	*optional*	**`schema-id`**	ID of the table's current schema when the snapshot was created
		required	**`first-row-id`**	The first `_row_id` assigned to the first row in the first data file in the first manifest, see Row Lineage

v1	v2	v3	Field	Description
		required	`added-rows`	The upper bound of the number of rows with assigned row IDs, see Row Lineage
		optional	`key-id`	ID of the encryption key that encrypts the manifest list key metadata

The snapshot summary's `operation` field is used by some operations, like snapshot expiration, to skip processing certain snapshots. Possible `operation` values are:

- `append` -- Only data files were added and no files were removed.
- `replace` -- Data and delete files were added and removed without changing table data; i.e., compaction, changing the data file format, or relocating data files.
- `overwrite` -- Data and delete files were added and removed in a logical overwrite operation.
- `delete` -- Data files were removed and their contents logically deleted and/or delete files were added to delete rows.

For other optional snapshot summary fields, see Appendix F.

Data and delete files for a snapshot can be stored in more than one manifest. This enables:

- Appends can add a new manifest to minimize the amount of data written, instead of adding new records by rewriting and appending to an existing manifest. (This is called a “fast append”.)

- Tables can use multiple partition specs. A table’s partition configuration can evolve if, for example, its data volume changes. Each manifest uses a single partition spec, and queries do not need to change because partition filters are derived from data predicates.
- Large tables can be split across multiple manifests so that implementations can parallelize job planning or reduce the cost of rewriting a manifest.

Manifests for a snapshot are tracked by a manifest list.

Valid snapshots are stored as a list in table metadata. For serialization, see Appendix C.

Snapshot Row IDs

A snapshot's `first-row-id` is assigned to the table's current `next-row-id` on each commit attempt. If a commit is retried, the `first-row-id` must be reassigned based on the table's current `next-row-id`. The `first-row-id` field is required even if a commit does not assign any ID space.

The snapshot's `first-row-id` is the starting `first_row_id` assigned to manifests in the snapshot's manifest list.

The snapshot's `added-rows` captures the upper bound of the number of rows with assigned row IDs. It can be used safely to increment the table's `next-row-id` during a commit. It can be more than the number of rows added in this snapshot and include some existing rows, see Row Lineage Example.

Manifest Lists

Snapshots are embedded in table metadata, but the list of manifests for a snapshot are stored in a separate manifest list file.

A new manifest list is written for each attempt to commit a snapshot because the list of manifests always changes to produce a new snapshot. When a manifest list is written, the (optimistic) sequence number of the snapshot is written for all new manifest files tracked by the list.

A manifest list includes summary metadata that can be used to avoid scanning all of the manifests in a snapshot when planning a table scan. This includes the number of added, existing, and deleted files, and a summary of values for each field of the partition spec used to write the manifest.

A manifest list is a valid Iceberg data file: files must use valid Iceberg formats, schemas, and column projection.

Manifest list files store `manifest_file`, a struct with the following fields:

v1	v2	v3	Field id, name	Type	Description
required	*required*	*required*	**`500 manifest_path`**	`string`	Location of tl manifest file
required	*required*	*required*	**`501 manifest_length`**	`long`	Length of the manifest file bytes
required	*required*	*required*	**`502 partition_spec_id`**	`int`	ID of a partiti spec used to write the manifest; mu be listed in table metada `partition-specs`
	required	*required*	**`517 content`**	`int` with meaning: `0: data`,	The type of f tracked by th manifest, eitl data or delet

v1	v2	v3	Field id, name	Type	Description
				`1: deletes`	files; 0 for all manifests
	required	*required*	**`515 sequence_number`**	`long`	The sequenc number whe the manifest was added t the table; us when readin v1 manifest l
	required	*required*	**`516 min_sequence_number`**	`long`	The minimu data sequen number of al live data or delete files i the manifest use 0 when reading v1 manifest lists
required	*required*	*required*	**`503 added_snapshot_id`**	`long`	ID of the snapshot wh the manifest was added

v1	v2	v3	Field id, name	Type	Description
optional	*required*	*required*	**`504 added_files_count`**	`int`	Number of entries in the manifest that have status `ADDED` (1), when `null` i is assumed t be non-zero
optional	*required*	*required*	**`505 existing_files_count`**	`int`	Number of entries in the manifest that have status `EXISTING` (C when `null` i is assumed t be non-zero
optional	*required*	*required*	**`506 deleted_files_count`**	`int`	Number of entries in the manifest that have status `DELETED` (2) when `null` i is assumed t be non-zero

v1	v2	v3	Field id, name	Type	Description
optional	*required*	*required*	**`512 added_rows_count`**	`long`	Number of ro in all of files i the manifest that have sta `ADDED`, wher `null` this is assumed to l non-zero
optional	*required*	*required*	**`513 existing_rows_count`**	`long`	Number of ro in all of files i the manifest that have sta `EXISTING`, when `null` t is assumed t be non-zero
optional	*required*	*required*	**`514 deleted_rows_count`**	`long`	Number of ro in all of files i the manifest that have sta `DELETED`, wh `null` this is assumed to l non-zero

v1	v2	v3	Field id, name	Type	Description
optional	*optional*	*optional*	**`507 partitions`**	`list<508: field_summary>` (see below)	A list of field summaries fo each partitio field in the spec. Each fi in the list corresponds a field in the manifest file' partition spe
optional	*optional*	*optional*	**`519 key_metadata`**	`binary`	Implementati specific key metadata for encryption
		optional	**`520 first_row_id`**	`long`	The starting `_row_id` to assign to row added by `ADDED` data files First Ro ID Assignme

`field_summary` is a struct with the following fields:

v1	v2	Field id, name	Type	Description
required	*required*	`509 contains_null`	`boolean`	Whether the manifest contains at least one partition with a null value for the field
optional	*optional*	`518 contains_nan`	`boolean`	Whether the manifest contains at least one partition with a NaN value for the field
optional	*optional*	`510 lower_bound`	`bytes` [1]	Lower bound for the non-null, non-NaN values in the partition field, or null if all values are null or NaN [2]
optional	*optional*	`511 upper_bound`	`bytes` [1]	Upper bound for the non-null, non-NaN values in the partition field, or null if all values are null or NaN [2]

Notes:

1. Lower and upper bounds are serialized to bytes using the single-object serialization in Appendix D. The type of used to encode the value is the type of the partition field data.

2. If -0.0 is a value of the partition field, the `lower_bound` must not be +0.0, and if +0.0 is a value of the partition field, the `upper_bound` must not be -0.0.

First Row ID Assignment

The `first_row_id` for existing manifests must be preserved when writing a new manifest list. The value of `first_row_id` for delete manifests is always `null`. The `first_row_id` is only assigned for data manifests that do not have a `first_row_id`. Assignment must account for data files that will be assigned `first_row_id` values when the manifest is read.

The first manifest without a `first_row_id` is assigned a value that is greater than or equal to the `first_row_id` of the snapshot. Subsequent manifests without a `first_row_id` are assigned one based on the previous manifest to be assigned a `first_row_id`. Each assigned `first_row_id` must increase by the row count of all files that will be assigned a `first_row_id` via inheritance in the last assigned manifest. That is, each `first_row_id` must be greater than or equal to the last assigned `first_row_id` plus the total record count of data files with a null `first_row_id` in the last assigned manifest.

A simple and valid approach is to estimate the number of rows in data files that will be assigned a `first_row_id` using the manifest's `added_rows_count` and `existing_rows_count`: `first_row_id = last_assigned.first_row_id + last_assigned.added_rows_count + last_assigned.existing_rows_count`.

Scan Planning

Scans are planned by reading the manifest files for the current snapshot. Deleted entries in data and delete manifests (those marked with status "DELETED") are not used in a scan.

Manifests that contain no matching files, determined using either file counts or partition summaries, may be skipped.

For each manifest, scan predicates, which filter data rows, are converted to partition predicates, which filter partition tuples. These partition predicates are used to select relevant data files, delete files, and deletion vector metadata.

Conversion uses the partition spec that was used to write the manifest file regardless of the current partition spec.

Scan predicates are converted to partition predicates using an *inclusive projection*: if a scan predicate matches a row, then the partition predicate must match that row's partition. This is called *inclusive* [1] because rows that do not match the scan predicate may be included in the scan by the partition predicate.

For example, an `events` table with a timestamp column named `ts` that is partitioned by `ts_day=day(ts)` is queried by users with ranges over the timestamp column: `ts > X`. The inclusive projection is `ts_day >= day(X)`, which is used to select files that may have matching rows. Note that, in most cases, timestamps just before `X` will be included in the scan because the file contains rows that match the predicate and rows that do not match the predicate.

The inclusive projection for an unknown partition transform is *true* because the partition field is ignored and not used in filtering.

Scan predicates are also used to filter data and delete files using column bounds and counts that are stored by field id in manifests. The same filter logic can be used for both data and delete files because both store metrics of the rows either inserted or deleted. If metrics show that a delete file has no rows that match a scan predicate, it may be ignored just as a data file would be ignored [2].

Data files that match the query filter must be read by the scan.

Note that for any snapshot, all file paths marked with "ADDED" or "EXISTING" may appear at most once across all manifest files in the snapshot. If a file path appears more than once, the results of the scan are undefined. Reader implementations may raise an error in this case, but are not required to do so.

Delete files and deletion vector metadata that match the filters must be applied to data files at read time, limited by the following scope rules.

- A deletion vector must be applied to a data file when all of the following are true:
  - The data file's `file_path` is equal to the deletion vector's `referenced_data_file`

    - The data file's data sequence number is *less than or equal to* the deletion vector's data sequence number
    - The data file's partition (both spec and partition values) is equal [4] to the deletion vector's partition
- A *position* delete file must be applied to a data file when all of the following are true:
    - The data file's `file_path` is equal to the delete file's `referenced_data_file` if it is non-null
    - The data file's data sequence number is *less than or equal to* the delete file's data sequence number
    - The data file's partition (both spec and partition values) is equal [4] to the delete file's partition
    - There is no deletion vector that must be applied to the data file (when added, such a vector must contain all deletes from existing position delete files)
- An *equality* delete file must be applied to a data file when all of the following are true:
    - The data file's data sequence number is *strictly less than* the delete's data sequence number
    - The data file's partition (both spec id and partition values) is equal [4] to the delete file's partition *or* the delete file's partition spec is unpartitioned

In general, deletes are applied only to data files that are older and in the same partition, except for two special cases:

- Equality delete files stored with an unpartitioned spec are applied as global deletes. Otherwise, delete files do not apply to files in other partitions.

- Position deletes (vectors and files) must be applied to data files from the same commit, when the data and delete file data sequence numbers are equal. This allows deleting rows that were added in the same commit.

Notes:

1. An alternative, *strict projection*, creates a partition predicate that will match a file if all of the rows in the file must match the scan predicate. These projections are used to calculate the residual predicates for each file in a scan.
2. For example, if `file_a` has rows with `id` between 1 and 10 and a delete file contains rows with `id` between 1 and 4, a scan for `id = 9` may ignore the delete file because none of the deletes can match a row that will be selected.
3. Floating point partition values are considered equal if their IEEE 754 floating-point "single format" bit layout are equal with NaNs normalized to have only the most significant mantissa bit set (the equivalent of calling `Float.floatToIntBits` or `Double.doubleToLongBits` in Java). The Avro specification requires all floating point values to be encoded in this format.
4. Unknown partition transforms do not affect partition equality. Although partition fields with unknown transforms are ignored for filtering, the result of an unknown transform is still used when testing whether partition values are equal.

Snapshot References

Iceberg tables keep track of branches and tags using snapshot references. Tags are labels for individual snapshots. Branches are mutable named references that can be updated by committing a new snapshot as the branch's referenced snapshot using the Commit Conflict Resolution and Retry procedures.

The snapshot reference object records all the information of a reference including snapshot ID, reference type and Snapshot Retention Policy.

v1	v2	Field name	Type	Description
required	*required*	**`snapshot-id`**	`long`	A reference's snapshot ID. The tagged snapshot or latest snapshot of a branch.
required	*required*	**`type`**	`string`	Type of the reference, `tag` or `branch`
optional	*optional*	**`min-snapshots-to-keep`**	`int`	For `branch` type only, a positive number for the minimum number of snapshots to keep in a branch while expiring snapshots. Defaults to table property `history.expire.min-snapshots-to-keep`.
optional	*optional*	**`max-snapshot-age-ms`**	`long`	For `branch` type only, a positive number for the max age of snapshots to keep when expiring, including the latest snapshot. Defaults to table property

v1	v2	Field name	Type	Description
				`history.expire.max-snapshot-age-ms`.
optional	*optional*	**`max-ref-age-ms`**	`long`	For snapshot references except the `main` branch, a positive number for the max age of the snapshot reference to keep while expiring snapshots. Defaults to table property `history.expire.max-ref-age-ms`. The `main` branch never expires.

Valid snapshot references are stored as the values of the `refs` map in table metadata. For serialization, see Appendix C.

Snapshot Retention Policy

Table snapshots expire and are removed from metadata to allow removed or replaced data files to be physically deleted. The snapshot expiration procedure removes snapshots from table metadata and applies the table's retention policy. Retention policy can be configured both globally and on snapshot reference through properties `min-snapshots-to-keep`, `max-snapshot-age-ms` and `max-ref-age-ms`.

When expiring snapshots, retention policies in table and snapshot references are evaluated in the following way:

1. Start with an empty set of snapshots to retain
2. Remove any refs (other than main) where the referenced snapshot is older than `max-ref-age-ms`
3. For each branch and tag, add the referenced snapshot to the retained set
4. For each branch, add its ancestors to the retained set until:
    a. The snapshot is older than `max-snapshot-age-ms`, AND
    b. The snapshot is not one of the first `min-snapshots-to-keep` in the branch (including the branch's referenced snapshot)
5. Expire any snapshot not in the set of snapshots to retain.

Table Metadata

Table metadata is stored as JSON. Each table metadata change creates a new table metadata file that is committed by an atomic operation. This operation is used to ensure that a new version of table metadata replaces the version on which it was based. This produces a linear history of table versions and ensures that concurrent writes are not lost.

The atomic operation used to commit metadata depends on how tables are tracked and is not standardized by this spec. See the sections below for examples.

Table Metadata Fields

Table metadata consists of the following fields:

v1	v2	v3	Field	Description
required	*required*	*required*	`format-version`	An integer version number for the format. Implementations must throw an exception if a

v1	v2	v3	Field	Description
				table's version is higher than the supported version.
optional	*required*	*required*	`table-uuid`	A UUID that identifies the table, generated when the table is created. Implementations must throw an exception if a table's UUID does not match the expected UUID after refreshing metadata.
required	*required*	*required*	`location`	The table's base location. This is used by writers to determine where to store data files, manifest files, and table metadata files.
	required	*required*	`last-sequence-number`	The table's highest assigned sequence number, a monotonically increasing long that tracks the order of snapshots in a table.

v1	v2	v3	Field	Description
required	*required*	*required*	**`last-updated-ms`**	Timestamp in milliseconds from the unix epoch when the table was last updated. Each table metadata file should update this field just before writing.
required	*required*	*required*	**`last-column-id`**	An integer; the highest assigned column ID for the table. This is used to ensure columns are always assigned an unused ID when evolving schemas.
required			**`schema`**	The table's current schema. (**Deprecated**: use `schemas` and `current-schema-id` instead)
optional	*required*	*required*	**`schemas`**	A list of schemas, stored as objects with `schema-id`.
optional	*required*	*required*	**`current-schema-id`**	ID of the table's current schema.

v1	v2	v3	Field	Description
required			**`partition-spec`**	The table's current partition spec, stored as only fields. Note that this is used by writers to partition data, but is not used when reading because reads use the specs stored in manifest files. (**Deprecated**: use `partition-specs` and `default-spec-id` instead)
optional	*required*	*required*	**`partition-specs`**	A list of partition specs, stored as full partition spec objects.
optional	*required*	*required*	**`default-spec-id`**	ID of the "current" spec that writers should use by default.
optional	*required*	*required*	**`last-partition-id`**	An integer; the highest assigned partition field ID across all partition specs for the table. This is used to ensure partition fields are always assigned an

v1	v2	v3	Field	Description
				unused ID when evolving specs.
optional	*optional*	*optional*	**`properties`**	A string to string map of table properties. This is used to control settings that affect reading and writing and is not intended to be used for arbitrary metadata. For example, `commit.retry.num-retries` is used to control the number of commit retries.
optional	*optional*	*optional*	**`current-snapshot-id`**	`long` ID of the current table snapshot; must be the same as the current ID of the `main` branch in `refs`.
optional	*optional*	*optional*	**`snapshots`**	A list of valid snapshots. Valid snapshots are snapshots for which all data files exist in the file system. A data file must not be deleted from the file system until the last

v1	v2	v3	Field	Description
				snapshot in which it was listed is garbage collected.
optional	*optional*	*optional*	`snapshot-log`	A list (optional) of timestamp and snapshot ID pairs that encodes changes to the current snapshot for the table. Each time the current-snapshot-id is changed, a new entry should be added with the last-updated-ms and the new current-snapshot-id. When snapshots are expired from the list of valid snapshots, all entries before a snapshot that has expired should be removed.
optional	*optional*	*optional*	`metadata-log`	A list (optional) of timestamp and metadata file location pairs that encodes changes to the previous metadata files for the

v1	v2	v3	Field	Description
				table. Each time a new metadata file is created, a new entry of the previous metadata file location should be added to the list. Tables can be configured to remove oldest metadata log entries and keep a fixed-size log of the most recent entries after a commit.
optional	*required*	*required*	`sort-orders`	A list of sort orders, stored as full sort order objects.
optional	*required*	*required*	`default-sort-order-id`	Default sort order id of the table. Note that this could be used by writers, but is not used when reading because reads use the specs stored in manifest files.
	optional	*optional*	`refs`	A map of snapshot references. The map keys are the unique snapshot reference

v1	v2	v3	Field	Description
				names in the table, and the map values are snapshot reference objects. There is always a `main` branch reference pointing to the `current-snapshot-id` even if the `refs` map is null.
optional	*optional*	*optional*	**`statistics`**	A list (optional) of table statistics.
optional	*optional*	*optional*	**`partition-statistics`**	A list (optional) of partition statistics.
		required	**`next-row-id`**	A `long` higher than all assigned row IDs; the next snapshot's `first-row-id`. See Row Lineage.
		optional	**`encryption-keys`**	A list (optional) of encryption keys used for table encryption.

For serialization details, see Appendix C.

When a new snapshot is added, the table's `next-row-id` should be increased by the sum of `record_count` for all data files that will be assigned a `first_row_id` via inheritance in the snapshot. The `next-row-id` must always be higher than any assigned row ID in the table.

A simple and valid approach is estimate of the number of rows in data files that will be assigned a `first_row_id` using the manifests' `added_rows_count` and `existing_rows_count`. Using the last assigned manifest, this is `next-row-id = last_assigned.first_row_id + last_assigned.added_rows_count + last_assigned.existing_rows_count`.

Table Statistics

Table statistics files are valid Puffin files. Statistics are informational. A reader can choose to ignore statistics information. Statistics support is not required to read the table correctly. A table can contain many statistics files associated with different table snapshots.

Statistics files metadata within `statistics` table metadata field is a struct with the following fields:

v1	v2	Field name	Type	Description
required	*required*	**`snapshot-id`**	`long`	ID of the Iceberg table's snapshot the statistics file is associated with.
required	*required*	**`statistics-path`**	`string`	Path of the statistics file. See Puffin file format.

v1	v2	Field name	Type	Description
required	*required*	`file-size-in-bytes`	`long`	Size of the statistics file.
required	*required*	`file-footer-size-in-bytes`	`long`	Total size of the statistics file's footer (not the footer payload size). See Puffin file format for footer definition.
optional	*optional*	`key-metadata`	Base64-encoded implementation-specific key metadata for encryption.	
required	*required*	`blob-metadata`	`list<blob metadata>` (see below)	A list of the blob metadata for statistics contained in the file with structure described below.

Blob metadata is a struct with the following fields:

v1	v2	Field name	Type	Description
required	*required*	**`type`**	`string`	Type of the blob. Matches Blob type in the Puffin file.
required	*required*	**`snapshot-id`**	`long`	ID of the Iceberg table's snapshot the blob was computed from.
required	*required*	**`sequence-number`**	`long`	Sequence number of the Iceberg table's snapshot the blob was computed from.
required	*required*	**`fields`**	`list<integer>`	Ordered list of fields, given by field ID, on which the statistic was calculated.
optional	*optional*	**`properties`**	`map<string, string>`	Additional properties associated with the statistic. Subset of Blob properties in the Puffin file.

Partition Statistics

Partition statistics files are based on partition statistics file spec. Partition statistics are not required for reading or planning and readers may ignore them. Each table snapshot may be associated with at most one partition statistics file.

A writer can optionally write the partition statistics file during each write operation, or it can also be computed on demand. Partition statistics file must be registered in the table metadata file to be considered as a valid statistics file for the reader.

`partition-statistics` field of table metadata is an optional list of structs with the following fields:

v1	v2	v3	Field name	Type	Description
required	*required*	*required*	`snapshot-id`	`long`	ID of the Iceberg table's snapshot the partition statistics file is associated with.
required	*required*	*required*	`statistics-path`	`string`	Path of the partition statistics file. See Partition statistics file.
required	*required*	*required*	`file-size-in-bytes`	`long`	Size of the partition statistics file.

PARTITION STATISTICS FILE

Statistics information for each unique partition tuple is stored as a row in any of the data file format of the table (for example, Parquet or ORC). These rows must be sorted (in ascending manner with NULL FIRST) by `partition` field to optimize filtering rows while scanning.

The schema of the partition statistics file is as follows:

v1	v2	v3	Field id, name	Type	Description
required	*required*	*required*	**`1 partition`**	`struct<..>`	Partition data tuple, schema based on the unified partition type considering all specs in a table
required	*required*	*required*	**`2 spec_id`**	`int`	Partition spec id
required	*required*	*required*	**`3 data_record_count`**	`long`	Count of records in data files
required	*required*	*required*	**`4 data_file_count`**	`int`	Count of data files

v1	v2	v3	Field id, name	Type	Description
required	*required*	*required*	**`5 total_data_file_size_in_bytes`**	`long`	Total size of data files in bytes
optional	*optional*	*required*	**`6 position_delete_record_count`**	`long`	Count of position deletes across position delete files and deletion vectors
optional	*optional*	*required*	**`7 position_delete_file_count`**	`int`	Count of position delete files ignoring deletion vectors
		required	**`13 dv_count`**	`int`	Count of deletion vectors
optional	*optional*	*required*	**`8 equality_`**	`long`	Count of records in

v1	v2	v3	Field id, name	Type	Description
			`delete_record_count`		equality delete files
optional	*optional*	*required*	**`9 equality_delete_file_count`**	`int`	Count of equality delete files
optional	*optional*	*optional*	**`10 total_record_count`**	`long`	Accurate count of records in a partition after applying deletes if any
optional	*optional*	*optional*	**`11 last_updated_at`**	`long`	Timestamp in milliseconds from the unix epoch when the partition was last updated

v1	v2	v3	Field id, name	Type	Description
optional	*optional*	*optional*	`12 last_updated_snapshot_id`	`long`	ID of snapshot that last updated this partition

Note that partition data tuple's schema is based on the partition spec output using partition field ids for the struct field ids. The unified partition type is a struct containing all fields that have ever been a part of any spec in the table and sorted by the field ids in ascending order. In other words, the struct fields represent a union of all known partition fields sorted in ascending order by the field ids.

For example,

1. `spec#0` has two fields `{field#1, field#2}` and then the table has evolved into `spec#1` which has three fields `{field#1, field#2, field#3}`. The unified partition type looks like `Struct<field#1, field#2, field#3>`.
2. `spec#0` has two fields `{field#1, field#2}` and then the table has evolved into `spec#1` which has just one field `{field#2}`. The unified partition type looks like `Struct<field#1, field#2>`.

When a v2 table is upgraded to v3 or later, the `position_delete_record_count` field must account for all position deletes, including those from remaining v2 position delete files and any deletion vectors added after the upgrade.

Calculating `total_record_count` for a table with equality deletes or v2 position delete files requires reading data. In such cases, implementations may omit this field and must write `NULL`, indicating that the exact record count in a partition is unknown. If a table has no deletes or only deletion vectors, implementations are encouraged to populate `total_record_count` using metadata in manifests.

Encryption Keys

Keys used for table encryption can be tracked in table metadata as a list named `encryption-keys`. The schema of each key is a struct with the following fields:

v1	v2	v3	Field name	Type.	Description
		required	**`key-id`**	`string`	ID of the encryption key
		required	**`encrypted-key-metadata`**	`string`	Encrypted key and metadata, base64 encoded [1]
		optional	**`encrypted-by-id`**	`string`	Optional ID of the key used to encrypt or wrap `key-metadata`

v1	v2	v3	Field name	Type.	Description
		optional	**`properties`**	`map<string, string>`	A string to string map of additional metadata used by the table's encryption scheme

Notes:

1. The format of encrypted key metadata is determined by the table's encryption scheme and can be a wrapped format specific to the table's KMS provider.

Commit Conflict Resolution and Retry

When two commits happen at the same time and are based on the same version, only one commit will succeed. In most cases, the failed commit can be applied to the new current version of table metadata and retried. Updates verify the conditions under which they can be applied to a new version and retry if those conditions are met.

- Append operations have no requirements and can always be applied.
- Replace operations must verify that the files that will be deleted are still in the table. Examples of replace operations include format changes (replace an Avro file with a Parquet file) and compactions (several files are replaced with a single file that contains the same rows).
- Delete operations must verify that specific files to delete are still in the table. Delete operations based on expressions can always be applied

(e.g., where timestamp < X).

- Table schema updates and partition spec changes must validate that the schema has not changed between the base version and the current version.

File System Tables

*Note: This file system based scheme to commit a metadata file is **deprecated** and will be removed in version 4 of this spec. The scheme is **unsafe** in object stores and local file systems.*

An atomic swap can be implemented using atomic rename in file systems that support it, like HDFS [1].

Each version of table metadata is stored in a metadata folder under the table's base location using a file naming scheme that includes a version number, `V`: `v<V>.metadata.json`. To commit a new metadata version, `V+1`, the writer performs the following steps:

1. Read the current table metadata version `V`.
2. Create new table metadata based on version `V`.
3. Write the new table metadata to a unique file: `<random-uuid>.metadata.json`.
4. Rename the unique file to the well-known file for version `V`: `v<V+1>.metadata.json`.
    a. If the rename succeeds, the commit succeeded and `V+1` is the table's current version
    b. If the rename fails, go back to step 1.

Notes:

1. The file system table scheme is implemented in HadoopTableOperations.

Metastore Tables

The atomic swap needed to commit new versions of table metadata can be implemented by storing a pointer in a metastore or database that is updated with a check-and-put operation [1]. The check-and-put validates that the version of the table that a write is based on is still current and then makes the new metadata from the write the current version.

Each version of table metadata is stored in a metadata folder under the table's base location using a naming scheme that includes a version and UUID: `<V>-<random-uuid>.metadata.json`. To commit a new metadata version, `V+1`, the writer performs the following steps:

1. Create a new table metadata file based on the current metadata.
2. Write the new table metadata to a unique file: `<V+1>-<random-uuid>.metadata.json`.
3. Request that the metastore swap the table's metadata pointer from the location of `V` to the location of `V+1`.
    a. If the swap succeeds, the commit succeeded. `V` was still the latest metadata version and the metadata file for `V+1` is now the current metadata.
    b. If the swap fails, another writer has already created `V+1`. The current writer goes back to step 1.

Notes:

1. The metastore table scheme is partly implemented in BaseMetastoreTableOperations.

Delete Formats

This section details how to encode row-level deletes in Iceberg delete files. Row-level deletes are added by v2 and are not supported in v1. Deletion vectors are added in v3 and are not supported in v2 or earlier. Position delete files must not be added to v3 tables, but existing position delete files are valid.

There are different formats for encoding row-level deletes:

- Deletion vectors (DVs) identify deleted rows within a single referenced data file by position in a bitmap
- Position delete files identify deleted rows by file location and row position (**deprecated** in v3)
- Equality delete files identify deleted rows by the value of one or more columns

Deletion vectors are a binary representation of deletes for a single data file that is more efficient at execution time than position delete files. Unlike equality or position delete files, there can be at most one deletion vector for a given data file in a snapshot. Writers must ensure that there is at most one deletion vector per data file and must merge new deletes with existing vectors or position delete files. When removing a data file, writers must also remove any deletion vector that applies to that data file from delete manifests. Writers are not required to rewrite Puffin files that contain the removed deletion vectors.

Row-level delete files (both equality and position delete files) are valid Iceberg data files: files must use valid Iceberg formats, schemas, and column projection. It is recommended that these delete files are written using the table's default file format.

Row-level delete files and deletion vectors are tracked by manifests. A separate set of manifests is used for delete files and DVs, but the same manifest schema is used for both data and delete manifests. Deletion vectors are tracked individually by file location, offset, and length within the containing file. Deletion vector metadata must include the referenced data file.

Both position and equality delete files allow encoding deleted row values with a delete. This can be used to reconstruct a stream of changes to a table.

Deletion Vectors

Deletion vectors identify deleted rows of a file by encoding deleted positions in a bitmap. A set bit at position P indicates that the row at position P is deleted.

These vectors are stored using the `deletion-vector-v1` blob definition from the Puffin spec.

Deletion vectors support positive 64-bit positions, but are optimized for cases where most positions fit in 32 bits by using a collection of 32-bit Roaring bitmaps. 64-bit positions are divided into a 32-bit "key" using the most significant 4 bytes and a 32-bit sub-position using the least significant 4 bytes. For each key in the set of positions, a 32-bit Roaring bitmap is maintained to store a set of 32-bit sub-positions for that key.

To test whether a certain position is set, its most significant 4 bytes (the key) are used to find a 32-bit bitmap and the least significant 4 bytes (the sub-position) are tested for inclusion in the bitmap. If a bitmap is not found for the key, then it is not set.

Delete manifests track deletion vectors individually by the containing file location (`file_path`), starting offset of the DV blob (`content_offset`), and total length of the blob (`content_size_in_bytes`). Multiple deletion vectors can be stored in the same file. There are no restrictions on the data files that can be referenced by deletion vectors in the same Puffin file.

At most one deletion vector is allowed per data file in a snapshot. If a DV is written for a data file, it must replace all previously written position delete files so that when a DV is present, readers can safely ignore matching position delete files.

Position Delete Files

Position-based delete files identify deleted rows by file and position in one or more data files, and may optionally contain the deleted row.

*Note: Position delete files are **deprecated** in v3. Existing position deletes must be written to delete vectors when updating the position deletes for a data file.*

A data row is deleted if there is an entry in a position delete file for the row's file and position in the data file, starting at 0.

Position-based delete files store `file_position_delete`, a struct with the following fields:

Field id, name	Type	Description
`2147483546` `file_path`	`string`	Full URI of a data file with FS scheme. This must match the `file_path` of the target data file in a manifest entry
`2147483545` `pos`	`long`	Ordinal position of a deleted row in the target data file identified by `file_path`, starting at `0`
`2147483544` `row`	`required struct<...>` [1]	Deleted row values. Omit the column when not storing deleted rows.

1. When present in the delete file, `row` is required because all delete entries must include the row values.

When the deleted row column is present, its schema may be any subset of the table schema and must use field ids matching the table.

To ensure the accuracy of statistics, all delete entries must include row values, or the column must be omitted (this is why the column type is `required`).

The rows in the delete file must be sorted by `file_path` then `pos` to optimize filtering rows while scanning.

- Sorting by `file_path` allows filter pushdown by file in columnar storage formats.
- Sorting by `pos` allows filtering rows while scanning, to avoid keeping deletes in memory.

Equality Delete Files

Equality delete files identify deleted rows in a collection of data files by one or more column values, and may optionally contain additional columns of the deleted row.

Equality delete files store any subset of a table's columns and use the table's field ids. The *delete columns* are the columns of the delete file used to match data rows. Delete columns are identified by id in the delete file metadata column `equality_ids`. The column restrictions for columns used in equality delete files are the same as those for identifier fields with the exception that optional columns and columns nested under optional structs are allowed (if a parent struct column is null it implies the leaf column is null).

A data row is deleted if its values are equal to all delete columns for any row in an equality delete file that applies to the row's data file (see `Scan Planning`).

Each row of the delete file produces one equality predicate that matches any row where the delete columns are equal. Multiple columns can be thought of as an `AND` of equality predicates. A `null` value in a delete column matches a row if the row's value is `null`, equivalent to `col IS NULL`.

For example, a table with the following data:

```
 1: id | 2: category | 3: name
-------|-------------|---------
 1     | marsupial   | Koala
 2     | toy         | Teddy
 3     | NULL        | Grizzly
 4     | NULL        | Polar
```

The delete `id = 3` could be written as either of the following equality delete files:

```
equality_ids=[1]

 1: id
-------
 3
```

```
equality_ids=[1]

 1: id | 2: category | 3: name
-------|-------------|---------
 3     | NULL        | Grizzly
```

The delete `id = 4 AND category IS NULL` could be written as the following equality delete file:

```
equality_ids=[1, 2]

 1: id | 2: category | 3: name
-------|-------------|---------
 4     | NULL        | Polar
```

If a delete column in an equality delete file is later dropped from the table, it must still be used when applying the equality deletes. If a column was added to a table and later used as a delete column in an equality delete file, the column value is read for older data files using normal projection rules (defaults to `null`).

Delete File Stats

Manifests hold the same statistics for delete files and data files. For delete files, the metrics describe the values that were deleted.

Appendix A: Format-specific Requirements

Avro

Data Type Mappings

Values should be stored in Avro using the Avro types and logical type annotations in the table below.

Optional fields, array elements, and map values must be wrapped in an Avro `union` with `null`. This is the only union type allowed in Iceberg data files.

Optional fields without an Iceberg default must set the Avro field default value to null. Fields with a non-null Iceberg default must convert the default to an equivalent Avro default.

Maps with non-string keys must use an array representation with the `map` logical type. The array representation or Avro’s map type may be used for maps with string keys.

Type	Avro type	Notes
`unknown`	`null` or omitted	
`boolean`	`boolean`	
`int`	`int`	
`long`	`long`	
`float`	`float`	
`double`	`double`	
`decimal(P,S)`	`{ "type": "fixed", "size": minBytesRequired(P), "logicalType": "decimal", "precision": P, "scale": S }`	Stored as fixed using the minimum number of bytes for the given precision.
`date`	`{ "type": "int", "logicalType": "date" }`	Stores days from 1970-01-01.
`time`	`{ "type": "long", "logicalType": "time-micros" }`	Stores microseconds from midnight.
`timestamp`	`{ "type": "long", "logicalType":`	Stores microseconds from 1970-01-01 00:00:00.000000. [1]

Type	Avro type	Notes
	`"timestamp-micros",` `"adjust-to-utc": false }`	
`timestamptz`	`{ "type": "long",` `"logicalType":` `"timestamp-micros",` `"adjust-to-utc": true }`	Stores microseconds from 1970-01-01 00:00:00.000000 UTC. [1]
`timestamp_ns`	`{ "type": "long",` `"logicalType":` `"timestamp-nanos",` `"adjust-to-utc": false }`	Stores nanoseconds from 1970-01-01 00:00:00.000000000. [1], [2]
`timestamptz_ns`	`{ "type": "long",` `"logicalType":` `"timestamp-nanos",` `"adjust-to-utc": true }`	Stores nanoseconds from 1970-01-01 00:00:00.000000000 UTC. [1], [2]
`string`	`string`	
`uuid`	`{ "type": "fixed",` `"size": 16,` `"logicalType": "uuid" }`	
`fixed(L)`	`{ "type": "fixed",` `"size": L }`	
`binary`	`bytes`	
`struct`	`record`	

Type	Avro type	Notes
`list`	`array`	
`map`	`array` of key-value records, or `map` when keys are strings (optional).	Array storage must use logical type name `map` and must store elements that are 2-field records. The first field is a non-null key and the second field is the value.
`variant`	`record` with `metadata` and `value` fields. `metadata` and `value` must not be assigned field IDs and the fields are accessed through names.	Shredding is not supported in Avro.
`geometry`	`bytes`	WKB format, see Appendix G
`geography`	`bytes`	WKB format, see Appendix G

Notes:

1. Avro type annotation `adjust-to-utc` is an Iceberg convention; default value is `false` if not present.
2. Avro logical type `timestamp-nanos` is an Iceberg convention; the Avro specification does not define this type.

Field IDs

Iceberg struct, list, and map types identify nested types by ID. When writing data to Avro files, these IDs must be stored in the Avro schema to support ID-based column pruning.

IDs are stored as JSON integers in the following locations:

ID	Avro schema location	Property	Example

Struct field	Record field object	`field-id`	`{ "type": "record",`
			`...`
			` "fields": [`
			`  { "name": "l",`
			`    "type": ["null", "long"],`
			`    "default": null,`
			`    "field-id": 8 }`
			` ] }`

List element	Array schema object	`element-id`	`{ "type": "array",`
			` "items": "int",`
			` "element-id": 9 }`

String map key	Map schema object	`key-id`	`{ "type": "map",`
			` "values": "int",`
			` "key-id": 10,`
			` "value-id": 11 }`

String map value	Map schema object	`value-id`	

Map key, value	Key, value fields in the element record.	`field-id`	`{ "type": "array",`
			` "logicalType": "map",`
			` "items": {`
			`  "type": "record",`
			`  "name":`

ID	Avro schema location	Property	Example
			`"k12_v13",` `"fields": [` `{ "name": "key",` `"type": "int",` `"field-id": 12` `},` `{ "name":` `"value",` `"type":` `"string",` `"field-id": 13` `}` `] } }`

Note that the string map case is for maps where the key type is a string. Using Avro's map type in this case is optional. Maps with string keys may be stored as arrays.

Parquet

Data Type Mappings

Values should be stored in Parquet using the types and logical type annotations in the table below. Column IDs are required to be stored as field IDs on the parquet schema.

Lists must use the 3-level representation.

Type	Parquet physical type	Logical type	Notes
`unknown`	None		Omit from data files

Type	Parquet physical type	Logical type	Notes
`boolean`	`boolean`		
`int`	`int32`		
`long`	`int64`		
`float`	`float`		
`double`	`double`		
`decimal(P,S)`	`P <= 9`: `int32`, `P <= 18`: `int64`, `fixed` otherwise	`DECIMAL(P,S)`	Fixed must use the minimum number of bytes that can store `P`.
`date`	`int32`	`DATE`	Stores days from 1970-01-01.
`time`	`int64`	`TIME_MICROS` with `adjustToUtc=false`	Stores microseconds from midnight.
`timestamp`	`int64`	`TIMESTAMP_MICROS` with `adjustToUtc=false`	Stores microseconds from 1970-01-01 00:00:00.000000.
`timestamptz`	`int64`	`TIMESTAMP_MICROS` with	Stores microseconds from 1970-01-01 00:00:00.000000 UTC.

Type	Parquet physical type	Logical type	Notes
		`adjustToUtc=true`	
`timestamp_ns`	`int64`	`TIMESTAMP_NANOS` with `adjustToUtc=false`	Stores nanoseconds from 1970-01-01 00:00:00.000000000.
`timestamptz_ns`	`int64`	`TIMESTAMP_NANOS` with `adjustToUtc=true`	Stores nanoseconds from 1970-01-01 00:00:00.000000000 UTC.
`string`	`binary`	`UTF8`	Encoding must be UTF-8.
`uuid`	`fixed_len_byte_array[16]`	`UUID`	
`fixed(L)`	`fixed_len_byte_array[L]`		
`binary`	`binary`		
`struct`	`group`		
`list`	`3-level list`	`LIST`	See Parquet docs for 3-level representation.
`map`	`3-level map`	`MAP`	See Parquet docs for 3-level representation.

Type	Parquet physical type	Logical type	Notes
`variant`	`group` with `metadata` and `value` fields. `metadata` and `value` must not be assigned field IDs and the fields are accessed through names.	`VARIANT`	See Parquet docs for Variant encoding and Variant shredding encoding.
`geometry`	`binary`	`GEOMETRY`	WKB format, see Appendix G.
`geography`	`binary`	`GEOGRAPHY`	WKB format, see Appendix G.

When reading an `unknown` column, any corresponding column must be ignored and replaced with `null` values.

ORC

Data Type Mappings

Type	ORC type	ORC type attributes	Notes
`unknown`	None		Omit from data files
`boolean`	`boolean`		
`int`	`int`		ORC `tinyint` and `smallint` would also map

Type	ORC type	ORC type attributes	Notes
			to `int`.
`long`	`long`		
`float`	`float`		
`double`	`double`		
`decimal(P,S)`	`decimal`		
`date`	`date`		
`time`	`long`	`iceberg.long-type`=`TIME`	Stores microseconds from midnight.
`timestamp`	`timestamp`	`iceberg.timestamp-unit`=`MICROS`	Stores microseconds from 2015-01-01 00:00:00.000000. [1], [2]
`timestamptz`	`timestamp_instant`	`iceberg.timestamp-unit`=`MICROS`	Stores microseconds from 2015-01-01 00:00:00.000000 UTC. [1], [2]
`timestamp_ns`	`timestamp`	`iceberg.timestamp-unit`=`NANOS`	Stores nanoseconds from 2015-01-01 00:00:00.000000000. [1]

Type	ORC type	ORC type attributes	Notes
`timestamptz_ns`	`timestamp_instant`	`iceberg.timestamp-unit`=`NANOS`	Stores nanoseconds from 2015-01-01 00:00:00.000000000 UTC. [1]
`string`	`string`		ORC `varchar` and `char` would also map to **`string`**.
`uuid`	`binary`	`iceberg.binary-type`=`UUID`	
`fixed(L)`	`binary`	`iceberg.binary-type`=`FIXED` & `iceberg.length`=`L`	The length would not be checked by the ORC reader and should be checked by the adapter.
`binary`	`binary`		
`struct`	`struct`		
`list`	`array`		
`map`	`map`		
`variant`	`struct` with `metadata` and `value` fields. `metadata` and	`iceberg.struct-type`=`VARIANT`	Shredding is not supported in ORC.

Type	ORC type	ORC type attributes	Notes
	`value` must not be assigned field IDs.		
`geometry`	`binary`	`iceberg.binary-type`=`GEOMETRY`	WKB format, see Appendix G.
`geography`	`binary`	`iceberg.binary-type`=`GEOGRAPHY`	WKB format, see Appendix G.

Notes:

1. ORC's TimestampColumnVector consists of a time field (milliseconds since epoch) and a nanos field (nanoseconds within the second). Hence the milliseconds within the second are reported twice; once in the time field and again in the nanos field. The read adapter should only use milliseconds within the second from one of these fields. The write adapter should also report milliseconds within the second twice; once in the time field and again in the nanos field. ORC writer is expected to correctly consider millis information from one of the fields. More details at https://issues.apache.org/jira/browse/ORC-546
2. ORC `timestamp` and `timestamp_instant` values store nanosecond precision. Iceberg ORC writers for Iceberg types `timestamp` and `timestamptz` **must** truncate nanoseconds to microseconds. `iceberg.timestamp-unit` is assumed to be `MICROS` if not present.

One of the interesting challenges with this is how to map Iceberg's schema evolution (id based) on to ORC's (name based). In theory, we could use Iceberg's column ids as the column and field names, but that would be inconvenient.

The column IDs must be stored in ORC type attributes using the key `iceberg.id`, and `iceberg.required` to store `"true"` if the Iceberg column is required, otherwise it will be optional.

Iceberg would build the desired reader schema with their schema evolution rules and pass that down to the ORC reader, which would then use its schema evolution to map that to the writer's schema. Basically, Iceberg would need to change the names of columns and fields to get the desired mapping.

Iceberg writer	ORC writer	Iceberg reader	ORC reader
`struct<a (1): int, b (2): string>`	`struct<a: int, b: string>`	`struct<a (2): string, c (3): date>`	`struct<b: string, c: date>`
`struct<a (1): struct<b (2): string, c (3): date>>`	`struct<a: struct<b:string, c:date>>`	`struct<aa (1): struct<cc (3): date, bb (2): string>>`	`struct<a: struct<c:date, b:string>>`

Appendix B: 32-bit Hash Requirements

The 32-bit hash implementation is 32-bit Murmur3 hash, x86 variant, seeded with 0.

Primitive type	Hash specification	Test value
`int`	`hashLong(long(v))` [1]	`34` → `2017239379`

Primitive type	Hash specification	Test value
`long`	`hashBytes(littleEndianBytes(v))`	`34L` → `2017239379`
`decimal(P,S)`	`hashBytes(minBigEndian(unscaled(v)))` [2]	`14.20` → `-500754589`
`date`	`hashInt(daysFromUnixEpoch(v))`	`2017-11-16` → `-653330422`
`time`	`hashLong(microsecsFromMidnight(v))`	`22:31:08` → `-662762989`
`timestamp`	`hashLong(microsecsFromUnixEpoch(v))`	`2017-11-16T22:31:08` → `-2047944441` `2017-11-16T22:31:08.000001` → `-1207196810`
`timestamptz`	`hashLong(microsecsFromUnixEpoch(v))`	`2017-11-16T14:31:08-08:00` → `-2047944441` `2017-11-16T14:31:08.000001-08:00` → `-1207196810`
`timestamp_ns`	`hashLong(microsecsFromUnixEpoch(v))` [3]	`2017-11-16T22:31:08` → `-2047944441` `2017-11-16T22:31:08.000001001` → `-1207196810`
`timestamptz_ns`	`hashLong(microsecsFromUnixEpoch(v))` [3]	`2017-11-16T14:31:08-08:00` → `-2047944441`

Primitive type	Hash specification	Test value
		`2017-11-16T14:31:08.000001001-08:00` → `-1207196810`
`string`	`hashBytes(utf8Bytes(v))`	`iceberg` → `1210000089`
`uuid`	`hashBytes(uuidBytes(v))` [4]	`f79c3e09-677c-4bbd-a479-3f349cb785e7` → `1488055340`
`fixed(L)`	`hashBytes(v)`	`00 01 02 03` → `-188683207`
`binary`	`hashBytes(v)`	`00 01 02 03` → `-188683207`

The types below are not currently valid for bucketing, and so are not hashed. However, if that changes and a hash value is needed, the following table shall apply:

Primitive type	Hash specification	Test value
`unknown`	always `null`	
`boolean`	`false: hashInt(0)`, `true: hashInt(1)`	`true` → `1392991556`
`float`	`hashLong(doubleToLongBits(double(v))` [5]	`1.0F` → `-142385009`, `0.0F` → `1669671676`, `-0.0F` → `1669671676`
`double`	`hashLong(doubleToLongBits(v))` [5]	`1.0D` → `-142385009`, `0.0D` → `1669671676`, `-0.0D` → `1669671676`

A 32-bit hash is not defined for `variant` because there are multiple representations for equivalent values.

Notes:

1. Integer and long hash results must be identical for all integer values. This ensures that schema evolution does not change bucket partition values if integer types are promoted.
2. Decimal values are hashed using the minimum number of bytes required to hold the unscaled value as a two's complement big-endian; this representation does not include padding bytes required for storage in a fixed-length array. Hash results are not dependent on decimal scale, which is part of the type, not the data value.
3. Nanosecond timestamps must be converted to microsecond precision before hashing to ensure timestamps have the same hash value.
4. UUIDs are encoded using big endian. The test UUID for the example above is: `f79c3e09-677c-4bbd-a479-3f349cb785e7`. This UUID encoded as a byte array is: `F7 9C 3E 09 67 7C 4B BD A4 79 3F 34 9C B7 85 E7`
5. `doubleToLongBits` must give the IEEE 754 compliant bit representation of the double value. All `NaN` bit patterns must be canonicalized to `0x7ff8000000000000L`. Negative zero (`-0.0`) must be canonicalized to positive zero (`0.0`). Float hash values are the result of hashing the float cast to double to ensure that schema evolution does not change hash values if float types are promoted.

Appendix C: JSON serialization

Schemas

Schemas are serialized as a JSON object with the same fields as a struct in the table below, and the following additional fields:

v1	v2	Field	JSON representation	Example
optional	*required*	**`schema-id`**	`JSON int`	`0`
optional	*optional*	**`identifier-field-ids`**	`JSON list of ints`	`[1, 2]`

Types are serialized according to this table:

Type	JSON representation	Example
`unknown`	`JSON string: "unknown"`	`"unknown"`
`boolean`	`JSON string: "boolean"`	`"boolean"`
`int`	`JSON string: "int"`	`"int"`
`long`	`JSON string: "long"`	`"long"`
`float`	`JSON string: "float"`	`"float"`
`double`	`JSON string: "double"`	`"double"`
`date`	`JSON string: "date"`	`"date"`
`time`	`JSON string: "time"`	`"time"`
`timestamp, microseconds, without zone`	`JSON string: "timestamp"`	`"timestamp"`

Type	JSON representation	Example
`timestamp, microseconds, with zone`	`JSON string: "timestamptz"`	`"timestamptz"`
`timestamp, nanoseconds, without zone`	`JSON string: "timestamp_ns"`	`"timestamp_ns"`
`timestamp, nanoseconds, with zone`	`JSON string: "timestamptz_ns"`	`"timestamptz_ns"`
`string`	`JSON string: "string"`	`"string"`
`uuid`	`JSON string: "uuid"`	`"uuid"`
`fixed(L)`	`JSON string: "fixed[<L>]"`	`"fixed[16]"`
`binary`	`JSON string: "binary"`	`"binary"`
`decimal(P, S)`	`JSON string: "decimal(<P>, <S>)"`	`"decimal(9,2)"`, `"decimal(9, 2)"`
`struct`	`JSON object: {` `"type": "struct",` `"fields": [ {` `"id": <field id int>,` `"name": <name string>,` `"required": <boolean>,` `"type": <type JSON>,`	`{` `"type": "struct",` `"fields": [ {` `"id": 1,` `"name": "id",` `"required": true,` `"type": "uuid",`

Type	JSON representation	Example
	`"doc": <comment string>, "initial-default": <JSON encoding of default value>, "write-default": <JSON encoding of default value> }, ... ] }`	`"initial-default": "0db3e2a8-9d1d-42b9-aa7b-74ebe558dceb", "write-default": "ec5911be-b0a7-458c-8438-c9a3e53cffae" }, { "id": 2, "name": "data", "required": false, "type": { "type": "list", ... } } ] }`
`list`	`JSON object: { "type": "list", "element-id": <id int>, "element-required": <bool> "element": <type JSON> }`	`{ "type": "list", "element-id": 3, "element-required": true, "element": "string" }`
`map`	`JSON object: { "type": "map", "key-id": <key id int>, "key": <type JSON>,`	`{ "type": "map", "key-id": 4, "key": "string",`

Type	JSON representation	Example
	`"value-id": <val id int>,` `"value-required": <bool>` `"value": <type JSON>` `}`	`"value-id": 5,` `"value-required": false,` `"value": "double"` `}`
`variant`	`JSON string: "variant"`	`"variant"`
`geometry(C)`	`JSON string: "geometry(<C>)"`	`"geometry(srid:4326)"`
`geography(C, A)`	`JSON string: "geography(<C>,<E>)"`	`"geography(srid:4326,spherical)"`

Note that default values are serialized using the JSON single-value serialization in Appendix D.

Partition Specs

Partition specs are serialized as a JSON object with the following fields:

Field	JSON representation	Example
`spec-id`	`JSON int`	`0`
`fields`	`JSON list: [` `<partition field JSON>,` `...` `]`	`[ {` `"source-id": 4,` `"field-id": 1000,` `"name": "ts_day",` `"transform": "day"` `}, {`

Field	JSON representation	Example
		`"source-id": 1,` `"field-id": 1001,` `"name": "id_bucket",` `"transform": "bucket[16]"` `} ]`

Each partition field in `fields` is stored as a JSON object with the following properties.

V1	V2	V3	Field	JSON representation	Example
required	required	optional	**`source-id`**	`JSON int`	1
		optional	**`source-ids`**	`JSON list of ints`	`[1,2]`
	required	required	**`field-id`**	`JSON int`	1000
required	required	required	**`name`**	`JSON string`	`id_buck`
required	required	required	**`transform`**	`JSON string`	`bucket[` `]`

Notes:

1. For partition fields with a transform with a single argument, only `source-id` is written. In case of a multi-argument transform, only `source-ids` is written.

Supported partition transforms are listed below.

Transform or Field	JSON representation	Example
`identity`	`JSON string: "identity"`	`"identity"`
`bucket[N]`	`JSON string: "bucket[<N>]"`	`"bucket[16]"`
`truncate[W]`	`JSON string: "truncate[<W>]"`	`"truncate[20]"`
`year`	`JSON string: "year"`	`"year"`
`month`	`JSON string: "month"`	`"month"`
`day`	`JSON string: "day"`	`"day"`
`hour`	`JSON string: "hour"`	`"hour"`

In some cases partition specs are stored using only the field list instead of the object format that includes the spec ID, like the deprecated `partition-spec` field in table metadata. The object format should be used unless otherwise noted in this spec.

The `field-id` property was added for each partition field in v2. In v1, the reference implementation assigned field ids sequentially in each spec starting at 1,000. See Partition Evolution for more details.

Older versions of the reference implementation can read tables with transforms unknown to it, ignoring them. But other implementations may break if they encounter unknown transforms. All v3 readers are required to read tables with unknown transforms, ignoring them. Writers should not write using partition specs that use unknown transforms.

Sort Orders

Sort orders are serialized as a list of JSON object, each of which contains the following fields:

Field	JSON representation	Example
`order-id`	`JSON int`	`1`

`fields`	`JSON list: [`
	`<sort field JSON>,`
	`...`
	`]`
		`[ {`
		`"transform": "identity",`
		`"source-id": 2,`
		`"direction": "asc",`
		`"null-order": "nulls-first"`
		`}, {`
		`"transform": "bucket[4]",`
		`"source-id": 3,`
		`"direction": "desc",`
		`"null-order": "nulls-last"`
		`} ]`

Each sort field in the fields list is stored as an object with the following properties:

V1	V2	V3	Field	JSON representation	Example
required	required	required	`transform`	`JSON string`	`bucket[`
required	required	optional	`source-id`	`JSON int`	1
		optional	`source-ids`	`JSON list of ints`	`[1,2]`
required	required	required	`direction`	`JSON string`	`asc`

V1	V2	V3	Field	JSON representation	Example
required	required	required	**`null-order`**	`JSON string`	`nulls-last`

Notes:

1. For sort fields with a transform with a single argument, only `source-id` is written. In case of a multi-argument transform, only `source-ids` is written.

Older versions of the reference implementation can read tables with transforms unknown to it, ignoring them. But other implementations may break if they encounter unknown transforms. All v3 readers are required to read tables with unknown transforms, ignoring them.

The following table describes the possible values for the some of the field within sort field:

Field	JSON representation	Possible values
`direction`	`JSON string`	`"asc", "desc"`
`null-order`	`JSON string`	`"nulls-first", "nulls-last"`

Table Metadata and Snapshots

Table metadata is serialized as a JSON object according to the following table. Snapshots are not serialized separately. Instead, they are stored in the table metadata JSON.

A metadata JSON file may be compressed with GZIP.

Metadata field	JSON representation	Example
`format-version`	`JSON int`	`1`
`table-uuid`	`JSON string`	`"fb072c92-a02b-11e9-ae9c-1bb7bc9eca94"`
`location`	`JSON string`	`"s3://b/wh/data.db/table"`
`last-updated-ms`	`JSON long`	`1515100955770`
`last-column-id`	`JSON int`	`22`
`schema`	`JSON schema (object)`	`See above, read schemas instead`
`schemas`	`JSON schemas (list of objects)`	`See above`
`current-schema-id`	`JSON int`	`0`
`partition-spec`	`JSON partition fields (list)`	`See above, read partition-specs instead`
`partition-specs`	`JSON partition specs (list of objects)`	`See above`
`default-spec-id`	`JSON int`	`0`

Metadata field	JSON representation	Example
`last-partition-id`	`JSON int`	`1000`
`properties`	`JSON object: {` `"<key>": "<val>",` `...` `}`	`{` `"write.format.default": "avro",` `"commit.retry.num-retries": "4"` `}`
`current-snapshot-id`	`JSON long`	`3051729675574597004`
`snapshots`	`JSON list of objects: [ {` `"snapshot-id": <id>,` `"timestamp-ms": <timestamp-in-ms>,` `"summary": {` `"operation": <operation>,` `... },` `"manifest-list": "<location>",` `"schema-id": "<id>"` `},` `...` `]`	`[ {` `"snapshot-id": 3051729675574597004,` `"timestamp-ms": 1515100955770,` `"summary": {` `"operation": "append"` `},` `"manifest-list": "s3://b/wh/.../s1.avro"` `"schema-id": 0` `} ]`

Metadata field	JSON representation	Example
`snapshot-log`	`JSON list of objects: [` `{` `"snapshot-id": ,` `"timestamp-ms":` `},` `...` `]`	`[ {` `"snapshot-id": 30517296...,` `"timestamp-ms": 1515100...` `} ]`
`metadata-log`	`JSON list of objects: [` `{` `"metadata-file": ,` `"timestamp-ms":` `},` `...` `]`	`[ {` `"metadata-file":` `"s3://bucket/.../v1.json",` `"timestamp-ms": 1515100...` `} ]`
`sort-orders`	`JSON sort orders (list of sort field object)`	`See above`
`default-sort-order-id`	`JSON int`	`0`
`refs`	`JSON map with string key and object value:` `{` `"<name>": {` `"snapshot-id": <id>,` `"type": <type>,` `"max-ref-age-ms":`	`{` `"test": {` `"snapshot-id": 123456789000,` `"type": "tag",` `"max-ref-age-ms": 10000000` `}` `}`

Metadata field	JSON representation	Example
	`<long>,` `...` `}` `...` `}`	
`encryption-keys`	`JSON list of encryption key objects`	`[ {"key-id": "5f819b", "key-metadata": "aWNlYmVyZwo="} ]`

Name Mapping Serialization

Name mapping is serialized as a list of field mapping JSON Objects which are serialized as follows

Field mapping field	JSON representation	Example
`names`	`JSON list of strings`	`["latitude", "lat"]`
`field-id`	`JSON int`	`1`

Field mapping field	JSON representation	Example
`fields`	`JSON field mappings (list of objects)`	`[{ "field-id": 4, "names": ["latitude", "lat"] }, { "field-id": 5, "names": ["longitude", "long"] }]`

Example

```
[ { "field-id": 1, "names": ["id", "record_id"] },
   { "field-id": 2, "names": ["data"] },
   { "field-id": 3, "names": ["location"], "fields": [
       { "field-id": 4, "names": ["latitude", "lat"] },
       { "field-id": 5, "names": ["longitude", "long"] }
     ] } ]
```

Appendix D: Single-value serialization

Binary single-value serialization

This serialization scheme is for storing single values as individual binary values.

Type	Binary serialization
`unknown`	Not supported
`boolean`	`0x00` for false, non-zero byte for true

Type	Binary serialization
`int`	Stored as 4-byte little-endian
`long`	Stored as 8-byte little-endian
`float`	Stored as 4-byte little-endian
`double`	Stored as 8-byte little-endian
`date`	Stores days from the 1970-01-01 in an 4-byte little-endian int
`time`	Stores microseconds from midnight in an 8-byte little-endian long
`timestamp`	Stores microseconds from 1970-01-01 00:00:00.000000 in an 8-byte little-endian long
`timestamptz`	Stores microseconds from 1970-01-01 00:00:00.000000 UTC in an 8-byte little-endian long
`timestamp_ns`	Stores nanoseconds from 1970-01-01 00:00:00.000000000 in an 8-byte little-endian long
`timestamptz_ns`	Stores nanoseconds from 1970-01-01 00:00:00.000000000 UTC in an 8-byte little-endian long
`string`	UTF-8 bytes (without length)
`uuid`	16-byte big-endian value, see example in Appendix B
`fixed(L)`	Binary value

Type	Binary serialization
`binary`	Binary value (without length)
`decimal(P, S)`	Stores unscaled value as two’s-complement big-endian binary, using the minimum number of bytes for the value
`struct`	Not supported
`list`	Not supported
`map`	Not supported
`variant`	Not supported
`geometry`	WKB format, see Appendix G
`geography`	WKB format, see Appendix G

Bound serialization

The binary single-value serialization can be used to store the lower and upper bounds maps of manifest files, except as specified by the following table.

Type	Binary serialization
`geometry`	A single point, encoded as a x:y:z:m concatenation of its 8-byte little-endian IEEE 754 coordinate values. x and y are mandatory. This becomes x:y if z and m are both unset, x:y:z if only m is unset, and x:y:NaN:m if only z is unset.

Type	Binary serialization
`geography`	A single point, encoded as a x:y:z:m concatenation of its 8-byte little-endian IEEE 754 coordinate values. x and y are mandatory. This becomes x:y if z and m are both unset, x:y:z if only m is unset, and x:y:NaN:m if only z is unset.
`variant`	A serialized Variant of encoded v1 metadata concatenated with an encoded variant object. Object keys are normalized JSON paths to identify fields; values are lower or upper bound values.

JSON single-value serialization

Single values are serialized as JSON by type according to the following table:

Type	JSON representation	Example	Description
`boolean`	`JSON boolean`	`true`	
`int`	`JSON int`	`34`	
`long`	`JSON long`	`34`	
`float`	`JSON number`	`1.0`	
`double`	`JSON number`	`1.0`	
`decimal(P, S)`	`JSON string`	`"14.20"`, `"2E+20"`	Stores the string representation of the decimal value, specifically, for values with a positive scale, the number of digits to the right of

Type	JSON representation	Example	Description
			the decimal point is used to indicate scale, for values with a negative scale, the scientific notation is used and the exponent must equal the negated scale
`date`	`JSON string`	`"2017-11-16"`	Stores ISO-8601 standard date
`time`	`JSON string`	`"22:31:08.123456"`	Stores ISO-8601 standard time with microsecond precision
`timestamp`	`JSON string`	`"2017-11-16T22:31:08.123456"`	Stores ISO-8601 standard timestamp with microsecond precision; must not include a zone offset
`timestamptz`	`JSON string`	`"2017-11-16T22:31:08.123456+00:00"`	Stores ISO-8601 standard timestamp with microsecond precision; must include a zone offset and it must be '+00:00'
`timestamp_ns`	`JSON string`	`"2017-11-16T22:31:08.123456789"`	Stores ISO-8601 standard timestamp with nanosecond precision; must not include a zone offset

Type	JSON representation	Example	Description
`timestamptz_ns`	`JSON string`	`"2017-11-16T22:31:08.123456789+00:00"`	Stores ISO-8601 standard timestamp with nanosecond precision; must include a zone offset and it must be '+00:00'
`string`	`JSON string`	`"iceberg"`	
`uuid`	`JSON string`	`"f79c3e09-677c-4bbd-a479-3f349cb785e7"`	Stores the lowercase uuid string
`fixed(L)`	`JSON string`	`"000102ff"`	Stored as a hexadecimal string
`binary`	`JSON string`	`"000102ff"`	Stored as a hexadecimal string
`struct`	`JSON object by field ID`	`{"1": 1, "2": "bar"}`	Stores struct fields using the field ID as the JSON field name; field values are stored using this JSON single-value format
`list`	`JSON array of values`	`[1, 2, 3]`	Stores a JSON array of values that are serialized using this JSON single-value format

Type	JSON representation	Example	Description
`map`	`JSON object of key and value arrays`	`{ "keys": ["a", "b"], "values": [1, 2] }`	Stores arrays of keys and values; individual keys and values are serialized using this JSON single-value format
`geometry`	`JSON string`	`POINT (30 10)`	Stored using WKT representation, see Appendix G
`geography`	`JSON string`	`POINT (30 10)`	Stored using WKT representation, see Appendix G

Appendix E: Format version changes

Version 3

Default values are added to struct fields in v3.

- The `write-default` is a forward-compatible change because it is only used at write time. Old writers will fail because the field is missing.
- Tables with `initial-default` will be read correctly by older readers if `initial-default` is always null for optional fields. Otherwise, old readers will default optional columns with null. Old readers will fail to read required fields which are populated by `initial-default` because that default is not supported.

Types `variant`, `geometry`, `geography`, `unknown`, `timestamp_ns`, and `timestamptz_ns` are added in v3.

All readers are required to read tables with unknown partition transforms, ignoring the unsupported partition fields when filtering.

Writing v3 metadata:

- Partition Field and Sort Field JSON:
  - `source-ids` was added and must be written in the case of a multi-argument transform.
  - `source-id` must be written in the case of single-argument transforms.

Row-level delete changes:

- Deletion vectors are added in v3, stored using the Puffin `deletion-vector-v1` blob type
- Manifests are updated to track deletion vectors:
  - `referenced_data_file` was added and can be used for both deletion vectors (required) and v2 position delete files that contain deletes for only one data file (optional)
  - `content_offset` was added and must match the deletion vector blob's offset in a Puffin file
  - `content_size_in_bytes` was added and must match the deletion vector blob's length in a Puffin file
- Deletion vectors are maintained synchronously: Writers must merge DVs (and older position delete files) to ensure there is at most one DV per data file
  - Readers can safely ignore position delete files if there is a DV for a data file
- Writers are not allowed to add new position delete files to v3 tables
- Existing position delete files are valid in tables that have been upgraded from v2

    - These position delete files must be merged into the DV for a data file when one is created
    - Position delete files that contain deletes for more than one data file need to be kept in table metadata until all deletes are replaced by DVs

Row lineage changes:

- Writers must set the table's `next-row-id` and use the existing `next-row-id` as the `first-row-id` when creating new snapshots
    - When a table is upgraded to v3, `next_row_id` should be initialized to 0
    - When committing a new snapshot `next-row-id` must be incremented by at least the number of newly assigned row ids in the snapshot
    - It is recommended to increment `next-row-id` by the total `added_rows_count` and `existing_rows_count` of all manifests assigned a `first_row_id`
- Writers must assign a `first_row_id` to new data manifests when writing a manifest list
    - When writing a new manifest list each `first_row_id` must be incremented by at least the number of newly assigned row ids in the manifest
    - It is recommended to increment `first_row_id` by a manifest's `added_rows_count` and `existing_rows_count`
- When writing a manifest, new data files must be written with a null `first_row_id` so that the value is assigned at read time based on the manifest's `first_row_id`

- When a manifest has a non-null `first_row_id`, readers must assign a `first_row_id` to any data file that has a missing or null value in that manifest
    - Readers must increment `first_row_id` by the data file's `record_count`
- When writing an existing data file into a new manifest, its `first_row_id` must be written into the manifest
- When a data file has a non-null `first_row_id`, readers must:
    - Replace any null or missing `_row_id` with the data file's `first_row_id` plus the row's `_pos`
    - Replace any null or missing `_last_updated_sequence_number` to the data file's `data_sequence_number`
    - Read any non-null `_row_id` or `_last_updated_sequence_number` without modification
- When a data file has a null `first_row_id`, readers must produce null for `_row_id` and `_last_updated_sequence_number`
- When writing an existing row into a new data file, writers must write `_row_id` and `_last_updated_sequence_number` if they are non-null

Encryption changes:

- Encryption keys are tracked by table metadata `encryption-keys`
- The encryption key used for a snapshot is specified by `key-id`

Version 2

Writing v1 metadata:

- Table metadata field `last-sequence-number` should not be written
- Snapshot field `sequence-number` should not be written

- Manifest list field `sequence-number` should not be written
- Manifest list field `min-sequence-number` should not be written
- Manifest list field `content` must be 0 (data) or omitted
- Manifest entry field `sequence_number` should not be written
- Manifest entry field `file_sequence_number` should not be written
- Data file field `content` must be 0 (data) or omitted

Reading v1 metadata for v2:

- Table metadata field `last-sequence-number` must default to 0
- Snapshot field `sequence-number` must default to 0
- Manifest list field `sequence-number` must default to 0
- Manifest list field `min-sequence-number` must default to 0
- Manifest list field `content` must default to 0 (data)
- Manifest entry field `sequence_number` must default to 0
- Manifest entry field `file_sequence_number` must default to 0
- Data file field `content` must default to 0 (data)

Writing v2 metadata:

- Table metadata JSON:
    - `last-sequence-number` was added and is required; default to 0 when reading v1 metadata
    - `table-uuid` is now required
    - `current-schema-id` is now required
    - `schemas` is now required
    - `partition-specs` is now required

  - `default-spec-id` is now required
  - `last-partition-id` is now required
  - `sort-orders` is now required
  - `default-sort-order-id` is now required
  - `schema` is no longer required and should be omitted; use `schemas` and `current-schema-id` instead
  - `partition-spec` is no longer required and should be omitted; use `partition-specs` and `default-spec-id` instead
- Snapshot JSON:
  - `sequence-number` was added and is required; default to 0 when reading v1 metadata
  - `manifest-list` is now required
  - `manifests` is no longer required and should be omitted; always use `manifest-list` instead
- Manifest list `manifest_file`:
  - `content` was added and is required; 0=data, 1=deletes; default to 0 when reading v1 manifest lists
  - `sequence_number` was added and is required
  - `min_sequence_number` was added and is required
  - `added_files_count` is now required
  - `existing_files_count` is now required
  - `deleted_files_count` is now required
  - `added_rows_count` is now required
  - `existing_rows_count` is now required
  - `deleted_rows_count` is now required

- Manifest key-value metadata:
  - `schema-id` is now required
  - `partition-spec-id` is now required
  - `format-version` is now required
  - `content` was added and is required (must be "data" or "deletes")
- Manifest `manifest_entry`:
  - `snapshot_id` is now optional to support inheritance
  - `sequence_number` was added and is optional, to support inheritance
  - `file_sequence_number` was added and is optional, to support inheritance
- Manifest `data_file`:
  - `content` was added and is required; 0=data, 1=position deletes, 2=equality deletes; default to 0 when reading v1 manifests
  - `equality_ids` was added, to be used for equality deletes only
  - `block_size_in_bytes` was removed (breaks v1 reader compatibility)
  - `file_ordinal` was removed
  - `sort_columns` was removed

Note that these requirements apply when writing data to a v2 table. Tables that are upgraded from v1 may contain metadata that does not follow these requirements. Implementations should remain backward-compatible with v1 metadata requirements.

Appendix F: Implementation Notes

This section covers topics not required by the specification but recommendations for systems implementing the Iceberg specification to help maintain a uniform experience.

Point in Time Reads (Time Travel)

Iceberg supports two types of histories for tables. A history of previous "current snapshots" stored in "snapshot-log" table metadata and parent-child lineage stored in "snapshots". These two histories might indicate different snapshot IDs for a specific timestamp. The discrepancies can be caused by a variety of table operations (e.g. updating the `current-snapshot-id` can be used to set the snapshot of a table to any arbitrary snapshot, which might have a lineage derived from a table branch or no lineage at all).

When processing point in time queries implementations should use "snapshot-log" metadata to lookup the table state at the given point in time. This ensures time-travel queries reflect the state of the table at the provided timestamp. For example a SQL query like `SELECT * FROM prod.db.table TIMESTAMP AS OF '1986-10-26 01:21:00Z';` would find the snapshot of the Iceberg table just prior to '1986-10-26 01:21:00 UTC' in the snapshot logs and use the metadata from that snapshot to perform the scan of the table. If no snapshot exists prior to the timestamp given or "snapshot-log" is not populated (it is an optional field), then systems should raise an informative error message about the missing metadata.

Optional Snapshot Summary Fields

Snapshot summary can include metrics fields to track numeric stats of the snapshot (see Metrics) and operational details (see Other Fields). The value of these fields should be of string type (e.g., `"120"`).

Metrics

Field	Description
`added-data-files`	Number of data files added in the snapshot

Field	Description
`deleted-data-files`	Number of data files deleted in the snapshot
`total-data-files`	Total number of live data files in the snapshot
`added-delete-files`	Number of positional/equality delete files and deletion vectors added in the snapshot
`added-equality-delete-files`	Number of equality delete files added in the snapshot
`removed-equality-delete-files`	Number of equality delete files removed in the snapshot
`added-position-delete-files`	Number of position delete files added in the snapshot
`removed-position-delete-files`	Number of position delete files removed in the snapshot
`added-dvs`	Number of deletion vectors added in the snapshot
`removed-dvs`	Number of deletion vectors removed in the snapshot
`removed-delete-files`	Number of positional/equality delete files and deletion vectors removed in the snapshot
`total-delete-files`	Total number of live positional/equality delete files and deletion vectors in the snapshot
`added-records`	Number of records added in the snapshot

Field	Description
`deleted-records`	Number of records deleted in the snapshot
`total-records`	Total number of records in the snapshot
`added-files-size`	The size of files added in the snapshot
`removed-files-size`	The size of files removed in the snapshot
`total-files-size`	Total size of live files in the snapshot
`added-position-deletes`	Number of position delete records added in the snapshot
`removed-position-deletes`	Number of position delete records removed in the snapshot
`total-position-deletes`	Total number of position delete records in the snapshot
`added-equality-deletes`	Number of equality delete records added in the snapshot
`removed-equality-deletes`	Number of equality delete records removed in the snapshot
`total-equality-deletes`	Total number of equality delete records in the snapshot
`deleted-duplicate-files`	Number of duplicate files deleted (duplicates are files recorded more than once in the manifest)

Field	Description
`changed-partition-count`	Number of partitions with files added or removed in the snapshot
`manifests-created`	Number of manifest files created in the snapshot
`manifests-kept`	Number of manifest files kept in the snapshot
`manifests-replaced`	Number of manifest files replaced in the snapshot
`entries-processed`	Number of manifest entries processed in the snapshot

Other Fields

Field	Example	Description
`wap.id`	"12345678"	The Write-Audit-Publish id of a staged snapshot
`published-wap-id`	"12345678"	The Write-Audit-Publish id of a snapshot already been published
`source-snapshot-id`	"12345678"	The original id of a cherry-picked snapshot
`engine-name`	"spark"	Name of the engine that created the snapshot
`engine-version`	"3.5.4"	Version of the engine that created the snapshot

Assignment of Snapshot IDs and `current-snapshot-id`

Writers should produce positive values for snapshot ids in a manner that minimizes the probability of id collisions and should verify the id does not conflict with existing snapshots. Producing snapshot ids based on timestamps alone is not recommended as it increases the potential for collisions.

The reference Java implementation uses a type 4 uuid and XORs the 4 most significant bytes with the 4 least significant bytes then ANDs with the maximum long value to arrive at a pseudo-random snapshot id with a low probability of collision.

Java writes `-1` for "no current snapshot" with V1 and V2 tables and considers this equivalent to omitted or `null`. This has never been formalized in the spec, but for compatibility, other implementations can accept `-1` as `null`. Java will no longer write `-1` and will use `null` for "no current snapshot" for all tables with a version greater than or equal to V3.

Naming for GZIP compressed Metadata JSON files

Some implementations require that GZIP compressed files have the suffix `.gz.metadata.json` to be read correctly. The Java reference implementation can additionally read GZIP compressed files with the suffix `metadata.json.gz`.

Position Delete Files with Row Data

Although the spec allows for including the deleted row itself (in addition to the path and position of the row in the data file) in v2 position delete files, writing the row is optional and no implementation currently writes it. The ability to write and read the row is supported in the Java implementation but is deprecated in version 1.11.0.

Appendix G: Geospatial Notes

The Geometry and Geography class hierarchy and its Well-known text (WKT) and Well-known binary (WKB) serializations (ISO supporting XY, XYZ, XYM, XYZM) are defined by OpenGIS Implementation Specification for Geographic information – Simple feature access – Part 1: Common architecture, from OGC (Open Geospatial Consortium).

Points are always defined by the coordinates X, Y, Z (optional), and M (optional), in this order. X is the longitude/easting, Y is the latitude/northing, and Z is usually the height, or elevation. M is a fourth optional dimension, for example a linear reference value (e.g., highway milepost value), a timestamp, or some other value as defined by the CRS.

The version of the OGC standard first used here is 1.2.1, but future versions may also be used if the WKB representation remains wire-compatible.

Features

Schema Evolution
Hidden Partitioning
Partition Evolution
Serializable Isolation
Branching and Tagging
Optimistic Concurrency
Advanced Filtering
Compute Engine Integrations
REST Catalog
Multiple language APIs

Get Started

Spark Quickstart
Hive Quickstart
Open Table Spec
Docs
Talks

Community

Support
Mailing Lists
Iceberg Events
Issues
Contribute Guidelines

ASF

Apache Software Foundation
Thanks
Sponsorship
Security
License