Writing Puffin files

Overview

This document describes a design for writing Puffin files with Apache Impala. Puffin file writing is an enabler for supporting Iceberg V3 spec as it introduces deletion vectors, and these vectors are stored as Puffin sidecar files. Also, column statistics could leverage Puffin file writing.

Goals

- A C++ implementation for writing Puffin files with
  - apache-datasketches-theta-v1 blob type
  - deletion-vector-v1 blob type
- COMPUTE STATS implementation for Iceberg tables with Puffin files
- Give seamless support for v3 Iceberg tables on DML statements
  - DELETE, UPDATE, MERGE statements
  - Metadata persistence with Puffin blobs
  - Table maintenance

Non-goals

- Introducing a new blob type that is in line with Impala's current column stat layout

Background

The Puffin format is designed to be a lightweight, random-access container for arbitrary binary blobs.

File Structure and Layout

A Puffin file acts as a linear container. It relies on metadata "blobs" to describe the contents.

Offset	Component	Length	Description
0	**Magic**	4 bytes	Fixed sequence 0x50 0x46 0x41 0x31 (ASCII: PFA1).
4	**Blob 1**	Variable	Raw binary data for the first blob.
...	...	...	...
N	**Blob M**	Variable	Raw binary data for the M-th blob.
...	**Footer**	Variable	JSON metadata describing the blobs.

A critical implementation detail is endianness. While the file footer flags and sizes typically follow the machine's endianness, the Deletion Vector blob wrapper enforces Big-Endian for length and CRC fields to align with legacy Delta Lake formats. The Roaring Bitmap payload itself utilises a portable format that is Little-Endian.

Impala's file-writer

HdfsTableWriter

HdfsTableWriter is an abstract base class that defines the interface for writing rows to HDFS table partition files in various formats (Parquet, Text, etc.). It serves as the core abstraction for all table writers in Impala. Can be reused for HdfsPuffinWriter.

Proposed design

Puffin file writer

Puffin file writer will be a separate Exec component in the backend code, the object mapping will be in line with the Puffin spec:

```
HdfsTableWriter (abstract base in be/src/exec/)
  ↓
PuffinWriter (be/src/exec/puffin/puffin-writer.h)
  ├─ Contains: puffin::File file_
  └─ Methods: inherited methods from HdfsTableWriter, and methods for blob handling

puffin::File (be/src/exec/puffin/file.h)
  └─ Data: FileMetadata file_metadata, std::vector<Blob> blobs

puffin::Blob (be/src/exec/puffin/blob.h)
  ├─ Data: BlobMetadata metadata, uint8_t* data (non-owning pointer)
  └─ Subclasses (for typed access):
    ├─ DeletionVector (be/src/exec/puffin/delete-vector.h)
    └─ ThetaDatasketch (be/src/exec/puffin/theta-datasketch.h)

puffin::BlobMetadata (be/src/exec/puffin/file-metadata.h)
  ├─ Enums: BlobType, CompressionCodec
  └─ Metadata structure (properties, snapshot-id, sequence-number,..)

puffin::FileMetadata (be/src/exec/puffin/file-metadata.h)
  └─ Data: std::vector<std::string> properties
```

COMPUTE STATS extension

COMPUTE STATS Process Flow

1. QUERY PARSING & ANALYSIS

- SQL Parser (sql-parser.cup) parses COMPUTE STATS statement
- Creates ComputeStatsStmt object
- Two variants:
    - COMPUTE STATS <table>
    - COMPUTE INCREMENTAL STATS <table> [PARTITION(...)]

2. STATEMENT ANALYSIS (ComputeStatsStmt)

- Generates child queries:
    - tblStatsQueryStr_: Row count & partition stats
    - colStatsQueryStr_: Column-level statistics

3. QUERY EXECUTION (Frontend.java)

- doCreateExecRequest() creates TExecRequest
- Query planner (Planner.java) generates execution plan
- Distributed execution across cluster nodes
- Results collected as TRowSet objects

4. RESULTS PROCESSING (catalog-op-executor.cc)

ExecComputeStats() receives:

- tbl_stats_data: Row counts per partition
- col_stats_data: Column statistics per partition

SetTableStats():

- Extracts row counts from tbl_stats_data
- Maps to partition keys
- Creates TPartitionStats objects

SetColumnStats() - directly processes column stats

5. CATALOG UPDATE (TAlterTableUpdateStatsParams)

Creates ALTER TABLE request with:

- table_stats: Row counts per partition
- column_stats: Aggregated column statistics
- partition_stats: Per-partition intermediate stats (incremental)
- snapshot_id: For Iceberg tables
- is_incremental: Flag for incremental computation

6. HMS & CATALOG CACHE UPDATE

- Stats written to Hive Metastore (HMS)

For Iceberg tables:

- May read Puffin stats (PuffinStatsLoader.java)
- Merges HMS and Puffin NDV stats
- Uses the most recent stats based on snapshot timestamps

Approach 1: Extend Step 4

Creating an additional method for Step 4 (client-request-state.cc) that will check if the table is capable of handling statistics sidecars, and if it's possible, the query output from the child query will be pushed down to a PuffinWriter in ExecComputeStats method.
This solution injects the Puffin file writing directly after query execution, requiring separate memory management and being outside the query context.

Approach 2: Extend Step 2

Modifying the query generation could yield a plan that has a multi-table sink, and one of the table sinks has a Puffin Writer.
This solution follows the usual query execution flow; the list of written files is available after the query finishes and could be easily passed down to Step 5.
Open question: Is it possible to use a PlanRootSink and TableSink in a multi-sink setup?

Delete vector writing and modification

Current delete file writing design:
https://docs.google.com/document/d/1GuRiJ3jjqkwINsSCKYaWwcfXHzbMrsd3WEMDOB11Xqw/edit?tab=t.0#heading=h.5bmfhbmb4qdk

The query execution workflow is very similar to deletion vectors; the main differentiator is that in the Iceberg V3 spec, in one snapshot, only one deletion vector is allowed for one file, which means that with a series of deletes on the same file, the writer must merge the previous deletion vector with the newly written one: this can be done by providing every data file and deletion vector on partition level to the writed, and then the writer can merge the vectors before writing (old + new).

Storing deletes in a blob

The deletion vector uses a Roaring bitmap for storing the deleted positions, and the spec optimizes for sub-32-bit-sized deletion entries; therefore, a deletion higher in position than 2^32 is keyed to a different bitmap, for example:

DELETE FROM table where id in (100, 50000, 4500000000);

Stored deletes:

Bitmap #0: Marks {100, 50,000}
Bitmap #1: Marks {205,032,804}

The serialization/deserialization is exercised through CRoaring's API in Impala's code; this is in line with the spec of the deletion vectors.