Concurrency in Iceberg

Ryan Blue
Data updates in HDFS-like stores – Feb 2019

NETFLIX

What is Iceberg?

- A new **table format**: how to track data files as a table

- Goals
  - For **users**: tables that work, without unpleasant surprises
  - For **platform**: atomic commits, logical/physical separation, etc.

- A community standard, to replace the Hive table format

Technical Landscape

- Migration to object stores
    - These are not file systems
    - Append-only or not, formats are **immutable**
    - Commits are coarse
- Metadata is big data
    - Metastore must be a distributed system
      . . . when only tracking **partitions**

So that means...

NETFLIX

Nope.

- Agreement on a distributed database isn’t happening
- Efforts have fizzled as a result

Instead?

Avoid needing consensus!

Avoid needing community consensus.

How?

- Use components everyone can agree on (Avro, Parquet, etc.)
- Take advantage of coarse commits: optimistic concurrency
- Perform metastore tasks in drivers to scale with the cluster
- Metastore maintains a top-level pointer to metadata

Commits

- To commit, a writer must:
  - Note the current metadata version - the base version
  - Create a new metadata file (and manifests, and data files…)
  - Atomically swap the base version for the new version

- Atomic swap can be implemented by:
  - Locking a table in Hive Metastore
  - Atomic rename in HDFS
  - Distributed ZK lock

Atomic swap ensures linear history

Commits: Conflict Resolution

- Writers **optimistically** write new versions
  - Assume no other write will conflict
  - On commit failure, retry based on the latest metadata

- To support retry, actions are structured as:
  - **Assumptions** about the current table state
  - **Pending changes** to the current table state

- Changes are safe if the assumptions hold

Append Resolution Example

- Use case: append data to a table
  - New files: file_A.avro

- Append action:
  - **Assumptions**: None
  - **Pending changes**: add file_A.avro

- Never fails validation and retry is fast

Replace Resolution Example

- Use case: safely merge small files
  - Input: file_A.avro, file_B.avro
  - Output: merged.parquet

- Replace action:
  - **Assumptions**: file_A.avro and file_B.avro are still in the table
  - **Pending changes**:
    remove file_A.avro and file_B.avro, add merged.parquet

- If file_A.avro or file_B.avro is deleted, the commit fails

Small API: TableOperations

```
interface TableOperations {

  TableMetadata current();

  TableMetadata refresh();

  void commit(TableMetadata base,
              TableMetadata metadata);
  ...
}
```


NETFLIX

Using TableOperations

```
class TableChange {
  ...
  public void commit() {
    Tasks.retry(4).run(() -> {
      TableMetadata base = ops.refresh();
      TableMetadata changed = applyChanges(base);
      ops.commit(base, changed);
    }
  }
}
```

Benefits

- Small API surface
- Flexible, can implement snapshot isolation
- No need to ignore uncommitted work

Limitations

- Coarse global synchronization sets a speed limit
- Relies on implementations with fast retries
- Long-running operations – like replace – can starve

Upsert

- Not yet supported
- Based on file append with idempotent delta – fast
- Scope of deltas TBD

Thank you

NETFLIX