The Evolution of Netflix's S3 Data Warehouse

Ryan Blue & Dan Weeks
September 2018 - Strata NY

NETFLIX

Overview

- Netflix Architecture
- S3 Data Warehouse
- Iceberg Tables
- What’s Next

Netflix Architecture

September 2018 - Strata NY

NETFLIX

Cloud native data warehouse

Architectural Principles

- Separate Compute and Storage
- Isolate Different Workloads
- Single Source of Truth

Tech Stack

- **S3 as storage layer**
  - Metadata in Hive Metastore
- **EC2 as compute layer**
  - Hadoop + YARN
- **Spark, Presto (and a little Hive and Pig)**

S3 Data Warehouse

September 2018 - Strata NY

NETFLIX

Hadoop file system compatibility with S3

S3 as a File System

HDFS	S3
`create()`	`REST.PUT.OBJECT`
`open()`	`REST.GET.OBJECT`
`listStatus()`	`REST.GET.BUCKET`
`delete()`	`REST.DELETE.OBJECT`
`rename()`	`REST.COPY.OBJECT +` `REST.DELETE.OBJECT`

What about performance?

Performance & Compatibility

- **Performance**
  - Individual operations take longer
  - Some operations do not map cleanly
  - Break contracts to optimize

- **Commit Path**
  - Relies on expensive rename
  - Creates multiple copies with versioning

Optimizing Commits

- **The “batch” pattern**
  - Never delete data as part of a job
  - Always write data to new paths
  - Atomically swap data locations

- **S3 Committer**
  - Use features like multi-part upload
  - Allows for “append” support

What about consistency?

Consistent Listing (s3mper)

- **Overlay a consistent view of metadata**
  - Track file system metadata externally
  - Expire old metadata and rely on S3

- **Check listings against consistent system**
  - Fail or delay until view is consistent
  - Manually resolve collisions

Challenges

- **Maintenance Cost is High**
  - Custom changes per execution engine
  - Never implemented in Presto or Hive
  - Behaviors differ slightly by implementation

- **Platform issues are surfaced to users**
  - Append is not atomic
  - Automatic overwrite
  - Table operations can be inconsistent

Common Threads

- **File System**
  - Works around differences in behaviors
  - Trades correctness for fewer S3 calls
- **s3mper**
  - Works around S3 prefix-listing inconsistency
- **S3 committers and Batch Pattern**
  - Works around lack of atomic changes to file listings
  - Works around lack of cheap rename in S3
  - Needed to avoid using S3 file system for silly operations

Maybe the problem is using S3 as a file system?

Why are we using S3 this way?

Iceberg

September 2018 - Strata NY

NETFLIX

Hive Table Design

- Key idea: **organize data in a directory tree**

```
date=20180513/
 |- hour=18/
 |    |- ...
 |- hour=19/
 |    |- part-000.parquet
 |    |- ...
 |    |- part-031.parquet
 |- hour=20/
 |    |- ...
 |- ...
```

Hive Table Design

- Filter by directories as columns
  `SELECT ... WHERE date = '20180513' AND hour = 19`

```
date=20180513/
 |- hour=18/
 |    |- ...
 |- hour=19/
 |    |- part-000.parquet
 |    |- ...
 |    |- part-031.parquet
 |- hour=20/
 |    |- ...
 |- ...
```

Design Problems

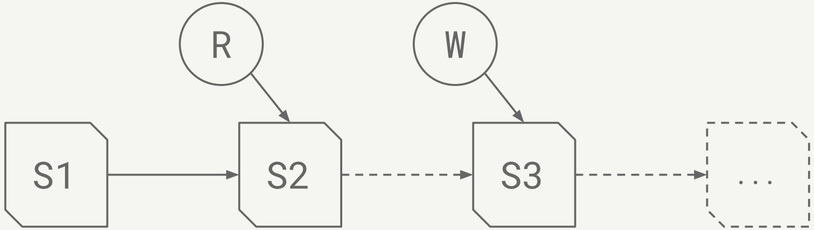
- Table state is stored in two places
  - Partitions in the Hive Metastore
  - Files in a FS with **no transaction support**
- Still requires directory listing to plan jobs
  - **O(n) listing calls, n = # matching partitions**
  - **Eventual consistency breaks correctness**
- Requires elaborate locking for “correctness”
  - **Nothing respects the locking scheme**

Iceberg’s Design

- Key idea: **track all files in a table** over time
  - A **snapshot** is a complete list of files in a table
  - Each write produces and commits a new snapshot

Snapshot Design Benefits

- Snapshot isolation without locking
  - Readers use a current snapshot
  - Writers produce new snapshots in isolation, then commit



- Any change to the file list is an atomic operation
  - Append data across partitions
  - Merge or rewrite files

Design Benefits

- No expensive or eventually-consistent FS operations:
  - No directory or prefix listing
  - No rename: data files written in place

- Reads and writes are isolated and all changes are atomic

- Faster scan planning, distributed across the cluster
  - `O(1)` manifest reads, not `O(n)` partition list calls
  - Upper and lower bounds used to eliminate files

- Reliable CBO metrics

Iceberg replaces s3mper, batch pattern, and S3 committers

Want more specifics?

Come to the **Iceberg talk!**

At **5:25** today in **1E09**

What’s next?

September 2018 - Strata NY

NETFLIX

Today: A narrow paved path

- **New to Hadoop? Big data is great!** Just remember . . .
  - You need to know the physical layout of tables you read
  - Make sure you don’t write too many files – or too few
  - Appends are actually overwrites, except in Presto
  - Don’t write from Presto (but nothing will stop you)
  - You shouldn’t use timestamps or nested types
  - You can’t drop columns in CSV tables
  - And by CSV, we don’t really mean CSV
  - You can’t rename columns in JSON tables
  - If you rename columns in Parquet,
    either Presto or Spark will work, but not both
  - . . .

While we're fixing tables . . .

- **Hidden partitioning**
  - Partition filters derived from data filters
  - No more accidental full table scans

- **Full schema evolution**
  - Supports add, drop, and rename columns

- **Reliable support for types**
  - date, time, timestamp, and decimal
  - struct, list, map, and mixed nesting

Table Layout is Hidden

- Queries are not broken by layout changes
- Physical layout can evolve without painful migration
  - Mistakes can be fixed
  - Prototypes can move to production faster
  - Tables can change as volume grows over time
- **Data Platform can transparently fix table layout**

Snapshot-based Tables

- Any write is atomic – either complete or invisible
  - Rewrite files instead of partitions
  - Tables never have partially committed data

- Simple, built-in change detection
  - Cache and materialized view maintenance
  - Incremental processing

- **Data Platform can monitor and fix data files**
  - Compact small files
  - Repartition to a new layout

Table Format Library

- Common implementation for table operations
  - Write settings are per table, like row group size
  - Read defaults are set in one place, like split combination

- Simple data gathering
  - Log scan predicates and projection to Kafka
  - Recommend optimizations based on actual use

- **Data Platform can automate tuning recommendations**
  - Test file format tuning settings *per table*
  - Update table to affect all writes

Questions?

NETFLIX

Additional Iceberg Slides

September 2018 - Strata NY

NETFLIX

Case Study: Atlas

- Historical Atlas data:
  - Time-series metrics from Netflix runtime systems
  - 1 month: 2.7 million files in 2,688 partitions
  - Problem: cannot process more than a few days of data

- Sample query:

```
select distinct tags['type'] as type
from iceberg.atlas
where
  name = 'metric-name' and
  date > 20180222 and date <= 20180228
order by type;
```

Case Study: Atlas Performance

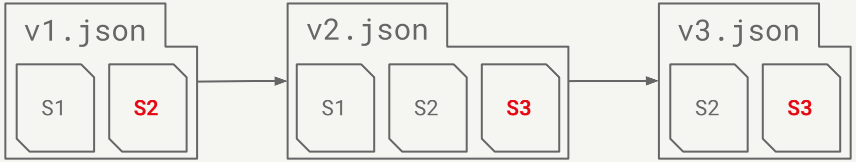
- Hive table – with Parquet filters:
  - 400k+ splits per day, not combined
  - EXPLAIN query: **9.6 min** (planning wall time)

- Iceberg table – partition data filtering:
  - 15,218 splits, combined
  - **13 min** (wall time) / 61.5 hr (task time) / 10 sec (planning)

- Iceberg table – partition and min/max filtering:
  - 412 splits
  - **42 sec** (wall time) / 22 min (task time) / 25 sec (planning)

Iceberg Metadata

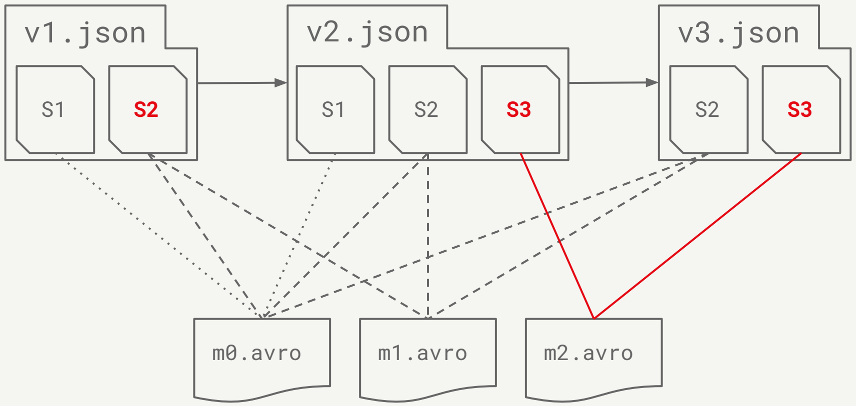
- Implementation of snapshot-based tracking
  - Adds table schema, partition layout, string properties
  - Tracks old snapshots for eventual garbage collection



- Table metadata is immutable and always moves forward
- The current snapshot (pointer) can be rolled back

Manifest Files

- Snapshots are split across one or more **manifest files**
  - Manifests store partition data for each data file
  - Reused to avoid high write volume



Manifest File Contents

- Basic data file info:
  - File location and format
  - Iceberg tracking data
- Values to filter files for a scan:
  - Partition data values
  - Per-column lower and upper bounds
- Metrics for cost-based optimization:
  - File-level: row count, size
  - Column-level: value count, null count, size

Commits

- To commit, a writer must:
  - Note the current metadata version – the base version
  - Create new metadata and manifest files
  - Atomically swap the base version for the new version

- This atomic swap ensures a linear history

- Atomic swap is implemented by:
  - A custom metastore implementation
  - Atomic rename for HDFS or local tables

Commits: Conflict Resolution

- Writers *optimistically* write new versions:
  - Assume that no other writer is operating
  - On conflict, retry based on the latest metadata

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

- Changes are safe if the assumptions are all true

Commits: Resolution Example

- Use case: safely merge small files
  - Merge input: `file1.avro`, `file2.avro`
  - Merge output: `merge1.parquet`

- Rewrite operation:
  - **Assumption**: `file1.avro` and `file2.avro` are still present
  - **Pending changes**:
    Remove `file1.avro` and `file2.avro`
    Add `merge1.parquet`

- Deleting `file1.avro` or `file2.avro` will cause a commit failure