High Throughput Ingestion with Iceberg

Authors: Andrei Ionescu (http://linkedin.com/in/andreiionescu), Shone Sadler (http://linkedin.com/in/shonesadler), Anil Malkani (https://www.linkedin.com/in/anil-malkani-52861a)

(NOTE - Blog 2)

Overview

Customers use Adobe Experience Platform to centralize and standardize their data across the enterprise resulting in a 360 degree view of their data. In our earlier blog “Iceberg at Adobe”, we introduced our scale and consistency challenges and the need to move to apache Iceberg. Adobe Experience Platform is the infrastructure capable of processing exabytes of data ingested through either **Streaming** for low-latency use-cases or **Batch** where processing larger chunks of data efficiently is the concern.

Prior to Iceberg we had a problem with high frequency small files being sent to Adobe Experience Platform for batch ingestion. We suffered from a textbook case known in the big data industry as the "Small File Problem." We immediately saw problems with attempting to commit these small files to Iceberg tables at scale and needed a solution. This blog details that uphill battle focusing on the **buffered writes** ingestion pattern, and how we address the challenge via the key solution of ingesting data.

Streaming Ingestion

Streaming ingestion allows you to send data from client and server-side devices to Experience Platform in near real-time. Platform supports the use of data inlets to stream incoming experience data, which is persisted in streaming-enabled datasets within the Data Lake. Data inlets can be configured to automatically authenticate the data they collect, ensuring that the data is coming from a trusted source.

The key component in charge of streaming ingestion is Siphon Stream. It uses Apache Spark Structured Streaming, a distributed and scalable data processing engine with support for writing multiple kinds of files, and connectors. Apache Spark is the default data processing engine for Adobe Experience Platform, partly due to its rich processing semantics.

The general architecture for streaming ingestion encountered challenges, and improvements are detailed in other articles. The links to the articles are available under the resources section.

Batch Ingestion

Direct ingestion is part of the batch ingestion. The batched files are uploaded into a staging storage zone, processed and committed into the main storage zone, batch by batch.

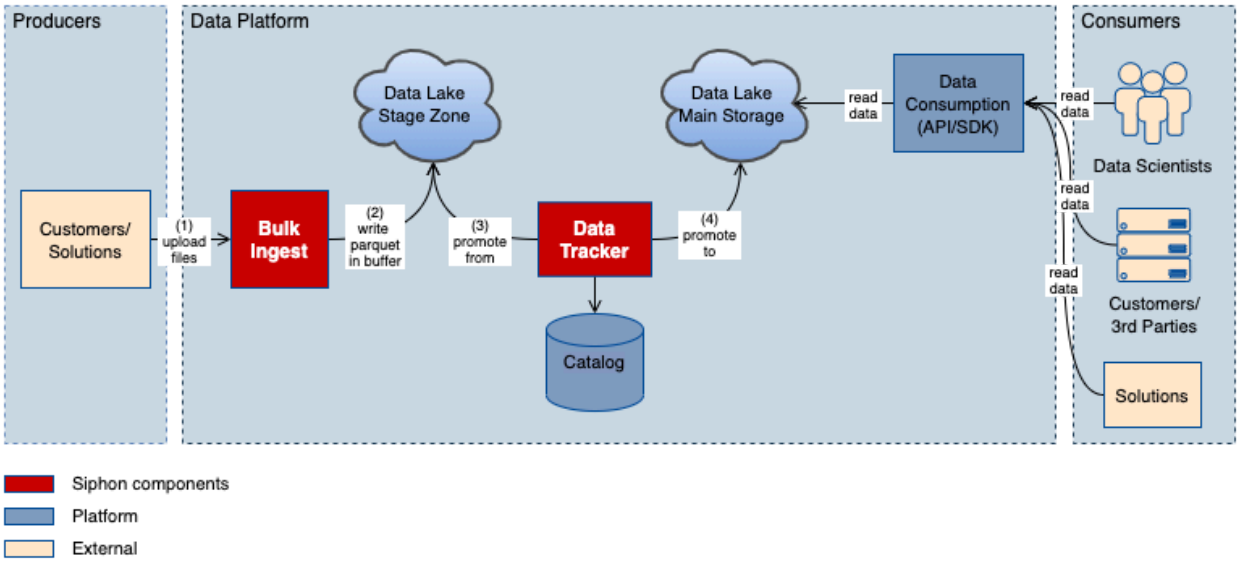

Figure 1. Direct Ingestion Architecture

For customers, this process comes with the advantage of faster data ingestion but with the major disadvantage of sub-optimal file storage leading to issues on the consumption side.

The disadvantage above is in fact the effect of the two technical problems that arise in a high throughput file ingestion environment:

- Small Files Problem
- High concurrent writes leading to race conditions

The Throughput

The high throughput we are discussing is measured in files per day per dataset. We mostly see high frequency small files in TimeSeries DataSets that record clickstream and other event data.

Currently, on any given day, say we process:

- Average number of files ingested is about **700000** over all TimeSeries DataSets.
- Average data size ingested is about **1.5 Tb** over all TimeSeries DataSets.
- Supported throughput per dataset is about **100000 files**.
- Supported throughput per dataset is about **250 Gb**.

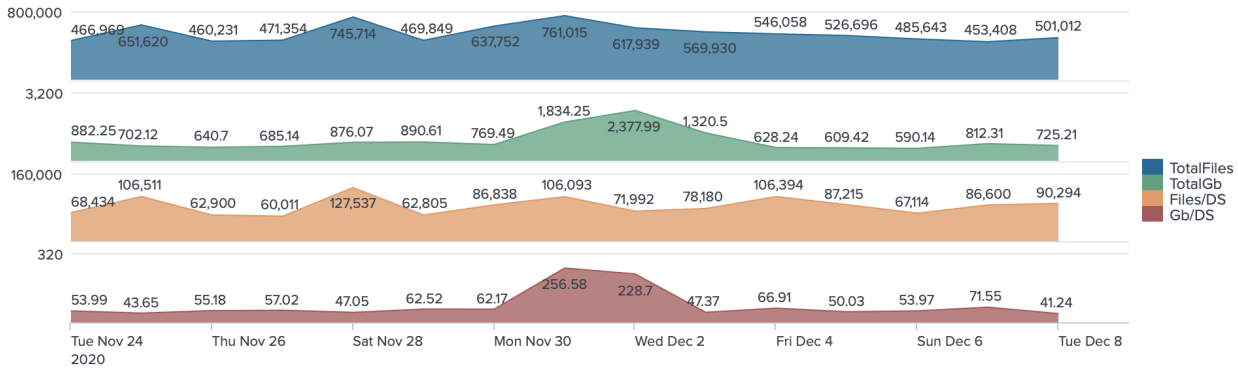

Figure 2. Daily Throughput of Buffered Writes

Small Files Problem

This is a problem already known in distributed storages. For HDFS the issue appears when storing multiple files smaller than block size. HDFS is built to work with large amounts of data stored as big files.

Every file, directory and block in HDFS is represented as an object in the namenode's memory, each of which occupies 150 bytes, as a rule of thumb. So 10 million files, each using a block, would use about 3 gigabytes of memory. Scaling up much beyond this level is a problem with current hardware. Certainly a billion files is not feasible.

Reading through small files normally causes lots of seeks and lots of hopping from datanode to datanode to retrieve each small file, all of which is an inefficient data access pattern.

Iceberg helps on the read side by minimizing file scanning and more accurately locating the files that need to be loaded. When the Iceberg reader is used the data files are pruned with partition and column-level stats, using table metadata.

The above solution helps up to a point but in a high throughput ingestion system such as the Adobe Experience platform we will next run into issues with **high concurrency writes on Iceberg tables**.

High Concurrent Writes on Iceberg Tables

Iceberg uses optimistic concurrency to allow concurrent writes. When conflicts arise, Iceberg automatically retries to ensure updates succeed when compatible. Iceberg may get into accruing a queue of multiple retries and delaying data being ingested in a timely manner.

Iceberg uses snapshots to have isolation between concurrent writes, and these snapshots are created at each commit (data update or change) on the Iceberg table.

Initial tests with Iceberg showed a limit of 15 commits per minute per dataset, whereas we needed to process at least 30 times more per minute per dataset. Each commit starts from a previous snapshot and results in a new one.

This limiting throughput comes from the fact that Iceberg is scanning metadata files of the previous snapshot and composing a new snapshot after the data is written. This process of reading the previous snapshot and then creating a new one takes time, and increases with time as more and more metadata is added.

We needed to solve this limitation in Iceberg and address the throughput expectation of our ingestion mechanism without introducing unpredictable data delays.

Solution

The strategy we took to advance with our Iceberg integration was buffering our writes in Adobe Experience Platform.

Buffered writes is a batch ingestion pattern to address our data needs since it overcomes our main two problems that arise in a high throughput file ingestion environment as Adobe Experience Platform:

- The known Small Files Problem in Hadoop Distributed File System
- High concurrent writes on Iceberg tables

The solution represents a separate service that offers a buffering point, responsible for determining when and how to package and move data from this buffer point to the data lake.

The benefits of this service are:

- **Optimizes the writes** – instead if many writes with small amounts of data we do less writes with larger amounts of data
- **Optimizes the reads** – instead of scanning lots of files (both data and metadata files) readers will have a smaller number of files to open
- **Auto-scaling** – since we use separate on-demand jobs to ingest and optimize data, auto-scaling is inherently available

Architecture

The buffered writes solution for the ingested data from producer to consumer in the Adobe Experience Platform is explained in the figure below.

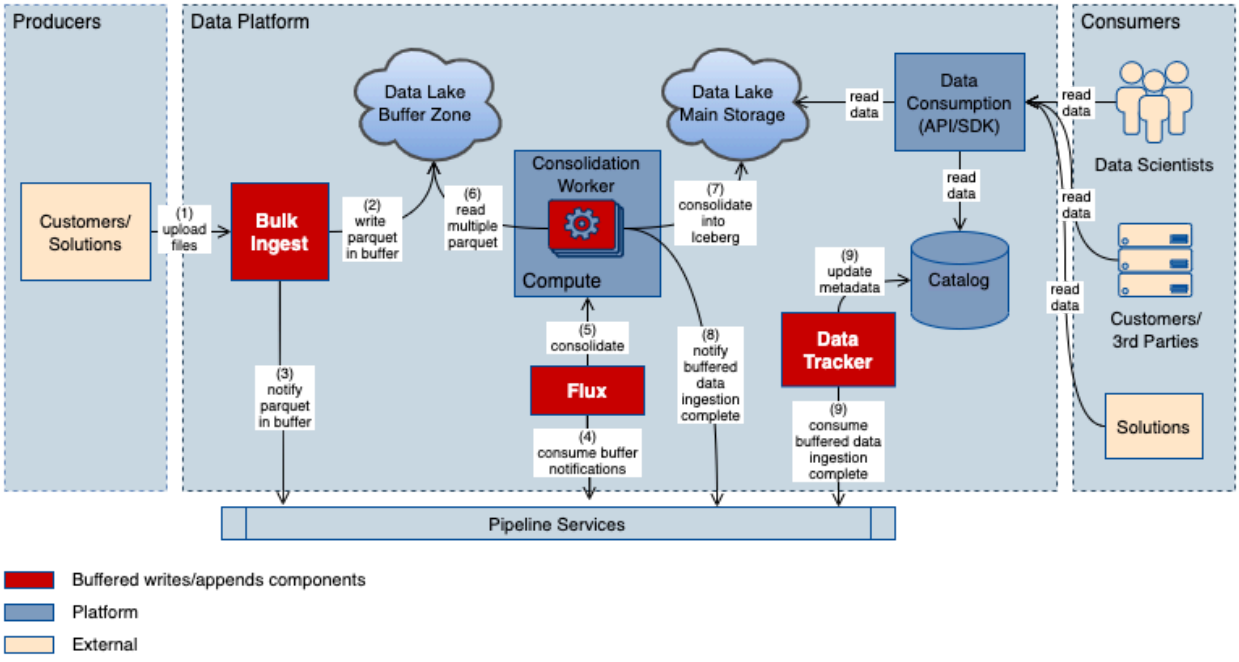

Figure 3. Buffered Writes Architecture

This architecture can be briefly explained in three main stages:

Producers: In this first stage, the customers or external solutions generate data that can be events, CRM data, or dimensional data and usually these are files. These are uploaded or pushed through the Bulk Ingest service.

Data Platform: Here, the data is consumed and written as Parquet files into the buffer zone where it waits for some time. This buffer zone is the first stop of the ingestion process. At this time the Flux component is notified that new data has landed and is read for consolidation. Based on rules and conditions. Flux decides when enough data has accumulated and/or a specific time interval has passed, then it gets the buffered data and writes it efficiently into the main store, in a location where the customer has access to. Data Tracker in turn takes care of generating high level metadata and metrics.

Consumers: In this final stage, customers can pull the consolidated data efficiently using Adobe Experience Platform SDKs and APIs to train their machine learning models, run SQL queries, build dashboards and reports, hydrate the Unified Profile store, run audience segmentations and much more.

The Data Platform components are:

- **Bulk Ingest** - takes care of ingesting data and storing it into buffering storage
- **Data Lake Buffer Zone** - storage location where data is buffered
- **Pipeline Services** - Kafka for Adobe Experience Platform, mainly to manage status

- **Flux** - Spark Stateful Streaming Application containing the logic of buffering the data
- **Compute** - Adobe Experience Platform Compute Service built on top of Azure Databricks which offers Apache Spark support for running both short lived jobs and long running applications
- **Consolidation Worker** - Specialised worker that moves and writes the data from buffer storage into main storage using Iceberg semantics
- **Data Lake Main Storage** - storage location where data is available to customers
- **Data Tracker** - service taking care of higher level metadata
- **Catalog** - high level metadata service for our customers

Data Flow

The files arriving have varying file sizes and can belong to different data partitions. These files are buffered as they arrive and then rearranged to write minimum number of files and generating the least number of commits.

To understand how it works, let's walk through the two scenarios, writes without buffering and write after buffering.

When no buffered writes are applied, any new file arriving will be right away ingested as is without any optimizations.

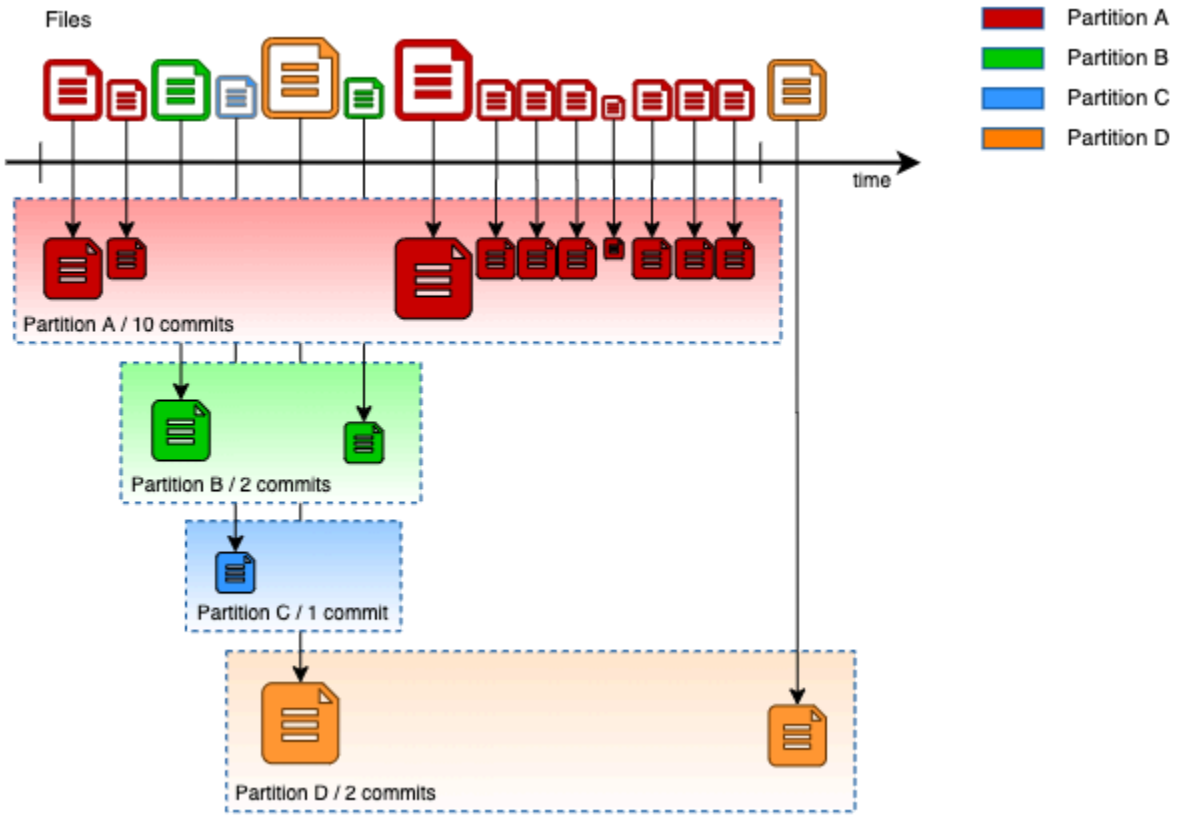

Figure 4. Data Flow without Buffered Writes

This means 15 small files and 15 commits.

When the buffered promotion is used, data is waiting for the proper time to be ingested and optimized to have the least number of files and commits.

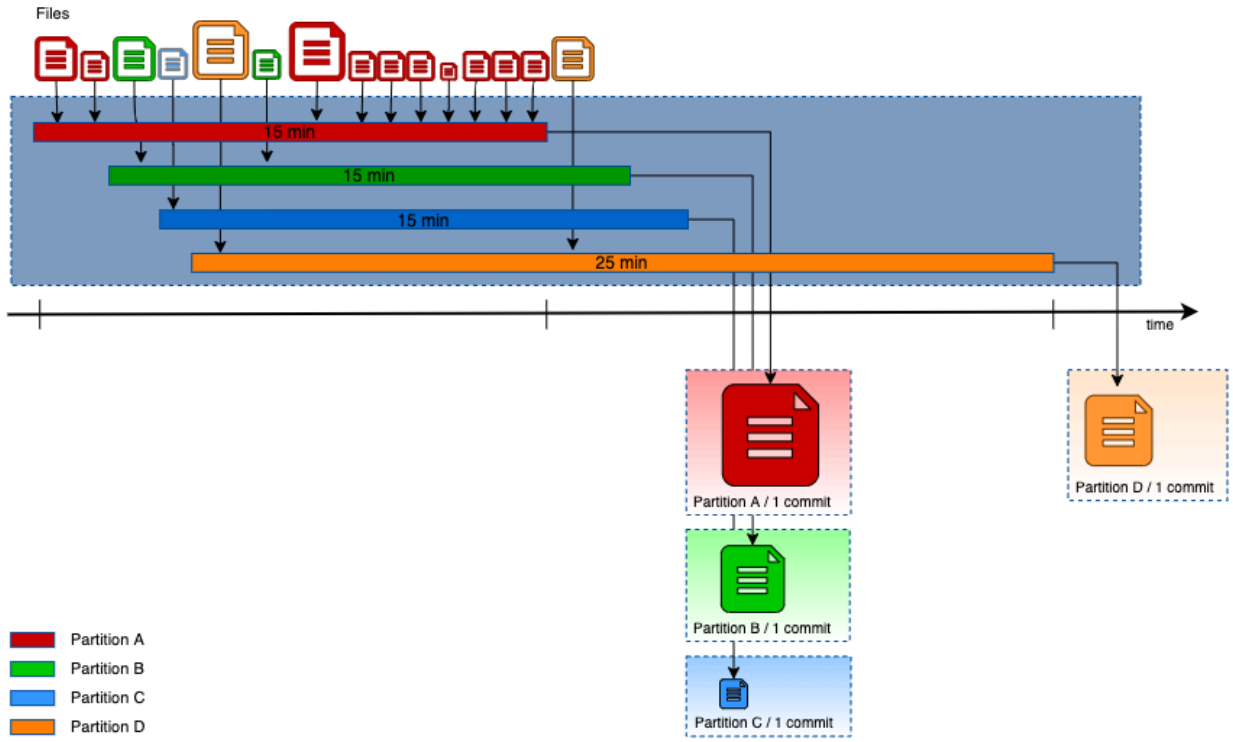

Figure 5. Data Flow with Buffered Writes

With buffered promotion we have 4 commits and 4 files.

Consolidation Worker

One major component of the buffered writes implementation is the "consolidation worker". This is the short lived process that is triggered when enough data is buffered and it must be written in the main storage.

The worker works at tenant level and tries to optimize the data mostly inside partitions but is not limited to this as it accommodates files spanning across multiple partitions too.

The Iceberg Community helped us work through two blockers we had for the Consolidation Worker. First was achieving exactly once guarantees when committing data by using the Write Audit Publish flow (WAP) functionality already present in Apache Iceberg. Second was reducing commit errors due to parallel overwrites of a `version-hint.txt` file. For those interested, details on those discussions are in the resources section below.

Write Audit Publish flow (WAP)

Iceberg has the Write Audit Publish functionality that gives the possibility to store the amount of data as a staged commit later on cherry picking it up and making it available in the table. WAP functionality relies on a specific outside given id - called wapId - by which the staged commit can be later retrieved. The most important aspect is the fact that assures uniqueness of staged commits - ensuring there cannot be two staged commits with the same given ID.

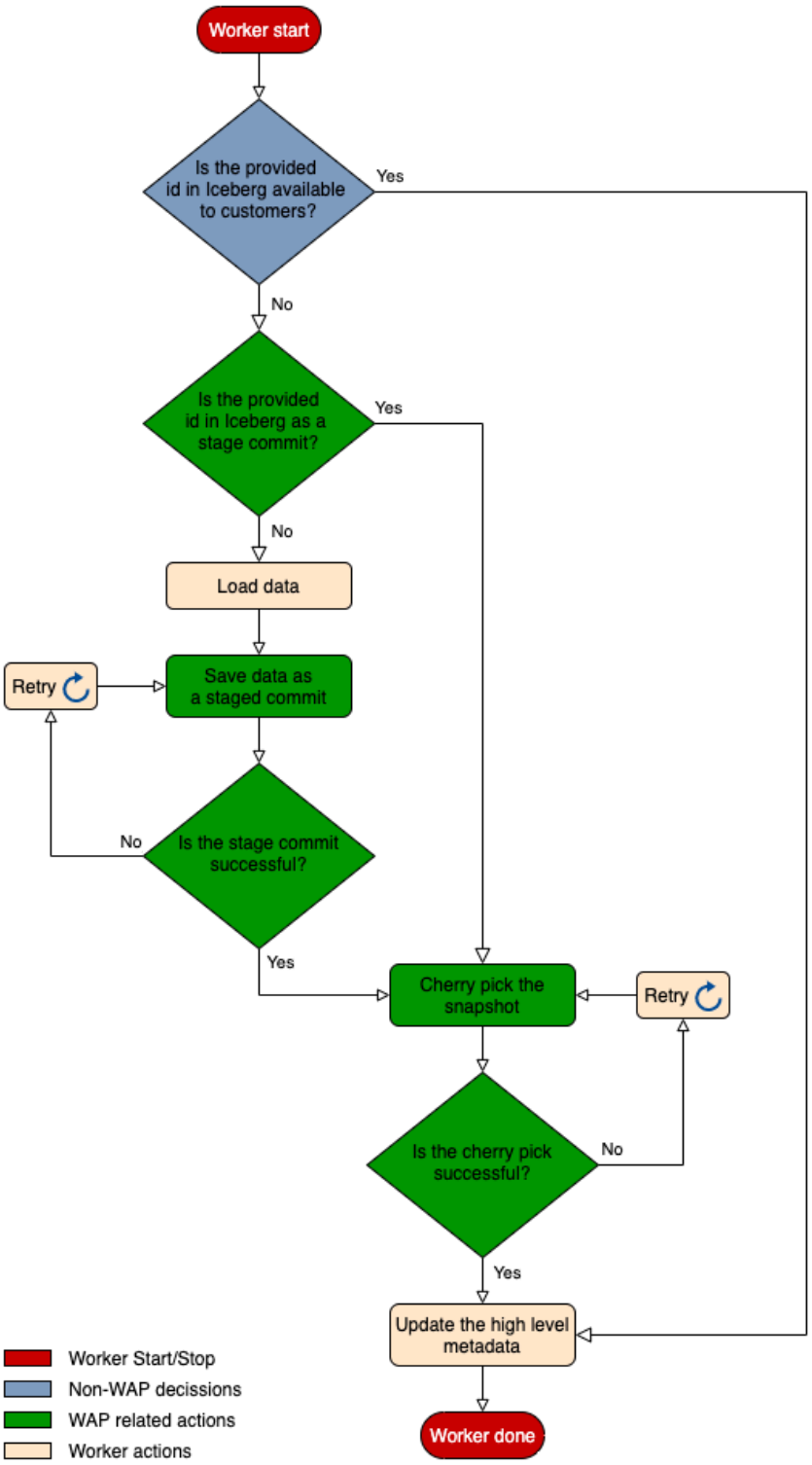

Figure 6. The Write Audit Publish in Consolidation Worker

The WAP workflow is implemented in Consolidation Worker:

- Check if the data is already present in Iceberg by provided id and if so just update the high level metadata
- Check if the data is staged as a separate commit and cherry pick it into the table making it available to customers
- If not present in either of the above cases load the data and write it using the WAP functionality: stage the data with a specific id then cherry pick it.
- At the end update the high level metadata store.

Iceberg's version-hint.txt file improvement23231

An issue that Iceberg has with high throughput writes is properly resolving the latest table version to operate on. In some cases the meta file (`version-hint.txt`) that provides Iceberg's SDK the current version goes missing due to a known non atomic rename/move operation.

When Iceberg commits a new version it will do so by using the HDFS create(overwrite=true) API to replace the current content of version-hint.text with the new version value (an increment of the current one).

Internally ADLS (Azure DataLake Store) chose to implement this API as a DELETE + CREATE operation instead. This creates the possibility for a particular window of time, on each commit operation, where there is no version-hint.text file available.

This raises inconsistency, since both the Iceberg reader and writer depend on the version-hint file to resolve the version and select the appropriate metadata version files to load.

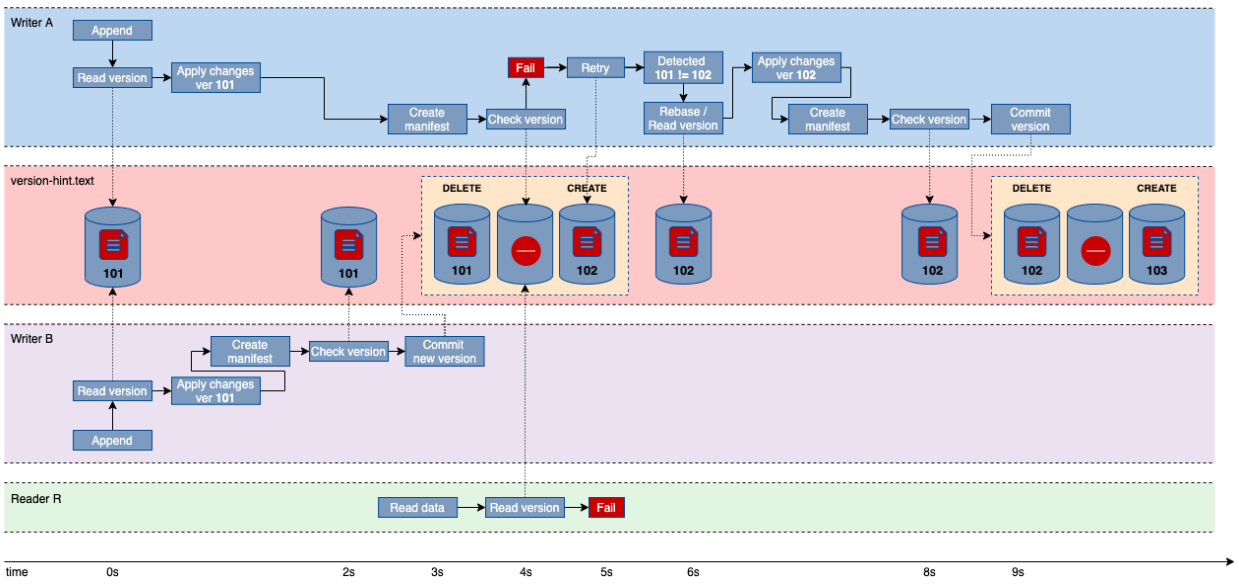

Figure 7. Iceberg's Default Version File Implementation

The solution relies on moving to a different implementation for persisting table version by leveraging the ADLS filesystem APIs guarantees - such as atomicity of CREATE(overwrite=false) and read-after-write consistency of list directory.

Hence the implementation was to switch to preserving version using directory listing instead, so each writer will use CREATE(overwrite=false) to create a new file to signal the new version while readers will have to list the versions directory and pick the highest value present at that particular moment in time.

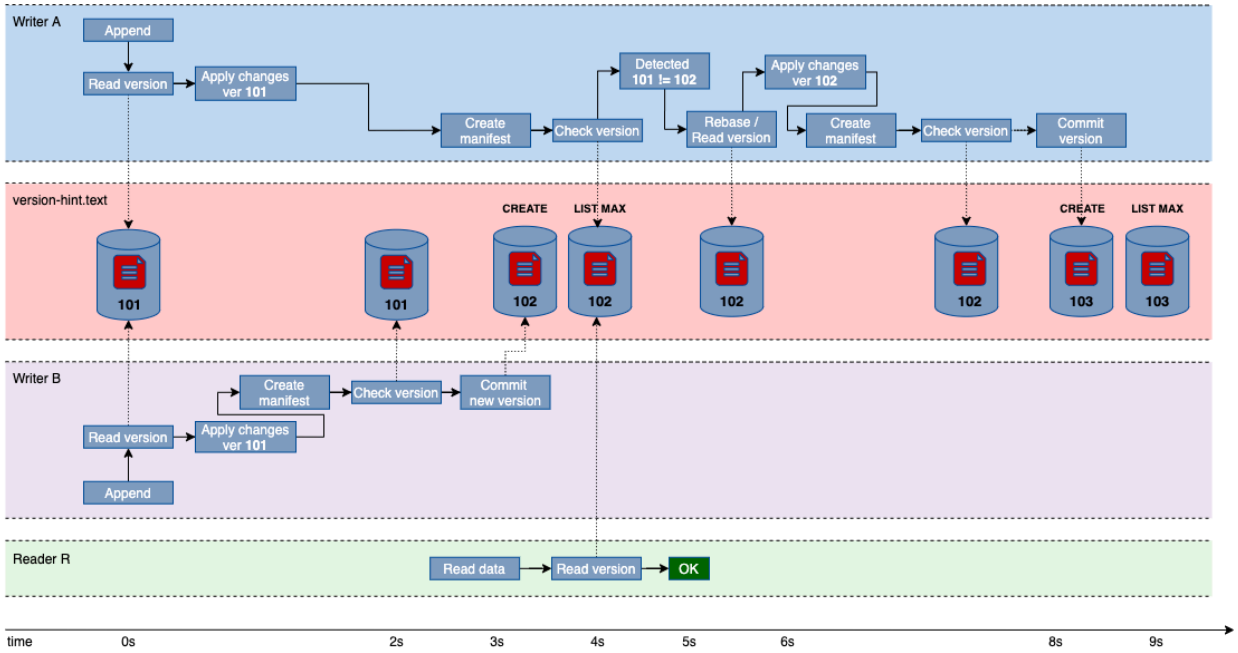

Figure 8. Iceberg version-hint.txt File Improvement

This improvement is Adobe's own way of fixing the issue. While Iceberg Community has been discussing the issue, the chosen way of fixing it is different.

Conclusion

With the Buffered Writes solution we can now go well beyond the targeted throughput goals we had. We have successfully benchmarked ingesting ~200k small files per hour into a single Iceberg Table. Moreover we can horizontally scale to accommodate future data processing needs. The only limitation we have is the available resources on our compute setup. A good problem to have.

In terms of reading the data, we are in a better place now that the data is optimized for reading. Iceberg brings a lot of benefits by itself when reading the data, like vectorized reading, metadata filtering, data filtering, etc. and beside these runtime benefits there is additional tooling available for an even better data optimization. We will talk more about these benefits in our next blog "Taking Query Optimizations to the Next Level with Iceberg".

Resources

1. Adobe Experience Platform

2. Adobe Experience Platform Data Ingestion Help
3. Spark Structured Streaming
4. Redesigning Siphon Stream for Streamlined Data Processing (Part 1)
5. Redesigning Siphon Stream for Streamlined Data Processing in Adobe Experience Platform (Part 2)
6. Creating Adobe Experience Platform Pipeline with Kafka
7. [DISCUSS] Write-audit-publish support
8. Update version-hint.txt atomically (Github ticket discussion)
9. Core: Enhance version-hint.txt recovery with file listing (Github Pull Request)