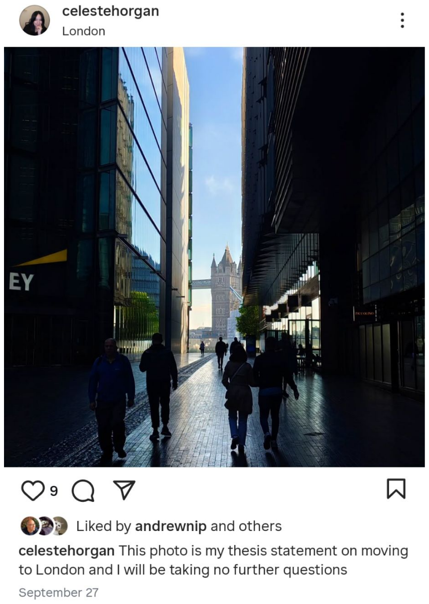
Writes, 3 Ways

Newbie explanations and explorations

Hi! I’m Celeste and I’m new



celestehorgan
London

9

Liked by **andrewnip** and others

celestehorgan This photo is my thesis statement on moving to London and I will be taking no further questions

September 27

… to London

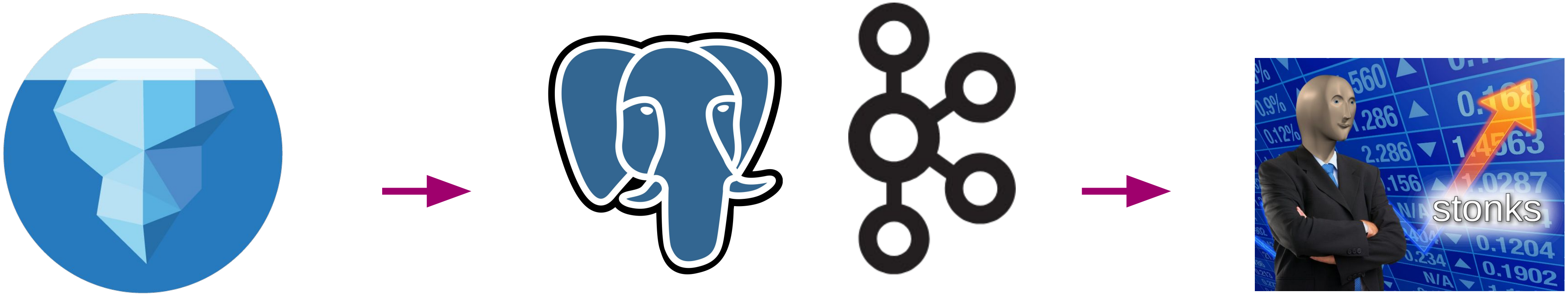
Celeste Horgan · You
OSS Advocate & OSPO stuff @ Snowflake
1mo ·

I'm happy to announce that she's a London girl now! As of today I'm a Senior OSS Developer Advocate at Snowflake! I'm so excited to help run the OSPO here and continue connecting with open source data ...more

… to Snowflake

… to Apache Iceberg™

The way I learn best



Something new

Stuff I already know

Profit

Why Postgres? Kafka? Iceberg?

Mostly because I know Kafka and Postgres well but also:

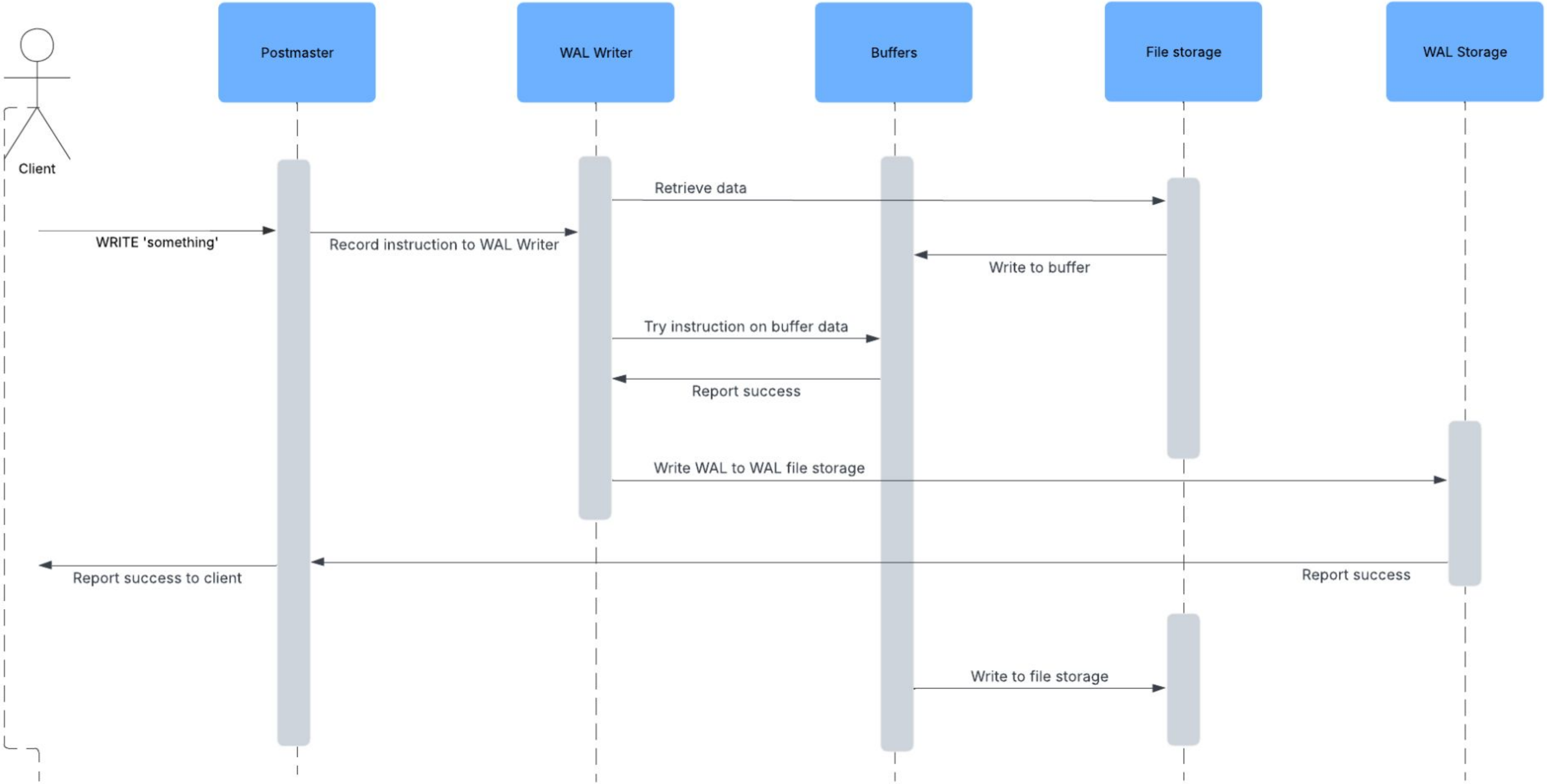
- 1990s
  - **Postgres**: Created at UC Berkeley. Stability and Flexibility are priorities
- Early 2010s
  - **Kafka:** Created at LinkedIn. Handling data in real time and parallel processing are priorities
- Late 2010s
  - **Iceberg:** Created at Netflix. End user (developer) experience is a priority.

With that said…

Here's what we'll do:

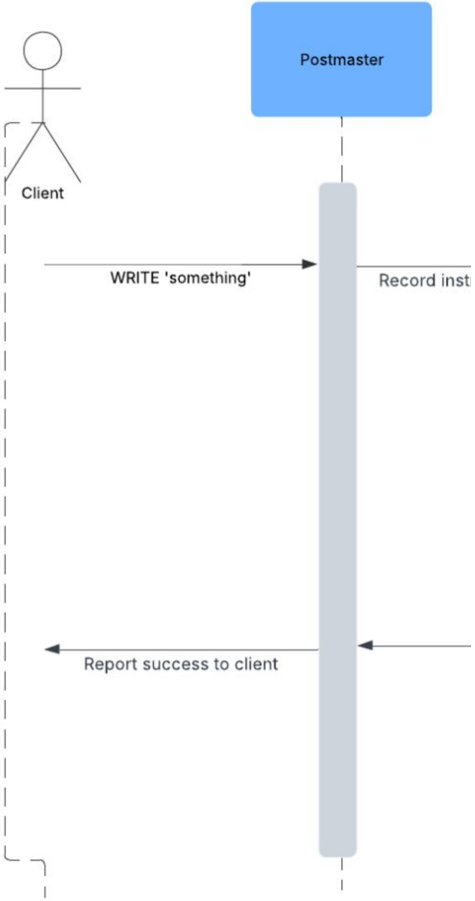
- The lifetime of a write in Postgres
    - & What this says about the system's design
- The lifetime of a write in Apache Kafka®
    - & What this says about the system's design
        - … Compared to Postgres
        - … And how system design trends evolved for Web 2.0
- The lifetime of a write in Apache Iceberg™
    - & What this says about the system's design
        - … Compared to Postgres and Kafka
        - … And how system design trends evolved once more for big data
        - … And how they'll continue to evolve?

Writes in Postgres


Client
Postmaster
WAL Writer
Buffers
File storage
WAL Storage
Retrieve data
WRITE 'something'
Record instruction to WAL Writer
Write to buffer
Try instruction on buffer data
Report success
Write WAL to WAL file storage
Report success to client
Report success
Write to file storage

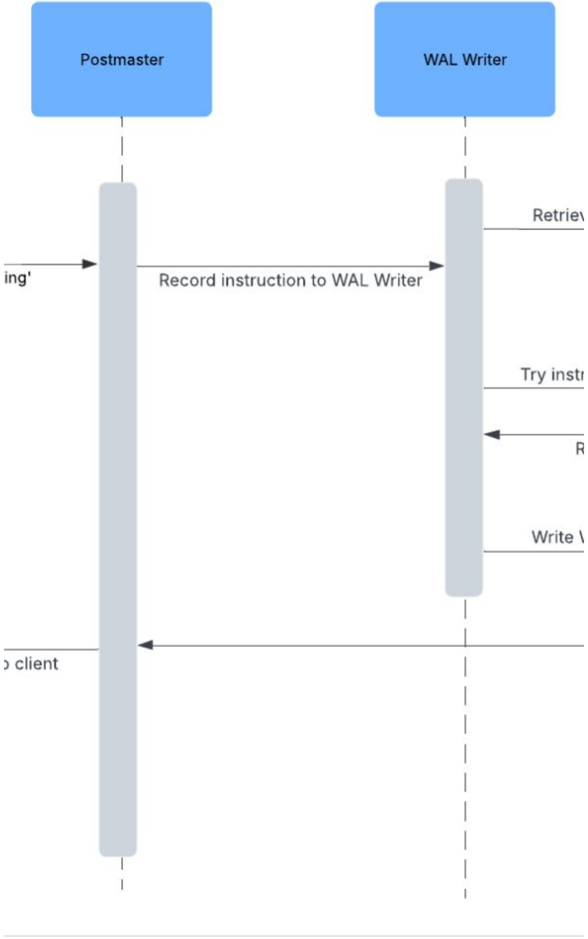
Client → Postmaster

- Clients never connect directly to the database processes, they go through the Postmaster
- Postmaster creates a background process per-connection
  - **Isolation and separation of concerns**



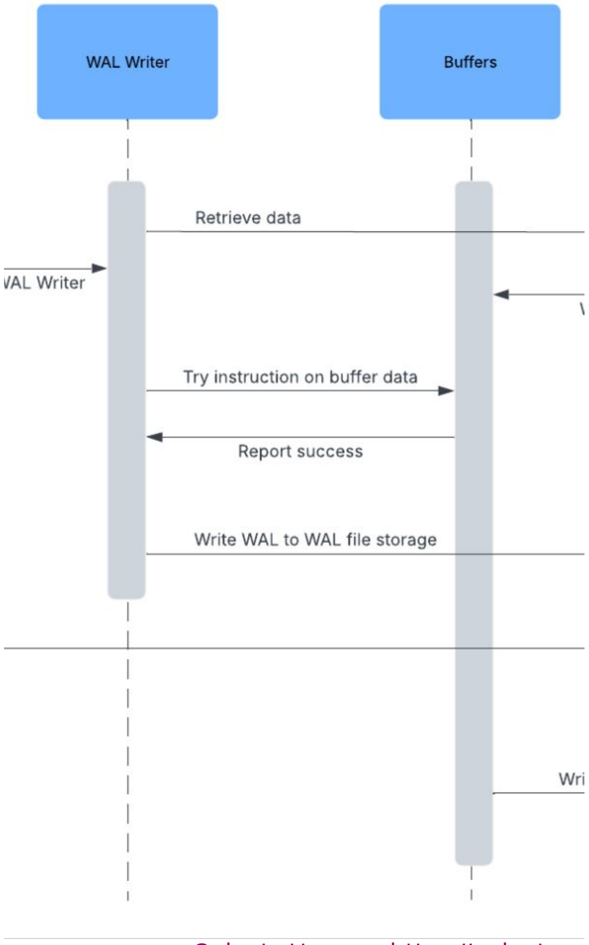
Postmaster → WAL

- The WAL is Postgres' keystone
- All SQL statements (database transactions initiated by the client) are recorded in the Write Ahead Log first
- WAL allows for replay of events, high availability use cases thru replay, rollbacks, etc
    - Because transactions are **atomic**



WAL → Buffer

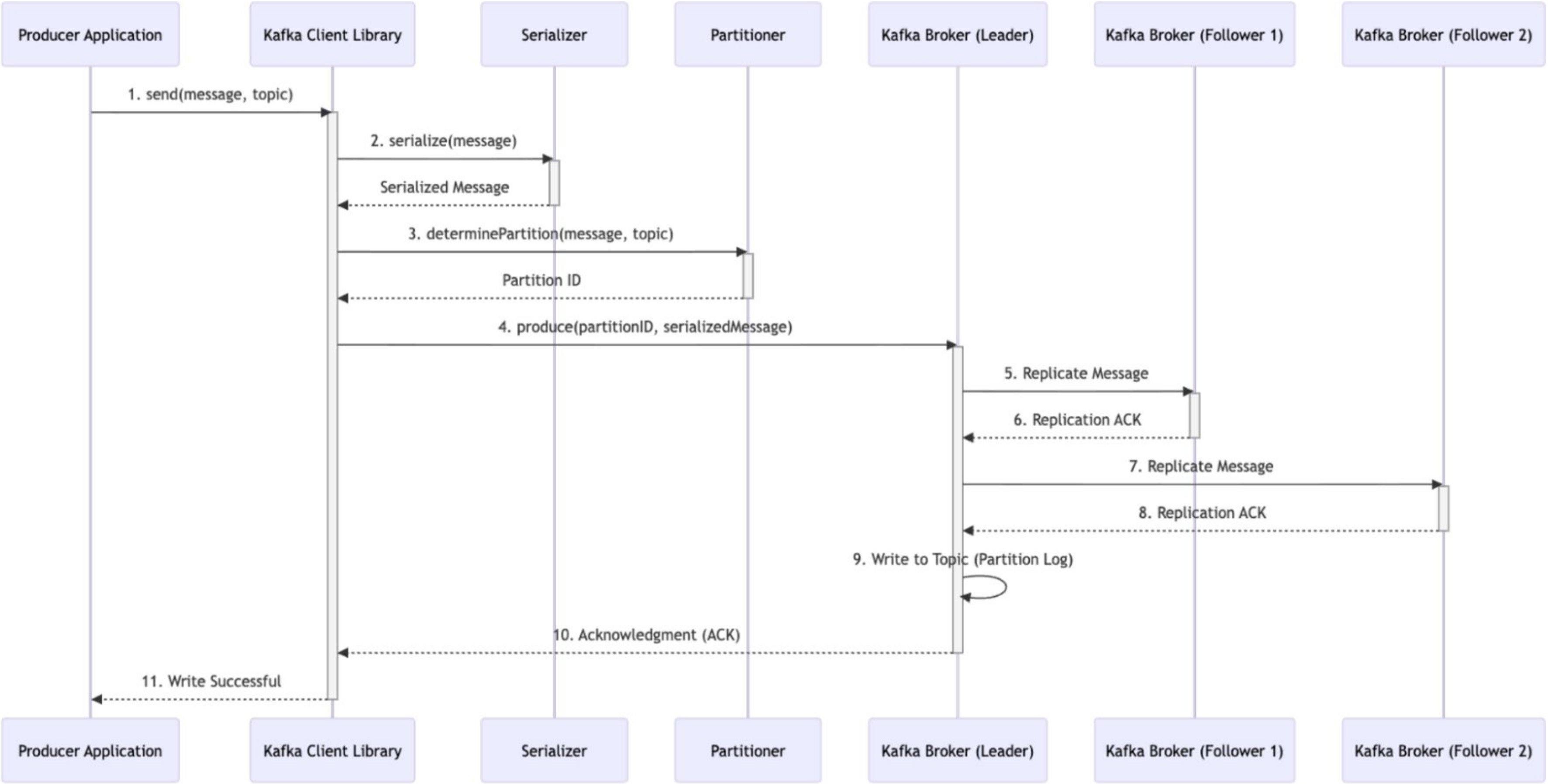
- The WAL records the transaction, then retrieves the affected data and writes it to a buffer
- It then **tries** the transaction on the buffered data
  - If all goes well, Postgres updates the pointers and the buffer becomes the source of truth (we will see this pattern again)



What did we learn?

- Postgres cares about accuracy more than it cares about speed
  - Writes are linear to the WAL and cannot be done concurrently
  - The Autovacuum is famously slow and a bit annoying to manage
- Postgres cares about transactionality
  - So that it can replay all those WAL
  - Hardware was less reliable

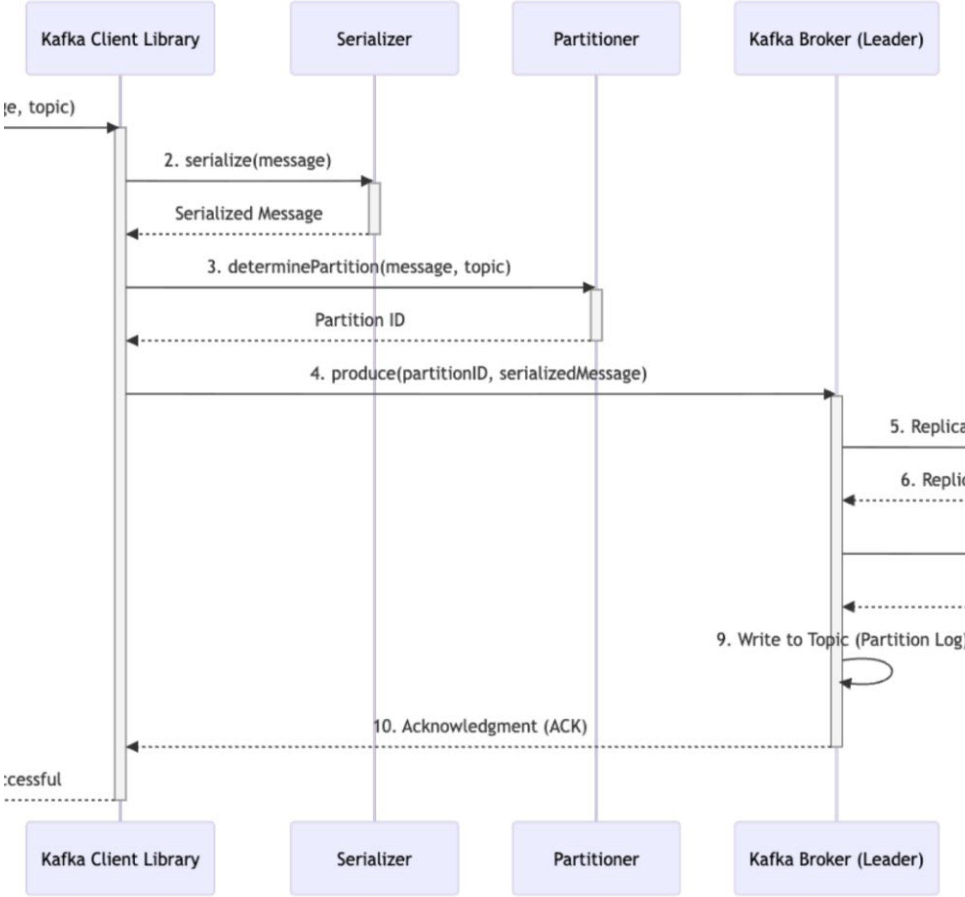
Writes in Kafka



Celeste Horgan · https://celeste.works

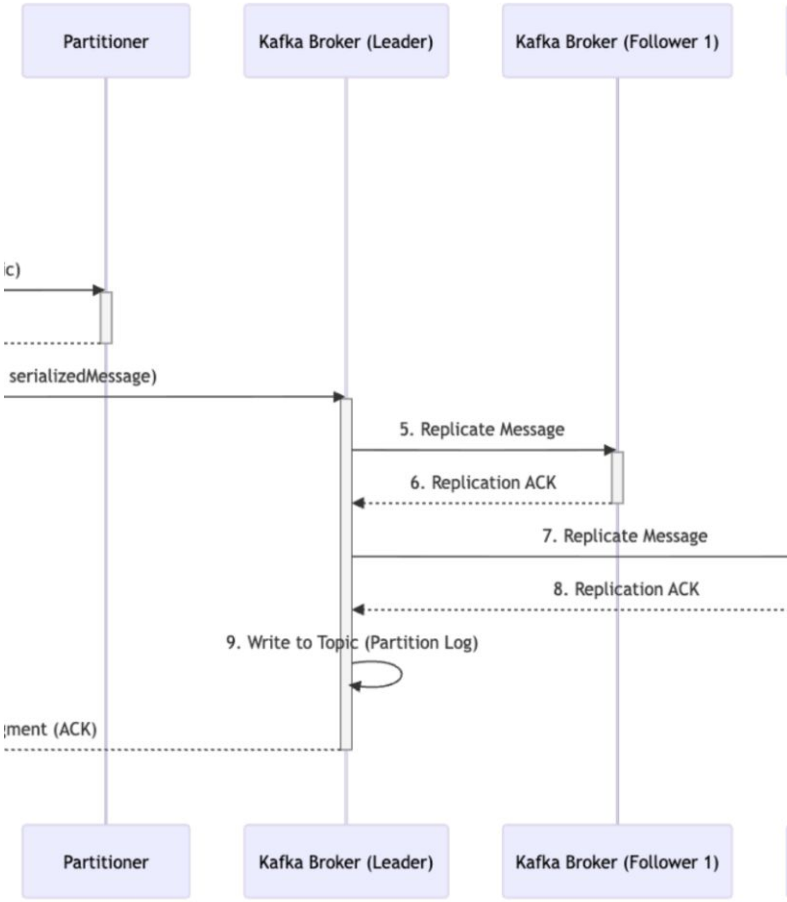
Client → Serializer
Client → Partitioner
Client → Broker

- All messages go through the **serializer**
- Kafka's **partitioner** uses the message key to determine the partition to send the message to
- It then sends the message to the partition's lead **broker**, from where it is replicated to other brokers in the pool
- If all goes well,



Partitioning and broker pools; Implications

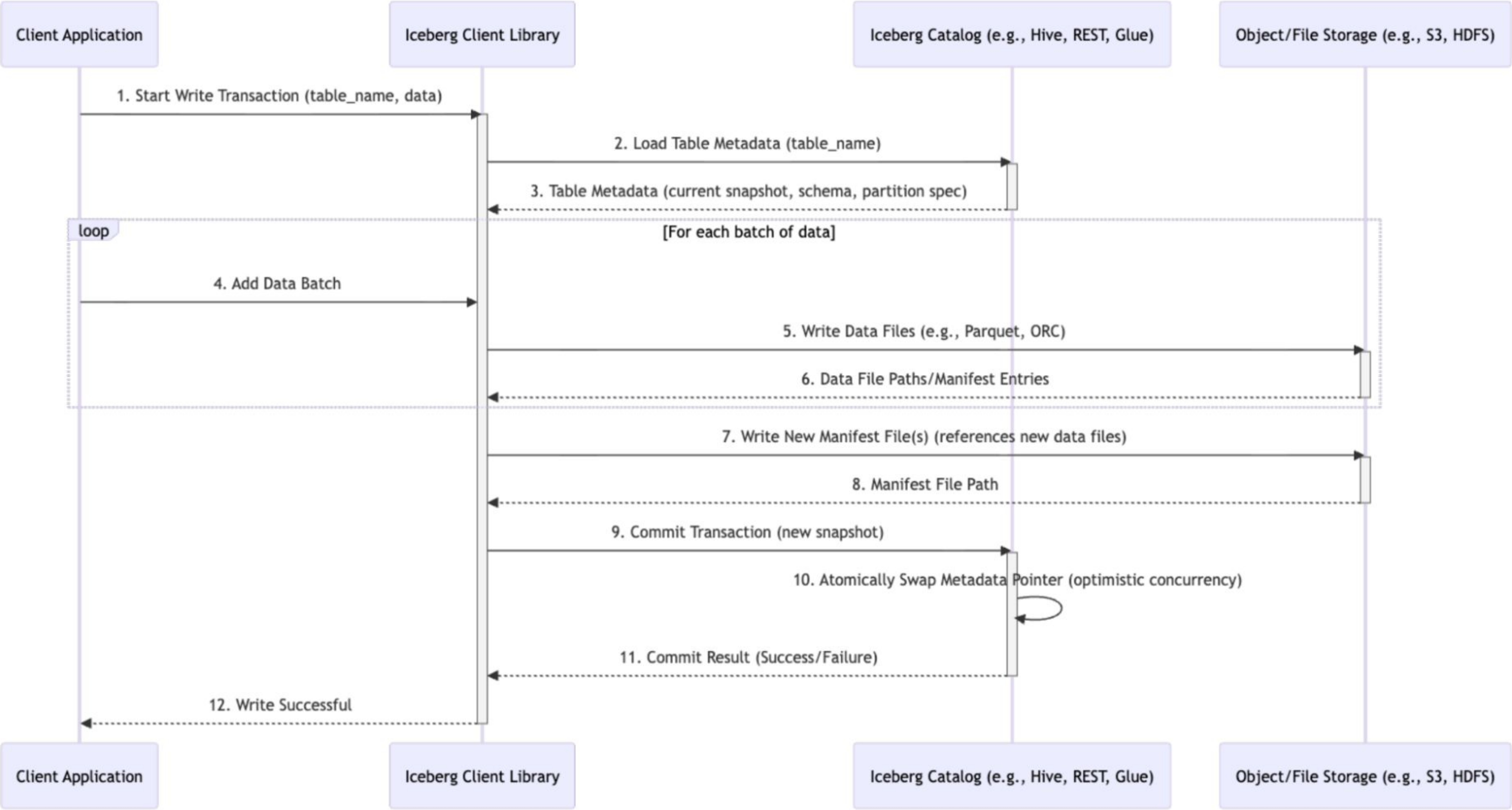
- Topics are partitioned to enable **parallel processing** of messages
  - There is no guarantee that a partition receives all messages
- The brokers replicate partitions for **fault tolerance** purposes
  - There is no guarantee that a broker has replicas of all partitions
- Consumer groups are assigned to a *broker* not a topic
  - There is no guarantee that a consumer will receive all messages in a topic



What did we learn?

- Kafka is *fundamentally* distributed in its design and consistency on reads is a secondary concern
- It builds in distributed computing as a primary system concern
- It introduces an abstraction layer between writing and where the data is written

Writes in Iceberg



Client Application

Iceberg Client Library

Iceberg Catalog (e.g., Hive, REST, Glue)

Object/File Storage (e.g., S3, HDFS)

1. Start Write Transaction (table_name, data)

2. Load Table Metadata (table_name)

3. Table Metadata (current snapshot, schema, partition spec)

loop [For each batch of data]

4. Add Data Batch

5. Write Data Files (e.g., Parquet, ORC)

6. Data File Paths/Manifest Entries

7. Write New Manifest File(s) (references new data files)

8. Manifest File Path

9. Commit Transaction (new snapshot)

10. Atomically Swap Metadata Pointer (optimistic concurrency)

11. Commit Result (Success/Failure)

12. Write Successful

Client Application

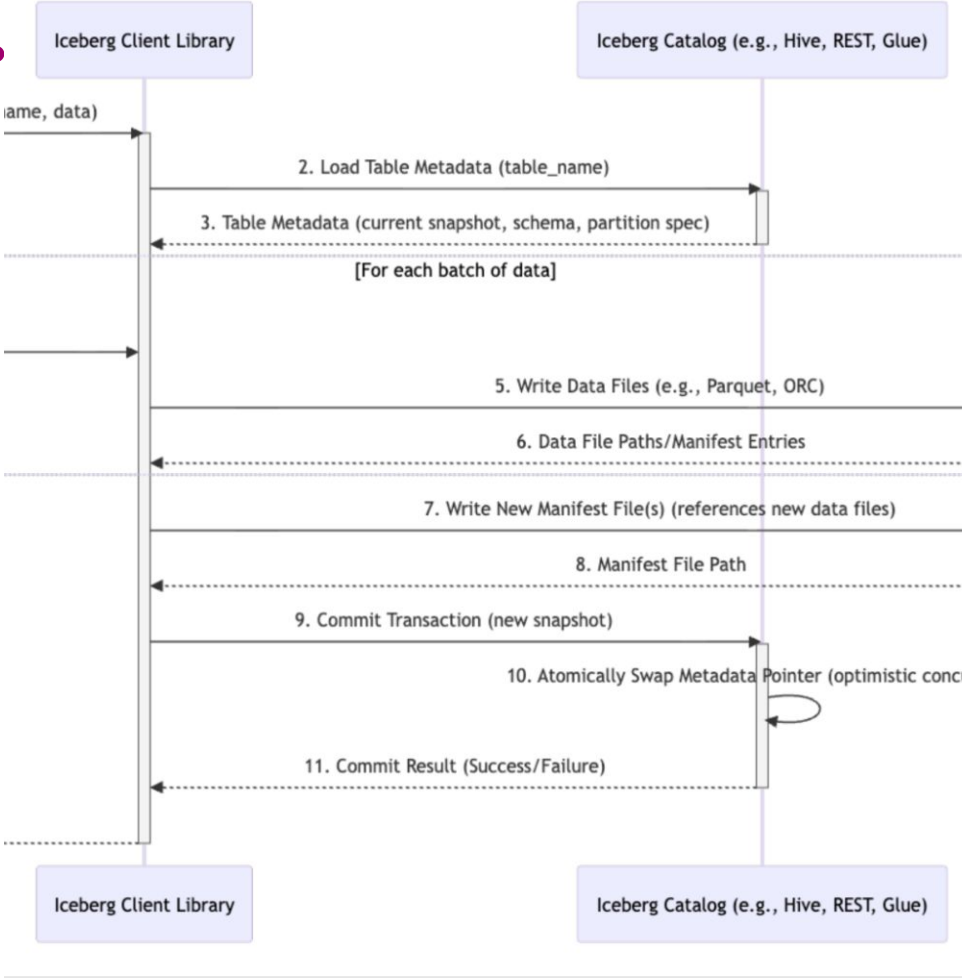
Iceberg Client Library

Iceberg Catalog (e.g., Hive, REST, Glue)

Object/File Storage (e.g., S3, HDFS)

Client → Catalog

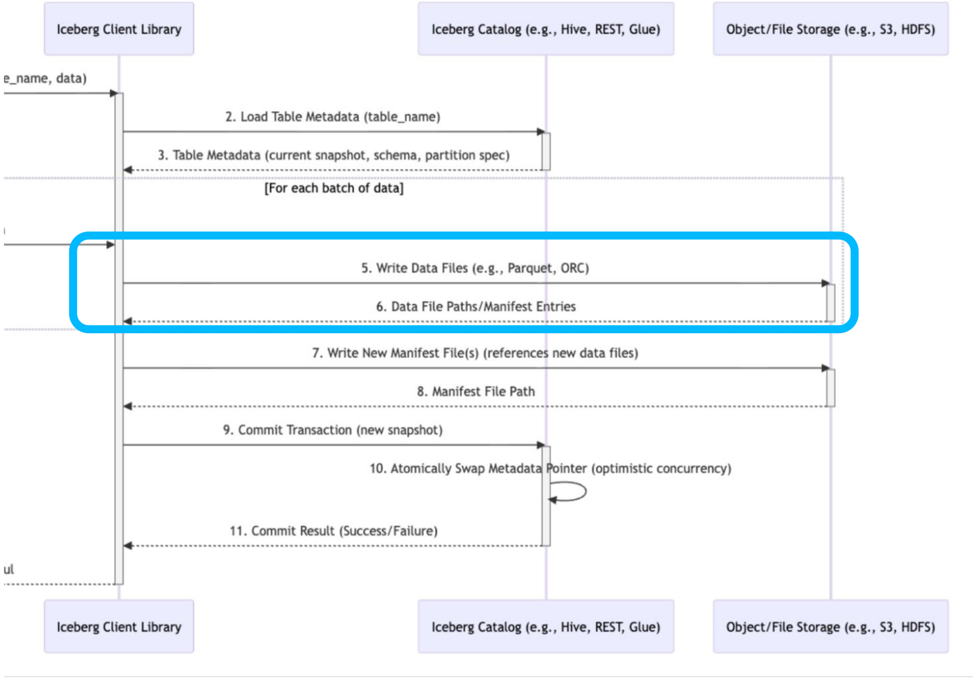
- Retrieves the metadata and snapshot information for the table
- Initiates a write of the data files



Iceberg Client Library
Iceberg Catalog (e.g., Hive, REST, Glue)
ame, data)
2. Load Table Metadata (table_name)
3. Table Metadata (current snapshot, schema, partition spec)
[For each batch of data]
5. Write Data Files (e.g., Parquet, ORC)
6. Data File Paths/Manifest Entries
7. Write New Manifest File(s) (references new data files)
8. Manifest File Path
9. Commit Transaction (new snapshot)
10. Atomically Swap Metadata Pointer (optimistic conc
11. Commit Result (Success/Failure)
Iceberg Client Library
Iceberg Catalog (e.g., Hive, REST, Glue)

Client → Object storage

- Initiates a write of the data files to object storage



Iceberg Client Library

Iceberg Catalog (e.g., Hive, REST, Glue)

Object/File Storage (e.g., S3, HDFS)

e_name, data)

2. Load Table Metadata (table_name)

3. Table Metadata (current snapshot, schema, partition spec)

[For each batch of data]

5. Write Data Files (e.g., Parquet, ORC)

6. Data File Paths/Manifest Entries

7. Write New Manifest File(s) (references new data files)

8. Manifest File Path

9. Commit Transaction (new snapshot)

10. Atomically Swap Metadata Pointer (optimistic concurrency)

11. Commit Result (Success/Failure)

ul

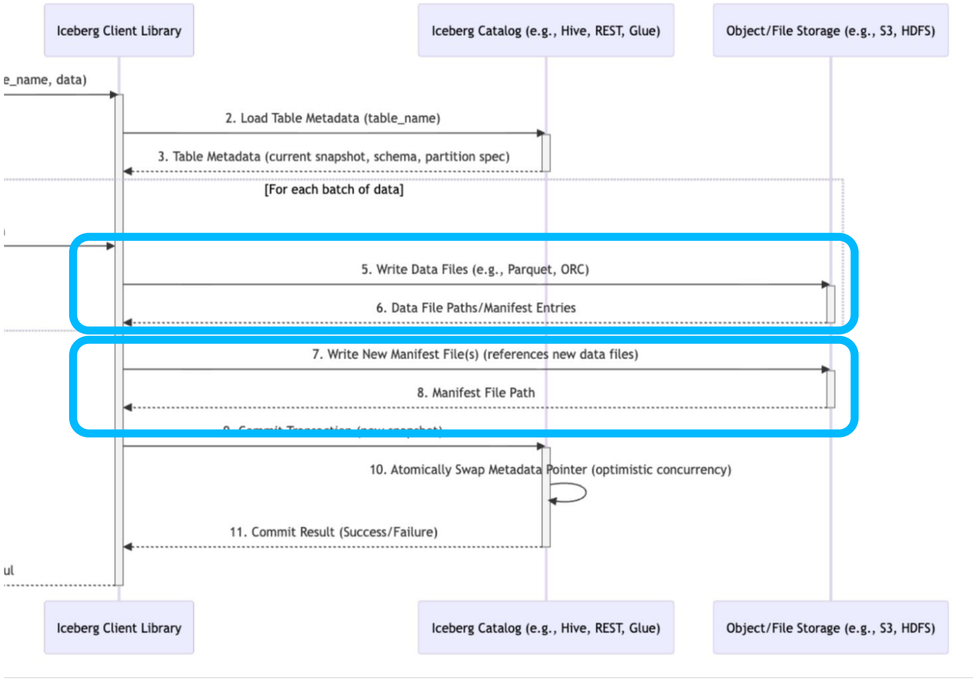
Iceberg Client Library

Iceberg Catalog (e.g., Hive, REST, Glue)

Object/File Storage (e.g., S3, HDFS)

Client → Object storage

- Initiates a write of the data files to object storage
- Then updates the manifest



Iceberg Client Library

Iceberg Catalog (e.g., Hive, REST, Glue)

Object/File Storage (e.g., S3, HDFS)

e_name, data)

2. Load Table Metadata (table_name)

3. Table Metadata (current snapshot, schema, partition spec)

[For each batch of data]

5. Write Data Files (e.g., Parquet, ORC)

6. Data File Paths/Manifest Entries

7. Write New Manifest File(s) (references new data files)

8. Manifest File Path

9. Commit Transaction (new snapshot)

10. Atomically Swap Metadata Pointer (optimistic concurrency)

11. Commit Result (Success/Failure)

ul

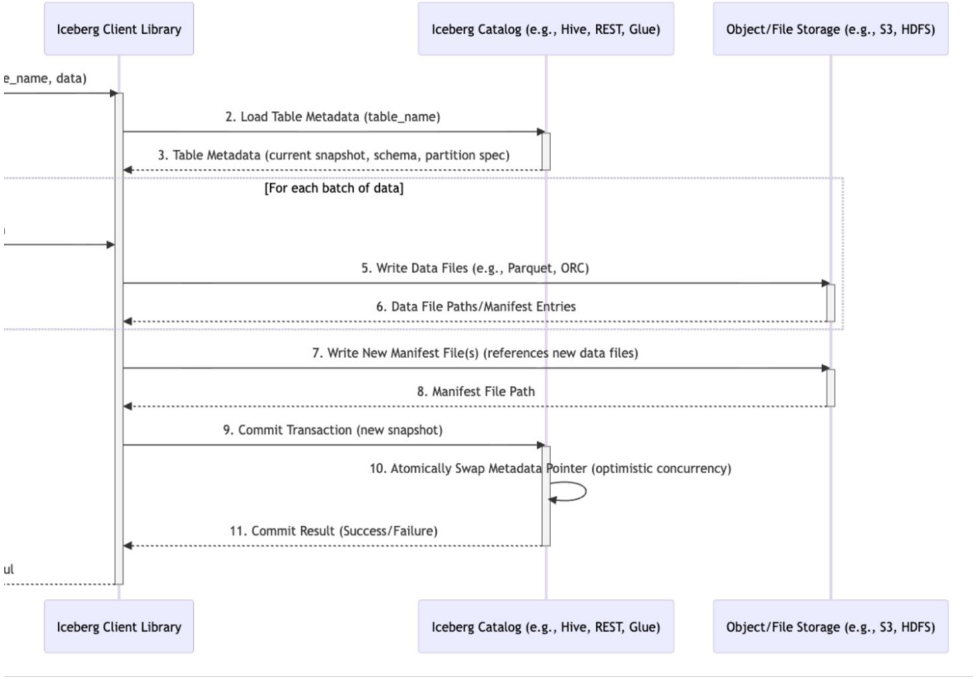
Iceberg Client Library

Iceberg Catalog (e.g., Hive, REST, Glue)

Object/File Storage (e.g., S3, HDFS)

Client → Object storage

- Initiates a write of the data files to object storage
- Updates the manifest for the table
- Commits the transaction to the catalog
  - Creates a new snapshot of the table for the next transaction
  - Updates the metadata pointer for the table



Iceberg Client Library

Iceberg Catalog (e.g., Hive, REST, Glue)

Object/File Storage (e.g., S3, HDFS)

e_name, data)

2. Load Table Metadata (table_name)

3. Table Metadata (current snapshot, schema, partition spec)

[For each batch of data]

5. Write Data Files (e.g., Parquet, ORC)

6. Data File Paths/Manifest Entries

7. Write New Manifest File(s) (references new data files)

8. Manifest File Path

9. Commit Transaction (new snapshot)

10. Atomically Swap Metadata Pointer (optimistic concurrency)

11. Commit Result (Success/Failure)

ul

Iceberg Client Library

Iceberg Catalog (e.g., Hive, REST, Glue)

Object/File Storage (e.g., S3, HDFS)

What have we learned

- Iceberg is table shaped, but is not inherently a data store
  - It’s an **abstraction** layer
- It provides a consistent way of talking to data services because it’s implementation is lightweight
  - **User experience** for the human developer
- Decoupling **storage and compute** is useful for reasons beyond scalability

What does this mean for data engineers?

What we’ve learned

- **Distributed computing** is the way forward
  - Flink and Kafka were both designed ground-up for distributed computing
- **Event processing** is the way forward
  - Because we’re handling large amounts of data generated in near realtime from various systems

However…

- **SQL and tables are conceptually useful ideas** for users
  - Even when the underlying system isn’t really table-shaped, i.e. Flink
- **Data eventually needs to ‘rest’**
  - Preferably in the cheapest way possible

Meanwhile

- **Agentic AI is on the rise and has different needs than humans**
  - MCP Servers are one part of the solution through strong adherence to JSON-RPC protocol
- A big need is for **semantic meaning** to be embedded alongside data in a consistent fashion:
  - Check out Snowflake's Open Semantic Interchange, launching soon: https://www.snowflake.com/en/blog/open-semantic-interchange-ai-standard/

Thank you so much!

https://www.linkedin.com/in/celeste-horgan/
https://bsky.app/profile/celestehorgan.bsky.social
https://celeste.works

P.S. - We [Snowflake] are hosting an OSS data meetup on February 2nd. Follow me on LinkedIn for the official announce later this week!

Title

- Body text
    - `this is code`

Title

- Body text
  - `this is code`

This is a section page