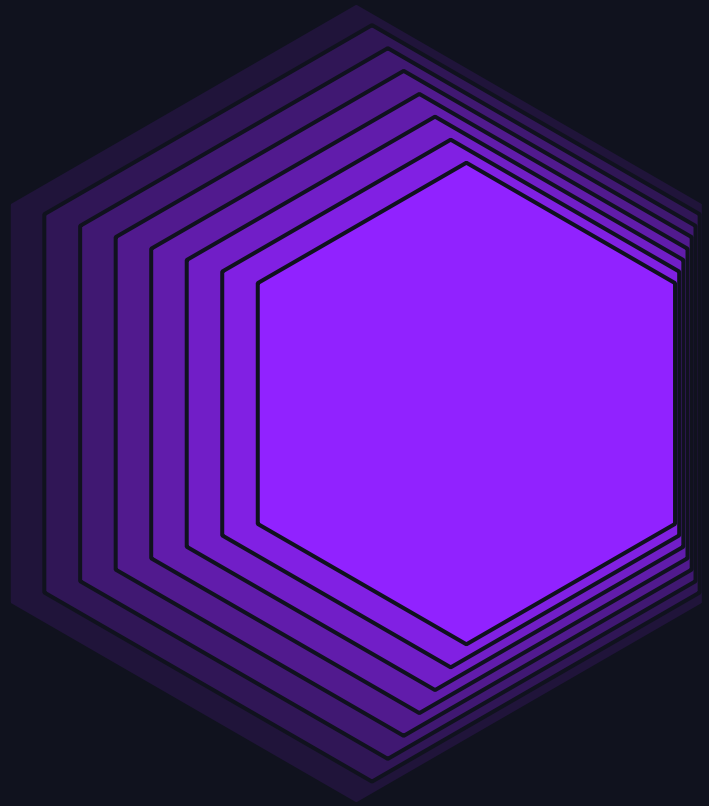
DATA+AI SUMMIT BY databricks

DELTA LAKE AND MICROSERVICES



Alex Wilcoxson and Greg Ott
2024-06-10

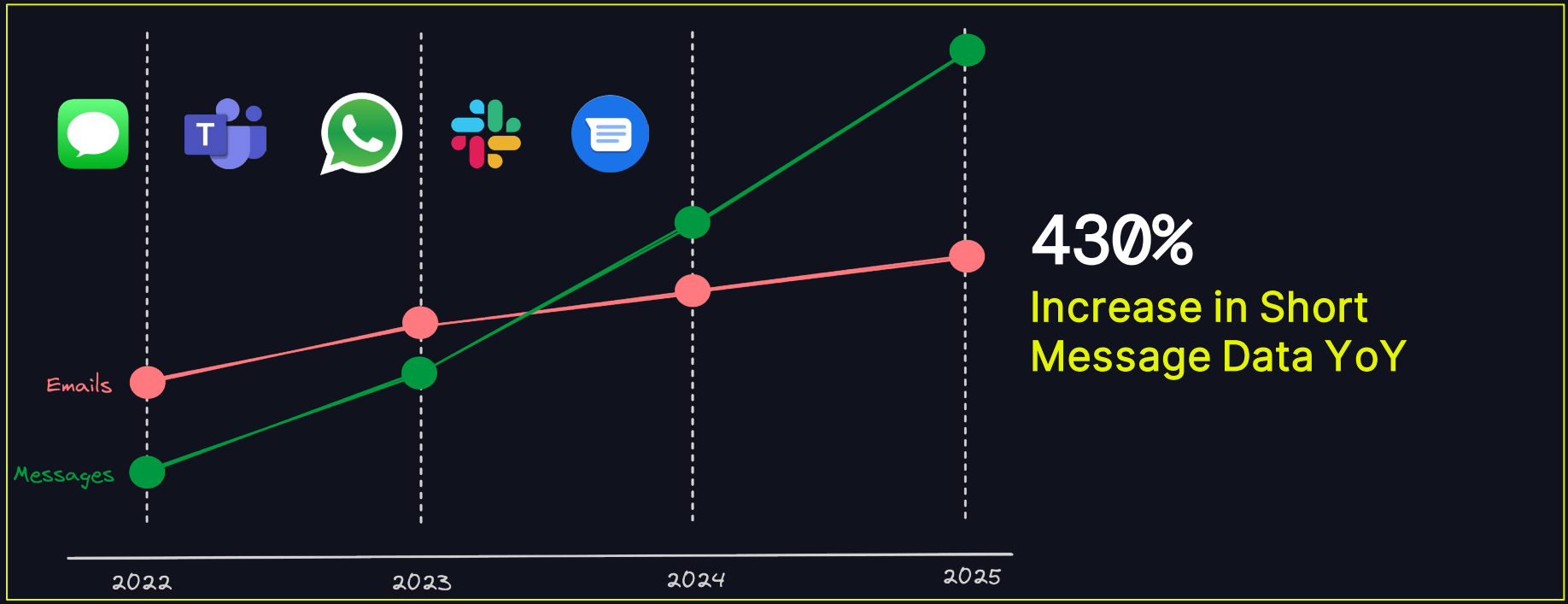
DATA+AI SUMMIT BY databricks

SEMI-STRUCTURED COMMUNICATION DATA WITH DELTA LAKE AND MICROSERVICES

Evolving our data store to handle increasing complexity and volume of communication data

VOLUME AND COMPLEXITY

Short message data is the new normal

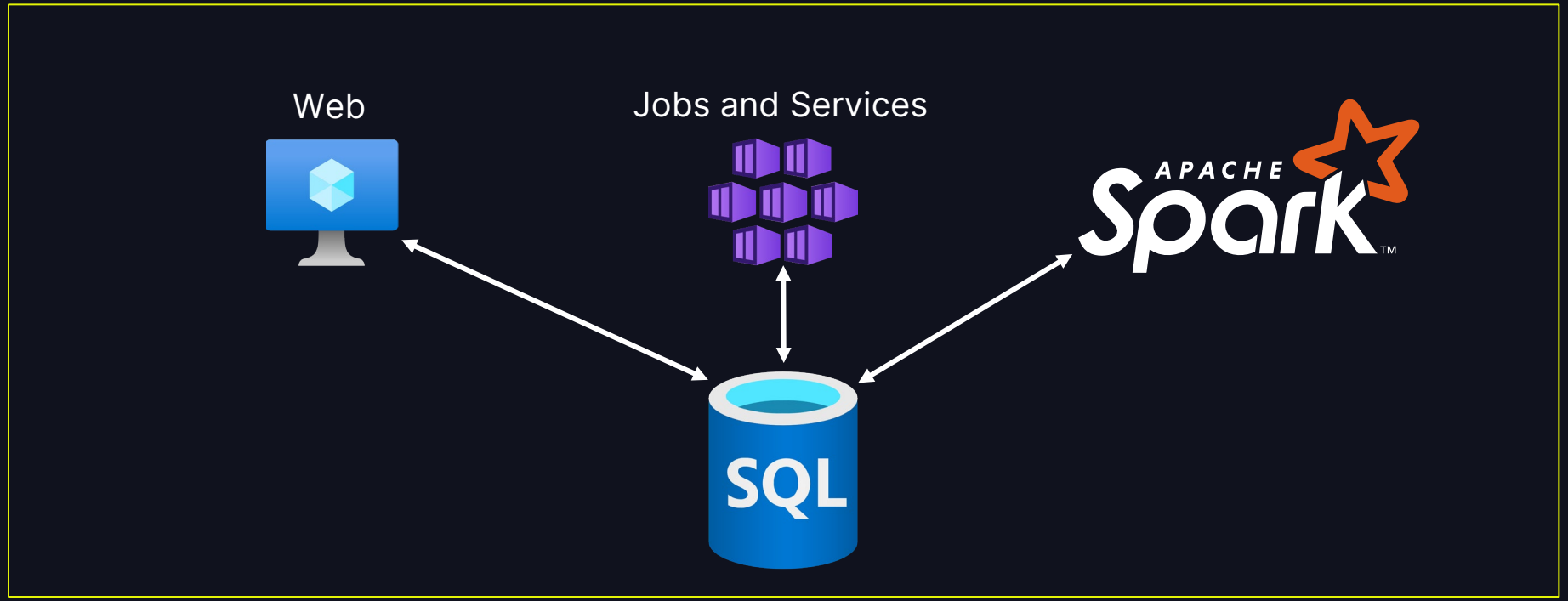


430%

Increase in Short Message Data YoY

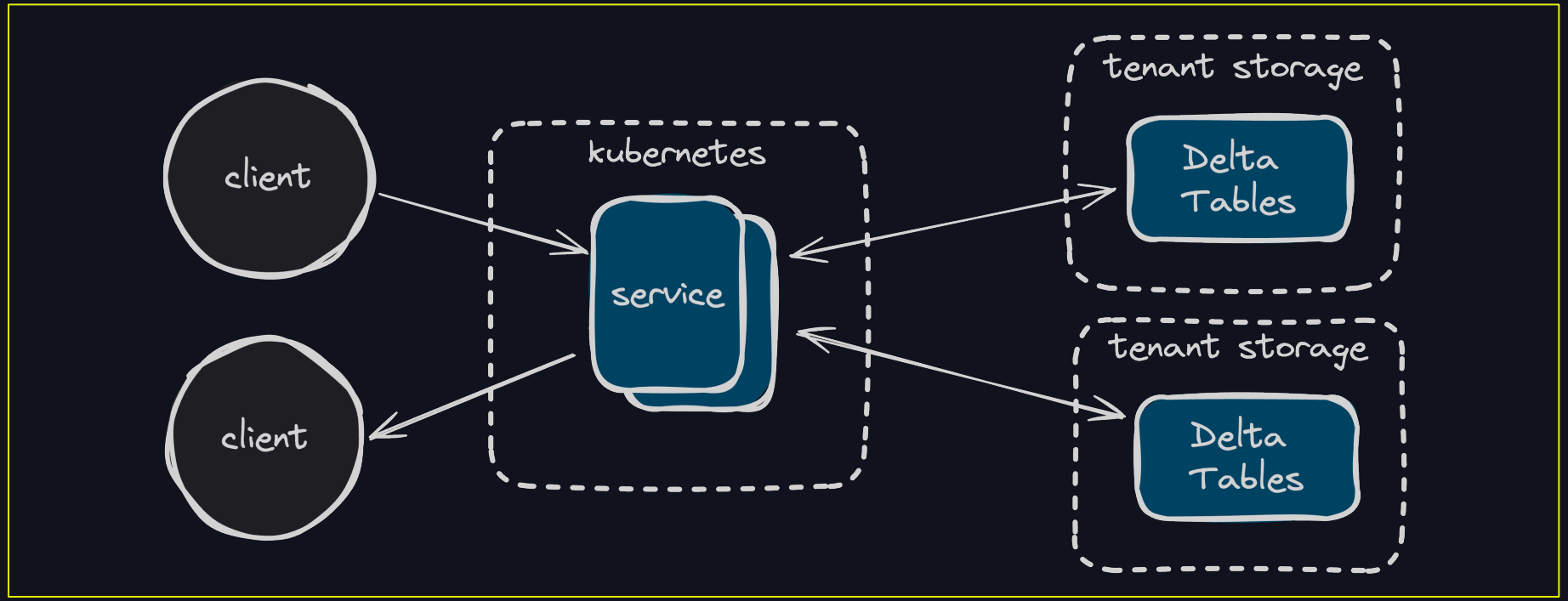
VOLUME AND COMPLEXITY

Challenges scaling for individual message volume



NEW DATA STORE

Scaling for volume and complexity



ALEX WILCOXSON

Staff Software Engineer

- I've worked at Relativity for 11 years.
- .NET, SQL Server, and front-end full stack
- Distributed systems, Kubernetes, Spark
- Data Engineering and Machine Learning

GREG OTT

Staff Software Engineer

- I've worked as a developer at Relativity for 7 years
- Originally started with DevOps, then onto C# and JS and finally into Scala and Rust
- Worked with AI/Analytics/Big data for the past 5 years

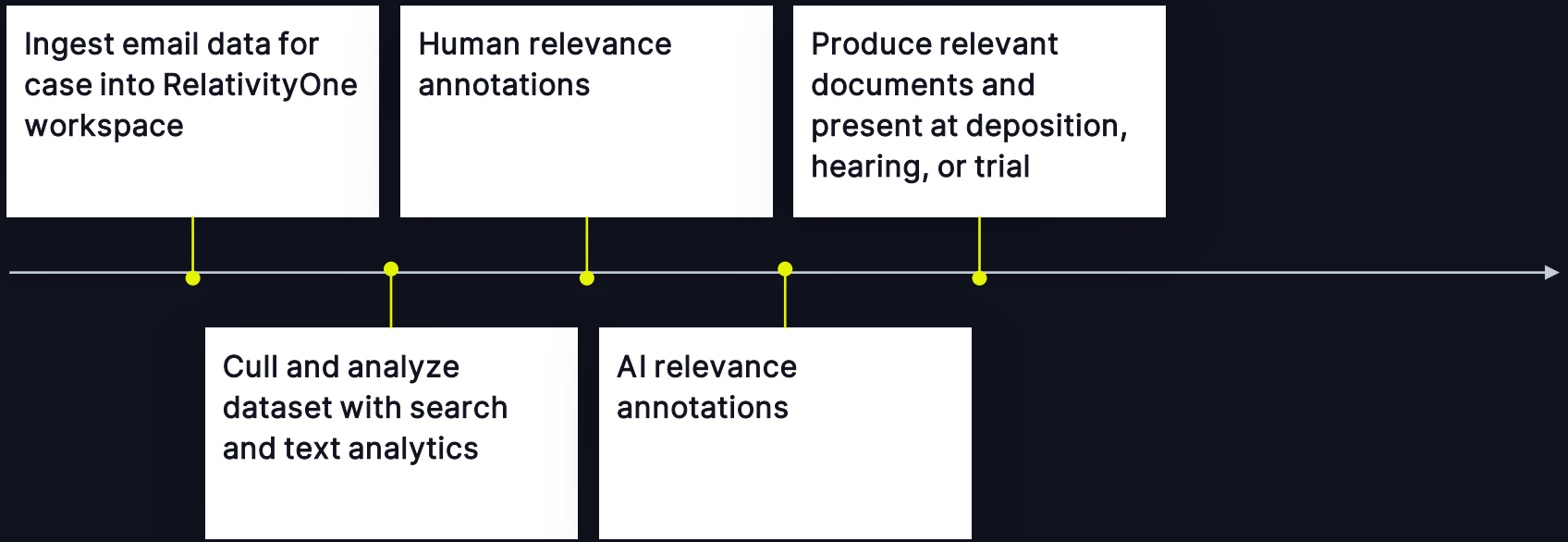
AGENDA

- Introduction to RelativityOne and short message data
- Architecture overview: Dive into the tech stack
- Deep dive into system and lessons learned
- Closing remarks and next steps

RELATIVITY AND SHORT MESSAGE DATA

WHAT IS RELATIVITYONE?

Example workflow



SHORT MESSAGE DATA

Messages as documents?

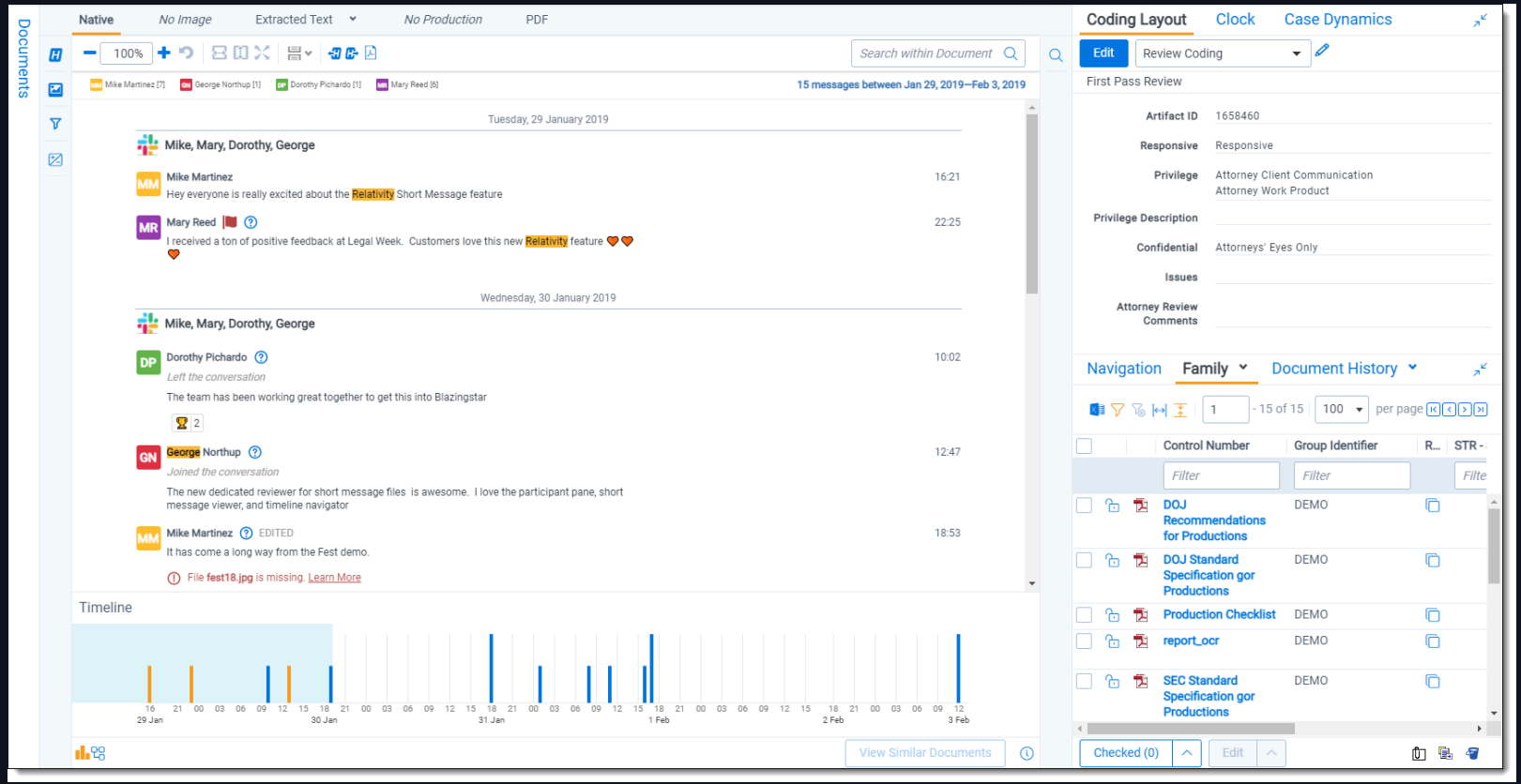
FILE 1

```
[{
  "date": "2024-06-09",
  "body": "Hey Greg! I'm going to DAIS this year, are you?",
  "participant": "Alex"
}, {
  "date": "2024-06-09",
  "body": "Hey Alex, I am. Want to catch up?",
  "participant": "Greg"
}, {
  "date": "2024-06-09",
  "body": "Sure, how about coffee on Tuesday?",
  "participant": "Alex",
  "reactions": [{
    "emoji": ":thumbsup:",
    "participant": "Greg"
  }]
}]
```

FILE 2

```
[{
  "date": "2024-06-11",
  "body": "Hey Greg, meet at 9?",
  "participant": "Alex"
}, {
  "date": "2024-06-11",
  "body": "Greg?",
  "participant": "Alex"
}, {
  "date": "2024-06-11",
  "body": "Greg where are you?!",
  "participant": "Alex"
}]
```

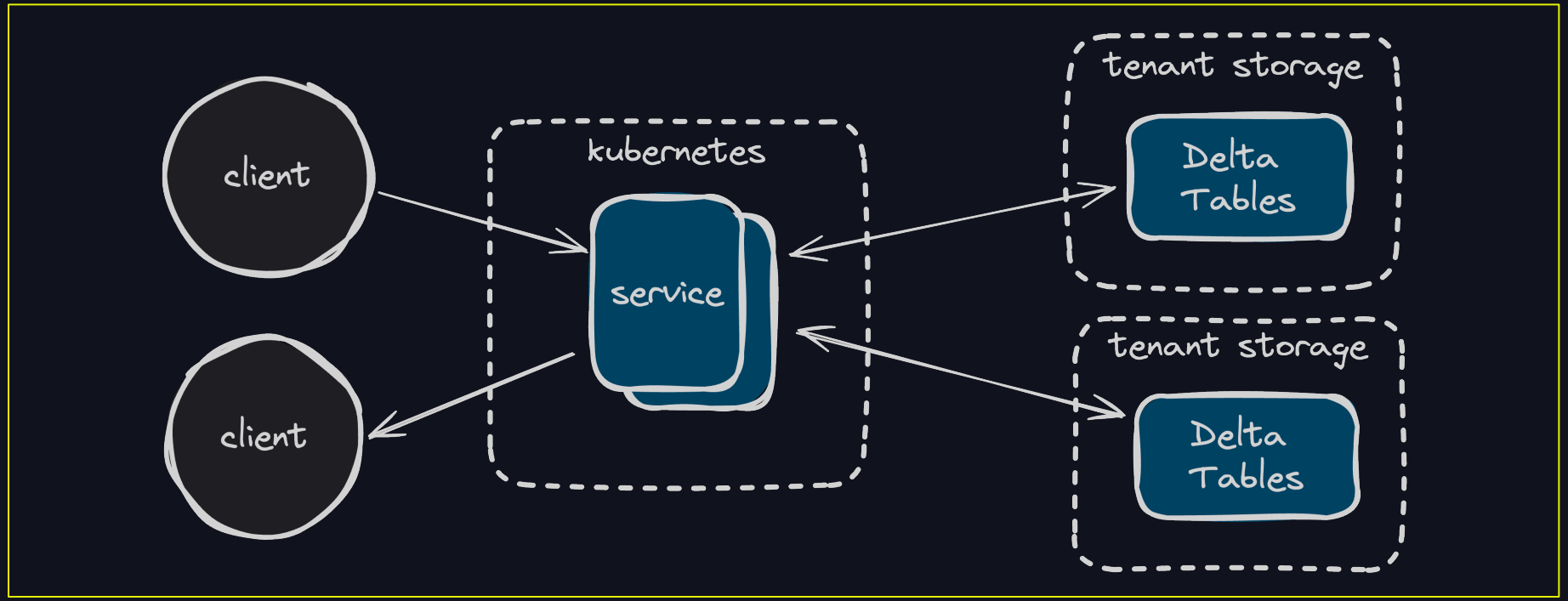
THE RELATIVITY REVIEW INTERFACE



ARCHITECTURE OVERVIEW AND TECHNOLOGY SELECTION

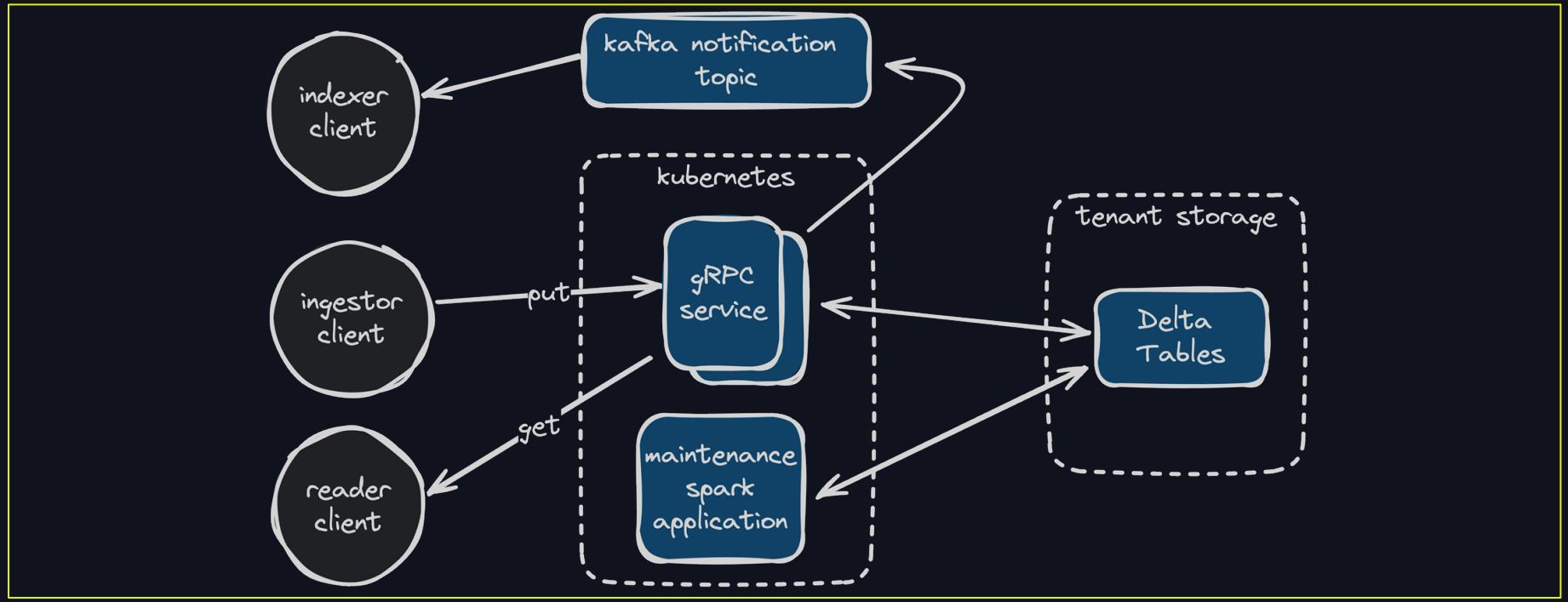
ARCHITECTURE OVERVIEW

Evolving RelativityOne's Data Platform



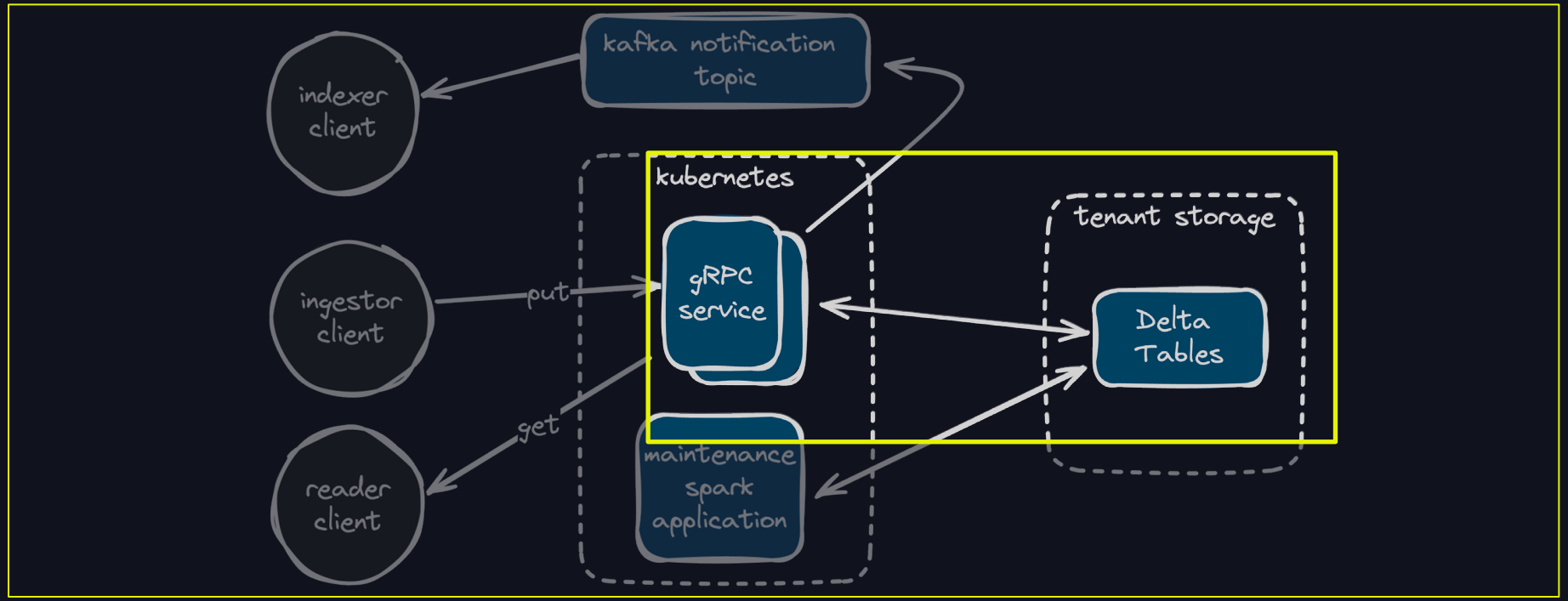
ARCHITECTURE OVERVIEW

Evolving RelativityOne's Data Platform



ARCHITECTURE OVERVIEW

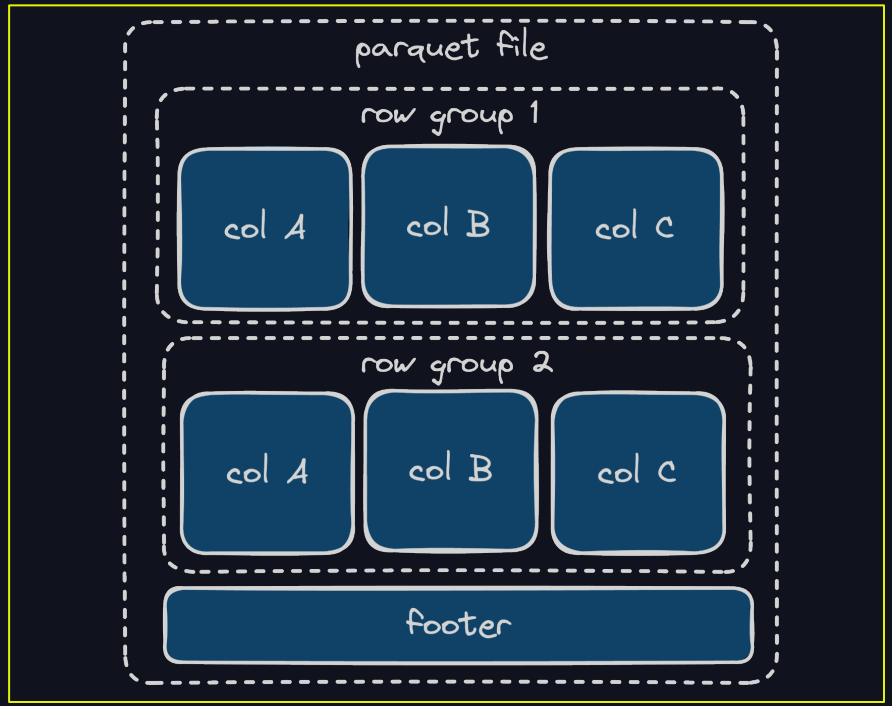
Evolving RelativityOne's Data Platform



PARQUET

Columnar storage

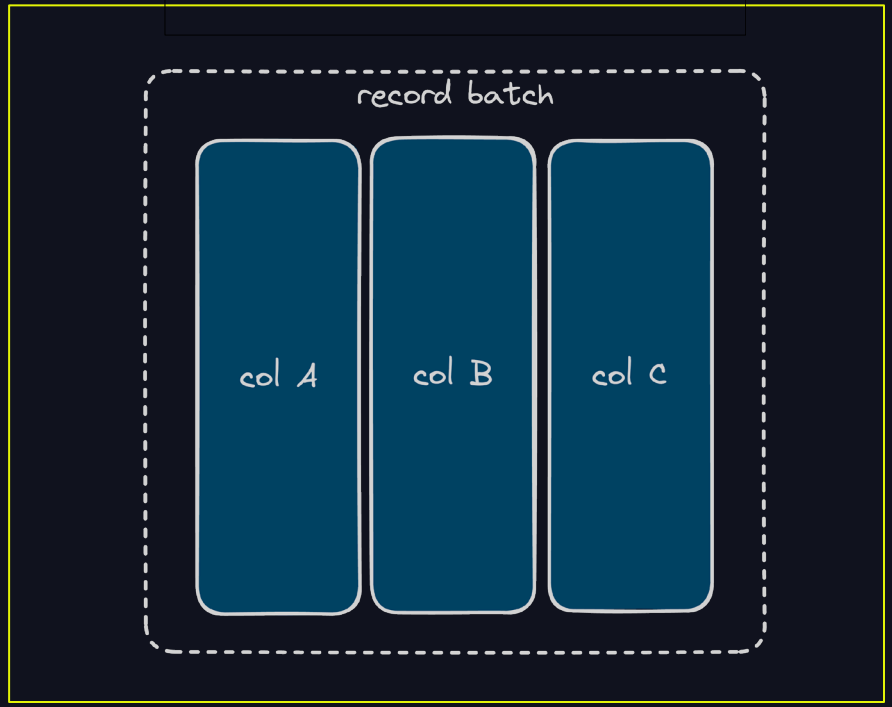
- Columnar
- Compression, encodings
- Schema
- Statistics, metadata
- Arrow integration



ARROW

In-memory columnar

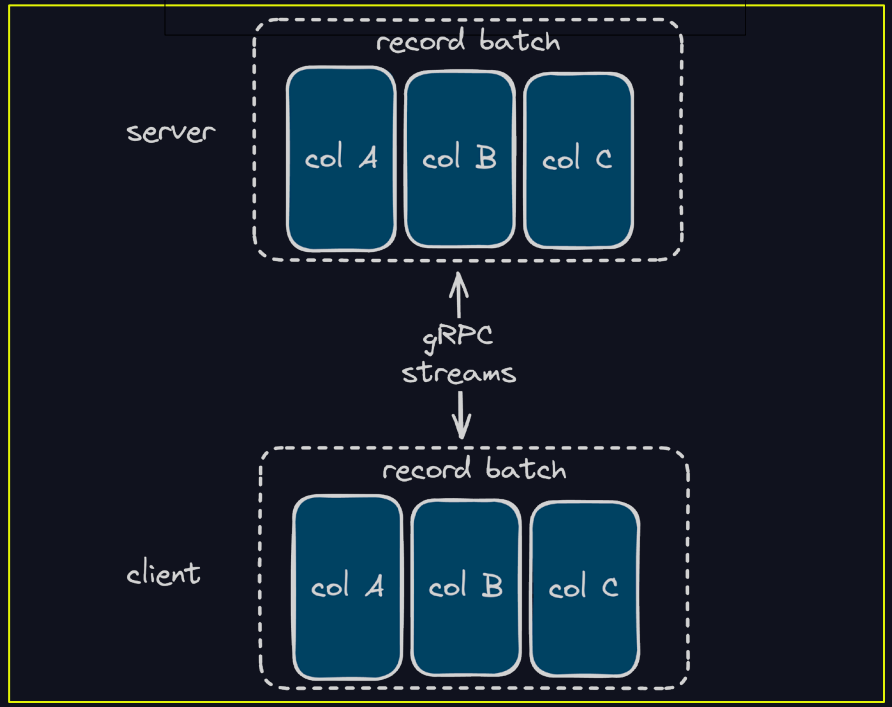
- Columnar, vectorized
- Encoding
- Schema



ARROW

In-memory columnar

- Columnar, vectorized
- Encoding
- Schema
- Flight gRPC
- DataFusion's format

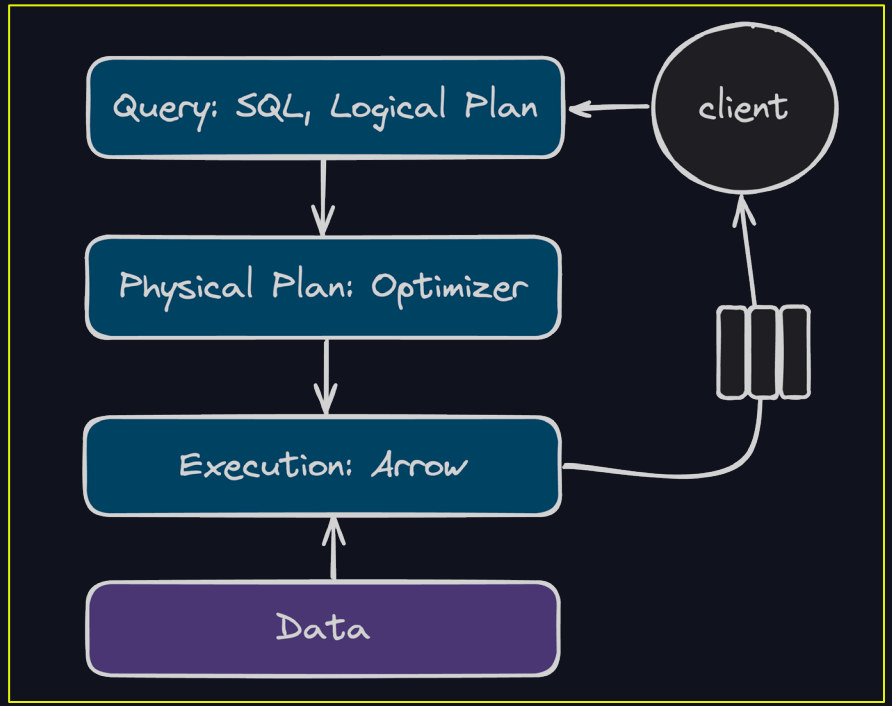


DATAFUSION

Embeddable Query Engine

APACHE DATAFUSION™

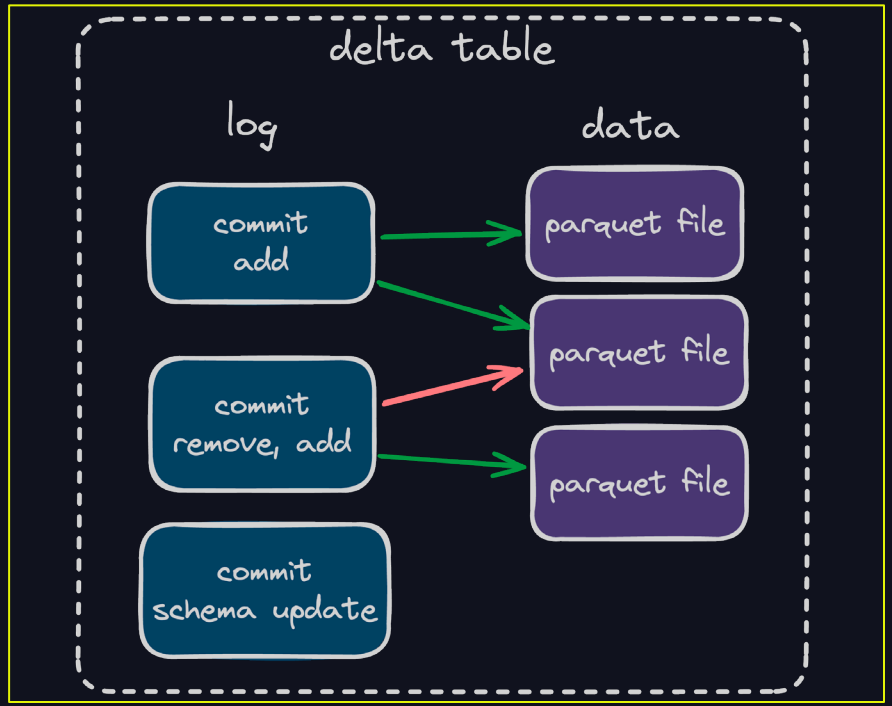
- More of a library
- Extensible, reusable
- Columnar, vectorized (Arrow!)
- Delta Lake Rust integration



DELTA LAKE

Table framework for Parquet

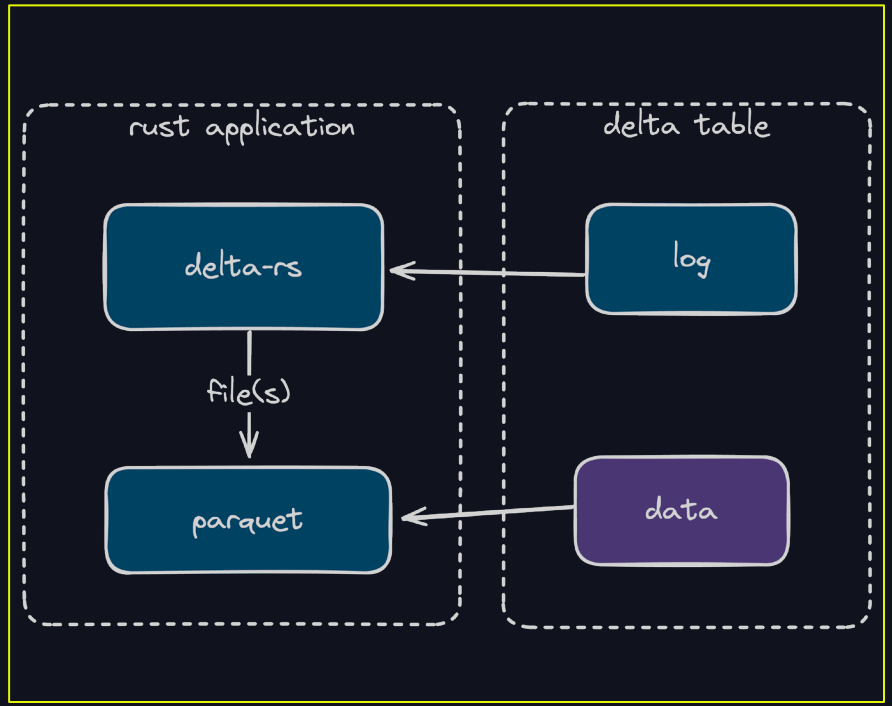
- Commits, transactions
- Schema evolution
- Optimizations



DELTA LAKE

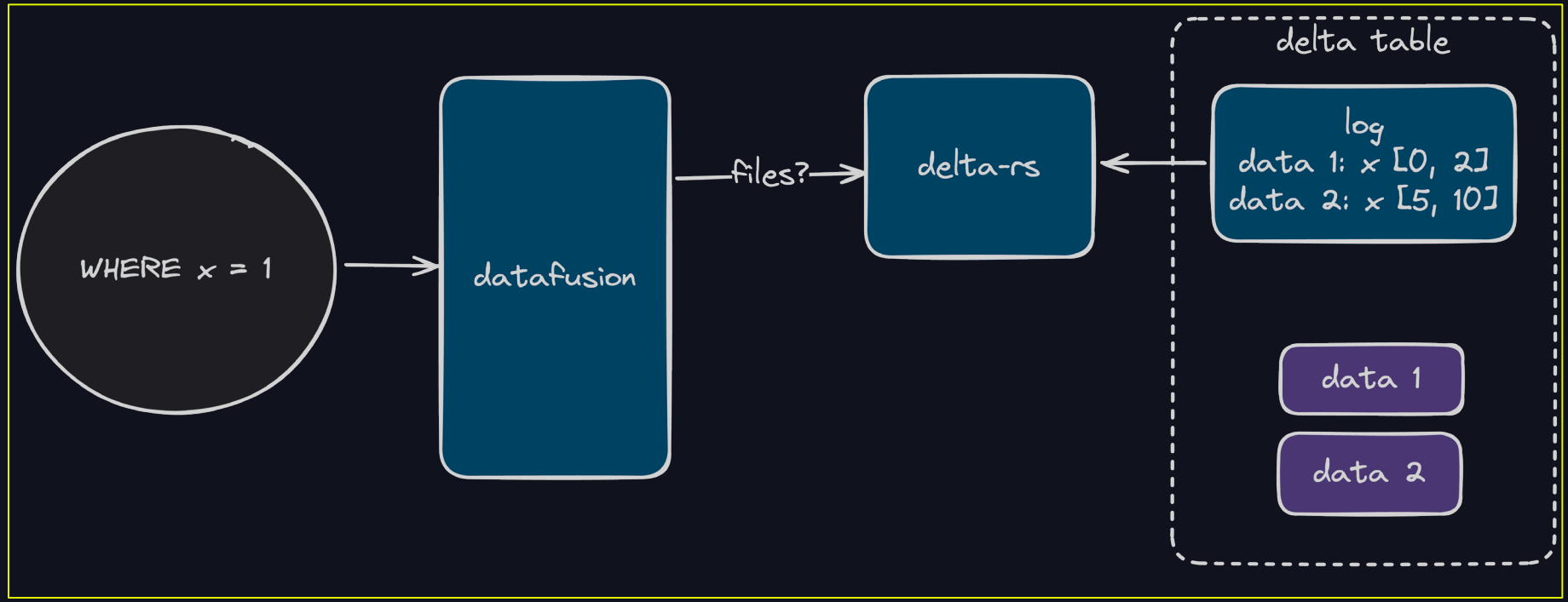
Table framework for Parquet

- Commits, transactions
- Schema evolution
- Optimizations
- Rust standalone
- Statistics, metadata
- Data skipping
- DataFusion example



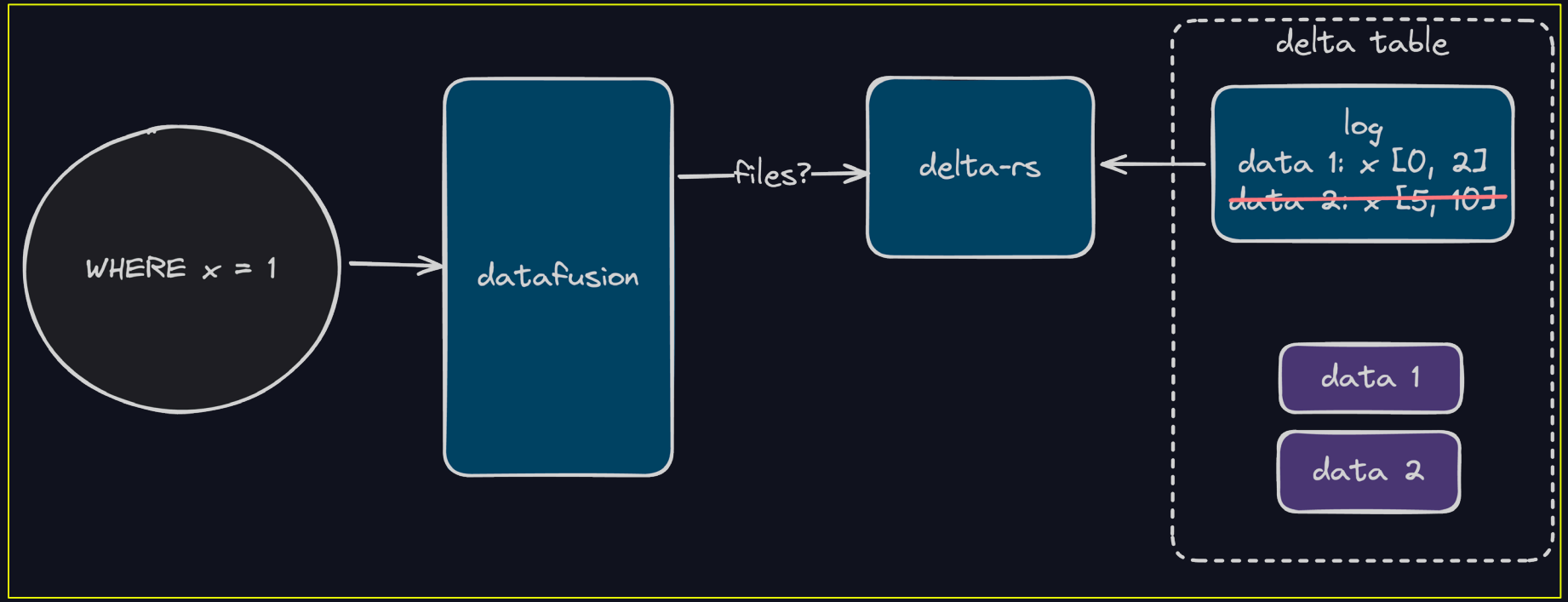
DATAFUSION QUERY

Example of delta-rs DataFusion integration



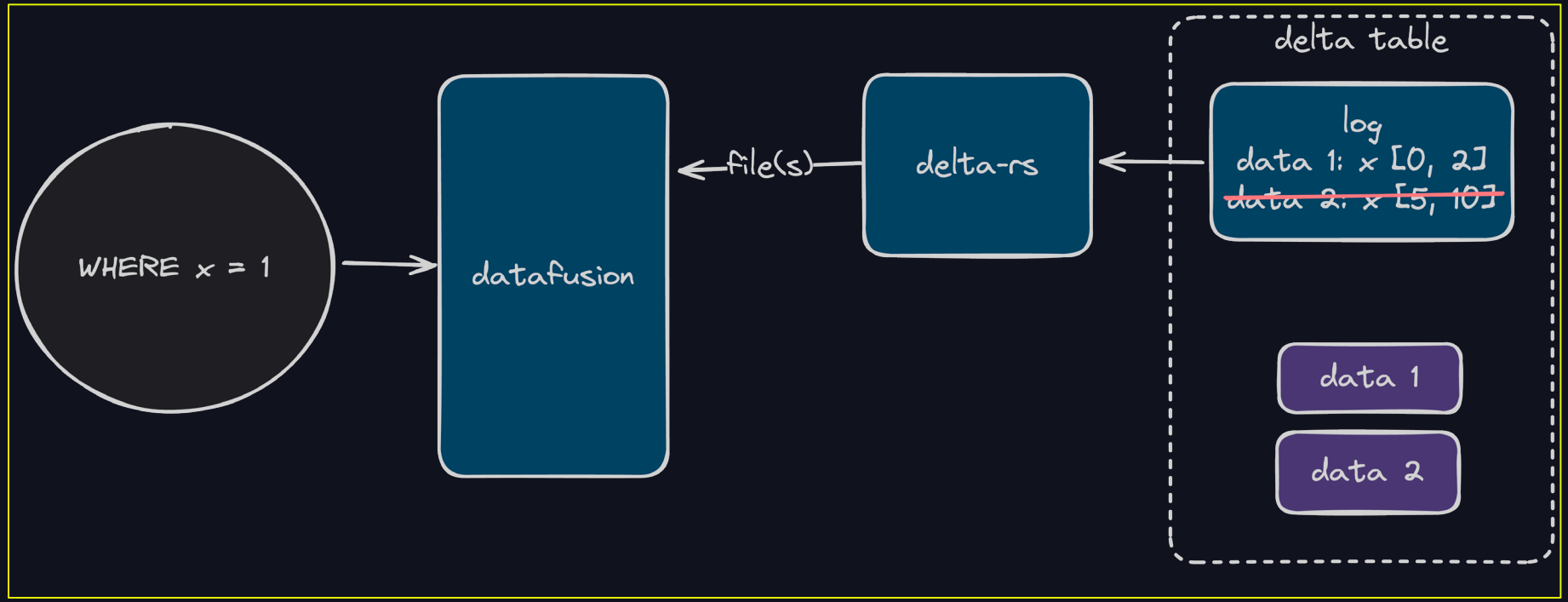
DATAFUSION QUERY

Example of delta-rs DataFusion integration



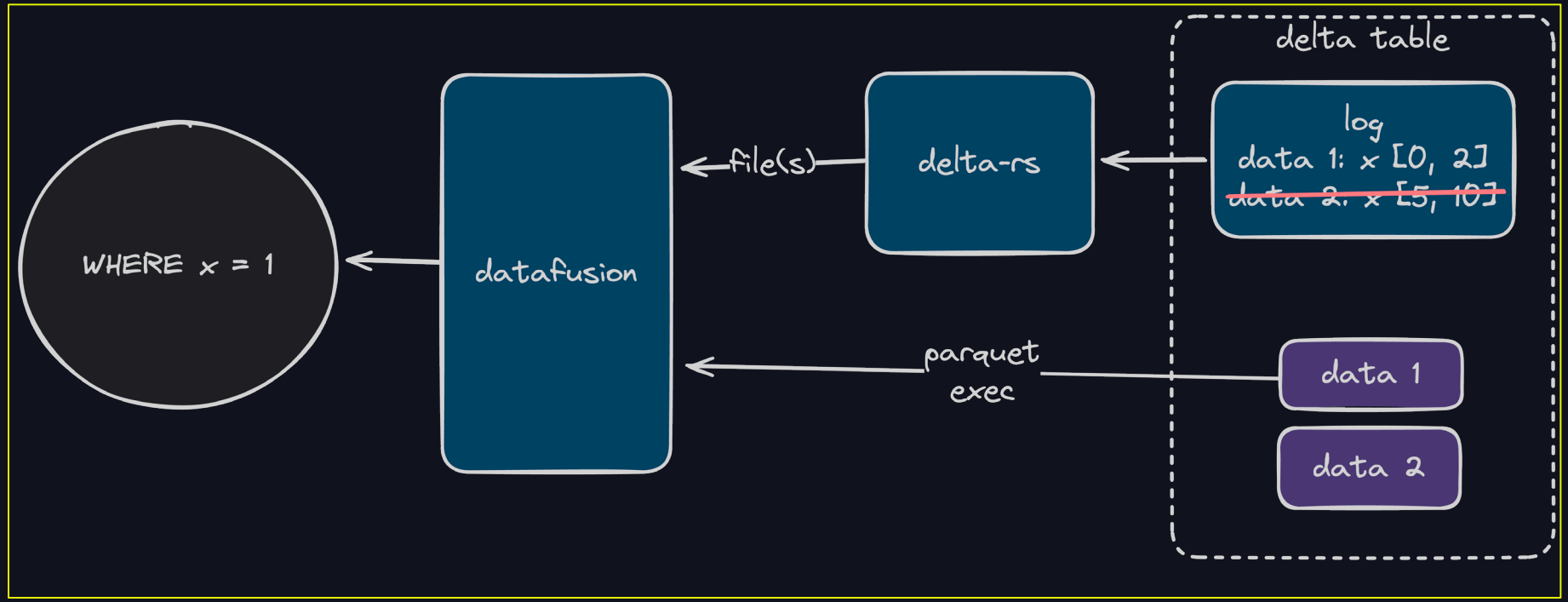
DATAFUSION QUERY

Example of delta-rs DataFusion integration



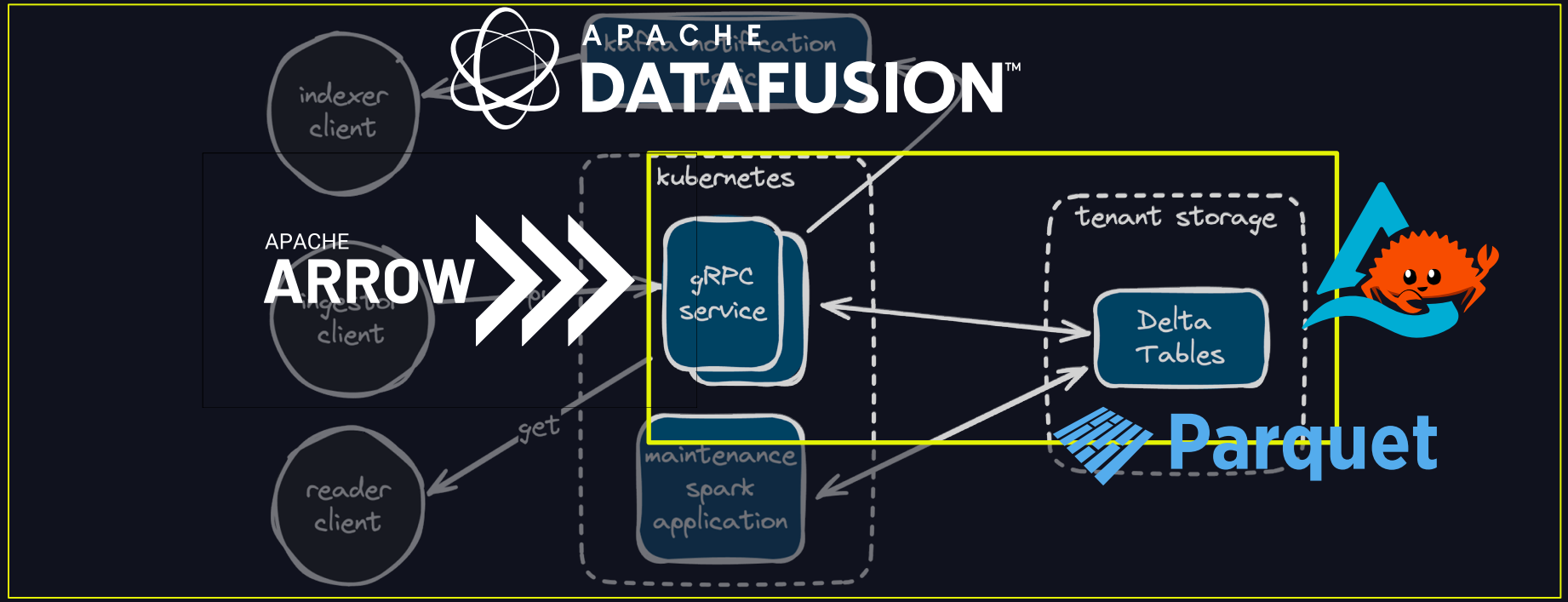
DATAFUSION QUERY

Example of delta-rs DataFusion integration



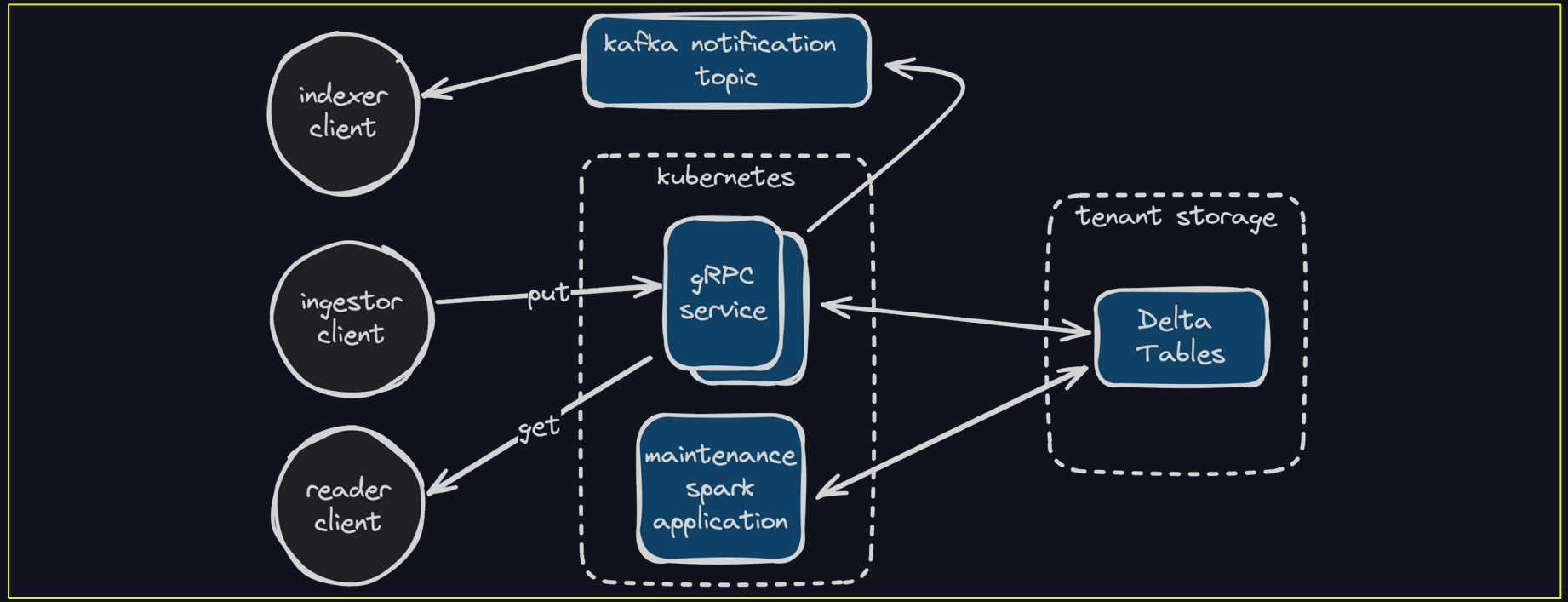
ARCHITECTURE RECAP

Scalable, structured, high throughput, multi-tenant storage system

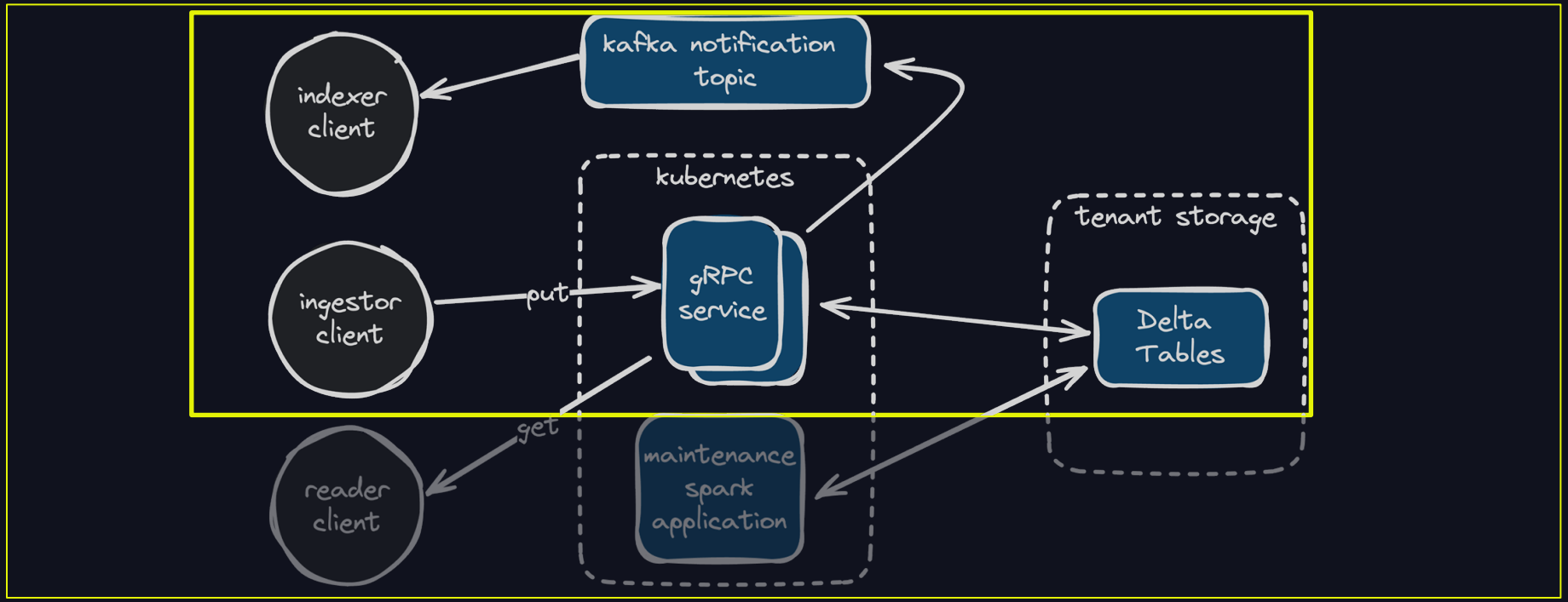


DATA INGESTION CHALLENGES

DATA INGESTION

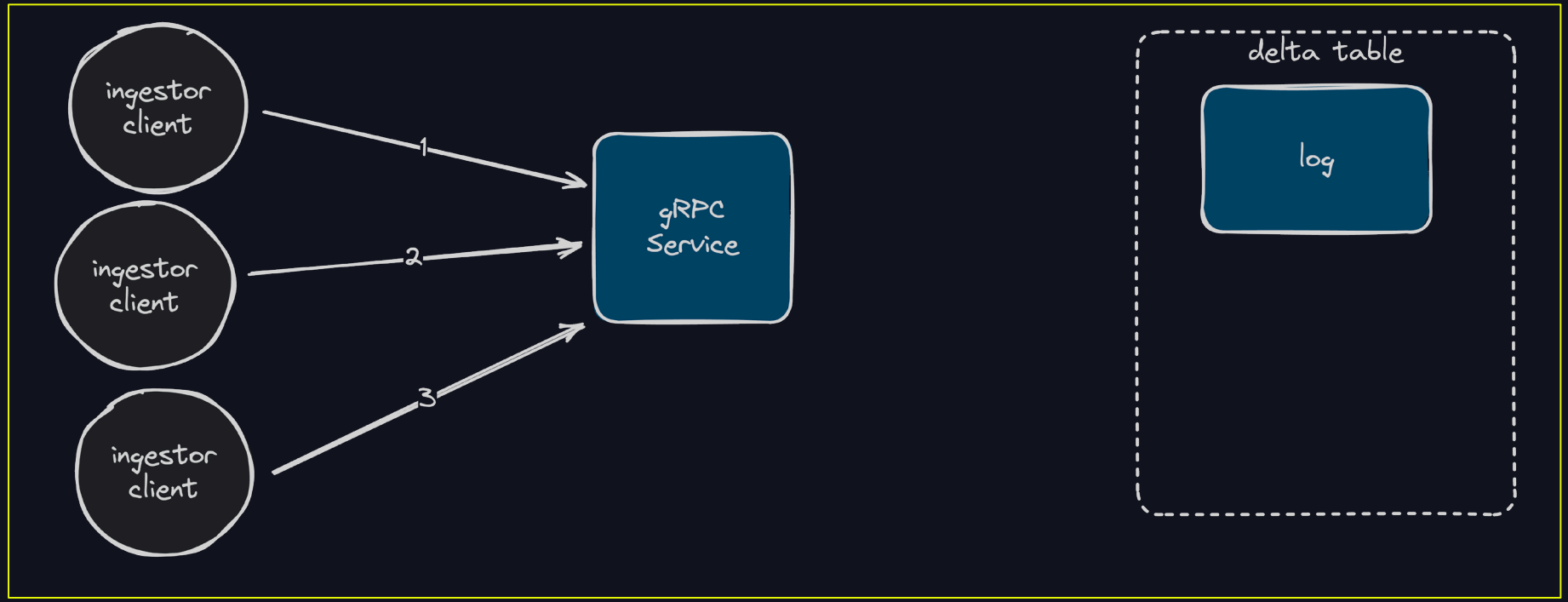


DATA INGESTION



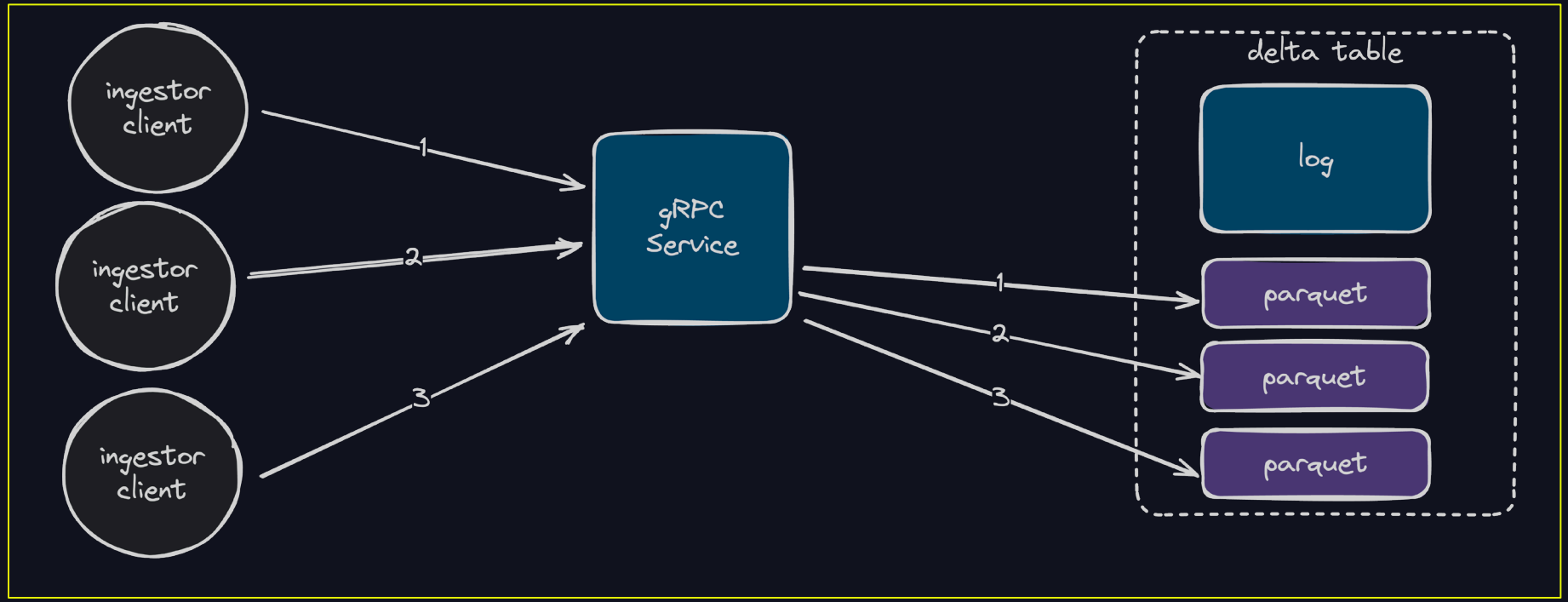
INGESTION CHALLENGES

Too much concurrency!



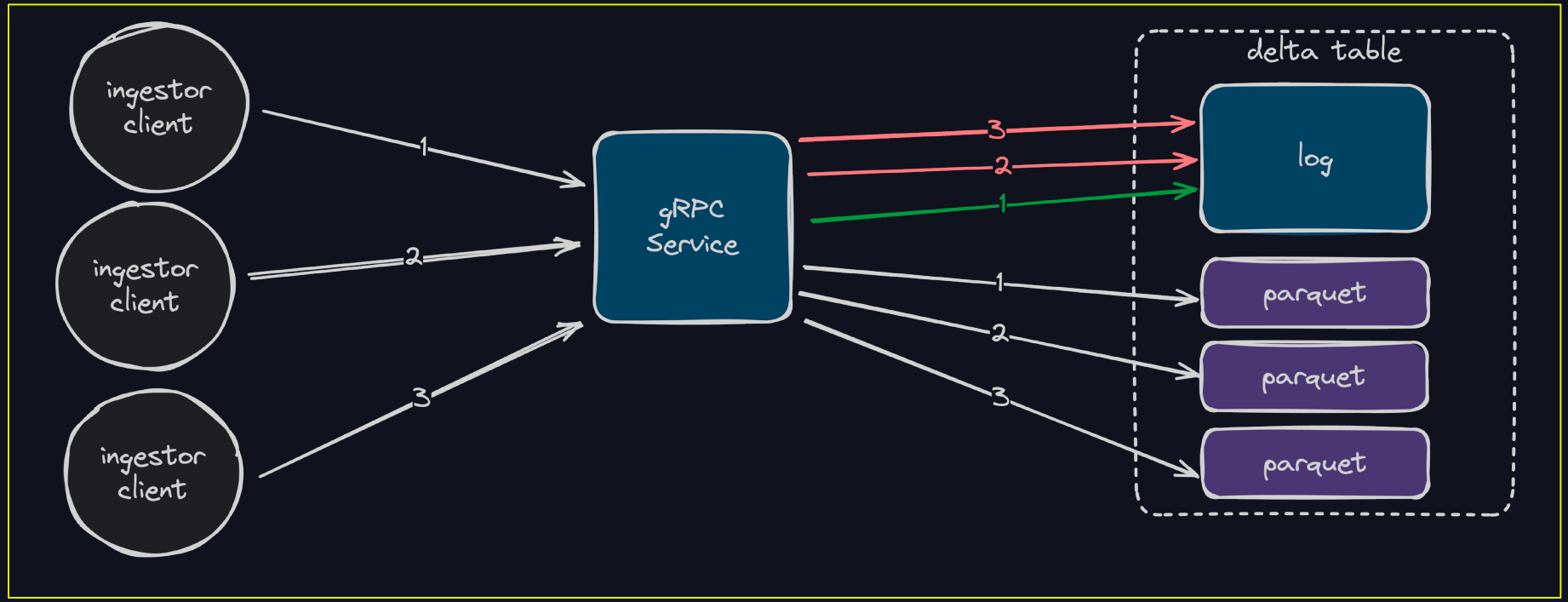
INGESTION CHALLENGES

Too much concurrency!



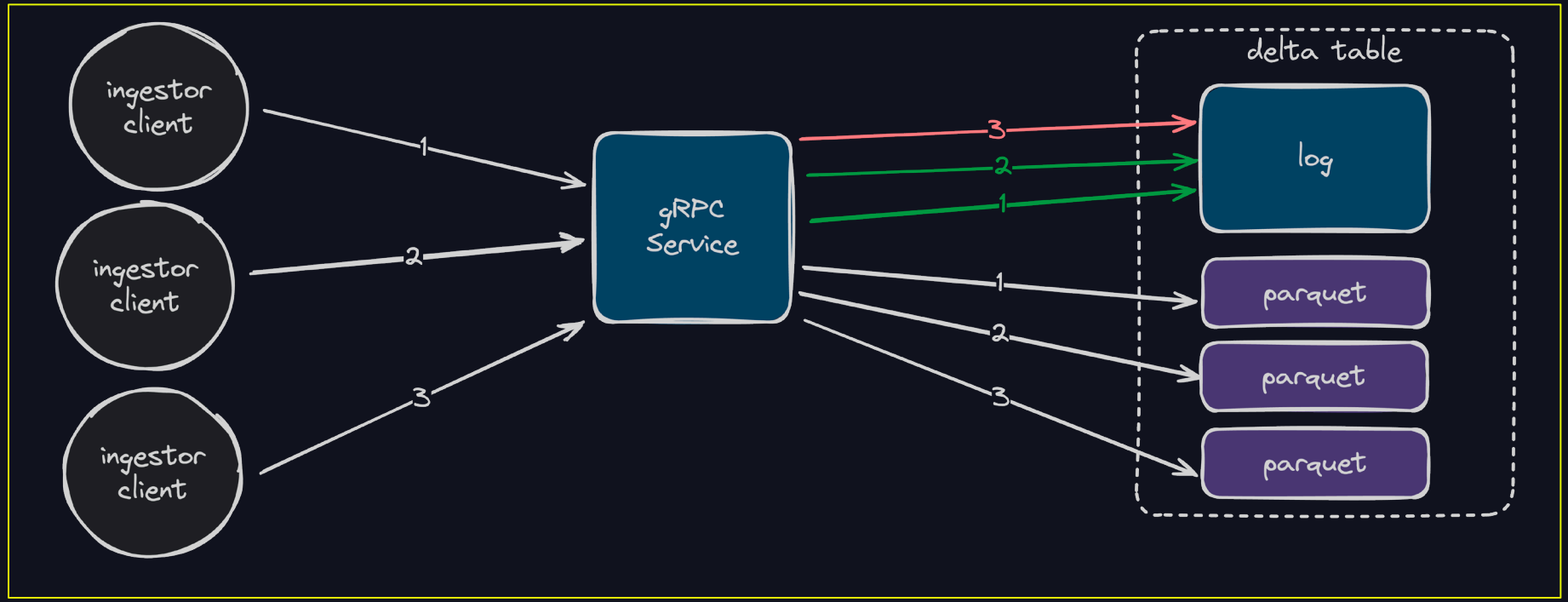
INGESTION CHALLENGES

Too much concurrency!



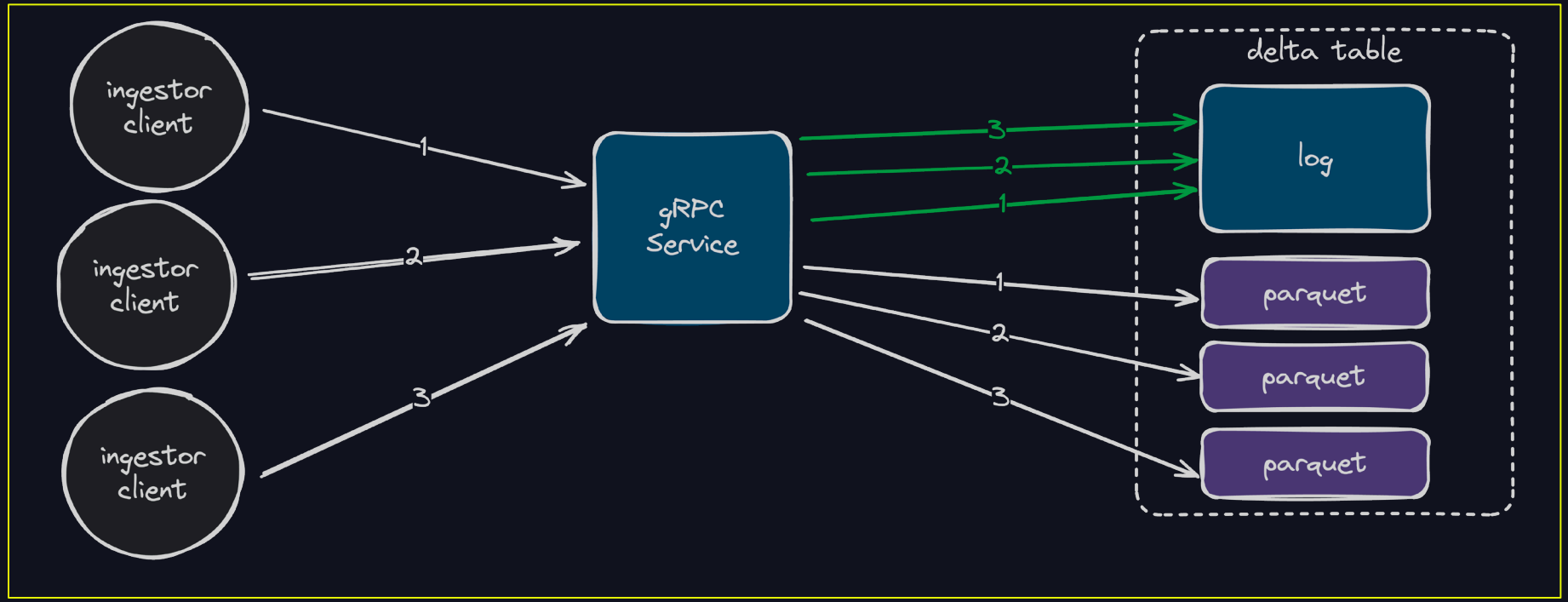
INGESTION CHALLENGES

Too much concurrency!



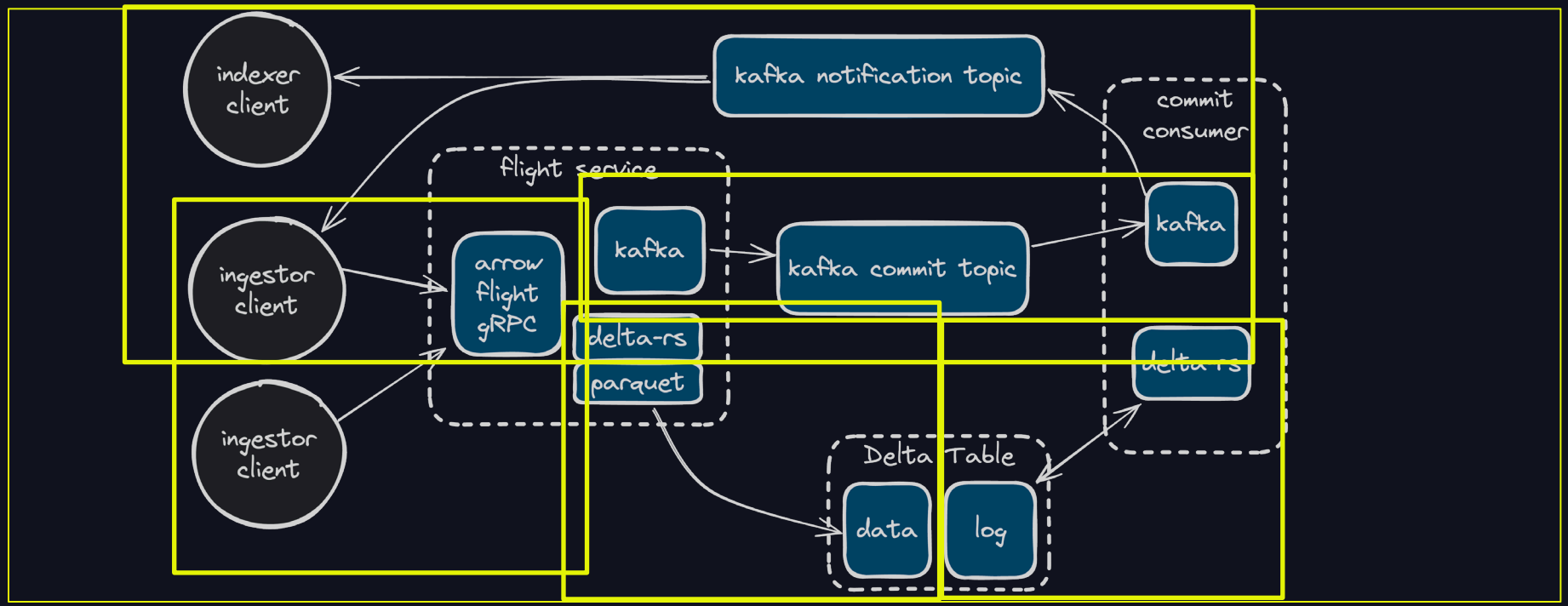
INGESTION CHALLENGES

Too much concurrency!



INGESTION SOLUTION

Supporting low latency scalable puts



HOW IS IT WORKING?

Analyzing early production metrics (April)

Percentile	Put Latency
50	90ms
95	132ms
99	204ms

Put Requests	3M
Failures	54
Rows Written	85M
Bytes Written	121GB

STRATEGIES AND OPTIMIZATIONS FOR DATA READS

HOW IS THE DATA USED?

Use cases for short message data after it's imported

Review

- Clients expect to be able to view batches of messages quickly
- They might also expect to be able to slice the messages in different ways

Annotations

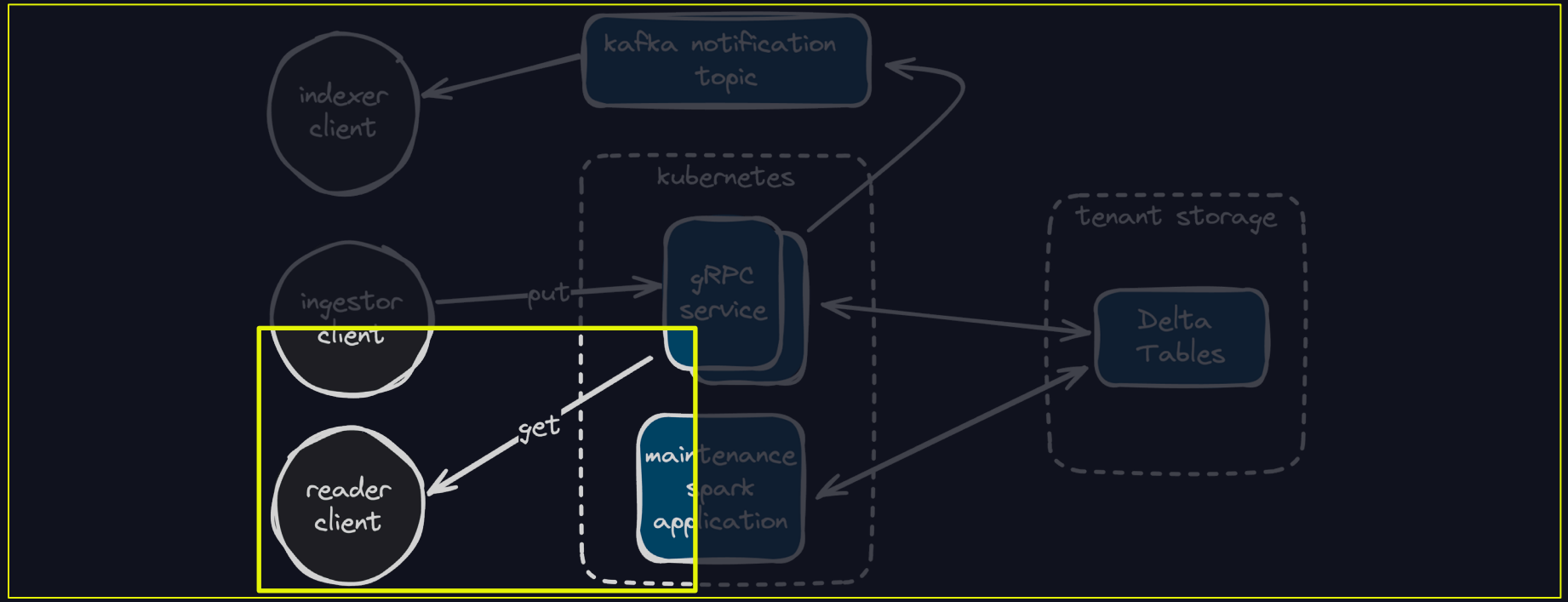
- Message annotations – such as notes and flags – will come in as messages are reviewed
- These annotations will be attached to individual messages

Search Indexing

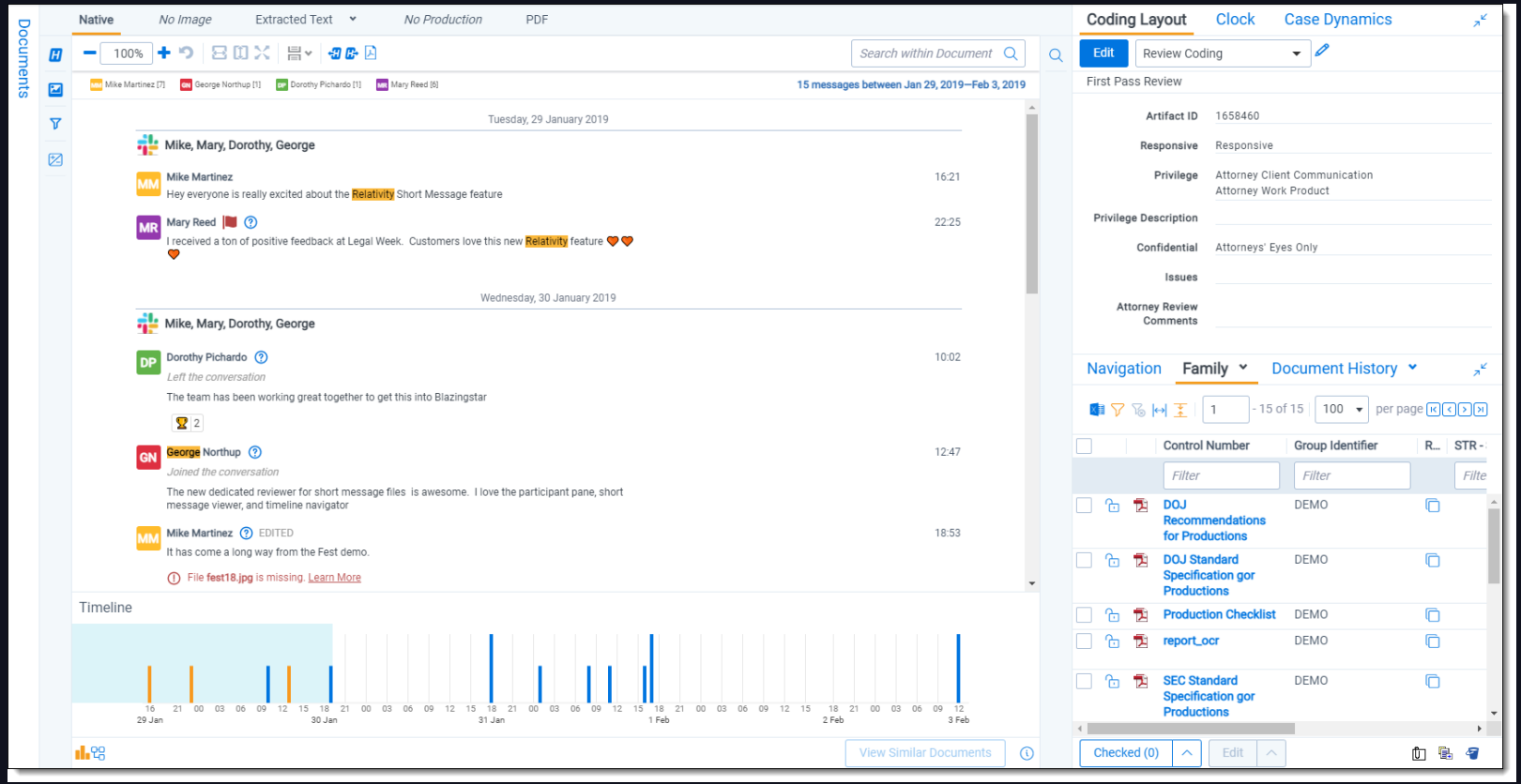
- Automated tools at Relativity will be retrieving messages in bulk for indexing
- Small updates will have to happen as messages are annotated

FOCUSING ON THE CLIENT WORKFLOW

Discussing optimizations to handle incoming annotations

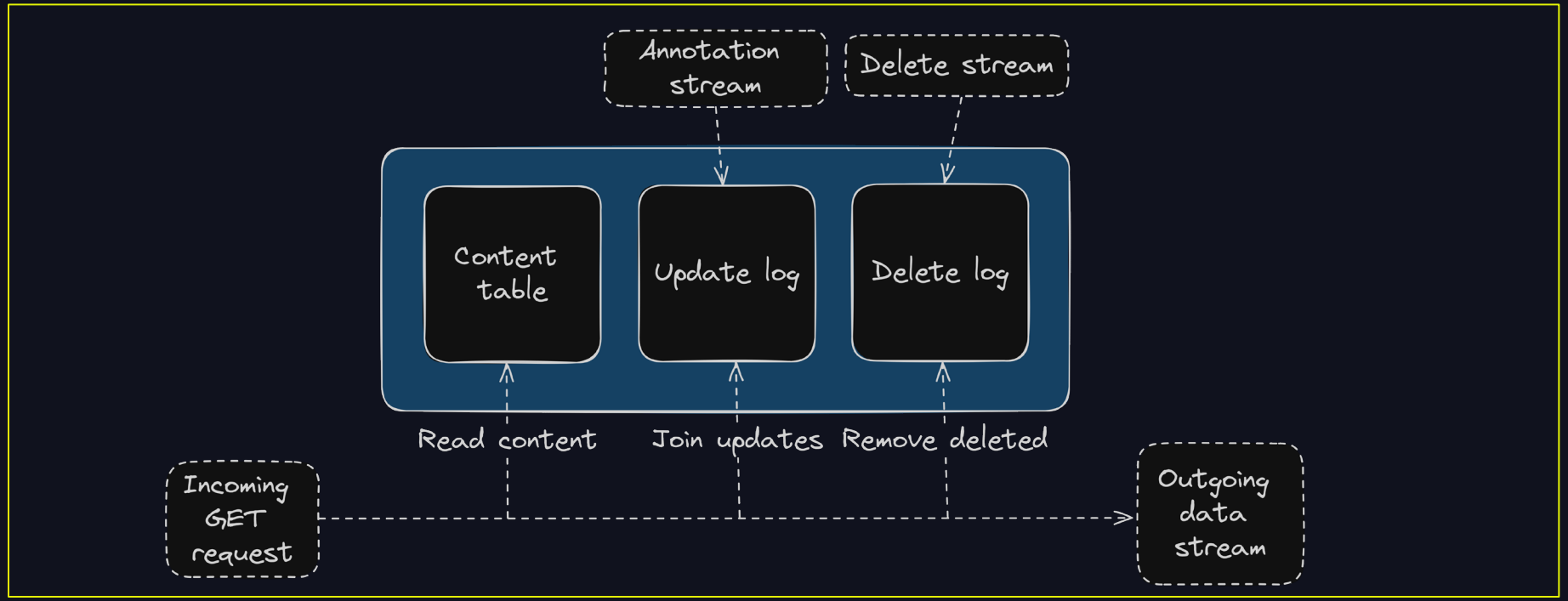


THE RELATIVITY REVIEW INTERFACE



OUR WRITE AHEAD LOG

The update and delete log tables allow for massively parallel updates



UPDATE LOG PERFORMANCE

The update log allows us to support thousands of concurrent annotations

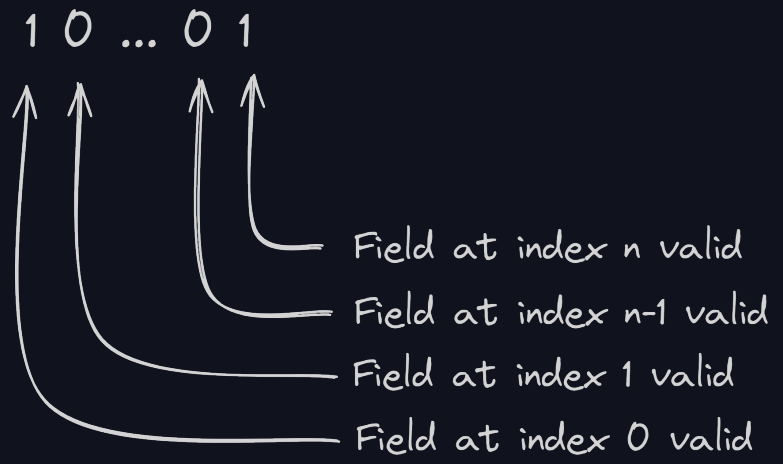
ID	Field	Value
1	Useful	False
1	Useful	True
1	Notes	Some notes
2	Useful	False
1	Notes	Some notes. Another sentence.
3	Notes	Another note
2	Notes	Yet another note

- Update log is append-only, avoids conflicts between 2 processes attempting to update the same table
- Query plan folds in updates based on most recent updates per field/message
- Query planning is very fast – peak read throughput is 150k rows per second

UPDATE LOG CONTINUED

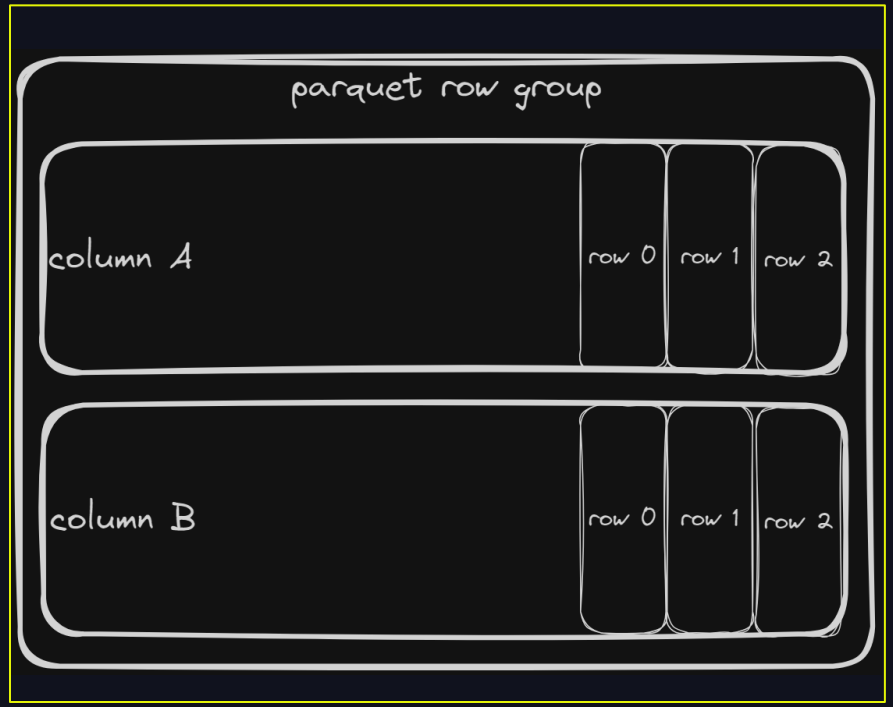
Bit maps for merging data

ID	Useful	Notes	Bit map
1	False		10
1	True		10
2		Some notes here	01
1	False	Some more notes	11



BYTE RANGE CACHING

Optimizations around caching reduce our data retrieval latency by 50%

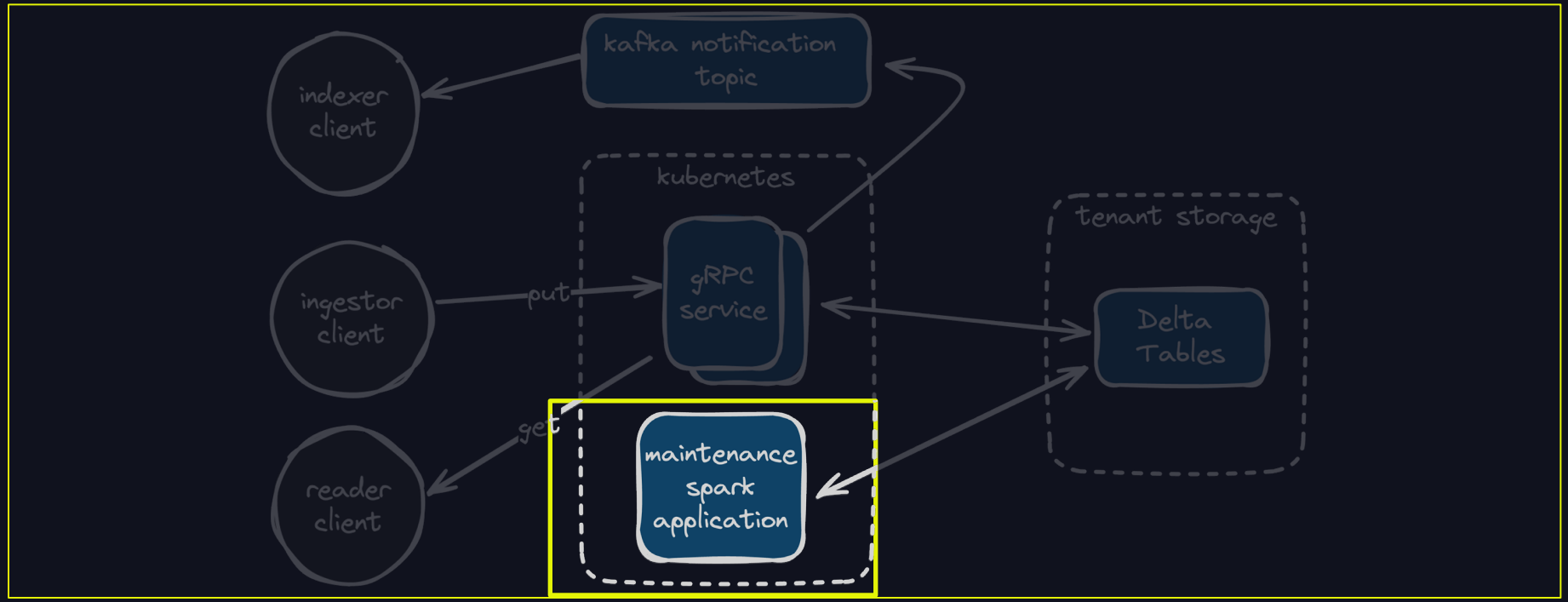


- We implemented a caching layer for parquet loads
- Caching accepts the column/byte range and parquet or checkpoint file path as a key
- Since parquet files and checkpoints are never altered once on disk, cache invalidation can be handled naturally as new cache entries evict old ones

DATA HYGIENE: SCHEDULED MAINTENANCE JOBS

A LOOK AT THE MAINTENANCE JOB

Our service relies on a background maintenance job to optimize



METADATA MAINTENANCE JOB

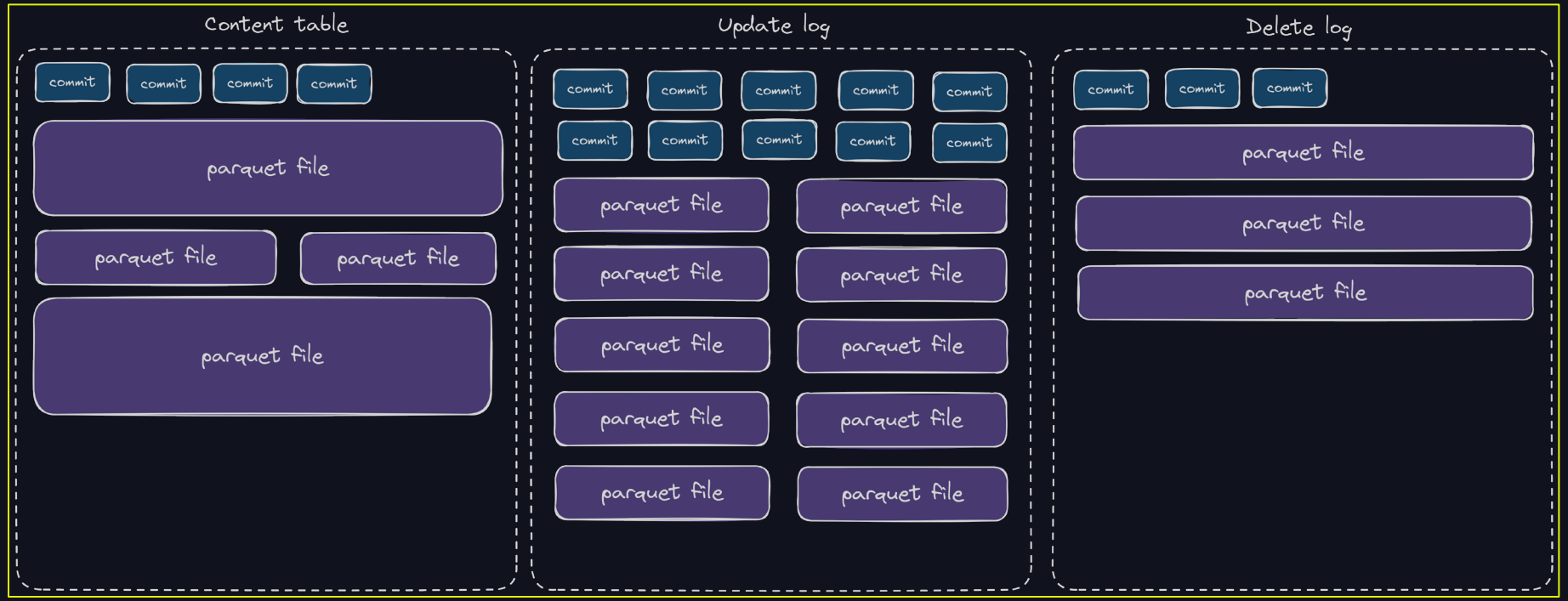
We use a Spark maintenance job to handle data hygiene

- Maintenance job first flushes the update and delete logs into the content table
- Next, it optimizes the tables. The content table is optimized with z-order.
- Next, old commits are cleared out. This is done automatically in Spark, unlike Rust.
- Finally, tables are vacuumed to remove unused parquet files

- We run a maintenance job in Spark on a schedule
- This maintenance job handles data cleanup: optimize, vacuum, update log flush
- Optimize reduces data volume by >90%

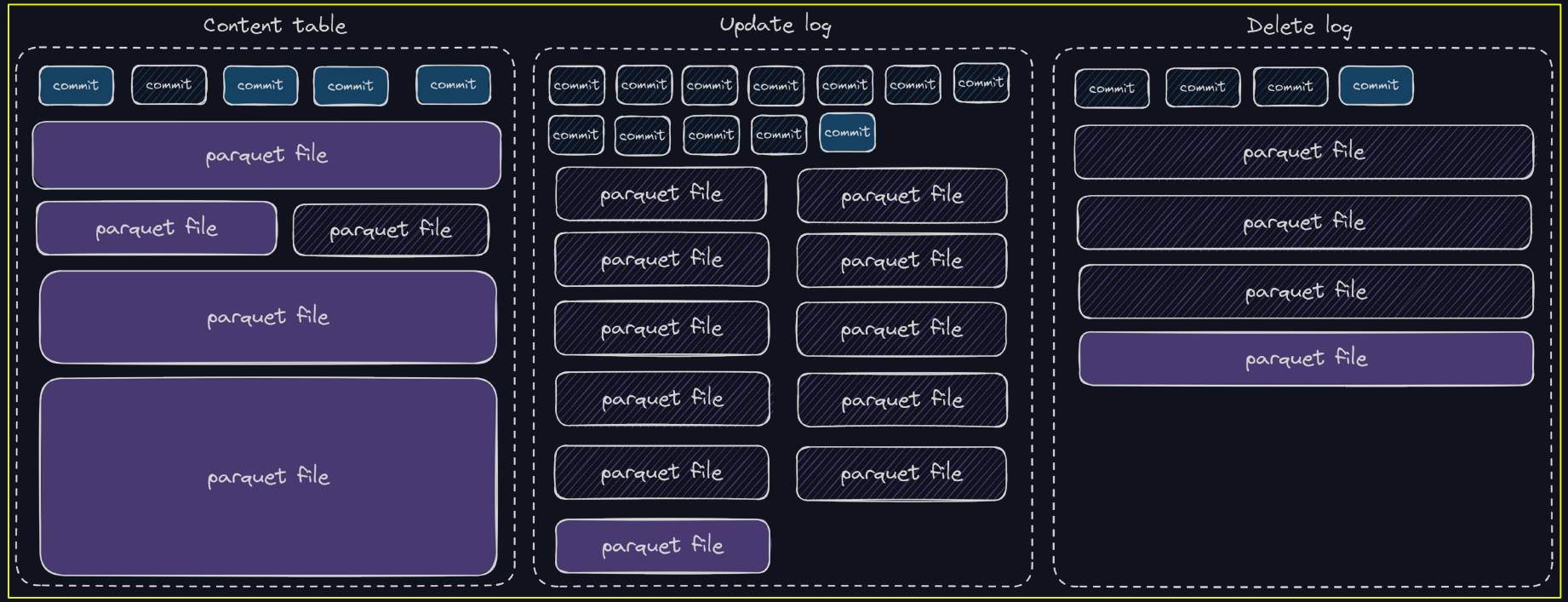
STARTING STATE OF THE TABLES

Many update commits, few content commits and fewer delete commits



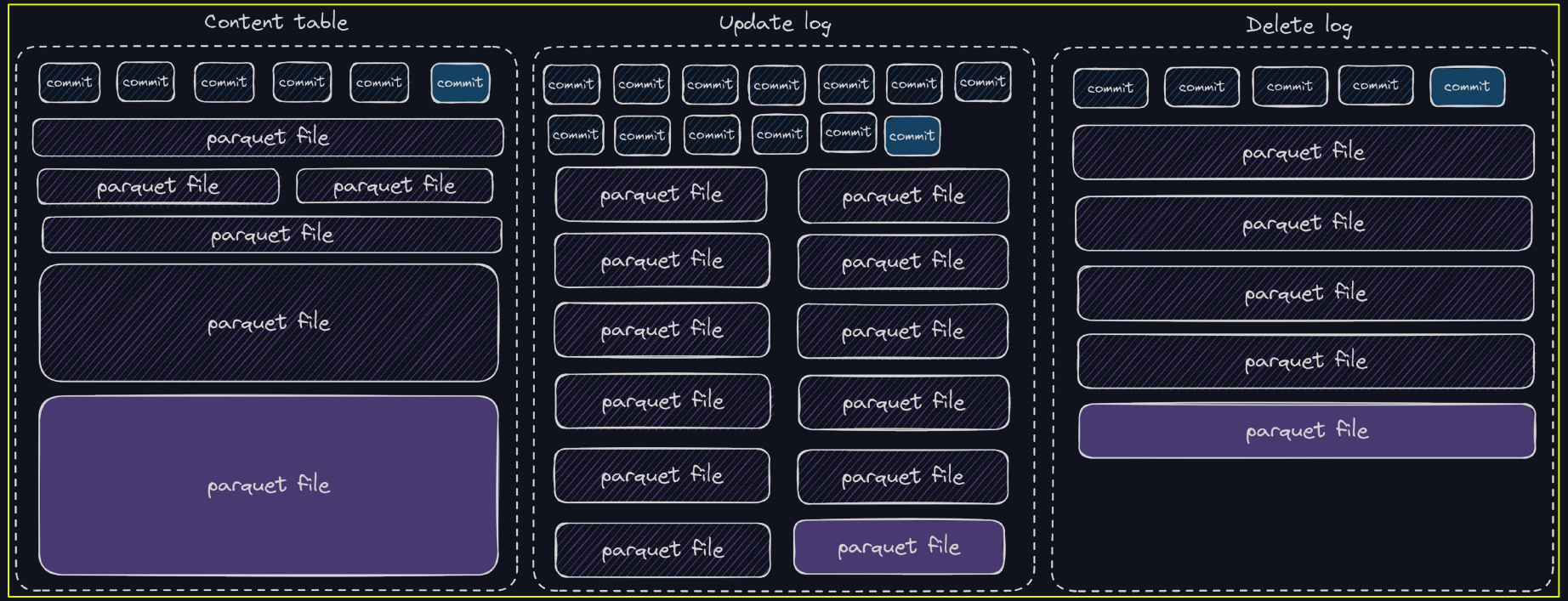
UPDATE AND DELETE LOG FLUSH

Some data in the content table is updated



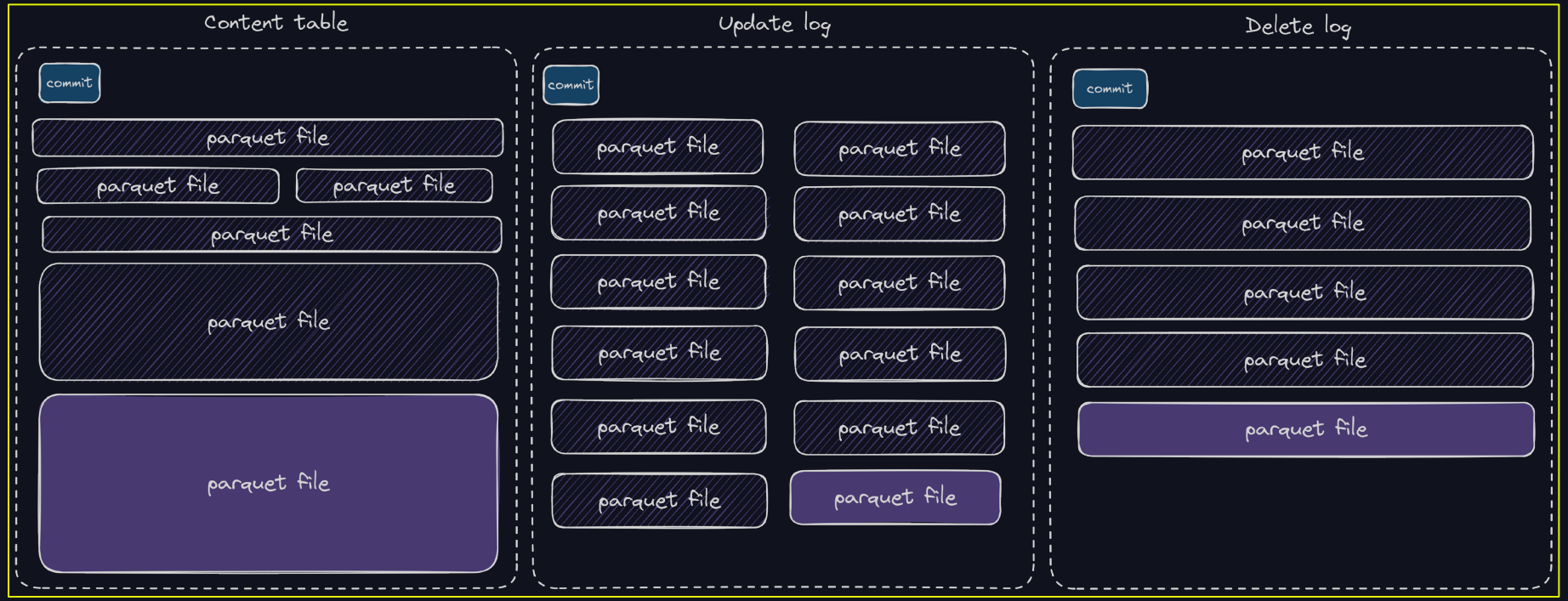
OPTIMIZED AND Z-ORDERED

One (or more) new commit and data file per table



OLD COMMITS CLEARED OUT

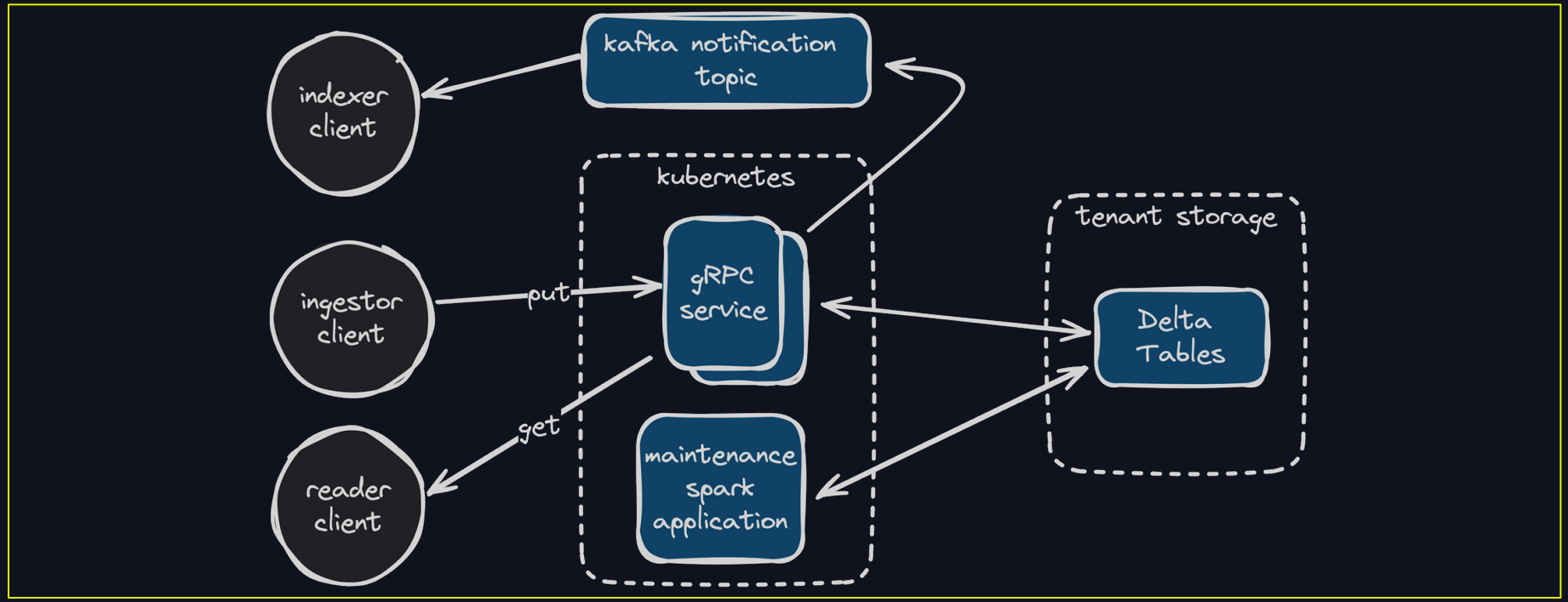
Parquet files remain



OLD DATA FILES CLEARED OUT

Tables are now returned to a clean state

A FINAL LOOK AT THE ARCHITECTURE



CLOSING REMARKS AND FUTURE POSSIBILITES

CLOSING REMARKS

- Excited about future of this emerging tech and ecosystem
- Looking forward to applying it to more data types and RelativityOne workflows
- Shout-out to the delta-rs, DataFusion, and arrow-rs maintainers and the various communities for their support