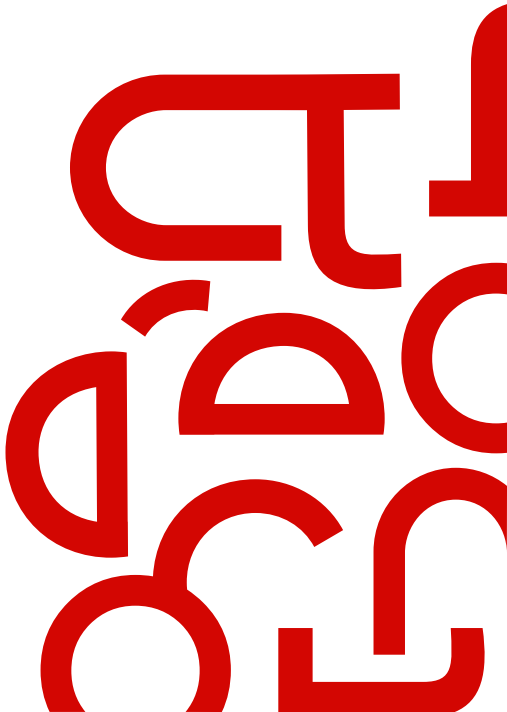
Building a Unified Feature Platform with DuckDB and Arrow

Michael Eastham
Chief Architect, Tecton AI

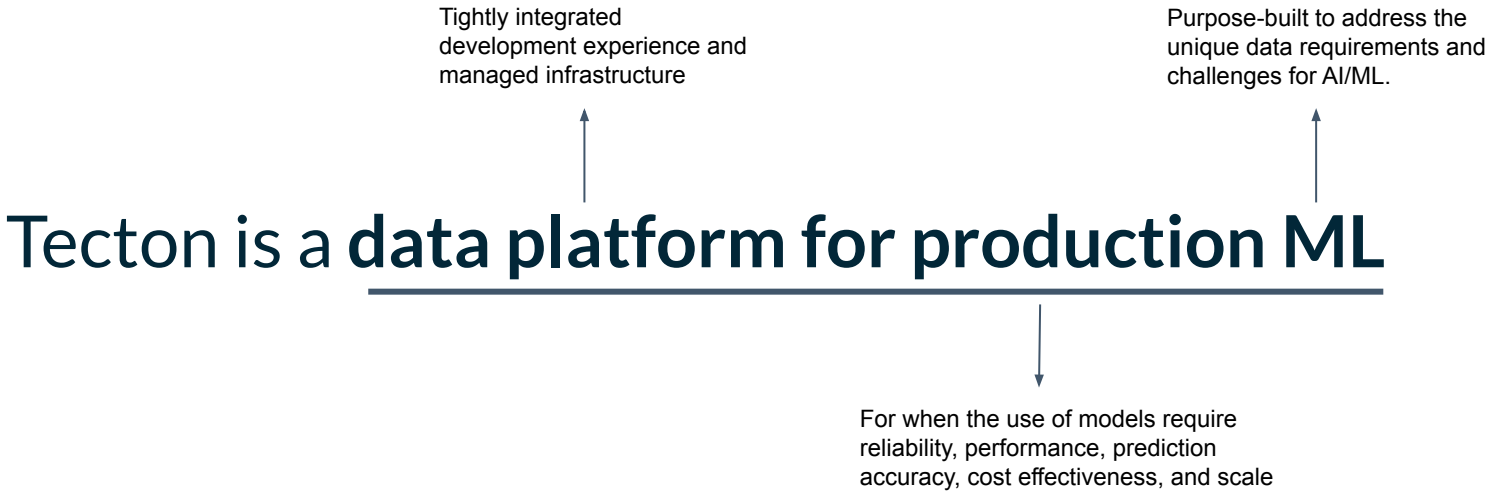
tecton



Who's Tecton?

PURPOSE	Activating data to power every experience with AI/ML
FOUNDERS	The creators of Uber's Michelangelo AI Platform • Created the first machine learning "feature store" • Scaled from 0 to 1000's of models in production
MATURITY	Founded 2018 • $160M raised
CUSTOMERS	Examples include: ATLASSIAN PLAID depop ROBLOX tide
INVESTORS	andreessen horowitz SEQUOIA KLEINER PERKINS

What's Tecton?

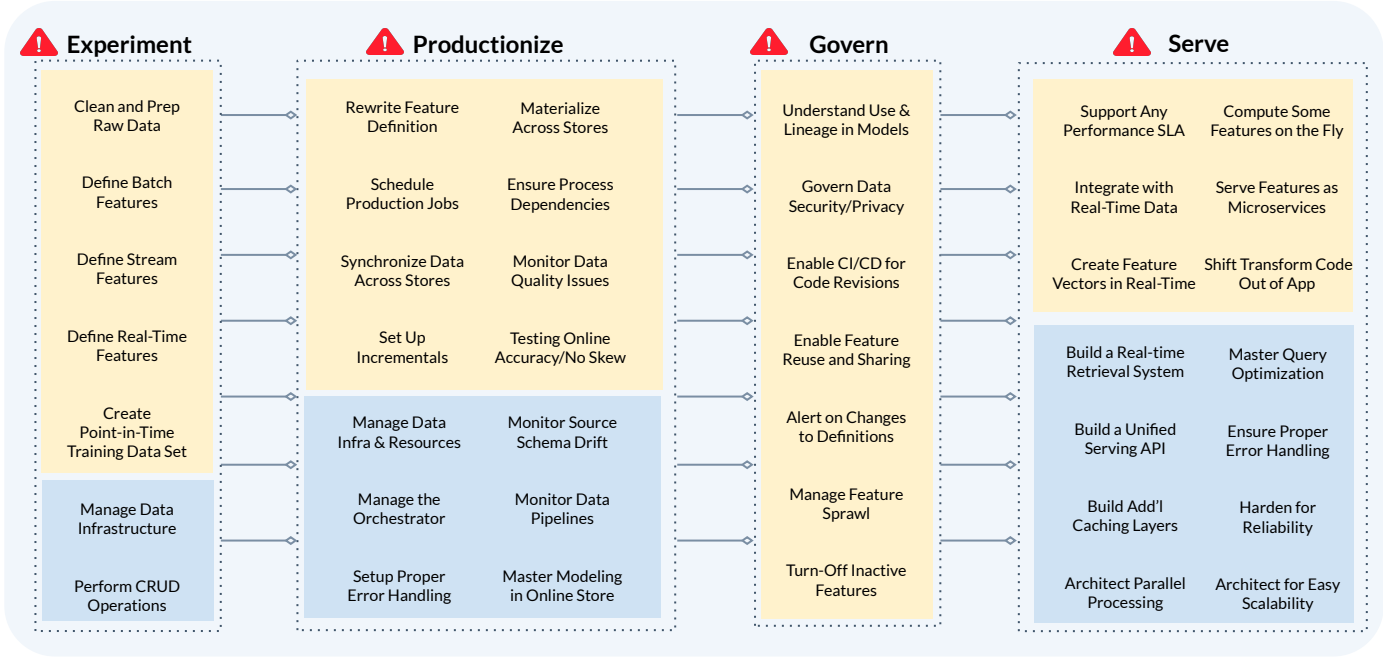


Why Tecton?

Production ML is a Massive Data Problem

The data journey from source-to-serve is consuming

Development-related
Infrastructure-related



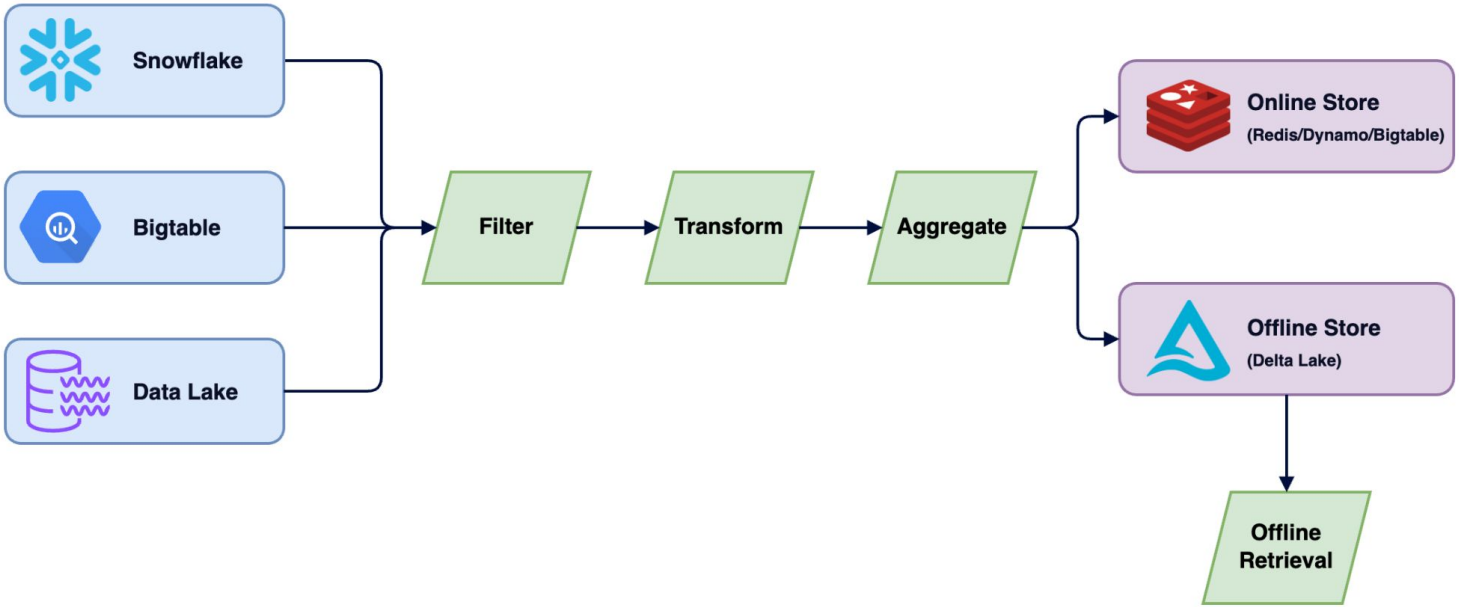
⚠ Slow to **build**, **maintain**, and **manage** ⚠ Manual, **error-prone** systems ⚠ Difficult **to scale**

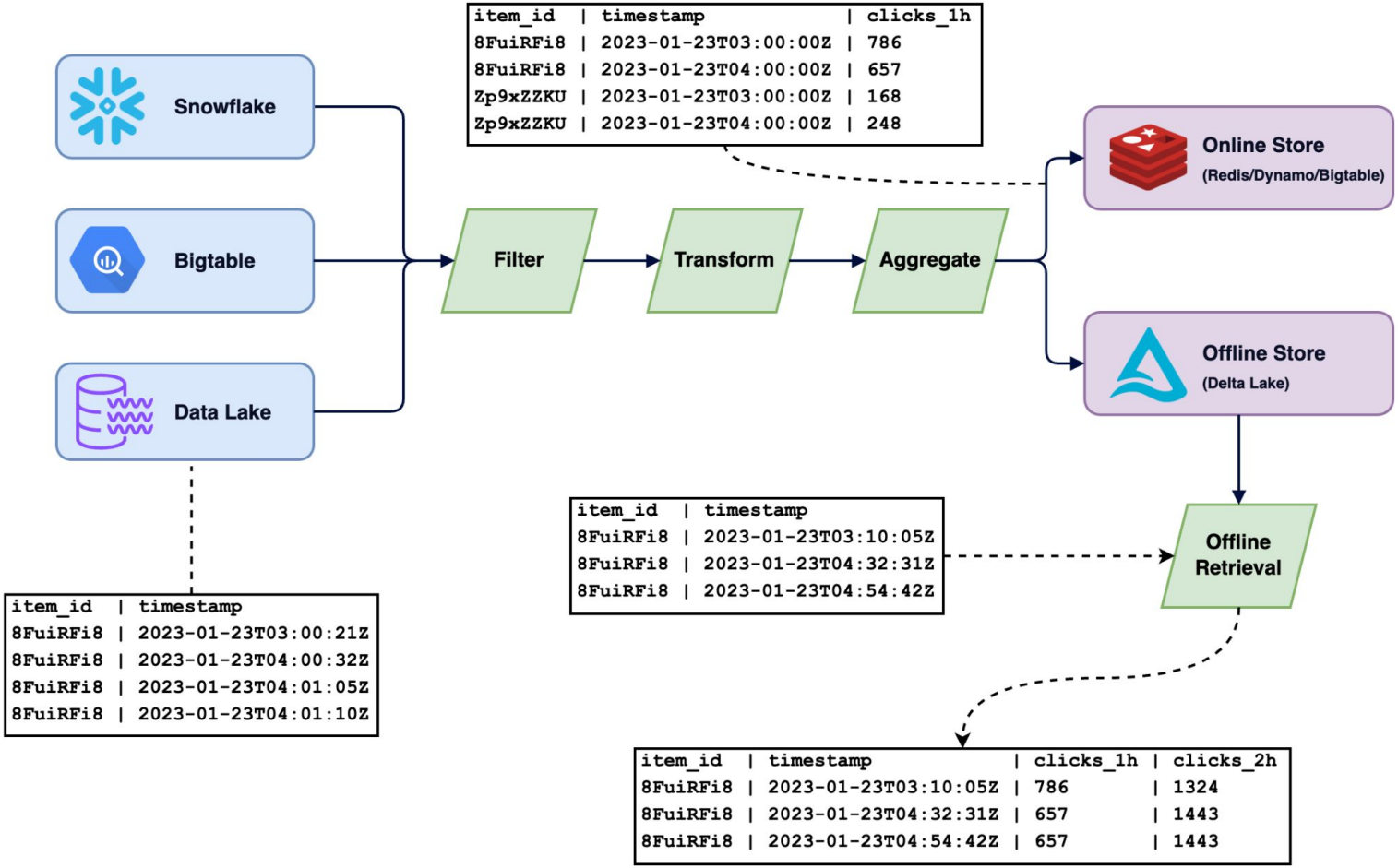
How?

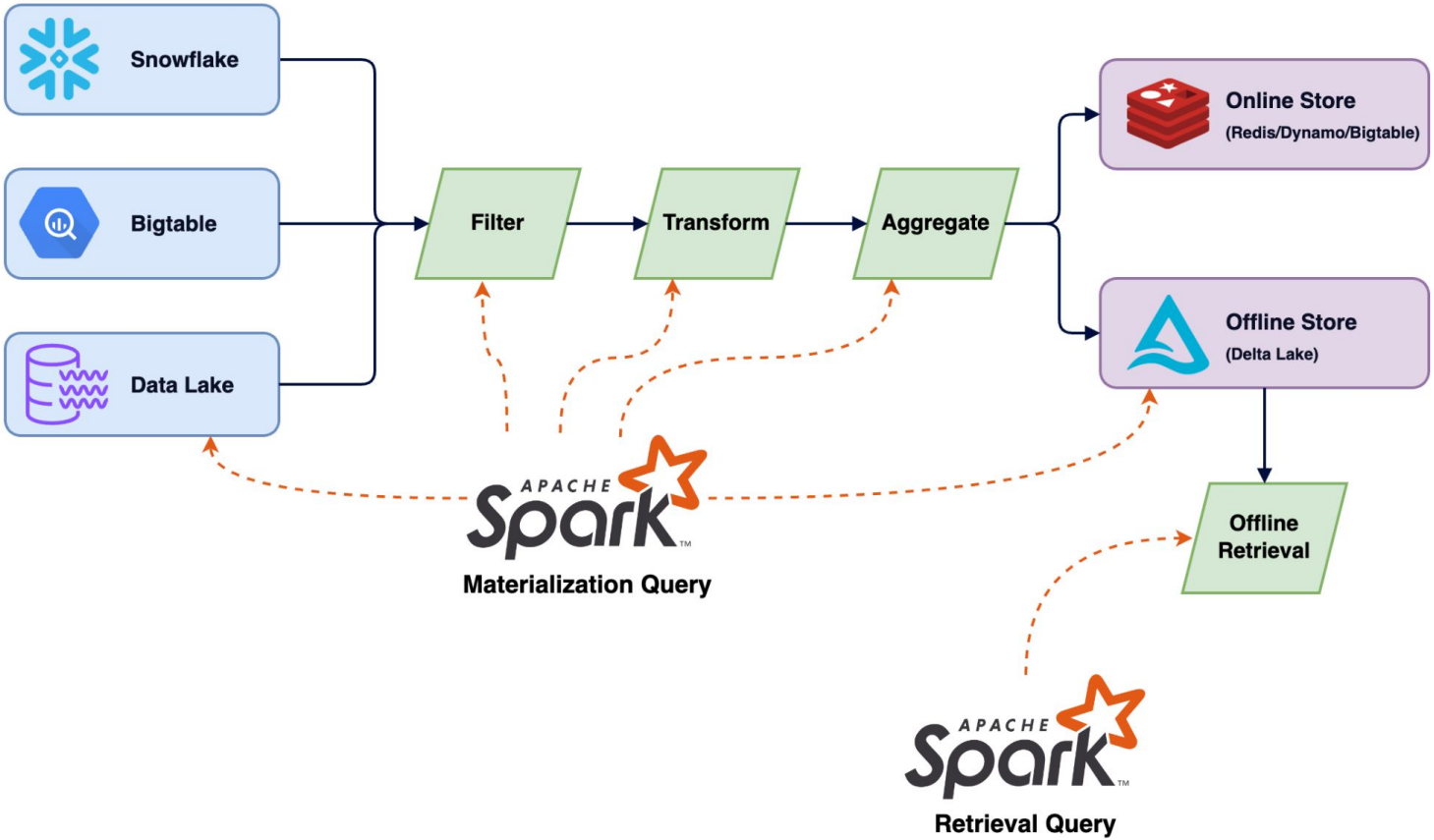
- Provide a declarative framework in which users (Data Scientists, Data Engineers, ML Engineers) define the transformation from their raw data into production-ready features.
- Orchestrate pipelines which implement these transformations on:
  - Historical Data (i.e. Backfills)
  - Recent Data
    - In Batches, or Streaming
- These pipelines populate storage (i.e. the Feature Store), which serves:
  - Offline Queries:
    - Run in batches, each query is for a given `(entity, timestamp)`
    - Optimized for throughput
    - Use Case: Training; Offline/Batch Inference
  - Online Queries:
    - Each query is for the most recent data for a given `entity`
    - Optimized for latency
    - Use Case: Online Inference

Why?

- A single, declarative Feature definition drives your whole ML system:
  - Backfilling, stream ingestion, re-ingesting data to recover from upstream errors, all use the same feature definition
  - The training and serving systems use the same feature definition
  - Details of pipeline management are abstracted away
  - This includes **experimentation**: user can develop with our framework iteratively in notebooks, so there is not a separate “productionisation” step once useful features are identified
- Uniform Feature definitions form the basis for monitoring, governance, lineage, feature sharing and reuse, etc. functionality.


Snowflake
Bigtable
Data Lake
Filter
Transform
Aggregate
Online Store
(Redis/Dynamo/Bigtable)
Offline Store
(Delta Lake)
Offline
Retrieval


Snowflake
Bigtable
Data Lake
Filter
Transform
Aggregate
Online Store
(Redis/Dynamo/Bigtable)
Offline Store
(Delta Lake)
Offline Retrieval
item_id | timestamp | clicks_1h
8FuiRFi8 | 2023-01-23T03:00:00Z | 786
8FuiRFi8 | 2023-01-23T04:00:00Z | 657
Zp9xZZKU | 2023-01-23T03:00:00Z | 168
Zp9xZZKU | 2023-01-23T04:00:00Z | 248
item_id | timestamp
8FuiRFi8 | 2023-01-23T03:00:21Z
8FuiRFi8 | 2023-01-23T04:00:32Z
8FuiRFi8 | 2023-01-23T04:01:05Z
8FuiRFi8 | 2023-01-23T04:01:10Z
item_id | timestamp
8FuiRFi8 | 2023-01-23T03:10:05Z
8FuiRFi8 | 2023-01-23T04:32:31Z
8FuiRFi8 | 2023-01-23T04:54:42Z
item_id | timestamp | clicks_1h | clicks_2h
8FuiRFi8 | 2023-01-23T03:10:05Z | 786 | 1324
8FuiRFi8 | 2023-01-23T04:32:31Z | 657 | 1443
8FuiRFi8 | 2023-01-23T04:54:42Z | 657 | 1443


Snowflake
Bigtable
Data Lake
Filter
Transform
Aggregate
Online Store
(Redis/Dynamo/Bigtable)
Offline Store
(Delta Lake)
Offline
Retrieval
APACHE
Spark
Materialization Query
APACHE
Spark
Retrieval Query

Spark

- Customers write pyspark/Spark SQL transformations themselves
- We combine these transformations with our own query library which handles time filtering, incremental aggregations, and writing into the Feature Store
- These combined jobs are deployed in the customer's Spark environment (Databricks/EMR/Dataproc). Customers control the configuration of these jobs, including cluster size.

Problem: Spark is Confusing

What does this error mean?

Source: https://stackoverflow.com/questions/75522599/what-is-the-cause-of-this-error-code-was-working-before-and-randomly-started-to

```
Py4JJavaError: An error occurred while calling o137.showString.
: java.util.concurrent.ExecutionException: Boxed Error
    at scala.concurrent.impl.Promise$.resolver(Promise.scala:83)
    at scala.concurrent.impl.Promise$.scala$concurrent$impl$Promise$$resolveTry(Promise.scala:75)
    at scala.concurrent.impl.Promise$KeptPromise$.apply(Promise.scala:402)
    at scala.concurrent.Promise$.fromTry(Promise.scala:138)
    at scala.concurrent.Promise$.failed(Promise.scala:124)
    at scala.concurrent.Future$.failed(Future.scala:619)
    at org.apache.spark.sql.execution.exchange.ShuffleExchangeLike.$anonfun$materializeFuture$1(ShuffleExchangeExec.scala:104)
    at org.apache.spark.sql.util.LazyValue.getOrInit(LazyValue.scala:41)
    at org.apache.spark.sql.execution.exchange.Exchange.getOrInitMaterializeFuture(Exchange.scala:68)
    at org.apache.spark.sql.execution.exchange.ShuffleExchangeLike.materializeFuture(ShuffleExchangeExec.scala:96)
    at org.apache.spark.sql.execution.exchange.ShuffleExchangeLike.materialize(ShuffleExchangeExec.scala:84)
    at org.apache.spark.sql.execution.exchange.ShuffleExchangeLike.materialize$(ShuffleExchangeExec.scala:83)
    at org.apache.spark.sql.execution.exchange.ShuffleExchangeExec.materialize(ShuffleExchangeExec.scala:128)
    at org.apache.spark.sql.execution.adaptive.ShuffleQueryStageExec.doMaterialize(QueryStageExec.scala:161)
    at org.apache.spark.sql.execution.adaptive.QueryStageExec.$anonfun$materialize$1(QueryStageExec.scala:74)
    at org.apache.spark.sql.execution.SparkPlan.$anonfun$executeQuery$1(SparkPlan.scala:223)
    at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
    at org.apache.spark.sql.execution.SparkPlan.executeQuery(SparkPlan.scala:220)
    at org.apache.spark.sql.execution.adaptive.QueryStageExec.materialize(QueryStageExec.scala:74)
    at org.apache.spark.sql.execution.adaptive.MaterializeExecutable.tryStart(AdaptiveExecutable.scala:396)
    at org.apache.spark.sql.execution.adaptive.AdaptiveExecutorRuntime.startChild(AdaptiveExecutor.scala:225)
    at org.apache.spark.sql.execution.adaptive.ExecutionHelper.start(ExecutionHelper.scala:47)
    at org.apache.spark.sql.execution.adaptive.QueryStageExecutable$$anon$2.$anonfun$new$1(AdaptiveExecutable.scala:251)
    at org.apache.spark.sql.execution.adaptive.ExecutionHelper$Listener.$anonfun$onChildSuccess$2(ExecutionHelper.scala:55)
    at org.apache.spark.sql.execution.adaptive.ExecutionHelper$Listener.$anonfun$onChildSuccess$2$adapted(ExecutionHelper.scala:54)
    at scala.Option.foreach(Option.scala:257)
    at org.apache.spark.sql.execution.adaptive.ExecutionHelper$Listener.$anonfun$onChildSuccess$1(ExecutionHelper.scala:54)
    at org.apache.spark.sql.execution.adaptive.ExecutionHelper$Listener.$anonfun$onChildSuccess$1$adapted(ExecutionHelper.scala:53)
    at scala.collection.mutable.ResizableArray.foreach(ResizableArray.scala:58)
    at scala.collection.mutable.ResizableArray.foreach$(ResizableArray.scala:51)
    at scala.collection.mutable.ArrayBuffer.foreach(ArrayBuffer.scala:47)
    at org.apache.spark.sql.execution.adaptive.ExecutionHelper$Listener.onChildSuccess(ExecutionHelper.scala:53)
    at org.apache.spark.sql.execution.adaptive.AdaptiveExecutorRuntime.$anonfun$onActiveChildSuccess$2(AdaptiveExecutor.scala:314)
    at org.apache.spark.sql.execution.adaptive.AdaptiveExecutorRuntime.$anonfun$onActiveChildSuccess$2$adapted(AdaptiveExecutor.scala:314)
    at scala.collection.mutable.ResizableArray.foreach(ResizableArray.scala:58)
    at scala.collection.mutable.ResizableArray.foreach$(ResizableArray.scala:51)
    at scala.collection.mutable.ArrayBuffer.foreach(ArrayBuffer.scala:47)
    at org.apache.spark.sql.execution.adaptive.AdaptiveExecutorRuntime.onActiveChildSuccess(AdaptiveExecutor.scala:314)
```

```
Caused by: java.lang.NoSuchMethodError: org.apache.sedona.core.utils.SedonaConf.<init>(Lorg/apache/spark/SparkConf;)V
    at org.apache.spark.sql.sedona_sql.strategy.join.TraitJoinQueryExec.doExecute(TraitJoinQueryExec.scala:63)
    at org.apache.spark.sql.sedona_sql.strategy.join.TraitJoinQueryExec.doExecute$(TraitJoinQueryExec.scala:56)
    at org.apache.spark.sql.sedona_sql.strategy.join.RangeJoinExec.doExecute(RangeJoinExec.scala:37)
    at org.apache.spark.sql.execution.SparkPlan.$anonfun$execute$1(SparkPlan.scala:185)
    at org.apache.spark.sql.execution.SparkPlan.$anonfun$executeQuery$1(SparkPlan.scala:223)
    at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
    at org.apache.spark.sql.execution.SparkPlan.executeQuery(SparkPlan.scala:220)
    at org.apache.spark.sql.execution.SparkPlan.execute(SparkPlan.scala:181)
    at org.apache.spark.sql.execution.InputAdapter.inputRDD(WholeStageCodegenExec.scala:525)
    at org.apache.spark.sql.execution.InputRDDCodegen.inputRDDs(WholeStageCodegenExec.scala:453)
    at org.apache.spark.sql.execution.InputRDDCodegen.inputRDDs$(WholeStageCodegenExec.scala:452)
    at org.apache.spark.sql.execution.InputAdapter.inputRDDs(WholeStageCodegenExec.scala:496)
    at org.apache.spark.sql.execution.joins.BroadcastHashJoinExec.inputRDDs(BroadcastHashJoinExec.scala:178)
    at org.apache.spark.sql.execution.WholeStageCodegenExec.doExecute(WholeStageCodegenExec.scala:746)
    at org.apache.spark.sql.execution.SparkPlan.$anonfun$execute$1(SparkPlan.scala:185)
    at org.apache.spark.sql.execution.SparkPlan.$anonfun$executeQuery$1(SparkPlan.scala:223)
    at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
    at org.apache.spark.sql.execution.SparkPlan.executeQuery(SparkPlan.scala:220)
    at org.apache.spark.sql.execution.SparkPlan.execute(SparkPlan.scala:181)
    at org.apache.spark.sql.execution.ProjectExec.doExecute(basicPhysicalOperators.scala:92)
    at org.apache.spark.sql.execution.SparkPlan.$anonfun$execute$1(SparkPlan.scala:185)
    at org.apache.spark.sql.execution.SparkPlan.$anonfun$executeQuery$1(SparkPlan.scala:223)
    at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
    at org.apache.spark.sql.execution.SparkPlan.executeQuery(SparkPlan.scala:220)
    at org.apache.spark.sql.execution.SparkPlan.execute(SparkPlan.scala:181)
    at org.apache.spark.sql.execution.InputAdapter.inputRDD(WholeStageCodegenExec.scala:525)
    at org.apache.spark.sql.execution.InputRDDCodegen.inputRDDs(WholeStageCodegenExec.scala:453)
    at org.apache.spark.sql.execution.InputRDDCodegen.inputRDDs$(WholeStageCodegenExec.scala:452)
    at org.apache.spark.sql.execution.InputAdapter.inputRDDs(WholeStageCodegenExec.scala:496)
    at org.apache.spark.sql.execution.aggregate.HashAggregateExec.inputRDDs(HashAggregateExec.scala:137)
    at org.apache.spark.sql.execution.WholeStageCodegenExec.doExecute(WholeStageCodegenExec.scala:746)
    at org.apache.spark.sql.execution.SparkPlan.$anonfun$execute$1(SparkPlan.scala:185)
    at org.apache.spark.sql.execution.SparkPlan.$anonfun$executeQuery$1(SparkPlan.scala:223)
    at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
    at org.apache.spark.sql.execution.SparkPlan.executeQuery(SparkPlan.scala:220)
    at org.apache.spark.sql.execution.SparkPlan.execute(SparkPlan.scala:181)
    at org.apache.spark.sql.execution.exchange.ShuffleExchangeExec.inputRDD$lzycompute(ShuffleExchangeExec.scala:160)
    at org.apache.spark.sql.execution.exchange.ShuffleExchangeExec.inputRDD(ShuffleExchangeExec.scala:160)
    at org.apache.spark.sql.execution.exchange.ShuffleExchangeExec.mapOutputStatisticsFuture$lzycompute(ShuffleExchangeExec.scala:164)
    at org.apache.spark.sql.execution.exchange.ShuffleExchangeExec.mapOutputStatisticsFuture(ShuffleExchangeExec.scala:163)
    at org.apache.spark.sql.execution.exchange.ShuffleExchangeLike.$anonfun$materializeFuture$2(ShuffleExchangeExec.scala:100)
    at org.apache.spark.sql.catalyst.errors.package$.attachTree(package.scala:52)
    at org.apache.spark.sql.execution.exchange.ShuffleExchangeLike.$anonfun$materializeFuture$1(ShuffleExchangeExec.scala:100)
    ... 76 more
```

ChatGPT

The error message indicates that a `NoSuchMethodError` occurred, specifically referencing the constructor of the `SedonaConf` class from the `org.apache.sedona.core.utils` package. This error typically happens when there's a mismatch between the compiled code and the runtime environment, often due to different versions of libraries being used.

We want to help our users when they run into problems, but the surface area of Spark failures makes this really difficult

Spark Configuration

- Spark Properties
  - Dynamically Loading Spark Properties
  - Viewing Spark Properties
  - Available Properties
    - Application Properties
    - Runtime Environment
    - Shuffle Behavior
    - Spark UI
    - Compression and Serialization
    - Memory Management
    - Execution Behavior
    - Executor Metrics
    - Networking
    - Scheduling
    - Barrier Execution Mode
    - Dynamic Allocation
    - Thread Configurations
    - Spark Connect
      - Server Configuration
    - Security
    - Spark SQL
      - Runtime SQL Configuration
      - Static SQL Configuration
    - Spark Streaming
    - SparkR
    - GraphX
    - Deploy
    - Cluster Managers
      - YARN
      - Mesos
      - Kubernetes

Source: https://spark.apache.org/docs/latest/configuration.html

Problem: Spark complicates deployments

- We require a managed Spark environment (Databricks/EMR/DataProc) to use our product
- When a customer isn't an existing user of Spark, this often involves pulling in additional parties that have to be consulted as part of the sales process

Problem: Spark is heavyweight

- Setting up Spark to run on your laptop is possible, but cumbersome and prone to behavior differences to “production.”
- Spinning up a fresh Spark cluster to try something out can take 10+ minutes
- => Not a rewarding environment for experimentation and iteration

Spark is often not the right tool for the problems our users are trying to solve

When Spark is not a good fit for the use case, it degrades Tecton's value proposition:

- The experimentation experience is not smooth
- The pipelines are managed, but the area of shared responsibility is awkward. Customers might still have to learn a lot about configuring Spark

Source: https://motherduck.com/blog/big-data-is-dead/

Can we do better?

Product Priorities

- Enable domain experts to do their feature engineering using tools that they're familiar with, which is often SQL or Pandas.
- Polished integrations with Data Warehouses
- Iteration and Experimentation experience that's 10x better than what we previously had with Spark

Requirements: Devex

- Customers should have flexibility to express their transformations in the language/framework they want to use.
- Often times this means Pandas or SQL
- Might also want to support things like Polars or Ibis
- It should be easy for us to add support for different options, without sacrificing performance.

Requirements: Local First

- We want a delightful local development experience. Usually this is within a notebook running on a laptop or a service like Collab, Deepnote, or Hex.
- It should be fast to start up
- It should not require complex dependencies which can't easily be distributed through pip (such as a JVM and Hadoop JARs)
- It must be able to process non-trivial queries with constrained resources.

Requirements: Easy to Deploy

- No *mandatory* third party vendors like we have with Spark
- Be extremely frugal with configuration options

Requirements: Data Sources

We must be able to integrate with a wide variety of Data Sources

- Snowflake
- BigQuery
- Data Lakes
- Local Files
- Down the road: RedShift, Postgres…

Building Blocks: Apache Arrow

What is Arrow?

Format

Apache Arrow defines a language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware like CPUs and GPUs. The Arrow memory format also supports zero-copy reads for lightning-fast data access without serialization overhead.

Learn more about the design or read the specification.

Libraries

Arrow's libraries implement the format and provide building blocks for a range of use cases, including high performance analytics. Many popular projects use Arrow to ship columnar data efficiently or as the basis for analytic engines.

Libraries are available for C, C++, C#, Go, Java, JavaScript, Julia, MATLAB, Python, R, Ruby, and Rust. See how to install and get started.

Ecosystem

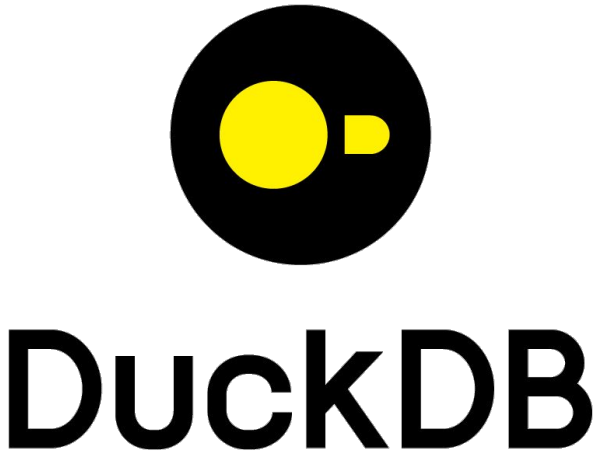
Apache Arrow is software created by and for the developer community. We are dedicated to open, kind communication and consensus decisionmaking. Our committers come from a range of organizations and backgrounds, and we welcome all to participate with us.

Learn more about how you can ask questions and get involved in the Arrow project.

Source: https://arrow.apache.org/

Building Blocks: DuckDB

- “DuckDB is a fast in-process analytical database”
- Or, the SQLite of analytical queries



Why DuckDB

Arrow

Seamless integration allows us to flexibly integrate new data sources, transformation modes, etc.

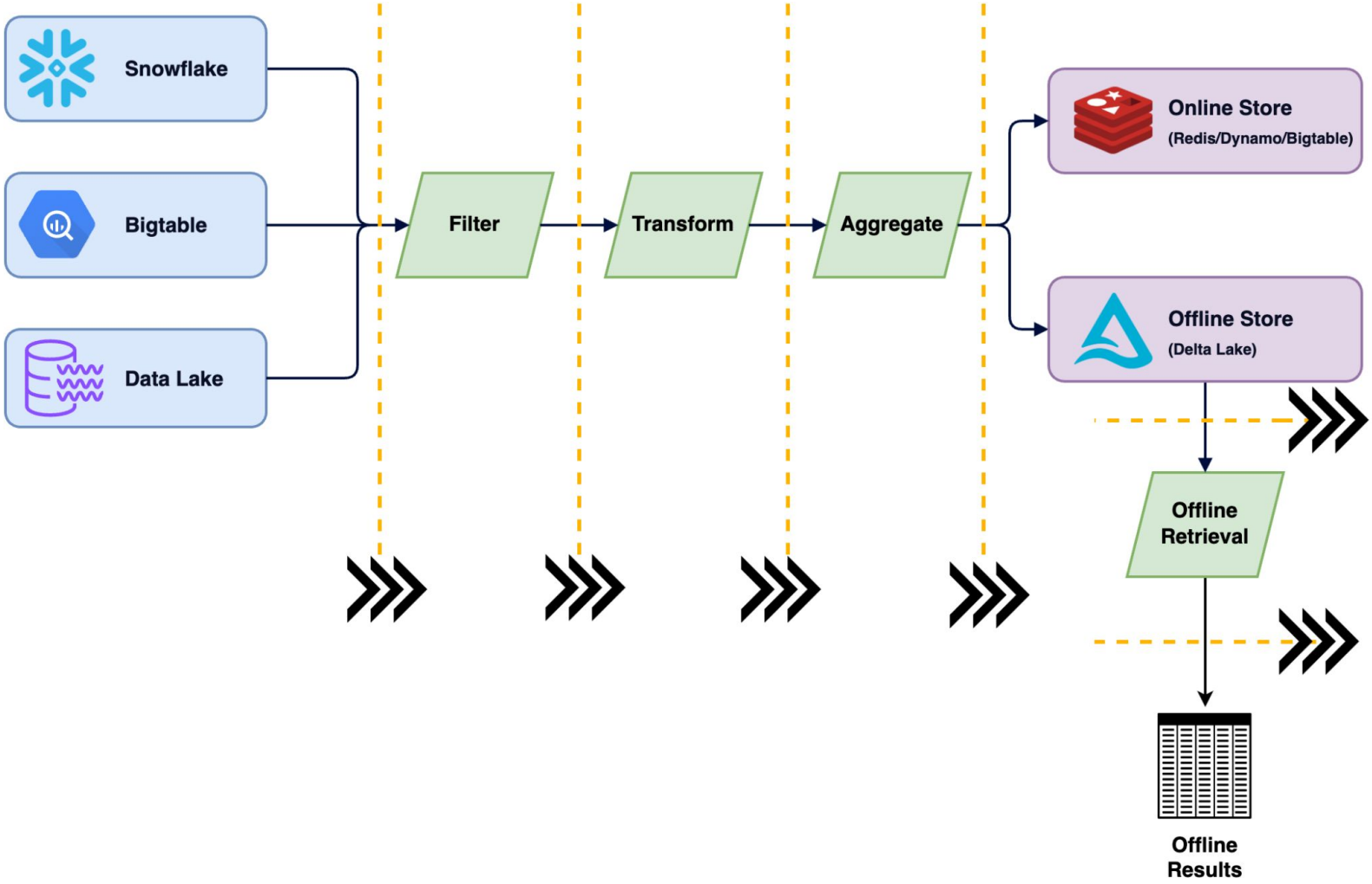
Performance

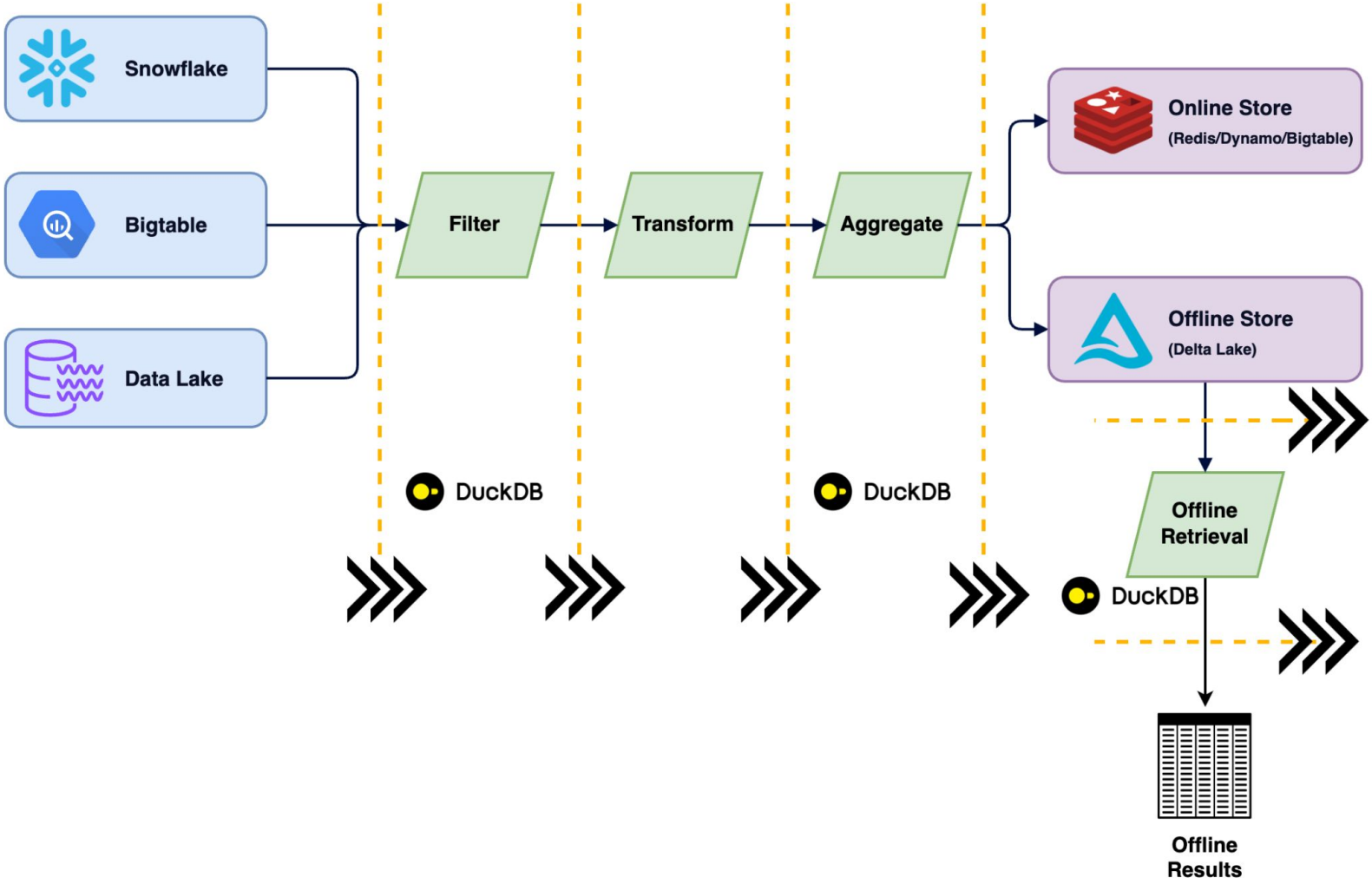
Fast & Lightweight

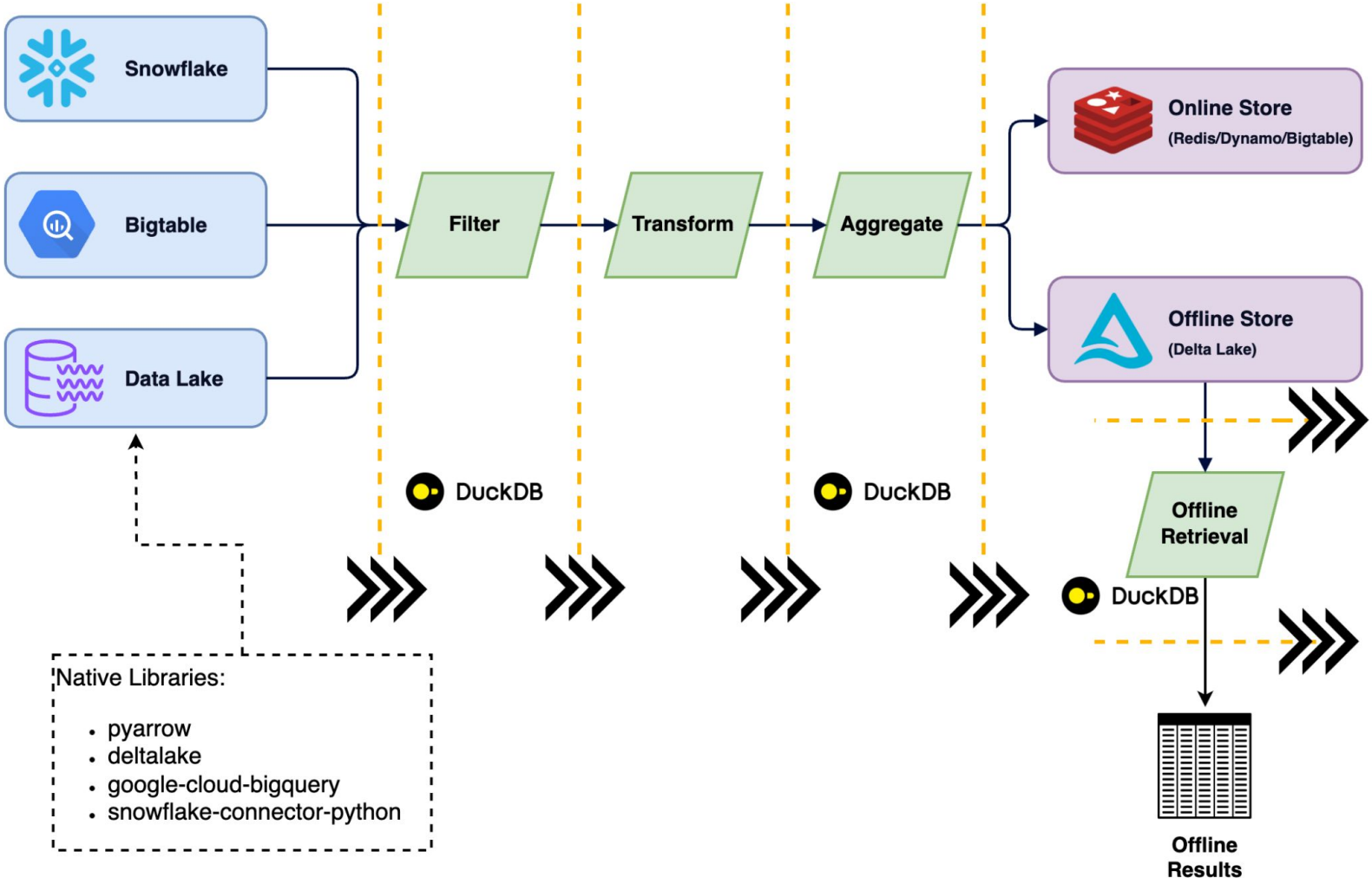
Can process non-trivial datasets with modest amounts of RAM due to streaming, out-of-core

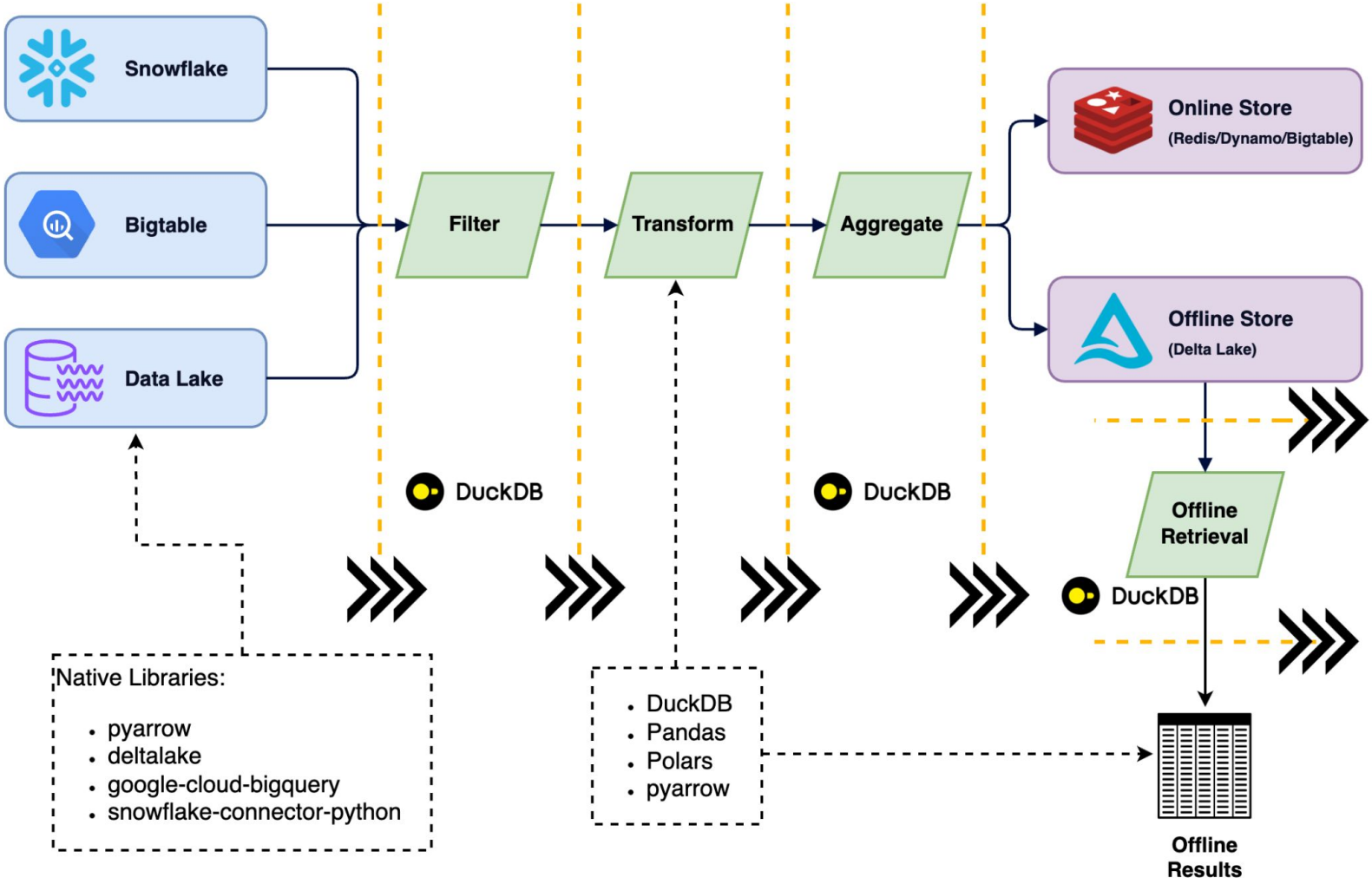
Simplicity

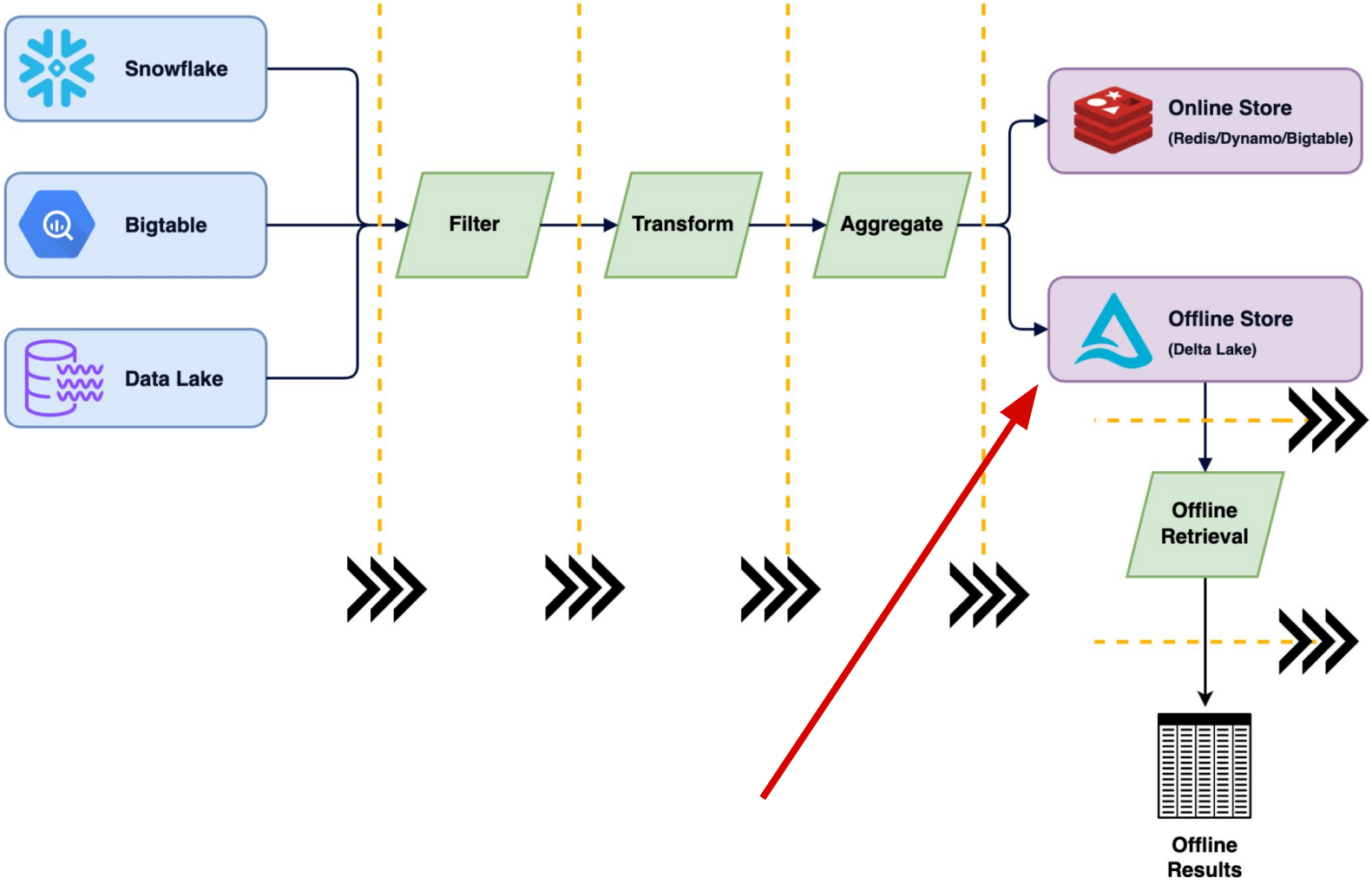
Eliminates the complexities of distributed query engines, Spark/Hadoop configuration, etc., which are unnecessary for the datasets we're typically dealing with.


Snowflake
Bigtable
Data Lake
Filter
Transform
Aggregate
Online Store
(Redis/Dynamo/Bigtable)
Offline Store
(Delta Lake)
Offline
Retrieval
Offline
Results


Snowflake
Bigtable
Data Lake
Filter
Transform
Aggregate
Online Store
(Redis/Dynamo/Bigtable)
Offline Store
(Delta Lake)
DuckDB
DuckDB
Offline
Retrieval
DuckDB
Offline
Results


Snowflake
Bigtable
Data Lake
Filter
Transform
Aggregate
Online Store
(Redis/Dynamo/Bigtable)
Offline Store
(Delta Lake)
DuckDB
DuckDB
Offline
Retrieval
DuckDB
Native Libraries:
pyarrow
deltalake
google-cloud-bigquery
snowflake-connector-python
Offline
Results


Snowflake
Bigtable
Data Lake
Filter
Transform
Aggregate
Online Store
(Redis/Dynamo/Bigtable)
Offline Store
(Delta Lake)
DuckDB
DuckDB
Offline
Retrieval
DuckDB
Native Libraries:
pyarrow
deltalake
google-cloud-bigquery
snowflake-connector-python
DuckDB
Pandas
Polars
pyarrow
Offline
Results


Snowflake
Bigtable
Data Lake
Filter
Transform
Aggregate
Online Store
(Redis/Dynamo/Bigtable)
Offline Store
(Delta Lake)
Offline
Retrieval
Offline
Results

Why do we have Delta?

In a word, atomicity:

- When appending new time ranges
- When partially deleting data
- When optimizing object size

Additionally, speeds up some types of operations by storing table metadata in a more convenient format

Integrations

Source: https://delta.io/

Delta Rust API

docs | source code

Rust **Python**

This library allows Rust (with Python bindings) low level access to Delta tables and is intended to be used with data processing frameworks like datafusion, ballista, rust-dataframe, vega, etc.

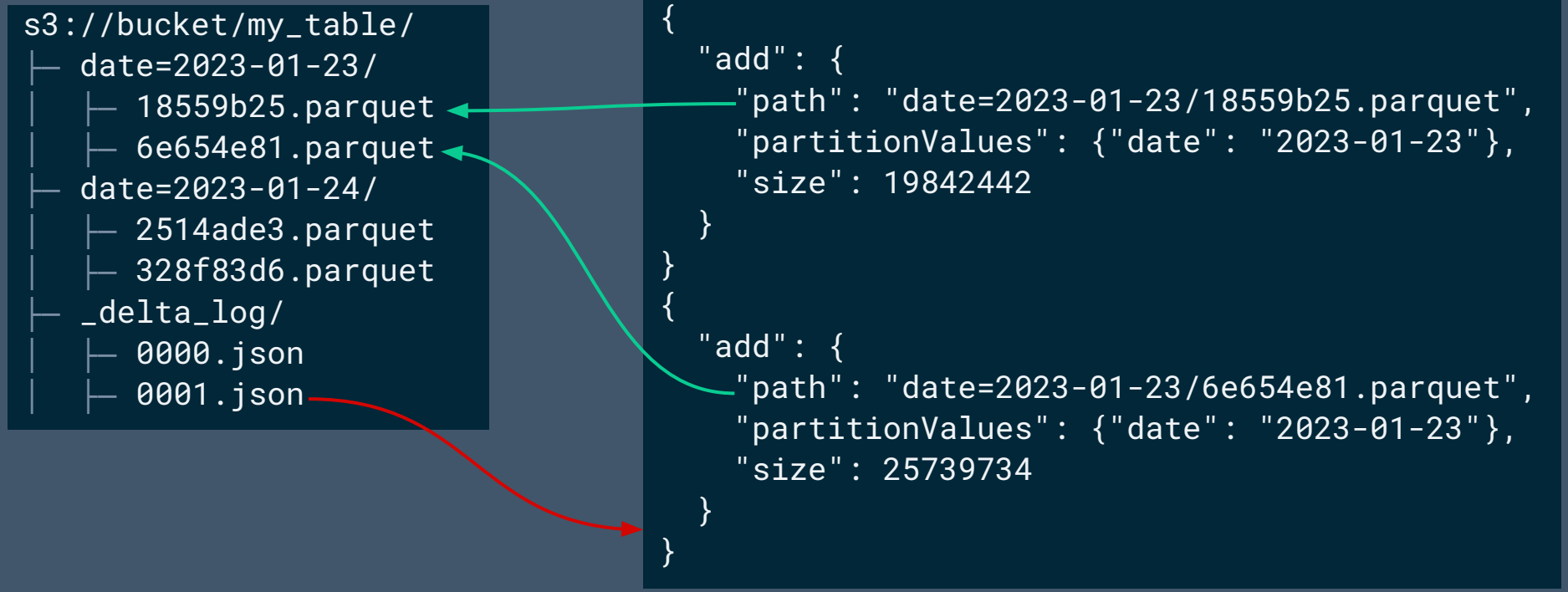
- Reads Delta Tables into pyarrow tables and/or datasets 🎉
- Limited write functionality:
  - Does not support the full range of write operations
  - Not compatible with Spark's DynamoDB locking protocol
  - Certain commit metadata not supported
- Limited AWS auth options

Are we stuck?

Thankfully not, but we need to understand a bit of how Delta works under the covers..

```
s3://bucket/my_table/
├─ date=2023-01-23/
│   ├─ 18559b25.parquet
│   ├─ 6e654e81.parquet
├─ date=2023-01-24/
│   ├─ 2514ade3.parquet
│   ├─ 328f83d6.parquet
├─ _delta_log/
│   ├─ 0000.json
│   ├─ 0001.json
```

```
"metaData": {
  "schema": [
    {"name": "ts", "type": "timestamp"},
    {"name": "date", "type": "string"},
    {"name": "clicks_1h", "type": "long"},
  },
  "partitionColumns": [
    "date"
  ]
}
```


s3://bucket/my_table/
├─ date=2023-01-23/
│ ├─ 18559b25.parquet
│ ├─ 6e654e81.parquet
├─ date=2023-01-24/
│ ├─ 2514ade3.parquet
│ ├─ 328f83d6.parquet
├─ _delta_log/
│ ├─ 0000.json
│ ├─ 0001.json
{
"add": {
"path": "date=2023-01-23/18559b25.parquet",
"partitionValues": {"date": "2023-01-23"},
"size": 19842442
}
}
{
"add": {
"path": "date=2023-01-23/6e654e81.parquet",
"partitionValues": {"date": "2023-01-23"},
"size": 25739734
}
}

Deleting a subset of keys

```
s3://bucket/my_table/
├─ date=2023-01-23/
│   ├─ 18559b25.parquet
│   ├─ 6e654e81.parquet
│   ├─ 7fea8d91.parquet
├─ date=2023-01-24/
│   ├─ 2514ade3.parquet
│   ├─ 328f83d6.parquet
├─ _delta_log/
│   ├─ 0000.json
│   ├─ 0001.json
│   ├─ 0002.json
```

```
{
  "remove": {
    "path": "date=2023-01-23/18559b25.parquet",
    "partitionValues": {"date": "2023-01-23"},
    "size": 19842442
  }
}
{
  "add": {
    "path": "date=2023-01-23/7fea8d91.parquet",
    "partitionValues": {"date": "2023-01-23"},
    "size": 17498312
  }
}
```

Solution

- delta-standalone is a JVM library which is part of the upstream Delta project. It does all of the transaction log management without touching the underlying data (parquet) files.
- We utilize it by spinning up a JVM as a sidecar to our main Python process, and send it messages when we need to mutate the Delta log.

Solution

- To delete a list of keys using this method:
  - Use delta-standalone to read the list of files in the current version of the table
  - Load the parquet files as a pyarrow dataset, filter out the keys to delete using DuckDB, and write the result to a new set of parquet files
  - Tell delta-standalone to write “add” and “remove” logs for all of the affected files.
  - Do the entire thing over again if you hit a conflict writing a log.

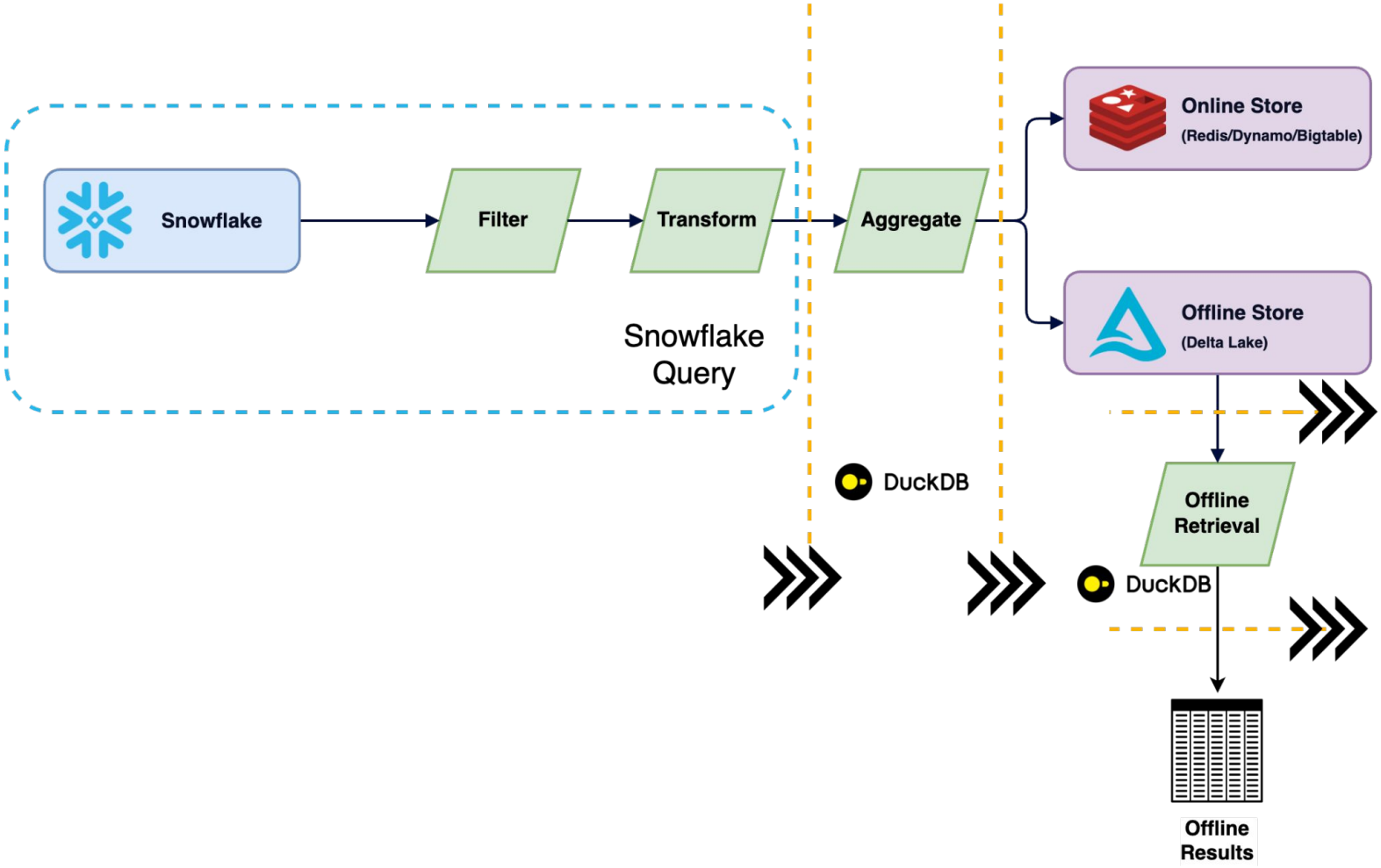
Alternatives Explored

- Switch to (Iceberg|Hudi)
- Improve on delta-rs

DuckDB Extension

- Problem: We have some esoteric aggregations we need to support (incremental versions of approximate quantile and approximate count distinct)
- DuckDB allows adding new aggregations in extensions.
- Maintenance and distribution is a little painful:
  - Extension APIs are not stable; every minor release so far has required changes.
  - Need to distribute a compiled extension for every OS/Architecture combination you want to support.
  - Built-in distribution mechanism is not versioned: you only get to publish one extension version per DuckDB version.

Snowflake Integration


Snowflake
Filter
Transform
Aggregate
Snowflake Query
Online Store
(Redis/Dynamo/Bigtable)
Offline Store
(Delta Lake)
DuckDB
Offline Retrieval
DuckDB
Offline Results

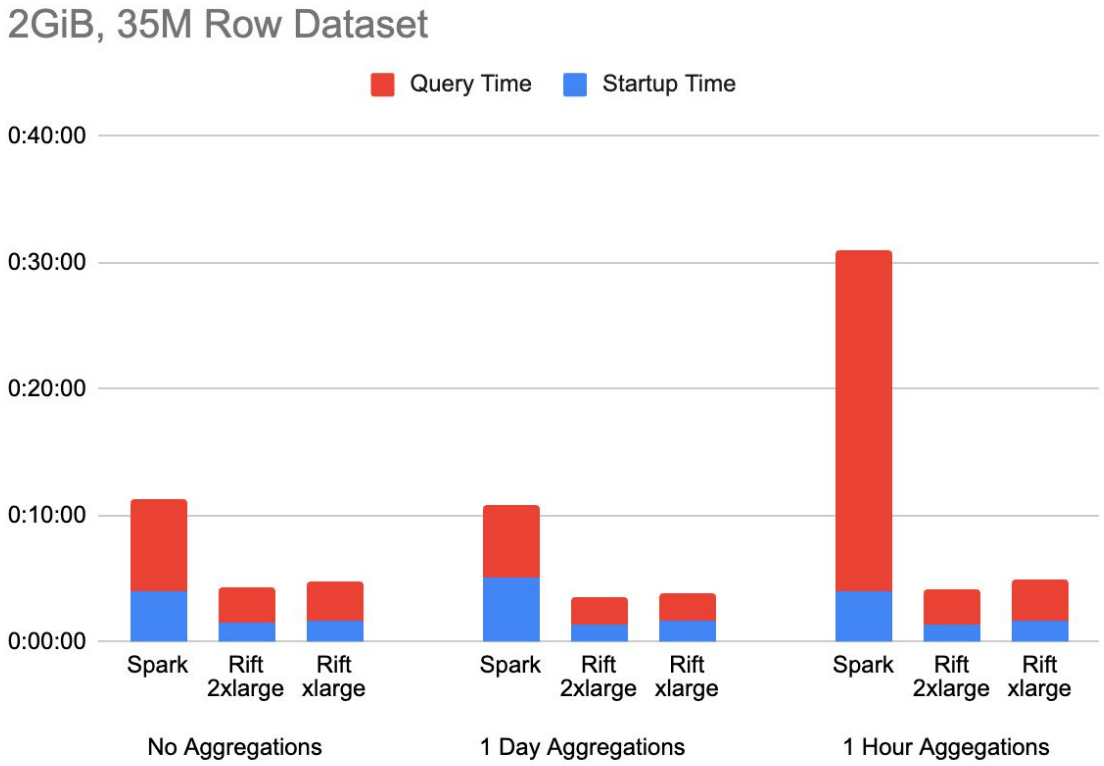
Results (so far)

- Released in Private Preview in January, along with some additional streaming ingest functionality which is outside of the scope of this talk.
- Together with the streaming functionality we are calling this configuration “Rift”
- Private preview has included a mix of new customers who never used Spark, customers who did use Spark, and a number of prospects.
- Hypothesis about Rift being easier to deploy and faster to production than Spark has been borne out so far.

Performance: Materialization Query

Label	Instance(s)	Hourly
Spark	2x m6id.xlarge	$0.4746*
Rift 2xlarge	1x r5.2xlarge	$0.5600
Rift xlarge	1x r5.xlarge	$0.2800

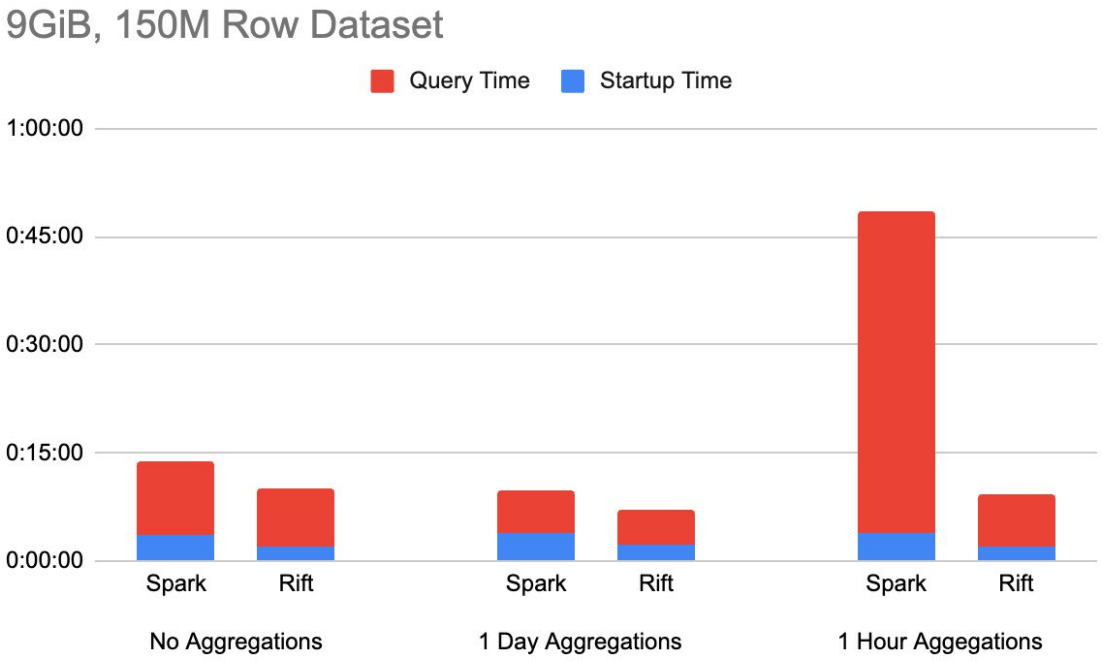
**Hourly cost does not include Driver or DBUS*



Performance: Materialization Query

Label	Instance(s)	Hourly
Spark	4x m6id.xlarge	$0.9492*
Rift	1x r5.2xlarge	$1.1200

**Hourly cost does not include Driver or DBUs*



The End

- Find me afterwards to ask questions!
- Tecton is hiring Remotely as well as in NYC and SF!