

Streaming SQL for Data Engineers: The Next Big Thing?



Materialize



decodable



timeplus



popsink



DELTASTREAM



SINGULARITY
DATA



Materialize



decodable



timeplus



popsink



DELTASTREAM



SINGULARITY
DATA

Streaming SQL Products

- Apache Flink
- Apache Spark
- Apache Beam
- AWS Kinesis
- Google Cloud Dataflow
- Databricks
- ksqlDB
- ...

Open source and
vendor solutions

- Meta
- LinkedIn
- Pinterest
- DoorDash
- Alibaba
- ...

Companies building
internal platforms

STREAMING SQL PIPELINES

SO HOT RIGHT NOW

👏 Hi, I'm Yaroslav

👋 Hi, I'm Yaroslav

- Principal Software Engineer @ Goldsky
- Staff Data Engineer @ Shopify
- Software Architect @ Activision
- ...

👋 Hi, I'm Yaroslav

- Principal Software Engineer @ Goldsky
- Staff Data Engineer @ Shopify
- Software Architect @ Activision
- ...

❤️ Apache Flink

```
TableEnvironment tableEnv = TableEnvironment.create(/*...*/);

Table revenue = tableEnv.sqlQuery(
    "SELECT cID, cName, SUM(revenue) AS revSum " +
    "FROM Orders " +
    "WHERE cCountry = 'FRANCE' " +
    "GROUP BY cID, cName"
);
```



... but why SQL?

Why SQL?

- Wide adoption
- Declarative transformation model
- Planner!
- Common type system

What instead of How

Intention

User



Execution

Runtime

Imperative Style

Intention

User



Planning

Planner



Execution

Runtime

Declarative SQL Style

Declarative Transformation Model

```
SELECT * FROM Orders  
INNER JOIN Product  
ON Orders.productId = Product.id
```

- LOTS of code!
- Create an operator to connect two streams
- Define and accumulate state
- Implement a mechanism for emitting the latest value per key

SQL API

DataStream API

Declarative Transformation Model

```
SELECT * FROM Orders  
INNER JOIN Product  
ON Orders.productId = Product.id
```

SQL API

```
val orders = tEnv.from("Orders")  
  .select($"productId", $"a", $"b")
```

```
val products = tEnv.from("Products")  
  .select($"id", $"c", $"d")
```

```
val result = orders  
  .join(products)  
  .where($"productId" === $"id")  
  .select($"a", $"b", $"c")
```

Why not Table API?

Declarative Transformation Model

```
SELECT * FROM (  
    SELECT *, ROW_NUMBER() OVER (PARTITION BY ticker  
ORDER BY price DESC) AS row_num  
    FROM stock_table)  
WHERE row_num <= 10;
```

Top-N Query

```

SELECT *
FROM stock_table
  MATCH_RECOGNIZE(
    PARTITION BY ticker
    ORDER BY event_time
    MEASURES
      A.event_time AS initialPriceTime,
      C.event_time AS dropTime,
      A.price - C.price AS dropDiff,
      A.price AS initialPrice,
      C.price AS lastPrice
    ONE ROW PER MATCH
    AFTER MATCH SKIP PAST LAST ROW
    PATTERN (A B* C) WITHIN INTERVAL '10' MINUTES
    DEFINE
      B AS B.price > A.price - 500
  )

```

Row Pattern Recognition in SQL (ISO/IEC TR 19075-5:2016)

Planner Decoupling

	Without Object Reuse		With Object Reuse
Old Planner	84,279	→	88,426 (+5%)
Blink Planner	89,048	→	136,710 (+54%)

From <https://www.ververica.com/blog/a-journey-to-beating-flinks-sql-performance>

Flink Planner Migration

Planner Optimizations & Query Rewrite

- Predicate push down
- Projection push down
- Join rewrite
- Join elimination
- Constant inlining
- ...

Common Type System

```
CREATE TABLE TABLE (  
  id BIGINT,  
  number INTEGER,  
  timestamp TIMESTAMP,  
  author STRING,  
  difficulty STRING,  
  size INTEGER,  
  vid BIGINT,  
  block_range STRING  
  PRIMARY KEY (vid) NOT ENFORCED  
) WITH (  
  'connector' = 'jdbc',  
  'table-name' = 'table'  
);
```

SQL API

```
val postgresSink: SinkFunction[Envelope] = JdbcSink.sink(  
  "INSERT INTO table " +  
    "(id, number, timestamp, author, difficulty, size, vid, block_range) " +  
    "VALUES (?, ?, ?, ?, ?, ?, ?, ?) " +  
    "ON CONFLICT (id) DO UPDATE SET " +  
    "number = excluded.number, " +  
    "timestamp = excluded.timestamp, " +  
    "author = excluded.author, " +  
    "difficulty = excluded.difficulty, " +  
    "size = excluded.size, " +  
    "vid = excluded.vid, " +  
    "block_range = excluded.block_range " +  
    "WHERE excluded.vid > table.vid",  
  new JdbcStatementBuilder[Envelope] {  
    override def accept(statement: PreparedStatement, record: Envelope): Unit = {  
      val payload = record.payload  
      payload.id.foreach { id => statement.setString(1, id) }  
      payload.number.foreach { number => statement.setBigDecimal(2, new java.math.BigDecimal(number)) }  
      payload.timestamp.foreach { timestamp => statement.setBigDecimal(3, new java.math.BigDecimal(timestamp)) }  
      payload.author.foreach { author => statement.setString(4, author) }  
      payload.difficulty.foreach { difficulty => statement.setBigDecimal(5, new java.math.BigDecimal(difficulty)) }  
      payload.size.foreach { size => statement.setBigDecimal(6, new java.math.BigDecimal(size)) }  
      payload.vid.foreach { vid => statement.setLong(7, vid.toLong) }  
      payload.block_range.foreach { block_range => statement.setObject(8, new PostgresIntRange(block_range), Types.  
    }  
  },  
)
```

DataStream API



*When you start using SQL
you get access to the
decades of advancements
in database design*

When NOT to use

- Complex serialization / deserialization logic
- Low-level optimizations, especially with state and timers
- Not always debugging-friendly

Dealing with Complexity

UDFs for heavy lifting

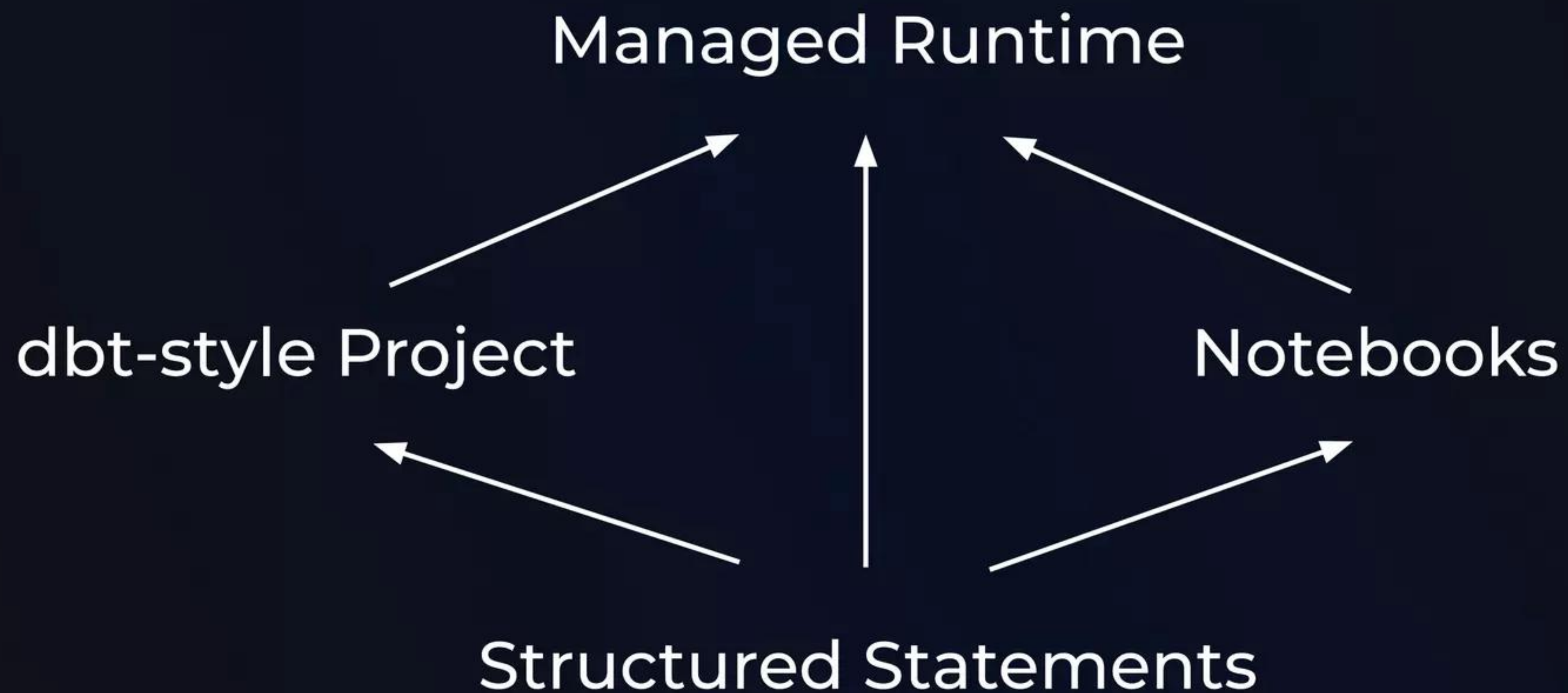
- Calling 3rd-party libraries
- External calls
- Enrichments

Templating

- Control structures
- dbt-style macros and references

Convinced? Let's use it!

Ways to use



Requirements

- Version control
- Code organization
- Testability
- CI/CD
- Observability

Structured Statements

```
def revenueByCountry(country: String): Table = {  
  tEnv.sqlQuery(  
    s"""  
      |SELECT name, SUM(revenue) AS totalRevenue  
      |FROM Orders  
      |WHERE country = '${country}'  
      |GROUP BY name""").stripMargin  
  )  
}
```

Structured Statements

✓ *structure*

✓ *mock/stub
for testing*

```
def revenueByCountry(country: String): Table = {  
  tEnv.sqlQuery(  
    s"""  
      |SELECT name, SUM(revenue) AS totalRevenue  
      |FROM Orders  
      |WHERE country = '${country}'  
      |GROUP BY name""".stripMargin  
    )  
}
```

Structured Statements

- **Treat them like code**
- Only make sense when Table API is not available
- Mix with other API flavours
- SQL also has style guides
- Otherwise it's a typical streaming application!

Structured Statements

- Version control: ●
- Code organization: ●
- Testability: ●
- CI/CD: ●
- Observability: ●

dbt-style Project

→ models

- ◆ common

- users.sql
- users.yml

- ◆ sales.sql

- ◆ sales.yml

- ◆ ...

→ tests

- ◆ ...

dbt-style Project

→ models

◆ common

- users.sql
- users.yml

◆ sales.sql

◆ sales.yml

◆ ...

→ tests

◆ ...

✓ *structured*

✓ *schematized*

✓ *testable*

dbt-style Project

```
{{ config(materialized='materializedview') }}
```

```
SELECT symbol,  
        AVG(bid_price) AS avg  
FROM {{ REF('market_orders') }}  
GROUP BY symbol
```

```
SELECT  
  ((text::jsonb)->>'bid_price')::FLOAT AS bid_price,  
  (text::jsonb)->>'order_quantity' AS order_quantity,  
  (text::jsonb)->>'symbol' AS symbol,  
  (text::jsonb)->>'trade_type' AS trade_type,  
  to_timestamp(((text::jsonb)->'timestamp')::BIGINT) AS ts  
FROM {{ REF('market_orders_raw') }}
```

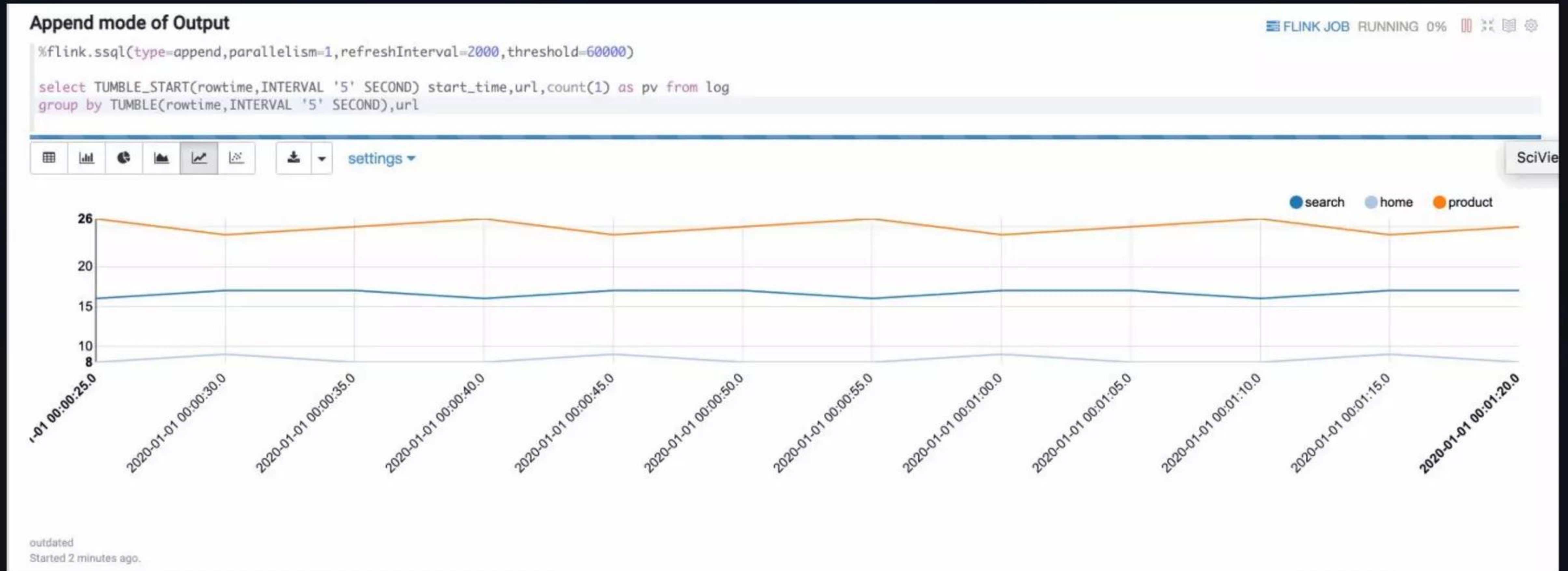
dbt-style Project

- Works well for heavy analytical use-cases
- Could write tests in Python/Scala/etc.
- Probably needs more tooling than you think (state management, observability, etc.)
- Check dbt adapter from Materialize!

dbt-style Project

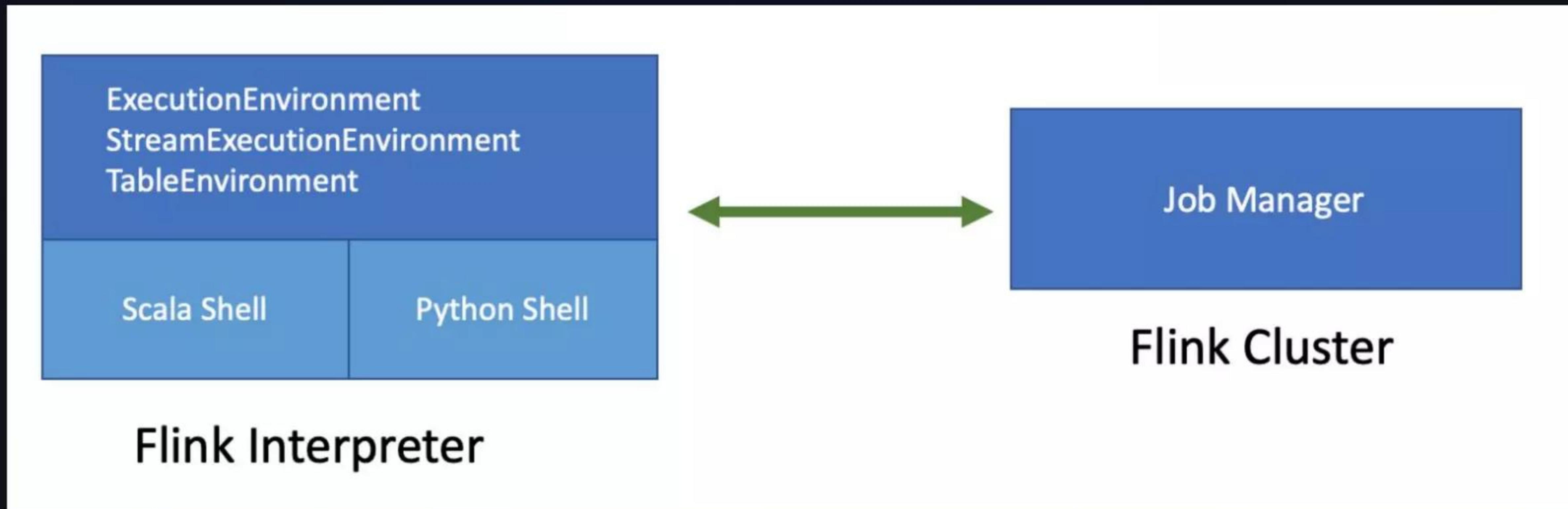
- Version control: ●
- Code organization: ●
- Testability: ●
- CI/CD: ●
- Observability: ●

Notebooks



Apache Zeppelin

Notebooks



Apache Zeppelin

Notebooks

- **Great UX**
- Ideal for exploratory analysis and BI
- Complements all other patterns really well
- Way more important for realtime workloads

Notebooks

We don't recommend productionizing notebooks and instead encourage empowering data scientists to build production-ready code with the right programming frameworks

<https://www.thoughtworks.com/en-ca/radar/techniques/productionizing-notebooks>

Notebooks

- Version control: ●
- Code organization: ●
- Testability: ●
- CI/CD: ●
- Observability: ●

Managed Runtime

process-usage-metrics-cell-1

4fbc4649 • Processes usage metrics for S3

Running

Stop

...

Overview

SQL

Version: 1

Version: 1

```
1 INSERT INTO `processed-usage-metrics`
2   SELECT
3     jobId AS job_id,
4     '1' AS cell_id,
5     accountId AS account_id,
6     'num_tasks' AS metric_name,
7     coalesce(usageMetrics.numTasks, 0) AS metric_value,
8     TO_TIMESTAMP_LTZ('timestamp', 3) AS ts
9   FROM `user-metrics-cell-1-raw`
```

Completed in 24 seconds

Run Preview

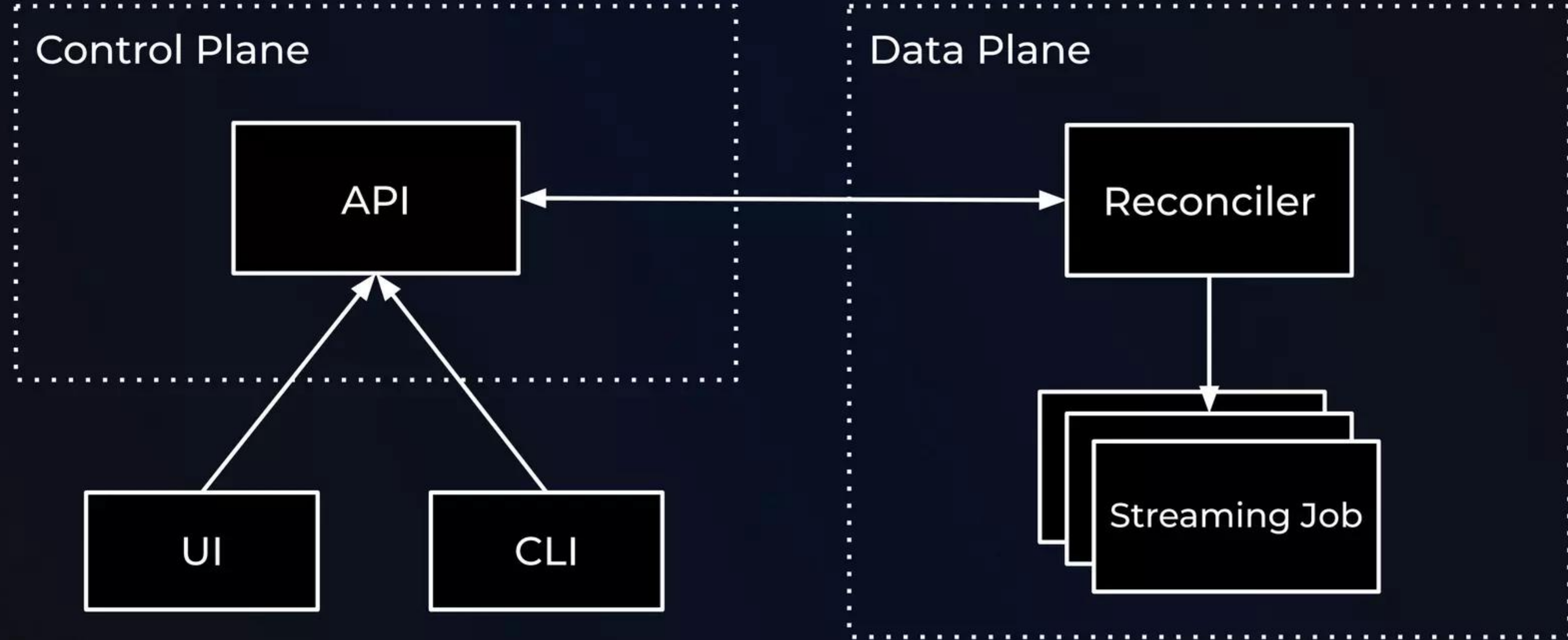
...

decodable

Managed Runtime

- Managed $\approx$ “Serverless”
- Auto-scaling
- Automated deployments, rollbacks, etc.
- Testing for different layers is decoupled (runtime vs jobs)

Managed Runtime



Reference Architecture

*Any managed runtime
requires excellent
developer experience
to succeed*

Managed Runtime: Ideal Developer Experience

SELECT * ...



SELECT * ...



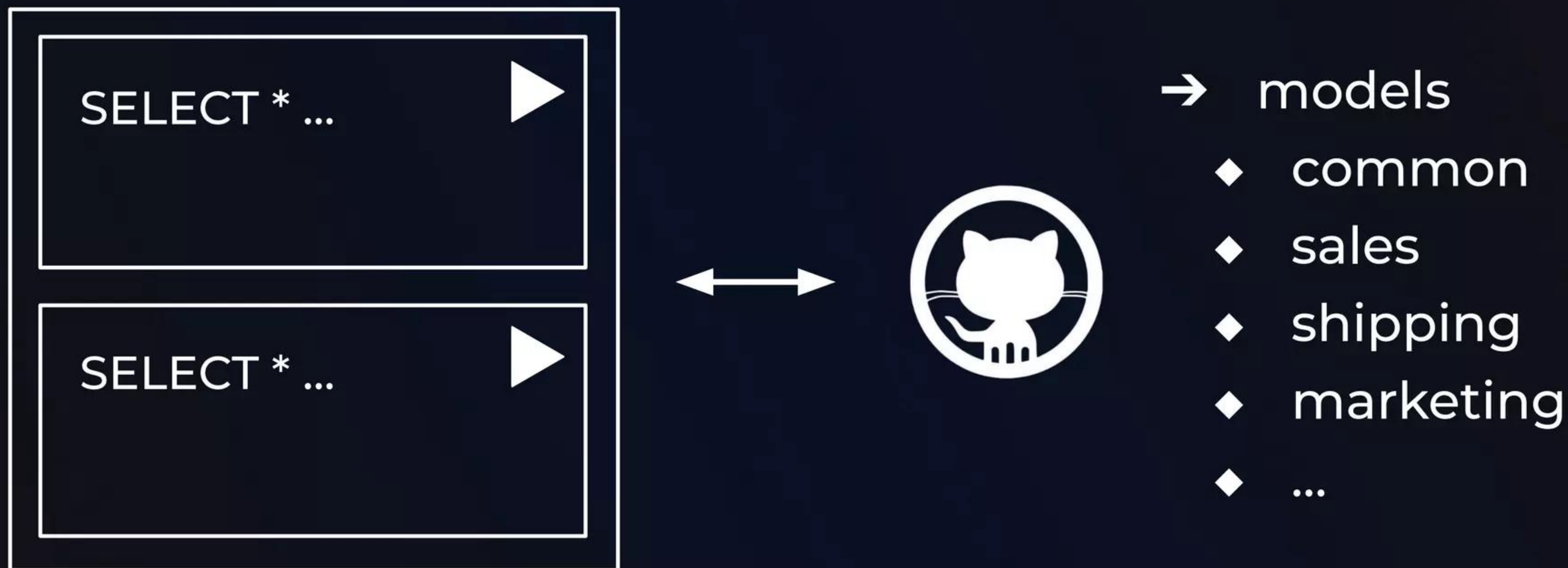
Notebooks UX

Managed Runtime: Ideal Developer Experience



Version Control Integration

Managed Runtime: Ideal Developer Experience



dbt-style Project Structure

Managed Runtime: Ideal Developer Experience

SELECT * ...



SELECT * ...



- Version 1
- Version 2
- Version 3
- ...

Versioning

Managed Runtime: Ideal Developer Experience

SELECT * ... ▶

SELECT * ... ▶

User	Count
Irene	100
Alex	53
Josh	12
Jane	1

Previews

Managed Runtime

- Version control: ●
- Code organization: ●
- Testability: ●
- CI/CD: ●
- Observability: ●

Summary

	Structured Statements	dbt-style Project	Notebooks	Managed Runtime
Version Control	●	●	●	●
Code Organization	●	●	●	●
Testability	●	●	●	●
CI/CD	●	●	●	●
Observability	●	●	●	●
Complexity	●	●	●	●

General Guidelines

- Long-running streaming apps require special attention to state management
- Try to avoid mutability: every change is a new version
- Integration testing > unit testing
- Embrace the SRE mentality

Really dislike SQL?

In SQL

```
SELECT
  base.fac_type as fac_type,
  COUNT( 1) as airport_count
FROM `malloy-data.faa.airports` as base
WHERE base.state='CA'
GROUP BY 1
ORDER BY 2 desc
```

Equivalent in Malloy

```
query: table('malloy-data.faa.airports') -> {
  group_by: fac_type
  aggregate: airport_count is count()
  where: state = 'CA'
  order_by: airport_count desc
}
```

Malloy

```
from employees
# Filter before aggregations
filter start_date > @2021-01-01
group country (
  aggregate [max_salary = max salary]
)
# And filter after aggregations!
filter max_salary > 100000
```

```
SELECT
  country,
  MAX(salary) AS max_salary
FROM
  employees
WHERE
  start_date > DATE '2021-01-01'
GROUP BY
  country
HAVING
  MAX(salary) > 100000
```

PRQL

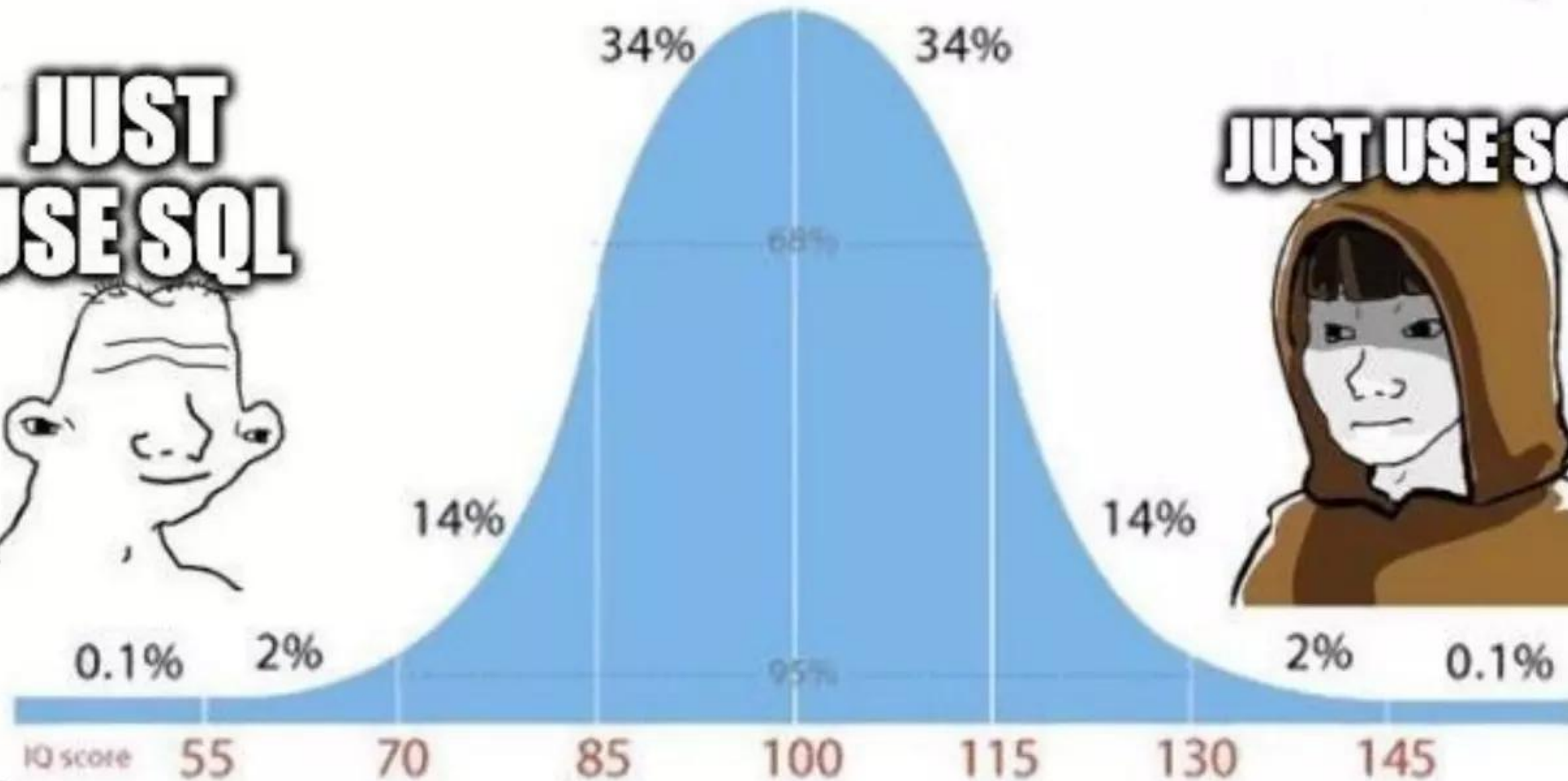
**SQL IS OLD AND ISN'T A REAL LANGUAGE.
WRITE ACTUAL CODE WITH PANDAS OR SPARK**



**JUST
USE SQL**



JUST USE SQL



Questions?

@saplens