Delta Lake: Making Cloud Data Lakes Transactional and Scalable

Reynold Xin

@rxin

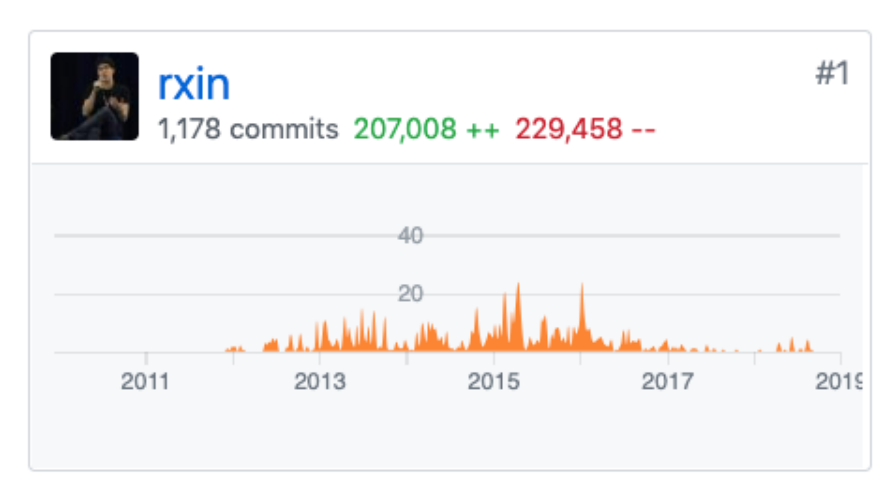
Stanford University, 2019-05-15

databricks

About Me

Databricks co-founder & Chief Architect
- Designed most major things in “modern day” Apache Spark
- #1 contributor to Spark by commits and net lines deleted

PhD in databases from Berkeley

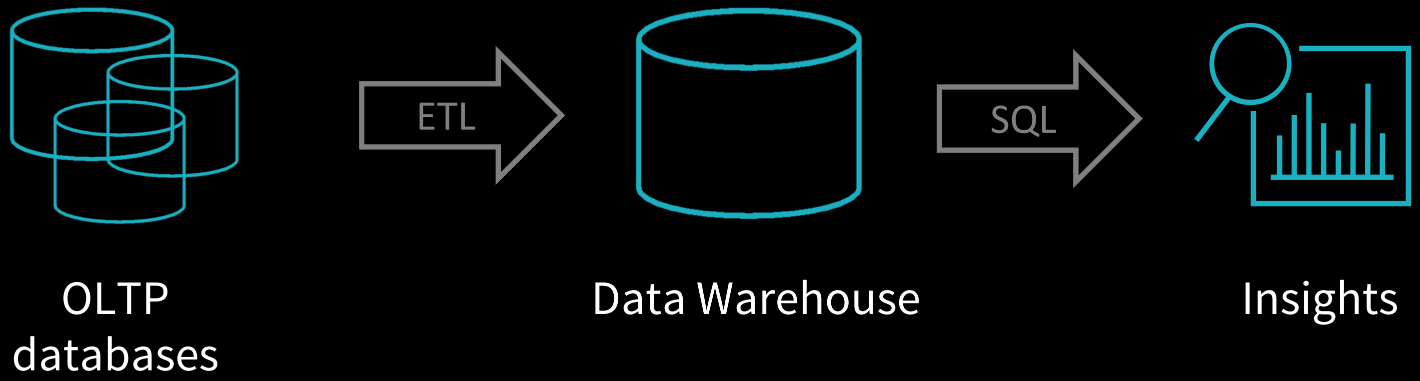


Building data analytics platform is hard

Data streams

Insights

Traditional Data Warehouses



Challenges with Data Warehouses

ETL pipelines are often complex and slow

Ad-hoc pipelines to process data and ingest into warehouse

No insights until daily data dumps have been processed

Workloads often limited to SQL and BI tools

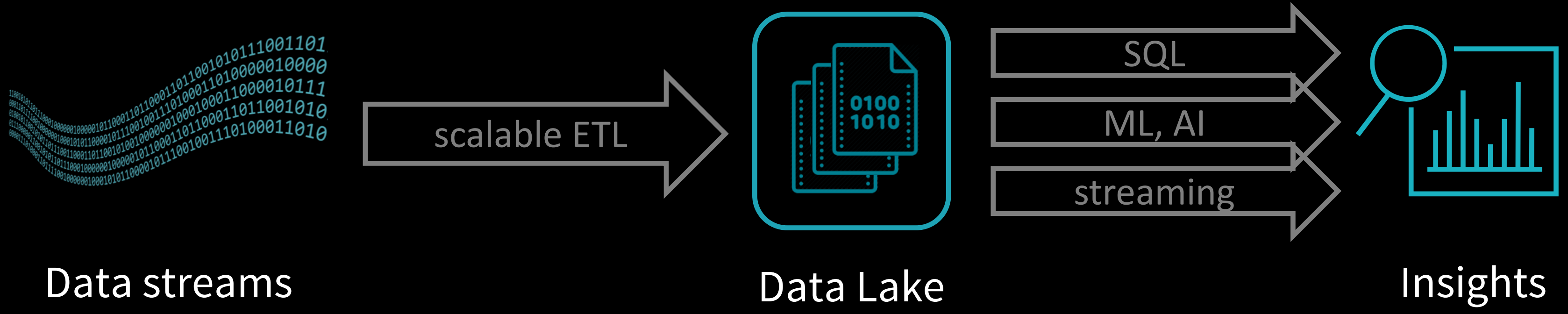
Data in proprietary formats

Hard to do integrate streaming, ML, and AI workloads

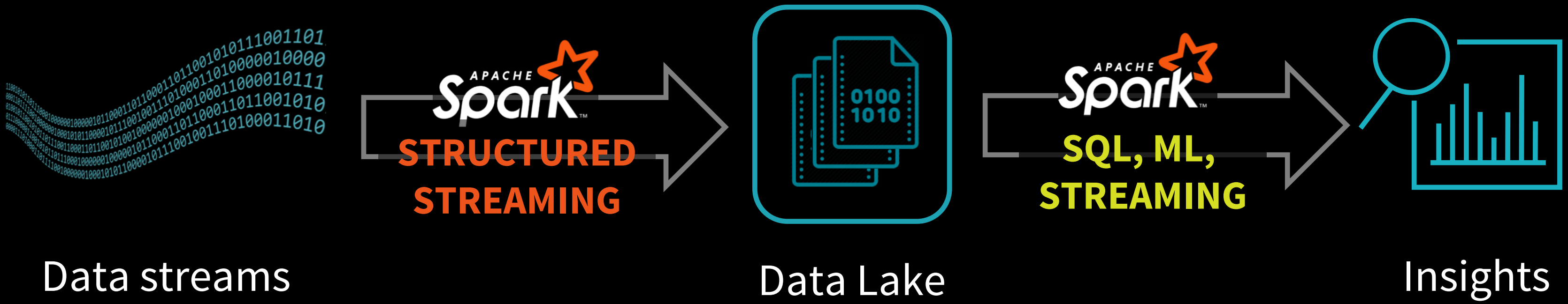
Performance is expensive

Scaling up/out usually comes at a high cost

Dream of Data Lakes



Data Lakes + Spark = Awesome!



APACHE Spark The 1st Unified Analytics Engine

Advantages of Data Lakes

ETL pipelines are ~~complex and slow~~ simpler and fast

Unified Spark API between batch and streaming simplifies ETL

Raw unstructured data available as structured data in minutes

Workloads ~~limited~~ not limited anything!

Data in files with open formats

Integrate with data processing and BI tools

Integrate with ML and AI workloads and tools

Performance is ~~expensive~~ cheaper

Easy and cost-effective to scale out compute and storage

Challenges of Data Lakes in practice

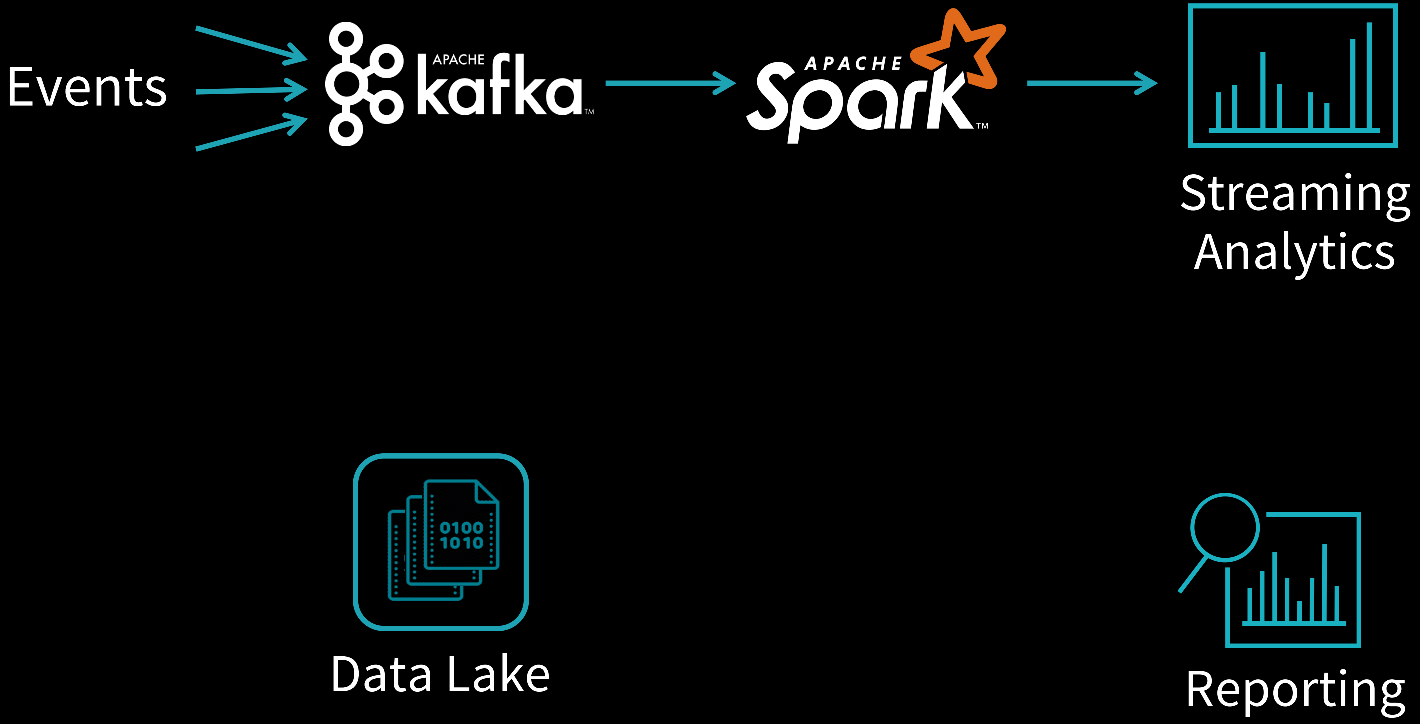
databricks

Challenges of Data Lakes in practice

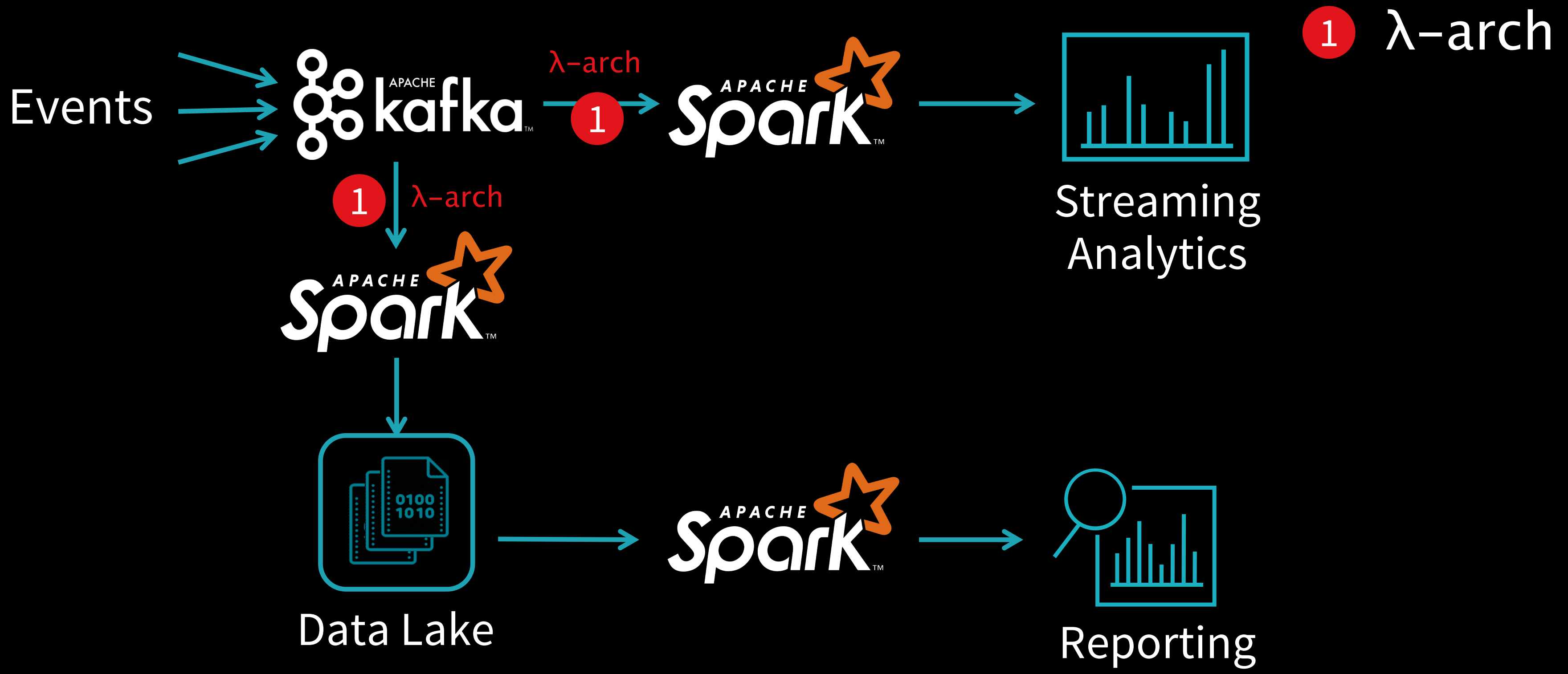
ETL @ databricks

Evolution of a Cutting-Edge Data Pipeline

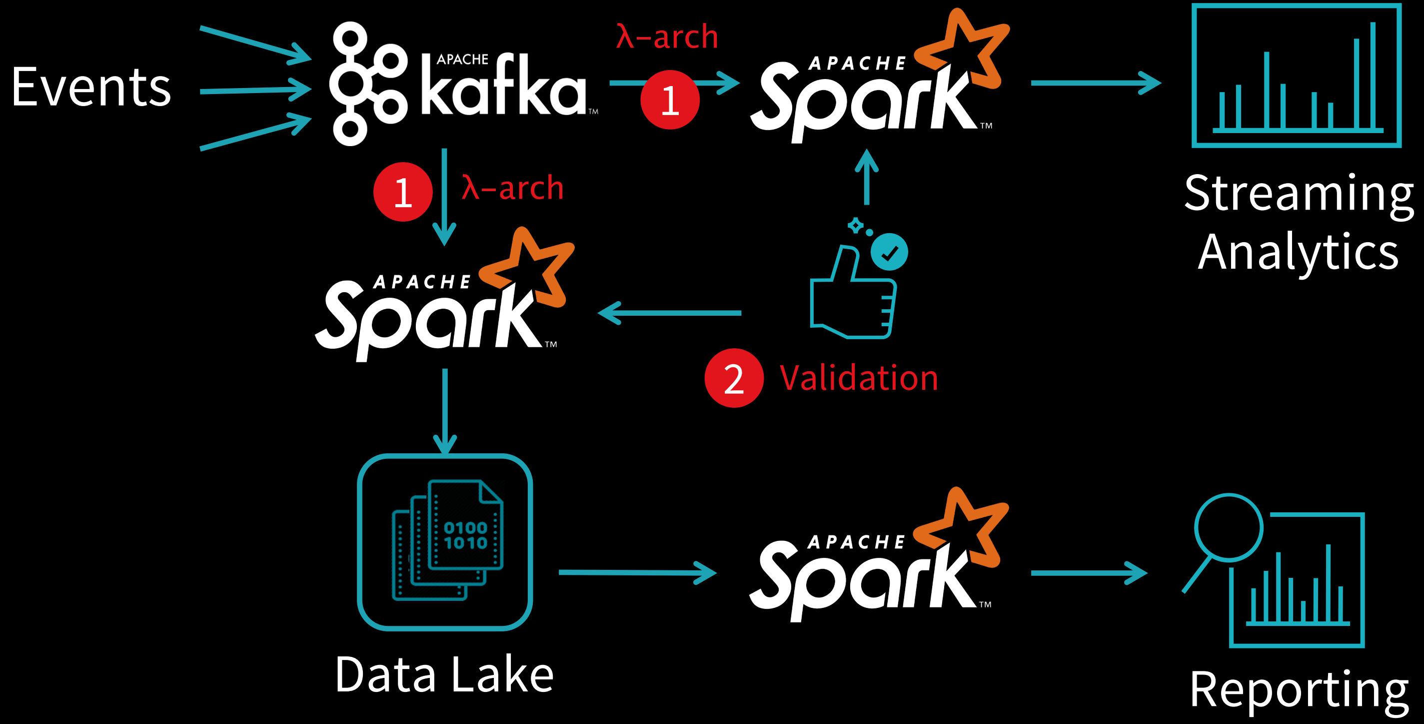
Evolution of a Cutting-Edge Data Pipeline



Challenge #1: Historical Queries?



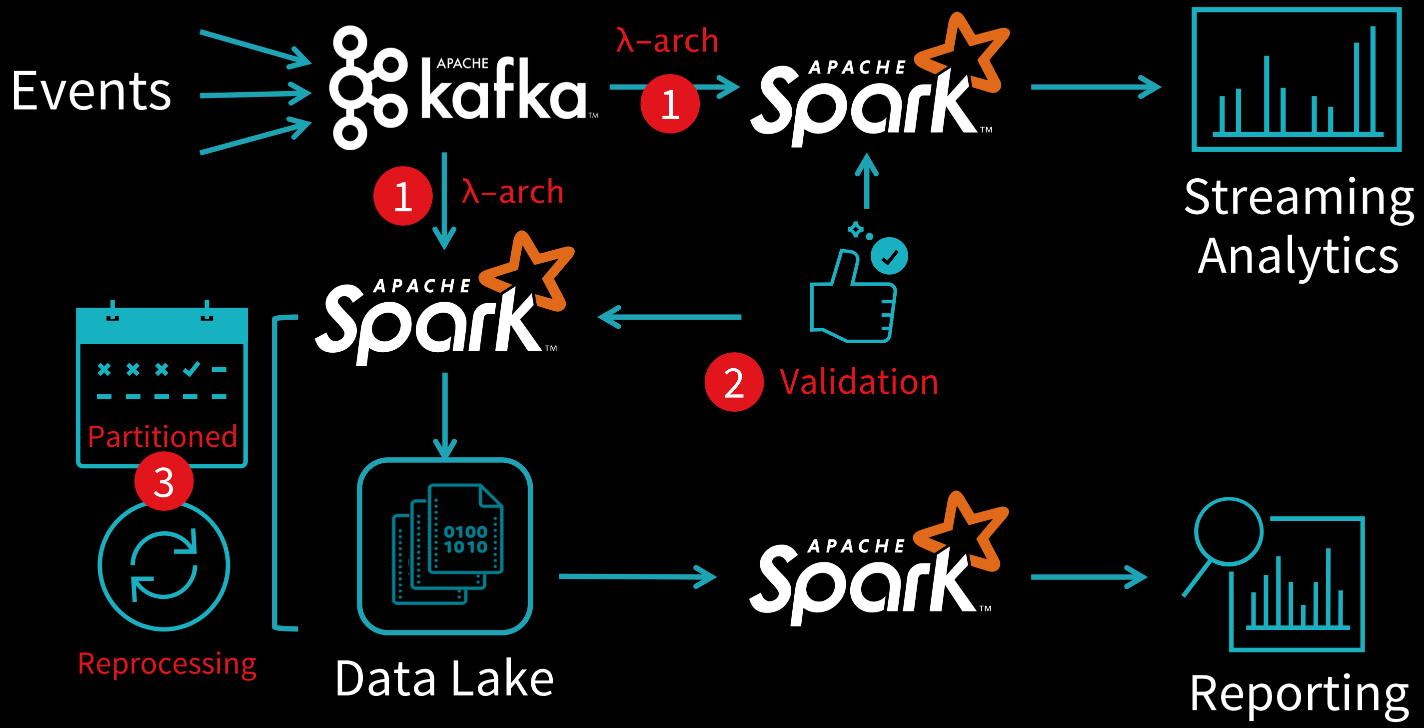
Challenge #2: Messy Data?



1 λ-arch

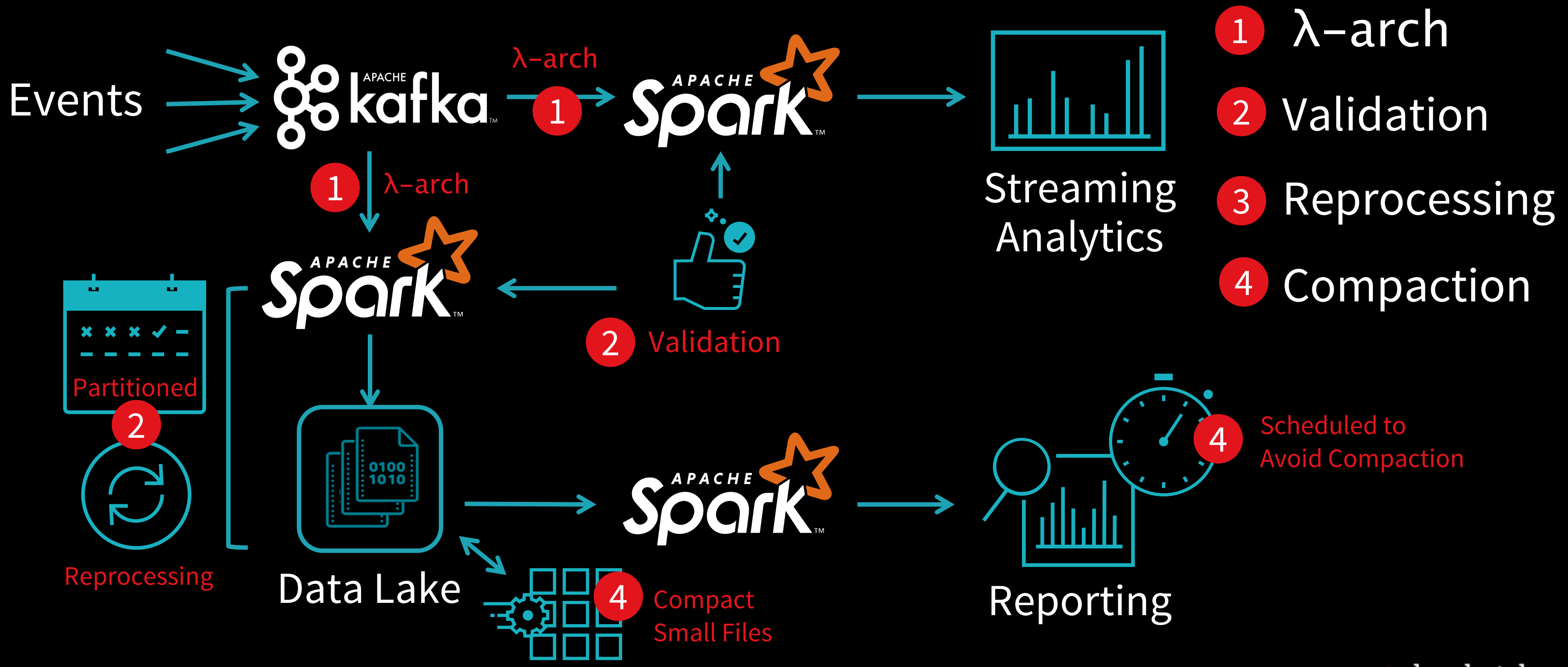
2 Validation

Challenge #3: Mistakes and Failures?



1 λ-arch
2 Validation
3 Reprocessing

Challenge #4: Query Performance?



databricks

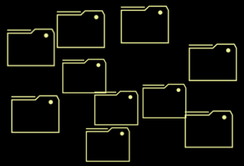
Data Lake Reliability Challenges

Failed production jobs leave data in corrupt state requiring tedious recovery

Lack of consistency makes it almost impossible to mix appends, deletes, upserts and get consistent reads

Lack of schema enforcement creates inconsistent and low quality data

Data Lake Performance Challenges



Too many small or very big files - more time opening & closing files rather than reading content (worse with streaming)

Partitioning aka “poor man’s indexing”- breaks down when data has many dimensions and/or high cardinality columns

Neither storage systems, nor processing engines are great at handling very large number of subdir/files

Figuring out what to read is too slow

Extremely slow dataframe loading

[redacted]

Commands Blocked on Metadata Operations

[redacted]

Data integrity is hard

Keep getting FileNotFound for tempView
[redacted]

Different field types cause conflicting schemas w...
[redacted]

CRITICAL production problem: inconsistent job e...
[redacted]

Appending new data to a partitioned table
[redacted]

Band-aid solutions made it worse!

refresh table issue - status ?

[redacted]

refresh table

[redacted]

Keep getting FileNotFound for tempView

[redacted]

Everyone has the same problems

Concatenate small files [redacted]

how to control number of parquet files within par... [redacted]

Reading many small JSON files on ADLS in Databricks

[redacted]

parquet file optimization Inbox x

[redacted]

THE GOOD
OF DATA WAREHOUSES

- Pristine Data
- Transactional Reliability
- Fast SQL Queries

THE GOOD
OF DATA LAKES

- Massive scale out
- Open Formats
- Mixed workloads

databricks

DELTA

The **SCALE** of data lake

The **RELIABILITY & PERFORMANCE** of data warehouse

The **LOW-LATENCY** of streaming

databricks DELTA = Scalable storage + Transactional log databricks **DELTA**

Scalable storage

table data stored as Parquet files
on HDFS, AWS S3, Azure Blob Stores

Transactional log

sequence of metadata files to track
operations made on the table

stored in scalable storage along with table

```
pathToTable/
      +---- 000.parquet
      +---- 001.parquet
      +---- 002.parquet
      +      ...
      |
      +---- _delta_log/
             +---- 000.json
             +---- 001.json
             ...
```

Log Structured Storage

Changes to the table are stored as *ordered, atomic* commits

Each commit is a set of actions file in directory `_delta_log`

```
_delta_log/
      000.json
      001.json
```

INSERT actions

- Add 001.parquet
- Add 002.parquet

UPDATE actions

- Remove 001.parquet
- Remove 002.parquet
- Add 003.parquet

Log Structured Storage

Readers read the log in atomic units thus reading consistent snapshots

000.json

001.json

INSERT actions

- Add 001.parquet
- Add 002.parquet

UPDATE actions

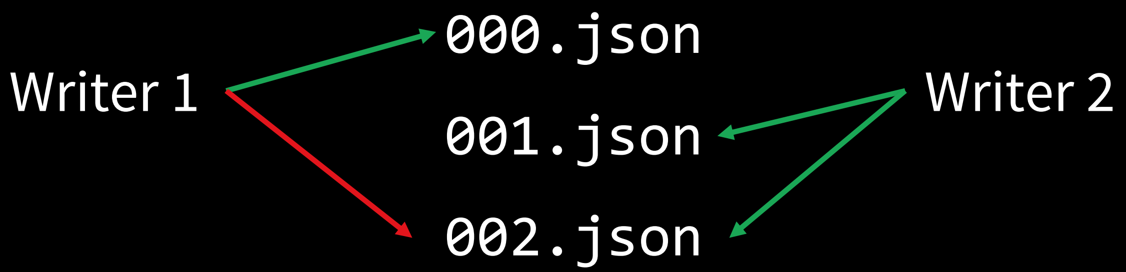
- Remove 001.parquet
- Remove 002.parquet
- Add 003.parquet

readers will read
either [001+002].parquet
or 003.parquet
and nothing in-between

Mutual Exclusion

Concurrent writers need to agree on the order of changes

New commit files must be created mutually exclusively



only one of the writers trying to concurrently write 002.json must succeed

Challenges with cloud storage

Different cloud storage systems have different semantics to provide atomic guarantees

Cloud Storage	Atomic Files Visibility	Atomic Put if absent	Solution
Azure Blob Store, Azure Data Lake	✗	✔	Write to temp file, rename to final file if not present
AWS S3	✔	✗	Separate service to perform all writes directly (single writer)

Concurrency Control

Pessimistic Concurrency
Block others from writing anything
Hold lock, write data files, commit to log

✔ Avoid wasted work

✘ Distributed locks

Optimistic Concurrency
Assume it'll be okay and write data files
Try to commit to the log, fail on conflict
Enough as write concurrency is usually low

✔ Mutual exclusion is enough!

✘ Breaks down if there a lot of conflicts

Solving Conflicts Optimistically

1. Record start version
2. Record reads/writes
3. If someone else wins, check if anything you read has changed.
4. Try again.

User 1		User 2
R: A	← `000000.json` →	R: A
W: B	→ `000001.json` ←	W: C
	`000002.json` ←	

new file C does not conflict with new file B, so retry and commit successfully as 2.json

Solving Conflicts Optimistically

1. Record start version
2. Record reads/writes
3. If someone else wins, check if anything you read has changed.
4. Try again.

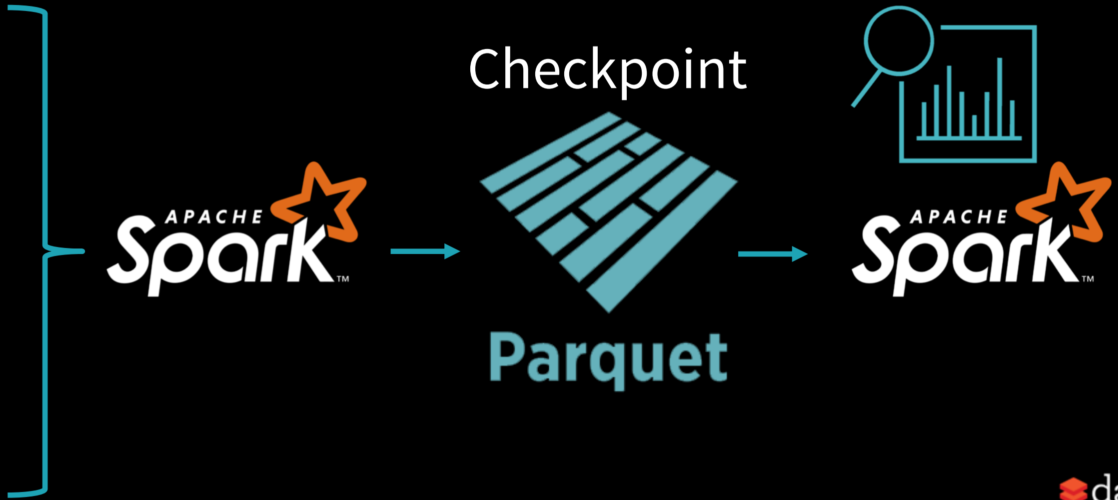
User 1				User 2
R: A	←	000000.json	→	R: A
W: A,B	→	000001.json	←	W: A,C

Deletions of file A by user 1 conflicts with deletion by user 2, user 2 operation fails

Metadata/Checkpoints as Data

Large tables can have millions of files in them! Even pulling them out of Hive [MySQL] would be a bottleneck.

```
Add 1.parquet
Add 2.parquet
Remove 1.parquet
Remove 2.parquet
Add 3.parquet
```



Challenges solved: Reliability

Problem:

Failed production jobs leave data in corrupt state requiring tedious recovery

databricks DELTA

Solution:

Failed write jobs do not update the commit log, hence partial / corrupt files not visible to readers

Challenges solved: Reliability

Challenge :

Lack of consistency makes it almost impossible to mix appends, deletes, upserts and get consistent reads

databricks DELTA

Solution:

All reads have full snapshot consistency
All successful writes are consistent
In practice, most writes don't conflict
Tunable isolation levels (serializability by default)

Challenges solved: Reliability

Challenge :

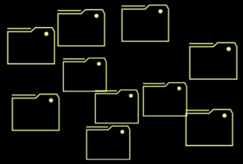
Lack of schema enforcement creates inconsistent and low quality data

databricks
DELTA

Solution:

Schema recorded in the log
Fails attempts to commit data with incorrect schema
Allows explicit schema evolution
Allows invariant and constraint checks (high data quality)

Challenges solved: Performance



Challenge:

Too many small files increase resource usage significantly

Solution:

Transactionally performed compaction using OPTIMIZE

```
OPTIMIZE table WHERE date = '2019-04-04'
```

Challenges solved: Performance

Challenge:

Partitioning breaks down with many dimensions and/or high cardinality columns

Solution:

Optimize using multi-dimensional clustering on multiple columns

```
OPTIMIZE conns WHERE date = '2019-04-04'
ZORDER BY (srcIP, destIP)
```

databricks

DELTA

Querying connection data at Apple

Ad-hoc query of connection data based on different columns

Connections

- date
- srcIp
- dstIp

> PBs

> trillions of rows

partitioning is bad as cardinality is high

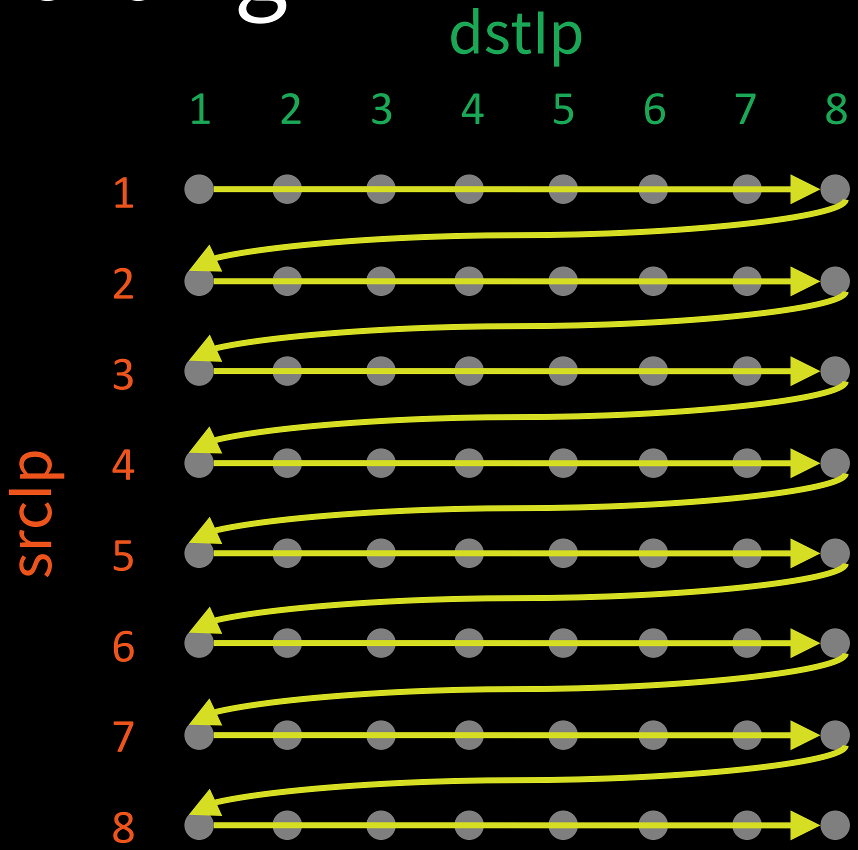
```
SELECT count(*) FROM conns
WHERE date = '2019-04-04'
AND srcIp = '1.1.1.1'
```

```
SELECT count(*) FROM conns
WHERE date = '2019-04-04'
AND dstIp = '1.1.1.1'
```

Multidimensional Sorting

```
SELECT count(*) FROM conns
WHERE date = '2019-04-04'
AND srcIp = '1.1.1.1'
```

```
SELECT count(*) FROM conns
WHERE date = '2019-04-04'
AND dstIp = '1.1.1.1'
```

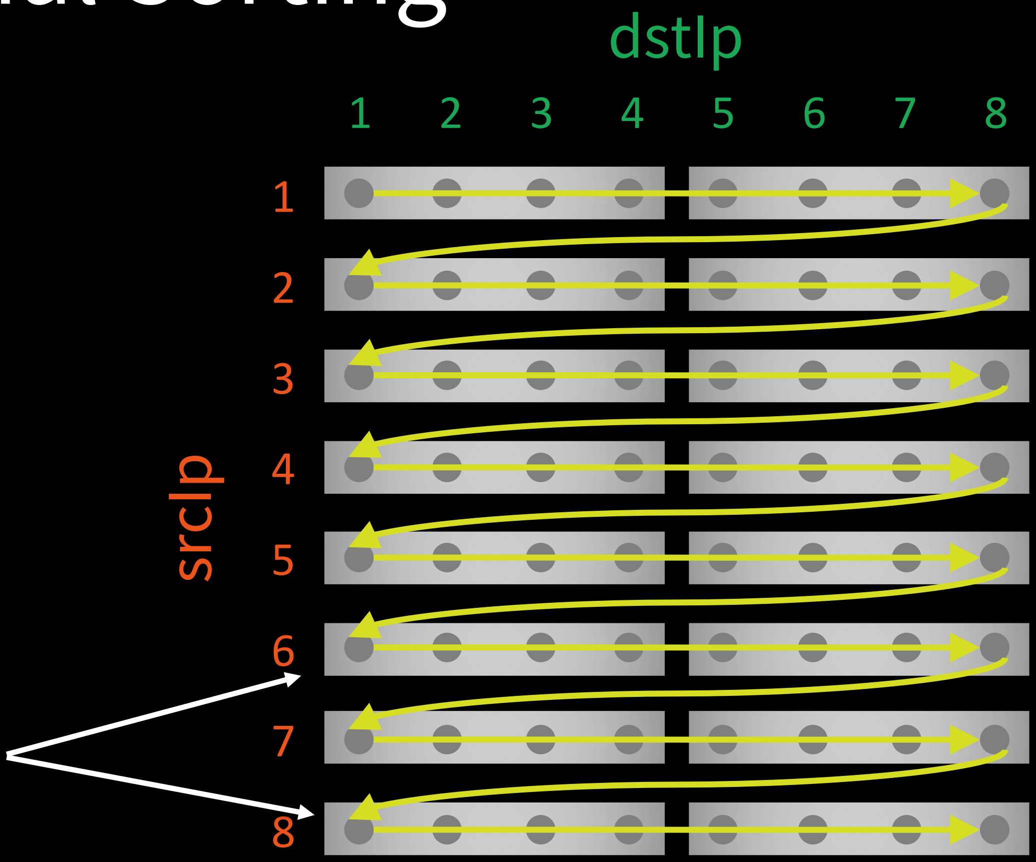


Multidimensional Sorting

```
SELECT count(*) FROM conns
WHERE date = '2019-04-04'
AND srcIp = '1.1.1.1'
```

```
SELECT count(*) FROM conns
WHERE date = '2019-04-04'
AND dstIp = '1.1.1.1'
```

ideal file size = 4 rows

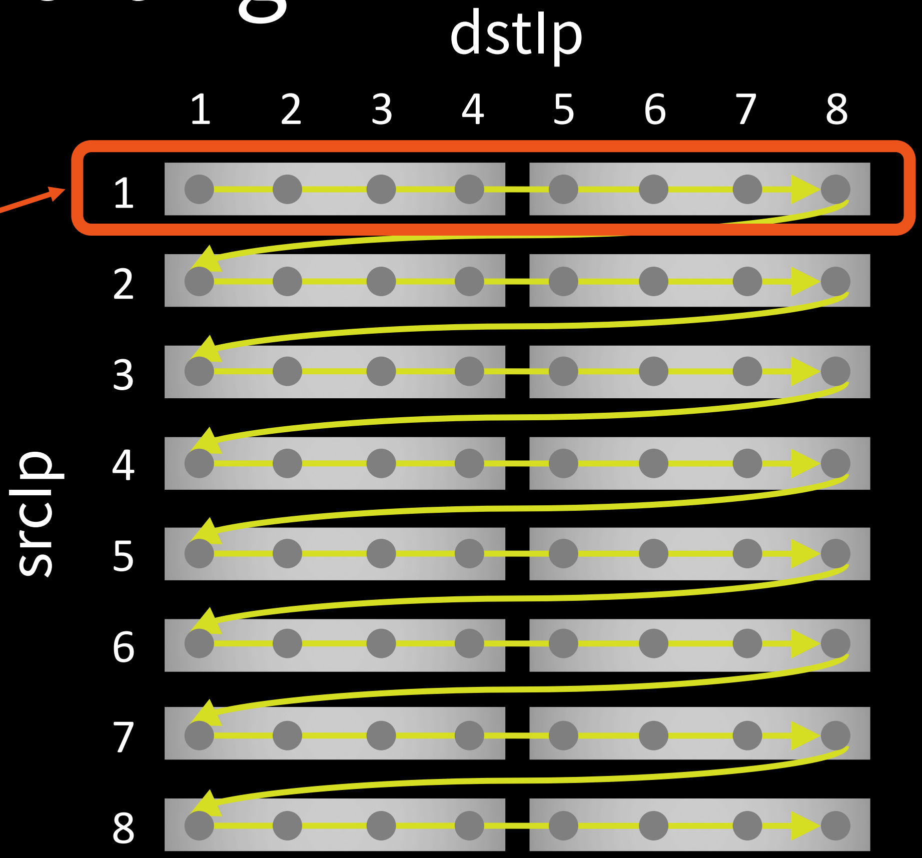


Multidimensional Sorting

```
SELECT count(*) FROM conns
WHERE date = '2019-04-04'
AND srcIp = '1.1.1.1'
```

2 files

```
SELECT count(*) FROM conns
WHERE date = '2019-04-04'
AND dstIp = '1.1.1.1'
```



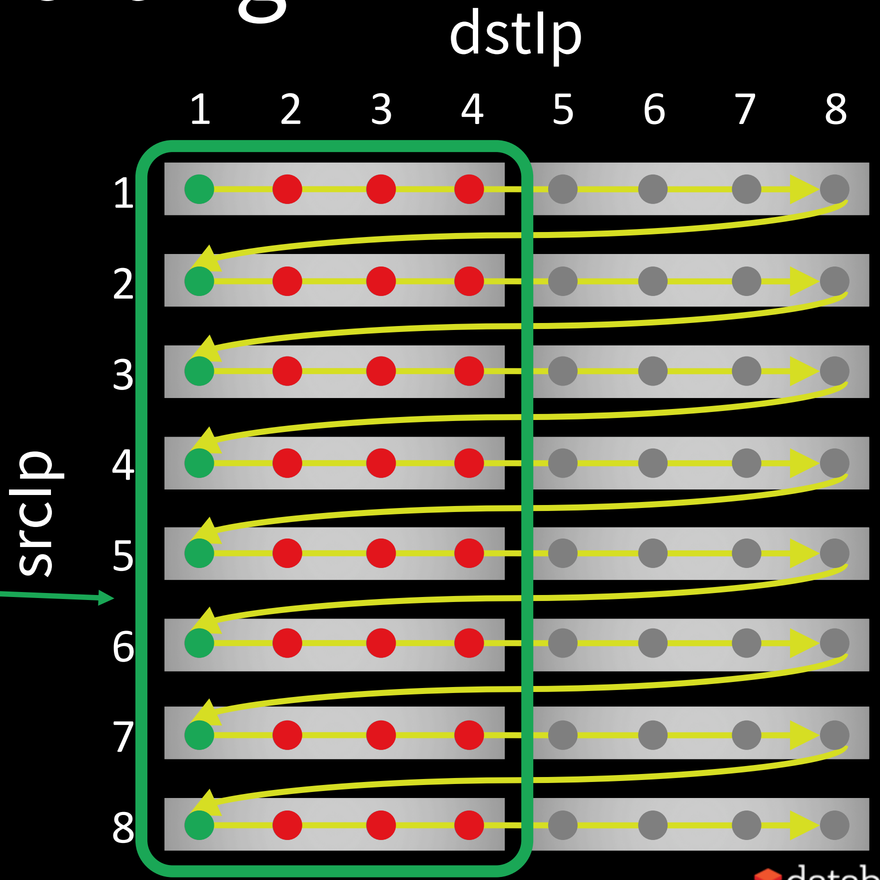
Multidimensional Sorting

```
SELECT count(*) FROM conns
WHERE date = '2019-04-04'
AND srcIp = '1.1.1.1'
```

2 files

```
SELECT count(*) FROM conns
WHERE date = '2019-04-04'
AND dstIp = '1.1.1.1'
```

8 files



great for major sorting dimension, not for others

Multidimensional Clustering

```
SELECT count(*) FROM conns
WHERE date = '2019-04-04'
AND srcIp = '1.1.1.1'
```

```
SELECT count(*) FROM conns
WHERE date = '2019-04-04'
AND dstIp = '1.1.1.1'
```

zorder space
filling curve

databricks

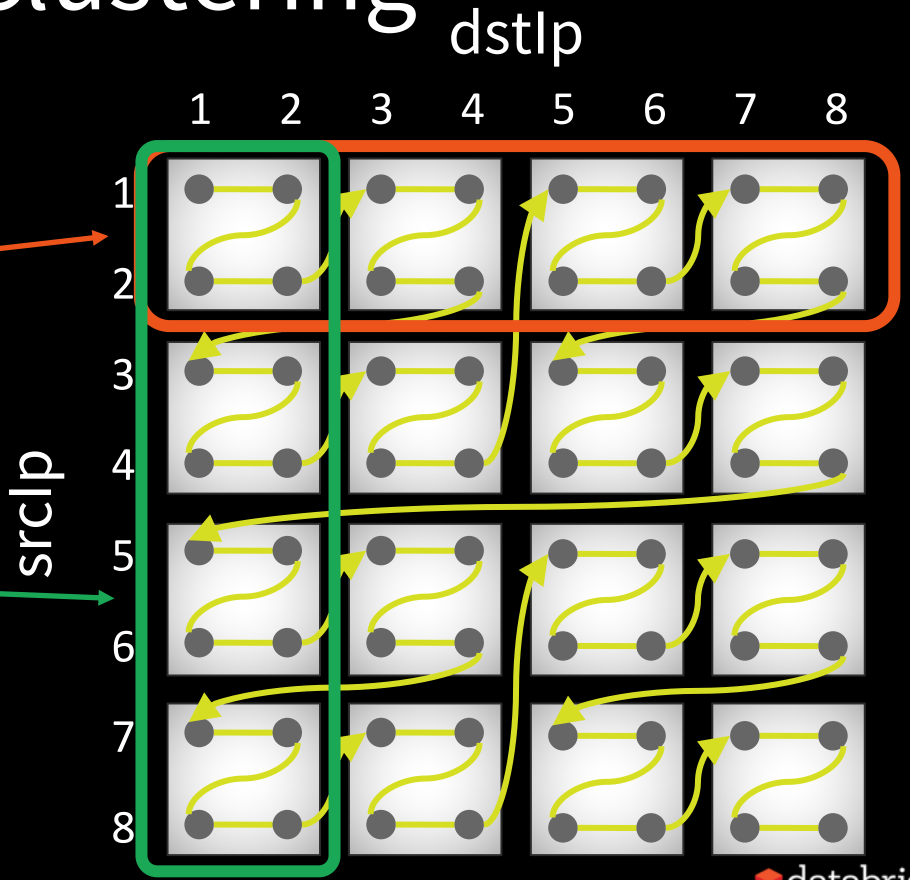
Multidimensional Clustering

```
SELECT count(*) FROM conns
WHERE date = '2019-04-04'
AND srcIp = '1.1.1.1'
```

4 files

```
SELECT count(*) FROM conns
WHERE date = '2019-04-04'
AND dstIp = '1.1.1.1'
```

4 files



reasonably good for all dimensions

databricks

Data Pipeline @ Apple

Security Infra
IDS/IPS, DLP, antivirus, load balancers, proxy servers

Cloud Infra & Apps
AWS, Azure, Google Cloud

Servers Infra
Linux, Unix, Windows

Network Infra
Routers, switches, WAPs, databases, LDAP

Detect signal across user, application and network logs

Quickly analyze the blast radius with ad hoc queries

Respond quickly in an automated fashion

Scaling across petabytes of data and 100's of security analysts

> 100TB new data/day

> 300B events/day

Data Pipeline @ Apple

Security Infra
IDS/IPS, DLP, antivirus, load balancers, proxy servers

Cloud Infra & Apps
AWS, Azure, Google Cloud

Servers Infra
Linux, Unix, Windows

Network Infra
Routers, switches, WAPs, databases, LDAP

Dump

Messy data not ready for analytics

DATALAKE1

DATALAKE2

Complex ETL

Separate warehouses for each type of analytics

DW1 Incidence Response

DW2 Alerting

DW3 Reports

> 100TB new data/day

> 300B events/day

Data Pipeline @ Apple

Security Infra
IDS/IPS, DLP, antivirus, load balancers, proxy servers

Cloud Infra & Apps
AWS, Azure, Google Cloud

Servers Infra
Linux, Unix, Windows

Network Infra
Routers, switches, WAPs, databases, LDAP

Dump

Messy data not ready for analytics

DATALAKE1

DATALAKE2

Complex ETL

Separate warehouses for each type of analytics

DW1 Incidence Response

DW2 Alerting

DW3 Reports

Took 20 engineers + 24 weeks

Hours of delay in accessing data

Very expensive to scale

Only 2 weeks of data in proprietary formats

No advanced analytics (ML)

Data Pipeline @ Apple

APACHE Spark
STRUCTURED STREAMING

databricks
DELTA

APACHE Spark
SQL, ML, STREAMING

Incidence Response

Alerting

Reports

Took 2 engineers + 2 weeks

Data usable in minutes/seconds

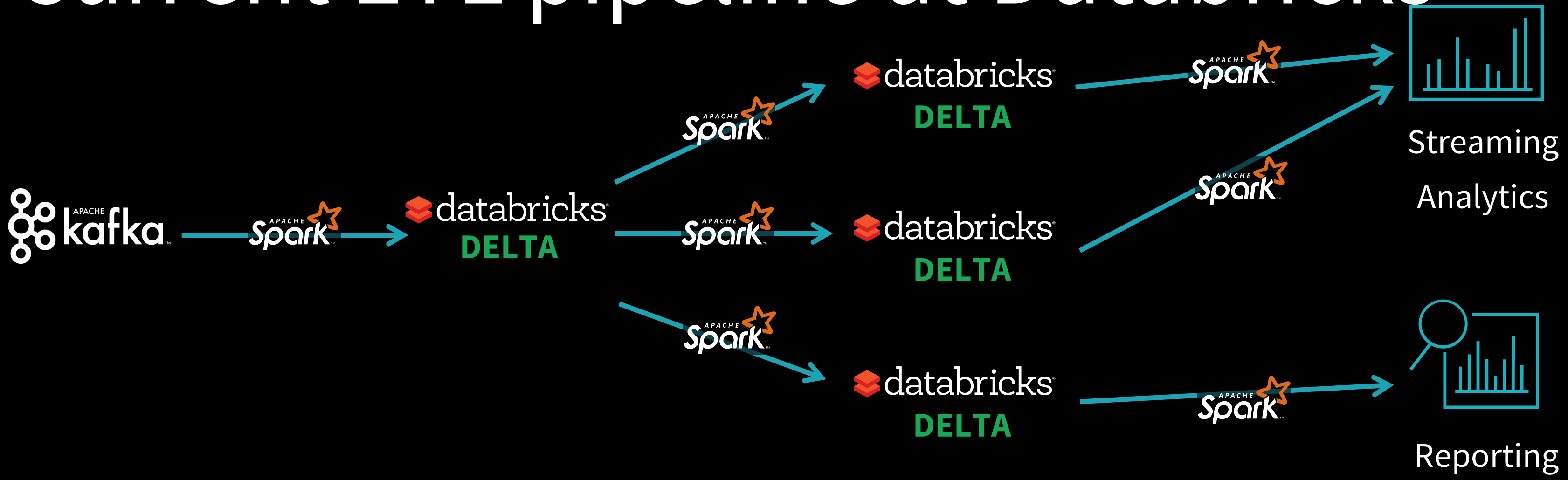
Easy and cheaper to scale

Store 2 years of data in open formats

Enables advanced analytics

KEYNOTE TALK

Current ETL pipeline at Databricks



1. ~~λ arch~~ → Not needed, Delta handles both short and long term data
2. Validation ✓ → Easy as data in short term and long term data in one location
3. Reprocessing ✓
4. Compaction ✓

→ Easy and seamless with Delta's transactional guarantees

Easy to use Delta with Spark APIs

Instead of **parquet**...

```
CREATE TABLE ...
USING parquet
...

dataframe
    .write
    .format("parquet")
    .save("/data")
```

... simply say **delta**

```
CREATE TABLE ...
USING delta
…

dataframe
    .write
    .format("delta")
    .save("/data")
```

databricks DELTA

APACHE Spark

MASSIVE SCALE	Scalable Compute & Storage
RELIABILITY	ACID Transactions & Data Validation
PERFORMANCE	Data Indexing & Caching (10-100x)
OPEN	Open source & data stored as Parquet
LOW-LATENCY	Integrated with Structured Streaming

Questions?