

Lecture 19: Cloud Data Systems (Snowflake & Databricks)

Sam Madden



Some slides from Ashish Motivala, Jiaqi Yan (Snowflake); Reynold Zin (Databricks)

Agenda

Recap: parallel query processing

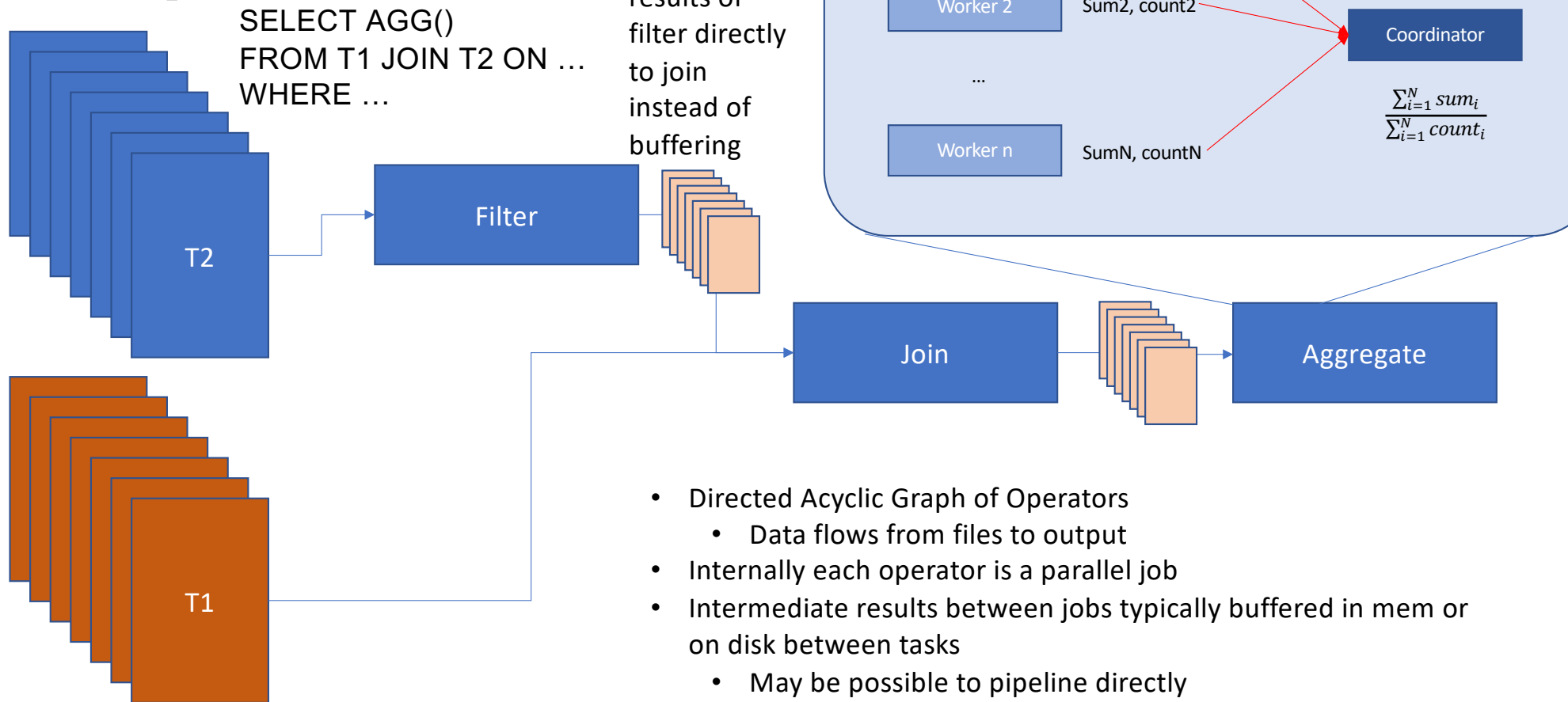
Cloud data system & data warehousing

Snowflake Architecture

Databricks Lakehouse Architecture



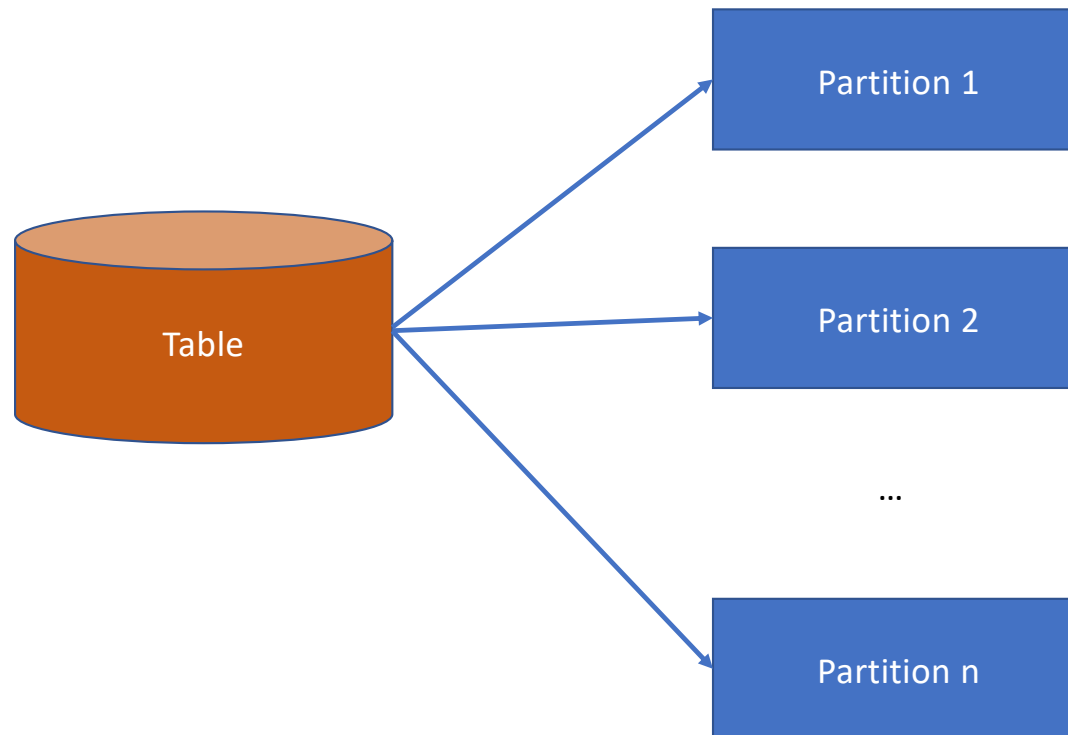
Parallel Dataflow Example



Partitioning Strategies

- Random / Round Robin
 - Evenly distributes data (no skew)
 - Requires us to repartition for joins
- Range partitioning
 - Allows us to perform joins/merges without repartitioning, when tables are partitioned on join attributes
 - Subject to skew
- Hash partitioning
 - Allows us to perform joins/merges without repartitioning, when tables are partitioned on join attributes.
 - Only subject to skew when there are many duplicate values

Round Robin Partitioning



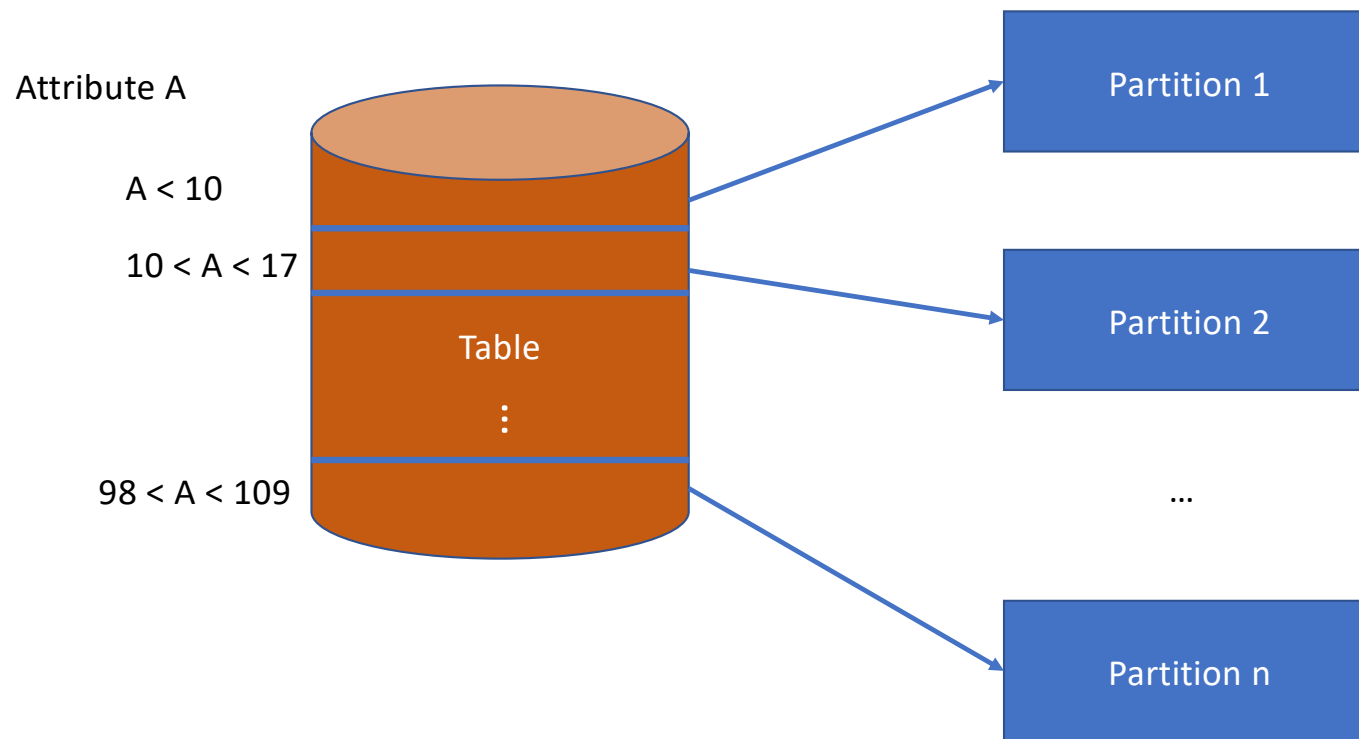
Advantages:

Each partition has the same number of records

Disadvantage:

No ability to push down predicates to filter out some partitions

Range Partitioning



Advantages:

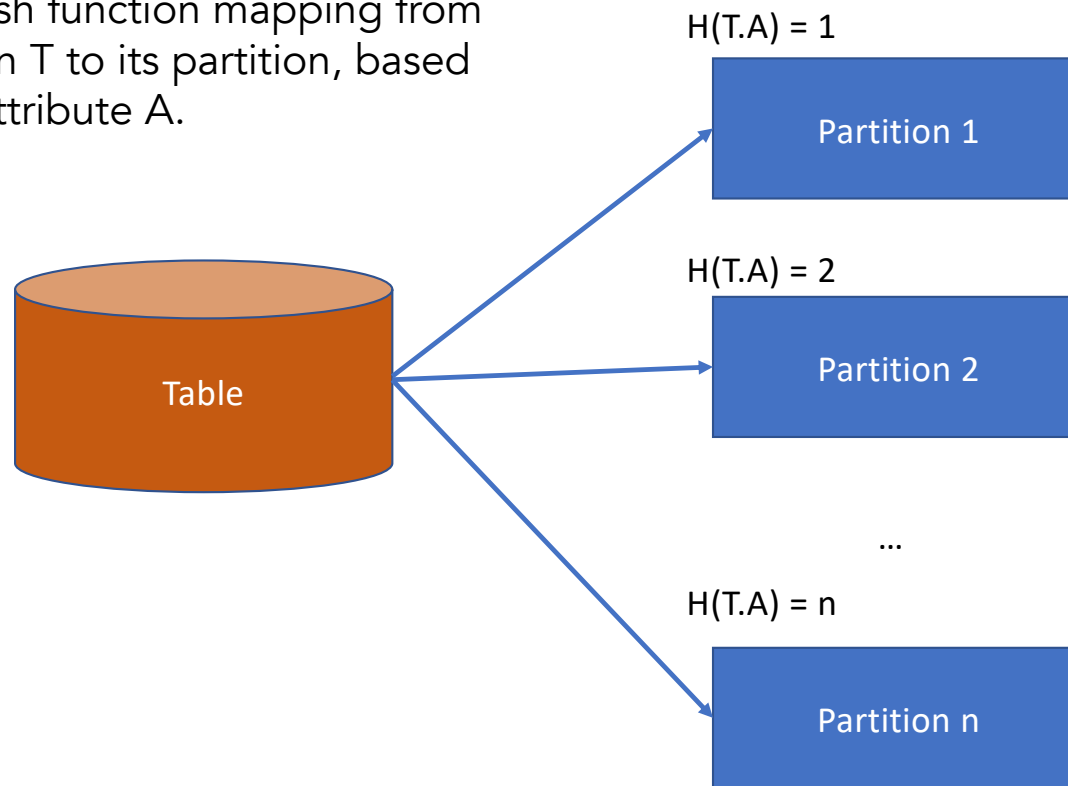
Easy to push down predicates (on partitioning attribute)

Disadvantage:

Difficult to ensure equal sized partitions, particularly in the face of inserts and skewed data

Hash Partitioning

$H(T.A)$ is a hash function mapping from each record in T to its partition, based on value of attribute A .



Advantages:

Each partition has about the same number of records, unless one value is very frequent

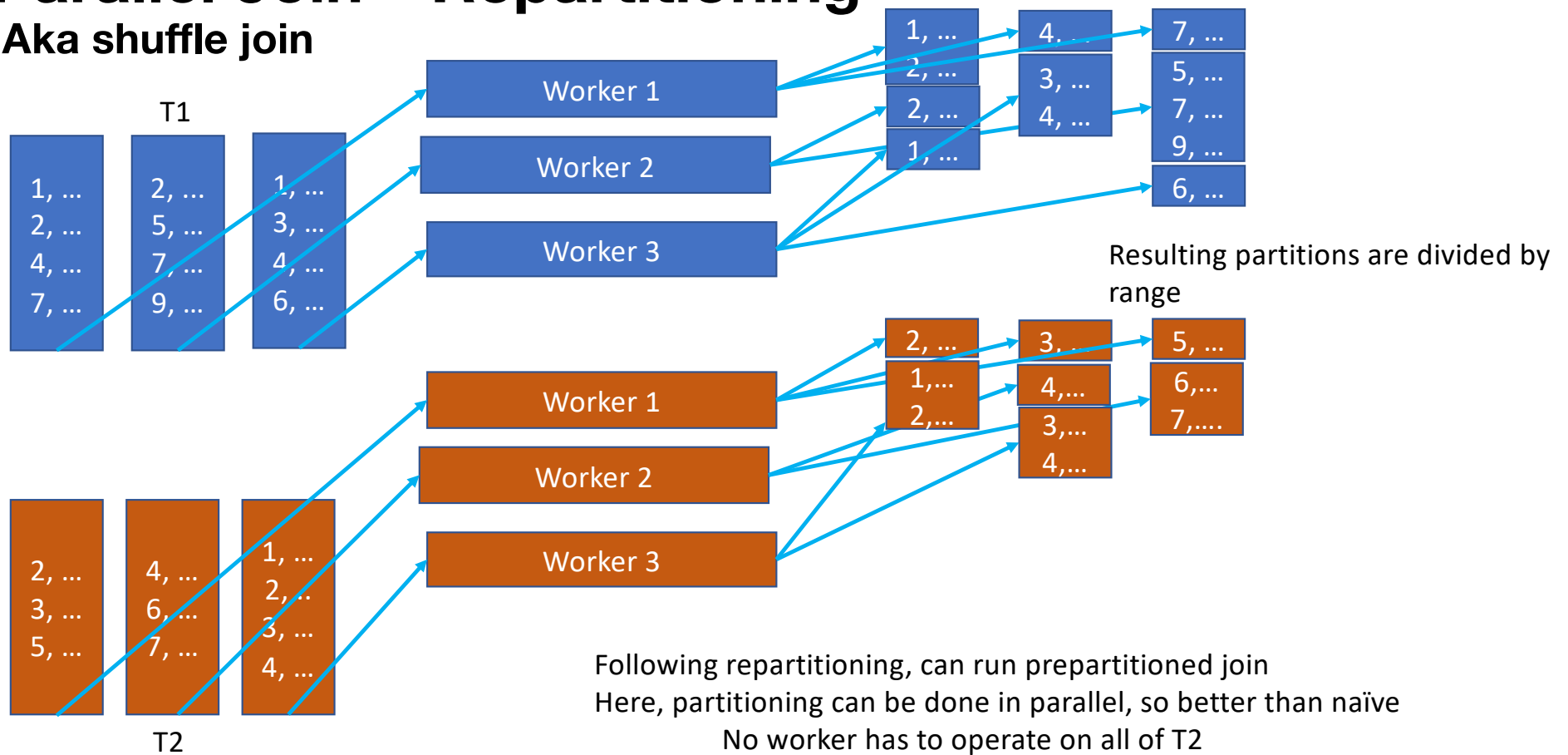
Possible to push down equality predicates on partitioning attribute

Disadvantages:

Can't push down range predicates

Parallel Join – Repartitioning

Aka shuffle join

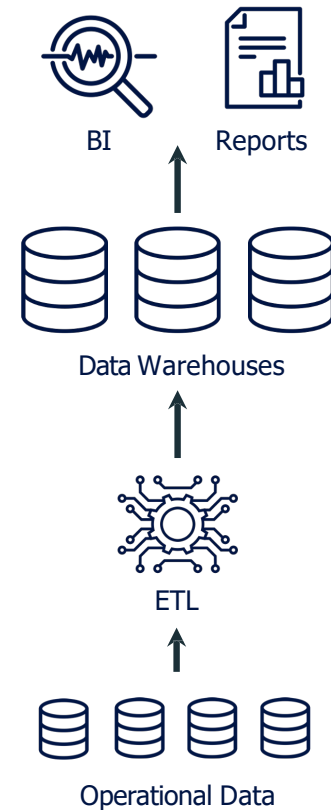


Agenda

- Recap: parallel query processing
- Cloud data system & data warehousing
- Snowflake Architecture
- Databricks Lakehouse Architecture

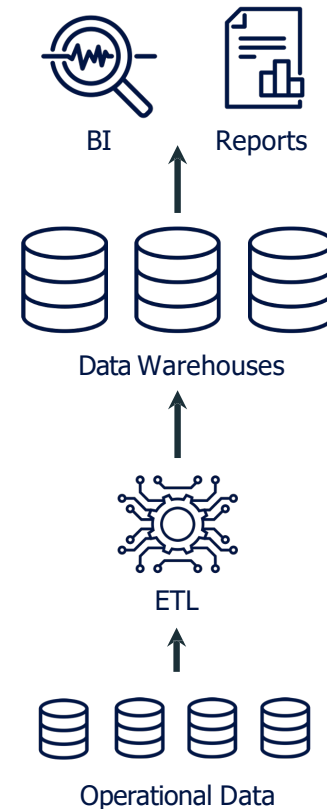
1980s: Data Warehouses

- ETL data directly from operational database systems
- Rich management and performance features for SQL analytics: schemas, indexes, transactions, etc



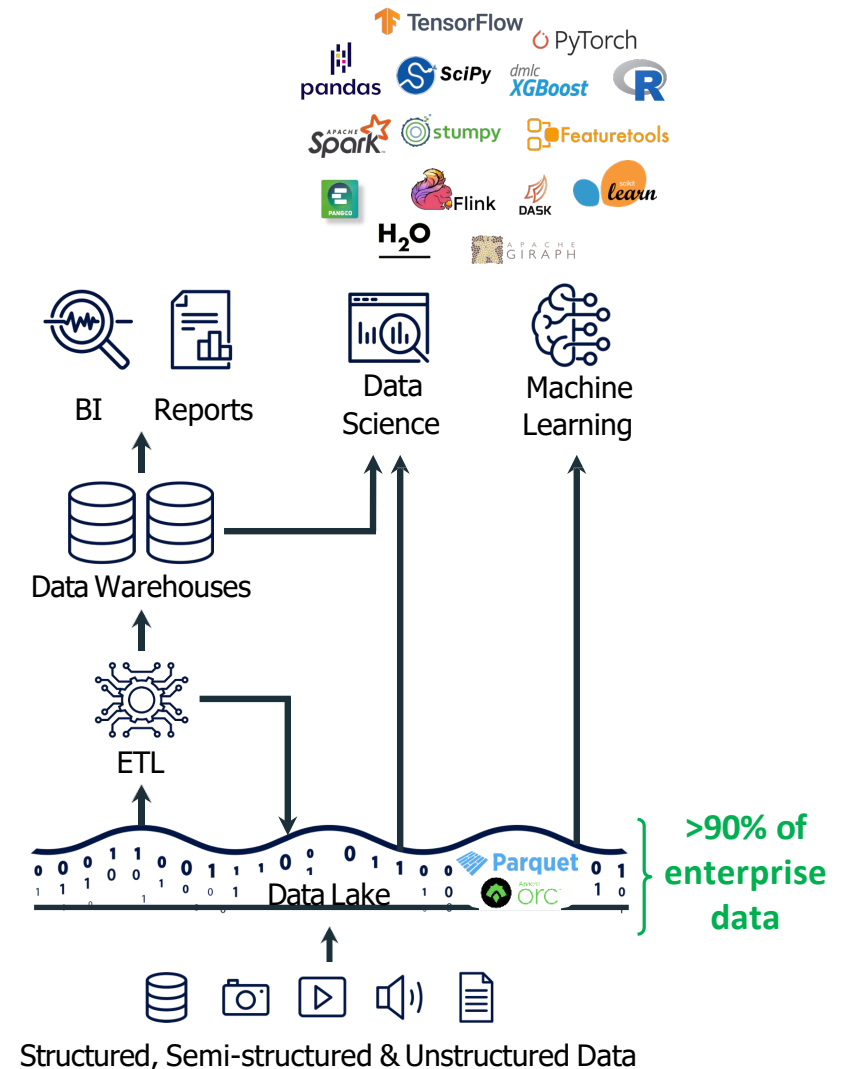
2010s: New Problems for Data Warehouses

- Could not support rapidly growing **unstructured and semi-structured data**: time series, logs, images, documents, etc
- **High cost** to store large datasets
- No support for **data science & ML**



2010s: Data Lakes

- Low-cost storage to hold all raw data with a file API (e.g. S3, HDFS)
- Open file formats (e.g. Parquet) accessible directly by ML / DS engines
- ETL jobs load specific data into warehouses, possibly for further ELT



At the Same Time: The Rise of Cloud Data Systems

- Most modern enterprises now run on the cloud
- ~70% of the commercial database market now goes to cloud vendors

How is the Cloud Different?

- Machines can be added / removed on demand
- Durable networked storage layer (e.g., Amazon S3)
 - Typically “append only”
 - Data is replicated N times
 - Very high availability
 - High latency (hundreds of ms/request) & modest BW (hundreds of MB/sec)

Agenda

- Recap: parallel query processing
- Cloud data system & data warehousing
- Snowflake Architecture
- Databricks Lakehouse Architecture

Snowflake Overview

“Elastic Data Warehouse” purpose built for the cloud

Leverages extremely reliable cloud storage (S3) for durability

“Shared disk” style design

Modern, efficient query executor



Extremely Widely Adopted

~20% of the cloud data warehouse marketshare

~\$2B annual revenue

Remarkable growth for a company founded in 2012 – in a market with some of the biggest names in technology!

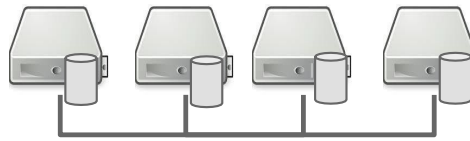


Why Cloud?

- **Amazing platform for building distributed systems**
 - Virtually unlimited, elastic compute and storage
 - Pay-per-use model (with strong economies of scale)
 - Efficient access from anywhere
- **Software as a Service (SaaS)**
 - No need for complex IT organization and infrastructure
 - Pay-per-use model
 - Radically simplified software delivery, update, and user support



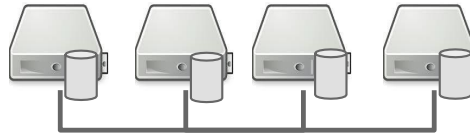
Shared-nothing Architecture



- Tables are horizontally partitioned across nodes
- Every node has its own local storage
- Every node is only responsible for its local table partitions
- Elegant and easy to reason about
- Scales well for star-schema queries
- Dominant pre-cloud architecture in data warehousing
 - Teradata, Vertica, Netezza...



The Perils of Coupling

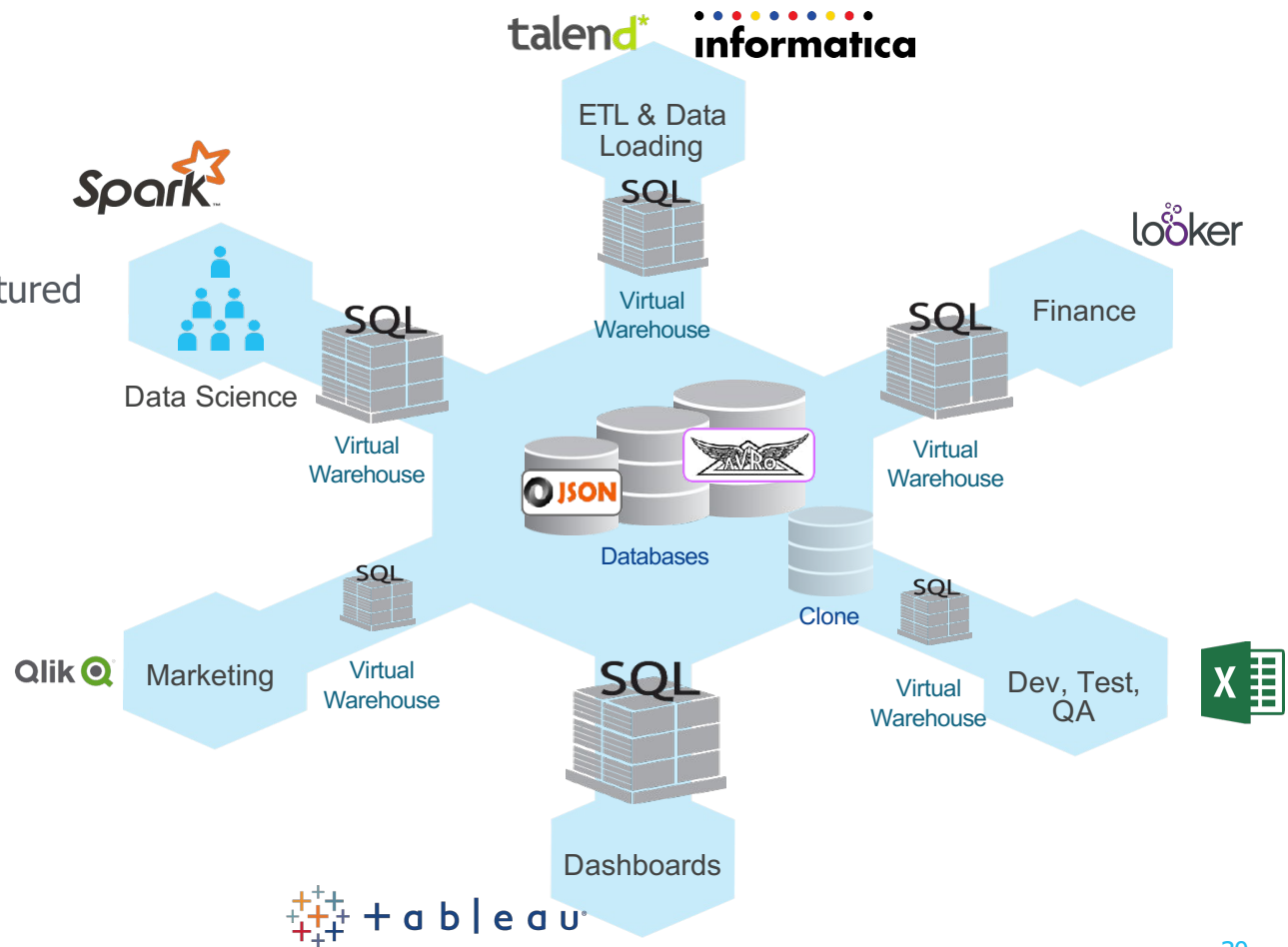


- Shared-nothing *couples* compute and storage resources
- Elasticity
 - Resizing compute cluster requires redistributing (lots of) data
 - Cannot simply shut off unused compute resources → no pay-per-use
- Limited availability
 - Membership changes (failures, upgrades) significantly impact performance and may cause downtime
- Homogeneous resources vs. heterogeneous workload
 - Bulk loading, reporting, exploratory analysis



Multi-cluster, shared data architecture

- **No data silos**
Storage decoupled from compute
- **Any data**
Native for structured & semi-structured
- **Unlimited scalability**
Along many dimensions
- **Low cost**
Compute on demand
- **Instantly cloning**
Isolate production from DEV & QA
- **Highly available**
11 9's durability, 4 9's availability



Concerns?



Photo realistic render
scary snowflake crown
fantasy elements with
the words “What could
go wrong” glowing
letters above

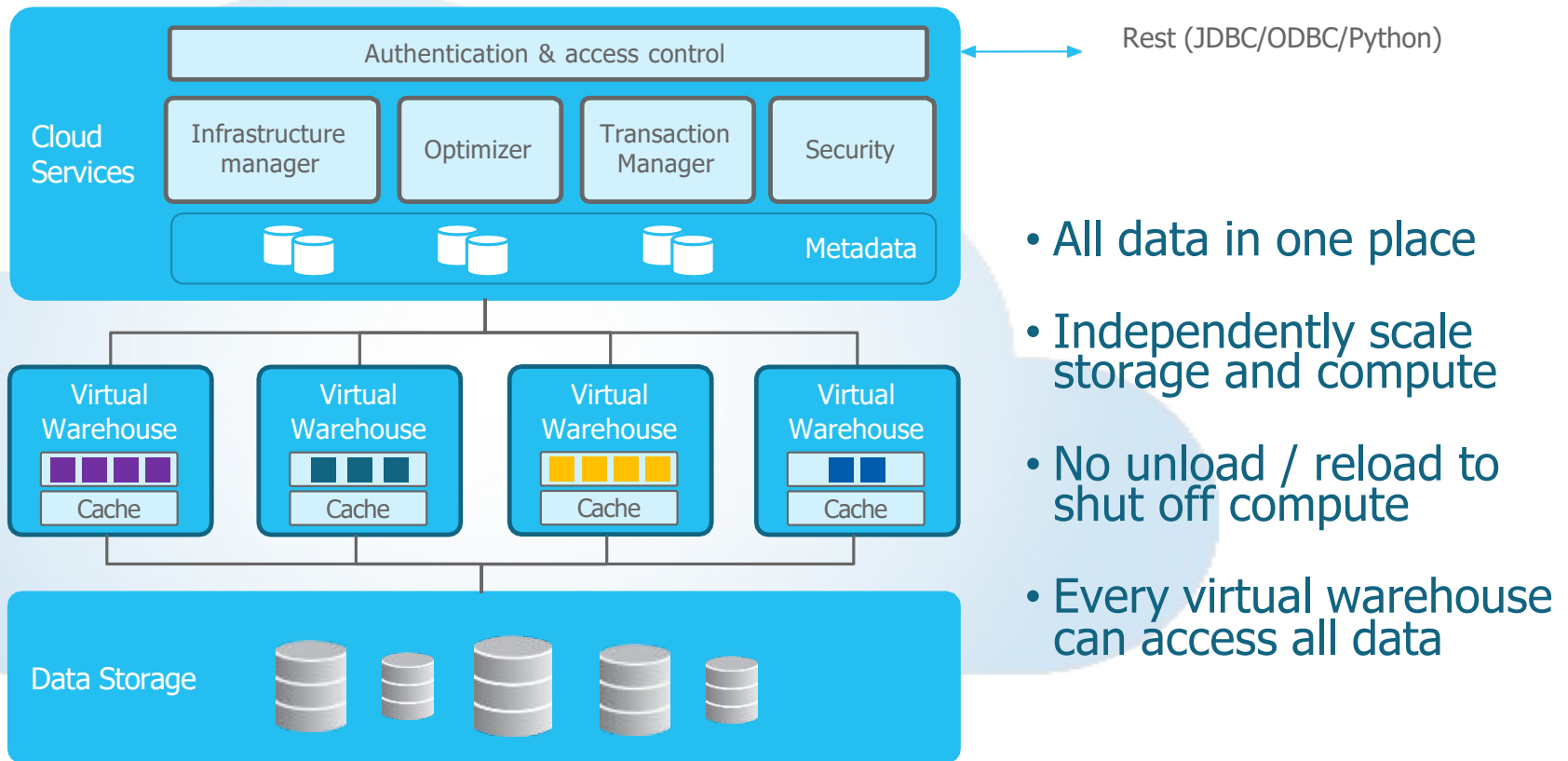
Performance?

Updates?

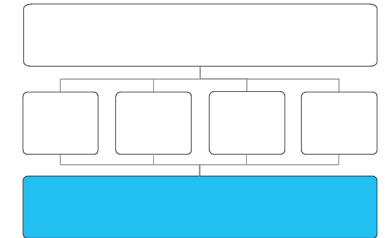
Lack of Control?



Multi-cluster Shared-data Architecture



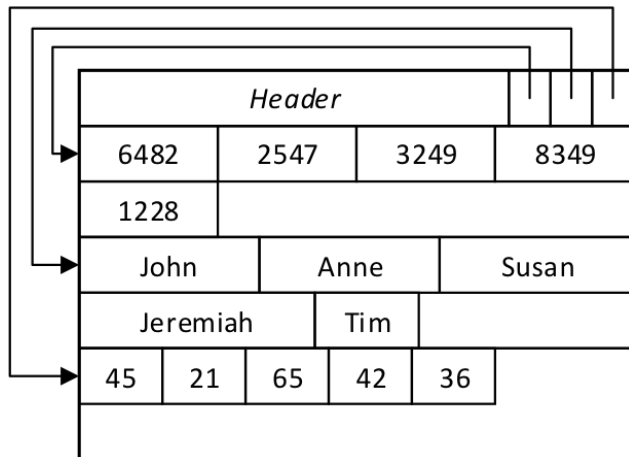
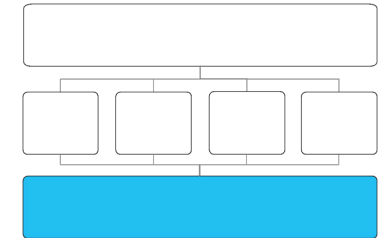
Data Storage Layer



- Stores table data and query results
 - Table is a set of immutable micro-partitions
- Uses tiered storage with Amazon S3 at the bottom
 - Object store (key-value) with HTTP(S) PUT/GET/DELETE interface
 - High availability, 3x replicated, extreme durability (11-9's)
- Some important differences w.r.t. local disks
 - Latency and BW to a single node is poor relative to disk
 - No update-in-place, objects must be written in full
 - Highly concurrent



Table Files



Not great for point updates / deletes!



- Snowflake uses PAX [Ailamaki01] aka hybrid columnar storage (similar to Parquet)

- Tables horizontally partitioned into immutable micro-partitions (~16 MB)

- Updates add or remove entire files
- Values of each column grouped together and compressed
- Queries read header + columns they need

Table Clustering

Tables can be *clustered* on a particular key

Partitions records by ranges of the key attribute, such that each micro-partition (mostly) contains a contiguous range of attributes

Clustering is lazy, not eager

Reclustering done automatically



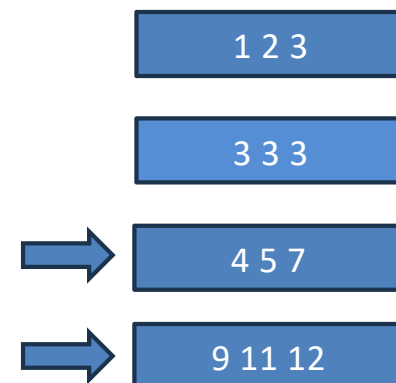
Block Skipping (“Pruning”) vs Indexing

Snowflake has no indexes - how does table clustering help?

Allows “skipping” – each partition has a min/max value, only read partitions that satisfy query.

Systems stores block metadata separately to enable this

SELECT a2 from T
WHERE a > 5



T.a metatdata

Min	Max
1	3
3	3
4	7
9	12



Block Skipping (“Pruning”) vs Indexing

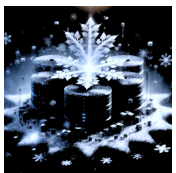
Snowflake has no indexes - how does table clustering help?

Allows “skipping” – each partition has a min/max value, only read partitions that satisfy query.

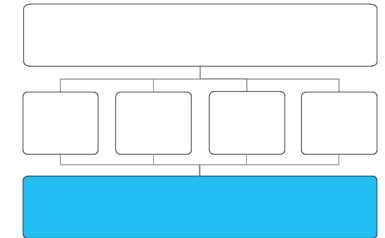
Systems stores block metadata separately to enable this

Partitions may overlap; easy to update / maintain partitions

Why not just use B+Trees + clustering?



Intermediate Data

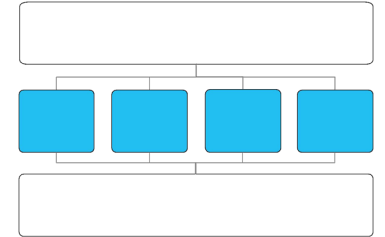


- Tiered storage also used for temp data and query results
 - Arbitrarily large queries, never run out of disk
 - New forms of client interaction
 - No server-side cursors
 - Retrieve and reuse previous query results
- Metadata stored in a transactional key-value store (not S3)
 - Which table consists of which S3 objects
 - Optimizer statistics, lock tables, transaction logs etc.
 - Part of Cloud Services layer





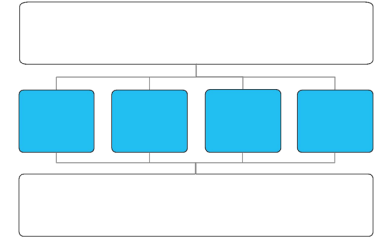
Virtual Warehouse



- Virtual warehouse = Cluster of EC2 instances – “workers”
- Pure compute resources
 - Created, destroyed, resized on demand
 - Users may run multiple warehouses at same time
 - Each warehouse has access to all data but isolated performance
 - Users may shut down *all* warehouses when they have nothing to run
- T-Shirt sizes: XS to 4XL
 - Users do not know which type or how many EC2 instances
 - Service and pricing can evolve independent of cloud platform



Worker Nodes

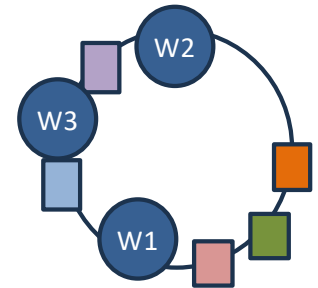


- **Worker processes are ephemeral and idempotent**
 - Worker node forks new worker process when query arrives
 - Do not modify micro-partitions directly but queue removal or addition of micro-partitions
- **Each worker node maintains local table cache**
 - Collection of table files i.e., S3 objects accessed in past
 - Shared across concurrent and subsequent worker processes
 - Assignment of micro-partitions to nodes using consistent hashing, with deterministic stealing.



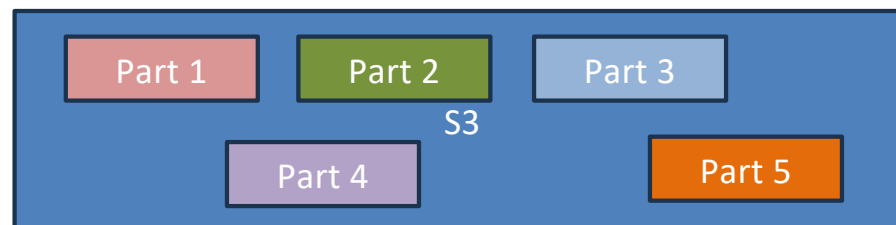
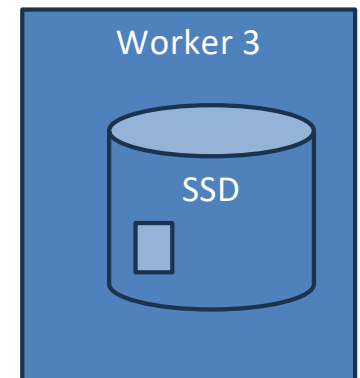
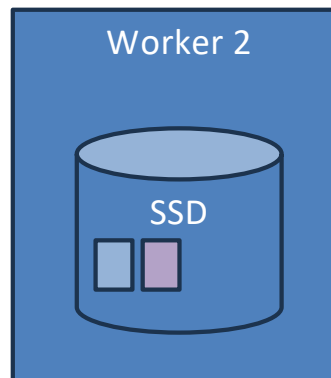
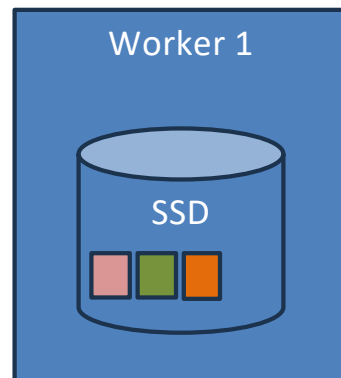
Data Affinity and Caching

Data cached on local storage; managed via LRU
Affinity between workers and partitions via consistent hashing

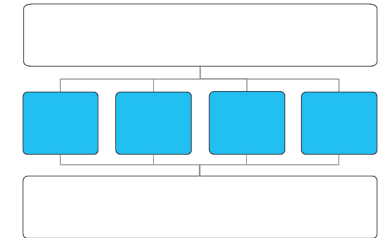


SELECT MAX(a)
FROM T

SELECT MIN(a)
FROM T



Execution Engine



- **Columnar** [MonetDB, C-Store, many more]
 - Effective use of CPU caches, SIMD instructions, and compression
- **Vectorized** [Zukowski05]
 - Operators handle batches of a few thousand rows in columnar format
 - Avoids materialization of intermediate results
- **Push-based** [Neumann11 and many before that]
 - Operators push results to downstream operators (no Volcano iterators)
 - Removes control logic from tight loops
 - Works well with DAG-shaped plans
- **No transaction management, no buffer pool**
 - But: most operators (join, group by, sort) can spill to disk and recurse
 - Queries are transactionally isolated from concurrent updates

*Uses partitioned
parallelism for query
execution*



Vectorized Execution

What's wrong with tuple at a time execution?

Alternatives:

Column/table-at-a-time?

- No pipelining
- May unnecessarily materialize intermediates, e.g.,:

```
SELECT ... WHERE sal + bonus > x
```

sal + bonus doesn't need to be stored in tuple-at-a-time

- Not great for cache locality



Vectorized Execution

Vectorized = “batch at a time”, e.g., ~ 1000 tuples

- Improves cache locality
- Avoids large intermediates
- Can be pipelined
- $\sim 1000\times$ lower functional call overhead

Picking batch size a bit difficult; what happens with very selective operators?



Boncz et al. MonetDB/X100: Hyper-Pipelining Query Execution. CIDR 2005.

Illustrative Example

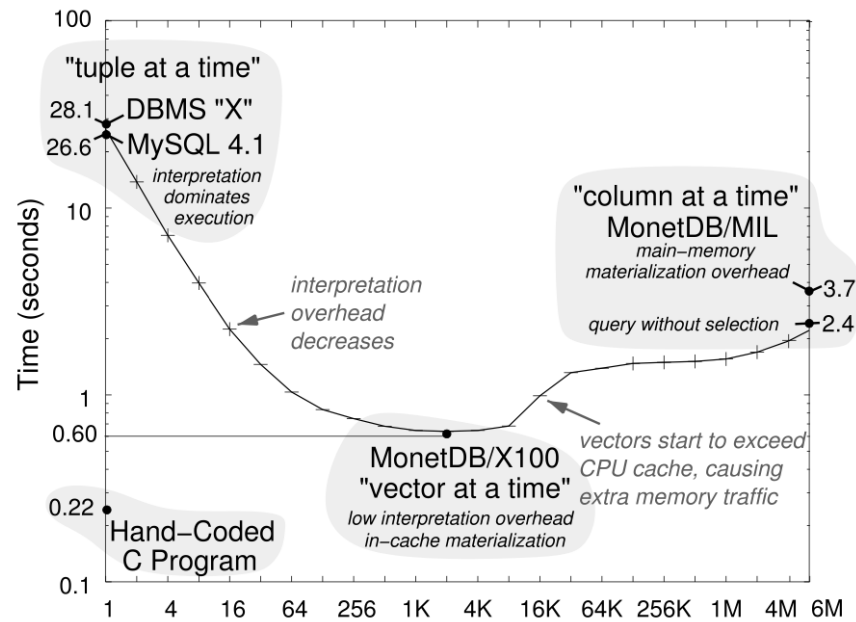


Figure 1: Hand-written code vs. execution engines for TPC-H Query 1 (Figure 3 of [16])



Neumann et al, *Efficiently Compiling Efficient Query Plans for Modern Hardware*, VLDB 2011.

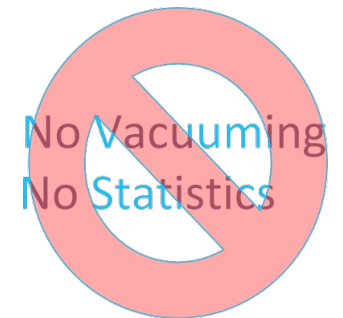
No Buffer Pool ?!

What do they mean by this?

(Presumably just that they rely on the OS cache to keep data in memory)



- Adaptive
- Self-tuning
- Do no harm!
- Automatic
- Default



Semi-Structured and Schema-Less Data

- **Three new data types:** `VARIANT`, `ARRAY`, `OBJECT`
 - `VARIANT`: holds values of any standard SQL type + `ARRAY` + `OBJECT`
 - `ARRAY`: offset-addressable collection of `VARIANT` values
 - `OBJECT`: dictionary that maps strings to `VARIANT` values
 - Like JavaScript objects or MongoDB documents
- **Self-describing, compact binary serialization**
 - Designed for fast key-value lookup, comparison, and hashing
- **Supported by all SQL operators (joins, group by, sort...)**



Post-relational Operations

- Extraction from `VARIANTS` using path syntax

```
SELECT sensor.measure.value, sensor.measure.unit
FROM sensor_events
WHERE sensor.type = 'THERMOMETER';
```

- Flattening (pivoting) a single `OBJECT` or `ARRAY` into multiple rows

```
SELECT p.contact.name.first AS "first_name",
       p.contact.name.last AS "last_name",
       (f.value.type || ': ' || f.value.contact) AS "contact"
FROM person p,
     LATERAL FLATTEN(input => p.contact) f;
```

first_name	last_name	contact
"John"	"Doe"	email: <u>john@doe.xyz</u>
"John"	"Doe"	phone: 555-123-4567
"John"	"Doe"	phone: 555-666-7777

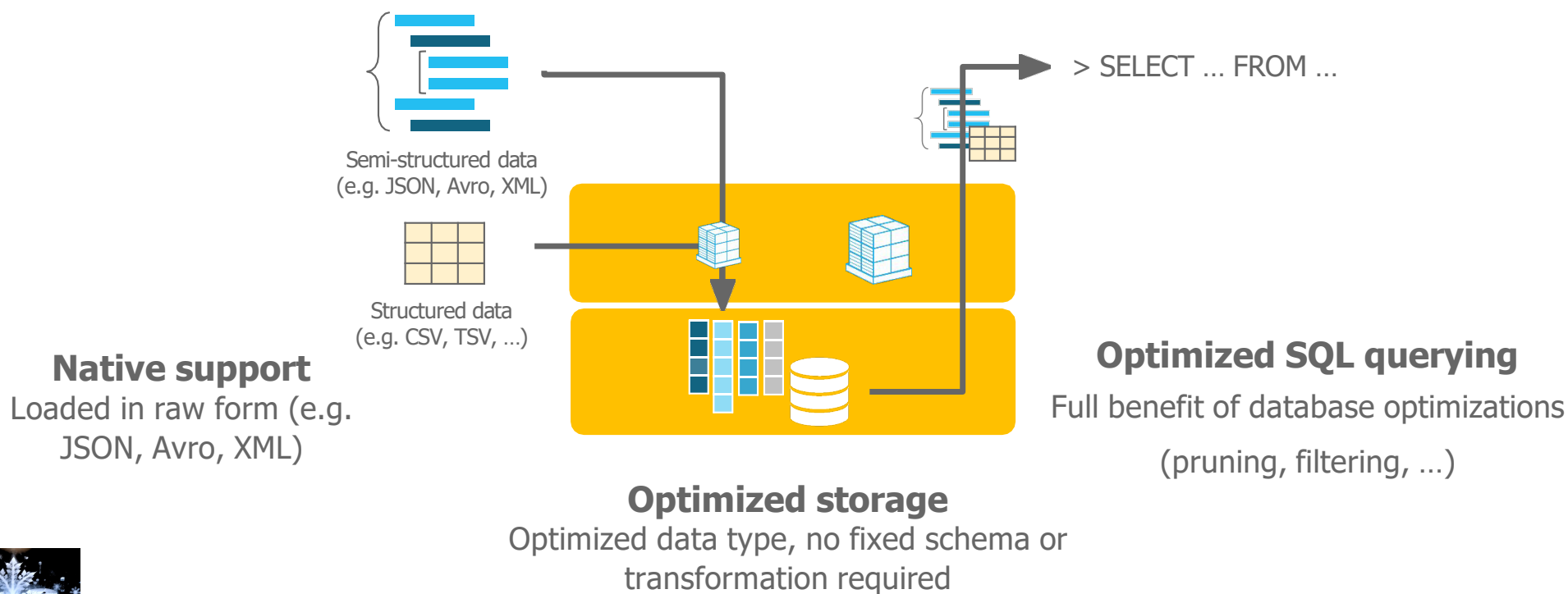


Schema-Less Data

- Cloudera Impala, Google BigQuery/Dremel
 - Columnar storage and processing of semi-structured data
 - But: full schema required up front!
- Snowflake introduces *automatic* type inference and columnar storage for *schema-less* data (VARIANT)
 - Frequently common paths are detected, projected out, and stored in separate (typed and compressed) columns in table file
 - Collect metadata on these columns for use by optimizer → pruning
 - Independent for each micro-partition → schema evolution



Automatic Columnarization of semi-structured data

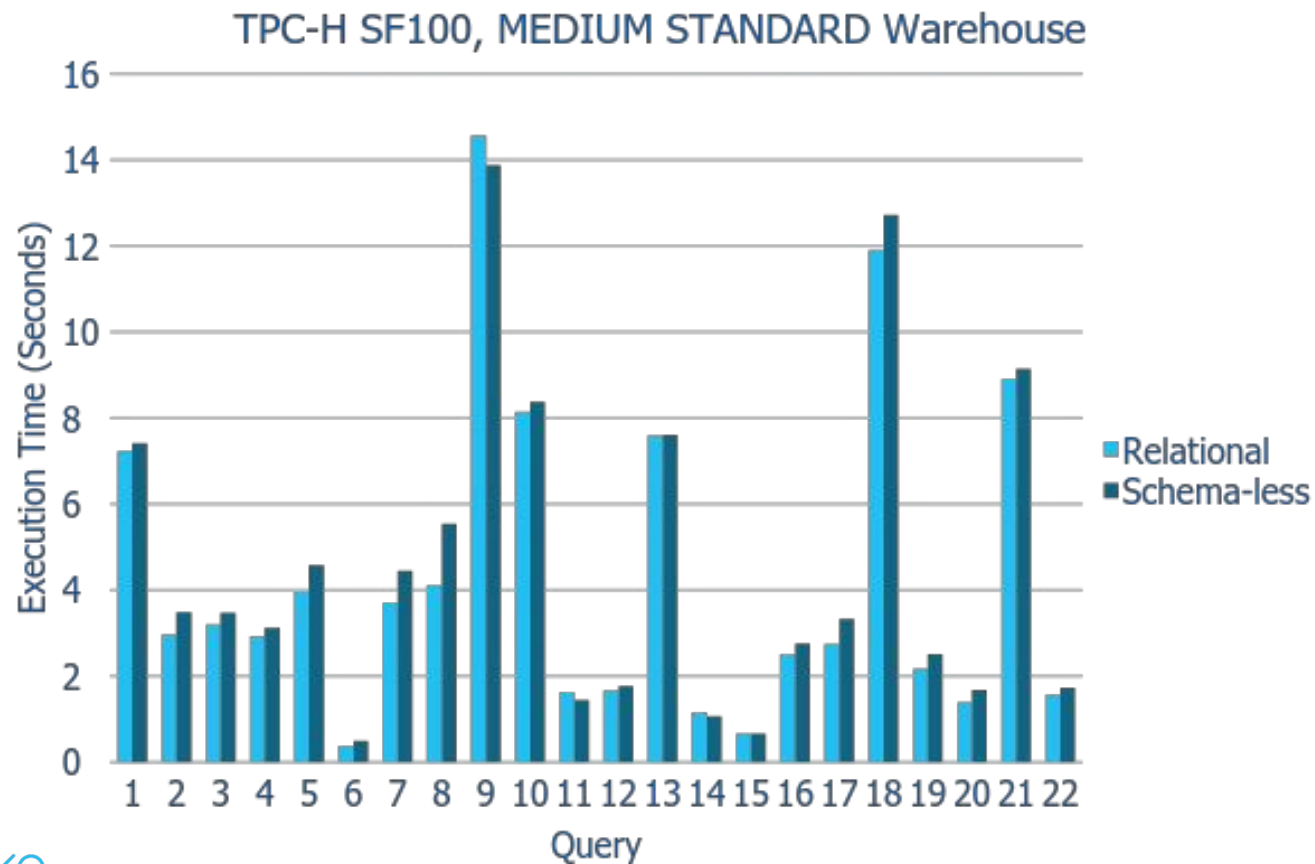




["row-acpd~3xuz-spjz", "00000000-0000-0000-8937-94B69691FF22", 0, 1700086033, null, 1700086227, null, "{ }", "1N4AZ1CP2J", "King", "Bothell", "WA", "98011", "2018", "NISSAN", "LEAF", "Battery Electric Vehicle (BEV)", "Clean Alternative Fuel Vehicle Eligible", "151", "0", "1", "239393178", "POINT (-122.20578 47.762405)", "PUGET SOUND ENERGY INC||CITY OF TACOMA - (WA)", "53033022102", "3009", "1", "1"]
["row-w97i-93tb~sstr", "00000000-0000-0000-DCF4-624CB67EA957", 0, 1700086033, null, 1700086227, null, "{ }", "JN1AZ0CP3B", "King", "Kirkland", "WA", "98034", "2011", "NISSAN", "LEAF", "Battery Electric Vehicle (BEV)", "Clean Alternative Fuel Vehicle Eligible", "73", "0", "1", "192638967", "POINT (-122.209285 47.71124)", "PUGET SOUND ENERGY INC||CITY OF TACOMA - (WA)", "53033022201", "3009", "1", "1"]
["row-ehsn~3tfd-bs3s", "00000000-0000-0000-10A3-AE8F3AF08C18", 0, 1700086033, null, 1700086227, null, "{ }", "1N4BZ0CP6H", "King", "Kirkland", "WA", "98034", "2017", "NISSAN", "LEAF", "Battery Electric Vehicle (BEV)", "Clean Alternative Fuel Vehicle Eligible", "107", "0", "45", "145384389", "POINT (-122.209285 47.71124)", "PUGET SOUND ENERGY INC||CITY OF TACOMA - (WA)", "53033021904", "3009", "1", "1"]
["row-zppk.fdn2_qd4h", "00000000-0000-0000-41D2-A4A4D5AD86D3", 0, 1700086033, null, 1700086227, null, "{ }", "3C3CFFGE4F", "Snohomish", "Lake Stevens", "WA", "98258", "2015", "FIAT", "500", "Battery Electric Vehicle (BEV)", "Clean Alternative Fuel Vehicle Eligible", "87", "0", "44", "292932748", "POINT (-122.112265 48.0047)", "PUGET SOUND ENERGY INC", "53061052706", "3213", "1", "45"]
["row-744h.m6bd.ijbn", "00000000-0000-0000-A6A8-EE9BE4A46669", 0, 1700086033, null, 1700086227, null, "{ }", "5YJ3E1EA5K", "Kitsap", "Bainbridge Island", "WA", "98110", "2019", "TESLA", "MODEL 3", "Battery Electric Vehicle (BEV)", "Clean Alternative Fuel Vehicle Eligible", "220", "0", "23", "214907430", "POINT (-122.5235781 47.6293323)", "PUGET SOUND ENERGY INC", "53035090700", "848", "6", "29"]
["row-a7d4_x75h_wbhn", "00000000-0000-0000-D8AB-ED869B6FFA74", 0, 1700086033, null, 1700086227, null, "{ }", "JTDKARFP3K", "Thurston", "Olympia", "WA", "98501", "2019", "TOYOTA", "PRIUS PRIME", "Plug-in Hybrid Electric Vehicle (PHEV)", "Not eligible due to low battery range", "25", "0", "22", "289260751", "POINT (-122.89692 47.043535)", "PUGET SOUND ENERGY INC", "53067010400", "2742", "10", "28"]
["row-bvdk.tugx.mth2", "00000000-0000-0000-9C08-04D22DB8DB21", 0, 1700086033, null, 1700086227, null, "{ }", "5YJYGDEEXM", "King", "Seattle", "WA", "98102", "2021", "TESLA", "MODEL Y", "Battery Electric Vehicle (BEV)", "Eligibility

<u>Make</u>	<u>Model</u>
NISSAN	LEAF
NISSAN	LEAF
NISSAN	LEAF
FIAT	500
TESLA	MODEL 3
TOYOTA	PRIUS PRIME
TESLA	MODEL Y

Schema-Less Performance



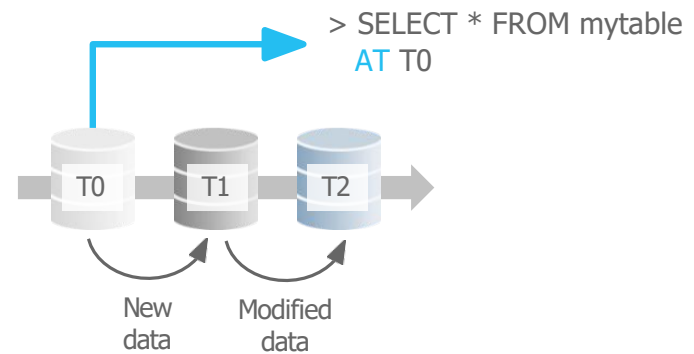
ETL vs. ELT

- ETL = Extract-Transform-Load
 - Classic approach: extract from source systems, run through some transformations (perhaps using Hadoop), then load into relational DW
- ELT = Extract-Load-Transform (“transform on demand”)
 - Schema-later or schema-never: extract from source systems, leave in or convert to JSON or XML, load into DW, transform there if desired
 - Decouples information producers from information consumers
- Snowflake: ELT with speed and expressiveness of RDBMS



Time Travel and Cloning

- Previous versions of data automatically retained
 - Same metadata as Snapshot Isolation
- Accessed via SQL extensions
 - UNDROP recovers from accidental deletion
 - SELECT AT for point-in-time selection
 - CLONE [AT] to recreate past versions



Lessons Learned

- Building a relational DW was a controversial decision in 2012
 - But turned out correct; Hadoop did not replace RDBMSs
- Multi-cluster, shared-data architecture game changer for org
 - Business units can provision warehouses on-demand
 - Fewer data silos
 - Dramatically lower load times and higher load frequency
- Semi-structured extensions were a bigger hit than expected
 - People use Snowflake to replace Hadoop clusters



Lessons Learned (2)

- SaaS model dramatically helped speed of development
 - Only one platform to develop for
 - Every user running the same version
 - Bugs can be analyzed, reproduced, and fixed very quickly
- Users love “no tuning” aspect
 - But creates continuous stream of hard engineering challenges...
- Core performance less important than anticipated
 - Elasticity matters more in practice



Snowflake Summary

- Snowflake is a cloud-native data warehouse as a service
 - Novel multi-cluster, shared-data architecture
 - Highly elastic and available
 - Semi-structured and schema-less data at the speed of relational data



Agenda

Recap: parallel query processing

Cloud data system & data warehousing

Snowflake Architecture

Databricks Lakehouse Architecture



Problems with Data Lakes

Cheap to store all the data, but the 2-tier architecture is much more complex!

Data reliability suffers:

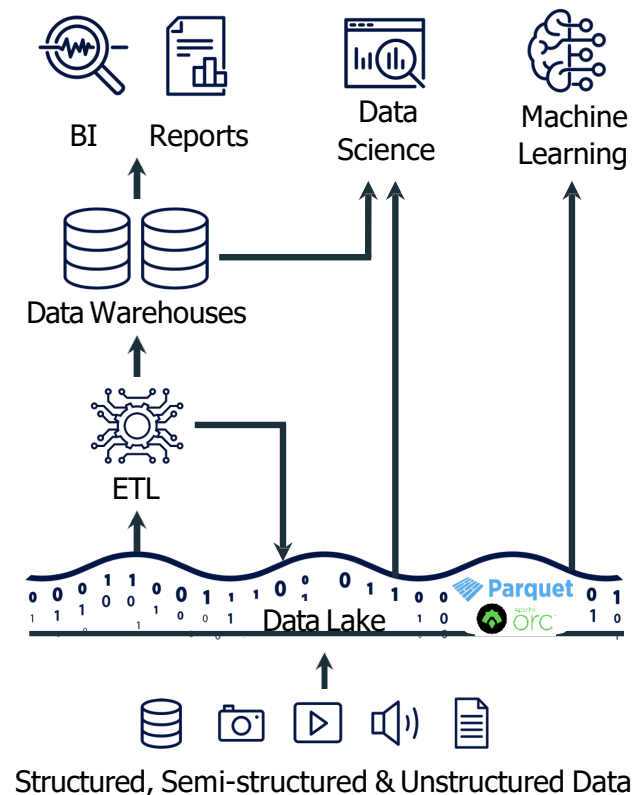
- Multiple storage systems with different semantics, SQL dialects, etc
- Extra ETL steps that can go wrong

Timeliness suffers:

- Extra ETL steps before data available in DW

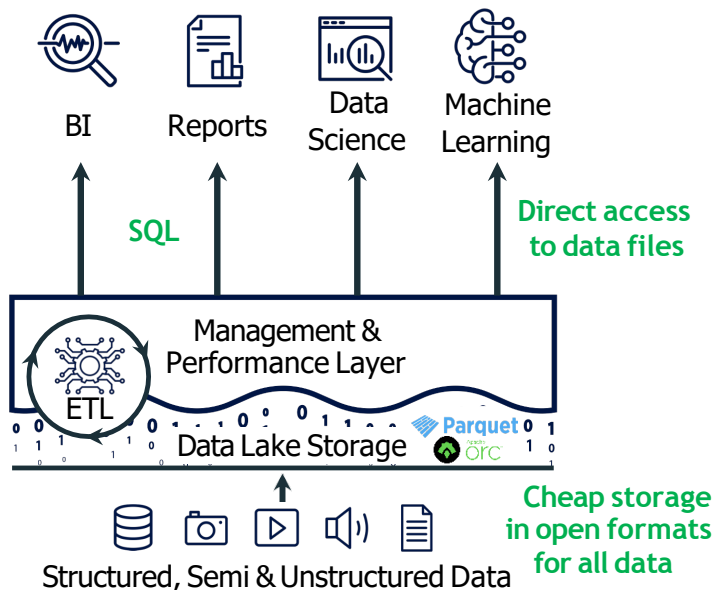
High cost:

- Continuous ETL, duplicated storage



Lakehouse Systems

Implement data warehouse management and performance features on top of **directly-accessible data in open formats**



Can we get state-of-the-art performance & governance features with this design?

**Snowflake puts this all in the DBMS;
Databricks advocates using Spark to do this**

Lakehouse Technology

New techniques to provide data warehousing features directly on data lake storage

- Retain existing open file formats (e.g. Apache Parquet, ORC)
- Add management and performance features on top (transactions, data versioning, indexes, etc)
- Can also help eliminate other data systems, e.g. message queues

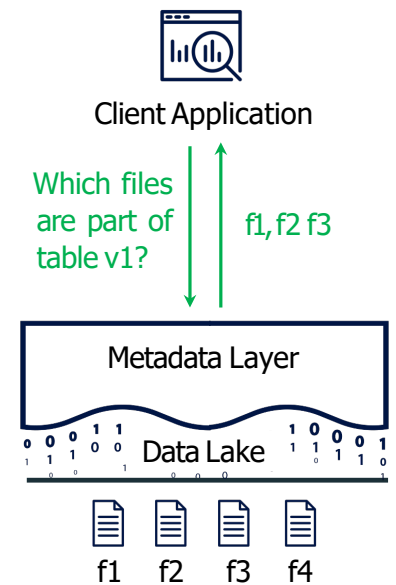
Key Technologies Enabling Lakehouse

1. **Metadata layers on data lakes:** add transactions, versioning & more
2. **Lakehouse engine designs:** performant SQL on data lake storage
3. **Declarative I/O interfaces** for data science & ML

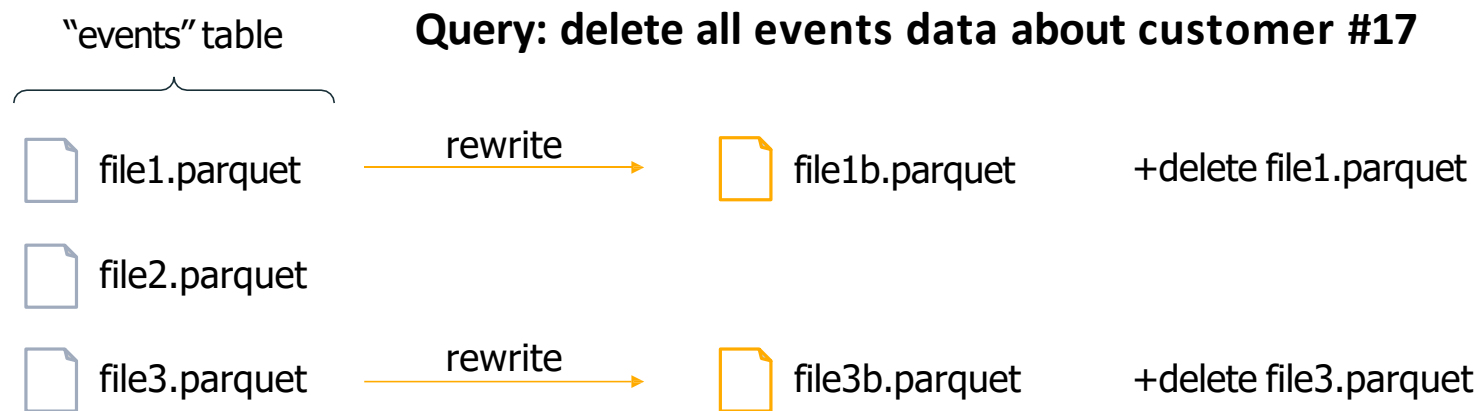
Metadata Layers on Data Lakes

- Track **which files** are part of a table version to offer rich management features like transactions
Clients can then access the underlying files at high speed

- Examples:

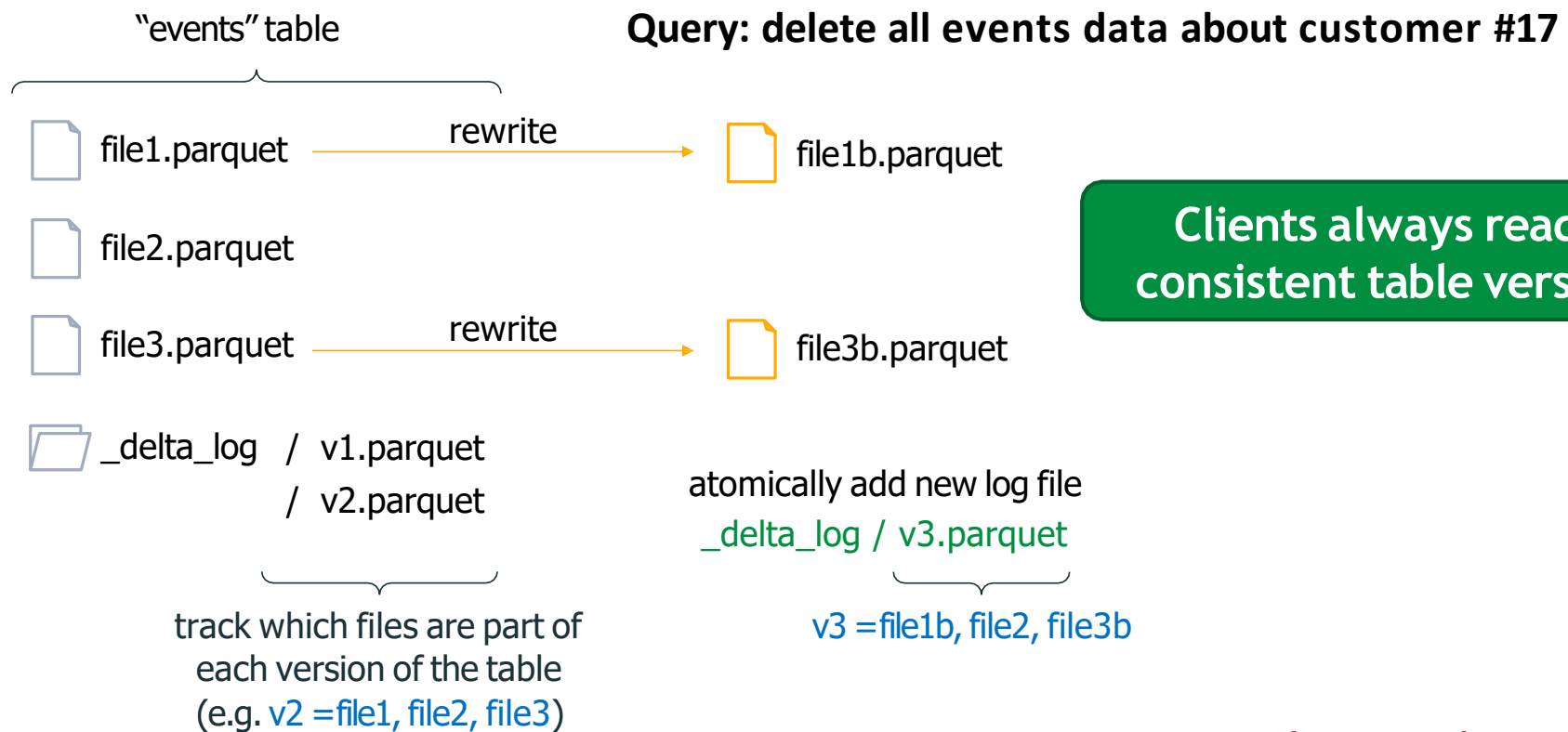


Example: Traditional Data Lake



Problem: What if a query reads the table while the delete is running?

Example: DELTA LAKE



Other Management Features with DELTA LAKE

- Time travel to old table versions
- Zero-copy CLONE by forking the log
- DESCRIBE HISTORY
- Schema enforcement & constraints

```
SELECT * FROM my_table  
TIMESTAMP AS OF "2020-05-01"
```

```
CREATE TABLE my_table_dev  
SHALLOW CLONE my_table
```

1 DESCRIBE HISTORY flightdelays

▶ (1) Spark Jobs

version ▼	timestamp	userid ▼	userName ▼	operation ▼	notebook ▼
7	2019-10-08T16:47:22	101543	...@databricks.com	MERGE	▶ {"notebookId":"25"}
6	2019-10-08T16:44:16	101543	...@databricks.com	MERGE	▶ {"notebookId":"25"}
5	2019-10-06T19:26:53	101543	...@databricks.com	UPDATE	▶ {"notebookId":"25"}

Key Technologies Enabling Lakehouse

1. **Metadata layers on data lakes:** add transactions, versioning & more
2. **Lakehouse engine designs:** performant SQL on data lake storage
3. **Declarative I/O interfaces** for data science & ML

The Challenge

- Most data warehouses have full control over the data storage system and query engine, so they design them together
- The key idea in a Lakehouse is to store data in **open** storage formats (e.g. Parquet) for direct access from many systems
- How can we get great performance with these standard, open formats?

Enabling Lakehouse Performance

Even with a fixed, directly-accessible storage format, 4 optimizations help:

- **Auxiliary data structures** like statistics and indexes
 - **Data layout** optimizations within files
 - **Caching** hot data in a fast format
 - **Execution optimizations** like vectorization
- } Minimize I/Os for cold data
- } Match DW performance on hot data

New query engines such as Databricks Photon Engine use these ideas

Optimization 1: Auxiliary Data Structures

- Even if the base data is in Parquet, we can build other data structures to speed up queries, and maintain them transactionally
- **Example:** min/max zone maps for data skipping

 file1.parquet

year: min 2018, max 2019
uid: min 12000, max 23000

 file2.parquet

year: min 2018, max 2020
uid: min 12000, max 14000

 file3.parquet

year: min 2020, max 2020
uid: min 23000, max 25000

updated transactionally
with Delta table log

Query: `SELECT * FROM events
WHERE year=2020 AND uid=24000`

Optimization 1: Auxiliary Data Structures

- Even if the base data is in Parquet, we can build other data structures to speed up queries, and maintain them transactionally
- **Example:** min/max zone maps for data skipping

 file1.parquet

year: min 2018, max 2019
uid: min 12000, max 23000

 file2.parquet

year: min 2018, max 2020
uid: min 12000, max 14000

 file3.parquet

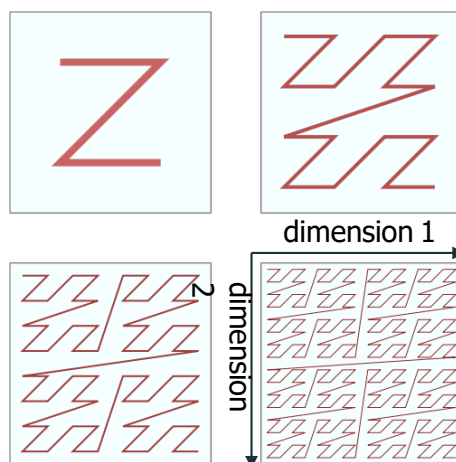
year: min 2020, max 2020
uid: min 23000, max 25000

updated transactionally
with Delta table log

Query: SELECT * FROM events
WHERE year=2020 AND uid=24000

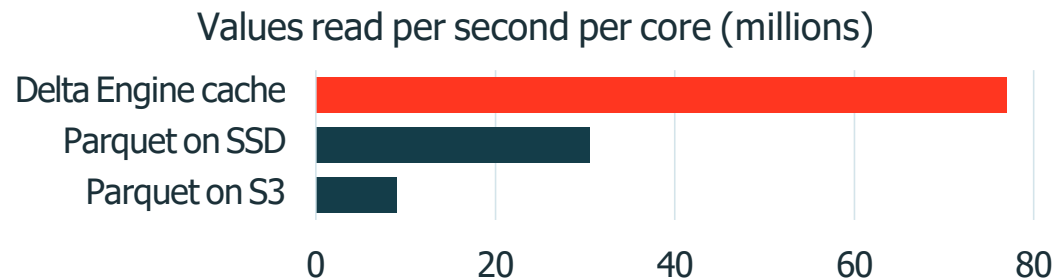
Optimization 2: Data Layout

- Even with a fixed storage format such as Parquet, we can optimize the data layout *within* tables to minimize I/O
- **Example:** Z-order sorting for multi-dimensional clustering



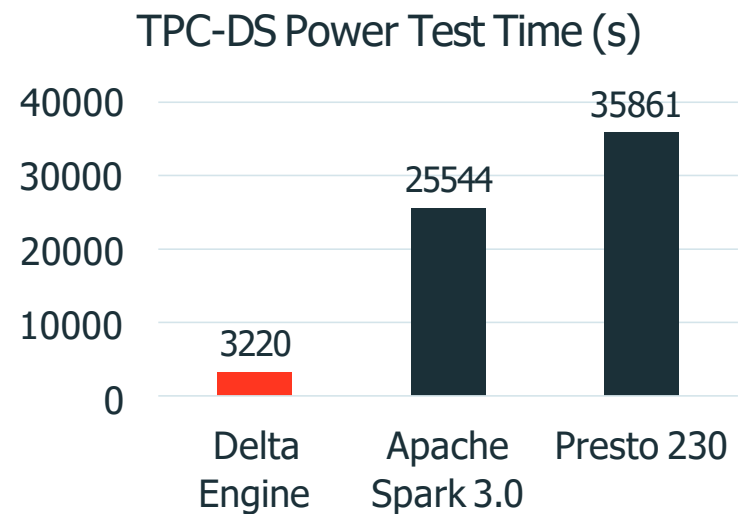
Optimization 3: Caching

- Most data warehouses cache hot data in SSD or RAM
- Can do the same in Lakehouse, using the metadata layer for consistency
- **Example: SSD cache in Photon Engine**



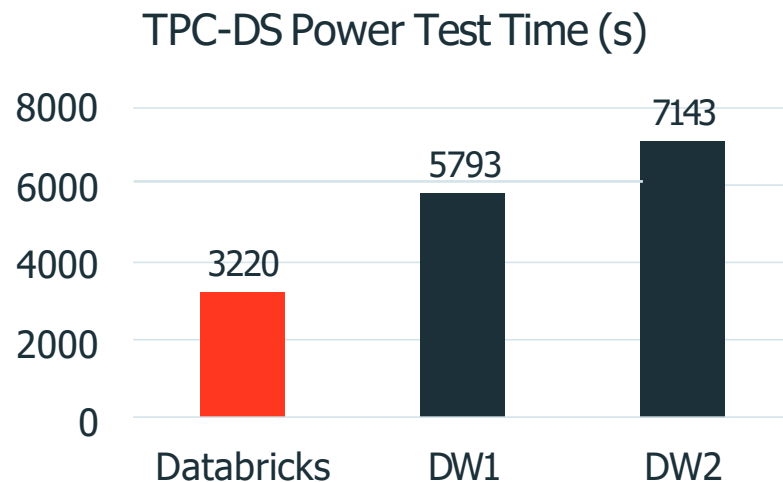
Optimization 4: Vectorized Execution

- Many existing ideas can also be applied over open formats like Parquet
- **Example:** Databricks Photon vectorized engine



Putting These Ideas Together

Lakehouse engines can match DW performance on either hot or cold data!



Databricks Sets Official Data Warehousing Performance Record



by Reynold Xin and Mostafa Mokhtar

Posted in COMPANY BLOG | November 2, 2021

Today, we are proud to announce that **Databricks SQL** has set a **new world record in 100TB TPC-DS**, the gold standard performance benchmark for data warehousing. **Databricks SQL outperformed the previous record by 2.2x**. Unlike most other benchmark news, this result has been formally audited and reviewed by the TPC council.

Summary

Both Databricks and Snowflake offer "cloud-native" data solutions.

Both employ sophisticated data layouts, vectorization, workload management, etc.

Differ in their approach: Snowflake wants you to load all data into your RDBMS, Databricks wants you to put all of your data into Delta Lake files and process with Spark.