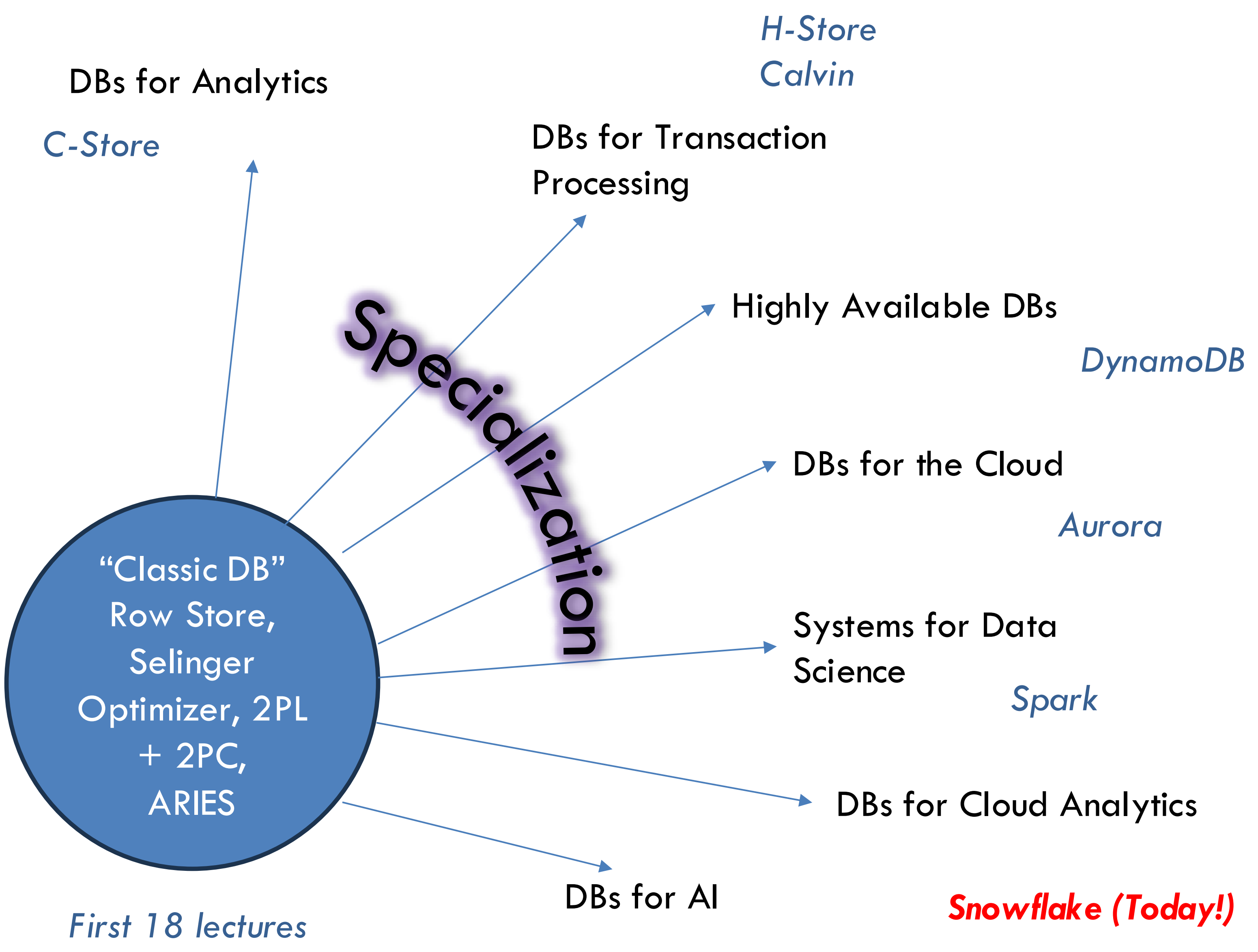
The Hunters in the Snow,
Pieter Bruegel the Elder, 1565

Snowflake

December 2, 2024

Some slides from Sam Madden, Ashish Motivala, Jiaqi Yan, Snowflake Inc

Where Are We???


H-Store
Calvin
DBs for Analytics
C-Store
DBs for Transaction Processing
Specialization
Highly Available DBs
DynamoDB
DBs for the Cloud
Aurora
"Classic DB"
Row Store,
Selinger
Optimizer, 2PL
+ 2PC,
ARIES
Systems for Data Science
Spark
DBs for Cloud Analytics
DBs for AI
Snowflake (Today!)
First 18 lectures

Snowflake Overview

"Elastic Data Warehouse" (db focused on analytics, not transactions) purpose-built for the cloud

Leverages extremely reliable cloud storage (S3) for durability

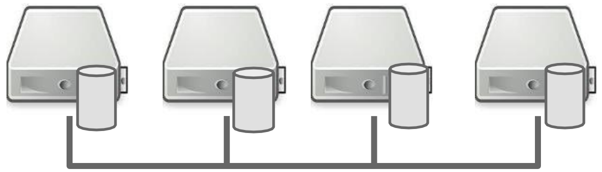
"Shared disk" style design

Modern, efficient query executor

Why Use The Cloud?

- New(-ish) 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)
  - Reduced need for complex IT organization and infrastructure
  - Pay-per-use model
  - Simplified software delivery, update, and user support

Shared-nothing Architecture

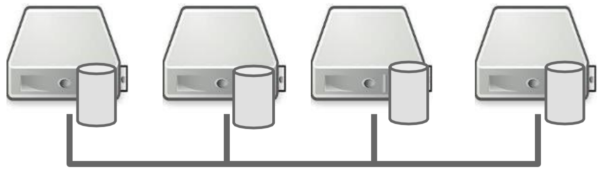


- Tables are horizontally partitioned across nodes
- Every node has its own local storage
- Every node is only responsible for its local table partitions

- Simple 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
- Yields bad elasticity
  - Resizing compute cluster requires redistributing (lots of) data
  - Cannot simply shut off unused compute resources → no pay-per-use
- And limited availability
  - Membership changes (failures, upgrades) significantly impact performance and may cause downtime
- Faces problems with **homogeneous resources** vs. **heterogeneous workload**
  - Bulk loading, reporting, exploratory analysis

A Data Warehouse Design for the Cloud

Snowflake is an RDBMS redesigned entirely for cloud-based operation

Storage decoupled from compute

Native for structured & semi-structured

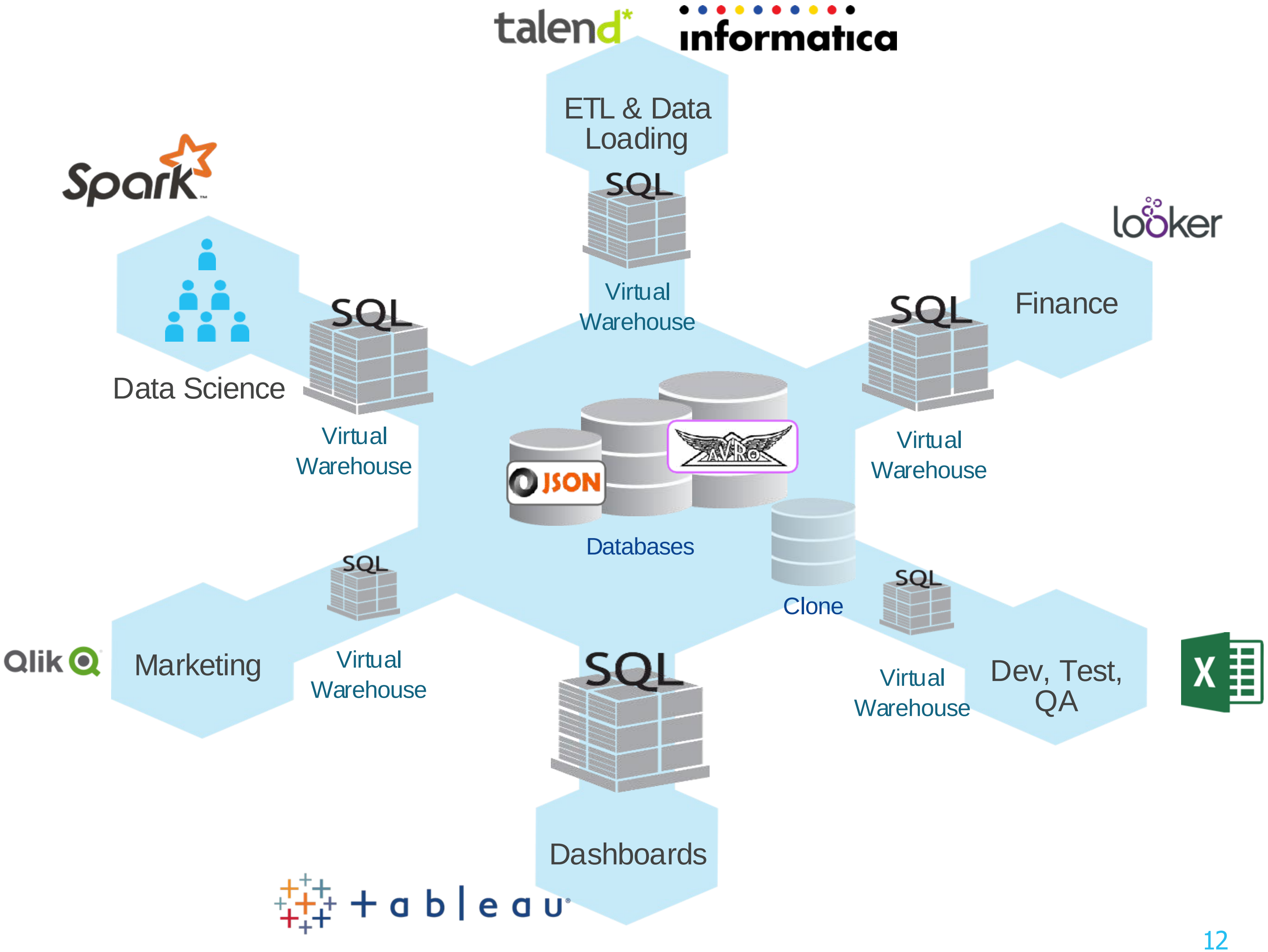
Scalability along many dimensions

Low cost, w/compute on demand

Instant db cloning

Isolate production from Dev & QA

Highly available



Concerns?

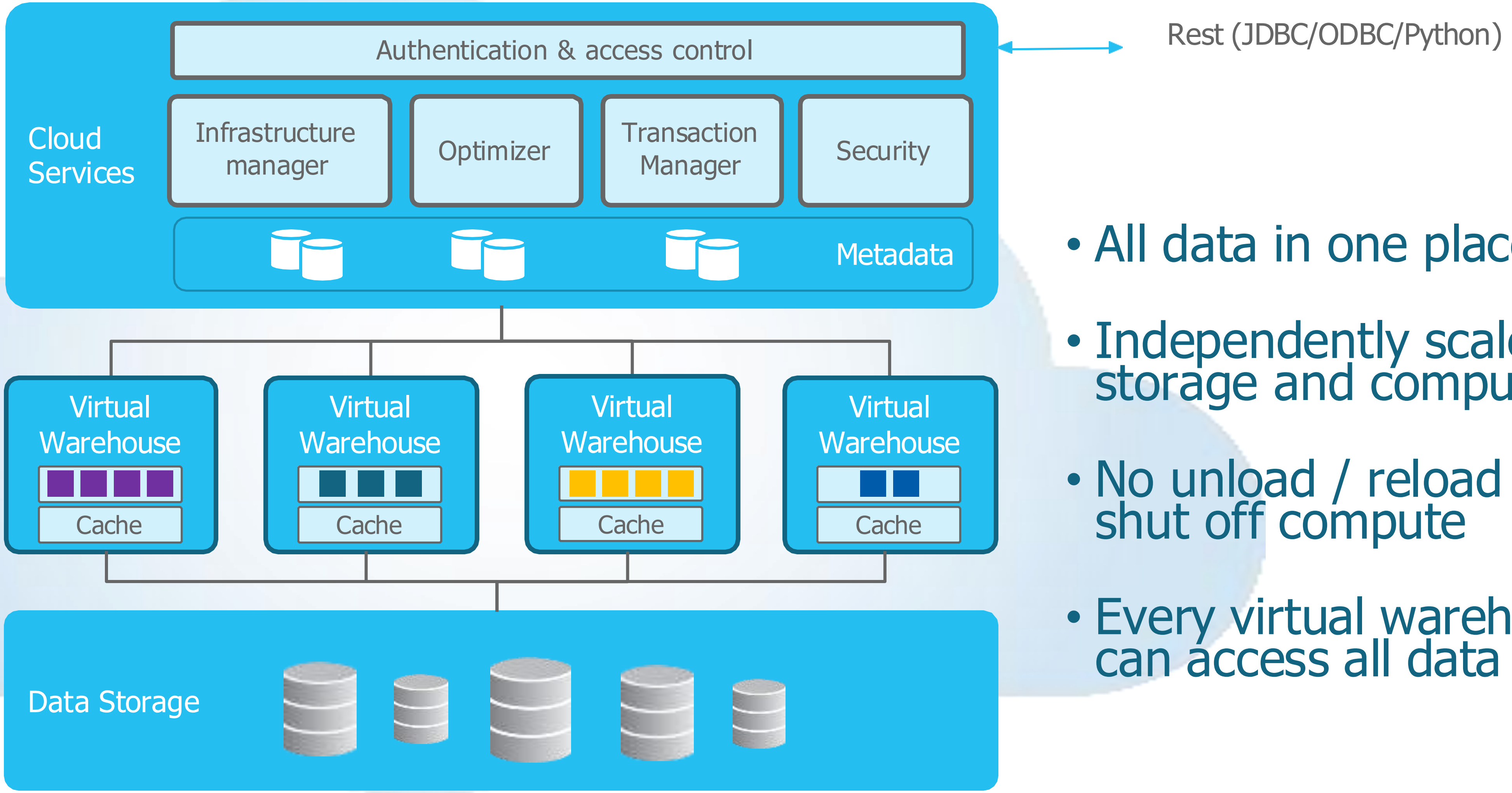
Photo realistic render scary snowflake crown fantasy elements with the words “What could go wrong” glowing letters above

Performance?

Updates?

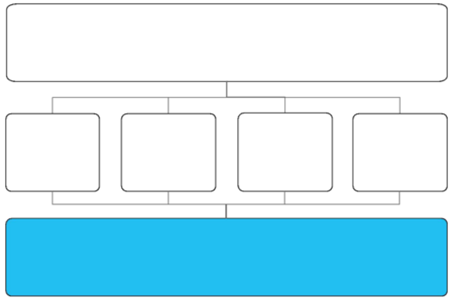
Lack of Control?

Multi-cluster Shared-data Architecture



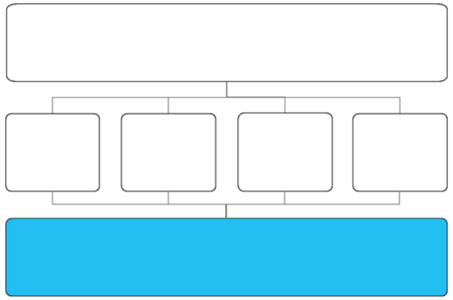
- All data in one place
- Independently scale storage and compute
- No unload / reload to shut off compute
- Every virtual warehouse can access all data

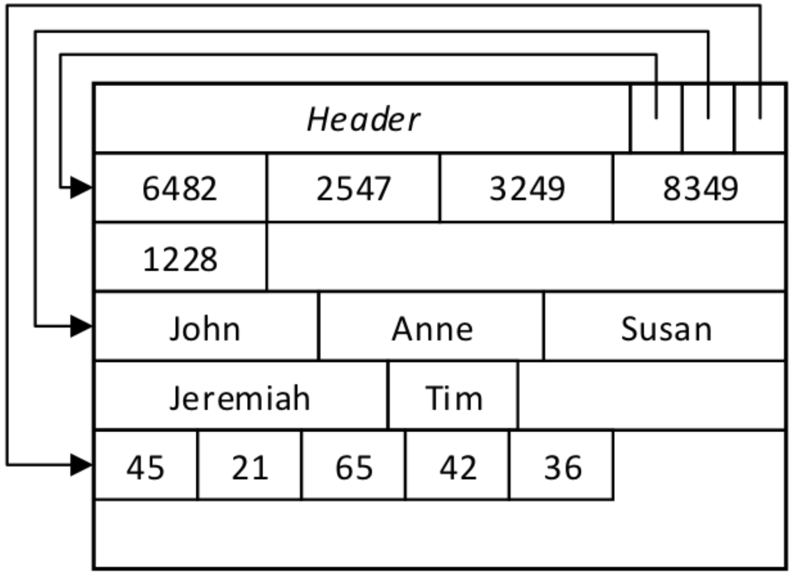
Data Storage Layer



- 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
- 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

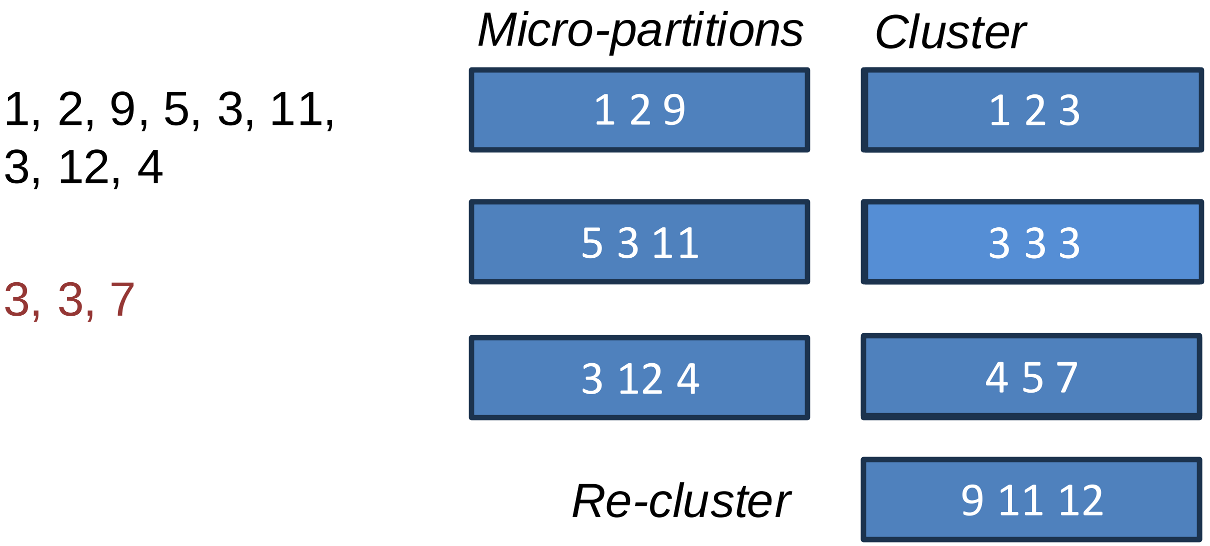
Table Clustering

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



https://docs.snowflake.com/en/user-guide/tables-clustering-keys

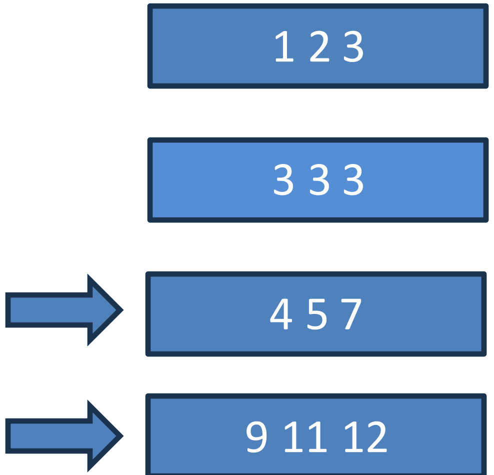
Block Skipping ("Pruning") vs Indexing

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?

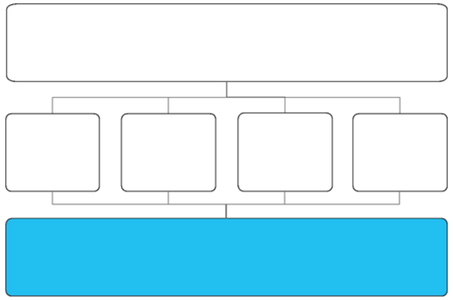
Allows "skipping" – each partition has a min/max value, only read partitions that satisfy query.

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

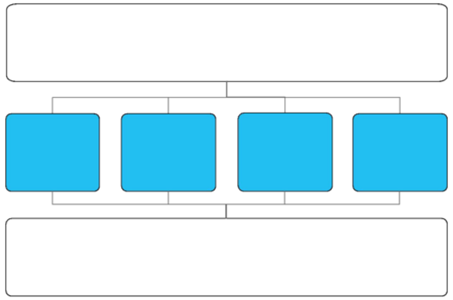


- Tiered storage also used for temp data and query results
  - Arbitrarily large queries, never run out of disk
  - New forms of client interaction
    - No server-side cursors, since clients can directly access S3 to retrieve and reuse previous query results

- 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 (see later)

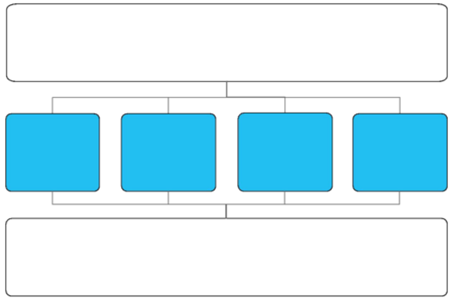

Break Time

Virtual Warehouse



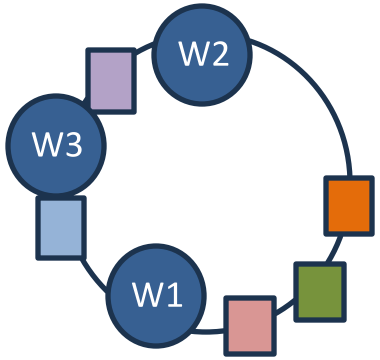
- A virtual warehouse is a cluster of EC2 instances serving as query processor “workers”
- Pure compute resources, decoupled from data storage
  - Created, destroyed, resized on demand
  - Users may run multiple warehouses at same time
  - Each warehouse has access to all data but isolated performance
  - 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
  - They 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 uses consistent hashing, with deterministic stealing

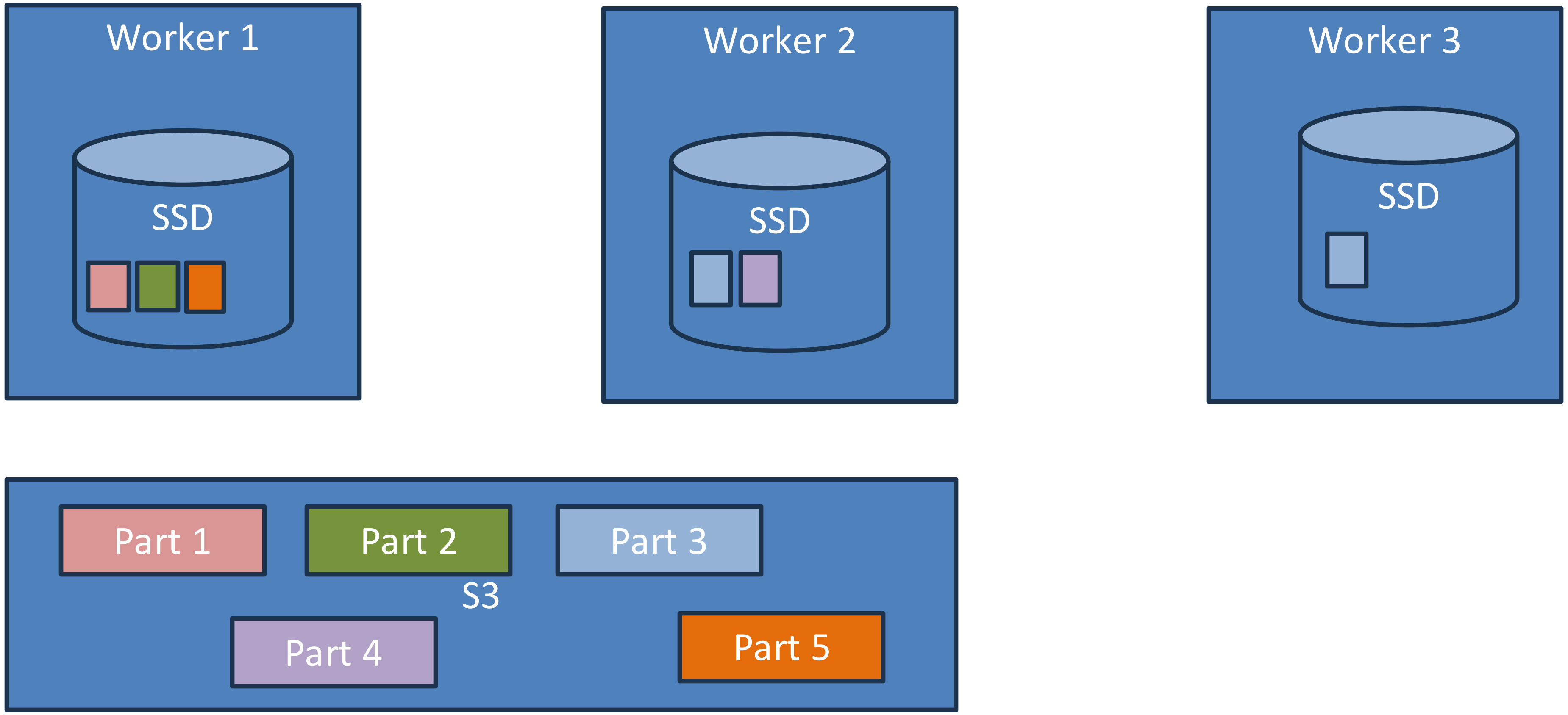
Data Affinity and Caching



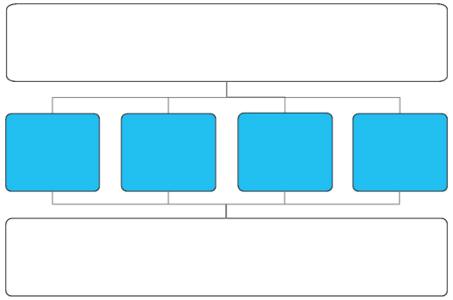
Data cached on local storage; managed via LRU
Affinity between workers and partitions via consistent hashing

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

Vectorized Execution

- Iterator model invokes getNext() many times, imposes huge function call overhead
- Virtual functions make branch prediction in tight loop bad
- Large amounts of code per tuple means bad code locality

What's wrong with tuple at a time execution?

Alternatives:

Column/table-at-a-time?

- No pipelining (passing data to ops without copies)
- May unnecessarily materialize intermediates, e.g.,:

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

sal + bonus doesn't need to be stored in tuple-at-a-time (MapReduce materializes tons of intermediate data)

- 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
- ~1000x 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

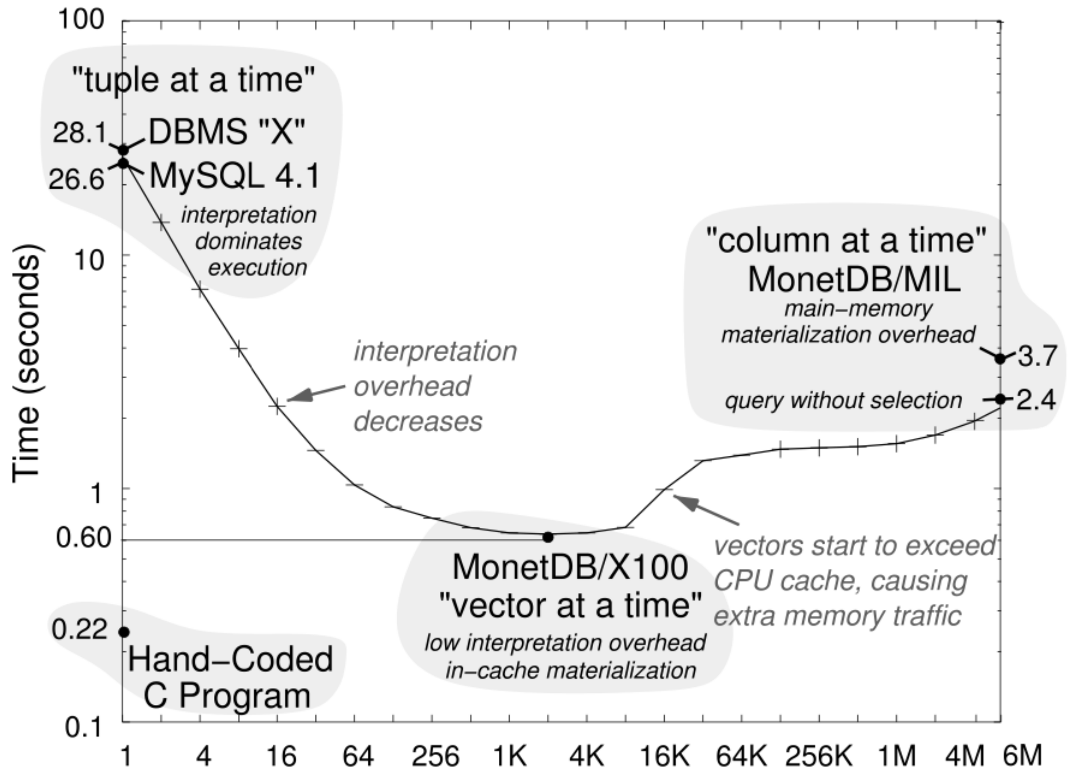


Figure 1: Hand-written code vs. execution engines for TPC-H Query 1 (Figure 3 of [16])

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

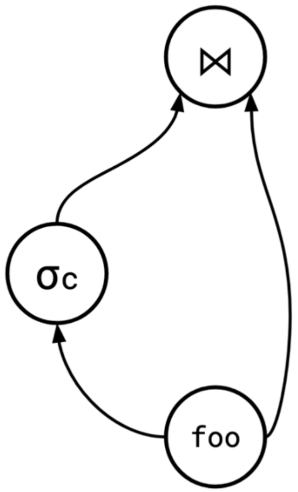
Push-Based

They cite the paper below, which really is about compilation

But say: that a push-based system "enable Snowflake to efficiently process DAG-shaped plans" – paper below does not make it clear how they help with DAG shaped plans

In fact, not clear Snowflake implements codegen / compilation at all!

```
WITH foo as (...)
SELECT * FROM
        (SELECT * FROM foo WHERE c) AS foo1
        JOIN foo AS foo2 ON foo1.a = foo2.b
```



Example from https://justinjaffray.com/query-engines-push-vs.-pull/

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

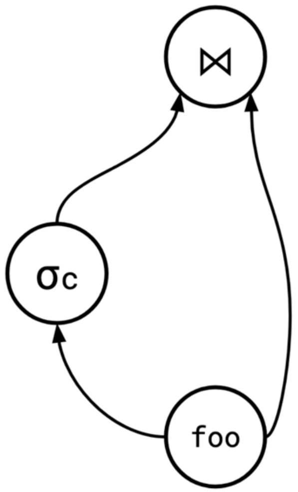
Push-Based

In an iterator model, this is annoying to deal with:

- foo has to buffer results until upstream operators *pull, and*
- keep track of which consumers have consumed which results

In a *push* model, this is greatly simplified – foo just sends results to both operators (who may have to buffer)

```
WITH foo as (...)
SELECT * FROM
        (SELECT * FROM foo WHERE c) AS foo1
        JOIN foo AS foo2 ON foo1.a = foo2.b
```



Example from https://justinjaffray.com/query-engines-push-vs.-pull/

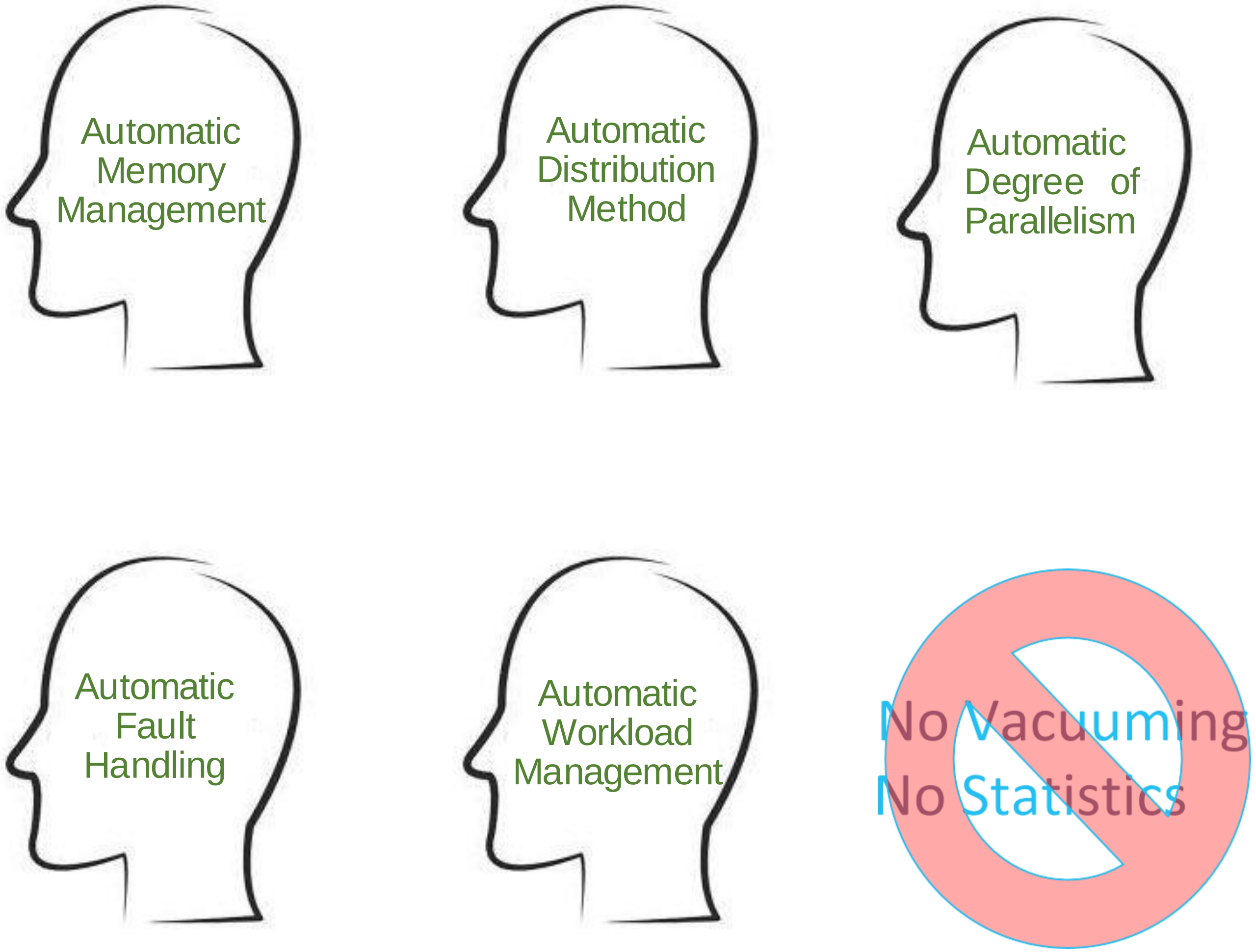
No Buffer Pool?!

What do they mean by this?

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

Self Tuning & Self Healing

- Adaptive
- Self-tuning
- Do no harm!
- Automatic
- Work by default

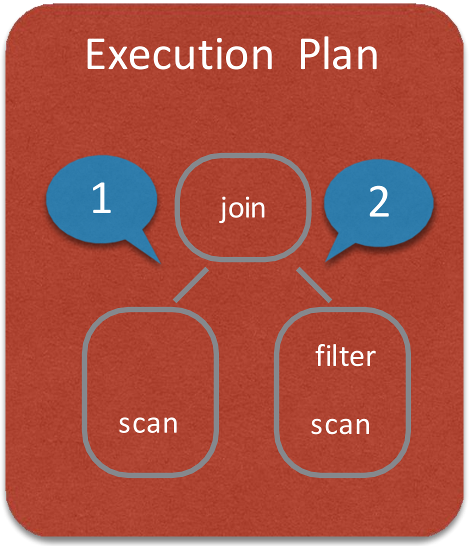


This part of the paper has a heavy marketing quality to it

Example: Automatic Skew Avoidance

1

2 Detect popular values on the build side of the join
Use broadcast for those and directed join for the others



- Adaptive → popular values detected at runtime
- Self-tuning → number of values
- Transparent → no performance degradation
- Automatic → kicks in when needed
- Default → enabled by default for all joins

Q: What is the issue with popular values? Why do those mess up the build side of joins?

A: Naively, values in a join get sent to a particular node for processing; popular values mean some nodes will be overwhelmed. So broadcast the popular ones everywhere so that certain keys are not tied to certain nodes

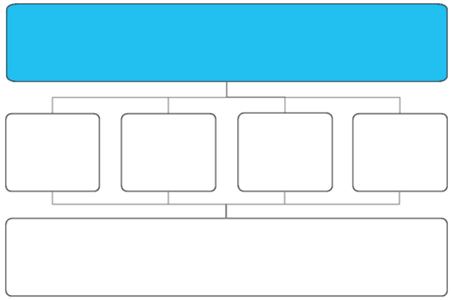
Example: Workload Stealing

Consistent hashing determines which files workers will retrieve for processing a query before execution.

When a worker process completes scanning its input files, it might mean one worker has finished ahead of others; it can ask peer worker processes that it scan their files for them

The requestor always downloads from storage instead of the peer to avoid additional burden.

Concurrency Control



- Designed for analytic workloads
  - Large reads, bulk or trickle inserts, bulk updates
- Snapshot Isolation (SI) [Berenson95]
- SI based on multi-version concurrency control (MVCC)
  - DML statements (insert, update, delete, merge) produce new table versions of tables by adding or removing whole files
  - Natural choice because table files on S3 are immutable
  - Additions and removals tracked in metadata (key-value store)
- Versioned snapshots used also for time travel and cloning

Snapshot Isolation

Recall:

- "Snapshot" state of database at start of query
- Abort any xaction that does a WX for some X updated by another concurrent xaction
- This prevents a mixture of writes from two concurrent transactions being written
- Conflicting reads are permitted
- It does not enforce a total ordering of txs (like Serializability)

In a system with immutable storage, this is easy: just need to track the set of files that were on S3 when the query started and read from those. Later writes create new files.

Is Snapshot Isolation Serializable?

No! Write skew:

T1:

RX = 1

WY = 2

T2:

RY = 1

WX = 3

Neither observed each other's write!

Many database systems use snapshot isolation anyway, since this scenario is somewhat unusual.

Unlikely to be a problem in Snowflake as updates are not common.

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, protobufs, JSON, 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 `VARIANT`s 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

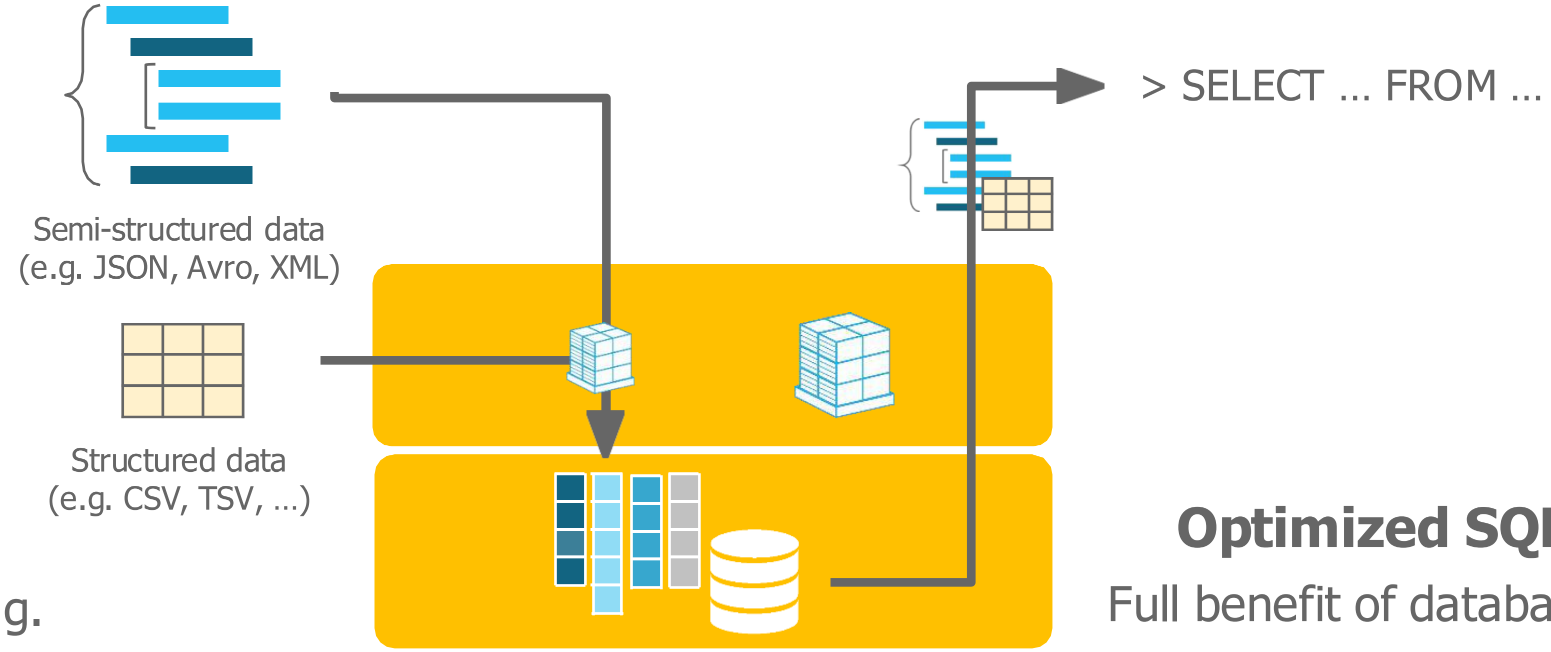
```
SELECT p.contact.name.first AS "first_name",
       p.contact.name.last AS "last_name",
       (f.value.type || ': ' || f.value.contact) AS "contact"
FROM person p,
     LATERAL FLATTEN(input => p.contact) f;

------------+-----------+---------------------+
 first_name | last_name |       contact       |
------------+-----------+---------------------+
 "John"     | "Doe"     | email: john@doe.xyz |
 "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



Native support
Loaded in raw form (e.g. JSON, Avro, XML)

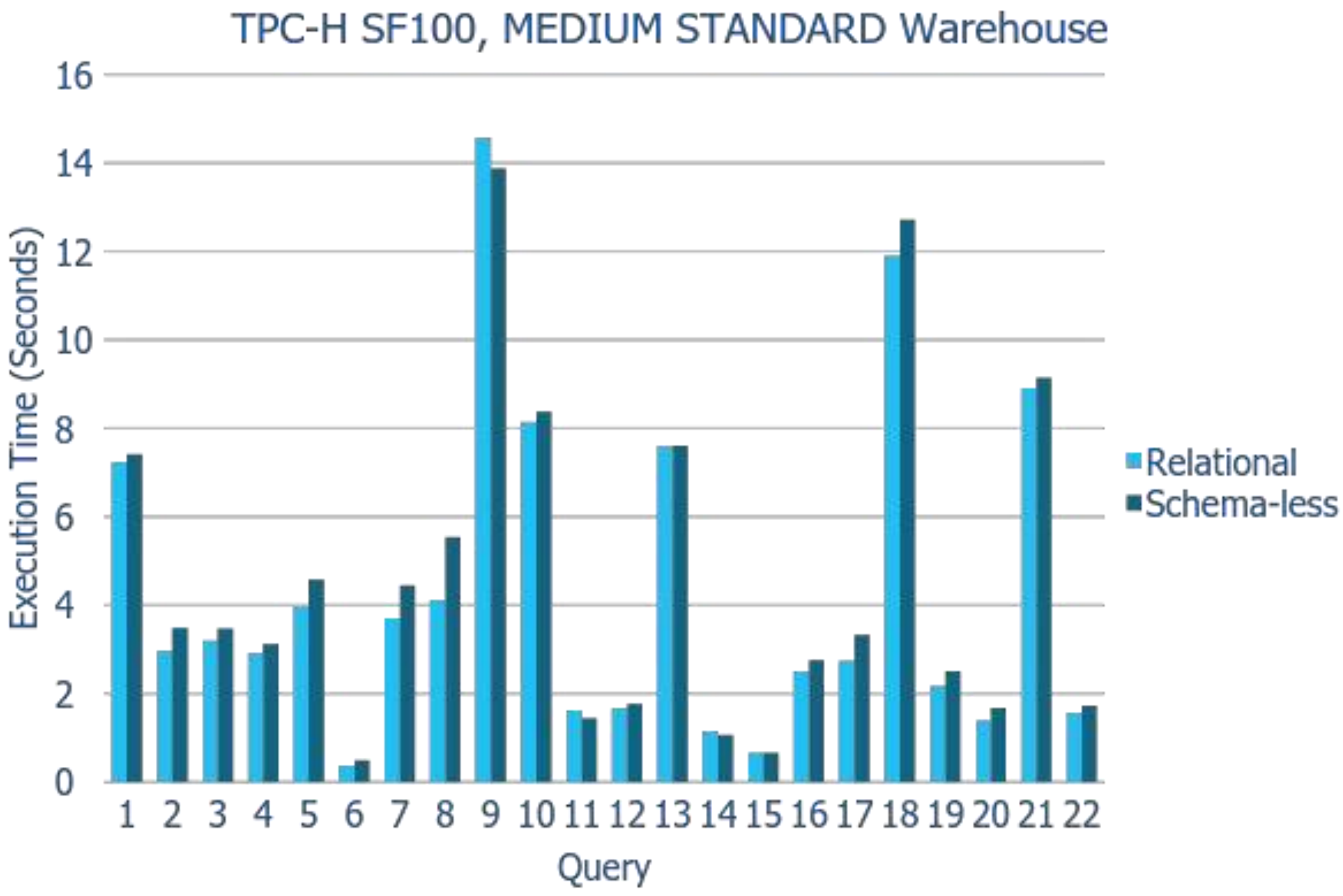
Optimized storage
Optimized data type, no fixed schema or transformation required

Optimized SQL querying
Full benefit of database optimizations (pruning, filtering, …)

["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

Make	Model
NISSAN	LEAF
NISSAN	LEAF
NISSAN	LEAF
FIAT	500
TESLA	MODEL 3
TOYOTA	PRIUS PRIME
TESLA	MODEL Y

Schema-Less Performance

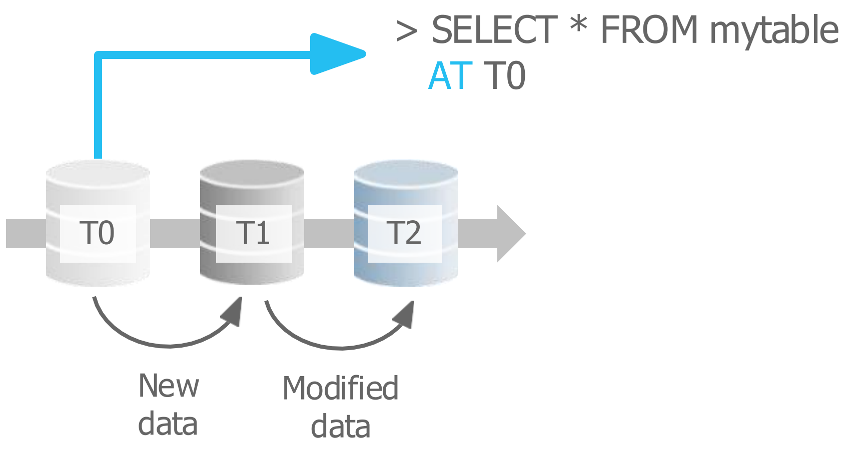


ETL vs. ELT

- ETL = Extract-Transform-Load
  - Classic approach: extract from source systems, run through some transformations (perhaps using MapReduce), then load into RDBMS
- 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 data warehouse, transform there if desired
  - Decouples information producers from information consumers
- Snowflake aims for 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

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