Lecture 20: Snowflake
Sam Madden

Some slides from Ashish Motivala, Jiaqi Yan, Snowflake
Last Time: Spark and Resilient Distributed Datasets (RDDs)
Last Time: Spark and Resilient Distributed Datasets (RDDs)

- Utilize distributed memory while giving efficient fault tolerance
  Avoid storing data updates explicitly
  Instead, obtain fault tolerance by logging transformations (*lineage*)
Last Time: Spark and Resilient Distributed Datasets (RDDs)

• Utilize distributed memory while giving efficient fault tolerance
  Avoid storing data updates explicitly
  Instead, obtain fault tolerance by logging transformations (lineage)

• Operations via coarse-grained transformations (e.g., map, filter)
Last Time: Spark and Resilient Distributed Datasets (RDDs)

• Utilize distributed memory while giving efficient fault tolerance
  Avoid storing data updates explicitly
  Instead, obtain fault tolerance by logging transformations (lineage)

• Operations via coarse-grained transformations (e.g., map, filter)
• Allow user control of data persistence, partitioning, and caching
RDDs

• **Read-only**, partitioned collection of records
RDDS

- **Read-only**, partitioned collection of records
- Created from either data in stable storage or other RDDs
RDDs

- **Read-only**, partitioned collection of records
- Created from either data in stable storage or other RDDs
- A sequence of **transformations** defines an RDD:
  - Map, filter, flatmap, sample, groupbykey, reducebykey, join, union
RDDs

- **Read-only**, partitioned collection of records
- Created from either data in stable storage or other RDDs
- A sequence of **transformations** defines an RDD:
  - Map, filter, flatmap, sample, groupbykey, reducebykey, join, union
- **Actions** return value or export data to storage system
  - count, collect, save, reduce, lookup
RDDS

- **Read-only**, partitioned collection of records
- Created from either data in stable storage or other RDDS
- A sequence of **transformations** defines an RDD:
  - Map, filter, flatmap, sample, groupbykey, reducebykey, join, union

- **Actions** return value or export data to storage system
  - count, collect, save, reduce, lookup

No need to actually run code until there’s an **action**
RDDs

- **Read-only**, partitioned collection of records
- Created from either data in stable storage or other RDDs
- A sequence of **transformations** defines an RDD:
  - Map, filter, flatmap, sample, groupbykey, reducebykey, join, union
- **Actions** return value or export data to storage system
  - count, collect, save, reduce, lookup

No need to actually run code until there’s an **action**

**Read-only** means we can exploit speculative (re)execution

MapReduce also does this
Example: Console Log Mining

lines = spark.textFile("hdfs://...")

errors = lines.filter(_.startsWith("ERROR"))

errors.count()
Example: Console Log Mining

```scala
lines = spark.textFile("hdfs://...")

errors = lines.filter(_.startsWith("ERROR"))

errors.count()
```

"Base RDD" and "Transformed RDD" may never be actually stored on disk.
Where Are We???

“Classic DB”
Row Store,
Selinger
Optimizer,
2PL + 2PC,
ARIES

First 12 lectures
Where Are We???

"Classic DB"
Row Store,
Selinger
Optimizer,
2PL + 2PC,
ARIES

DBs for Analytics

DBs for Transaction Processing

Highly Available DBs

DBs for the Cloud

Systems for Data Science

DBs for Cloud Analytics

First 12 lectures
Where Are We???

- DBs for Analytics
  - C-Store
- DBs for Transaction Processing
- Highly Available DBs
- DBs for the Cloud
- Systems for Data Science
- DBs for Cloud Analytics

First 12 lectures

"Classic DB"
Row Store,
Selinger
Optimizer,
2PL + 2PC,
ARIES
Where Are We???

"Classic DB"
Row Store,
Selinger
Optimizer,
2PL + 2PC,
ARIES

DBs for Analytics
C-Store

DBs for Transaction Processing

H-Store
Calvin

Highly Available DBs

DBs for the Cloud

Systems for Data Science

DBs for Cloud Analytics

First 12 lectures
Where Are We???

"Classic DB"
Row Store,
Selinger Optimizer,
2PL + 2PC,
ARIES

First 12 lectures

C-Store

DBs for Analytics

H-Store
Calvin

DBs for Transaction Processing

Highly Available DBs
DynamoDB

DBs for the Cloud

Systems for Data Science

DBs for Cloud Analytics
Where Are We???

"Classic DB"
Row Store,
Selinger
Optimizer,
2PL + 2PC,
ARIES

First 12 lectures

DBs for Analytics
C-Store

H-Store
Calvin

DBs for Transaction Processing

Highly Available DBs
DynamoDB

DBs for the Cloud
Aurora

Systems for Data Science

DBs for Cloud Analytics
Where Are We???

"Classic DB"
Row Store,
Selinger
Optimizer,
2PL + 2PC,
ARIES

DBs for Analytics
C-Store

DBs for Transaction
Processing

Highly Available DBs
H-Store
Calvin

DBs for the Cloud
DynamoDB
Aurora

Systems for Data
Science
Spark

DBs for Cloud
Analytics

First 12 lectures
Where Are We???

"Classic DB"
Row Store,
Selinger Optimizer,
2PL + 2PC,
ARIES

First 12 lectures

DBs for Analytics
C-Store

DBs for Transaction Processing

Highly Available DBs

DBs for the Cloud

Systems for Data Science

DBs for Cloud Analytics

Cloud DBs

DynamoDB

Aurora

Spark

Snowflake (Today)
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
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 with scary snowflake crown and fantasy elements with the words “What could go wrong” glowing letters above.
Concerns?

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

Performance?
Concerns?

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

Performance?

Updates?
Concerns?

Photo realistic render of a scary snowflake crowned clown with fantasy elements and the words “What could go wrong” glowing letters above.

Performance?
Updates?
Lack of Control?
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

Cloud Services

- Authentication & access control
- Infrastructure manager
- Optimizer
- Transaction Manager
- Security

Data Storage

- Virtual Warehouse
  - Cache
- Virtual Warehouse
  - Cache
- Virtual Warehouse
  - Cache
- Virtual Warehouse
  - Cache

Rest (JDBC/ODBC/Python)
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
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

Not great for point updates / deletes!
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
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
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
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
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
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
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
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
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?
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

1 2 3
3 3 3
4 5 7
9 11 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

<table>
<thead>
<tr>
<th>T.a metadata</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>12</td>
</tr>
</tbody>
</table>
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
```
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
```
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
```
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

<table>
<thead>
<tr>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>12</td>
</tr>
</tbody>
</table>

SELECT a2 from T
WHERE a > 5
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
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
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 (see later)
Virtual Warehouse

- Virtual warehouse = Cluster of EC2 instances – “workers”
- Pure compute resources
  - Created, destroyed, resized on demand
  - Users may run multiple warehouses at the 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

Worker 1
SSD

Worker 2
SSD

S3
Data Affinity and Caching

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

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

What’s wrong with tuple at a time execution?

Alternatives:
Vectorized Execution

What’s wrong with tuple at a time execution?

Alternatives:
Column/table-at-a-time?
Vectorized Execution

What’s wrong with tuple at a time execution?

Alternatives:
Column/table-at-a-time?
• No pipelining
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
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

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

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?

Illustrative Example

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

Pushed-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!

Pushed-Based

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

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/
Pushed-Based

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

- foo has to buffer results until upstream operators pull, and

```sql
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/
Pushed-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

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/
**Pushed-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)

```sql
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 ?!
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
• Default

Automatic Memory Management
Automatic Distribution Method
Automatic Degree of Parallelism
Automatic Fault Handling
Automatic Workload Management

No Vacuuming
No Statistics
- Adaptive
- Self-tuning
- Do no harm!
- Automatic
- Default
Example: Automatic Skew Avoidance

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

2. 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
Example: Automatic Skew Avoidance

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

2. • 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?
Example: Workload Steading

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

When a worker process completes scanning its input files, it can request from peer worker processes that it scans 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]
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)
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

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?
Is Snapshot Isolation Serializable?

No!
Is Snapshot Isolation Serializable?

No! Write skew:

T1:                  T2:  RX = 1
Is Snapshot Isolation Serializable?

No! Write skew:

T1:
  RX = 1

T2:
  RY = 1
Is Snapshot Isolation Serializable?

**No!** Write skew:

T1:  
RX = 1
RY = 1

T2:  

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.
Is Snapshot Isolation Serializable?

No! Write skew:

T1: 
   RX = 1
   WY = 2

T2: 
   RX = 1
   RY = 1

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.
Is Snapshot Isolation Serializable?

No! Write skew:

T1:
RX = 1
WY = 2

T2:
RY = 1
WX = 3

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.
Is Snapshot Isolation Serializable?

No! Write skew:

T1:
RX = 1
WY = 2

T2:
RY = 1
WX = 3

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.
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 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;
```

<table>
<thead>
<tr>
<th>first_name</th>
<th>last_name</th>
<th>contact</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;John&quot;</td>
<td>&quot;Doe&quot;</td>
<td>email: <a href="mailto:john@doe.xyz">john@doe.xyz</a></td>
</tr>
<tr>
<td>&quot;John&quot;</td>
<td>&quot;Doe&quot;</td>
<td>phone: 555-123-4567</td>
</tr>
<tr>
<td>&quot;John&quot;</td>
<td>&quot;Doe&quot;</td>
<td>phone: 555-666-7777</td>
</tr>
</tbody>
</table>

```
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

Structured data (e.g. CSV, TSV, ...)

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

Semi-structured data (e.g. JSON, Avro, XML)
<table>
<thead>
<tr>
<th>Make</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>NISSAN</td>
<td>LEAF</td>
</tr>
<tr>
<td>NISSAN</td>
<td>LEAF</td>
</tr>
<tr>
<td>FIAT</td>
<td>500</td>
</tr>
<tr>
<td>TESLA</td>
<td>MODEL 3</td>
</tr>
<tr>
<td>TOYOTA</td>
<td>PRIUS PRIME</td>
</tr>
<tr>
<td>TESLA</td>
<td>MODEL Y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stuff</th>
<th>Stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
<tr>
<td>Stuff</td>
<td>Stuff</td>
</tr>
</tbody>
</table>
Schema-Less Performance

TPC-H SF100, MEDIUM STANDARD Warehouse

Execution Time (Seconds)

Query

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

Relational
Schema-less
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

> SELECT * FROM mytable AT T0

T0  T1  T2
New data Modified data
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