CS4221
Cloud Databases II. Data Lakes and Warehouses

Yao LU

2024 Semester 2

National University of Singapore
School of Computing

Data lakes and warehouses: outline

- Data lakes and warehouses
- Case studies
  - Snowflake
  - Other offerings

Recall cloud systems

- Vendors provide database-as-a-service (DBaaS) offerings that are managed DBMS environments.

- Newer systems are starting to blur the lines between shared-nothing and shared-disk.
  - Example: You can do simple filtering on Amazon S3 before copying data to compute nodes.

Recall cloud systems

Approach #1: **Managed DBMSs**

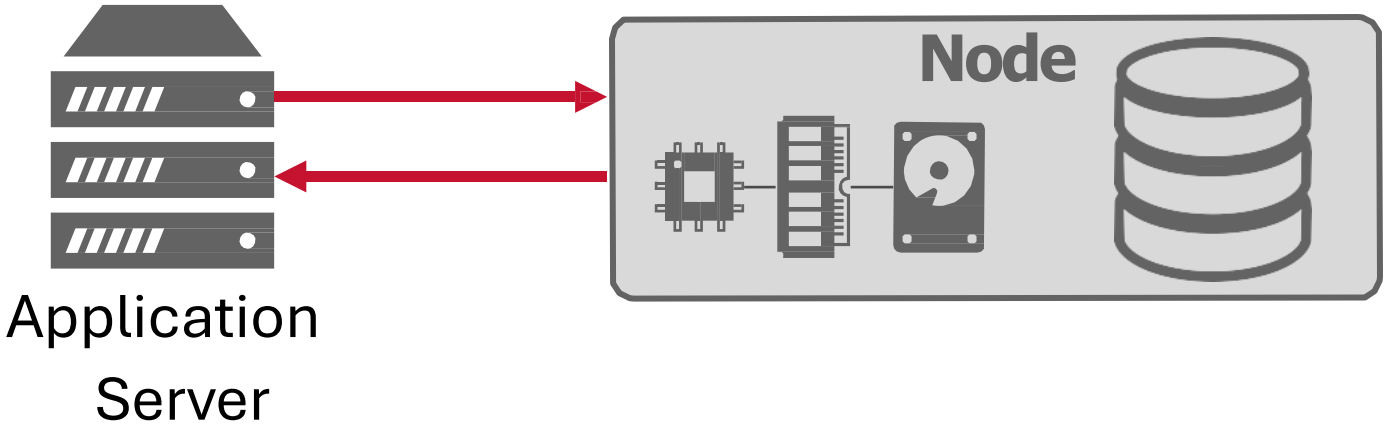
- No significant modification to the DBMS to be "aware" that it is running in a cloud environment.
- Examples: Most vendors

Approach #2: **Cloud-Native DBMS**

- System designed explicitly to run in a cloud environment.
- Usually based on a shared-disk architecture.
- Examples: Snowflake, Google BigQuery

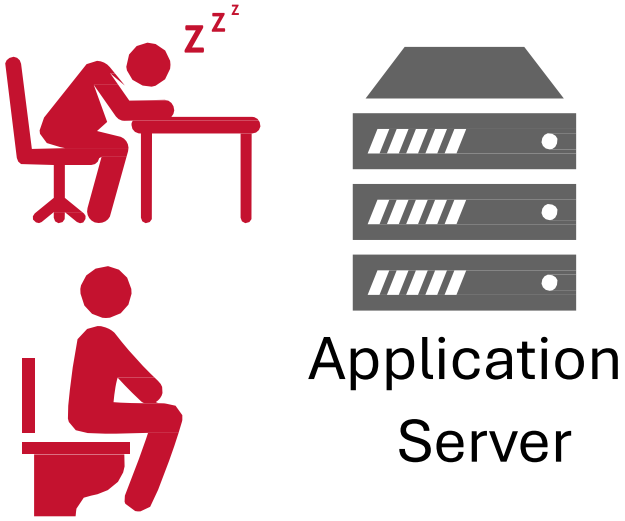
Serverless databases

- Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.



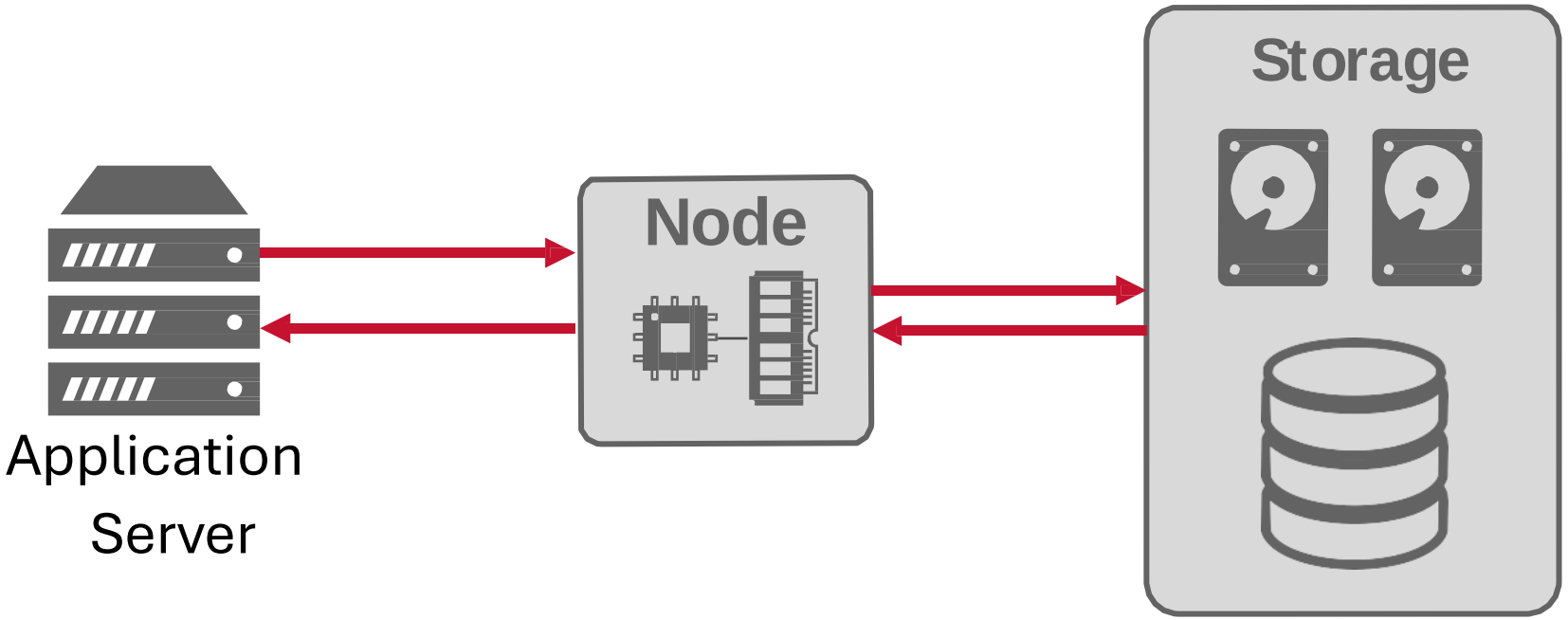
Serverless databases

- Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.



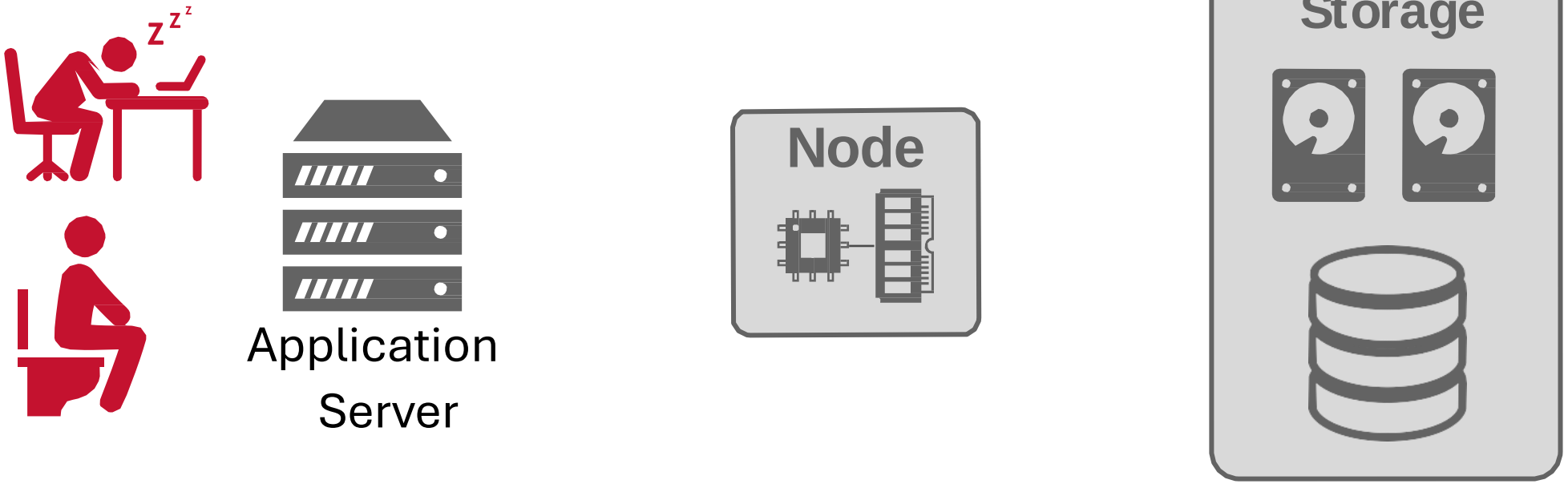
Serverless databases

- Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.



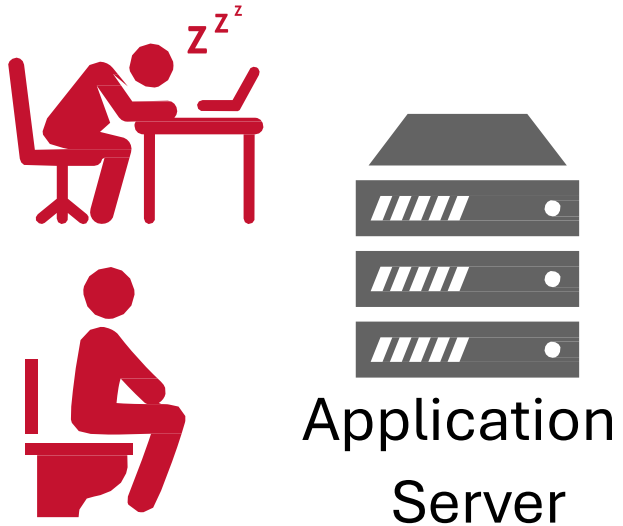
Serverless databases

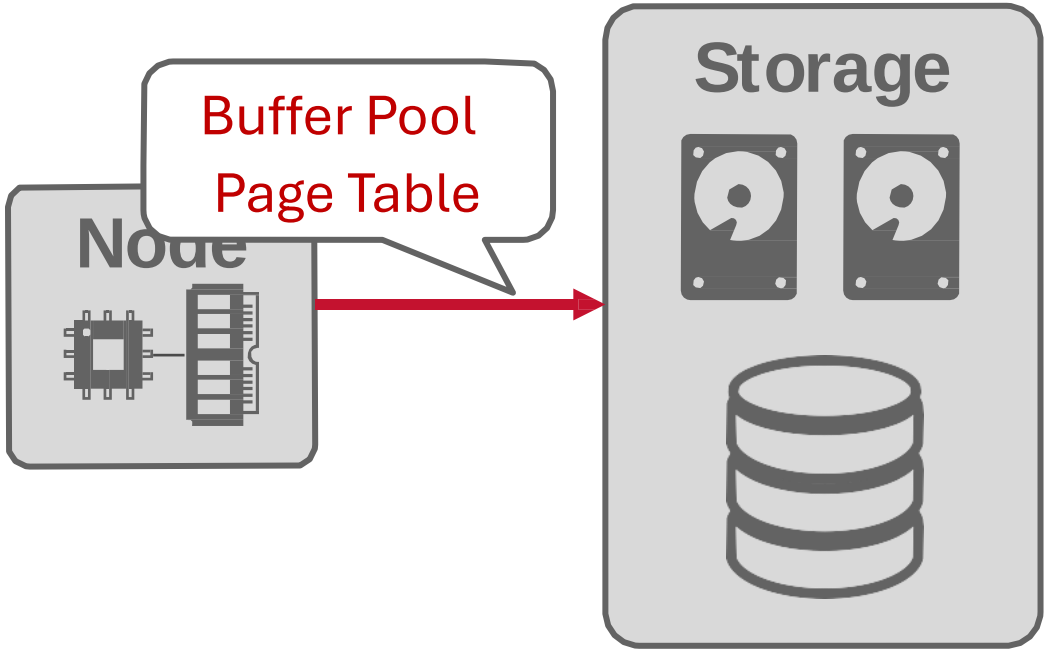
- Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.



Serverless databases

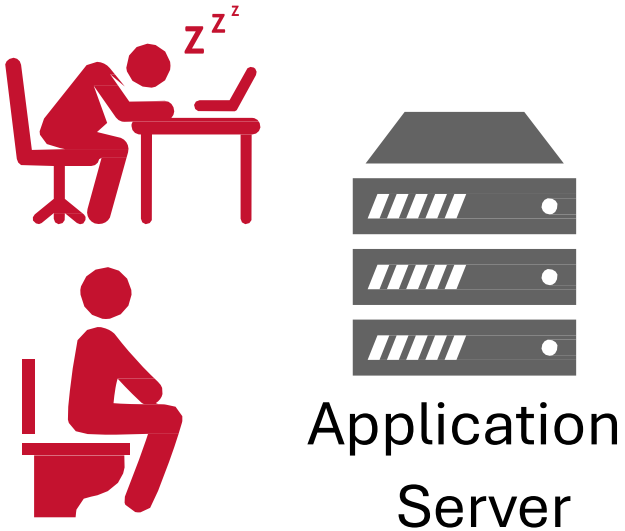
- Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.

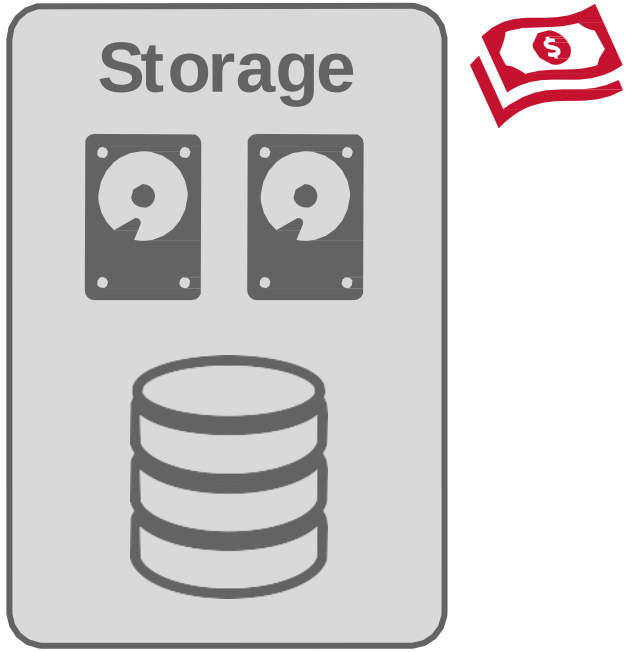




Serverless databases

- Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.

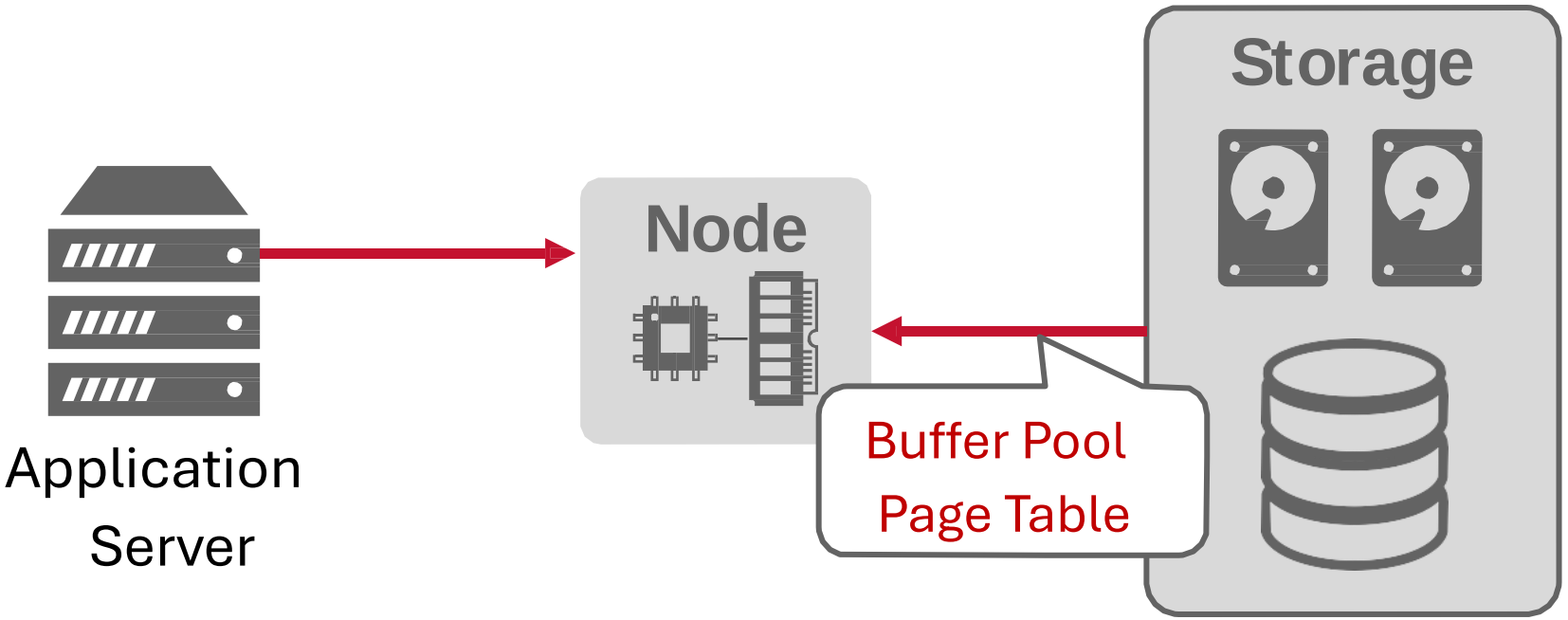




Serverless databases

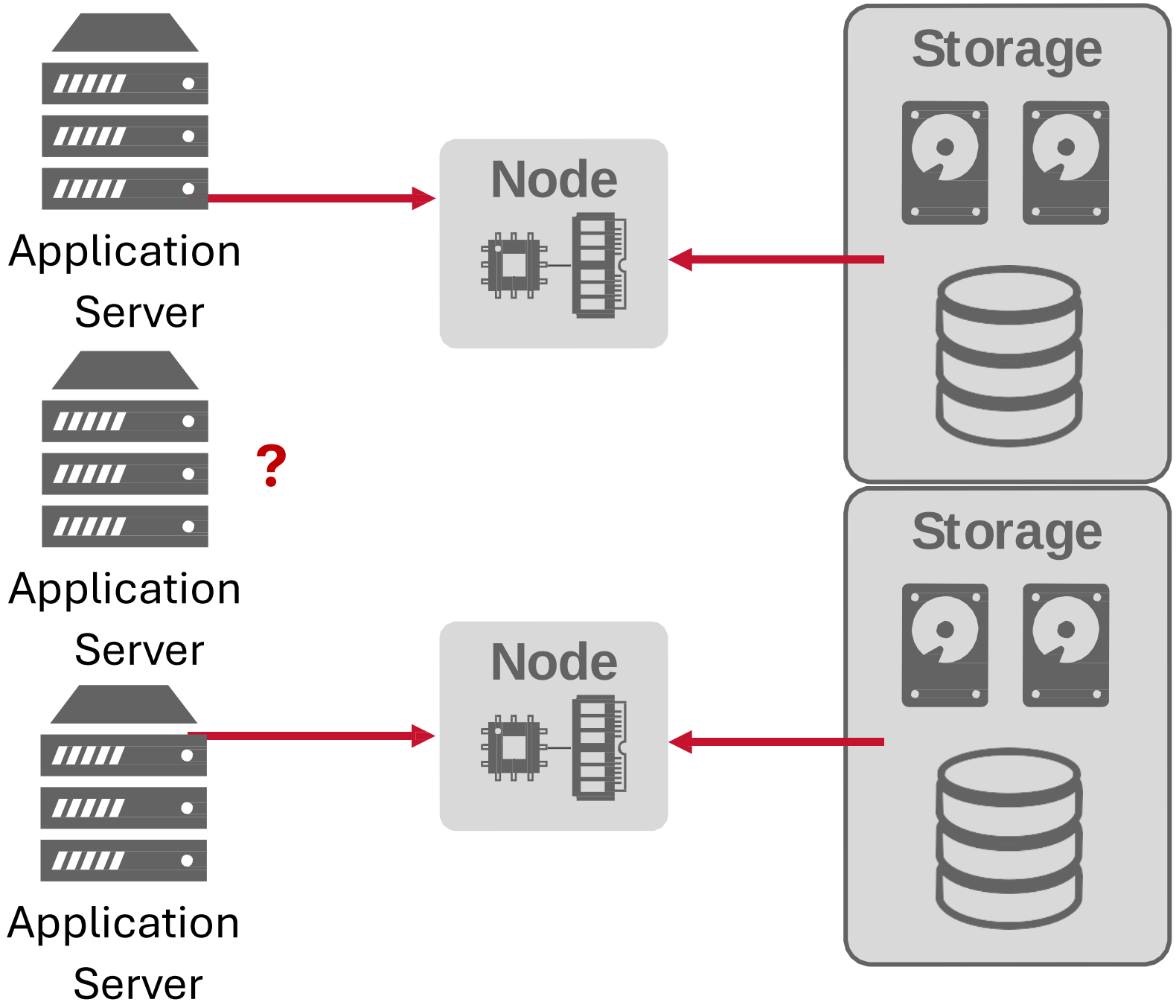
- Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.

planetscale
CockroachDB
NEON
amazon
fauna
Microsoft SQL Azure



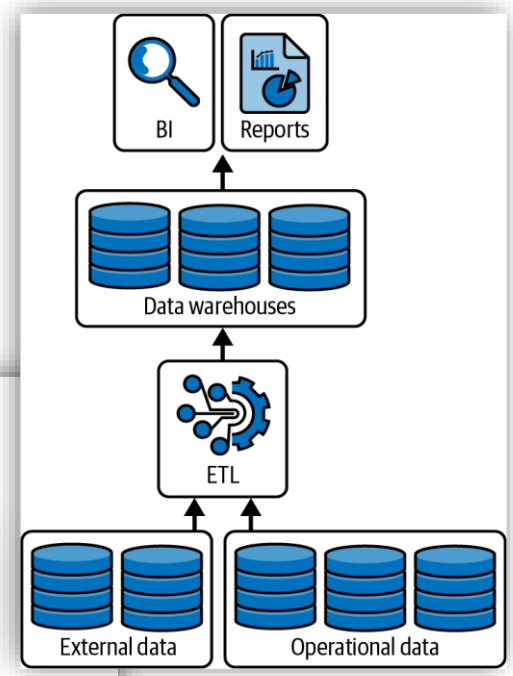
Overbooking?

- Sell more than have.



Data warehouses

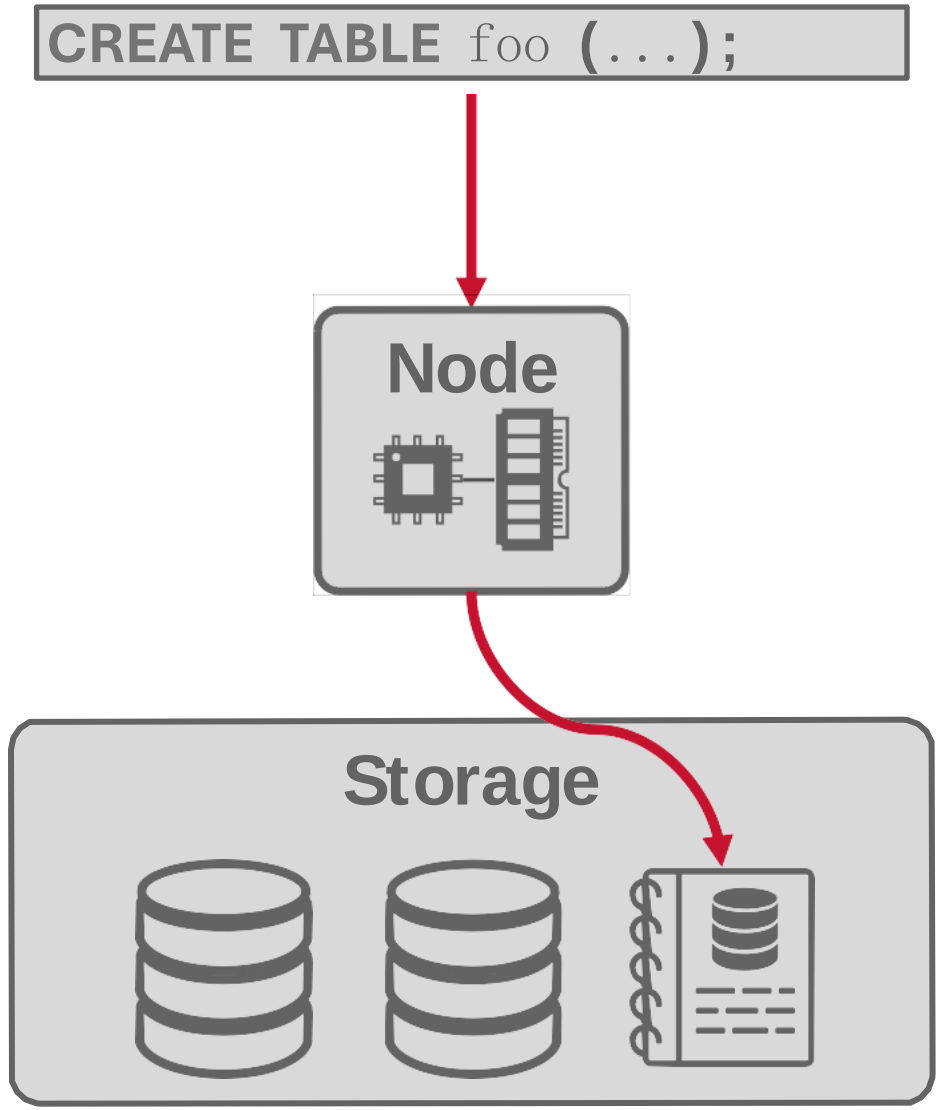
- A data management system that stores current and historical data from multiple sources in a business friendly manner for easier insights and reporting. Typically used for business intelligence (BI).
  - ACID transactions
  - Management features (backup and recovery controls, gated controls, etc.)
  - Performance optimizations (indexes, partitioning, etc.)
  - Limited support for ML and unstructured data.



https://www.future-processing.com/blog/top-8-data-warehouse-solutions/

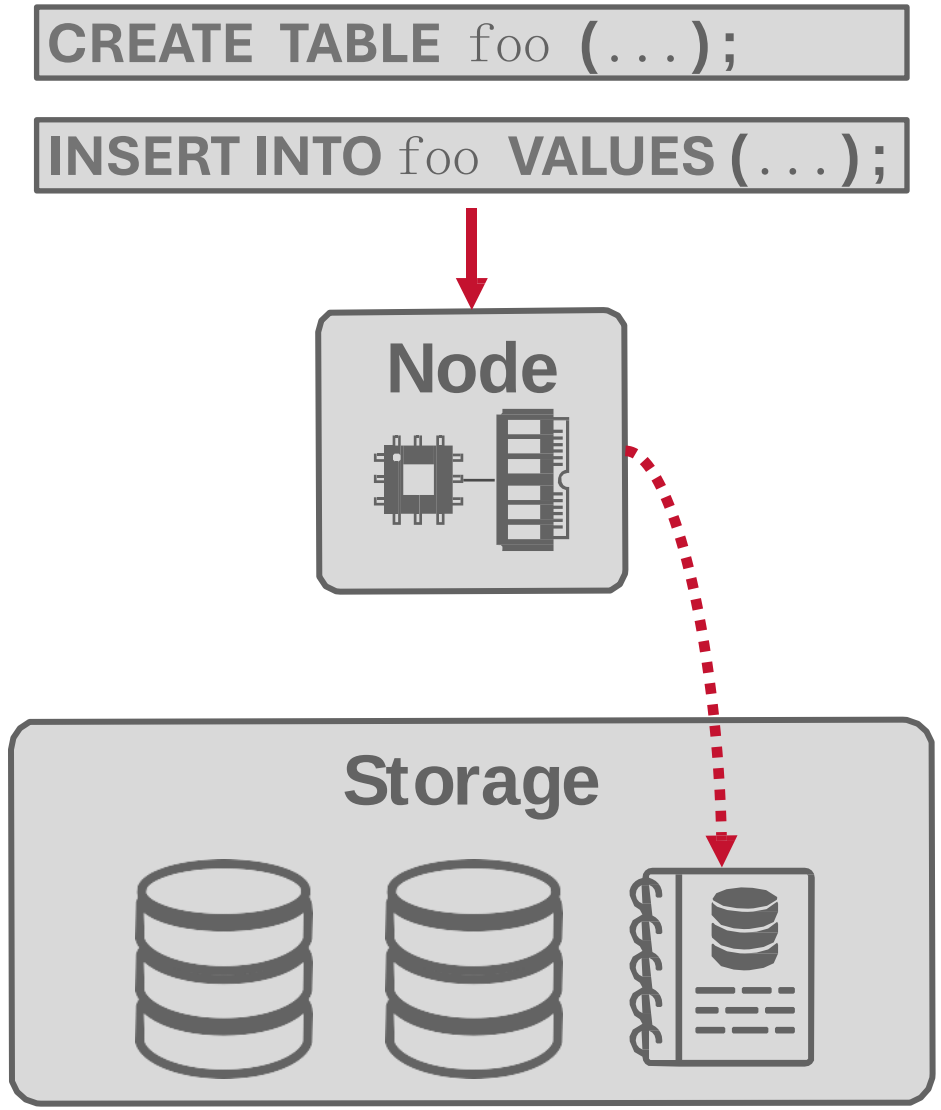
Data lakes

- Repository for storing large amounts of structured, semi-structured, and unstructured data without having to define a schema or ingest the data into proprietary internal formats.



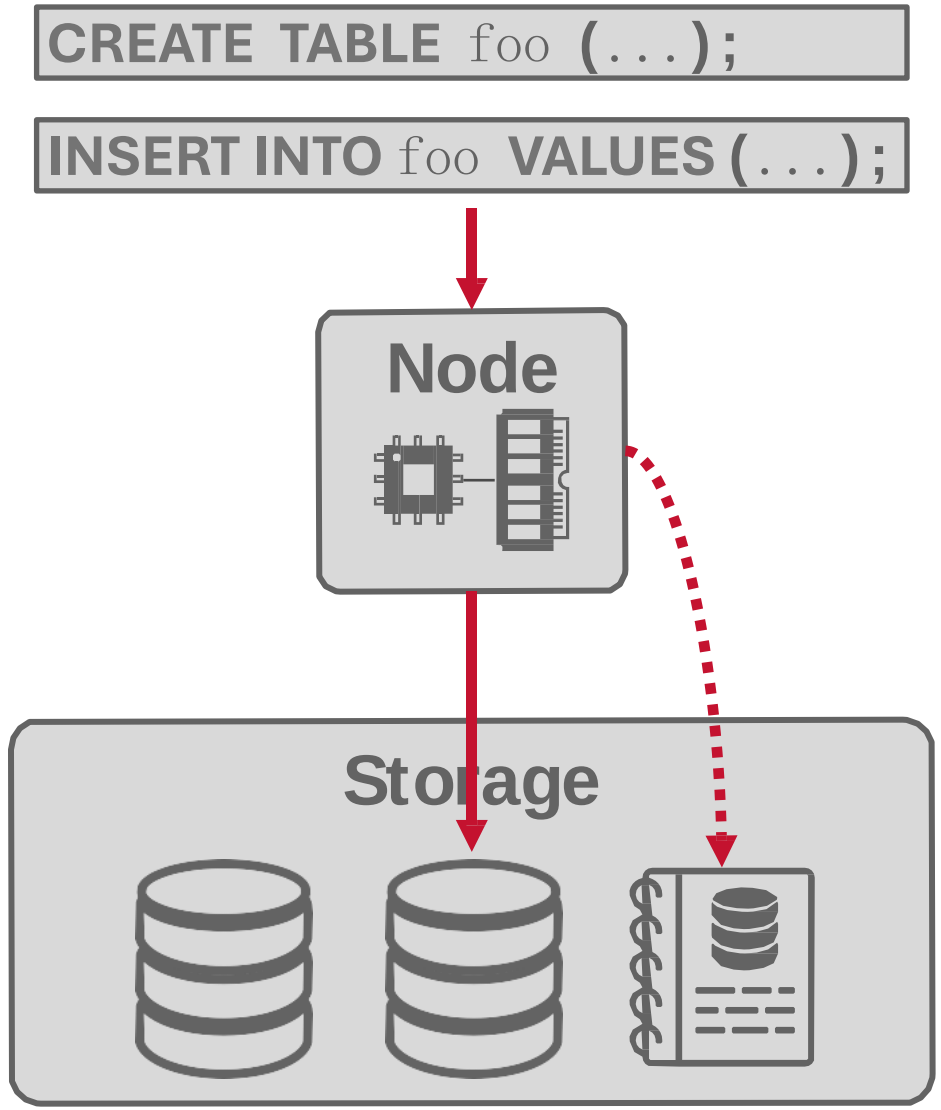
Data lakes

- Repository for storing large amounts of structured, semi-structured, and unstructured data without having to define a schema or ingest the data into proprietary internal formats.



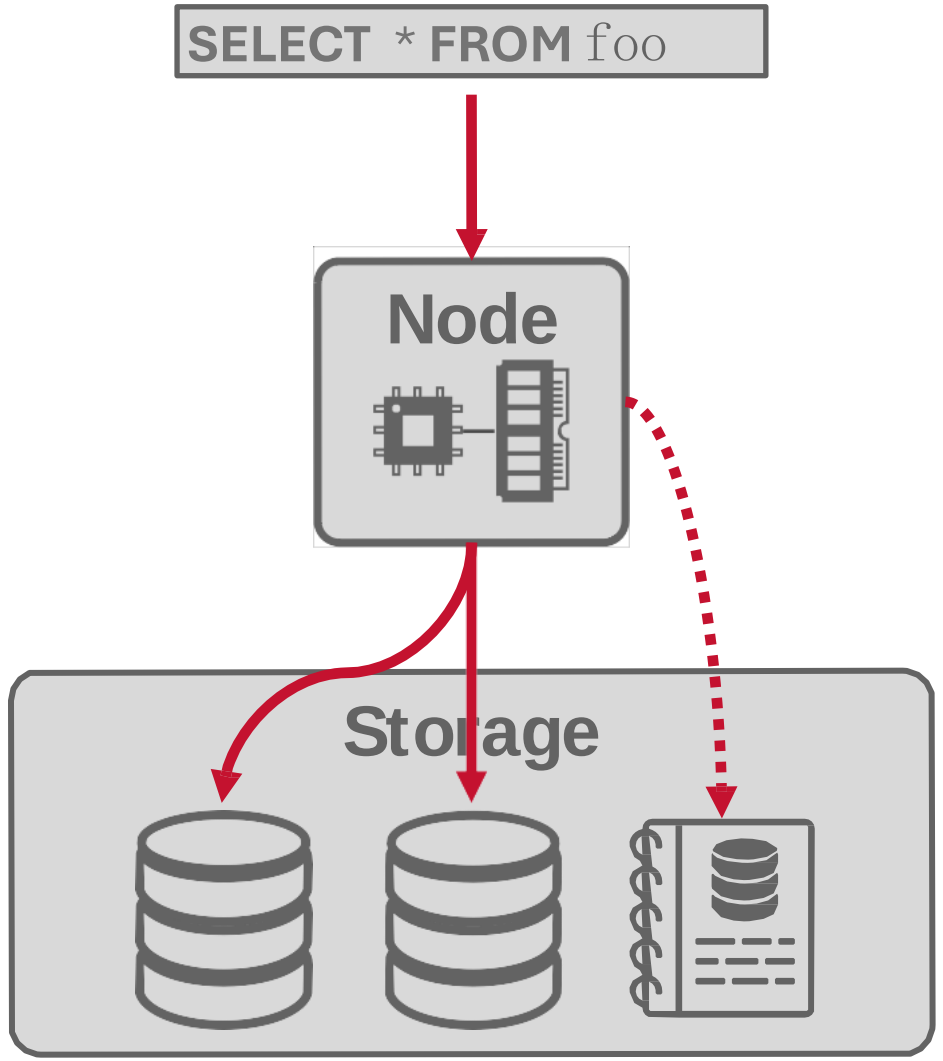
Data lakes

- Repository for storing large amounts of structured, semi-structured, and unstructured data without having to define a schema or ingest the data into proprietary internal formats.



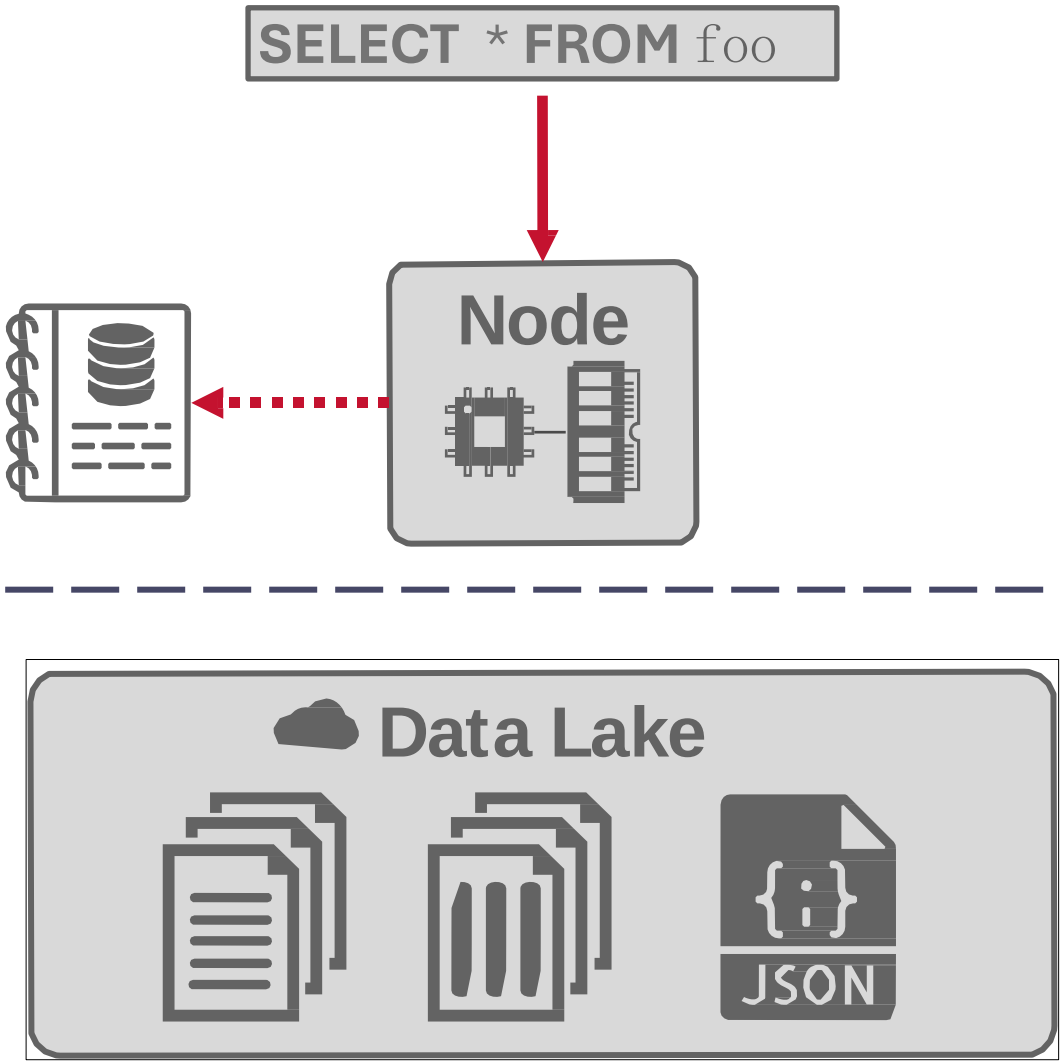
Data lakes

- Repository for storing large amounts of structured, semi-structured, and unstructured data without having to define a schema or ingest the data into proprietary internal formats.



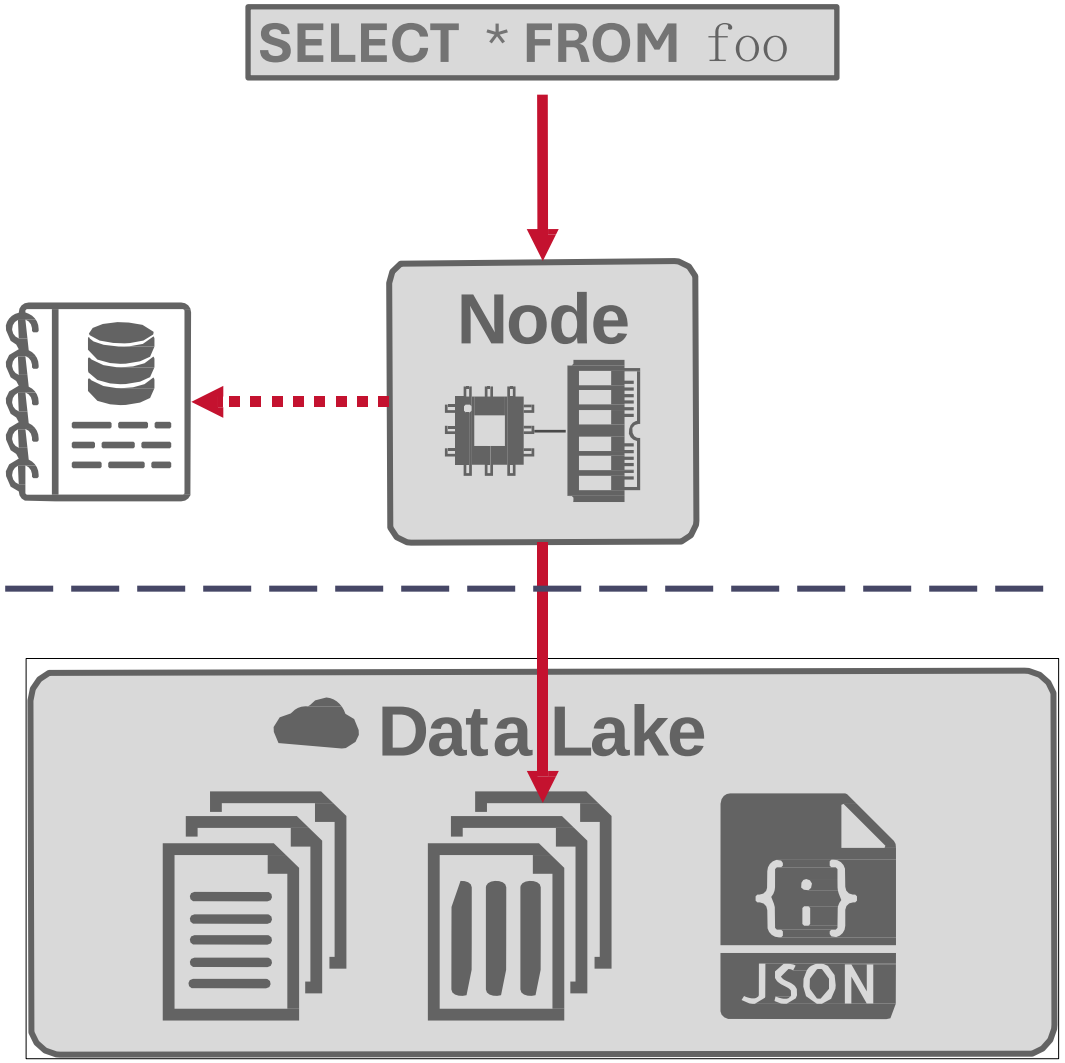
Data lakes

- Repository for storing large amounts of structured, semi-structured, and unstructured data without having to define a schema or ingest the data into proprietary internal formats.



Data lakes

- Repository for storing large amounts of structured, semi-structured, and unstructured data without having to define a schema or ingest the data into proprietary internal formats.



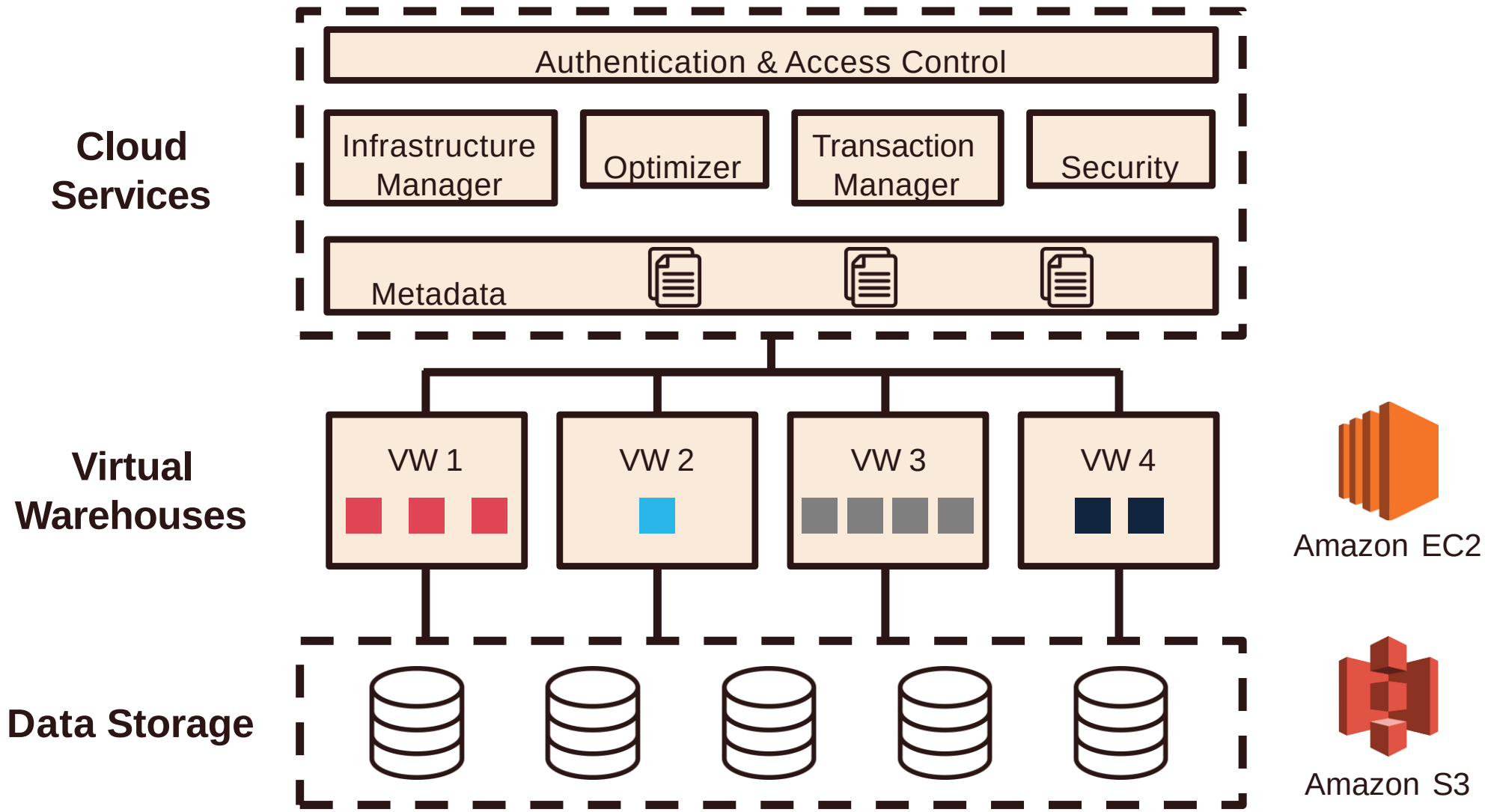
trino amazon REDSHIFT Google Big Query HIVE

databricks snowflake presto

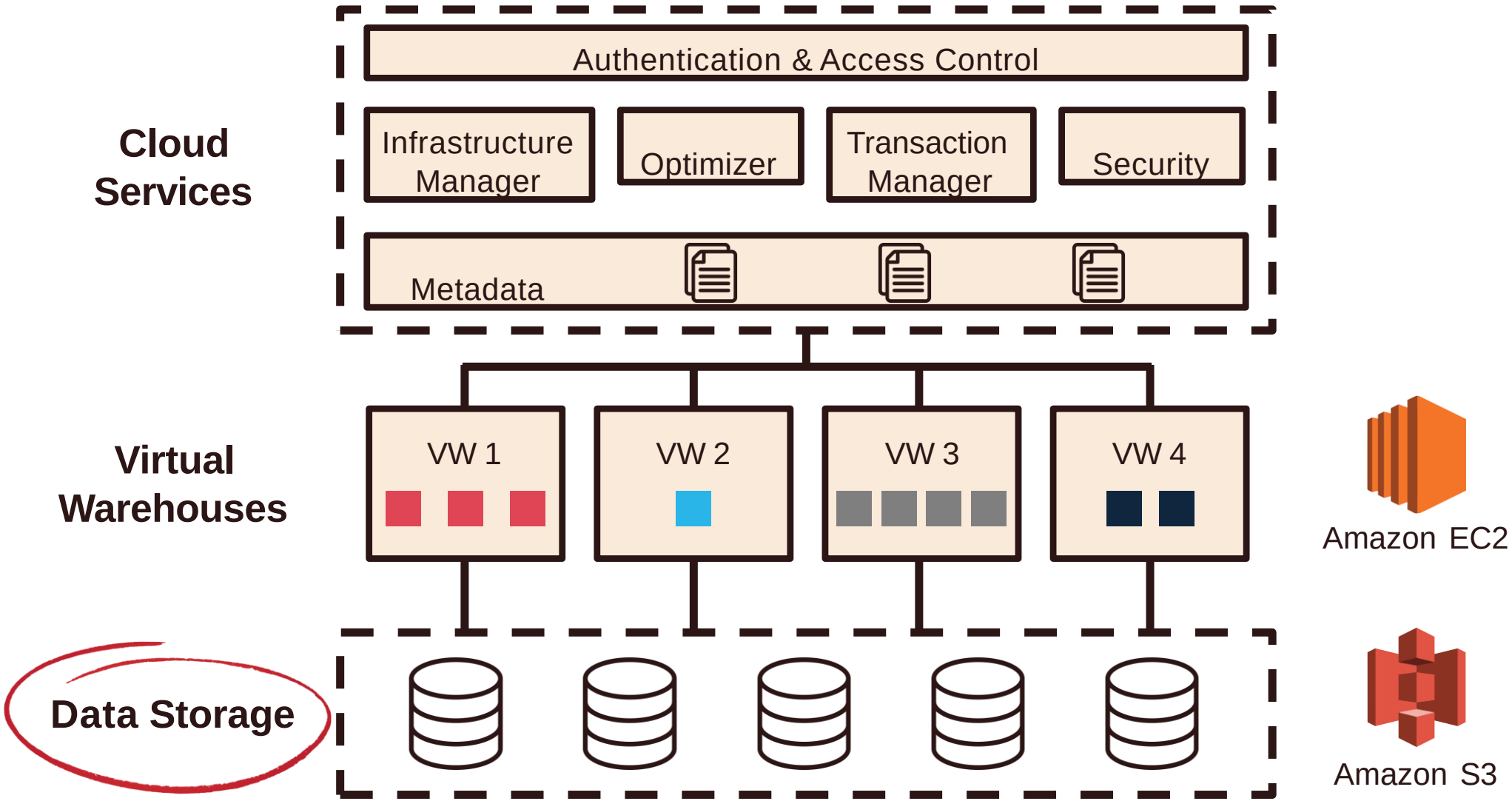
Snowflake

- The Snowflake Elastic Data Warehouse
  - Target analytical queries to support business intelligence (BI)
  - Founded at 2012, growing fast, largest software IPO (2020) ever
- Pure SaaS
  - Nothing to install, always on, always up-to-date
  - Ease of use, only pay for what you use
- Multi-Cloud Support

Snowflake architecture



Snowflake architecture



Recall from last lecture: storage models

Choice #1: N-ary Storage Model **(NSM)**

- This is for OLTP and what we learned before.

Choice #2: Decomposition Storage Model **(DSM)**

- This is for OLAP

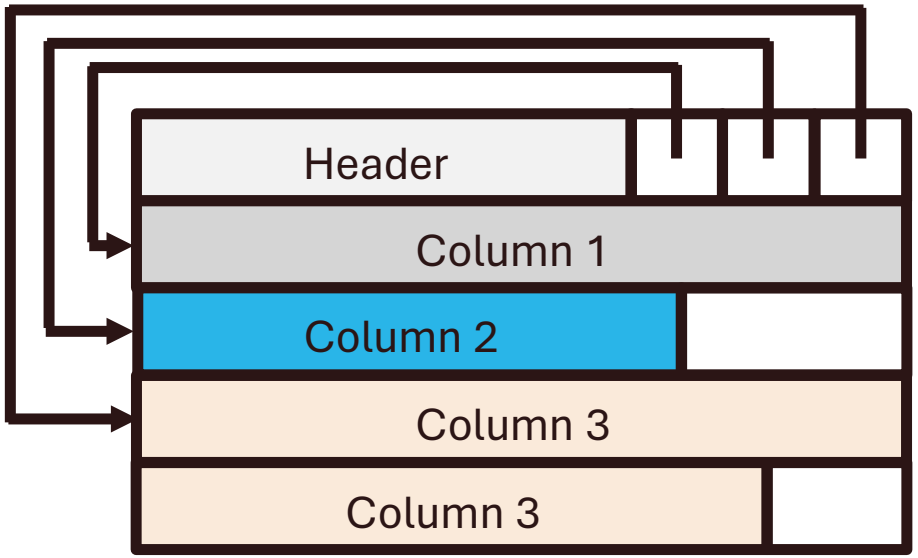
Choice #3: Hybrid Storage Model **(PAX)**

- This is for HTAP (Hybrid Transactional and Analytical Processing) and OLAP workloads.

Table file format

- Tables are horizontally partitioned
  - Micro-partition: size = 10s MB, natural ingestion order

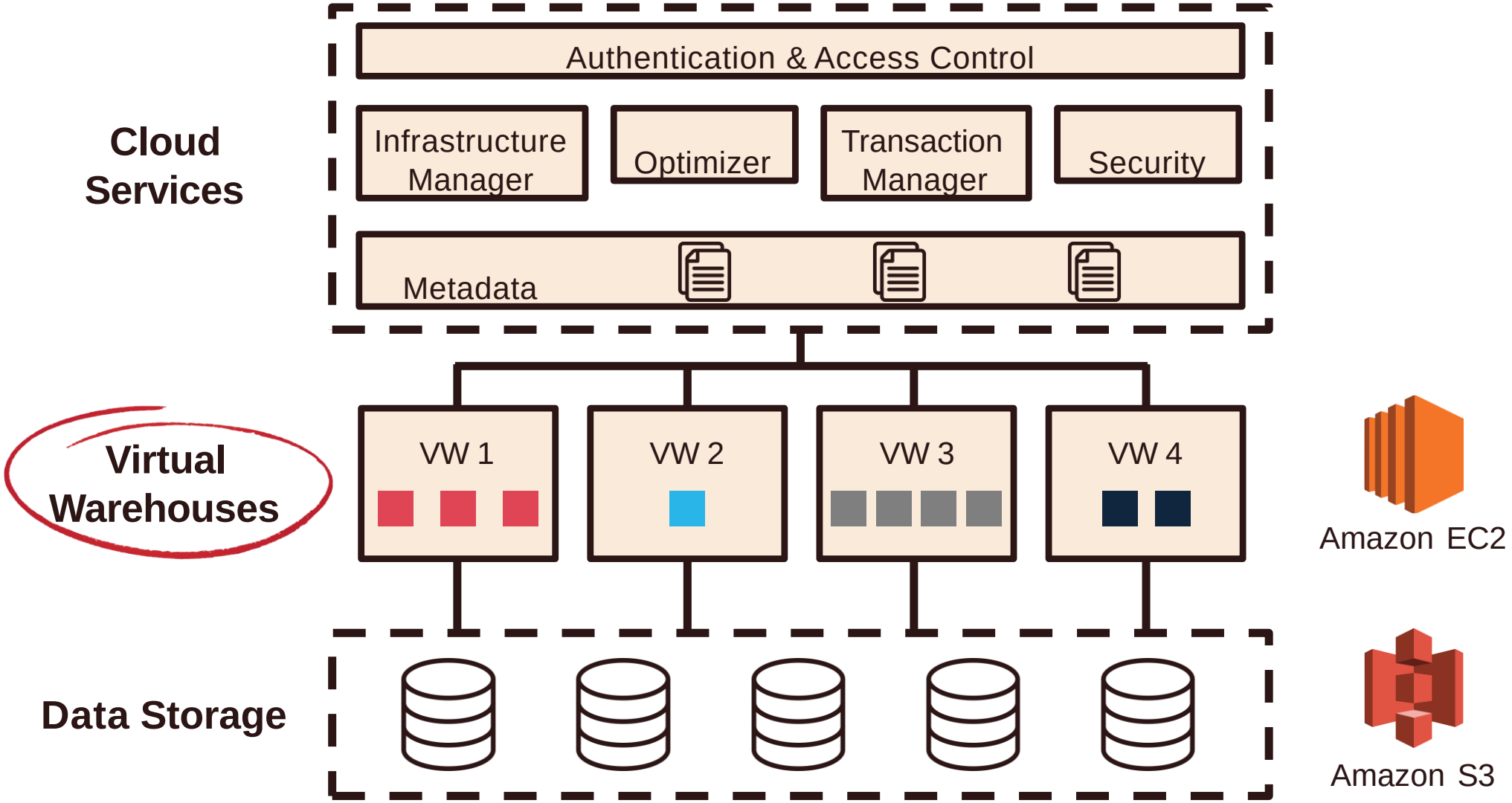
- Hybrid columnar (PAX) format



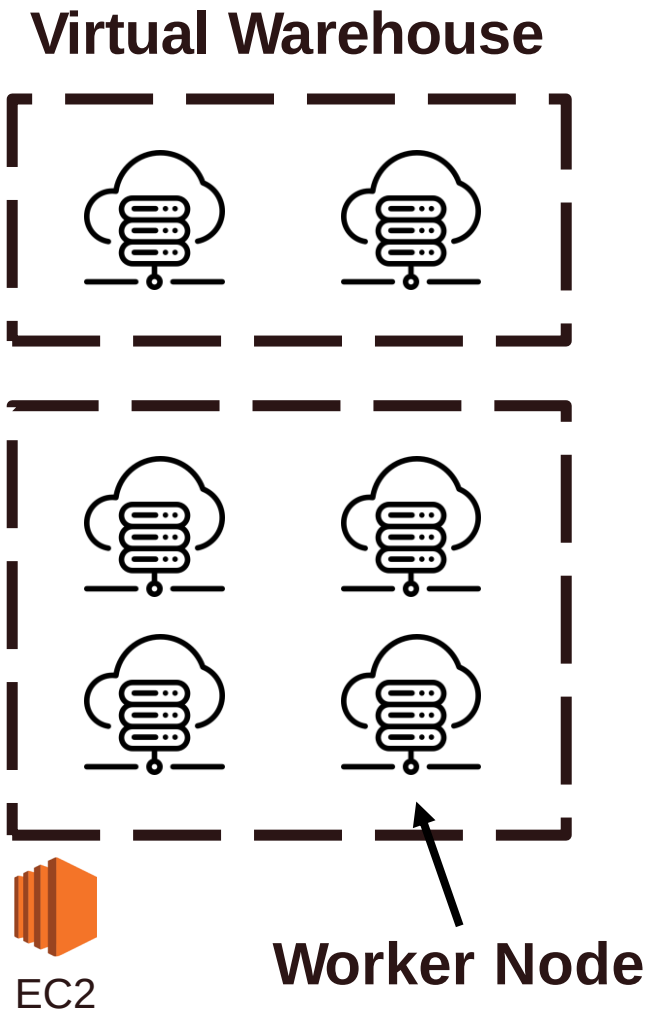
Queries first read header

Heavily Compressed

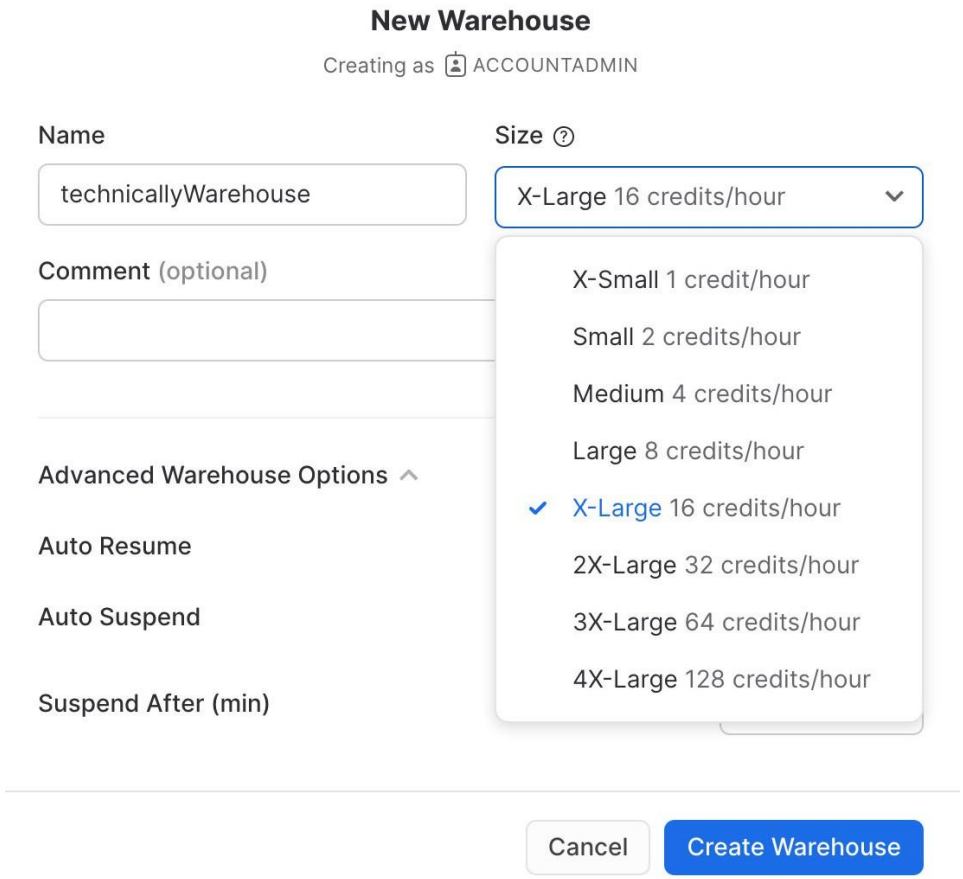
Snowflake architecture



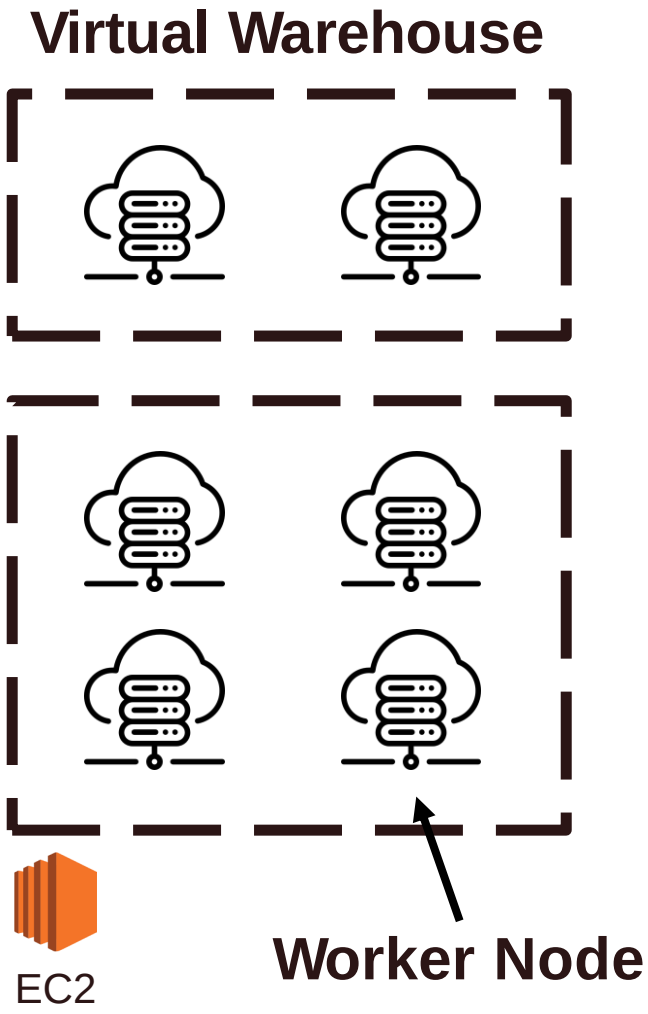
Virtual warehouses: the muscle



- Create, destroy, resize on demand



Virtual warehouses: the muscle



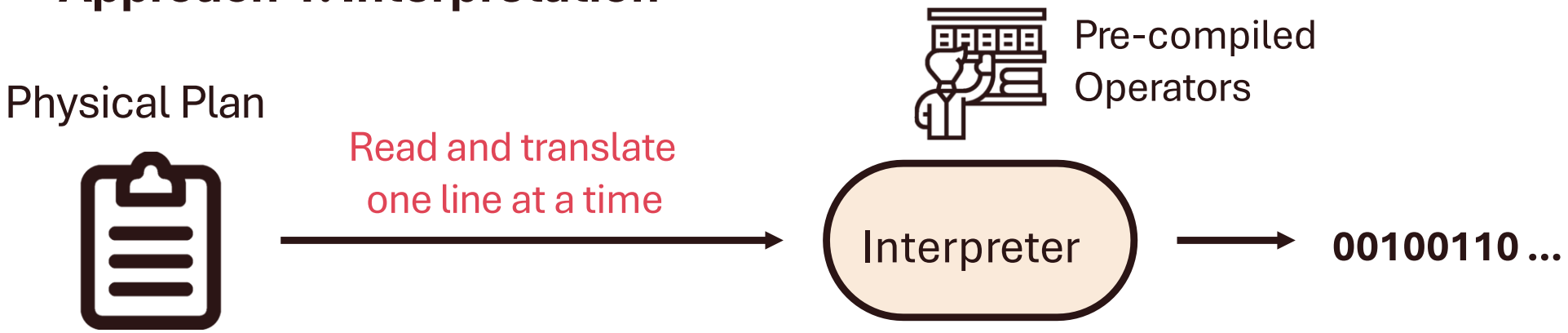
- Create, destroy, resize on demand
- Performance Isolation
  - Shared data, private compute
  - Typical usage pattern
    - Continuously-running VWs for repeating jobs
    - On-demand VWs for ad-hoc tasks
- Ephemeral worker processes
- **Columnar, Vectorized, Push-Based**

Execution engine design space

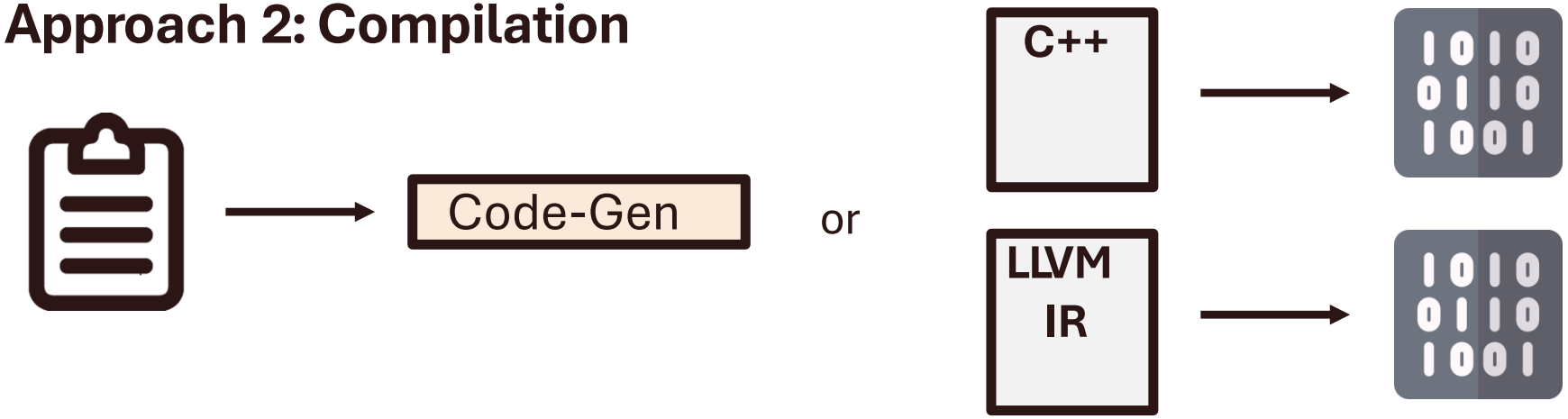
- **Engine Type**
  - Interpretation
  - Compilation (Code-Gen)
- **Execution Model**
  - Iterator / Volcano
  - Fully-Materialized
  - Vectorization
- **Pipeline Direction**
  - Pull
  - Push

Executing the plan

- **Approach 1: Interpretation**

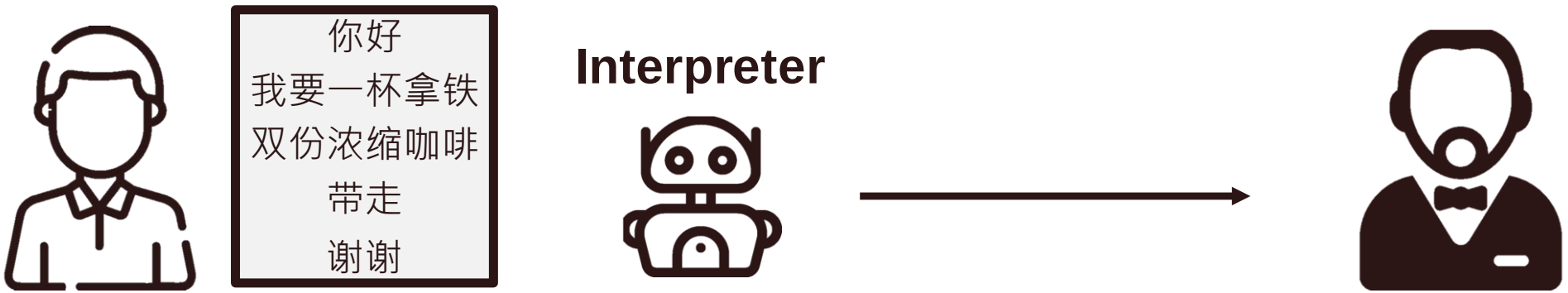


- **Approach 2: Compilation**



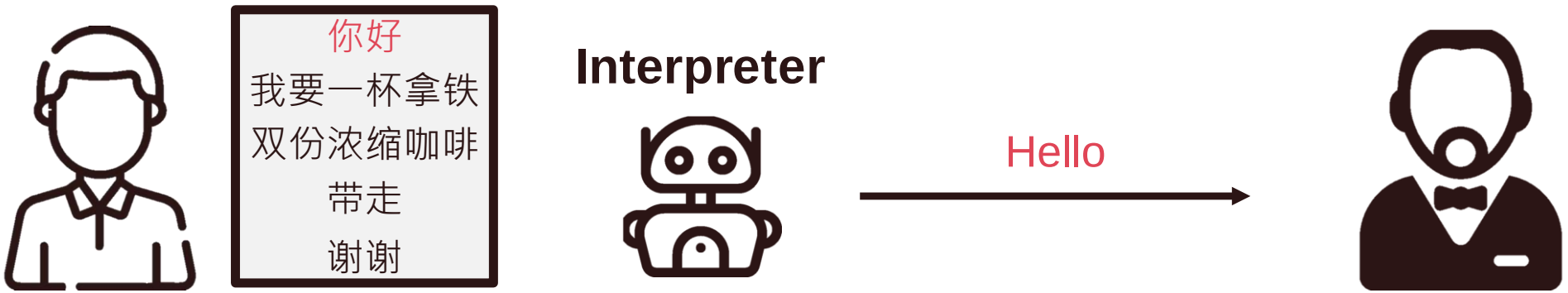
Aside: interpreting vs. compiling

- **Complier**: bring the pre-translated sheet



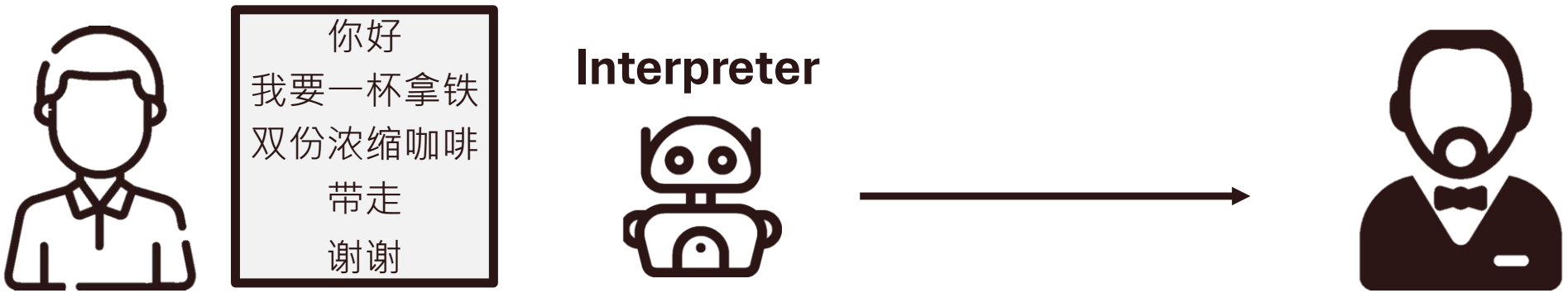
Aside: interpreting vs. compiling

- **Complier**: bring the pre-translated sheet



Aside: interpreting vs. compiling

- **Complier**: bring the pre-translated sheet



Pros

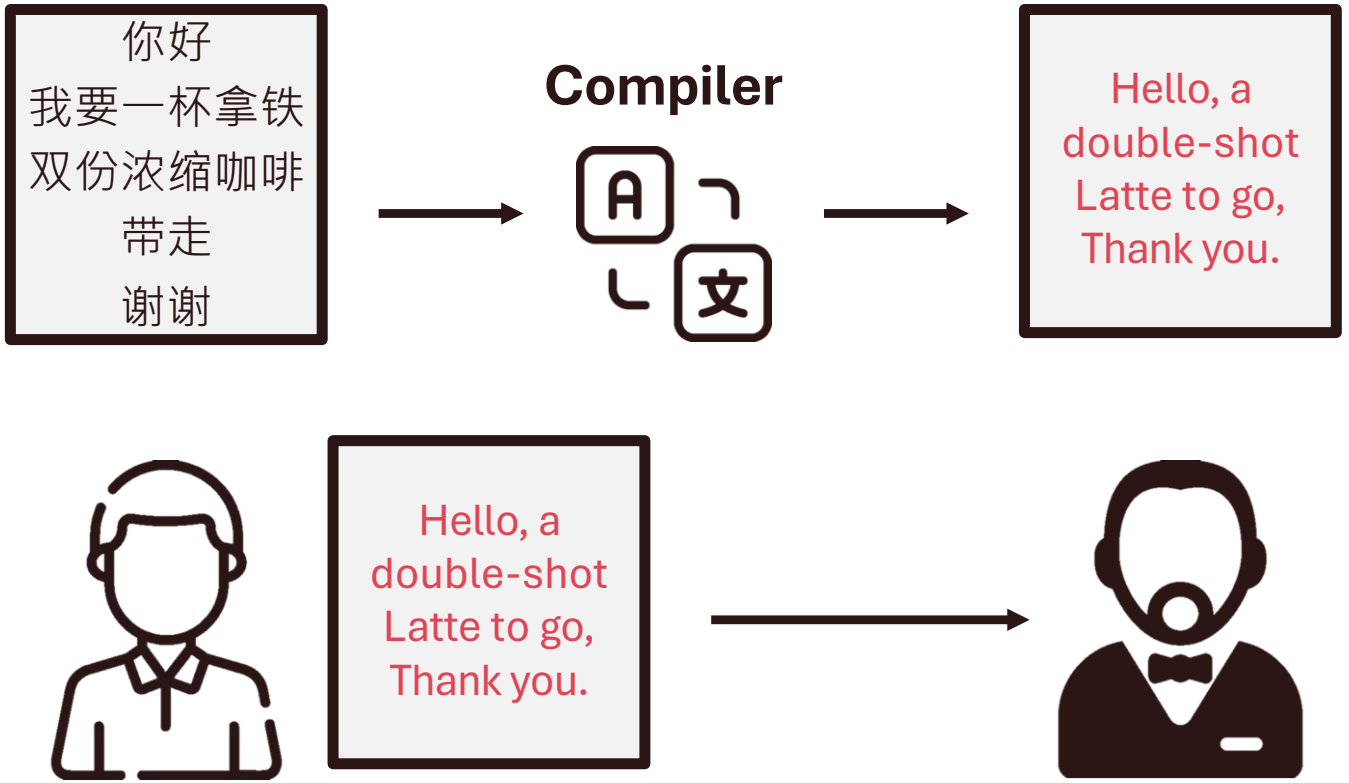
- No need for code analysis beforehand
- Easier to test and debug
- Cross-platform

Cons

- Need interpreters
- Execution is often slower

Aside: interpreting vs. compiling

- **Complier**: bring the pre-translated sheet



Pros

- Faster, ready-to-run
- Code better optimized

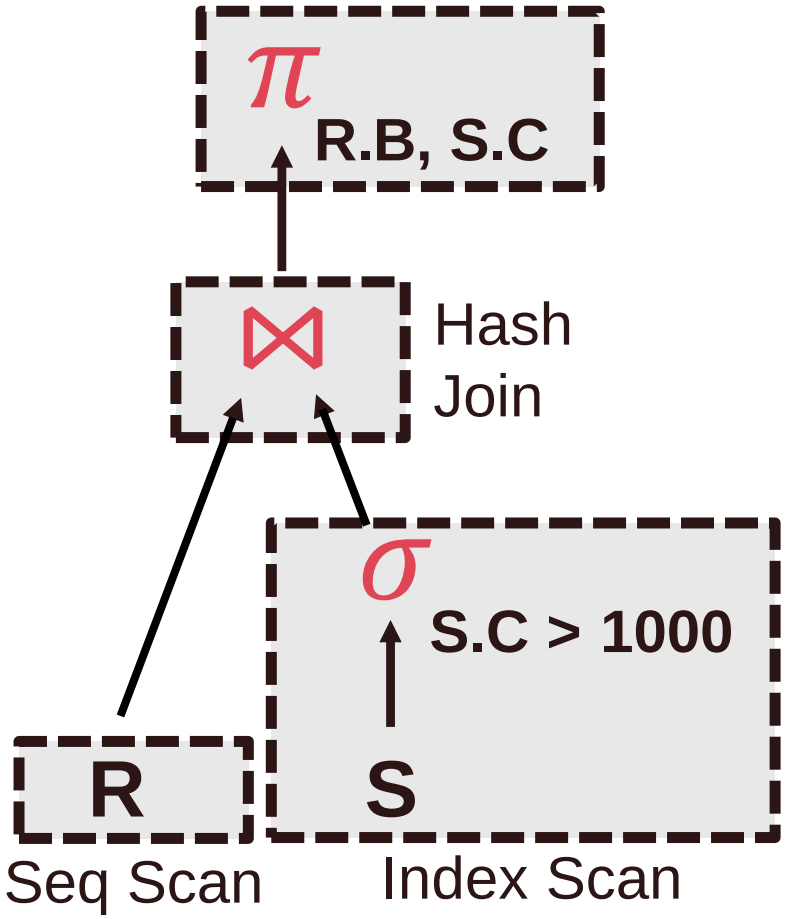
Cons

- Long extra compile time
- Requires more memory
- Hard to get it right
- Worse portability

Execution engine design space

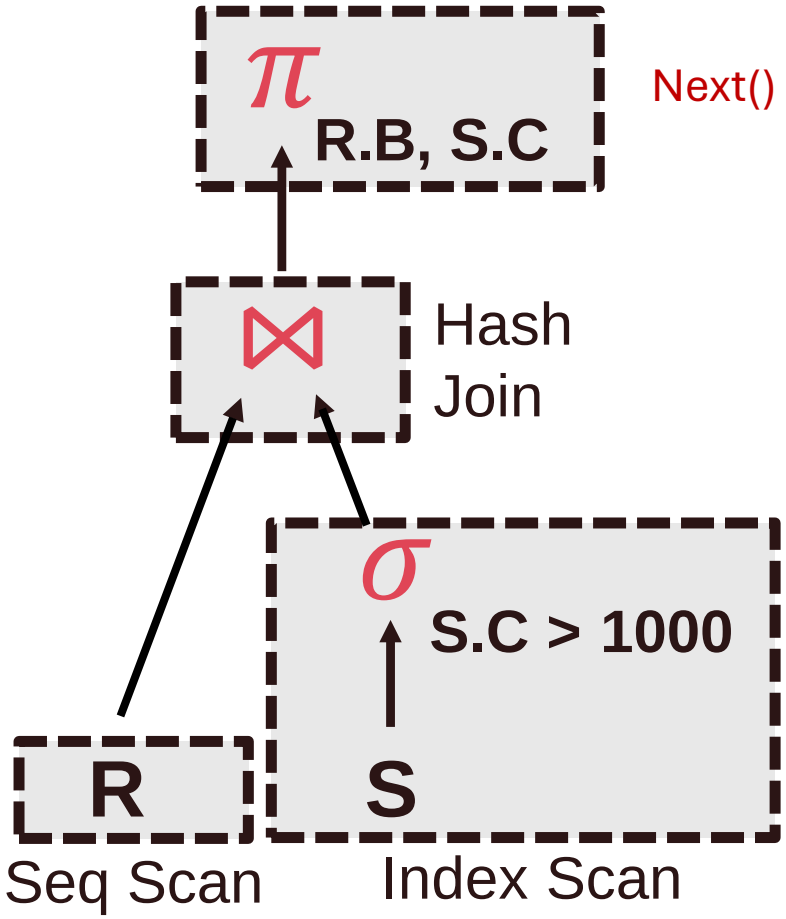
- **Engine Type**
  - **Interpretation**
  - Compilation (Code-Gen)
- **Execution Model**
  - Iterator / Volcano
  - Fully-Materialized
  - **Vectorization**
- **Pipeline Direction**
  - Pull
  - **Push**

Iterator/volcano model



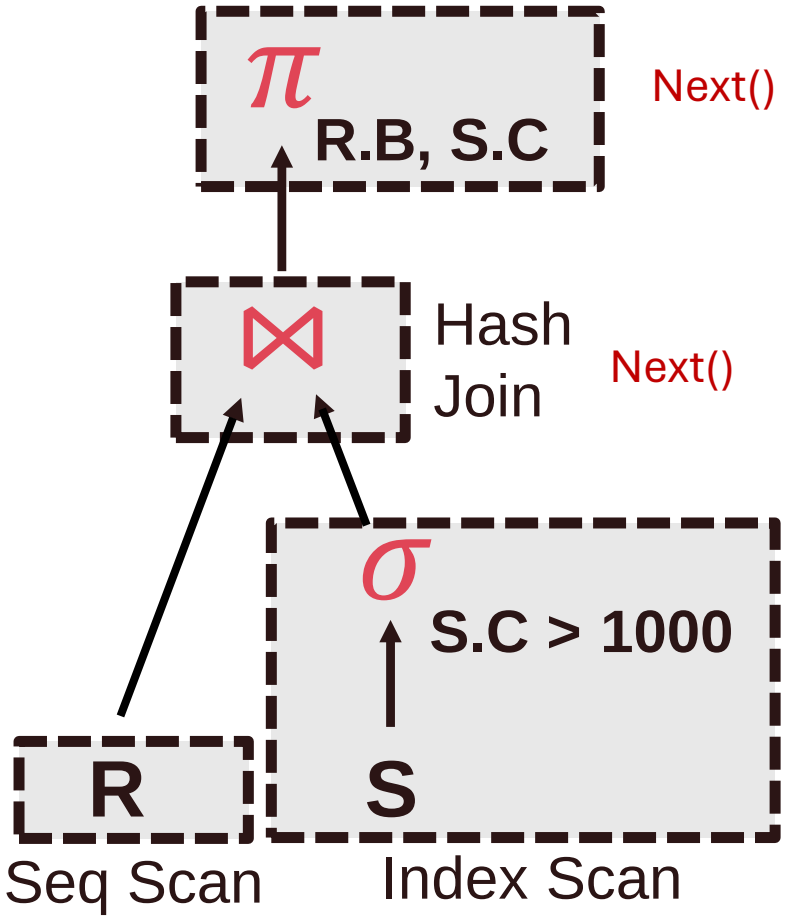
- Each operator implements Next()
  - Emits an output tuple or NULL
- The root operator implements a **loop** that keeps invoking **Next()** on its child
- Execution can be **pipelined**
- Could have **pipeline breakers**
  - E.g., join, order by
- Elegant implementation

Iterator/volcano model



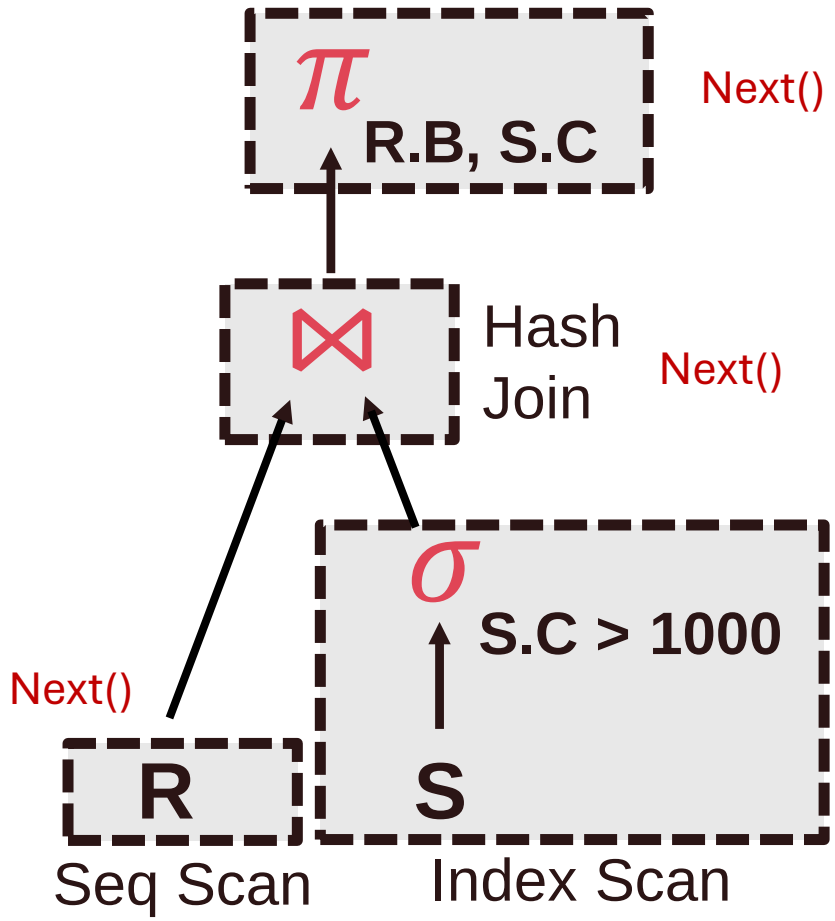
- Each operator implements Next()
  - Emits an output tuple or NULL
- The root operator implements a **loop** that keeps invoking **Next()** on its child
- Execution can be **pipelined**
- Could have **pipeline breakers**
  - E.g., join, order by
- Elegant implementation

Iterator/volcano model



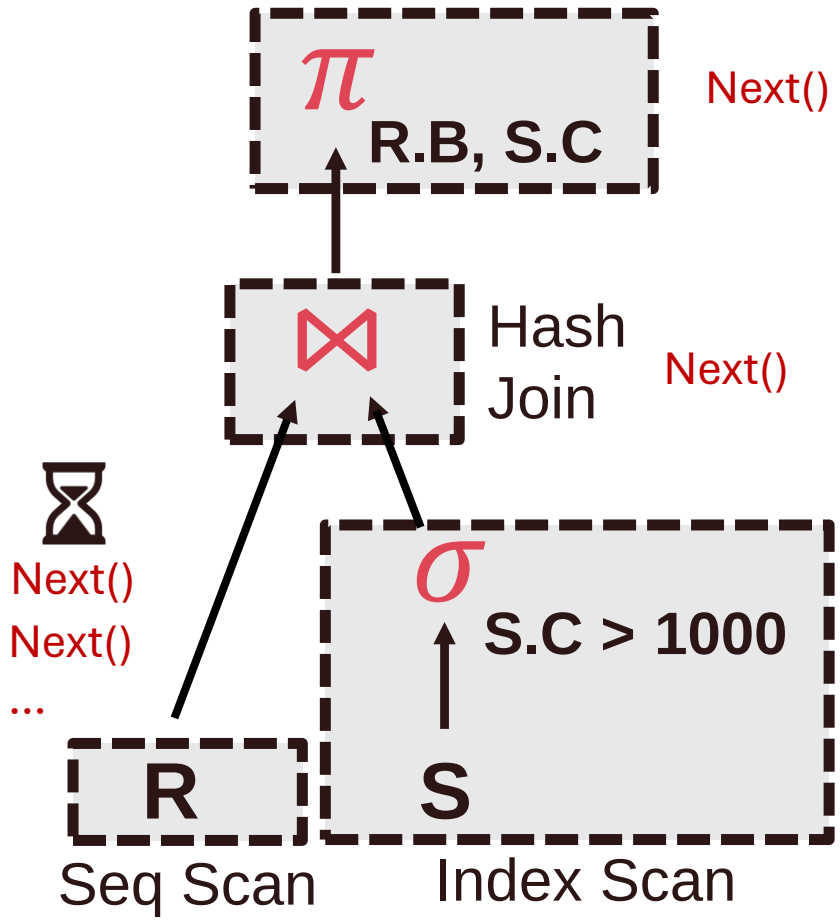
- Each operator implements Next()
  - Emits an output tuple or NULL
- The root operator implements a **loop** that keeps invoking **Next()** on its child
- Execution can be **pipelined**
- Could have **pipeline breakers**
  - E.g., join, order by
- Elegant implementation

Iterator/volcano model



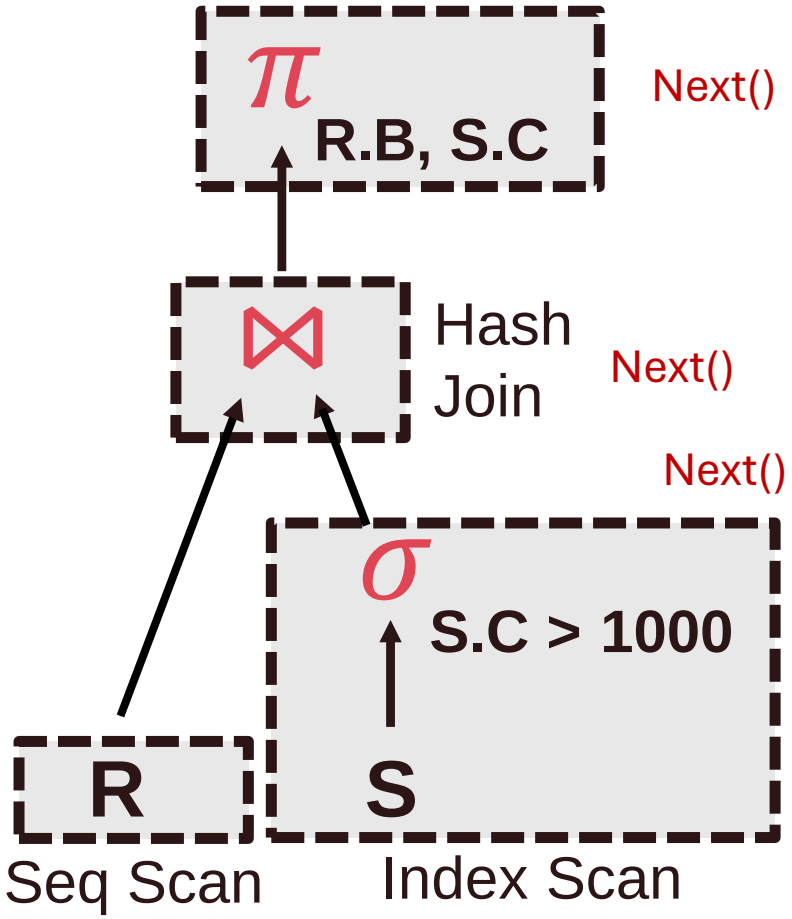
- Each operator implements Next()
  - Emits an output tuple or NULL
- The root operator implements a **loop** that keeps invoking **Next()** on its child
- Execution can be **pipelined**
- Could have **pipeline breakers**
  - E.g., join, order by
- Elegant implementation

Iterator/volcano model



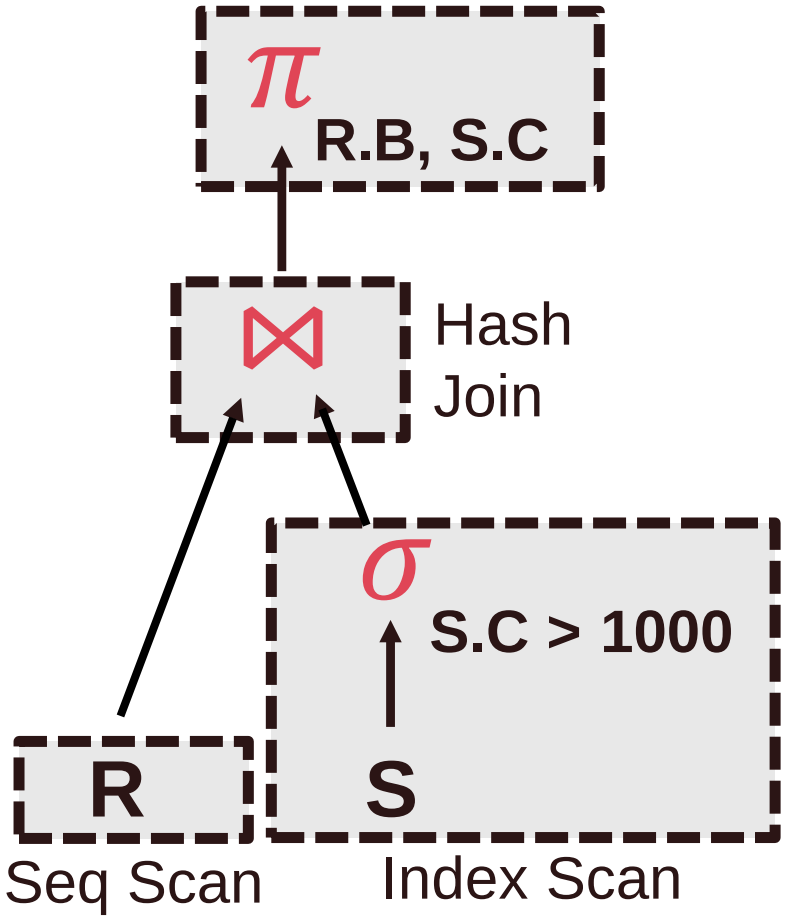
- Each operator implements Next()
  - Emits an output tuple or NULL
- The root operator implements a **loop** that keeps invoking **Next()** on its child
- Execution can be **pipelined**
- Could have **pipeline breakers**
  - E.g., join, order by
- Elegant implementation

Iterator/volcano model



- Each operator implements Next()
  - Emits an output tuple or NULL
- The root operator implements a **loop** that keeps invoking **Next()** on its child
- Execution can be **pipelined**
- Could have **pipeline breakers**
  - E.g., join, order by
- Elegant implementation

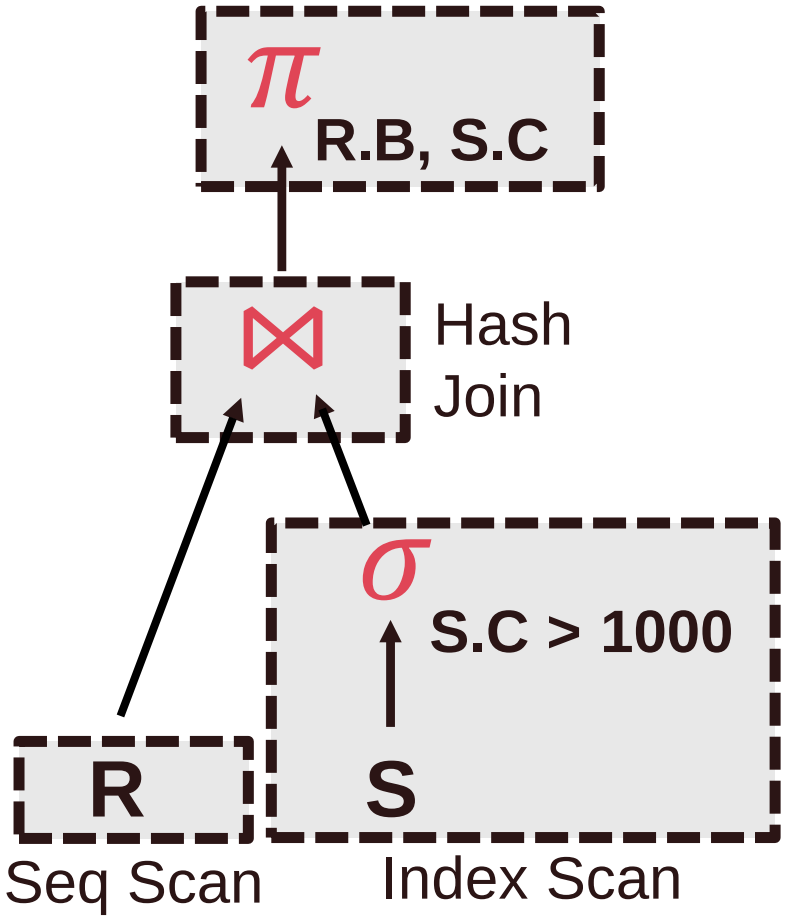
Iterator/volcano model




```
class AbstractExecutor {
    virtual void Init() = 0;
    virtual Tuple* Next() = 0;
protected:
    Context *ctx;
};
```

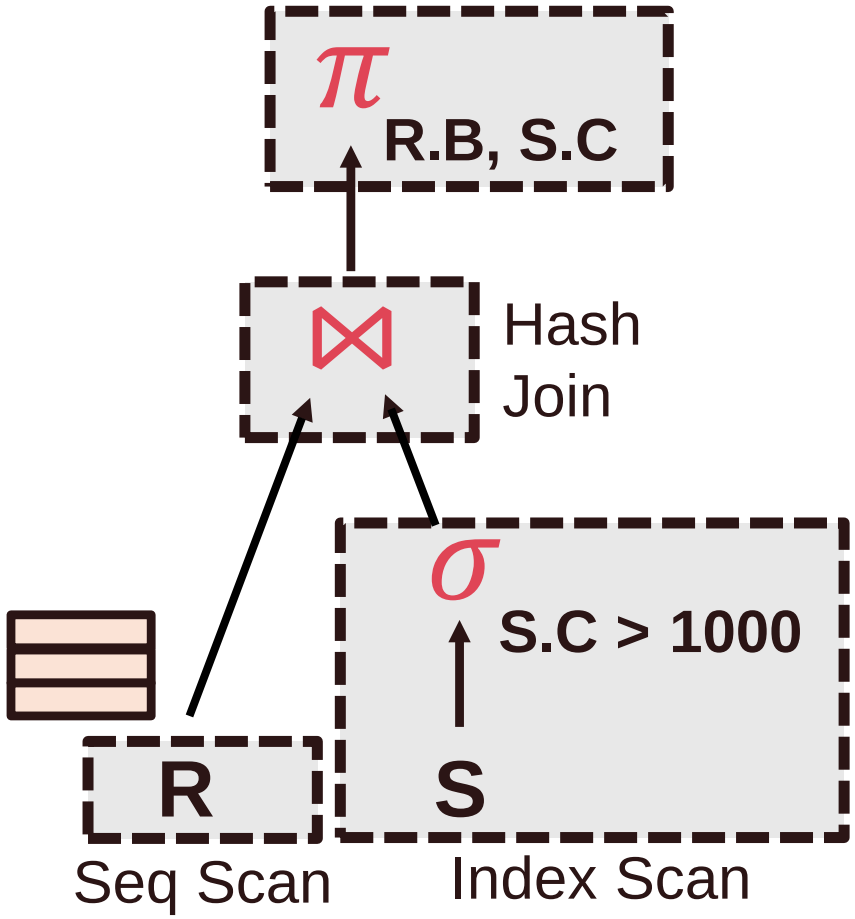
- Every operator implements the same interface
- Operators may have internal states

Fully-materialized model



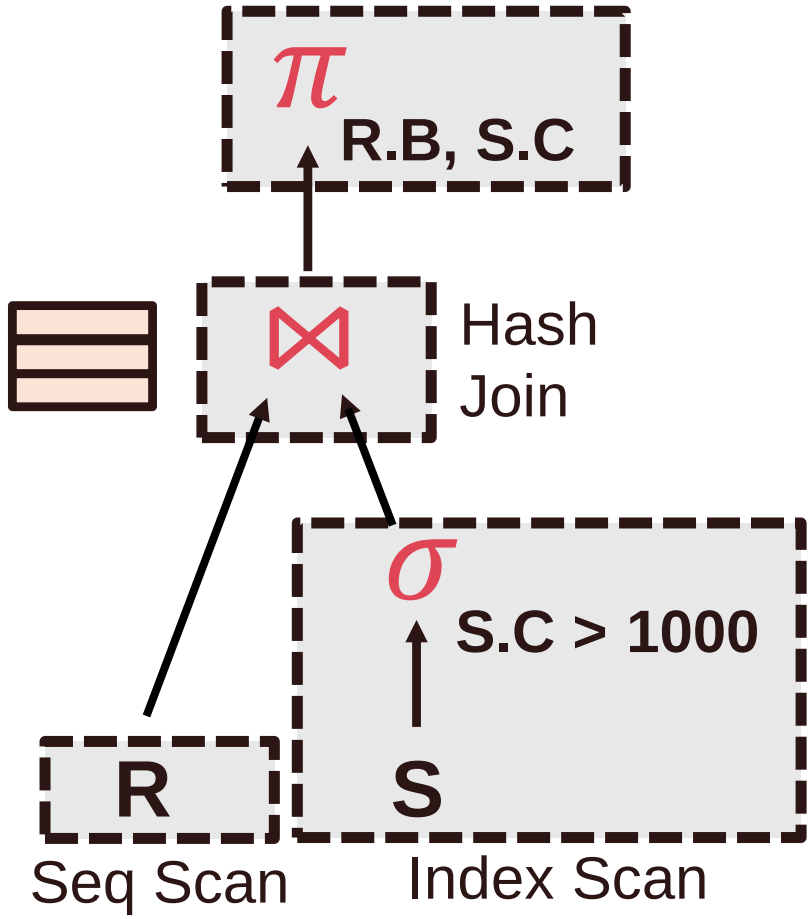
- Each operator stores its output in a single buffer and returns all at once
- Parent operator does not start until its children finish
- Non-pipelined

Fully-materialized model



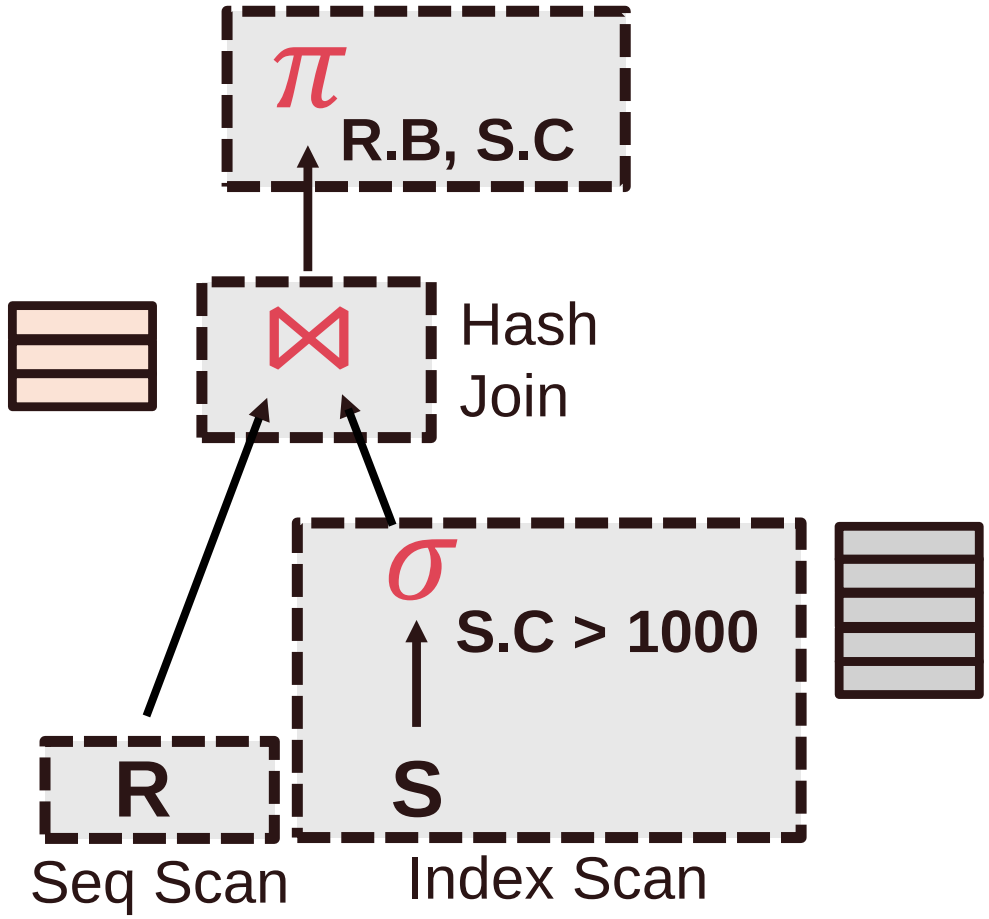
- Each operator stores its output in a single buffer and returns all at once
- Parent operator does not start until its children finish
- Non-pipelined

Fully-materialized model



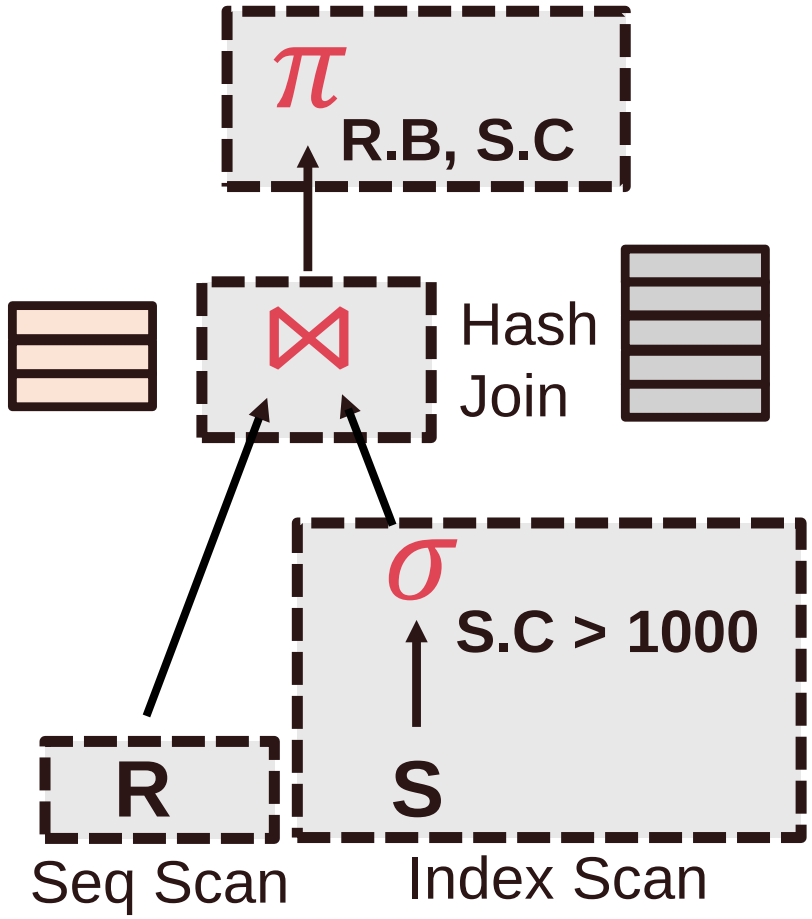
- Each operator stores its output in a single buffer and returns all at once
- Parent operator does not start until its children finish
- Non-pipelined

Fully-materialized model



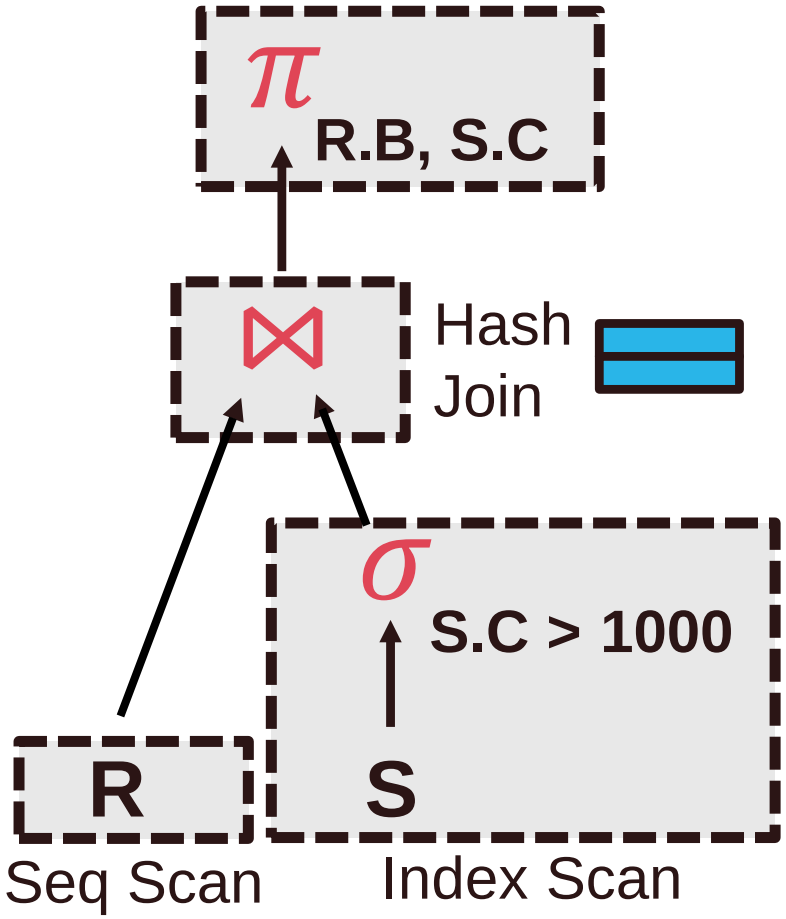
- Each operator stores its output in a single buffer and returns all at once
- Parent operator does not start until its children finish
- Non-pipelined

Fully-materialized model



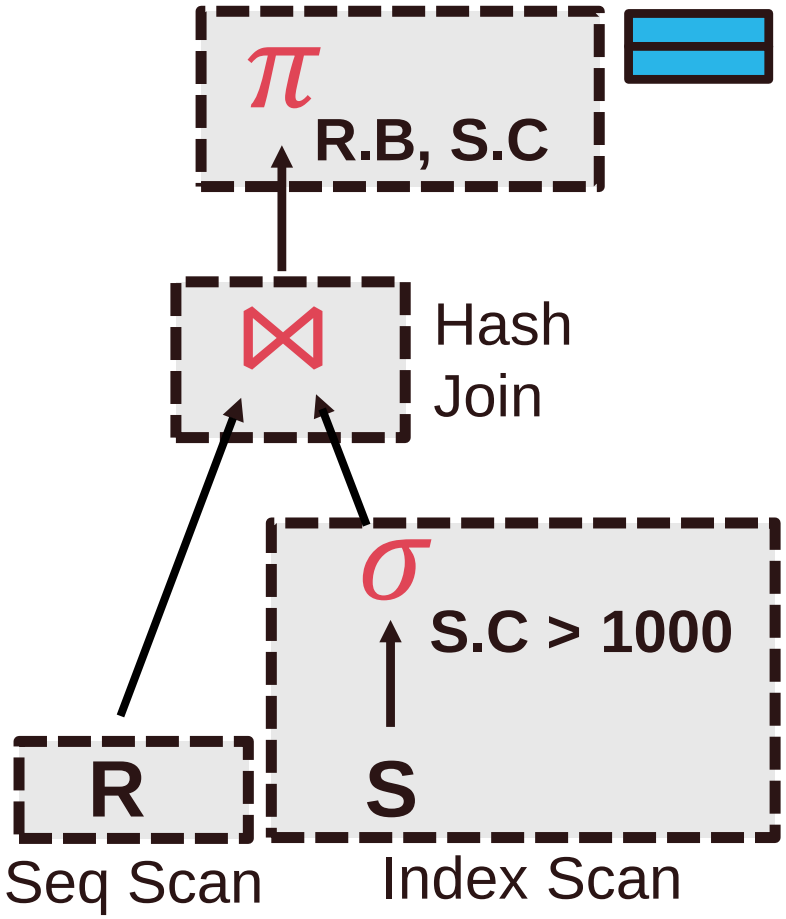
- Each operator stores its output in a single buffer and returns all at once
- Parent operator does not start until its children finish
- Non-pipelined

Fully-materialized model



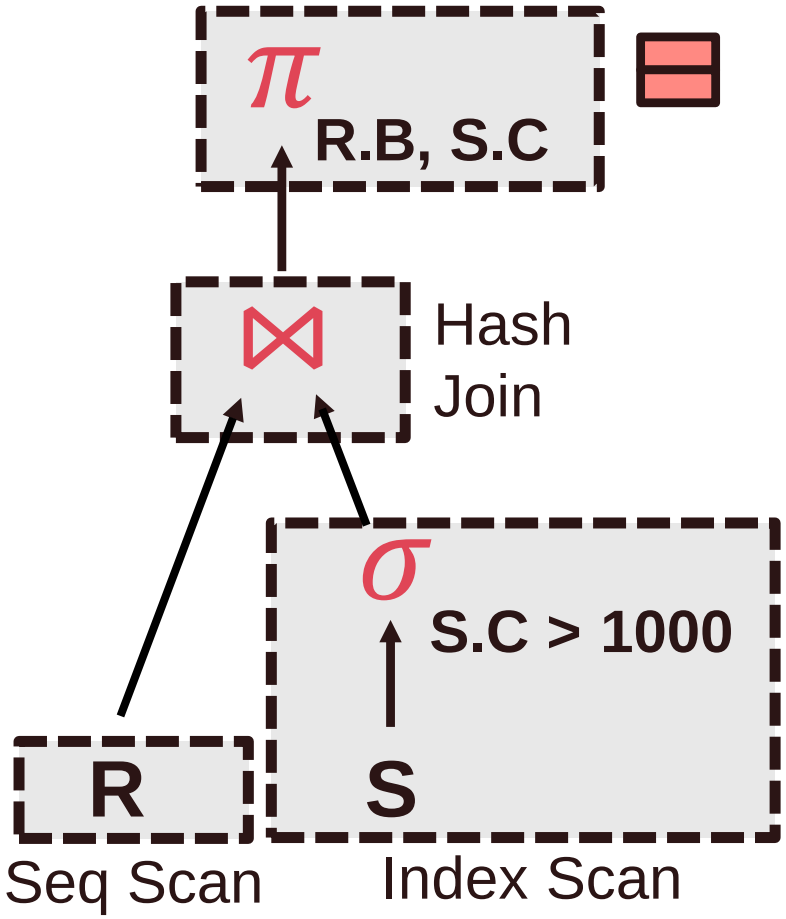
- Each operator stores its output in a single buffer and returns all at once
- Parent operator does not start until its children finish
- Non-pipelined

Fully-materialized model



- Each operator stores its output in a single buffer and returns all at once
- Parent operator does not start until its children finish
- Non-pipelined

Fully-materialized model



- Each operator stores its output in a single buffer and returns all at once
- Parent operator does not start until its children finish
- Non-pipelined

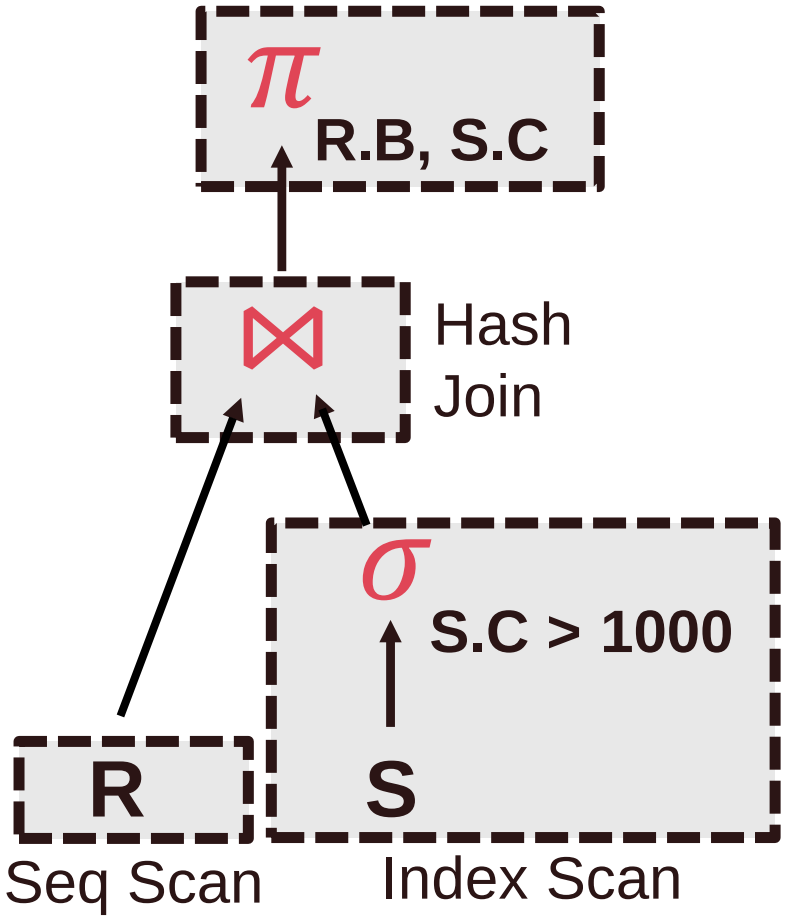
Iterator vs. fully-materialized

Iterator / Volcano	**Fully-Materialized**
Tuple at a time	Operator at a time
Small intermediate results	Need extra memory for intermediate results
A lot of virtual function calls	Fewer function calls
Can benefit from pipelining	Can benefit from batch processing (e.g, SIMD)
Adopted by almost every OLTP DBMS	A few DBMSs (e.g., monetDB, VoltDB)

Iterator vs. fully-materialized

Iterator / Volcano	**Fully-Materialized**
Tuple at a time	Operator at a time
Small intermediate results	Need extra memory for intermediate results
A lot of virtual function calls	Fewer function calls
Can benefit from pipelining	Can benefit from batch processing (e.g, SIMD)
Adopted by almost every OLTP DBMS	A few DBMSs (e.g., monetDB, VoltDB)

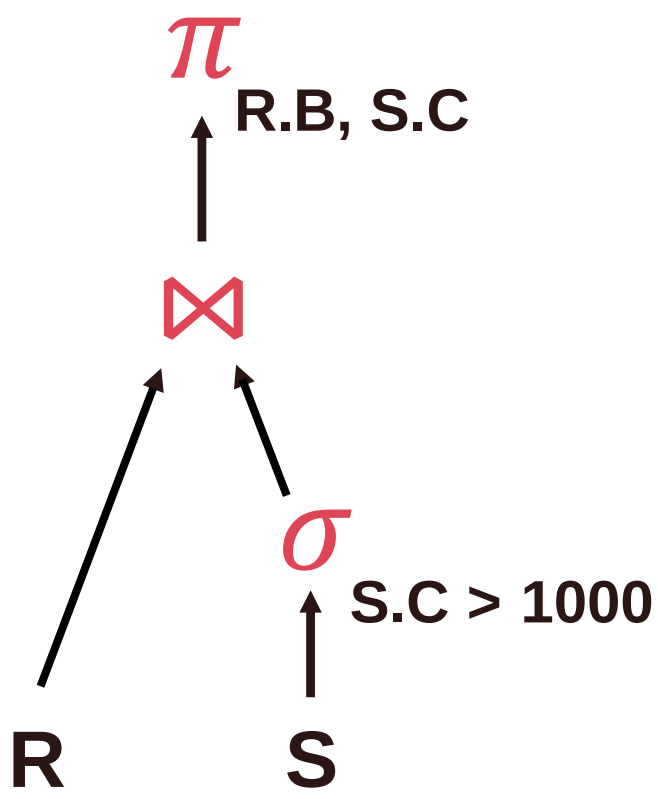
Vectorization model



- Every operator implements **NextBatch( )**
  - Emits a batch of tuples
- Can use SIMD instructions in operator's internal loop to accelerate processing
- Much fewer function calls compared to the iterator model
- Ideal for OLAP
  - Adopted by most interpreted OLAP engines today

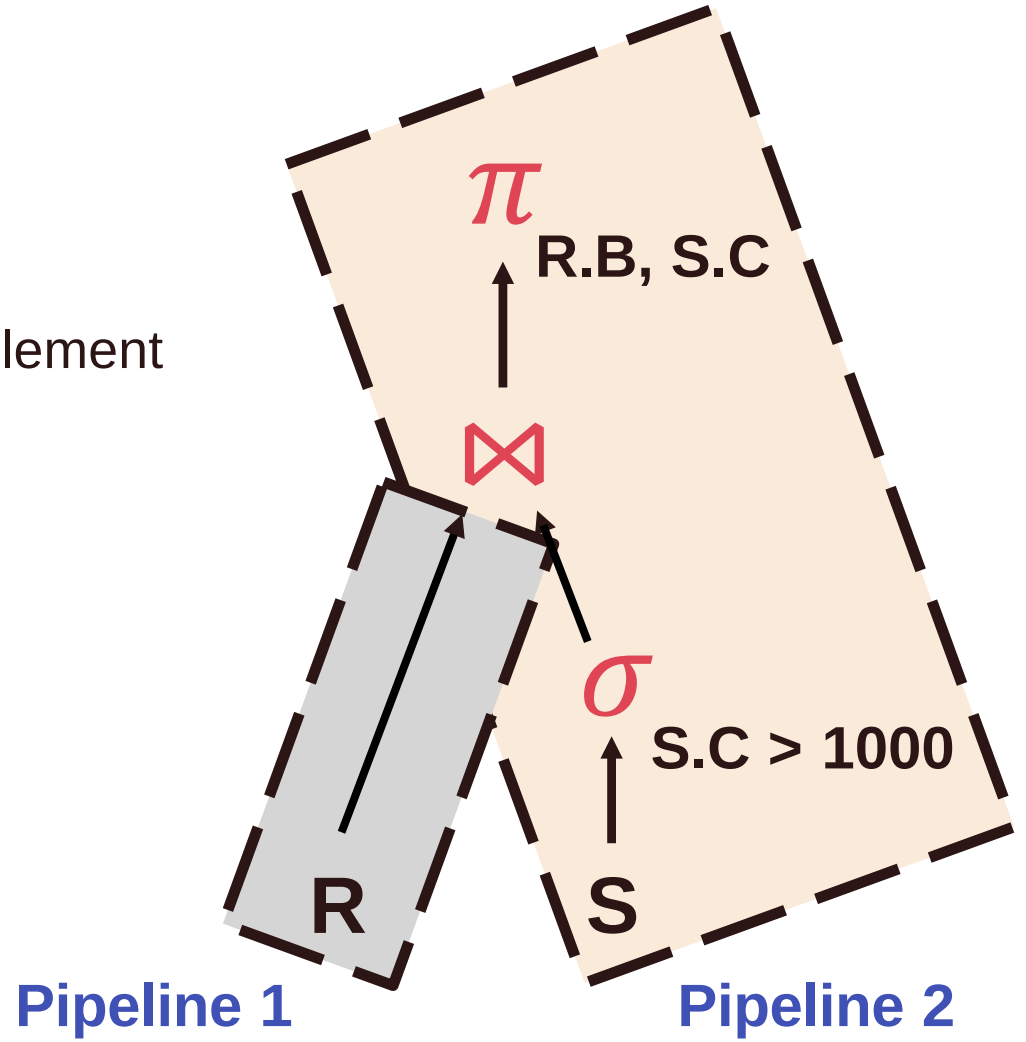
Pipeline direction

- **Pull**
  - Parent operator “pulls” data up from its children
  - Via function calls such as Next( )
  - Most common way, easier to understand and implement
- **Push**
  - Child operator “pushes” data to its parent
  - Similar to producer-consumer model
  - Easier to “fuse” operations so that data stays in CPU register as long as possible

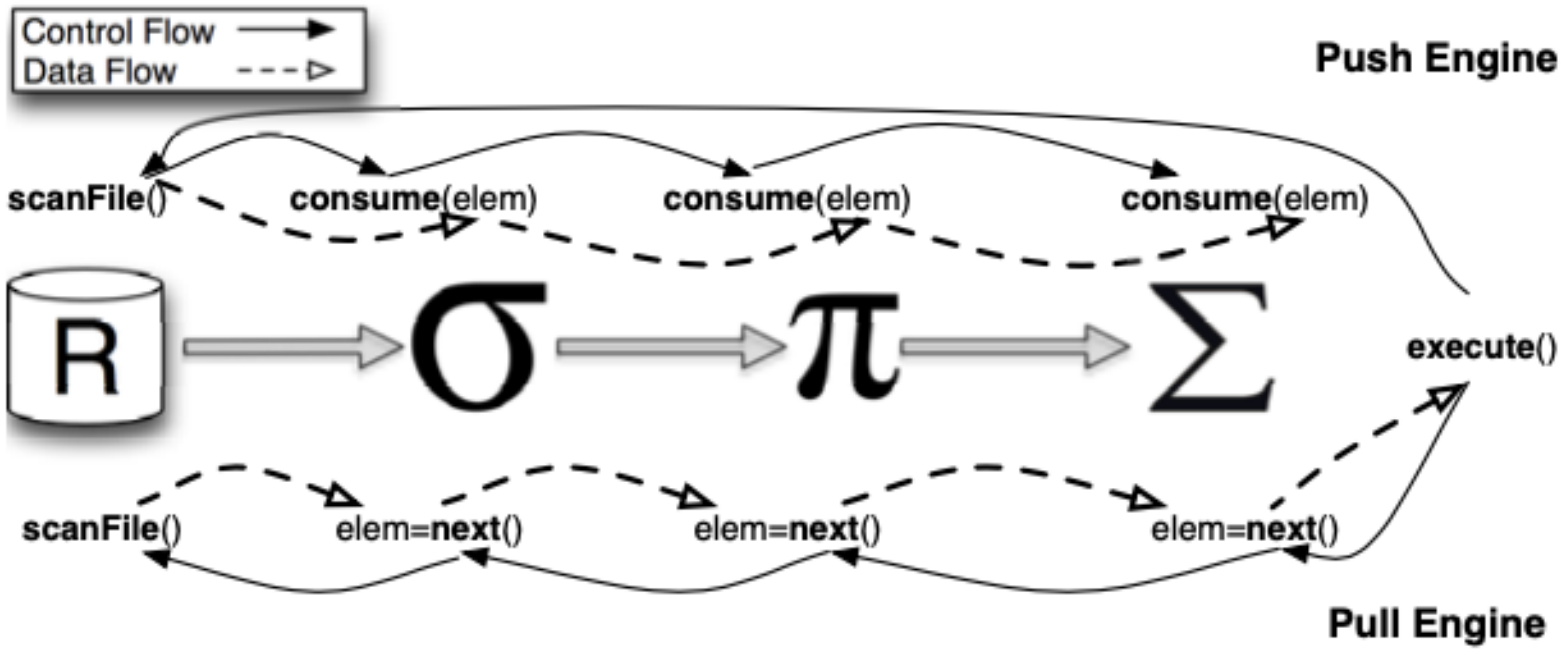


Pipeline direction

- **Pull**
  - Parent operator “pulls” data up from its children
  - Via function calls such as Next( )
  - Most common way, easier to understand and implement
- **Push**
  - Child operator “pushes” data to its parent
  - Similar to producer-consumer model
  - Easier to “fuse” operations so that data stays in CPU register as long as possible

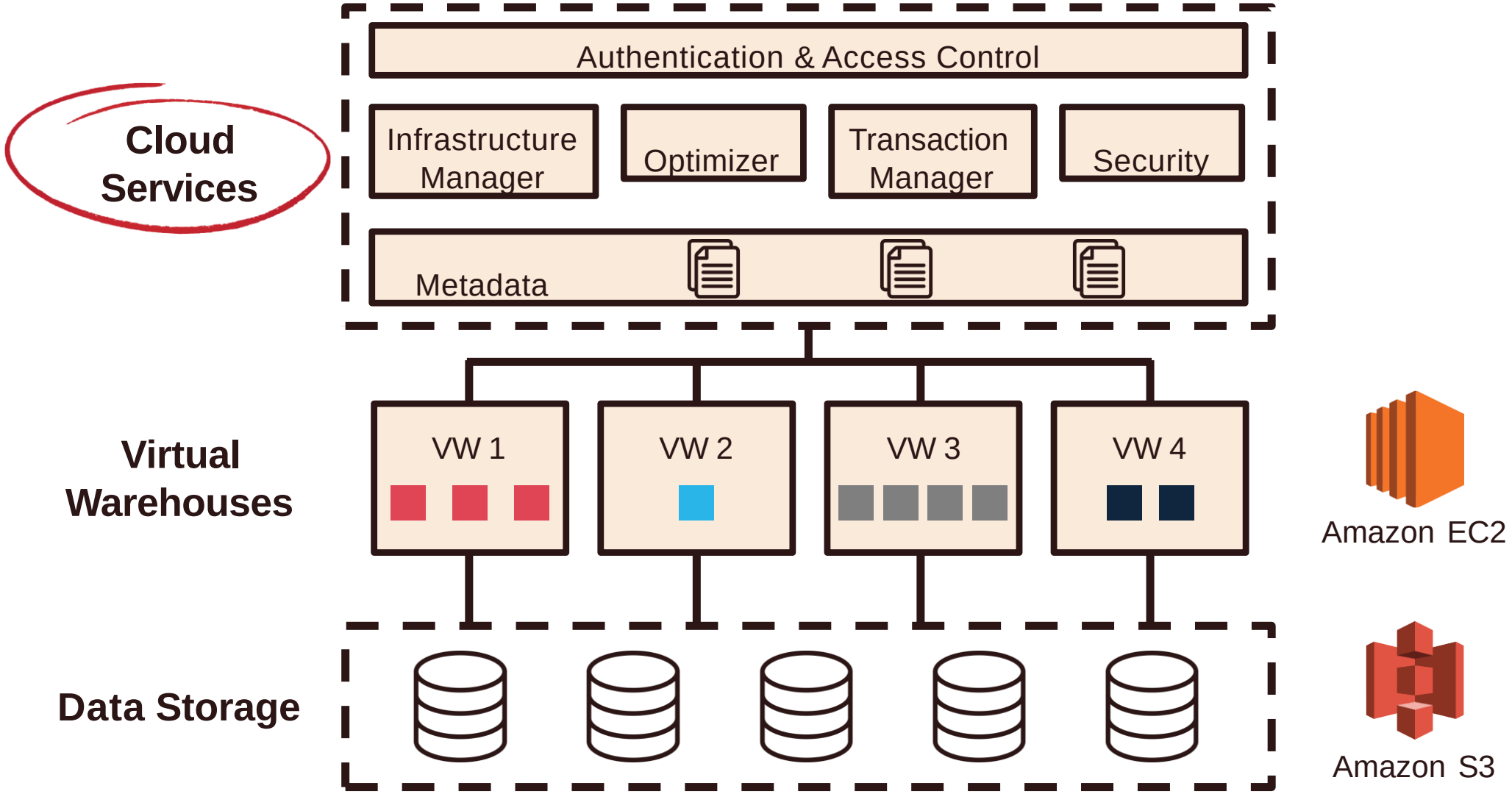


Pipeline direction

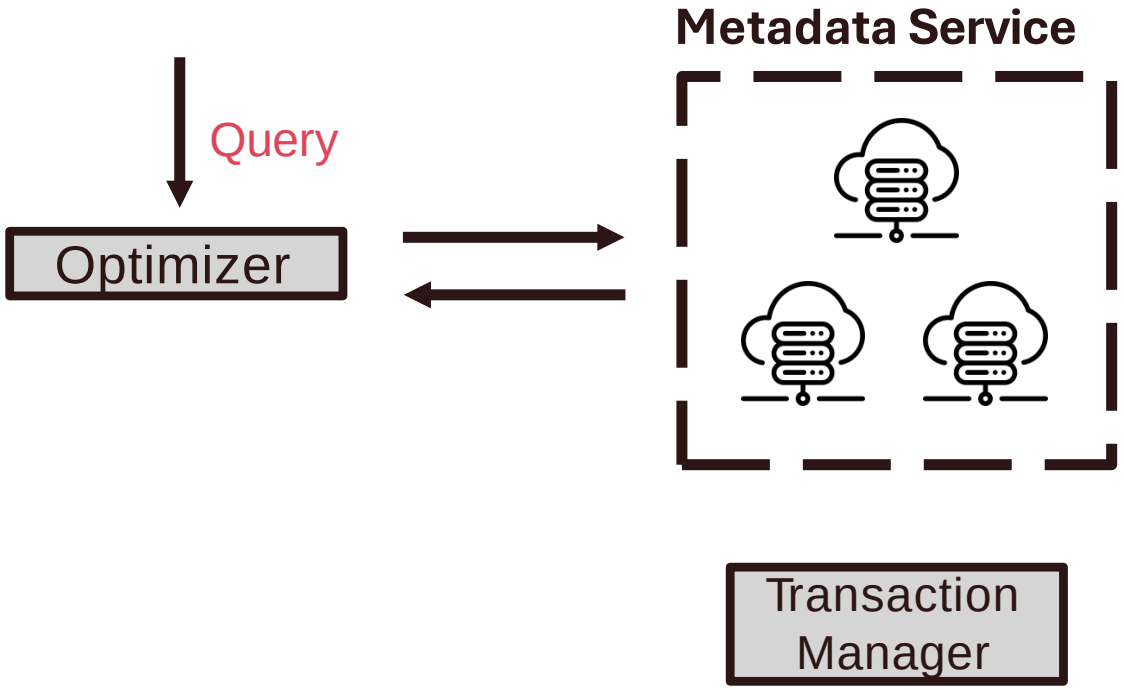



```
SELECT SUM(R.B)
FROM R
WHERE R.A < 10
```

Snowflake architecture



Cloud services: the brain



- **Optimizer**
  - Cascade-style
  - Scan set **pruning**

Pruning and re-clustering

How to avoid full table scan?

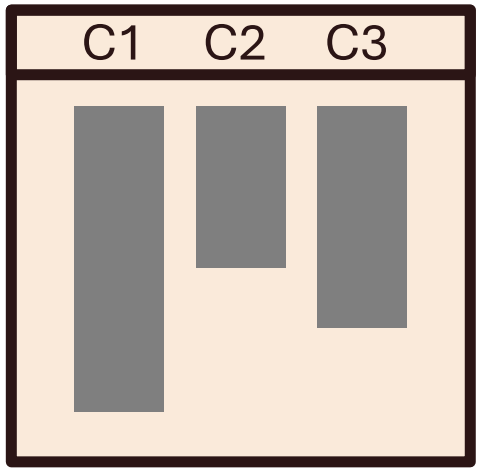
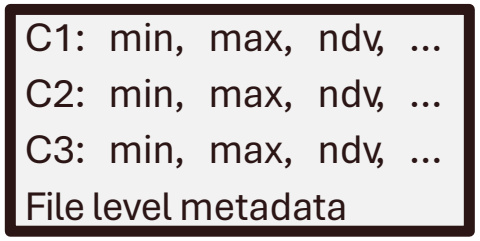


Table File

Pruning and re-clustering

How to avoid full table scan?

Zone Map



Cached in Metadata Service

Prune at compile and run time

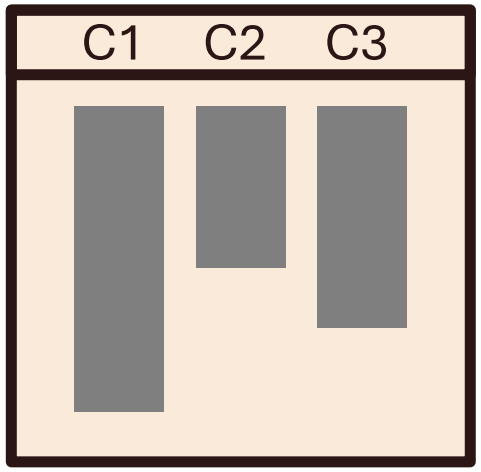


Table File

But it only works with data locality

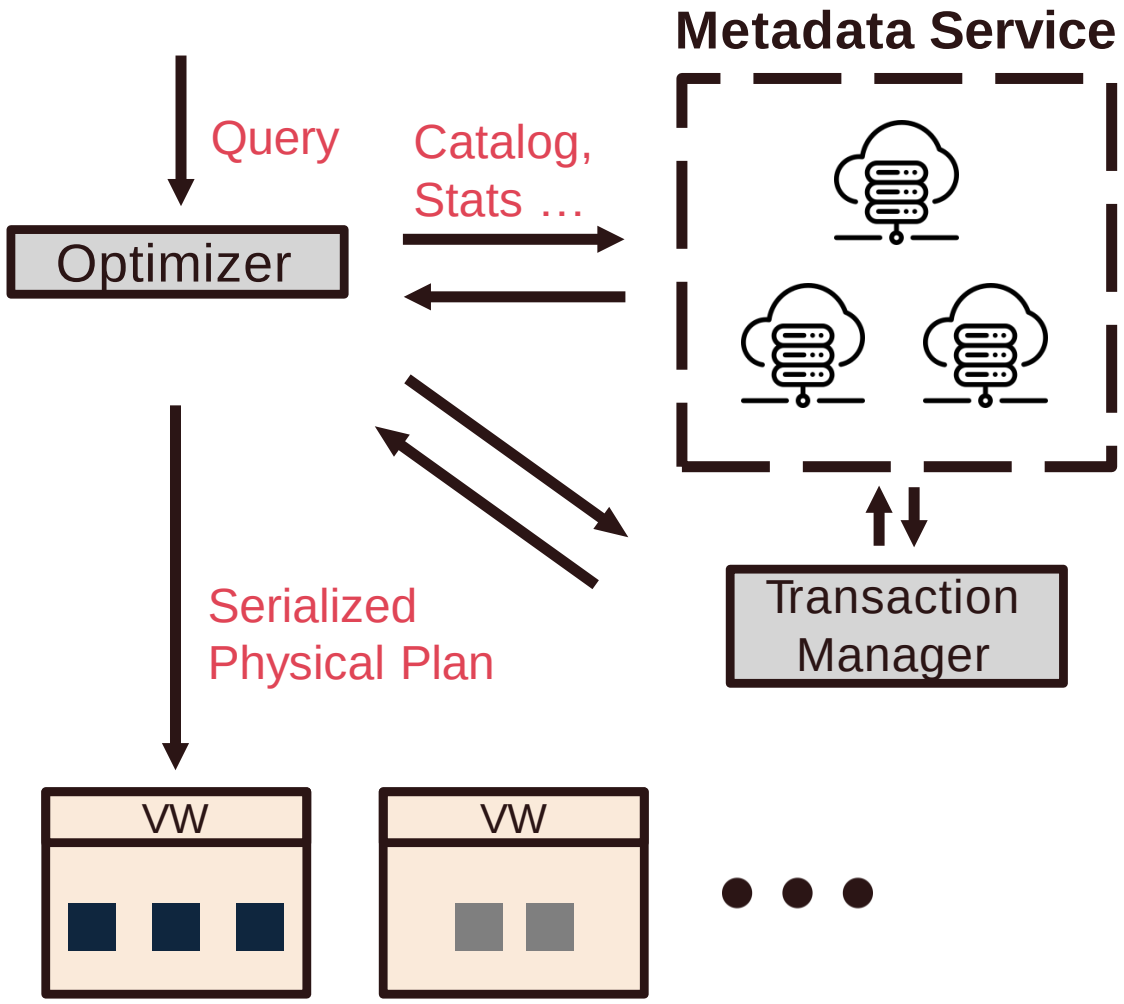
select count(*) from TBL where C1 > 'S'

Reclustering

Keep data "mostly" sorted

Automatic, incremental in the background

Cloud services: the brain



- **Optimizer**
  - Cascade-style
  - Scan set **pruning**
- **Metadata Service**
  - Stand-alone **FoundationDB** cluster for low latency accesses
  - Info needed for query compilation
    - Catalog, Stats
    - Lock status, version info
    - Zone maps
- **Multi-Version Concurrency Control (Snapshot Isolation)**

Snowflake architecture summary

- Disaggregated compute and storage
- Immutable hybrid columnar files in object storage
- Virtual warehouses provide elasticity and performance isolation
- Vectorized push-based execution engine
- Ephemeral storage system for caching intermediate results and persistent files
- Multi-tenant, always-on cloud services
- Separate fast metadata store
- Cascades-style optimizer, zone maps for scan pruning

Data lakes and warehouses: outline

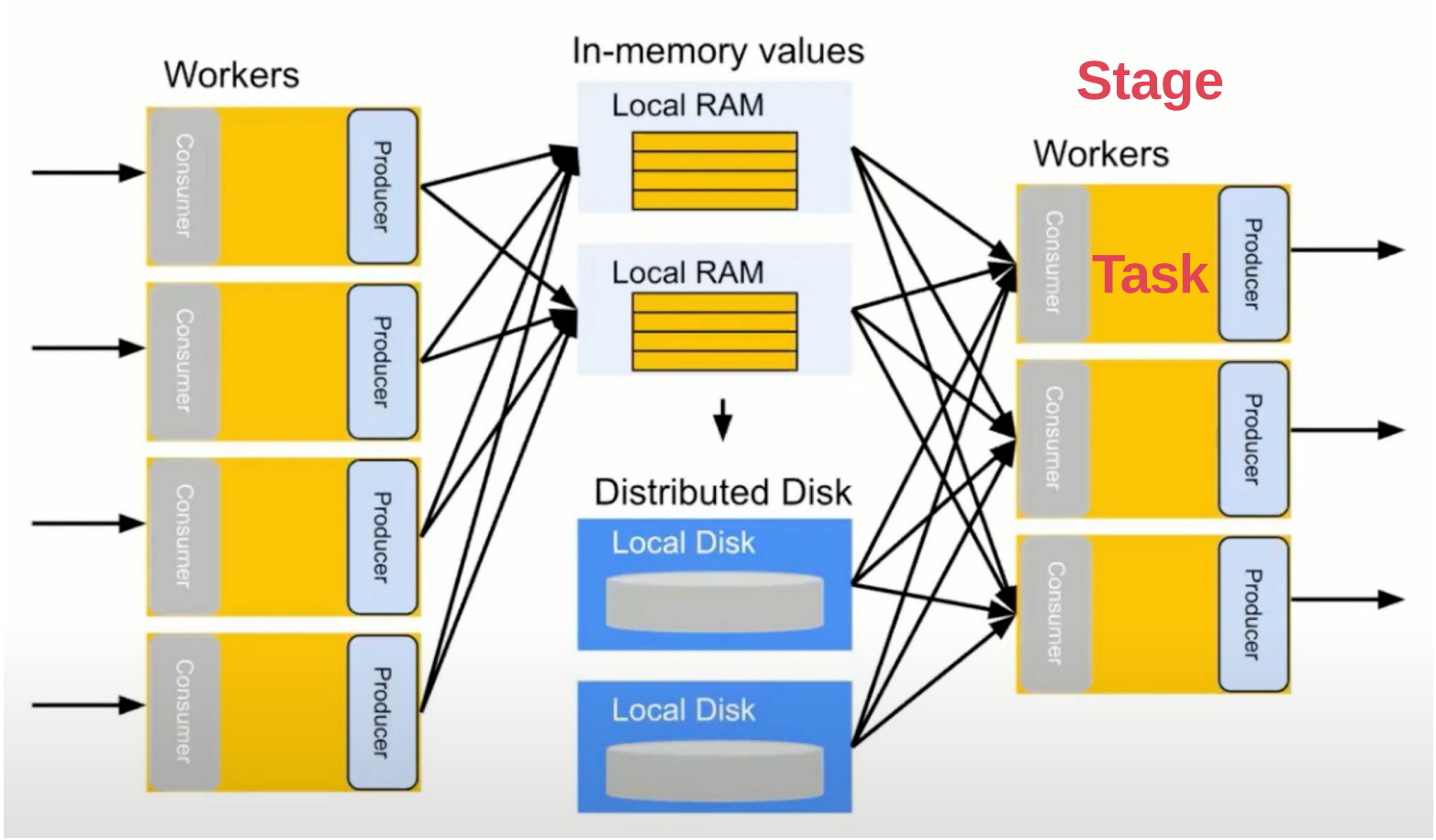
- Data lakes and warehouses
- Case studies
  - Snowflake
  - **Other offerings**

Google BigQuery

- Originated from the Google **Dremel** project
  - First database system with disaggregated compute and
  - storage In-situ data processing ⟶ **data lake**
  - Become commercial product BigQuery in 2012
- Serverless scalable analysis
  - On-demand pricing & capacity-based pricing
  - Columnar storage (Capacitor) similar to Parquet & ORC
  - Vectorized engine
  - **In-memory shuffle service**

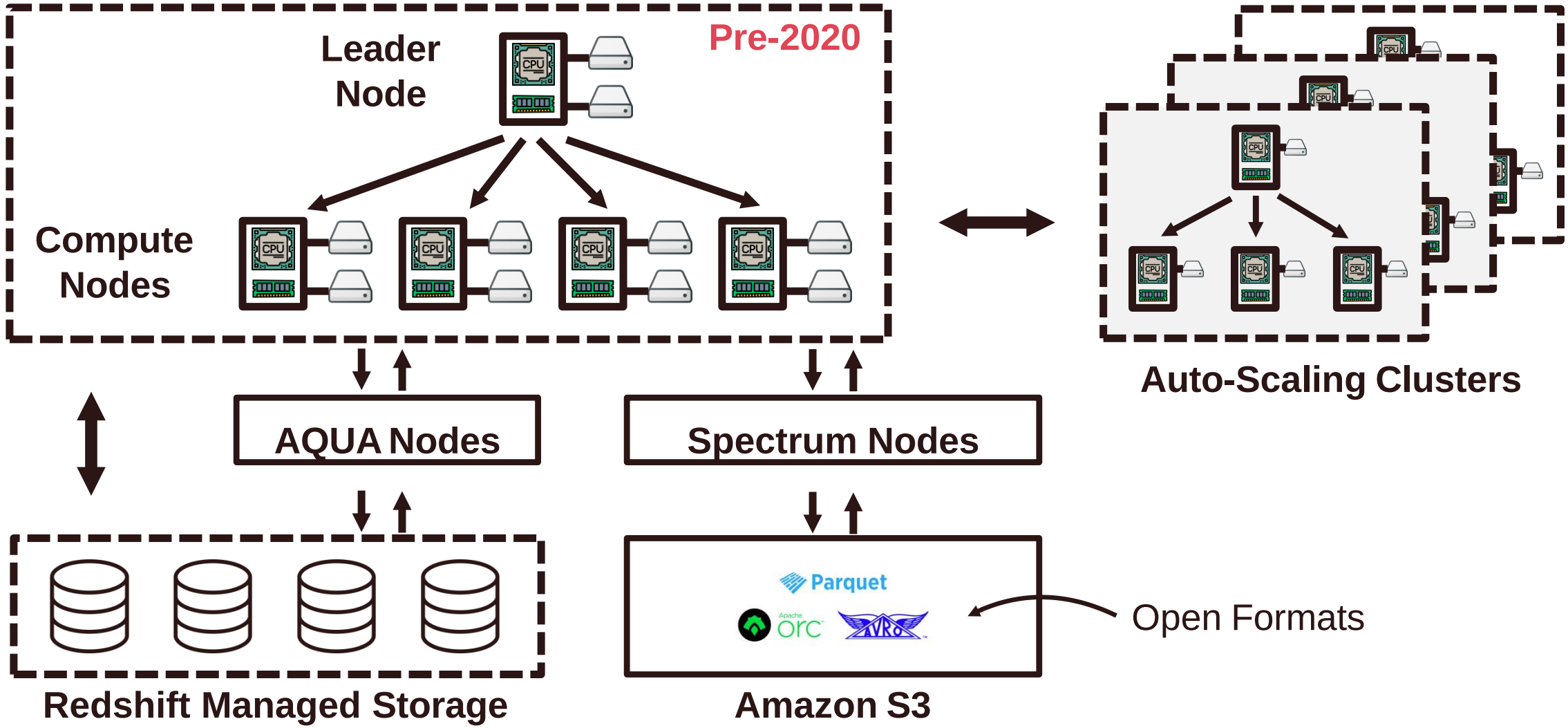
In-memory shuffle service

Google BigQuery



- Fault-Tolerance
- Straggler Avoidance
- Dynamic Resource Allocation
- Performance Overhead

Amazon Redshift



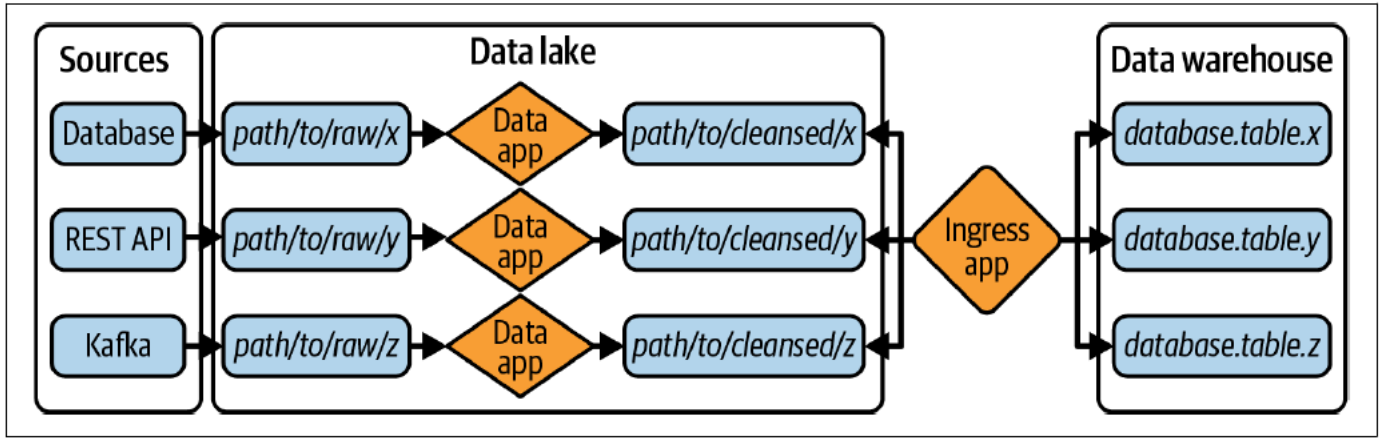
Amazon Redshift features

- Code-Gen (C++) plan fragments
- Compilation Service
  - Compiled-plan cache with 99.95% hit ratio
- Performance Optimizations
  - Min-max pruning
  - SIMD scan from local
  - SSDs AZ64 encoding

…

The dual-tier data architecture

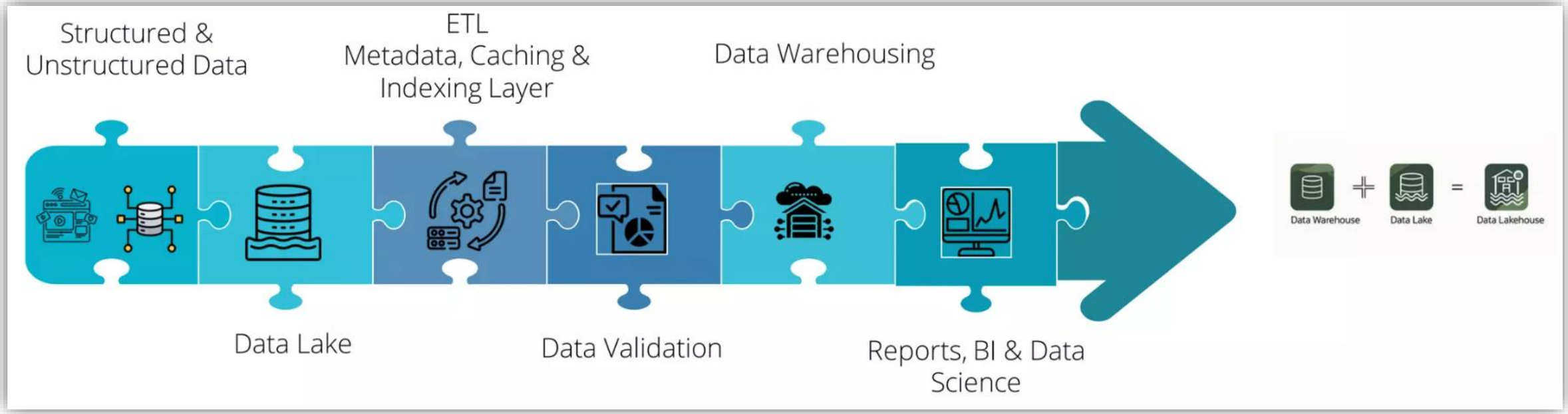
- Repository for storing large amounts of structured, semi-structured, and unstructured data without having to define a schema or ingest the data into proprietary internal formats.
    - Extract operational data from siloed data sources for writing into landing zones (/raw).
    - Read, clean, and transform the data from /raw and write the changes to /cleansed.
    - Read from /cleansed (could do additional joining and normalization) before writing out the warehouse.



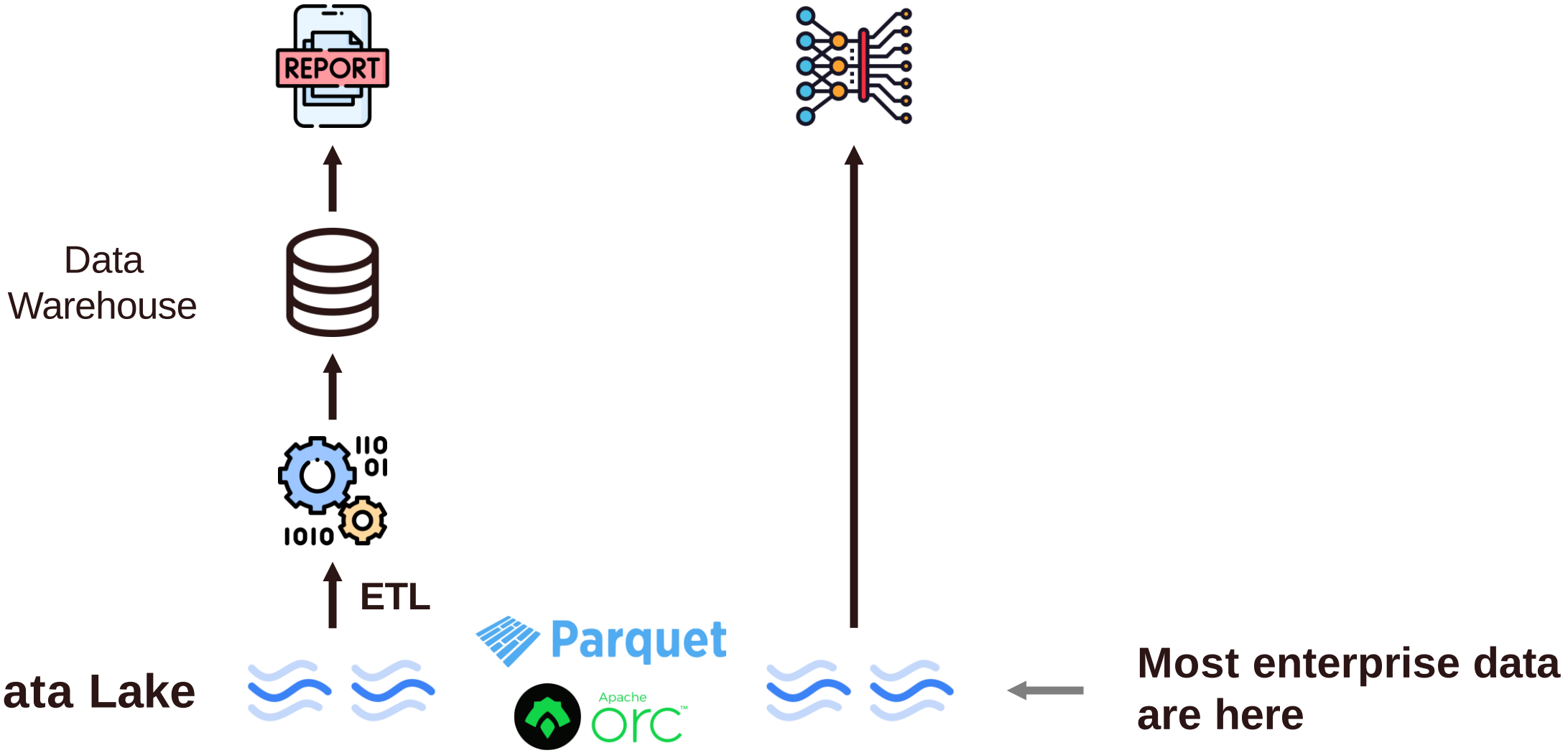
- Complex staging, redundant storage and less efficient

Lakehouse

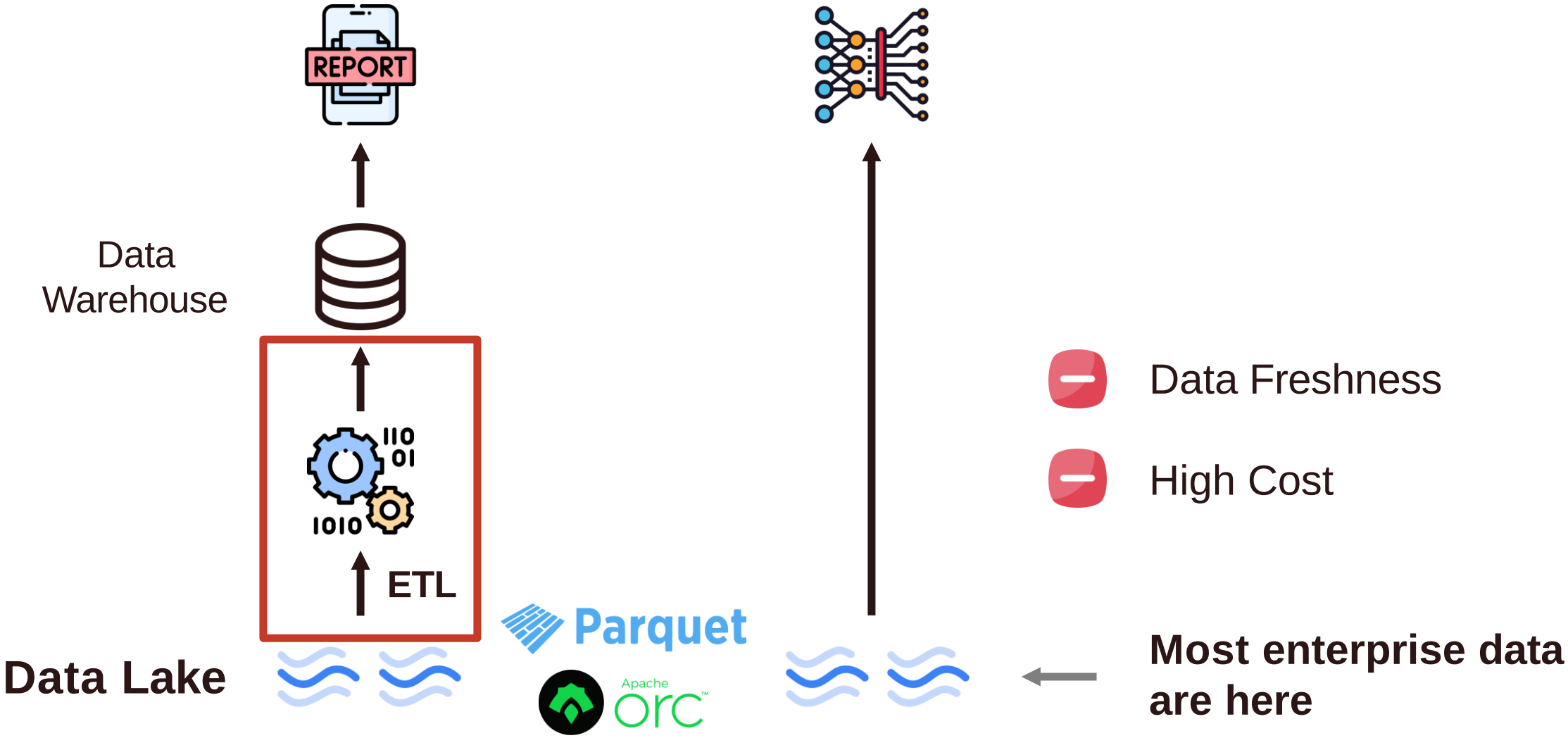
- A combination of data warehouse and data lake for better flexibility, low cost, and ACID transactions.
  - No need to copy data to data lake and warehouse separately.
  - Saves cost of infrastructure and staff.
  - Scalability and resilience.



Lakehouse

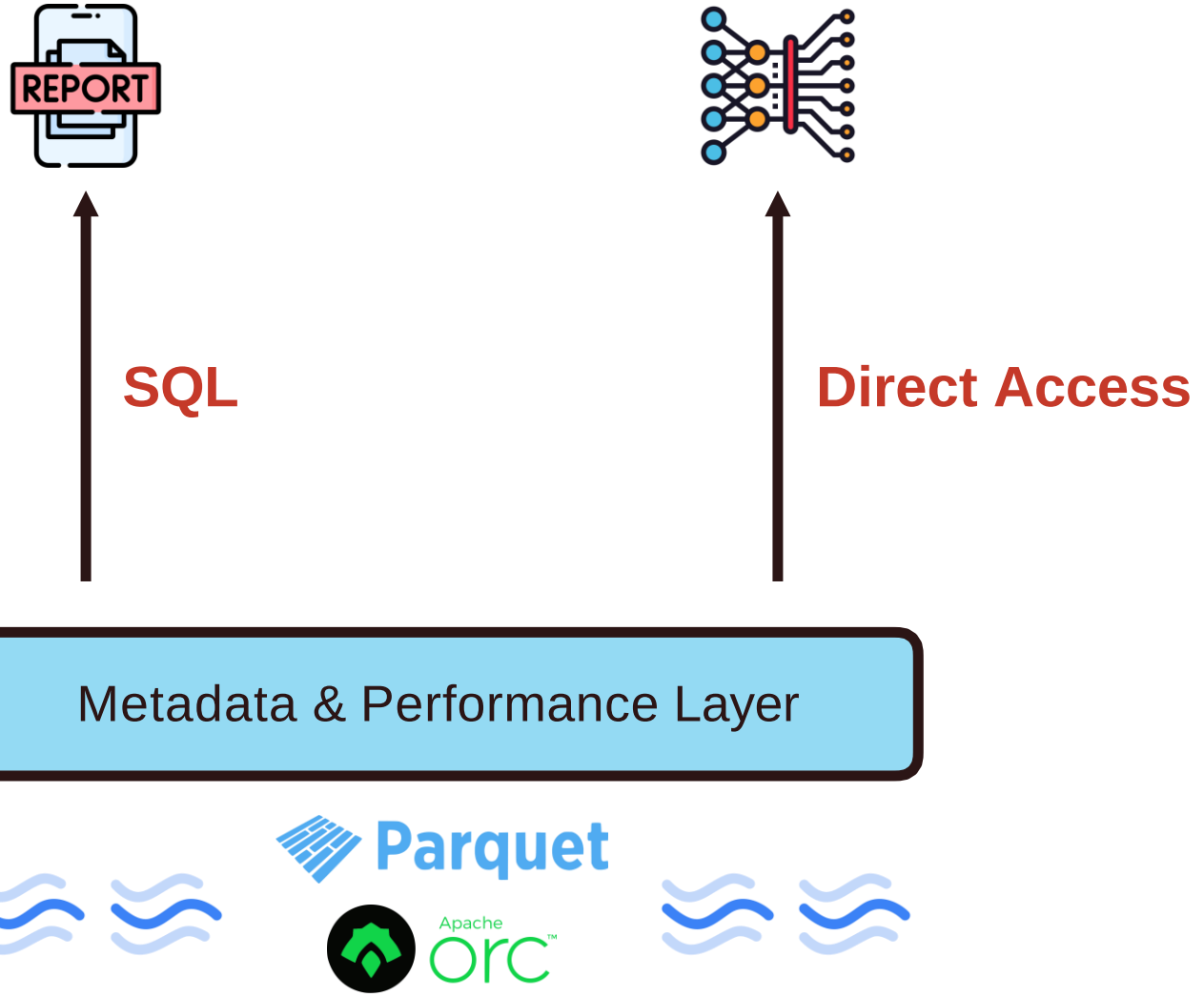

REPORT
Data Warehouse
ETL
Data Lake
Parquet
Apache orc
Most enterprise data are here

Lakehouse


REPORT
Data Warehouse
ETL
Data Lake
Parquet
Apache orc
Data Freshness
High Cost
Most enterprise data are here

Lakehouse

databricks

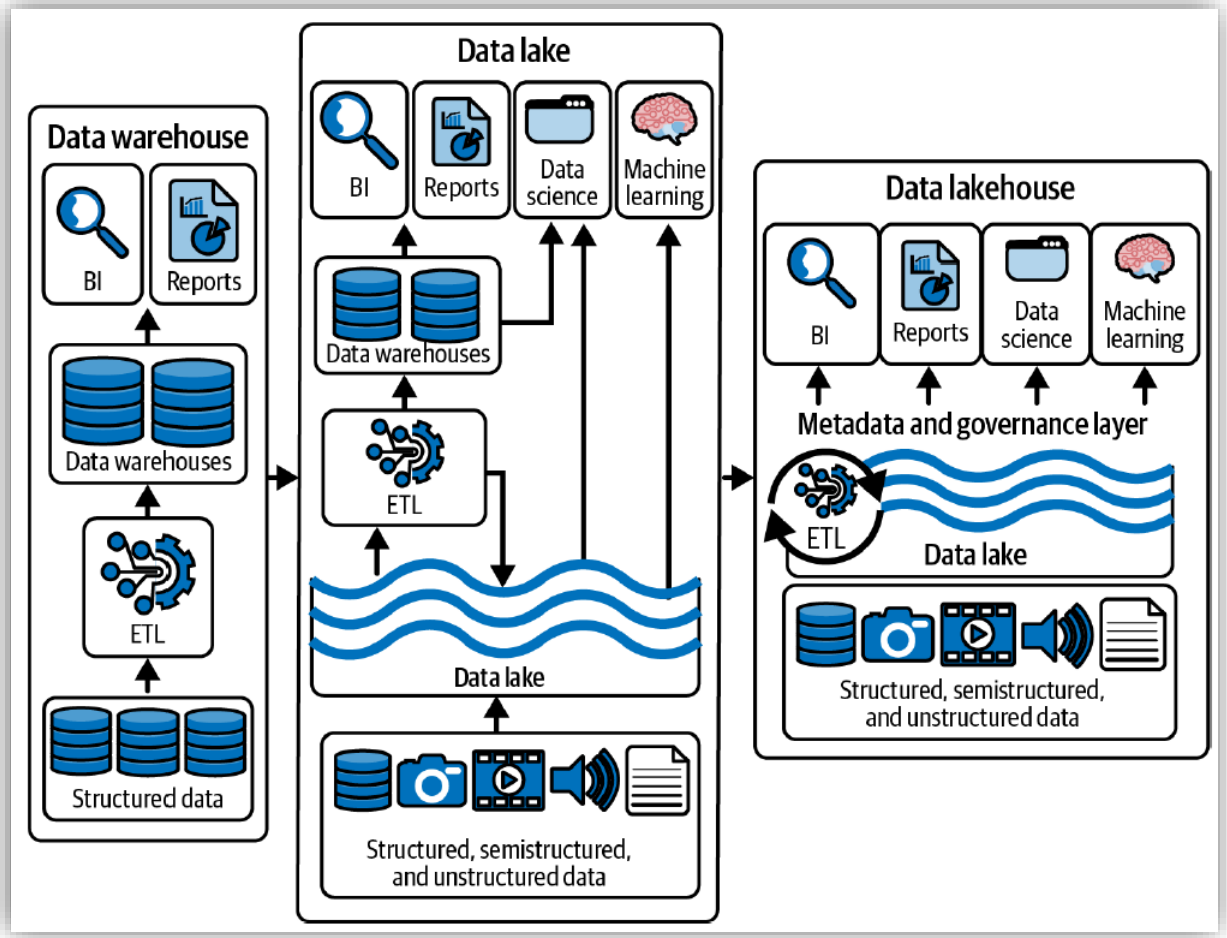


Lakehouse performance optimization

- ➜ Zone-maps, indexes, … stored as Delta tables
- ➜ Caching hot data in SSD or DRAM
- ➜ New vectorized engine: **Photon**
  - Pull-based vectorized query processing
  - Precompiled operator primitives
  - Use position list rather than bitmap for late materialization

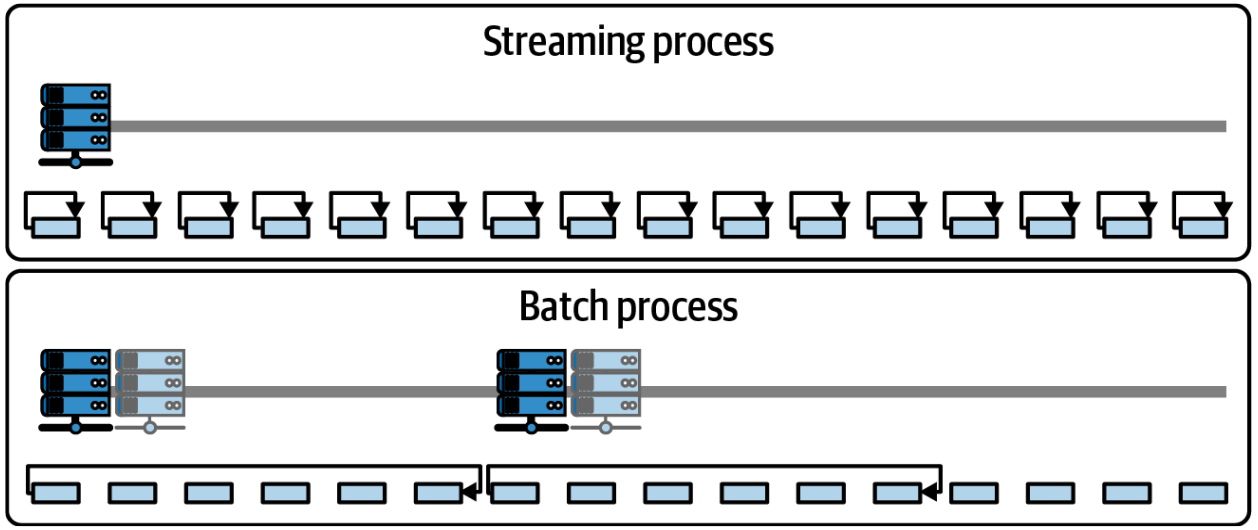
Delta Lake

- A combination of data warehouse and data lake for better flexibility, low cost, and ACID transactions.
  - No need to copy data to data lake and warehouse separately.
  - Saves cost of infrastructure and staff.
  - Scalability and resilience.



Streaming vs. batch processing

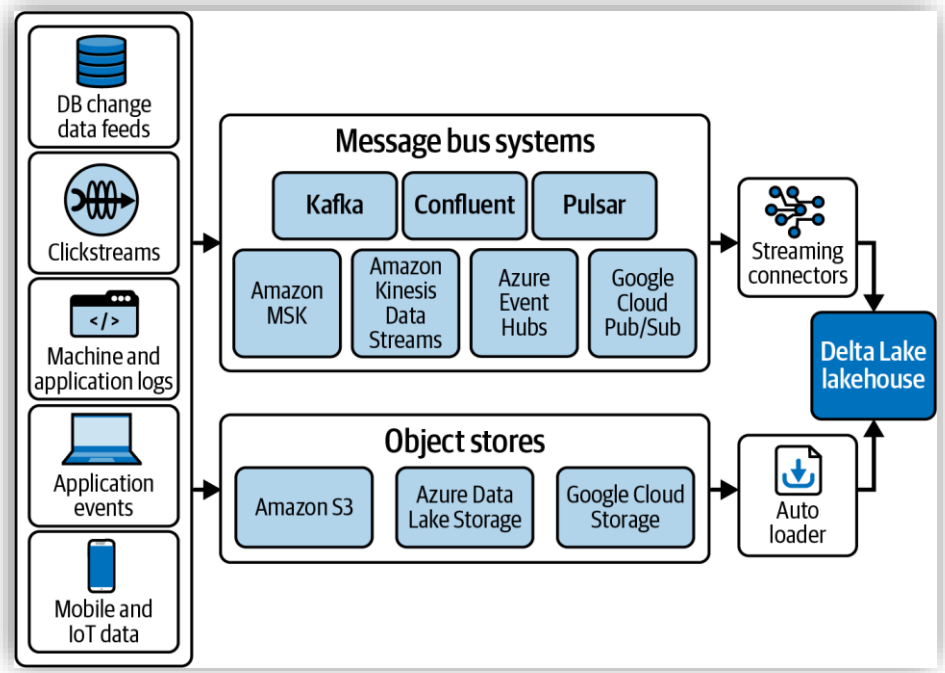
- **Streaming processing**: continuously processes data streaming, enabling instant insights and actions.
- **Batch processing**: deals with large volumes of data in chunks at scheduled intervals.



Streaming processing optimizes for **latency**, while batch processing optimizes for **throughput**.

Streaming vs. batch processing

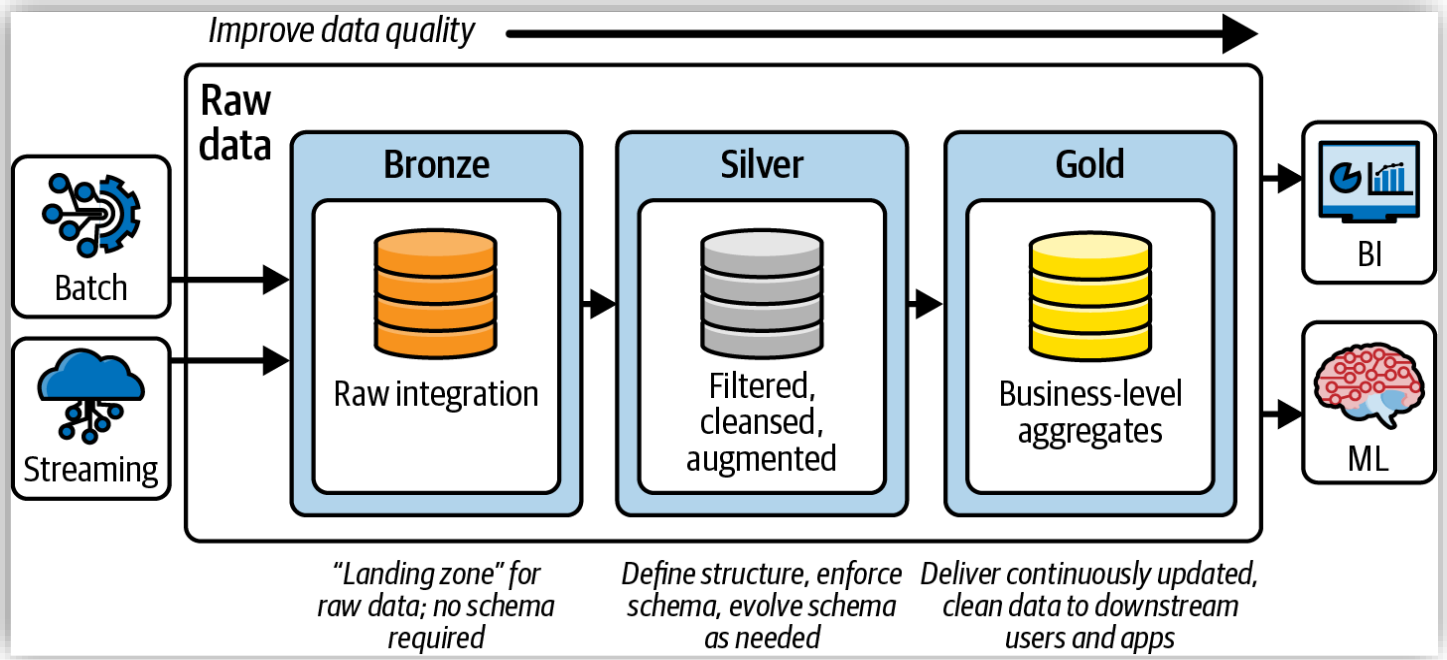
- **Streaming processing**: continuously processes data streaming, enabling instant insights and actions.
- **Batch processing**: deals with large volumes of data in chunks at scheduled intervals.



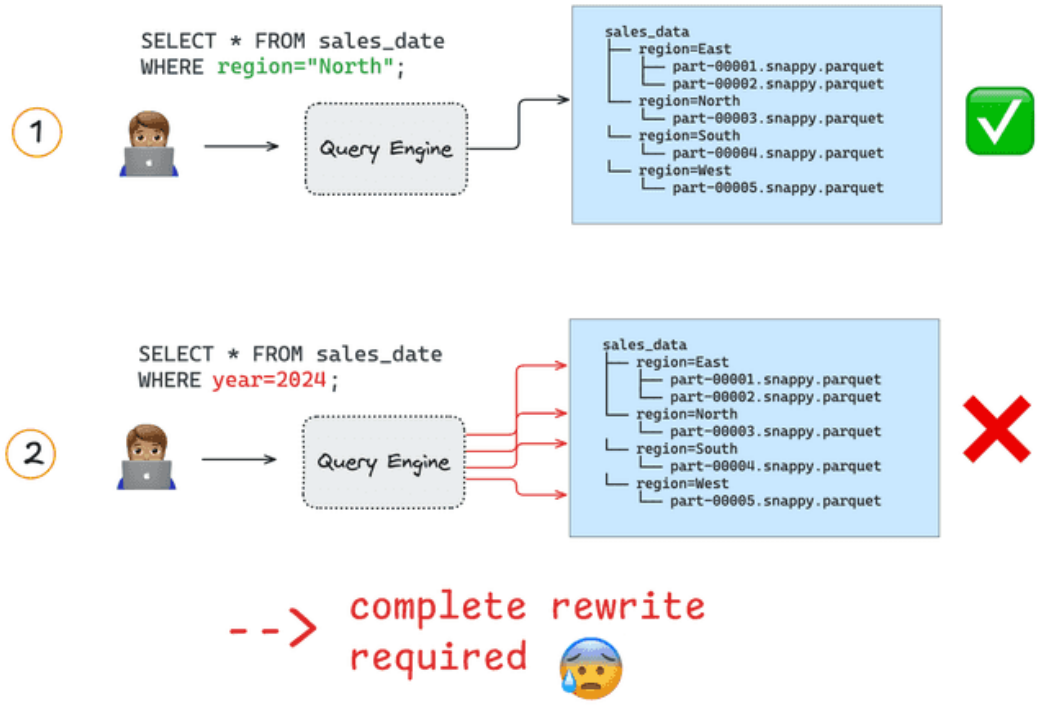
An example architecture diagram for stream processing applications with a Delta Lake sink from Databricks.

Medallion architecture

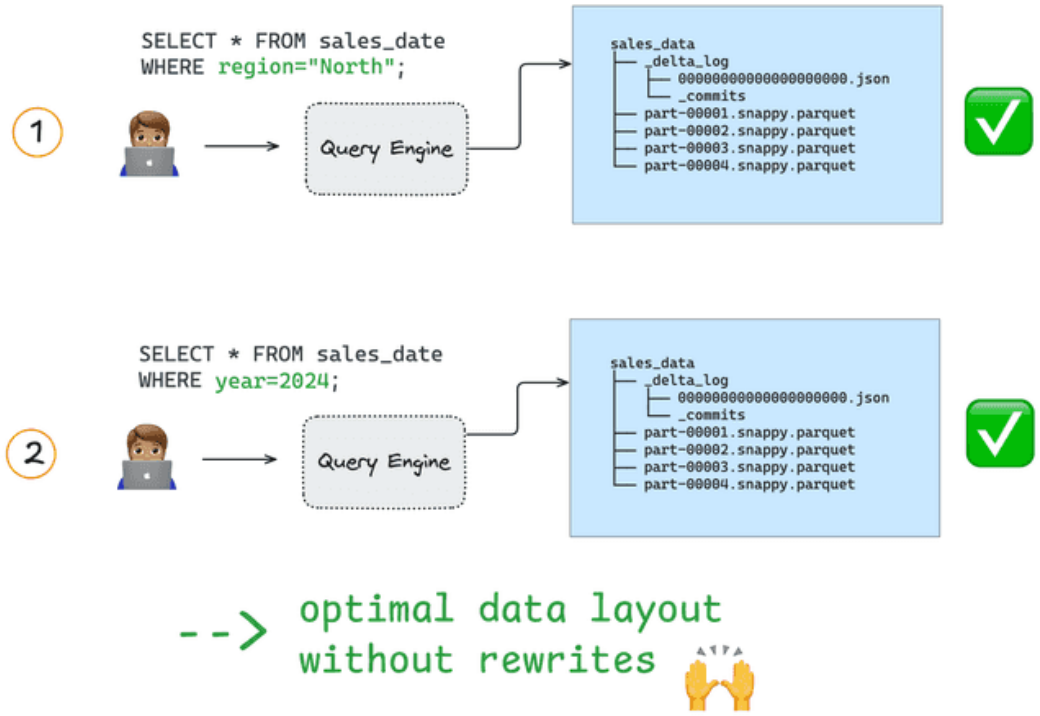
- A scheme to progressively refine datasets in the lakehouse.
  - Works for both batch or streaming sources.
  - Bronze: as simple as possible. E.g., Json parsing.
  - Silver: more complex preprocessing. E.g., text extraction from HTMLs.
  - Gold: complex joins and aggregates, w/ external data.



Liquid clustering



Hive-style partitioning on a partitioning column.



Liquid clustering automatically picks partitioning columns based on query patterns.

https://delta.io/blog/liquid-clustering/

Data lakes and warehouses: outline

- Data lakes and warehouses
- Case studies
  - Snowflake
  - Other offerings

Credits and references

- Denny Lee et al. Delta Lake: The Definitive Guide.
- Andy Pavlo, CMU
- Dixin Tang, UT Austin
- Huanchen Zhang, Tsinghua