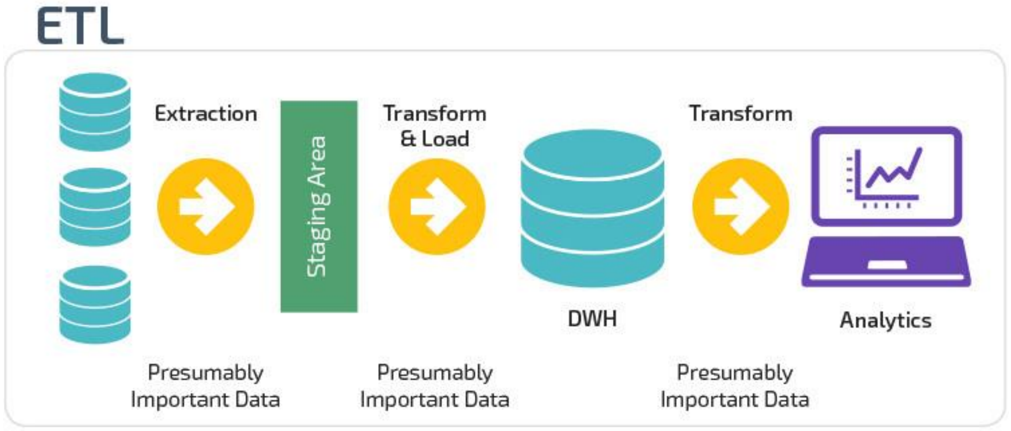
Cloud-Based Data Processing

OLAP in the cloud

Jana Giceva

Traditional OLAP / data warehouses

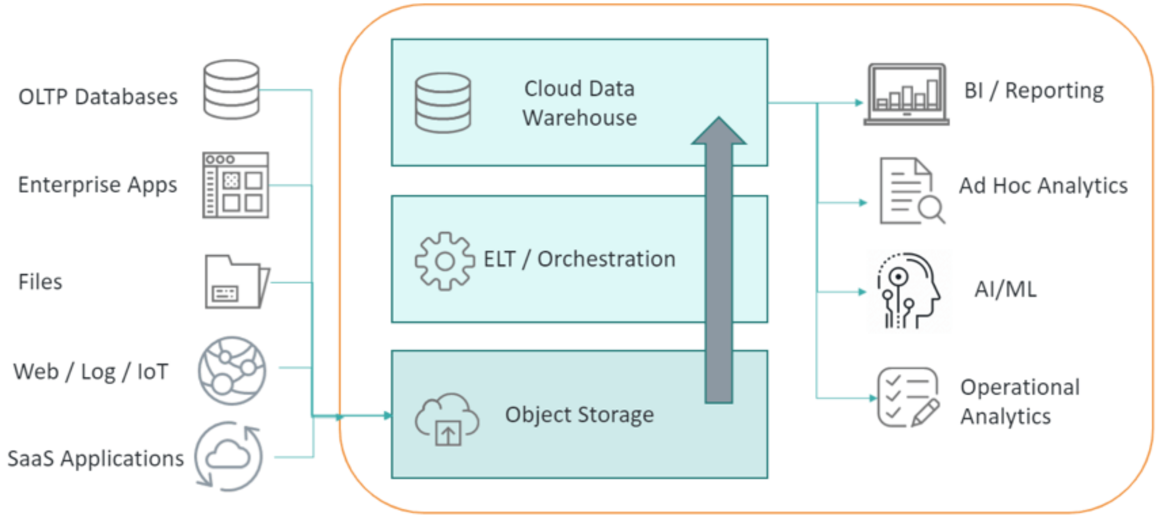
- Traditional data warehousing systems are built for:
  - Predictable, slow-evolving internal data
  - *Relational* data, structured in a *star*- or *snowflake* schema
  - Complex *ETL* (extract-transform-load) pipelines and physical tuning (compression, layout, etc.)
  - Limited number of users and use-cases


ETL
Extraction
Staging Area
Transform & Load
Transform
DWH
Analytics
Presumably Important Data
Presumably Important Data
Presumably Important Data

img src: https://panoply.io/data-warehouse-guide

Cloud-based data

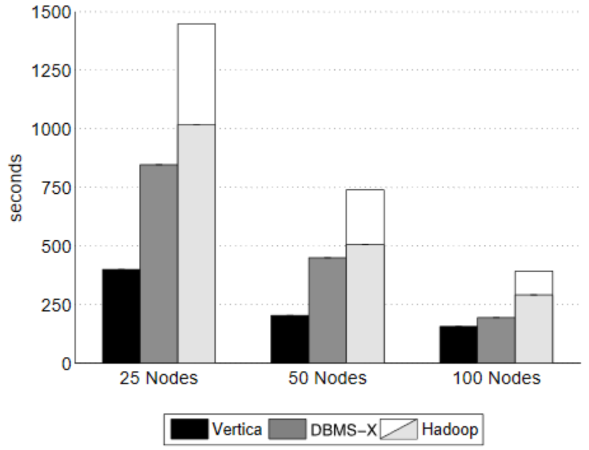
- Data in the cloud:
  - Dynamic, external sources: web, logs, mobile devices, sensor data, etc.
  - ELT instead of ETL (extract-load-transform) – data transformation is done inside the system
  - Often in semi-structured data format (e.g., JSON, XML, Avro)
  - Access required by many users, with very different use-cases



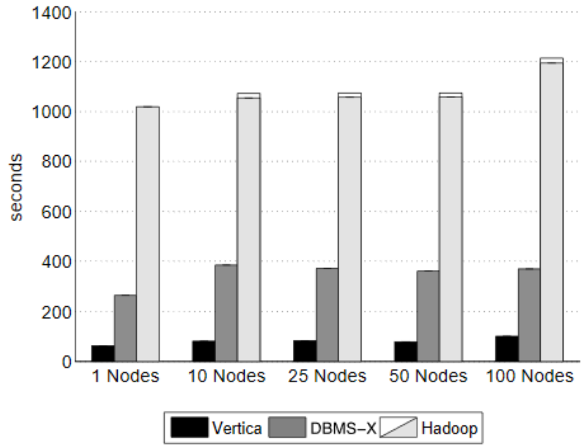
img src: InterWorks

Are DW still relevant in the era of BigData?

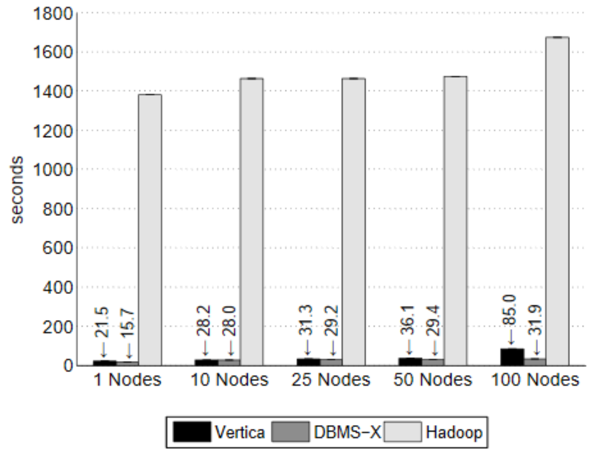
```
SELECT * FROM Data WHERE
field LIKE '%XYZ%';
```



```
SELECT pageURL, pageRank
FROM Rankings WHERE
pageRank > X;
```



```
JOIN
```



- Schemas are a good idea (parsing text is slow)
- Auxiliary data structures help boost performance (value indexes, join indexes)
- Optimized algorithms and storage structures: layout, data formatting, order of execution, choice of algorithm

Source: Pavlo et al. (2009). A Comparison of Approaches to Large-Scale Data Analysis. SIGMOD

Data warehouse system architecture

- Architecture: **shared-nothing**

- Important **architectural dimensions** and methods
  - **Storage**:
    - Columnar storage, compression, data pruning
    - Table partitioning, distribution

  - **Query engine**:
    - Vectorized or JIT code-gen
    - Query optimization
    - Fine- and coarse-grained parallelism

  - **Cluster:**
    - (Meta-) data sharing
    - Resource allocation and management

Data warehouse storage

- **Data layout:** column-store vs. row-store
  - Column-stores only read relevant data (skip irrelevant data by skipping unrelated columns)
  - Suitable for analytical workloads (better use of CPU caches, SIMD registers, lightweight compression).
  - E.g., Parquet, ODC, etc.

- **Storage format**: compression is key!
  - Trades I/O for CPU and good fit for large datasets (storage) and I/O intensive workloads
  - Excellent synergy with column-stores
  - E.g., RLE, gzip, LZ4, etc.

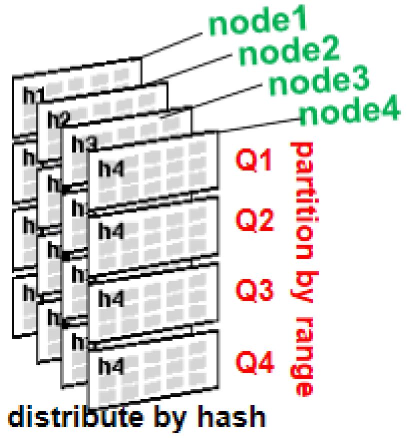
- **Pruning:** skip irrelevant data using a MinMax index.
  - Data is usually ordered → can maintain a sparse MinMax index
  - Allows to skip irrelevant data horizontally (rows).

Table partitioning and distribution

- **Data is spread based on a key**
  - Functions: hash, range, list

- **Distribution** (system-driven)
  - Goal: parallelism
    - Give each compute node a piece of the data
    - Each query has work on every piece (keep everyone busy)

- **Partitioning** (user-specified)
  - Goal: data lifecycle management
    - Data warehouse e.g., keeps last six months
    - Every night: load one new day, drop the oldest partition
  - Goal: improve access pattern
    - When querying for May, drop P1,P3,P4 (partition pruning).



img src: P.Boncz (CWI)

Query Execution

Scalability is not as important unless you can make the most out of the underlying hardware.

- **Vectorized execution**
  - Data is not materialized (as in Volcano or MapReduce), but pipelined (push-based) in batches
  - Batch-size (few thousand rows) to save I/O, and greatly improve cache efficiency.
  - E.g., Actian Vortex, Hive, Drill, Snowflake, etc.

- And/or **JIT code generation**
  - Generate a separate program that executes (only) the exact query plan
  - Compile the program (JiT compilation) and run on your data.
  - E.g., Spark, Tableau/HyPer, MemSQL, etc.

Cloud-native warehouses

- Design considerations for scalability, elasticity, fault-tolerance, good performance.
  - **Should we keep compute and storage tightly coupled?**

- **Separate tiers**, **different challenges** at **cloud-scale** for cloud-data:
  - **Storage**:
    - Abstracting from the storage format
    - Distribution and partitioning of data even more relevant at cloud-scale
    - Data caching models across the (deep) storage hierarchy (cost/performance)
  - **Query execution**:
    - Distributed query execution: combine scale-out and scale-up
    - Global resource-aware scheduling
    - Distributed query optimization

- **Service form factor**
  - *Reserved-capacity* services vs. *serverless* instances

Data placement in the Cloud

- There is **no data locality**
  - To create elasticity, compute needs to be de-coupled from storage
  - i.e., AWS S3 files are always stored remotely
    - high latency (100-200ms) and slow bandwidth (20-125MB/s)

- **Distribution and partitioning is very common**
  - Distribution – allows jobs to be parallelized
  - Partitioning – partition-pruning, data lifecycle management

- Some **locality** can be created **by caching**
  - Caching in memory (e.g., Spark)
  - Caching on local ephemeral disk (e.g., DBIO cache in Databricks, Vertica EON, etc.)
    - 0.03ms latency, ~500MB/S bandwidth, ~500GB size (per core)

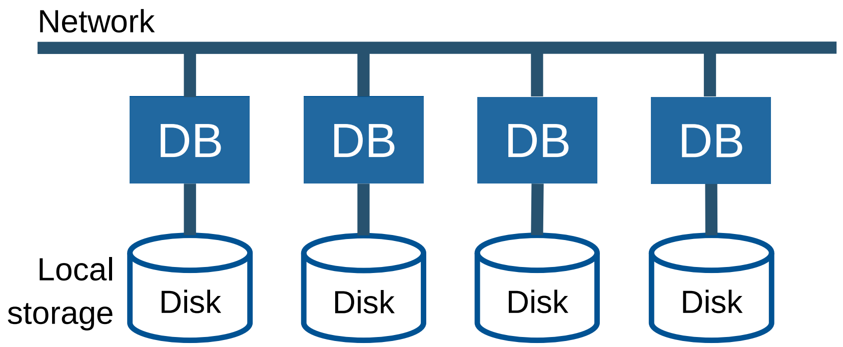
Shared-nothing cloud data warehouse

Shared-nothing architecture

- Shared nothing data warehouse
  - dominant system architecture for high-performance data warehousing.

- Scales well for star-schema queries as very little bandwidth is required to join
  - a small (broadcast) dimensions table with
  - a large (partitioned) fact table.

- Elegant design with homogeneous nodes



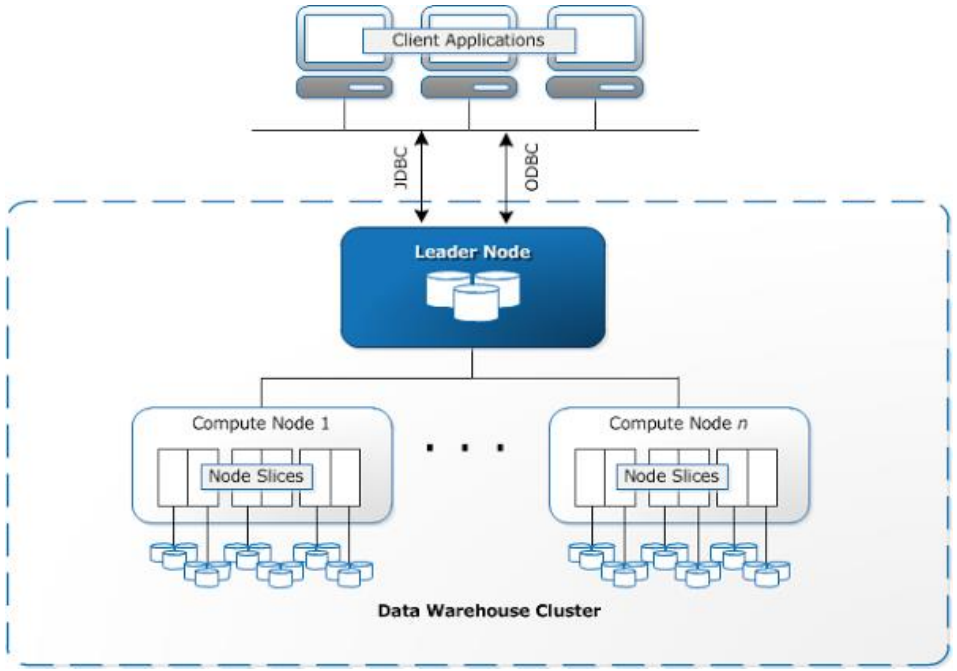
Every query processor node (DB) has its own local attached storage (disk).

Data is horizontally partitioned across the processor nodes.

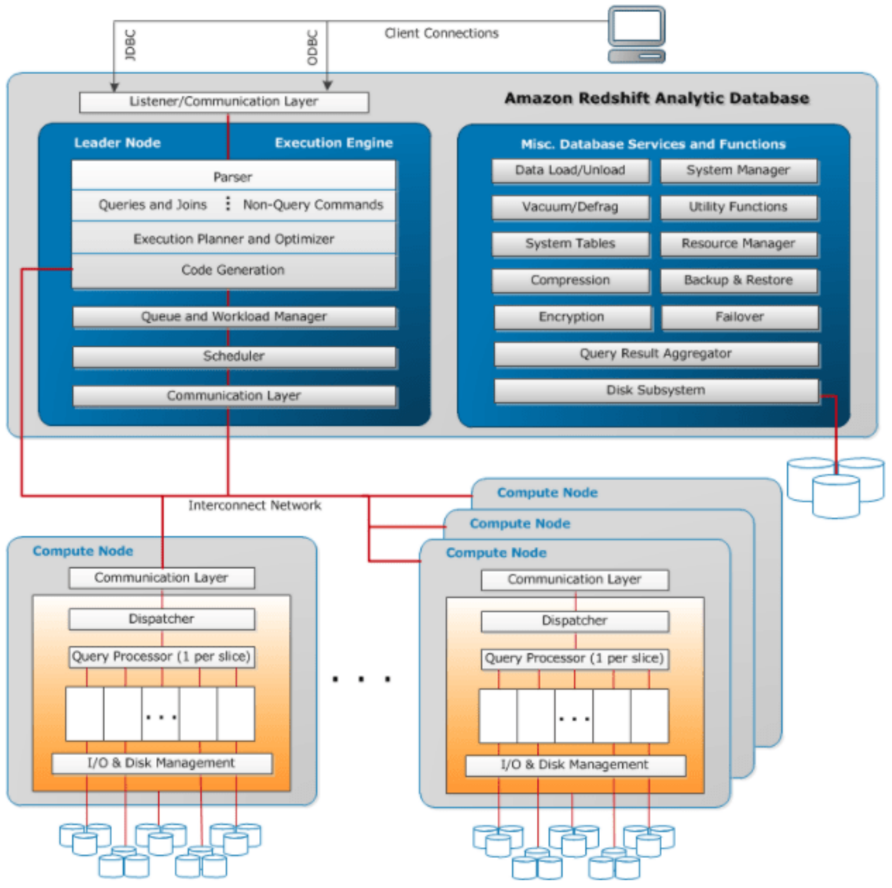
Each node is only responsible for the rows on its own local disks

Example: Amazon (AWS) Redshift

- Classic **shared-nothing** design with locally attached storage
- The execution engine is ParAccel DBMS
  - **Classic MPP, JIT C++**
- Leverages **standard AWS services**:
  - EC2 + EBS + S3, Virtual Private Cloud
- **Redshift cluster: Leader + Compute nodes**
  - Leader parses a query and builds an optimal execution plan.
  - Creates compiled code and distributes it to the compute nodes for processing.
  - Aggregates the results before returning the result to the client.



Redshift detailed architecture



Credits: docs.aws.amazon.com

The leader node:

- SQL end point. Connects to SQL client / BI tools by JDBC/ODBC
- Stores metadata
- Query compilation
- Query optimization
- Coordinate parallel SQL processing

The compute node:

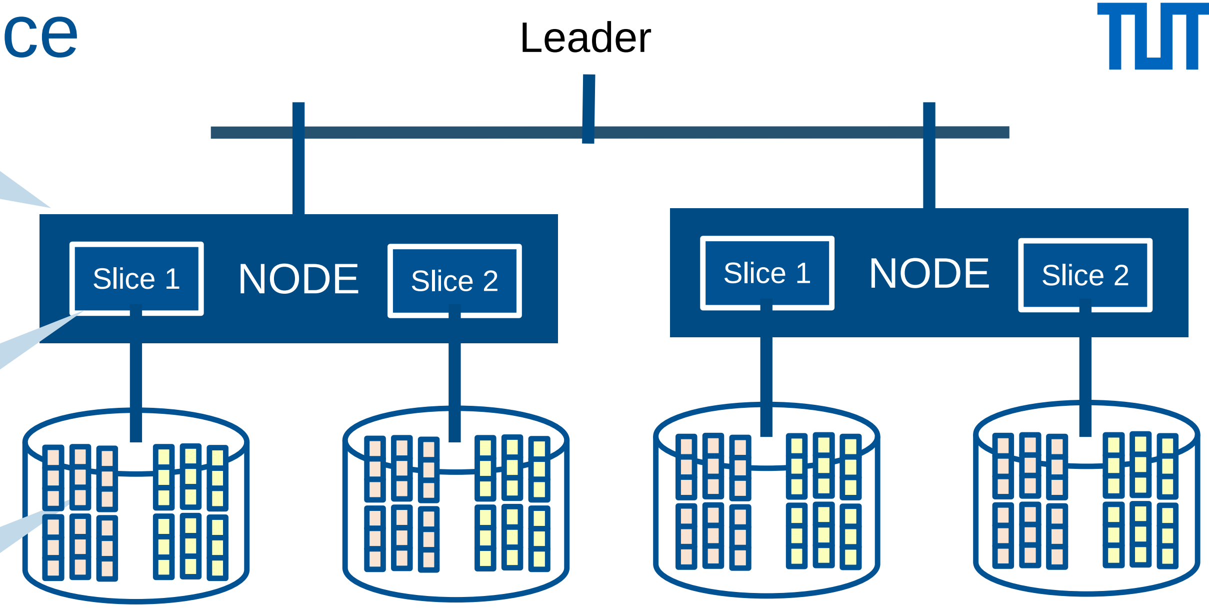
- Local, columnar storage
- Executes queries in parallel, by slices
- Handles load, back-up, restore
- Pricing: based on the compute node instance and the number of nodes used.

A Redshift Instance

Each compute **node** has dedicated CPU, memory and locally attached disk storage.

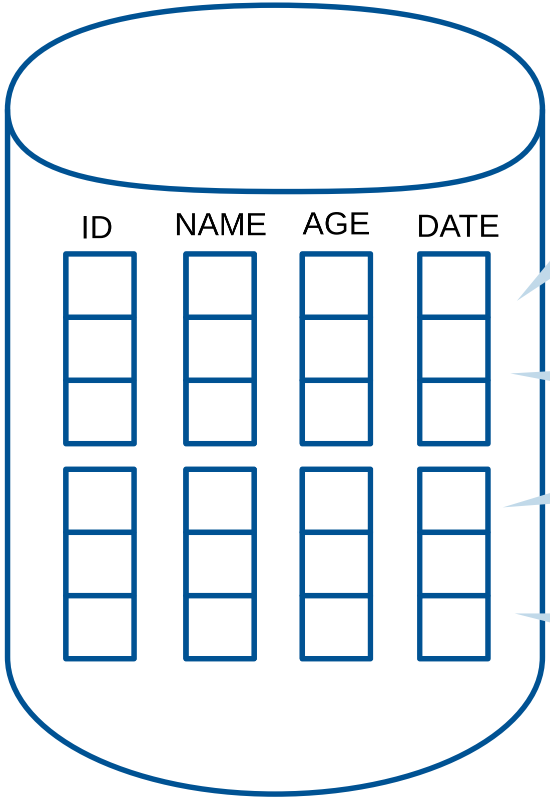
Memory, storage, and data partitioned among the **slices**.

Hash and round-robin table partitioning / distribution.



- The leader distributes data to the slices and apportions workload to them.
- The number of slices per node depends on the node size.
- Within a node, Redshift can decide how to distribute data between the slices (or the user can specify the **distribution key**, to better match the query's joins and aggregations).

Within a slice



Data stored in columns, sorted by:

- Compound sort key
- *Interleaved* sort key (multidimensional sorting)

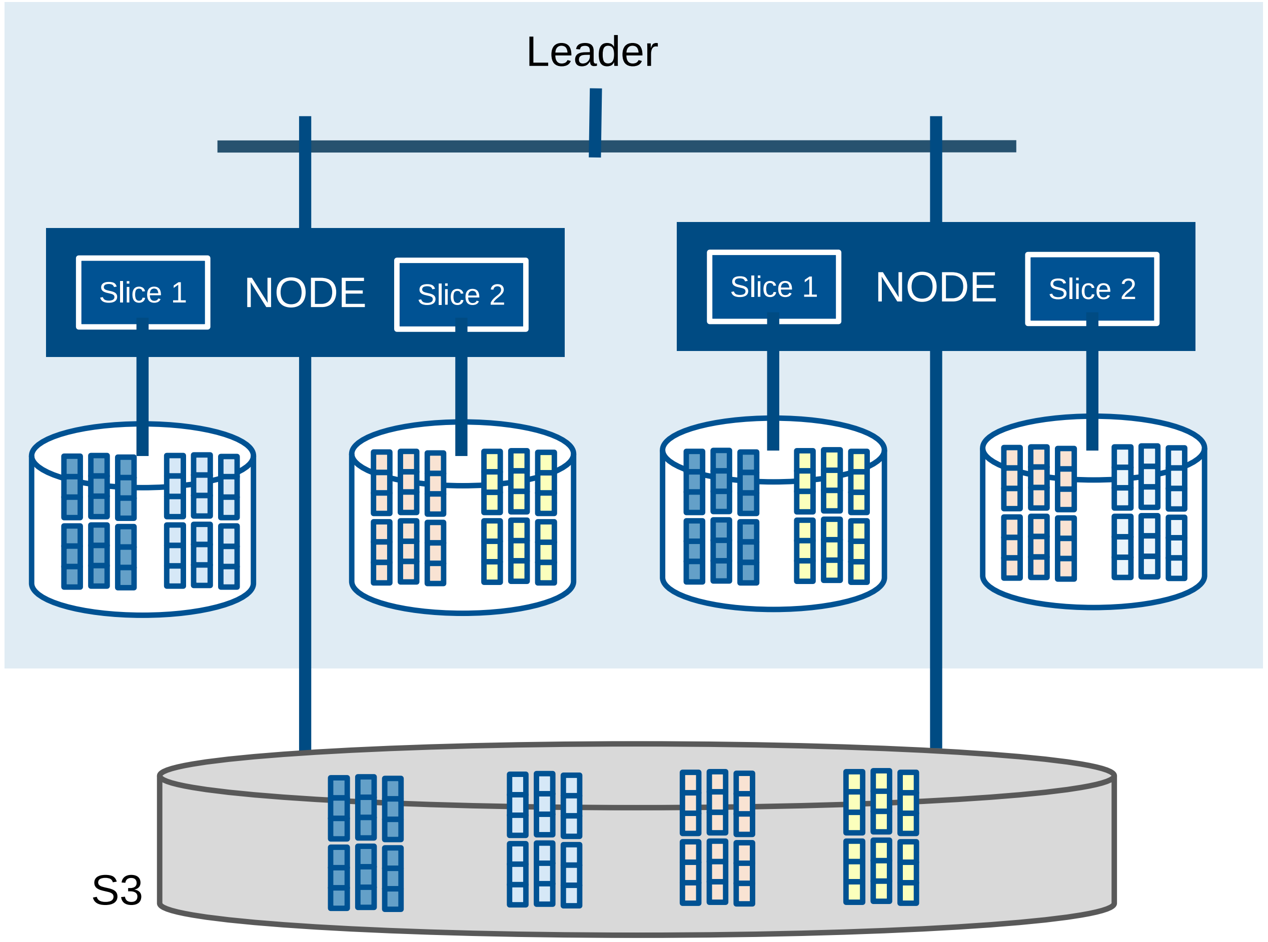
Columns stored in 1MB blocks.

Min and Max value of each block retained in a *zone map.*

Rich collection of compression options (RLE, dictionary, gzip, etc.)

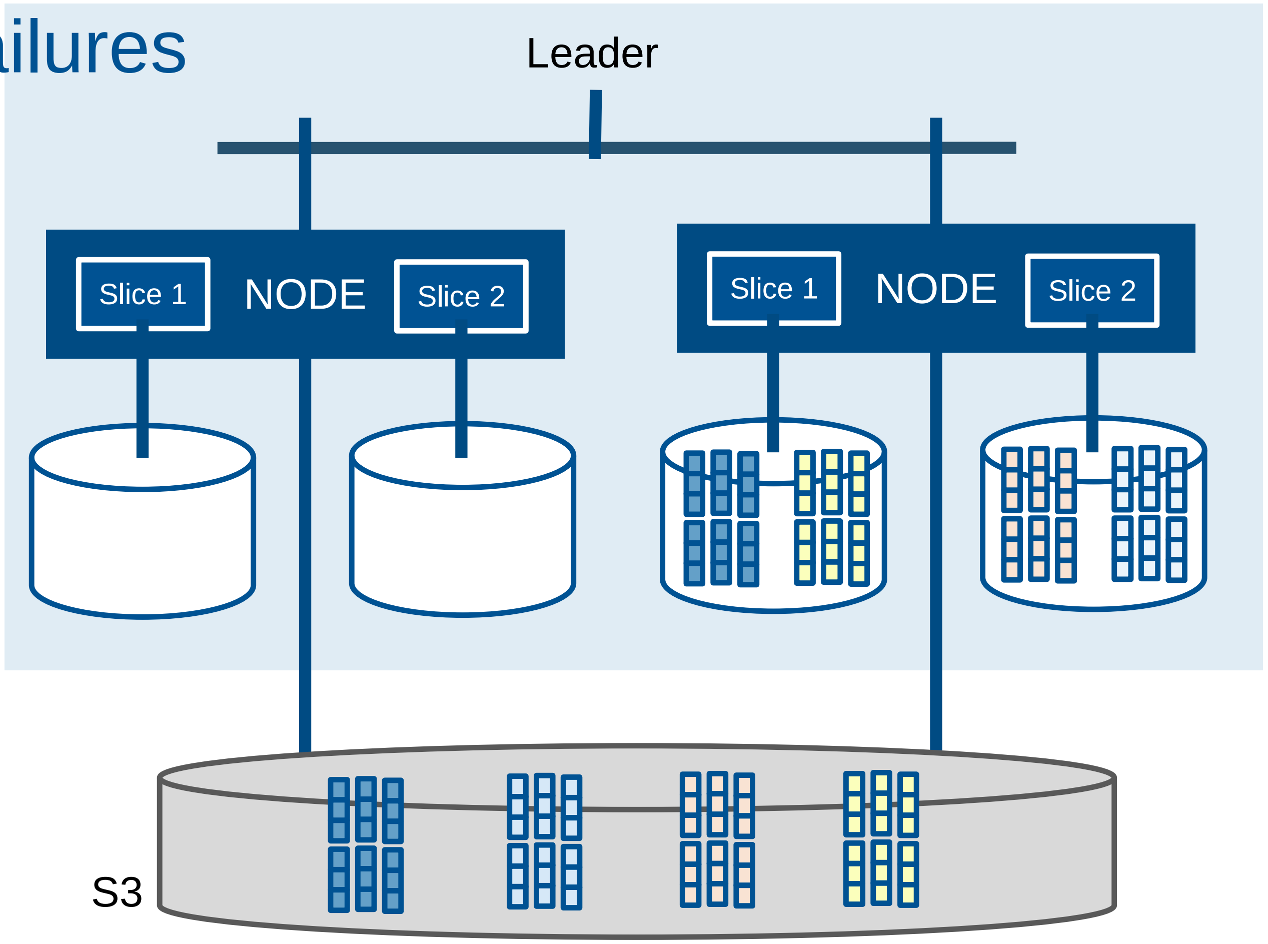
Fault tolerance

- Each 1MB block is replicated on a different compute node
- Data blocks (1MB) are also stored on S3
- S3 triply replicates each block



Handling node failures

- Assume node 1 fails:
  - Option #1:
    node 2 processes load until node 1 is restored
  - Option #2:
    new node is instantiated
    - node 3 processes workload using data in S3
    - until the local disks are restored



Redshift summary

- Highly successful cloud SaaS DW service
- Classic shared-nothing design
- Leverages S3 to handle node and disk failures
- Key strength:
  - performance through use of local storage
- Key weakness:
  - compute cannot be scaled independent of storage (and vice-versa)

Drawbacks of shared-nothing architecture

- **Tightly couples compute and storage resources**
- **Heterogeneous workloads**
  - a system configuration that is ideal for bulk loading (high I/O bandwidth, light compute)
  - is poor fit for complex queries (low I/O bandwidth, heavy compute).
- **Membership changes**
  - if the set of nodes changes potentially a large volume of data needs to be reshuffled
- **Online upgrades**
  - possible but very hard when everything is coupled and expected to be homogeneous.
- This makes it **problematic** to use it **in the cloud setting**

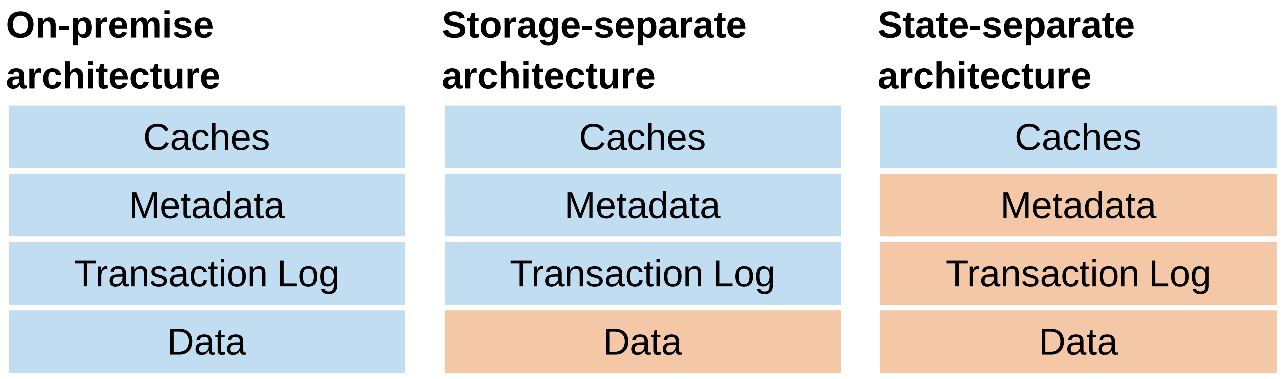
Shared-storage architectures

Separating compute and storage

Separating Compute and Storage

- Evolution of cloud-data warehouse architectures over the years

- Engines maintain state comprised of: cache, metadata, transaction log, and data



- The first step is **decoupling of storage and compute** – more flexible scaling
  - Both layers can scale-up or down independently
  - Storage is abundant and cheaper than compute
  - User only pays for the compute needed to query a working subset of the data

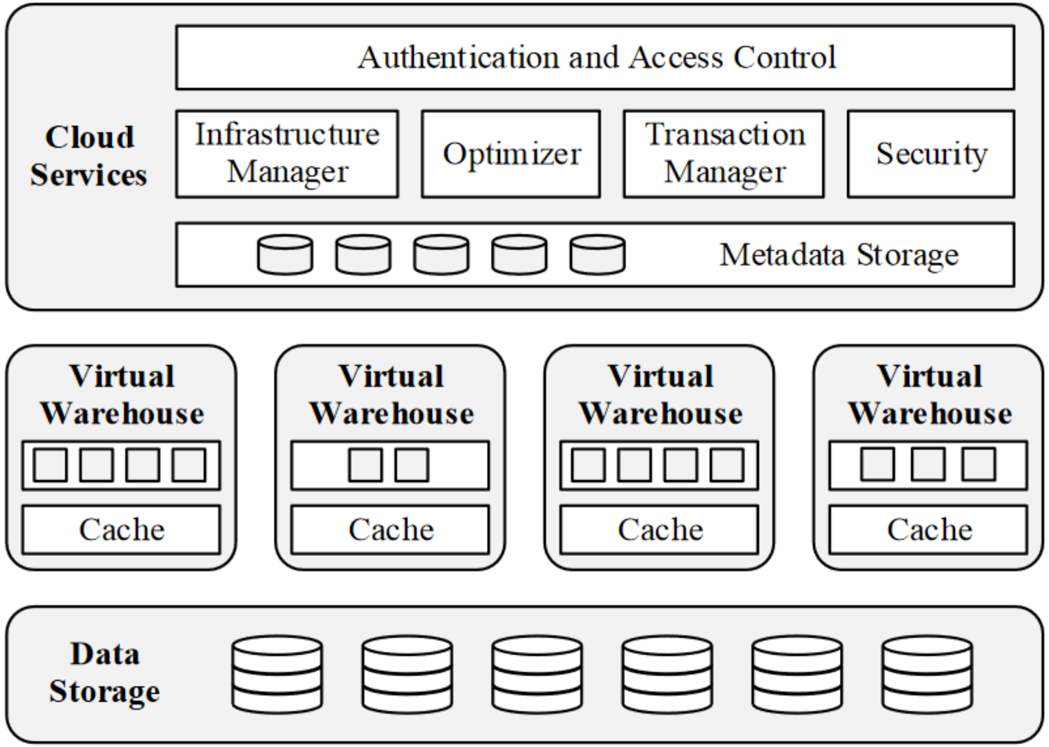
Example: Snowflake

- Snowflake separates storage and compute.
  - two loosely coupled, independently scalable services.

- Compute
  - handled by a proprietary shared-nothing execution engine.
  - highly elastic.

- Storage
  - handled by Amazon S3, Azure Blob storage, or Google Cloud storage.
  - dynamically cached on local storage clusters used to execute the queries

Snowflake Architecture



The Brain: Key data management services.

The Muscle: Shared-nothing execution engine (virtual warehouse)

The Storage: Shared-storage for data and query results.

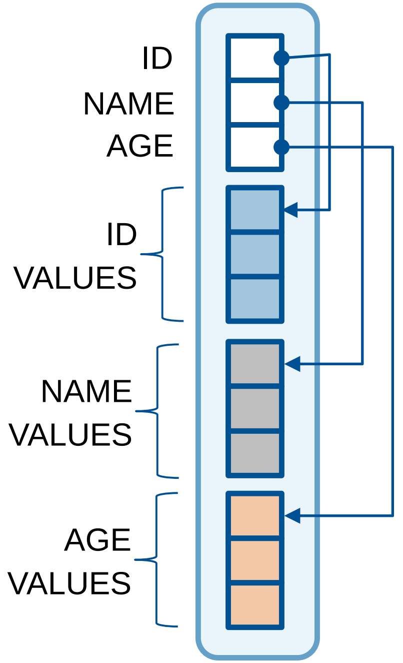
img src: Dageville et al. (2016) The Snowflake Elastic Data Warehouse. SIGMOD

Table storage

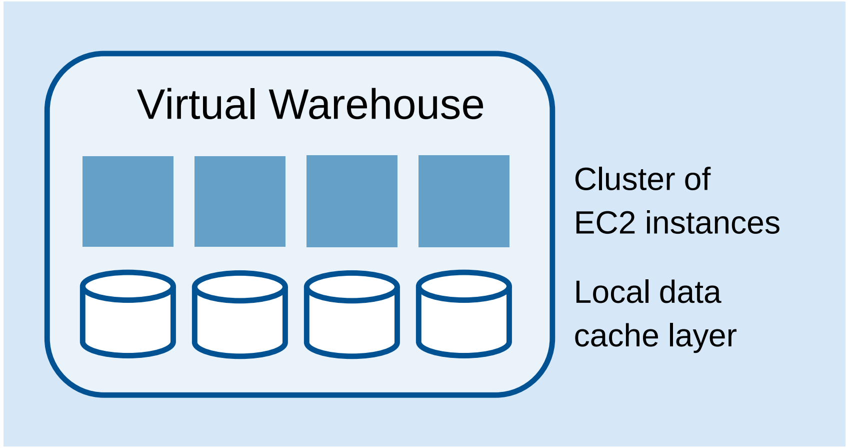
- Tables are horizontally partitioned into large immutable files
  - Similar to blocks or pages in a traditional database system

- Within each file:
  - The values of each attribute (column) are grouped together
  - Heavily compressed (e.g., gzip, RLE, etc.)

- For accelerated query processing:
  - MinMax value of each column of each file of each table are kept in a catalog
  - used for pruning at runtime.



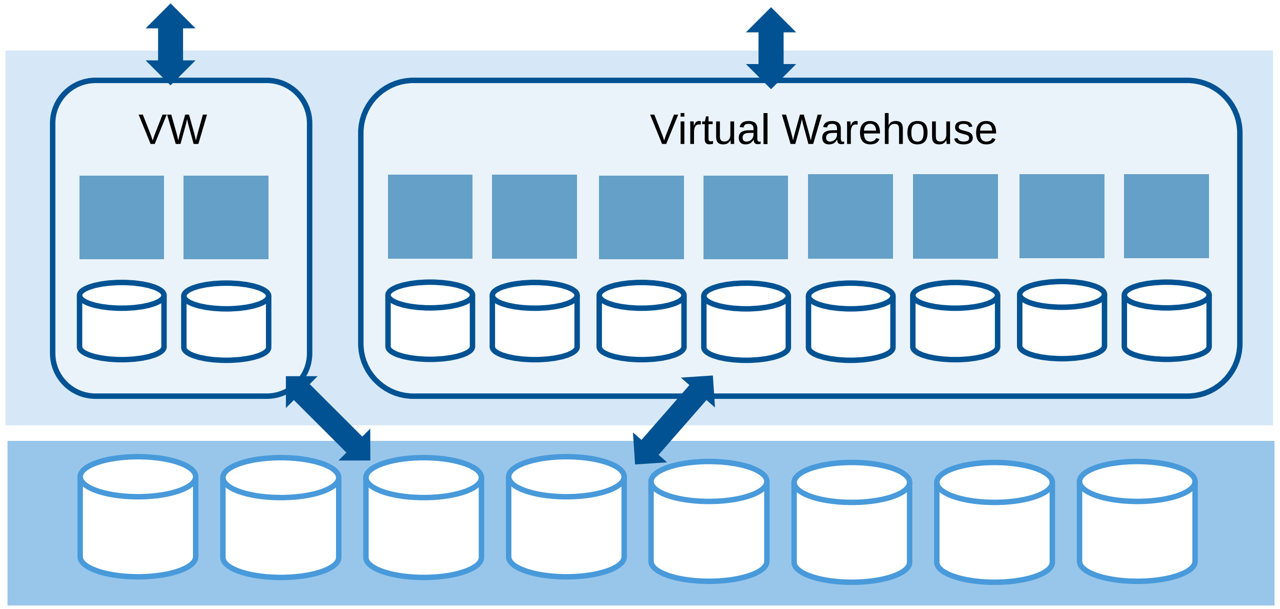
Virtual warehouses



- **Dynamically created cluster of EC2 instances**
  - Pure compute resources
  - Can be created, destroyed, and resized at any time
- **Local disk cache** file headers and **table columns**
- Three sizing mechanisms:
  - Number of EC2 instances
  - Size of each instance (#cores, I/O capacity)

- Each query mapped to exactly one virtual warehouse
- Each VW may run multiple queries in parallel
- Every VW has access to the same **shared table** without needed to copy data

Separate compute and storage



- Queries against the same database can be given the resources to meet their needs
- This flexibility is not feasible with shared-nothing approach (e.g., in Redshift)

Snowflake summary

- Designed for the cloud

- Compute and storage independently scalable
  - Data stored in S3/Azure/GFS but with own closed format (you need to load/trasform)
  - Virtual warehouses composed of clusters of compute (AWS EC2) instances
  - Queries can be given exactly the compute resources they need
  - Query execution is still statefull
  - and is not “serverless”

- No management knobs
  - No indices, no create/update stats, no distribution keys, etc.

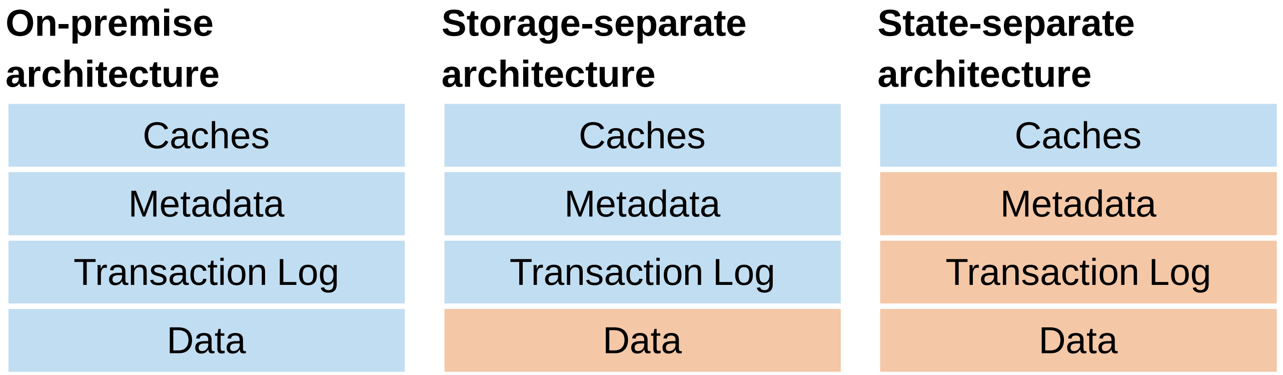
- Can directly query unstructured data (JSON)

Stateless shared-storage architectures

Separating compute and state

Separating Compute and Storage / State

- In stateful architectures, state of in-flight transaction is stored in the compute node and is not hardened into persistent storage until the transaction commits.



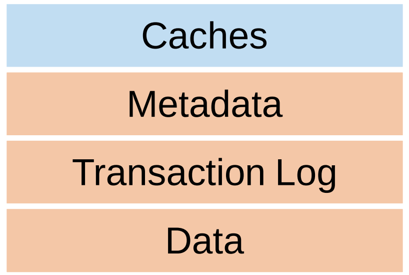
- When a compute node fails, the state of non-committed transaction is lost → fail the transaction

- Resilience to compute node failure and elastic assignment of data to compute are not possible in stateful architectures → the need to move to **stateless architectures.**

Stateless compute architectures

- Compute nodes should not hold any state information

State-separate architecture

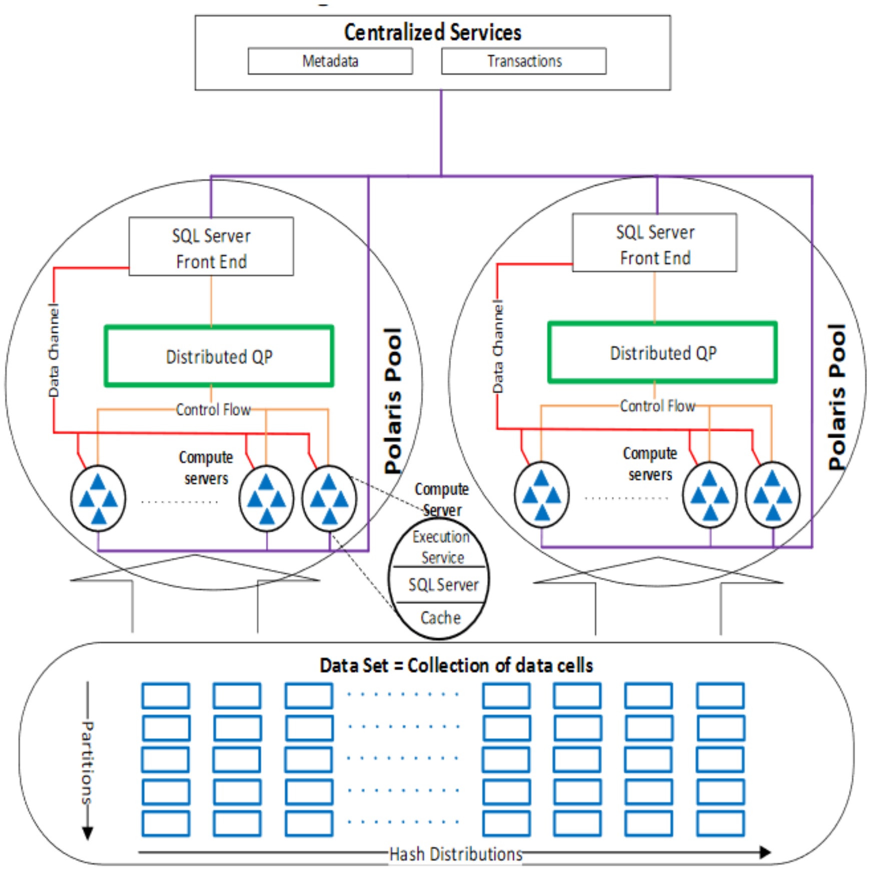


- Caches need to be as close to the compute as possible
  - Can be lazily reconstructed from persistent storage
  - No need to be decoupled from compute

- All data, transactional logs and metadata need to be externalized

- Enables partial restart of query execution in the event of compute node failures and online changes of the cluster topology

Example: POLARIS



- Separation of **storage** and **compute**
  - Compute done by **Polaris pools**
- Shared centralized services
  - Metadata and Transactions
- **Stateless architecture within a pool**
  - Data stored durably in remote storage
  - Metadata and transactional log is offloaded to centralized services (built for high availability and performance)
- Multiple pools can transactionally access the same logical database.

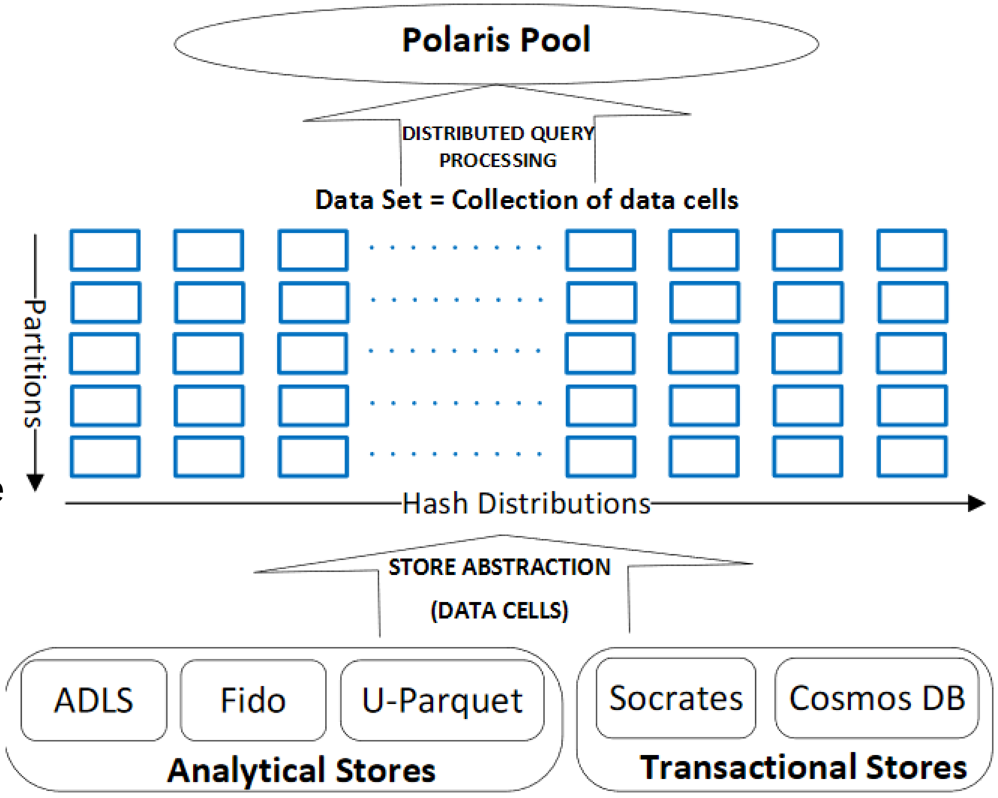
img src: Aguilar-Saborit et al. (2020) POLARIS: The Distributed SQL Engine in Azure Synapse. VLDB

Storage layer considerations

- **Data cells** – abstraction from the underlying data format and storage system
  - Converging data lakes and warehouses

- **Hash-based distribution**
  - To enable easy and balanced distribution of data to compute (e.g., Polaris pool).
  - Hash-distribution h(r) is a system-defined function applied to (a user-defined composite key) r that returns the hash bucket (distribution) that r belongs to – mapping cells to compute nodes.

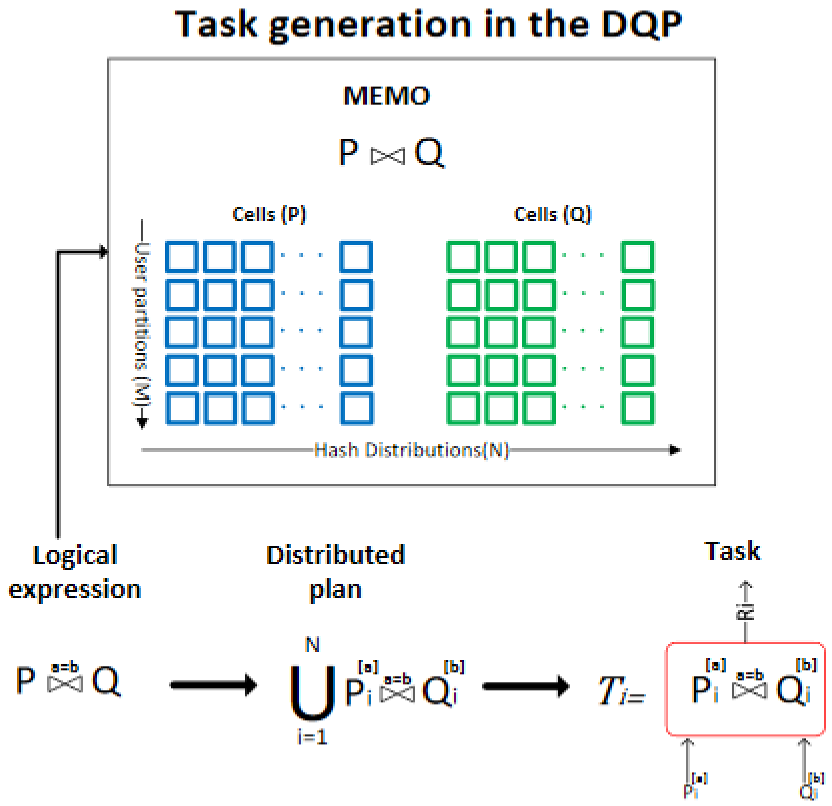
- **The Partitioning function** p(r) is useful for partition range pruning when range or equality predicates are defined over r.



img src: Aguilar-Saborit et al. (2020) POLARIS: The Distributed SQL Engine in Azure Synapse. VLDB

Distributed query processing

- All incoming queries are compiled in two phases:
  - Stage 1 uses SQL server cascades query optimizer to generate the logical search space
    - Contains all logical equivalent alternative plans to execute a query
  - Stage 2 does distributed cost-based query optimizatioı to enumerate all physical distributed implementations of these logical query plans.
    - Picks one with the least estimate cost (taking data movement cost into account).

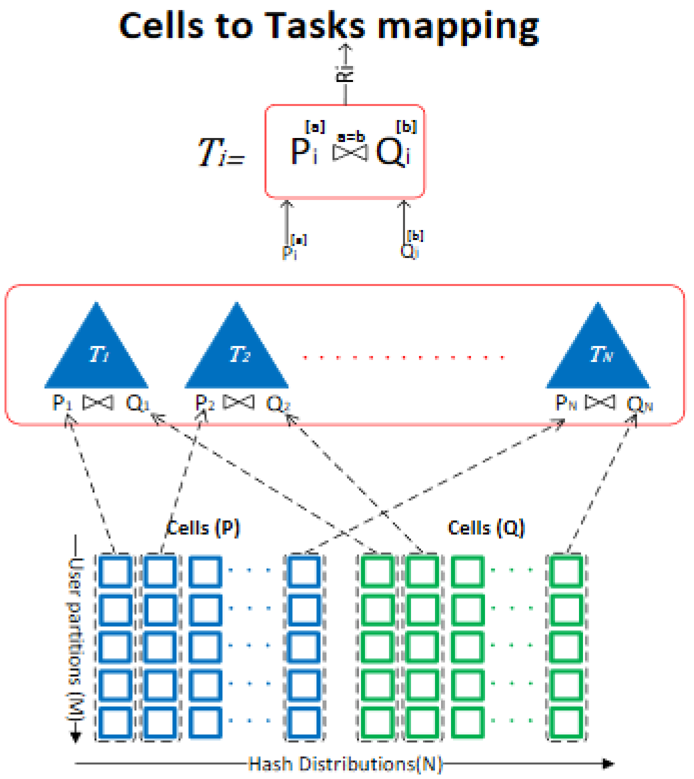

Task generation in the DQP
MEMO
$P \bowtie Q$
Cells (P)
Cells (Q)
User partitions (M)
Hash Distributions(N)
Logical expression
Distributed plan
Task
$P \overset{a=b}{\bowtie} Q$
$\bigcup_{i=1}^{N} P_i^{[a]} \overset{a=b}{\bowtie} Q_i^{[b]}$
$T_i = P_i^{[a]} \overset{a=b}{\bowtie} Q_i^{[b]}$
R_i
$P_i^{[a]}$
$Q_i^{[b]}$

Distributed query processing II

- Task T_i – physical execution of an operator E on the i^{th} hash-distribution of its inputs.

- Tasks are instantiated templates of (the code executing) expression E that run in parallel across N hash-distributions of the inputs.

- A task has three components:
  - **Inputs**: collection of cells for each input's data partition stored either in local or remote storage
  - **Task template**: code to execute on the compute nodes, representing the operator expression E.
  - **Output**: collection of cells produced by the task. Used either input for another task or the final result.



Task organization

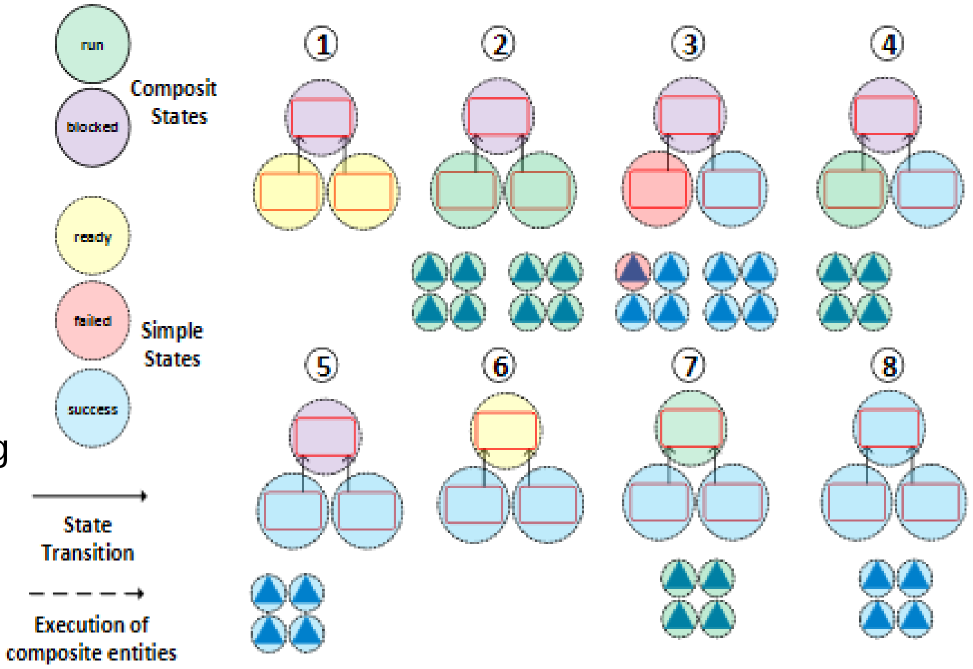
- Model the distribute query execution of queries via hierarchical state machines

- Execution of the query task DAG is top-down in topological sort-order.

- State machines to have fine-grained control at task-level and define a predictable model for recovering from failures.

- States and transitions are logged at each step – necessary for debugging and resuming after failover.

- Low resource overhead for tracking concurrent execution of many queries.



POLARIS Summary

- The separation of state and compute enable offering different service form-factors:
  - Serverless, capacity reservations, multiple pools.

- Data cell abstraction for efficient processing of diverse collection of data formats and storage systems

- Combining scale-up and scale-out
  - Scale-up: intra-partition parallelism, vectorized processing, columnar storage, careful control flow, cache-hierarchy optimizations, deep enhancements to query optimization, etc.
  - Fine-grained scale-out: distributed query processing inspired by big data query execution frameworks

- Elastic query processing via
  - Separation of state and compute
  - Flexible abstraction of datasets as cells
  - Task inputs defined in terms of cells
  - Fine-grained orchestration of tasks using state machines.

Summary

- Architecture of the analytical database system
  - Understand the basic design areas (storage, query processing, system)
    - Column storage, compression
    - Vectorization/JIT, MinMax pushdown
    - Clustering, partitioning/distribution, update infrastructure

- Cloud Database Systems
  - Motivation, Characteristics – differences to on-premise deployments
  - Evolving the architecture of DW for the cloud environment
    - Shared-nothing, shared-storage, stateless query processing
  - Overview of some of the popular systems
    - Redshift, Snowflake, POLARIS

References

The material covered in this class is mainly based on:

- Slides from “Big Data for Data Science” from Prof. Peter Boncz, CWI (link)

Papers:

- Dageville et al. The Snowflake Elastic Data Warehouse SIGMOD 2016
- Aguilar-Saborit et a.. POLARIS: The Distributed SQL Engine in Azure Synapse. VLDB 2020
- Pavlo et al. A Comparison of Approaches to Large-Scale Data Analysis. SIGMOD 2009
- Vuppalapati et al. Building an Elastic Query Engine on Disaggregated Storage. NSDI 2020
- Gupta et al. Amazon Redshift and the Case for Simpler Data Warehouses. SIGMOD 2015
- AWS Redshift: Data warehouse system architecture (link)

Further reading:

- Tan et al. Choosing a Cloud DBMS: Architectures and Tradeoffs. VLDB 2019
- Redshift Spectrum (link)
- Redshift AQUA (link)
- Melnik et al. Dremel: A Decade of Interactive SQL Analysis at Web-Scale. VLDB 2019