Practical Course: Cloud-Based Data Processing

Michalis Georgoulakis

Chair for Database Systems
School of Computation, Information and Technology
Technical University of Munich

09.06.2026

Schedule

	Content	Lab Start	Lab End	Oral Discussions
09.06	Cloud Compute & Networking	Lab 6	Lab 5	Lab 3
16.06	Cloud Benchmarking	Project	Lab 6	Lab 4, Lab 5
23.06	Topic Selection			Lab 6
30.06	Q&A			
07.07	Q&A			
14.07	Project Presentations			

Lab 3 Takeaways

Pre-aggregation helps the wire but not the wall clock

The coordinator is the real broadcast bottleneck

- Q4 broadcast: >75% of wall time is the coordinator reading + aggregating + serializing the right side
- Coordinator I/O throughput ≈ 130 MB/s single-threaded — so 7.9 GB takes ~60 s regardless of network speed

The big side also matters

- When the bigger side is 4 GB → shuffle wins Q3
- When the bigger side is 15–40 GB → broadcast wins Q3

Methodology Good Practices

- ≥ 3 runs per cell, drop the cold-start outlier
- Drop OS page cache between runs to get reproducible cold numbers
- Measure B instead of assuming the NIC line rate
- Sweep M on Q4 with the yellow filelist_{small,medium,large} to see the crossover
- Separate timing for coordinator-build vs send vs decode

Data Warehouse Storage Considerations

- **Data layout**: column-store vs. row-store
  - Column-stores only read relevant data (skip irrelevant data by skipping unrelated columns)
  - Suitable for analytical workloads
  - Prime Example: Apache Parquet

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

- **Pruning**: skip irrelevant data
  - 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
    - each compute node gets a piece of the data
    - each query has work on every piece (keep everyone busy)
- **Partitioning** (user-specified)
  - Goal 1: data lifecycle management
    - Data warehouse e.g., keeps last six months
    - Every night: load one new day, drop the oldest partition
  - Goal 2: improve access pattern with partition pruning
    - When querying for month “May”, drop partitions P1,P3,P4.

Moving to the cloud

The shared-nothing cluster doesn’t fit

Cloud-native warehouses

- Design considerations for good performance, scalability, elasticity, fault-tolerance, etc.

- 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 across the (deep) storage hierarchy (cost/performance)
  - Query execution:
    - Distributed query execution: combine scale-out and scale-up
    - Distributed query optimization
    - Global resource-aware scheduling

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

How cloud warehouses evolved: four moves

- Start: a **shared-nothing cluster** binds compute, storage and state to the same nodes
  - Fast on a fixed cluster, but brittle in the cloud: rigid scaling, node failures, hard upgrades
- **Move 1**: separate storage from compute
  - Shared storage, each layer scales on its own (*Snowflake virtual warehouses)*

How cloud warehouses evolved: four moves

- Start: a **shared-nothing cluster** binds compute, storage and state to the same nodes
  - Fast on a fixed cluster, but brittle in the cloud: rigid scaling, node failures, hard upgrades
- **Move 1**: separate storage from compute
  - Shared storage, each layer scales on its own (*Snowflake virtual warehouses)*
- **Move 2**: disaggregate memory as well
  - A shuffle-memory tier: spill less, join faster (*BigQuery)*

How cloud warehouses evolved: four moves

- Start: a **shared-nothing cluster** binds compute, storage and state to the same nodes
  - Fast on a fixed cluster, but brittle in the cloud: rigid scaling, node failures, hard upgrades
- **Move 1**: separate storage from compute
  - Shared storage, each layer scales on its own (*Snowflake virtual warehouses)*
- **Move 2**: disaggregate memory as well
  - A shuffle-memory tier: spill less, join faster (*BigQuery)*
- **Move 3:** make compute stateless
  - Externalize data, log and metadata so any node can restart (*POLARIS, BigQuery shuffle tier)*

How cloud warehouses evolved: four moves

- Start: a **shared-nothing cluster** binds compute, storage and state to the same nodes
  - Fast on a fixed cluster, but brittle in the cloud: rigid scaling, node failures, hard upgrades
- **Move 1**: separate storage from compute
  - Shared storage, each layer scales on its own (*Snowflake virtual warehouses)*
- **Move 2**: disaggregate memory as well
  - A shuffle-memory tier: spill less, join faster (*BigQuery)*
- **Move 3:** make compute stateless
  - Externalize data, log and metadata so any node can restart (*POLARIS, BigQuery shuffle tier)*
- **Move 4**: drop provisioning entirely: go serverless
  - Pay per query — serverless DBs, or FaaS + object storage

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 a poor fit for complex queries (low I/O bandwidth, heavy compute).

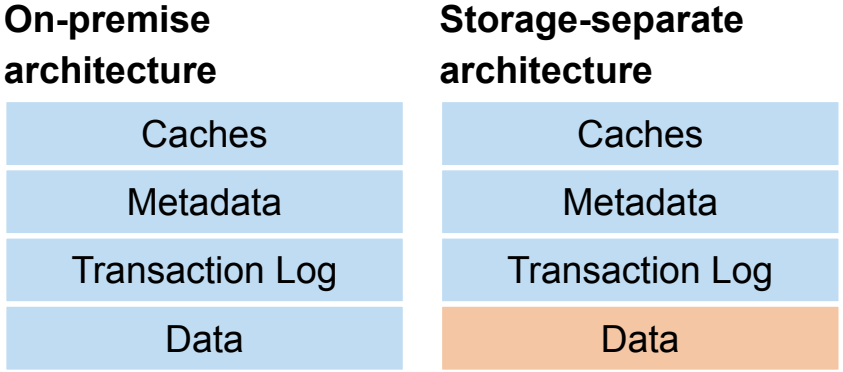
- **Membership changes**
  - if the set of nodes changes potentially a large volume of data needs to be reshuffled

- **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**

1. Separating Compute and Storage

- Evolution of cloud-data warehouse architectures over the years
- Engines maintain state comprised of: cache, metadata, transaction log, and data

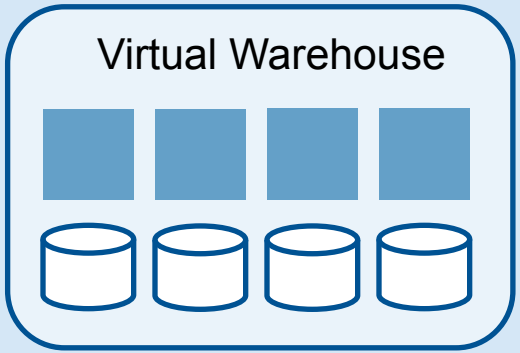


- The first step is **decoupling of storage and compute**

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

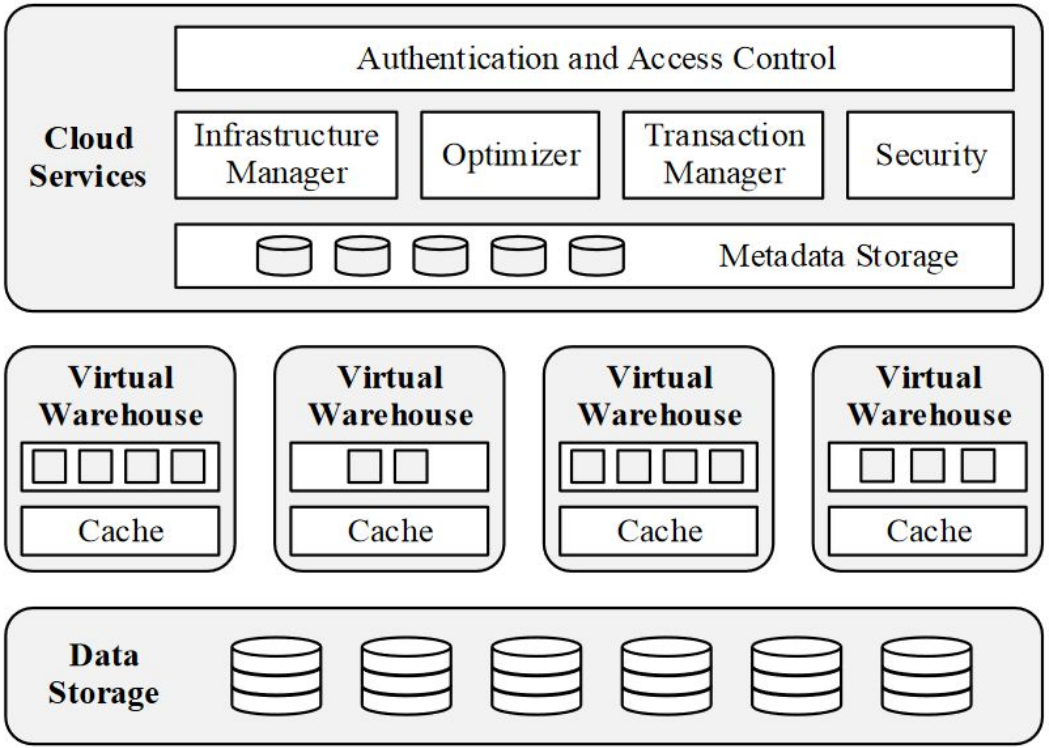
Logic behind virtual warehouses



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

- Each query mapped to exactly one virtual warehouse (VW)
- Each VW can run multiple queries in parallel
- Every VW has access to the same **shared data** without needing to copy it.

Example: Snowflake



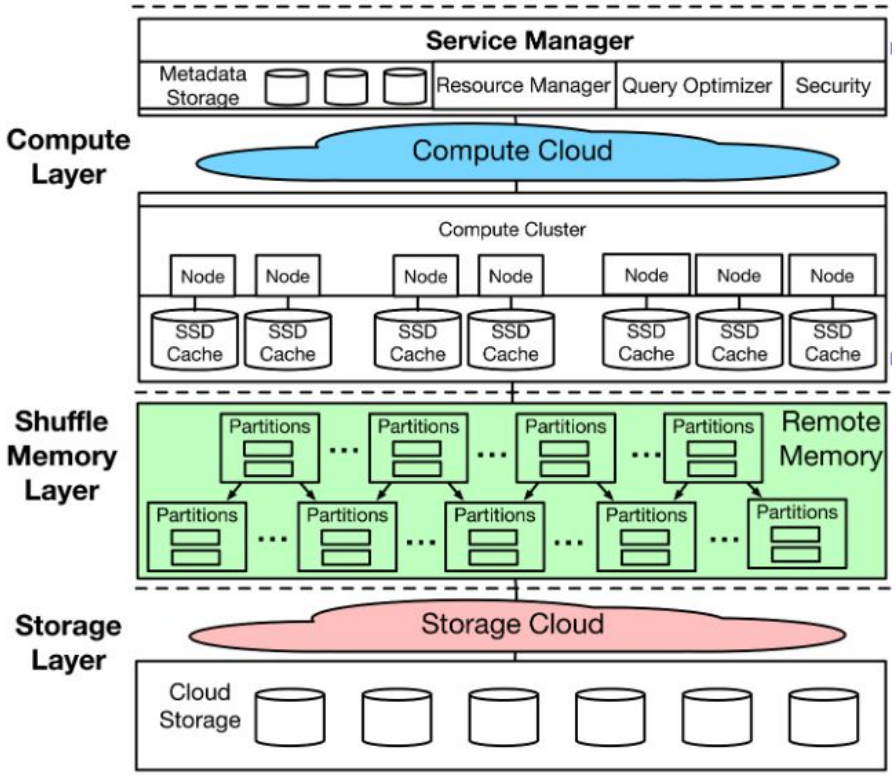
The Brain: Key data management services.

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

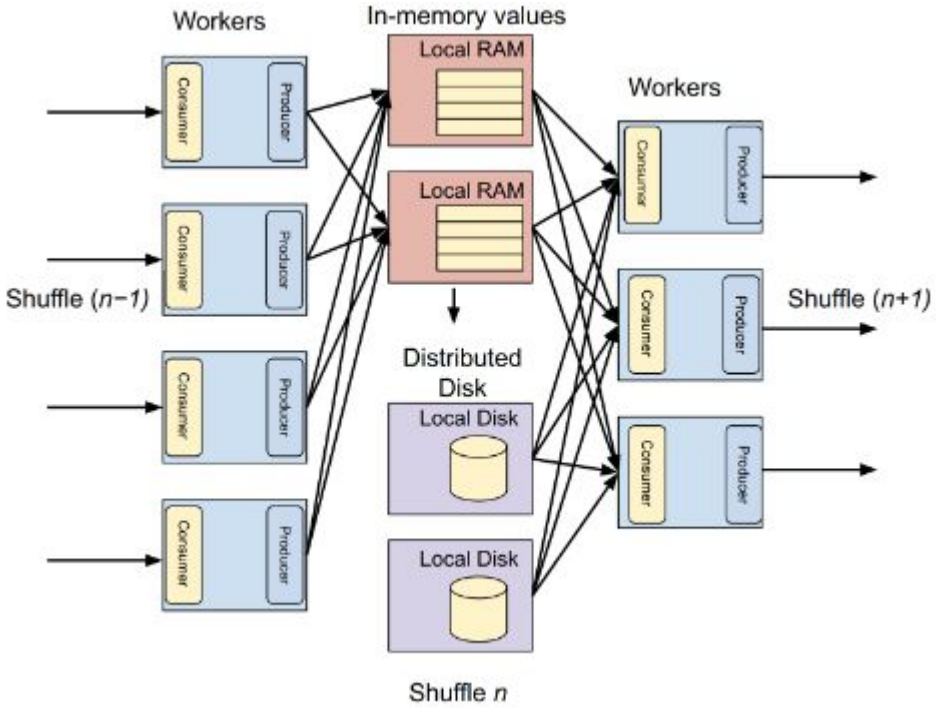
Disaggregated Compute-Memory-Storage Arch.



Src: Li et al. Cloud-Native Databases, VLDB'22

- **Advantages:**
  - Elasticity: storage, compute *and* memory can be scaled independently
  - Schedule the resources for better utilization
  - Complex workloads: cope with the large intermediate results
- **Key features:**
  - Shuffle-memory layer for speeding up joins
  - Reduces I/O cost by avoiding to write the intermediate results to disks
- **Examples:**
  - BigQuery

BigQuery's Shuffle Workflow



- **Producer** in each worker:
  - generates partitions and
  - sends them to the shuffling layer
- **Consumer**:
  - combines the partitions and
  - performs the operations locally
- Large intermediate results can be spilled to local disks

Melnik et al. Dremel: A Decade of Interactive SQL Analysis at Web Scale

Stateless shared-storage architectures

Move 3: separating compute from 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.

On-premise architecture	**Storage-separate architecture**	**State-separate architecture**
Caches	Caches	Caches
Metadata	Metadata	Metadata
Transaction Log	Transaction Log	Transaction Log
Data	Data	Data

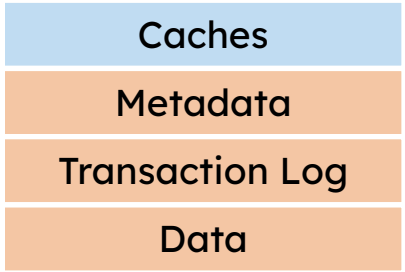
- When a compute node fails, the state of non-committed transaction is lost □ fail the transaction

- 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



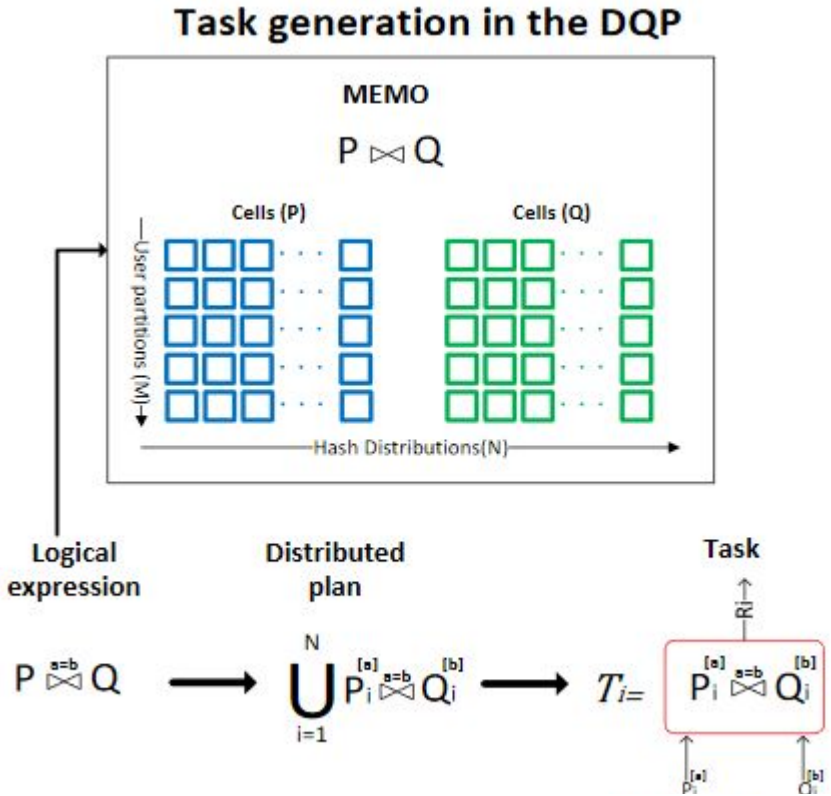
- Caches need to be as close to the compute as possible
  - Can be lazily reconstructed from persistent storage
  - No need to be decoupled from compute
- All data, transactional logs and metadata need to be *externalized*

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

- Examples: BigQuery using the shuffle tier and dynamic scheduler, POLARIS

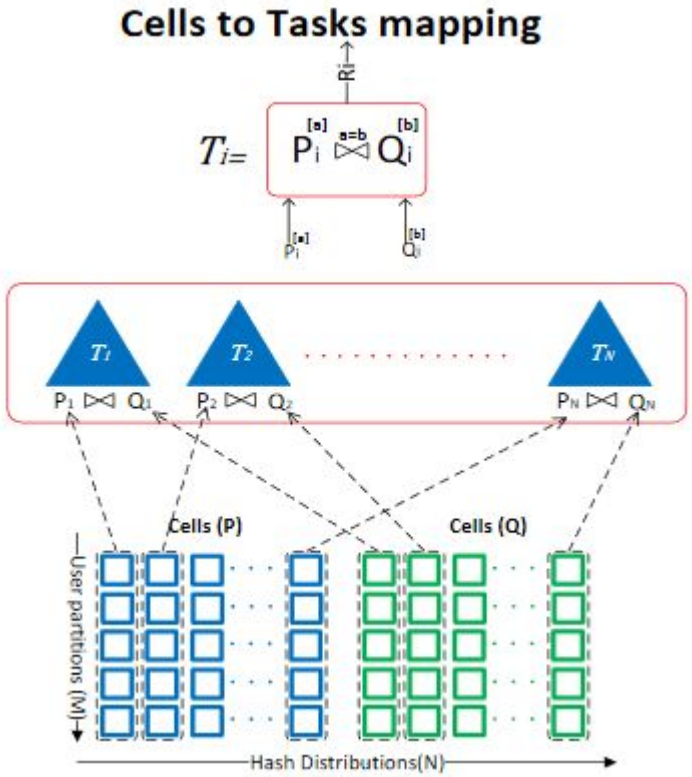
Distributed query processing – part 1

- All incoming queries are **compiled in two phases:**
  - **Stage 1 uses SQL server query optimizer** to generate all logical equivalent plans to execute a query
    - Uses data for the collection of files/tables, partitions, and distributions
  - **Stage 2 does distributed cost-based query optimization** to enumerate all physical distributed implementations of these logical query plans.
    - Picks one with the least estimate cost (taking data movement cost into account).



Distributed query processing – part 2

- Task Ti: physical execution of an operator E on the ith hash-distribution of its inputs
- Tasks are instantiated templates of (the code executing) an expression E that run in parallel across N hash-distributions of the inputs
- A task has three components:
  - **Inputs**: collections of cells for each inputs 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. Either input for another task or the final result.



Serverless

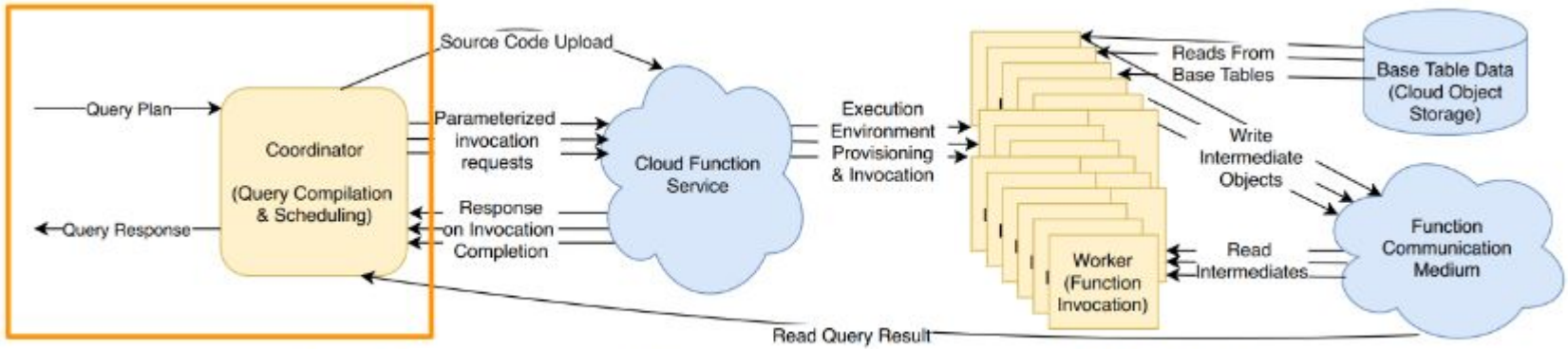
Move 4: drop provisioning entirely — pay per query

Serverless Computing for Queries

- Issue queries in the cloud without worrying about resource provisioning and pay by query granularity
- Two main approaches:
  - **Serverless Databases:**
    - Rely on the cloud SQL engine and storage to execute the queries with dynamic resource provisioning
    - The DB service can pause when idle and resume when a query comes in
  - **Serverless functions + Cloud storage:**
    - Rely on Function-as-a-Service (FaaS) and cloud storage to run queries with on-demand resources

Serverless Functions + Cloud storage

- **Two challenges:**
  - Functions are **stateless**
  - **Stragglers** increase the overall latency of the parallel query processing
- **Approach:**
  - Use cloud **storage** to exchange **state** □ similar to state-separate query processing
  - Use tuned models to **detect stragglers** and invoke functions with **duplication computation**

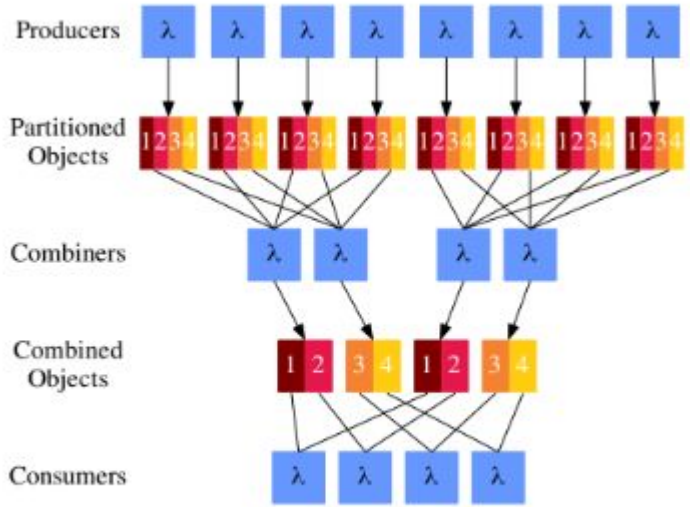


Perron, Matthew, et al. "Starling: A scalable query engine on cloud functions." *In SIGMOD*. 2020.

Serverless Functions + Cloud storage

- **Query processing using lambda functions**
  - Invoke many tasks in each stage
  - Each task writes the intermediate results to a single object file
  - Combiners can be used to reduce the read cost of the large shuffle

- **Trade-off between the number of invoked tasks (performance) and cost**



Multi-Stage Shuffling based on Functions

Src: Starling: A scalable query engine on cloud functions. SIGMOD'2

From FaaS query engines to your Lab

- **The pattern across Snowflake, BigQuery and Starling:** compute is disposable, all state lives in cloud object storage
  - A serverless/FaaS worker spends most of its life reading inputs and intermediates from that storage
- **One primitive decides FaaS query performance:** how fast a worker saturates throughput to object storage
  - Producers read base tables, consumers read shuffle objects — the read path is the bottleneck
  - A single-threaded reader caps out fast

Lab 6 - build a data processing pipeline over FaaS

Deadline: **16.06, 13:59:59**