Cloud-Native Databases

Guoliang Li **Haowen Dong** **Chao Zhang**

Tsinghua University

Motivation of Cloud Database

☐Market Trends: Databases are moving to Cloud

➢ Growth Speed

- 68% of the growth of the DBMS Market came from cloud.
- 38.2% annual growth rate from 2021 to 2026.

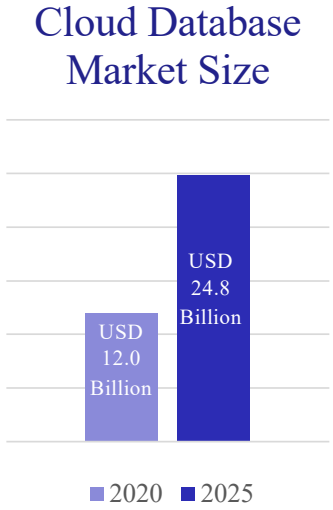
➢ Revenue Rate

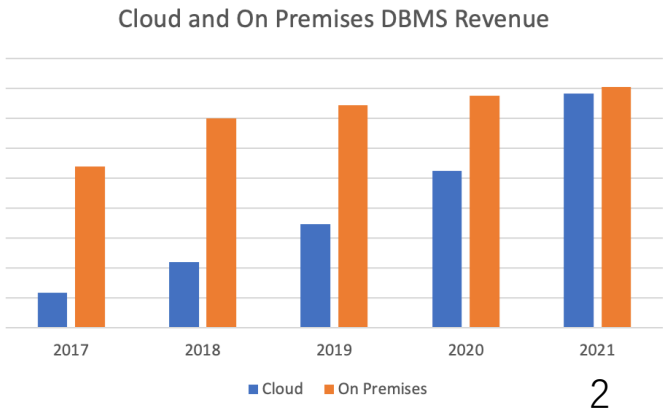
- $39.2 billion, 49% of all DBMS revenue from cloud(2021).

➢ Market Share

- 75% of Databases will be on Cloud in 2023

➢ "The future of DBMS Market is Cloud."





Motivation of Cloud Database

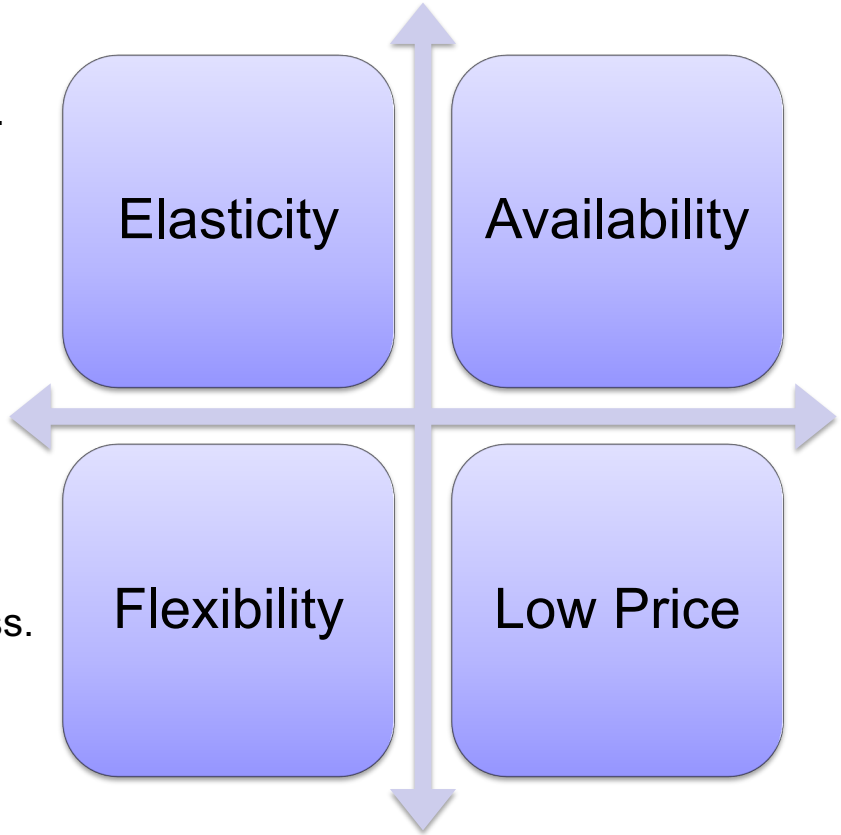
☐Advantages for Cloud Database Customers

Auto Scaling Service.
- Service can auto scale up or down based on workload.

Infinite Capacity.
- The system can provide nearly infinite resources to users.

Out-of-the-box Feature.
- Users can use the service without worrying about the complex deployment process.

Auto Tuning & Optimizing.
- Support auto tuning & optimizing.



High Service Availability.
- The system maintains multiple replicas to support high service availability.

Strong Data Durability.
- Replicas deployed in different locations guarantee data durability over extreme disasters.

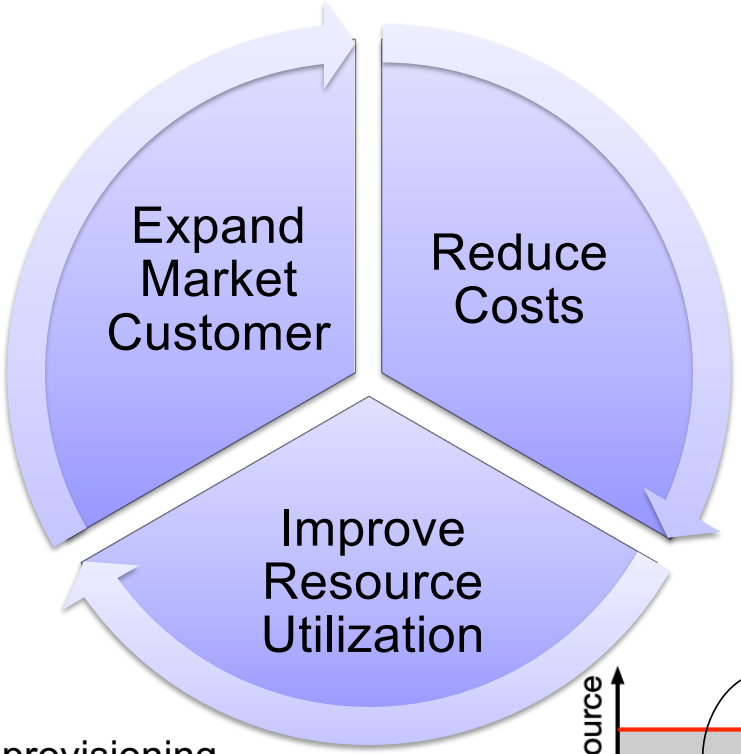
Pay-as-you-go Pricing Model.
- Converting capital expenses to operating expenses. Users only need to pay for usage, instead of the maximum capacity of the whole workload.

Motivation of Cloud Database

☐Advantages for Cloud Database Providers

Demand Expansion.
- ➢ The era of big data, the demand of data processing is expanding rapidly.

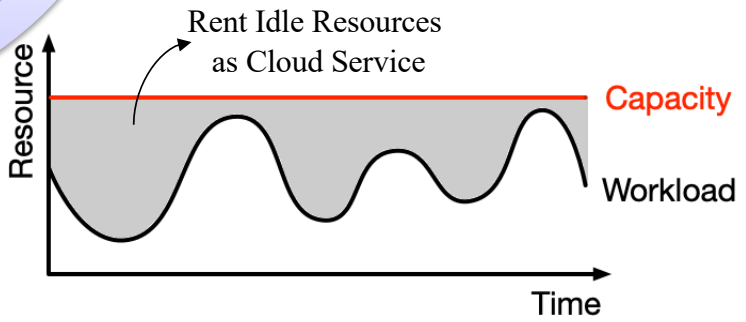
New Target Customers.
- ➢ Flexibility of service attracts grand amount of small business or individual users without professional data management team.



Scale Effect in Deployment.
- ➢ Wholesale price on hardware purchase.
- ➢ Hyperscale data centers.

Scale Effect in Operation.
- ➢ Unified management team.
- ➢ Low electricity price, benefits of data center locations.

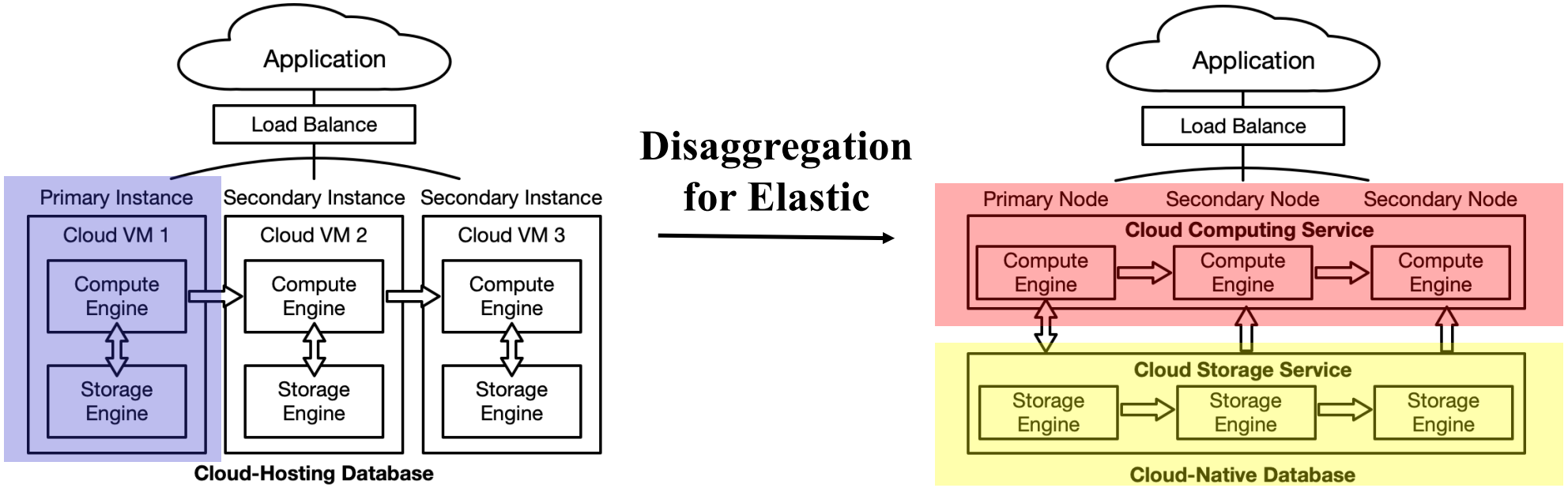
Resource Utilization.
- ➢ Fixed-sized resource provisioning meets dynamic workloads.
- ➢ Rent resources as cloud service to improve utilization.

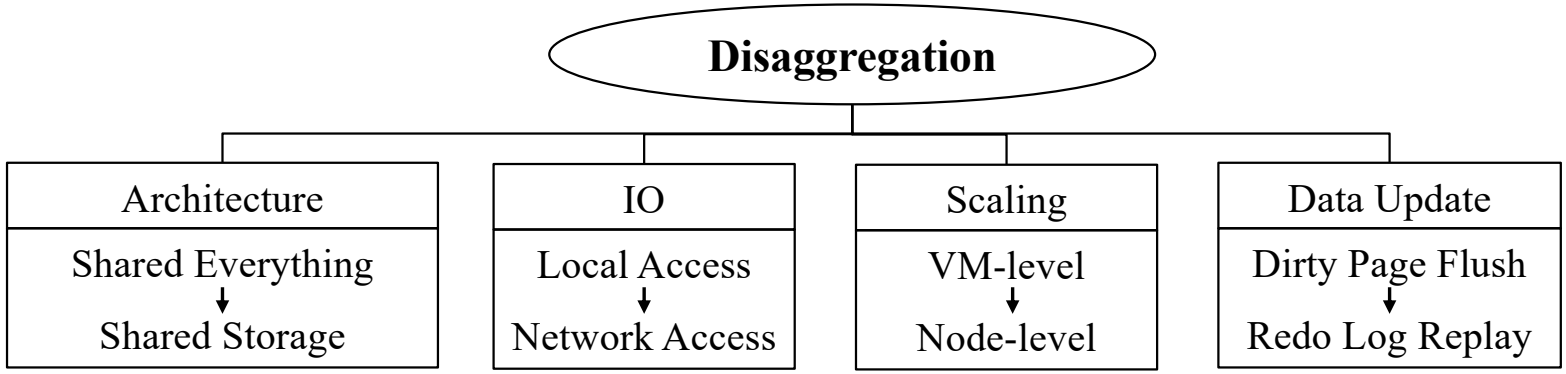


Outline of Tutorial

- From Cloud-Hosting to Cloud-Native
- Cloud-Native OLTP Architectures
- Cloud-Native OLTP Techniques
- Cloud-Native OLAP Architectures
- Cloud-Native OLAP Techniques
- Open Problems & Opportunities

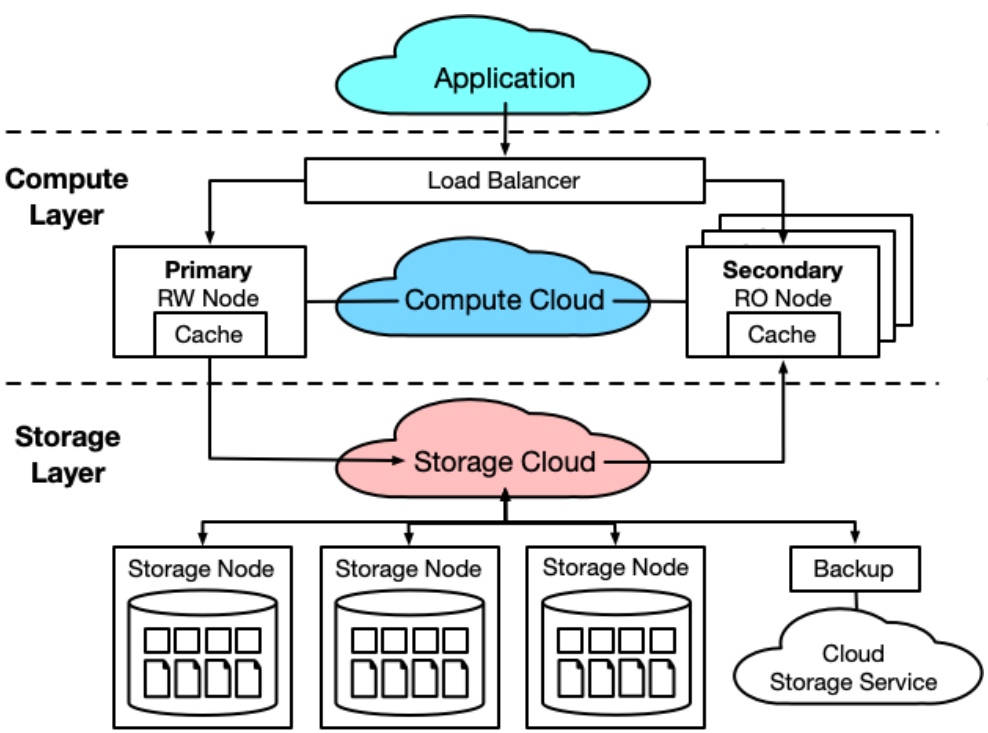
From Cloud-Hosting to Cloud-Native



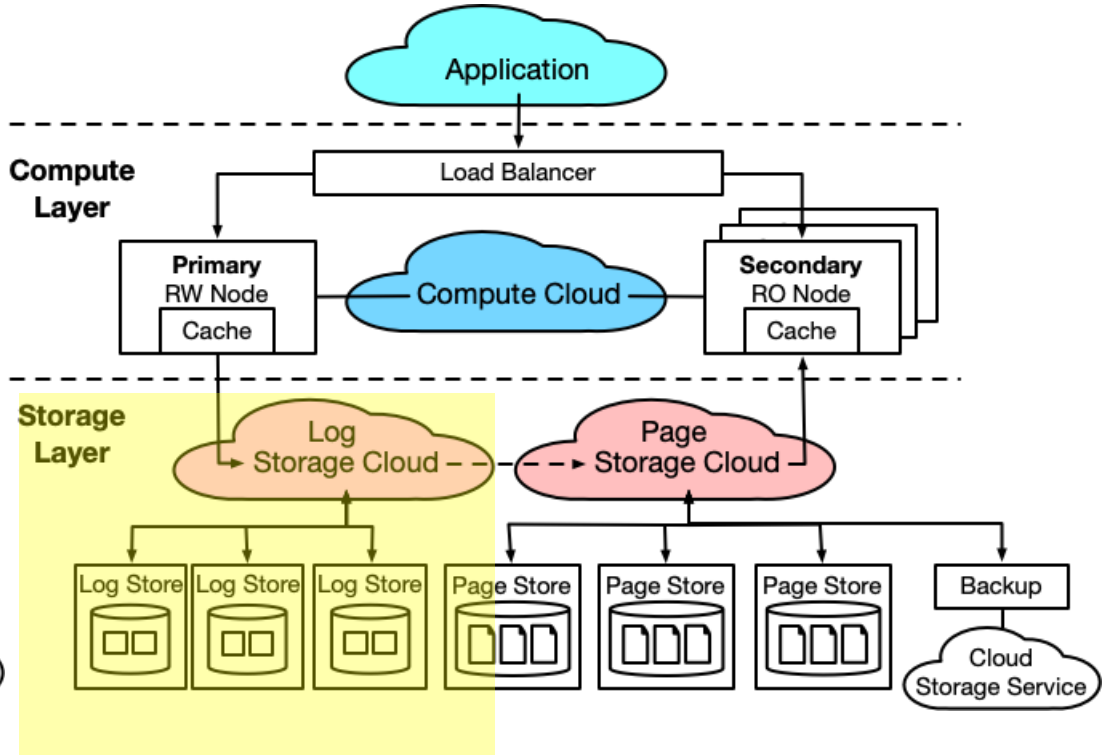


Cloud-Native OLTP Architectures

An Overview of Cloud OLTP Architectures

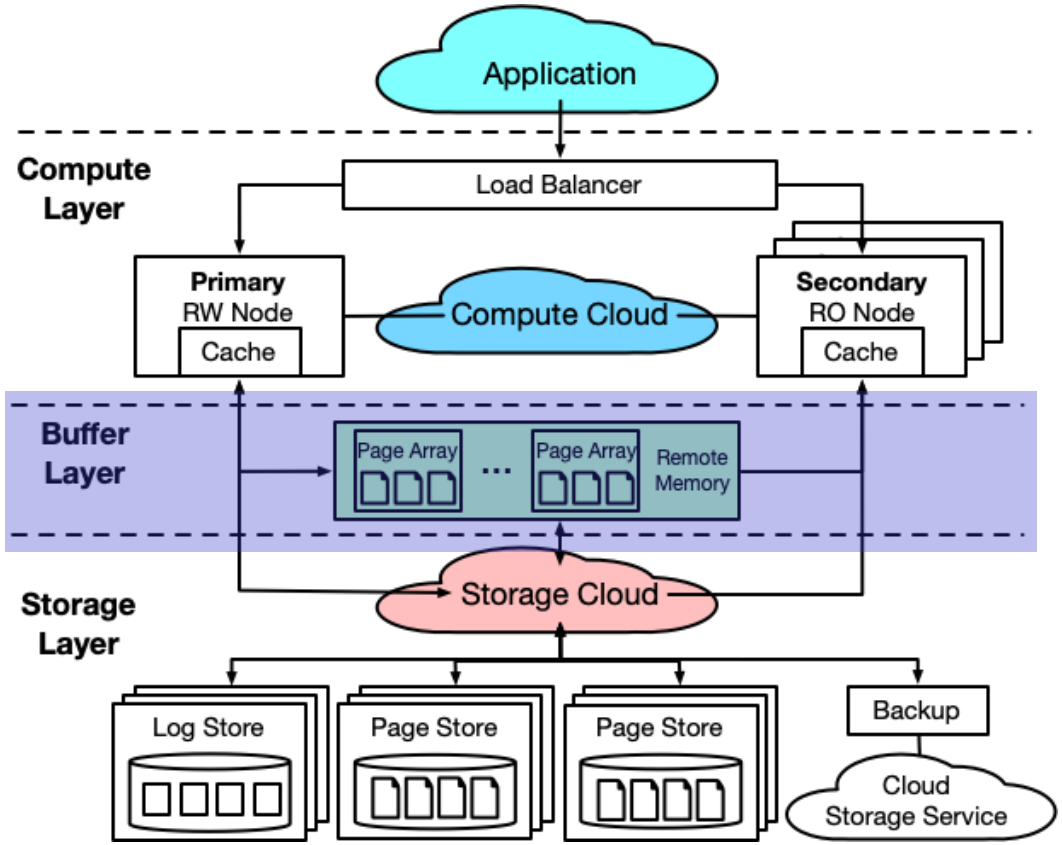


(1) Disaggregated Compute-Storage



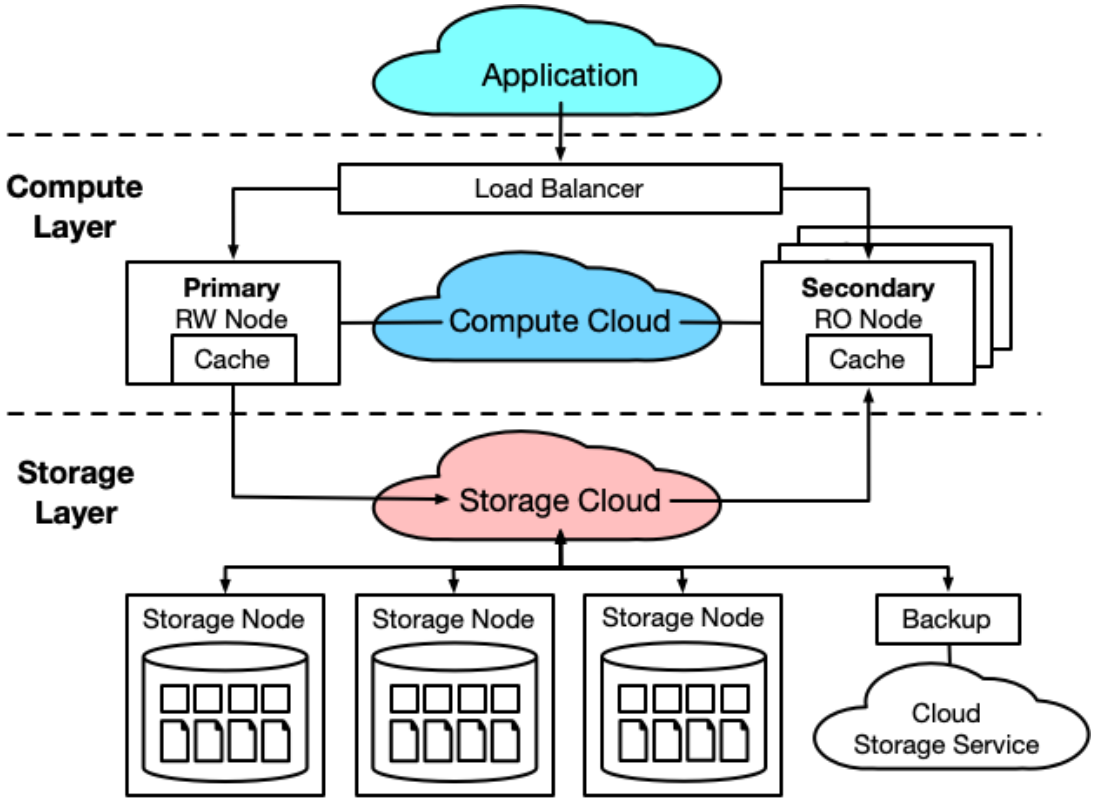
(2) Disaggregated Compute-Log-Storage

An Overview of Cloud OLTP Architectures



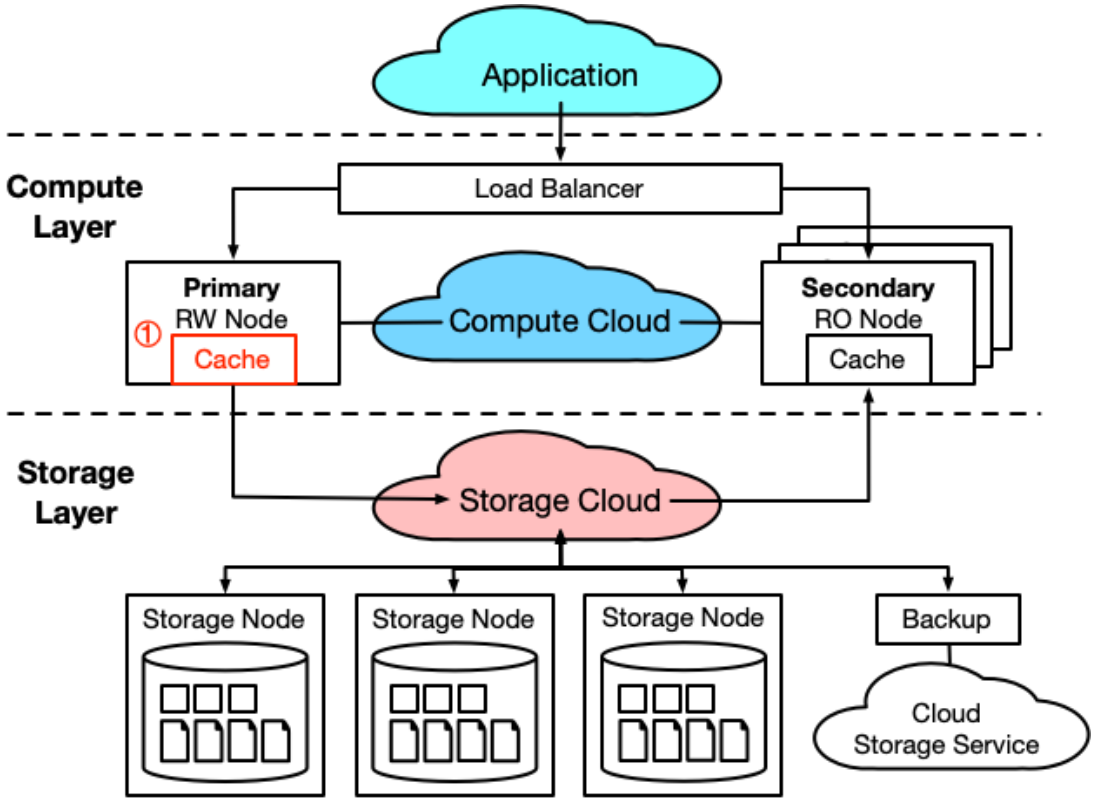
(3) Disaggregated Compute-Buffer-Storage

(1) Disaggregated Compute-Storage Architecture



- Motivations:
  - **Elasticity.** Compute and Storage can be scheduled individually
  - **Efficiency.** Reduce write amplification.
  - **Availability.** Multi-layer recovery mechanism to handle various exceptions.
- Key Features:
  - Disaggregation of Compute & Storage.
  - Log is the database.

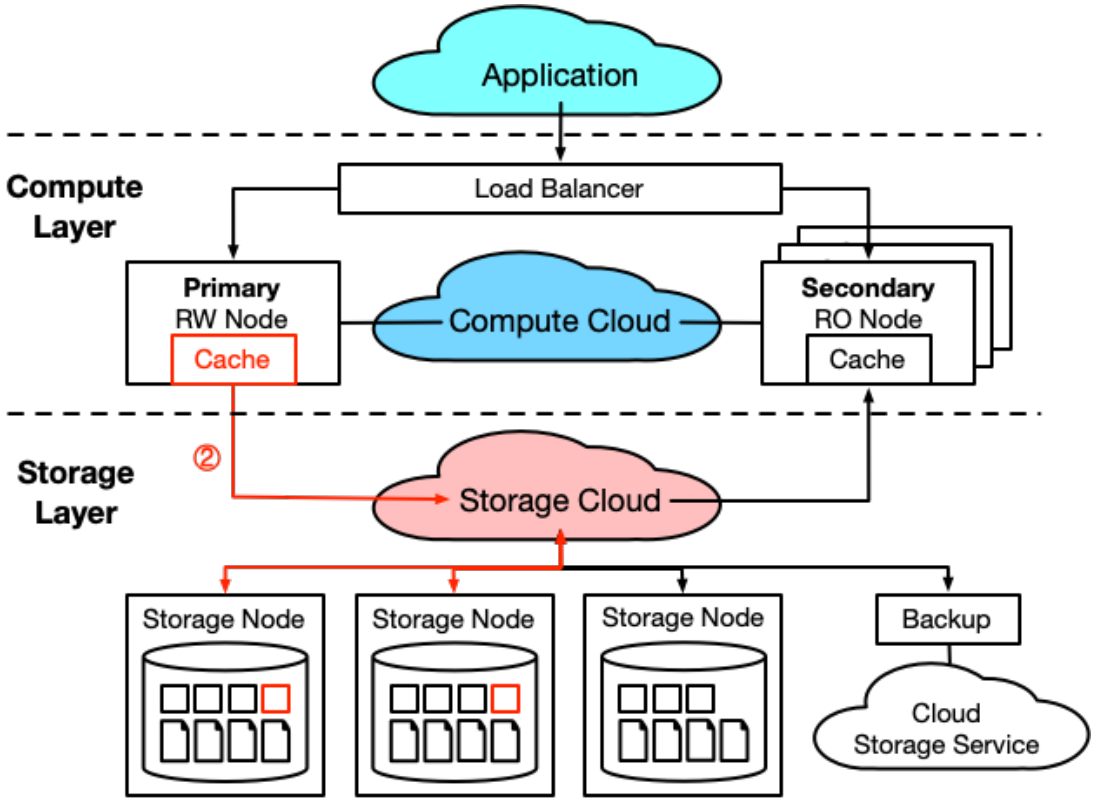
(1) Disaggregated Compute-Storage Architecture



- Data write path:

1. **Primary node** updates local page in cache and generates redo log.
2. Primary node writes redo log to majority nodes in the storage cloud.
3. Commit the write after the majority of data replicas finish the log writing process for durability.

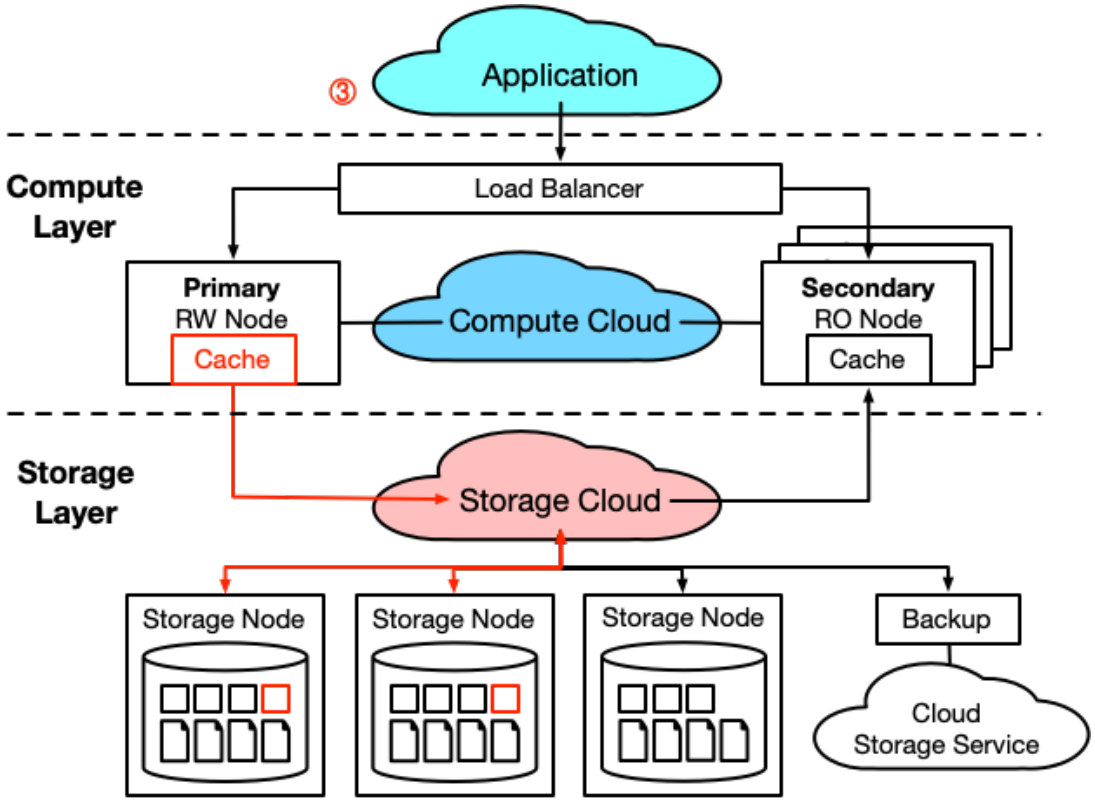
(1) Disaggregated Compute-Storage Architecture



- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. Primary node writes redo log to **majority nodes** in the storage cloud.
3. Commit the write after the majority of data replicas finish the log writing process for durability.

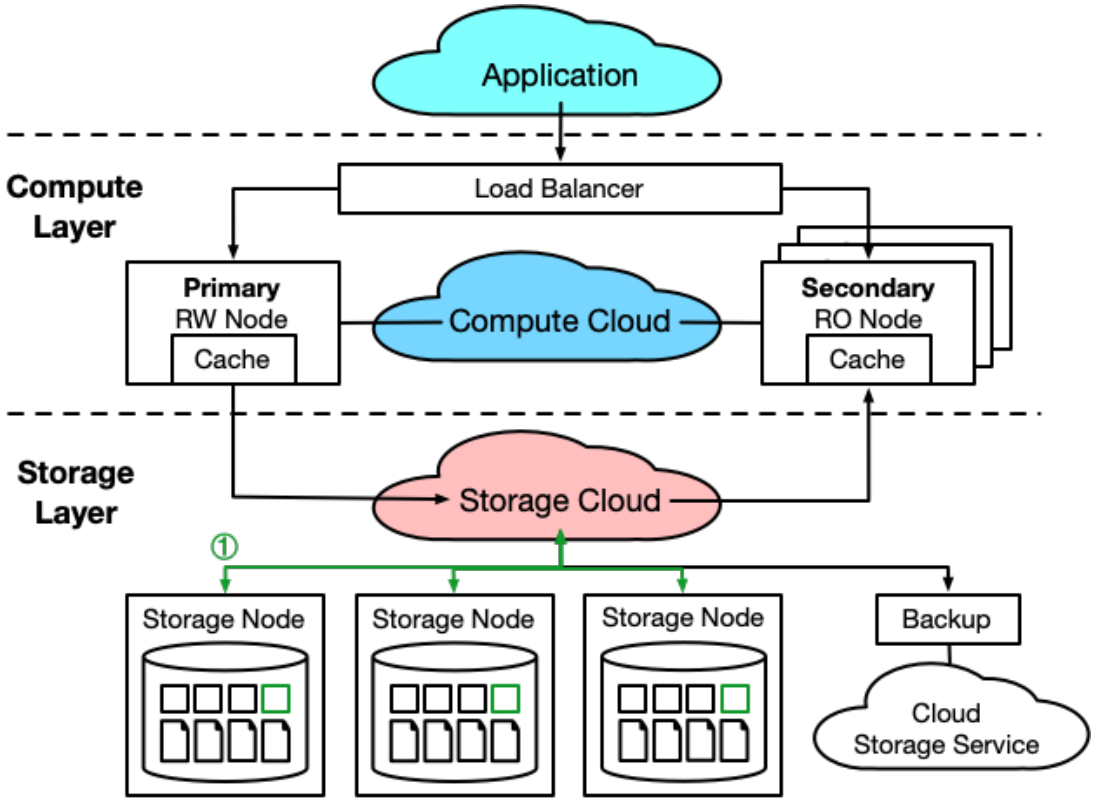
(1) Disaggregated Compute-Storage Architecture



- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. Primary node writes redo log to majority nodes in the storage cloud.
3. Commit the write after the majority of data replicas finish the log writing process for durability.

(1) Disaggregated Compute-Storage Architecture



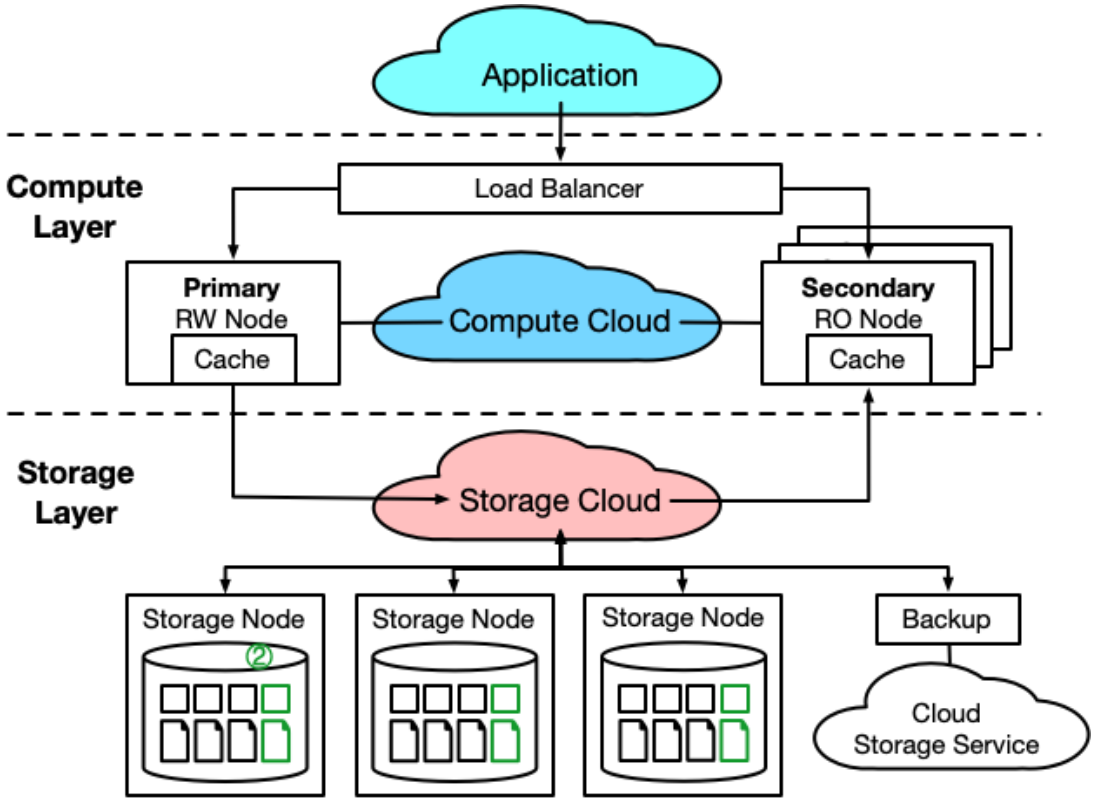
- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. Primary node writes redo log to majority nodes in the storage cloud.
3. Commit the write after the majority of data replicas finish the log writing process for durability.

- Data sync path:

1. The consistency of replicas is maintained based on **the log data**.
2. Redo logs are replayed to the page asynchronously in the storage node.
3. Directly transfer logs to secondary nodes to reduce the update latency.

(1) Disaggregated Compute-Storage Architecture



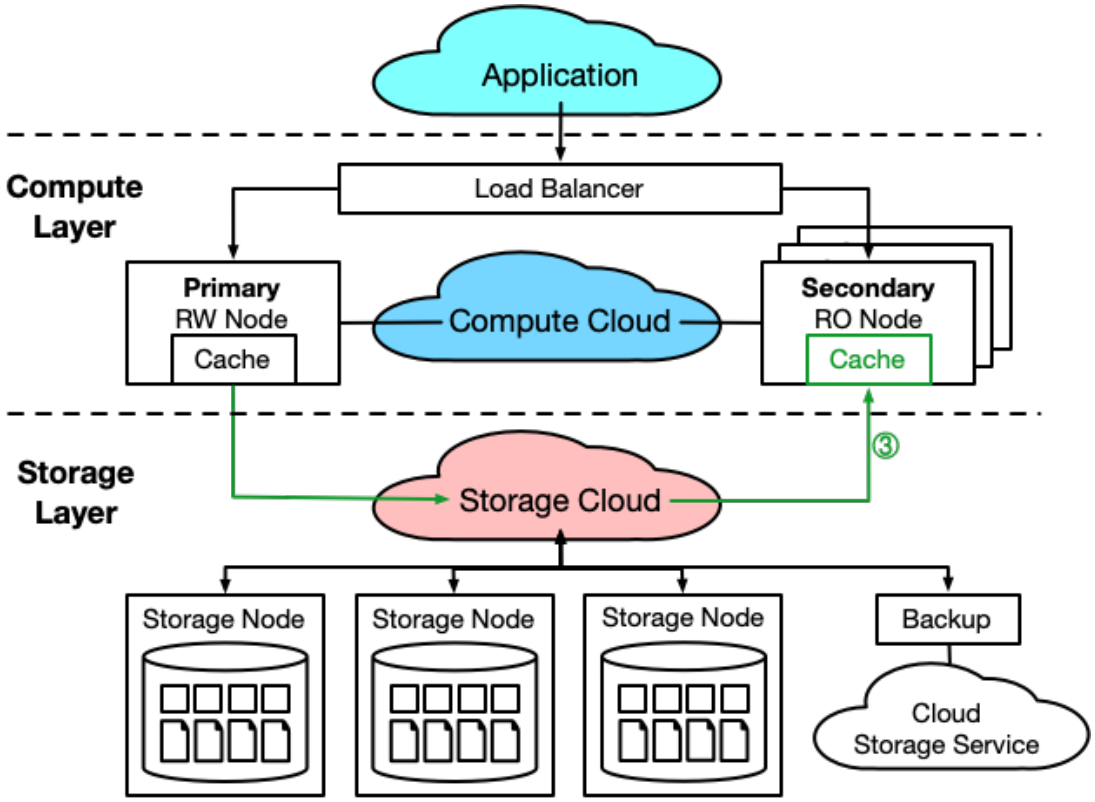
- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. Primary node writes redo log to majority nodes in the storage cloud.
3. Commit the write after the majority of data replicas finish the log writing process for durability.

- Data sync path:

1. The consistency of replicas is maintained based on the log data.
2. Redo logs are replayed to the page **asynchronously** in the storage node.
3. Directly transfer logs to secondary nodes to reduce the update latency.

(1) Disaggregated Compute-Storage Architecture



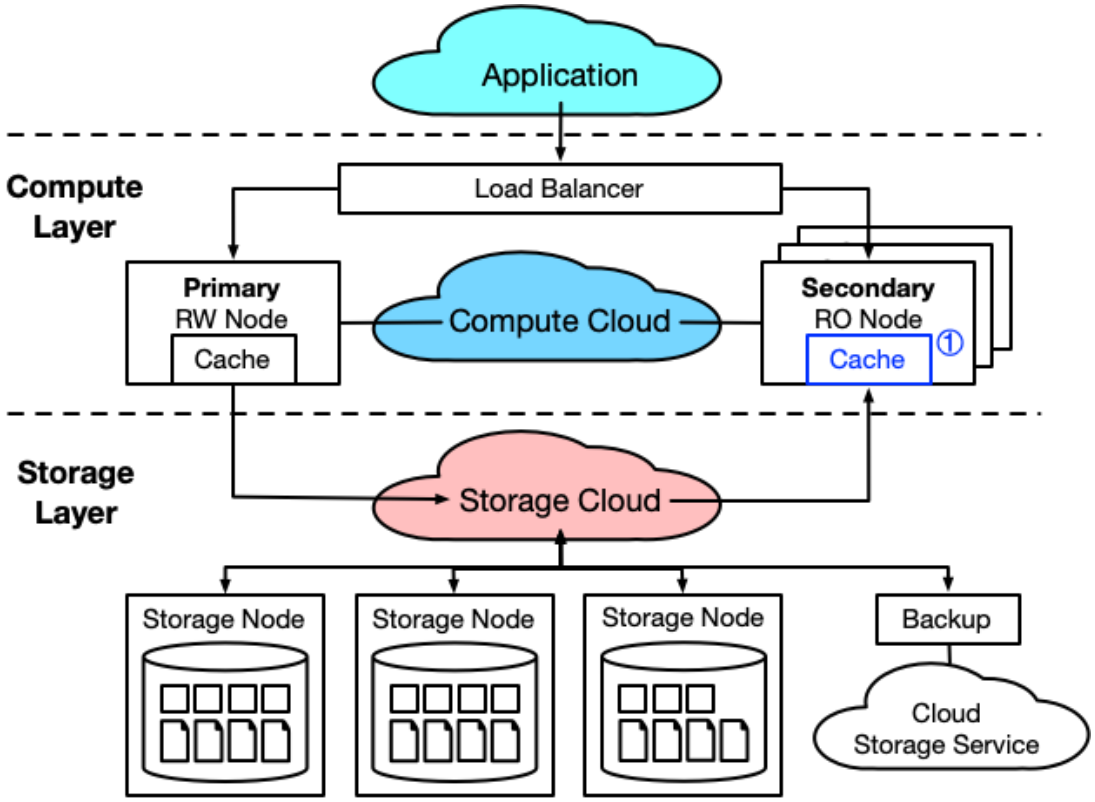
- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. Primary node writes redo log to majority nodes in the storage cloud.
3. Commit the write after the majority of data replicas finish the log writing process for durability.

- Data sync path:

1. The consistency of replicas is maintained based on the log data.
2. Redo logs are replayed to the page asynchronously in the storage node.
3. Directly transfer **logs** to secondary nodes to reduce the update latency.

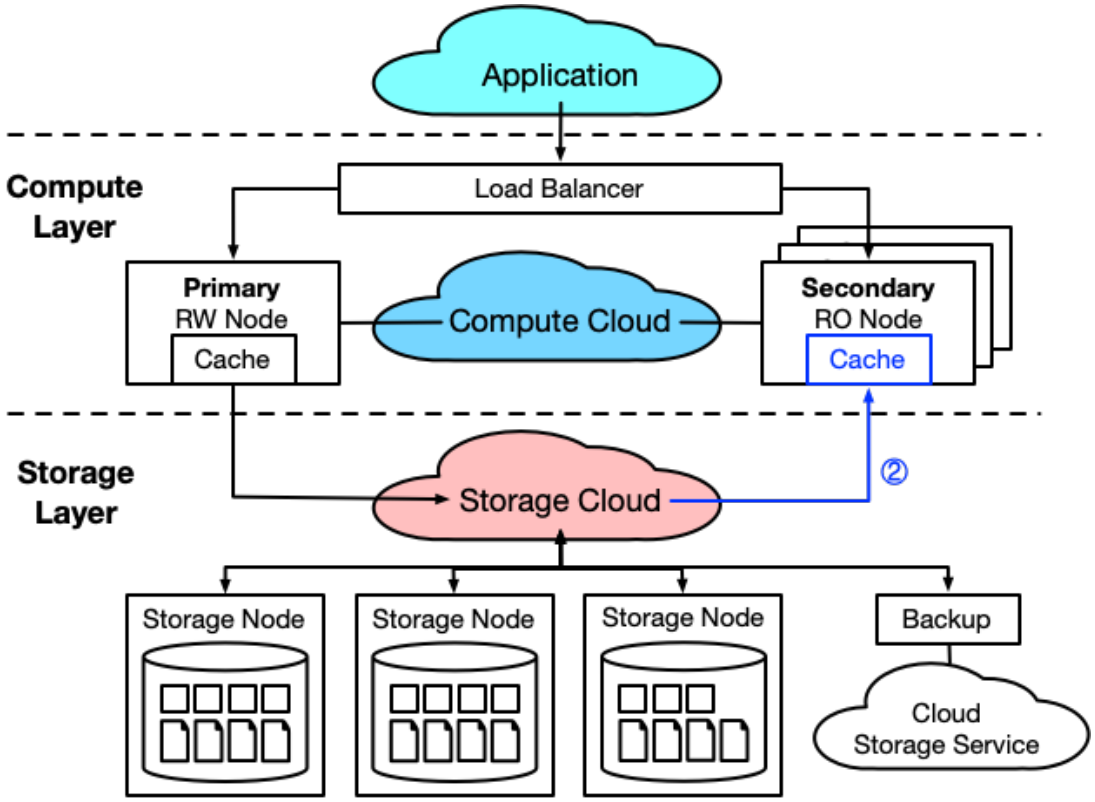
(1) Disaggregated Compute-Storage Architecture



- Data read path:

1. First, compute node will check its local cache. When data is **in the local cache and valid**, it can directly load data from the local cache and **return**.
2. When the cache misses in ①, compute node will send read requests to the storage layer.
3. For each node in the storage layer:
   - When data in the page is new, directly load data from the page.
   - Else replay the page from redo logs.
4. Compute node receives data from nodes in storage layer and runs cache replacement strategy in the local cache, then returns.

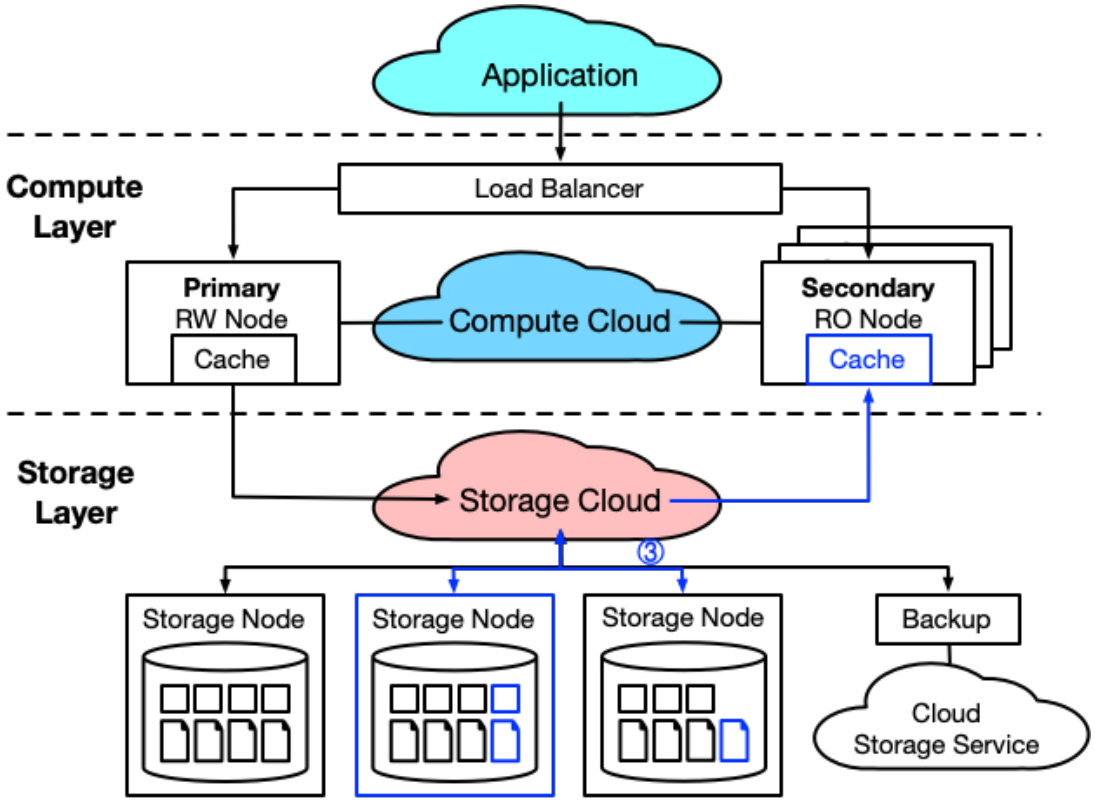
(1) Disaggregated Compute-Storage Architecture



- Data read path:

1. First, compute node will check its local cache. When data is in the local cache and valid, it can directly load data from the local cache and return.
2. When the cache misses in ①, compute node will send read requests to the storage layer.
3. For each node in the storage layer:
   - When data in the page is new, directly load data from the page.
   - Else replay the page from redo logs.
4. Compute node receives data from nodes in storage layer and runs cache replacement strategy in the local cache, then returns.

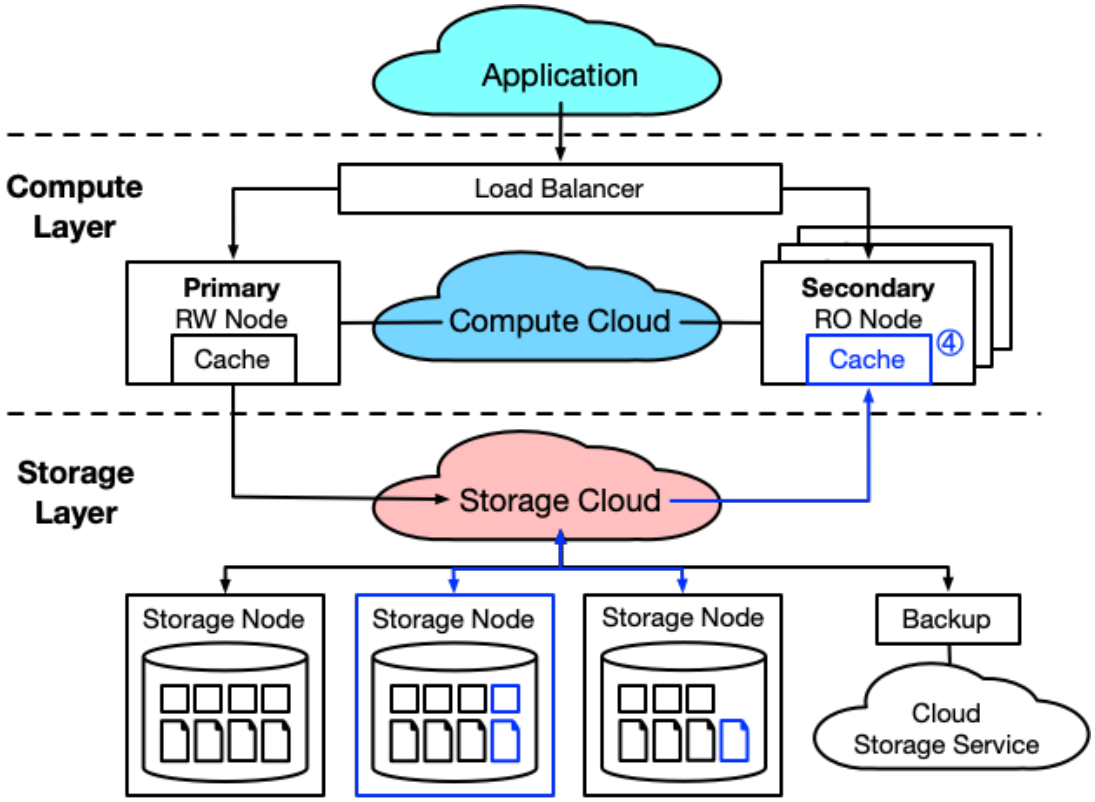
(1) Disaggregated Compute-Storage Architecture



- Data read path:

1. First, compute node will check its local cache. When data is in the local cache and valid, it can directly load data from the local cache and return.
2. When the cache misses in ①, compute node will send read requests to the storage layer.
3. For each node in the storage layer:
   - When data in the page is new, directly load data from the page.
   - Else replay the page from redo logs.
4. Compute node receives data from nodes in storage layer and runs cache replacement strategy in the local cache, then returns.

(1) Disaggregated Compute-Storage Architecture



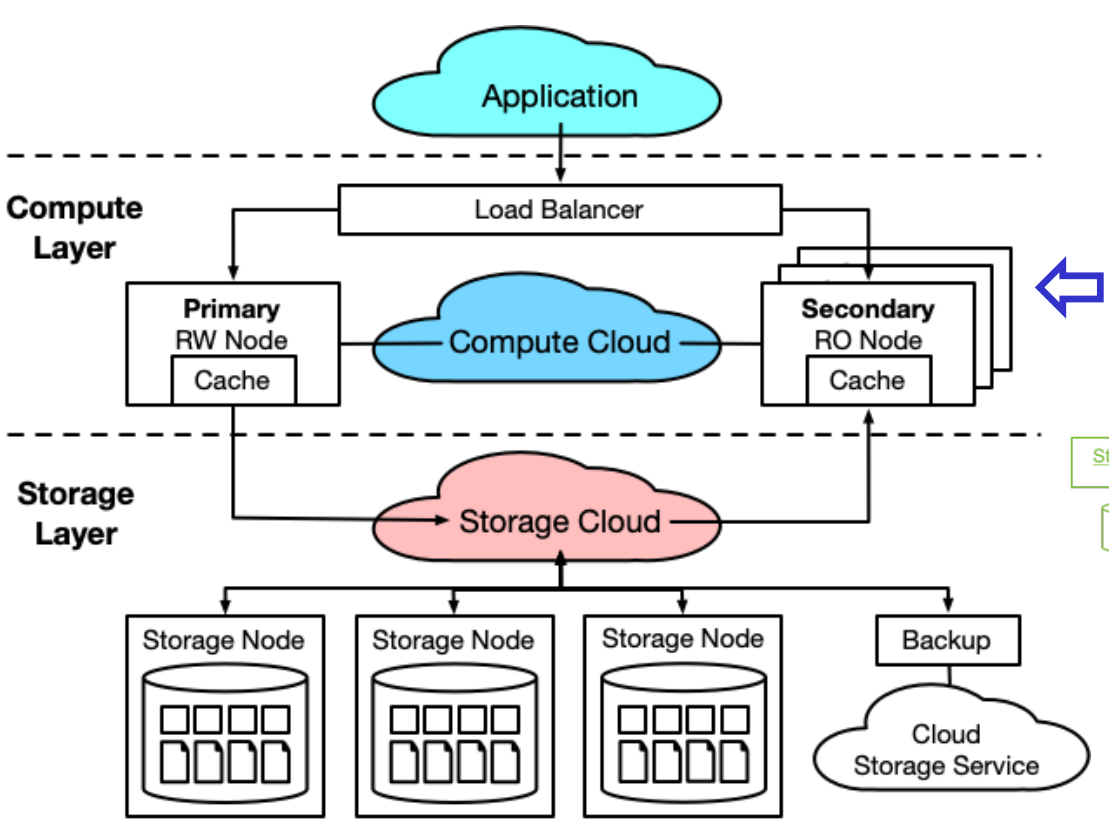
- Data read path:

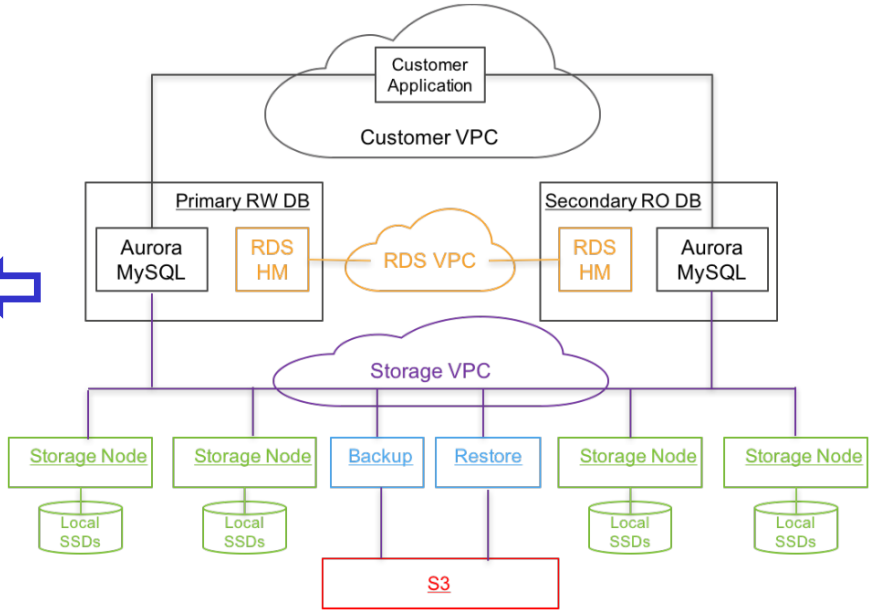
1. First, compute node will check its local cache. When data is in the local cache and valid, it can directly load data from the local cache and return.
2. When the cache misses in ①, compute node will send read requests to the storage layer.
3. For each node in the storage layer:
   - When data in the page is new, directly load data from the page.
   - Else replay the newest page from redo logs.
4. Compute node receives data from nodes in storage layer and runs cache replacement strategy in the local cache, then **returns**.

(1) Disaggregated Compute-Storage Architecture

- **Advantages compared to Traditional Architecture:**
  - **Low write latency.**
    - Write can be committed without waiting for updating pages in storage nodes.
  - **Reduce write amplification.**
    - Log replay is pushdown to the storage layer. Avoid dirty page flush during data writing.
    - Shared-storage architecture. Avoid that single instance maintains multiple storage replicas.
  - **Better elasticity.**
    - Disaggregation of compute and storage resources, which can be scheduled independently.
- **Limitations:**
  - **High read latency when cache misses.**
    - The update data may not be replayed to the page, leading to extra read latency for log replaying.

(1) Disaggregated Compute-Storage Architecture

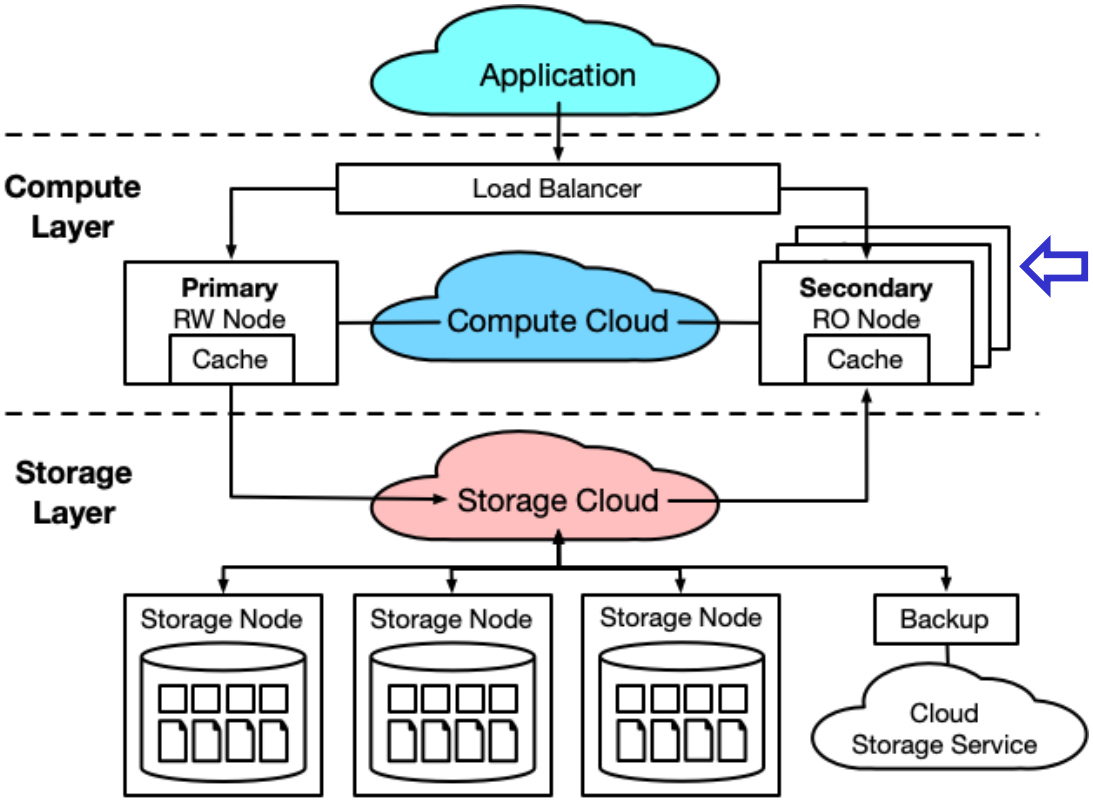


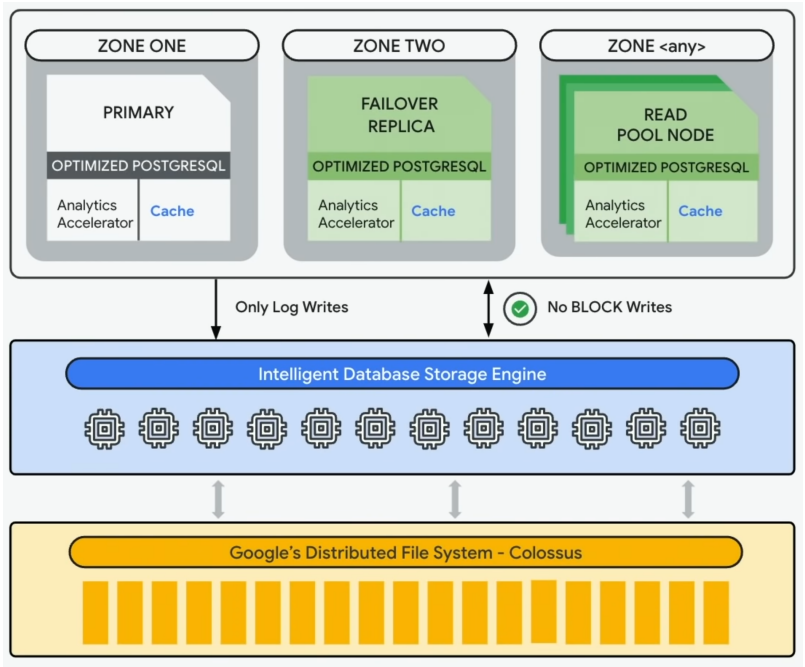


- Examples:
  - AWS Aurora

Verbitski, Alexandre, et al. Amazon aurora: Design considerations for high throughput cloud-native relational databases. SIGMOD, 2017.

(1) Disaggregated Compute-Storage Architecture

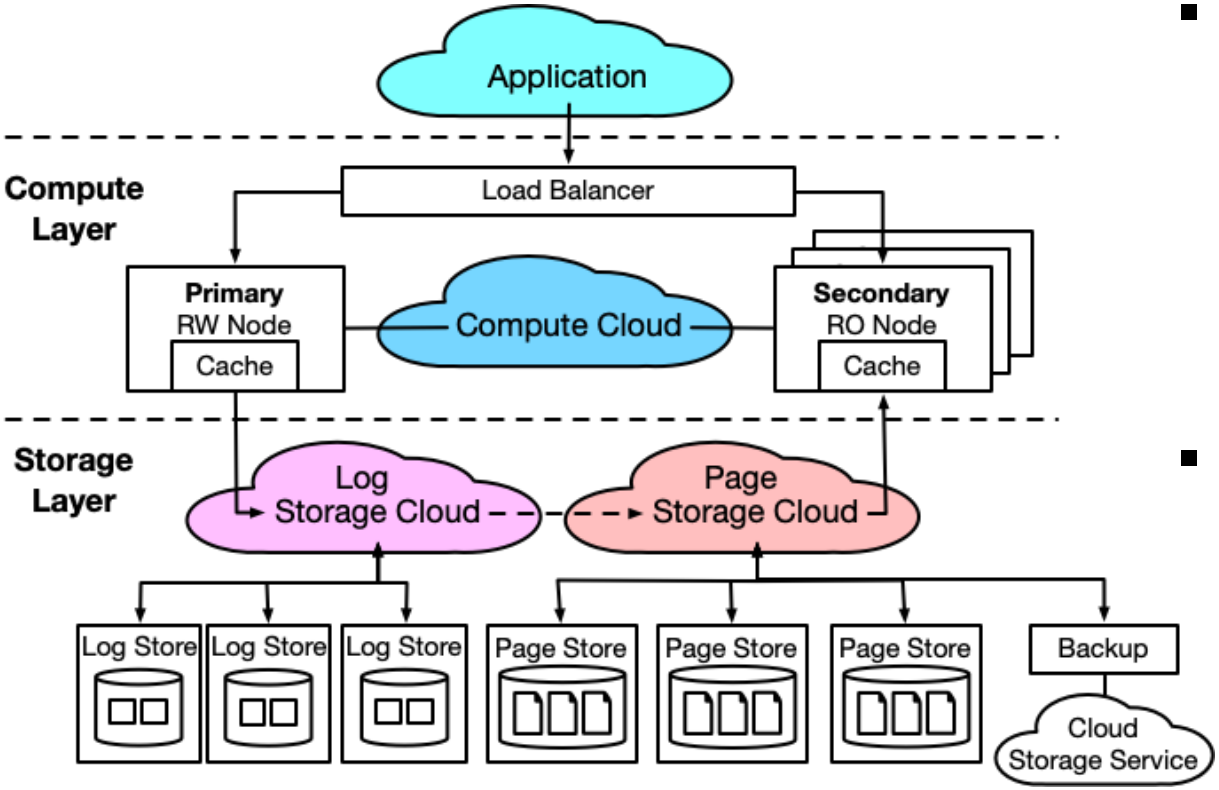




- Examples:
  - AWS Aurora
  - GCP AlloyDB

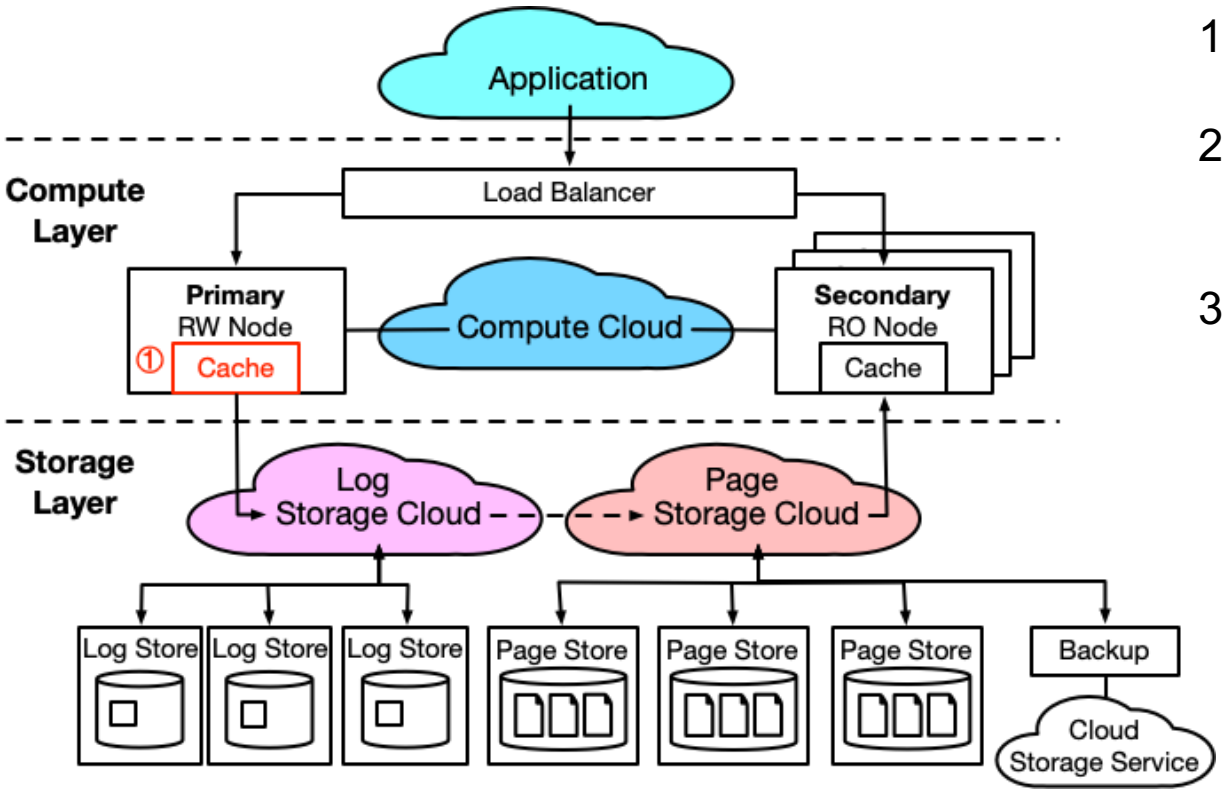
Figure from https://cloud.google.com/blog/products/databases/alloydb-for-postgresql-columnar-engine

(2) Disaggregated Compute-Log-Storage Architecture



- Motivations:
  - **Efficiency.** Better write performance with faster Log storage service.
  - **Elasticity.** The scaling of Log & Page Storage is independent of each other.
- Key Features:
  - Disaggregation of Log & Page Storage.
  - Different features of Log & Storage services: fast Log Store & cheap Page Store.

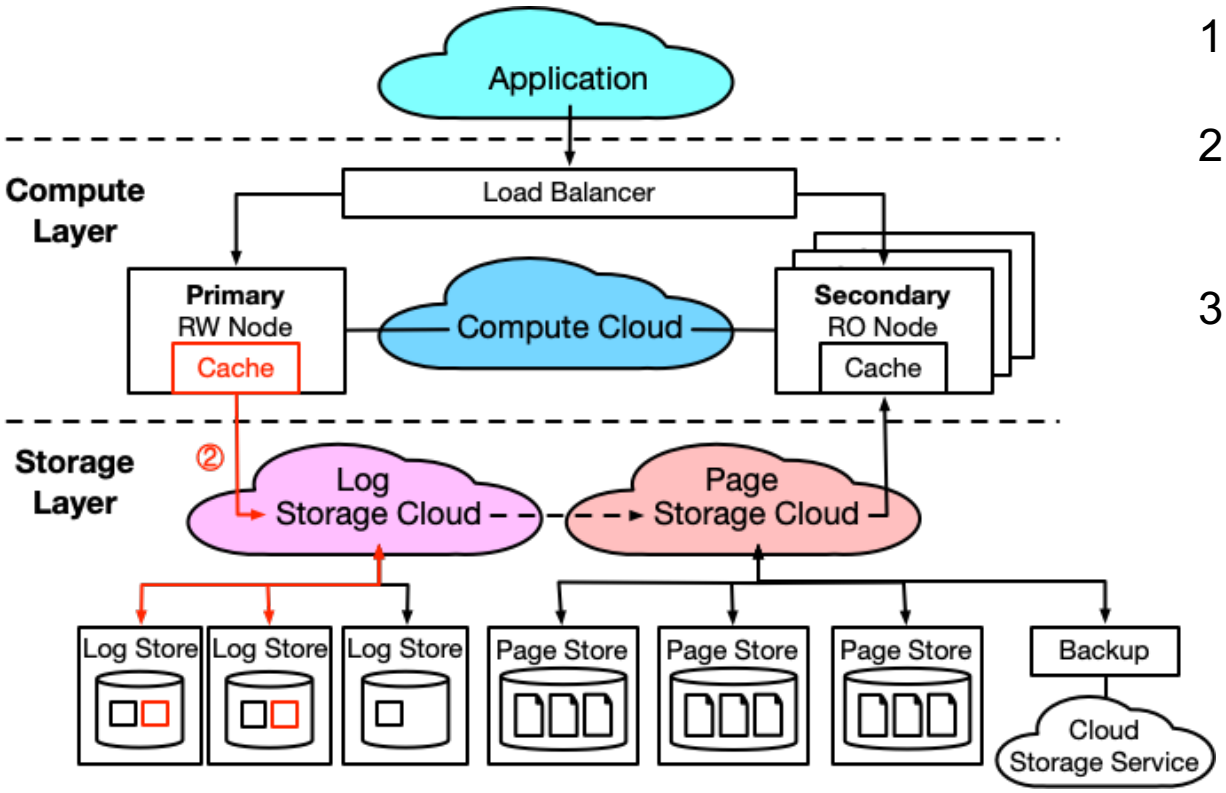
(2) Disaggregated Compute-Log-Storage Architecture



- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. Primary node writes redo log to majority nodes in the log storage cloud.
3. Commit the write after the majority of log replicas finish the log writing process for durability.

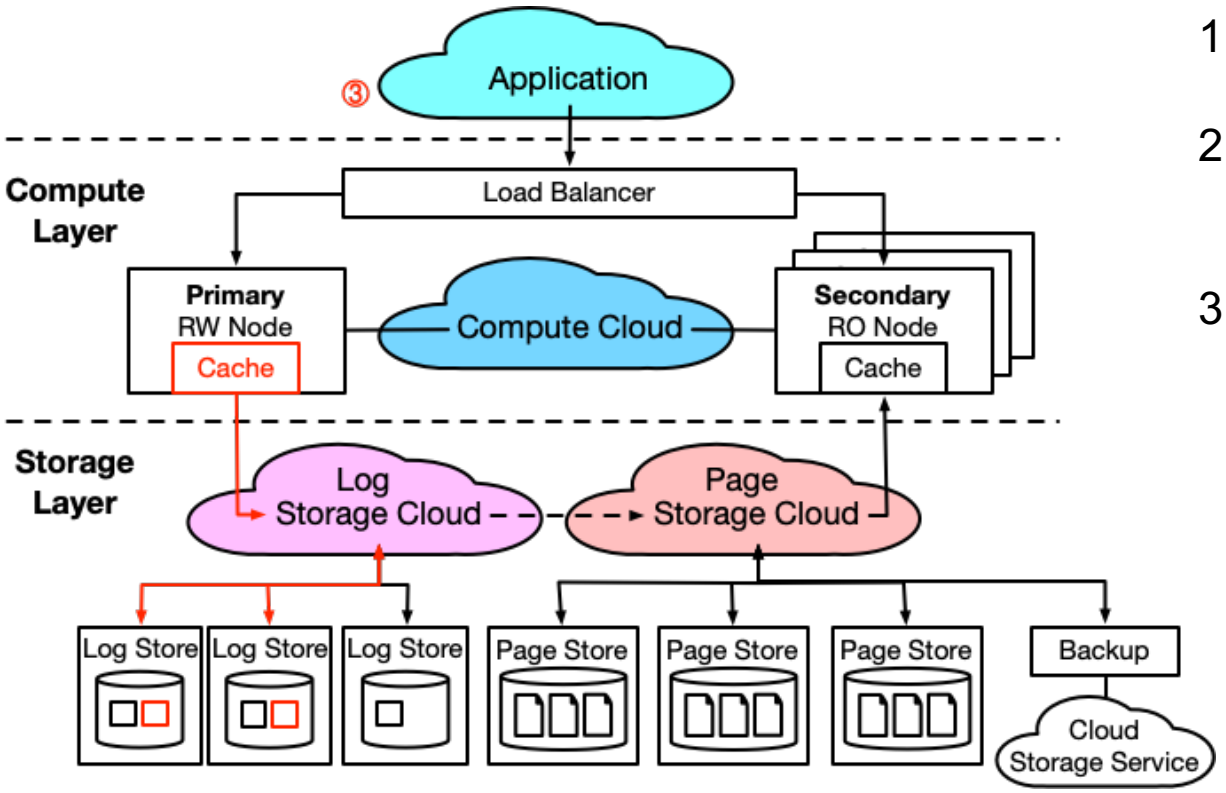
(2) Disaggregated Compute-Log-Storage Architecture



- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. Primary node writes redo log to majority nodes in the **log storage cloud**.
3. Commit the write after the majority of log replicas finish the log writing process for durability.

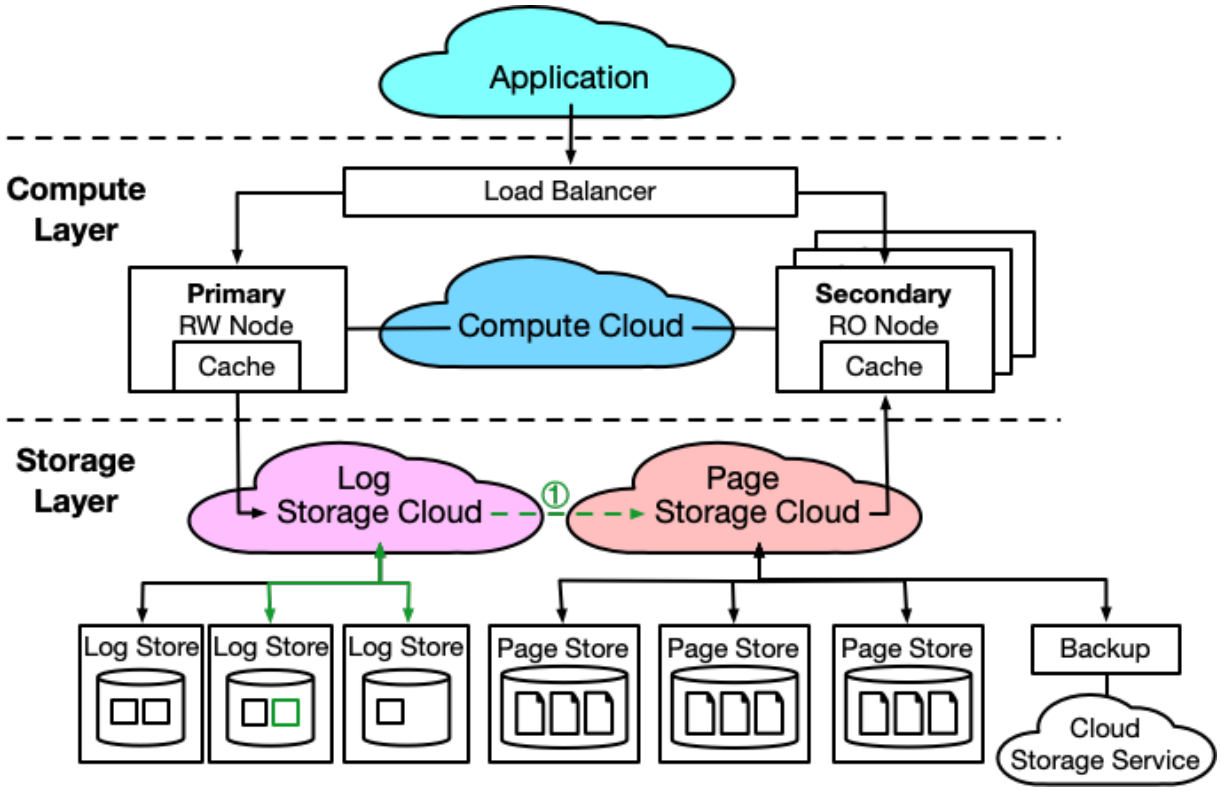
(2) Disaggregated Compute-Log-Storage Architecture



- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. Primary node writes redo log to majority nodes in the log storage cloud.
3. Commit the write after the majority of **log replicas** finish the log writing process for durability.

(2) Disaggregated Compute-Log-Storage Architecture



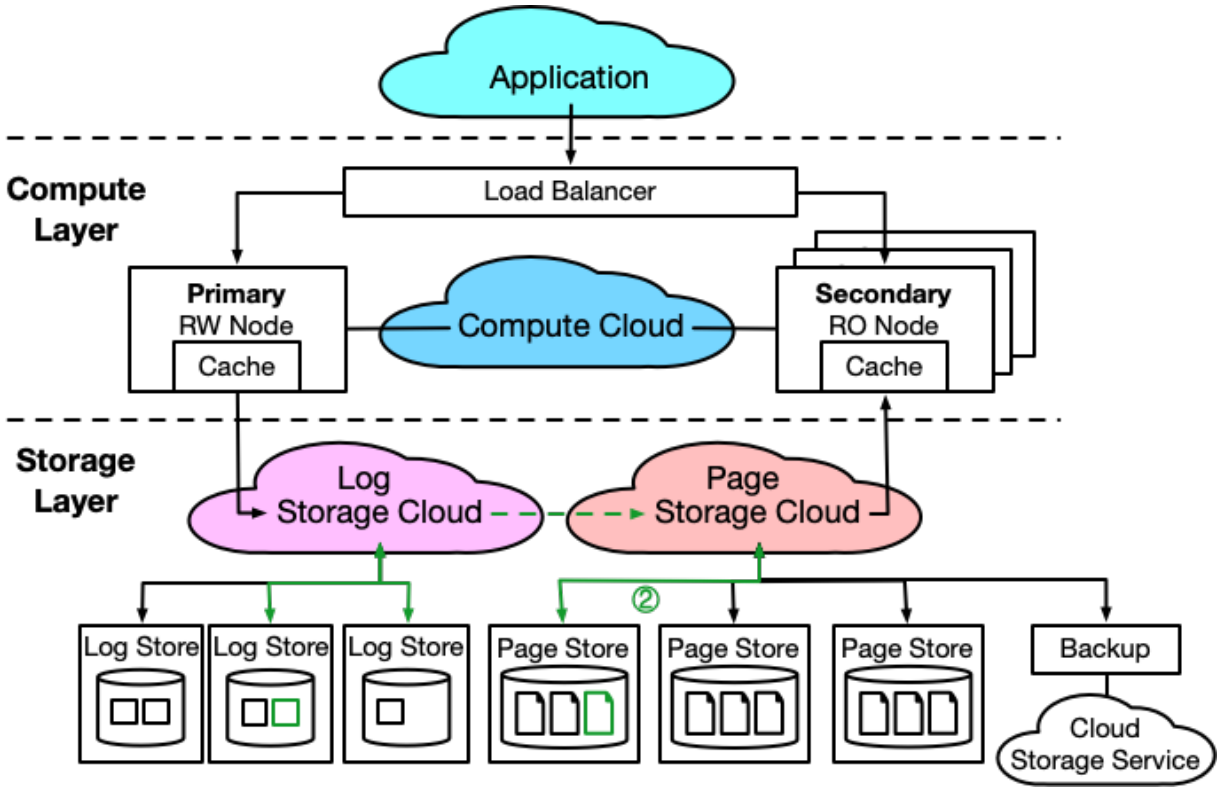
- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. Primary node writes redo log to majority nodes in the log storage cloud.
3. Commit the write after the majority of log replicas finish the log writing process for durability.

- Data sync path:

1. Read redo logs from the log storage and send them to the page storage.
2. Redo logs are replayed to page storage nodes asynchronously.
3. Different page storage nodes maintain consistency with the gossip protocol.
4. When redo logs are replayed in all page storage nodes, the log storage can be truncated.
5. Directly transfer logs to secondary nodes to reduce the update latency.

(2) Disaggregated Compute-Log-Storage Architecture



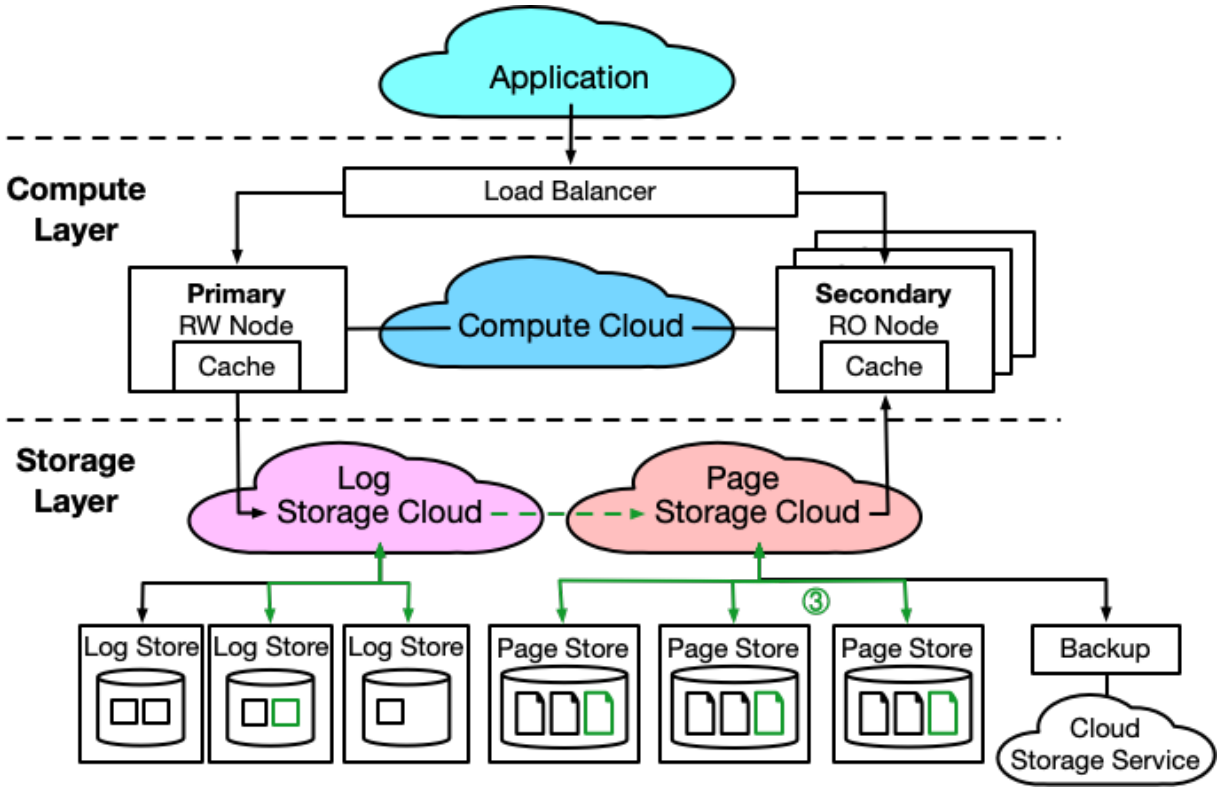
- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. Primary node writes redo log to majority nodes in the log storage cloud.
3. Commit the write after the majority of log replicas finish the log writing process for durability.

- Data sync path:

1. Read redo logs from the log storage and send them to the page storage.
2. Redo logs are replayed to page storage nodes **asynchronously**.
3. Different page storage nodes maintain consistency with the gossip protocol.
4. When redo logs are replayed in all page storage nodes, the log storage can be truncated.
5. Directly transfer logs to secondary nodes to reduce the update latency.

(2) Disaggregated Compute-Log-Storage Architecture



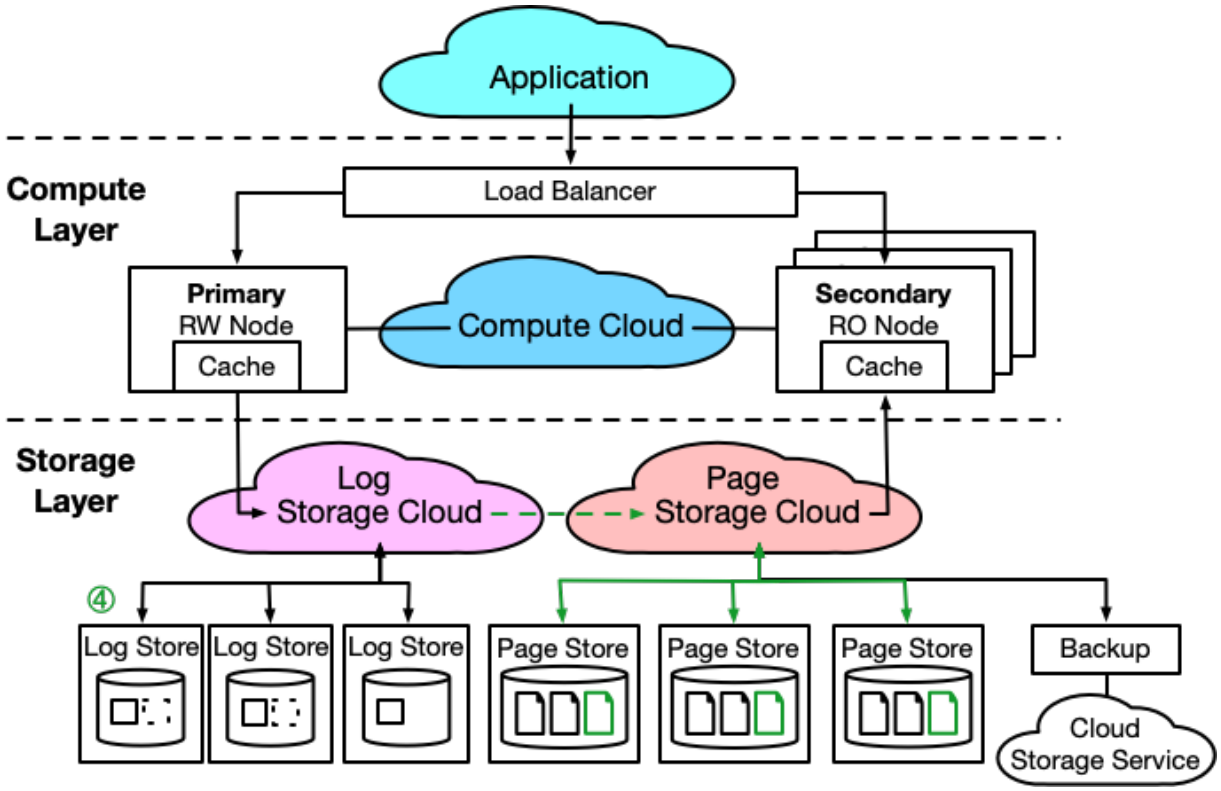
- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. Primary node writes redo log to majority nodes in the log storage cloud.
3. Commit the write after the majority of log replicas finish the log writing process for durability.

- Data sync path:

1. Read redo logs from the log storage and send them to the page storage.
2. Redo logs are replayed to page storage nodes asynchronously.
3. Different page storage nodes **maintain consistency with the gossip protocol**.
4. When redo logs are replayed in all page storage nodes, the log storage can be truncated.
5. Directly transfer logs to secondary nodes to reduce the update latency.

(2) Disaggregated Compute-Log-Storage Architecture



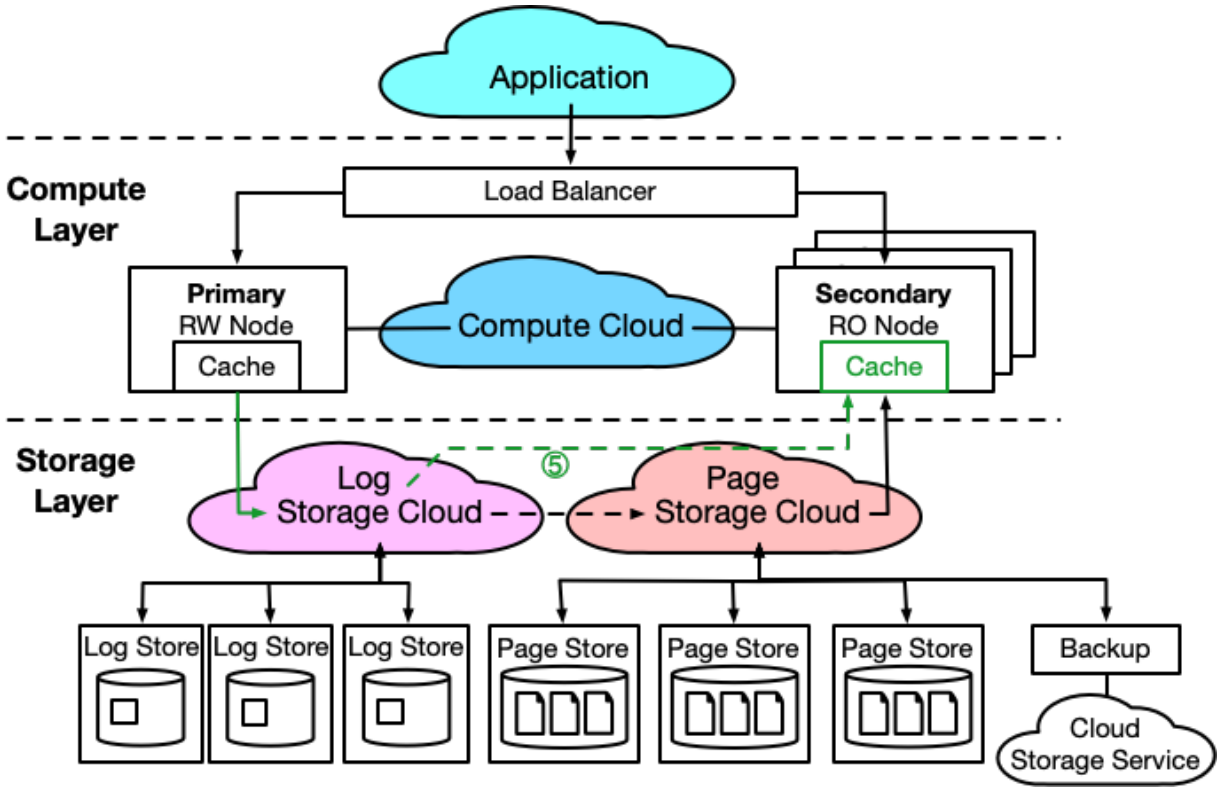
- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. Primary node writes redo log to majority nodes in the log storage cloud.
3. Commit the write after the majority of log replicas finish the log writing process for durability.

- Data sync path:

1. Read redo logs from the log storage and send them to the page storage.
2. Redo logs are replayed to page storage nodes asynchronously.
3. Different page storage nodes maintain consistency with the gossip protocol.
4. When redo logs are replayed in all page storage nodes, **the log storage can be truncated**.
5. Directly transfer logs to secondary nodes to reduce the update latency.

(2) Disaggregated Compute-Log-Storage Architecture



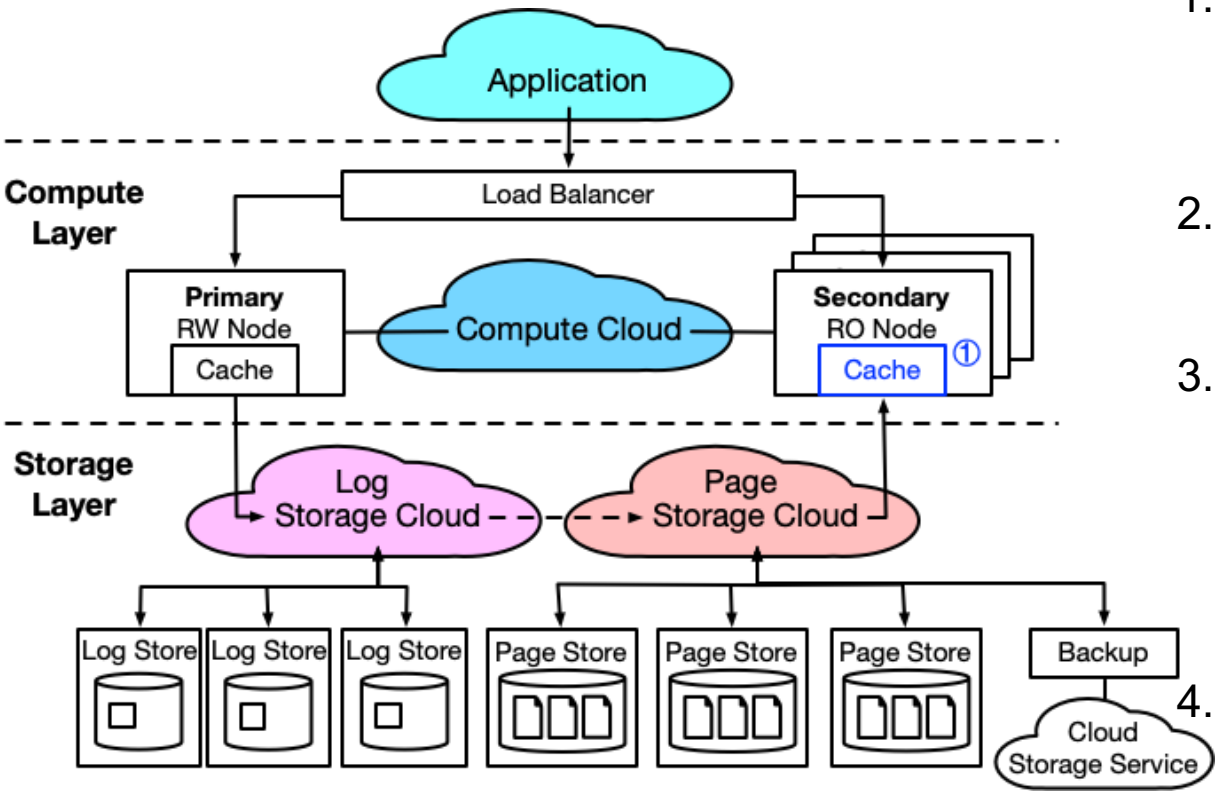
- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. Primary node writes redo log to majority nodes in the log storage cloud.
3. Commit the write after the majority of log replicas finish the log writing process for durability.

- Data sync path:

1. Read redo logs from the log storage and send them to the page storage.
2. Redo logs are replayed to page storage nodes asynchronously.
3. Different page storage nodes maintain consistency with the gossip protocol.
4. When redo logs are replayed in all page storage nodes, the log storage can be truncated.
5. Directly transfer logs to secondary nodes to reduce the update latency.

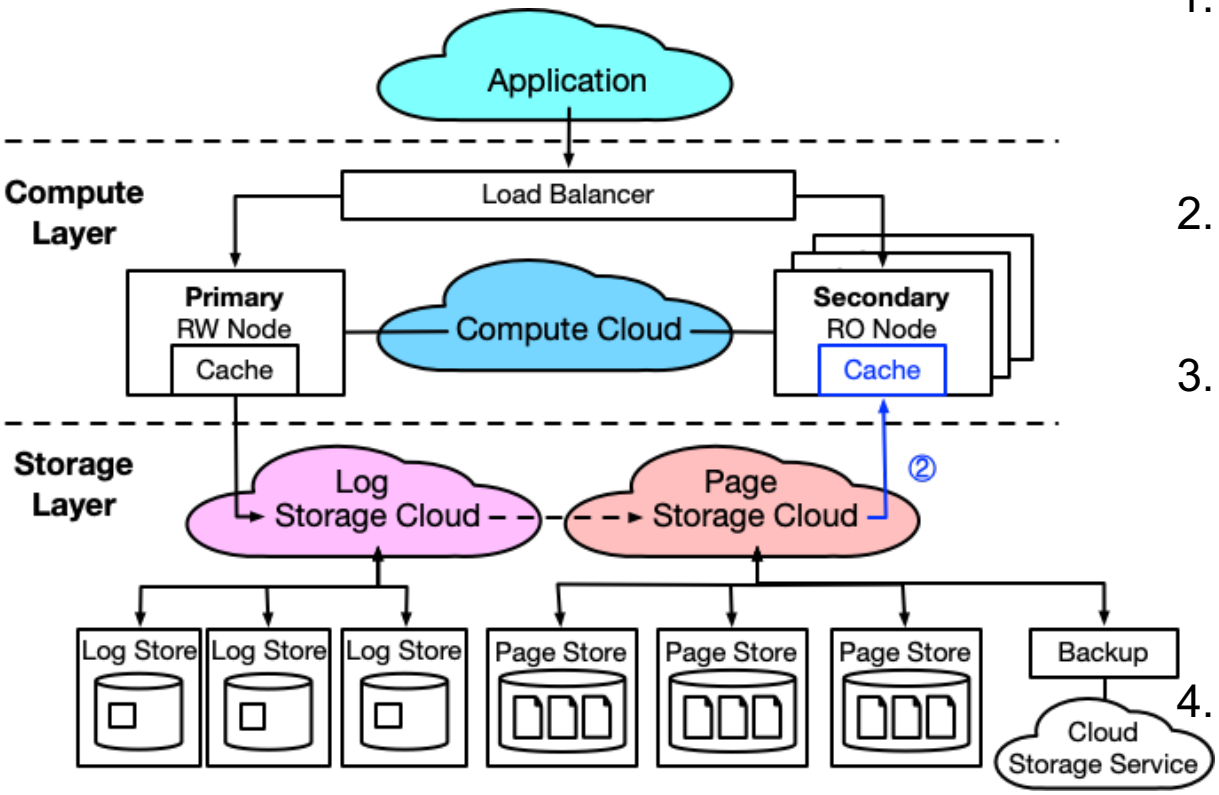
(2) Disaggregated Compute-Log-Storage Architecture



- Data read path:

1. First, compute node will check its local cache. When data is in the local cache and valid, it can directly load data from the local cache and **return**.
2. When the cache misses in ①, compute node will send read requests to the page storage cloud.
3. For each node in the page storage cloud:
   - When data in the page is new, directly load data from the page.
   - Else, wait for the sync process until the data is new.
4. Compute node receives data from page storage cloud and runs cache replacement strategy in the local cache, then returns.

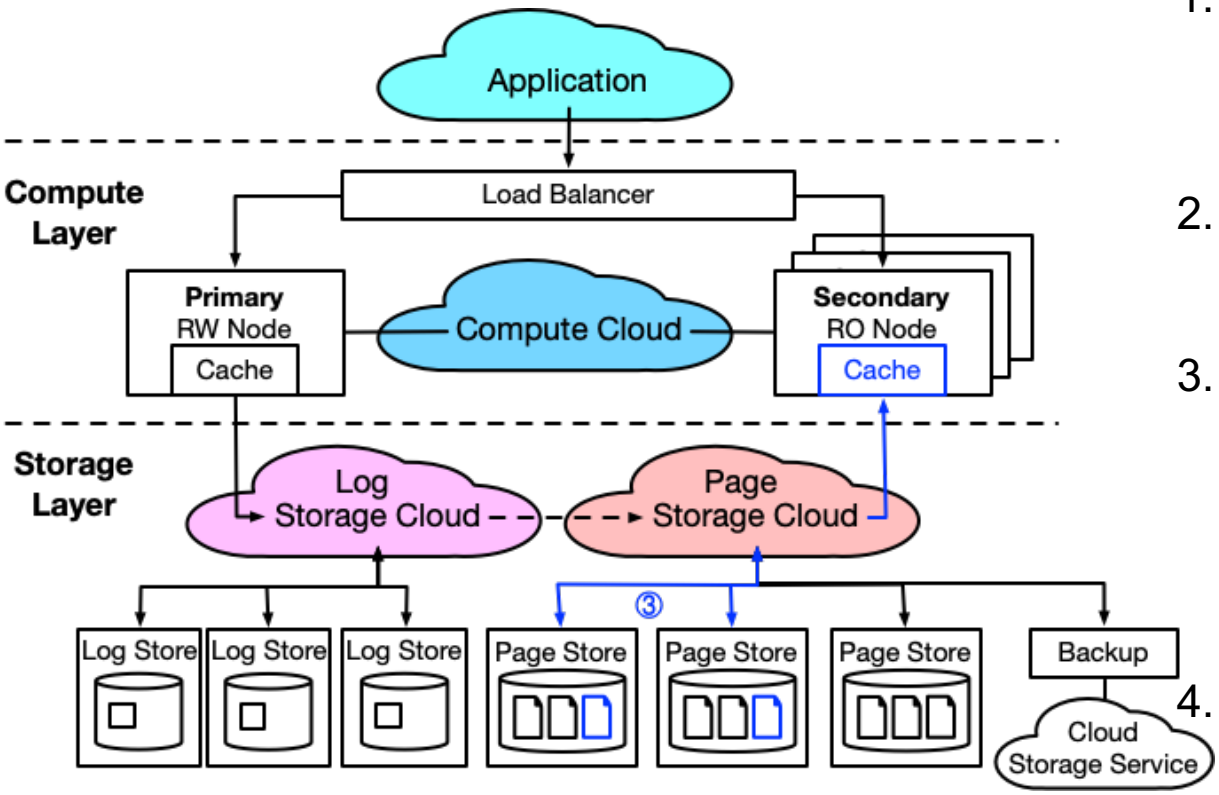
(2) Disaggregated Compute-Log-Storage Architecture



- Data read path:

1. First, compute node will check its local cache. When data is in the local cache and valid, it can directly load data from the local cache and return.
2. When the cache misses in ①, compute node will send read requests to the **page storage cloud**.
3. For each node in the page storage cloud:
   - When data in the page is new, directly load data from the page.
   - Else, wait for the sync process until the data is new.
4. Compute node receives data from page storage cloud and runs cache replacement strategy in the local cache, then returns.

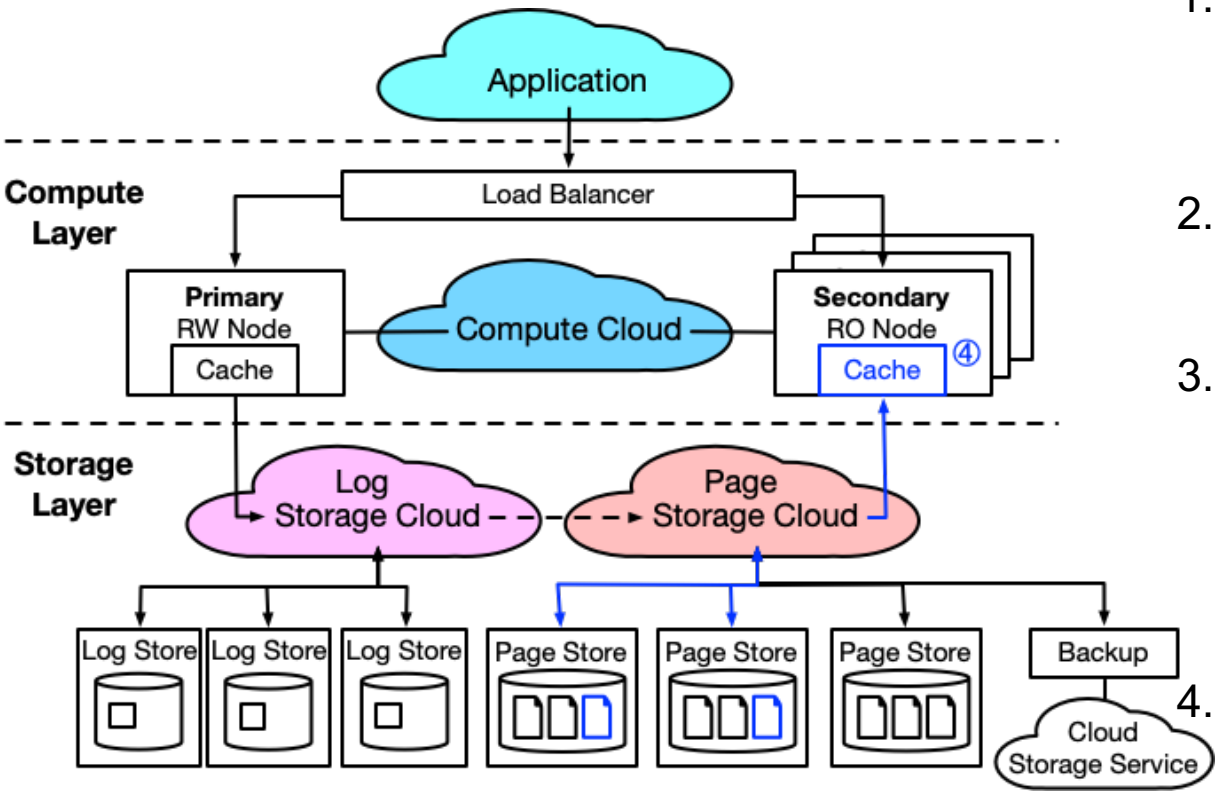
(2) Disaggregated Compute-Log-Storage Architecture



- Data read path:

1. First, compute node will check its local cache. When data is in the local cache and valid, it can directly load data from the local cache and return.
2. When the cache misses in ①, compute node will send read requests to the page storage cloud.
3. For each node in the page storage cloud:
   - When data in the page is new, directly load data from the page.
   - Else, **wait for the sync process** until the data is new.
4. Compute node receives data from page storage cloud and runs cache replacement strategy in the local cache, then returns.

(2) Disaggregated Compute-Log-Storage Architecture



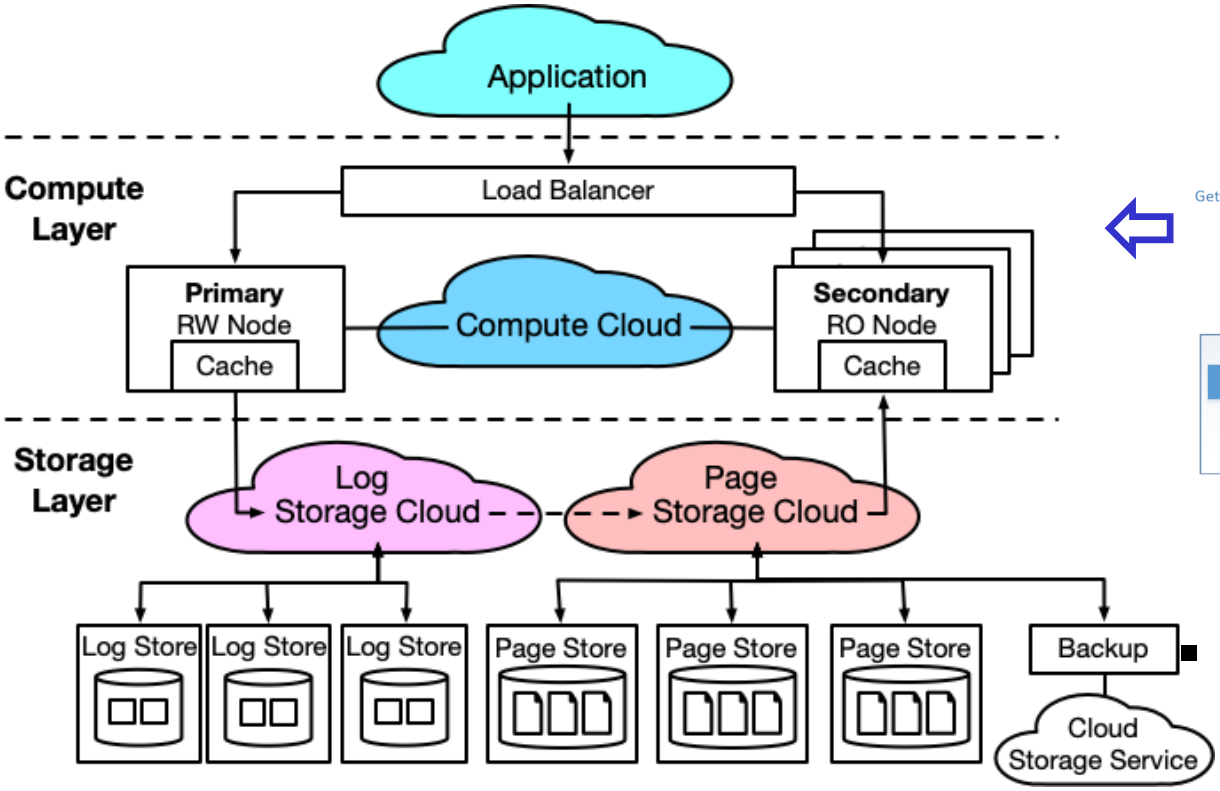
- Data read path:

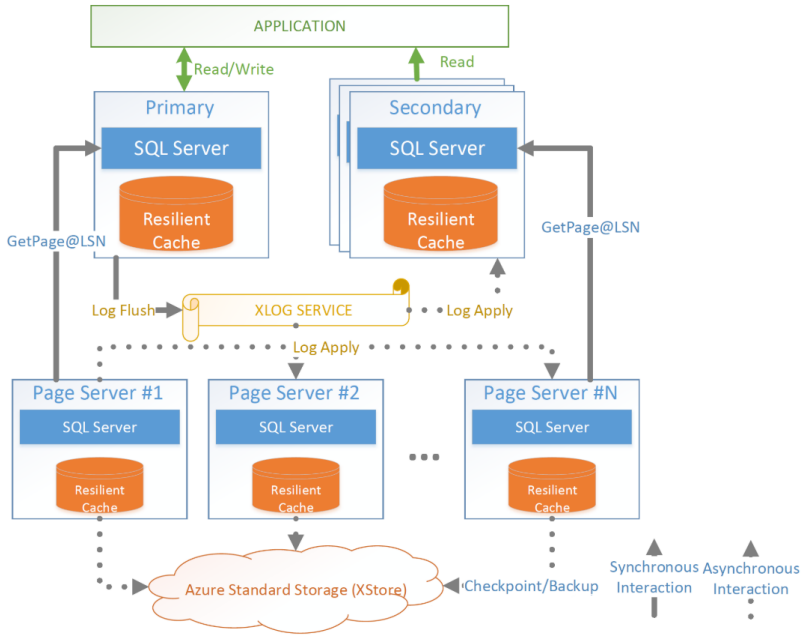
1. First, compute node will check its local cache. When data is in the local cache and valid, it can directly load data from the local cache and return.
2. When the cache misses in ①, compute node will send read requests to the page storage cloud.
3. For each node in the page storage cloud:
   - When data in the page is new, directly load data from the page.
   - Else, wait for the sync process until the data is new.
4. Compute node receives data from page storage cloud and runs cache replacement strategy in the local cache, then **returns**.

(2) Disaggregated Compute-Log-Storage Architecture

- **Advantages:**
  - **Low write latency.**
    - With the fast log storage service, write can be committed faster compared with disaggregated compute-storage architecture.
  - **Better elasticity.**
    - The log and page storage can be scheduled independently, achieving a balance between the cost and the performance.
- **Limitations:**
  - **High read latency when cache misses.**
    - The queries in computing nodes must wait for the log replay when the cache misses.
  - **More complex recovery algorithm.**
    - Data may be recovered from log storage, which requires a complex mechanism.

(2) Disaggregated Compute-Log-Storage Architecture

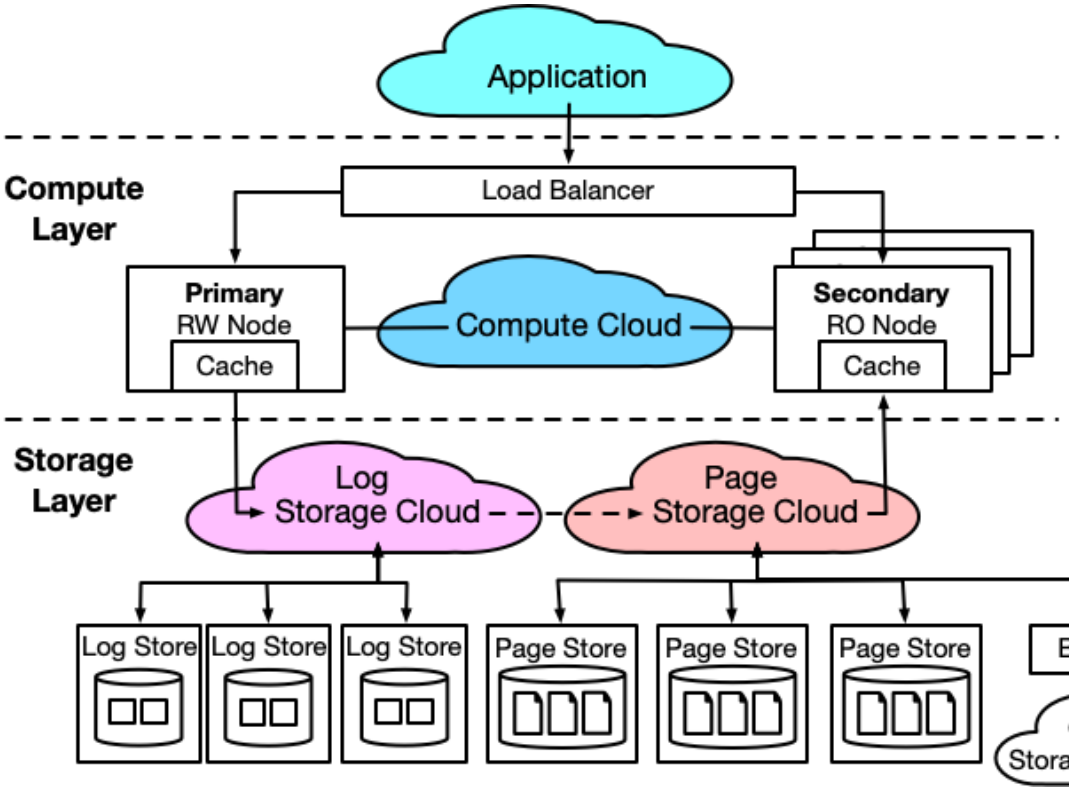


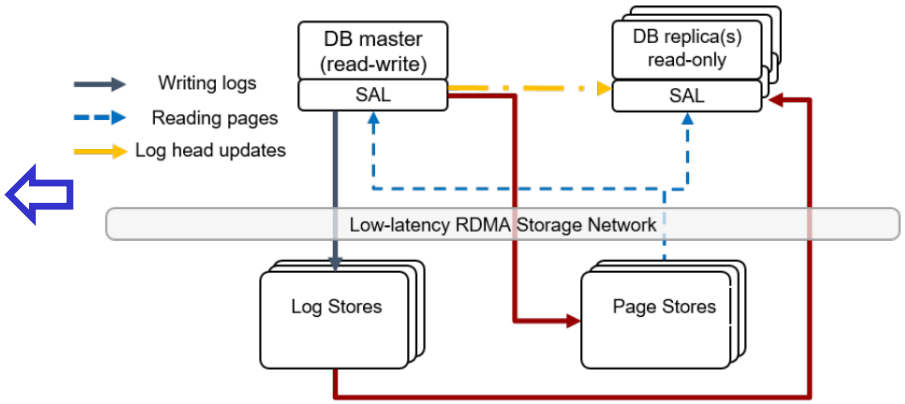


- Examples:
  - Azure HyperScale(Socrates)

Antonopoulos, Panagiotis, et al. Socrates: The new sql server in the cloud. SIGMOD, 2019.

(2) Disaggregated Compute-Log-Storage Architecture

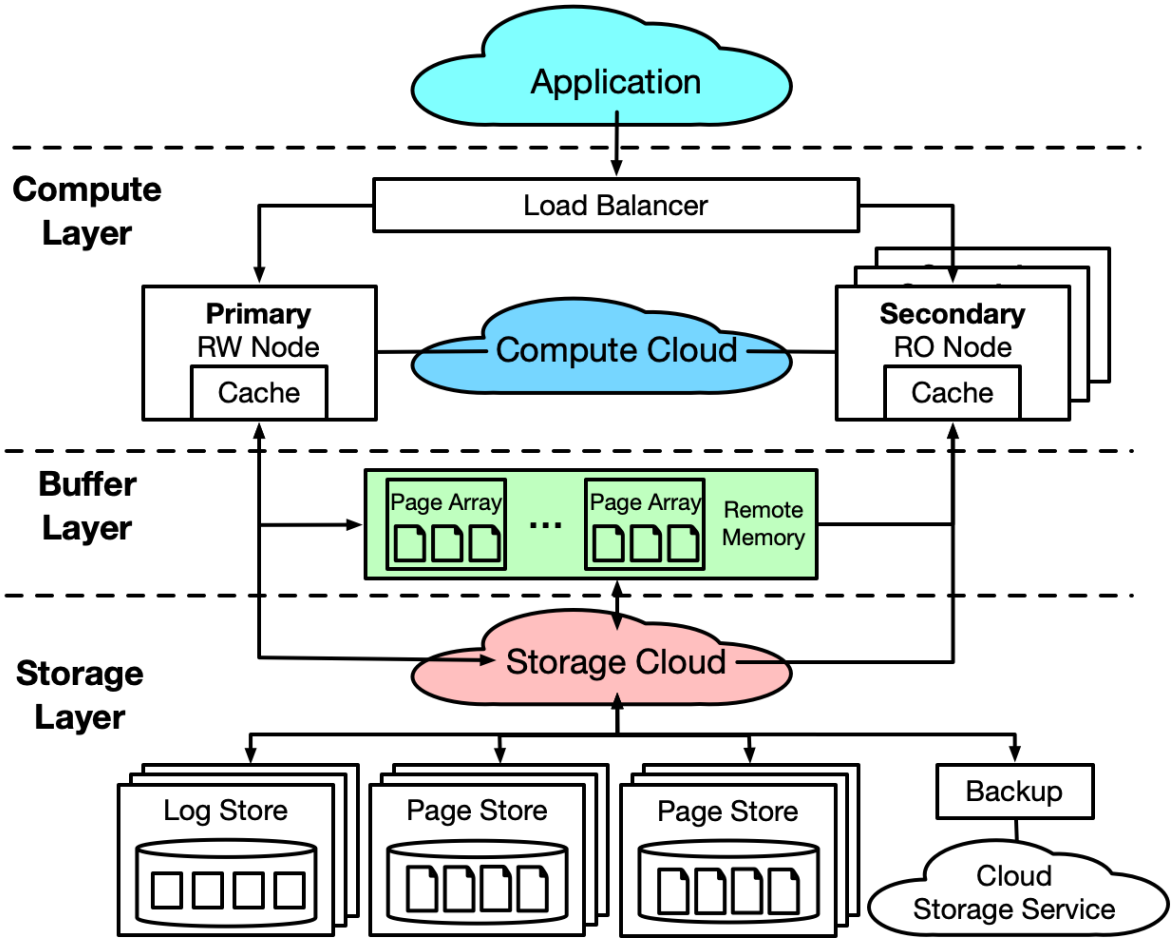




- Examples:
  - Azure HyperScale
  - HUAWEI Taurus

Depoutovitch, Alex, et al. Taurus database: How to be fast, available, and frugal in the cloud. SIGMOD, 2020.

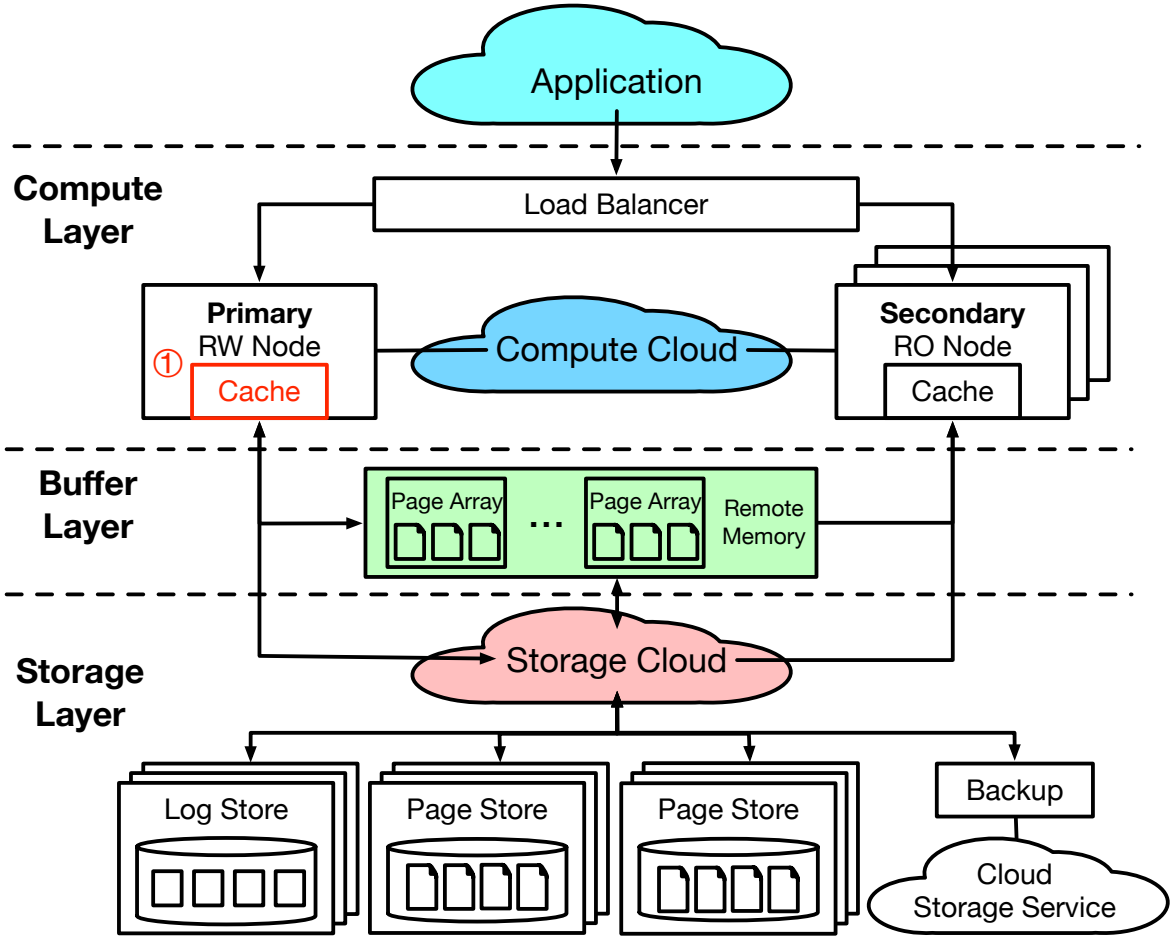
(3) Disaggregated Compute-Buffer-Storage Architecture



- Motivations:
  - **Latency.** Reduce read latency with shared remote memory.
  - **Throughput.** Reduce duplicate data loading process of different compute nodes.
  - **Elasticity.** The memory resources can be dynamically allocated on demand.

- Key Features:
  - Elastic shared remote buffer of all compute nodes(RW node & RO nodes)

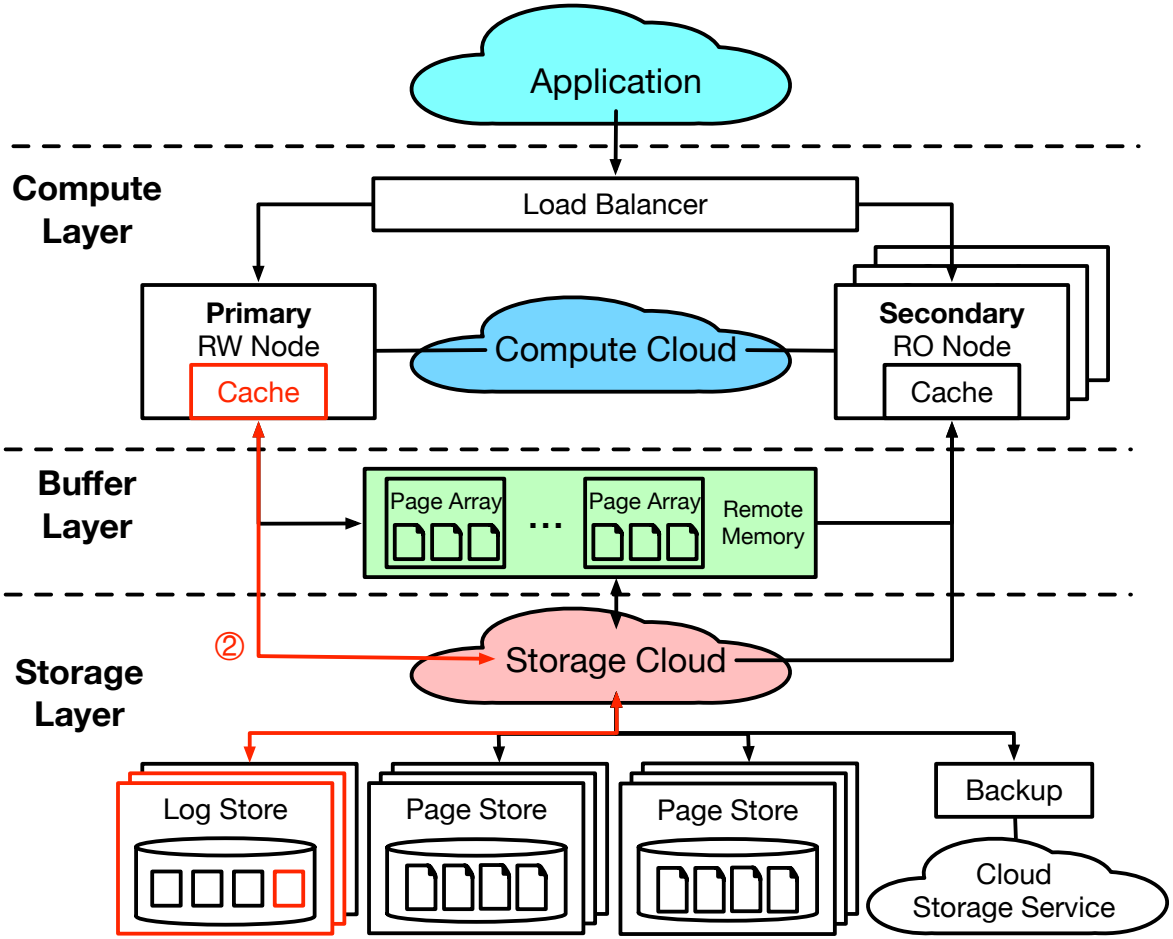
(3) Disaggregated Compute-Buffer-Storage Architecture



- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. The redo log writes to log storage(multi-replicas) for durability.
3. Commit the write after the redo log is durable.
4. The corresponding page in the shared buffer will be updated simultaneously.

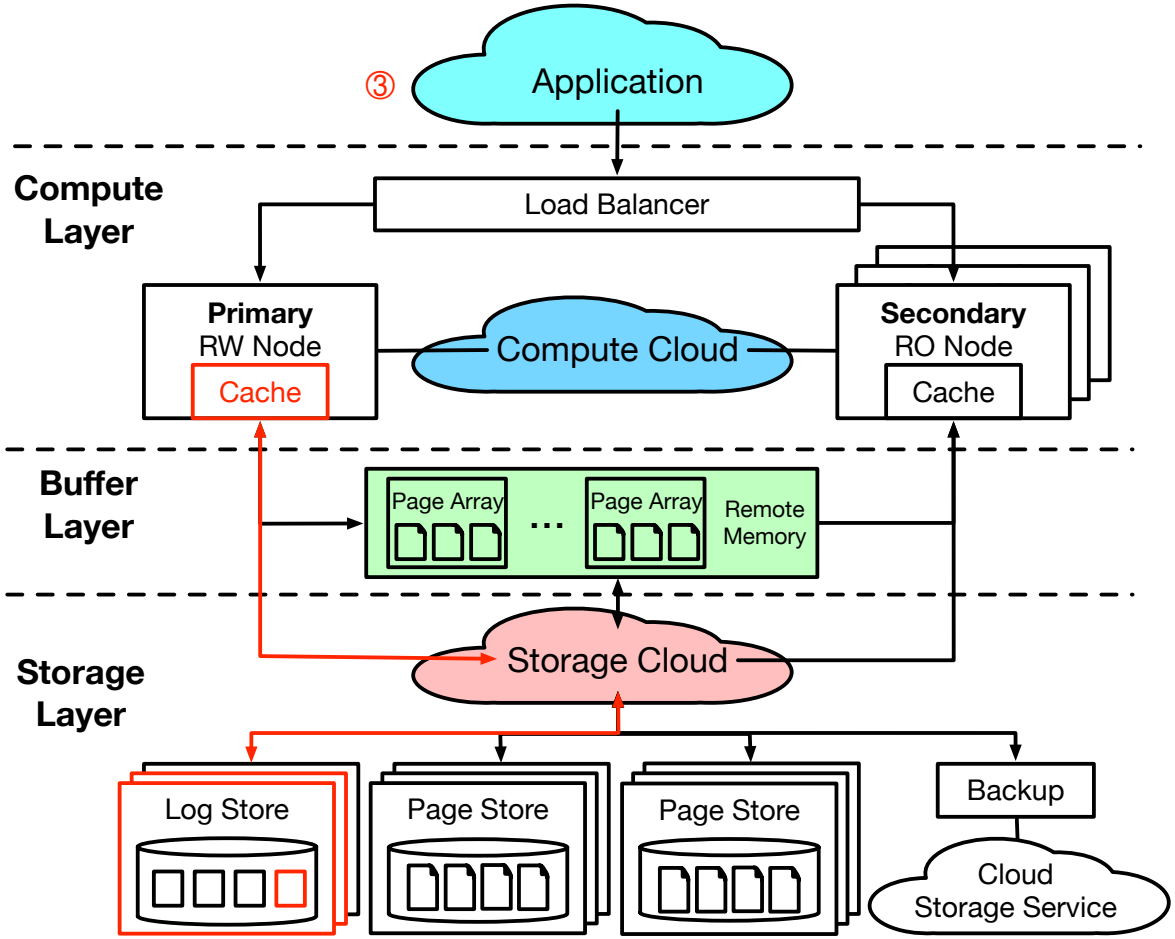
(3) Disaggregated Compute-Buffer-Storage Architecture



- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. The redo log writes to **log storage(multi-replicas)** for durability.
3. Commit the write after the redo log is durable.
4. The corresponding page in the shared buffer will be updated simultaneously.

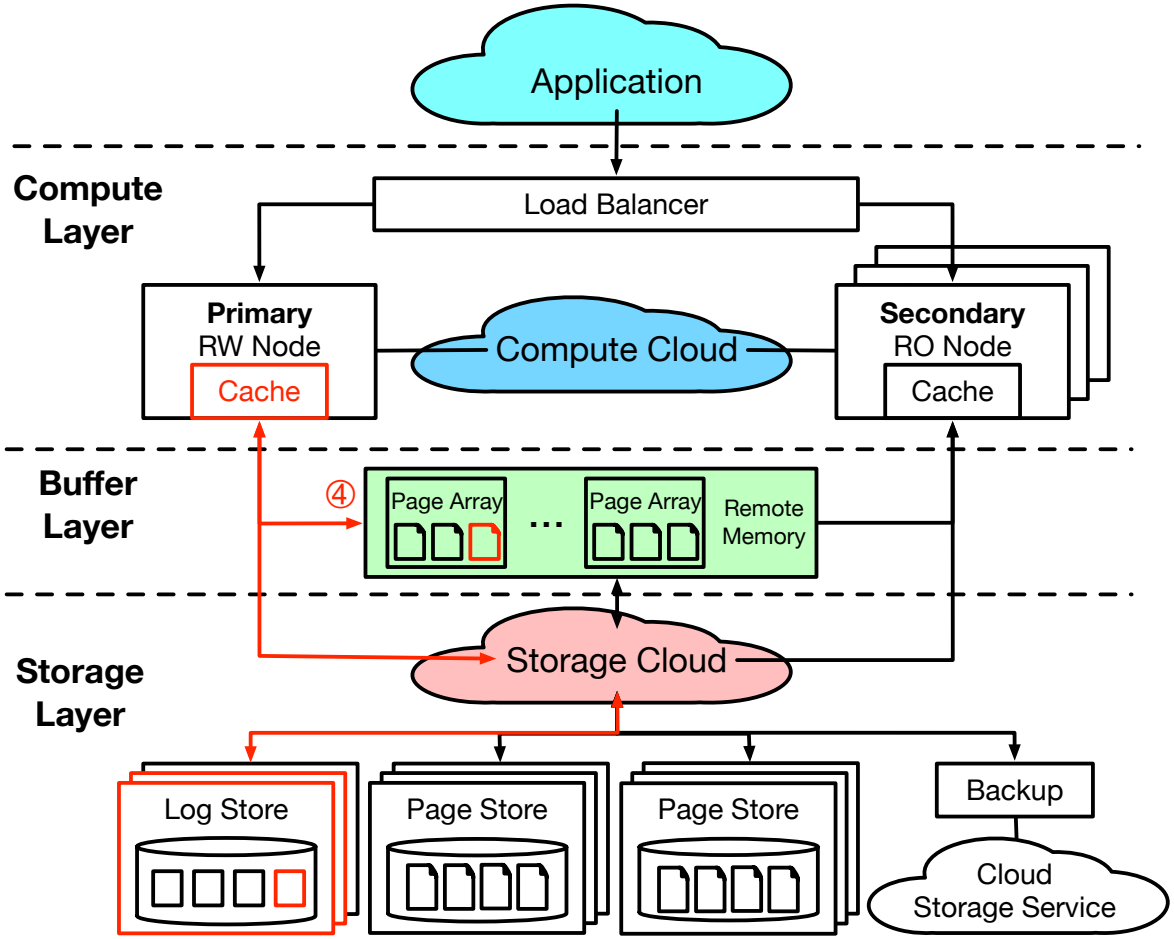
(3) Disaggregated Compute-Buffer-Storage Architecture



- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. The redo log writes to log storage(multi-replicas) for durability.
3. Commit the write after **the redo log is durable.**
4. The corresponding page in the shared buffer will be updated simultaneously.

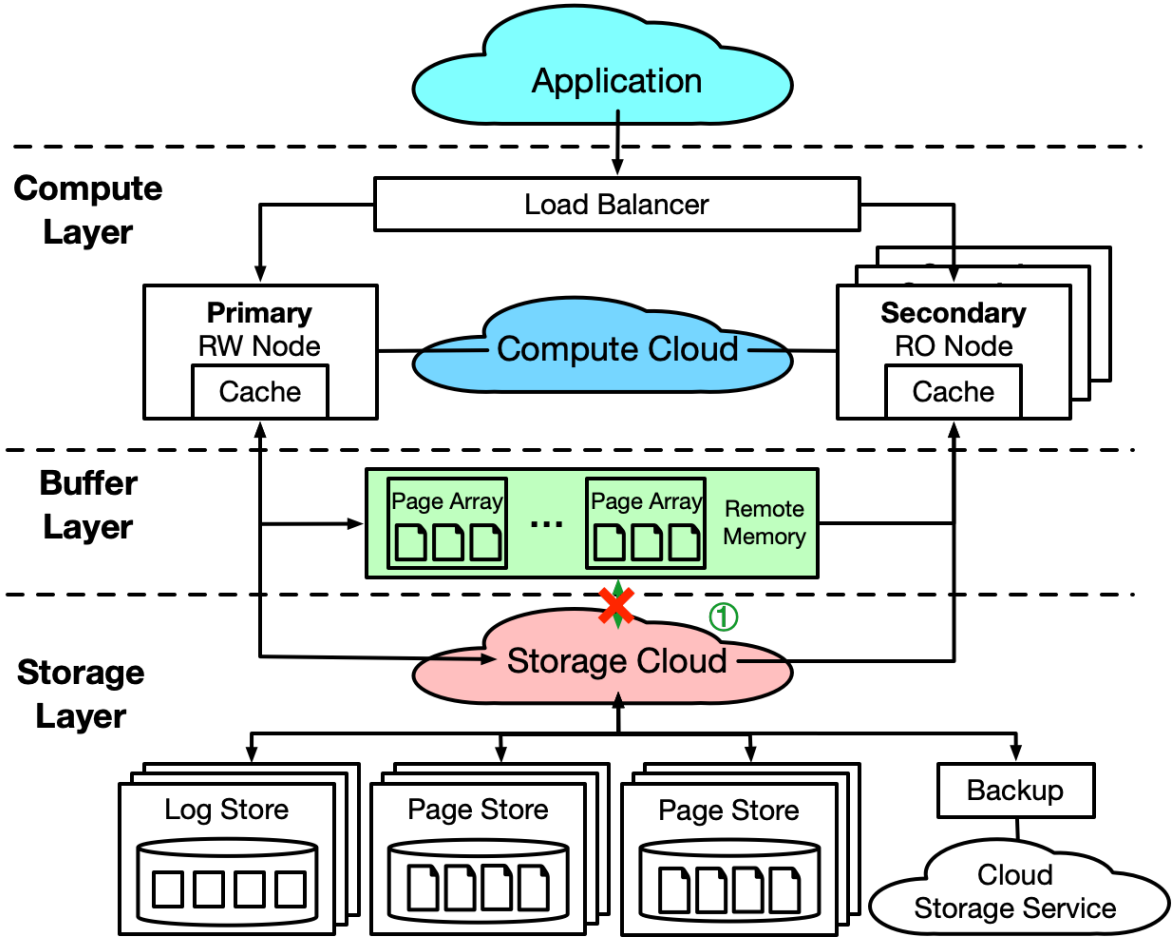
(3) Disaggregated Compute-Buffer-Storage Architecture



- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. The redo log writes to log storage(multi-replicas) for durability.
3. Commit the write after the redo log is durable.
4. The corresponding page in **the shared buffer** will be updated simultaneously.

(3) Disaggregated Compute-Buffer-Storage Architecture



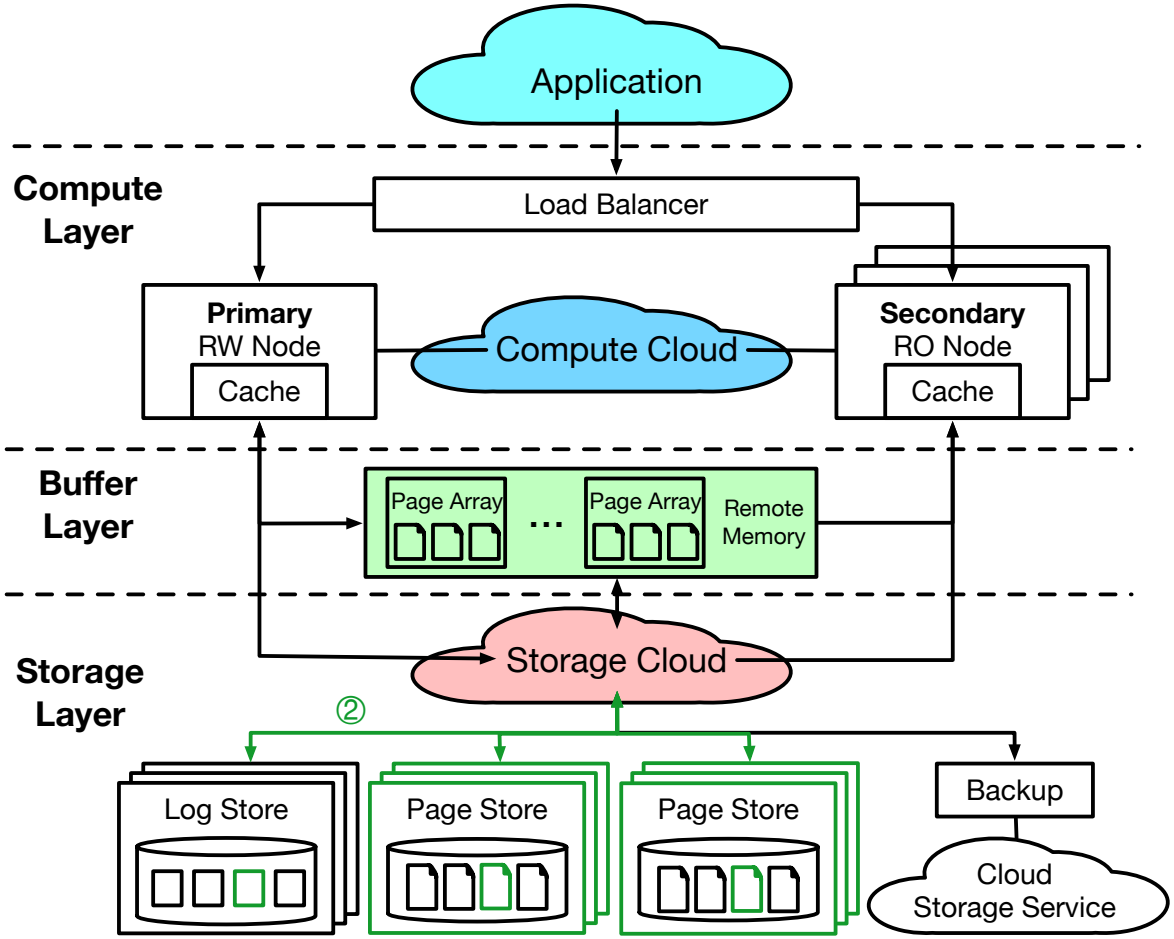
- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. The redo log writes to log storage(multi-replicas) for durability.
3. Commit the write after the redo log is durable.
4. The corresponding page in the shared buffer will be updated simultaneously.

- Data sync path:

1. Page in the shared buffer will **never** be written to storage nodes.
2. Redo logs are replayed to page storage asynchronously.
3. Directly transfer logs to secondary nodes to reduce the update latency.

(3) Disaggregated Compute-Buffer-Storage Architecture



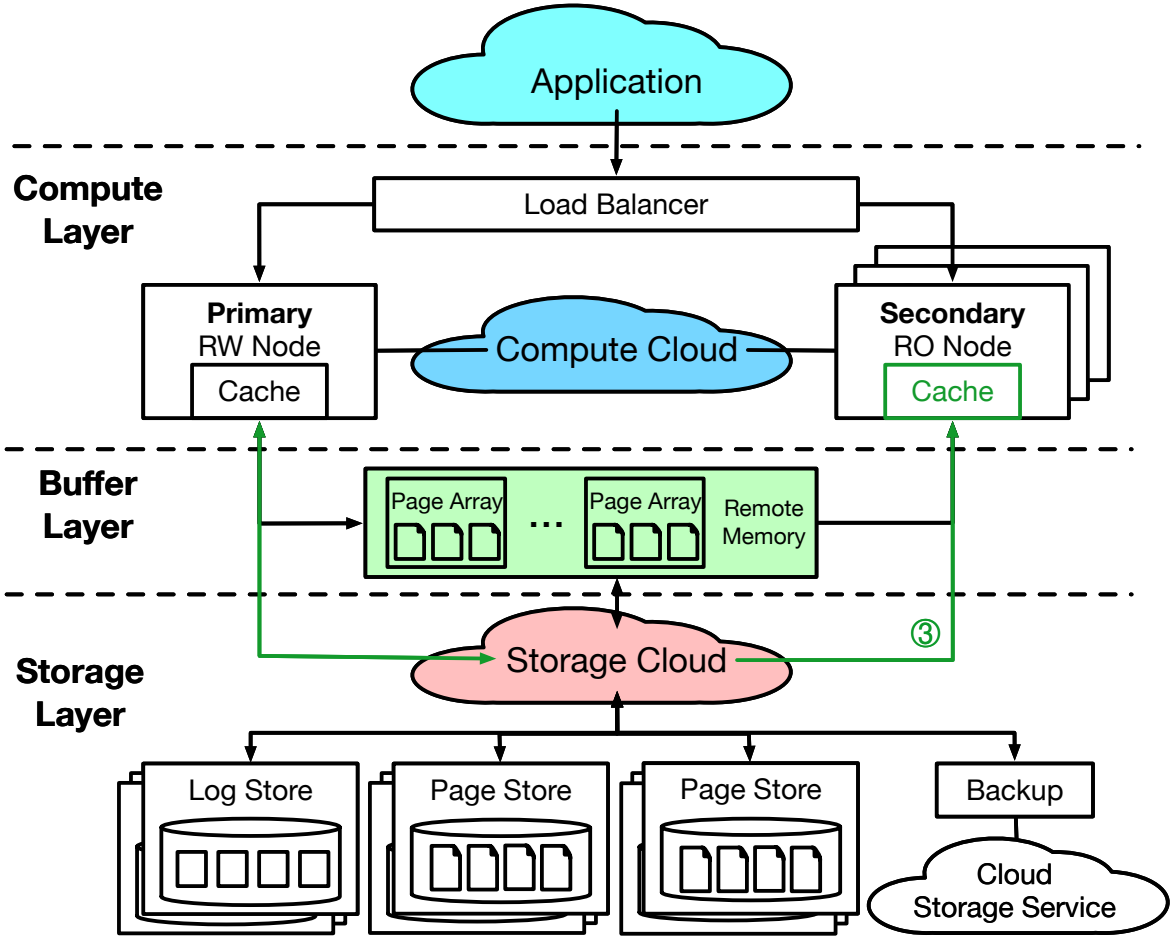
- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. The redo log writes to log storage(multi-replicas) for durability.
3. Commit the write after the redo log is durable.
4. The corresponding page in the shared buffer will be updated simultaneously.

- Data sync path:

1. Page in the shared buffer will never be written to storage nodes.
2. Redo logs are replayed to page storage **asynchronously**.
3. Directly transfer logs to secondary nodes to reduce the update latency.

(3) Disaggregated Compute-Buffer-Storage Architecture



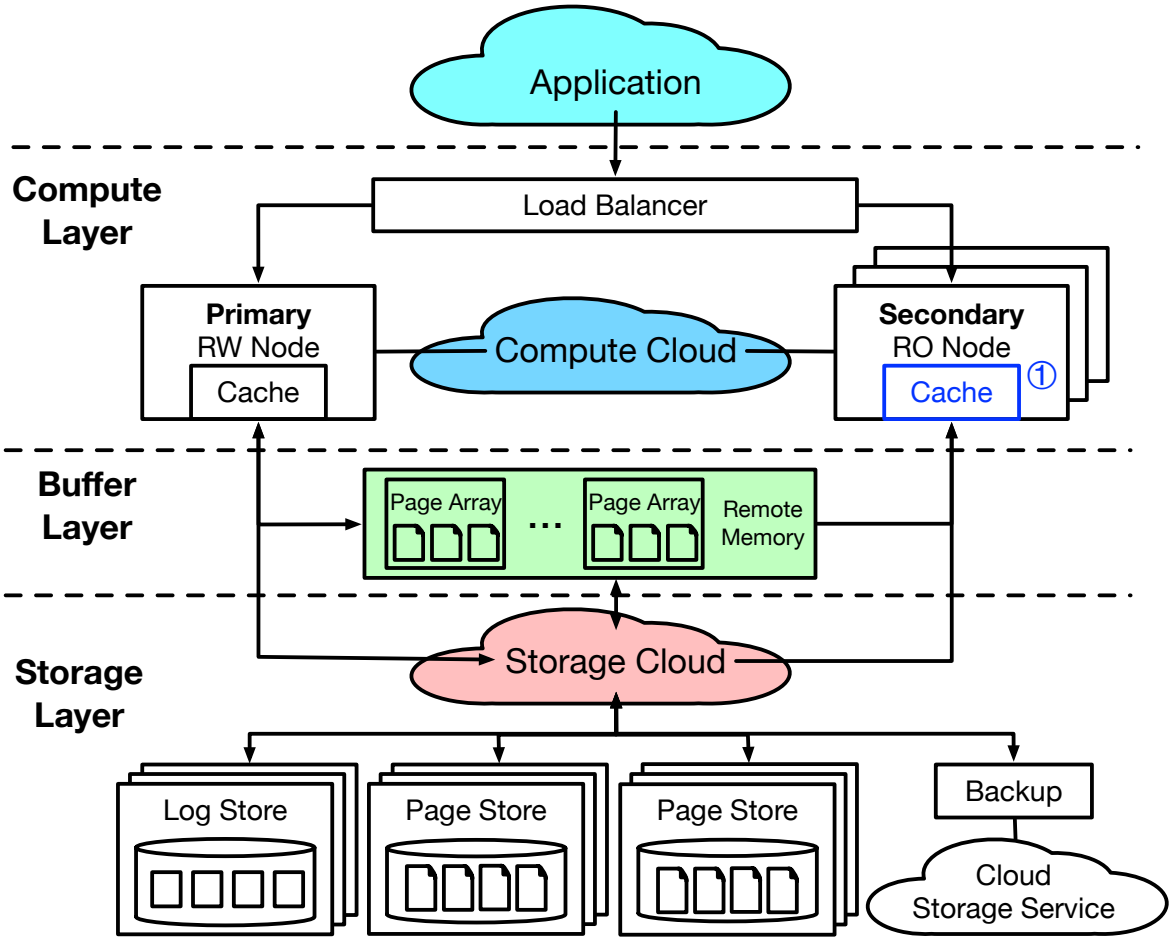
- Data write path:

1. Primary node updates local page in cache and generates redo log.
2. The redo log writes to log storage(multi-replicas) for durability.
3. Commit the write after the redo log is durable.
4. The corresponding page in the shared buffer will be updated simultaneously.

- Data sync path:

1. Page in the shared buffer will never be written to storage nodes.
2. Redo logs are replayed to page storage asynchronously.
3. Directly transfer logs to secondary nodes to reduce the update latency.

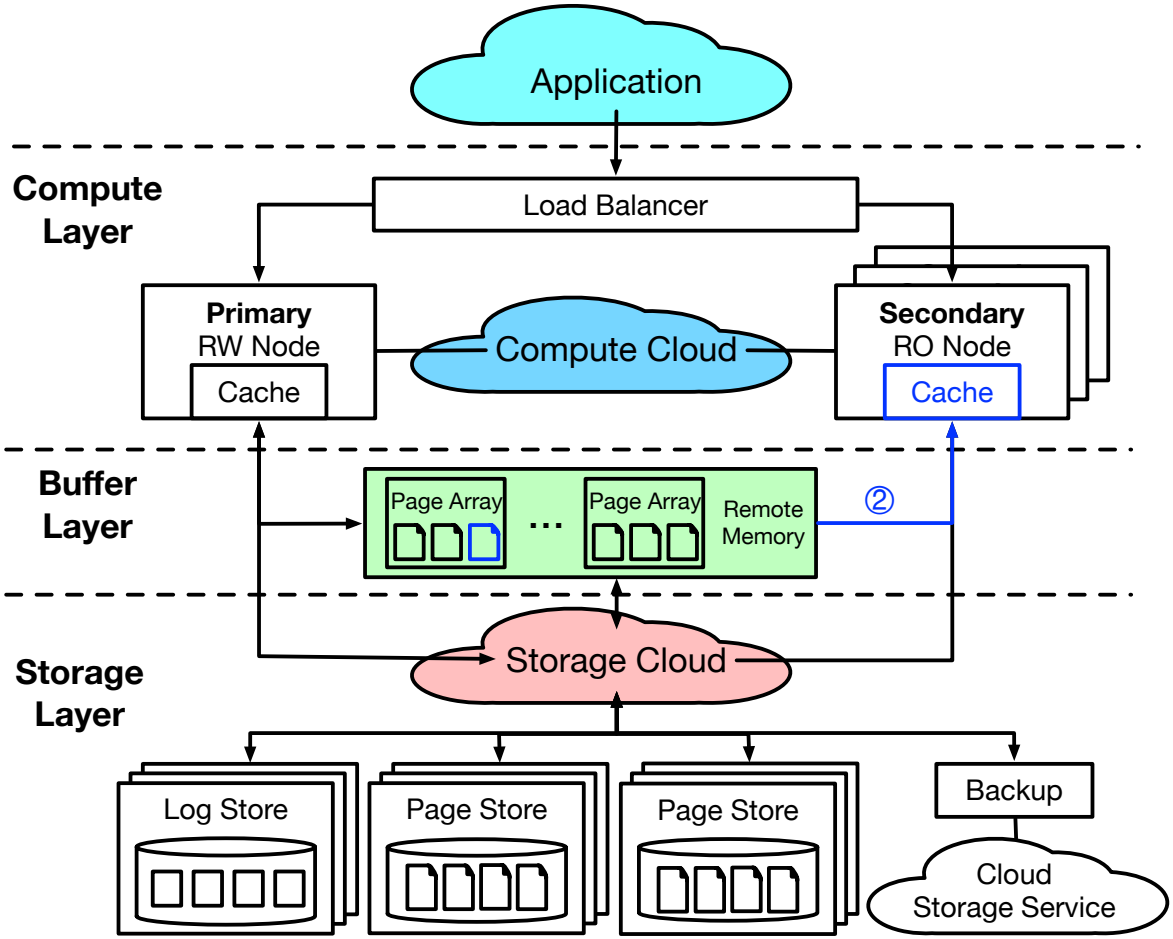
(3) Disaggregated Compute-Buffer-Storage Architecture



- Data read path:

1. First, compute node will check its local cache. When data is in the local cache and valid, it can directly load data from the local cache and **return**.
2. If the local cache misses in ①, compute node will check the remote shared buffer. When data is in the remote buffer and valid, it can load data from the remote buffer and return.
3. If the remote buffer misses in ②, it will read data from page storage nodes, and run the cache replacement algorithm in the remote buffer.
4. Compute node receives data from the remote buffer and runs cache replacement strategy in the local cache, then returns.

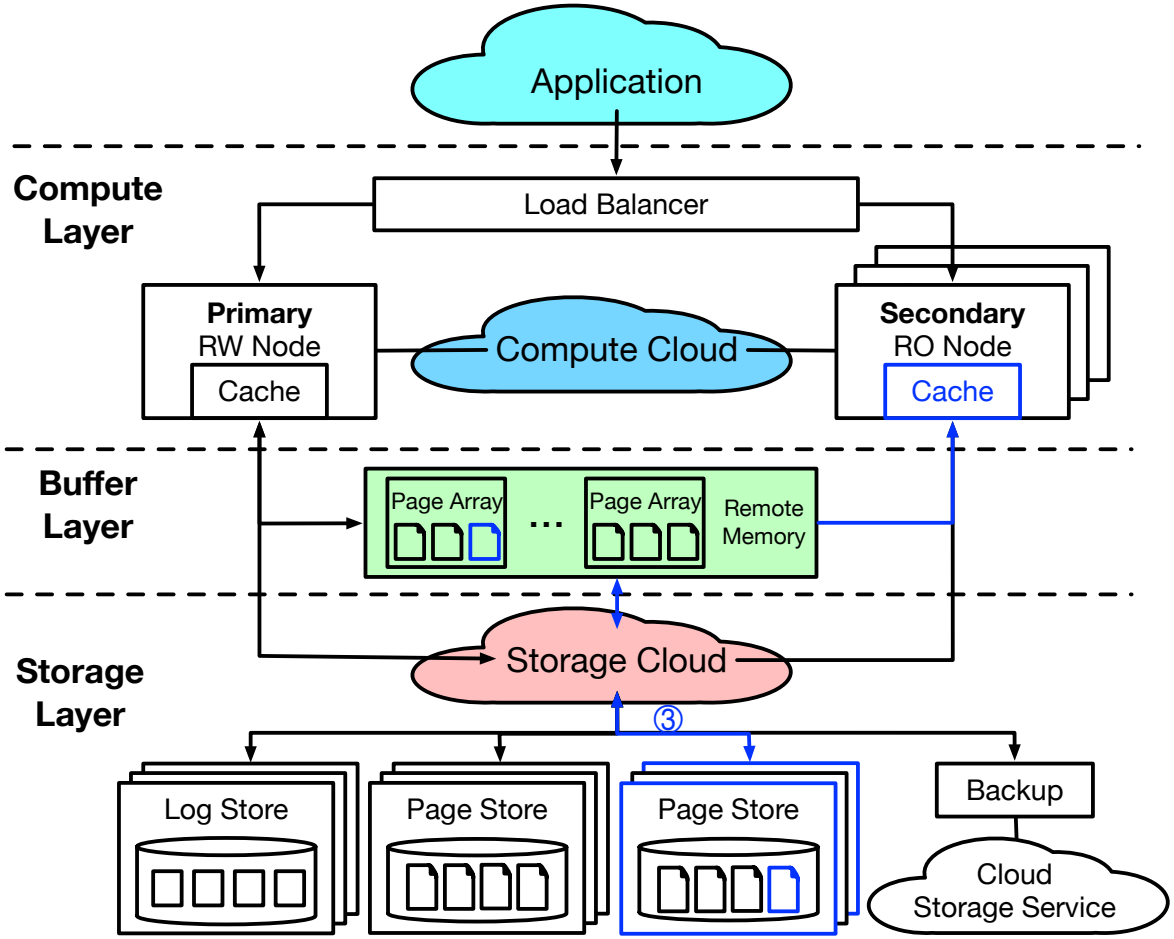
(3) Disaggregated Compute-Buffer-Storage Architecture



- Data read path:

1. First, compute node will check its local cache. When data is in the local cache and valid, it can directly load data from the local cache and return.
2. If the local cache misses in ①, compute node will check the remote shared buffer. When data is **in the remote buffer and valid**, it can load data from remote buffer and **return**.
3. If the remote buffer misses in ②, it will read data from page storage nodes, and run the cache replacement algorithm in the remote buffer.
4. Compute node receives data from the remote buffer and runs cache replacement strategy in the local cache, then returns.

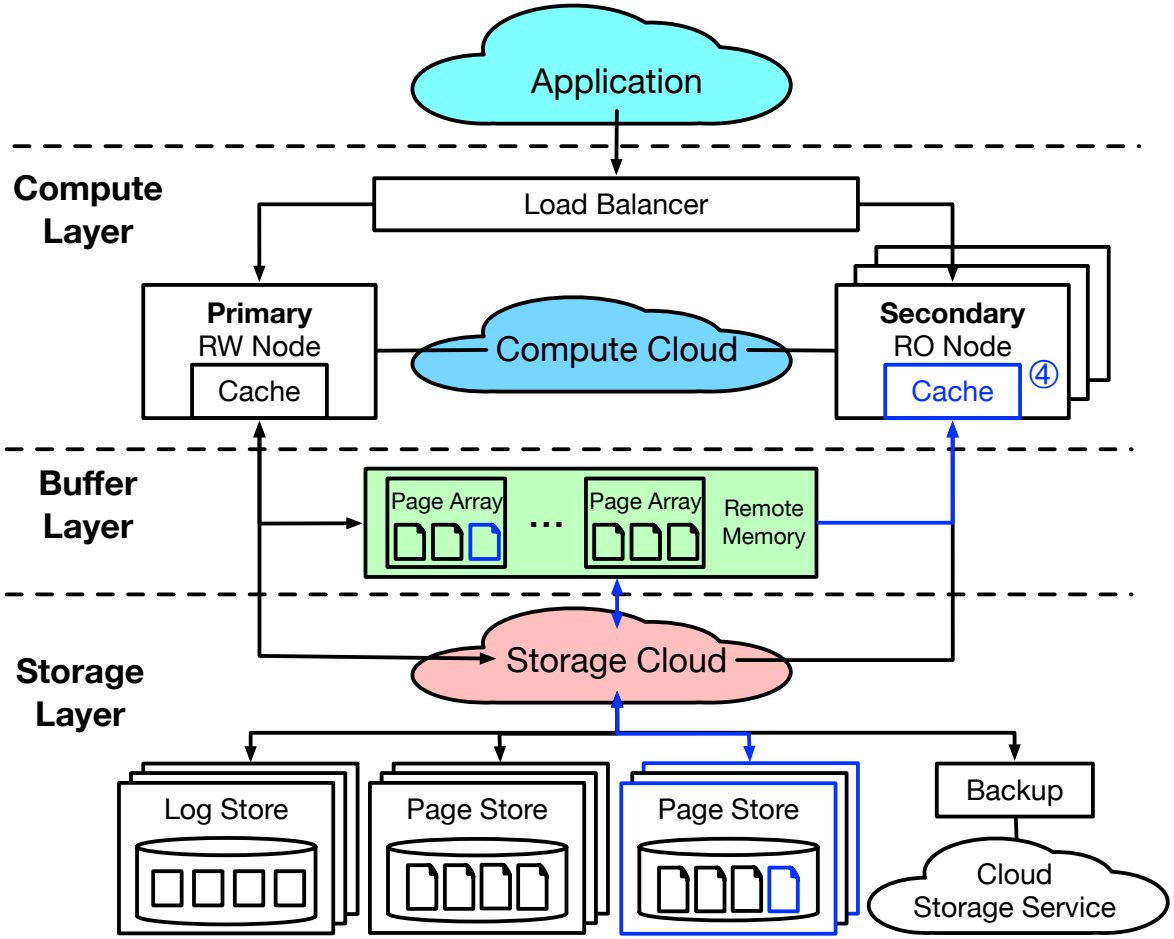
(3) Disaggregated Compute-Buffer-Storage Architecture



- Data read path:

1. First, compute node will check its local cache. When data is in the local cache and valid, it can directly load data from the local cache and return.
2. If the local cache misses in ①, compute node will check the remote shared buffer. When data is in the remote buffer and valid, it can load data from the remote buffer and return.
3. If the remote buffer misses in ②, it will read data from page storage nodes, and run the **cache replacement algorithm in the remote buffer**.
4. Compute node receives data from the remote buffer and runs cache replacement strategy in the local cache, then returns.

(3) Disaggregated Compute-Buffer-Storage Architecture



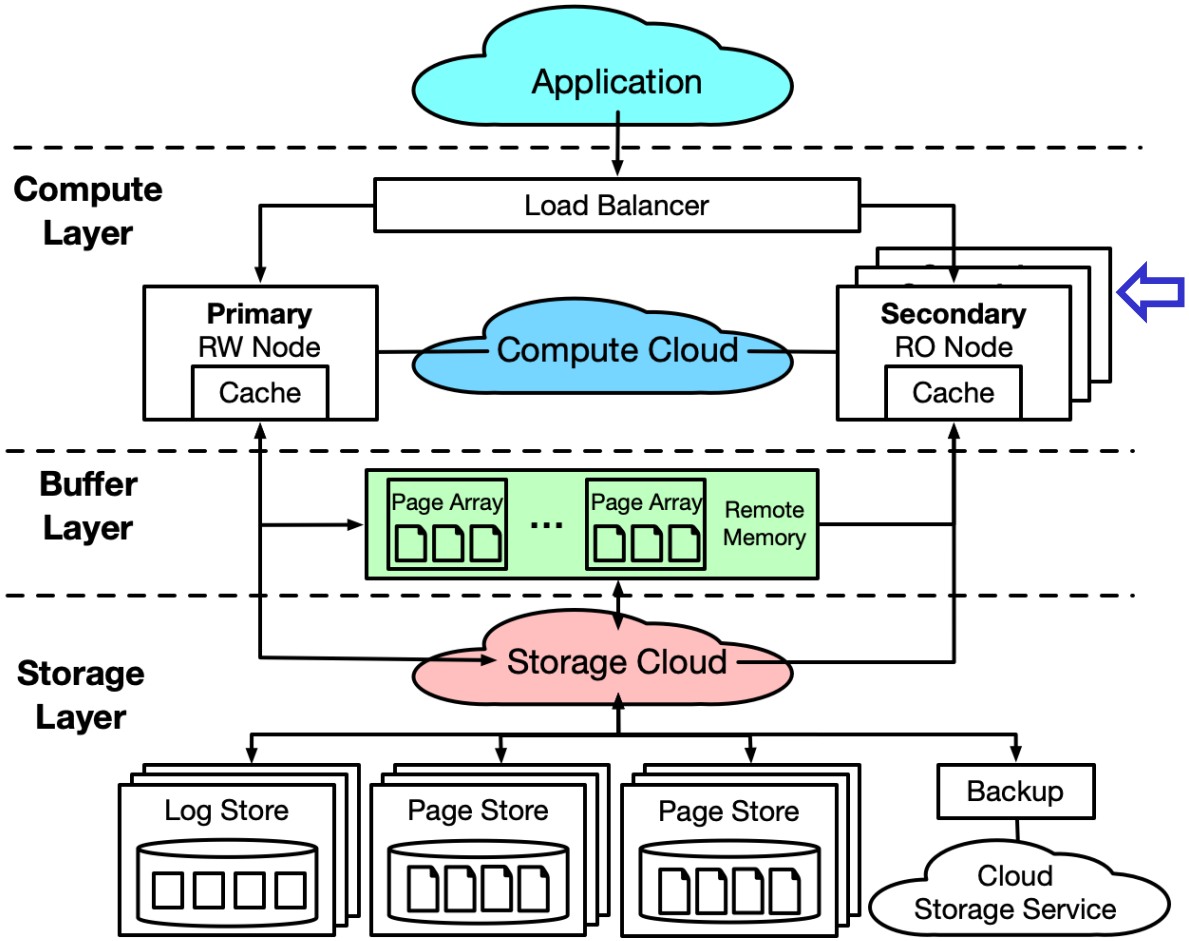
- Data read path:

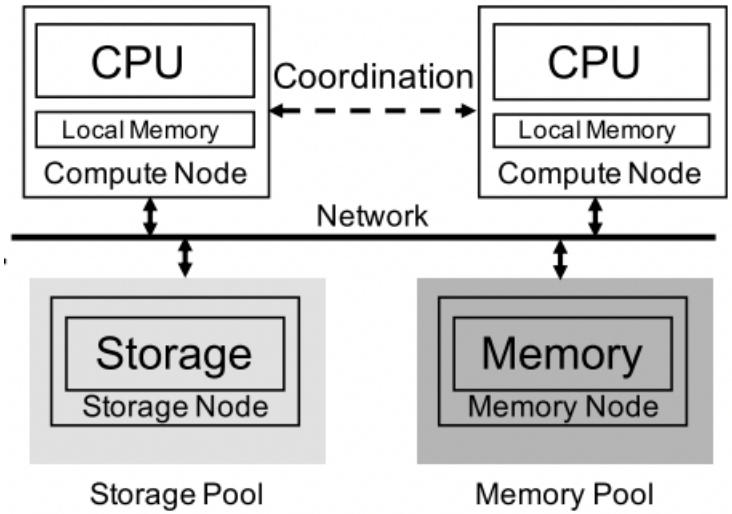
1. First, compute node will check its local cache. When data is in the local cache and valid, it can directly load data from the local cache and return.
2. If the local cache misses in ①, compute node will check the remote shared buffer. When data is in the remote buffer and valid, it can load data from the remote buffer and return.
3. If the remote buffer misses in ②, it will read data from page storage nodes, and run the cache replacement algorithm in the remote buffer.
4. Compute node receives data from the remote buffer and runs cache replacement strategy in the local cache, then **returns**.

(3) Disaggregated Compute-Buffer-Storage Architecture

- **Advantages:**
  - **Low read latency.**
    - Compute nodes can read data from remote memory, which is faster than durable storage services.
  - **High read throughput.**
    - Different compute nodes share the same remote buffer area, which reduces the duplicate data read for the same read requests on different compute nodes.
  - **Better elasticity.**
    - Memory resources can be dynamically allocated on demand, which is independent of compute and storage resources.
- **Limitations:**
  - **Network bottleneck of the buffer layer.**
    - Remote memory requires high network throughput and low latency at the same time. Therefore, the network of remote memory may become the bottleneck of the database system.

(3) Disaggregated Compute-Buffer-Storage Architecture





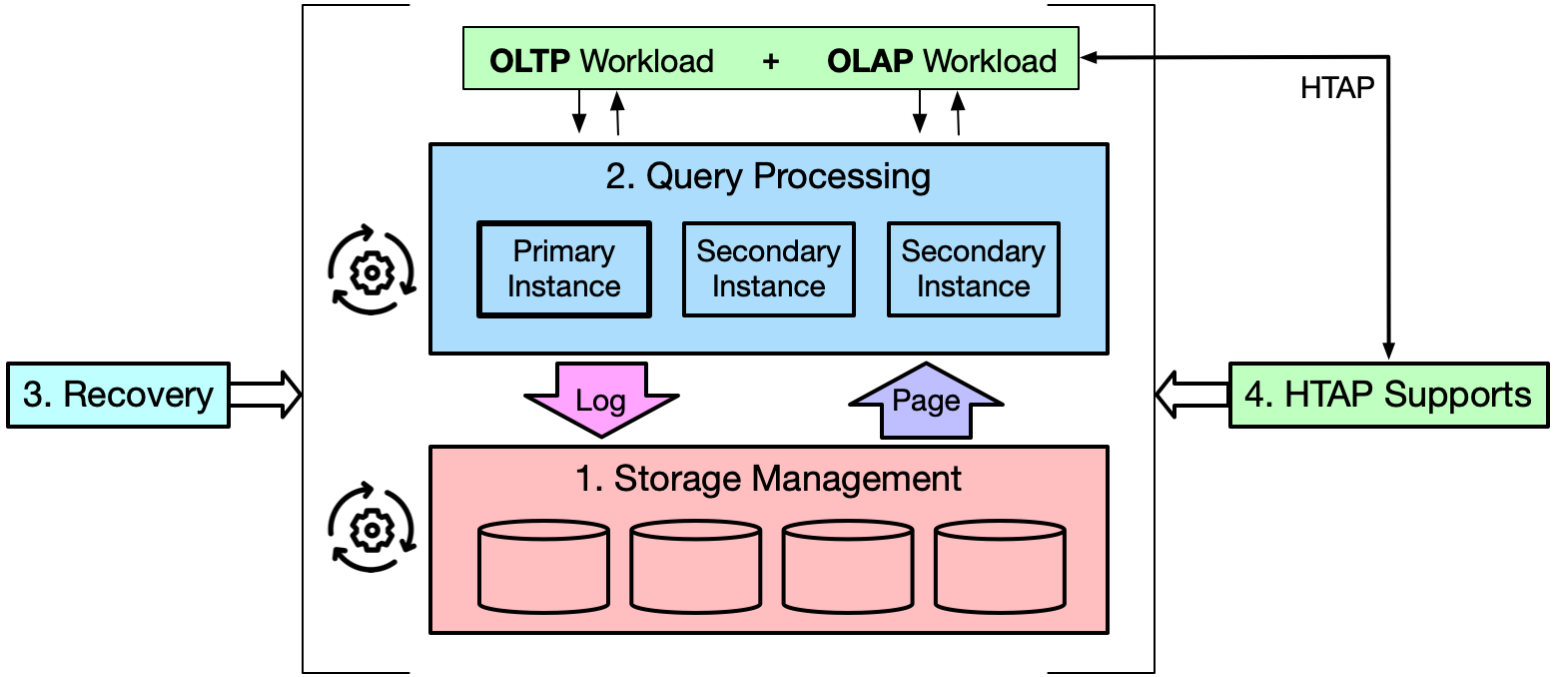
- Examples:
  - Alibaba PolarDB Serverless

Cao, Wei, et al. Polardb serverless: A cloud native database for disaggregated data centers. SIGMOD, 2021.

Cloud-Native OLTP Techniques

Overview of Cloud-Native OLTP Techniques

1. **Storage Management:** Maintain the consistency of different data replicas.
2. **Query Processing:** Synchronize data updates between different computing nodes.
3. **Recovery:** Different-level recovery algorithms for various failures.
4. **HTAP Supports:** Handle OLAP workload on OLTP systems.



1. Storage Management in OLTP

1) Data Placement

a. Coupled Log & Page Storage

b. Disaggregated Log & Page Storage

2) Consistency Protocol

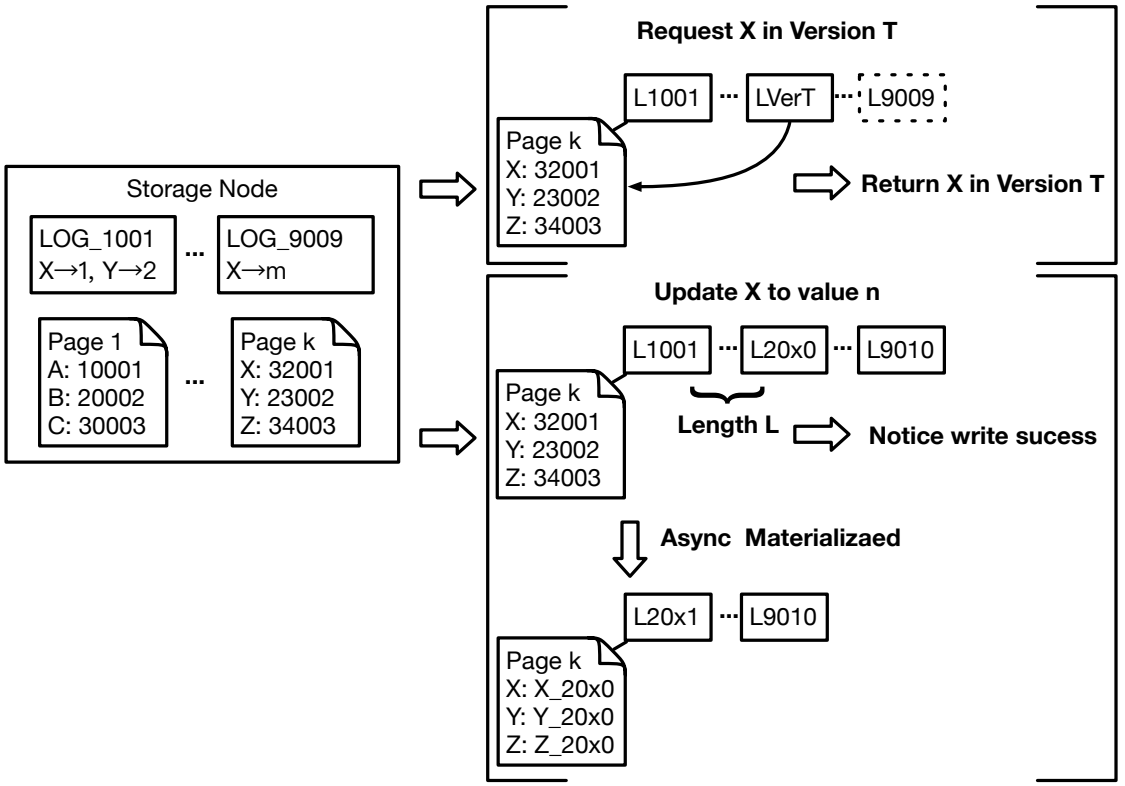
a. Quorum-based protocol: Aurora

b. Paxos-based protocol: PolarDB (PolarFS)

1. Storage Management in OLTP

1a) Coupled Log & Page Storage

- Log is the Database.
- Pages are **never** written from the database tier(including background writes, checkpointing, and cache eviction).
- Read data from logs, and pages are a cache of the log applications.
- Checkpointing & page materialization are both governed by the length of the log chain.
- **All pages are replayed from logs.**
- Cons: Extra time to analyze log chain



1. Storage Management in OLTP

1a) Coupled Log & Page Storage (e.g., Amazon Aurora)

- Compute layer will never transform page data to the storage layer. (Fig. 3)
- Log & page are stored in the same node, while log chains control page replay. (Fig. 4)

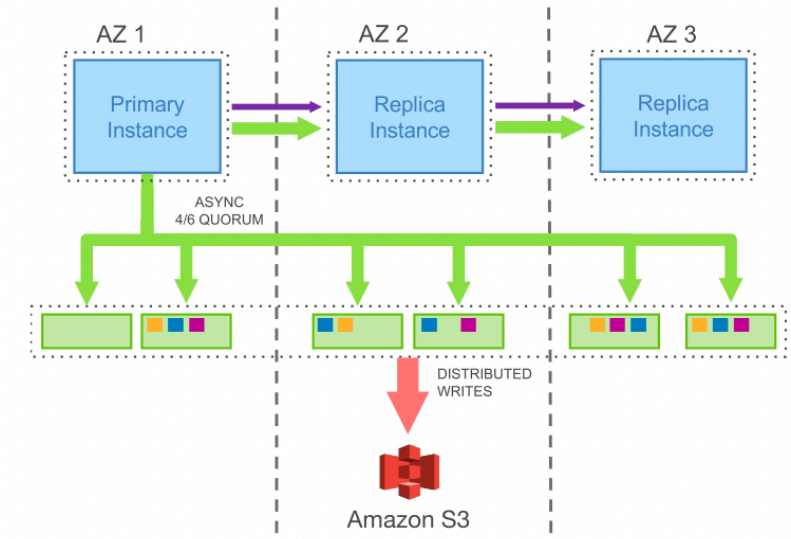


Figure 3: Network IO in Amazon Aurora

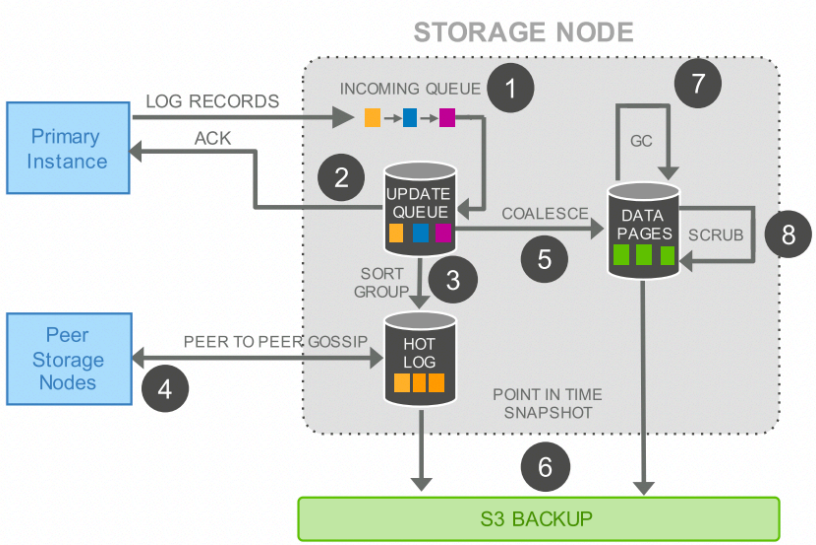


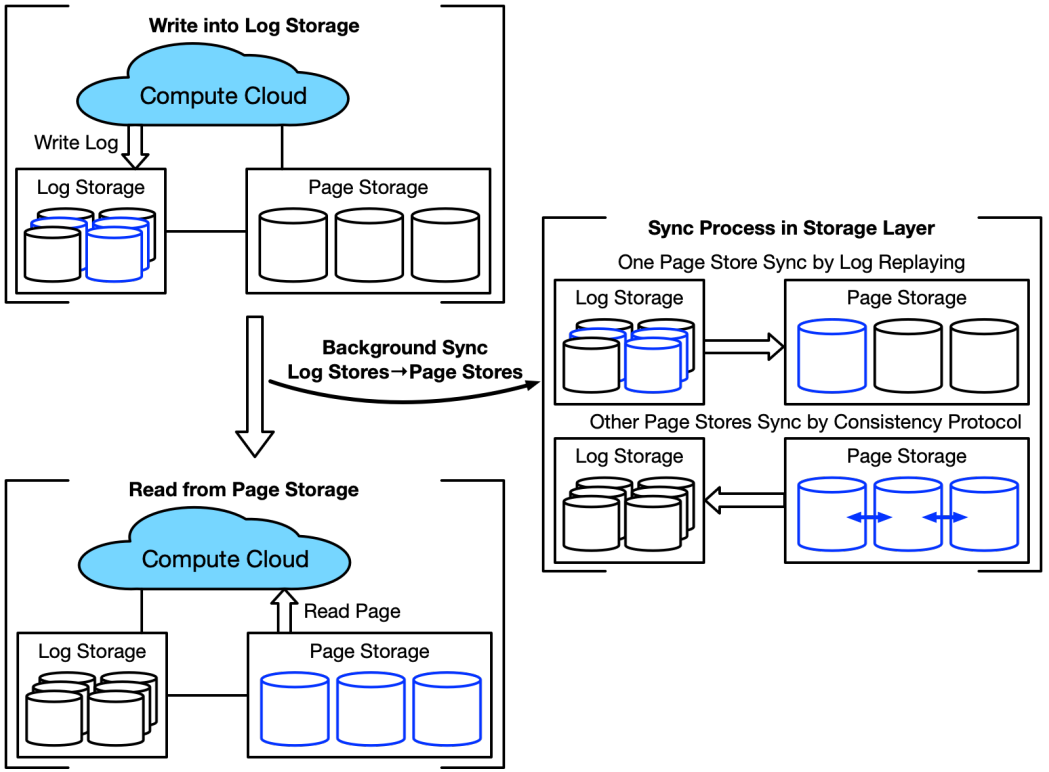
Figure 4: IO Traffic in Aurora Storage Nodes

Verbitski, Alexandre, et al. Amazon aurora: Design considerations for high throughput cloud-native relational databases. SIGMOD, 2017.

1. Storage Management in OLTP

1b) Disaggregated Log & Page Storage

- Disaggregation of Durability & Availability.
- Log storage for durability: reduce write latency (Transaction commits after the logs are durable).
- Page storage for availability: accelerate read processing (Avoid analyzing log chain).
- Page storage nodes replay log data asynchronously.
- **Some pages receive logs from Log Storage; others from the sync process.**
- Cons: Sync latency when read from page storage.



1. Storage Management in OLTP

1b) Disaggregated Log & Page Storage (e.g., Taurus Database)

- Log data for durability.
  - Commit the writes after the log is persistent (Step 1~3).
  - Truncate the logs after the page is persistent (Step 8).
- Page data for availability.
  - Compute nodes only read from page storage.
  - Storage nodes receive logs asynchronously (Step 4~7).
- Log & Page are stored in different nodes offered by different storage services.

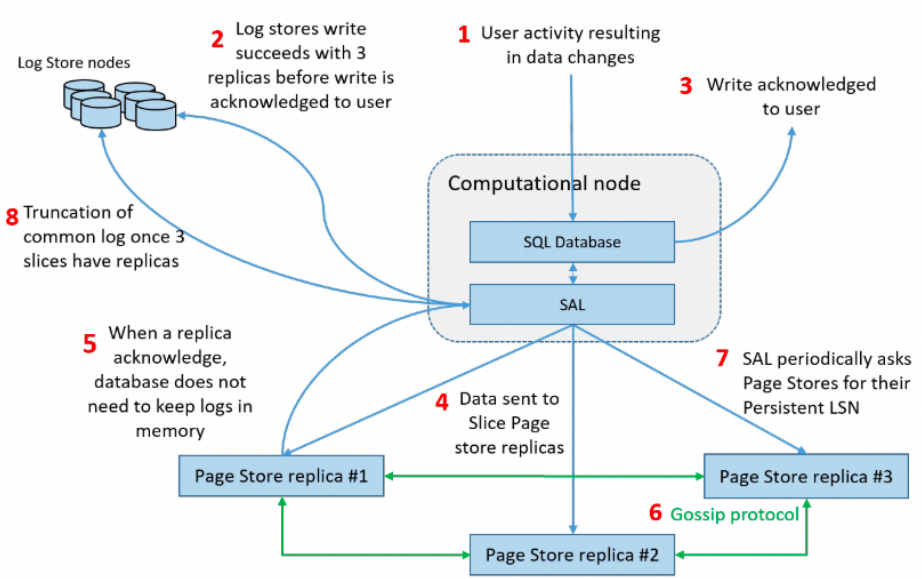


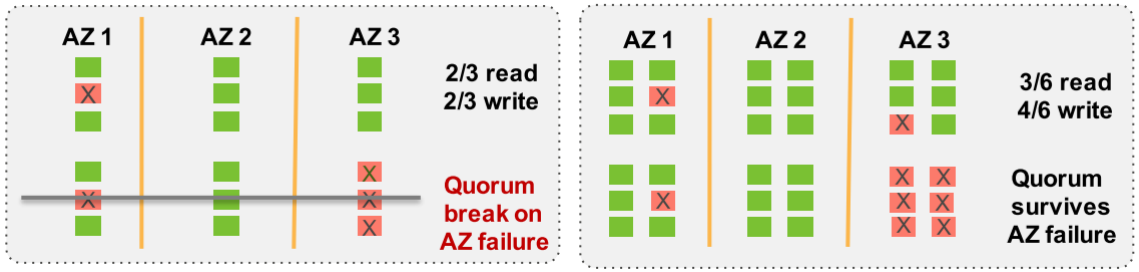
Figure 3: Taurus write path

Depoutovitch, Alex, et al. Taurus database: How to be fast, available, and frugal in the cloud. SIGMOD, 2020.

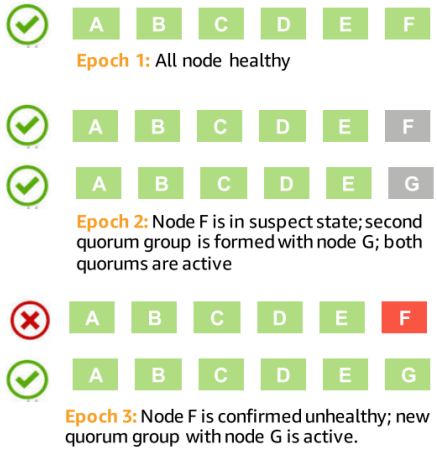
1. Storage Management in OLTP

2a) Quorum-based Protocol

- High concurrency, low fault tolerance.
- Simple & parallel procedure.
- Demand extra gossip process.



Necessity of 6 Replicas in Quorum



Quorum Membership Changes

Case study: Aurora

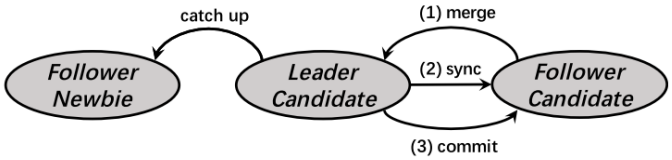
- 6/4/3(N/W/R) for “AZ+1” failure toleration.
- Quorum membership for non-blocking recovery.
- Gossip to fill the missing writes of replicas.

Verbitski, Alexandre, et al. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. SIGMOD, 2018.

1. Storage Management in OLTP

2b) Paxos-based Protocol

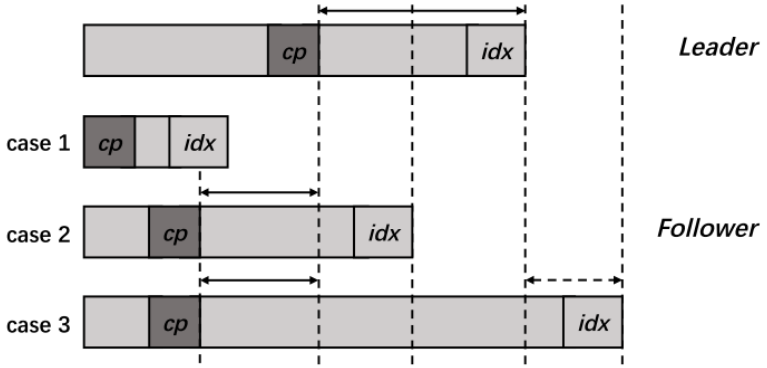
- Strong fault tolerance, low concurrency.
- Complex & linearized procedure.
- Demand concurrency optimization.



Overall of Parallel Raft

Case study: PolarFS

- Parallel Raft Protocol, Derived from Raft and better support for high concurrent I/Os
- Allowing out-of-order log acknowledging, committing, and applying.
- Optimized catch-up mechanism for lagging followers.



Fast Catch Up Process in Parallel Raft

Cao, Wei, et al. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. VLDB, 2018.

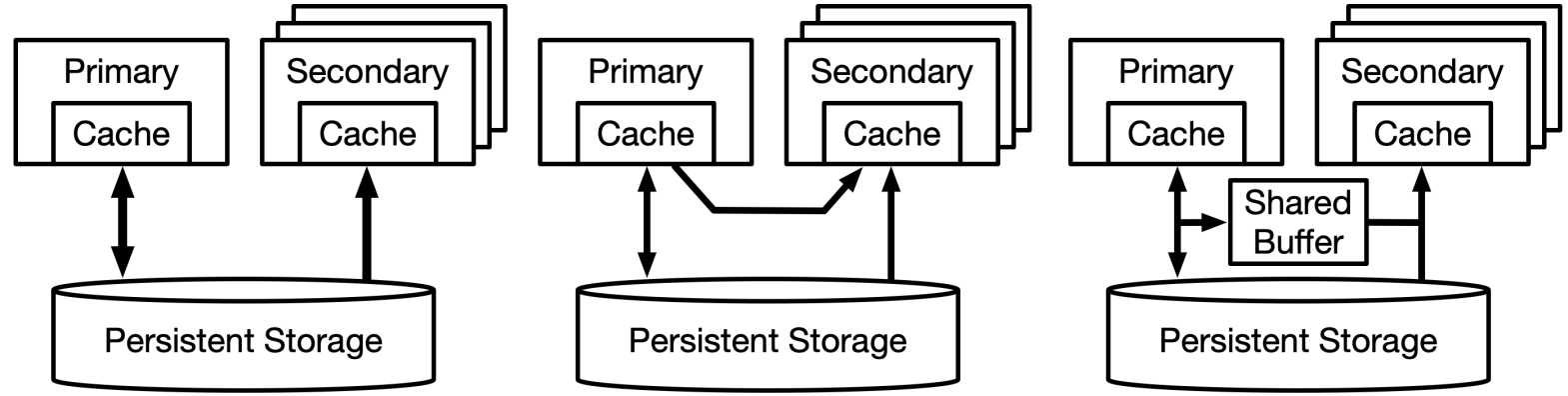
2. Query Processing in OLTP

1) Synchronization for Secondary Nodes

a. Read from Persistent Storage Services

b. Local Cache Synchronization with Redo Log

c. Read from Shared Remote Buffer

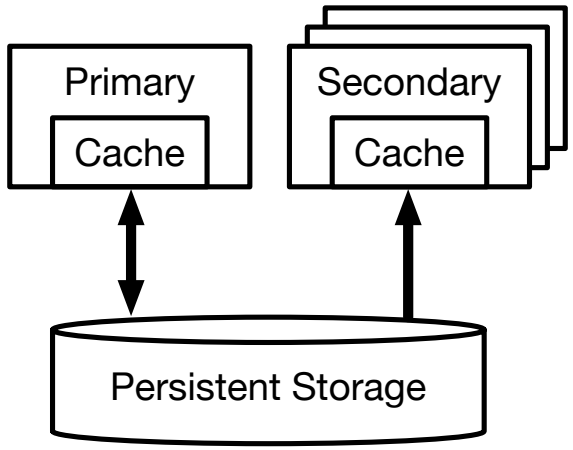


(a) Sync with Persistent Storage (b) Sync with Local Cache (c) Sync with Shared Buffer

2. Query Processing in OLTP

1a) Read from persistent storage services

- The most fundamental guarantee of data consistency.
- One primary RW node with multiple RO secondary nodes. To achieve high concurrency, data updates in the primary node will not wait for buffer synchronization of secondary nodes.
- Reading data from storage services will suffer from much longer latency than reading from the cache.

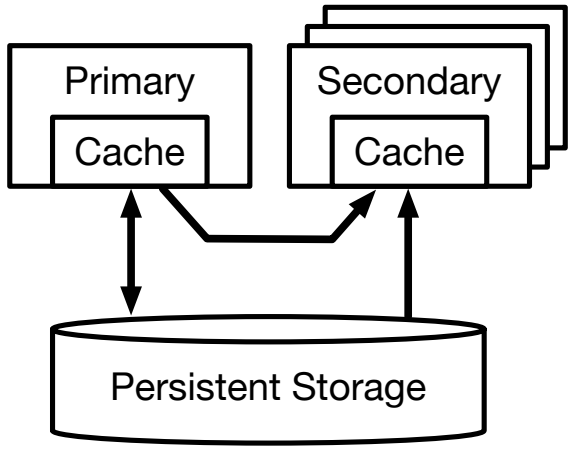


Sync with Persistent Storage

2. Query Processing in OLTP

1b) Local Cache Synchronization with Log

- Low sync latency, transmit redo log from the primary node to secondary nodes for cache status sync.
- High concurrency with lossy protocol, the primary does not require sync confirmation from secondaries.
- Offload log transmission from the primary node to log storage. Avoid network bottleneck at the primary node.



Sync with Local Cache

2. Query Processing in OLTP

1b) Local Cache Synchronization with Log (e.g., HyperScale)

- Disaggregated Compute-Log-Storage Architecture, independent log storage service.
- Motivation: Reduce the update delay between the primary node and secondary nodes.
- Lossy Protocol: Asynchronous and possibly unreliable (in a fire-and-forget style). Avoid blocking the data write process.
- Techniques:
  - Pending area: transaction caching for atomicity.
  - Destaging: Log truncation to save space.

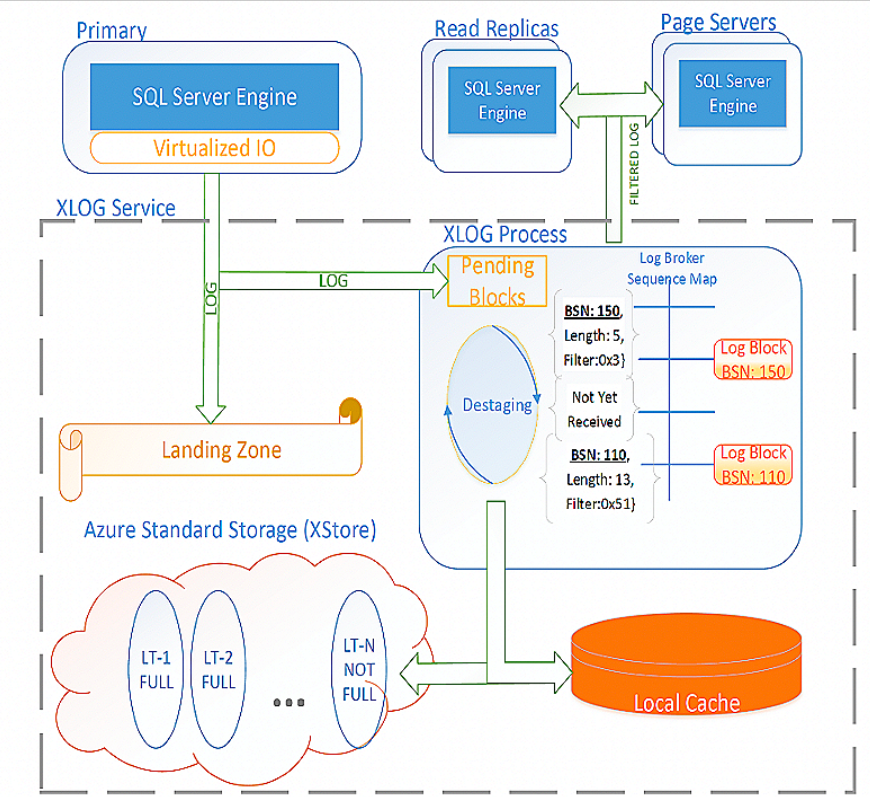


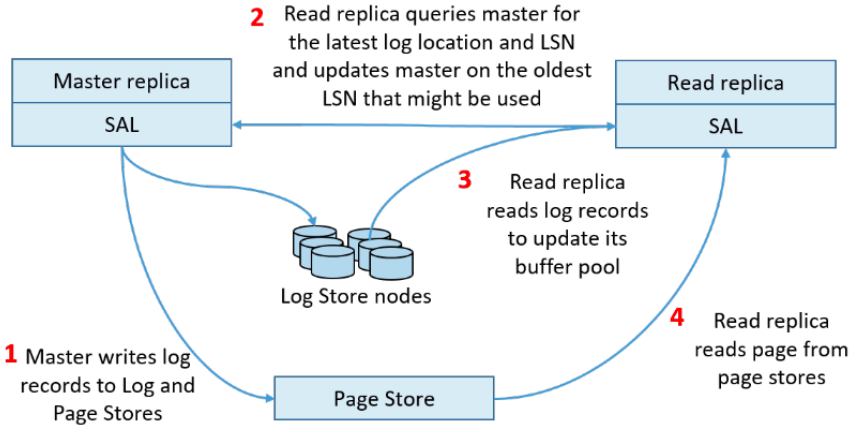
Figure 3: XLOG Service

Antonopoulos, Panagiotis, et al. Socrates: The new sql server in the cloud. SIGMOD, 2019.

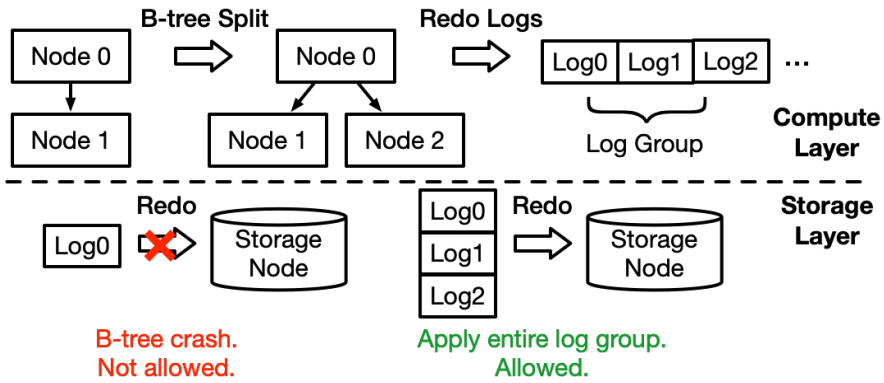
2. Query Processing in OLTP

1b) Local Cache Synchronization with Log (e.g., Taurus Database)

- Network offloading: Secondary nodes get logs from Log Storage instead of the Primary node.
- Two types of consistency: physical & logical.
- Physical consistency: internal structures like b-tree.
  - Primary node: Locking pages (only inside a node).
  - Secondary nodes: Log records group boundary.
- Logical consistency: transaction isolation.
  - Primary node: Generate commit log.
  - Secondary nodes: Receive logs from primary to update active transaction list & Buffer. Buffer stores multi-version pages.



Network offloading to Log Storage



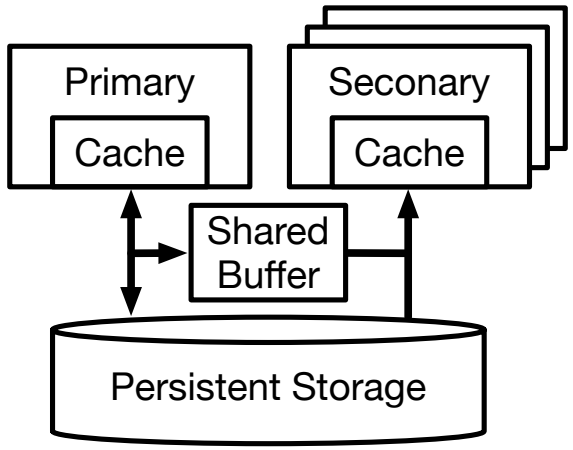
Physical Consistency with Log Group

Depoutovitch, Alex, et al. Taurus database: How to be fast, available, and frugal in the cloud. SIGMOD, 2020.

2. Query Processing in OLTP

1c) Read from Shared Remote Buffer

- High memory resource utilization, allowing remote memory access across different machines.
- Benefits from low page access latency of shared remote memory; reduce sync latency.
- Offloading log replaying from shared buffer to the storage layer. Avoid network bottlenecks at the shared buffer area.



Sync with Shared Buffer

2. Query Processing in OLTP

1c) Shared Remote Buffer: OS-level

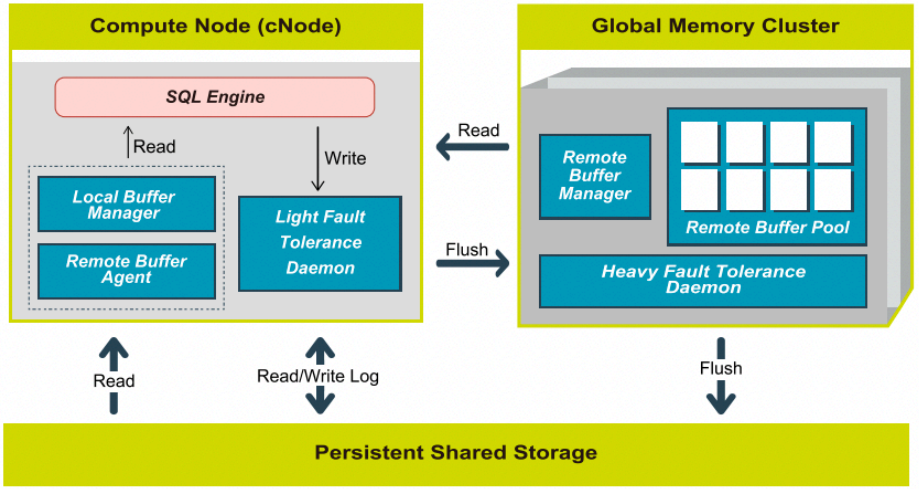
- Motivations:
  - High resource utilization: Expose unused memory across different machines.
  - General-purpose proposals: transparently to unmodified applications.
- Methods:
  - Exposing remote memory paging systems. (e.g., Infiniswap[1])
  - New OS architecture for hardware disaggregation. (e.g., LegoOS[2])
- Limitations:
  - **Full I/O stack**: Each remote page access must go through the full I/O stack, causing the IO latency to be much longer than network latency.
  - **High cache miss ratio**: Unique data access patterns of database workloads, causing a high cache miss ratio in general purpose design.

[1] Gu, Juncheng, et al. Efficient memory disaggregation with infiniswap. NSDI, 2017.
[2] Shan, Yizhou, et al. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. OSDI, 2018.

2. Query Processing in OLTP

1c) Shared Remote Buffer: DB-level(e.g., LegoBase)

- Database layer memory management & remote memory access.
- Motivations of database layer design:
  - Bypass the time-consuming kernel data path, and reduce access latency.
  - Retain the sophisticated design of the LRU mechanism used in the conventional database.
  - Explore database-specific optimizations(e.g., clever metadata caching).
- Key components:
  - **Persistent Shared Storage(pStorage):** Storing WAL, checkpoints and database tables.
  - **Compute Node(cNode):** Performing SQL queries by consuming data from pStorage.
  - **Global Memory Cluster(gmCluster):** Allocating remote memory to cNode.

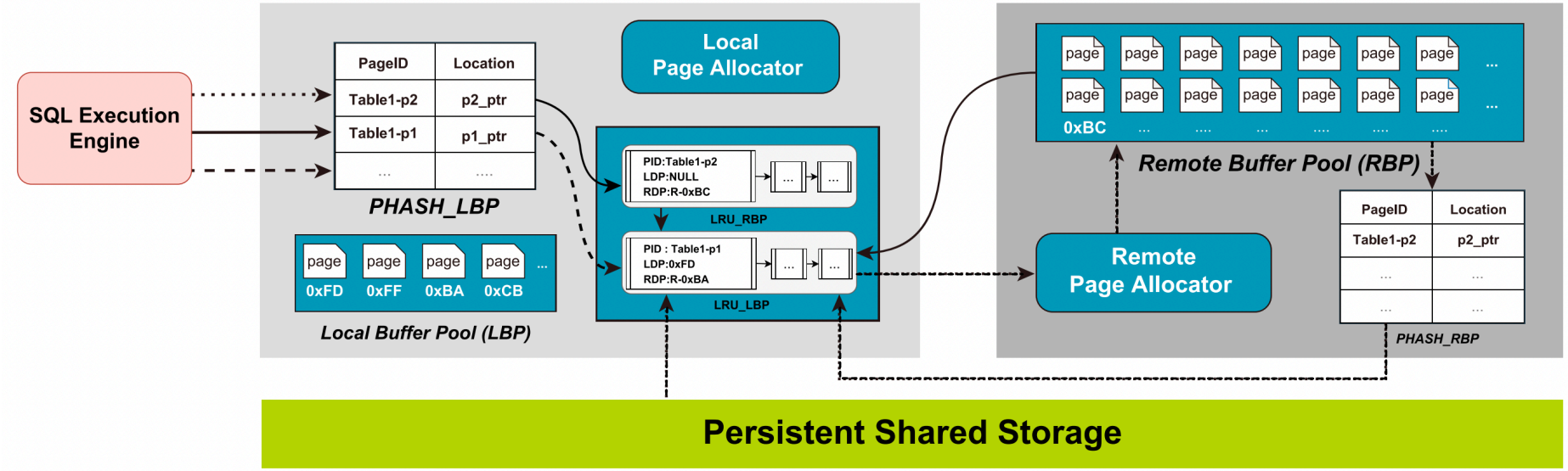


Zhang, Yingqiang, et al. Towards cost-effective and elastic cloud database deployment via memory disaggregation. VLDB, 2021.

2. Query Processing in OLTP

1c) Shared Remote Buffer Area: DB-level (e.g., LegoBase)

- Three page access paths:
  - **Dotted Arrows**: Page accesses from local buffer pool.
  - **Solid Arrows**: Page accesses from locally cached remote address pointers.
  - **Dashed Arrows**: Page accesses from persistent shared storage.



Zhang, Yingqiang, et al. Towards cost-effective and elastic cloud database deployment via memory disaggregation. VLDB, 2021.

2. Query Processing in OLTP

1c) Challenges of DB-level Shared Remote Buffer(e.g., PolarDB Serverless)

- **C1**: Consistency of shared remote buffer & local buffer of the primary node.
- **S1**: Cache invalidation mechanism: *Page Invalidation Bitmap*(PIB, page update status) and *Page Reference Directory*(PRD, page user list) in the home node to achieve cache coherency.
- **C2**: A large amount of network transmission of Shared Memory is caused by cache flush.
- **S2**: Page materialization offloading: Pages can be evicted without flushing back.

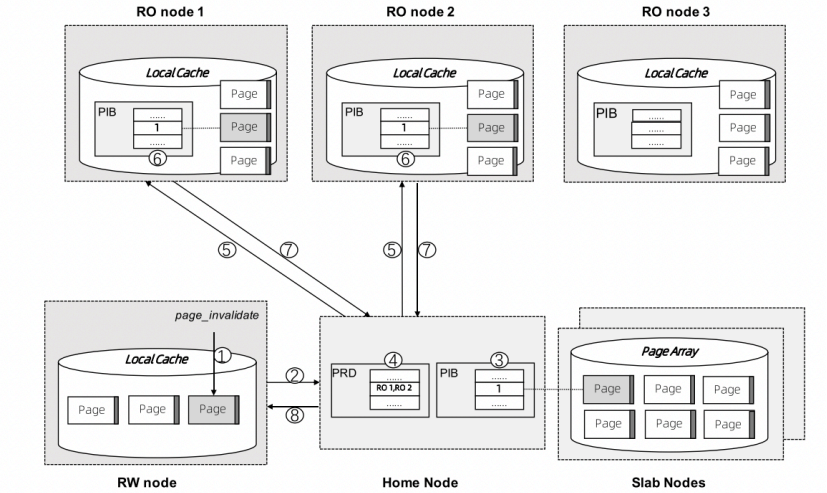


Figure 6: Cache Invalidation

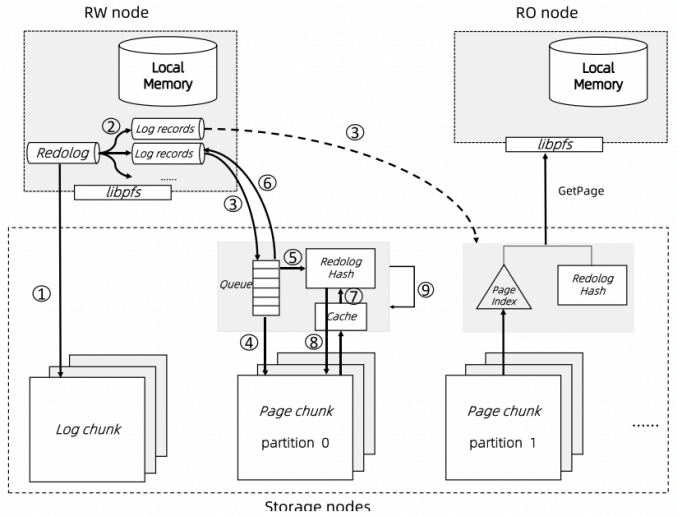


Figure 7: Page Materialization Offloading

Cao, Wei, et al. Polardb serverless: A cloud native database for disaggregated data centers. SIGMOD, 2021.

3. Recovery Mechanism in OLTP

1) Non-persistent Layer Recovery (Compute & Buffer)

a. Mechanism based on Disaggregation of Compute & Storage

b. Mechanism based on Disaggregation of Compute & Buffer

2) Persistent Layer Backup & Recovery (Log & Page Storage)

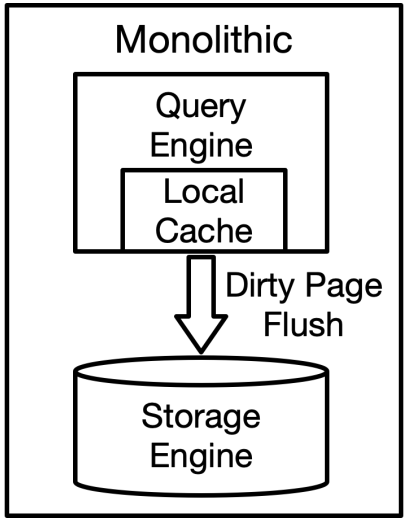
a. Mechanism based on Coupled Log & Page Storage

b. Mechanism based on Disaggregation of Log & Page Storage

3. Recovery Mechanism in OLTP

1) Traditional Database Recovery Algorithm (ARIES)

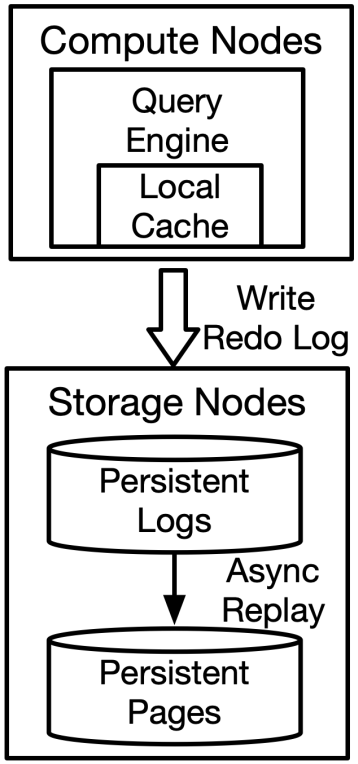
- Algorithm Workflow (3 Phases):
  - **Analysis Phase**: Determine the start point of the redo phase. Find the not persisted dirty pages for Redo Phase and uncommitted transactions for Undo Phase.
  - **Redo Phase**: Redo from the start point, replaying the transaction updates to pages in the local cache, and re-establishing a stable runtime state.
  - **Undo Phase**: Roll back the uncommitted transactions, async.
- Features:
  - Shared-Everything(Monolithic) Architecture.
  - Updates by **Pages Flush**: Dirty pages in local cache flush to persistent storage and updates old pages.



3. Recovery Mechanism in OLTP

1a) Mechanism based on Disaggregation of Compute & Storage

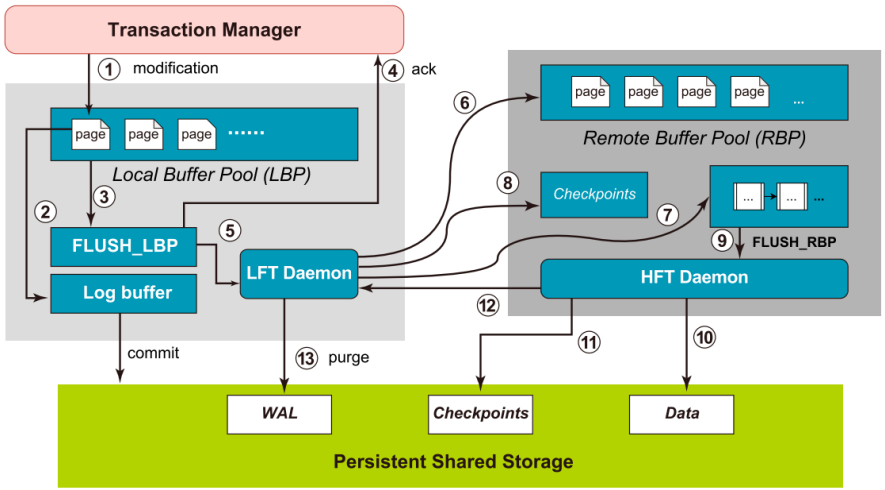
- Differences in disaggregation:
  - Shared Storage Architectures.
  - Updates by **Logs Replay**: Page updates are replayed from log data.
- Influences on recovery:
  - **No Redo Phase in Compute Layer**: Compute layer does not need to re-establish the local cache state. The storage layer will guarantee durability after crashes & asynchronously consuming redo logs. Significantly reduce recovery time.
  - **Cold Cache Issue**: Under the shared storage architecture, cache in the storage layer & shared buffer of all compute nodes will not lose together with compute nodes, diluting the negative influence of the cold cache issue.



3. Recovery Mechanism in OLTP

1b) Mechanism based on Disaggregation of Compute & Buffer

- Precondition: Compute nodes & remote buffer are unlikely to fail simultaneously.
- Motivation: Reduce the recovery time of compute node failure.
- Two-tier ARIES Fault Tolerance Protocol (e.g., LegoBase):
  1. Compute Node & Remote Buffer: High frequency with low recovery time, light fault tolerance.
  2. Remote Buffer & Persistent Storage: Ensuring worst-case data persistency, heavy fault tolerance.
- Protocol Procedure:
  1. Query Execution (Step 1~4).
  2. Remote Buffer Checkpoint (Step 5,6,8).
  3. Persistent Storage Checkpoint (Step 7,9~11).
  4. Log Truncation (Step 12~13).



Zhang, Yingqiang, et al. Towards cost-effective and elastic cloud database deployment via memory disaggregation. VLDB, 2021.

3. Recovery Mechanism in OLTP

2a) Persistent Storage Recovery with Coupled Log & Page Storage

- Motivations:
  - Avoid heavyweight distributed commit process, e.g., 2-phase commit.
  - Avoid complex recovery situations in the multi-replica situation.
- Methods:
  - **Storage Consistency Points**: The lower bound of LSN has met Quorum in a protection group. Gaps inside a node can be eliminated by gossiping with other peers inside the group. System Commit Number should not exceed Volume Complete LSN(minimum Consistency Point).
  - **Log Truncation during Recovery**: The existence of gaps inside nodes will generate complex recovery situations. To avoid these situations, the storage node will ignore the log after VCL.

Figure 3: Storage Consistency Points

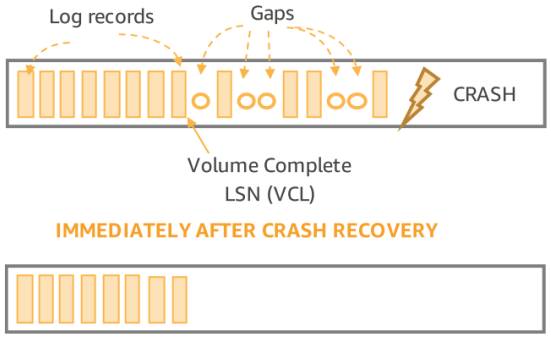


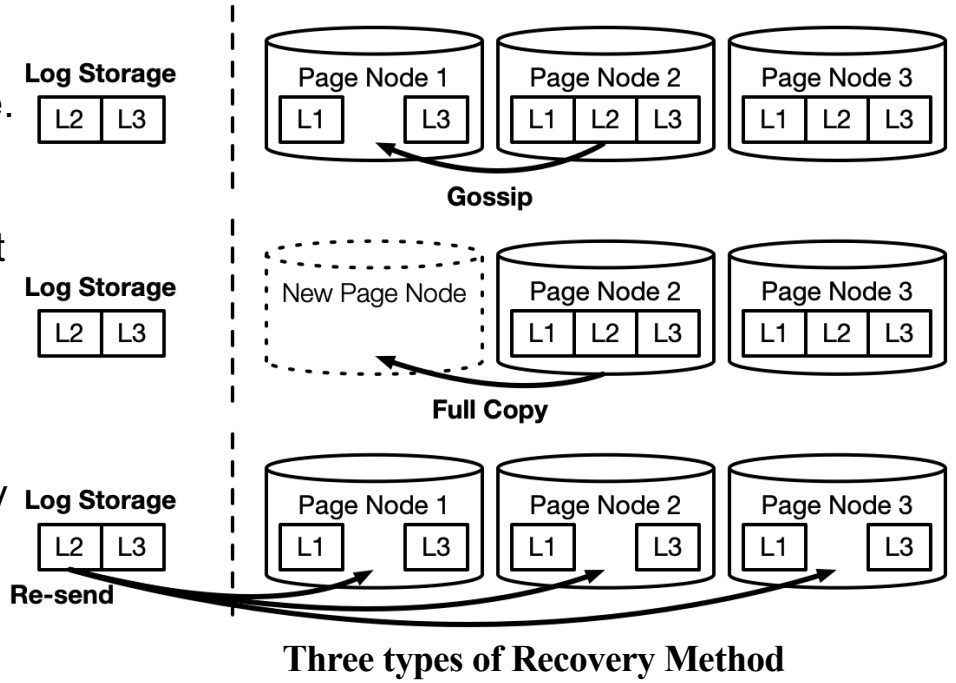
Figure 4: Log truncation during crash recovery

Verbitski, Alexandre, et al. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. SIGMOD, 2018.

3. Recovery Mechanism in OLTP

2b) Persistent Storage Recovery with Disaggregated Log & Page Storage

- Motivations:
  - Reduce the data transfers for the Log Storage.
  - Recovery from any type of failures in Page Storage.
- Methods:
  - **Gossip in Page Storage**: Page storage nodes that suffer short-term failure receive missing logs from the Gossip of the others.
  - **Full Copy in Page Storage**: Allocating new page storage node when original node is down. Full copy data from other normal node.
  - **Log re-send from Log Storage**: When the log is completely lost in the page store, the Log Storage will re-send the log to the Page Storage.



Three types of Recovery Method

Depoutovitch, Alex, et al. Taurus database: How to be fast, available, and frugal in the cloud. SIGMOD, 2020.

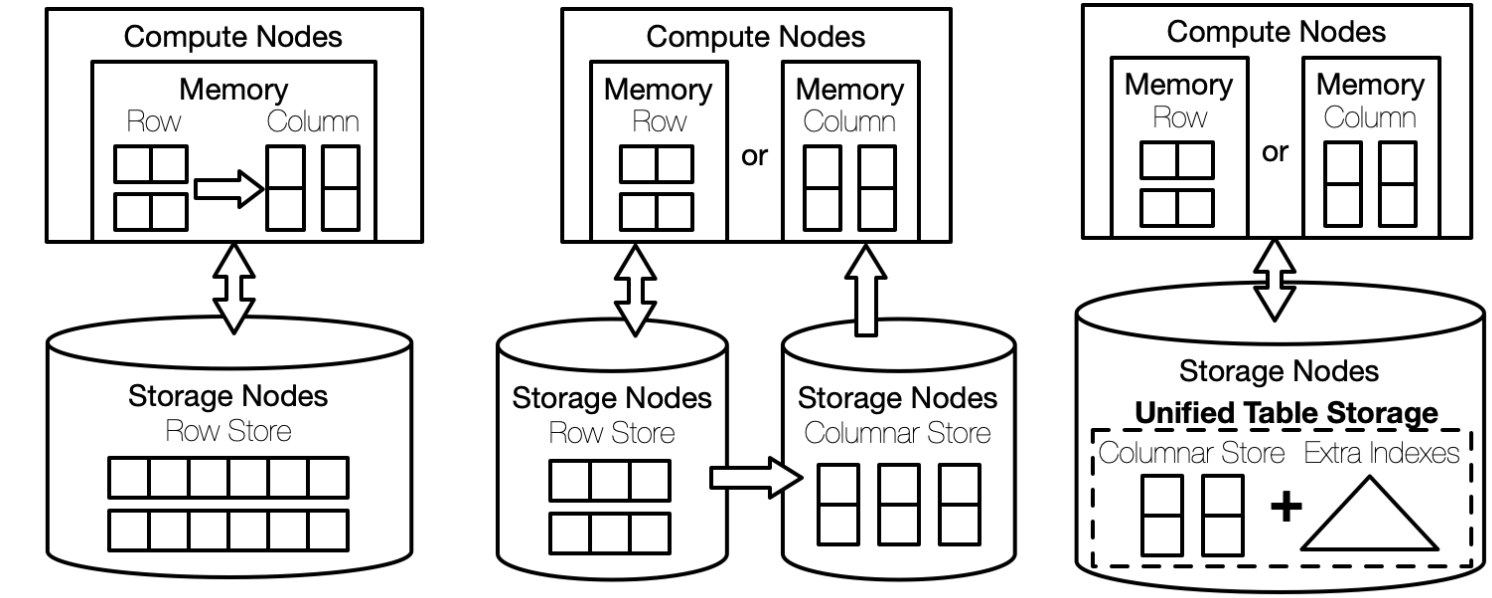
4. HTAP Supports in OLTP

1) HTAP on Disaggregation Architecture

a. Compute Layer Transformation.

b. Storage Layer Heterogeneous Replicas.

c. Unified Table Storage Structure.

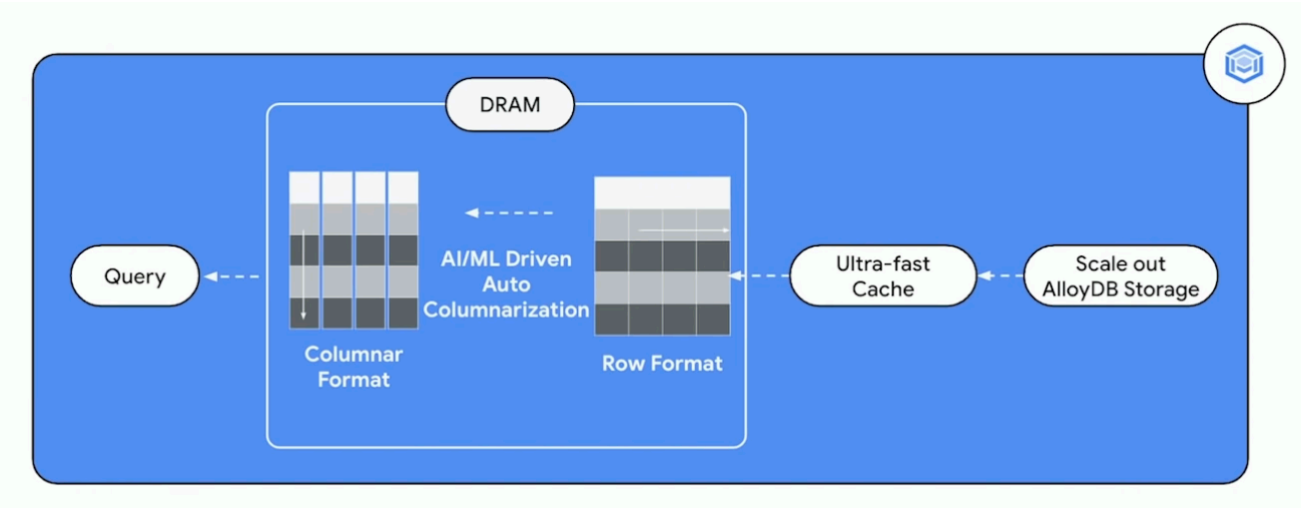


(a) Compute Layer Transformation (b) Storage Layer Heterogeneous Replicas (c) Unified TableStorage Structure

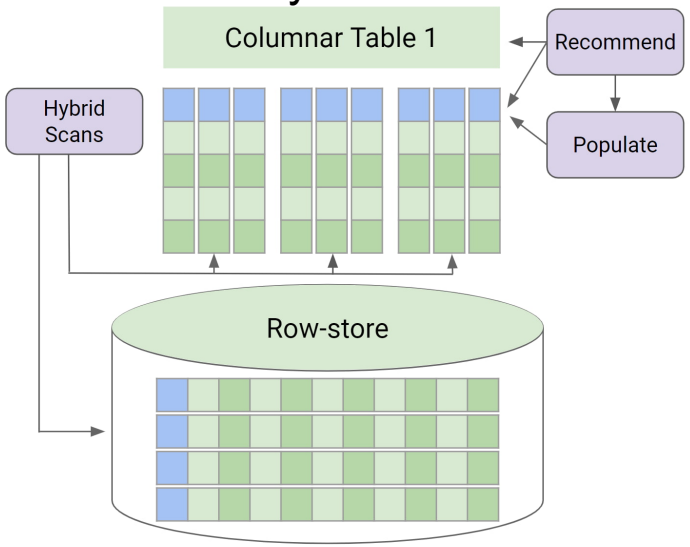
4. HTAP Supports in OLTP

1a) Compute Layer Transformation(e.g., Google AlloyDB)

- Data is stored in row format in persistent storage.
- Processing as row format under OLTP workload.
- Automatically converting row to columnar format to support OLAP queries.
- Support hybrid scan on columnar and row-oriented data simultaneously.



Hybrid Scans in AlloyDB



Hybrid Scans in AlloyDB

Figure from https://cloud.google.com/blog/products/databases/alloydb-for-postgresql-columnar-engine

4. HTAP Supports in OLTP

1b) Storage Layer Heterogeneous Replicas(e.g., TiDB)

- Heterogeneous storage replicas(TiKV for row format, TiFlash for columnar format)
- TiKV supports OLTP workload & maintains consistency under Raft Protocol.
- TiFlash async receives logs from TiKV for updates & not participate in Raft Protocol.
- Columnar Delta Tree structure to support efficient read and write, appends delta updates immediately and later merges to stable columnar chunks.

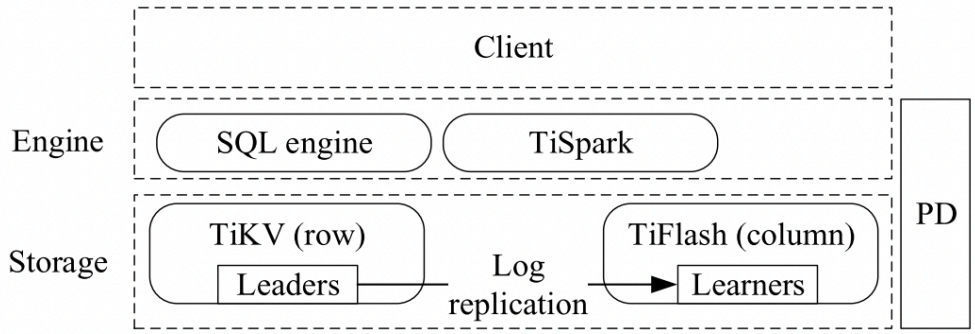


Figure 2: TiDB architecture

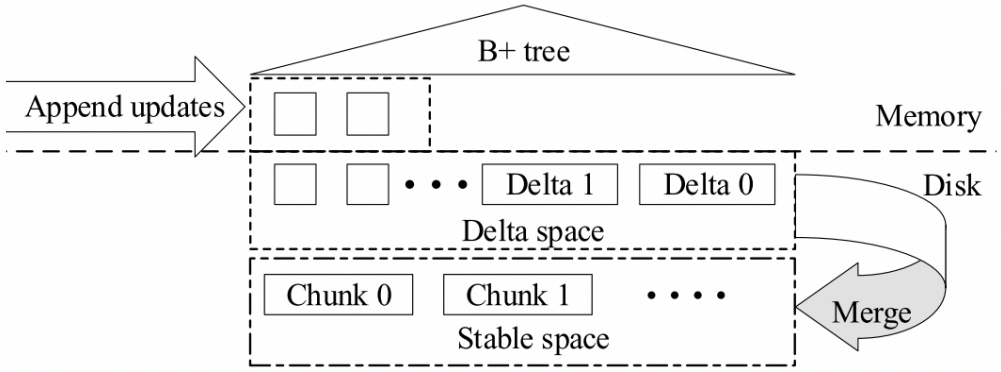


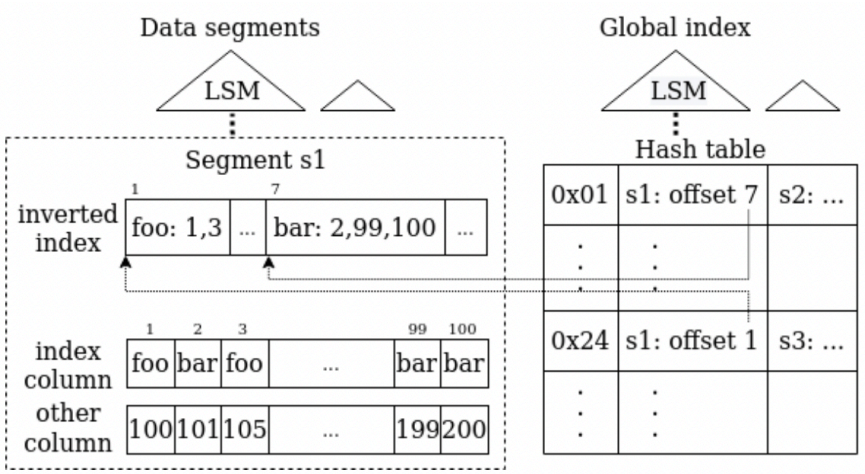
Figure 4: The columnar delta tree

Huang, Dongxu, et al. TiDB: a Raft-based HTAP database. VLDB, 2020.

4. HTAP Supports in OLTP

1c) Unified Table Storage Structure(e.g., SingleStoreDB)

- Row format storage in memory & Columnar storage in persistent storage.
- Combining scan performance of column store & seek performance of row store.
- Columnar LSM tree structure for OLAP + Secondary hash indexes for OLTP
- Two-level secondary indexes: Data Segments (immutable) & Global Index (mutable).

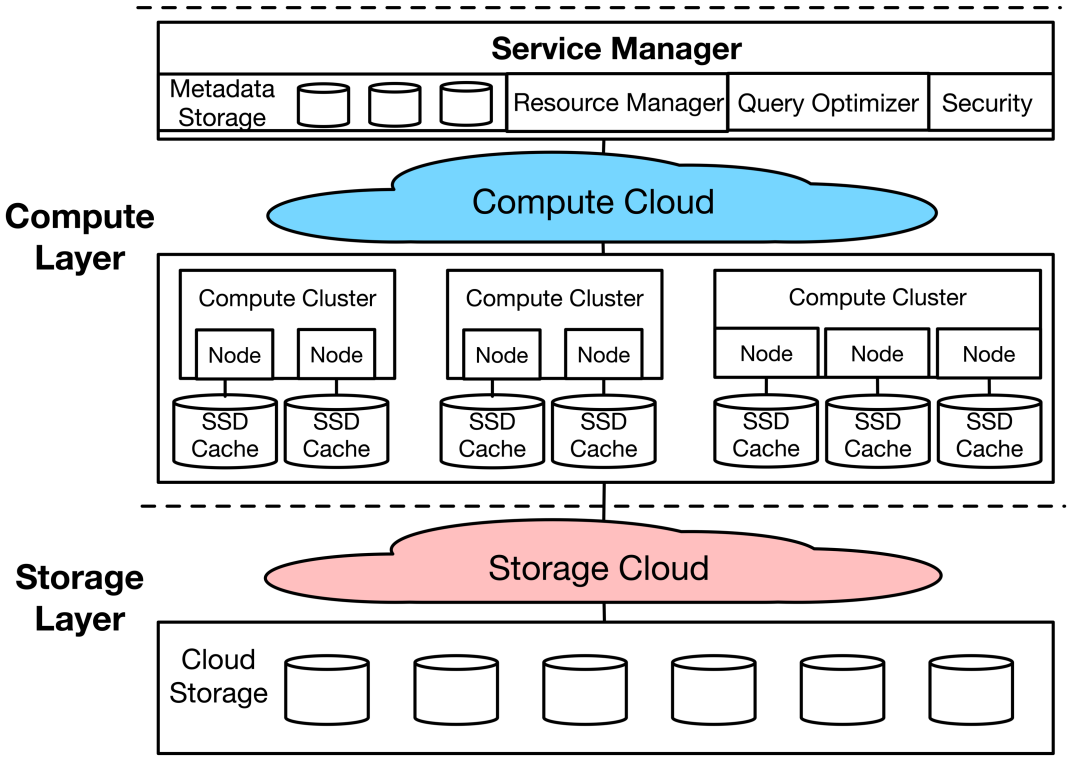


Two-level secondary indexes in S2DB

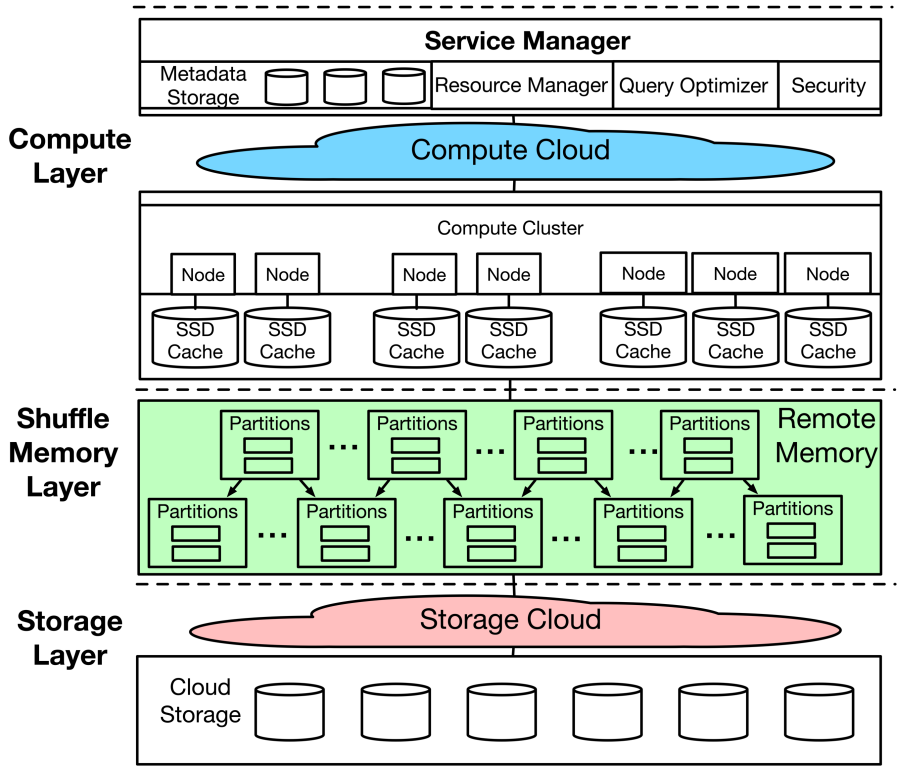
Prout, Adam, et al. Cloud-Native Transactions and Analytics in SingleStore. SIGMOD, 2022.

Cloud-Native OLAP Architectures

An Overview of Cloud OLAP Architectures

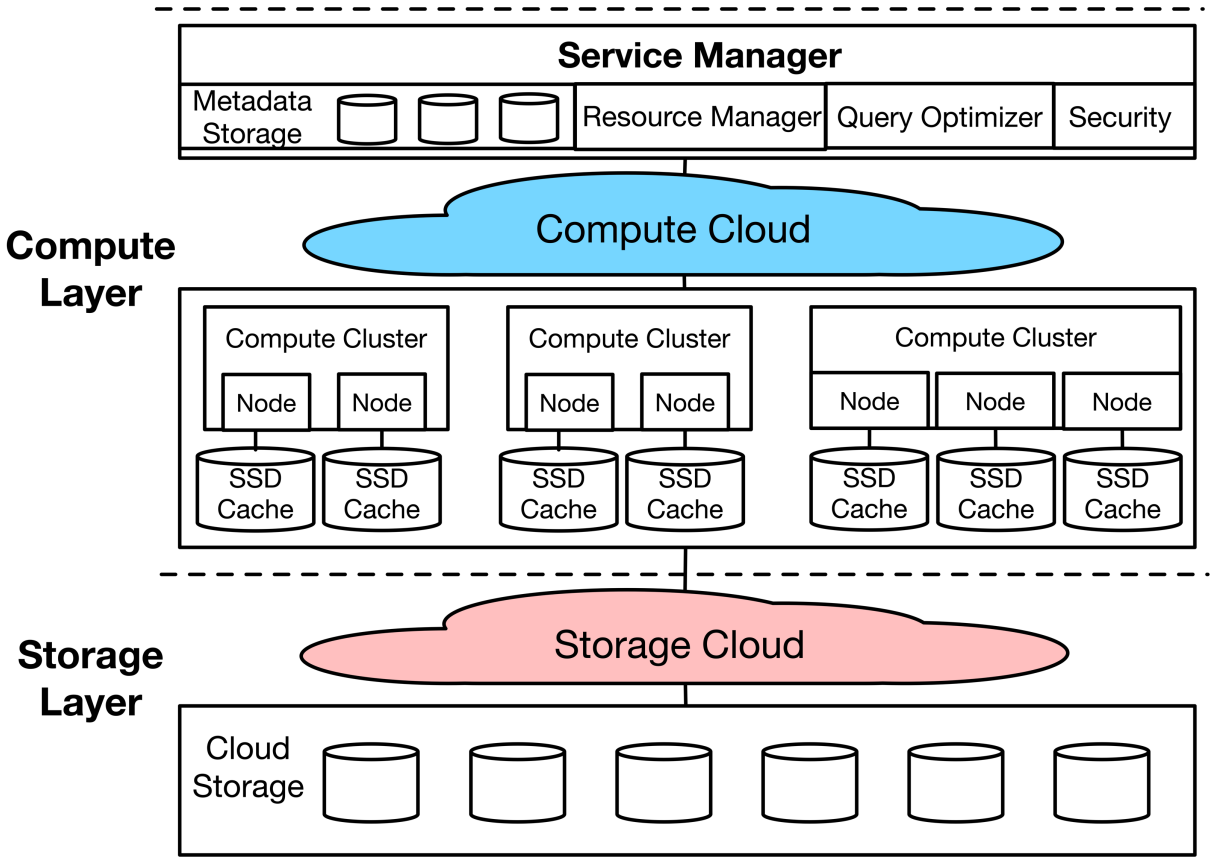


(1) Disaggregated Compute-Storage OLAP Architecture



(2) Disaggregated Compute-Memory-Storage OLAP Architecture

(1) Disaggregated Compute-Storage Architecture



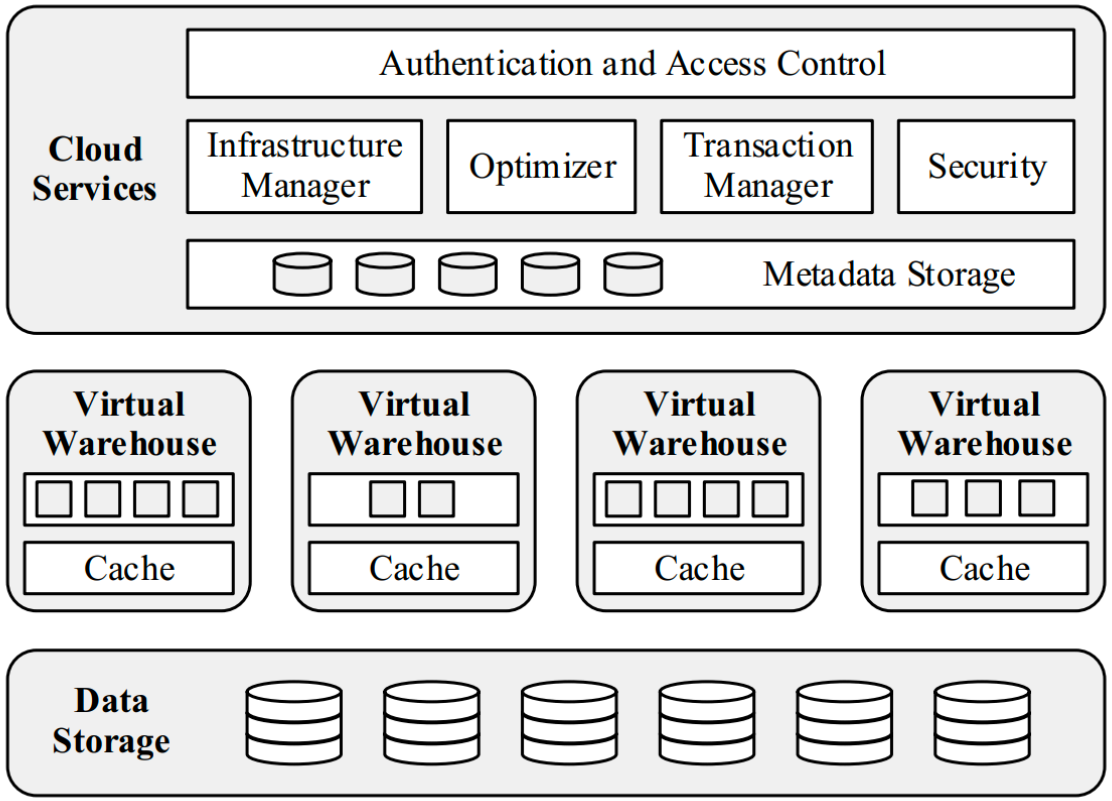
(1) Disaggregated Compute-Storage OLAP Architecture

- Motivation:
  - **Elasticity**: Storage and compute resources need to be scaled independently
  - **Availability**: Tolerate cluster and node failures
  - **Heterogeneous workloads:** high I/O bandwidth or heavy compute
- Key Features:
  - Disaggregation of Compute and Storage
  - Multi-tenancy and Serverless
  - Elastic Data Warehouses
  - Local SSDs Caching
  - Cloud Storage Service, e.g., AWS S3, Google Cloud Storage, Azure Blob Storage

(1) Disaggregated Compute-Storage Architecture

- **Advantages compared to On-Premise Share-Nothing OLAP Architectures:**
  - **Higher availability**
    - Cluster and node failures can be recovered quickly because of (1) the data replication across many availability zones and (2) the scalable cloud service.
  - **More cost-efficient**
    - Resources are virtualized and shared by multiple tenants.
    - Serverless computing provides the pay-as-you-go model in a query-level granularity.
  - **Better elasticity**
    - The compute and storage resources can be scheduled on demand individually.
- **Limitation:**
  - **Network traffic becomes the bottleneck when the local cache is not hit**
    - Need to design efficient and effective caching and computation pushdown strategies.

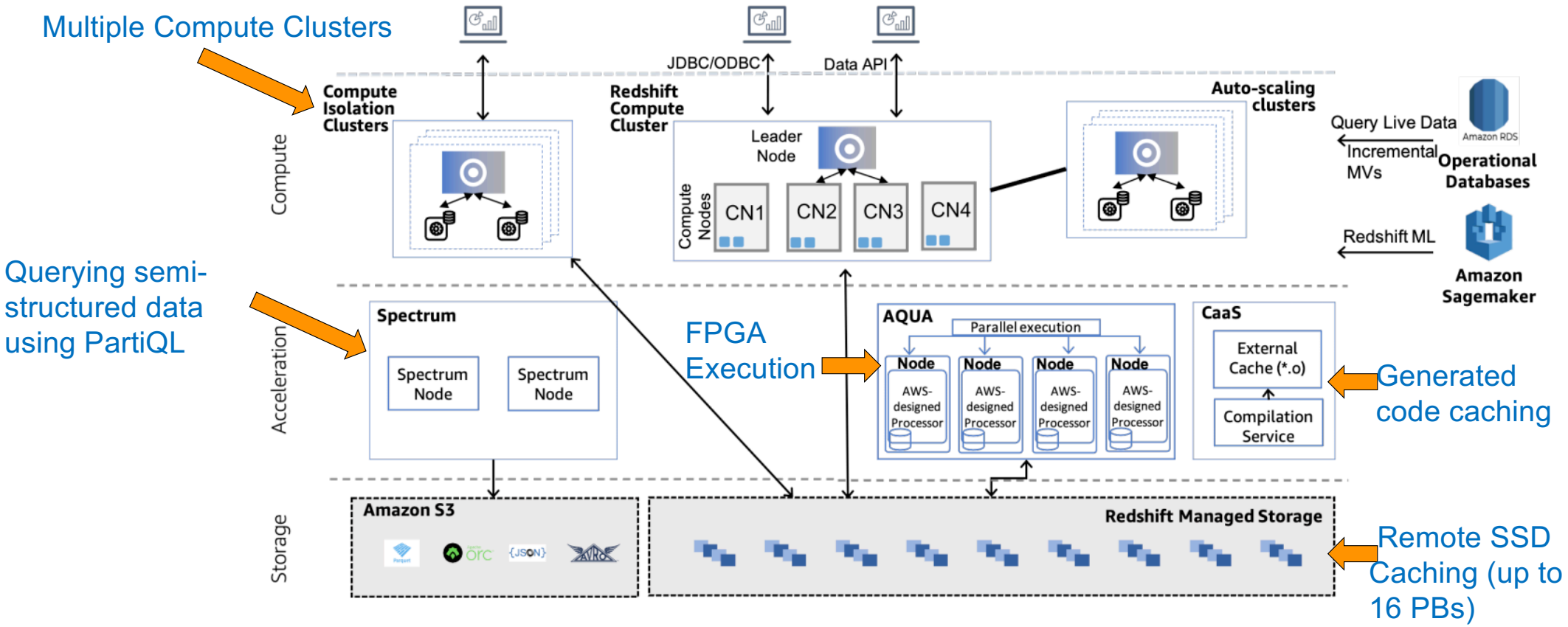
(1a) Case Study: Snowflake



- Feature Highlight
  - Cloud Services to manage VWs, workload, security, and metadata
  - Multiple Virtual Warehouses (Elastic Clusters of EC2 instances)
  - Cloud Storage, e.g., AWS S3 to store data as immutable table files
- Overview of Query Processing
  1. Parse the query using the cloud service, and generate and optimize query plan
  2. Execute the query in a virtual warehouse
  3. If the local cache is not hit, load the data from the cloud storage with pruning and computation pushdown

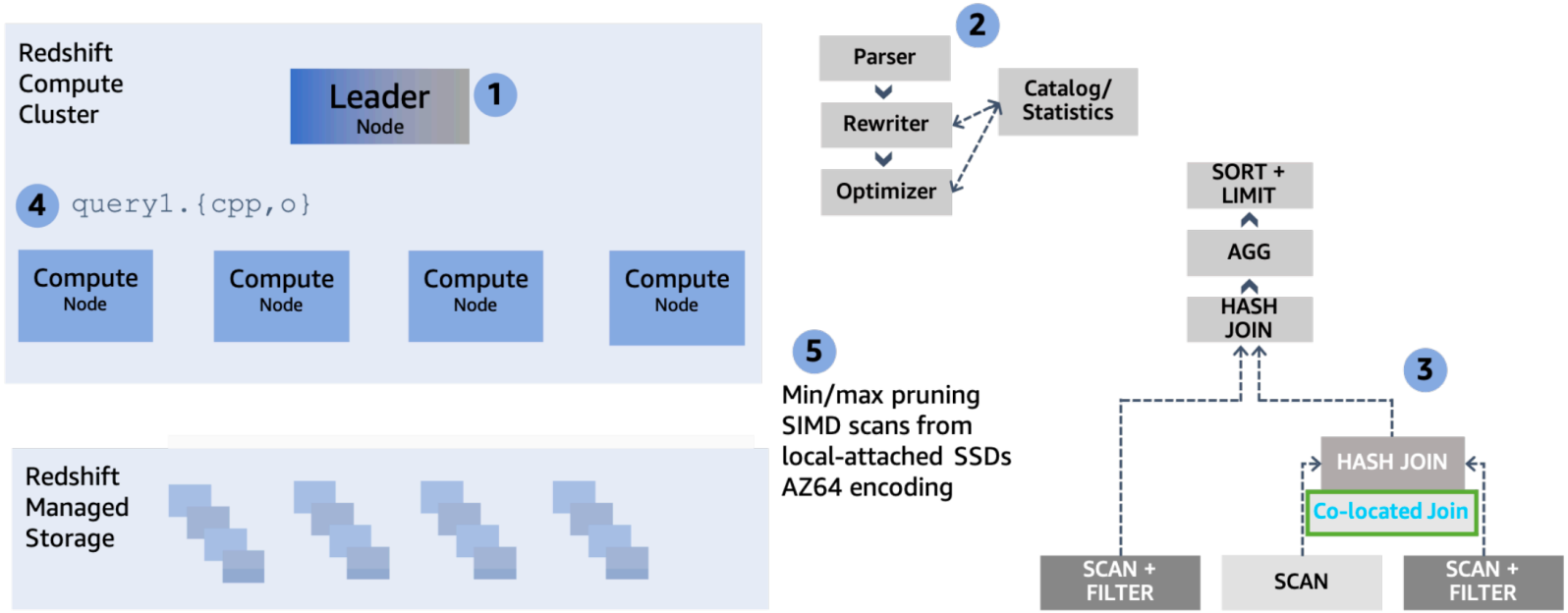
Dageville, Benoit, et al. “The snowflake elastic data warehouse.” In SIGMOD. 2016.

(1b) Case Study: Redshift



Nikos Armenatzoglou, et al. “Amazon Redshift Re-invented.” In SIGMOD. 2022.

(1b) Case Study: Redshift Query Flow



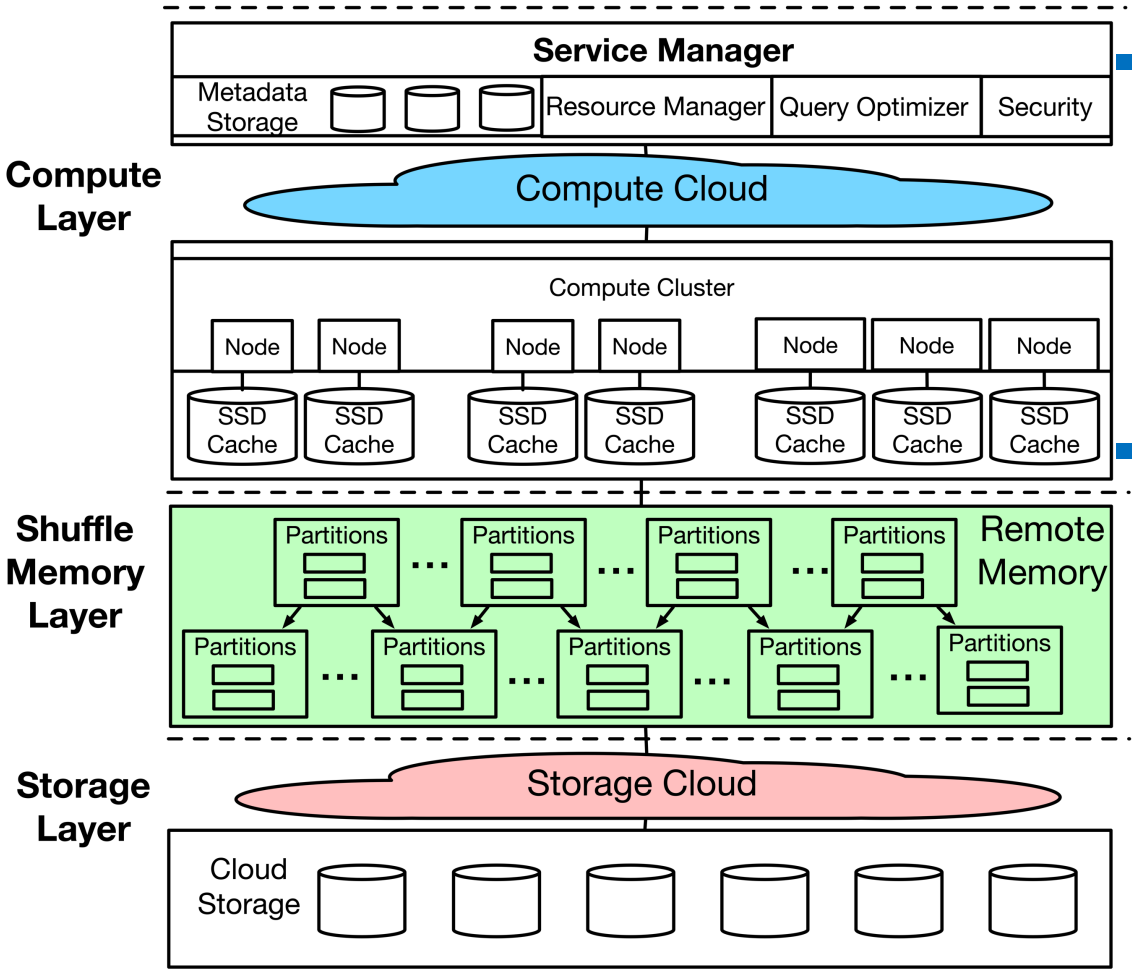
1. The leader node accepts a query
2. The query is parsed, rewritten, and optimized with the catalog statistics
3. The query plan is further optimized with the co-located join selection
4. The optimized plan is generated to C++ code, compiled and sent to compute nodes for execution
5. The columnar data is scanned from locally attached SSDs or is scanned from the cloud storage

Nikos Armenatzoglou, et al. “Amazon Redshift Re-invented.” In SIGMOD. 2022.

Comparison of Cloud Databases with Architecture (1)

Databases	Computation	Storage	Query Processing	Serverless
Snowflake	Isolated Virtual Warehouses (EC2 instances)	S3+Local Storage	Columnar Scan with Vectorized Engine	Instance-level Serverless Computing
Redshift	Isolated & Shared Compute Clusters + Acceleration Layer (Spectrum, AQUA)	S3+Redshift Managed Storage (RMS)	Columnar Scan with Code Generation	Instance-level Serverless Computing

(2) Disaggregated Compute-Memory-Storage Architecture



(2) Disaggregated Compute-Memory-Storage OLAP Architecture

Motivation:

- **Elasticity**: Storage and compute resources need to be scaled independently
- **Centralized Scheduling**: schedule the resources for better utilization
- **Complex workloads:** cope with the large intermediate results

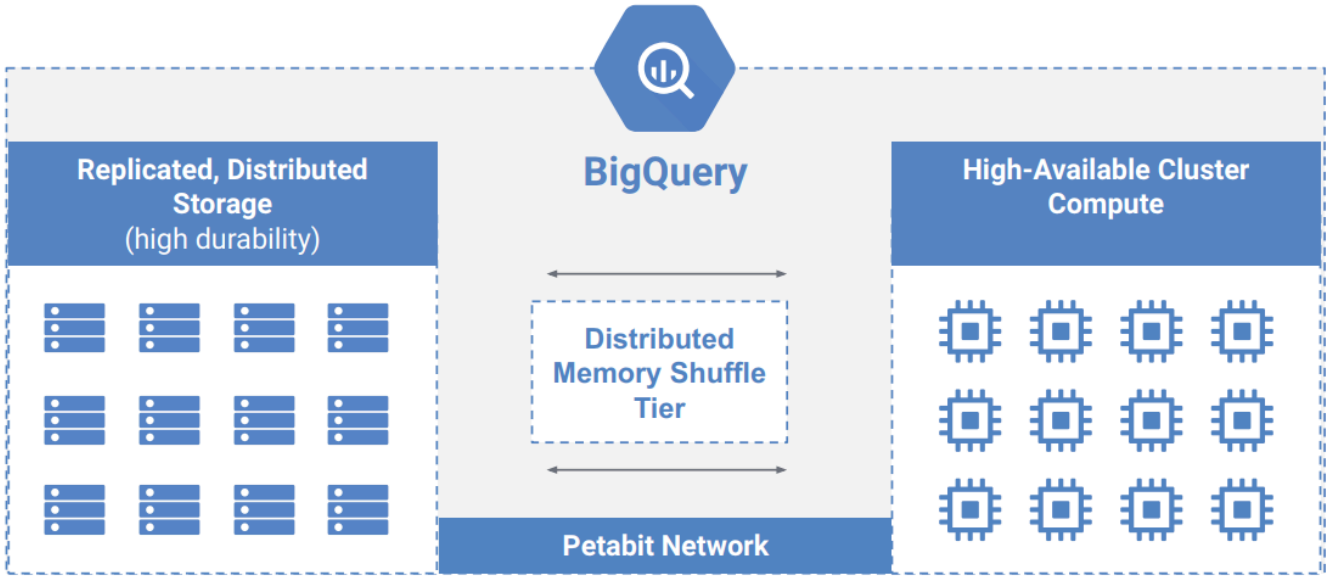
Key Features:

- Disaggregation of Compute and Storage
- Shuffle Memory Layer for speeding up joins
- Multi-tenancy and Serverless computing
- Local SSDs Caching
- Cloud Storage Service, e.g., AWS S3, Google Cloud Storage, Azure Blob Storage

(2) Disaggregated Compute-Memory-Storage Architecture

- **Advantages compared to On-Premise Share-Nothing OLAP Architectures:**
  - **Higher throughput.**
    - Shuffle memory tier reduces I/O cost by avoiding writing intermediate results to the disks.
  - **Higher resource utilization.**
    - Compute resources are virtualized and scheduled on demand.
  - **Better elasticity.**
    - The compute, memory, and storage resources can be scheduled individually.
- **Limitations:**
  - **Shuffle memory tier could incur a high cost**
    - Need to design efficient and effective pushdown and scheduling algorithms to reduce the data loaded to memory.

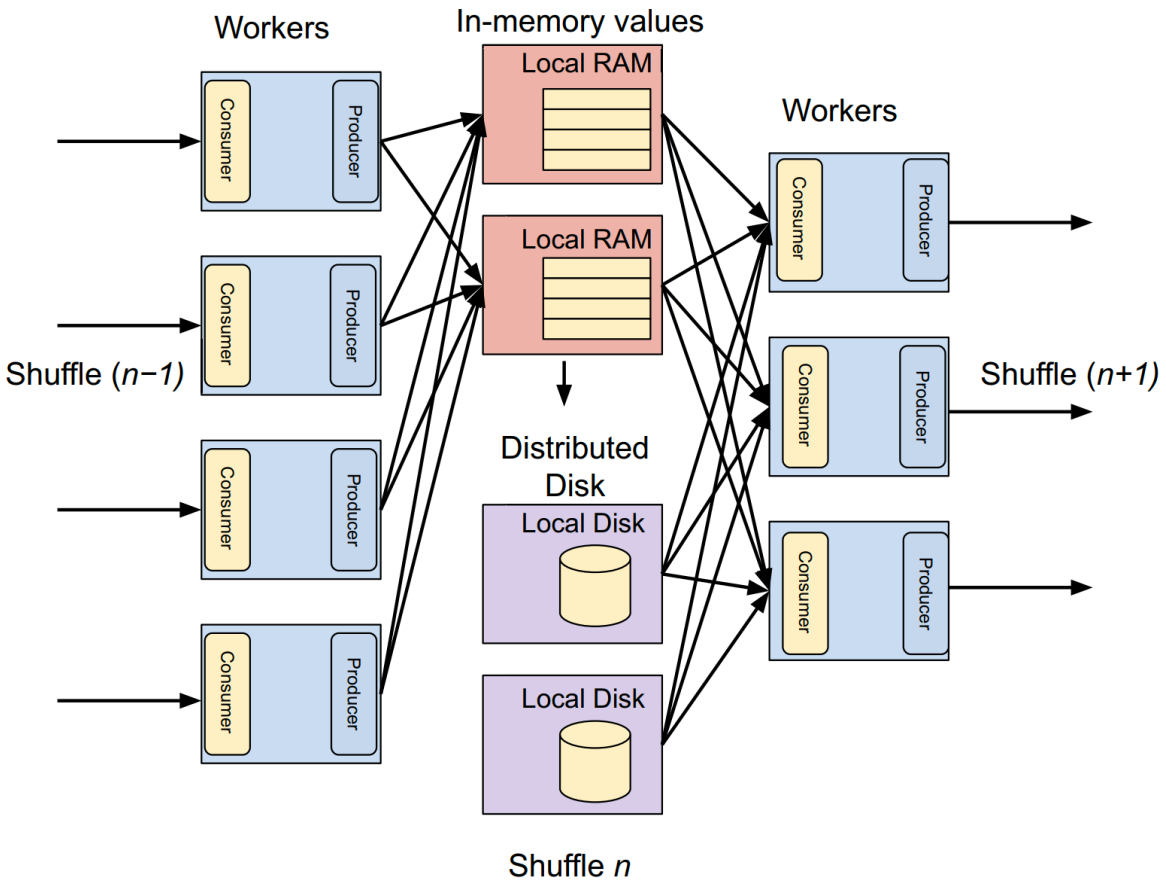
(2) Case Study: BigQuery



- Compute (Clusters + Memory Shuffle Tier) and Storage
- Dremel Query Engine with the support of semi-structured data querying
- Distributed Memory Shuffle Tier for Query Optimization
- Colossus Storage Clusters with Capacitor format

Sergey Melnik et al. Dremel: A Decade of Interactive SQL Analysis at Web Scale. PVLDB, 2020, 13(12):3461-3472.

(2) Case Study: BigQuery Shuffle Workflow



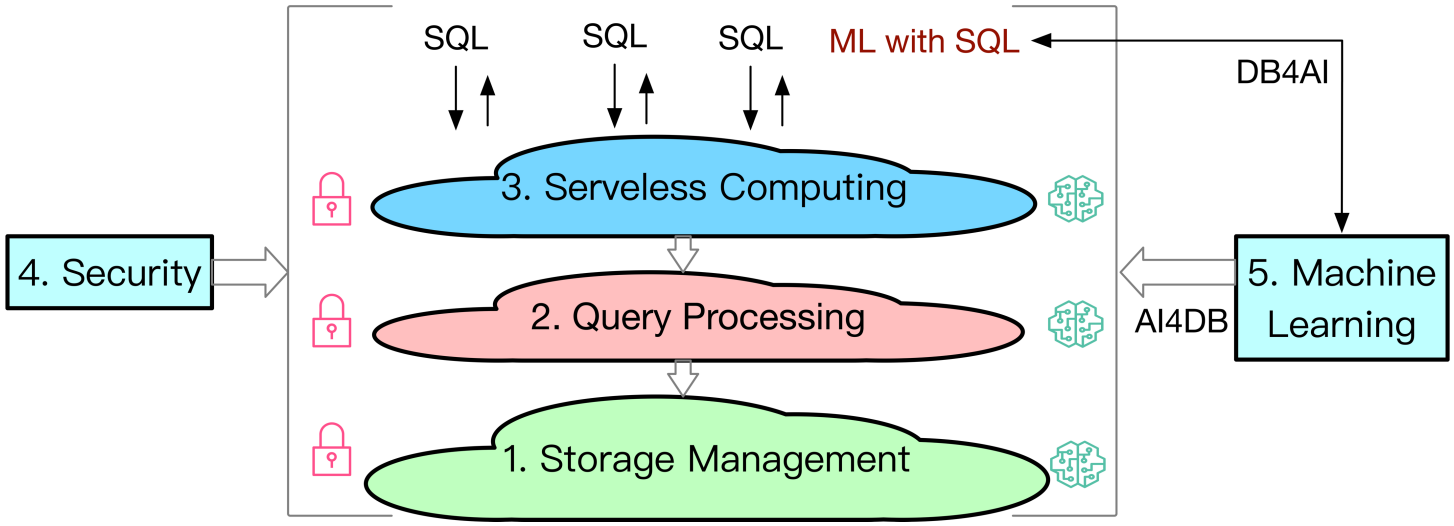
- Producer in each worker generates partitions and sends them to the in-memory nodes for shuffling
- Consumer combines the received partitions and performs the operations locally
- Large intermediate results can be spilled to the local disks

Sergey Melnik et al. Dremel: A Decade of Interactive SQL Analysis at Web Scale. PVLDB, 2020, 13(12):3461-3472.

Cloud-Native OLAP Techniques

Overview of Cloud-Native OLAP Techniques

1. **Storage Management:** Data organization in the cloud
2. **Query Processing:** Querying data with the local cache and cloud storage
3. **Serverless Computing:** Automated provisioning and scaling of resources
4. **Security:** Protect data from stealing and tampering in the cloud
5. **Machine Learning:** AI for cloud-native DBMS and cloud-native DBMS for AI



1. Storage Management in the Cloud

1) Data Organization

a. Metadata management

b. Tabular data management

c. Semi-structured data management

2) Data Placement

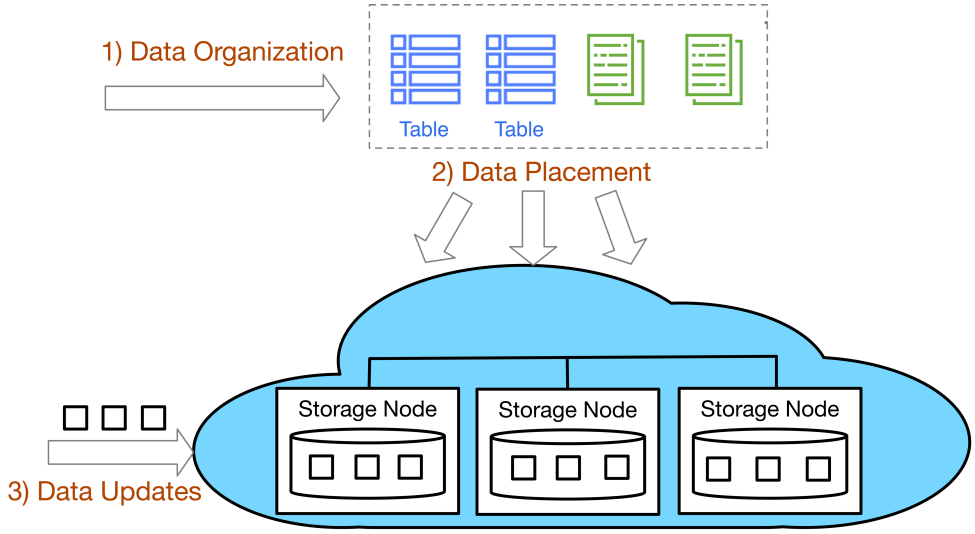
a. Data partitioning in the cloud

b. Data reshuffling in the cloud

3) Data Updates

a. Recreate objects with transaction updates

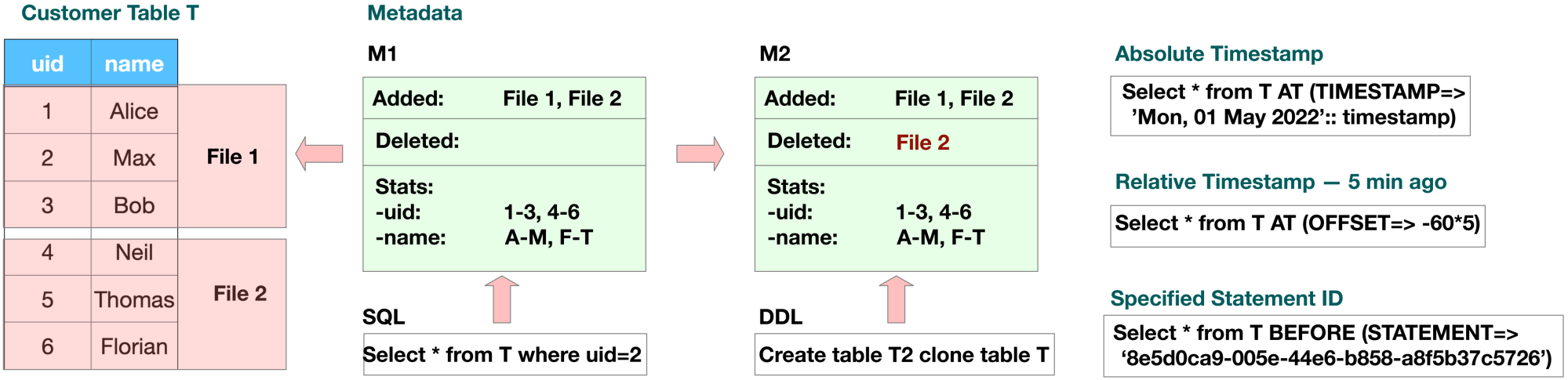
b. In-place update with internal table



1. Storage Management in the Cloud

1a) Metadata Management

- Metadata is stored in the layer of cloud service
- Contains information for schema, data version, location, statistics, logs, etc.
- Techniques: pruning, zero-copy cloning, and time traveling (MVCC)



(a) Pruning (b) Zero-cloning (c) Time-Traveling

1. Storage Management in the Cloud

1b) Data organization for tabular data

- Tables are partitioned into immutable files, i.e., micro-partitions
- Micro-partitions are replicated across multiple availability zones
- Tables are organized in columnar format, e.g., Parquet, and can be compressed to reduce the storage cost, e.g., run length encoding

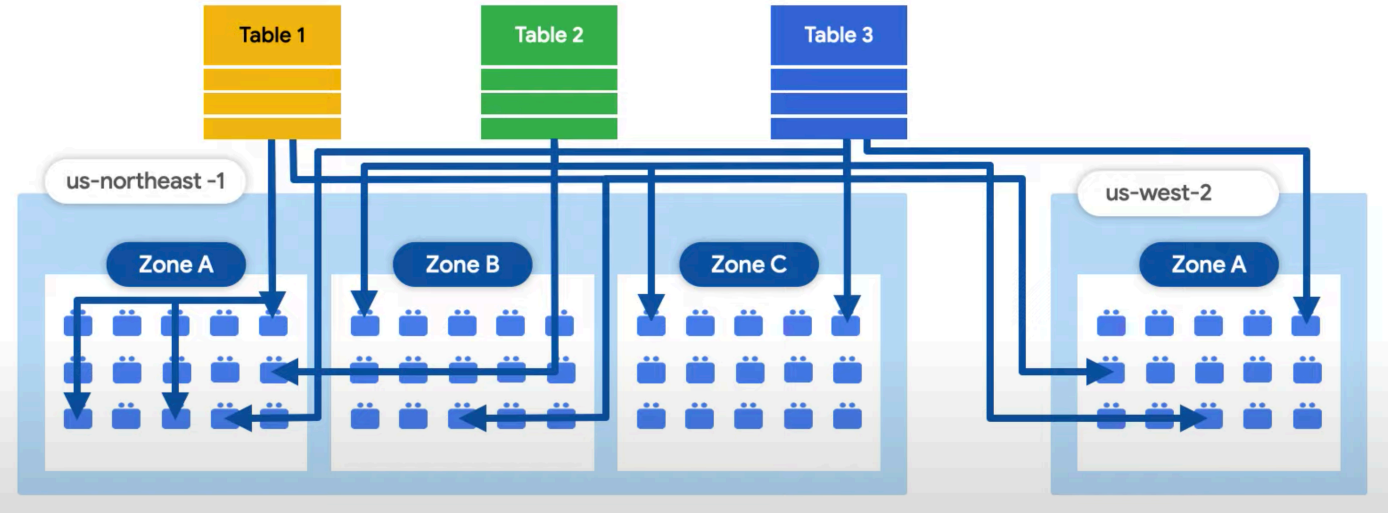


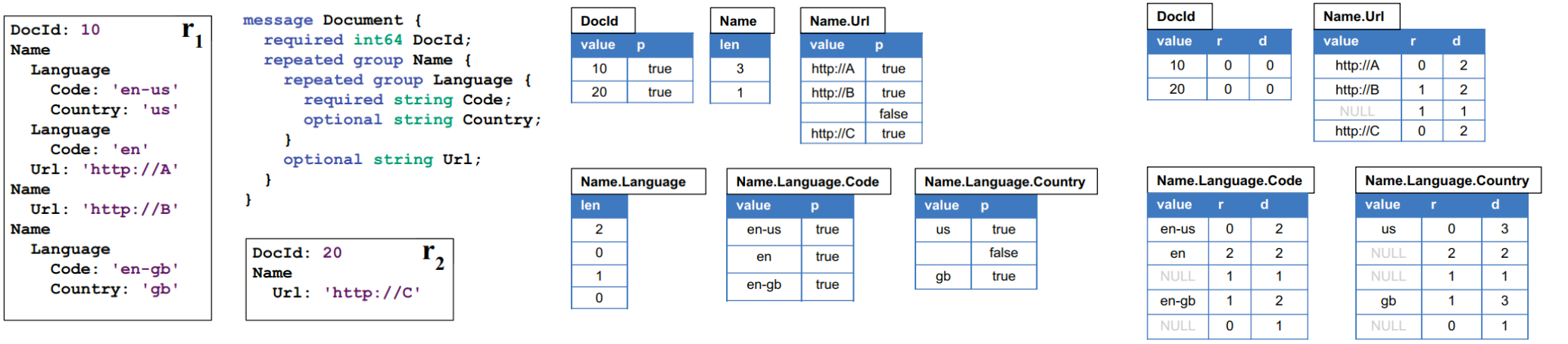
Figure from https://cloud.google.com/blog/topics/developers-practitioners/bigquery-admin-reference-guide-storage

1. Storage Management in the Cloud

1c) Data organization for semi-structured data

a. encoding the documents with length (len) and presence (p)

b. encoding the documents with repetition levels (r) and definition levels (d)



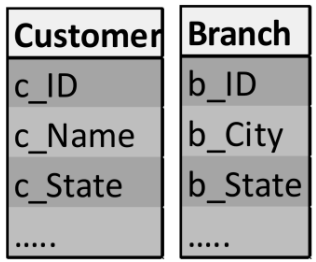
(a) Documents with a schema (b) Encoding with length and presence (c) Encoding with repetition and definition

Sergey Melnik et al. Dremel: A Decade of Interactive SQL Analysis at Web Scale. PVLDB, 2020, 13(12):3461-3472.

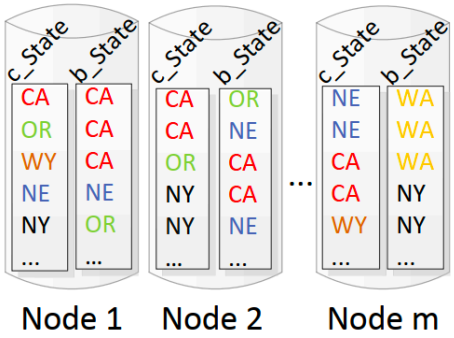
1. Storage Management in the Cloud

2a) The join graph approach for partition-key selection

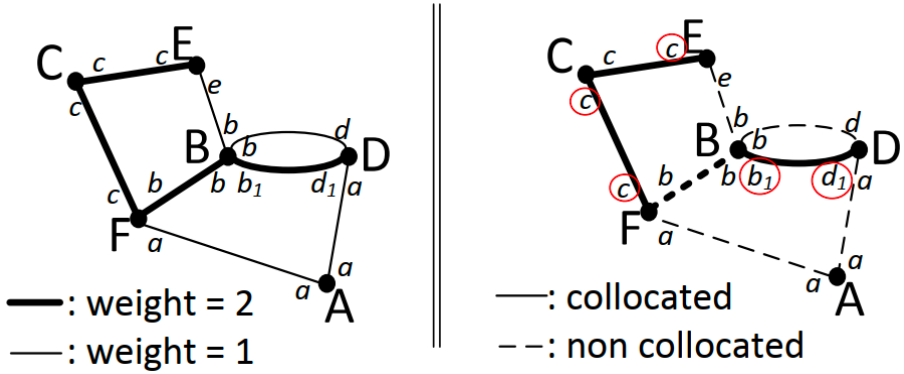
- Select the collocated join keys to minimize network communication
- Build a join multi-graph based on a query workload
- Use graph matching techniques and heuristics to select partition keys



(a) Schema.



(b) *"Even"* distribution.



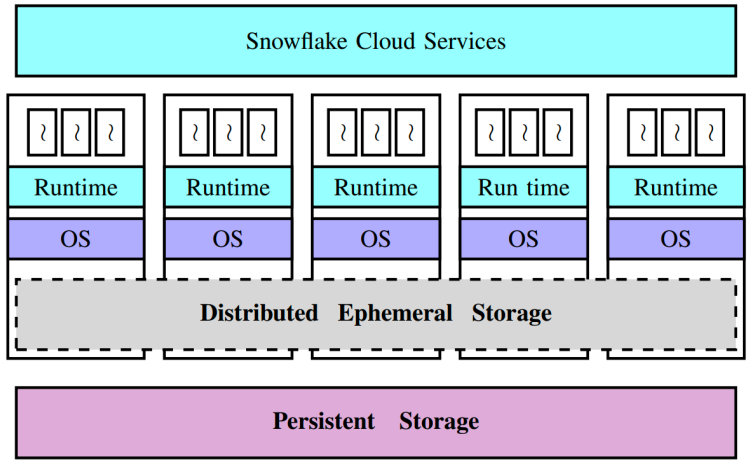
(c) The Join Graph Approach

Parchas, Panos, et al. "Fast and effective distribution-key recommendation for amazon redshift." *PVLDB Endowment* 13.12 (2020): 2411-2423.

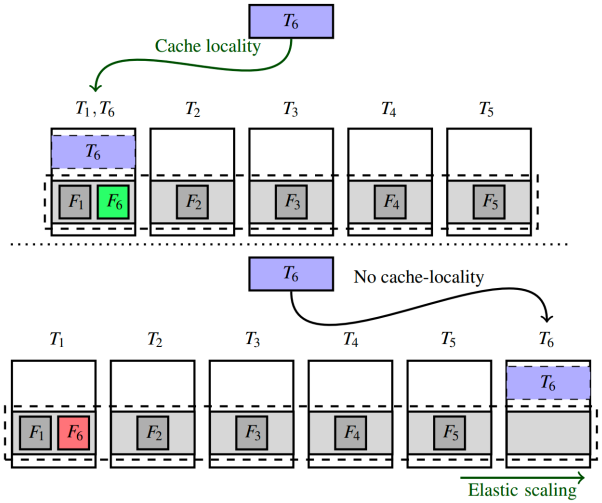
1. Storage Management in the Cloud

2b) Lazy consistent hashing during scaling of the ephemeral storage

- To achieve smooth scaling for the cloud databases
- When a new node is added to a VW, the consistent hashing is deferred
- When a related task is scheduled, the data is read and cached



(a) A Cloud Warehouse Architecture



(b) Lazy Consistent Hashing

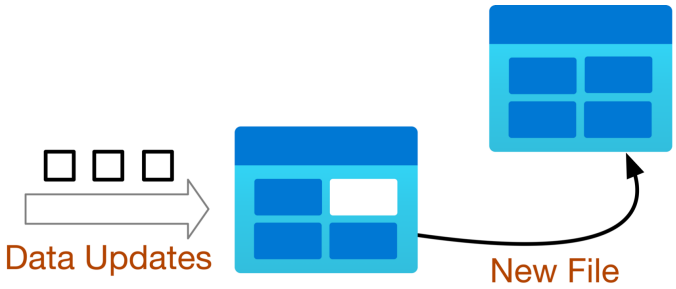
Vuppalapati, Midhul, et al. "Building an elastic query engine on disaggregated storage." *In NSDI* 2020.

1. Storage Management in the Cloud

3) Data Update

a) For blob storage, an update results in a creation of a new file due to the immutable table

b) mutable table with in-place updates, e.g., Snowflake's hybrid table in Unistore, supporting indexing, single-row retrieval, loading data from immutable tables



(a) Creation of a new object file

(b) In-place updates

2. Query Processing in the Cloud

1. Columnar Scan with Caching/Pushdown

- ❑ Caching: Snowflake, Redshift
- ❑ Pushdown Computation: PushdownDB
- ❑ Caching and Pushdown : FlexPushdownDB (FPDB)

2. Columnar Scan with Shuffle Memory Tier

- ❑ Multi-stage parallel columnar scan with a shuffle memory tier.
- ❑ E.g., BigQuery

2. Query Processing in the Cloud

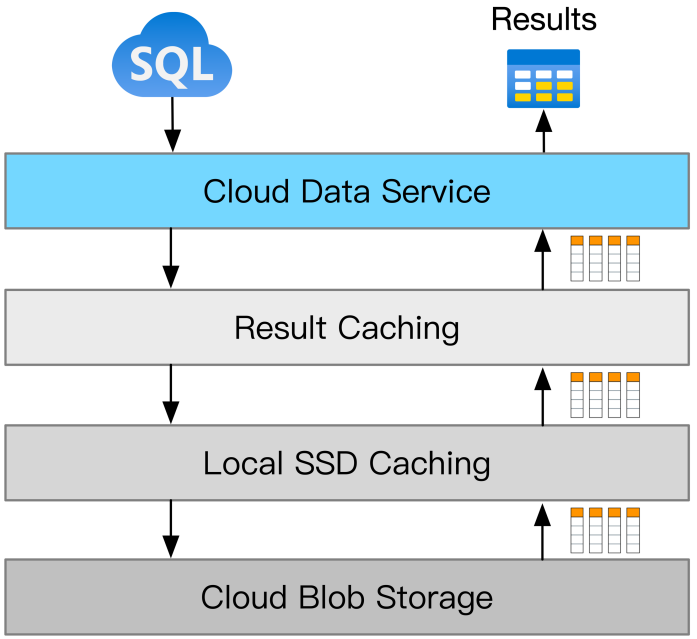
1a) Columnar Scan with Caching

i. Given a query, it searches for the query results in the cache

ii. If the cache is not hit, it searches for the results from the local SSD cache and processes the query with columnar scans in the local cluster

iii. If the local cache is not hit, it processes the data loading from cloud blob storage

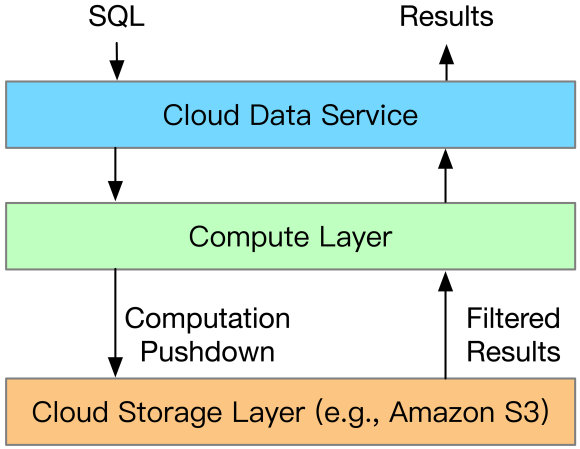
iv. File stealing from neighbors for load balance



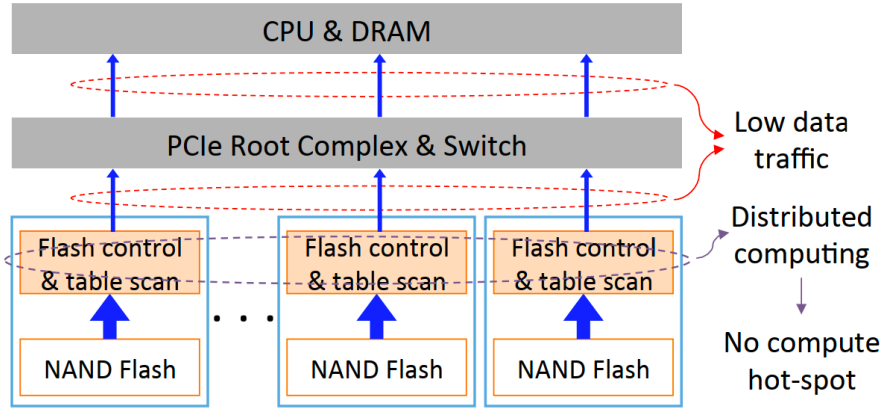
2. Query Processing in the Cloud

1b) Columnar Scan with Pushdown

- Pushdown computation to the cloud storage, e.g., Amazon S3 Select, Select API can be extended to support index scan, hash-join, group by, top-k
- Pushdown to the computational storage drive, e.g., FPGA-enabled table scan



(a) Pushdown to the cloud storage



(b) Pushdown to the computational storage

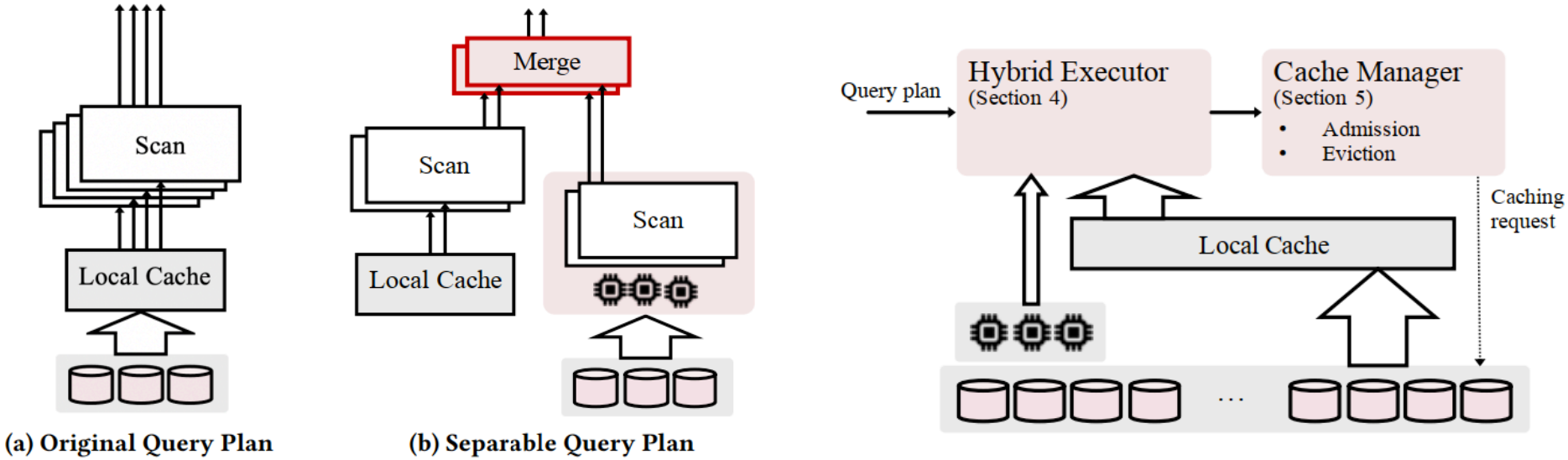
Yu, Xiangyao, et al. "PushdownDB: Accelerating a DBMS using S3 computation." *2020 ICDE*, 2020.

Cao, Wei, et al. "POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database." *FAST,* 2020.

2. Query Processing in the Cloud

1c) Columnar Scan with Caching and Pushdown

- Process the query with caching and computation pushdown simultaneously
- Enhance the cache replacement strategy, e.g., weighted LFU



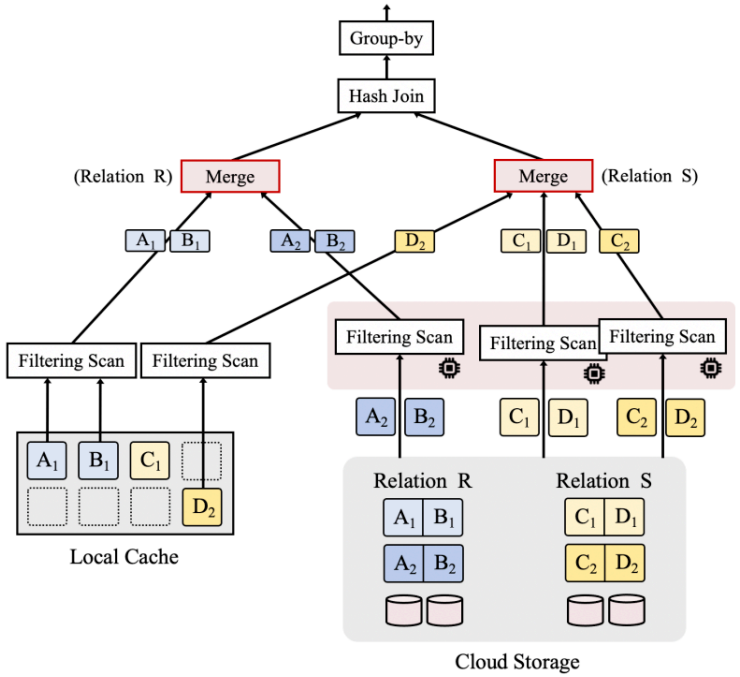
Yang, Yifei, et al. "Flexpushdowndb: Hybrid pushdown and caching in a cloud DBMS." *PVLDB,* 14.11 (2021): 2101-2113.

2. Query Processing in the Cloud

1c) Columnar Scan with Caching and Pushdown

- Hybrid Query Execution : (1) Local Cache is more efficient than Pushdown; (2) Pushdown is more efficient than loading all data from the cloud storage
- Two relations R(A,B) and S(C,D), each attribute has two partitions

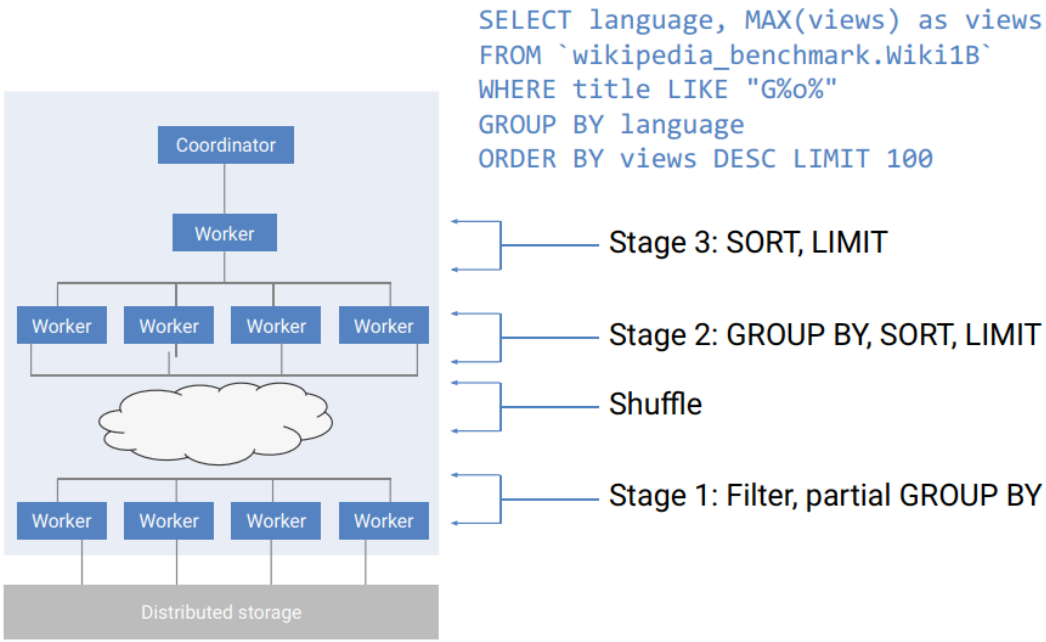
```
SELECT R.B, Sum(S.D)
FROM R, S
WHERE R.A=S.C and R.B>10
            and S.D>20
GROUP BY R.B
```



2. Query Processing in the Cloud

2. Columnar Scan with Shuffle Memory Tier

- Use a shuffle memory tier without writing the intermediate results to disks
- The query is executed by multiple workers with multiple stages



Melnik, Sergey, et al. "Dremel: A decade of interactive SQL analysis at web scale." *PVLDB,* 13.12 (2020): 3461-3472.

3. Serverless Computing for Queries

Motivation: tenants issue the queries in the cloud without caring about the resource provisioning and can pay for the resources in the query granularity

Two main approaches are as follows:

1. **Serverless Databases:** rely on the cloud SQL engine and storage to execute the queries with dynamic resource provisioning; the database service can pause for the idle period and resume when a query comes in
2. **Serverless Functions + Cloud Storage:** rely on Function-as-a-Service (Faas) and the cloud storage to perform the queries with on-demand resources

AWS Lambda

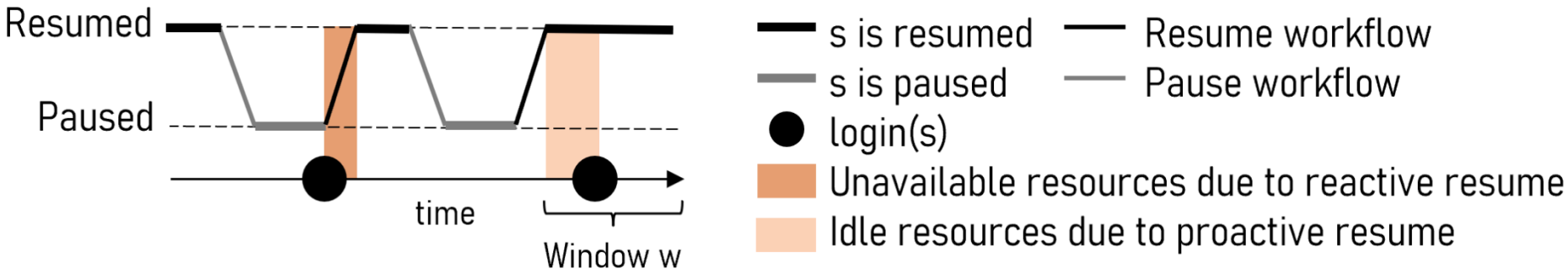
Google Cloud Functions

...

3. Serverless Computing for Queries

1. Serverless Database with Dynamic Resource Scheduling

- **Challenge: starting a database is expensive after a pause period**
- **Solution: predict the pause/resume patterns and proactively resume the resources for each database**

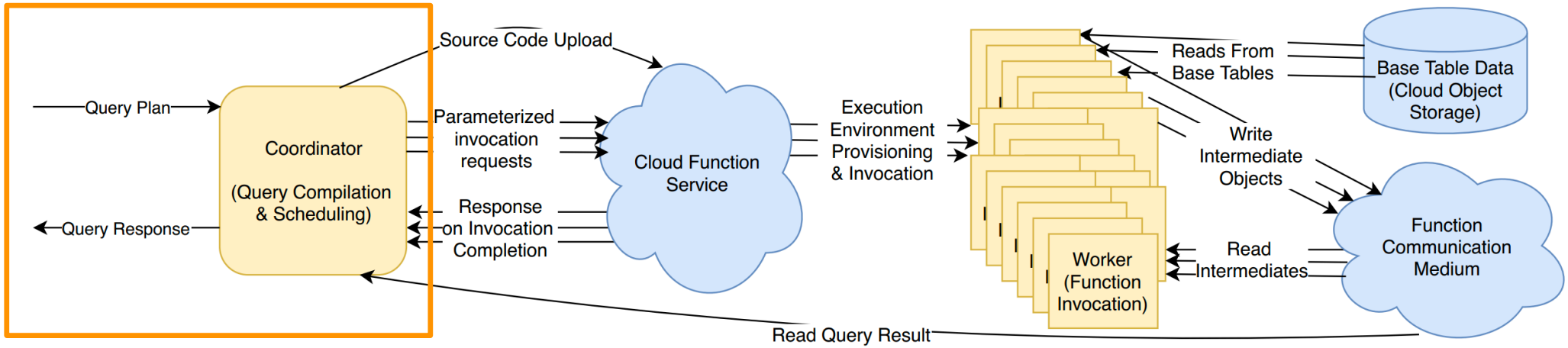


Poppe, Olga, et al. "Moneyball: proactive auto-scaling in Microsoft Azure SQL database serverless." *Proceedings of the VLDB Endowment* 15.6 (2022): 1279-1287.

3. Serverless Computing for Queries

2. Serverless Functions + Cloud Storage

- **Two Challenges: (1) functions are stateless; (2) stragglers increase the overall latency of the parallel query processing**
- **Solutions: use cloud storage to exchange states; use tuned models to detect stragglers and invoke functions with duplicate computation;**

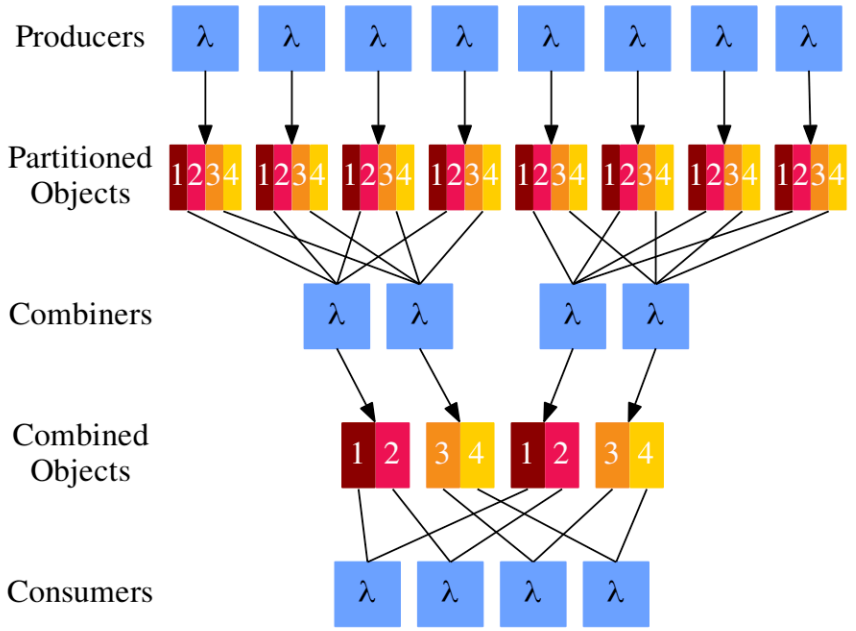


Perron, Matthew, et al. "Starling: A scalable query engine on cloud functions." *In SIGMOD*. 2020.

3. Serverless Computing for Queries

2. Query Processing based on Serverless Functions and Cloud Storage

- 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

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

Summary of Serverless Computing for Queries

Category	Database/ Prototype	Main Approach	Scaling	Pricing Model
Serverless Database	Azure SQL, Athena, BigQuery	Stateful SQL Engine + Auto-pausing and resuming mechanism	Scaling the resources with more CPU, memory, or stand-by nodes	Pay for active service with min-max bound
Function as a Service (FaaS)	Starling	Stateless Functions + Stateful Cloud Storage	Scaling the resources by invoking more function tasks	Pay for used functions and storage

4. Security in the Cloud

1. Software-based Data Protection

- E.g., Snowflake, Redshift
- Pros: high scalability and throughput, low cost
- Cons: decryption for query processing

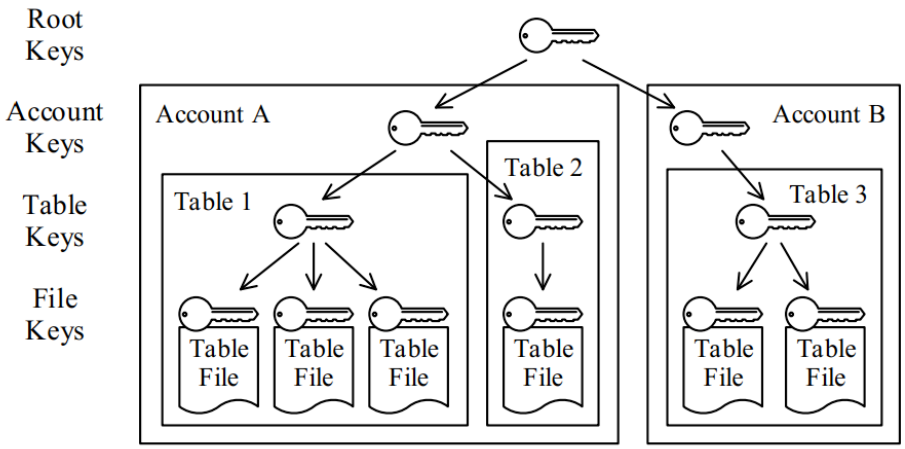
2. Hardware-based Data Protection

- E.g., Azure SQL
- Pros: high end-to-end security
- Cons: low scalability and throughput, high cost

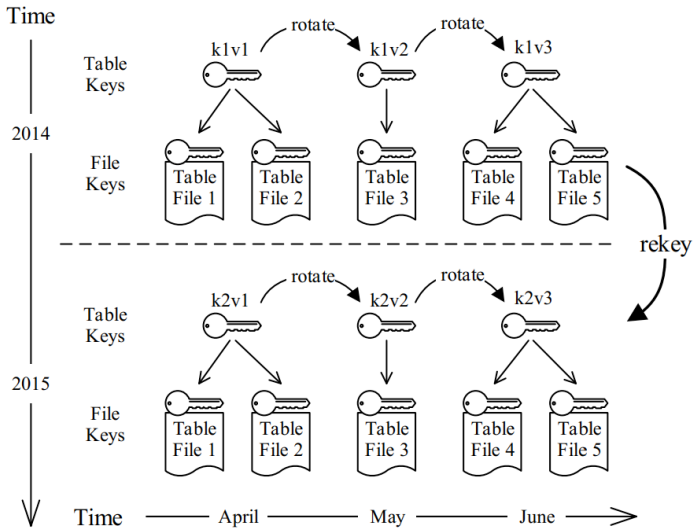
4. Security in the Cloud

1. Software-based Data Protection

- **Core Idea:** **encryption keys are automatically rotated and re-encrypted**
- **Challenges:** **data is decrypted for query processing; the cloud vendors may be untrusted**



(a) Encryption Key Hierarchy



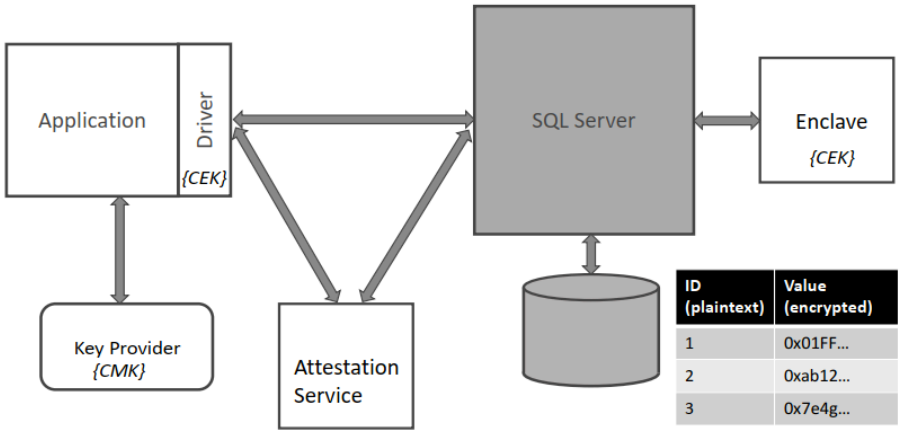
(b) Key Life Cycle

Dageville, Benoit, et al. “The snowflake elastic data warehouse.” In SIGMOD. 2016.

4. Security in the Cloud

2. Hardware-based Data Protection

- **Core Idea: database systems and cloud providers are untrusted; leverage customized hardware, e.g., Enclave, for data protection; bring-your-own-keys**
- **Challenges: computation over ciphertext directly; improve the efficiency of enclave**



the design of Enclave-based protection in Azure SQL

Step 1: Application issues a query "select * from T where value = @v"

Step 2: Driver encrypts the parameter @v and sent to the DBMS with attestation service

Step 3: DBMS fetches the data and invokes the enclave for evaluation

Step 4: Enclave decrypts the data to plaintext and evaluates the filter

Antonopoulos, Panagiotis, et al. "Azure SQL database always encrypted." *In SIGMOD*. 2020.

5. Cloud-Native Databases with Machine Learning

1. ML-Enabled Cloud-Native Databases

- ML-Enabled Workload Management
- RL-Enabled Partition-Key Advisor
- And many more: knob tuning, index tuning, root cause diagnose, etc.

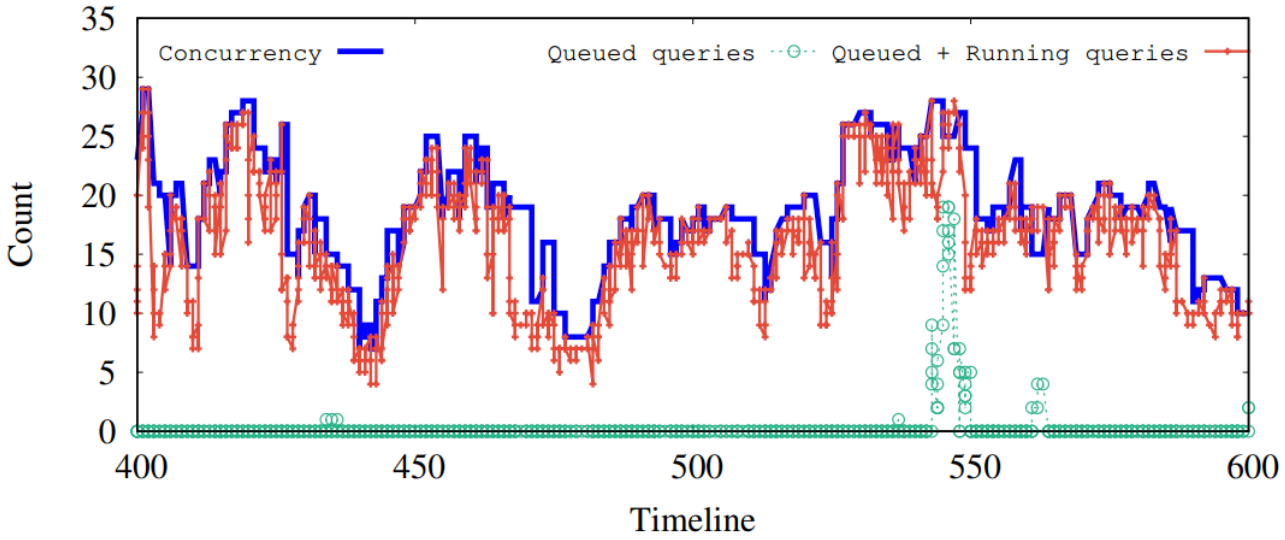
2. Cloud-Native Database for ML

- SQL-enabled ML pipeline
- Cloud Database with AutoML

5. Cloud-Native Databases with Machine Learning

ML-Enabled Cloud-Native Databases: Redshift Workload Management

- **Core Idea:** **tune the workload concurrency by predicting the memory consumption and execution time for the workload**
- **Challenges:** **schedule the workload; migrate to new access pattern**
- **Solution:** **Redshift AutoWLM; trains an XGBOOST model for each cluster**

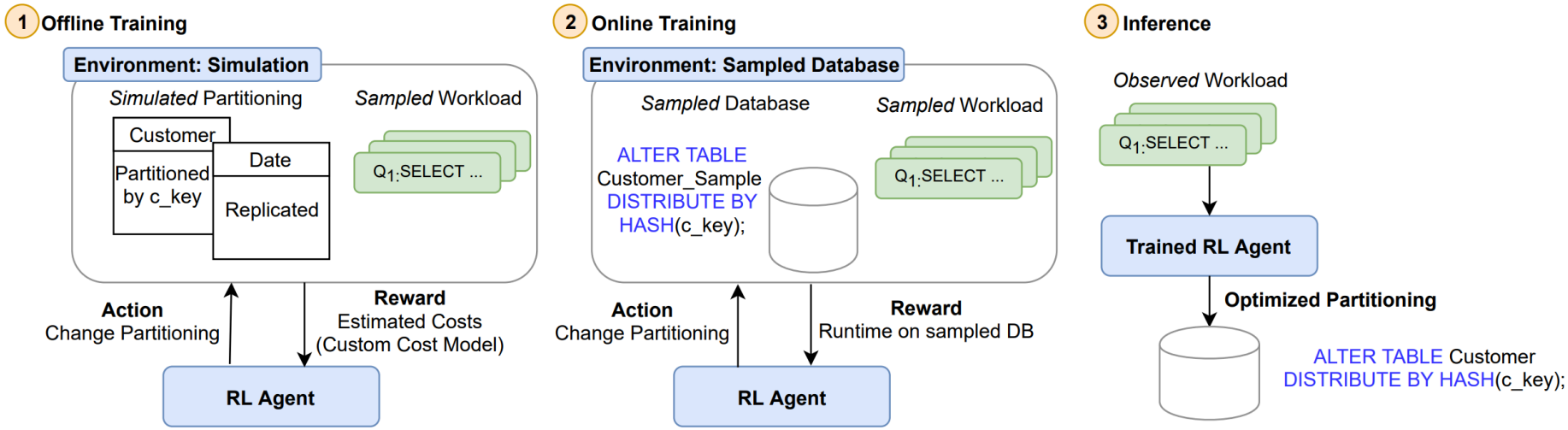


Nikos Armenatzoglou, et al. "Amazon Redshift Re-invented." In SIGMOD. 2022.

5. Cloud-Native Databases with Machine Learning

Partition-Key Advisor for Cloud Databases

- **Core Idea:** exploring column combinations as partition keys and learning with RL
- **Challenges:** characterize partition features; migrate models to new workloads
- **Solution:** (1) extract partition features as [tables, query frequencies, foreign keys] and use DQN to partition the tables for a workload; (2) train a cluster of DQN models on typical workloads and pick one with the most similar features;



Hilprecht, Benjamin, Carsten Binnig, and Uwe Röhm. "Learning a partitioning advisor for cloud databases." *SIGMOD*. 2020.

5. Cloud-Native Databases with Machine Learning

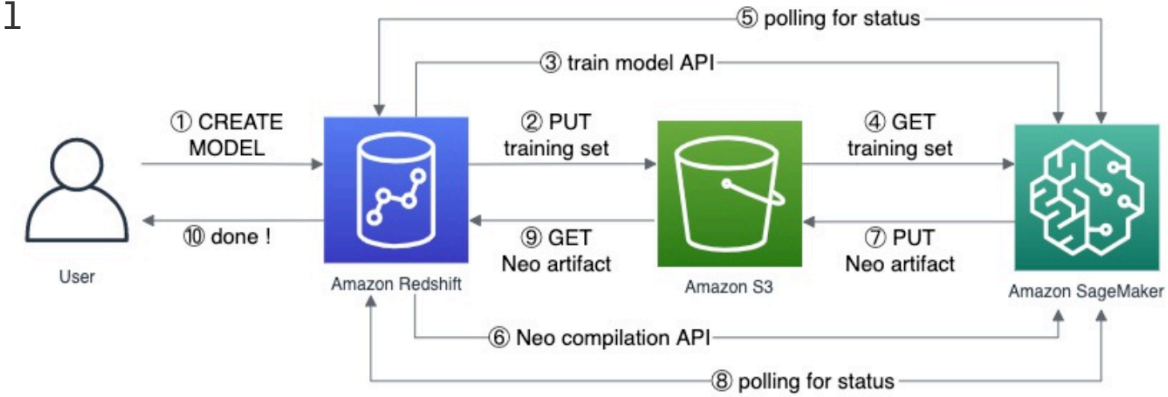
2. Cloud-Native Database for ML

- **Core Idea: (1) SQL-enabled machine learning in cloud databases; (2) bring the model to the data; (3) AutoML by the cloud providers, e.g., model selection, training and tuning**
- **Challenges: SLA-aware in-database ML; flexibility of SQL-based ML pipeline**

```
CREATE MODEL demo_ml.customer_churn_model
FROM
(SELECT state, area_code,
average_daily_spend,
average_daily_cases,
churn
FROM demo_ml.customer_activity
WHERE record_date < '2022-01-01' )

TARGET churn FUNCTION predict_customer_churn
```

(a) CREATE MODEL for predicting customer churn



(b) Redshift with SageMaker

Nikos Armenatzoglou, et al. "Amazon Redshift Re-invented." In SIGMOD. 2022.

Open Problems and Opportunities

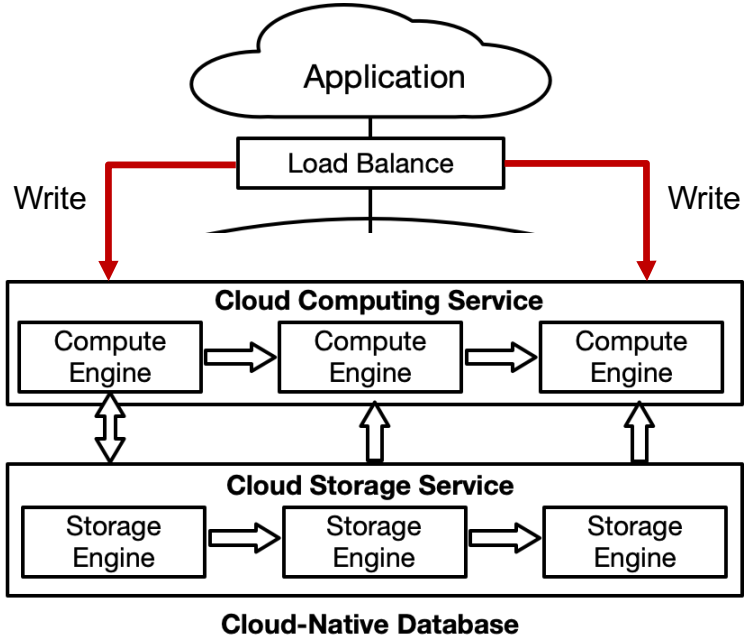
Multi-Write Architecture in the Cloud

- **Call for Multi-Write Solutions for Cloud-Native Databases**
  - **Multi-Write Protocol** (**How to handle write conflicts in the cloud**)
  - **Data Consistency** (**How to keep data consistency for dirty caches**)
  - **Log Management** (**How to replay and update the logs**)

Multi-Write Protocol? ⇨

Data Consistency? ⇨

Log Management? ⇨



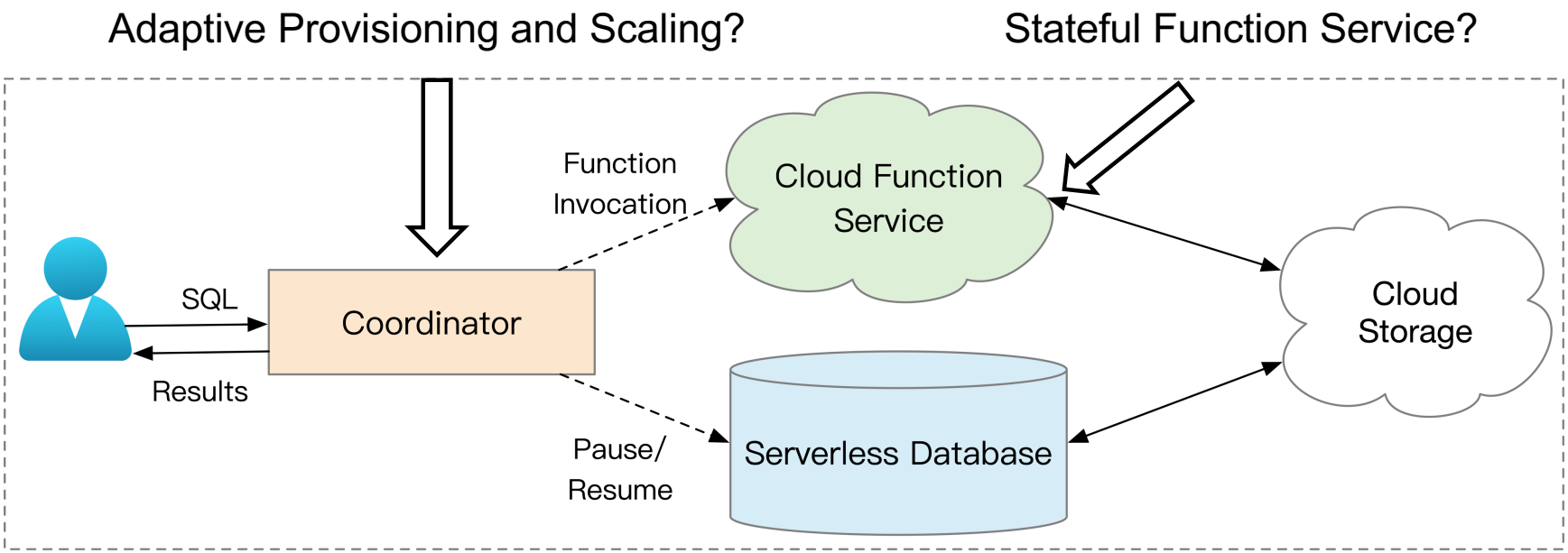
CRDT Protocol?

Memory Disaggregation?

Paxos Protocol?

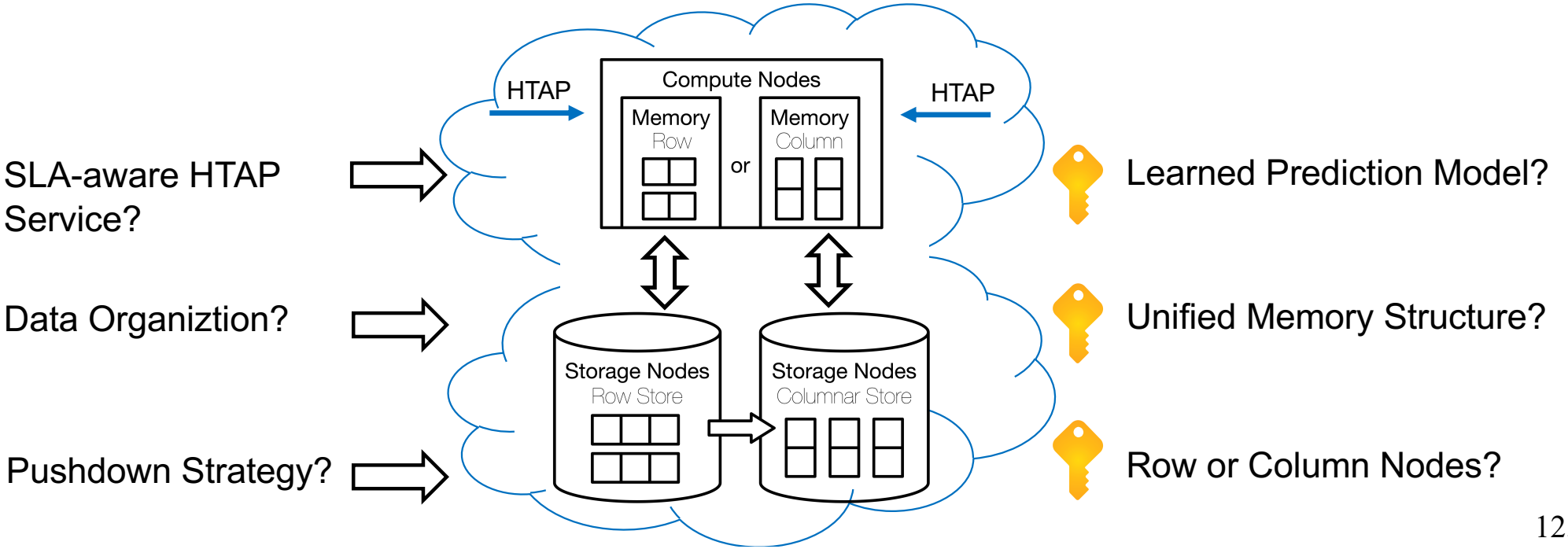
Fine-grained Serverless

- **Serverless Computing**
  - **Stateful Function Service** (**How to exchange the intermediate results**)
  - **Adaptive Provisioning and Scaling** (**How to schedule the resources adaptively**)



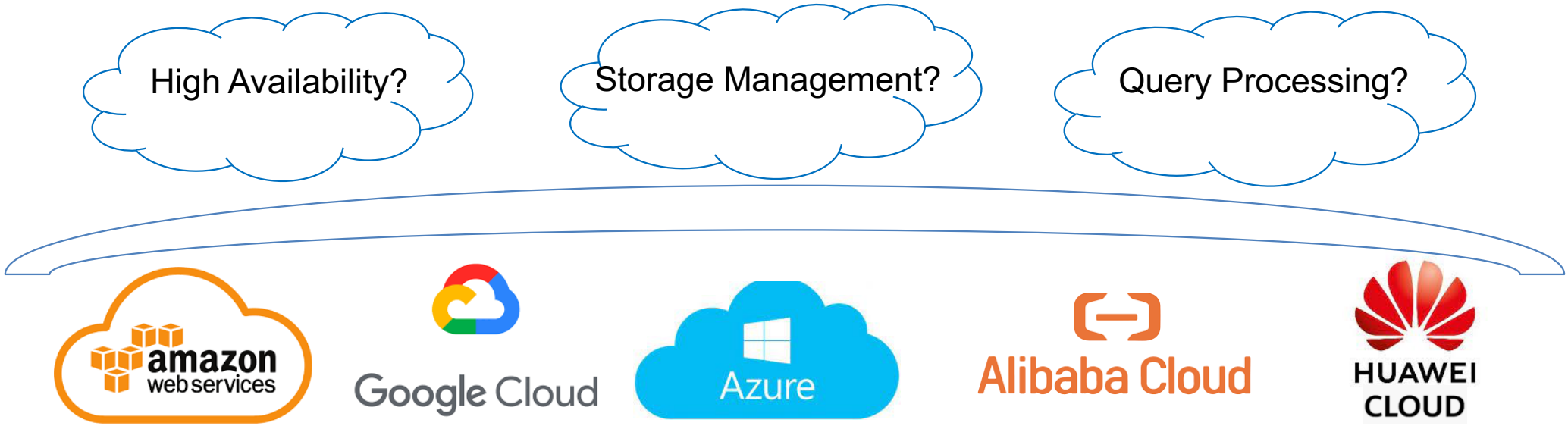
Cloud-Native HTAP Database

- **Call for Cloud-Native HTAP databases**
  - **SLA-aware HTAP service (How to balance performance, freshness, cost)**
  - **Data Organization for HTAP (How to organize the cloud data for HTAP)**
  - **Pushdown Strategy for HTAP (Pushdown operators to row or column nodes)**



Multi-Cloud Database

- **Call for Multi-Cloud Databases**
  - **High Availability** (**How to handle the failures and migrate data in multi-cloud**)
  - **Storage Management** (**How to organize and store the data in multi-cloud**)
  - **Query Processing** (**How to perform the query in multi-cloud**)



Q & A

Thanks for your listening!