Cloud-Based Data Processing

OLTP in the cloud (DBaaS)

Jana Giceva
Traditional OLTP database design

- Monolithic relational database stack did not change much in the last 30-40 years

<table>
<thead>
<tr>
<th>Application</th>
<th>SQL</th>
<th>Transactions</th>
<th>Caching</th>
<th>Logging</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SQL layer: compiler, optimizer, query processing, access methods</td>
<td>Transaction layer: transaction manager, locking, undo management</td>
<td>Caching: buffer cache</td>
<td>Logging: redo logging, crash recovery, back-up/restore</td>
<td>Storage: durable storage</td>
</tr>
</tbody>
</table>
Monolithic relational database stack did not change much in the last 30-40 years

- Works for on premise deployments:
  - In-memory databases (scale-up servers)
  - Fast storage solutions (SSDs, SAN)
  - Expensive reliable hardware
  - Managed by a DB administrator
    - Control software update cycles and plan downtimes
Different conventional approaches for scaling out databases:

**Sharding**
- coupled at the application layer

**Shared Nothing**
- coupled at the SQL layer

**Shared Disk**
- coupled at the caching and storage layer

But the tightly coupled monolithic stack remains.
Failures happen all the time
- Permanent vs transient failures
- Slow or overloaded components (nodes)

Geo-distributed datacenters

Different requirements and expectations
- Cost, elasticity, availability, throughput, tail latency at scale (see next slide)

Fast and expensive vs. slow and cheap storage
- Trade-off high-performance vs. durability and scalability

Different guarantees of ephemeral and persistent storage media
OLTP in the cloud (DBaaS)

Expectations:

- Durability guarantees
- Support large databases (e.g., 100TB)
- Scalability (e.g., workload, dataset size, etc.)
- Fault tolerance
- High availability (e.g., 99.999% availability)
- High performance (low tail latency, high throughput)
- Elastic with workload demand (pay-as-you-go)
- Small blast radius (the impact of a failure)
- Economic (good performance/cost ratio)
- Data locality guarantees (e.g., GDPR compliance)
Conflicting goals:
- Support both large databases (e.g., 100TB data) AND provide high availability (i.e., small mean-time-to-recovery) is contradictory.
- Small mean-time-to-recovery requires small data.

Many challenging questions:
- How do we make the system elastic (with load and data, respectively)?
- How do we ensure fast recovery?
- How do we provide high availability?
- How do we achieve good performance?
- How do we keep replicas consistent?
- How do we optimize for the new bottlenecks (e.g., reduce write amplification)?
- How do we efficiently execute distributed transactions in a geo-distributed environment?

Not possible with traditional monolithic database architectures.
There are four prominent DBaaS architectures offered today on the major cloud providers:

1. **Non-partitioned shared-nothing log replicated state machine**
   - Microsoft’s HADR for SQL Server

2. **Non-partitioned shared-data**
   - Disaggregated compute-storage: AWS Aurora, GCP AlloyDB
   - Disaggregated compute-log-storage: Azure Socrates, Huawei Taurus, Alibaba POLARDB
   - Disaggregated compute-buffer-log-storage: Alibaba PolarDB Serverless

3. **Partitioned shared-nothing log replicated state machine**
   - Requires distributed transactions
   - Spanner, CockroachDB, POLARDB-X

4. **Tightly coupled nodes over fast interconnect**
   - Oracle’s RAC and Exadata
Non-partitioned shared-nothing log replicated state machine
Non-partitioned, shared-nothing LRST

Today’s solution for SQL Server offered in Microsoft Azure

- Primary node processes all update transactions
- Ships update logs to all follower replicas
- Primary replica periodically backs-up data to Azure’s storage service (XStore)
  - A log is backed up every 5 min
  - A full back-up is done once per week
- Follower replicas process read-only queries
- Needs four nodes (1 primary, 3 secondary) to guarantee high availability and durability.
Advantages

- **The service is stable and mature**
  - The HADR architecture has been used successfully for million of databases in Azure

- **Has high performance**
  - Every compute node has a full, local copy of the database
The size of a database cannot grow beyond the storage capacity of a single node (machine).

Replicas required for resilience and load are coupled.

O(size-of-data) operations create issues.
- The cost of seeding a new node (replica) is linear with the size of the database
- Back-up / restore
- Scale-up and down
- Are all examples of operations whose cost grows linearly with the size of the database.

A special case occurs with long-running transactions when the log grows beyond the storage capacity of the machine and cannot be truncated until the long-running transaction commits.

Hence, today SQL DB limits the size of the database to 4 TB.
Non-partitioned shared-data log replicated state machine
Key insight: decompose RDBMS functionality

- Decompose monolithic database stack into its fundamental building blocks.

**Diagram:**

```
  Application
  SQL
  Transactions
  Caching
  Logging
  Storage

  Compute service layer executed by nodes optimized for query processing. Uses the **storage service** over the network.

  Purpose-built, scale-out, self-healing, multi-tenant database-optimized **storage service**.
```
Decoupling compute and storage

- **Advantages:**
  - Easy to scale up/down with the load demand
    - Simply add / remove compute nodes.
  - Easy to scale up/down the size of the database
  - Easy to replace misbehaving or unreachable compute nodes
  - Easy to fail over from a writer to a replica
    - The expensive part of the operation is handled by the storage layer.
  - **Replicating the storage** across multiple nodes not coupled with workload demand
Traditionally the interaction between the compute and storage layers has many inefficiencies

- RDBMSs organize data in pages.
  - Page modifications are periodically flushed to disk.
- Page modifications are also recorded in a redo log.
  - The log is written to disk in a continuous stream.

- Lot of redundant writes to disk, especially with replicas.
  - E.g., a single logical DB write can lead into multiple (up to five) physical disk writes → performance problems.
  - A lot of data needs to be sent over the network → the I/O bottleneck moves to the network.

- DB admins combat write amplification by reducing the frequency of page flushes.
  - At the cost of slower crash recovery.
Solution: **The Log is the Database**

- Example system **Amazon Aurora**
  - Compute nodes only write redo log records to the distributed storage layer.
    - For correctness purposes the Log is the database!
  - The storage layer re-constructs the page images from log records on demand
    - Any pages that the storage layer materializes (in the background) are a cache of log applications.

*img src: Aurora (SIGMOD’17)*
Advantages

- **Reduces network I/O**, as the primary only ships the log records + metadata updates to the replicas.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Transactions</th>
<th>IOs/Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mirrored MySQL</td>
<td>780,000</td>
<td>7.4</td>
</tr>
<tr>
<td>Aurora with Replicas</td>
<td>27,378,000</td>
<td>0.95</td>
</tr>
</tbody>
</table>

- **Less pressure on the network allows more aggressive data replication**
  - better durability, availability, and minimizing the impact of jitter (more parallel reads).

- **Moving processing (check-pointing, backup/recovery, etc.) to storage**:  
  - Improves availability as we have lower crash recovery time  
  - Minimizes jitter caused by background processes.
Durability at scale

- The lifetime of a database instance is not coupled with storing the data.
  - instances fail, are shut-down, become unavailable, are resized up/down based on workload, etc.

- The storage layer ensures the data is stored durably and resiliently to failure.
  - Goal is to be fault tolerant not only to transient failures but also to entire data centers / AZs.

- Additionally, both data and log should be partitioned for scalability.
Aurora’s quorum and cell based architecture

- Why are 3 replicas and 2/3 read/write quorum insufficient at cloud-scale?
  
  - Assume an AZ fails and another transient fault occurs.

- Aurora replicates all writes six times to three AZs with 4/6 write quorum and 3/6 read quorum.
  
  - can tolerate the loss of an entire AZ without losing data availability.

  - can recover rapidly from larger failures.

img src: Aurora (SIGMOD’18)
- Amazon Aurora’s replication is based on sharding and scale-out.

- When a fault occurs, need a fast recovery
  - Low mean-time-to-recover (MTTR) to increase availability.
  - Need to partition the database in small enough chunks (fixed size segments) that are fast to copy and repair.

- In Aurora:
  - A database is sharded into 10GB logical units.
  - Each unit is replicated six ways, spread across a large distributed storage fleet.
Each log record has an associated Log Sequence Number (LSN).

Storage node steps:
1. Receive a log record and add it to an in-memory queue
2. Persist record on disk and acknowledge
3. Organize records and identify gaps in the log since some batches may be lost
4. Gossip with peers to fill the gaps
5. Coalesce log records into new data pages
6. Periodically stage log and new pages to S3
7. Periodically garbage collect old versions
8. Periodically validate CRC codes on pages

img src: Aurora (SIGMOD'17)
What about reads?

- As with most databases, a read often hits the buffer cache.

- What happens in case of a buffer cache miss?
  
  - A quorum read is expensive and is best avoided.
  
  - Aurora’s client-side storage driver tracks which writes were successful for which segments.
    
    - No need for a quorum read on routine page reads.
  
  - The only time a read quorum is needed is during recovery on a database instance restart.
    
    - The initial set of LSN markers must be reconstructed by asking storage nodes.
Overview of compute-storage Architectures

- **Advantages compared to Traditional Architectures:**
  - **Low write latency:**
    - Write can be committed without waiting for updating pages in storage nodes
  - **Reduce write amplification**
    - Log replay is pushed down 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 scaled independently

- **Limitations:**
  - **High read latency when a cache misses:**
    - The udpate data may not be replayed to the page, leading to extra read latency for log replaying
Separate durability from availability

- Microsoft’s Hyperscale (Socrates) extends the shared-data architecture.

- **Key ideas:**
  1. *Separate Log from Storage* and make the Log first class citizen.
  2. Differentiate *when to use fast vs slow storage*.

- **Advantages:**
  - Separates **durability** (implemented by the log)
    - fundamental property to avoid data loss
  - from **availability** (implemented by the storage tier)
    - needed to provide good quality of service
  - In contrast to availability, durability does not require copies in fast storage
  - In contrast to durability, availability does not need a fixed number of replicas.
Why should the log be treated separately?

- The log is a **potential bottleneck** of any OLTP system
  - Every update must be logged before a transaction is committed
  - The log is shipped to all replicas to keep them consistent
  - Storing it on slow storage media can hurt performance.

- **How to provide a highly performant logging solution at cloud scale?**
  - Make the log durable in fast storage and fault-tolerant by replicating it
  - Reading and shipping log records is more scalable if the log is decoupled from other storage
Disaggregated Compute-Log-Storage Architecture

**Compute tier.**
Compute nodes. The primary instance handles all read/write transactions. Multiple followers handle read-only queries. Cache hot pages in memory.

**Log tier.** Separate log service to ensure low commit latencies and good scalability.

**Storage tier.**
Each page server keeps a copy of a partition of the database in-memory or fast local SSDs.

**Azure Storage Service tier.**
The page servers checkpoint data pages and create back-ups in the slow but cheap XStore tier.

*img src: Socrates (SIGMOD’19)*
Logging as a cloud-native service

1. The Primary Compute node writes log blocks directly to a fast durable storage LZ
   - Synchronous operation that needs low latency
   - For durability, keep 3 replicas of all data.

2. An XLOG process:
   - disseminates the log blocks to Page Servers and follower replicas.
     - asynchronous writes and possibly lossy.
     - LZ for durability, XLOG for availability.
     - waits for blocks to be made durable
     - orders the blocks after gossip protocol
     - archives to local and remote storage.

3. Followers/Page Servers pull log blocks from the XLOG service.

---

img src: Socrates (SIGMOD’19)
Socrates

- Uses less expensive copies of data in the fast local storage
- Fewer copies of data overall and less networking bandwidth
- Less compute resources to keep copies up-to-date

<table>
<thead>
<tr>
<th></th>
<th>Today</th>
<th>Socrates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max DB Size</td>
<td>4TB</td>
<td>100TB</td>
</tr>
<tr>
<td>Availability</td>
<td>99.99</td>
<td>99.999</td>
</tr>
<tr>
<td>Upsize/downsize</td>
<td>O(data)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Storage impact</td>
<td>4x copies (+backup)</td>
<td>2x copies (+backup)</td>
</tr>
<tr>
<td>CPU impact</td>
<td>4x single images</td>
<td>25% reduction</td>
</tr>
<tr>
<td>Recovery</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Commit Latency</td>
<td>3 ms</td>
<td>&lt; 0.5ms</td>
</tr>
<tr>
<td>Log Throughput</td>
<td>50MB/s</td>
<td>100+ MB/s</td>
</tr>
</tbody>
</table>

img src: Socrates (SIGMOD'19)
Overview of Compute-Log-Storage Architecture

**Advantages:**
- Low write latency:
  - With the fast log storage service → write can be committed faster
- Better elasticity:
  - The log and page storage can be scheduled independently, achieving a balance between the cost and the performance

**Disadvantages:**
- High read latency when a 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
Disaggregated Compute-Buffer-Storage Architecture

- **Motivation:**
  - **Latency:**
    reduce read latency with shared remote memory
  - **Throughput:**
    reduce duplicate data loading process of different compute nodes
  - **Elasticity:**
    Memory resources can be allocated on demand.

- **Key feature:**
  - Elastic shared remote buffer for all compute nodes incl. read-write node and read-only nodes.

**img src: Tutorial on Cloud-native databases (VLDB’22)**
Disaggregated Compute-Buffer-Storage Architecture

- **Data write path**
  - Primary updates the local page in cache and generates a redo log
  - The redo log writes to log storage (multi-replicas) for durability
  - Commit the write after redo-log is durable
  - The corresponding page in the buffer is updated simultaneously

- **Data sync path**
  - Page in the shared buffer is not written to storage nodes
  - Redo logs are replayed to page storage asynchronously
  - Directly transfer logs to RO nodes to reduce the update latency

- **Data read path**
  - Compute node checks ist local cache
  - On a miss, check the remote buffer
  - On a miss, read from page storage nodes, update caches based on replacement strategy, return.

Example system:
- Alibaba PolarDB Serverless (SIGMOD 2021)
Overview compute-buffer-storage architecture

- **Advantages:**
  - **Low read latency:**
    - Compute nodes can read data from remote memory – faster than durable storage services
  - **High read throughput:**
    - Different compute nodes share the same remote buffer → reduces the duplicate data for the same read queries on different compute nodes
  - **Better elasticity**
    - Memory resources can be allocated on demand, 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.
    - Thus, the network could become the bottleneck of the database system.
Distributed Transactions
(quick re-cap)
Recall **atomicity** in the context of ACID transactions
- A transaction either **commits** or **aborts**
- If it commits, its updates are durable
- If it aborts, it has no visible side-effects
- ACID consistency (preserving invariants) relies on atomicity

If the transaction updates data on multiple nodes, this implies
- Either all nodes must commit, or all must abort
- If any node crashes, all must abort

Ensuring this is the **atomic commitment** problem.
Two-phase commit (2PC)

- **Two-phase commit protocol**
  most common algorithm to ensure atomic commitment across multiple nodes.
  [Gray, 1978]

- Not to be confused with two-phase locking (2PL).

- **What if the coordinator crashes?**
  - Coordinator writes the decision to disk
  - When it recovers, it reads decision from disk and sends it to replicas
  - Problem if coordinator crashes after prepare, but before broadcasting decision
  - Algorithm is blocked until coordinator recovers

- **Solution: consensus (total order broadcast).**

img src: Martin Kleppmann (Cambridge)
Fault-tolerant 2PC (1/2)

- **Fault tolerant 2PC** based on **Paxos Commit**
  
  [Gray and Lamport, 2006]

- Every node that participates in the transaction uses total order broadcast to disseminate its vote (commit or abort).

- If node A suspects that node B has failed, then A may try to vote to abort on behalf of B.
Fault-tolerant 2PC (2/2)

- **Fault tolerant 2PC** based on **Paxos Commit**
  [Gray and Lamport, 2006]

- Potential race condition if two (conflicting) votes received from the same node (e.g., by node A voting on B’s behalf).

- Resolved due to total order broadcast + counting only the first vote to arrive.
Partitioned log-replicated state machine
Google’s Spanner

- A database system with million’s of nodes, petabytes of data, distributed across datacenters worldwide.

- **Consistency properties:**
  - **Serializable** transaction isolation
  - **Linearizable** reads and writes
  - Many **shards**, each holding a subset of the data;
    atomic commit of transactions across shards

- **Many standard techniques:**
  - State machine replication (Paxos) within a shard
  - Two-phase locking (2PL) for serializability
  - Two-phase commit (2PC) for cross-shard atomicity

- **The interesting bit:** read-only transactions require no locks!
Spanner – Partitioned (Log-replicated SM)

- **Spanner shards the data logically into partitions called splits**
  - To address the O(size of data) issues

- **Multiple copies of a split are kept consistent using Paxos**
  - Only one of the partition replicas can modify data – the leader.
  - Other replicas can only read.

- The main challenge in geo-replication is keeping the replicas consistent
  - **Key innovation with the True Time facility.**
Consistent snapshots

- A read-only transaction observes a **consistent snapshot**

- **Approach:** multi-version concurrency control (MVCC)
  - Each read-write transaction $T_w$ has commit timestamp $t_w$
  - Every value is tagged with timestamp $t_w$ of transaction that wrote it (not overwriting previous value)
  - Read-only transaction $T_r$ has snapshot $t_r$
  - $T_r$ ignores values with $t_w > t_r$; observes most recent value with $t_w \leq t_r$

- Observes **causal consistency** guarantees
- Must ensure that whenever $T_1 \rightarrow T_2$, we have $t_1 < t_2$

  - Physical clocks may be inconsistent with causality
  - Can we use Lamport clocks instead?

  - Problem: linearizability depends on read-time order, and logical clocks may not reflect on this!
True Time: explicit physical clock uncertainty

- Spanner’s TrueTime clock returns \([t_{earliest}, t_{latest}]\).

- True physical timestamp must lie within that range.

- On commit, wait for uncertainty \(\delta_i = t_{i,latest} - t_{i,earliest}\) before committing and releasing the locks and assign \(t_{latest}\) as timestamp for transaction \(T_i\).
  - Its writes become visible to other transactions after \(\delta_i\).
Determining clock uncertainty in TrueTime

- Clock servers with **atomic clock** or **GPS receiver** in each datacenter, servers report their clock uncertainty.
- Each node syncs its quartz clock with a server every 30 sec. Between syncs, assume worst case drift of 200ppm.
Other systems for geo-distributed transactions

- **Strong consistency and serializability for geo-distributed transactions.**
  - CockroachDB
    - Is open source: https://github.com/cockroachdb/cockroach
    - Great resource if you want to see how all the things we’ve covered so far in class are put into practice: https://www.cockroachlabs.com/docs/v22.2/architecture/overview
  - Paper: CockroachDB: The resilient Geo-Distributed SQL Database (Parallel Commits)
    - does not depend on specialized hardware to bound the uncertainty window to a few ms
    - relies on Hybrid-Logical Clocks (HLC): timestamps combining physical and logical time.
    - uncertainty intervals, and behaviour under clock skew
  - **Calvin:** Fast Distributed Transactions for Partitioned Database Systems
  - **Ocean Vista:** Gossip-Based Visibility Control for Speedy Geo-Distributed Transactions
  - **Building Consistent Transactions with Inconsistent Replication**
  - **SLOG:** Serializable, Low-latency, Geo-replicated Transactions
  - Cloud-native transactions and analytics in **SingleStore**
  - **FoundationDB** a distributed unbundled transaction KV store
FoundationDB

- Will host a **guest talk** for more details on **Jan 19th**.

- Adopts an **unbundled architecture**:
  - **Control plane** – manages metadata
    - Coordinators:
      - Cluster sequencer, data distributor, rate keeper
  - **Data plane** – manages data in a **disaggregated compute-log-storage arch.**
    - **Transaction system (TS)**
    - **Log system (LS)**
    - **Storage system (SS)**

- The **TS** provides **strict serializability** through a combination of **OCC and MVCC**.
Rack-scale solutions using fast interconnects
Distributed transactions can scale

- With low-latency high-bandwidth network interconnects using RDMA:
  - The End of a Myth: Distributed Transactions can Scale (NAM-DB, SI)
  - Strong consistency is not hard to get: 2PL and 2PC on Thousands of Cores
  - No compromises: distributed transactions with consistency, availability, and performance (FaRM, serializability)
  - FaSST: Fast, Scalable, and Simple Distributed Transactions with Two-sided (RDMA) datagram RPCs

- Oracle’s RAC and Exadata
  - All nodes of a cluster are tightly coupled on a fast InfiniBand interconnect
  - Shared cache fusion layer over a distributed storage tier with storage cells
    - Optimized network writes/transfers (zero-copy, OS-bypassing, parallel writing, prioritizing critical OLTP I/O writes/transfers, etc.)
The material covered in this class is mainly based on:

- All Things Distributed Blog by Werner Vogel (Amazon’s CTO) ([link](#))
- Slides from “Distributed Systems” course from University of Cambridge ([link](#))
- Tutorial on cloud native databases from VLDB 2022 ([link](#))

Papers:

- Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases (SIGMOD 2017)
- Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits and Membership Changes (SIGMOD 2018)
- Socrates: The New SQL Server in the Cloud (SIGMOD 2019)
- PolarDB Serverless: A Cloud-native database for disaggregated data centers (SIGMOD 2021)
- Spanner: Google's Globally-Distributed Database (OSDI 2012)
- FoundationDB: a distributed unbundled transactional key value store (SIGMOD 2021)

Other relevant (industry) systems:

- Taurus Database: How to be Fast, Available, and Frugal in the Cloud (SIGMOD 2020)
- Cloud-Native Database Systems at Alibaba: Opportunities and Challenges (VLDB 2019)
- CockroachDB: The Resilient Geo-Distributed SQL Database (SIGMOD 2020)