Apache Cassandra - A Decentralized Structured Storage System

Avinash Lakshman
Prashant Malik
from Facebook

Presented by:
Oded Naor
Acknowledgments

Some slides are based on material from:

• Idit Keidar, Topics in Reliable Distributed Computing, EE, Technion
• DataStax
• Database Systems course, CS, Technion
Databases

• Organized collection of data
  • Typically persistent
• Data retrieval queries
  • E.g. SQL
• Concurrent access by many users
• Managed by DBMS (Database Management System) software
  • MySQL, SimpleDB (Amazon), and many more
Relational DB

• Data stored in tables (relations)
• Each table has a fixed set of attributes (columns)
  • E.g. person table attributes: id, name, age, address
• Data stored in rows (tuples)
Querying Relational DBs

• High-level query languages based on set-theory relations
• Most popular is SQL

<table>
<thead>
<tr>
<th>ID</th>
<th>Course</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123456789</td>
<td>236832</td>
<td>100</td>
</tr>
</tbody>
</table>

What are the names of all the students that received 100 in course 236832?

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>123456789</td>
<td>Oded Naor</td>
</tr>
</tbody>
</table>

What are the names of all the students that received 100 in course 236832?
A collection of query and update operations executed atomically
- May succeed and commit
- May fail and abort
- So-called **ACID** properties:
  - **Atomicity**
    - A transaction can succeed or abort; all-or-nothing
  - **Consistency**
    - Correct execution
    - A TX that starts in the future sees all the committed TXs in the past.
  - **Isolation**
    - No partial result observed by clients
  - **Durability**
    - Persistence

Transactions [Gray 78]
NoSQL

• Not Only SQL
  • Not the other thing!
  • Data modeled in means other than tabular relations
  • E.g. Key-Value store, Document-Store, Tuple store...

• Seminal papers:
  • Google’s BigTable, 2006
  • Amazon’s DynamoDB, 2007
So Far...

- Talked mainly on concurrency on a single node:
  - Spin locks
  - Monitors
  - Multi-core scalability
  - GPGPUs
  - ...
- This means we have:
  - “Semi-synchronous”
  - Shared resources
  - Locks
This Talk – Distributed System

• Many Nodes
  • Each node may run concurrent threads and processes
  • We treat each node as an “object” in the general system
• Asynchronous communication
  • Messages between nodes can be arbitrarily delayed
  • A node cannot determine if another node crashed or just delayed communication
• Crashes
  • A node can crash arbitrarily, the system needs to keep working
ACID May Be Overly Expensive

• In quite a few modern applications:
  • ACID contrasts with key desiderata: high volume, high availability
  • We can live with some errors, to some extent
  • Or more accurately, we prefer to suffer errors than to be significantly less functional
Simple Model of a Distributed Service

- **Context**: distributed service
  - e.g., social network
- **Clients** make get / set **requests**
  - e.g., setLike(user, post), getLikes(post)
  - Each client can talk to any server
- **Servers** return **responses**
  - e.g., ack, \{user_1, ..., user_k\}

- **Failure**: the network may occasionally disconnect due to failures
- **Desiderata**: **C**onsistency, **A**vailability, **P**artition tolerance
CAP Service Properties

• **Consistency**
  • every read (to any node) gets a response that reflects the most recent version of the data

• **Availability**
  • every request (to a living node) gets an answer: set succeeds, get returns a value

• **Partition tolerance**
  • service continues to function on network failures
  • As long as clients can reach servers
The CAP Theorem

CAP Theorem [Brewer1999]:

*A distributed service can support *at most two out of* **C, A and P**.
Visual Guide to NoSQL Systems

Data Models
- Relational (comparison)
- Key-Value
- Column-Oriented/Tabular
- Document-Oriented

Pick Two

CA
- RDBMSs (MySQL, Postgres, etc)
- Aster Data Greenplum Vertica

AP
- Dynamo
- Voldemort
- Tokyo Cabinet
- KAI
- Cassandra
- SimpleDB
- CouchDB
- Riak

Availability:
Each client can always read and write.

Consistency:
All clients always have the same view of the data.

CP
- BigTable
- Hypertable
- Hbase
- MongoDB
- Terrastore
- Scalaris
- Berkeley DB
- MemcacheDB
- Redis

Partition Tolerance:
The system works well despite physical network partitions.

From: 2010 visual by Nathan Hurst
The BASE Model

• Applies to distributed systems of type AP

• **Basic Availability**
  • Provide high availability through distribution

• **Soft state**
  • Inconsistency (stale answers) allowed

• **Eventual consistency**
  • If updates stop, then after some time consistency will be achieved
    • Achieved by protocols to propagate updates and verify correctness of propagation (gossip protocols)

• Philosophy: best effort, optimistic, staleness and approximation allowed
Cassandra’s Parents - Amazon Dynamo

• What is it?
  • A highly-available and scalable storage system used by Amazon
  • Key-Value Store
  • Used to retrieve user shopping charts and other core services

• How it works?
  • Operations work even when network is partitioned
    • Conflicts resolved using conflicts resolution mechanism
  • AP model
  • Allows extensive customizations
Cassandra’s Parents – Google BigTable

• What is it?
  • A high performance data storage system built on Google File System and other Google technologies.

• How it works?
  • Provides both structure and data distribution but relies on a distributed file system for durability.
  • Richer data model from Dynamo
    • One key, many values
## Cassandra and Parents

<table>
<thead>
<tr>
<th></th>
<th>Bigtable</th>
<th>Dynamo</th>
<th>Cassandra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data model</td>
<td>Column-Oriented</td>
<td>Key-Value</td>
<td>Column-Oriented</td>
</tr>
<tr>
<td>CAP</td>
<td>AP</td>
<td>CP</td>
<td>AP</td>
</tr>
<tr>
<td>Architecture</td>
<td>Master-slave</td>
<td>Decentralized</td>
<td>Decentralized</td>
</tr>
<tr>
<td></td>
<td>P2P</td>
<td>P2P</td>
<td>P2P</td>
</tr>
</tbody>
</table>
Column Stores

• Common idea: don’t keep a row in a consecutive block, split via projection
  • Column store: each column is independent; column-family store: each column family is independent

• Both provide some major efficiency benefits in common read-mainly workloads
  • Given a query, load to memory only the relevant columns
  • Columns can often be highly compressed due to value similarity
  • Effective form for sparse information (no NULLs, no space)

• Column-family store is handled differently from RDBs, often requiring a designated query language
**Column Store**

<table>
<thead>
<tr>
<th>id</th>
<th>sid</th>
<th>name</th>
<th>address</th>
<th>year</th>
<th>faculty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>861</td>
<td>Alma</td>
<td>Haifa</td>
<td>2</td>
<td>NULL</td>
</tr>
<tr>
<td>2</td>
<td>753</td>
<td>Amir</td>
<td>Jaffa</td>
<td>NULL</td>
<td>CS</td>
</tr>
<tr>
<td>3</td>
<td>955</td>
<td>Ahuva</td>
<td>NULL</td>
<td>2</td>
<td>IE</td>
</tr>
</tbody>
</table>

**Standard RDB**

**Column Store**: each column stored separately (still SQL)

Why? **Efficiency** (fetch only required columns), compression, sparse data for free

**Column-Family Store**: NoSQL

**keyspace**

<table>
<thead>
<tr>
<th>column family</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

**Column Family**

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>address</th>
<th>year</th>
<th>faculty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alma</td>
<td>Haifa</td>
<td>2</td>
<td>NULL</td>
</tr>
<tr>
<td>2</td>
<td>Amir</td>
<td>Jaffa</td>
<td>2</td>
<td>CS</td>
</tr>
<tr>
<td>3</td>
<td>Ahuva</td>
<td>NULL</td>
<td>2</td>
<td>IE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>faculty</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>CS</td>
</tr>
<tr>
<td>3</td>
<td>IE</td>
</tr>
</tbody>
</table>

(Cassandra model)
Partitioning

• Offers the ability to scale incrementally

• How?
  • Dynamically partition the data over the set of nodes in the cluster.
  • Consistent hashing (order preserving hash function).
  • Output range: a ring
  • Each node: is assigned a random value which determines its place on the ring.
Partitioning

• Each data item:
  • Hashes its key to find its position on the ring
  • “Walks” clockwise to find the first node on the ring

• Each node:
  • Assigned a random value on the ring
  • Responsible for the region counterclockwise until its predecessor
  • Departure or arrival of a node only affects the immediate neighbours.
Partitioning

• Challenges:
  • Random positioning of each node leads to non-uniform data and load distribution.
  • The basic algorithm is oblivious to the heterogeneity in the performance of nodes.

• Addressed by:
  • Analyzing load information on the ring
  • Lightly loaded nodes move on the ring to alleviate heavily loaded ones.
Replications

- **Replication factor**: determines how many copies of your data exist
- **Coordinator node**: in charge of the replications of its part of the ring
- Each row is replicated across multiple datacenters
- Uses Zookeeper to elect a leader node
  - Leader decides replica ranges for each node in Cassandra
Cassandra’s Ring Model with Replications

write(k, t)

hash(k) = 2

Replication Factor = 3

Advantage: Flexibility / ease of cluster redesign
Failure Detector

• Each node can estimate what nodes are up or down
• Based on Φ accrual failure detector [DUHK2004]
• Emits a value Φ rather than a Boolean value
• Value represents likelihood of a mistake
  • Φ = 1 likelihood of mistake: 10%
  • Φ = 2 likelihood of mistake: 1%
Atomic Register Emulation in a Message-Passing System

[Attiya, Bar-Noy, Dolev]
Linearizability - Read/Write Variable Example

\[ \text{write}(0) \quad \text{read}(1) \quad \text{write}(2) \quad \text{write}(1) \quad \text{read}(2) \]
Linearizability - Read/Write Variable Example

write(1) happened after write(0)
Linearizability - Read/Write Variable Example

walk(0) -> read(1) -> write(2) -> write(1)
walk(1) already happened
read(1)
Distributed Shared Memory (DSM)

- Can we provide the illusion of atomic shared-memory registers in a message-passing system?
- In an asynchronous system?
- Where processes can fail?
Liveness Requirement

- **Wait-freedom**: every operation by a correct process \( p \) eventually completes
  - In a finite number of \( p \)'s steps
- Regardless of steps taken by other processes
  - In particular, the other processes may fail or take any number of steps between \( p \)'s steps
  - But \( p \) must be given a chance to take as many steps as it needs (**fairness**)


Register

• Holds a value
• Can be read
• Can be written

• Interface:
  • int read(); // returns last written value
  • void write(int v); // returns ack
ABD: Fault-Tolerant Emulation

[Attiya, Bar-Noy, Dolev]

• Assumes up to $f < n/2$ processes can fail
• Main ideas:
  • Store value at majority of processes before write completes
  • read from majority
  • read intersects write, hence sees latest value
1-Reader 1-Writer (SRSW)

• Single-reader – there is only one process that can read from the register
• Single-writer – there is only one process that can write to the register
• The reader and writer are just 2 processes
  • The other $n-2$ processes are there to help
Trivial Solution?

• Writer simply sends message to reader
  • When does it return ack?
  • What about failures?

• We want a *wait-free* solution:
  • If the reader (writer) fails, the writer (reader) should be able to continue writing (reading)
1-Writer

write(v)

write

ack

("write", v, t)

("ack")
1-Reader

read

R

read() → R

S2

("read") → S2

Sn

("read-ack", v, t) → Sn

R

return(v) → R
SRSW Algorithm: Variables

• At each process:
  • $x$, a copy of the register
  • $t$, initially 0, unique tag associated with latest write
SRSW Algorithm Emulating **Write**

- To perform \( \text{write}(x, v) \)
  - choose \( \text{tag} > t \)
  - set \( x \leftarrow v; \ t \leftarrow \text{tag} \)
  - send (“write”, \( v, t \)) to all

- Upon receive (“write”, \( v, \ t, \text{tag} \))
  - **if** \( \text{tag} > t \) **then** set \( x \leftarrow v; \ t \leftarrow \text{tag} \) **fi**
  - send (“ack”, \( v, \text{tag} \)) to writer

- When writer receives (“ack”, \( v, t \)) from majority (counting an ack from itself too)
  - return ack to client
SRSW Algorithm Emulating Read

• To perform read(x,v)
  • send (“read”) to all
• Upon receive (“read”)
  • send (“read-ack”, x, t) to reader
• When reader receives (“read-ack”, v, tag) from majority (including local values of x and t)
  • choose value v associated with largest tag
  • store these values in x,t
  • return x
Does This Work?

• Only possible overlap is between read and write
  • why?

• When a read does not overlap any write –
  • It reads at least one copy that was written by the latest write (why?)
  • This copy has the highest tag (why?)

• What if 2 reads overlap the same write?
Example

- read(1)
- write(1) already happened
- read(?)
- But local copy written by read
- finds a copy that was written

Linearizable
Wait-Freedom

• Only waiting is for majority of responses
• There is a correct majority
• All correct processes respond to all requests
  • Respond even if the tag is smaller
Take III: n-Reader 1-Writer (MRSW)

• n-reader – all the processes can read
• Does the previous solution work?
• What if 2 reads by different processes overlap the same write?
Example

finds a copy that was written

write(1) already happened

write(1) already happened

read(1)

does not find a written copy, returns 0
MRSW Algorithm
Extending the **Read**

• When reader receives (“read-ack”, \(v, \tag\)) from majority
  • choose value \(v\) associated with largest \(\tag\)
  • store these values in \(x,t\)
  • send (“propagate”, \(x, t\)) to all (except writer)

• Upon receive (“propagate”, \(v, \tag\)) from process \(i\)
  • **if** \((\tag > t)\) **then** set \(x \leftarrow v; t \leftarrow \tag\) **fi**
  • send (“prop-ack”, \(x, t\)) to process \(i\)

• When reader receives (“prop-ack”, \(v, \tag\)) from majority (including itself)
  • return \(x\)
The Complete Read

Phase 1: Read

Phase 2: Write-Back

Multi-reader only
Take IV: n-Reader n-Writer (MRMW)

• n-writer – all the processes can write to the register
• Does the previous solution work?
Playing Tag

• What if two writers use the same tag for writing different values?
• Need to ensure *unique* tags
  • That’s easy: break ties, e.g., by process id
• What if a later write uses a smaller tag than an earlier one?
  • Must be prevented (*why?*)
MRMW Algorithm
Extending the Write

- To perform write(x,v)
  - send (“query”) to all
- Upon receive (“query”) from i
  - send (“query-ack”, t) to i
- When writer receives (“query-ack”, tag) from majority (counting its own tag)
  - choose unique tag > all received tags
  - continue as in 1-writer algorithm
- What if another writer chooses a higher tag before write completes?
The Complete Write

Phase 1: Read
Multi-writer only

Phase 2: Write

**write(v)**

S1 → S1 → S1 → S1 → S1 → S1 → Sn

(“query”) → (“query-ack”, t) → ⋮ → (“write”, v, t) → (“ack”) → Sn

**ack**
How Long Does it Take?

• The write emulation
  • Single-writer: 2 rounds (steps)
  • Multi-writer: 4 rounds (steps)

• The read emulation
  • Single-reader: 2 rounds (steps)
  • Multi-reader: 4 rounds (steps)
What if A Majority Can Fail?

• You guessed it!
• Homework question
Can We Emulate *Every* Atomic Object the Same Way?

- We only emulated a read/write object
- Think of a general object type, with some data members and some methods
- Can we support it the same way?
R/W Registers vs. Consensus

• ABD works even if the system is completely asynchronous
• In Paxos, there is no progress when there are multiple leaders
• Here, there is always progress – multiple writers can write concurrently
  • One will prevail (which?)
Questions?
thank you!