ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 12 Distributed Database Systems I

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

September 30, 2024

Outline

Reading Assignment

- This lecture: Cassandra - A Decentralized Structured Storage System `https://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf`
- Next lecture: Spanner: Google's Globally-Distributed Database `http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//archive/spanner-osdi2012.pdf`

Outline

Distributed Data Store

- Horizontal scaling requires to use more servers to support a single data store.
  - Sharding improves write performance (multiple pieces of data).
  - Replication improves read performance (single piece of data).
- Let's still consider our banking exmaple where Alice need to transfer $100 to Bob.
- But we would assume due to scale-out requirements, Alice's account is stored in a server in the Chicago office and Bob's account is stored in a server in the New York office.
  - The two offices are connected by computer network so servers can communicate with each other.
  - Let's further assume the request to transfer money is processed by the New York server.

Distributed Data Store (Cont.)

- While apparently the two servers need to communicate with each other to have consensus on what is available on each account before and after the transation, there may exist failures in the system.
  - A server becomes unresponsive for a period of time, e.g. when a Java program need to "Stop-the-World" for garbage collection.
  - The computer network may be interrupted by some reason.
- What requirements should we have for this system to serve the users properly?

Consistency and Availability

- Consistency: servers have consensus on the data.
  - The New York office server will need to make sure that Alice has enough balance before it updates Bob's account.
  - As mentioned before, one method to achieve this is to allow the New York server to contact the Chicago server.
- Availability: a server, when works properly, will respond requests from users promptly.
  - The New York server should response that the transaction either completes successfully if Alice has enough balance or fails if Alice does not have enough balance.

Partition Tolerance

- But what if the Chicago server is unresponsive or the network communication is interrupted?
  - For consistency, the New York server need to resend messages and wait for responses from the Chicago server before it can proceed with the transaction – availability is violated.
  - For availability, the New York server may choose to allow Bob have $100 before validating that Alice has enough balance, or tell Bob that Alice does not have enough balance though Alice may have enough balance – consistency is violated.
- Partition tolerance: the system will continue to work while satisfying predefined guarantees if the messages are dropped.
  - Server failures may be modeled as communication failures.
  - Can predefined guarantees include both consistency and availablility?
  - Will adding replicas at Chicago and New York, or allowing the Chicago server to process the request help?

The CAP Theorem

You can only choose at most two out of Consistency, Availability, and Partition tolerance when building a distributed data store.

- Relational databases running on a single server has no partition tolerance so can have both consistency and availability.
- Distributed data store need to survive network failures so need to give up either consistency or availability.

Outline

Non-Relational Database

- Unlike banking, there are data management systems not requiring ACID guarantees.
  - E.g. consider an IoT system that stores readings from sensors.
- Distributed data stores usually need to sacrifice consistency or availability.
  - For larger capacity and better performance.
  - So cannot guarantee ACID.
- As most single server relational databases guarantee ACID, newer distributed data stores explore a different trade-off to give up ACID and to be non-relational at the same time.

BASE Guarantees

- A set of guarantees used by many distributed data stores.
- Basically Available: prioritize availability, even if consistency may be violated.
  - I.e. we allow each server to react by itself if the network communication is interrupted.
- Soft state: different servers may have different views of data.
  - As a consequence, application developers should be aware of inconsistency within the data if they are obtained from multiple servers.
- Eventually consistent: when there is no update, the different views will converge into a consistent one later.
  - End users may expect reasonable application behaviors once everything settles down.
  - Indeed, many network related behaviors are eventually consistent – if you send me an email, you would expect me to receive the email sometime later, but not immediately.

Outline

Apache Cassandra

- A distributed key-value database designed for high availability, scalability, and fault tolerance.
  - Run on a cluster of commodity servers, or nodes.
  - Allow nodes to join and leave easily for growth and maintenance.
  - Enable trade-offs between availability and consistency via tunable consistency.
  - No single point of failure.
- Developed at Facebook in 2008, later open-sourced as Apache Cassandra.
  - Combining ideas from Google's Bigtable and Amazon's Dynamo.

Data Model

- To leverage existing work force that are familiar with SQL, Cassandra uses similar wording to discribe its data model.
  - Data are organized into tables.
  - Each table consists of rows.
  - Each row consists of columns, some as the (primary) key, and the rest as the value.
- Columns for key are ordered and divided into two groups.
  - Partition key is used for sharding, which decides which server this row is stored. Thus range query is not supported.
  - Clustering key is used to sort the rows with the same partition key (on the same server) following the lexicographical order.
- Columns for values may be added to specific rows later.
  - No need to follow a predefined schema.
- No join and additional indexing.
  - While recent versions may support join and index, you will need to understand their consistency and performance implications as Cassandra is not a relational database.

Data Model Example

- A simple social network service.
  - Users have other users as friends: each pair are either friends of each other or not friends at all.
  - Users create posts that only their friends can read.
- TABLE Friends: partition key userId, clustering key friendId
  - Each row represents a friendship relation.
  - The rows are located in multiple servers depending on userId.
- Listing all friends of a specific user is efficient.
  - Query runs on a single server.
- Updating relations between two users may break consistency.
  - Need to update two rows that are on two different servers.
  - What if one server fails?
  - What if queries for listing friends arrived between one server is updated and the other is not?

Data Model Example (cont.)

- TABLE Posts: partition key userId, clustering key postId
  - Columns for value: postTime, content
  - Each row represents a single post.
  - Posts for the same user stay on a single server.
- How to list all posts from friends of a user?
  - Can be solved by joining Friends and Posts tables for a relational database.
  - For Cassandra, will need to query Friends to list all friends of the user (one server), and then query Posts for posts from each friend (multiple servers).
  - What about consistency if one delete a friend during the query?
- What if one would like to list recent posts from friends?
  - Option 1: use (postTime, postId) as clustering key for Posts table to query posts from a user within a time interval.
  - Option 2: create a new table FriendsPosts with primary key userId, clustering key (postTime, friendId, postId), and another column content – query can be resolved on a single server.

Nodes Management

- A keyspace consists of multiple tables.
- Each keyspace has a replication factor N.
  - Every row in the keyspace will be replicated to N nodes.
- When nodes join the cluster, some rows are moved from existing nodes to them to balance the load.
- When nodes leave the cluster, missing replicas of rows are recreated from remaining nodes (if possible).
- Example: with 5 nodes each with 1TB storage, a keyspace with N=3 can store 1.67TB of rows.
  - Adding another node will increase the size to 2TB.
  - There will be no data loss if two nodes fails.

Data Management

- MemTables store data in the memory.
  - Serve as cache for read and write operations.
- SSTables persist data on disks.
  - Created when flushing data from MemTables to disks periodically or when there is not enough memory.
  - Once written, SSTables are unchangeable – later deletes and updates are marked in follow-up SSTables.
  - When system is less busy, multiple SSTables can be compacted into one by removing deleted entries and merging updates.
- CommitLog is the transaction log for writes.
  - Append-only on dedicated local disk for high write throughput.
  - Managed as multiple files so those that are flushed to SSTables can be safely discarded.
- Overall, the performance objective is to saturate disks throughput via sequential writes.

Request and Replication

- A client connects to the cluster via with a node C.
  - The client may config or obtain from the node C a list of available nodes that can be used when the node C fails.
- A request should at least provide partition keys so the node C know where to forward it.
  - I.e. to the nodes that hold replicas of the rows.
  - The node C will further forward the responses from those nodes back to the client.
- Should the client wait for all the responses?
  - Yes: make sure all replicas behave the same for consistency.
  - No: use the fastest node for availability.
  - What about to treat reads and writes differently?

Tunable Consistency

- Option 1: wait for all replicas to acknowledge writes.
  - Best availablity and performance for read: just need to wait for the fastest available node to read the most recent write.
  - But it may slowdown writes and in the worst case violate availability as writes may need to wait for nodes and the network to recover from a failure.
- Option 2: wait for the fastest node to acknowledge write.
  - Best availablity and performance for write.
  - Need to wait for all replicas for reads in order to get the most recent write – bad availability.
  - What about wait for the fastest node for read? No guarantee of consistency (get the most recent write)!

Tunable Consistency (Cont.)

- Option 3: assuming the row is replicated to $N = 2k + 1$ nodes, then wait for $k + 1$ nodes for both writes and read.
  - Consistency is guaranteed to read the most recent write.
  - Availability may be violated if more than half of the nodes or their networks fail.
  - But the system should work properly most of the time when most nodes and networks work.
- Though you still need to remember that due to the CAP theorem, one among consistency, availability, and partition tolerance must be given up.
  - For Option 3, if a network partition happens, then clients connecting to the smaller partition with no more than k nodes need to wait for both write and read – availability cannot be guaranteed.