Unit 3 Syllabus

- Cassandra: Review of Relational databases, Sharding and Shared-Nothing Architecture, Web Scale, Rise of NoSQL, Distributed and Decentralized, Elastic Scalability, High Availability and Fault Tolerance, Tuneable Consistency, High Performance, Brewer's CAP Theorem, Cassandra's Data Model, CQL types, Other Simple Data Types, User-Defined Types

Review of Relational databases

- There are many reasons that the relational database has become so overwhelmingly popular over the last four decades.
- An important one is the Structured Query Language (SQL), which is feature-rich and uses a simple, declarative syntax. SQL was first officially adopted as an American National Standards Institute (ANSI) standard in 1986; since that time, it's gone through several revisions and has also been extended with vendor-proprietary syntax such as Microsoft's T-SQL and Oracle's PL/SQL to provide additional implementation-specific features.

Review of Relational databases

- SQL is powerful for a variety of reasons. It allows the user to represent complex relationships with the data, using statements that form the Data Manipulation Language (DML) to insert, select, update, delete, truncate, and merge data.
- SQL statements support grouping aggregate values and executing summary functions. SQL provides a means of directly creating, altering, and dropping schema structures at runtime using Data Definition Language (DDL). SQL also allows you to grant and revoke rights for users and groups of users using the same syntax.

Transactions, ACID-ity, and Two-Phase Commit

- ACID is an acronym for Atomic, Consistent, Isolated, Durable, which are the gauges you can use to assess that a transaction has executed properly and that it was successful
- Atomic
- Atomic means "all or nothing"; that is, when a statement is executed, every update within the transaction must succeed in order to be called successful.
- There is no partial failure where one update was successful and another related update failed. The common example here is with monetary transfers at an ATM: the transfer requires a debit from one account and a credit to another account. This operation cannot be subdivided; they must both succeed.

ACID-ity

- Consistent
- Consistent means that data moves from one correct state to another correct state, with no possibility that readers could view different values that don’t make sense together. For example, if a transaction attempts to delete a customer and their order history, it cannot leave order rows that reference the deleted customer’s primary key; this is an inconsistent state that would cause errors if someone tried to read those order records.

ACID-ity

- Isolated Isolated means that transactions executing concurrently will not become entangled with each other; they each execute in their own space. That is, if two different transactions attempt to modify the same data at the same time, then one of them will have to wait for the other to complete.
- Durable Once a transaction has succeeded, the changes will not be lost. This doesn’t imply another transaction won’t later modify the same data; it just means that writers can be confident that the changes are available for the next transaction to work with as necessary

Two-Phase Commit

- Transactions become difficult under heavy load. When you first attempt to horizontally scale a relational database, making it distributed, you must now account for distributed transactions, where the transaction isn’t simply operating inside a single table or a single database, but is spread across multiple systems. In order to continue to honor the ACID properties of transactions, you now need a transaction manager to orchestrate across the multiple nodes

Two-Phase Commit

- In order to account for successful completion across multiple hosts, the idea of a two-phase commit (sometimes referred to as "2PC") is introduced.
- The two-phase commit is a commonly used algorithm for achieving consensus in distributed systems, involving two sets of interactions between hosts known as the prepare phase and commit phase. Because the two-phase commit locks all associated resources, it is useful only for operations that can complete very quickly.
- Although it may often be the case that your distributed operations can complete in subsecond time, it is certainly not always the case. Some use cases require coordination between multiple hosts that you may not control yourself. Operations coordinating several different but related activities can take hours to update

Two-Phase Commit

- Two-phase commit blocks; that is, clients (“competing consumers”) must wait for a prior transaction to finish before they can access the blocked resource.
- The protocol will wait for a node to respond, even if it has died. It’s possible to avoid waiting forever in this event, because a timeout can be set that allows the transaction coordinator node to decide that the node isn’t going to respond and that it should abort the transaction. However, an infinite loop is still possible with 2PC; that’s because a node can send a message to the transaction coordinator node agreeing that it’s OK for the coordinator to commit the entire transaction.
- The node will then wait for the coordinator to send a commit response

Sharding and Shared-Nothing Architecture

- This has been used to good effect at large websites such as eBay, which supports billions of SQL queries a day, and in other modern web applications.
- The idea here is that you split the data so that instead of hosting all of it on a single server or replicating all of the data on all of the servers in a cluster, you divide up portions of the data horizontally and host them each separately.
- For example, consider a large customer table in a relational database. The least disruptive thing (for the programming staff, anyway) is to vertically scale by adding CPU, adding memory, and getting faster hard drives, but if you continue to be successful and add more customers, at some point

Feature-based shard or functional segmentation

- There are three basic strategies for determining shard structure:
- Feature-based shard or functional segmentation
- This is the approach taken by Randy Shoup, Distinguished Architect at eBay, who in 2006 helped bring the site's architecture into maturity to support many billions of queries per day.
- Using this strategy, the data is split not by dividing records in a single table (as in the customer example discussed earlier), but rather by splitting into separate databases the features that don't overlap with each other very much.
- For example, at eBay, the users are in one shard, and the items for sale are in another. This approach depends on understanding your domain so that you can segment data cleanly.

Key-based sharding

- In this approach, you find a key in your data that will evenly distribute it across shards.
- So instead of simply storing one letter of the alphabet for each server as in the (naive and improper) earlier example, you use a one-way hash on a key data element and distribute data across machines according to the hash. It is common in this strategy to find time-based or numeric keys to hash on

Lookup table

- In this approach, also known as directory-based sharding, one of the nodes in the cluster acts as a “Yellow Pages” directory and looks up which node has the data you’re trying to access. This has two obvious disadvantages.
- The first is that you’ll take a performance hit every time you have to go through the lookup table as an additional hop. The second is that the lookup table not only becomes a bottleneck, but a single point of failure.

Shared Nothing Architectue

- Sharding could be termed a kind of shared-nothing architecture that’s specific to databases.
- A shared-nothing architecture is one in which there is no centralized (shared) state, but each node in a distributed system is independent, so there is no client contention for shared resources. Shared-nothing architecture was more recently popularized by Google, which has written systems such as its Bigtable database and its MapReduce implementation that do not share state, and are therefore capable of nearinfinite scaling.
- The Cassandra database is a shared-nothing architecture, as it has no central controller and no notion of primary/secondary replicas; all of its nodes are the same.

Shared Nothing Architectectue

- Many nonrelational databases offer this automatically and out of the box is very handy; creating and maintaining custom data shards by hand is a wicked proposition.
- For example, MongoDB, which we'll discuss later, provides auto-sharding capabilities to manage failover and node balancing.
- It's good to understand sharding in terms of data architecture in general, but especially in terms of Cassandra more specifically. Cassandra uses an approach similar to key-based sharding to distribute data across nodes, but does so automatically.

Web Scale

- With the rapid growth in the web, there is great variety to the kinds of data that need to be stored, processed, and queried, and some variety to the businesses that use such data.
- Consider not only customer data at familiar retailers or suppliers, and not only digital video content, but also the required move to digital television and the explosive growth of email, messaging, mobile phones, RFID,
- Voice Over IP (VoIP) usage, and the Internet of Things (IoT). Companies that provide content—and the third-party value-add businesses built around them—require very scalable data solutions. Consider too that a typical business application developer or database administrator may be used to thinking of relational databases as the center of the universe.

The Rise of NoSQL

- The term "NoSQL" began gaining popularity around 2009 as a shorthand way of describing these databases.
- The term has historically been the subject of much debate, but a consensus has emerged that the term refers to nonrelational databases that support "not only SQL" semantics.

Key-value stores

- In a key-value store, the data items are keys that have a set of attributes.
- All data relevant to a key is stored with the key; data is frequently duplicated.
- Popular key-value stores include Amazon’s Dynamo DB, Riak, and Voldemort. Additionally, many popular caching technologies act as key-value stores, including Oracle Coherence, Redis, and Memcached.

Column stores

- In a column store, also known as a wide-column store or column oriented store, data is stored by column rather than by row.
- For example, in a column store, all customer addresses might be stored together, allowing them to be retrieved in a single query. Popular column stores include Apache Hadoop’s HBase, Apache Kudu, and Apache Druid.

Document stores

- The basic unit of storage in a document database is the complete document, often stored in a format such as JSON, XML, or YAML.
- Popular document stores include MongoDB, CouchDB, and several public cloud offerings.

Graph databases

- Graph databases represent data as a graph—a network of nodes and edges that connect the nodes.
- Both nodes and edges can have properties. Because they give heightened importance to relationships, graph databases such as Neo4j, JanusGraph, and DataStax Graph have proven popular for building social networking and semantic web applications

Object databases

- Object databases store data not in terms of relations and columns and rows, but in terms of objects as understood from the discipline of objectoriented programming.
- This makes it straightforward to use these databases from object-oriented applications. Object databases such as db4o and InterSystems Caché allow you to avoid techniques like stored procedures and object-relational mapping (ORM) tools. The most widely used object database is Amazon Web Services' Simple Storage Service (S3).

XML databases

- XML databases are a special form of document databases, optimized specifically for working with data described in the eXtensible Markup Language (XML). So-called “XML native” databases include BaseX and eXist.

Multimodel databases

- Databases that support more than one of these styles have been growing in popularity. These “multimodel” databases are based on a primary underlying database (most often a relational, key-value, or column store) and expose additional models as APIs on top of that underlying database. Examples of these include
- Microsoft Azure Cosmos DB, which exposes document, wide column, and graph APIs on top of a key-value store, and DataStax Enterprise, which offers a graph API on top of Cassandra’s wide column model. Multimodel databases are often touted for their ability to support an approach known as polyglot persistence

Cassandra

- "Apache Cassandra is an open source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tuneably consistent, row-oriented database.
- Cassandra bases its distribution design on Amazon's Dynamo and its data model on Google's Bigtable, with a query language similar to SQL. Created at Facebook, it now powers cloud-scale applications across many industries

Distributed and Decentralized

- Cassandra is distributed, which means that it is capable of running on multiple machines while appearing to users as a unified whole. In fact, there is little point in running a single Cassandra node.
- Although you can do it, and that's acceptable for getting up to speed on how it works, you quickly realize that you'll need multiple machines to really realize any benefit from running Cassandra.
- Much of its design and codebase is specifically engineered toward not only making it work across many different machines, but also for optimizing performance across multiple data center racks, and even for a single Cassandra cluster running across geographically dispersed data centers. You can confidently write data to anywhere in the cluster and Cassandra will get it.

Distributed and Decentralized

- Once you start to scale many other data stores (MySQL, Bigtable), some nodes need to be set up as primary replicas in order to organize other nodes, which are set up as secondary replicas.
- Cassandra, however, is decentralized, meaning that every node is identical; no Cassandra node performs certain organizing operations distinct from any other node. Instead,
- Cassandra features a peer-to-peer architecture and uses a gossip protocol to maintain and keep in sync a list of nodes that are alive or dead. We'll discuss this more in "Gossip and Failure Detection"

Distributed and Decentralized

- The fact that Cassandra is decentralized means that there is no single point of failure.
- All of the nodes in a Cassandra cluster function exactly the same.
- This is sometimes referred to as “server symmetry.” Because they are all doing the same thing, by definition there can’t be a special host that is coordinating activities, as with the primary/secondary setup that you see in MySQL, Bigtable, and so many other databases

Distributed and Decentralized

- Typically this process is not decentralized, as in Cassandra, but is rather performed by defining a primary/secondary relationship.
- That is, all of the servers in this kind of cluster don’t function in the same way. You configure your cluster by designating one server as the primary (or primary replica) and others as secondary replicas.

Distributed and Decentralized

- The primary replica acts as the authoritative source of the data, and operates in a unidirectional relationship with the secondary replicas, which must synchronize their copies.
- If the primary node fails, the whole database is in jeopardy. To work around the situation of the primary as a single point of failure, you often need to add complexity to the environment in the form of multiple primary nodes.

Distributed and Decentralized

- Note that while we frequently understand primary/secondary replication in the RDBMS world, there are NoSQL databases such as MongoDB that follow the primary/secondary scheme as well. Even Mongo's "replica set" mechanism is essentially a primary/secondary scheme in which the primary can be replaced by an automated leader election process.

Distributed and Decentralized

- Decentralization, therefore, has two key advantages:
- it's simpler to use than primary/secondary, and it helps you avoid outages.
- It is simpler to operate and maintain a decentralized store than a primary/secondary store because all nodes are the same.
- That means that you don't need any special knowledge to scale; setting up 50 nodes isn't much different from setting up one.
- There's next to no configuration required to support it. Because all of the replicas in Cassandra are identical, failures of a node won't disrupt service.

Elastic Scalability

- Elastic scalability refers to a special property of horizontal scalability.
- It means that your cluster can seamlessly scale up and scale back down.
- To do this, the cluster must be able to accept new nodes that can begin participating by getting a copy of some or all of the data and start serving new user requests without major disruption or reconfiguration of the entire cluster.
- You don’t have to restart your process. You don’t have to change your application queries. You don’t have to manually rebalance the data yourself. Just add another machine—Cassandra will find it and start sending it work.

High Availability and Fault Tolerance

- In general architecture terms, the availability of a system is measured according to its ability to fulfill requests.
- But computers can experience all manner of failure, from hardware component failure to network disruption to corruption.
- Any computer is susceptible to these kinds of failure.
- There are of course very sophisticated (and often prohibitively expensive) computers that can themselves mitigate many of these circumstances, as they include internal hardware redundancies and facilities to send notification of failure events and hot swap components.

High Availability

- Cassandra is highly available.
- You can replace failed nodes in the cluster with no downtime, and you can replicate data to multiple data centers to offer improved local performance and prevent downtime if one data center experiences a catastrophe such as fire or flood.