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

Tuneable Consistency

- *Consistency* is an overloaded term in the database world, but for our purposes we will use the definition that a read always returns the most recently written value.
- Consider the case of two customers attempting to put the same item into their shopping carts on an ecommerce site.
- If I place the last item in stock into my cart an instant after you do, you should get the item added to your cart, and I should be informed that the item is no longer available for purchase.
- This is guaranteed to happen when the state of a write is consistent among all nodes that have that data.

Tuneable Consistency

- Cassandra is frequently called “eventually consistent,” which is a bit misleading.
- Out of the box, Cassandra trades some consistency in order to achieve total availability. But Cassandra is more accurately termed “tuneably consistent,” which means it allows you to easily decide the level of consistency you require, in balance with the level of availability.

Static Consistency

- This is sometimes called sequential consistency, and is the most stringent level of consistency.
- It requires that any read will always return the most recently written value.
- In one single-processor machine, this is no problem to observe, as the sequence of operations is known to the one clock.
- But in a system executing across a variety of geographically dispersed data centers, it becomes much more slippery

Static Consistency

- Achieving this implies some sort of global clock that is capable of timestamping all operations, regardless of the location of the data or the user requesting it or how many (possibly disparate) services are required to determine the response.

Casual Consistency

- This is a slightly weaker form of strict consistency. It does away with the fantasy of the single global clock that can magically synchronize all operations without creating an unbearable bottleneck.
- Instead of relying on timestamps, causal consistency takes a more semantic approach, attempting to determine the cause of events to create some consistency in their order.
- It means that writes that are potentially related must be read in sequence. If two different, unrelated operations suddenly write to the same field at the same time, then those writes are inferred not to be causally related.
- But if one write occurs after another, we might infer that they are causally related. Causal consistency dictates that causal writes must be read in sequence.

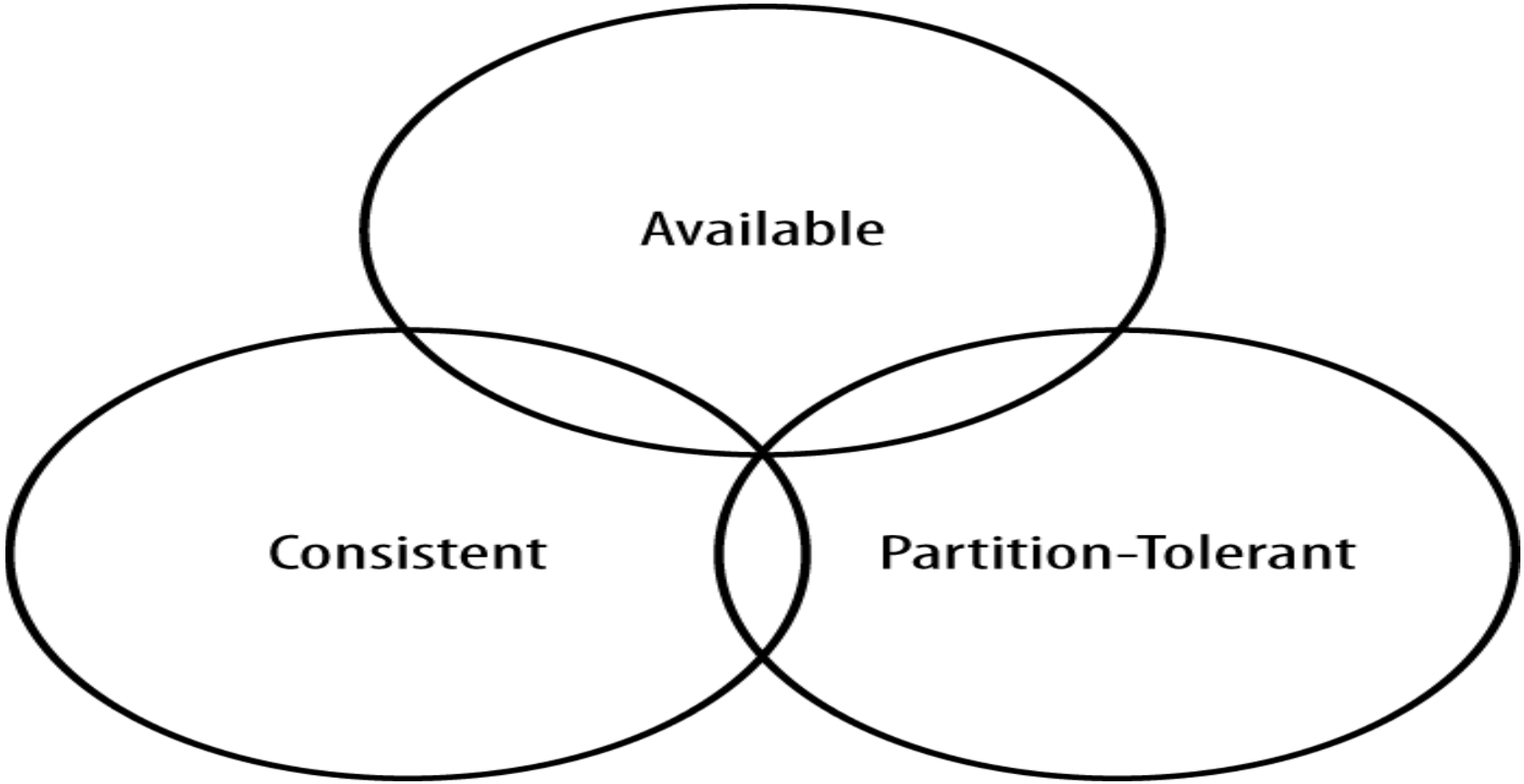
Weak Consistency

- Eventual consistency means on the surface that all updates will propagate throughout all of the replicas in a distributed system, but that this may take some time. Eventually, all replicas will be consistent

Brewer's CAP Theorem

- The CAP theorem is sometimes called Brewer's theorem after its author, Eric Brewer.
- *Consistency*
- All database clients will read the same value for the same query, even given concurrent updates.
- *Availability*
- All database clients will always be able to read and write data.
- *Partition tolerance*
- The database can be split into multiple machines; it can continue functioning in the face of network segmentation breaks.

Brewer's CAP Theorem



Brewer's CAP Theorem

- It's interesting to note that the design of the system around CAP placement is independent of the orientation of the data storage mechanism; for example, the CP edge is populated by graph databases and document-oriented databases alike.
- In this depiction, relational databases are on the line between consistency and availability, which means that they can fail in the event of a network failure (including a cable breaking). This is typically achieved by defining a single primary replica, which could itself go down, or an array of servers that simply don't have sufficient mechanisms built in to continue functioning in the case of network partitions

Brewer's CAP Theorem

- *CA*
- To primarily support consistency and availability means that you're likely using two-phase commit for distributed transactions.
- It means that the system will block when a network partition occurs, so it may be that your system is limited to a single data center cluster in an attempt to mitigate this.
- If your application needs only this level of scale, this is easy to manage and allows you to rely on familiar, simple structures
- *CP*
- To primarily support consistency and partition tolerance, you may try to advance your architecture by setting up data shards in order to scale. Your data will be consistent, but you still run the risk of some data becoming unavailable if nodes fail.

Brewer's CAP Theorem

- *AP*
- To primarily support availability and partition tolerance, your system may return inaccurate data, but the system will always be available, even in the face of network partitioning. DNS is perhaps the most popular example of a system that is massively scalable, highly available, and partition tolerant.

Row Oriented

- Cassandra's data model can be described as a *partitioned row store*, in which data is stored in sparse multidimensional hash tables. "Sparse" means that for any given row you can have one or more columns, but each row doesn't need to have all the same columns as other rows like it (as in a relational model).
- "Partitioned" means that each row has a unique partition key used to distribute the rows across multiple data stores.

High Performance

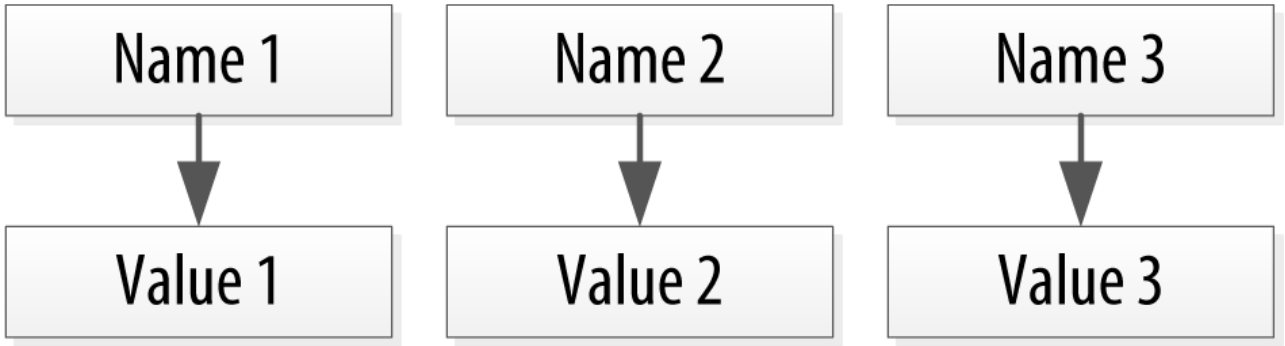
- Cassandra was designed specifically from the ground up to take full advantage of multiprocessor/multicore machines, and to run across many dozens of these machines housed in multiple data centers.
- It scales consistently and seamlessly to hundreds of terabytes. Cassandra has been shown to perform exceptionally well under heavy load.
- It consistently can show very fast throughput for writes per second on basic commodity computers, whether physical hardware or virtual machines.
- As you add more servers, you can maintain all of Cassandra's desirable properties without sacrificing performance

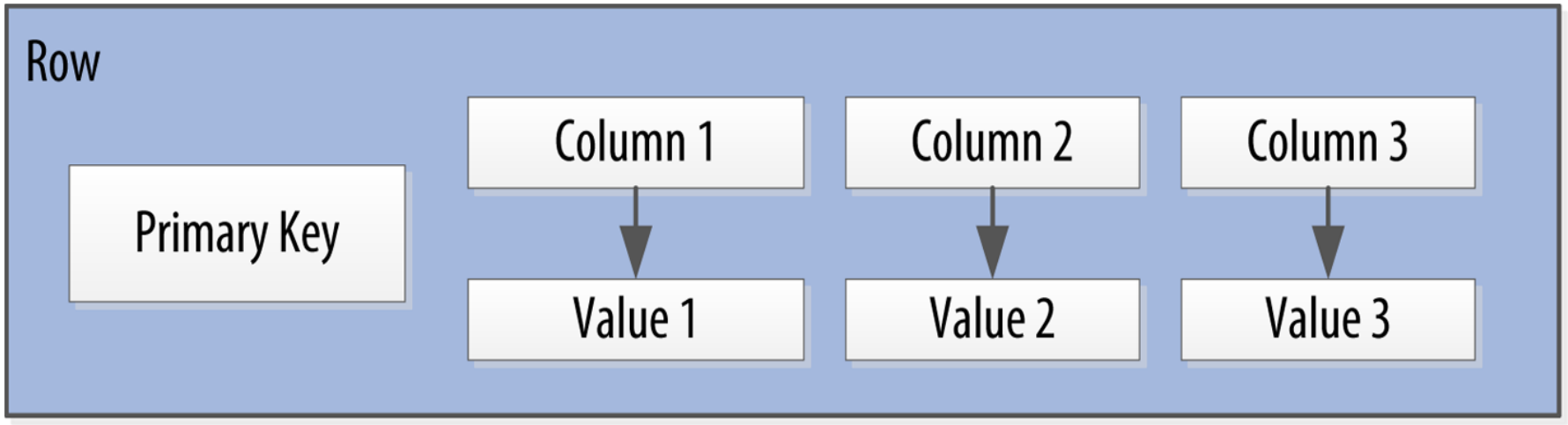
The Cassandra Query Language

Cassandra's Data Model

- The simplest data store you would conceivably want to work with might be an array or list.
- If you persisted this list, you could query it later, but you would have to either examine each value in order to know what it represented, or always store each value in the same place in the list and then externally maintain documentation about which cell in the array holds which values.

Cassandra's Data Model





Cassandra's Data Model

- The *column*, which is a name/value pair
- The *row*, which is a container for columns referenced by a primary key
- The *partition*, which is a group of related rows that are stored together on the same nodes
- The *table*, which is a container for rows organized by partitions
- The *keyspace*, which is a container for tables
- The *cluster*, which is a container for keyspaces that spans one or more nodes

Clusters

- The Cassandra database is specifically designed to be distributed over several machines operating together that appear as a single instance to the end user.
- So the outermost structure in Cassandra is the *cluster*, sometimes called the *ring*, because Cassandra assigns data to nodes in the cluster by arranging them in a ring.

Keyspaces

- A cluster is a container for keyspaces.
- A *keyspace* is the outermost container for data in Cassandra, corresponding closely to a *database* in the relational model.
- In the same way that a database is a container for tables in the relational model, a keyspace is a container for tables in the Cassandra data model.
- Like a relational database, a keyspace has a name and a set of attributes that define keyspace-wide behavior such as replication

Tables

- A *table* is a container for an ordered collection of rows, each of which is itself an ordered collection of columns.
- Rows are organized in partitions and assigned to nodes in a Cassandra cluster according to the column(s) designated as the partition key.
- The ordering of data within a partition is determined by the clustering columns.

Columns

- A *column* is the most basic unit of data structure in the Cassandra data model.
- So far you've seen that a column contains a name and a value. You constrain each of the values to be of a particular type when you define the column.
- You'll want to dig into the various types that are available for each column, but first let's take a look into some other attributes of a column that we haven't discussed yet: timestamps and time to live.
- These attributes are key to understanding how Cassandra uses time to keep data current

Timestamps

- Each time you write data into Cassandra, a timestamp, in microseconds, is generated for each column value that is inserted or updated.
- Internally, Cassandra uses these timestamps for resolving any conflicting changes that are made to the same value, in what is frequently referred to as a *last write wins* approach.

CQL Types

- CQL supports a flexible set of data types, including simple character and numeric types, collections, and user-defined types.
- **Numeric Data Types**
- *int*
- A 32-bit signed integer (as in Java)
- *bigint*
- A 64-bit signed long integer (equivalent to a Java long)
- *smallint*
- A 16-bit signed integer (equivalent to a Java short)
- *tinyint*
- An 8-bit signed integer (as in Java)

CQL Types

- *varint*
- A variable precision signed integer (equivalent to
- java.math.BigInteger)
- *float*
- A 32-bit IEEE-754 floating point (as in Java)
- *double*
- A 64-bit IEEE-754 floating point (as in Java)
- *decimal*
- A variable precision decimal (equivalent to java.math.BigDecimal)

Textual Data Types

- CQL provides two data types for representing text, one of which you've made quite a bit of use of already (text):
- *text, varchar*
- Synonyms for a UTF-8 character string
- *ascii*
- An ASCII character string

Time and Identity Data Types

- The identity of data elements such as rows and partitions is important in any data model in order to be able to access the data.
- Cassandra provides several types that prove quite useful in defining unique partition keys.

timestamp

- Each column has a timestamp indicating when it was last modified, you can also use a timestamp as the value of a column itself.
- The time can be encoded as a 64-bit signed integer, but it is typically much more useful to input a timestamp using one of several supported ISO 8601 date formats. For example:
- 2015-06-15 20:05-0700
- 2015-06-15 20:05:07-0700
- 2015-06-15 20:05:07.013-0700
- 2015-06-15T20:05-0700
- 2015-06-15T20:05:07-0700
- 2015-06-15T20:05:07.013+-0700

date, time

- date and time types that allowed these to be represented independently; that is, a date without a time, and a time of day without reference to a specific date.
- As with timestamp, these types support ISO 8601 formats.

uuid

- A *universally unique identifier* (UUID) is a 128-bit value in which the bits conform to one of several types, of which the most commonly used are known as Type 1 and Type 4.
- The CQL uuid type is a Type 4 UUID, which is based entirely on random numbers. UUIDs are typically represented as dash-separated sequences of hex digits. For example:
- 1a6300ca-0572-4736-a393-c0b7229e193e
- The uuid type is often used as a surrogate key, either by itself or in combination with other values.

timeuuid

- This is a Type 1 UUID, which is based on the MAC address of the computer, the system time, and a sequence number used to prevent duplicates.
- This type is frequently used as a conflict-free timestamp. CQL provides several convenience functions for interacting with the
- timeuuid type: now(), dateOf(), and unixTimestampOf().
- The availability of these convenience functions is one reason why timeuuid tends to be used more frequently than uuid.

timeuuid

- cqlsh:my_keyspace> UPDATE user SET id = uuid() WHERE first_name = 'Mary' AND last_name = 'Rodriguez’; cqlsh:my_keyspace> SELECT first_name, id FROM user WHERE first_name = 'Mary' AND last_name = 'Rodriguez';

Other Simple Data Types

- CQL provides several other simple data types that don't fall nicely into one of the preceding categories:

- *boolean*

- This is a simple true/false value. The cqlsh is case insensitive in accepting these values but outputs True or False.

Other Simple Data Types

- ***blob***
  - A *binary large object* (blob) is a colloquial computing term for an arbitrary array of bytes.
  - The CQL blob type is useful for storing media or other binary file types. Cassandra does not validate or examine the bytes in a blob.
  - CQL represents the data as hexadecimal digits—for example,
  - 0x00000ab83cf0

Other Simple Data Types

- ***inet***
- This type represents IPv4 or IPv6 internet addresses. cqlsh accepts any legal format for defining IPv4 addresses, including dotted or nondotted representations containing decimal, octal, or hexadecimal values.
- However, the values are represented using the dotted decimal format in cqlsh output—for example, 192.0.2.235

Other Simple Data Types

- *counter*

- The counter data type provides a 64-bit signed integer, whose value cannot be set directly, but only incremented or decremented.
- Cassandra is one of the few databases that provides race-free increments across data centers.
- Counters are frequently used for tracking statistics such as numbers of page views, tweets, log messages, and so on.
- The counter type has some special restrictions. It cannot be used as part of a primary key. If a counter is used, all of the columns other than primary key columns must be counters

User-Defined Types

- Cassandra gives you a way to define your own types to extend its data model.
- These user-defined types (UDTs) are easier to use than tuples since you can specify the values by name rather than position.
- Create your own address type:
- cqlsh:my_keyspace> CREATE TYPE address (
- street text,
- city text,
- state text,
- zip_code int);