Building Applications with Apache Cassandra

A quickstart guide to Cassandra for Java developers and architects

Goals - Revisited

By the end of this live, online course, you'll understand:

- Cassandra's architecture and how it works
- How Cassandra is different from traditional databases, and how those differences affect data modeling and application development
- How to identify use cases where Cassandra is a good fit
- Cassandra's features, especially as available through the client drivers
- Commonly used configuration options

Goals - Revisited

By the end of this live, online course, you'll be able to:

- Design Cassandra data models that will perform / scale effectively
- Create application programs using the DataStax Java Driver
- Use tools including *cqlsh*, *nodetool*, and CCM

Unit 4: Cluster Architecture

~35 minutes

Unit 4 Goals

- Learn about common cluster topologies and the role that gossip and snitches play in forming clusters
- Start a small cluster and examine its status and metrics using *nodetool*
- Understand Cassandra's mechanisms for keeping data highly available and in sync across replicas, such as hinted handoff and repair

Exercise 4a – Creating a Cluster

- Goal:
  - Create a “local” cluster consisting of multiple nodes on a single machine, examine its status using *nodetool*, and connect the Reservation Service to it
- Steps:
  - Use the Cassandra Cluster Manager (CCM)
    - https://github.com/pcmanus/ccm

```
$ sudo apt install python-pip # (already done)
$ pip install ccm # (already done)
$ ccm
```

  - Run the command “ccm” in your terminal to get a list of commands
  - Start a 3-node cluster of Cassandra 3.10.0 called “reservation-cluster”

Exercise 4a – Creating a Cluster

- CCM Command:

```
$ ccm create –v 3.11.0 –n 3 reservation_cluster --vnodes
```

- Create the schema in our new cluster
  - Connect to any node using *cqlsh*
  - Use the SOURCE command to load from file

```
$ ccm node1 cqlsh
cqlsh> SOURCE '~/reservation-service/src/main/resources/
reservation.cql';
```

Exercise 4a – Creating a Cluster

- Nodetool
  - To learn commands in general, can invoke nodetool directly from our installation

```
$ nodetool # list commands
$ nodetool help <command>
```

- Connect to a selected node in our CCM cluster with nodetool
  - Use *nodetool* commands including status, tpstats and netstats to examine cluster health

```
$ ccm node1 nodetool status
$ ccm node1 nodetool tpstats
$ ccm node1 nodetool netstats
```

Exercise 4a – Creating a Cluster

- Updating our application:

```
reservation-service$ git checkout exercise-4a
```

- Steps:
  - Examine new configuration class: CassandraConfiguration.java
  - Configure the properties file to point to nodes in our cluster
    - /reservation-service/src/main/resources/application.properties
    - Add: 127.0.0.1, 127.0.0.2
  - Update the way we build the Cluster object
    - use the addConnectionPoints() operation

Exercise 4a – Creating a Cluster

- Finishing up:
  - To save your work (locally):

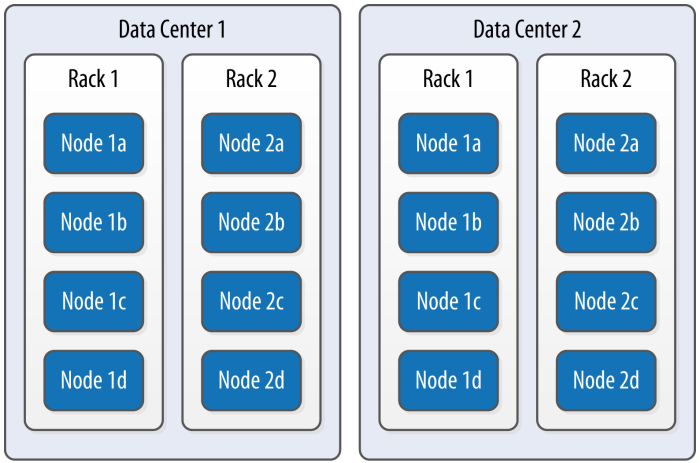
```
reservation-service$ git commit –a –m “my work”
```

  - To see the solution

```
reservation-service$ git checkout exercise-4a-solution
```

- Questions?

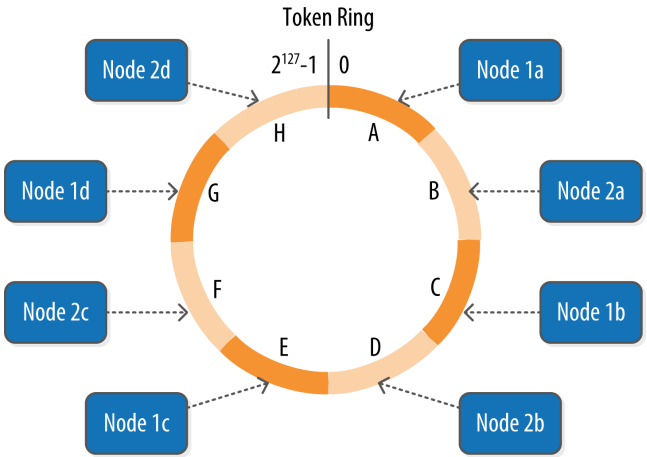
Cluster Topology



- Organization
  - Nodes
  - Racks
  - Data Centers
- Goals
  - Distribute copies (replicas) for high availability
  - Route queries to nearby nodes for high performance
- Approaches
  - Gossip
  - Snitches

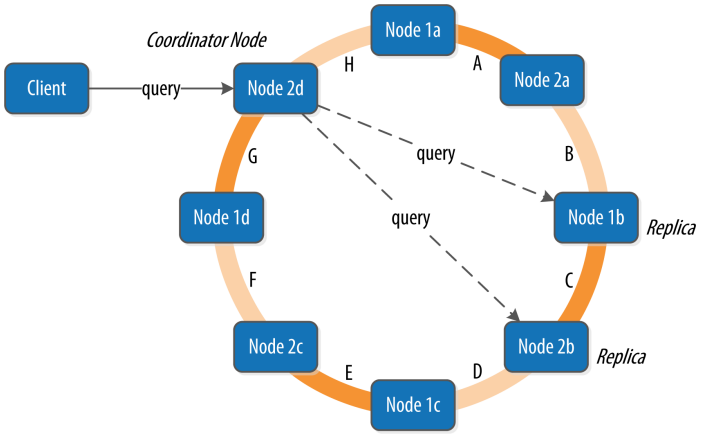
See also: Cassandra, The Definitive Guide, 2nd Edition, Chapter 6: The Cassandra Architecture

Clusters and Rings



- Organization
  - Tokens
  - Token Ring
- Goals
  - Distribute data evenly across nodes
- Approaches
  - Partitioners
  - Virtual nodes

Replication and Consistency



- Organization
  - Client (with driver)
  - Coordinator Node
  - Replica nodes
- Goals
  - Consistent data even when nodes are down/unresponsive
- Approaches
  - Replication Factor / Strategy
  - Consistency Level

Replication Strategy

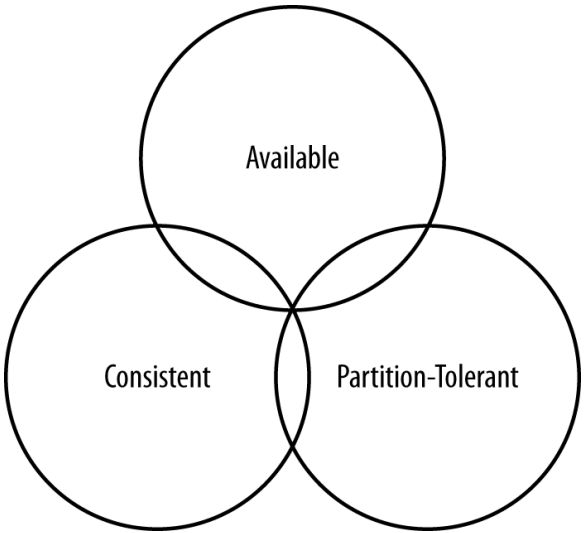
- Replication Factor
  - Number of replicas of your data to be stored
- Replication Strategies
  - SimpleStrategy − useful for development, single data center

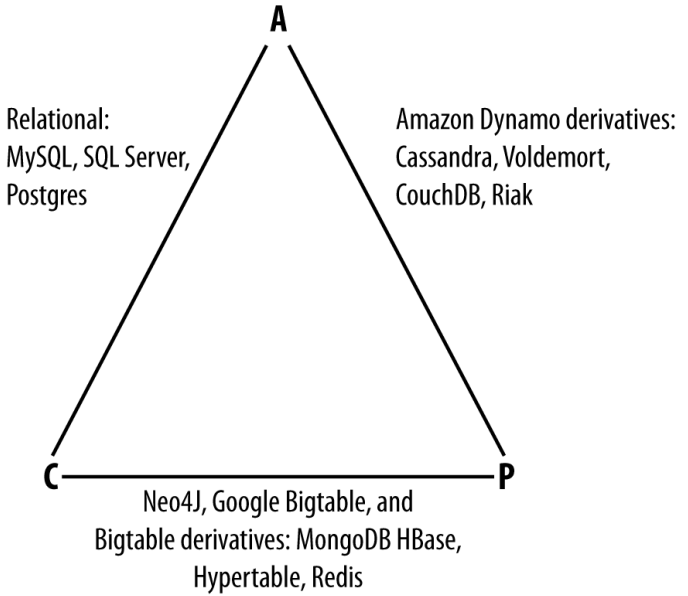
```
CREATE KEYSPACE hotel WITH replication = {'class':'SimpleStrategy',
'replication_factor':1};
```

```
CREATE KEYSPACE hotel_multi_dc WITH replication =
  { 'class':'NetworkTopologyStrategy', 'dc1':3, ’dc2':3, … };
```

  - Changing details in deployed system allowed, but maintenance required
    - *nodetool clean*, *nodetool repair*

The CAP Theorem and Eventual Consistency





Tuneable Consistency and Consistency Levels

- ONE, TWO, THREE
  - Useful for speed
  - ANY (Write only, use with care)
- ALL
  - Number of nodes to respond = RF
  - Overly restrictive?
- QUORUM
  - (RF / 2) + 1
  - Frequently very useful
- LOCAL_ONE, LOCAL_QUORUM
  - Similar to above, but nodes must be in local data center
- EACH_QUORUM
  - Quorum of nodes must respond in each data center

Strong Consistency vs. Eventual Consistency

- Eventual consistency
  - i.e. W → ONE, R → QUORUM
  - Use cases
    - Write heavy
    - Data not read immediately
- Strong Consistency
  - i.e. W→ QUORUM, R → QUORUM
  - Use cases
    - Read after write
    - Data loading with validation

- Strong Consistency Formula
  - *R + W > RF = strong consistency*
    - *R*: read replica count required by consistency level
    - *W*: the write replica count required by consistency level
    - *RF*: replication factor
  - Example: W→ QUORUM, R → QUORUM, RF = 3
    - 2 + 2 > 3
  - Implication: all client reads will see the most recent write

Configuring Consistency

- *cqlsh*

```
cqlsh> CONSISTENCY       # get current level
cqlsh> CONSISTENCY ONE   # set level to one for subsequent statements
```

- DataStax Java Driver - QueryOptions

```
QueryOptions queryOptions = new QueryOptions();
queryOptions.setConsistencyLevel(ConsistencyLevel.ONE);

Cluster cluster = Cluster.builder().addContactPoints(…)
          .withQueryOptions(queryOptions).build();

Statement statement = …

statement.setConsistencyLevel(ConsistencyLevel.QUORUM);
```

See also:

http://docs.datastax.com/en/drivers/java/3.2/com/datastax/driver/core/ConsistencyLevel.html

Exercise 4b – Configuring Consistency Levels

- Goal:
  - Assume our application needs to be able to support reads soon after writes
  - We can configure a default consistency level on the Cluster that will be used for both reads and writes – what should we use?
- Steps

```
reservation-service$ git checkout exercise-4b
```

  - Configure application property **com.cassandraguide.services.reservation.DEFAULT_CONSISTENCY_LEVEL**
  - Update ReservationService.java to create QueryOptions using that property and use those options in building Cluster

Exercise 4b – Configuring Consistency Levels

- Finishing up:
  - To save your work (locally):

```
reservation-service$ git commit –a –m “my work”
```

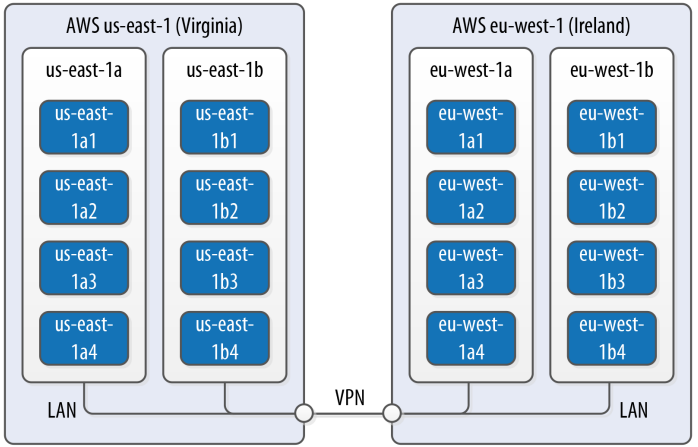
  - To see the solution

```
reservation-service$ git checkout exercise-4b-solution
```

- Questions?

Consistency Level and Deployment

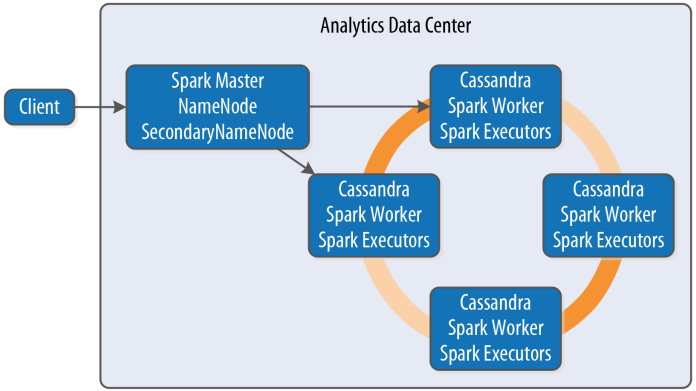
- Active-Active, Multi-Region
  - EACH_QUORUM may limit availability if connection to one region/data center is down
  - LOCAL_QUORUM, relying on Cassandra to complete writes to remote data centers
  - QUORUM is a reasonable middle-ground approach



See also: Cassandra, The Definitive Guide, 2nd Edition, Chapter 14: Deploying and Integrating

Consistency Level and Deployment

- Separate data center for analytics
  - Writes in ”online DC” – LOCAL_QUORUM
    - Writes to analytics DC in background
    - Analytic DC availability/performance decoupled from online DC
    - EACH_QUORUM, QUORUM overkill, unless “real-time” analytics required
  - Reads in “analytics DC” – LOCAL_QUORUM
    - Or even LOCAL_ONE



See also: Cassandra, The Definitive Guide, 2nd Edition, Chapter 14: Deploying and Integrating

Break

5 minutes

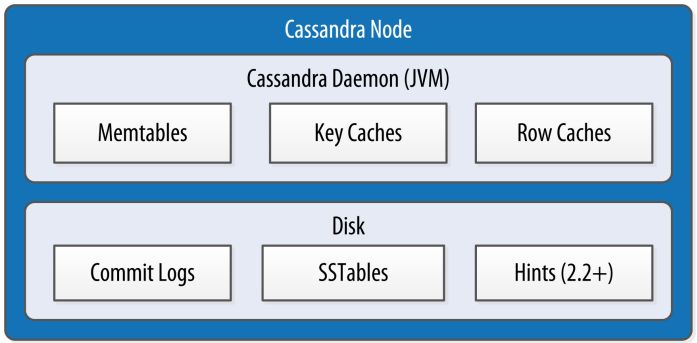
Unit 5: Read and Write Paths

60 minutes

Unit 5 Goals

- Understand how Cassandra handles our queries on the “read path” and “write path”, including mechanisms like hinted handoff and repair
- Learn how to perform more advanced write operations such as batches and lightweight transactions, and when it is appropriate to use them
- Learn how to perform more advanced read operations such as searching and paging

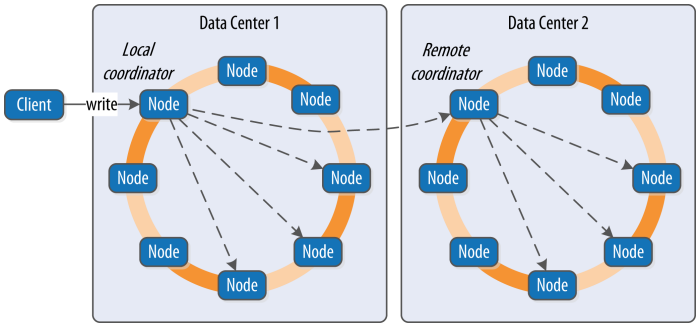
Cassandra Internals



- Organization
  - In-memory storage (memtables, caches, indexes)
  - On-disk storage (commit logs, SSTables, hints files)
- Goals
  - Fast reads and writes
  - Maximum availability of data

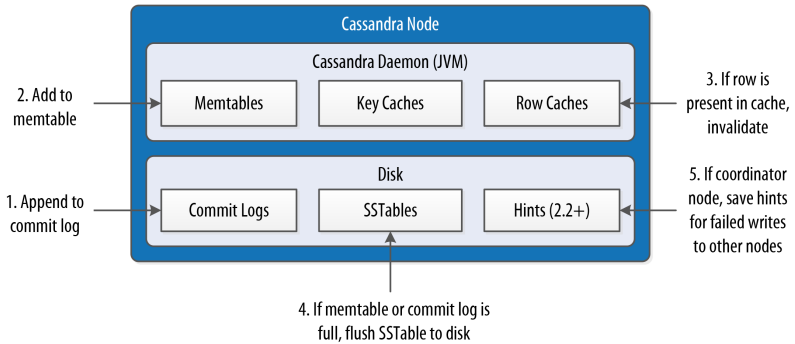
See also: Cassandra, The Definitive Guide, 2nd Edition, Chapter 9: Reading and Writing Data

Cassandra Write Path



- Client (using driver) selects coordinator node in local data center
  - Preferably a replica
- Coordinator records write locally and forwards to replica nodes in local DC and remote coordinator
- Coordinator can return once enough nodes respond to satisfy requested consistency level

Cassandra Write Path - Internals



- Immediately append to commit log
  - Only used when node is recovering from crash
- Add to memtable
- Acknowledge write to coordinator
- Update caches (if used)
- Flush memtables to disk as SSTables (as needed)
  - Compaction runs later (as needed)
- Store hints (on coordinator node as needed)

Hinted Handoff

- Coordinator node stores hints
  - Writes for nodes that are temporarily down/unreachable
- Does not count as a write
  - Exception: Consistency level ANY
- Hints shared when node comes back online
- Hints kept for a limited time
  - Default 3 hours
- After timeout, repair required

Write Consistency Levels

Consistency level	Implication
ANY	Ensure that the value is written to a minimum of one replica node before returning to the client, allowing hints to count as a write.
ONE, TWO, THREE	Ensure that the value is written to the commit log and memtable of at least one, two, or three nodes before returning to the client.
LOCAL_ONE	Similar to ONE, with the additional requirement that the responding node is in the local data center.
QUORUM	Ensure that the write was received by at least a majority of replicas ((RF / 2) + 1).
LOCAL_QUORUM	Similar to QUORUM, where the responding nodes are in the local data center.
EACH_QUORUM	Ensure that a QUORUM of nodes respond in each data center.
ALL	Ensure that the number of nodes specified by replication factor received the write before returning to the client. If even one replica is unresponsive to the write operation, fail the operation.

Batches

- Group modifications on multiple partitions into a single statement
- Atomic, not isolated
- Counter batch used for counters

```
cqlsh> BEGIN BATCH
  INSERT INTO hotel.hotels
    (id, name, phone) VALUES ('AZ123',
    'Super Hotel at WestWorld',
    '1-888-999-9999');
  INSERT INTO hotel.hotels_by_poi (
    poi_name, id, name, phone)
    VALUES ('West World', 'AZ123',
    'Super Hotel at WestWorld',
    '1-888-999-9999');
APPLY BATCH;
```

```
Statement hotelInsert,
hotelsByPoiInsert;
…
BatchStatement hotelBatch =
  new BatchStatement();
hotelBatch.add(hotelsByPoiInsert);
hotelBatch.add(hotelInsert);
```

Exercise 5a – Batches

- Goal:
  - Incorporate additional tables from the data model design into our application
  - Learn how to add additional consistency to writes by using batch
- Steps

```
reservation-service$ git checkout exercise-5a
```

  - Add PreparedStatements to create, update, delete on "reservations_by_hotel_date" table
  - Create BatchStatements to wrap operations using multiple tables
    - createReservation()
    - updateReservation()
    - deleteReservation()

Exercise 5a – Batches

- Discussion
  - Have we ensured uniqueness across multiple tables?
  - What problem did we encounter in the update method?
  - How do we handle data migration when adding a new table?
- Finishing up:
  - To save your work (locally):

```
reservation-service$ git commit –a –m “my work”
```

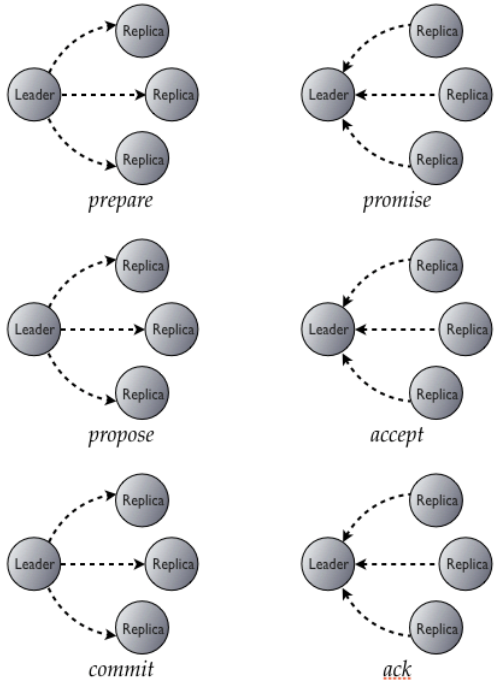
  - To see the solution

```
reservation-service$ git checkout exercise-5a-solution
```

Lightweight Transactions

- Serializable consistency, not ACID
- Based on Paxos algorithm
- Limited to a single partition
- Serial consistency level

Consistency level	Implication
SERIAL	A quorum of nodes must respond to the negotiation (default)
LOCAL_SERIAL	Similar to SERIAL, but indicates that the transaction will only involve nodes in the local data center.



Lightweight Transactions

- Syntax
  - Ensure uniqueness on creation - IF NOT EXISTS

```
INSERT INTO hotel.hotels (id, name, phone)
    VALUES ('AZ123', 'Super Hotel at WestWorld', '1-888-999-9999') IF
NOT EXISTS;
```

```
UPDATE hotel.hotels SET name='Super Hotel Suites at WestWorld' ...
WHERE id='AZ123'
    IF name='Super Hotel at WestWorld';
```

Exercise 5b – Lightweight Transactions

- Goal:
  - Learn how to add additional consistency to writes to individual rows by using Cassandra's Lightweight Transactions
- Steps

```
reservation-service$ git checkout exercise-5b
```

  - Modify PreparedStatements to ensure uniqueness
    - Insert into reservations_by_confirmation has unique confirmation number
    - Update to reservations_by_confirmation
      - don't allow changes to fields in primary key of reservations_by_hotel_date

Exercise 5b – Lightweight Transactions

- Finishing up:
  - To save your work (locally):

```
reservation-service$ git commit –a –m “my work”
```

  - To see the solution

```
reservation-service$ git checkout exercise-5b-solution
```

- Questions?

Searching and the SELECT command

- Best practice: Design tables to support planned queries
- WHERE clause
  - Composed of relations on primary key columns and columns with secondary indexes
  - Queries (other than table scans) must include the partition key
  - Relations other than equality not allowed on a partition key
    - IN clause is considered an equality relation
  - Selection of clustering columns and relations must indicate a contiguous set of rows
- ORDER BY and GROUP BY clauses
  - See documentation

See also: http://cassandra.apache.org/doc/latest/cql/dml.html?highlight=select

Secondary Indexes

- Original implementation − doesn't scale well

```
CREATE INDEX ON user ( last_name );
SELECT * FROM user WHERE last_name = 'Nguyen';
```

- SSTable Attached Storage Index (SASI) − better but not great

```
CREATE CUSTOM INDEX user_last_name_sasi_idx ON user (last_name) USING
'org.apache.cassandra.index.sasi.SASIIndex';
SELECT * FROM user WHERE last_name LIKE 'N%';
```

- Other search options − best performance
  - Elassandra (Elastic Search + Cassandra), Stratio Lucene Index
  - DataStax Enterprise Search (Based on Apache SOLR)

Materialized Views

- Let Cassandra maintain multiple views − higher performance
- Added in 3.0 release based on new storage engine

```
CREATE MATERIALIZED VIEW reservation.reservations_by_confirmation
   AS SELECT * FROM reservation.reservations_by_hotel_date   # Base
table
   WHERE confirmation_number IS NOT NULL and hotel_id IS NOT NULL and
start_date IS NOT NULL and room_number IS NOT NULL   # Filters
   PRIMARY KEY (confirm_number, hotel_id, start_date, room_number);
```

- Must provide a filter for every primary key column (at least IS NOT NULL)

```
SELECT * FROM reservations_by_confirmation WHERE
confirmation_number='LCY1BM'
```

Exercise 5c – Materialized Views

- Goal:
  - Use materialized views to simplify application creation and eliminate code required to maintain multiple tables
- Steps

```
reservation-service$ git checkout exercise-5c
```

  - Update our schema to drop the original reservations_by_confirmation table and re-create it as a materialized view on reservations_by_hotel_date

```
cqlsh> SOURCE '~/reservation-service/src/main/resources/reservation_mv.cql'
```

  - Eliminate writes to reservations_by_confirmation (keeping the read)
  - Implement the searchReservationsByHotelDate() method

Exercise 5c – Materialized Views

- Added command to simple_test_script.sh to test search by hotel/

```
curl 'http://localhost:8080/reservations/?
hotelId=NY456&startDate=2017-06-08'
```

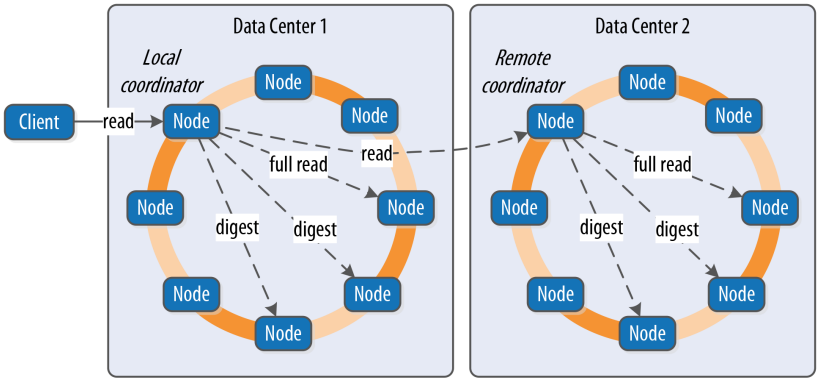
- Finishing up:

```
reservation-service$ git commit –a –m “my work”
```

```
reservation-service$ git checkout exercise-5c-solution
```

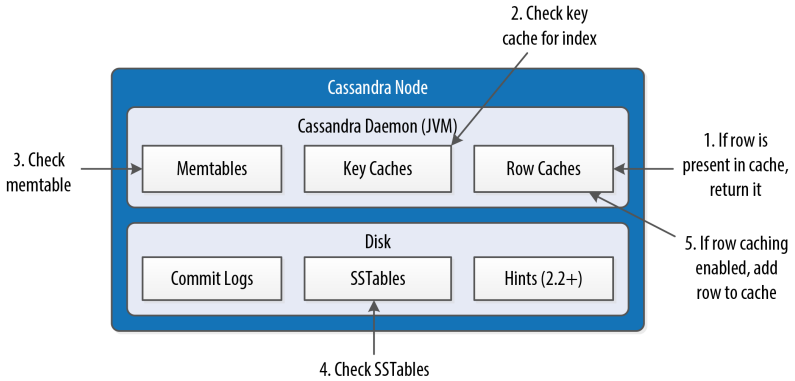
- Questions?

Cassandra Read Path



- Client (using driver) selects coordinator node in local data center
  - Preferably a replica
- Coordinator requests data
  - Full record from one replica
  - Digests from other replicas
- Coordinator reconciles results
  - Read repair if necessary
- Can return once enough replicas agree

Cassandra Read Path - Internals



- Check row cache (if enabled)
  - Return data if found
- Check memtable
- Check SSTables
- Merge results
- Update row cache (if enabled)

Read Consistency Levels

Consistency level	Implication
ONE, TWO, THREE	Immediately return the record held by the first node(s) that respond to the query. A background thread is created to check that record against the same record on other replicas. If any are out of date, a *read repair* is then performed.
LOCAL_ONE	Similar to ONE, with the additional requirement that the responding node is in the local data center.
QUORUM	Query all nodes. Once a majority of replicas ((RF / 2) + 1) respond, return to the client the value with the most recent timestamp. Then, if necessary, perform a read repair in the background on all remaining replicas.
LOCAL_QUORUM	Similar to QUORUM, where the responding nodes are in the local data center.
EACH_QUORUM	Ensure that a QUORUM of nodes respond in each data center.
ALL	Query all nodes. Wait for all nodes to respond, and return to the client the record with the most recent timestamp. Then, if necessary, perform a read repair in the background. If any nodes fail to respond, fail the read operation.

Repair

- Read repair
  - Performed when out of date values detected on a read
  - Or as a background task following a percentage of reads
    - read_repair_chance table property
- Anti-entropy repair
  - Uses Merkle trees for efficient comparison, but still resource intensive
  - Options
    - Full, incremental
  - Tooling
    - *nodetool repair*
    - Cassandra reaper (Apache Cassandra)
    - OpsCenter Repair Service (DataStax Enterprise)

Paging

- Prevents resource issues such as out of memory for large result sets
- Transparent to client unless desired
- Enabled by default − default page size is 5000
- Can override default on cluster object, or set on individual statement

```
Cluster cluster = Cluster.builder().addContactPoint(…)
    .withQueryOptions(new QueryOptions().setFetchSize(2000)).build();

statement.setFetchSize(2000);
```

See also:
http://docs.datastax.com/en/developer/java-driver/3.2/manual/paging/

Paging

- ResultSet API allows fine grained control of paging

```
for (Row row : resultSet) {

  if (resultSet.getAvailableWithoutFetching() < 100 && !resultSet.isFullyFetched())
          resultSet.fetchMoreResults();

    // process the row

}
```

- Preserving paging state across multiple application invocations

```
PagingState nextPage = resultSet.getExecutionInfo().getPagingState();

// convert to string and pass to caller

// on next call from client, paging state is passed back in.
hotelSelect.setPagingState(pagingState);
```

Paging

- Paging in *cqlsh*
  - Enable / Disable paging (enabled by default)

```
cqlsh> PAGING ON;
cqlsh> PAGING OFF;
```

  - Set page size

```
cqlsh> PAGING 1000;
```

  - Limit results (sampling)

```
cqlsh> SELECT * FROM hotel.hotels LIMIT 100;
```

Break

5 minutes

Unit 6: Advanced Client Development

60 minutes

Unit 6 Goals

- Learn how to view and interpret logging and tracing reports to debug and analyze application performance issues
- Understand how Cassandra handles deletion and tombstones
- Interact with advanced driver features such as logging, codecs, and policies
- Gain experience using asynchronous (non-blocking) interactions
- Awareness of other configuration and tuning options for the DataStax Java Driver

Advanced Driver Settings – Logging

- Loggers
  - SLF4J API – pluggable implementation
  - Reservation Service uses Logback implementation (spring-boot-starter-web)
  - Can set log levels in application.properties

```
logging.level.com.datastax.driver.core.Cluster=DEBUG
```

- QueryLogger
  - Tracks query latency and errors
  - Reports via 3 different Loggers: NORMAL, SLOW, and ERROR
  - Create and register with Cluster

See also: https://docs.datastax.com/en/developer/java-driver/3.2/manual/logging/

Advanced Driver Settings – Metrics

- Metrics
  - collected and reported using Dropwizard Metrics Library

```
Metrics metrics = cluster.getMetrics();
```

- JMX integration
  - Enabled by default

See also:
http://docs.datastax.com/en/drivers/java/3.2/com/datastax/driver/core/Metrics.html

Tracing

- Enable tracing to see interactions between nodes
  - Can help diagnose why queries are slow
- Available on any CQL client
  - DevCenter – enabled by default, just switch over to tracing tab
  - *cqlsh* – enable via TRACING command

```
cqlsh> TRACING ON;
```

  - DataStax Drivers: ExecutionInfo, QueryTrace

```
statement.enableTracing();
statement.execute().getExecutionInfo().getQueryTrace();
```

See also:

https://academy.datastax.com/resources/datastax-java-driver-request-tracing

Exercise 6a – Logging and Tracing

- Goal:
  - Gain experience accessing and interpreting logging and tracing output
- Steps

```
reservation-service$ git checkout exercise-6a
```

  - Configure log settings on the DataStax Java Driver to output debug log information related for the Cluster class “com.datastax.driver.core.Cluster”
    - Hint: application.properties
  - In remaining time, enable/view tracing in *cqlsh*, DevCenter and logs while running additional commands

Exercise 6a – Logging and Tracing

- Finishing up:
  - To save your work (locally):

```
reservation-service$ git commit –a –m “my work”
```

  - To see the solution

```
reservation-service$ git checkout exercise-6a-solution
```

- Questions?

Deletion and Tombstones

- Append-only storage model − SSTables are immutable
- Tombstones used to explicitly indicate deleted data
  - Prevent accidental restoration of deleted data
- Data actually cleaned up during compaction
- Large numbers of tombstones can affect reads
  - Example log output

```
Read 1 live and 123780 tombstoned cells | 19:48:36,710 | 127.0.0.1 | 128631
```

Compaction

- Combining multiple SSTables into a single table, while eliminating obsolete values, to reduce size of data on disk
  - Deleted data
  - Data superseded by later writes
  - Expired data (Time to Live)
- Runs as a background process
- Takeaway: consider the impact of many updates and deletions

- Select a CompactionStrategy per table
  - Size tiered – default, recommended for write-intensive loads
  - Leveled - recommended for read-intensive loads
  - Date tiered compaction strategy
  - Time window compaction strategy

Time to Live (TTL)

- Let Cassandra handle the cleanup of data you no longer need
- Set TTL on INSERT or UPDATE by a length of time in seconds
- Provided TTL applied applies to columns referenced in the query
- Use cases:
  - Reservation/transaction data that is no longer relevant
  - Data that can be archived for historical purposes

DataStax Java Driver – Codecs

- Configurable mapping of CQL types to arbitrary Java objects.
  - java.time.LocalDate ⇔CQL date
- Requires additional Maven dependency
- Register codecs with the Cluster

```
<dependency>

<groupId>com.datastax.cassandra</groupId>
  <artifactId>cassandra-driver-extras</artifactId>
  <version>3.2.0</version>
</dependency>
```

```
import com.datastax.driver.extras.codecs.jdk8.L

cluster.getConfiguration().getCodecRegistry().register(LocalDateCodec.instance);
```

See also:
http://docs.datastax.com/en/developer/java-driver/3.2/manual/custom_codecs/
http://docs.datastax.com/en/developer/java-driver/3.2/manual/custom_codecs/extras/

Exercise 6b – Codecs

- Goal:
  - Eliminate awkward type conversion code from the Reservation Service
- Steps

```
reservation-service$ git checkout exercise-6b
```

  - Add Maven dependency for codecs provided by the DataStax Java Driver
  - Register the LocalDateCodec with the Cluster
  - Remove messy conversion code that is no longer needed
  - Use the Row.get(String, Class) method to retrieve date attributes from row

```
java.time.LocalDate startDate = row.get("start_date",
java.time.LocalDate.class)
```

Exercise 6b - Codecs

- Finishing up:
  - To save your work (locally):

```
reservation-service$ git commit –a –m “my work”
```

  - To see the solution

```
reservation-service$ git checkout exercise-6c-solution
```

- Questions?

DataStax Java Driver Configuration/Tuning

- Resource management
  - PoolingOptions (per session)
    - Requests per connection
    - Min/max connections
- Networking
  - NettyOptions
  - SocketOptions
  - Address resolution
  - Socket settings, TCP timeouts, etc.
- Query Options
  - Default consistency level
  - Default page size
- Policies (more detail to come)
  - LoadBalancingPolicy
  - RetryPolicy
  - Reconnection Policy
  - SpeculativeExecutionPolicy

See also: Cassandra, The Definitive Guide, 2nd Edition, Chapter 9: Clients

DataStax Java Driver – Load Balancing Policy

- Policy for determining the “query plan”
  - i.e. the order of preference in which nodes will be used to service queries
  - TokenAwarePolicy - targets nodes responsible for a given token
    - If the coordinator node is one of the replicas, that saves a step
  - DCAwareRoundRobinPolicy - used to prefer ”local" data center nodes
  - Other options: LatencyAwarePolicy, RoundRobinPolicy, WhiteListPolicy
- Chainable/Nestable

```
Cluster cluster = Cluster.builder().addContactPoint(…)
    .withLoadBalancingPolicy(new TokenAwarePolicy(
        new DCAwareRoundRobinPolicy.builder().build()));
```

See also: http://docs.datastax.com/en/developer/java-driver/3.2/manual/load_balancing/

DataStax Java Driver – Reconnection Policy

- Driver maintains connections to nodes in the cluster behind the scenes
- When nodes are detected to be down, they are omitted from query plans
- ReconnectionPolicy determines how often the driver attempts to reconnect to down nodes
  - ConstantReconnectionPolicy
  - ExponentialReconnectionPolicy (default)

See also: http://docs.datastax.com/en/developer/java-driver/3.2/manual/reconnection/

DataStax Java Driver – Retry Policy

- Queries can fail due to networking issues, and the driver can retry
- RetryPolicy controls retry decisions for the driver based on the error
  - Retry on same host or next host in query plan, and at what consistency level
  - DefaultRetryPolicy – conservative in recommending retries
  - FallthroughRetryPolicy – never retries
  - DowngradingConsistencyRetryPolicy – retries with lower consistency
    - Not recommended
  - LoggingRetryPolicy – wraps other implementations and logs decisions

```
Cluster cluster = Cluster.builder().addContactPoint(…).
    withRetryPolicy(new
LoggingRetryPolicy(DowngradingConsistencyRetryPolicy.INSTANCE));
```

See also: http://docs.datastax.com/en/developer/java-driver/3.2/manual/retries/

DataStax Java Driver SpeculativeExecutionPolicy

- Working around slow-responding nodes
  - Network congestion, garbage collection, high utilization
  - Waiting for a timeout is wasteful

```
Statement.setReadTimeoutMillis(2000)
```

- SpeculativeExecutionPolicy
  - If coordinator hasn't responded, move to the next node in the query plan
  - ConstantSpeculativeExecutionPolicy – fixed delay
  - PercentileSpeculativeExecutionPolicy – based on observed latency to a node

```
Cluster cluster = Cluster.builder().addContactPoint(…).withSpeculativeExecutionPolicy(
        new ConstantSpeculativeExecutionPolicy(200 /* ms delay */, 3 /* spec executions */));
```

See also:
http://docs.datastax.com/en/developer/java-driver/3.2/manual/speculative_execution/

Exercise 6c – Policies

- Goal:
  - Configure driver to use common policy settings
- Steps

```
reservation-service$ git checkout exercise-6c
```

  - Update the cluster creation code to
    - Configure the load balancing policy to be token aware, round robin (not DC aware)
    - Enable a 99 percentile speculative execution policy with a maximum of 2 speculative executions

Exercise 6c - Policies

- Finishing up:
  - To save your work (locally):

```
reservation-service$ git commit –a –m “my work”
```

  - To see the solution

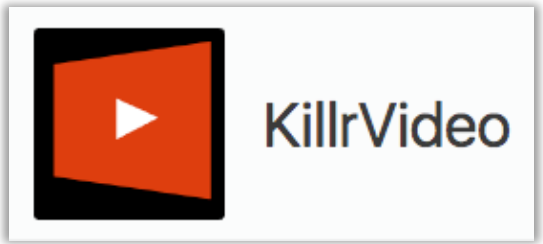
```
reservation-service$ git checkout exercise-6c-solution
```

- Questions?

Asynchronous Execution

- DataStax Java Driver provides asynchronous APIs for non-blocking operations
- Async methods return instances of Guava ListenableFuture
- Examples:

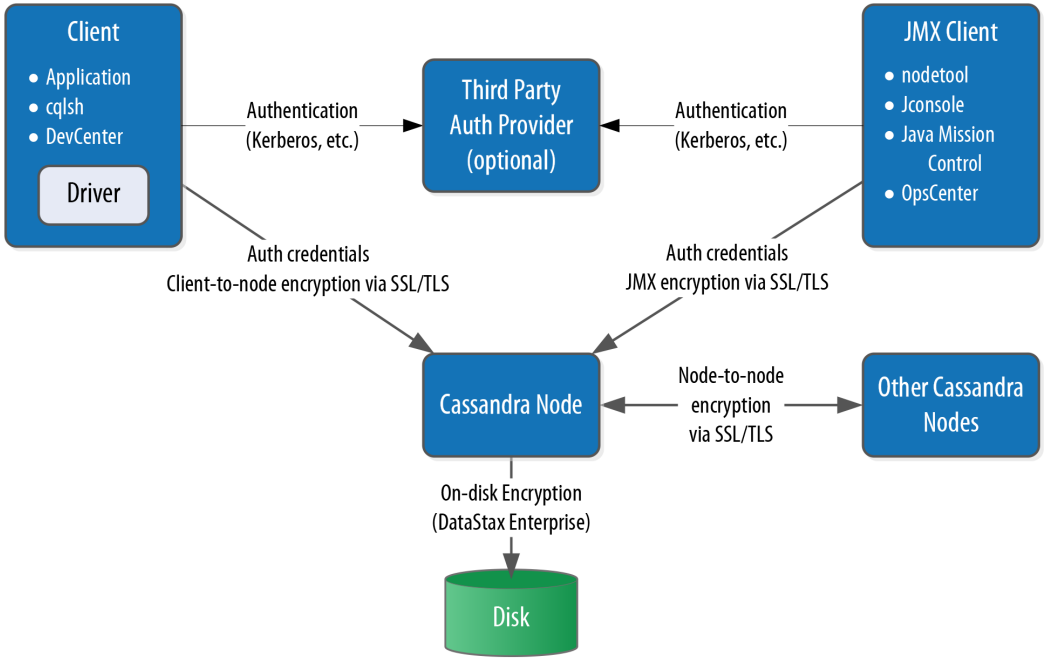
```
Cluster.connectAsync()
Session.executeAsync()
PreparedStatement.prepareAsync()
Mapper<T>.getAsync()
Mapper<T>.saveAsync()
Mapper<T>.deleteAsync()
```



developer/java-driver/3.2/manual/async/

Cassandra Security

- Features
  - Authentication
  - Authorization
  - Encryption
- Characteristics
  - Role-based
  - Pluggable



See also: Cassandra, The Definitive Guide, 2nd Edition, Chapter 13: Security

DataStax Java Driver Security – Authentication

- Basic username/password authentication supported natively
  - Pluggable authentication – DSE uses to add Kerberos, LDAP support
- Configure cassandra.yaml file on nodes to use PasswordAuthenticator

```
authenticator: ~~AllowAllAuthenticator~~ PasswordAuthenticator
```

- Create users using CQL

```
cqlsh> CREATE USER jeff WITH PASSWORD 'i6XJsj!k#9';
```

- Clients pass credentials when building Cluster (configure externally)

```
Cluster cluster = Cluster.builder().addContactPoint(…)
  .withCredentials("jeff", "i6XJsj!k#9").build(); // should load from
configuration
```

See also: Cassandra, The Definitive Guide, 2nd Edition, Chapter 13: Security

DataStax Java Driver Security – Authorization

- Enable via cassandra.yaml configuration file on each node

```
authorizer: AllowAllAuthorizer CassandraAuthorizer
```

- Users assigned to roles, permissions granted to users and roles

```
cqlsh> CREATE USER jeff WITH PASSWORD 'i6XJsj!k#9';
GRANT SELECT ON hotel.hotels TO jeff;
cqlsh> CREATE ROLE hotel_management;
cqlsh> GRANT ALL ON KEYSPACE hotel TO hotel_management;
cqlsh> GRANT hotel_management TO jeff;
```

application – except UnauthorizedException

See also: Cassandra, The Definitive Guide, 2nd Edition, Chapter 13: Security

DataStax Java Driver Security – Encryption

- Supports SSL encryption between your client and Cassandra nodes
- Request SSL when building Cluster

```
Cluster cluster = Cluster.builder().addContactPoint(…)
  .withCredentials(…).withSSL().build();
```

- Configure SSL using JSSE (Java Secure Socket Extension) or Netty
  - In code or via configuration file
- Requires distribution of keys to every node in the Cluster
  - Self-signed keys recommended for development only

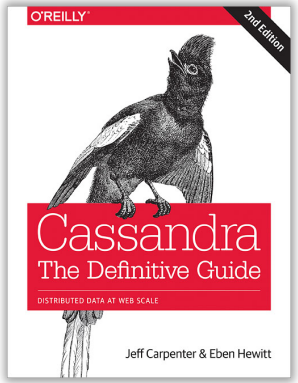
See also: Cassandra, The Definitive Guide, 2nd Edition, Chapter 13: Security

Unit and Integration Testing

- Unit Testing
  - What are you trying to unit test?
  - Mocking − Mockito, various libraries
  - Embedded C* - scassandra, cassandra-unit
- Integration Testing
  - External cluster − CCM
  - Docker
  - Load testing - JMeter

Continue your learning

- Finish up activities
- Reinforce learning with readings:
  - Chapter 8 – Clients
  - Chapter 9 – Reading and Writing



- Free self-paced courses at DataStax Academy
  - DS220: Data Modeling
  - DS310: DataStax Enterprise Search
  - DS320: DataStax Enterprise Analytics with Apache Spark™
  - DS330: DataStax Enterprise Graph

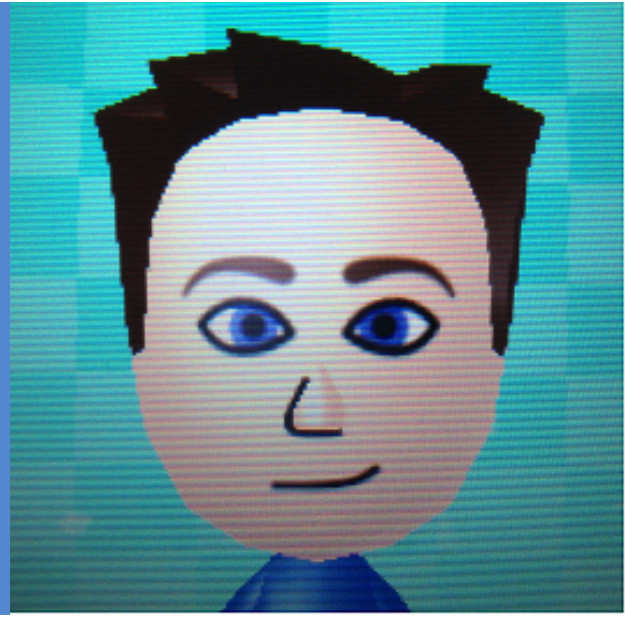
Let’s Stay in Contact

jeff.carpenter@datastax.com

@jscarp

jeffreyscarpenter

Blog: medium.com/@jscarp



End of Part 2

Thanks for joining this course!