

# DataStax Developer Days



# KillrVideo - a video sharing application

The screenshot displays a user interface for a video sharing application. At the top, there's a navigation bar with a dark grey background. On the right side of the bar, there are three items: a small icon followed by the text "Tour: Off", a link "What is this? ▾", and a small "D" icon. Below the navigation bar, the main content area has a teal header section on the left and a white content area on the right.

The content area contains a grid of video thumbnails. The first two rows each have four thumbnails. The third row has three thumbnails, and the fourth row has two thumbnails. Each thumbnail includes a video preview image, the title, the source channel, and the publish date.

- Row 1:**
  - THE SECRET SCRIPTURE TRAILER #1 (2017) | MOVIECLIPS TRAILERS**  
by Carmel Simonis  
1 views • an hour ago
  - THE GUARDIAN BROTHERS TRAILER #1 (2017) | MOVIECLIPS TRAILERS**  
by Austin Dooley  
1 views • an hour ago
  - FERNAND TRAILER #3 (2017) | MOVIECLIPS TRAILERS**  
by Jose Labadie  
4 views • 2 hours ago
  - THE MEYEROWITZ TRAILER #1 (2017) | MOVIECLIPS TRAILERS**  
by Taylor Turner  
13 views • 2 hours ago
- Row 2:**
  - TEST CAT VIDEOS FUNNY KITTEN SAYING**  
by Arma Witting  
3 views • 10 days ago
  - KEY & PEELE - PEGASUS SIGHTING**  
by Willie Bradtke  
2264 views • 12 days ago
  - HOME SWEET HELL OFFICIAL TRAILER #1 (2015) - KATHERINE HEIGL**  
by Newton Reinger  
2315 views • 12 days ago
  - INTRODUCTION TO THE CENTER: OVERVIEW**  
by Newton Reinger  
2499 views • 12 days ago

# KillrVideo-java

<https://github.com/KillrVideo/killrvideo-java>

The screenshot shows the homepage of the KillrVideo Java application. At the top, there's a navigation bar with a logo, 'Get Started', 'Docs', 'Blog', and a GitHub icon. Below the navigation is a section featuring three devices (laptop, tablet, smartphone) displaying the KillrVideo interface. A large green banner below them contains the text 'Learn to build applications with Apache Cassandra and DataStax Enterprise'. Underneath this, there's a section titled 'What is KillrVideo?' with a brief description: 'KillrVideo is a reference application for developers looking to learn how to build applications with Apache Cassandra and DataStax Enterprise.' At the bottom of this section is a blue 'Get Started' button.

<https://killrvideo.github.io>

The screenshot shows the GitHub repository page for 'KillrVideo'. The header includes the repository name, a description ('KillrVideo is a reference application for developers looking to learn how to build applications with Apache Cassandra and DataStax Enterprise.'), and a link to the info and docs site at <https://killrvideo.github.io>. Below the header, there are sections for 'Repositories' (12), 'People' (5), 'Teams' (2), 'Projects' (4), and 'Settings'. A 'Pinned repositories' section lists three repositories: 'killrvideo-nodejs' (JavaScript, 8 stars, 6 forks), 'killrvideo.github.io' (JavaScript, 3 stars, 2 forks), and 'killrvideo-data' (Groovy, 2 stars, 2 forks). Further down, a detailed view of the 'killrvideo-java' repository is shown, including its description ('Java implementation for KillrVideo project'), fork information ('Forked from doanduyhai/killrvideo-java'), language ('Java'), star count (5 stars), fork count (4 forks), and last update ('Updated a day ago'). To the right of the repository details, there are sections for 'Top languages' (JavaScript, Shell, Java, Groovy, PowerShell) and 'Most used topics'. A 'New' button is located in the top right corner of the main search area.

<https://github.com/KillrVideo>

# pom.xml

(get the driver)

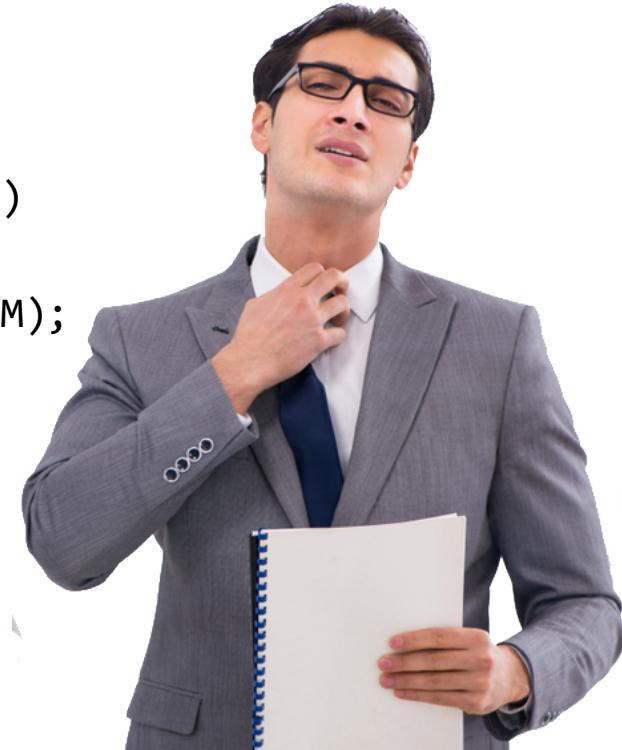
```
<dependency>
  <groupId>com.datastax.dse</groupId>
  <artifactId>dse-java-driver-core</artifactId>
  <version>1.6.8</version>
</dependency>
```



[Getting the driver](#) [POM Example](#)



```
PreparedStatement createUser = dseSession.prepare(  
    QueryBuilder  
        .insertInto("killrvideo", usersTableName)  
        .value("userid", QueryBuilder.bindMarker())  
        .value("firstname", QueryBuilder.bindMarker())  
        .value("lastname", QueryBuilder.bindMarker())  
        .value("email", QueryBuilder.bindMarker())  
        .value("created_date", QueryBuilder.bindMarker())  
        .ifNotExists() // use lightweight transaction  
    ).setConsistencyLevel(ConsistencyLevel.LOCAL_QUORUM);
```



```
BoundStatement insertUser = createUser.bind()  
    .setUUID("userid", userIdUUID)  
    .setString("firstname", firstName)  
    .setString("lastname", lastName)  
    .setString("email", email)  
    .setTimestamp("created_date", now);
```

```
DseSession session = configureAndCreateADseSession()
```

```
session.execute(insertUser);
```



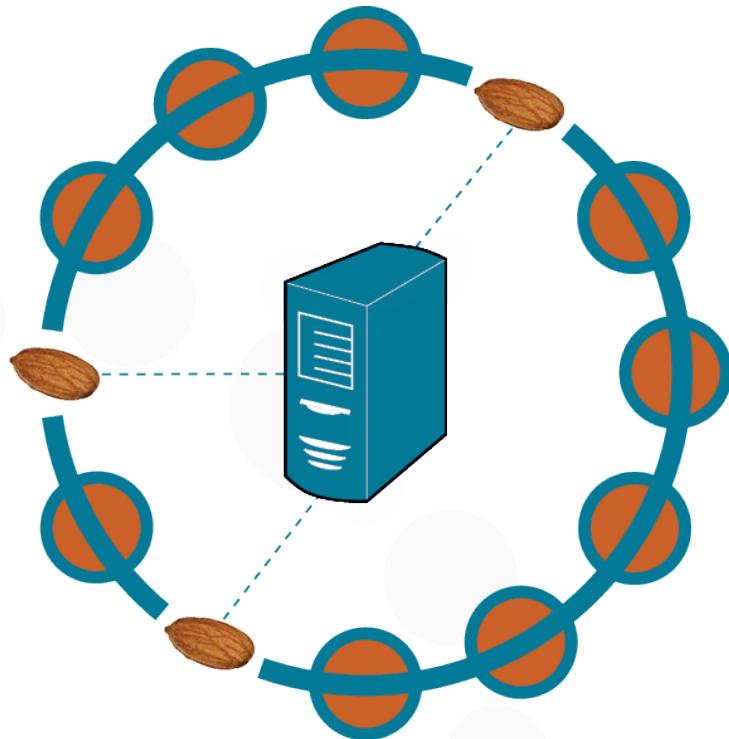
# Cluster Builder



# Builder

```
DseCluster meCluster =  
DseCluster  
    .builder()  
    .addContactPoint("100.100.100.100")  
    .build()
```

# Contact Points



Only one necessary

Unless that node is down 😬

More are good

Seed nodes are good; more  
“in the know”

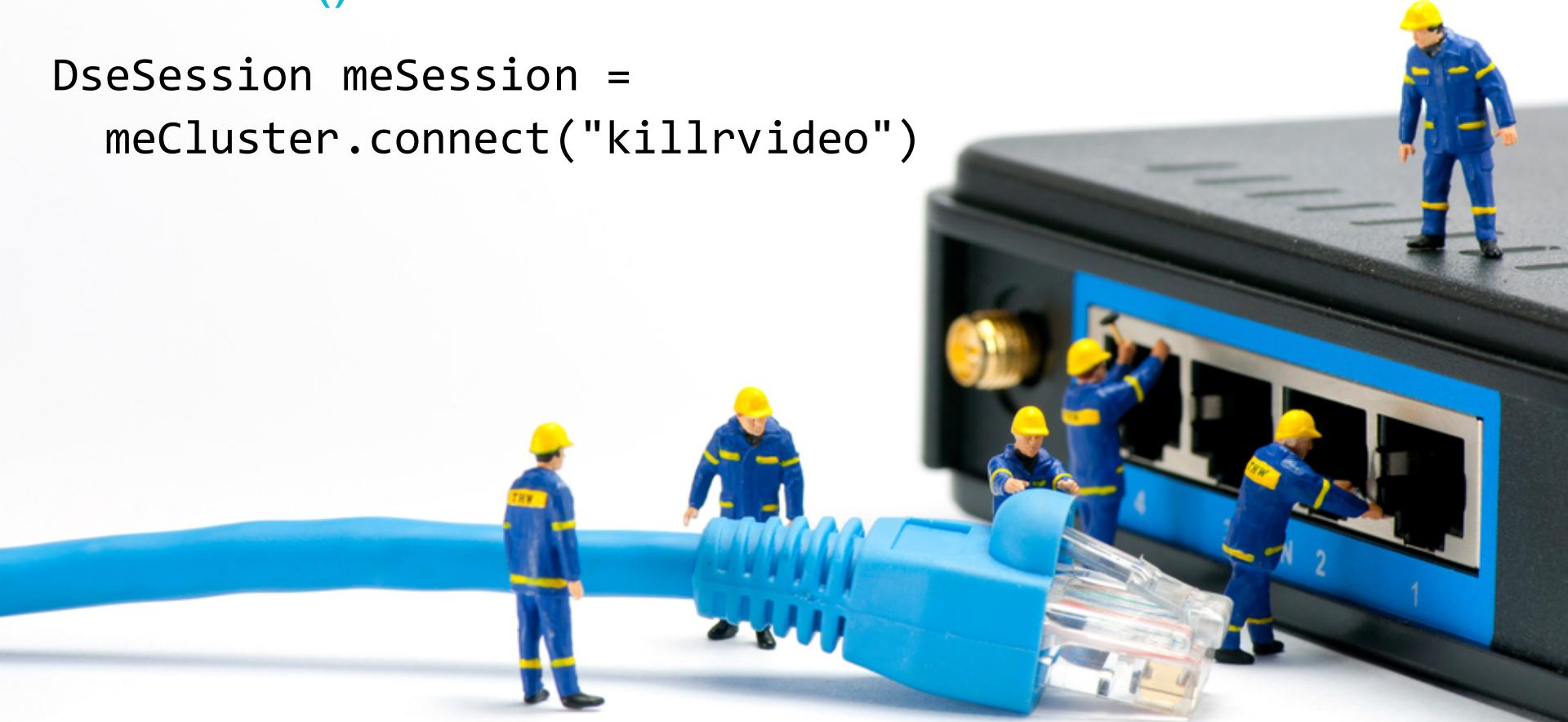
# DseSession

- The big dog in town
- Main entry point to the cluster
- Maintains metadata about the cluster
- Thread safe

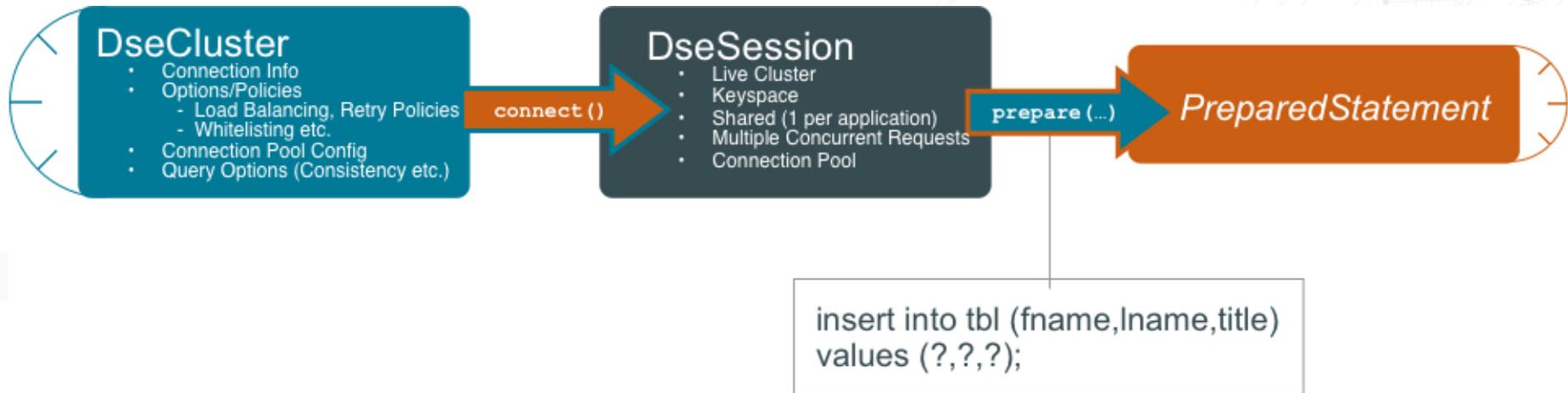


# connect()

```
DseSession meSession =  
    meCluster.connect("killrvideo")
```



# Major Players



# Ways to Talk to Your Cluster

- SimpleStatement
- PreparedStatement
- BoundStatement
- QueryBuilder
- Object Mapping
- Accessors



# Simple Command String + execute()



For when you need to do something quick and dirty

```
session.execute("SELECT * FROM users")
```

You know...sometimes you just got to get something done

# execute() → ResultSet

```
ResultSet rs = session.execute("SELECT * FROM t1");

Row firstRow = rs.one();

int col1 = firstRow.getInt("c1");

String col2 = firstRow.getString("c2");
```

# SimpleStatement

- Simple statements are interpreted at runtime and may include parameter placeholders
- You can set options on a `SimpleStatement` instance

```
Statement statement =  
    new SimpleStatement("select * from t1 where c1 = 5");  
  
Statement statement =  
    new SimpleStatement("select * from t1 where c1 = ?", 5);  
session.execute(statement);
```

# Prepared and Bound Statements

- *Compiled once on each node automatically as needed*
- Prepare each statement only once per application
- Use one of the many bind variations to create a *BoundStatement*

```
PreparedStatement ps = session.prepare("SELECT * from t1 where c1 = ?");  
BoundStatement bound = ps.bind(5);
```

# QueryBuilder

- Alternative to building CQL string queries manually
- Contains methods to build SELECT, UPDATE, INSERT and DELETE statements
- Generates a Statement as per the earlier techniques

<b>Method</b>	<b>Return Type</b>
<code>QueryBuilder.select()</code>	Selection
<code>QueryBuilder.insertInto()</code>	Insert
<code>QueryBuilder.update()</code>	Update
<code>QueryBuilder.delete()</code>	Selection

# QueryBuilder

QueryBuilder

```
.update("killrvideo", "video_ratings")
.with(QueryBuilder.incr("rating_counter"))
.and(QueryBuilder.incr("rating_total", QueryBuilder.bindMarker()))
.where(QueryBuilder.eq("videoid", QueryBuilder.bindMarker()))
```

# Object Mapping

Map Java entity beans to CQL tables

```
CREATE TABLE IF NOT EXISTS users (
    userid uuid,
    firstname text,
    lastname text,
    email text,
    created_date timestamp,
    PRIMARY KEY (userid));
```

```
public class User {  
    private UUID userid;  
    private String firstname;  
    private String lastname;  
    private String email;  
    private Date createdAt;  
  
    public User() {}  
    public User(UUID userid, String firstname,  
        String lastname, String email, Date createdAt) {  
        this.userid = userid;  
        this.firstname = firstname;  
        this.lastname = lastname;  
        this.email = email;  
        this.createdAt = createdAt;  
    }  
    public UUID getUserId() { return userid; }  
    public void setUserId(UUID userid) { this.userid = userid; }  
    // Etc.  
}
```



```
@Table(keyspace = "killrvideo", name = "users")
public class User {
    @PartitionKey
    private UUID userid;
    @Length(min = 1, message = "firstName must not be empty")
    @Column
    private String firstname;
    @Length(min = 1, message = "lastname must not be empty")
    @Column
    private String lastname;
    @Length(min = 1, message = "email must not be empty")
    @Column
    private String email;
    @NotNull
    @Column(name = "created_date")
    private Date createdAt;
}
```



# Mapper



```
MappingManager manager = new MappingManager(meSession);
Mapper<User> userMapper = manager.mapper(User.class);
Result<User> users = userMapper.map(meSession.execute(myQuery));
```

# Exercise!

Application Development  
Prepared Statements (Notebook)



# Accessors

```
@Accessor
public interface VideosAccessor {
    @Query("SELECT * FROM videos_by_genre" +
        "WHERE genre = :genre AND release_year >= :start" +
        " AND release_year <= :end ORDER BY release_year DESC")
    Iterable<Video> getVideosByGenreYearRange(
        @Param("genre") String genre,
        @Param("start") int yearRangeStart,
        @Param("end")   int yearRangeEnd);
    ...
}
```



```
VideosAccessor accessor =  
    manager.createAccessor(VideosAccessor.class)
```

```
Iterable<Video> videos =  
    accessor.getVideosByGenreYearRange("horror", 2013, 2016)
```

# Exercise!

## Application Development Accessor (Notebook)





**Best  
Practice  
Ahead**

# Instantiate Less

Single **Cluster** object per physical cluster

Single **Session** object per application

# Run Async

`mapper.map()` vs `mapper.mapAsync()`

`execute()` vs `executeAsync()`

`get()` vs `getAsync()`

`save()` vs `saveAsync()`

Returns a `ListenableFuture`

Maximizes Throughput

# Prepare Once – Bind Many

- Server parses/compiles query once
- Cached for lifetime of the application
- Reduces network traffic

# Avoid Deletes and Writing Nulls

```
INSERT INTO meTable (primary_key, clustering_key)  
VALUES ('pk1', 'ck1');
```

VS

```
INSERT INTO meTable (primary_key, clustering_key, regular_col)  
VALUES ('pk1', 'ck1', null);
```

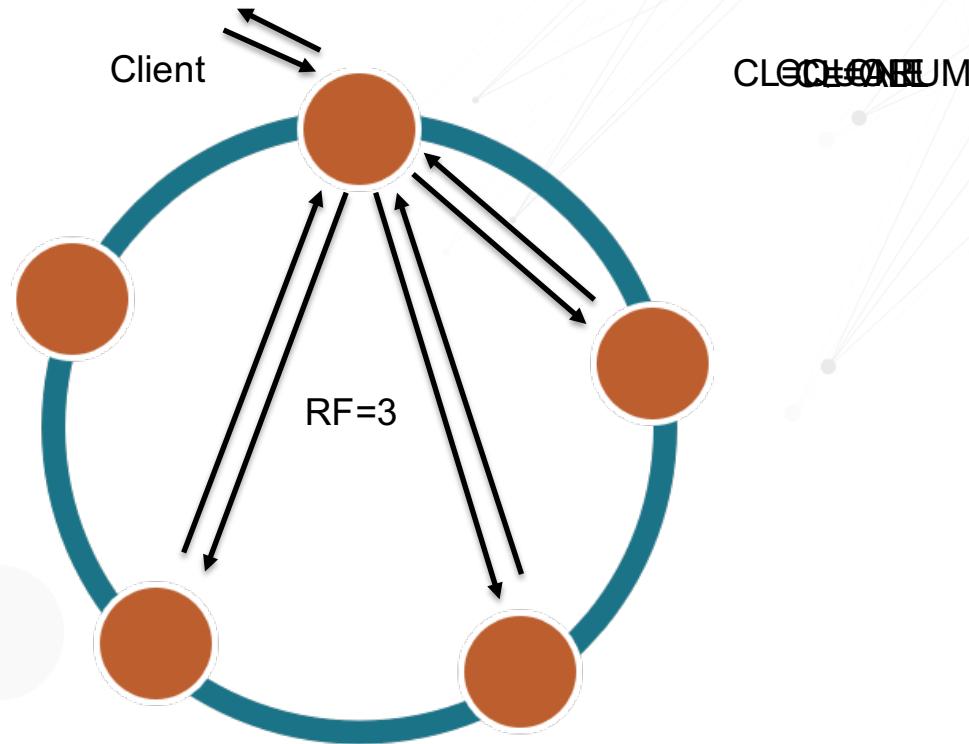
Second version writes a tombstone

# Be Datacenter Aware

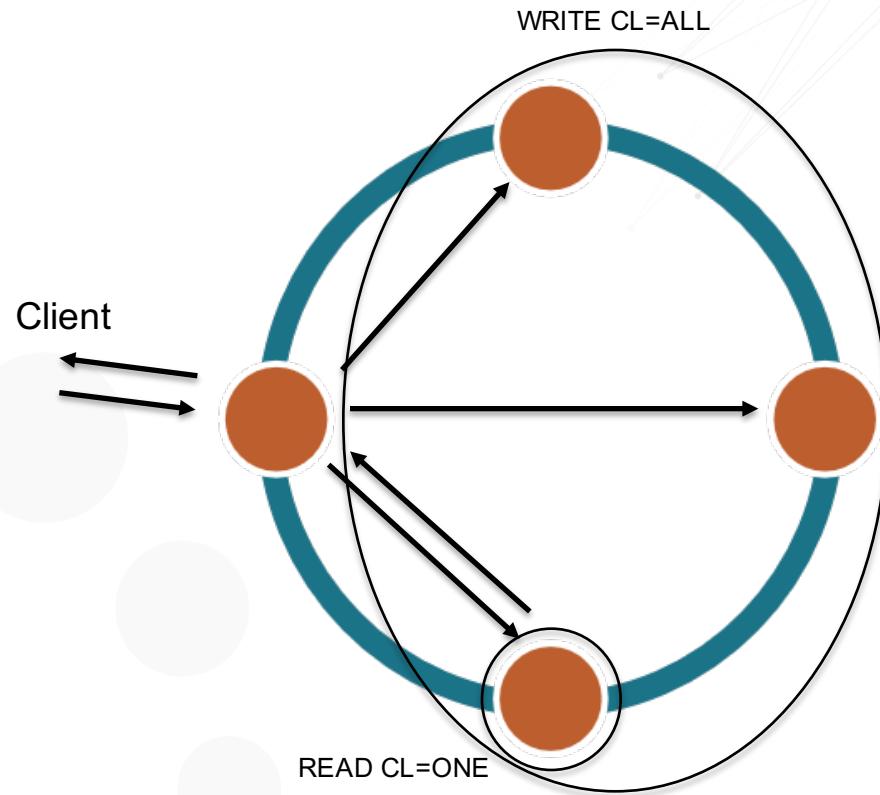
```
DCAwareRoundRobinPolicy  
    .builder()  
    .withLocalDc("myLocalDC")
```

- More on load balancing policies later

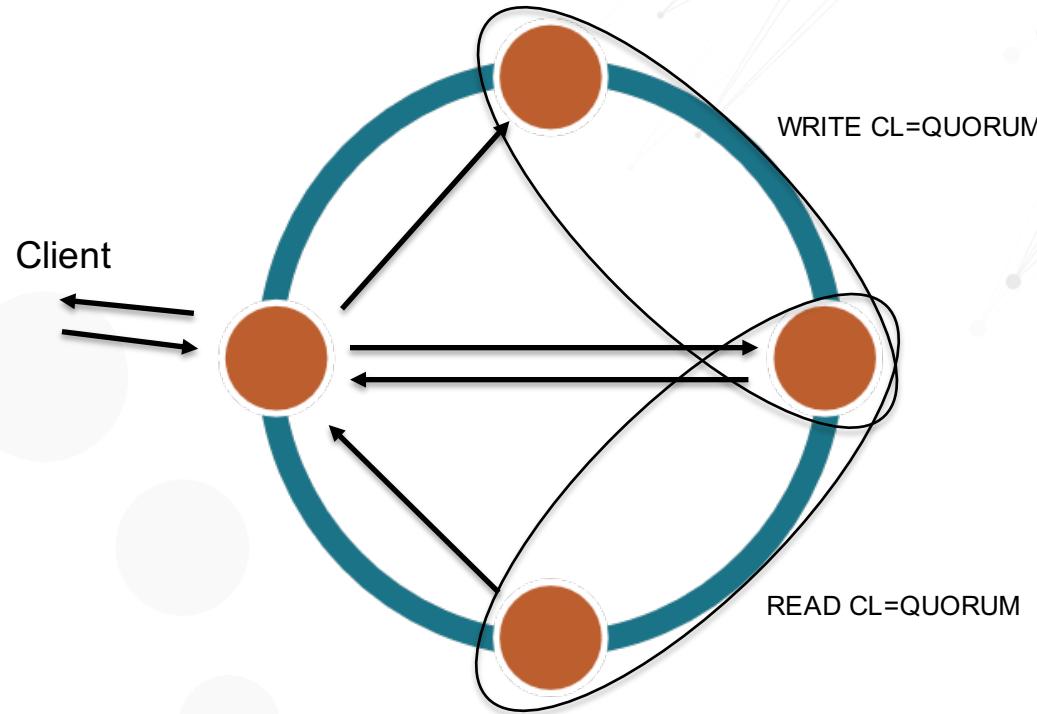
# Consistency Levels



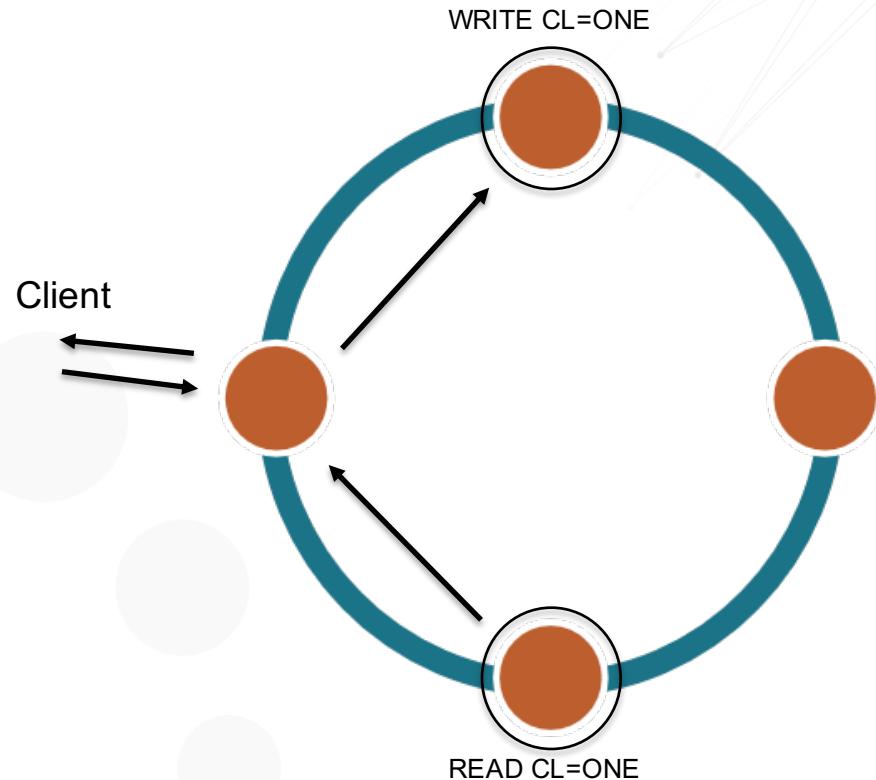
# Strong Consistency



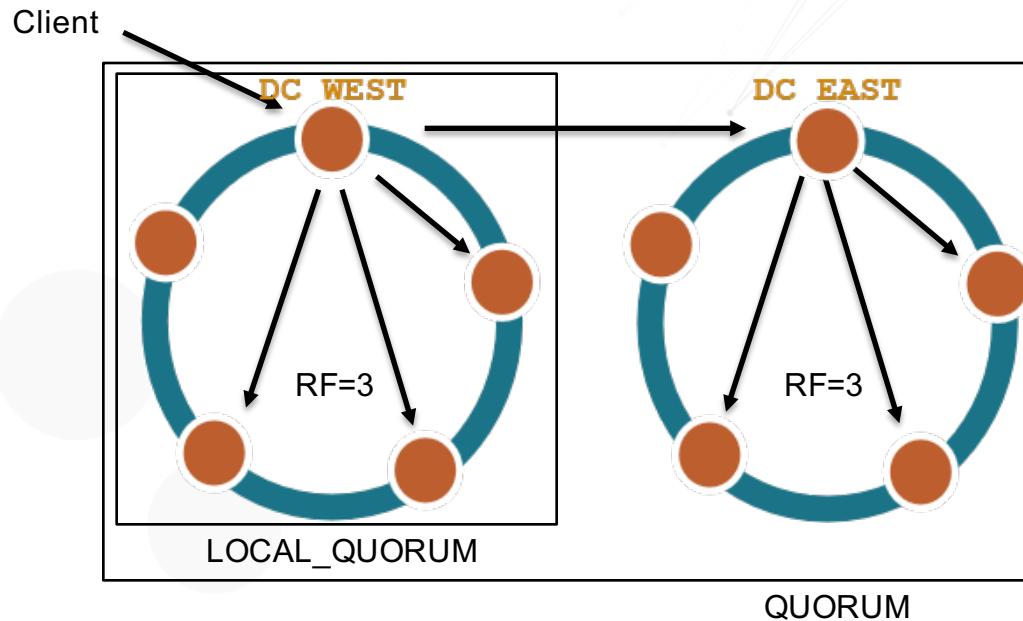
# Quorum



# CL=ONE



# Consistency Across Data Centers



# Time to Live (TTL)

Time-series data

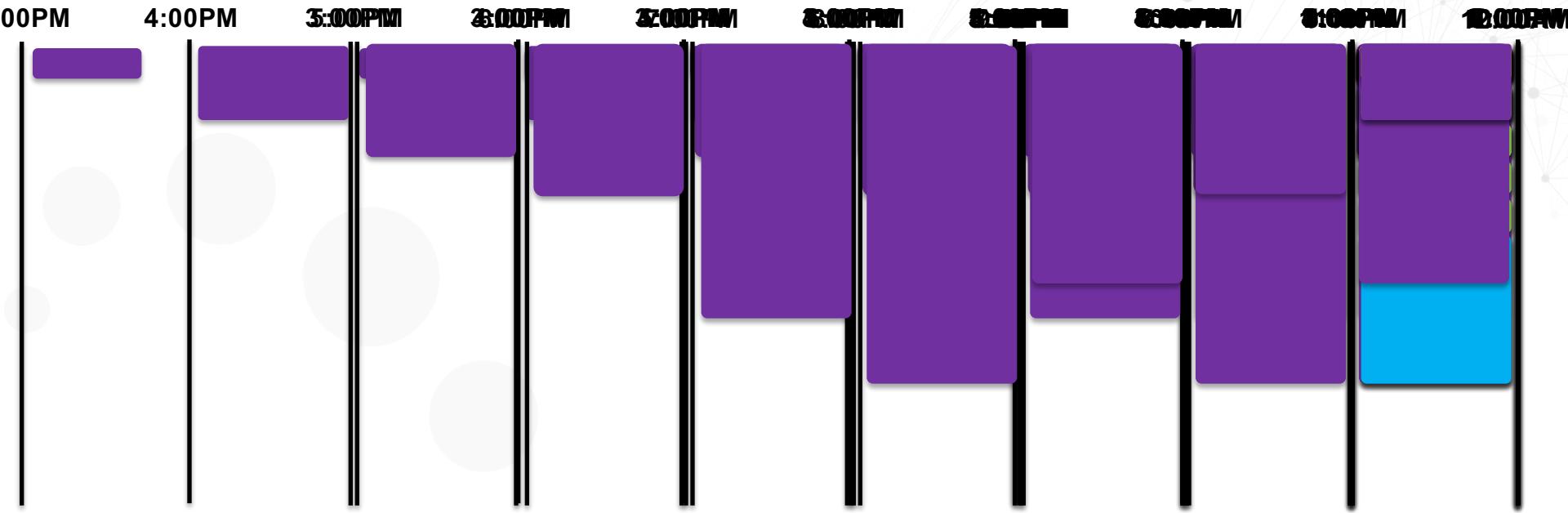
Delete entire partitions when possible

Better type of tombstones

TimeWindowCompactionStrategy  
VS  
DateTieredCompactionStrategy

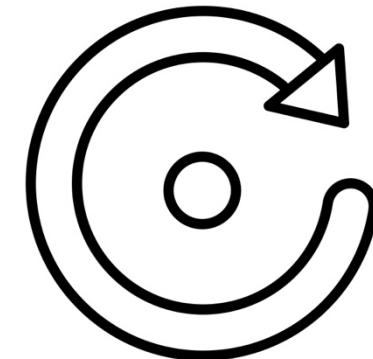


# Time Windowed Compaction



# Retry Policies

- Most are deprecated (long story; don't ask)
- DefaultRetryPolicy
  - Retries once onReadTimeout or onWriteTimeout
  - Enough replicas for your consistency level must be online
- FallthroughRetryPolicy
  - Doesn't retry
  - Sends exception to your client application



**retry**

# Reconnection Policies

Reconnects driver to a downed node

Two options:

- ConstantReconnectionPolicy
  - Check every N milliseconds
- ExponentialReconnectionPolicy (default)
  - Increases every interval
  - Caps out at a max



# Batches

- All commands will *eventually* succeed
- Atomicity
- *That's it*

```
INSERT INTO videos...
INSERT INTO videos_by_user...
INSERT INTO videos_by_genre...
INSERT INTO latest_videos...
```



# Batches and the Driver

```
BatchStatement insertVideoBatch = new BatchStatement();  
  
insertVideoBatch.add(insertToVideos);  
insertVideoBatch.add(insertToVideosByUser);  
insertVideoBatch.add(insertToVideosByGenre);  
insertVideoBatch.add(insertToLatestVideos);  
  
session.execute(insertVideoBatch);
```

# Batch Details

- One message to coordinator node
- Coordinator writes a batch log and replicates it to other nodes
- All affected nodes must ack the batch before coordinator removes batch log

# Batches != Data Loading

Batches about data integrity between tables  
All three inserts must succeed for batch to succeed

```
batchStatement.add(insertVideo);  
batchStatement.add(insertUserVideo);  
batchStatement.add(insertLatestVideo);
```



# Paging

- Pulling large result sets all at once may not make sense
- Retrieve sections or (pages)
- Retrieve more...if necessary
- Default of 5,000

# setFetchSize() → Page Size

- Global

```
Cluster cluster = Cluster.builder()
    .addContactPoint("127.0.0.1")
    .withQueryOptions(new QueryOptions().setFetchSize(2000))
    .build()

// Or at runtime:
cluster.getConfiguration().getQueryOptions().setFetchSize(2000)
```

- Single statement

```
Statement statement = new SimpleStatement("your query");
statement.setFetchSize(2000);
```



# Paging State

- Tracks location in a ResultSet
- “Continue where you left off...”
- Query + parameters must match

```
ResultSet resultSet = session.execute("your query");
// iterate the result set...
PagingState pagingState = resultSet.getExecutionInfo().getPagingState();
```

- Save that state

```
String meState = pagingState.toString()
```

- Later...

```
PagingState pagingState = PagingState.fromString(string);
Statement st = new SimpleStatement("your query");
st.setPagingState(pagingState);
ResultSet rs = session.execute(st);
```



# Load balancing

```
new TokenAwarePolicy(  
    new DCAwareRoundRobinPolicy("local-dc-name"))
```

