Cassandra Fundamentals

Module 4

Cassandra access from Java programs

LXFT
LISTED
NYSE

Module plan

- C* Java driver architecture
- Basic Java API

Prerequisites

- JDK (1.7)
- Eclipse (or other IDE like IntelliJIDEA)
- Maven
- set JAVA_HOME, ECLIPSE_HOME, M2_HOME, M2

C* Drivers

- Datastax supported: http://github.com/datastax

Driver	Version	2.1 Features	Out of the C* box
Java	2.1.4	Yes	Yes
Python	2.1.4	Yes	Yes
C/C++	1.0.0-rc1	No	No
C#	2.1.2	Yes	No
Ruby	2.0.0	Partial: C* 2.0 API	No
Node.js	1.0.3	No	No

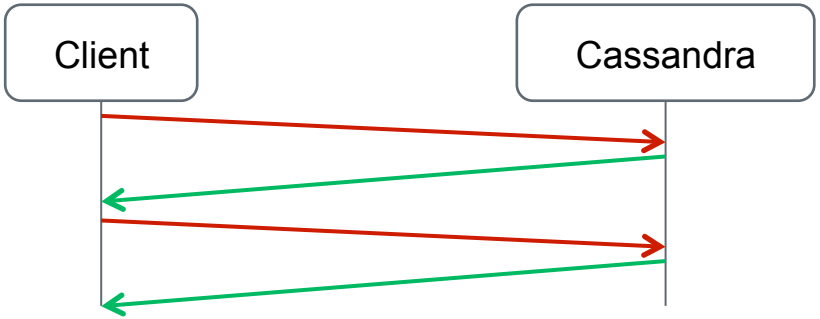
- ODBC, Clojure, Erlang, Go, Haskell, Perl, PHP, R, Rust, Scala

Java Driver

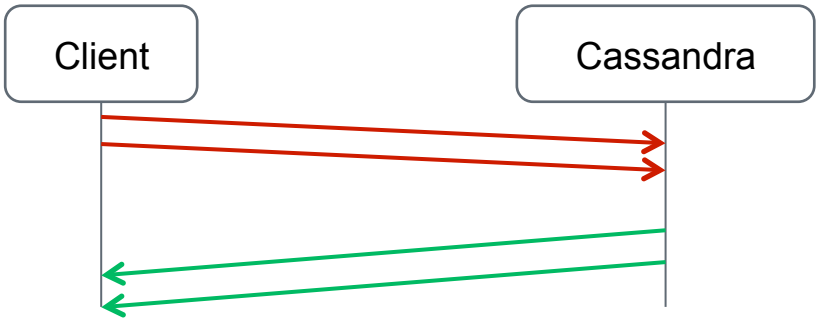
- Reference C* driver implementation
- Netty-based (asynchronous)
- Compression
- Security (SSL)
- Automatic fail-over
- Configurable policies
  - LoadBalancingPolicy, ReconnectionPolicy, RetryPolicy
- Query tracing support
- On Maven Central (cassandra-driver-core)
  - depends on Netty, Guava, Metrics

Request Pipelining

No pipelining



Pipelining



Configure and Connect

```
Cluster cluster = Cluster.builder()
      .addContactPoints("127.0.0.1", "192.168.1.100")
      .setConsistencyLevel(ConsistencyLevel.ONE)
      .withLoadBalancingPolicy(new DCAwareRoundRobinPolicy("DC1")
      .withRetryPolicy(DowngradingConsistencyRetryPolicy.INSTANCE)
      .withReconnectionPolicy(new ConstantReconnectionPolicy(100L))
      .build();

Session session = cluster.connect();

cluster.shutdown();
```

cassandra.yml

```
start_native_transport : true
# native_transport_port : 9042 (default)
rpc_address : IP address or hostname
```

Create Keyspace and Table

```
session.execute (“CREATE KEYSPACE myKeyspace WITH replication = ” +
“{‘class’:’SimpleStrategy’, ‘replication_factor’: 3};”);

session.execute (“CREATE TABLE myKeyspace.table1 ( ” +
            “id uuid PRIMARY KEY,” +
            “col1 integer,” +
            “col2 text,” +
            “col3 set<text> );” +
);
```

Write

```
Session session = cluster.connect(“myKeyspace”);

session.execute(
      “INSERT INTO table1 (id, col1, col2, col3) ” +
      “VALUES (12345678-1111-2222-3333-1234567890ab, ” +
      “123, ‘abc’, {‘first’, ‘last’});”
);

session.execute(
      “INSERT INTO table1 (col1, col2)
      VALUES (123, ‘abc’);”
);
```

Read

```
ResultSet results = session.execute(
      “SELECT * FROM table1 ” +
      “WHERE id=12345678-1111-2222-3333-1234567890ab or col1=123;”
);

for (Row row: results) {
      String userId = row.getString(“id”);
      Long col1 = row.getInt(“col1”);
      String col2 = row.getString(“col2”);
}
```

Bound Statement

```
PreparedStatement statement = session.prepare(
      “SELECT * FROM table WHERE id=? or col1=? or col3=?”
);

BoundStatement boundStatement = new BoundStatement(statement);

Set<String> stringSet = new HashSet<String>();
stringSet.add(“only”);
stringSet.add(“one more”);

ResultSet result = session.execute(boundStatement.bind(
      UUID.fromString(“12345678-1111-2222-3333-1234567890ab”),
      123,
      stringSet));
```

Paging Through a ResultSet (Cursor)

```
Statement statement = new SimpleStatement (
 “SELECT * FROM table WHERE id=1234”
);

statement.setFetchSize(10);

ResultSet result = session.execute(statement);

Iterator<Row> iter = result.iterator();

while (!result.isFullyFetched()) {

       result.fetchMoreResults();

       Row row = iter.next();

       System.out.println(row.getString(0));

}
```

Lightweight Transactions: Success Checking

```
ResultSet rs = session.execute(
      “INSERT INTO users(username, email, pass) ” +
      “VALUES (‘abreiman’,’abreiman@luxoft.com’,12341234ccdd) “ +
      “IF NOT EXISTS”);

Row row = rs.one();

row.getBool(“applied”);
```

Batch Statement

```
BatchStatement bs = new BatchStatement();

bs.add(new SimpleStatement("INSERT … VALUES(?)", v1));

bs.add(new SimpleStatement("INSERT … VALUES(?)", v2));

PreparedStatement ps = session.prepare("INSERT … VALUES(?)"));

bs.add(ps.bind(v1));

bs.add(ps.bind(v2));

Session.execute(bs);
```

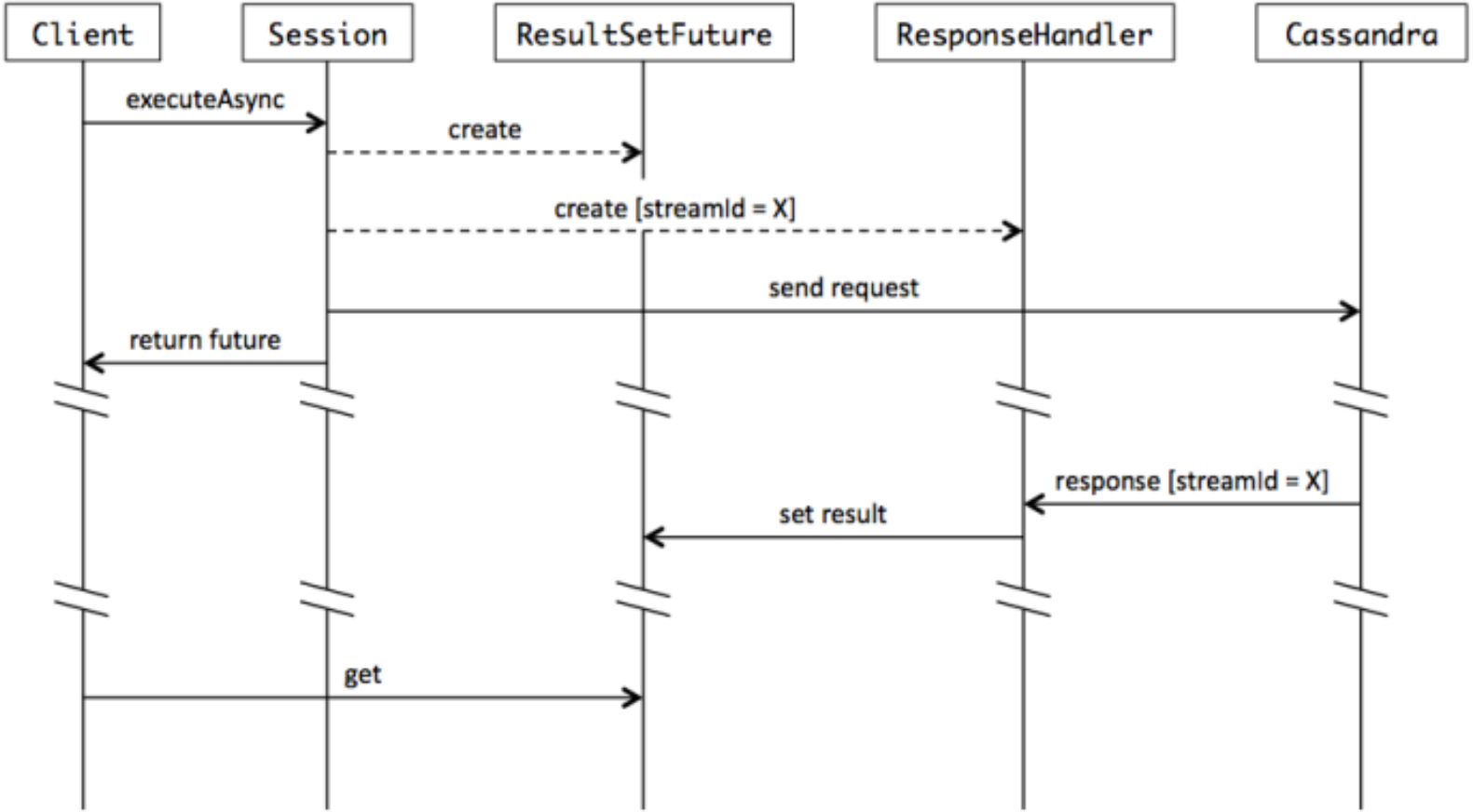
Dynamic Queries with QueryBuilder

```
Statement stmt = QueryBuilder.select()
      .all()
      .from(“keyspace”, “table”)
      .where(eq(“id”, UUID.fromString(“…”)));

Statement stmt = QueryBuilder.insertInto(“keyspace”,”table”)
      .value(“id”, UUID.fromString(“…”)
      .value(“name”, “abreiman”);

Statement stmt = QueryBuilder.update(“keyspace”,”table”)
      .with(set(“id”, UUID.fromString(“…”))
      .where(eq(“id”, UUID.fromString(“…”)));
```

Asynchronous I/O

```
PreparedStatement ps = session.prepare(“SELECT … ?”);
List<ResultSetFuture> futures = new ArrayList<ResulsSetFuture>();
for (int i=0; i<100; i++) {
  ResultSetFuture rsFuture = session.executeAsync(ps.bind(i));
  futures.add(rsFuture);
}
for (ResultSetFuture f: futures) {
  for (Row r: f.getUninterruptibly()) {
      row.getString(0);
  };
}
```

Asynchronous I/O



Asynchronous I/O

- Checking is result ready

```
while (!future.isDone()) {…}
```

- Blocking wait

```
ResultSet rs = future.get();
```

- Blocking wait with timeout

```
ResultSet rs = future.get(10, TimeUnit.SECONDS);

      => catch(TimeoutException e) { future.cancel(true); }
```

ListenableFuture: Guava Callbacks

```
import com.google.common.util.concurrent.Futures;
import com.google.common.util.concurrent.FutureCallback;
import com.google.common.util.concurrent.MoreExecutors;

ResultSetFuture rsFuture = session.executeAsync("SELECT …");

Futures.addCallback(rsFuture, new FutureCallback<ResultSet>() {
  @Override public void OnSuccess(ResultSet res) { res.get…() }
  @Override public void OnFailure(Throwable t) { … }
},
MoreExecutors.sameThreadExecutor()
);
```

Cluster Metadata

```
Cluster cluster = …

Metadata metadata = cluster.getMetadata();

String clusterName = metadata.getClusterName();

for (Host host: metadata.getAllHosts() ) {
      host.getDatacenter()
      host.getAddress()
      host.getRack
}
```

Cluster configuration

- Load Balancing Policies: which node to execute query on?
  - RoundRobinPolicy – round-robin all nodes (as if all nodes are local), first available wins
  - DCAwareRoundRobinPolicy – round-robin local DC nodes, then remote DC nodes
  - TokenAwarePolicy – wrap another policy with token awareness
- Reconnect Policies: how often to reconnect to a dead node?
  - ConstantReconnectionPolicy
  - ExponentialReconnectionPolicy (default)

Cluster configuration

- Retry Policies: what to do in case of node unavailability or request timeout?
  - DefaultRetryPolicy – keeps consistency level while retrying
  - DowngradingConsistencyRetryPolicy – lowers consistency if:
    - read timeout: 2+ nodes answered but that's not enough
    - write timeout: 1+ ack with UNLOGGED BATCH
    - unavailable exception: 1+ alive node
  - FallthroughRetryPolicy – rethrow, never retry
  - LoggingRetryPolicy – logging wrapper

Exceptions

- NoHostAvailableException
- QueryExecutionException
- QueryValidationException
- AlreadyExistsException
- AuthenticationException
- …

Tracing Query Execution

```
Statement statement = new SimpleStatement("SELECT…")
                                          .enableTracing();
ResultSet result = session.execute(statement);
ExecutionInfo exInfo = result.getExecutionInfo();
exInfo.getQueriedHost()
for (Host host: exInfo.getTriedHosts()) { host.toString() }
for (QueryTrace.Event ev: exInfo.getQueryTrace().getEvents()) {
        ev.getDescription(), ev.getSource(), ev.getTimestamp(), ev.getSourceElapsedMicros()
}
```

Best Practices

- Use QUORUM or LOCAL_QUORUM
- Use prepared statements
- Fully qualify tables
- Use DC and Token Aware policies
- Use Token Aware policy
- Use batches of up to 100 inserts
- CAS contention is at the partition

Thank you!

Questions?