A Guide to Cassandra with Java

1. Overview

This tutorial is an introductory guide to the Apache Cassandra database using Java.

You will find key concepts explained, along with a working example that covers the basic steps to connect to and start working with this NoSQL database from Java.

2. Cassandra

Cassandra is a scalable NoSQL database that provides continuous availability with no single point of failure and gives the ability to handle large amounts of data with exceptional performance.

This database uses a ring design instead of using a master-slave architecture. In the ring design, there is no master node – all participating nodes are identical and communicate with each other as peers.

This makes Cassandra a horizontally scalable system by allowing for the incremental addition of nodes without needing reconfiguration.

2.1. Key Concepts

Let's start with a short survey of some of the key concepts of Cassandra:

- **Cluster**– a collection of nodes or Data Centers arranged in a ring architecture. A name must be assigned to every cluster, which will subsequently be used by the participating nodes
- **Keyspace**– If you are coming from a relational database, then the schema is the respective keyspace in Cassandra. The keyspace is the outermost container for data in Cassandra. The main attributes to set per keyspace are the *Replication Factor*, the *Replica Placement Strategy* and the *Column Families*
- **Column Family**– Column Families in Cassandra are like tables in Relational Databases. Each Column Family contains a collection of rows which are represented by a *Map<RowKey, SortedMap<ColumnKey, ColumnValue>>*. The key gives the ability to access related data together
- **Column** – A column in Cassandra is a data structure which contains a column name, a value and a timestamp. The columns and the number of columns in each row may vary in contrast with a relational database where data are well structured

3. Using the Java Client

3.1. Maven Dependency

We need to define the following Cassandra dependency in the *pom.xml*, the latest version of which can be found here:

```
<dependency>
    <groupId>com.datastax.cassandra</groupId>
    <artifactId>cassandra-driver-core</artifactId>
    <version>3.1.0</version>
</dependency>
```

In order to test the code with an embedded database server we should also add the *cassandra-unit* dependency, the latest version of which can be found here:

```
<dependency>
    <groupId>org.cassandraunit</groupId>
    <artifactId>cassandra-unit</artifactId>
```

```
    <version>3.0.0.1</version>
</dependency>
```

3.2. Connecting to Cassandra

In order to connect to Cassandra from Java, we need to build a *Cluster* object.

An address of a node needs to be provided as a contact point. If we don't provide a port number, the default port (9042) will be used.

These settings allow the driver to discover the current topology of a cluster.

```
public class CassandraConnector {

    private Cluster cluster;

    private Session session;

    public void connect(String node, Integer port) {
        Builder b = Cluster.builder().addContactPoint(node);
        if (port != null) {
            b.withPort(port);
        }
        cluster = b.build();

        session = cluster.connect();
    }

    public Session getSession() {
        return this.session;
    }

    public void close() {
        session.close();
        cluster.close();
    }
}
```

3.3. Creating the Keyspace

Let's create our "*library*" keyspace:

```
public void createKeyspace(
  String keyspaceName, String replicationStrategy, int replicationFactor) {
  StringBuilder sb =
    new StringBuilder("CREATE KEYSPACE IF NOT EXISTS ")
      .append(keyspaceName).append(" WITH replication = {")
      .append("'class':'").append(replicationStrategy)
      .append("','replication_factor':").append(replicationFactor)
      .append("};");

    String query = sb.toString();
    session.execute(query);
}
```

Except from the *keyspaceName* we need to define two more parameters, the *replicationFactor*and the *replicationStrategy*. These parameters determine the number of replicas and how the replicas will be distributed across the ring, respectively.

With replication Cassandra ensures reliability and fault tolerance by storing copies of data in multiple nodes.

At this point we may test that our keyspace has successfully been created:

```
private KeyspaceRepository schemaRepository;
private Session session;

@Before
public void connect() {
    CassandraConnector client = new CassandraConnector();
    client.connect("127.0.0.1", 9142);
    this.session = client.getSession();
    schemaRepository = new KeyspaceRepository(session);
}
@Test
public void whenCreatingAKeyspace_thenCreated() {
    String keyspaceName = "library";
    schemaRepository.createKeyspace(keyspaceName, "SimpleStrategy", 1);

    ResultSet result =
      session.execute("SELECT * FROM system_schema.keyspaces;");

    List<String> matchedKeyspaces = result.all()
      .stream()
      .filter(r -> r.getString(0).equals(keyspaceName.toLowerCase()))
      .map(r -> r.getString(0))
      .collect(Collectors.toList());

    assertEquals(matchedKeyspaces.size(), 1);
    assertTrue(matchedKeyspaces.get(0).equals(keyspaceName.toLowerCase()));
}
```

3.4. Creating a Column Family

Now, we can add the first Column Family “books” to the existing keyspace:

private static final String TABLE_NAME = "books"; private Session session;

```
public void createTable() {
    StringBuilder sb = new StringBuilder("CREATE TABLE IF NOT EXISTS ")
      .append(TABLE_NAME).append("(")
      .append("id uuid PRIMARY KEY, ")
      .append("title text,")
      .append("subject text);");

    String query = sb.toString();
    session.execute(query);
}
```

The code to test that the Column Family has been created, is provided below:

```
private BookRepository bookRepository;
private Session session;

@Before
public void connect() {
    CassandraConnector client = new CassandraConnector();
    client.connect("127.0.0.1", 9142);
    this.session = client.getSession();
    bookRepository = new BookRepository(session);
}
@Test
public void whenCreatingATable_thenCreatedCorrectly() {
    bookRepository.createTable();

    ResultSet result = session.execute(
      "SELECT * FROM " + KEYSPACE_NAME + ".books;");

    List<String> columnNames =
      result.getColumnDefinitions().asList().stream()
      .map(cl -> cl.getName())
      .collect(Collectors.toList());

    assertEquals(columnNames.size(), 3);
    assertTrue(columnNames.contains("id"));
    assertTrue(columnNames.contains("title"));
    assertTrue(columnNames.contains("subject"));
}
```

3.5. Altering the Column Family

A book has also a publisher, but no such column can be found in the created table. We can use the following code to alter the table and add a new column:

```
public void alterTablebooks(String columnName, String columnType) {
    StringBuilder sb = new StringBuilder("ALTER TABLE ")
      .append(TABLE_NAME).append(" ADD ")
      .append(columnName).append(" ")
      .append(columnType).append(";");

    String query = sb.toString();
    session.execute(query);
}
```

Let's make sure that the new column *publisher* has been added:

```
@Test
public void whenAlteringTable_thenAddedColumnExists() {
    bookRepository.createTable();

    bookRepository.alterTablebooks("publisher", "text");

    ResultSet result = session.execute(
      "SELECT * FROM " + KEYSPACE_NAME + "." + "books" + ";");
```

```
    boolean columnExists = result.getColumnDefinitions().asList().stream()
      .anyMatch(cl -> cl.getName().equals("publisher"));

    assertTrue(columnExists);
}
```

3.6. Inserting Data in the Column Family

Now that the *books* table has been created, we are ready to start adding data to the table:

```
public void insertbookByTitle(Book book) {
    StringBuilder sb = new StringBuilder("INSERT INTO ")
      .append(TABLE_NAME_BY_TITLE).append("(id, title) ")
      .append("VALUES (").append(book.getId())
      .append(", '").append(book.getTitle()).append("');");

    String query = sb.toString();
    session.execute(query);
}
```

A new row has been added in the 'books' table, so we can test if the row exists:

```
@Test
public void whenAddingANewBook_thenBookExists() {
    bookRepository.createTableBooksByTitle();

    String title = "Effective Java";
    Book book = new Book(UUIDs.timeBased(), title, "Programming");
    bookRepository.insertbookByTitle(book);

    Book savedBook = bookRepository.selectByTitle(title);
    assertEquals(book.getTitle(), savedBook.getTitle());
}
```

In the test code above we have used a different method to create a table named *booksByTitle:*

```
public void createTableBooksByTitle() {
    StringBuilder sb = new StringBuilder("CREATE TABLE IF NOT EXISTS ")
      .append("booksByTitle").append("(")
      .append("id uuid, ")
      .append("title text,")
      .append("PRIMARY KEY (title, id));");

    String query = sb.toString();
    session.execute(query);
}
```

In Cassandra one of the best practices is to use one-table-per-query pattern. This means, for a different query a different table is needed.

In our example, we have chosen to select a book by its title. In order to satisfy the *selectByTitle* query, we have created a table with a compound *PRIMARY KEY* using the columns, *title* and *id*. *The column *title* is the partitioning key while the *id* column is the clustering key.

This way, many of the tables in your data model contain duplicate data. This is not a downside of this database. On the contrary, this practice optimizes the performance of the reads.

Let's see the data that are currently saved in our table:

```
public List<Book> selectAll() {
    StringBuilder sb =
      new StringBuilder("SELECT * FROM ").append(TABLE_NAME);

    String query = sb.toString();
    ResultSet rs = session.execute(query);

    List<Book> books = new ArrayList<Book>();

    rs.forEach(r -> {
        books.add(new Book(
          r.getUUID("id"),
          r.getString("title"),
          r.getString("subject")));
    });
    return books;
}
```

A test for query returning expected results:

```
@Test
public void whenSelectingAll_thenReturnAllRecords() {
    bookRepository.createTable();

    Book book = new Book(
      UUIDs.timeBased(), "Effective Java", "Programming");
    bookRepository.insertbook(book);

    book = new Book(
      UUIDs.timeBased(), "Clean Code", "Programming");
    bookRepository.insertbook(book);

    List<Book> books = bookRepository.selectAll();

    assertEquals(2, books.size());
    assertTrue(books.stream().anyMatch(b -> b.getTitle()
      .equals("Effective Java")));
    assertTrue(books.stream().anyMatch(b -> b.getTitle()
      .equals("Clean Code")));
}
```

Everything is fine till now, but one thing has to be realized. We started working with table *books,* but in the meantime, in order to satisfy the *select* query by *title* column, we had to create another table named *booksByTitle*.

The two tables are identical containing duplicated columns, but we have only inserted data in the *booksByTitle* table. As a consequence, data in two tables is currently inconsistent.

We can solve this using a *batch* query, which comprises two insert statements, one for each table. A *batch* query executes multiple DML statements as a single operation.

An example of such query is provided:

```
    public void insertBookBatch(Book book) {
    StringBuilder sb = new StringBuilder("BEGIN BATCH ")
      .append("INSERT INTO ").append(TABLE_NAME)
      .append("(id, title, subject) ")
      .append("VALUES (").append(book.getId()).append(", '")
      .append(book.getTitle()).append("', '")
      .append(book.getSubject()).append("');")
      .append("INSERT INTO ")
      .append(TABLE_NAME_BY_TITLE).append("(id, title) ")
      .append("VALUES (").append(book.getId()).append(", '")
      .append(book.getTitle()).append("');")
      .append("APPLY BATCH;");

    String query = sb.toString();
    session.execute(query);
}
```

Again we test the batch query results like so:

```
@Test
public void whenAddingANewBookBatch_ThenBookAddedInAllTables() {
    bookRepository.createTable();

    bookRepository.createTableBooksByTitle();

    String title = "Effective Java";
    Book book = new Book(UUIDs.timeBased(), title, "Programming");
    bookRepository.insertBookBatch(book);

    List<Book> books = bookRepository.selectAll();

    assertEquals(1, books.size());
    assertTrue(
      books.stream().anyMatch(
        b -> b.getTitle().equals("Effective Java")));

    List<Book> booksByTitle = bookRepository.selectAllBookByTitle();

    assertEquals(1, booksByTitle.size());
    assertTrue(
      booksByTitle.stream().anyMatch(
        b -> b.getTitle().equals("Effective Java")));
}
```

Note: As of version 3.0, a new feature called “Materialized Views” is available , which we may use instead of *batch* queries. A well-documented example for “Materialized Views” is available here.

3.7. Deleting the Column Family

The code below shows how to delete a table:

```
public void deleteTable() {
    StringBuilder sb =
      new StringBuilder("DROP TABLE IF EXISTS ").append(TABLE_NAME);

    String query = sb.toString();
    session.execute(query);
}
```

Selecting a table that does not exist in the keyspace results in an *InvalidQueryException: unconfigured table books*:

```
@Test(expected = InvalidQueryException.class)
public void whenDeletingATable_thenUnconfiguredTable() {
    bookRepository.createTable();
    bookRepository.deleteTable("books");

    session.execute("SELECT * FROM " + KEYSPACE_NAME + ".books;");
}
```

3.8. Deleting the Keyspace

Finally, let's delete the keyspace:

```
public void deleteKeyspace(String keyspaceName) {
    StringBuilder sb =
      new StringBuilder("DROP KEYSPACE ").append(keyspaceName);

    String query = sb.toString();
    session.execute(query);
}
```

And test that the keyspace has been deleted:

```
@Test
public void whenDeletingAKeyspace_thenDoesNotExist() {
    String keyspaceName = "library";
    schemaRepository.deleteKeyspace(keyspaceName);

    ResultSet result =
      session.execute("SELECT * FROM system_schema.keyspaces;");
    boolean isKeyspaceCreated = result.all().stream()
      .anyMatch(r -> r.getString(0).equals(keyspaceName.toLowerCase()));

    assertFalse(isKeyspaceCreated);
}
```

4. Conclusion

This tutorial covered the basic steps of connecting to and using the Cassandra database with Java. Some of the key concepts of this database have also been discussed in order to help you kick start.

The full implementation of this tutorial can be found in the Github project.