Develop for Lx-DB

Table of Contents

When developing for Lx-DB, you need to be aware of the basic concepts, and of particular differences in how Lx-DB works and how it behaves.

- LeanXcale Query Engine was forked from Apache Derby, so the Lx-DB SQL dialect is very similar to Apache Derby.

There are some differences, though. Please see LeanXcale Reference for details.

- Some concepts are used differently in Lx-DB compared to other databases, and intentionally so.
  - Asynchronous conflict management - Applications must handle exceptions because of aborted transactions Transactions do not wait until there is no conflict. Lx-DB will raise an exception they have to handle

For details about the concepts, see LeanXcale concepts for details.

For details about the architecture, see LeanXcale architecture for details.

- Lx-DB uses some specific connection parameters to build the connection URL. These should be taken into account:
  - Connection Mode - type of connection to use
  - Isolation Level - which sort of isolation to use

The connection string must contain the 'direct' parameter.

1. Schemas and metadata

Creating schemas and working with metadata is done via the is done through sql commands:

For details see, Derby SQL reference

2. APIs

To access data and perform transactions, you use the API interface.

LeanXcale is accessible through 2 different APIs. Use the one that fits your needs best:

- JDBC API https://docs.oracle.com/cd/B19306_01/java.102/b14355/toc.htm

- KiVi Direct API

You can use any of the APIs and you will have a consistent view of your data between them.

3. KiVi Direct API

Lx-DB provides a direct key/value API. The functions in this API - while keeping ACID properties - don’t use SQL and thus, SQL parsing, compilation, planning is not needed which results in a much faster interface.

KiVi Direct is efficient when you are doing simple operations. A typical use case, is data ingestion. Doing INSERT DML operations over SQL to put a record in the database is less efficient than using KiVi direct API to put your data directly into the database.

On the other hand, this API doesn’t provide the full capabilities and power of an SQL interface.

This is a high performance API, but when using this library, direct connection to most of the components is needed so applications using this API must be co-located in the security LAN of the database.

The KiVi API provides the following mechanisms:

- Connection management
- Transaction management
- Tuple management (access to fields and data in the tuples)
- Adds: Insert, Upsert or Updates of tuples
- Gets of tuples from the key
- Scans of ranges of tuples
- Predicates for scans
- Scans on secondary indexes also with Predicates
- Project (limit the fields you retrieve)
- Aggregations

To use this direct API you need to include the library `kivi-javaapi.jar` in your project.

3.1. Code examples

Connection Example:

```
import static Expressions.*;
import static Aggregations.*;
import static Constants.*;

ConnectionSettings settings = new ConnectionSettings()
.credentials(user, pass, "tpch")
.address(address,por)

Connection connection = ConnectionFactory.connect
("kivi:zk//zkserver:2181", settings);
Database database = connection.database();
```

Getting a Data Row directly by Key:

```
Tuplekey key = table.createTupleKey();
key.putLong("id", 0);

Tuple tuple = table.get(key);
```

Simple SCAN with basic options:

```
Table people = database.getTable("person");
TupleKey min = people.createTupleKey();
min.putLong("id", 20);

TupleKey max = people.createTupleKey();
max.putLong("id", 30);

people.find()
        .min(min)
        .max(max)
        .foreach(tuple->processTuple(tuple));

// Max 20 results
people.find()
        .first(20)
        .foreach(tuple->processTuple(tuple));
```

Scan a Table using a Secondary Index:

```
Table table people = database.getTable("person");
TupleKey minKey = new TupleKey(new Field("dni", string(10)));
minKey.putString("111111111Q");

people.find()
        .index("dniIdx")
        .indexMin(minKey)
        .foreach(tuple->processTuple(tuple));
```

Inserting one Record(Tuple):

```
Table people = database.getTable("person");
Tuple person = people.createTuple();

//Fill in tuple fields
person.putLong("id", 1).
      .putString("name", "John")
      .putString("lastName", "Doe")
      .putString("phone", "555333695")
      .putString("email", "johndoe@nowhere.no")
      .putDate("birthday",
          new SimpleDateFormat("yyyy/MM/dd").parse("1970/01/01"))
      .putInt("numChildren", 4);

//Insert tuple
people.insert(person);

//Tuples are sent to the datastore when COMMIT is done
connection.commit();
```

Insert using a Sequence:

```
Table people = database.getTable("person");
Tuple person = people.createTuple();

long personId = database.getSequence("personId").nextVal();

person.putLong("id", personId)
      .putString("name", "John")
      .putString("lastName", "Doe")
      .putString("phone", "555333695")
      .putString("email", "johndoe@nowhere.no")
      .putDate("birthday",
          new SimpleDateFormat("yyyy/MM/dd").parse("1970/01/01"))
      .putInt("numChildren", 4);

people.insert(person);
```

Insert a Record containing a BLOB read from a file:

```
Table people = database.getTable("person");
Tuple person = people.createTuple();

person.putString("name", "John")
      .putString("lastName", "Doe")
      .putString("phone", "555333695");

OutputStream os = people.createBlob(person, "photo");
Files.copy(Paths.get("/path/to/john.jpeg"), os);

people.insert(person);
```

Updating a Record:

```
// Tuple person to be updated has to be previosly retrieved through
// a get()
Tuplekey key = table.createTupleKey();
key.putLong("id", 0);

Tuple person = table.get(key);
person.putString("phone", "anotherPhone")
      .putString("email", "johndoe@somewhere.some")

people.update(person);
```

Deleting a Record by Key:

```
Table people = database.getTable("person");
TupleKey johnKey = people.createTupleKey();
johnKey.putLong("id", key);

people.delete(johnKey);
```

Advanced Finding and Filtering:

```
import static Filters.*;
import static Constants.*;
import static Expressions.*;

Table people = database.getTable("person");

// Basic comparisons
people.find()
      .filter(gt("numChildren", 4).and(eq("name", string("John"))))
      .foreach(tuple->processTuple(tuple));

// Between
SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd");
Date minDate = sdf.parse("1900/01/01");
Date maxDate = sdf.parse("2000/01/01");

people.find()
      .filter(between("birthday", date(minDate), date(maxDate))
      .foreach(tuple->processTuple(tuple));

// Using a Expression with operators
people.find()
      .filter(gt("numChildren",sub(field("numRooms"),int(1))))
      .foreach(tuple->processTuple(tuple));
```

Project:

```
import static Projections.*;
import static Expressions.*;
import static Arrays.*;

Table people = database.getTable("person");

// Basic inclusion
people.find()
      .project(include(asList("name", "lastName", "birthdate")))
      .foreach(tuple->{
            String name = tuple.getString("name");
            String lastname = tuple.getString("lastName");
            Date date = tuple.getDate("birthdate");
      });

// SELECT name, Lastname, weight/height^2 AS imc FROM person
// Alias and expression usage
people.find()
      .project(compose(Arrays.asList(
        alias("name"),
        alias("lastName"),
        alias("imc",div(field("weight"),pow(field("height"),2)))
        )))
      .foreach(tuple->{
        String name = tuple.getString("name");
        String lastName = tuple.getString("lastName");
        float imc = getFloat("imc");
});
```

Aggregations:

```
import static Aggregations.*;
import static Expressions.*;
import static Collections.*;
import static Arrays.*;

// Simple aggregation
int numPeople =
    people.find()
      .aggregate(emptyList(), count("numPeople"))
      .iterator.next().getLong("numPeople");

// Group By aggregation
people.find()
      .aggregate(.asList("name"),.asList(
        count("numPeople"),
        avg("averageHeight", field("height"))
        avg("averageIMC",div(field("weight"),pow(field("height"),2)))
        ))
      .foreach(tuple->{
```

```
        String name = tuple.getString("name");
        long numPeople = tuple.getLong("numPeople");
        float avgHeight = tuple.getFloat("averageHeight");
        float avgIMC = tuple.getFloat("averageIMC");
    });

// Filtering before aggregate
SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd");
Date date = sdf.parse("1970/01/01");

people.find()
      .filter(gt("birthdate", date(date)))
      .aggregate(.asList("name"),.asList(
        count("numPeople"),
        avg("averageHeight", field("height"))
        avg("averageIMC",div(field("weight"),pow(field("height),2)))
        ))
      .foreach(tuple->{
        String name = tuple.getString("name");
        long numPeople = tuple.getLong("numPeople");
        float avgHeight = tuple.getFloat("averageHeight");
        float avgIMC = tuple.getFloat("averageIMC");
    });

//Filtering after aggregate
people.find()
      .aggregate(.asList("name"),.asList(
        count("numPeople"),
        avg("averageHeight", field("height"))
        avg("averageIMC",div(field("weight"),pow(field("height),2)))
        ))
      .filter(gt("averageIMC",float(26)).or(ge("numPeople",100)))
      .foreach(tuple->{
        String name = tuple.getString("name");
        long numPeople = tuple.getLong("numPeople");
        float avgHeight = tuple.getFloat("averageHeight");
        float avgIMC = tuple.getFloat("averageIMC");
    });
```

Kivi_API_examples Delete with a Filter:

```
import static Filters.*;
import static Constants.*;
import static Expressions.*;

// Simple filter
people.delete(Filters.eq("name", string("John")));

// Expression usage
people.delete(Filters.gt(
        div( field("weight"),
          pow(field("height"),2)),
          float(26)
      ));
```

4. JDBC API

JDBC is a really well-known and wide-spread way of accessing database information. There are lots of resources in the WEB and SQL is very powerful helping you doing complex queries.

LeanXcale provides a JDBC driver so you can leverage all the power of JDBC to access LeanXcale and do any kind of operation in a similar way you could do it over any other database.

5. JDBC Resources

As there are a lot of resources about JDBC to do any kind of operation, we won't cover the JDBC interface in depth. What follows in the next pages is a full example of a simple JDBC program working over LeanXcale.

5.1. Code examples

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import java.util.ArrayList;
import java.util.Properties;
```

```java
/**
 * <p>
 * This sample program is a minimal Java application showing JDBC
access to a
 * LeanXcale database.</p>
 * The application runs in a different JVM from LeanXcale and only
needs to load the client
 * driver.</p>
 */
public class SimpleApp
{
    private String driver = "com.leanxcale.jdbc.ElasticDriver";
    private String protocol = "jdbc:leanxcale://";
    private String cluster="localhost";

    /**
     * <p>
     * Starts the demo by creating a new instance of this class and
running
     * the <code>go()</code> method.</p>
     * <p>
     * <p>
     * The LeanXcale cluster (or standalone JVM) must already be
running when trying to obtain
     * client connections. This demo program will will try to connect
to a ZooKeeper cluster on this
     * host (the localhost).
     * </p>
     * <p>
     * When running this demo, you must include the driver in the
     * classpath of the JVM.
     * </p>
     * @param args This program accepts one optional argument
specifying the Zookeeper cluster URL,
     *             such as as server:port. The default is to use the
localhost cluster.
     */
    public static void main(String[] args)
    {
        new SimpleApp().go(args);
        System.out.println("SimpleApp finished");
    }

    /**
     * <p>
     * Starts the actual demo activities. This includes loading the
correct
     * JDBC driver, creating a database by making a connection,
     * creating a table in the database, and inserting, updating and
retrieving
     * some data. Some of the retrieved data is then verified
(compared) against
     * the expected results. Finally, the table is deleted.</p>
```

```
     * <p>
     *
     * @param args - Optional argument specifying which framework or
JDBC driver
     *        to use to connect to LeanXcale cluster or standalone
JVM. Default is the embedded framework,
     *        see the <code>main()</code> method for details.
     * @see #main(String[])
     */
    void go(String[] args)
    {
        /* parse the arguments to determine which framework is
desired*/
        parseArguments(args);

        System.out.println("SimpleApp starting in.");

        /* load the desired JDBC driver */
        loadDriver();

        /* We will be using Statement and PreparedStatement objects
for
         * executing SQL. These objects, as well as Connections and
ResultSets,
         * are resources that should be released explicitly after
use, hence
         * the try-catch-finally pattern used below.
         * We are storing the Statement and Prepared statement object
references
         * in an array list for convenience.
         */
        Connection conn = null;
        ArrayList<Statement> statements = new ArrayList(); // list of
Statements, PreparedStatements
        PreparedStatement psInsert = null;
        PreparedStatement psUpdate = null;
        Statement s = null;
        ResultSet rs = null;
        try
        {
            Properties props = new Properties(); // connection
properties

            String dbName = "leanxcaleDB"; // the name of the
database

            /*
             * This connection specifies create=true in the
connection URL to
             * cause the database to be created when connecting for
the first
             * time.
             *
```

```
             */
            conn = DriverManager.getConnection(protocol + cluster +
"/" + dbName
                    + ";create=true", props);

            System.out.println("Connected to and created database " +
dbName);

            // We want to control transactions manually. Autocommit
is on by
            // default in JDBC.
            conn.setAutoCommit(false);

            /* Creating a statement object that we can use for
running various
             * SQL statements commands against the database.*/
            s = conn.createStatement();
            statements.add(s);

            // We create a table...
            s.execute("create table location(num int, addr
varchar(40))");
            System.out.println("Created table location");

            // and add a few rows...

            /* It is recommended to use PreparedStatements when you
are
             * repeating execution of an SQL statement.
PreparedStatements also
             * allows you to parameterize variables. By using
PreparedStatements
             * you may increase performance (because the LeanXcale
engine does not
             * have to recompile the SQL statement each time it is
executed) and
             * improve security (because of Java type checking).
             */
            // parameter 1 is num (int), parameter 2 is addr
(varchar)
            psInsert = conn.prepareStatement(
                        "insert into location values (?, ?)");
            statements.add(psInsert);

            psInsert.setInt(1, 1956);
            psInsert.setString(2, "Webster St.");
            psInsert.executeUpdate();
            System.out.println("Inserted 1956 Webster");

            psInsert.setInt(1, 1910);
            psInsert.setString(2, "Union St.");
            psInsert.executeUpdate();
            System.out.println("Inserted 1910 Union");
```

```
            // Let's update some rows as well...

            // parameter 1 and 3 are num (int), parameter 2 is addr
(varchar)
            psUpdate = conn.prepareStatement(
                        "update location set num=?, addr=? where
num=?");
            statements.add(psUpdate);

            psUpdate.setInt(1, 180);
            psUpdate.setString(2, "Grand Ave.");
            psUpdate.setInt(3, 1956);
            psUpdate.executeUpdate();
            System.out.println("Updated 1956 Webster to 180 Grand");

            psUpdate.setInt(1, 300);
            psUpdate.setString(2, "Lakeshore Ave.");
            psUpdate.setInt(3, 180);
            psUpdate.executeUpdate();
            System.out.println("Updated 180 Grand to 300 Lakeshore");


            /*
               We select the rows and verify the results.
             */
            rs = s.executeQuery(
                    "SELECT num, addr FROM location ORDER BY num");

            /* we expect the first returned column to be an integer
(num),
             * and second to be a String (addr). Rows are sorted by
street
             * number (num).
             *
             * Normally, it is best to use a pattern of
             *  while(rs.next()) {
             *    // do something with the result set
             *  }
             * to process all returned rows, but we are only
expecting two rows
             * this time, and want the verification code to be easy
to
             * comprehend, so we use a different pattern.
             */

            int number; // street number retrieved from the database
            boolean failure = false;
            if (!rs.next())
            {
                failure = true;
                reportFailure("No rows in ResultSet");
            }
```

```
            if ((number = rs.getInt(1)) != 300)
            {
                failure = true;
                reportFailure(
                        "Wrong row returned, expected num=300, got "
+ number);
            }

            if (!rs.next())
            {
                failure = true;
                reportFailure("Too few rows");
            }

            if ((number = rs.getInt(1)) != 1910)
            {
                failure = true;
                reportFailure(
                        "Wrong row returned, expected num=1910, got "
+ number);
            }

            if (rs.next())
            {
                failure = true;
                reportFailure("Too many rows");
            }

            if (!failure) {
                System.out.println("Verified the rows");
            }

            // delete the table
            s.execute("drop table location");
            System.out.println("Dropped table location");

            /*
               We commit the transaction. Any changes will be
persisted to
               the database now.
             */
            conn.commit();
            System.out.println("Committed the transaction");
        }
        catch (SQLException sqle)
        {
            printSQLException(sqle);
        } finally {
            // release all open resources to avoid unnecessary memory
usage

            // ResultSet
```

```
            try {
                if (rs != null) {
                    rs.close();
                    rs = null;
                }
            } catch (SQLException sqle) {
                printSQLException(sqle);
            }

            // Statements and PreparedStatements
            int i = 0;
            while (!statements.isEmpty()) {
                // PreparedStatement extend Statement
                Statement st = (Statement)statements.remove(i);
                try {
                    if (st != null) {
                        st.close();
                        st = null;
                    }
                } catch (SQLException sqle) {
                    printSQLException(sqle);
                }
            }

            //Connection
            try {
                if (conn != null) {
                    conn.close();
                    conn = null;
                }
            } catch (SQLException sqle) {
                printSQLException(sqle);
            }
        }
    }

    /**
     * Loads the appropriate JDBC driver.
     */
    private void loadDriver() {
        /*
         *  The JDBC driver is loaded by loading its class.
         *  If you are using JDBC 4.0 (Java SE 6) or newer, JDBC
drivers may
         *  be automatically loaded, making this code optional.
         *
         */
        try {
            Class.forName(driver).newInstance();
            System.out.println("Loaded the appropriate driver");
        } catch (ClassNotFoundException cnfe) {
            System.err.println("\nUnable to load the JDBC driver " +
driver);
```

```
            System.err.println("Please check your CLASSPATH.");
            cnfe.printStackTrace(System.err);
        } catch (InstantiationException ie) {
            System.err.println(
                    "\nUnable to instantiate the JDBC driver " +
driver);
            ie.printStackTrace(System.err);
        } catch (IllegalAccessException iae) {
            System.err.println(
                    "\nNot allowed to access the JDBC driver " +
driver);
            iae.printStackTrace(System.err);
        }
    }

    /**
     * Reports a data verification failure to System.err with the
given message.
     *
     * @param message A message describing what failed.
     */
    private void reportFailure(String message) {
        System.err.println("\nData verification failed:");
        System.err.println('\t' + message);
    }

    /**
     * Prints details of an SQLException chain to
<code>System.err</code>.
     * Details included are SQL State, Error code, Exception message.
     *
     * @param e the SQLException from which to print details.
     */
    public static void printSQLException(SQLException e)
    {
        // Unwraps the entire exception chain to unveil the real
cause of the
        // Exception.
        while (e != null)
        {
            System.err.println("\n----- SQLException -----");
            System.err.println("  SQL State:  " + e.getSQLState());
            System.err.println("  Error Code: " + e.getErrorCode());
            System.err.println("  Message:    " + e.getMessage());
            e = e.getNextException();
        }
    }

    /**
     * Parses the arguments given and sets the values of this class'
instance
     * variables accordingly.
     * @param args Zookeeper cluster URL
```

```
     * Only the first argument will be considered, the rest will be
ignored.
     */
    private void parseArguments(String[] args)
    {
        if (args.length > 0) {
            cluster=args[0];
        }
    }
}
```