EXERAY

Jaguar User Manual

(Rev. 2.0.9, 11/11/2015)

Contents

Introduction

The goal of this user manual is to provide a guide for customers to know and understand Exeray's Jaguar functionality and high performance. Jaguar leverages patented Indexing technology to outperform current MPP Databases in the market. This guide also provides instructions of benchmark testing Jaguar's performance in terms of data inserting, retrieving and concurrency.

Environment

Jaguar consists both Server and Client sides. You may install Server and Client packages either on the same machine or on different servers.

System minimum requirements for Server Side

Hardware： Pentium III 450 above, 1024MB RAM, 200G HD
Software： Linux 2.6.23 CentOS, RedHat, Fedora, x86, x86_64
File systems： ext4, or XFS
Environments have been tested: Fedora 3.1.0 x86_64, 8GB RAM, 512GB HD, ext4

System minimum requirements for Client Side

Hardware： Pentium III 450 above, 512MB RAM, 100G HD
Software： Linux 2.6.23 CentOS, RedHat, Fedora, x86, x86_64
File systems：ext4, or XFS
Environments have been tested: Fedora 3.1.0 x86_64, 8GB RAM, 512GB HD,ext4

Jaguar Deployment

There three deployment modes for Jaguar system:

Mode A

Native Mode: where only one server is used and the server provides enough storage space.

Mode B

Scalable Mode: a cluster of servers are deployed but they appear as one server to Jaguar as one distributed file system. Data operations are performed on one server.

Mode C

Distributed Mode, a cluster of servers are deployed and each server can accept connections from clients and perform data operations.

The characteristics of each mode are displayed in the following table:

Mode	Speed	Storage	Concurrency
A: Native Mode	Best	Good	Good
B: Scalable Mode	Good	Best	Good
C: Distributed Mode	Average	Best	Best

Jaguar Installation: Native Mode

You can download all binary packages for 64bit machines of Jaguar software from our Github account (https://github.com/exeray/jaguar) and then you can install Jaguar in just one step. You may install Server, Client binaries either in the same machine or different servers. Jaguar Server will listen on TCP/IP port 8888, and the process name is “jdbserv”. The process can be started by any user, each having a different listening port. Jaguar provides shell scripts jdbstart and jdbstop for you to start and stop the Server.

Jaguar Server Setup

In the server machine, you can execute the following script using :

```
$ tar zxf jaguarservern.n.tar.gz
```

Then related files will be unzipped under jaguarservern.n directory:

```
$ cd jaguarservern.n
$ ./install.sh
```

The installation shell script will generate jag system user, copy config file *jaguar.conf* to

$HOME/jaguar/conf/jaguar.conf and copy jdbserv, jadmin, jdbstart and jdbstop to $HOME/jaguar/bin/. You should setup your $PATH environment variable to include the directory $HOME/jaguar/bin.

Configuration file $HOME/jaguar*/conf/jaguar.conf* includes the following parameters:

- port is the server listening port number, which you can change as needed.
- HASHJOIN_BUFFER_SIZE During hashjoin (starjoin), if star tables are smaller than this size (MB), they will be loaded into memory. The center table (first table) is not loaded into memory.
- BUFF_READER_BLOCKS When scanning a table, blocks of underlying file are loaded into a buffer which size is specified by this number. Higher number can boost performance during join or any scan operations.
- BUFF_READER_MAXMEM As described by above parameter, scanning a table uses a chunk of memory buffer. The maximum size of the buffer is bound by this parameter (MB) to preventing the system from running out of memory.
- RAY_LOG_LEVEL Lower number (min is 0) makes the server generate less logging messages. A higher number (max is 9) makes the server generates more debugging information.

Create database and users

After you have set up Jaguar, you may create database and users as jag user by invoking the jadmin script:

```
$ jadmin createdb mydb
```
mydb is your database name.

Use *jadmin createuser* to create your database users:

```
$ jadmin createuser tom:tom123
```
This will create user *tom* with password as *tom123* . The default user privilege is read and write.

You may grant more privileges by doing:

```
$ jadmin perm tom:read
```

This will grant only *read* privilege to user *tom.* The default *users* generated are general users who cannot query and change database System metadata. General users can only query and change database System Meta data by upgrading to Admin account. You can achieve this by doing:

```
$ jadmin role tom:admin
```

This command will upgrade the user *tom* to Admin level user.

And here is the man page:

```
$ jadmin
jadmin command [argument]
```

jadmin createdb <databasename>

- createa new database

jadmin dropdb <databasename>

- drop a database

jadmin createuser <username:password>

- createa new database user with username and password

jadmin dropuser <username>

- drop a database user

jadmin role <username:admin/user>

- update role of user (admin or user)

jadmin perm <username:read/write>

- update permission of user (read: read only, write: read and write)

Jaguar Server Startup

After you have created database and users, you may start Jaguar server:

```
$ $HOME/jaguar/bin/jdbstart
```

Then Jaguar server will listen on port 8888. After Server is started up, you may still create more Databases and User Accounts. Newly created Databases will take effect immediately while newly create Users will take about 10 minutes to be recognized by *jdbserv server process* . The Server log will be under $HOME/jaguar/log/ directory.

Jaguar Client Setup

To install client platforms, you can excute this command :

```
$ tar zxf jaguarclientn.n.tar.gz
```
Related files will be unzipped to jaguarclientn.n.

```
$ cd jaguarclientn.n
$ ./install.sh
```

Jaguar Installation: Scalable Mode

In the Scalable Mode (Mode B), a cluster of servers will be used. You need to have a cluster of servers that are networked together in your Local Area Network or the Cloud. The following steps should be followed to install and setup the Scalable Mode environment:

One your first server in the cluster:

Step 1: Download the jaguar-nnn.tar.gz from any user account
Step 2: Unpack the gzipped tar file (tar zxf jaguar-nnn.tar.gz)
Step 3: Execute as root or sudo SetupUserAccount.sh to create user account
Step 4: sudo to jaguar user account and execute install.sh
Step 5: sudo to jaguar account and complete conf/jaguar.conf and host.conf
(in jaguar.conf set CLUSTER_MODE=no)
Step 6: Execute as root or sudo /home/jaguar/jaguar/bin/SetupGlusterOnFirstHost.sh
Step 7: sudo to jaguar account and run jadmin to create databases and user accounts.
Step 8: sudo to jaguar account and run bin/jobstart

On other servers in the cluster:
Nothing should be done.

After all of the above steps have been finished, the on every server execute the bin/jdbstart command as jaguar user.

Jaguar Installation: Distributed Mode

In the Distributed Mode (Mode C), a cluster of servers will be used and each user is used as a master to take connections and queries. You need to have a cluster of servers that are networked together in your Local Area Network or the Cloud. The following steps should be followed to install and setup the Distributed Mode environment:

One your first server in the cluster: (note CLUSTER_MODE=yes)

Step 1: Download the jaguar-nnn.tar.gz from any user account
Step 2: Unpack the gzipped tar file (tar zxf jaguar-nnn.tar.gz)
Step 3: Execute as root or sudo SetupUserAccount.sh to create user account
Step 4: sudo to jaguar user account and execute install.sh
Step 5: sudo to jaguar account and complete conf/jaguar.conf and host.conf
(in jaguar.conf set CLUSTER_MODE=yes)
Step 6: Execute as root or sudo /home/jaguar/jaguar/bin/SetupGlusterOnFirstHost.sh
Step 7: sudo to jaguar account and run jadmin to create databases and user accounts.
Step 8: sudo to jaguar account and run bin/jobstart

On other servers in the cluster:

Step 1: Download the jaguar-nnn.tar.gz from any user account
Step 2: Unpack the gzipped tar file (tar zxf jaguar-nnn.tar.gz)
Step 3: Execute as root or sudo SetupUserAccount.sh to create user account
Step 4: sudo to jaguar user account and execute install.sh
Step 5: sudo to jaguar account and complete conf/jaguar.conf and host.conf
(in jaguar.conf set CLUSTER_MODE=yes)
Step 6: Execute as root or sudo ~jaguar/jaguar/bin/SetupGlusterOnOtherHosts.sh
Step 7: sudo to jaguar account and run bin/jobstart

After all of the above steps have been finished, the on every server execute the bin/jdbstart command as jaguar user.

Platform Provisioning

Mount noatime

File Input and Output (IO) is one of the most important performance indicator for Database. We suggest that you may close the *access time* option for your file system. You may disable this in */etc/fstab* as *root* :

defaults,noatime

Installation Verification

After you install the Jaguar Server, please make sure：
1) No other service or processes use port 8888
2) User *$HOME/jaguar/* was created
3) Following files exist and can be executed:

*$HOME/jaguar/*bin/jdbserv
*$HOME/jaguar/*bin/jadmin
*$HOME/jaguar/*bin/jdbstart
*$HOME/jaguar/*bin/jdbstop

Test Execution

Test Approaches

There are three ways to benchmarking test against Jaguar:

1. Interaction between Jaguar Client and Server side:
Test by running jag client, typing SQL-like commands. Then the Server will respond when receiving queries.

2. APIs calls

Test by writing C programs which calls Jaguar APIs to perform related data Select, Insert operations.

3. Operations on kernel data files
Test by using Jaguar C class to perform related data Select, Insert operations on Server Kernel data files

Baseline Performance

Following benchmarks can demonstrate the performance advantages of Jaguar:

- Data Load/Insert
- Date Query
- Indexing performance
- Memory usage
- Join performance
- Above performance in a concurrent way

Insert performance

There are 3 ways to test Insert performance:
1) Perform batch load on the Server side ;
2) Insert single record from Client side;
3) Perform Insert operation directly on Kernel data files.

Please make sure Server, Client were correctly installed and executed.

Preparations：

1. Create the user

```
$ jadmin createuser test:test
```

2. Create the table

```
$ jag –u test –p test –d test –h 127.0.0.1:8888
jaguar> create table test ( key: uid char(16), value: addr char(16) );
```

(A) Batch Load
You can test Jaguar Server by loading 3 million records. And the sample 3 million records can be generated by program *genrand* which comes with Client bin.

```
$ genrand 3000000 71
Please enter a line of header which contains name of keys and values.
Example: uid,v1,v2,v3
Example: k1,k2,v1,v2,v3,v4
Your input: uid,v1,v2,v3  (hit enter)
```

```
$ mv genrand.out /tmp/3M.txt
```

Then in jag client side (any user can run jag client side) use the following command to load 3 million records to test table:

```
jaguar> load file /tmp/3M.txt into test format H;
```

Expected behavior： After about 2 minutes, jag will tell how long it takes to load data in milliseconds.

When loading data, the input file can take these formats: H, and V. In format H, the first line is a header line where the names of key and value fields are listed, separated by a comma. Each data line is a list of keys and values, separated by a field separator (such as comma). In format V, each line begins with keys, then follows a list of name=value pairs.

For example:

H format：

```
uid,value1,value2,value3
abcdefghij123456,DDDD,DDDD,KKK
mbcdefghkj123056,PPPP,KKKK,DDD
cbcdefghdj193456,SSS,SSS,DDFDD
…
```

(except the first line, the values of each key and value can be quoted by single quote or double quote)

```
uid,value1,value2,value3
‘abcdefghij123456’,‘DDDD’,‘DDDD’,‘KKK’
‘mbcdefghkj123056’,‘PPPP’,‘KKKK’,‘DDD’
‘cbcdefghdj193456’,‘SSS’,‘SSS’,‘DDFDD’
…
```

V format：

```
abcdefghij123456,phone=13800002222,name=zhao
mbcdefghkj123056,phone=13422223333,name=li
cbcdefghdj193456,phone=13588882222,name=zhang
```

…

(except the first line, the values of each key and value can be quoted by single quote or double quote)

```
"abcdefghij123456",phone="13800002222",name="lee"
"mbcdefghkj123056",phone="13422223333",name="adam"
"cbcdefghdj193456",phone="13588882222",name="jenny"
…
```

You can also write all your SQL commands in a file and feed the file to jag program:

```
$ vi mycommands.rql

create table test1 ( key: uid char(16), value: addr char(16) );
load file /tmp/1000.txt into test1 format V;
quit;
```

Then execute shell command:

```
$ jag < mycommands.rql
```

(B) Client single record Insert

jbench program in Client package will help insert, modify and query records on Server. The following command will generate 10000 numbers randomly and insert record to *jbench* table in the Server.

```
$ jbench  –r "10000:0:0"  | tee –a test.log
```

In *"10000:0:0"* the first "10000" the times for Insertion, the second "0" is the times for Update and the third "0" is the times for Query.

The database used in jbench is 'test', and the table in the jbench program is 'jbench'. The table 'jbench' has a key named 'uid' of 16 bytes, a value named 'addr' of 32 bytes.

Expected behavior: jbench will insert data about 6000-8000 record per second including network overhead.

Join Performance

You may follow these steps to test table Join performance:
1) Use *genrand* generate some files containing 5 million records each.

```
$ genrand 5000000  71
Please enter a line of header which contains name of keys and values.
Example: uid,v1,v2,v3
Example: k1,k2,v1,v2,v3,v4
Your input: uid,v1,v2,v3   (then hit enter)

$ mv genrand.out /tmp/5M1.txt
$ genrand 5000000  71
$ mv genrand.out /tmp/5M2.txt
$ genrand 5000000  71
$ mv genrand.out /tmp/5M3.txt
```

2) Create tables

```
jaguar> create table t1 ( key: uid char(32), value: v1 char(16), v2 char(16), v3 char(16) );
jaguar> create table t2 ( key: uid char(32), value: v1 char(16), v2 char(16), v3 char(16) );
jaguar> create table t3 ( key: uid char(32), value: v1 char(16), v2 char(16), v3 char(16) );
```

3) Load Data

```
jaguar> load file /tmp/5M1.txt into t1 format H;
jaguar> load file /tmp/5M2.txt into t2 format H;
jaguar> load file /tmp/5M3.txt into t3 format H;
```

Each Load will need about 5 minutes.

4) Join

```
jaguar> select * join ( t1, t2, t3 ) limit 10;
```

This will Join 3 tables by joined key and you may also monitor memory usage by using *top* command.

Expected behavior: 3 million records tables will be joined and the result will be returned in 0.9 seconds.

Concurrent request performance

You may use *jbench* to simulate concurrent connections and requests. The concurrent testing will include following:
1) 100% users data insert concurrently
2) 100% users data select concurrently
3) 20% users data insert and 80% users data select concurrently

100% users data insert concurrently

```
$ jbench  –r “10000:0:0” –c 4 | tee –a test.log
```

“c 4” means 4 threads, and every thread will request 10000 insert operations from Server. Actual concurrent threads number should align with the number of Server Cores: for example if the Server had 16 cores, the concurrent threads should be set to 16 to make sure the best performance of Insertion.

100% users data query concurrently

```
$ jbench –r “0:0:10000”–c 4 | tee –a test.log
```

“c 4” means 4 threads, and every thread will request 10000 select operations from Server. Actual concurrent threads number should align with the number of Server Cores: for example if the Server had 16 cores, the concurrent threads should be set to 16 to make sure the best performance of Selection.

20% users data insert and 80% users data select concurrently

```
$ jbench  –r “2000:0:8000”–c 4 | tee –a test.log
```

“c 4” means 4 threads, and every thread will request 8000 select operations and 2000 insert operation from Server. The concurrent threads number should be set the same as above. You may also use *top* to monitor memory usage here as well.

Expected behavior: The performance will increase when you add more threads and will stop improvement when threads are more than the number of cores on the server.

Baseline Functionality

Following features of Database can also be tested using the above method:
1) create database
2) create user
3) create table
4) insert data into table
5) select data from table
6) update data in table
7) delete data from table
8) join tables and select data
9) load file into table
10) drop a table
11) drop database

The man page is here to help:

```
jaguar> help
You can enter the following commands:
help use (how to use databases)
help desc (how to describe tables)
help show (how to show tables)
help create (how to create tables)
help insert (how to insert data)
help load (how to batch load data)
help select (how to select data)
help update (how to update data)
help delete (how to delete data)
help join (how to join two or more tables)
help drop (how to drop a table completely)
```

Programming Guide

Connection to Jaguar Server

Shell

```
$  $HOME/jaguar/bin/jag –u USERNAME –p PASSWORD –h HOST:PORT –d DATABASE
```

```
Example:  $  $HOME/jaguar/bin/jag –u test –p mysecret –h dbhost:8888 –d mydb
jaguar>
```

C++/C

```
JaguarCPPClient adb;
adb.connect( adb, host, port, username, passwd, dbname, NULL, 0 );
```

Java JDBC

```
DataSource ds = new JaguarDataSource("127.0.0.1", 8888, "mydb");
Connection connection = ds.getConnection("testuser", "testpasswd");
```

Insert data

Shell

```
jaguar> insert into mytable ( uid, addr, age ) values ( ‘Joe’, ‘123 A Street, CA’, 35 );
```

C++/C

```
adb.query( adb, “insert into mytable ( uid, addr, age ) values ( ‘Joe’, ‘123 A Street, CA’, 35 ) “ );
```

Java JDBC

```
Statement statement = connection.createStatement();
statement.executeUpdate("insert into tab (uid, addr) values ( ‘Jill’, ‘333 B Ave, CA’ );");
```

Query data

Shell

```
jaguar> select * from t1;
```

Commands must be entered in one line.

```
Jaguar> select * from mytable where uid=jack or uid like ‘jen%’ and phone like ‘925%’;

Jaguar> select * from mytable where uid in (‘tom’, ‘jack’ )’;
```

C++/C

```
adb.query( adb, “select * from t1;“ );
while ( adb.reply() ) {
            adb.printRow( stdout );
            char *p = adb.getValue( &row,  “uid” );
            printf(“uid=%s\n”, p );
            free( p );
}
adb.freeResult();
```

Java JDBC

```
Statement statement = connection.createStatement();
ResultSet rs = statement.executeQuery("select * from tab;");
String val;
String m1;
```

```
while(rs.next()) {
        val = rs.getString("uid");
        m1 = rs.getString("m1");
        System.out.println( "uid: " + val + " m1: " + m1  );
 }
 rs.close();
statement.close();
```

Query with Index

Suppose table mytable contains key: uid and value: v1, v2, v3. If you need to query data in mytable according to a non--key column (or several columns), then you can create an index on the column(s) and query mytable by using the index. For example:

```
create index mytable_idx23  on mytable( v2 char(16), v3 char(16) );
```

Shell

```
jaguar> select * from mytable use index(mytable_idx23) where v2=’somevalue' and v3=’somevalue’;
```

C++/C

```
adb.query(  “select * from mytable use index(mytable_idx23) where v2>=’somevalue' ; “ );
while ( adb.reply(  ) ) {
            adb.printRow( stdout );
            char *p = adb.getValue( “uid” );
            printf(“uid=%s\n”, p );
            free( p );
}
adb.freeResult(  );
```

Java JDBC

```
Statement statement = connection.createStatement();
ResultSet rs = statement.executeQuery("select * from mytable use index(mytable_idx23) where
v2 >= ‘myvalue’;");
String val;
String m1;
while(rs.next()) {
        val = rs.getString("uid");
        m1 = rs.getString("m1");
        System.out.println( "uid: " + val + " m1: " + m1  );
 }
 rs.close();
statement.close();
```

Table Join

Join

```
jaguar>  select *  join ( hosttable, tab1, tab2, …) where …;
```

Tables tab1, tab2, … must have the same set of keys. Their set of keys must be the same or subset of the keys of the first table hosttable. For example:

Hosttable has keys k1,k2,k3 and tab2 has keys k1,k2 and tab3 has keys k1,k2.

Hosttable has keys k1,k2,k3 and tab2 has keys k1,k2,k3 and tab3 has keys k1,k2,k3.

Star Join

Star join works for tables where the fact table has columns that are keys in dimension tables.

```
jaguar>  select fact.trxnid, fact.volume, product.prodid, customer.phone starjoin(fact, product, customer);
```

fact table:

key	values			
trxnid	prodid	custid	volume	tdate

product table:

key	values						
prodid	price	producer	name	weight	height	width	depth

customer table:

key	values					
custid	fname	lname	mobilephone	homephone	email	address

Star join is preferred when the dimension tables are small so that they can be loaded into memory for fast join with the fact table.

Index Join

Index join is similar to join except the host table is an index.

```
jaguar> select * indexjoin( index(table1_index), tab2, tab3 ) where …;
```

Here tab2 and tab3 must have same keys, which are full set or sub set of the index table1_index. For example:

1) Index table1_index has keys m1,m2. Table tab2 has keys m1,m2 and so does table tab3.
2) Index table1_index has keys m1,m2. Table tab2 has key m1 and so does table tab3.

Index join is useful when table1 (from which the index table1_index is derived) cannot easily join tab2 and tab3 because table1 has different keys than tab2 and tab3. Table1 may have the following schema:

Table1:

Key	value
k1,k2,k3	m1,m2

table1_index:

key
m1,m2,k1,k2,k3

tab2:

key	value
m1	v1,v2,v3,v4

tab3:

key	value
m1	z1,z2,z3,z4

The index and other tables have similar keys so that fast merge join is performed to obtain the results, regardless the size of the index and the tables.

Operation

Stop a job

In jag interactive shell, if you want to stop a long-running job, ctrl-C keys can be entered to terminate the client shell as well as the server thread that performs the job.

```
jaguar> some long running task; …..  (Ctrl-C entered)
$  Server thread killed successfully
```

Refresh configuration

If any parameter in $HOME/jaguar/jaguar.conf is modified, or a new user is created with jadmin program, you can send a SIGHUP signal to the jdbserv process by this command:

```
$ kill –HUP  38293
```

Where 38293 is the process ID of jdbserv that can be obtained by command "ps –aux|grep jdbserv".

Jaguar Data Types

Currently Jaguar supports these data types:

1. Character string
   char(length) -- it is a fixed length character string in key columns. It is a variable length string in the value columns.

2. Integer
   int(length) - it is a fixed length integer (including short and long integers) in the key columns and variable length integer in the value columns.

3. Float
   float(L,d) -- a float decimal number with length L and number of digits after the decimal point.

4. Double
   double(L,d) -- similar to float except in internal representation and calculation, it is treated as double precision float number.

5. Time
   time(*) – a 16 digits time value in terms of microseconds. When a time data is loaded or inserted into Jaguar, the following format must be used:

   ```
   YYYY-MM-DD hh:mm:ss.[uuuuuu] +HH:MM
   YYYY-MM-DD hh:mm:ss.[uuuuuu] -HH:MM
   ```

   Where YYYY is the 4-digit year symbol, such as 2005
   MM is the month (1-12), such as 10
   DD is the date in 1-31, such 04
   hh:mm:ss is hour:minute:seconds such as 02:23:21
   uuuuuu is optional fractional seconds (or microseconds)

   +HH:MM and -HH:MM are optional time zone difference from GMT standard time.

   Example:

   From California, USA:
   ```
   insert into  sales (uid, sttime) values ( 1232, ‘2014-11-23 16:32:21 -08:00’ );
   insert into  sdata (deviceid, logtime) values ( 1232, ‘2015-10-23 13:32:21.234019 -08:00’ );
   select * from sales where sdate > ‘2014-12-10 03:12:23’;
   ```

Spark Data Analysis

Since Jaguar provide JDBC connectivity, developers can use Apache Spark to load data from Jaguar and perform data analytics and machine learning. The advantages provided by Jaguar is that Spark can load data faster, especially for loading data satisfying complex conditions, from Jaguar than from other data sources. The following code is based on two tables that have the following structure:

```
create table int10k ( key:  uid int(16),  score float(16.3), value:  city char(32) );

create table int10k_2 ( key: uid int(16),  score float(16.3), value: city char(32) );
```

Scala program:

```
import org.apache.spark.SparkConf

import org.apache.spark.SparkContext

import org.apache.spark.SparkContext._

import scala.collection._

import org.apache.spark.sql._

import org.apache.spark.sql.types._

import org.apache.log4j.Logger

import org.apache.log4j.Level

import com.jaguar.jdbc.internal.jaguar._

import com.jaguar.jdbc.JaguarDataSource



object TestScalaJDBC {

   def main(args: Array[String]) {

      sparkfunc()

   }


   def sparkfunc()

   {
```

```
Class.forName("com.jaguar.jdbc.JaguarDriver");
val sparkConf = new SparkConf().setAppName("TestScalaJDBC")
val sc = new SparkContext(sparkConf)
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext.implicits._

Logger.getLogger("org").setLevel(Level.OFF)
Logger.getLogger("akka").setLevel(Level.OFF)

val people = sqlContext.read.format("jdbc")
  .options(
     Map( "url" -> "jdbc:jaguar://127.0.0.1:8888/test",
         "dbtable" -> "int10k",
         "user" -> "test",
         "password" -> "test",
         "partitionColumn" -> "uid",
         "lowerBound" -> "2",
         "upperBound" -> "2000000",
         "numPartitions" -> "4",
         "driver" -> "com.jaguar.jdbc.JaguarDriver"
  )).load()

// work fine
people.registerTempTable("int10k")
people.printSchema()

val people2 = sqlContext.read.format("jdbc")
```

```
  .options(
     Map( "url" -> "jdbc:jaguar://127.0.0.1:8888/test",
         "dbtable" -> "int10k_2",
         "user" -> "test",
         "password" -> "test",
         "partitionColumn" -> "uid",
         "lowerBound" -> "2",
         "upperBound" -> "2000000",
         "numPartitions" -> "4",
         "driver" -> "com.jaguar.jdbc.JaguarDriver"
  )).load()
people2.registerTempTable("int10k_2")

// sort by columns


people.sort("score").show()
people.sort($"score".desc).show()
people.sort($"score".desc, $"uid".asc).show()
people.orderBy($"score".desc, $"uid".asc).show()



// select by expression
people.selectExpr("score", "uid"  ).show()
people.selectExpr("score", "uid as keyone"  ).show()
people.selectExpr("score", "uid as keyone", "abs(score)" ).show()

// select a few columns
```

```
val uid2 = people.select("uid", "score")
uid2.show();


// filter rows
val below60 = people.filter(people("uid") > 20990397 ).show()


// group by
people.groupBy("city").count().show()


// groupby and average
people.groupBy("city").avg().show()


people.groupBy(people("city"))
       .agg(
            Map(
                 "score" -> "avg",
                 "uid" -> "max"
               )
           )
       .show();



// rollup
people.rollup("city").avg().show()
people.rollup($"city")
       .agg(
            Map(
                "uid" -> "avg",
                "score" -> "max"
              )
           )
       .show();



// cube
people.cube($"city").avg().show()
people.cube($"city")
       .agg(
            Map(
                "uid" -> "avg",
                "score" -> "max"
              )
           )
       .show();

// describe statistics
people.describe( "uid", "score").show()

// find frequent items
people.stat.freqItems( Seq("uid") ).show()

// join two tables
people.join( people2, "uid" ).show()
people.join( people2, "score" ).show()
```

```
        people.join(people2).where ( people("uid") === people2("uid")  ).show()

        people.join(people2).where ( people("city") === people2("city")  ).show()

        people.join(people2).where ( people("uid") === people2("uid") and people("city") ===
people2("city")  ).show()

        people.join(people2).where ( people("uid") === people2("uid") && people("city") ===
people2("city") ).show()

        people.join(people2).where ( people("uid") === people2("uid") && people("city") ===
people2("city") ) .limit(3).show()


        // union

        people.unionAll(people2).show()


        // intersection

        people.intersect(people2).show()


        // exception

        people.except(people2).show()


        // Take samples

        people.sample( true, 0.1, 100 ).show()


        // distinct

        people.distinct.show()


        // same as distinct

        people.dropDuplicates().show()


        // cache and persist
        people.dropDuplicates.cache.show()

        people.dropDuplicates.persist.show()



        //  SQL dataframe

        val df = sqlContext.sql("SELECT * FROM int10k where uid < 200000000 and city between
'Alameda' and 'Berkeley' ")

        df.distinct.show()
```

The class generated from the above Scala program can be submitted to Spark as follows:

```
/bin/spark-submit --class TestScalaJDBC  \
    --master spark://masterhost:7077  \
    --driver-class-path   /path/to/your/jaguar-jdbc-2.0.jar \
    --driver-library-path  $HOME/jaguar/lib \
    --conf spark.executor.extraClassPath=/path/to/your/jaguar-jdbc-2.0.jar  \
    --conf spark.executor.extraLibraryPath=$HOME/jaguar/lib   \
     /path/to/your_project/target/scala-2.10/testjdbc_2.10-1.0.jar
```

SparkR with Jaguar

Once you have R and SparkR packages installed, you can start the SparkR program by executing the following command:

```
#!/bin/bash
```

```
export JAVA_HOME=/usr/lib/java/jdk1.7.0_75
LIBPATH=/usr/lib/R/site-library/rJava/libs:$HOME/jaguar/lib
LDLIBPATH=$LIBPATH:$JAVA_HOME/jre/lib/amd64:$JAVA_HOME/jre/lib/amd64/server
JDBCJAR=$HOME/jaguar/lib/jaguar-jdbc-2.0.jar
```

```
sparkR \
–driver-class-path $JDBCJAR \
–driver-library-path $LDLIBPATH \
–conf spark.executor.extraClassPath=$JDBCJAR \
–conf spark.executor.extraLibraryPath=$LDLIBPATH
```

Then in the SparkR command line prompt, you can execute the following R commands:

```
library(RJDBC)
library(SparkR)

sc <- sparkR.init(master=”spark://mymaster:7077″, appName=”MyTest”)

sqlContext <- sparkRSQL.init(sc )

drv <- JDBC(“com.jaguar.jdbc.JaguarDriver”, “/home/exeray/jaguar/lib/jaguar-jdbc-2.0.jar”,
“`”)

conn <- dbConnect(drv, “jdbc:jaguar://localhost:8888/test”, “test”, “test” )

dbListTables(conn)

df <- dbGetQuery(conn, “select * from int10k where uid > ‘anxnfkjj2329’ limit 5000;”)

head( df )
```

```
#correlation
> cor(df$uid,df$score)
[1] 0.05107418
```

```
#build the simple linear regression
> model<-lm(uid~score,data=df)
> model
```

```
Call:
lm(formula = uid ~ score, data = df)

Coefficients:
(Intercept) score
2.115e+07 1.025e-03

#get the names of all of the attributes
> attributes(model)
$names
[1] “coefficients” “residuals” “effects” “rank”
[5] “fitted.values” “assign” “qr” “df.residual”
[9] “xlevels” “call” “terms” “model”

$class
[1] “lm”
```