# ParaLite User's Guide

May 2012

**Ting Chen**

**University of Tokyo**
**7-3-1 Hongo Bunkyo-ku Tokyo, 113-0033 Japan**

# Table of Contents

# 1 Getting Started

## 1.1 Prerequisites

To play with ParaLite, you need:

Python interpreter

GXP (Grid and Cluster Shell)

SQLite

ParaLite is developed on Linux in python and only tested on Linux. ParaLite supports python 2.7 and probably newer versions, but python 2.5 is not supported.

It based on SQLite, a popular single-node database system.

```
http://www.sqlite.org
```

You have to install SQLite on each data node.

GXP is used to explore nodes. It is a parallel shell tool to let you run an identical or a similar command line to many machines in parallel and get results back interactively. You can obtain it from sourceforge:

```
http://sourceforge.net/projects/gxp/
```

## 1.2 Installation

unpack the tarball:

```
$ tar zxvf paralite-xx.tar.bz2
```

add `paralite` to the execution path:

```
$ export PATH=$PATH:/absolute/path/to/paralite-xx
```

or set the PATH in ~/.bashrc to make it work permanently.

To test your installation, type `paralite` to your shell prompt and see something like this.

```
$ paralite
  Usage: paralite /path/to/db STATEMENT
  Please enter "paralite help" for more information.
```

Before a query is issued, you should assure that all related nodes have already been explored by GXP:

```
$ gxpc use ssh huscs
$ gxpc explore huscs[[001-002]]
$ gxpc e hostname
  huscs001
  huscs002
```

# 2 Tutorial

## 2.1 Processes coordination methods

ParaLite is server-less and zero-configuration system, that is, you don't need start any process and specify any configuration before SQL is executed. To achieve this, a super process should be started to coordinate all other processes after they are started. Then where to start the super process and how everybody knows it? You need to specify hub information at the end of each query using three methods:

(1) `--hub db:hub_db` : Using a database as a hub;

```
$ paralite /path/to/db "query" --hub db:hub_db
```

If no hub info is specified, paraLite uses `/path/to/db` right after `paralite` as default hub.

(2) `--hub file://path/to/file` : Using a separate file which can be opened by all processes;

(3) `--hub host://hostname:port` : Starting the master process that listens to the `port` on machine `hostname`. This is commonly used because you can specify the super process running on a machine that explored all related nodes.

Note that, (1) or (2) is used only if the database and file can be shared among all processes, e.g. NFS is used.

## 2.2 Performing general SQL query

First, you can create table by the following command:

```
$ paralite test.db "create table x(a int, b varchar(10))"
```

This query is to create table `x` locally. If you want to specify the data nodes on which the table is created, `on` option is required as the following query:

```
$ paralite test.db "create table x(a, b) on huscs000"
```

Several nodes can be specified in a single query delimited by space.

If you have many nodes, typing each node name is very troublesome and painful. An improved description for multiple nodes is `[[xxx-yyy]]`, which represents a set of numbers between `xxx` and `yyy` (inclusive).

```
paralite test.db "create table x(a, b) on huscs[[000-002]]"
```

You can also use a configuration file to specify data nodes:

```
$ paraLite test.db create table x(a, b) on file node.conf
$ cat node.conf
huscs[[000-003]]
hongo[[100-112]]
```

Each partition can be divided into chunks and the number of chunks can be specified in the create statement:

```
$ paraLite test.db create table x(a, b) on huscs[[000-002]] chunk 3
```

The number of chunks is set based on the size of the database. As SQLite has bad performance on big data, it is better to make the size of each chunk to be small enough. However, you should not set it too small as with large number of chunks, the scheduling overhead is high. For example, if a whole table is 10GB and partitioned across 10 nodes, each node has about 1GB data. If the number of chunks is set to be 10, each chunk is only about 100MB.

This table can be partitioned either by hash fashion based on a specific key or round-robin fashion (by default). You can also specify the number of replica for each partition, 1 by default.

```
$ paralite test.db "create table x(a, b) [partition by key] [replica 3]"
```

The table also can hash-partitioned by multiple keys:

```
$ paralite test.db "create table x(a, b) [partition by key1, key2] [replica 3]"
```

Then, you need to load data to database:

```
$ cat test.dat
aa|test1
bb|test2
aa|test3
$ paralite test.db ".import test.dat x"
```

The default row separator and col separator in ParaLite are `\n` and `|` respectively. If your source data is not separated by them, assuming that row separator is '===' and col separator is '###', you can either change them by

```
$ paralite test.db ".row_separator ==="
$ paralite test.db ".col_separator ###"
```

or specify them in the `.import` command

```
$ paralite test.db ".import test.dat x -column_separator ### -row_separator ==="
```

Note that, at this point, paraLite cannot support session, that is, the values of all settings are changed explicitly.

Next, you can perform some selection on the table:

```
$ paralite test.db "select * from x where a='aa'"
aa|test1
aa|test3
$ paralite test.db "select count(*) from x"
3
$ paralite test.db "select a, count(*) from x group by a"
aa|2
bb|1
```

To specify SQL queries, you can write multiple queries in a single file and issue the command:

```
$ paralite test.db < a.sql
$ cat a.sql
create table x(a) on huscs[[001-002]];
.import x_file x;
```

```
select * from x;
```

Note: Queries (including special command (starts with .)) are separated by “;” even the special command does not need “;”.

Currently, ParaLite has some limitation in general SQL as follows:

(1) ParaLite cannot support compound operator: union, intersect, except

(2) ParaLite can support only nature join. Other joins like left join, right join are not supported.

(3) ParaLite cannot support complex selection with CASE, WHEN and IF....

These limitations will be fixed in the next version.

## 2.3 Performing collective query

collective query extends SQL syntax in which a user can define User-Defined Executable (UDX). The syntax of collective query with UDX is:

```
select a, F(b) as bb, c, ... from t where ....
with F="cmd_line"
     input stdin input_row_delimiter NEW_LINE input_col_delimiter NULL
     output stdout output_row_delimiter NEW_LINE output_col_delimiter NULL
     output_record_delimiter EMPTY_LINE
collective by 1
[--hub host://masternode:port] [-b blocksize]
```

Collective query has two features:

(1) Supportive of User-Defined Executable (UDX): an UDX is an executable binary or scrip which is defined in the query as a command line. Following the command line, you may want to specify the format of input and output data.

`input` : stdin by default or ’/path/to/file’

`input_row_delimiter` : NEW_LINE by default or ’any_string’

`input_col_delimiter` : NULL by default or ’any_string’

`output` : stdout by default or ’/path/to/file’

`output_row_delimiter` : NEW_LINE by default or ’any_string’

`output_col_delimiter` : NULL by default or ’any_string’

`output_record_delimiter` : EMPTY_LINE by default or ’any_string’

You can understand all these options by the following example:

```
$ paralite test.db "select * from document"
document_id | text
32819 | It is sunny today. I have a good mood.
82718 | I am studying in the lab. I want to go outside and play pingpong.
```

A Perl code below splits text into sentences and add an identification to each sentence.

```
$ cat ss
#!/usr/bin/perl
while (my $l = <STDIN>)
```

```
  chomp($l);
  my s = split(/\./, $l);
  for (my $i = 0; $i < scalar(s); ++$i)
    print "$i==$s[$i]\n";



$ cat a.dat
Sentence1. Sentence2. Sentence3.

$ cat a.dat | perl ss
1==Sentence1.
2== Sentence2.
3== Sentence3.

$ paralite test.db "select document_id, F(text) from document with F=\"ss\" out-
put_row_delimiter EMPTY_LINE"
32819 | 1==It is sunny today.
        2== I have a good mood.
82718 | 1==I am studying in the lab.
        2== I want to go outside and play pingpong.
$ paralite test.db "select document_id, F(text) from document with F=\"perl ss\" out-
put_col_delimiter '==' output_record_delimiter EMPTY_LINE"
32819 | 1 | It is sunny today.
32819 | 2 |  I have a good mood.
82718 | 1 | I am studying in the lab.
82718 | 2 |  I want to go outside and play pingpong.
```

An UDX can have more than one argument and output more than one column, e.g. F(a) as aa, G(a, b) as bb, G(a, b, c) as(bb, cc).

(2) Identification of Collective Query: Each query has an identification specified by `collective by ID`. Computing clients are grouped based on the ID. If a calculation is performed by 5 clients in parallel, 5 clients on any machines should issue a same query with same ID. Some clients can join the group during the calculation but before all data in data nodes are distributed.

To tune the performance, user can specify the block size by the option -b.

## 2.4 Analyzing a query

For a given SQL query, ParaLite first parses it grammatically using `pyparsing` (http://pyparsing.wikispaces.com/). You can see the syntax tree by issuing the following command:

```
$ paralite test.db "select sum(a) from T group by a" --parse-syntax
['SELECT', [['sum(a)']], 'FROM', [['T']], 'GROUP', 'BY', [['a']]]
- fromList: [['T']]
- group_by: [['a']]
- select: [['sum(a)']]
```

Each clause has an alias name, e.g. "group by a" is named by "group_by" and [['a']] is its value.

Then based on the syntax tree, the query is converted into an execution plan which composes of operators such as join, group and sub-query. You can get the execution plan of a query with the following command:

```
$ paralite test.db "select sum(a) from T group by a" --parse-plan
 (- group_by 0 expression=a input=['sum(a)'] output=['sum(a)'] group_key=['a']
key=[] split_key=[] tables=['T'] func=['sum(a)'] is_sql=0)
- (- sql 1 expression=select sum(a) from T group by a  input=[] output=['sum(a)']
key=[] split_key=['a'] tables=['T'] is_sql=None)

$ ./paralite test.db "select T.a from T, X where T.a = X.b" --parse-plan
 (- join 0 expression=['T.a = X.b '] input=['T.a', 'X.b'] output=['T.a']
key=['T.a', 'X.b'] split_key=[] tables=['T', 'X'] is_sql=0 is_sub_plan=False)
- (- sql 1 expression=select T.a from T  input=[] output=['T.a']
key=[] split_key=['T.a'] tables=['T'] is_sql=None)
- (- sql 2 expression=select X.b from X  input=[] output=['X.b']
key=[] split_key=['X.b'] tables=['X'] is_sql=None)
```

## 2.5 Special command

```
$ paralite test.db "special command"
```

special command:

```
.import FILE|DIR table [record_tag] [-column_separator col_sep]
                 [-row_separator row_sep] Import data from FILE into TABLE
.output FILE        Send output to FILENAME
.output stdout      Send output to the screen
.indices [TABLE]     Show names of all indices.
                     If TABLE specified, only show indices for tables
.row_separator STRING   Change row separator used by output mode and .import
.col_separator STRING   Change col separator used by output mode and .import
.show               Show the current values for various settings
.analyze SQL        Show the logical plan of a SQL query;
```

## 2.6 Configurations for performance

ParaLite allows you to make your own configuration file to control some parameters of SQLite or query execution plan. If you want to create the file, firstly create a file called `paralite.conf` in the current directory. Of course, you have to assure that the super process I mentioned above can access it.

```
$ cat paralite.conf
[db]
cache_size=-1        ; unit = KB,  -1: use the default size of sqlite
temp_store=0         ; where to store the temporary tables and indices:
```

```
                       0--default, 1--file, 2--memory
[runtime]
worker_num=-1, -1, -1  ; they are the number of worker for operators.
                         -1: use all data nodes
```

The value of worker_num depends on the shape of execution plan for a query. Before you set it, you probably need to use the analyze command to get the execution plan:

```
$ paralite test.db ".analyze 'select F(a) from x, y where x.a = y.b' with F=\"cmd\""
- udx: cmd
  -- join: x.a = y.b
     --- sql1: select x.a from x
     --- sql2: select y.b from y
```

So the `worker_num` could be -1, -1, 3, -1. You can only give non -1 value to operators who are not *sql* and *udx* since *sql* should be executed by every data node and *udx* is performed by each computing client.

## 2.7 Logs and databases

Once a query is issued, you have a directory named `.paralite-log` in home directory. You can find all logging information there. When you firstly use a database, for instance, when you issue the next SQL:

```
$ paralite test.db "create table x (a int, b) on hongo[[100-101]] chunk 2"
```

A `test.db` is firstly created to store all metadata info and 3 other files named `test.db-hostname-partitionID-chunkID` are created on hongo100 and hongo101 respectively in the same directory with `test.db`.

You can also specify the directories for logs and temporary files by the option `-log` and `-temp` respectively:

```
$ paralite test.db "create table x (a)" -log /log/dir -temp /temp/dir"
```

# 3 Using ParaLite in Real-World Workflows

TODO