ZimaDB - User Documentation

Vilém Zouhar, Petr Chmel

January 2019

1 General

ZimaDB supports a subset of the SQLite functionality. The program can be launched as an executable. It expects the input to be on the stdin. If a command line argument is passed to the executable, then it is treated as a filename to be opened. Otherwise the user is expected (according to the SQLite syntax) to open the database file using the *.open* command.

1.1 Data types

The following data types are supported by ZimaDB.

- BOOLEAN
- (UNSIGNED) INT, TINYINT
- VARCHAR(n) (if no size specified, 32 is used)
- DOUBLE

1.2 Allowed operations

The following operations are supported by ZimaDB.

- SELECT
- INSERT
- UPDATE
- DELETE
- CREATE TABLE
- DROP TABLE
- TRUNCATE TABLE

Aggregation functions are not permitted as well as nested SELECT statements. This is because the implementation is beyond the size of this project and requires advanced knowledge of databases. The JOIN ON operation can be simulated using selects on multiple tables. We don't support the table dot notation, so any multiple select must be done on distinct tables.

1.3 Other modifiers

Several other modifiers are supported as well:

- PRIMARY KEY attribute
- ASC, DESC attribtute
- wildcard asterisk (SELECT)

1.4 Meta commands

Similarly to the SQLite system, ZimaDB provides a set of meta commands.

1.4.1 .about

Display info about this project.

1.4.2 .debug [on—off]

Turn debug on/off. Omit arguments for info.

1.4.3 .dump TABLE*

Not implemented. Would output SQL code to dump a whole table (use regex for table name).

1.4.4 .exit

Exits ZimaDB (Ctrl-D can be also used)

1.4.5 .help

Shows help for ZimaDB meta commands.

1.4.6 .open DATABASE

Open a database file (ends with .zima).

1.4.7 .schema TABLE*

Not implemented. Would output SQL code to replicate a table schema (use regex for table name).

2 Examples

2.1 Employees database

We will create a simple employee database, which will keep track on the person's *id, name, age* and *monthly_salary*. To do so we launch **ZimaDB** and open a new database file called *emp.zima*. Even though it is generally considered a bad practice (softwareengineering.stackexchange.com/questions/85764/), we will use a three character prefix *emp_* for each of our column name.

```
.open employees.zima
CREATE TABLE emp (emp_id INT, emp_name VARCHAR(255), emp_age INT);
```

ZimaDB will respond with an information that the file was opened and table created. Running an empty *SELECT* query will tell us that the table contains no data.

```
SELECT * FROM emp;
```

We can insert the actual data using the **INSERT** query. The column order can be changed. Ommiting a column will make it set to 0 for numbers and an empty string for text fields. Mathematical expressions can also be used.

```
INSERT INTO emp (emp_id, emp_name, emp_age) VALUES (0, "Jarda", 20);
INSERT INTO emp (emp_age, emp_name) VALUES (25, "Milan");
INSERT INTO emp (emp_age, emp_name) VALUES (20+15, "Lenka");
INSERT INTO emp (emp_age, emp_name) VALUES ("50"-13, "Jarmila");
```

A wildcard **SELECT** would now return the whole content of the table. We can alter the **SELECT** query to compute eg. the age in months or to multiply person's id and their age.

```
SELECT *, emp_age*12, emp_age*emp_id FROM emp;
```

```
emp_age|emp_id|emp_name|emp_age*12|emp_age*emp_id
-------|------|--------|----------|--------------
20     |0     |Jarda   |240.000000|0.000000
25     |1     |Milan   |300.000000|25.000000
35     |2     |Lenka   |420.000000|70.000000
37     |3     |Jarmila |444.000000|111.000000
```

Now we create a different table with employee security level.

```
CREATE TABLE sec (sec_id INT, sec_level TINYINT);
INSERT INTO sec (sec_id, sec_level) VALUES (0, 10);
INSERT INTO sec (sec_id, sec_level) VALUES (1, 5);
INSERT INTO sec (sec_id, sec_level) VALUES (2, 5);
INSERT INTO sec (sec_id, sec_level) VALUES (3, 50);
```

Running a combined **SELECT** will return a cross product of these tables. We can add a binding condition (just as a regular **JOIN ON** would work) and display security level for each user.

```
SELECT * FROM emp, sec;
SELECT * FROM emp, sec WHERE emp_id = sec_id;
```

```
emp_age|emp_id|emp_name|sec_id|sec_level
-------|------|--------|------|---------
20     |0     |Jarda   |0     |10
25     |1     |Milan   |0     |10
35     |2     |Lenka   |0     |10
37     |3     |Jarmila |0     |10
20     |0     |Jarda   |1     |5
25     |1     |Milan   |1     |5
35     |2     |Lenka   |1     |5
37     |3     |Jarmila |1     |5
20     |0     |Jarda   |2     |5
25     |1     |Milan   |2     |5
35     |2     |Lenka   |2     |5
37     |3     |Jarmila |2     |5
20     |0     |Jarda   |3     |50
25     |1     |Milan   |3     |50
35     |2     |Lenka   |3     |50
37     |3     |Jarmila |3     |50
Rows: 16

emp_age|emp_id|emp_name|sec_id|sec_level
-------|------|--------|------|---------
20     |0     |Jarda   |0     |10
25     |1     |Milan   |1     |5
35     |2     |Lenka   |2     |5
37     |3     |Jarmila |3     |50
Rows: 4
```

Finally we can extract the ages of employees with their security level higher or equal to 7.

```
SELECT emp_name, emp_age FROM emp, sec WHERE (emp_id=sec_id) AND (sec_level>=7);
```

```
emp_name|emp_age
--------|-------
Jarda   |20
Jarmila |37
```