MINIDB – A MINIMAL DATA BASE MEMORY MANAGER

14/05/2014
PROINNOVA PROJECT ITESO - ORACLE
Rafael Alcaraz Mercado.

DR. RAÚL CAMPOS RODRÍGUEZ, ITESO PROJECT LEADER,

MARCELO LEÓN. ORACLE PROJECT LEADER.

ABSTRACT

In order to have working code in which we can make tests about how to use SIMD instructions on a database, we have decided to implement our minimal data base memory manager. We have called this MiniDB. MiniDB is a simple implementation of a column-wise storage for a database so we can make tests on it to show the concept of using SIMD instructions to accelerate procedures in a database.

INTRODUCTION

A database is an organized collection of data. The data is typically organized to model relevant aspects of reality in a way that supports processes requiring this information. A database management system (DBMS) is specially a designed application that interacts with the user, other applications, and the database itself to capture and analyze data. A general-purpose **database management system (DBMS)** is a software system designed to allow the definition, creation, querying, update, and administration of databases.

Depending in the DBMS we are talking about or even developing, we can have two major ways of organizing data: row-wise storage and column-wise storage.

SSN	Name	Age	Addr	City	St
101259797	SMITH	88	899 FIRST ST	JUNO	AL
892375862	CHIN	37	16137 MAIN ST	POMONA	CA
318370701	HANDU	12	42 JUNE ST	CHICAGO	IL

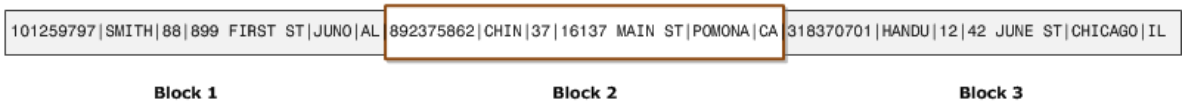

Figure1. A row-wise storage representation.

SSN	Name	Age	Addr	City	St
101259797	SMITH	88	899 FIRST ST	JUNO	AL
892375862	CHIN	37	16137 MAIN ST	POMONA	CA
318370701	HANDU	12	42 JUNE ST	CHICAGO	IL

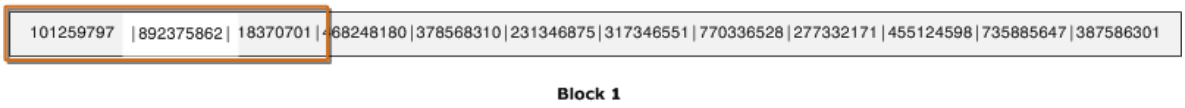

Figure2. A column-wise storage representation.

As we can see on Figure1, a row-wise storage have stored in consecutive addresses of memory, having a complete row per block. However, as in Figure2 we can clearly see that column-wise storage have all data from a column in consecutive addresses of memory having a block of memory per column.

In order to use SIMD instructions in a full potential, or at least what we are expecting, is to implement a minimal DBMS with column-wise storage, since SIMD instructions work on consecutive data memory addresses. Working with chunks of data multiple to the SIMD instruction's register's byte size, this storage layout make it easier to use.

RELATED WORK

We haven't found yet a database that uses SIMD to exploit its potential to accelerate common procedures within it. Regarding our work, we haven't done something like this before. Everything done so far is "brand new", putting it in simple terms.

OBJECTIVES

The main objective of this project is to use SIMD instructions instead of scalar operations in order to improve performance on common database procedures such as scans, aggregation functions and joins. This performance would be measured in terms of time ignoring the parsing time and execution time of MiniDB specific features and other specific code, just putting attention to the SIMD implementation. If we have a notable performance boost, this MiniDB will be used to present the core idea of using SIMD on a real commercial database in a simple way.

EXPERIMENTAL DETAILS OR THEORETICAL ANALYSIS

In order to develop this MiniDB, we had to define the general Table layout in memory so all the remaining code work based on this specifications. In the following text we'll describe this layout and the commands available in MiniDB that manipulate this table memory representation.

First of all, MiniDB just support the following data types:

- char - 1
- int - 2
- float - 3
- double - 4

Defined in this C enumeration:

```
enum {
  start,    //000
  char_t,   //001
  int_t,    //010
  float_t,  //011
  double_t, //100
  end       //101
};
```

To avoid specific detail implementations, we'll explain the table layout in a simple way.

A table in MiniDB is defined by its table type, which contains the table name, column count, row size in bytes, column names and a pointer to header type.

```
//Table type
typedef struct table_t{
  header_p hdr;
  char name[STR_MAX];
  int columnCount;
  int row_size;
  char (*column_names)[STR_MAX];
}table_t;
```

A header type has a table descriptor, which mainly purpose is to describe the column data types and in which order they are stored in memory. It contains a pointer to a page type.

```
//Header type
typedef struct header_t{
  page_p firstPage; //Pointer to meta-data
  char descriptor[1]; //Table column data types descriptor
}header_t;
```

A page type have a pointer to a next page type, meaning the first page type referenced at the header is the head of a single linked list that contains all the data distributed in pages.

```
//Page type
typedef struct page_t{
  page_p nextPage;
  int bitmap; //32 rows per page
  char data[1];
}page_t;
```

On a page we have all the data distributed in consecutive memory addresses in bytes, but their interpretations completely depends on the descriptor found at the header. For example, if we have a table with the following layout:

Column1 (char)	Column2 (int)
A	34
B	35
C	36

The page would have a bitmap at 7 (0000 0000 0000 0000 0000 0000 0000 0111 in binary) which means that the first three elements of each column data are reserved for the first three rows. The data would like something like this:

A	B	C	\0	\0	\0	\0	\0
\0	\0	\0	\0	\0	\0	\0	\0
\0	\0	\0	\0	\0	\0	\0	\0
\0	\0	\0	\0	\0	\0	\0	\0
34	35	36	\0	\0	\0	\0	\0
\0	\0	\0	\0	\0	\0	\0	\0
\0	\0	\0	\0	\0	\0	\0	\0
\0	\0	\0	\0	\0	\0	\0	\0

A row is retrieved using pointer arithmetic.

Figure3 gives an overall picture of this memory layout.

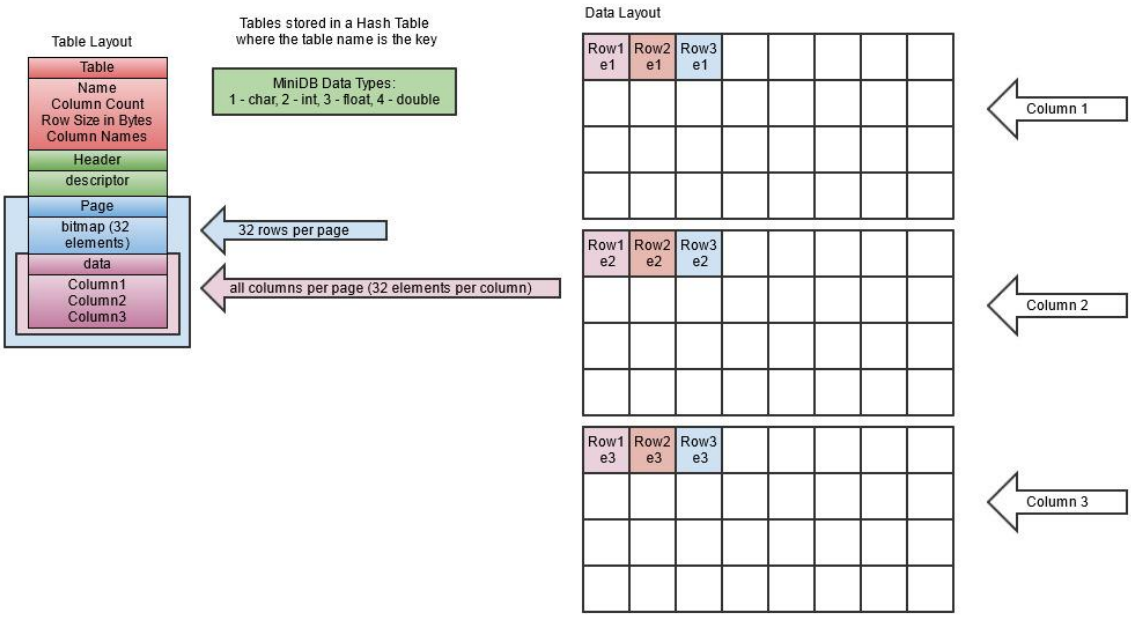

Figure3. MiniDB Memory layout.

Regarding what we can do with this memory representation, the question is best answered with the inclusion of “TableFunctions.h”. This header defines the function prototypes which work on the previously explained structure.

All available commands:

'createTable' - "creates a new table with the given table name. Eg. createTable Table_name"

'deleteTable' - "deletes a table that match the given name. Eg. deleteTable Table_name"

'addColumn' - "adds a Column to the given table with the given Column Name and data type. Data Types(1 = char, 2 = int, 3 = float, 4 = double). Eg. addColumn Table_name Column_name 2"

'insertRow' - "inserts a new row to the given table with the following format: (element,)+. For example, if we want to insert a new row to a table wich layout is a char, char, float we would type: 'm,b,12.3,' without the single commas. Eg. insertRow Table_name m,b,12.3"

'clear' - "clears the actual screen. Eg. clear"

'printTable' - "prints all the rows of a given table. Eg. printTable Table_name"

'mentionTables' - "print the names of all existent tables. Eg. mentionTables"

'saveTable' - "save to disk the given table as a flat file named after the table name without extension. Eg. saveTable Table_name"

'loadTable' - "loads to the actual process the given table. If table's flat file is not found, it doesn't load anything. Eg. loadTable Table_name"

'saveDB' - "save to disk a zip file that contains all tables' flat files with the given name. Eg. saveDB DB_name"

'loadDB' - "loads to the actual process the given db unzipping and loading it's content with the given name. Eg. loadDB DB_name"

'connectTo' - "connects to the given table, so in the commands <addColumn, insertRow, printTable, saveTable> you are not prompted for the table name. Eg. connectTo Table_name"

'disconnect' - "disable current table connection feature. Eg. disconnect"

'query' - "open query parser. Waits for a typed query and it's parsed and executed afterwards (read from stdin). Eg. query select * from Table_name;"

'loadQuery' - "loads a file with one or more queries in it and parses/execute each one. Eg. loadQuery query.qry"

'exit/quit' - "deletes all existent tables (Complete DB) and close MiniDB process. Eg. quit"

'help' - "show help text. Eg. help"

The query engine implemented for MiniDB supports select query, drop query, insert query and create query. In other words, you can make basic query about the data stored in tables, create tables, insert rows to tables, and drop tables.

Figure4 shows the basic flow for a select query.

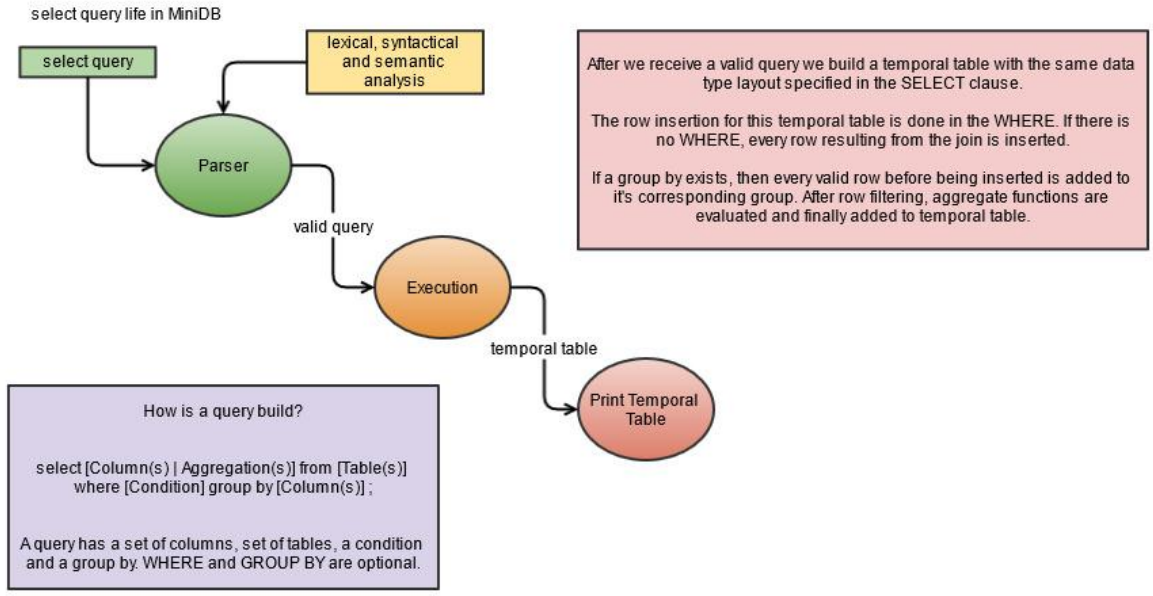

Figure4. Life of a Select Query

Figure5 makes a graphical representation of the overall MiniDB Arquitecture.

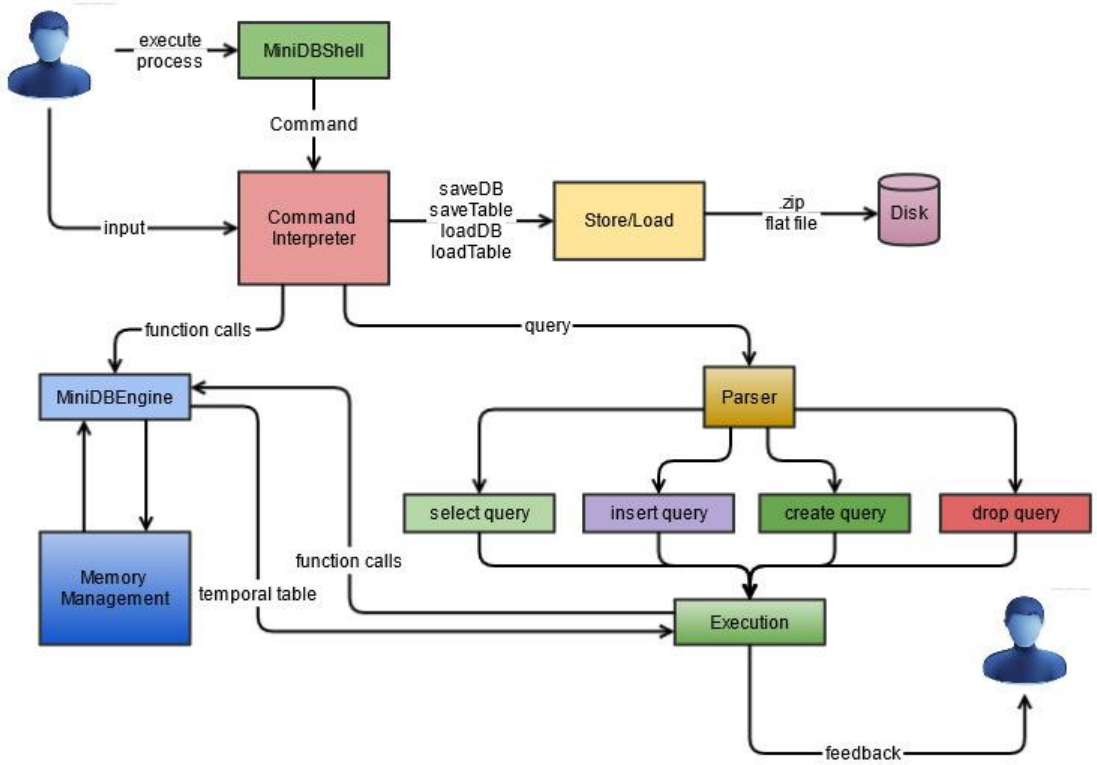

Figure5. MiniDB Arquitecture

For a complete reference for all available commands, query grammar and code references please go to the Code Documentation.

RESULTS

The result from this work is the ongoing development of MiniDB with some features already functional. The idea is to continue from where we left to start implementing SIMD instructions.

DISCUSSION

These current version for MiniDB is just the introduction to our database memory manager that will be used to benchmark SIMD usage on typical table operations. The actual state of MiniDB have a fully functional query engine that supports basic querying and aggregate functions. All of the supported functionality for the query engine can be seen in the query grammar available at the Code Documentation.

The next steps are to make an analysis on how is it possible to implement SIMD instructions in the query engine, mainly because this module of a database is the one in charge of executing the most common operations and tasks. Since the query engine is what makes possible to the user interact with the data stored at the database, it's important to improve the query response times.

CONCLUSIONS AND SUMMARY

The fully implementation of this database memory manager will serve us as a database simulator to avoid making changes in real and running commercial databases in order to showcase how SIMD instructions can be exploited to accelerate and even outperform previous implementations of DBMS. We hope this shows our point and help us achieve a kind of generic core idea on how to use them.

REFERENCES	
Makefile creation	**http://www.gnu.org/software/make/manual/make.html**
C Preprocessor	http://gcc.gnu.org/onlinedocs/cpp/
C Memory Management	http://en.wikibooks.org/wiki/C_Programming/Memory_management