**Design Document for CS165 Project : Memory Optimized Column Store**

*Introduction:*

This document details the design approach for the column store. It begins by presenting the overall design and then details the approach for some critical features of the system that have been implemented so far. It ends with analyzing the cost of implementing such a system in low level language like C vs implementing the same thing in a Strongly typed language like Scala.

*Design:*

At a high level the system design follows a typical client/server architecture, where the server handles the client requests. The following figure captures the overall design ( with important source files )

Server

**server.c**

**Other utilities (display format etc)**

**Client (client.c)**

Parser (parser.c)

dml handler (dml.c)

Request

**handles.c**

Response

**utils.c**

ddl handler (db.c)

**tuple.c**

col\_scan

(query.c)

sync.c

bplus\_tree (bplus\_tree.c bplus\_tree\_utils.c)

sorted

(sort.c)

hash\_join

(hash.c)

aggregator

(aggregator.c)

join

(join.c)

Index\_scan

(index.c)

load.c

select

(query.c)

*Life cycle of a request:*

The client and the server talk over a socket connection. When a request a made from the client it is

1. First parsed by the parser – to validate the valid and identify the type of request. Request from the client can be of any of the following types
   1. DDL – create table, index, shutdown the server
   2. DML – insert, update, select, join select, delete
2. Upon validation the based on the type of request it is either handled by the ddl handler or a dml handler.
3. A DDL request changes the state of the database – requesting in either creation of a new database, table or index.
4. A DML request – usually is made to either insert, select, update or delete.
   1. Select queries : if the column that is selected has an index , it is used during the query operation. Otherwise data is selected by doing a column scan.
   2. Insertion – Again insertion can result in update of the index.
5. Each request receives a response from the server, upon which the client can make subsequent requests.

*Indexing functionality:*

There can be two types of indices that are supported by the system

(1) Clustered Index : which is the primary index and there can only be one of these.

(2) Secondary Indicies : can be defined on any number of cols of the table.

These indexes can be either in the form of

(i) sorted – implemented as a sorted array of positions . Note this array is a separate array , and does not alter the order of the original positions of the indexed column.

(ii) Bplus tree – The tree is modeled such that the index node – is an array that holds values, while the leaves of the tree have both values and positions.

*Bulk Loading functionality:*

When a server shutdown message is received, the state of the system is saved in binary files. These files are then loaded on subsequent start up of the server process.

The files are designed as follows and are saved in directory cs165\_db.

1. schema.dat : It starts with a MAGIC word ( cs165). This will be used during load to validate the schema file. It stores the metadata such as , database name, number of tables, each table name, and number of columns per table
2. tblname.dat : Each table is written to a file – named after the table name. It

has information abt the tbl, the column length and the actual col data and if that column has index – the type of index.

1. tblIndex.dat – Any indices on the table ( across all it columns ) are stored in

file named tblname\_index.dat. The files contains, name of the column, type of index and index data itself.

*Hash Joins :*

The system supports hash join functionality. The hash table used a Prime number - such as 1021 – to create initial set of buckets. This allows for more even distribution of the data across buckets. In case when the bucket needs to be expanded – it is increased to the next prime number. All of the current data is rehashed.

In order to minimize the rehashing ( and size increase ) an estimate is made based on the data, and an optimal size which can holds all of this without creating long chains is created.

**Thoughts on how C compares with Scala:**

Below are some of my thoughts on how programming in C compares with something that is functional and strongly typed like Scala

My view is that when writing code in C , programmer has a lot of power to control every byte allocation and data movement. So smart developers can use to build really powerful systems – creating less garbage and reducing memory footprint.

But this comes with a cost – of long(er) development cycles perhaps – due to Segmentation faults, invalid reads/write encountered during run time. Using a strongly typed language such as Scala can alleviate some of this cost. The type safety

ensures that if the code compiles it will guarantee( 99%) that the code will run fine. Also language constructs allows for cleaner and leaner code base.