Proposed Storage Engine (this was done last year in a another project which got terminated!)

The proposed storage engine allocates memory as and when new rows are inserted. Memory for each row is allocated based on the size of the respective row. Therefore, rows can have varying length. In other words, rows can have columns with variable-length data-types like VARCHAR, TEXT, BLOB etc. The storage engine must keep track of all the inserted rows and for that purpose a simple table will be used to manage all the rows. As new rows are inserted, it is just appended at end of the table. Each entry in the table consists of a pointer to a row and a metadata field. The table helps for internal indexing and makes consecutive row retrieval faster. Thus, the standard APIs like index\_next(), index\_prev(), rnd\_next() and rnd\_prev() will be using this table to read rows. It is useful in doing joins.

The most important benefit of using a table is the fast insertion. However, the size of the table will be fixed during allocation and will eventually face overflow issues. There are two methods to resolve this issue. First, a new larger table is created and the pointers are relocated from the old table into the new table. However, there will be overhead of regular table growth and relocation. Another method is to create a new table of similar size and add the new pointers to this table. As requirement grows more tables will be appended. The tables are linked with each other forming a linked list of tables. The tables will be in equal sized chunks. Each chunk will hold a fixed number of entries, say 100 (=CSIZE). The last entry in each chunk will point to the base of the next chunk. To access row 250, we will access the last entry of chunk 0 to get a pointer to chunk 1. Then the last entry of chunk 1 will be used to get to chunk2 and the 50th entry in chunk 2 will be the entry for row 250. It needs mention here that a random access to a given row will not be based on the row number. Rather, random access to a row will be through an index which will store pointers to rows directly. Access to the table will usually be by using routines such as rnd\_next().

The storage engine must also manage entries or rows that are deleted. Therefore, a bit in metadata field will be reserved to denote deleted rows. The bit is set for deleted row. The memory reserved by the deleted pointer is then freed. Most commercial databases has high insertion rate and thus our focus is mostly on faster insertions.

Indices will also be created based on keys provided by users. These indices will store pointers to the table entry for the respective row.

The index structure can be a B+ tree, a binary tree, or a hashed table. We will explore the effectiveness of different indexing structures through experiments.

ROW 1

ROW 2

ROW 3

ROW 4

ROW N

ROW N +1

Tree index

Table 1a

Table 1

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Pointer to root of table index and first entry of the table (Table 1 above)

Top Hash Table

\_row().

Should the rows themselves be stored in two levels? Each field will have the same length: value, if the value fits. For varchar, we will store a length, and a pointer to memory containing the actual varchar. This scheme will help in quickly accessing a field in a row. But it will mean space overheads, and more complex memory management.

The trees and tables will be in RAM only. In NVM, the rows will be stored (one level or two), and a has table to access a row based on the primary key. Will be used during re-boot time and during recovery to build the RAM data structures.

Please consider this alternative too.