Design for new Hibase

3rd October 2003

Contents

1 Aims

Aim is to create a new implementation of hibase that addresses the problems of scale of databases handled, persistence, and the provision of a standard SQL interface.

It should be implemented in such a way that it will run on either 32 bit or 64 bit architectures without code change. It should store the relations in some form of stable virtual memory, and it should be capable of accepting SQL queries.

2 Store design

Store will be in a heap that is held in memory mapped files to handle the problem of persistence. New 64 bit architectures allow these to be as big as we are likely to need. Since one can not always guarantee that a

memory mapped file will be located in the same position in the address space, all pointers within the heap will be relative to the heap base.

Pointers to heap objects will be termed PIDs or Persistent IDentifiers.

The heap will be garbage collected. At run time all active PIDs being manipulated by the database code must be stored on Pstacks, stacks of PIDs whose size and location are known to the garbage collector. In additon to this there will be one dedicated root pointer from which all database objects descend.

2.1 PStack Interface

In order to ensure that all pointers can be found by the garbage collector PIDs will not be stored in ordinary C variables. Instead PIDs must reside on Pstacks, pointer stacks. Operations of higher layers of software will be expressed as a sort of reverse polish code on the current Pstack.

2.2 Heap call interface

2.2.1 Call down interface

void openHeap(char * filename) Opens the heap by mapping the named file into the address space.

void closeHeap() Flushes buffers and closes the memory mapped file.

void halloc(Pstack *s, int k, char t) This allocates a heap block containing $2^k - 2$ bytes of usable store, with type code *t*. Note that the addresses of pre-existing heap blocks may change following a halloc call due to the heap having grown and having had to be moved within the cpu address space. The PID for the block will be pushed ont the Pstack pointed to by *s*.

char * getAddr(Pstack *s) maps the PID on the top of the Pstack to an address, the address returned is after the header byte and type code described in section 2.3. Note that addresses obtained by this call are not guaranteed over calls to halloc.

unsigned char getHeader(Pstack *s) This returns the header described in table 1 of the heap block on top of the Pstack.

long getSize(Pstack *s) This returns the size in bytes of a block not including its header and type code.

unsigned char getType(Pstack *s) This returns the type byte for the heap on top of the Pstack.

void duplicate(Pstack *s) Duplicates the top of the pstack.

void drop(Pstack *s) Drops the top item from the pstack.

void load(Pstack *s, int n) Loads the n from top item on the pstack onto the top, duplicate(s)=load(s,0).

2.2.2 Callup interface

The heap can call the layers above to get information required for garbage collection.

void appStack(void (*marker)(PID)) Applies the function *marker* to every PID on the current Pstacks.

2.3 Physical organisation

The heap will be organised as a 1Kb administrative header followed by the heap proper. Within the heap a buddy system will be used. Blocks of store will be allocated in powers of 2 bytes. Each block will be described by a 1 byte allocation header with the format shown in table 1.

Following the header byte will be a 1 byte type code, which will define the structure of the data held in the block.

bit position	7	6	5-0
	mark bit	free bit	k

mark bit This is set true during the mark phase of a mark sweep garbage collection to indicate that a block is reachable.

free bit This is set if the block is not currently in use

k This encodes the size of the block as 2^kbytes. Size includes the header and type bytes.

Table 1: Format of a header byte

Table 2: Length information in type codes.

type codes	element length
`xxxx x000`	1 bit
`xxxx x001`	2 bits
`xxxx x010`	4 bits
`xxxx x011`	8 bits
`xxxx x100`	16 bits
`xxxx x101`	32 bits
`xxxx x110`	64 bits
`xxxx x111`	reserved

3 Vector store

3.1 Element sizes

All objects held on the heap will be vectors. That is to say they will be sequences of elements with each element in a vector having the same number of bits in it. The bit width of the vector will be encoded in the bottom 3 bits of the vector's type code. The permited bit widths are encoded as shown in table 2.

Note that the encodeings in the table mean that there will be a wastage of approximately 25% in individual elements. This is regarded as worth having to enable use of efficient access to byte aligned quantitites of length above 8 bits. One code is reserved and may in future be used to provide another length if efficiency considerations demand it. The actual length that will produce the greatest reduction in database size can only be determined after some period of use of the system, snce it depends on the statistical properties of domain sizes.

3.2 Vector Length Codes

A vector will be preceeded by a length code saying how many elements are present in the vector. Let the maximum number of elements that the vector could have V_{max}, given the element size and the heap block size within which it is embeded. This is determined by the equation 1.

$$V_{max} = \frac{8 \times H_{len} - 16 - L_{bits}}{E_{bits}} \tag{1}$$

where H_{len} is the length in bytes of the heap block, E_{bits} is the number of bits in each element and L_{bits} is the number of bits in the vector length code. This in turn is determined by equation 2.

$$L_{bits} = \begin{cases} 8 & if \quad V_{max} < 2^8 \\ 16 & if \quad 2^8 \leq V_{max} < 2^{16} \\ 32 & if \quad 2^{16} \leq V_{max} < 2^{32} \\ 64 & if \quad V_{max} > 2^{32} \end{cases} \tag{2}$$

The solutions to these equations can be tabulated in a 8×64 array encoding the values of L_{bits} and V_{max} as a function of element size and heap block length. As an illustration of the structure of a vector see Figure 1.

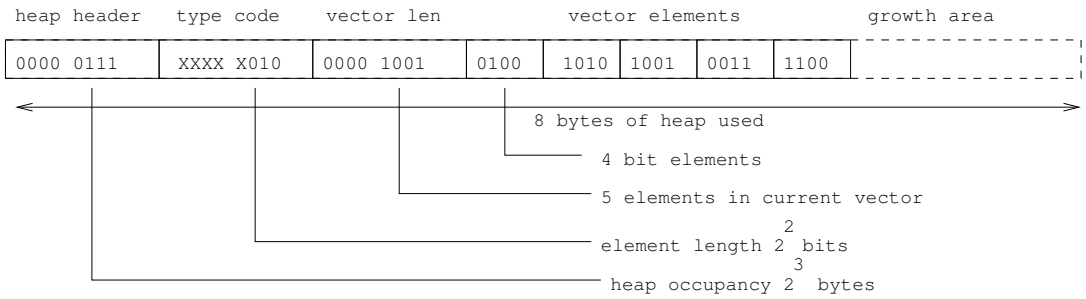

Figure 1: How a 5 element vector of 4 bit numbers would be stored on the heap.

Table 3: Specialisation of the vector codes.

Qualifier	Length	C style meaning
0000 0	000	Boolean
0000 0	001	unsigned 2 bit integer
0000 0	010	unsigned 4 bit integer
0000 0	011	unsigned 8 bit char
0000 0	100	unsigned 16 bit short
0000 0	101	unsigned 32 bit int
0000 0	110	unsigned 64 bit long
0000 1	011	signed 8 bit char
0000 1	100	signed 16 bit short
0000 1	101	signed 32 bit int
0001 0	101	32 bit PID
0001 0	110	64 bit PID
0001 1	101	32 bit float
0001 1	110	64 bit double

3.3 Specialised Vector Types

Vector elements may be of the following basic types:

1. Booleans - one bit elements.
2. Unsigned integers of lengths between 2 and 64 bits.
3. Signed integers of lengths between 8 and 32 bits
4. Floating point values of 32 or 64 bits.
5. PIDs of either 32 or 64 bits - pids are also unsigned integers.

The resulting combinations can be coded using a total of 5 bits in the type code, a redundant coding given that there are only 13 combinations. These are listed in table 3. Note that vectors containing PIDs have 3 unused bits in their type codes allowing for these to be further refined into specific types appropriate to relational data.

3.4 Vector Interface Calls

Semi-applicative updates

Note that since updates operations may cause vectors to be extended or widened, which may require them to be relocated, the interface to the vectors is semi-applicative. That is to say, all updates return a new PID, which will ususally be the PID of the original vector, but may occasionally be different. Thus whenever a vector that is pointed to by a larger data-structure is updated, the pointers to it in the larger data-structure should also be updated

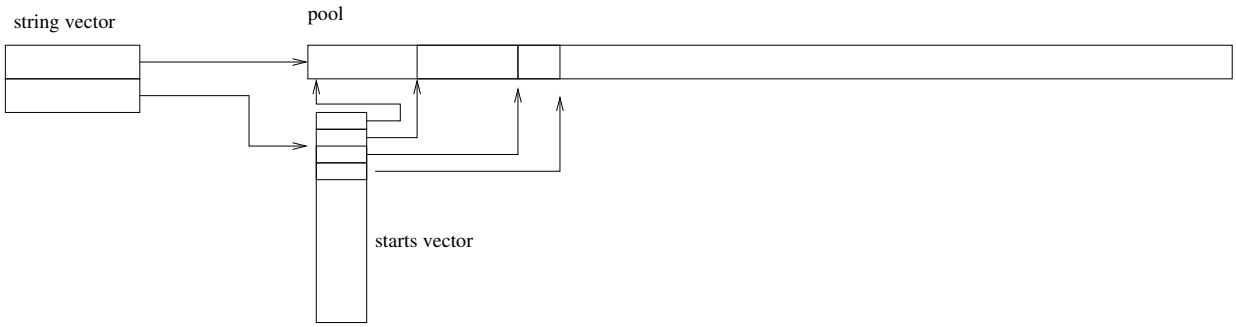

Figure 2: Layount of an uncompressed string vector.

void makeVector(Pstack *s, long length, unsigned char typecode) Creates an unitialised vector of specified intial length, and specified type code. The PID for the vector will be pushed onto the Pstack.

void setPidAt(Pstack *s, long n) The top pid on the pstack is written into the n'th position in the vector pointed to by the second from top position on the pstack. Indexes start from 0. The top two items on the pstack are then popped, and a PID to the, possibly extended, vector is placed on the pstack.

void getPidAt(Pstack *s, long n) Replace the top item on the pstack with the nth pid of the vector.

void setNumAt(Pstack *s, double num, long n) The number num written into the n'th position in the vector pointed to by the top position on the pstack. Indexes start from 0. If need be, the vector is widened to handle the larger number being written. The top item on the pstack is then popped and a, possibly new, PID to the vector is pushed onto the stack.

double getNumAt(Pstack *s, long n) The number in the nth position of the vector pointed to by the top of the pstack is returned as a double, whatever its compressed representation. The pstack is then popped. The result of an out of bounds fetch it undefined.

long getLength(Pstack *s) Returns the number of elements in the vector. The pstack is then popped.

4 String vectors

In addition to vectors of numbers and pointers, a basic requirement will be to store vectors of character strings. We will initially provide uncompressed arrays of strings. It may prove useful to provide compressed arrays of strings later. In order to provide suitable internationalisation we will allow Unicode characters in strings.

The implementation of an uncompressed string vector is shown in figure 2. Note that a string vector is implemented as a pool in which the characters are stored, and an index array pointing into the pool. The order of the index array need not correspond to the order of the entries in the pool. Pool entries will be null terminated. Note that this structure is optimised for reading. No random write operator is provided.

4.1 String Vector Access Functions

void makeStringVector(Pstack *s, long length) Creates an unitialised vector of specified intial length. The PID for the vector will be pushed onto the Pstack.

void setStringAt(Pstack *s, short *str, long n) The null terminated Unicode string str is written to position n of the vector pointed to by the top position on the pstack. The pid to the string vector remains on top of the stack. This will typically involve an exension of the pool.

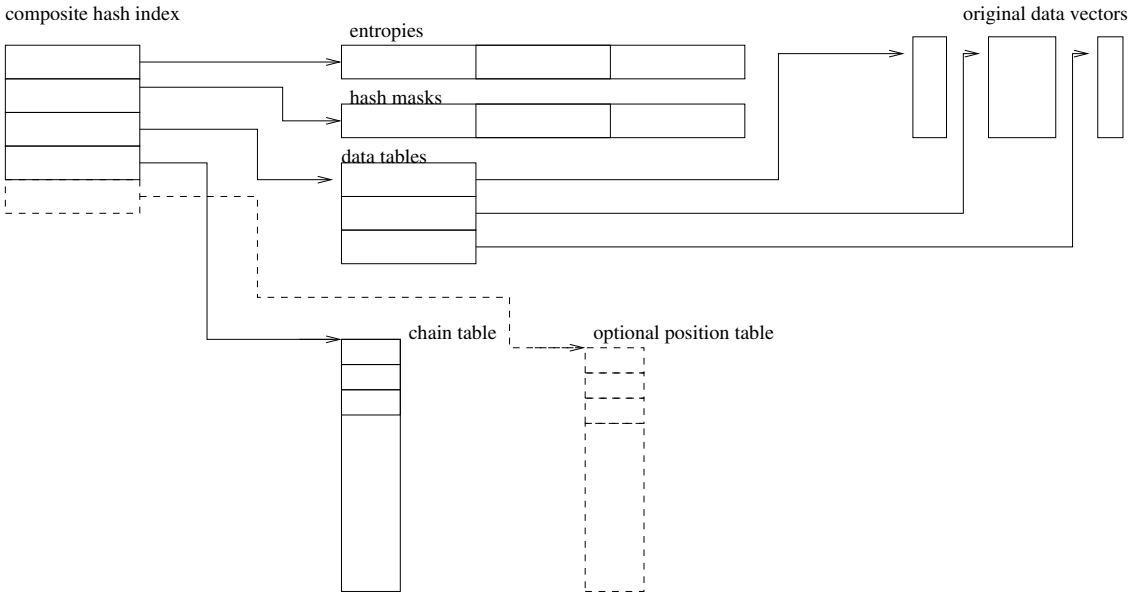

Figure 3: Structure of a composite hash index.

void getStrAt(Pstack *s, short * str,long n,long max) The string in the nth position of the vector pointed to by the top of the pstack is returned in str as a as a null terminated Unicode string, whatever its compressed representation. The pstack is then popped. The parameter max specifies the size of the buffer into which the string can be placed.

long getLength(Pstack *s) Returns the number of strings in the vector. The pstack is then popped.

5 Hash indices

5.1 Purpose

In the original Hibase hash tables were used in three contexts:

1. Dictionaries
2. Master indices to relations
3. Auxilliary indices to relations

In order to prevent code duplication a single datatype is provided for hash tables which can be utilised for all these purposes. It is illustrated in figure 3. In the three uses above for master indices the order in which the data is stored in the relation can be the same as the order in the hash table. For dictionaries which index an array of strings, this will not generally be the case, nor will it be the case with auxiliarry indices. We thus provide two variants of the hash tables dependent upon whether they contain an optional position table (shown in dashed lines in the diagram.

5.2 Hash table fields

Entropies this is the 0th field of the hash table structure, a PID that points to a vector holding the entropies of each column being indexed..

Masks this is field 1 of the hash table structure, a PID that points to a vector holding a bitmask specifying for each indexed column which bits from it are to be used to form the composite hash key. The bits used in different columns must be mutually exclusive.

DataTables a two dimensional array implemented as an array of PIDs to the column vectors of the data

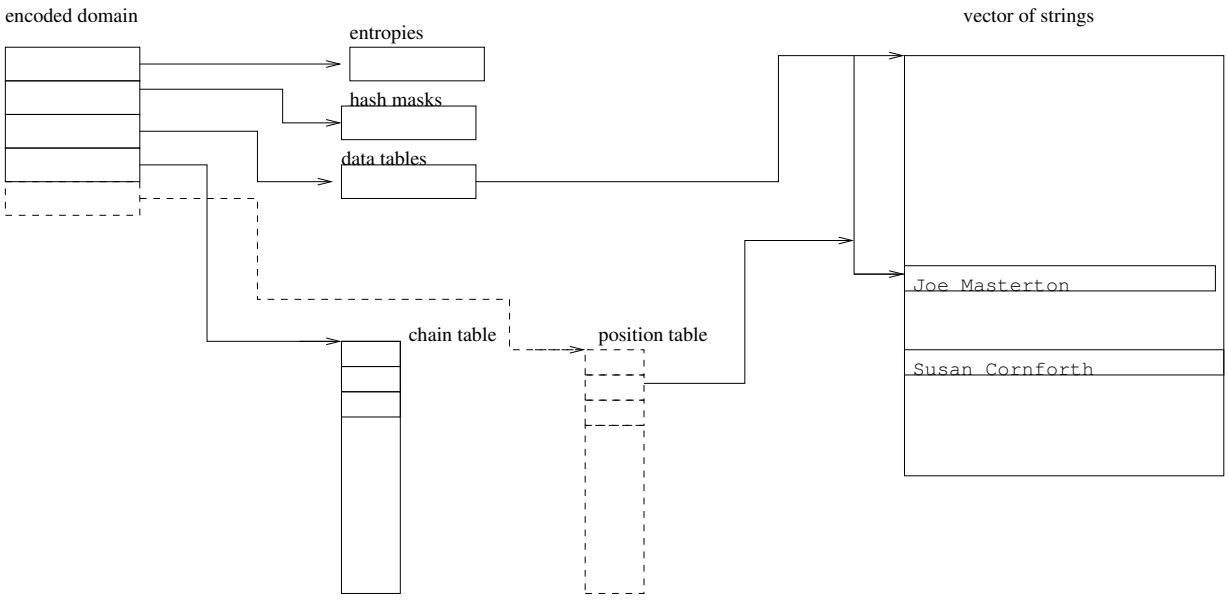

Figure 4: Representation of encoded domains

ChainTable We assume that if the hash table is of length $2m$ and if the hashcode obtained is h then the first probe will be at $h \mod m$ if this turns out to be a collision, then the location pointed to by $ChainTable[h \mod m]$ is probed and so on down the chain until either a hit or a null chain table entry is found. The overflow area of the chain table will be located at indices $m..2m-1$.

Positions This is an optional table used in the cases where the data table can not be ordered according to the hashcodes used here. This would be the case for a secondary index. In that case, the location of the actual row in the relation is not given by the h but by $Positions[h]$.

6 Encoded Domains

An encoded domain is represented by an indirect hash table indexing a vector of strings such that the vector of strings contains no repeated entries.

The output of a lookup here returns the hash position table entry.

7 Columns

8 Relations

9 SQL Interface

10 Composite type code table

Qualifier	Length	Represents
0000 0	000	Boolean
0000 0	001	unsigned 2 bit integer
0000 0	010	unsigned 4 bit integer
0000 0	011	unsigned 8 bit char
0000 0	100	unsigned 16 bit short
0000 0	101	unsigned 32 bit int
0000 0	110	unsigned 64 bit long
0000 1	011	signed 8 bit char
0000 1	100	signed 16 bit short
0000 1	101	signed 32 bit int
0001 0	101	32 bit PID
0011 0	101	Uncompressed String Vector
0101 0	101	Direct hash table
0111 0	101	Indirect hash table
0001 0	110	64 bit PID
0011 0	110	Uncompressed String Vector
0001 1	101	32 bit float
0001 1	110	64 bit double