Department of Informatics, University of Zürich

Vertiefungsarbeit
A Database Systems's Storage Layer Implementation

Jonas Schmid

Matriculation number: 06-908-677

Email: jonas.schmid@uzh.ch

19. September, 2011

University of Zurich
Department of Informatics

1 Introduction

This paper deals about the storage layer of a database management system. A storage layer consists mostly of three main parts, namely the file manager, buffer manager and the storage manager. Figure 1.1 shows an overview over the interaction of the three managers of the storage layer. Important to mention is that the file manager does not know about the content nor structure of the data it writes, it is just some data of its point of view. On the other side does the storage manager not know how the file manager stores the data, namely splitting up the data in blocks as explained later.
This report can be viewed as a summary of my implementation of the storage and file manager, it describes how I organized the files, how I deal with fixed and variable length data types and the block buffering.

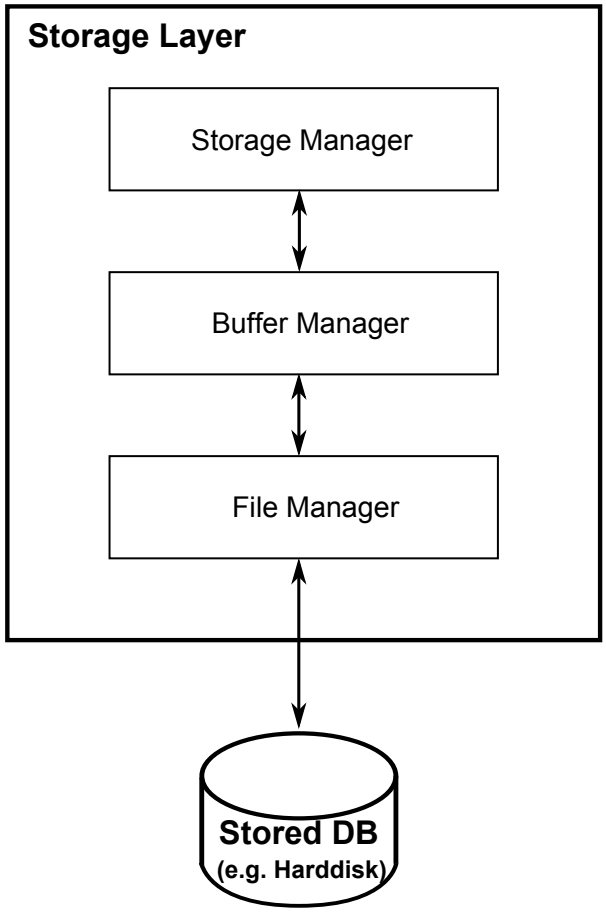

Figure 1.1: Detailed Storage Layer

2 Implementation

To abstract data types and provide extensibility, I created a file named *ctypes.c* and its header *ctypes.h* (see Listing 2.1). There I defined a typedef *bool* which should represent the boolean data type. Further I wrote an enumeration *attrType* for the types of an attribute of a relation. At the moment there are only Integer, Float and String implemented. I defined also two helper functions *getSizeOfType()* and *isVarlenType()*, whereas the first one returns the size of the attribute type given in the function parameter and the second one returns TRUE respectively 1 or FALSE respectively 0 depending on the function parameter its data type is variable length or not. The following Listing 2.1 shows the structure / function declaration of ctypes.

Listing 2.1: ctypes.h

```
typedef char bool;
#define TRUE  (bool)1
#define FALSE (bool)0

enum attrType
{
  INT = 1,
  FLOAT,
  STRING
};

size_t getSizeOfType(enum attrType type);

bool isVarlenType(enum attrType type);
```

2.1 File manager

The file manager is the connection between the hard disc and the level we would like to operate with files. Instead of working with offsets on a file, the file manager abstracts this by arranging a file into list of blocks. The argument for this decision is that files divided into blocks are way more "cache-friendly" than without. As soon as a buffer manager is implemented, this will matter because this one will then cache several blocks. This means every time a tuple is wanted, the buffer manager first checks if the tuple is already in the cache. If yes, it can be read directly from the cache and needs no disk access. If not, the buffer manager then reads and buffers the whole block where the wanted tuple is in it.

Figure 2.1 shows an example with a file consisting of three blocks and a cache with a size of one block. As the figure shows, block 1 is already in the cache. If a query now needs access to tuple T1 or tuple T2, the cache can immediately return these two. If a query needs access to tuple T3 or T4, the buffer manager then needs to read and load the corresponding block from the hard disk / file into the cache.

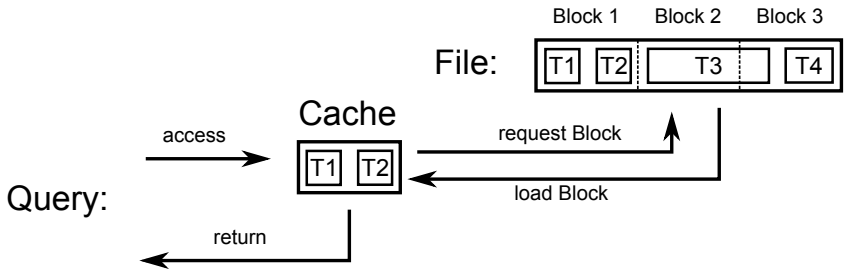

Figure 2.1: Caching example

A relation can consist of multiple fork files, depending on the storage structure implemented in the storage manager. With that architecture the file manager stays more generic and keeps the opportunity of modularity in the storage manager. In practical this means that the implementation can also handle for example column store.

The implementation of the file manager consists of the file *fmgr.c* and its header *fmgr.h* (see Listing 2.2 and Listing 2.3). The file manager has two structs, *fmgrRelation* and *fmgrRelInfo*. *fmgrRelInfo* contains the path of the tablespace, database and relation, whereas *fmgrRelation* has an array of file names and an array of fork files. The following Listing 2.2 shows the structure of the structs of fmgr.

Listing 2.2: Structs of fmgr.h

```
typedef struct fmgrRelation
{
  char * fileNames[MAX_FKN];
  FILE * files[MAX_FKN];
} fmgrRelation;

typedef struct fmgrRelInfo
{
  char tblSp[MAX_FILENAMESIZE + 1];
  char db[MAX_FILENAMESIZE + 1];
  char rel[MAX_FILENAMESIZE + 1];
} fmgrRelInfo;
```

Further the file manager must provide a couple of functions to ensure its minimal functionality. First it must be able to create a file to the corresponding relation where to write the data. In addition it must also provide a function to remove a relation. As a second "function couple" it needs to provide functions to write and read data. And as a third one it should provide some functionalities to either extend or truncate a file which represents a relation.

As the first and last functions, *fmgrInit()* and *fmgrShutdown()* have to be called at the startup and the shutdown respectively of the file manager. With *fmgrCreate()* and *fmgrClose()* my implementation of the file manager provides the functionality to create and remove a relation. Before one can read or write with *fmgrRead()* and *fmgrWrite()*, the file on which one would like to operate, needs to be open. *fmgrOpen()* provides this. To extend or truncate some data from a relation, one can call either *fmgrExtend()* respectively *fmgrTruncate()*. As the file manager works with multiple fork files per relation, *fmgrUnlink()* allows additionally to remove single fork files. To ensure that the file manager writes to an existing file, it can validate with *fmgrExists()* whether the specific fork file already exists or not. With *fmgrNrBlocks()* it can be checked how many blocks of data the fork file contains. As the last function *fmgrFlush()*

provides to write the content of a fork file permanently to the disk. The following Listing 2.3 shows the function declarations of fmgr.

Listing 2.3: fmgr.h

```
/* Initialize File Manager */
void fmgrInit();

/* Shutdown File Manager */
void fmgrShutdown();

/* Create a Relation */
fmgrRelation *fmgrCreate(const fmgrRelInfo *relInfo);

/* Close a Relation */
int fmgrClose(fmgrRelation *rel);

/* Open a Fork File from Relation */
int fmgrOpen(fmgrRelation *rel, unsigned int frkNr, bool create);

/* Read one Block from a Fork file from Relation */
size_t fmgrRead(fmgrRelation *rel, int frkNr, int blckNr, void *buffer);

/* Write one Block to a Fork file from Relation */
size_t fmgrWrite(fmgrRelation *rel, int frkNr, int blckNr, void *buffer);

/* Drop a Fork file from Relation */
int fmgrUnlink(fmgrRelation *rel, unsigned int frkNr);

/* Check if a Fork file from Relation exist */
bool fmgrExists(fmgrRelation *rel, unsigned int frkNr);

/* Number of Blocks from a Fork file from Relation */
int fmgrNrBlocks(fmgrRelation *rel, unsigned int frkNr);

/* Extend a Fork file from Relation */
bool fmgrExtend(fmgrRelation *rel, unsigned int frkNr, int blckNr, void *buffer);

/* Truncate a Fork file from Relation */
int fmgrTruncate(fmgrRelation *rel, unsigned int frkNr, int blckNr);

/* Write the Content of a Fork file from Relation permanently to disk */
int fmgrFlush(fmgrRelation *rel, unsigned int frkNr);
```

2.2 Heap file / Heap tuple

A heap file is a file without any order. It is implemented as a struct as shown in Listing 2.4. This struct contains a data section where the actual data is saved, a tuple descriptor and the size of the tuple itself. Important to mention is the structure of the heap tuple (see Figure 2.2).

There are 9 bytes reserved for the header. The first byte of the header is indicating whether the tuple is used or not (more about an unused tuple later in this report). The next 4 bytes contain the size of the whole tuple. And in the last 4 bytes of the header one can find the offset of the heap tuple in the heap file.

The content data is directly written into the data section if the attribute type is of fixed length (like Integer or Float), since the schema gives enough information about it. For variable length data types, we first write the length of the attribute and then the value itself. As an advantage of this, we don't need to parse the file since we know the size of the variable length data. This design decision was also made due to the aim of keeping the tuple as small as possible. In

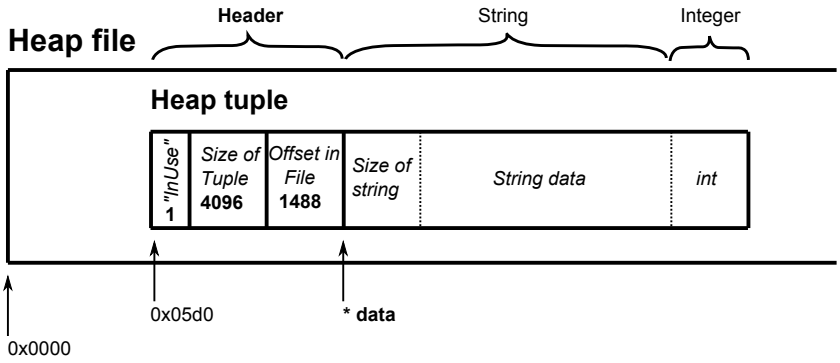

Figure 2.2: Structure of the data section of a heap tuple

example, if "hello" should be stored in a VARCHAR(100) field, we do not need 100 bytes but rather only 4 bytes to save its length as a number plus 5 bytes to store the data.

The heap file itself is written in binary rather then ASCII. The advantage of this decision is noticeable when writing numbers like integers or floats. To write a number as an integer may requires 4 bytes (depending on the operating system). That means, it takes 4 bytes to store a number like 1’000’000’000’000’000, but it takes 16 characters in ASCII, each 1 byte long, which means 16 bytes (see Figure 2.3).

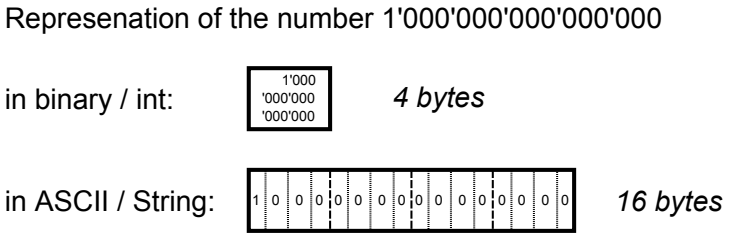

Figure 2.3: Presentation of a large number in ASCII and binary

Additionally there is a tuple descriptor, which has all necessary information about the schema of a tuple / relation. The tuple descriptor itself is implemented as a struct too (see Listing 2.4). It keeps track of the quantity of attributes a tuple has and keeps also a list of the description of the attributes.

An attribute itself is implemented as a struct again (see Listing 2.4). It keeps the name of itself, its type (string, int, float...) and its position within the array of all attributes.

If we now want to read a tuple, we go to the start position of it and read the first byte to ensure, if this tuple is used or empty. To access the data, we jump to the data section, which is the starting address plus 9 bytes from the header. To get the values of each attribute, we go trough the data part and read either the size of its type, if an attribute has a fixed length type, or read then first 4 bytes to get the size of the value and then read the data itself. Due to the tuple descriptor we are able to check if the actual attribute is fixed length or variable length.
The last part is the empty tuple we introduce. As one can later see an empty tuple is very useful when reading multiple tuples like a *select* * request. To handle with the free space between

two data tuples, we create an empty tuple instead of the free space. As a result of this, a heap file consists only of tuples, either used or empty ones. The header of an empty tuple consists only of a used byte, which value is 0, and a 4 byte number which declares the size of the the free space. The remains of the unused tuple is just some old data.

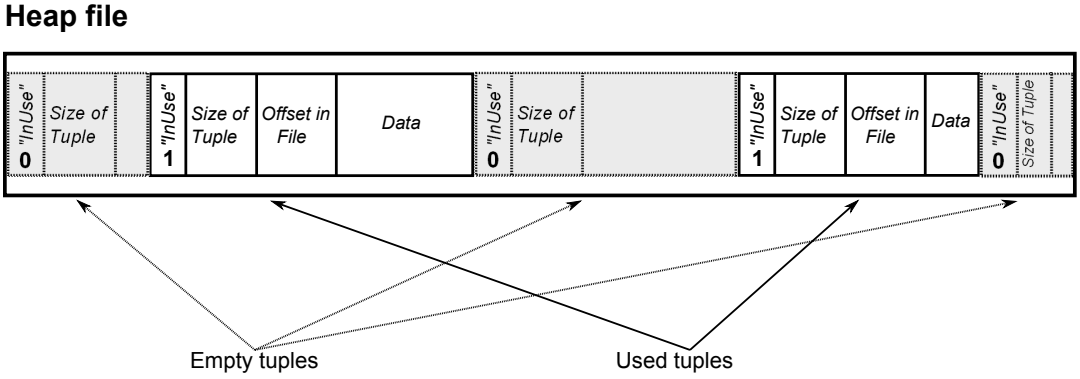

Figure 2.4: Heap file with empty and used tuples

To navigate through a heap file should be very easy now. One can start at the beginning of the file and check if the first byte in the first tuple is an empty one or one with data. If it is an empty one, we can read the next 4 bytes to determine the size of the empty tuple. So we just add the size of this to the starting position of the heap file to get to the second tuple. The same procedure is also applicable for a used tuple. That means we have now the possibility to navigate through the heap file by jumping from one tuple to the other without care about free space between them. The following Listing 2.4 shows the structure / function declaration of htup.

Listing 2.4: htup.h

```
typedef struct Attribute
{
  char attName[MAX_ATTL];
  enum attrType type;
  int attNum;
} Attribute;

typedef struct tupDesc
{
  int nrOfAttr;
  Attribute * attrs;
} tupDesc;

typedef struct hTup
{
  void * data;
  tupDesc *desc;
  size_t size;
} hTup;


void *getAttr(hTup *tup, unsigned int attrNr, size_t *size);
```

2.3 Storage manager

As the file manager deals with files and offsets, the storage manager should operate now on a higher level with relations, forks and blocks. Because the storage manager has access to the catalog, it knows how a relation is structured and saved. As it can be seen in the Listing 2.5 below, the storage manager is not that far implemented yet.
The storage manager should be able to read and write tuples as well as delete them. The latter comprised also to insert an empty tuple for the deleted ones. As long as there is no structure implemented which administrate the empty tuples, it should be possible to jump from one empty tuple to the next one. Further this jump mechanism should also be available to used tuples.

Listing 2.5: smgr.h

```
void smgrInit();

void smgrShutdown();

hTup *readHtup(fmgrRelation *rel, tupDesc *descriptor);

int writeHtup(fmgrRelation *rel, hTup *tup);
```

2.4 Testing and Errors that can occur

Because I have most of the projects developed in Java by now I tried to create some kind of unit test. For that issue I used cuTest [1], because it is small and light, therefore easy to understand, but nevertheless quite useful as it also create Test suite with multiple test functions and it provides already a few handy assertion functions.
But as the project is written in C, the testability is limited due to for example no boundary checking at arrays (one can write theoretically at the position -1 of an array and no error will be printed) and so on. That means that a theoretically correct implemented function can produce malicious data if it will be called with the wrong inputs.

3 Difficulties

As I had already some courses which where database related, the main problem was primarily with the programming language C since my experience with C is only based on some small exercises and examples from other courses. The remaining was more a question of design decisions which I made together with my supervisor.

Bibliography

[1] *CuTest: C Unit Testing Framework.* `http://cutest.sourceforge.net/`, [19.08.2011]