>>

File Management

/\ >>

❖ File Management

Aims of file management subsystem:

- organise layout of data within the filesystem
- handle mapping from database ID to file address
- transfer blocks of data between buffer pool and filesystem
- also attempts to handle file access error problems (retry)

Builds higher-level operations on top of OS file operations.

❖ File Management (cont)

Typical file operations provided by the operating system:

```
fd = open(fileName,mode)
  // open a named file for reading/writing/appending
close(fd)
  // close an open file, via its descriptor
nread = read(fd, buf, nbytes)
  // attempt to read data from file into buffer
nwritten = write(fd, buf, nbytes)
  // attempt to write data from buffer to file
lseek(fd, offset, seek_type)
  // move file pointer to relative/absolute file offset
fsync(fd)
  // flush contents of file buffers to disk
```

<< ^ >>

❖ DBMS File Organisation

How is data for DB objects arranged in the file system?

Different DBMSs make different choices, e.g.

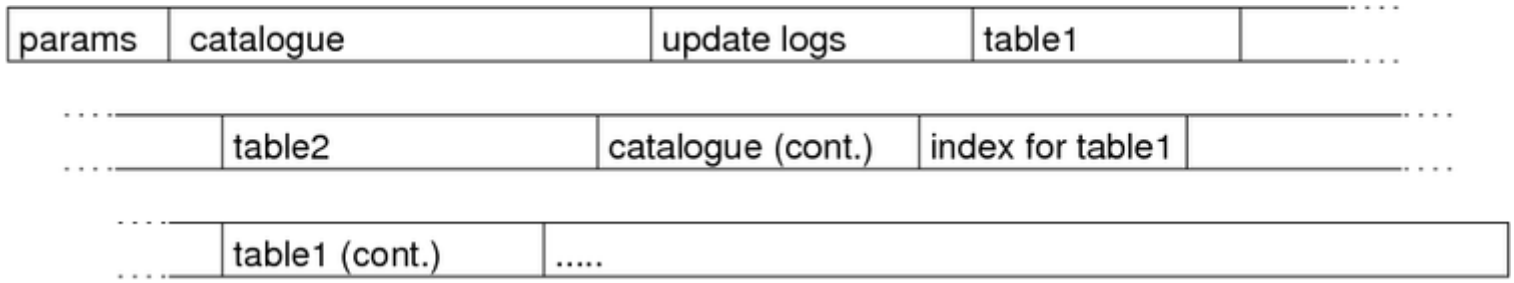
- by-pass the file system and use a raw disk partition
- have a single very large file containing all DB data
- have several large files, with tables spread across them
- have multiple data files, one for each table
- have multiple files for each table
- etc.

<< ^ >>

❖ Single-file DBMS

Consider a single file for the entire database (e.g. SQLite)

Objects are allocated to regions (segments) of the file.



If an object grows too large for allocated segment, allocate an extension.

What happens to allocated space when objects are removed?

<< ^ >>

❖ Single-file DBMS (cont)

Allocating space in Unix files is easy:

- simply seek to the place you want and write the data
- if nothing there already, data is appended to the file
- if something there already, it gets overwritten

If the seek goes way beyond the end of the file:

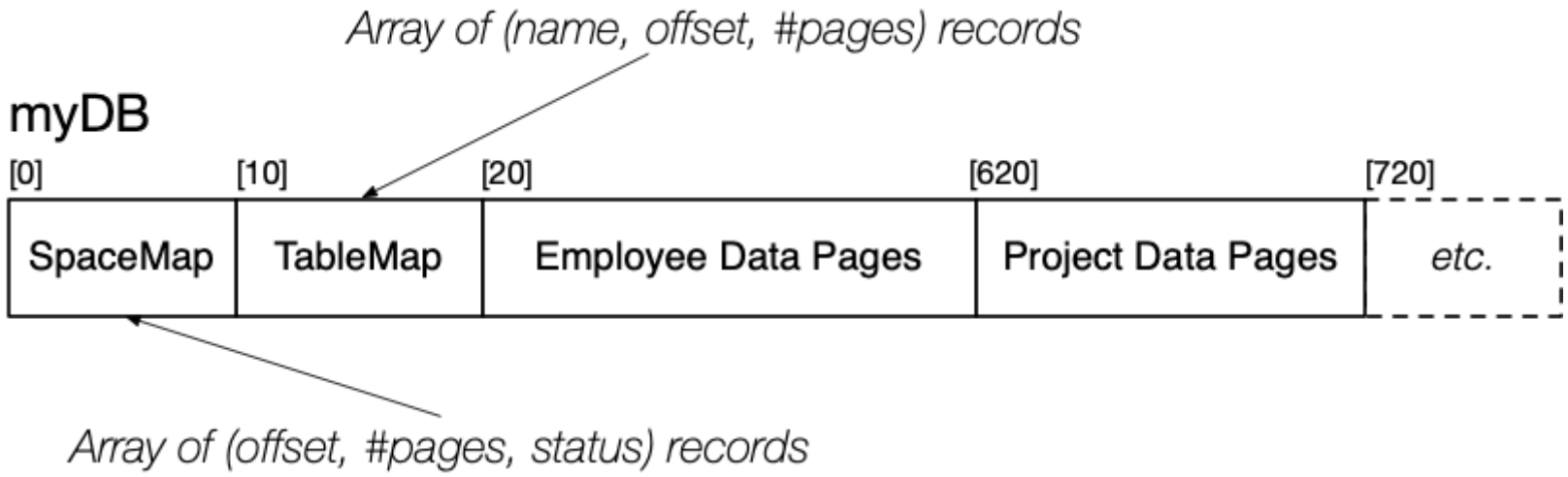
- Unix does not (yet) allocate disk space for the "hole"
- allocates disk storage only when data is written there

With the above, a disk/file manager is easy to implement.

<< ^ >>

❖ Single-file Storage Manager

Consider the following simple single-file DBMS layout:



E.g.

SpaceMap = [(0,10,U), (10,10,U), (20,600,U), (620,100,U), (720,20,F)]

TableMap = [("employee",20,500), ("project",620,40)]

❖ Single-file Storage Manager (cont)

Each file segment consists of a number fixed-size blocks

The following data/constant definitions are useful

```
#define PAGESIZE 2048    // bytes per page

typedef long PageId;     // PageId is block index
                         // pageOffset=PageId*PAGESIZE

typedef char *Page;      // pointer to page/block buffer
```

Typical **`PAGESIZE`** values: 1024, 2048, 4096, 8192

❖ Single-file Storage Manager (cont)

Possible storage manager data structures for opened DBs & Tables

```
typedef struct DBrec {
    char *dbname;     // copy of database name
    int fd;           // the database file
    SpaceMap map;     // map of free/used areas
    TableMap names;   // map names to areas + sizes
} *DB;

typedef struct Relrec {
    char *relname;    // copy of table name
    int   start;      // page index of start of table data
    int   npages;     // number of pages of table data
    ...
} *Rel;
```

<< ^ >>

❖ Example: Scanning a Relation

With the above disk manager, a query like

```
select name from Employee
```

might be implemented as

```
DB db = openDatabase("myDB");
Rel r = openRelation(db,"Employee");
Page buffer = malloc(PAGESIZE*sizeof(char));
for (int i = 0; i < r->npages; i++) {
   PageId pid = r->start+i;
   get_page(db, pid, buffer);
   for each tuple in buffer {
      get tuple data and extract name
      add (name) to result tuples
   }
}
```

❖ Single-File Storage Manager

```
// start using DB, buffer meta-data
DB openDatabase(char *name) {
   DB db = new(struct DBrec);
   db->dbname = strdup(name);
   db->fd = open(name,O_RDWR);
   db->map = readSpaceTable(db->fd);
   db->names = readNameTable(db->fd);
   return db;
}
// stop using DB and update all meta-data
void closeDatabase(DB db) {
   writeSpaceTable(db->fd,db->map);
   writeNameTable(db->fd,db->map);
   fsync(db->fd);
   close(db->fd);
   free(db->dbname);
   free(db);
}
```

COMP9315 21T1 ♢ File Management ♢ [10/19]

<< ^ >>

❖ Single-File Storage Manager (cont)

```
// set up struct describing relation
Rel openRelation(DB db, char *rname) {
    Rel r = new(struct Relrec);
    r->relname = strdup(rname);
    // get relation data from map tables
    r->start = ...;
    r->npages = ...;
    return r;
}

// stop using a relation
void closeRelation(Rel r) {
    free(r->relname);
    free(r);
}
```

❖ Single-File Storage Manager (cont)

```
// assume that Page = byte[PageSize]
// assume that PageId = block number in file

// read page from file into memory buffer
void get_page(DB db, PageId p, Page buf) {
   lseek(db->fd, p*PAGESIZE, SEEK_SET);
   read(db->fd, buf, PAGESIZE);
}

// write page from memory buffer to file
void put_page(Db db, PageId p, Page buf) {
   lseek(db->fd, p*PAGESIZE, SEEK_SET);
   write(db->fd, buf, PAGESIZE);
}
```

<< ^ >>

❖ Single-File Storage Manager (cont)

Managing contents of space mapping table can be complex:

```
// assume an array of (offset,length,status) records

// allocate n new pages
PageId allocate_pages(int n) {
    if (no existing free chunks are large enough) {
        int endfile = lseek(db->fd, 0, SEEK_END);
        addNewEntry(db->map, endfile, n);
    } else {
        grab "worst fit" chunk
        split off unused section as new chunk
    }
    // note that file itself is not changed
}
```

❖ Single-File Storage Manager (cont)

Similar complexity for freeing chunks

```
// drop n pages starting from p
void deallocate_pages(PageId p, int n) {
    if (no adjacent free chunks) {
        markUnused(db->map, p, n);
    } else {
        merge adjacent free chunks
        compress mapping table
    }
    // note that file itself is not changed
}
```

Changes take effect when **`closeDatabase()`** executed.

<< ^ >>

❖ Multiple-file Disk Manager

Most DBMSs don't use a single large file for all data.

They typically provide:

- multiple files partitioned physically or logically
- mapping from DB-level objects to files (e.g. via catalog meta-data)

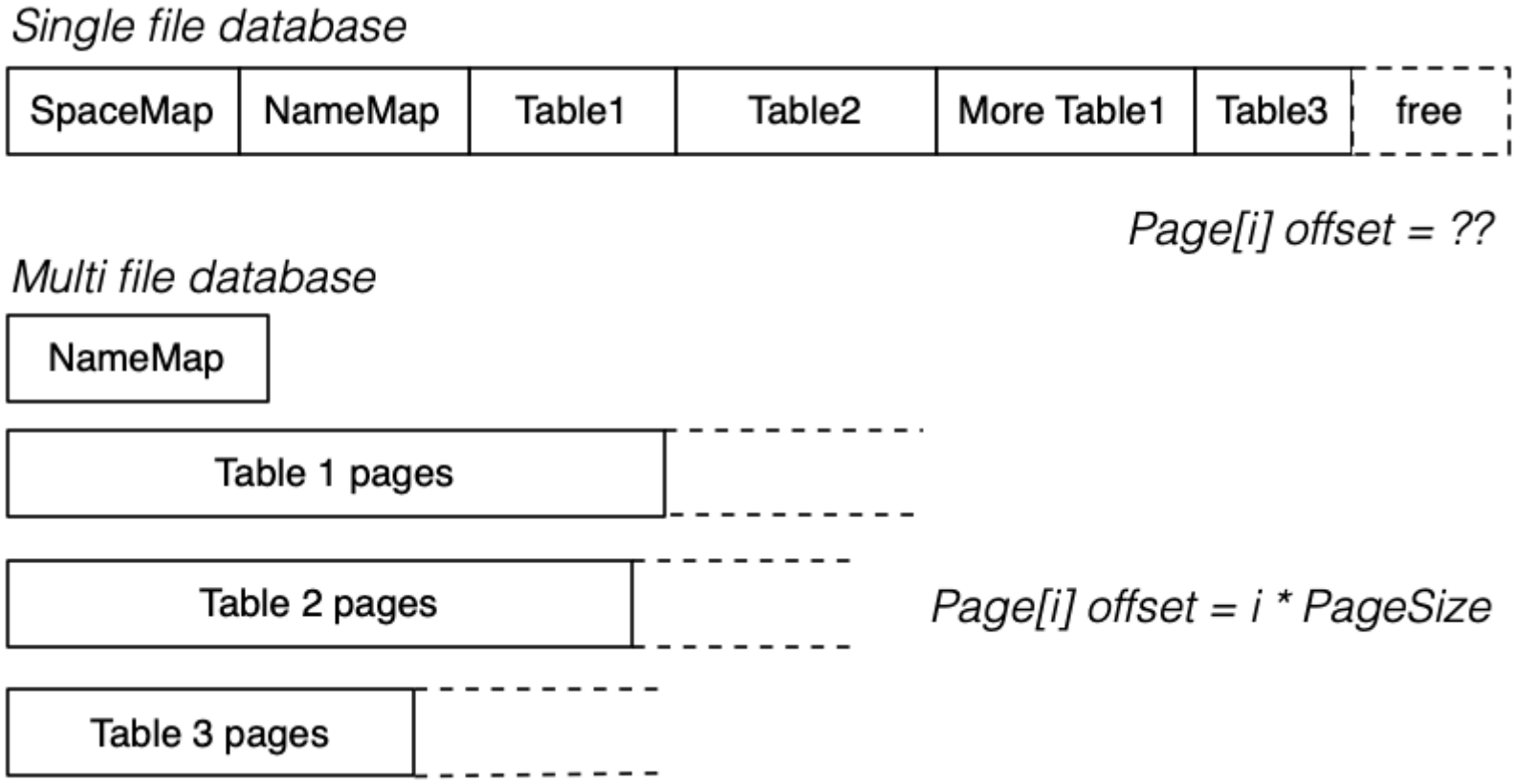
Precise file structure varies between individual DBMSs.

Using multiple files (one file per relation) can be easier, e.g.

- adding a new relation
- extending the size of a relation
- computing page offsets within a relation

❖ Multiple-file Disk Manager (cont)

Example of single-file vs multiple-file:



Consider how you would compute file offset of page[i] in table[1] ...

COMP9315 21T1 ♢ File Management ♢ [16/19]

<< ^ >>

❖ Multiple-file Disk Manager (cont)

Structure of **`PageId`** for data pages in such systems ...

If system uses one file per table, **`PageId`** contains:

- relation indentifier (which can be mapped to filename)
- page number (to identify page within the file)

If system uses several files per table, **`PageId`** contains:

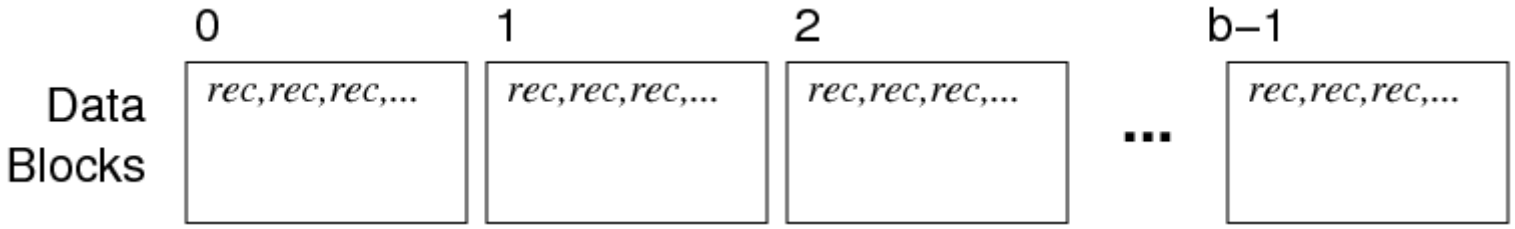
- relation identifier
- file identifier (combined with relid, gives filename)
- page number (to identify page within the file)

<< ^ >>

❖ DBMS File Parameters

Our view of relations in DBMSs:

- a relation is a set of r tuples, with average size R bytes
- the tuples are stored in b data pages on disk
- each page has size B bytes and contains up to c tuples
- data is transferred disk↔memory in whole pages
- cost of disk↔memory transfer T_r, T_w dominates other costs



<< ^

❖ DBMS File Parameters (cont)

Typical DBMS/table parameter values:

Quantity	Symbol	E.g. Value
total # tuples	r	10^6
record size	R	128 bytes
total # pages	b	10^5
page size	B	8192 bytes
# tuples per page	c	60
page read/write time	T_r, T_w	10 msec
cost to process one page in memory	-	$\cong 0$

Produced: 21 Feb 2021