Week 02 Lecture

Storage Management

Storage Management

2/58

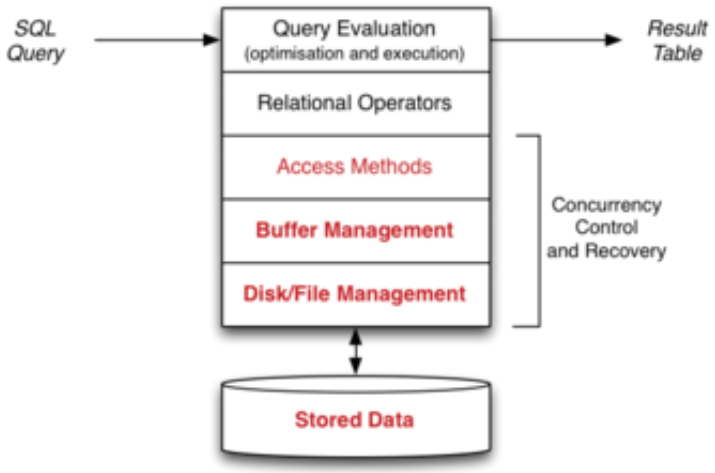
Aims of storage management in DBMS:

- provide view of data as collection of tables/tuples
- map from database objects (e.g. tables) to disk files
- manage transfer of data to/from disk storage
- use buffers to minimise disk/memory transfers
- interpret loaded data as tuples/records
- give foundation for file structures used by access methods

... Storage Management

3/58

Levels of DBMS related to storage management:



... Storage Management

4/58

Consider the query:

```
select student, course from Enrolments;
```

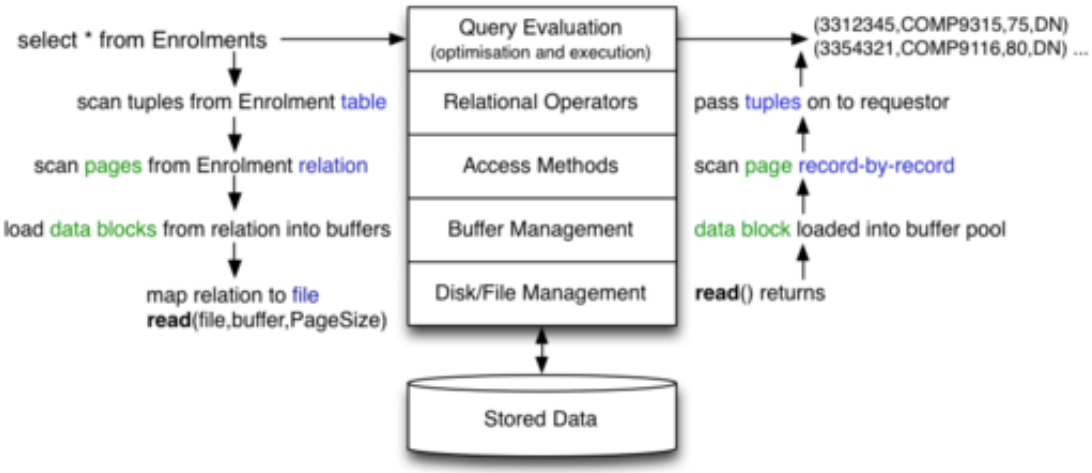
How the query executor deals with the database ...

```
DB db = openDatabase("myDB");
Reln r = openRel(db,"Enrolments");
Scan s = startScan(r);
Tuple t;  Results res = NULL;
while ((t = nextTuple(s)) != NULL)
{
    int stuid = getField(t,"student");
    char *course = getField(t,"course");
    res = addTuple(res, mkTuple(stuid,course));
```

}

... Storage Management

5/58

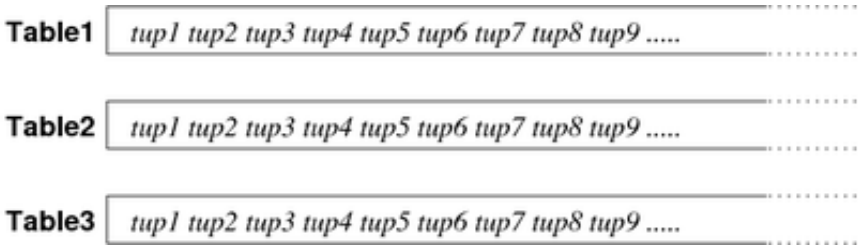


Views of Data

6/58

Users and top-level query evaluator see data as

- a collection of tables, each with a schema (tuple-type)
- where each table contains a set (sequence) of tuples

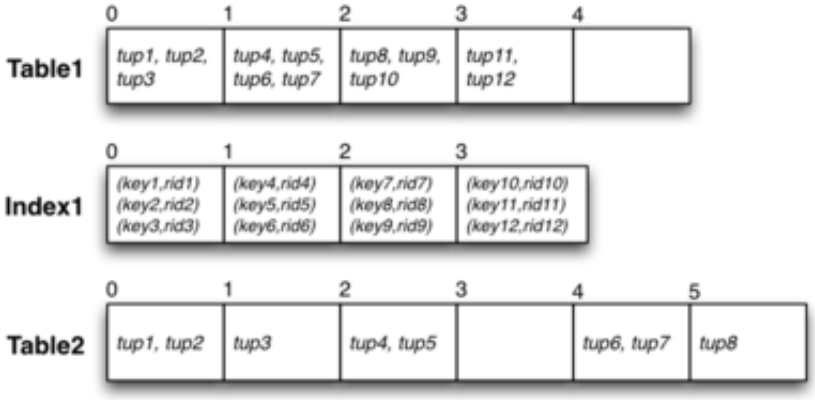


... Views of Data

7/58

Relational operators and *access methods* see data as

- sequence of fixed-size pages, typically 1KB to 8KB
- where each page contains tuple data or index data

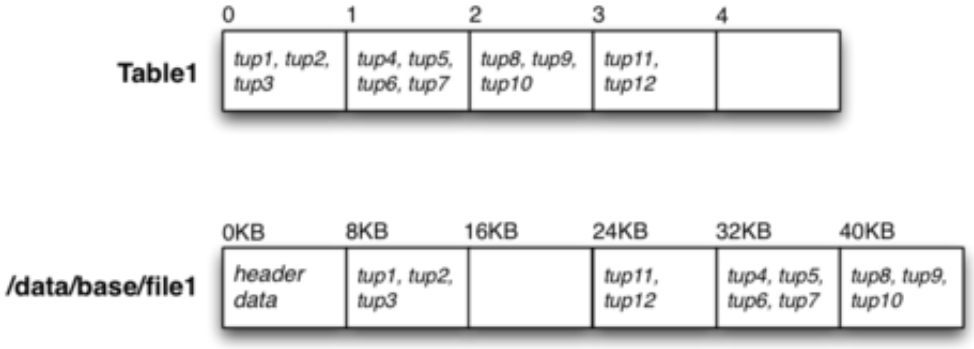


... Views of Data

8/58

File manager sees both DB objects and file store

- maps (tableName, pageIndex) to (file, offset)



Storage Management Topics

9/58

Topics to be considered:

- DB Object Management (Catalog)
  - how tables/functions/types, etc. are represented
- Disks and Files
  - performance issues and organisation of disk files
- Buffer Management
  - using caching to improve DBMS system throughput
- Tuple/Page Management
  - how tuples are represented within disk pages

Each topic illustrated by its PostgreSQL implementation.

Storage Manager Interface

10/58

The storage manager provides higher levels of system

- with an abstraction based on relations/pages/tuples

- which maps down to files/blocks/records (via buffers)

Example: simple scan of a relation:

```
select student,course from Enrolments
```

High-level view of result: sequence of tuples.

How is this mapped to accesses to files/blocks/records?

... Storage Manager Interface

11/58

Implementation at level of query executor: (not PostgreSQL)

```
DB db = openDatabase("myDB");
Reln r = openRelation(db,"Enrolments");
Scan s = startScan(r);
Tuple t;  Results res = NULL;
while ((t = nextTuple(s)) != NULL)
{
    int stuid = getField(t,"student");
    char *course = getField(t,"course");
    res = addTuple(res, mkTuple(stuid,course));
}
```

... Storage Manager Interface

12/58

The storage manager provides mechanisms for:

- representing database objects during query execution
  - `DB` (handle on an authorised/opened database)
  - `Reln` (handle on an opened relation)
  - `Page` (memory buffer to hold contents of data block)
  - `Tuple` (memory holding data values from one tuple)
- referring to database objects (addresses)
  - symbolic (e.g. database/schema/table/field names)
  - abstract physical (e.g. `PageId`, `TupleId`)

... Storage Manager Interface

13/58

Examples of references (addresses) used in DBMSs:

- `PageID` ... identifies (locates) a block of data
  - typically, `PageID = FileID + Offset`
  - where `Offset` gives location of block within file
- `TupleID` ... identifies (locates) a single tuple
  - typically, `TupleID = PageID + Offset`
  - where `Offset` gives location of tuple within page

Note that `Offset`s may be indexes into mapping tables giving real address.

... Storage Manager Interface

14/58

Possible implementation for `DB` object ...

```
typedef struct Database {
```

```
   char    *name; // database name
   Catalog  cat;  // meta-data
   ...
} *DB;
```

Possible implementation of `Reln` object ...

```
typedef struct Relation {
   char    *name; // table name
   File     file; // fd for table file
   ...
} *Reln;
```

... Storage Manager Interface

15/58

Possible implementation for `Scan` object ...

- query executor wants to see result tuple-at-a-time
- DBMS read *blocks* from files (page-of-tuples-at-a-time)

```
typedef struct ScanData {
   File  file;   // file holding table data
   Page  page;   // most recently read data
   int   pageno; // current block within file
   int   tupno;  // current tuple within page
   ...
} *Scan;
```

... Storage Manager Interface

16/58

`startScan()` might be implemented as:

```
Scan startScan(Reln r) {
   Scan s = MemAlloc(struct ScanData);
   s->file = r->file;
   s->page = null;
   s->pageno = 0;
   s->tupno = 0;
   return s;
}
```

... Storage Manager Interface

17/58

And `nextTuple()` might be implemented as:

```
Tuple nextTuple(Scan s) {
   if (noMoreTuplesIn(s->page,s->tupno))
      if (noMorePagesIn(s->file))
         return NULL;
      s->page = getPage(s->file,s->pageno);
      s->pageno++;
      s->tupno = 0;
   }
   Tuple t = getTuple(s->page,s->tupno);
   s->tupno++;
   return t;
}
```

From Symbolic to Internal

18/58

How do we determine ...

- information about a database, given its name
- information about a table, given its name

DBMSs use catalog data in special tables

E.g. for PostgreSQL

```
pg_database(oid, datname, datdba, datacl[], ...)
pg_namespace(oid, nspname, nspowner, nspacl[], ...)
pg_class(oid, relname, relnamespace, ..., relkind,
         reltuples, relnatts, relhaspkey, relacl[] ...)
pg_attribute(oid, attrelid, attname, atttypid, attnum, ...)
pg_type(oid, typname, typnamespace, typowner, typlen, ...)
```

Exercise 1: Table Statistics

19/58

Using the PostgreSQL catalog, write a view

- `create view pop("table",ntuples) as ...`
- to return name of table and estimated #tuples
- for all tables in the `public` schema

It should behave as follows:

```
db=# select * from pop;
  table   | ntuples
----------+---------
 table1   |     162
 table2   |    2788
 table3   |    1500
...
```

Exercise 2: Extracting a Schema

20/58

Write a PLpgSQL function:

- `function schema() returns setof text`
- giving a list of table schemas in the `public` schema

It should behave as follows:

```
db=# select * from schema();
            tables
---------------------------
 table1(x, y, z)
 table2(a, b)
 table3(id, name, address)
...
```

Exercise 3: More accurate tuple counts

21/58

The earlier example:

```
create view pop("table",ntuples) as ...
```

gives *estimated* tuple counts which may not be accurate.

Write a PLpgSQL function that returns accurate counts:

```
create type TableInfo as (table text, ntuples int);
create function pop2() returns setof TableInfo ...
```

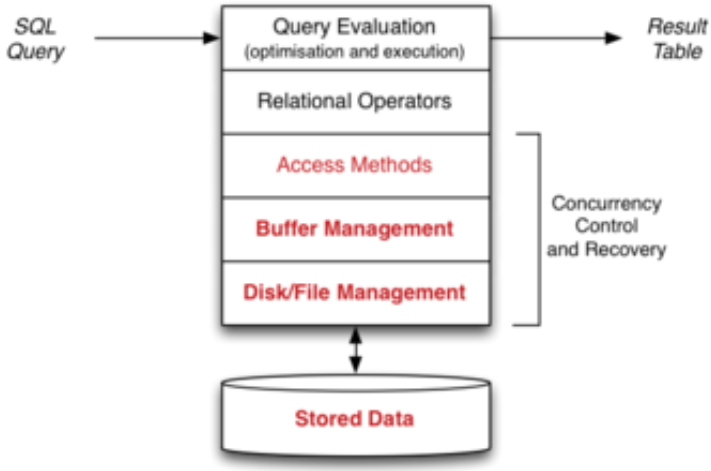
Hint: you will need to use dynamically-generated queries.

Storage Manager

DBMS Storage Manager

23/58

Levels of DBMS related to storage management:



Storage Technology

24/58

HDD (hard disk drive)

- large (10TB), cheap $/B, high power consumption
- R/W: slow (ms), latency, block xfer
- reliability via redundancy (RAID)

SSD (solid state drive)

- medium (500GB), moderate $/B, low power consumption
- R: fast (μs), byte/page xfer; W: mod (100μs), block erase
- reliability: limited block erasures

RAM (random access memory)

- small (10GB), high $/B, low power consumption, *volatile*
- R/W: fast (ns), byte xfer

... Storage Technology

25/58

Persistent storage is

- large, cheap, relatively slow, accessed in blocks
- used for long-term storage of data

Computational storage is

- small, expensive, fast, accessed by byte/word
- used for all analysis of data

Access cost HDD:RAM ≅ 100000:1, e.g.

- 100ms to read block containing two tuples
- 1μs to compare fields in two tuples

Disk Management

26/58

Aim of disk management subsystem:

- handles mapping from database ID to disk address
- transfer blocks of data between buffer pool and disk
- also attempts to handle disk access error problems (retry)

Note: DBMSs typically do not deal with the raw disk

- mapping from `PageId` to disk is via filesystem

File Management

27/58

Aims of file management subsystem:

- organise layout of data within the filesystem
- handles 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)

Essentially the same as those for disk management.

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

... File Management

28/58

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

29/58

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

Different DBMSs make different choices, e.g.

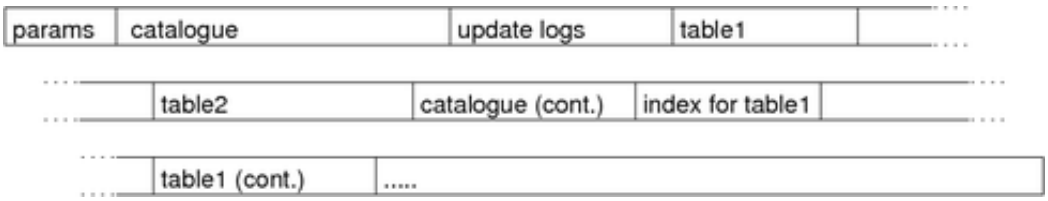
- by-pass the file system and use a raw disk partition
- have a single very large file containing all DB data
- have several large files, with tables spread across them
- have multiple data files, one for each table
- have multiple files for each table
- etc.

Single-file DBMS

30/58

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

31/58

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 Disk Manager

32/58

Simple disk manager for a single-file database:

```
// Disk Manager data/functions
#define PAGESIZE 2048   // bytes per page
typedef int PageId;     // PageId is block index

typedef struct DBdescriptor {
   char *dbname;     // copy of database name
   int   fd;         // the database file
```

```
   SpaceTable map;   // map of free/used areas
   NameTable names;  // map names to areas + sizes
   ...
} *DB;

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

... Single-file Disk Manager


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

... Single-file Disk Manager


```
// set up struct describing relation
Reln openRelation(DB db, char *rname) {
   Reln r = new(RelDescriptor);
   r->relname = strdup(rname);
   // get relation data from map tables
   r->start = ...;
   r->npages = ...;
   return r;
}
// stop using a relation
void closeRelation(Reln r) {
   free(r);
}
#define nPages(r)  (r->npages)
#define makePageId(r,i)  (r->first + i)
```

... Single-file Disk Manager


```
// assume that Page = buffer of PageSize bytes
// 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, pageOffset(p), 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, pageOffset(p), SEEK_SET);
   write(db->fd, buf, PAGESIZE);
}
```

... Single-file Disk Manager

36/58

The `pageOffset()` function uses the DB map

- takes a `PageId` value
- uses the DB space map
- returns an absolute file offset

E.g. each table is allocated large contiguous segment of file

- get start address of `relation(PageId)` from map
- add `pageNumber(PageId)*PAGESIZE` to give offset

... Single-file Disk Manager

37/58

```
// managing contents of mapping table is complex
// assume a list of (offset,length,status) tuples

// allocate n new pages at end of file
PageId allocate_pages(int n) {
   int endfile = lseek(db->fd, 0, SEEK_END);
   addNewEntry(db->map, endfile, n);
   // note that file itself is not changed
}
// drop n pages starting from p
void deallocate_pages(PageId p, int n) {
   markUnused(db->map, p, n);
   // note that file itself is not changed
}
```

Example: Scanning a Relation

38/58

With the above disk manager, the query:

```
select name from Employee
```

might be implemented as something like

```
DB db = openDatabase("myDB");
Reln r = openRelation(db,"Employee");
Page buffer = malloc(PAGESIZE*sizeof(char));
for (int i = 0; i < nPages(r); i++) {
   PageId pid = makePageId(r,i);
   get_page(db, pid, buffer);
   foreach tuple in buffer {
      get tuple data and extract name
   }
}
```

Exercise 4: Relation Scan Cost

Consider a table *R* with 10^5 tuples, implemented as

- number of records/tuples *r = 100,000*
- average size of records *R = 200* bytes
- size of data pages *B = 4096* bytes
- time to read one data page T_r *= 10msec*
- time to check one tuple *1 usec*
- time to form one result tuple *1 usec*
- overhead from scanning one page *40 usec*

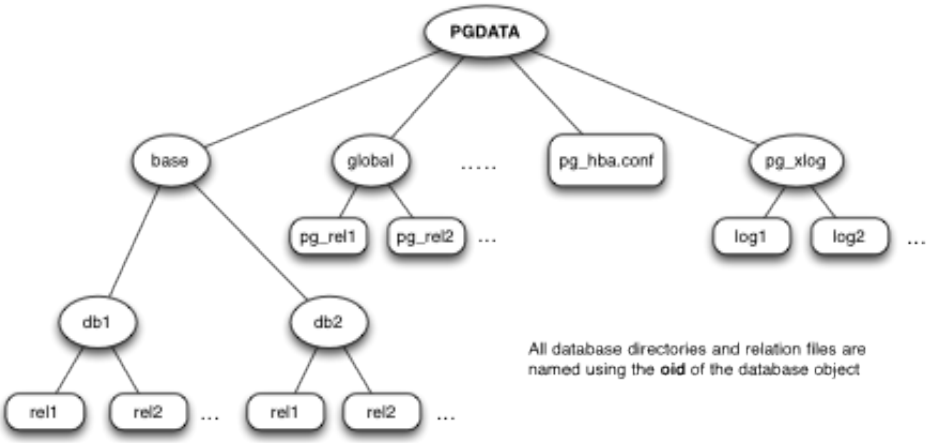
Calculate the total time-cost for answering the query:

```
select * from R where x > 10;
```

if 50% of the tuples satisfy the condition.

PostgreSQL Storage Manager

PostgreSQL uses the following file organisation ...



... PostgreSQL Storage Manager

Components of storage subsystem:

- mapping from relations to files (**`RelFileNode`**)
- abstraction for open relation pool (**`storage/smgr`**)
- functions for managing files (**`storage/smgr/md.c`**)
- file-descriptor pool (**`storage/file`**)

PostgreSQL has two basic kinds of files:

- heap files containing data (tuples)
- index files containing index entries

Note: `smgr` designed for many storage devices; only mag disk handler used

Relations as Files

42/58

PostgreSQL identifies relation files via their OIDs.

The core data structure for this is **`RelFileNode`**:

```
typedef struct RelFileNode {
    Oid  spcNode;  // tablespace
    Oid  dbNode;   // database
    Oid  relNode;  // relation
} RelFileNode;
```

Global (shared) tables (e.g. `pg_database`) have

- `spcNode == GLOBALTABLESPACE_OID`
- `dbNode == 0`

... Relations as Files

43/58

The **`relpath`** function maps **`RelFileNode`** to file:

```
char *relpath(RelFileNode r)  // simplified
{
   char *path = malloc(ENOUGH_SPACE);

   if (r.spcNode == GLOBALTABLESPACE_OID) {
      /* Shared system relations live in PGDATA/global */
      Assert(r.dbNode == 0);
      sprintf(path, "%s/global/%u",
              DataDir, r.relNode);
   }
   else if (r.spcNode == DEFAULTTABLESPACE_OID) {
      /* The default tablespace is PGDATA/base */
      sprintf(path, "%s/base/%u/%u",
              DataDir, r.dbNode, r.relNode);
   }
   else {
      /* All other tablespaces accessed via symlinks */
      sprintf(path, "%s/pg_tblspc/%u/%u/%u", DataDir
              r.spcNode, r.dbNode, r.relNode);
   }
   return path;
}
```

Exercise 5: PostgreSQL Files

44/58

In my PostgreSQL server

- examine the content of the `$PGDATA` directory
- find the directory containing the `pizza` database
- find the file in this directory for the `People` table
- examine the contents of the `People` file
- what are the other files in the directory?
- are there *forks* in any of my databases?

File Descriptor Pool

45/58

Unix has limits on the number of concurrently open files.

PostgreSQL maintains a pool of open file descriptors:

- to hide this limitation from higher level functions
- to minimise expensive `open()` operations

File names are simply strings: **`typedef char *FileName`**

Open files are referenced via: **`typedef int File`**

A **`File`** is an index into a table of "virtual file descriptors".

... File Descriptor Pool

Interface to file descriptor (pool):

```
File FileNameOpenFile(FileName fileName,
                      int fileFlags, int fileMode);
     // open a file in the database directory ($PGDATA/base/...)
File OpenTemporaryFile(bool interXact);
     // open temp file; flag: close at end of transaction?
void FileClose(File file);
void FileUnlink(File file);
int  FileRead(File file, char *buffer, int amount);
int  FileWrite(File file, char *buffer, int amount);
int  FileSync(File file);
long FileSeek(File file, long offset, int whence);
int  FileTruncate(File file, long offset);
```

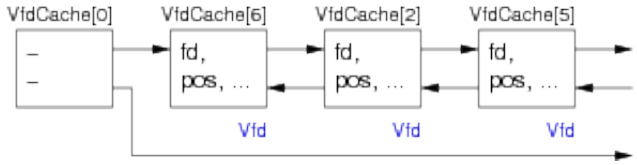
Analogous to Unix syscalls `open()`, `close()`, `read()`, `write()`, `lseek()`, ...

... File Descriptor Pool

Virtual file descriptors (**`Vfd`**)

- physically stored in dynamically-allocated array

- also arranged into list by recency-of-use



`VfdCache[0]` holds list head/tail pointers.

... File Descriptor Pool

Virtual file descriptor records (simplified):

```
typedef struct vfd
{
    s_short  fd;              // current FD, or VFD_CLOSED if none
```

```
    u_short  fdstate;         // bitflags for VFD's state
    File     nextFree;        // link to next free VFD, if in freelist
    File     lruMoreRecently; // doubly linked recency-of-use list
    File     lruLessRecently;
    long     seekPos;         // current logical file position
    char     *fileName;       // name of file, or NULL for unused VFD
    // NB: fileName is malloc'd, and must be free'd when closing the VFD
    int      fileFlags;       // open(2) flags for (re)opening the file
    int      fileMode;        // mode to pass to open(2)
} Vfd;
```

File Manager

The "magnetic disk storage manager"

- manages its own pool of open file descriptors
- each one represents an open relation file (Vfd)
- may use several Vfd's to access data, if file > 2GB
- manages mapping from **`PageId`** to file+offset.

PostgreSQL `PageId` values are structured:

```
typedef struct
{
    RelFileNode rnode;    // which relation/file
    ForkNumber  forkNum;  // which fork (of reln)
    BlockNumber blockNum; // which page/block
} BufferTag;
```

... File Manager

Access to a block of data proceeds as follows:

```
offset = BlockNumber * BLCKSZ
fileID = RelFileNode+ForkNumber
if (fileID is already in Vfd pool) {
    if (offset is in this file)
        fd = use Vfd from pool
    else
        fd = allocate new Vfd for next part of file
} else {
    fd = allocate new Vfd for this file
}
seek to offset in fd
read/write data page (BLCKSZ bytes)
```

`BLCKSZ` is a global configurable constant (default: 8192).

Buffer Pool

Buffer Pool

Aim of DBMS buffer pool:

- reduce number of disk reads and disk writes

Assumption:

- some pages are accessed many times during query eval^n

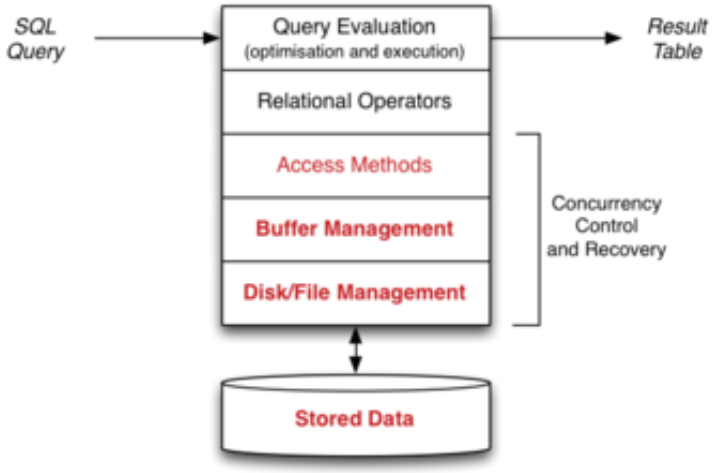
Achieves read/write reduction by:

- holding many pages in memory for re-use
- only removing them when "absolutely necessary"
- sharing pages among multiple transactions (global pool)

... Buffer Pool

53/58

Buffer pool lies between access methods and disk manager



... Buffer Pool

54/58

Access methods manipulate *pages* filled with tuples

- could be achieved via `get/put_page()` operations

Interface becomes `request/release` rather than `get/put`

- many relational ops work as
  - *get, process, [write], get, process, [write], get, process, ...*
- with a buffer pool, relational ops become
  - *request, process, release, request, process, release, ...*
  - where only some of the *requests* result in `read`
  - where only some of the *releases* result in `write`

... Buffer Pool

55/58

Buffer pool data structures:

- a fixed-size, memory-resident collection of *frames* (page-slots)
- a directory containing information about the status of each frame

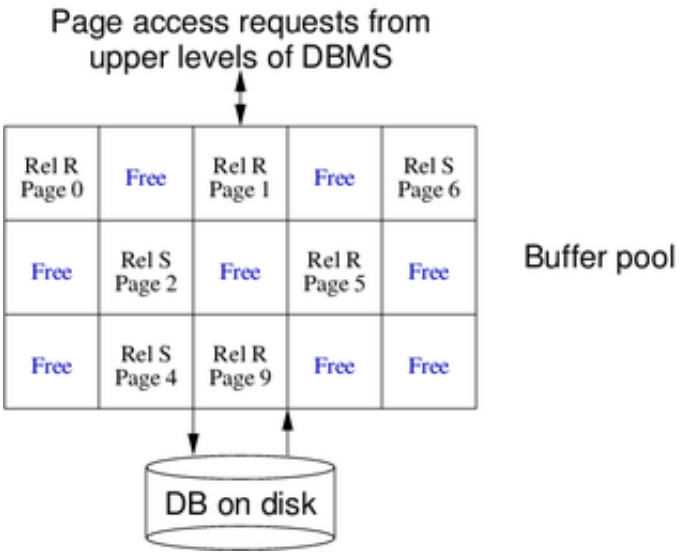
For each frame, we need to know:

- contents = `PageId` *(dbid,relid,page#)* or Empty flag (*tag*)
- whether it has been modified since loading (*dirty bit*)

- how many transactions are currently using it (*pin count*)
- time-stamp for most recent access (assists with replacement)
- pointer to next free frame (free list)

... Buffer Pool

56/58



... Buffer Pool

57/58

Basic buffer pool interface

```
Page request_page(PageId pid);
```

- get disk block corresponding to page `pid` into buffer pool

```
void release_page(PageId pid);
```

- indicate that page `pid` is no longer in use (advisory)

```
void mark_page(PageId pid);
```

- indicate that page `pid` has been modified (advisory)

```
void flush_page(PageId pid);
```

- write contents of page `pid` from buffer pool onto disk

```
void hold_page(PageId pid);
```

- recommend that page `pid` should not be swapped out

Exercise 6: Buffer Pool Functions

58/58

Assuming a `Frame` data structure like

```
typedef struct FrameData *FrameData;
struct FrameData {
   PageId  tag;     // contents
   Page    page;    // buffer containing data
   int     dirty;   // modified flag
```

```
    int     pin;     // pin count
    time_t  mra;     // most recent access
    FrameData *next; // free list
};
```

Give implementations for `request_page()`, `release_page()`, `replace_frame()`

Produced: 7 Mar 2016