course 2 Storage

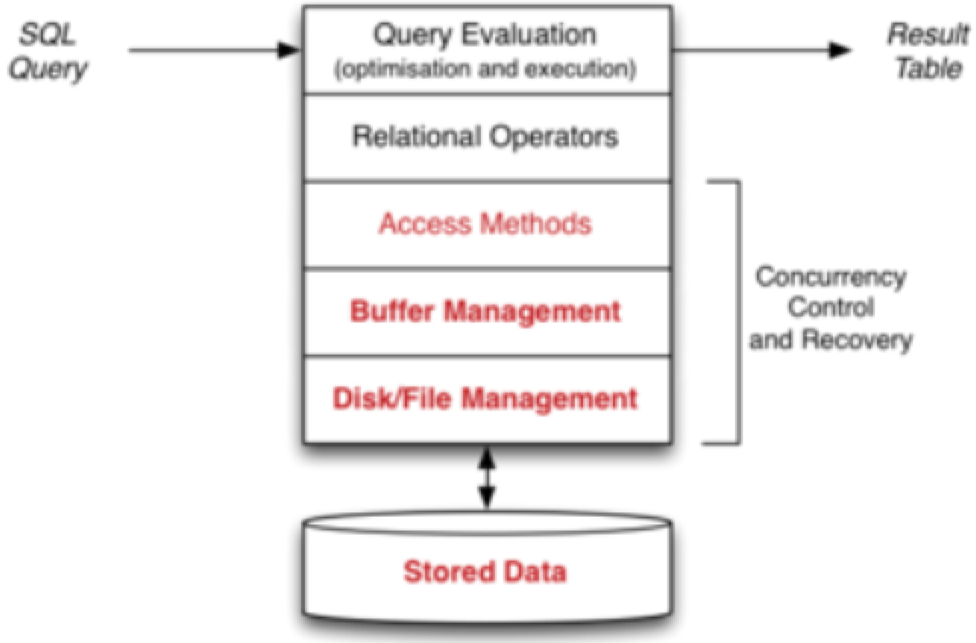
this section is about storage in postgreSQL like: devices, files, pages, tuples, buffers, catalogs

1. Storage Management

aims of storage management in DBMS are shown below:

- map from database objects (e.g. tables) to disk files
- use buffer to minimise disk/memory data transfers
    - and also manage transfer of data to/from disk storage
- provide view of data as collection of pages/tuples

levels of DBMS related to storage management:



there are severals topics to be considered in storage management:

- Disk and files
    - performance issue and organisation of disk files
- Buffer management
    - cache & page replacement strategies
- Tuple/Page management
    - how tuples are represented within disk pages
- DB object management (Catalog)
    - how tables/views/functions/types, etc are represented

1.1 views of data

- Users and top-level query see data as:
    - a collection a tables, each table contains a set of tuples

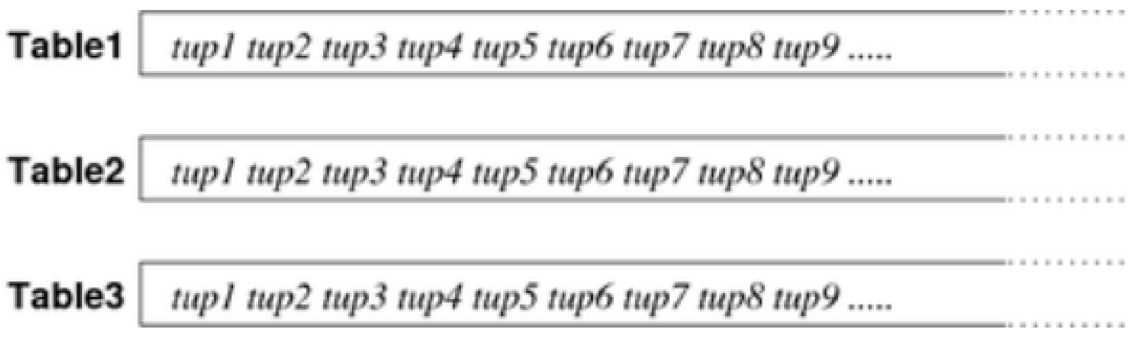


Table1
tup1 tup2 tup3 tup4 tup5 tup6 tup7 tup8 tup9 .....
Table2
tup1 tup2 tup3 tup4 tup5 tup6 tup7 tup8 tup9 .....
Table3
tup1 tup2 tup3 tup4 tup5 tup6 tup7 tup8 tup9 .....

- Relational operators and access methods see data as:
    - sequence of **fixed-size pages**, typically 1KB to 8KB, **each page contains tuple or index data**

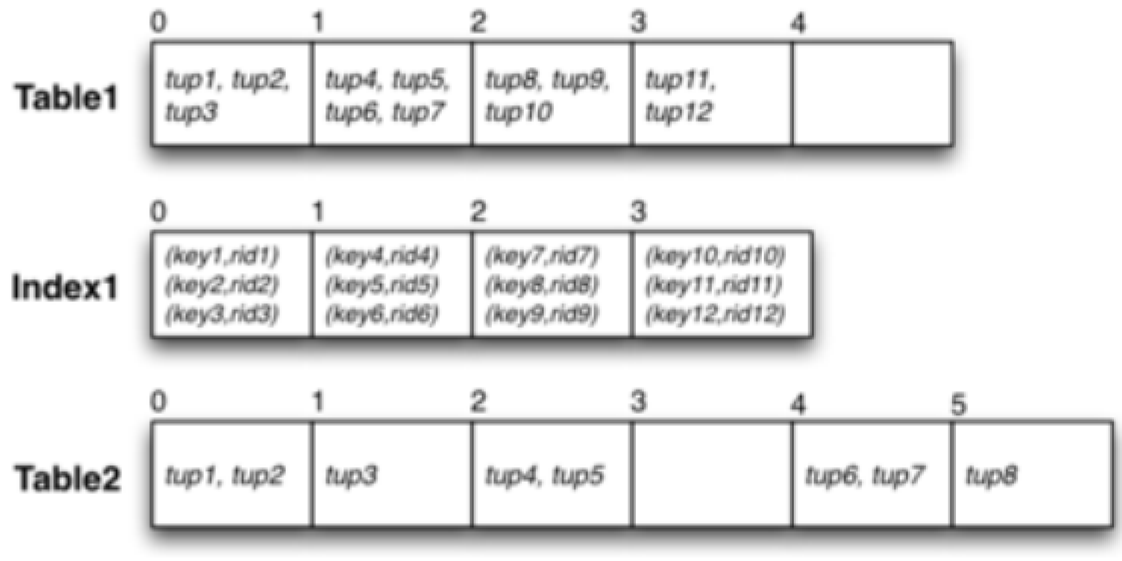
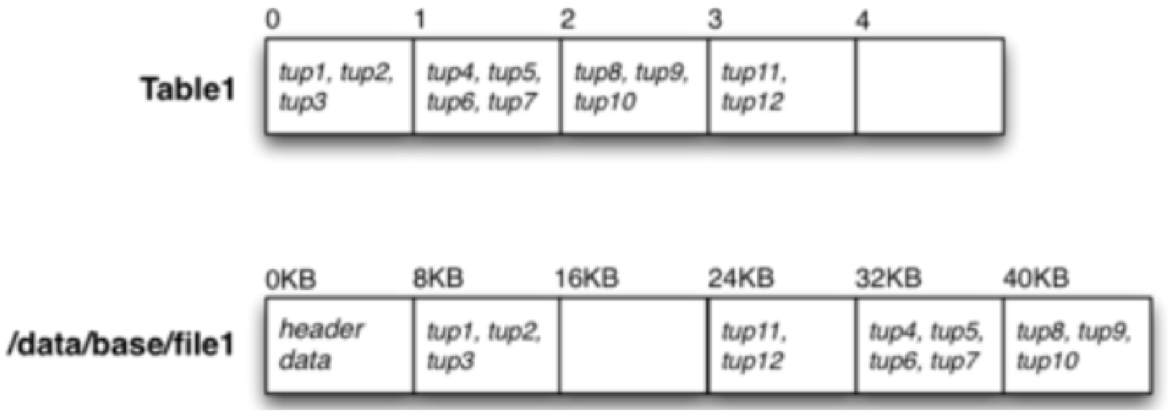
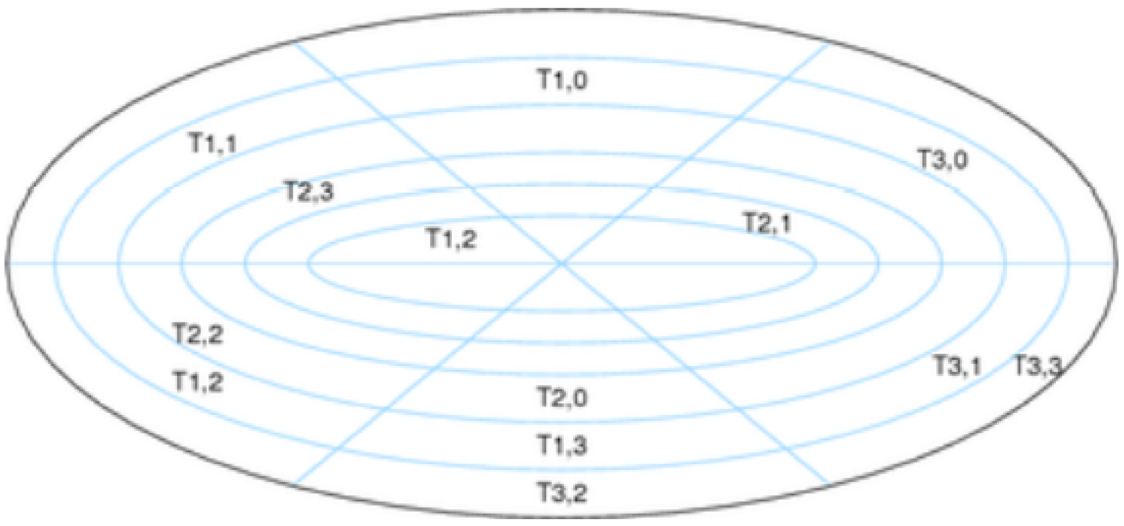


Table1
0
1
2
3
4
tup1, tup2, tup3
tup4, tup5, tup6, tup7
tup8, tup9, tup10
tup11, tup12
Index1
(key1,rid1) (key2,rid2) (key3,rid3)
(key4,rid4) (key5,rid5) (key6,rid6)
(key7,rid7) (key8,rid8) (key9,rid9)
(key10,rid10) (key11,rid11) (key12,rid12)
Table2
5
tup1, tup2
tup3
tup4, tup5
tup6, tup7
tup8

- File manager see data as:
  - maps table name + page index to file + offset

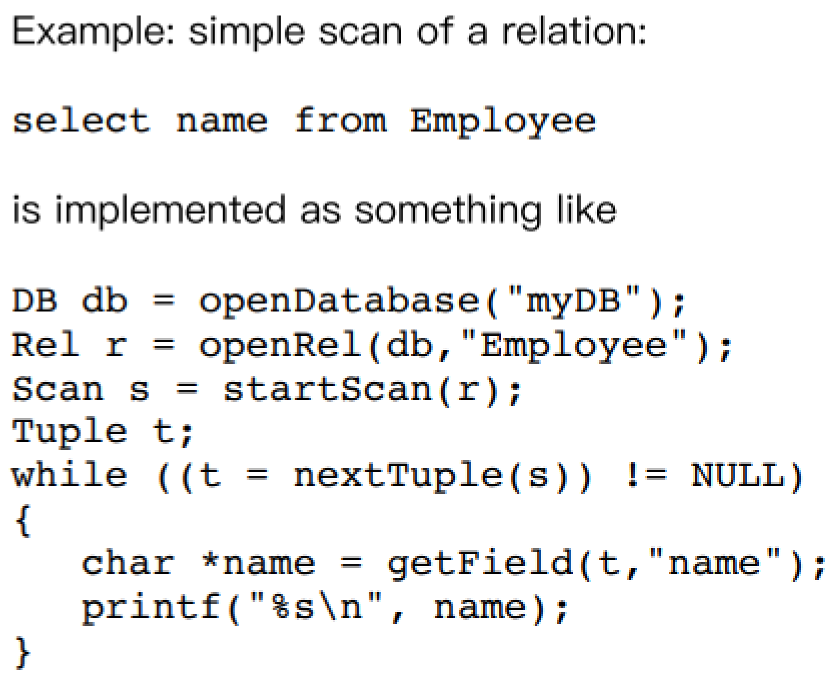


- Disk manager see data as:
  - fixed-size sectors of bytes, typically 512B
  - sectors are scattered across a disk device



1.2 storage manager interface

look how storage manager conduct a query (how it work):


Example: simple scan of a relation:

```
select name from Employee
```

is implemented as something like

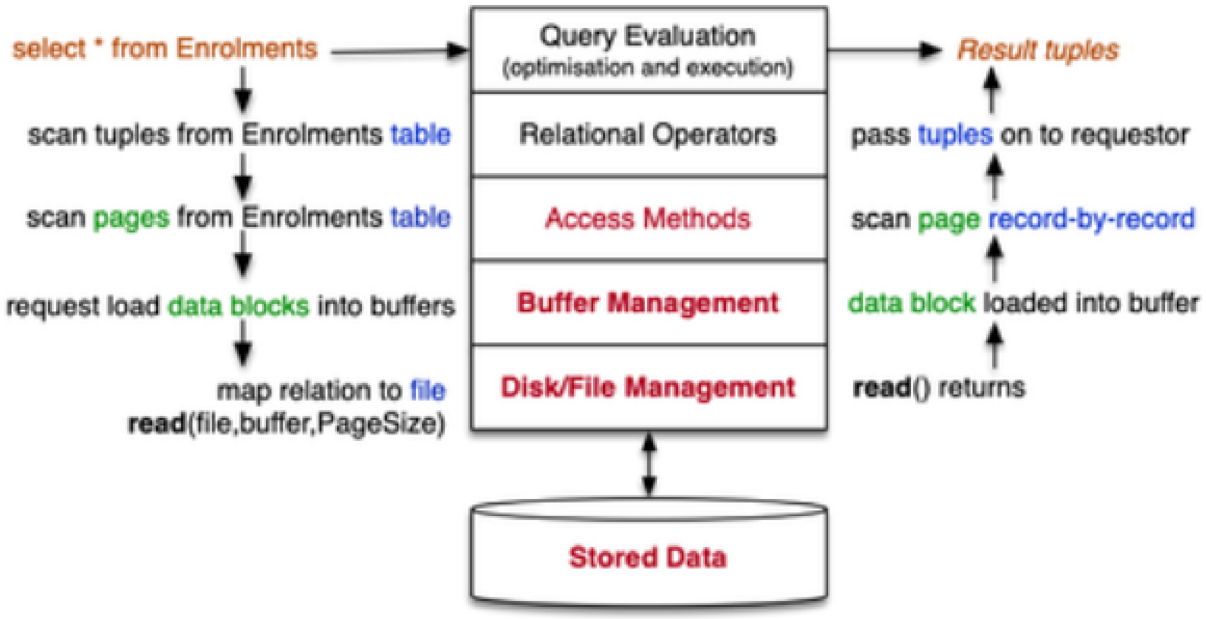
```
DB db = openDatabase("myDB");
Rel r = openRel(db,"Employee");
Scan s = startScan(r);
Tuple t;
while ((t = nextTuple(s)) != NULL)
{
   char *name = getField(t,"name");
   printf("%s\n", name);
}
```

the above shows several kinds of operations/mappings

- using database-name to access meta-data
- mapping a relation-name to a file

- performing page-by-page scans of files
- extracting tuples from pages
- extracting fields from tuples

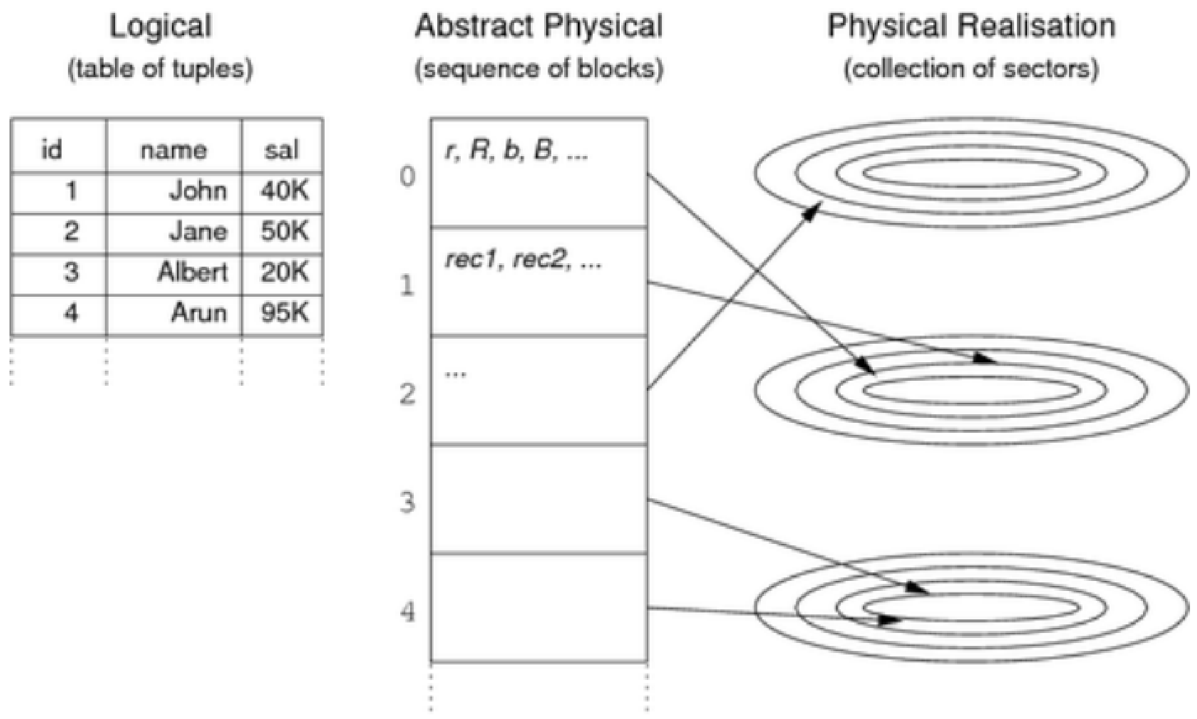
1.3 data flow in query evaluation


select * from Enrolments
Query Evaluation
(optimisation and execution)
Result tuples
scan tuples from Enrolments table
Relational Operators
pass tuples on to requestor
scan pages from Enrolments table
Access Methods
scan page record-by-record
request load data blocks into buffers
Buffer Management
data block loaded into buffer
map relation to file
read(file,buffer,PageSize)
Disk/File Management
read() returns
Stored Data

files in DBMS

Data sets (file) can be viewed at several levels of abstraction in DBMS:

- logical view:
- abstract physical: a file a sequence of fixed-size data blocks
- physical realisation:



1.4 storage technology

there are two methods of storage technology:

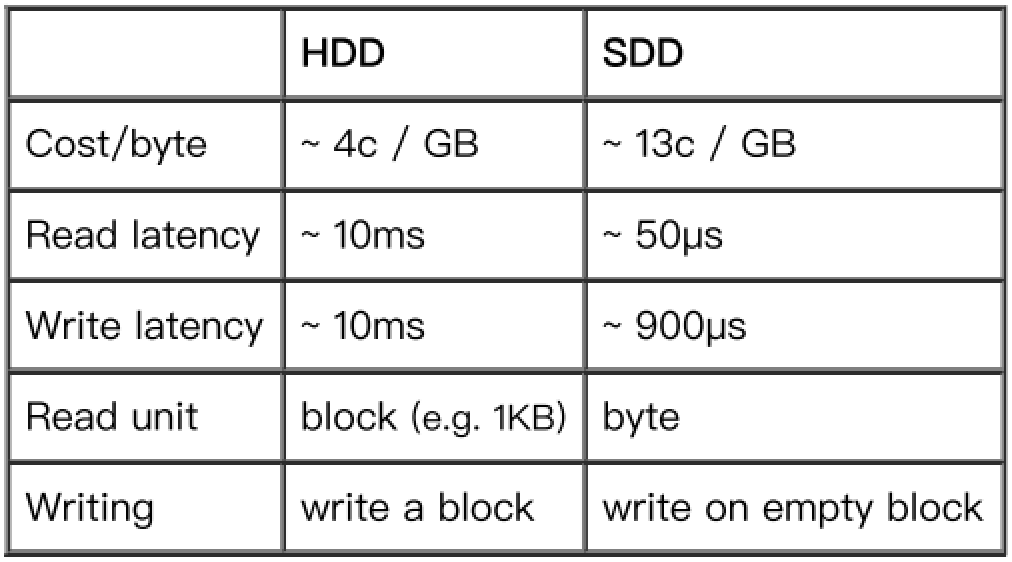
- computational storage: based on RAM
- bulk data storage: based on Disk

Computational storage: mainly use main memory(RAM)

bulk data storage: there are several kinds of bulk storage technology currently exitst:

- magnetic disks, optical disks, flash memory(SSD)

comaring HDD and SDD:



	HDD	SDD
Cost/byte	~ 4c / GB	~ 13c / GB
Read latency	~ 10ms	~ 50μs
Write latency	~ 10ms	~ 900μs
Read unit	block (e.g. 1KB)	byte
Writing	write a block	write on empty block

2. Disk Management

Aim:

- handles mapping from **database ID to disk address**(filesystem)
- tranfer blocks of data **between buffer pool and disk**
- also attempts to handle disk access error problem

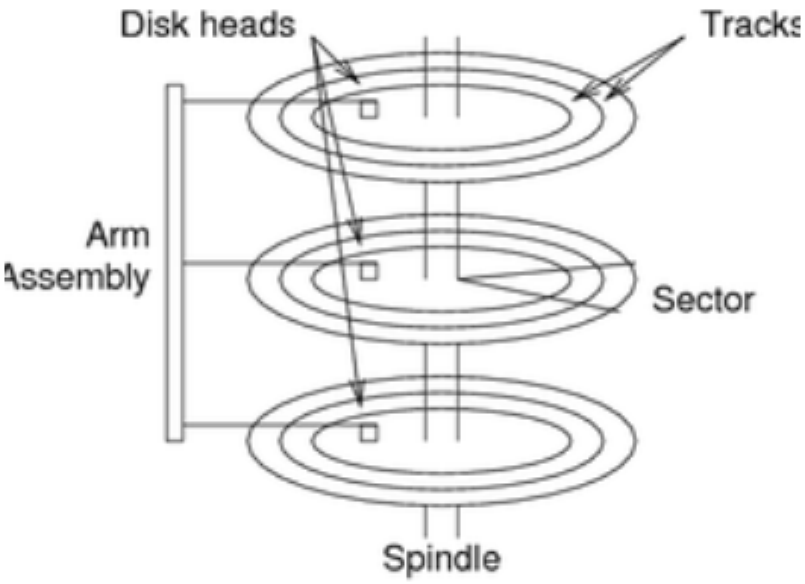
2.1 Disk Manager

there are severl interface about basic disk management:

- get_page: read disk block corresponding to PageId into buffer Page
- put_page: write block Page to disk block identified by PageId
- allocate_page: allocate a group of n disk blocks, optimised for sequential access
- deallocate_page: deallocate ….(just a reversal option of allocate)

2.2 Disk Technology

illustration of disk architecture:



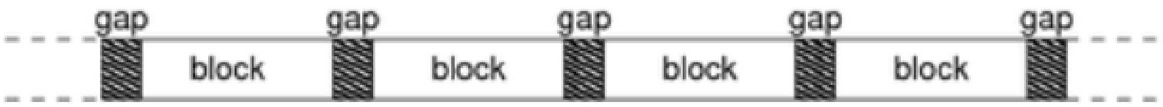
characteristics of disk:

- platters → tracks → sectors(blocks)
- transfer unit: 1block(e.g. 512B, 1KB, 2KB)

2.3 Disk access costs

- access time = seek time + rotational delay + tranfer time
- cost to write a block is similar to access time
- verify data on disk cost: add full rotation delay + block tranfer time

examples-1:



- 3.5 inches (8cm) diameter, 3600RPM, 1 surface (platter)
- 16MB usable capacity ($16 \times 2^{20} = 2^{24}$)
- 128 tracks, 1KB blocks (sectors), 10% gap between blocks
- #bytes/track = $2^{24}/128 = 2^{24}/2^{7} = 128KB$
- #blocks/track = $(0.9 * 128KB)/1KB = 115$
- seek time: min: 5ms (adjacent cyls), avg: 25ms max: 50ms

Time T_r to read one random block on disk #1:

- 3600 RPM = 60 revs per sec, rev time = 16.7 ms
- Time over blocks = $16.7 \times 0.9 = 15$ ms
- Time over gaps = $16.7 \times 0.1 = 1.7$ ms
- Transfer time for 1 block = 15/115 = 0.13 ms
- Time for skipping over gap = 1.7/115 = 0.01 ms

T_r = seek + rotation + transfer

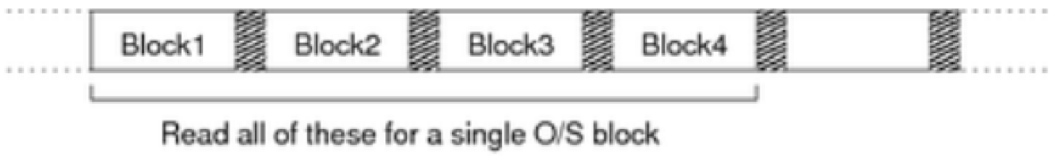
Minimum $T_r = 0 + 0 + 0.13 = 0.13$ ms

Maximum $T_r = 50 + 16.7 + 0.13 = 66.8$ ms

Average $T_r = 25 + (16.7/2) + 0.13 = 33.5\ ms$

examples-2:

if OS deals in 4KB blocks



$$T_r(4\text{–blocks}) = 25 + (16.7/2) + 4\times0.13 + 3\times0.01 = 33.9 \text{ ms}$$

$$T_r(1\text{–block}) = 25 + (16.7/2) + 0.13 = 33.5 \text{ ms}$$

compared with 1KB blocks, 4KB blocks accessing takes a bit more time

Note: sequential access reduces average block read cost significantly, but

- is limited to 115 blocks sequences
- is only used if blocks need to be sequentially scanned

- 3.5 inches (8cm) diameter, 3600RPM, 8 surfaces (platters)
- 8GB usable capacity ($8 \times 2^{30} = 2^{33}$ bytes)
- 8K (2^{13}) cylinders = 8k tracks per surface
- 256 sectors/track, 512 (2^9) bytes/sector

disk blocks addressing: 3bits(surface) + 13bits(cylinder) + 8bits(sector)

so if you use 32-bit OS, there will be 8bit left

2.4 Disk characteristics

disk has three most importantly characteristics:

- capacity
- access time
- reliability

so how to improve access time?

- minimise block tranfers: clustering, buffering, scheduled access

    - clustering: if there are two blocks are frequently access together, then we put them in the same block
- reduce seek: scheduling algorithm
- reduce latency:
- layout of data on disk (file organisation) can also assist

2.5 improve disk access

scheduled disk access

~~you can learn it from Operate System~~

2.5.1 disk layout

if data set is going to be acccessd in a pre-determined manner, arrange data on disk to minimise access time

E.g. if we want to conduct sequential scan, there are severl ways to reduce access time(by change disk layout)

- place subsequent blocks in same cylinder, different platters
- stagger so that as soon as block i was readed, block i+1 can be read
- once cylinder exhasuted, move to adjacent cylinder

2.5.2 improving writes

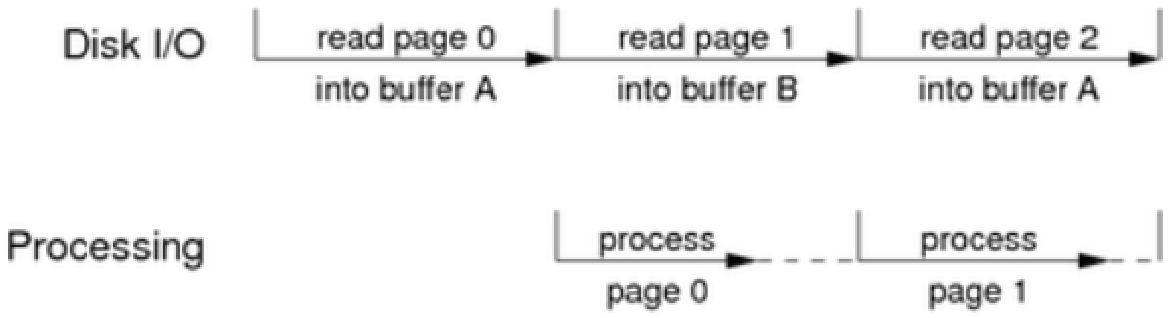
there are two mainly approaches to improving write option in disk:

- Nonvolatile write buffer
    - write all blocks to mem buffers in nonvolatile RAM
    - transfer to disk when idle
- Log disk
    - write all blocks to a special sequential access file-system
    - transfer to disk when idle

2.5.3 double buffering

double buffering exploits potential concurrency between disk and memory, whle reads/writes to disk are underway, other processing can be done.

Note: it just like the **I/O buffer management in OS-course**


Disk I/O
read page 0 into buffer A
read page 1 into buffer B
read page 2 into buffer A
Processing
process page 0
process page 1

let's compare single buffer and double buffer with a examples:

```
select sum(salary) from Employee
```

Note: we know that realtion = file = a sequence of n blocks A, B, C, D

- with a single buffer

```
read A into buffer then process buffer content
read B into buffer then process buffer content
read C into buffer then process buffer content
...
```

Costs:

- cost of reading a block = T_r
- cost of processing a block = T_p
- total elapsed time = $b.(T_r+T_p) = bT_r + bT_p$

Typically, $T_p < T_r$ (depends on kind of processing)

- with a double buffer

```
read A into buffer1
process A in buffer1
  and concurrently read B into buffer2
process B in buffer2
  and concurrently read C into buffer1
...
```

Costs:

- overall cost depends on relative sizes of T_r and T_p
- if $T_p \cong T_r$, total elapsed time = $T_r + bT_p$ (cf. $bT_r + bT_p$)

2.6 Multiple Disk Systems

essentially, multiple disks allow；

- improved reliablity by redundant storage of data
- reduced access cost by exploting parallelism

Note: we know that capacity increase naturally by adding multiple disks

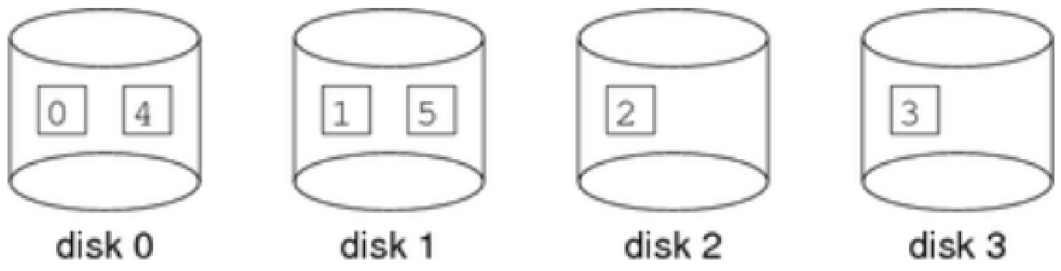
RAID: redundant arrays on independent disks, whose main purpose is to improve reading speed

2.6.1 RAID Level 0

uses striping to partition data for one file over several disks

E.g. for *n* disks, block *i* in the file is written to disk *(i mod n)*

Example: file with 6 data blocks striped onto 4 disks using *(pid mod 4)*



Increases capacity, improves data transfer rates, reduces reliability.

the operation will change accordingly：

```
writePage(PageId)
```

to

```
disk = diskOf(PageId,ndisks)
cyln = cylinderOf(PageId)
plat = platterOf(PageId)
sect = sectorOf(PageId)
writeDiskPage(disk, cyln, plat, sect)
```

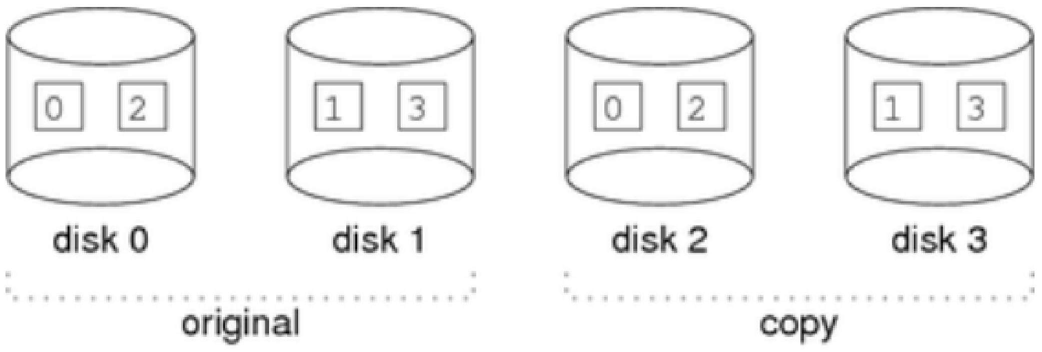
2.6.2 RALD Level 1

uses mirroring to store multiple copies of each blocks

Since disks can be read/written in parallel, transfer cost unchanged.

Multiple copies allows for single-disk failure with no data loss.

Example: file with 4 data blocks mirrored on two 2-disk partitions



Reduces capacity, improves reliability, no effect on data transfer rates.

the operation will change accordingly：

```
writePage(PageId)
```

to

```
n = ndisksInPartition
disk = diskOf(PageId,n)
cyln = cylinderOf(PageId)
plat = platterOf(PageId)
sect = sectorOf(PageId)
writeDiskPage(disk, cyln, plat, sect)
writeDiskPage(disk+n, cyln, plat, sect)
```

2.6.3 RAID levels 2-6

the higher levels of raid incorporate various combinations of:

- block/bits-level striping, mirroring, and error correcting codes（奇偶校验位）

the differences are primarily in:

- the kind of error correcting codes that are used

- where the ECC parity bits(奇偶校验位) are stored

RAID levels 2-5 can recover from failure in a single disk

RAID levels 6 can recover from smultaneous failures in two disks

3. Database Objects

the most important concept that we can learn from recent course is that:

- **how DB object are mapped to file system by Disk Manager?**

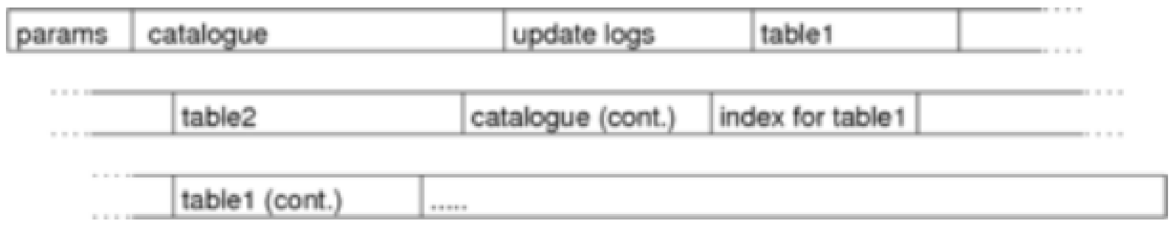
there are several DB objects:

- database
- parameters: global configuration information
- catalogue: meta-information describing database contents
- tables
- tuples
- indexes: access methods for efficient searching
- update logs: for handling rollback/recovery
- procedures: active elements

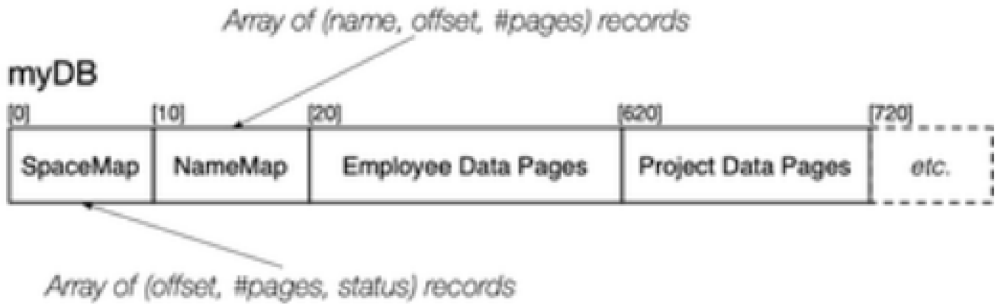
4. Storage Manager

4.1 Single-File storage manager

in single-file storage manager, objects are allocated to regions(segments) of the file



examples:

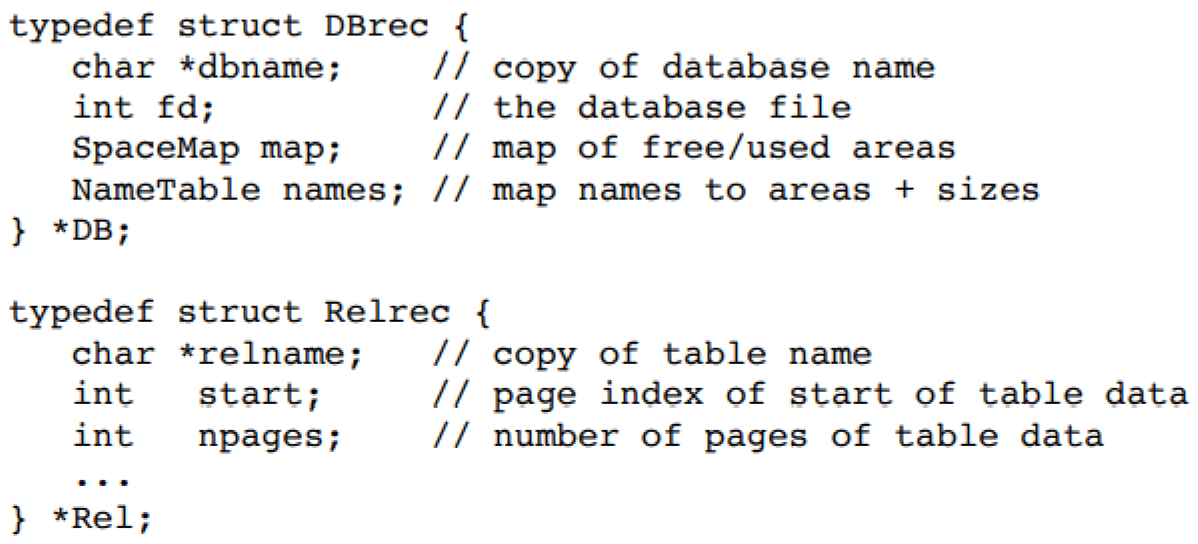


E.g.

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

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

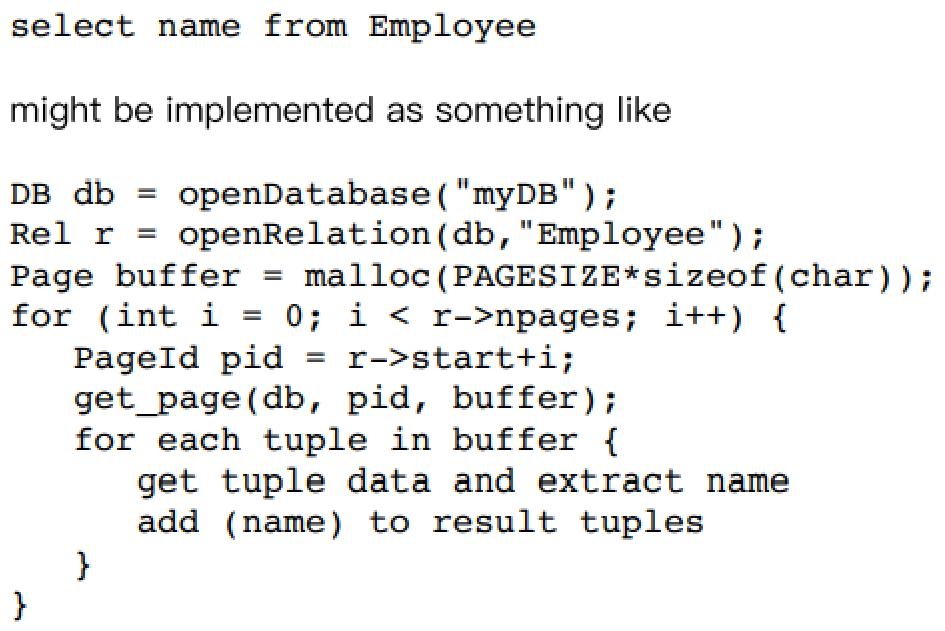
storage manager data structures for Database and tables:



```
typedef struct DBrec {
   char *dbname;    // copy of database name
   int fd;          // the database file
   SpaceMap map;    // map of free/used areas
   NameTable 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;
```

examples:

```
select name from Employee
```



```
select name from Employee
```

might be implemented as something like

```
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
   }
}
```

```
// 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);
}
```

```
// 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);
}
```

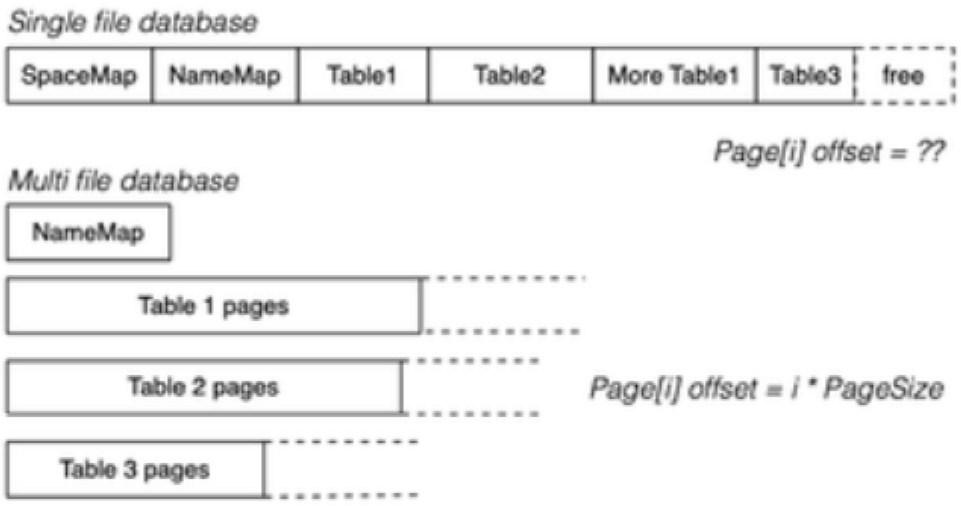
```
   // assume that Page = byte[PageSize]
   // assume that PageId = block number in file

   // 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);
   }
```

```
// 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
}
```

```
// 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

}
```

4.2 Multiple-File Disk Manager



if system use several files per table, PageId contains:

- relation identifier
- file identifier
- page number

4.3 Oracle File Structures

oracle uses five different kinds of files:

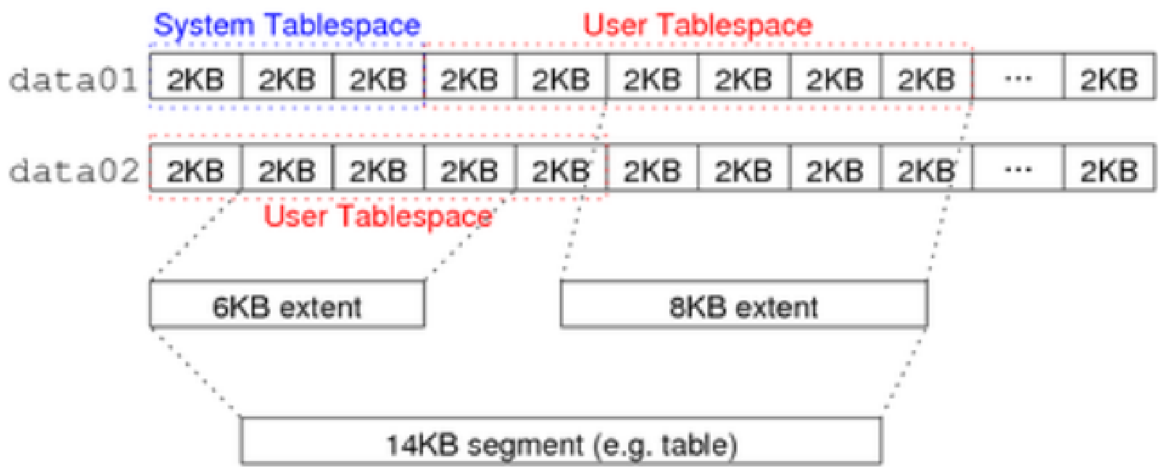
- data files: catalogue, tables, proceudures
- redo log files: update logs
- alert log files: record system events
- control files: configuration info
- archive files: off-line collected updates

layout of data within oracle file storage:

Tablespaces are logical units of storage, and every database object resides in exactly one tablespace

Units of storage within a tablespace:

- data block:
- extent
- segment



5. PostgreSQL Storage Manager

PostgreSQL has two basic kinds of files:

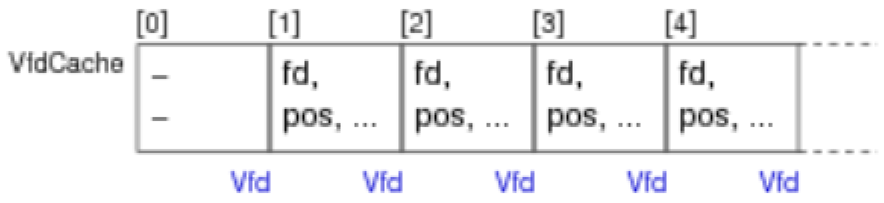
- **heap files** containing zdata(tuples)
- **index files** containing index entries

5.1 file descriptor pool

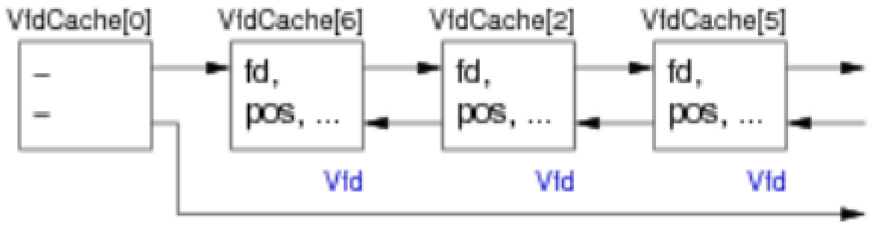
unix has limits on the number of concurrently open files, and PostgreSQL maintain a pool of open file descriptors.

Vfd: Virtual file descriptors

- physically stored in dynamically–allocated array



- also arranged into list by recency–of–use



the following is the defination of vfd:

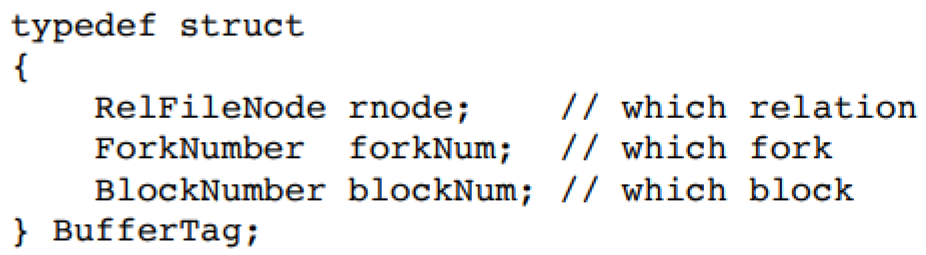
```
typedef struct vfd
{
    s_short  fd;              // current FD, or VFD_CLOSED if none
    u_short  fdstate;         // bitflags for VFD's state
    File     nextFree;        // link to next free VFD, if in freelist
    File     lruMoreRecently; // doubly linked recency-of-use list
    File     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;
```

5.2 File Manager

for magnetic disk storage manager:

- manage it own pool of file descriptors
- each one represents an open relation file (Vfd)
- may use several Vfds to access data, if file > 2GB
- manage mapping from PageId to file + offset

Notes: PageId structure is shown below



```
typedef struct
{
    RelFileNode rnode;    // which relation
    ForkNumber  forkNum;  // which fork
    BlockNumber blockNum; // which block
} BufferTag;
```

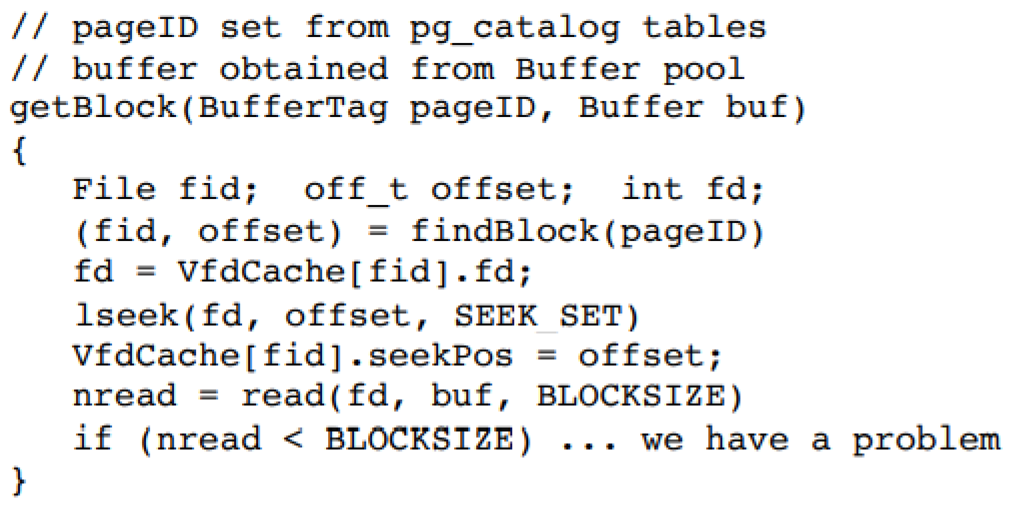
PostgreSQL stores each table in dir PGDATA/pg_database.oid in several files

emmm, there are multiple part, like Data files, Free space map, visibility map

- Data files
  - Data files includes a sequence of blocks/Pages, and each of it contains tuple-data and admin-data, whose size is typically 8KB (maxsize is 1GB, limited by Unix)
- Free space map
  - indicates where free spaces is in data pages
- visibility map
  - indicates pages where all tuples are visiable(accessible to all currently active trasactions)

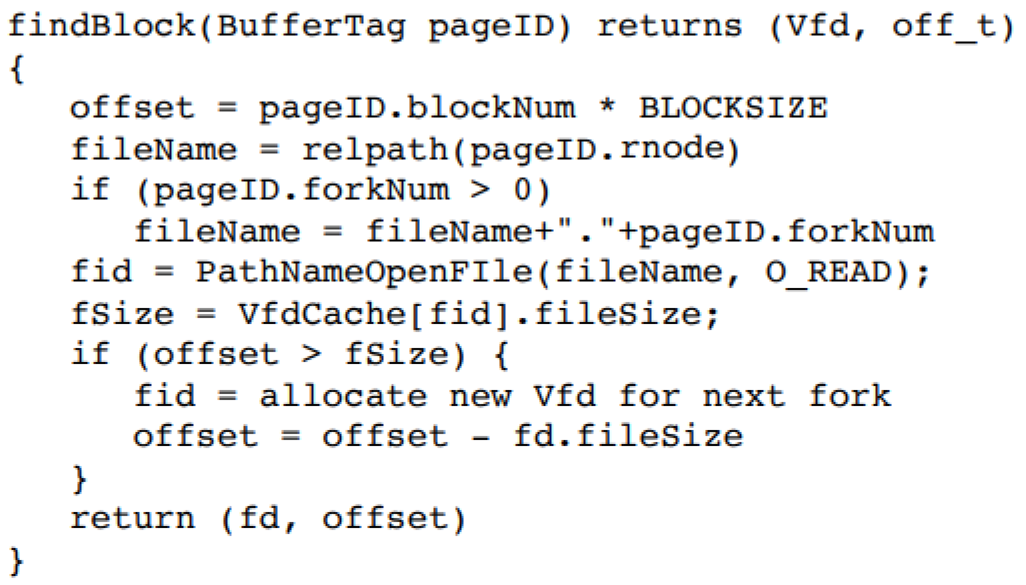
access to a block of data proceeds as follows:

- getBlock(BufferTag pageID, Buffer buf)



```
// pageID set from pg_catalog tables
// buffer obtained from Buffer pool
getBlock(BufferTag pageID, Buffer buf)
{
   File fid;  off_t offset;  int fd;
   (fid, offset) = findBlock(pageID)
   fd = VfdCache[fid].fd;
   lseek(fd, offset, SEEK_SET)
   VfdCache[fid].seekPos = offset;
   nread = read(fd, buf, BLOCKSIZE)
   if (nread < BLOCKSIZE) ... we have a problem
}
```

- findBlock(BufferTag pageID) returns (Vfd, off_t)



```
findBlock(BufferTag pageID) returns (Vfd, off_t)
{
   offset = pageID.blockNum * BLOCKSIZE
   fileName = relpath(pageID.rnode)
   if (pageID.forkNum > 0)
      fileName = fileName+"."+pageID.forkNum
   fid = PathNameOpenFIle(fileName, O_READ);
   fSize = VfdCache[fid].fileSize;
   if (offset > fSize) {
      fid = allocate new Vfd for next fork
      offset = offset - fd.fileSize
   }
   return (fd, offset)
}
```

6. Buffer Pool

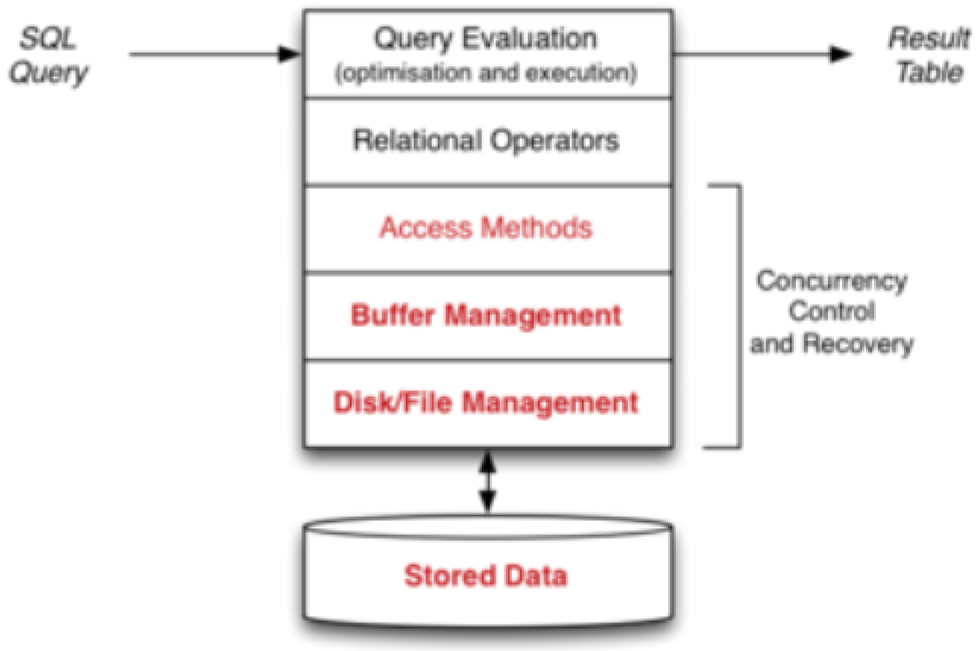
6.1 Buffer manager

buffer manager’s aim is :

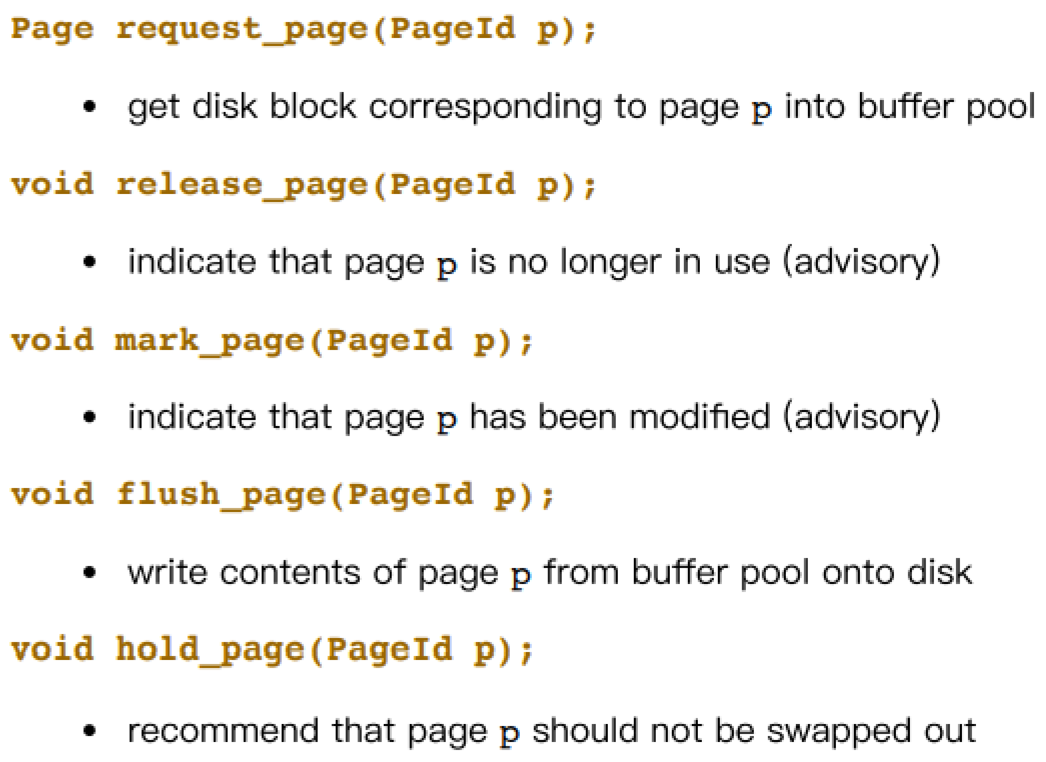
- minimise traffic between disk and memory via caching
- maintain a shared buffer pool in mem

Buffer Pool:

- collection of page slots(aka frames)
- each frame could be filled with a copy of data from a disk block



Buffer manager provides multiple interface:



```
Page request_page(PageId p);
```

- get disk block corresponding to page p into buffer pool

```
void release_page(PageId p);
```

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

```
void mark_page(PageId p);
```

- indicate that page p has been modified (advisory)

```
void flush_page(PageId p);
```

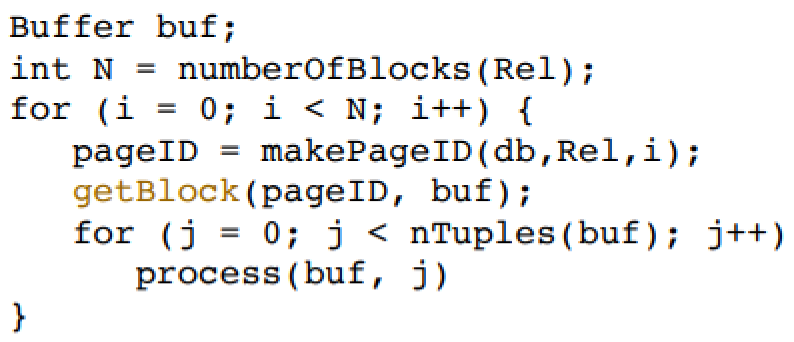
- write contents of page p from buffer pool onto disk

```
void hold_page(PageId p);
```

- recommend that page p should not be swapped out

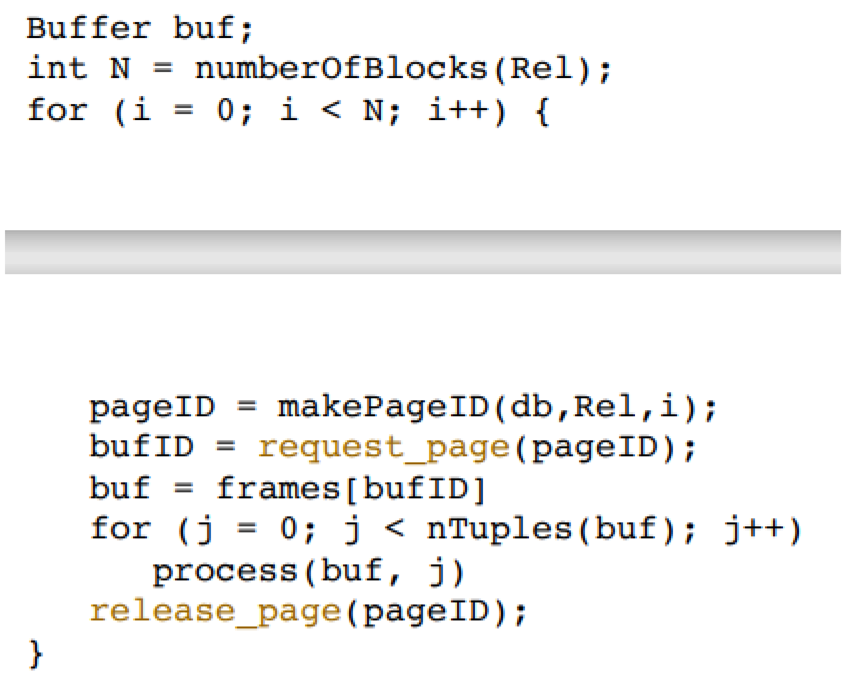
6.2 Buffer Pool Usage

- how scans are performed without Buffer pool



```
Buffer buf;
int N = numberOfBlocks(Rel);
for (i = 0; i < N; i++) {
   pageID = makePageID(db,Rel,i);
   getBlock(pageID, buf);
   for (j = 0; j < nTuples(buf); j++)
      process(buf, j)
}
```

- how scans are performed with Buffer pool



```
Buffer buf;
int N = numberOfBlocks(Rel);
for (i = 0; i < N; i++) {
```

```
   pageID = makePageID(db,Rel,i);
   bufID = request_page(pageID);
   buf = frames[bufID]
   for (j = 0; j < nTuples(buf); j++)
      process(buf, j)
   release_page(pageID);
}
```

6.3 Buffer Pool Data

Buffer Pool is a fixed-size, mem-resident collection of frames, and for each frames, we need to know that:

- whether it is currently used
- which page it contains
- has been modified?
- how many trasaction are currently use this frame
- time-stamp for most recent access

6.4 Requesting Pages

call from client: request_page(pid)

this process is like Page-relplacement in OS

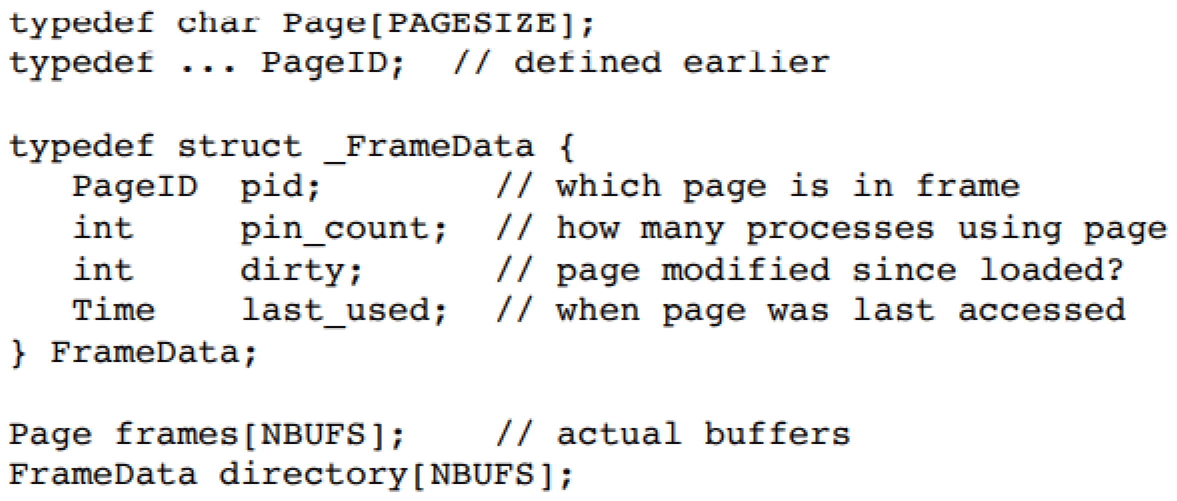
- if page pid is already in buffer pool, just use it
- or we need to read a page from disk into a new frame

6.5 Releasing Pages

call from client: release_page(pid)

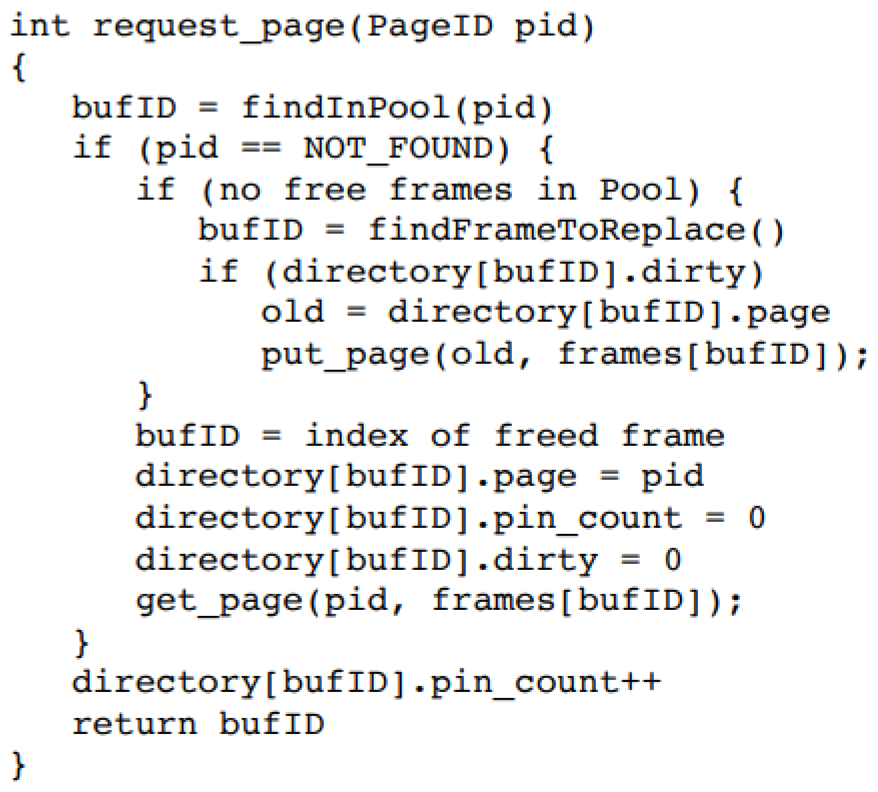
6.6 Buffer Pool Implementation

buffer pool data structures:



```
typedef char Page[PAGESIZE];
typedef ... PageID;  // defined earlier

typedef struct _FrameData {
   PageID  pid;        // which page is in frame
   int     pin_count;  // how many processes using page
   int     dirty;      // page modified since loaded?
   Time    last_used;  // when page was last accessed
} FrameData;

Page frames[NBUFS];    // actual buffers
FrameData directory[NBUFS];
```

implementation of request_page()



```
int request_page(PageID pid)
{
   bufID = findInPool(pid)
   if (pid == NOT_FOUND) {
      if (no free frames in Pool) {
         bufID = findFrameToReplace()
         if (directory[bufID].dirty)
            old = directory[bufID].page
            put_page(old, frames[bufID]);
      }
      bufID = index of freed frame
      directory[bufID].page = pid
      directory[bufID].pin_count = 0
      directory[bufID].dirty = 0
      get_page(pid, frames[bufID]);
   }
   directory[bufID].pin_count++
   return bufID
}
```

6.7 Page Replacement Policies

just like OS, include LRU, FIFO, etc

6.8 Buffer Pool Data Objects

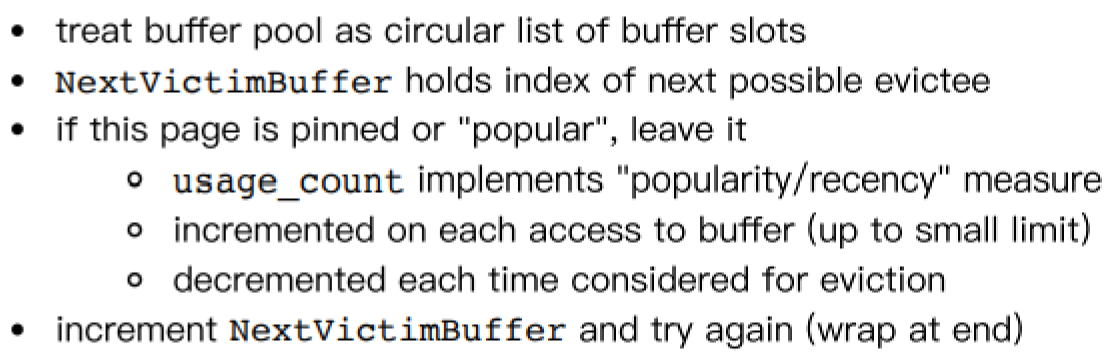
- BufferDescriptors: array of strut describe buffers
- Buffer: index into BufferDescriptors
- BufMgrLock: global lock on buffer pool
- BufferTag

6.9 Buffer Pool Function

- BufferGetPage:
- BufferIsPinned: check whether this backend holds a pin on buffer
- CheckPointBuffers: for recoveray

6.10 Clock-sweep replacement strategy

PostgreSQL use clock-sweep replacement strategy



- treat buffer pool as circular list of buffer slots
- `NextVictimBuffer` holds index of next possible evictee
- if this page is pinned or "popular", leave it
    - `usage_count` implements "popularity/recency" measure
    - incremented on each access to buffer (up to small limit)
    - decremented each time considered for eviction
- increment `NextVictimBuffer` and try again (wrap at end)

7. Record/Tuple Management

7.1 views of data

the disk and buffer manager provide the following view:

- data is a sequence of fixed-size blocks(pages)
- block can be randomly access via a PageID

but for Database applications, they view data as:

- a collection of records(tuples)
- records could be accessed via RecordID(RID)

tips: records are also called tuples, items, rows

what is the representation of a relation?

the logical view of a relation:

- a named and ordered sequence of tuples
- with some additional access method data structures

the physical representation of a relation:

- an indexed sequence of pages in one or more files

- along with data structure to manage the records

what is the representation of a tuple?

- Record: physical view of a table row, a sequence of bytes
- Tuple: logical view of a table row, a collection of typed fields

7.2 Record Management

aims: provide mapping from RecordID to Tuple

Page-level operations:

- operation to access records from a page
- operation to make changes to records in a page

Tuple-level operations:

- extract fields from a tuple
- set the value of the field of a tuple

Relation-level operations:

- fetch the tuple by relation-id
- return reference to record first Tuple in a page
- return the next-one

Example Query:

```
select name from Employee
```

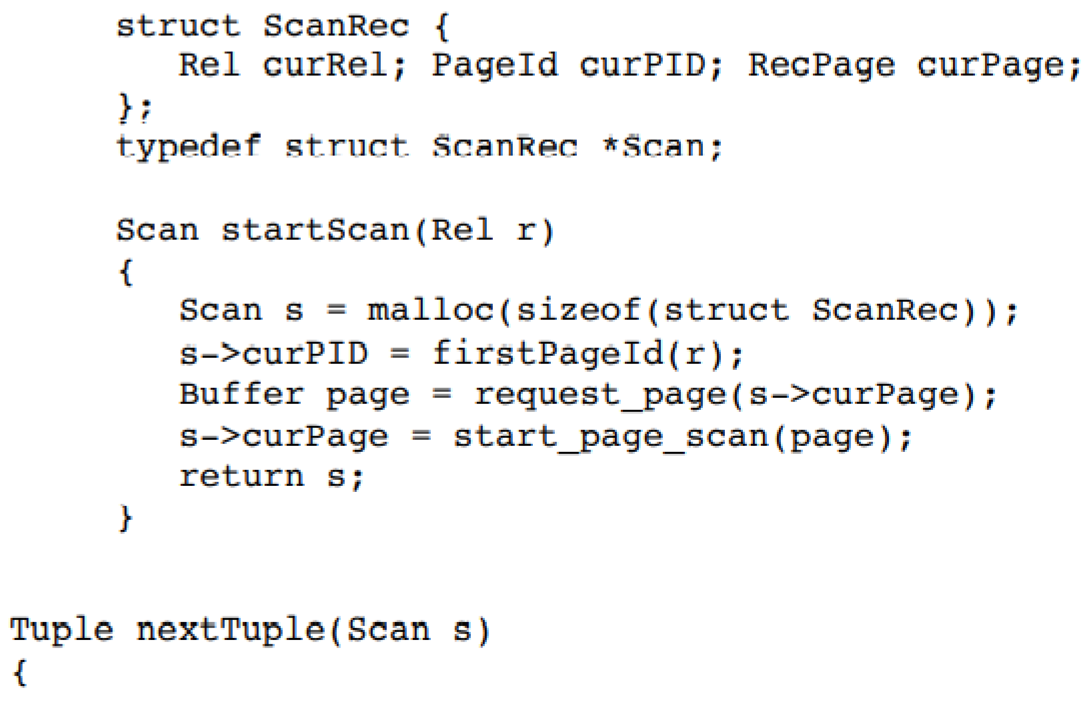
implemented as:

```
DB db = openDatabase("myDB");
Rel r = openRel(db,"Employee");
Scan s = startScan(r);
Tuple t;
while ((t = nextTuple(s)) != NULL)
{
   char *name = getField(t,"name");
   printf("%s\n", name);
}
```

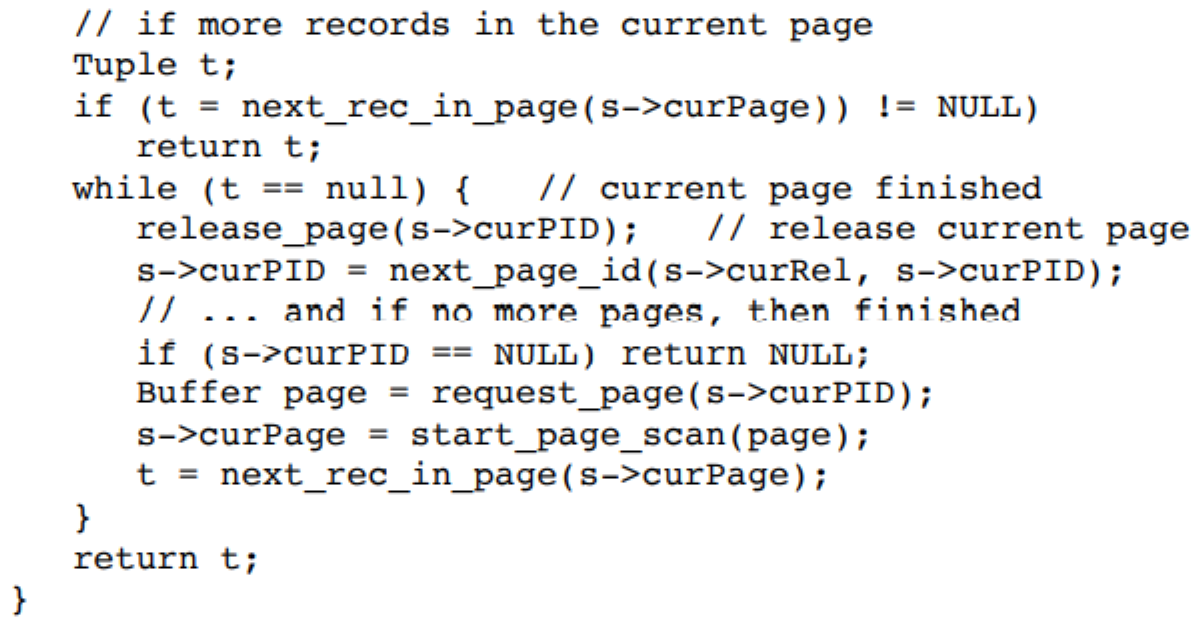
conceptually, the scanning implementation is simple:

```
// maintain "current" state of scan
struct ScanRec { Rel curRel; RecId curRec };
typedef struct ScanRec *Scan;

Scan startScan(Rel r) {
   Scan s = malloc(sizeof(struct ScanRec));
   s->curRec = firstRecId(r);
   return s;
}

Tuple nextTuple(Scan s) {
   Tuple t = fetchTuple(s->curRec);
   s->curRec = nextRecId(r,s->curRec);
   return t;
}
```

the real implementation relies on the buffer manager:



```
      struct ScanRec {
         Rel curRel; PageId curPID; RecPage curPage;
      };
      typedef struct ScanRec *Scan;

      Scan startScan(Rel r)
      {
         Scan s = malloc(sizeof(struct ScanRec));
         s->curPID = firstPageId(r);
         Buffer page = request_page(s->curPage);
         s->curPage = start_page_scan(page);
         return s;
      }

Tuple nextTuple(Scan s)
{
```



```
   // if more records in the current page
   Tuple t;
   if (t = next_rec_in_page(s->curPage)) != NULL)
      return t;
   while (t == null) {   // current page finished
      release_page(s->curPID);   // release current page
      s->curPID = next_page_id(s->curRel, s->curPID);
      // ... and if no more pages, then finished
      if (s->curPID == NULL) return NULL;
      Buffer page = request_page(s->curPID);
      s->curPage = start_page_scan(page);
      t = next_rec_in_page(s->curPage);
   }
   return t;
}
```

7.3 Record Identifiers

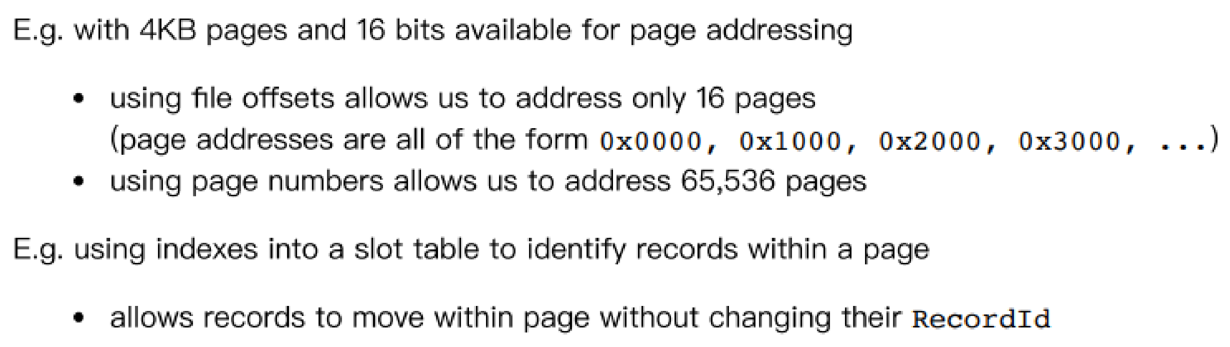
RecordID has two components:

- page number (which page it contained in)
- slot number (where it located within the page)
- (optional, if there are multiple files for a relation) file number (which file the page contained in)

Notes: PostgreSQL provide a unique OID for **every row** in the database

RecordID components are implement as **table indexs**

- (like using 16bit as address) rather than absolute offsets, casuse it can save space and allow flexibility in storage management


E.g. with 4KB pages and 16 bits available for page addressing

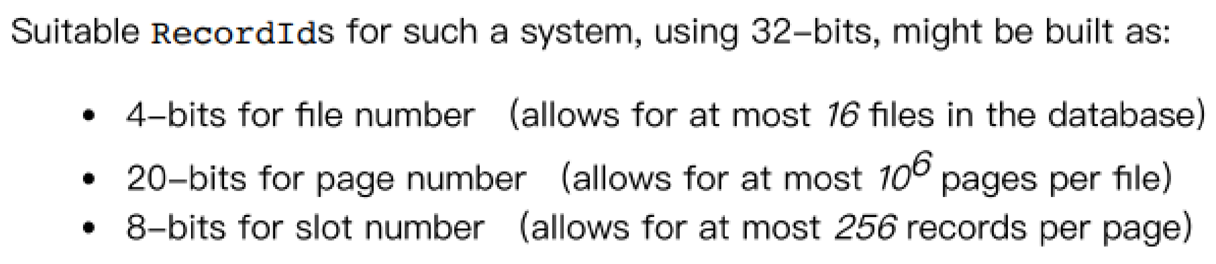
- using file offsets allows us to address only 16 pages (page addresses are all of the form `0x0000, 0x1000, 0x2000, 0x3000, ...`)
- using page numbers allows us to address 65,536 pages

E.g. using indexes into a slot table to identify records within a page

- allows records to move within page without changing their `RecordId`

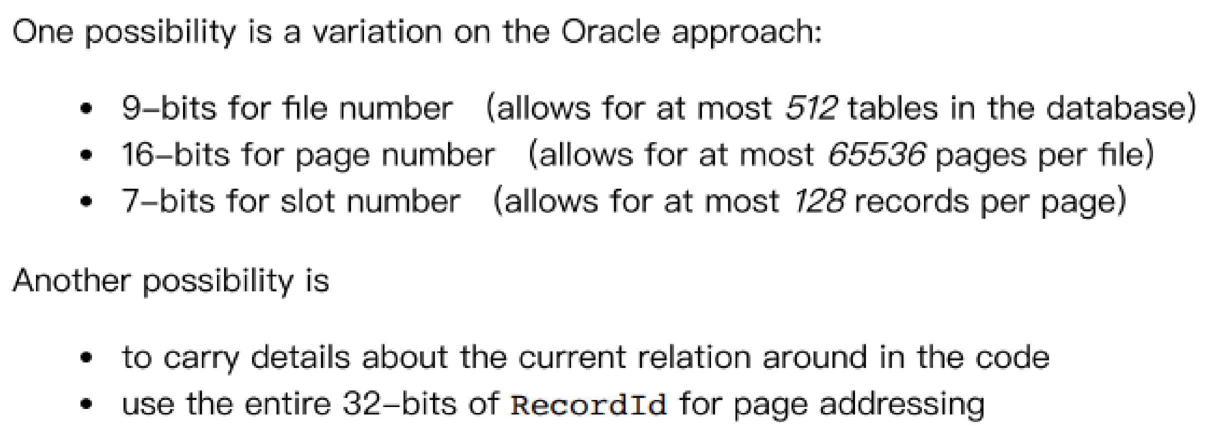
RecordID structure:

- example-1 Oracle:


Suitable `RecordIds` for such a system, using 32–bits, might be built as:

- 4–bits for file number (allows for at most *16* files in the database)
- 20–bits for page number (allows for at most 10^6 pages per file)
- 8–bits for slot number (allows for at most *256* records per page)

- example-2 MiniSQL:


One possibility is a variation on the Oracle approach:

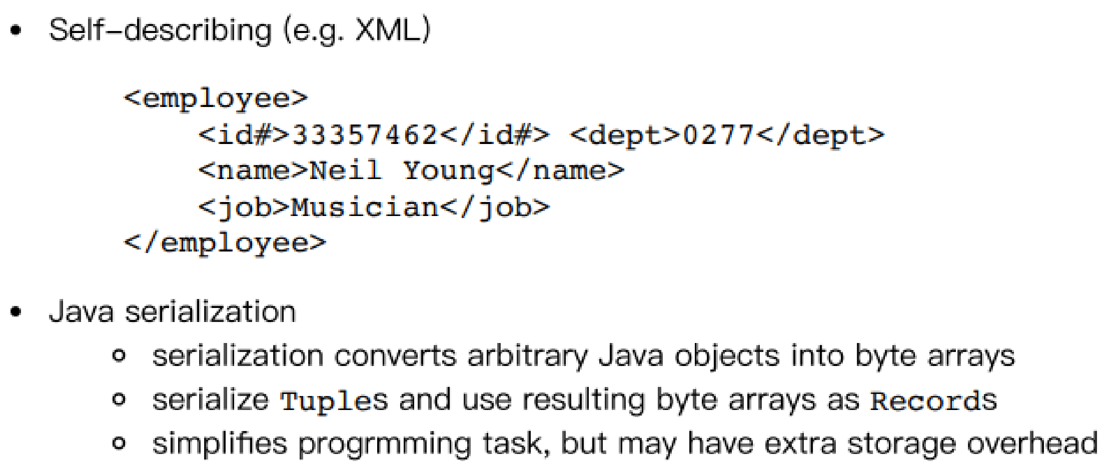
- 9–bits for file number (allows for at most *512* tables in the database)
- 16–bits for page number (allows for at most *65536* pages per file)
- 7–bits for slot number (allows for at most *128* records per page)

Another possibility is

- to carry details about the current relation around in the code
- use the entire 32–bits of `RecordId` for page addressing

Record Formats:

- fixed-length Records
    - advantages:
        - don't need slot dir in page (compute record offset as number*size)
        - intra-page mem management is simplified
    - disadvantages:
        - leads to considerable space wastage a
- variable-length Records:
    - there are two mainly encoding schemes: mainly are **(marks, length, delimeters)**


- Self–describing (e.g. XML)

```
<employee>
    <id#>33357462</id#> <dept>0277</dept>
    <name>Neil Young</name>
    <job>Musician</job>
</employee>
```

- Java serialization
    - serialization converts arbitrary Java objects into byte arrays
    - serialize `Tuples` and use resulting byte arrays as `Records`
    - simplifies progrmming task, but may have extra storage overhead

- advantages:
    - less-space-wastage
    - flexibility
- disadvantages:
    - more complex intra-page space management algorithms
    - potential for space-fragmentation within pages

what can we do if a record's size over page size?

- spanned records:
    - it means span a record between two pages
        - if can maximizes the utilization of space, but we require to access one more page
- store large data values outside record in separate file

7.4 Converting Records to Tuples

Records is set of bytes, if we need some information to interpret bytes, these information are stored at:

- maybe contained in a schema in the DBMS catalogue
- may be stored in the header for the data files
- may be stored partly in the record and partly in a DTD

to convert a record to tupe, we need to know:

- start location of each fields in the byte array
- number of bytes in each field
- type of value in each field

this leads to two struct: **FieldDesc and RelnDesc:**

```
typedef struct {
    short offset;   // index of starting byte
    short length;   // number of bytes
    Types type;     // reference to Type data
} FieldDesc;
typedef struct {
    char      *relname;   // relation name
    ushort     nfields;   // # of fields
    FieldDesc fields[]; // field descriptors
} RelnDesc;
```

example:

```
FieldDesc fields[] = malloc(4*sizeof(FieldDesc);
fields[0] = FieldDesc(0,4,INTEGER);
fields[1] = FieldDesc(4,20,VARCHAR);
fields[2] = FieldDesc(24,10,CHAR);
fields[3] = FieldDesc(34,4,NUMBER);
```

7.5 Defining Tuples

```
typedef struct {
    Record    data;      // pointer to data
    ushort    nfields;   // # fields
    FieldDesc fields[]; // field descriptions
} Tuple;
```

and the procedure of how the record→tuple mapping might occur:

```
Tuple mkTuple(RelnDesc schema, Record record)
{
    int i, pos = 0;
    int size = sizeof(Tuple) +
               (nfields-1)*sizeof(FieldDesc);
    Tuple *t = malloc(size);
    t->data = record;
    t->nfields = schema.nfields;
    for (i=0; i < schema.nfields; i++) {
        int len = record[pos++];
        t->fields[i].offset = pos;
        t->fields[i].length = len;
        // could add checking for over-length fields, etc.
        t->fields[i].type = schema.fields[i].type;
        pos += length;
    }
    return t;
}
```

7.6 Page Formats

7.7 Storage Utilisation

7.8 Overflows

8. Representing Database Objects

RDBMS manage different kinds of objects；

- databases, schemas, tablespaces
- relations/tables, attributes, tuples/records
- constraints, assertions
- views, stored procedures, triggers, rules

many objects have name (and in PostgreSQL, all have OIDS)

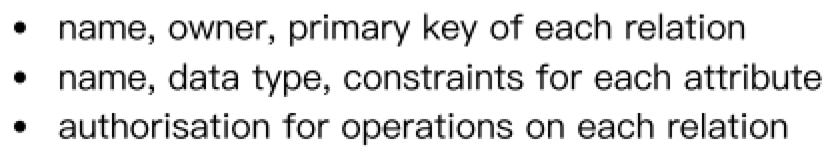
so how are the different types of objects represented?

how do we go from a name(or OID) to bytes stored on disk?

Top-level “objects” in typical SQL standard databases:

- catalog: SQL terminology for a database
  - users connect to a database
  - sets context for interaction
- schema: collection of DB object definations
  - each schema is defined with a database/catalog
  - used for name-space management
- tablespace: collection of DB files
  - each file contains DB objects from multiple catalog/schemas
  - used for file-space management
- (PostgreSQL)cluster: a server managing a set of DBs

a relation needs:



- name, owner, primary key of each relation
- name, data type, constraints for each attribute
- authorisation for operations on each relation

all of the information above is stored in the system catalog

(interesting, the system catalog is also stored as tables)

8.1 System Catalog

catalog structure contains information such as:

```
Users(id:int, name:string, ...)

Databases(id:int, name:string, owner:ref(User), ...)

Schemas(id:int, name:string, owner:ref(User), ...)

Types(id:int, name:string, defn:string, size:int, ...)

Tables(id:int, name:string, owner:ref(User),
                                inSchema:ref(Schema), ...)

Attributes(id, name:string, table:ref(Table),
                           type:ref(Type), pkey:bool, ...)
etc. etc.
```

the catalog is manipulated by a range of SQL operations:

- `create` *Object* `as` *Definition*
- `drop` *Object* ...
- `alter` *Object*   *Changes*
- `grant` *Privilege* `on` *Object*

where objects is one of table, view, function, trigger, schema

examples:

```
create table ABC (
    x integer primary key,
    y integer
);
```

this will produce a set of changes like:

```
userID := current_user();
schemaID := current_schema();
tabID := nextval('tab_id_seq');
select into intID id
from Types where name='integer';
insert into Tables(id,name,owner,inSchema,...)
  values (tabID, 'abc', userID, schema, ...)
attrID := nextval('attr_id_seq');
insert into Attributes(id,name,table,type,pkey,...)
    values (attrID, 'x', tabID, intID, true, ...)
attrID := nextval('attr_id_seq');
insert into Attributes(id,name,table,type,pkey,...)
    values (attrID, 'y', tabID, intID, false, ...)
```

8.3 Users/Groups

8.4 Tables

8.5 Functions

8.6 Types