COMP9315: Storage: Devices, Files, Pages, Tuples, Buffers, Catalogs

---

Storage Management

Storage Management

Aims of storage management in DBMS:

- provide view of data as collection of pages/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
- basis for file structures used by access methods

... Storage Management

The storage manager provides mechanisms for:

- representing database objects during query execution
  - `DB` (handle on an authorised/opened database)
  - `Rel` (handle on an opened relation)
  - `Page` (memory buffer to hold contents of disk 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`)

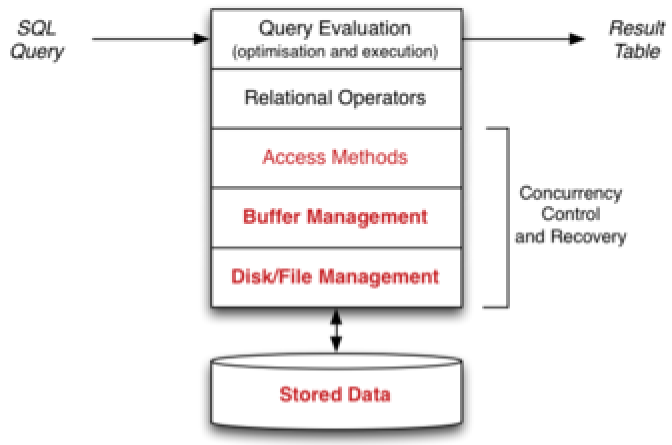
... Storage Management

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 + Index`
  - where `Index` gives location of tuple within page

... Storage Management

Levels of DBMS related to storage management:



... Storage Management

6/247

Topics to be considered:

- Disks and Files
  - performance issues and organisation of disk files
- Buffer Management
  - using caching to improve DBMS system throughput
  - involves discussion of 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

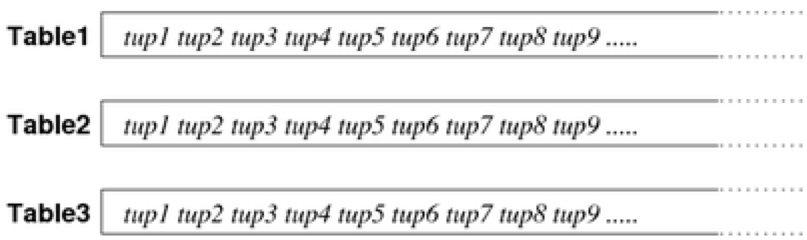
Each topic will be illustrated by its implementation in PostgreSQL.

Views of Data

7/247

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

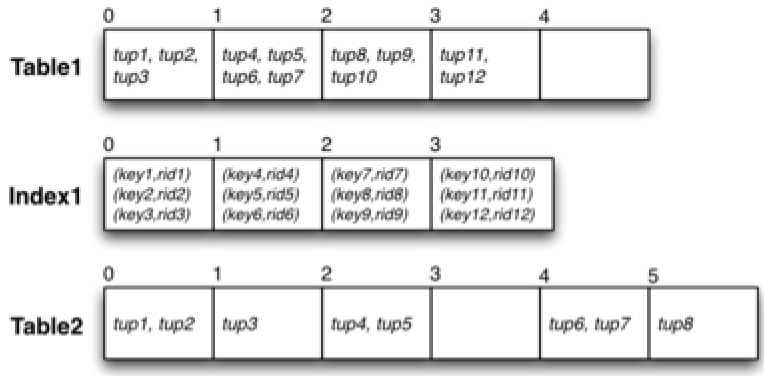


... Views of Data

8/247

Relational operators and access methods see data as

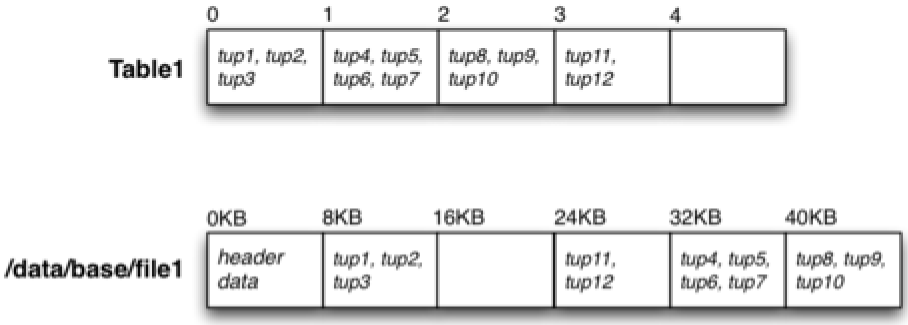
- sequence of fixed-size pages, typically 1KB to 8KB
- where each page contains tuple or index data



... Views of Data

File manager sees both DB objects and file store

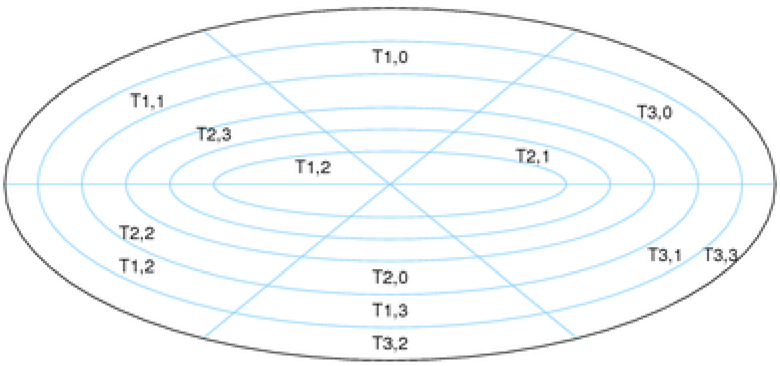
- maps table name + page index to file + offset



... Views of Data

Disk manager sees data as

- fixed-size sectors of bytes, typically 512B
- sectors are scattered across a disk device



On typical modern databases, handled by operating system filesystem.

Storage Manager Interface

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 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);
}
```

... Storage Manager Interface

12/247

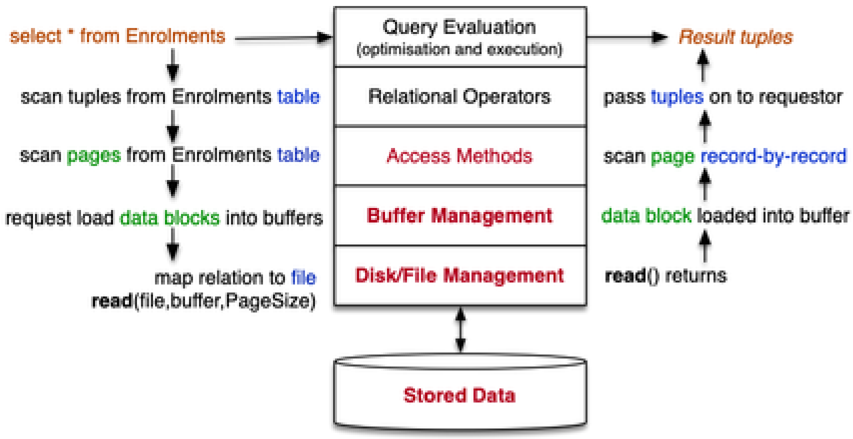
The above shows several kinds of operations/mappings:

- using a 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

The DBMS storage manager provides all of these, broken down across several modules.

Data Flow in Query Evaluation

13/247



... Data Flow in Query Evaluation

14/247

Notes on typical implementation strategies:

- addresses implemented as partitioned `ints`
  (e.g. `PageId = (FileNum<24)||PageNum`)
- addresses replaced by multiple arguments
  (e.g. `get_page(r,i,buf)` rather than `get_page(pid,buf)`)
- types such as `DB` and `Rel` are dynamic `structs`

```
struct RelRec { int fd; int npages; int blksize; }
typedef struct RelRec *Rel;
```

Files in DBMSs

15/247

Data sets can be viewed at several levels of abstraction in DBMSs.

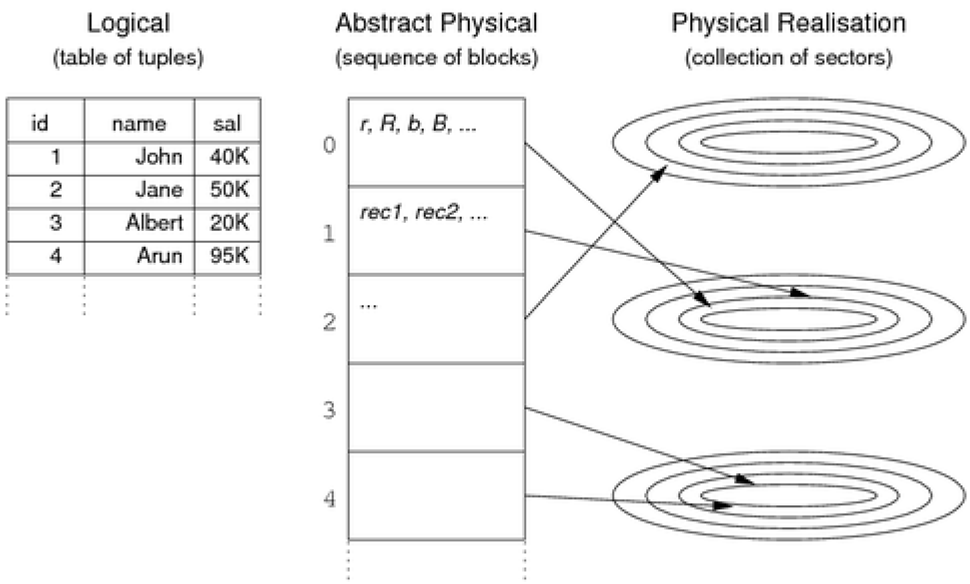
Logical view: a file is a named collection of data items (e.g. a table of tuples)

Abstract physical view: a file is a sequence of fixed-size data blocks.

Physical realisation: a collection of sectors scattered over *≥1* disks.

The abstraction used for managing this: `PageId`.

... Files in DBMSs



... Files in DBMSs

Two possibilities for DBMS disk managers to handle data:

- deal with the physical realisation (via disk partition)
  - the DBMS implementor has to write own disk management
  - gives fine-grained control for performance-critical systems
  - Oracle (at least) *can* execute from a raw Unix disk partition
- deal with the abstract physical view (via OS filesystem)
  - tables, indexes, etc. are represented as regions of *≥1* files
  - disk manager handles mapping from logical → abstract physical
  - different DBMSs use substantially different mappings

File System Interface

Most access to data on disks in DBMSs is via a file system.

Typical 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
```

Storage Technology

At this point in memory technology development:

- computational storage: fast, expensive, "small" storage is based on RAM
- bulk data storage: "slow", cheaper, large storage is based on disks

New technologies may eventually change this picture entirely

- e.g. holographic memory, large/cheap/non-volatile RAM, ...

But expect spinning disk technology to dominate for at least 5 more years.

Computational Storage

Characteristics of main memory (RAM):

- linear array of bytes (or words)
- transfer unit: 1 byte (or word)
- constant time random access ($\simeq 10^{-7} sec$)

Accessing memory:

```
load   reg,byte_address
store  reg,byte_address
```

Cache memory has similar characteristics to RAM, but is

- faster, more expensive *Rightarrow* smaller

Typical capacities: RAM (256MB..64GB), Cache (64MB..2GB)

Bulk Data Storage

Requirements for bulk data storage:

- non-volatile/permanent (unlike RAM)
- high capacity ($\gg$ RAM)
- fast retrieval speed (ideally $\simeq$ RAM)
- low cost (ideally, $\ll$ RAM)
- addressibility (ideally, smallest unit possible)

... Bulk Data Storage

Several kinds of bulk data storage technology currently exist:

- magnetic disks, optical disks, flash memory

Characteristics of bulk data storage technologies:

- low unit cost (relative to RAM)
- latency in accessing data (disks)
- must read/write "blocks" of data (disks)
- block transfer size typically 512B to 4KB

- can read bytes, must write blocks (flash)
- limited number of write cycles (flash)

Magnetic Disks

Classical/dominant bulk storage technology.

Characteristics:

- typical capacity (16GB..1TB)
- data transferred per block (512B)
- slow seek times (10msec)
- slow rotation speed (20msec)
- reasonable data transfer rate (8MB/sec)

Capacity increase over last decade: 4MB → 1GB → 1TB

Modest increase in speed; good reduction in cost.

Optical Disks

Optical disks provides an alternative spinning disk storage technology.

Several varieties: CD-ROM, CD-R, CD-RW, DVD-RW

Compared to magnetic disks, CD's have

- typical capacity (300..900GB)
- limited number of write/erase cycles (CD-RW)
- data transferred per block (2KB)
- slower seek times (100msec)
- slower rotation speed (20msec)
- lower data transfer rate (150KB/sec)
-

More suited to write-once, read-many applications (static DBs).

Flash Memory (SSD)

Flash memory is a non-mechanical alternative to disk storage.

Compared to disks, flash memory has

- moderate capacity (up to 4TB)
- limited number of write/erase cycles
- can read individual memory items
- can only erase complete blocks
- can only write onto an erased block
- good data transfer rate (16MB/sec)
- no read latency

... Flash Memory (SSD)

Properties of flash memory require specialised file system

Example: updating data in flash storage

- write new copy of changed data to a fresh block
- remap file pointers
- erase old block later when storage is relatively idle

Limitations on updating reduce potential DB applications.

- acceptable for mostly-write (e.g. logs)
- not useful for frequently updated (e.g. TPS)

Overall, not yet a serious contender as a DBMS substrate.

Comparing HDD and SSD

Comparison of HDD and SSD properties:

	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

Will SSDs ever replace HDDs?

Disk Management

Disk Manager

Aim:

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

... Disk Manager

Basic disk management interface is simple:

```
void get_page(PageId p, Page buf)
```

- read disk block corresponding to `PageId` into buffer `Page`

```
void put_page(PageId p, Page buf)
```

- write block in buffer `Page` to disk block identified by `PageId`

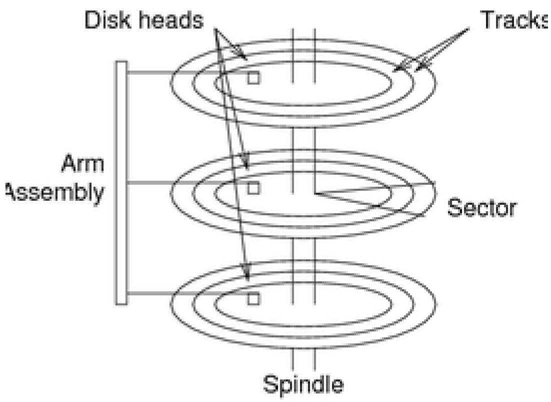
```
PageId allocate_pages(int n)
```

- allocate a group of `n` disk blocks, optimised for sequential access `void deallocate_page(PageId p, int n)`

- deallocate a group of `n` disk blocks, starting at `PageId`

Disk Technology

Disk architecture:



... Disk Technology

Characteristics of disks:

- collection of platters
- each platter = set of tracks (cylinders)
- each track = sequence of sectors (blocks)
- transfer unit: 1 block (e.g. 512B, 1KB, 2KB)
- access time depends on proximity of heads to required block

Accessing disk:

```
read  block at address (p,t,s)
write block at address (p,t,s)
```

Disk Access Costs

Access time includes:

- seek time   (find the right track, e.g. *10-50msec*)
- rotational delay   (find the right sector, e.g. *5-20msec*)
- transfer time   (read/write block, e.g. *0.1msec*)

Cost to write a block is similar to cost of reading

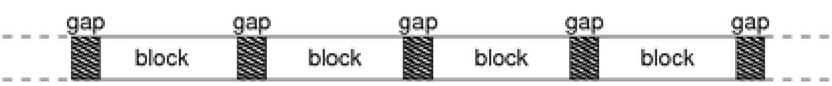
- i.e. seek time + rotational delay + block transfer time

But if we need to *verify* data on disk

- add full rotation delay + block transfer time

... Disk Access Costs

Example disk #1 characteristics:



- 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

Note that this analysis is simplified because #bytes/track and #sectors/track varies between outer and inner tracks (same storage density, reduced track length.

... Disk Access Costs 35/247

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 × 0.9 = 15 ms
- Time over gaps = 16.7 × 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*

... Disk Access Costs 36/247

If operating system deals in 4KB blocks:

T_r(4–blocks) = 25 + (16.7/2) + 4×0.13 + 3×0.01 = 33.9 ms

T_r(1–block) = 25 + (16.7/2) + 0.13 = 33.5 ms

Note that the cost of reading 4KB is comparable to reading 1KB.

Sequential access reduces average block read cost significantly, but

- is limited to 115 block sequences
- is only useful if blocks *need* to be sequentially scanned

... Disk Access Costs 37/247

Example disk #2 characteristics:

- 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

Addressing = 3 bits (surface) + 13 bits (cylinder) + 8 bits (sector)

If using 32-bit addresses, this leaves 8 bits ($2^8=256$ items/block).

Disk Characteristics

Three important characteristics of disk subsystems:

- capacity (how much data can be stored on the disk)
- access time (how long does it take to fetch data from the disk)
- reliability (how often does the disk fail? temporarily? catastrophically?)

Mean time to (complete) failure: 3–10 years.

... Disk Characteristics

Increasing capacity:

- buy a larger disk, or buy more disks
- make the data smaller (using compression techniques)

Improving access time:

- minimise block transfers: clustering, buffering, scheduled access
- reduce seek: faster moving heads, fixed heads, scheduled access
- reduce latency: faster spinning disks, scheduled access
- layout of data on disk (file organisation) can also assist

Improving reliability:

- add redundancy by adding more disks

Increasing Disk Capacity

Compress data (e.g. LZ encoding)

+ more data fits on disk

– compression/expansion overhead

For large compressible data (e.g. `text`), significant savings.

For most relational data (e.g. `int`, `char(8)`), no significant saving.

For high-performance memory caching, may never want to expand
(there is current research working on "computable" compressed data formats).

Improving Disk Access Costs

Approach #1: Use knowledge of data access patterns.

E.g. two records frequently accessed together
⇒ put them in the same block (clustering)

E.g. records scanned sequentially
⇒ place them in "staggered" blocks, double-buffer

Arranging data to match access patterns can improve throughput by 10–20 times.

Approach #2: Avoid reading blocks for each item access.

E.g. buffer blocks in memory, assume likely re-use

Scheduled Disk Access

Low-level disk manager (driver, controller):

- collects list of read/write requests from multiple requestors
- schedules their execution to minimise head movement and latency
- using a queue with priority function based on disk states

Example head movement scheduler: elevator algorithm

- head moves uniformly out towards edge of disk, handling requests "on the way"
- reaches edge, then moves uniformly towards centre of disk, handling requests
- reaches center, then moves out towards edge of disk ...

Disk Layout

If data sets are going to be frequently accessed in a pre-determined manner, arrange data on disk to minimise access time.

E.g. sequential scan

- place subsequent blocks in same cylinder, different platters
- stagger so that as soon as block *i* read, block *i+1* is available
- once cylinder exhausted, move to adjacent cylinder

Older operating systems provided fine-grained control of disk layout.

Modern systems generally don't, because of programmer complexity.

Unix has raw disk partitions: no file system, you write driver to manage disk.

Improving Writes

Nonvolatile write buffers

- "write" all blocks to memory buffers in nonvolatile RAM
- transfer to disk when idle, or when disk head in "good" location
- some operating systems (e.g. Solaris) support this

Log disk

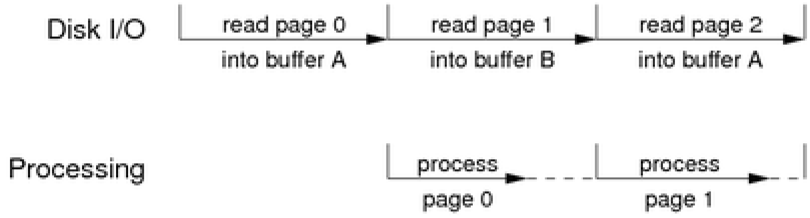
- write all blocks to a special sequential access file system
- transfer to real disk when idle
- additional advantage of having information available for recovery

---

Double Buffering

Double-buffering exploits potential concurrency between disk and memory.

While reads/writes to disk are underway, other processing can be done.



With at least two buffers, can keep disk working full-time.

... Double Buffering

Example: `select sum(salary) from Employee`

- relation = file (= a sequence of *b* blocks A, B, C, D, ...)
- processing data 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)

... Double Buffering

Double-buffering approach:

```
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. $bT_r + bT_p$)

General observation: use of multiple buffers can lead to substantial cost savings.
We will see numerous examples where multiple memory buffers are exploited.

Multiple Disk Systems

Various strategies can be employed to improve capacity, performance and reliability when multiple disks are available.

RAID (redundant arrays on independent disks) defines a standard set of such techniques.

Essentially, multiple disks allow

- improved reliability by redundant storage of data
- reduced access cost by exploiting parallelism

Capacity increases naturally by adding multiple disks

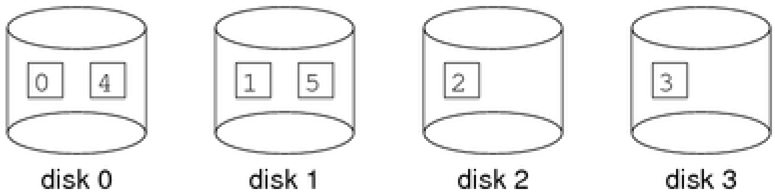
(although there is obviously a trade-off between increased capacity and increased reliability via redundancy)

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.

... RAID Level 0

The disk manager and RAID controller have to perform a mapping something like:

```
writePage(PageId)
```

to

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

(We discuss later how the `pid` might be represented and mapped)

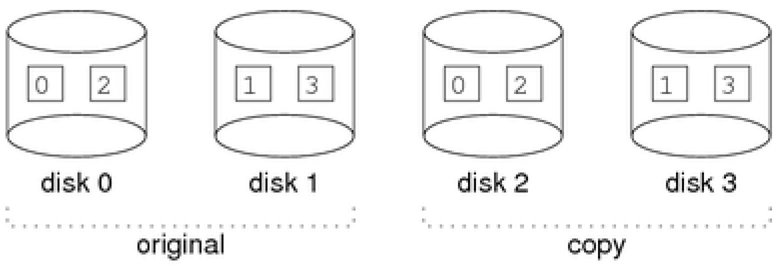
RAID Level 1

Uses *mirroring* (or *shadowing*) to store multiple copies of each block.

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.

... RAID Level 1 52/247

The disk manager and RAID controller have to perform a mapping something like:

```
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)
```

RAID levels 2-6

53/247

The higher levels of raid incorporate various combinations of:

- block/bit-level striping, mirroring, and error correcting codes (ECC)

The differences are primarily in:

- the kind of error checking/correcting codes that are used
- where the ECC parity bits are stored

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

RAID level 6 can recover from smultaneous failures in two disks.

Disk Media Failure

54/247

Rarely, a bit will be transferred to/from the disk incorrectly.

Error-correcting codes can check for and recover from this.

If recovery is not possible, the operation can simply be repeated.

If repeated reads/writes on the same block fail:

- the low-level disk manager assumes permanent media failure
- marks the offending block physical address in a *bad block table*
- the block will be deallocated and never re-used
- if a copy of data is available, can be restored elsewhere on disk

---

Database Objects

55/247

DBMSs maintain various kinds of objects/information:

database	can be viewed as an super-object for all others
parameters	global configuration information
catalogue	meta-information describing database contents
tables	named collections of tuples
tuples	collections of typed field values

indexes	access methods for efficient searching
update logs	for handling rollback/recovery
procedures	active elements

... Database Objects

56/247

The disk manager implements how DB objects are mapped to file system.

References to data objects typically reduce to e.g.

- access object in buffer at position `Offset`
- buffer is obtained as page `PageId` in system
- object is addressed via a `RecordID` = `PageId+Offset`

The disk manager needs to convert buffer access to

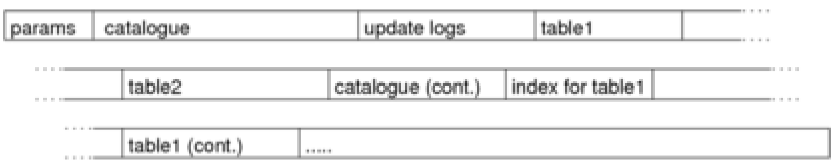
- ensure that the relevant file is open
- locate the physical page within the file
- read/write the appropriate amount of data to/from buffer

Single-file DBMS

57/247

One possible storage organisation is a single file for the entire database.

All objects are allocated to regions of this file.



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

58/247

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:

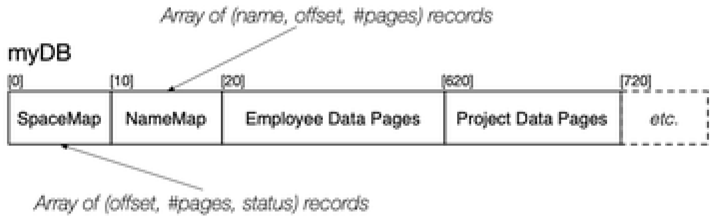
- Unix does not allocate disk space for the "hole" until it is written

Under these circumstances, a disk manager is easy to implement.

Single-file Storage Manager

59/247

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

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

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
   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;
```

Example: Scanning a Relation

With the above disk manager, our example:

```
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
   }
}
```

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);
}
```

... Single-File Storage Manager


```
// 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


```
// 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

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

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:

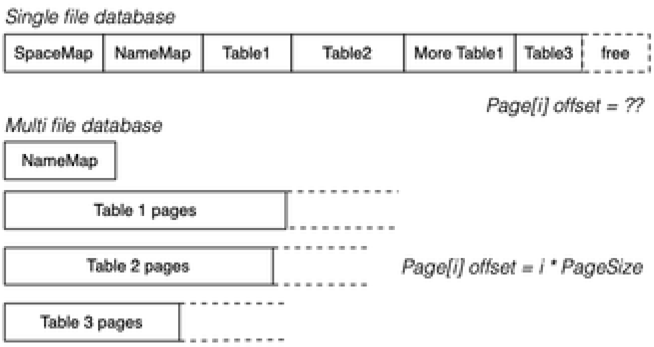
- multiple files partitioned physically or logically
- mapping from DB-level objects to files (e.g. via meta-data)

Precise file structure varies between individual DBMSs.

... Multiple-file Disk Manager

Using multiple files (one file per relation) can be easier

E.g. extending the size of a relation



... Multiple-file Disk Manager

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)

Oracle File Structures

71/247

Oracle uses five different kinds of files:

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

... Oracle File Structures

72/247

There may be multiple instances of each kind of file:

- they may be spread across several disk devices (for load balancing)
- they may be duplicated (for redundancy/reliability)

Data files are

- typically very large (> 100MB)
- typically allocated to several different file systems
- logically partitioned into *tablespaces* (`SYSTEM`, plus dba-defined others)

... Oracle File Structures

73/247

Tablespaces are logical units of storage (cf directories).

Every database object resides in exactly one tablespace.

Units of storage within a tablespace:

data block	fixed size unit of storage (cf 2KB page)
extent	specific number of contiguous data blocks
segment	set of extents allocated to a single database object

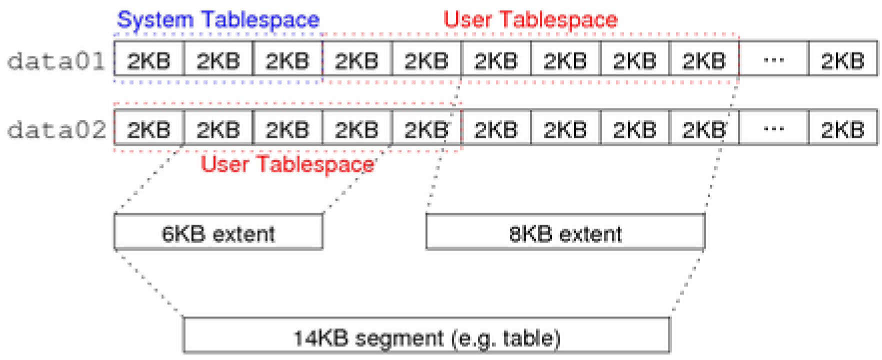
Segments can span multiple data files; extents cannot.

To be confusing, tables are called *datafiles* internally in Oracle.

... Oracle File Structures

74/247

Layout of data within Oracle file storage:



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 disk handler provided

Relations as Files

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

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

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

    if (rnode.spcNode == GLOBALTABLESPACE_OID) {
        /* Shared system relations live in PGDATA/global */
        Assert(rnode.dbNode == 0);
        sprintf(path, "%s/global/%u",
                DataDir, rnode.relNode);
    }
```

```
    else if (rnode.spcNode == DEFAULTTABLESPACE_OID) {
        /* The default tablespace is PGDATA/base */
        sprintf(path, "%s/base/%u/%u",
                DataDir, rnode.dbNode, rnode.relNode);
    }
    else {
        /* All other tablespaces accessed via symlinks */
        sprintf(path, "%s/pg_tblspc/%u/%u/%u", DataDir
                rnode.spcNode, rnode.dbNode, rnode.relNode);
    }
    return path;
}
```

File Descriptor Pool

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".

Source: **`include/storage/fd.h`**, **`backend/storage/file/fd.c`**

... File Descriptor Pool

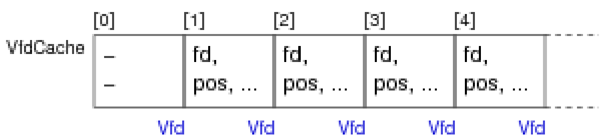
Interface to file descriptor (pool):

```
File PathNameOpenFilePerm(char *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);
```

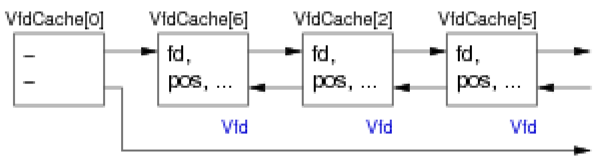
... File Descriptor Pool

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

82/247

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

83/247

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
    ForkNumber  forkNum;  // which fork
    BlockNumber blockNum; // which block
} BufferTag;
```

... File Manager

84/247

PostgreSQL stores each table

- in the directory *PGDATA*/`pg_database.oid`
- often in multiple files (aka *forks*)

... File Manager

85/247

Data files   (*Oid*, *Oid*`.1`, ...):

- sequence of fixed-size blocks/pages (typically 8KB)
- each page contains tuple data and admin data (see later)
- max size of data files 1GB (Unix limitation)

... File Manager

86/247

Free space map   (*Oid*`_fsm`):

- indicates where free space is in data pages
- "free" space is only free after `VACUUM`
  (`DELETE` simply marks tuples as no longer in use `xmax`)

Visibility map   (*Oid*`_vm`):

- indicates pages where all tuples are "visible"
  (*visible* = accessible to all currently active transactions)
- such pages can be ignored by `VACUUM`
- also used for index pages, to indicate all index entries visible
  (allows *index-only scans* to be done more efficiently)

... File Manager

Access to a block of data proceeds (roughly) as follows:

```
// 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
}
```

`BLOCKSIZE` is a global configurable constant (default: 8192)

... File Manager


```
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)
}
```

Buffer Pool

Buffer Manager

Aim:

- minimise traffic between disk and memory via caching
- mantains a (shared) *buffer pool* in main memory

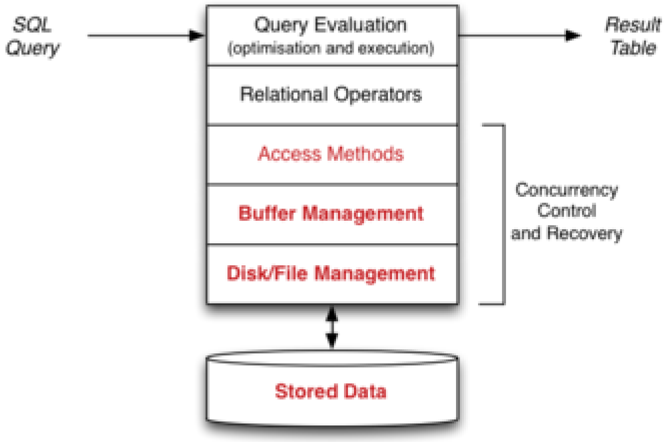
Buffer pool

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

---

... Buffer Manager

Buffer pool interposed between access methods and disk manager



Access methods/page manager normally work via get_page() calls;
now work via calls to get_page_via_buffer_pool() (aka request_page())

... Buffer Manager

Basic buffer pool 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

Buffer pool typically provides interface to `allocate_page` and `deallocate_page` as well.

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)
}
```

Requires `N` page reads.

If we read it again, `N` page reads.

... Buffer Pool Usage

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);
}
```

Requires `N` page reads on the first pass.

If we read it again, 0 ≤ page reads ≤ `N`

Buffer Pool Data

... Buffer Pool Data

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:

- whether it is currently in use
- which Page it contains   (i.e. PageId = *(relid,page#)*)
- whether it has been modified since loading (*dirty bit*)
- how many transactions are currently using it (*pin count*)
- time-stamp for most recent access

... Buffer Pool Data

... Buffer Pool Data

In subsequent discussion, we assume:

- cost of manipulating in-memory buffer pool data is insignificant
- all file access methods use request_page() instead of get_page()

Requesting Pages

Call from client: `request_page(pid)`

If page `pid` *is* already in buffer pool:

- no need to read it again
- use the copy in the pool   (unless write-locked)

If page `pid` is *not* already in buffer pool:

- need to read page from disk into a free frame
- if no free frames, need to remove a page from the pool

... Requesting Pages

Advantages:

- if a page is required several times for an operation, only read once

Disadvantages:

- overhead of managing buffer pool for each page request (insignificant)

- if page access pattern clashes with replacement, no effective caching

Releasing Pages

The `release_page` function indicates that a page

- is no longer required by this transaction
- is a good candidate for replacement   (**iff** noone else using it)

If the page hasn't been modified, simply overwritten when replaced.

If the page has been modified, must be written to disk before replaced.

Possible problem: changes not immediately reflected on disk

... Releasing Pages

Advantages:

- if page modified several times while in the pool, only written once

Disadvantages:

- overhead of managing buffer pool for each page request (insignificant)

If a page remains in pool over multiple transactions

- e.g. (requested,modified,released) several times but not *replaced*
- need to ensure that changes are guaranteed to be reflected on disk
- even if the system crashes before page is replaced

(This is generally handled by some kind of logging mechanism (e.g. Oracle redo log files).

Buffer Manager Example #1

Self join: an example where buffer pool achieves major efficiency gains.

Consider a query to find pairs of employees with the same birthday:

```
select e1.name, e2.name
from   Employee e1, Employee e2
where  e1.id < e2.id and e1.birthday = e2.birthday
```

This might be implemented inside the DBMS via nested loops:

```
for each tuple t1 in Employee e1 {
    for each tuple t2 in Employee e2 {
        if (t1.id < t2.id &&
                t1.birthday == t2.birthday)
            append (t1.name,t2.name) to result set
    }
}
```

... Buffer Manager Example #1

In terms of page-level operations, the algorithm looks like:

```
DB db = openDatabase("myDB");
Rel emp = openRel(db,"Employee");
int npages = nPages(emp);

for (int i = 0; i < npages; i++) {
    PageId pid1 = makePageId(emp,i);
    Page p1 = request_page(pid1);
    for (int j = 0; j < npages; i++) {
        PageId pid2 = makePageId(emp,j);
        Page p2 = request_page(pid2);
        // compare all pairs of tuples from p1,p2
```

```
        // construct solution set from matching pairs
        release_page(pid2);
    }
    release_page(pid1);
}
```

... Buffer Manager Example #1

105/247

Consider a buffer pool with 200 frames and a relation with $b \leqslant 200$ pages:

- first request for `p1` loads page 0 into buffer pool
- first request for `p2` finds page 0 already loaded
- rest of first `p2` iteration loads all other pages from `Employee`
- all subsequent requests find required page already loaded

Total number of page reads = b (entire relation is read exactly once)

... Buffer Manager Example #1

106/247

Now consider a buffer pool with 2 frames (the minimum required for the join):

- first request for `p1` loads page 0 into buffer pool
- first request for `p2` finds page 0 already loaded
- next request for `p2` loads page 1 into buffer pool
- next request for `p2` finds buffer pool full
  ⇒ need to free frame (but note that no write is required)
- because page 0 is "in use", we replace page 1

(continued ...)

... Buffer Manager Example #1

107/247

(... continued)

- request/release ⇒ page 0 remains in buffer while scanning on `p2`
- on each of the *b–1* subsequent `p2` scans ...
  - the `p1` page remains resident, while we iterate over the `p2` pages
  - we don't need to read the `p1` page (it's already resident)

Total number of page reads = $b * (b–1)$

Cf. 200–frame buffer vs 2–frame buffer ... if *b=100*, 100 reads vs 10000 reads.

Buffer Pool Implementation

108/247

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];
```

... Buffer Pool Implementation

109/247

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
}
```

Other Buffer Operations

The `release_page` operation:

- Decrement pin count for specified page

Note: no effect on disk or buffer contents until replacement required.

The `mark_page` operation:

- Set dirty bit on for specified page

Note: doesn't actually write to disk; indicates that frame needs to be written if used for replacement;

The `flush_page` operation:

- Write the specified page to disk (using `write_page`)

Note: not generally used by higher levels of DBMS; they rely on request/release protocol.

Page Replacement Policies

Several schemes are commonly in use:

- Least Recently Used (LRU)
    - often used for VM in operating systems; intuitively appealing but can perform badly
- First in First Out (FIFO)
    - need to maintain a queue of frames; enter tail of queue when read in
- Most Recently Used
- Random

LRU works for VM because of working set model
(recent past accesses determines future accesses)

For DBMS, we can predict patterns of page access better
(from our knowledge of how the relational operations are implemented)

... Page Replacement Policies

The cost benefits from a buffer pool (with n frames) is determined by:

- number of available frames (more ⇒ better)
- interaction between replacement strategy and page access patterns

Example (a): sequential scan, LRU or MRU, $n \geqslant b$, no competition

First scan costs b reads; subsequent scans are ``free''.

Example (b): sequential scan, MRU, $n < b$, no competition

First scan costs b reads; subsequent scans cost $b - n$ reads.

Example (c): sequential scan, LRU, $n < b$, no competition

All scans cost b reads; known as *sequential flooding*.

Page Access Times

How to determine when a page in the buffer was last accessed?

Could simply use the time of the last `request_page` for that `PageId`.

But this doesn't reflect *real* accesses to page.

For more realism, could use last `request_page` or `release_page` time.

Or could introduce operations for examining and modifying pages in pool:

- `examine_page(PageId, TupleId)` and `modify_page(PageId, TupleId, Tuple)`
- add "last access time" field to directory entry for each frame
- above operations access the page and also update the access time field

---

Buffer Manager Example #2

Standard join: an example where replacement policy can have large impact.

Consider a query to find customers who are also employees:

```
select c.name
from   Customer c, Employee e
where  c.ssn = e.ssn;
```

This might be implemented inside the DBMS via nested loops:

```
for each tuple t1 in Customer {
    for each tuple t2 in Employee {
        if (t1.ssn == t2.ssn)
            append (t1.name) to result set
    }
}
```

... Buffer Manager Example #2

Assume that:

- the `Customer` relation has b_C pages (e.g. 20)
- the `Employee` relation has b_E pages (e.g. 10)
- the buffer pool has n frames (e.g. 10)
- it cannot hold either relation completely   ($n < b_C$ and $n < b_E$)

... Buffer Manager Example #2

Works well with MRU strategy:

- pins `Customer` page, then processes all `Employee` pages against it
- each `Customer` page read exactly once
- two `Customer` pages occupy memory on all iterations bu the first
- $n - 3$ `Employee` pages read once
- the rest are read once on each of the b_C–1 iterations

Total page reads = $b_C + b_E + (b_C-1) \times (b_E - (n-3))$ = 20 + 10 + 19*(10−3) = 163

Note: assumes that both `request_page` and `release_page` set the last usage timestamp.

... Buffer Manager Example #2

Works less well with LRU strategy:

- pins `Customer` page, then starts to process `Employee` pages
- when pool fills starts replacing `Employee` pages from beginning
- each `Customer` page read exactly once
- each `Employee` page read once on each iteration

Total page reads = $b_C + b_C \times b_E$ = 20 + 20*10 = 220

PostgreSQL Buffer Manager

PostgreSQL buffer manager:

- provides a shared pool of memory buffers for all backends
- all access methods get data from disk via buffer manager

Same code used by backends which need a local buffer pool.

Buffers are located in a large region of shared memory.

Functions: **`src/backend/storage/buffer/*.c`**

Definitions: **`src/include/storage/buf*.h`**

... PostgreSQL Buffer Manager

Buffer pool consists of:

- shared fixed array (size `Nbuffers`) of **`BufferDesc`**
- shared fixed array (size `Nbuffers`) of **`Buffer`**
- each **`BufferDesc`** contains:
  - reference to memory for `Buffer`
  - status information (e.g. pin count, lock state)
- number of buffers set in *postgresql.conf*, e.g.

```
shared_buffers = 16MB      # min 128KB, at least max_connections*2, 8KB each
```

... PostgreSQL Buffer Manager

... PostgreSQL Buffer Manager

Definitions related to buffer manager:

`include/storage/buf.h`

- basic buffer manager data types (e.g. **`Buffer`**)

`include/storage/bufmgr.h`

- definitions for buffer manager function interface
  (i.e. the functions that other parts of the system call to user buffer manager)

`include/storage/buf_internals.h`

- definitions for buffer manager internals (e.g. **`BufferDesc`**)

Code in: **`backend/storage/buffer/`**

Buffer Pool Data Objects

BufferDescriptors: array of structures describing buffers

- holds data showing buffer usage; implements free list

Buffer: index into **BufferDescriptors**

- index values run from 1..Nbuffers ⇒ need −1
- local buffers have negative indexes

BufMgrLock: global lock on buffer pool

- needs to be obtained when modifying content in buffer pool

BufferTag

- data structure holding *(r,b)* pair; used to hash buf ids

... Buffer Pool Data Objects

Buffer manager data types:

```
BufFlags: BM_DIRTY, BM_VALID, BM_TAG_VALID, BM_IO_IN_PROGRESS, ...

typedef struct buftag {
   RelFileNode rnode;     /* physical relation identifier */
   ForkNumber  forkNum;
   BlockNumber blockNum;  /* relative to start of reln */
} BufferTag;

typedef struct BufferDesc {  (simplified)
   BufferTag   tag;          /* ID of page contained in buffer */
   int       buf_id;      // buffer's index number (from 0)
   Bits32    state;       // dirty, refcount, usage
   int       freeNext;    // link in freelist chain
   ...                    // others related to concurrency
} BufferDesc;
```

Buffer Pool Functions

Buffer manager interface:

Buffer ReadBuffer(Relation r, BlockNumber n)

- ensures n^{th} page of file for relation `r` is loaded
  (may need to remove an existing unpinned page and read data from file)
- increments reference (pin) count and usage count for buffer
- returns index of loaded page in buffer pool (`Buffer` value)
- assumes main fork, so no `ForkNumber` required

Actually a special case of `ReadBuffer_Common`, which also handles variations like different replacement strategy, forks, temp buffers, ...

... Buffer Pool Functions

Buffer manager interface (cont):

void ReleaseBuffer(Buffer buf)

- decrement pin count on buffer
- if pin count falls to zero,
  ensures all activity on buffer is completed before returning

void MarkBufferDirty(Buffer buf)

- marks a buffer as modified
- requires that buffer is pinned and locked
- actual write is done later (e.g. when buffer replaced)

... Buffer Pool Functions

Additional buffer manager functions:

`Page BufferGetPage(Buffer buf)`

- finds actual data associated with buffer in pool
- returns reference to memory where data is located

`BufferIsPinned(Buffer buf)`

- check whether this backend holds a pin on buffer

`CheckPointBuffers`

- write data in checkpoint logs (for recovery)
- flush all dirty blocks in buffer pool to disk

etc. etc. etc.

... Buffer Pool Functions

Important internal buffer manager function:

```
BufferDesc *BufferAlloc(
             Relation r, ForkNumber f,
             BlockNumber n, bool *found)
```

- used by **`ReadBuffer`** to find a buffer for *(r,f,n)*
- if *(r,f,n)* already in pool, pin it and return descriptor
- if no available buffers, select buffer to be replaced
- returned descriptor is pinned and marked as holding *(r,f,n)*
- **`ReadBuffer`** has to do the actual I/O

Clock-sweep Replacement Strategy

PostgreSQL page replacement strategy: *clock-sweep*

- 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)

For specialised kinds of access (e.g. sequential scan), can allocate a private "buffer ring" with different replacement strategy.

Record/Tuple Management

Views of Data

The disk and buffer manager provide the following view:

- data is a sequence of fixed-size blocks (pages)
- blocks can be (random) accessed via a `PageId`

Database applications view data as:

- a collection of records (tuples)
- records can be accessed via a `RecordId` (`RID`)

Standard terminology: *records* are also called *tuples*, items, rows, ...

... Views of Data

131/247

The abstract view of a relation:

- a named and (possibly) ordered sequence of *tuples*
- with (possibly) some additional access method data structures

The physical representation of a relation:

- an indexed sequence of *pages* in one or more files
- where each page contains a collection of *records*
- along with data structures to manage the records

... Views of Data

132/247

We use the following low-level abstractions:

`RecPage`

- a view of a disk page ... record data + storage management info
- provides an interpretation of `byte[]` provided by buffer manager

`Record`

- physical view of a table row ... a sequence of bytes
- format of table row data used for storing on disk

... Views of Data

133/247

We use the following high-level abstractions:

`Relation`

- logical view of a database table ... collection of tuples
- implemented via multiple pages in multiple files

`Tuple`

- logical view of a table row ... a collection of typed fields
- format of table row data used for manipulating in memory

Records vs Tuples

134/247

A *table* is defined by a collection of attributes (*schema*), e.g.

```
create table Employee (
   id#  integer primary key,
   name varchar(20),   -- or char(20)
   job  varchar(10),   -- or char(10)
   dept number(4)
);
```

A *tuple* is a collection of attribute values for such a schema, e.g.

```
    (33357462, 'Neil Young', 'Musician', 0277)
```

A *record* is a sequence of bytes, containing data for one tuple.

Record Management

135/247

Aim:

- provide `Tuple` and `Record` abstractions
- provide mapping from `RecordId` to `Tuple`
- allocate/maintain space within blocks (via `RecPage` abstraction)

In other words, the record manager reconciles the views of a block:

- array of bytes (physical)   vs   collection of tuples (logical)
- via the notion of records and intra-block storage management

Assumptions   (neither of which are essential):

- each block contains tuples from one relation
- every tuple is (much) smaller than a single page

Page-level Operations

Operations to access records from a page ...

`Record get_record(RecordId rid)`

- get record `rid` from page; returns reference to `Record`

`Record first_record()`

- return reference to `Record` first record in page

`Record next_record()`

- return reference to `Record` immediately following last accessed one
- returns `null` if no more records left in the page

... Page-level Operations

Operations to make changes to records in a page ...

`void update_record(RecordId rid, Record rec)`

- change value of record `rid` to the value stored in `rec`

`RecordId insert_record(Record rec)`

- insert new record into page and return its `rid`

`void delete_record(RecordId rid)`

- remove the record `rid` from the page

Tuple-level Operations

***Typ* `get`*Typ*`Field(int fno)`**

- extract the `fno`'th field from a `Tuple` as a value of type *Typ*

Examples: `getIntField(1)`, `getStringField(2)`

`void` `set`*Typ*`Field(int fno,` *Typ* `val)`

- set the value of the `fno`'th field of a `Tuple` to `val`

Examples: `setIntField(1,42)`, `setStringField(2,"abc")`

Also need operations to convert between `Record` and `Tuple` formats.

Relation-level Operations

`Tuple get_tuple(RecordId rid)`

- fetch the tuple specified by `rid`; return reference to `Tuple`

`Tuple first_tuple()`

- return reference to record first `Tuple` in page

`Tuple next_tuple()`

- return reference to `Tuple` immediately following last accessed one
- returns `null` if no more `Tuples` left in the relation

Plus operations to insert, delete and modify `Tuples` (analogous to `Records`)

Example Query

Recall previous example of simple scan of a relation:

```
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);
}
```

... Example Query

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;
}
```

... Example Query

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;
}
```

... Example Query

And similarly the `nextTuple()` function:

```
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;
}
```

Record Identifiers

The implementation of `RecordIDs` is determined by the physical storage structure of the DBMS.

A `RecordId` always has at least two components:

- a *page number* to indicate which page the record is contained in
- a *slot number* to indicate where the record is located within the page

If multiple files for a relation, then also need:

- a *file number* to indicate which file the page is contained in

(Or, more likely, use a `PageId` which combines both the file number and page number)

Some DBMSs provide `ROWIDs` in SQL to permit efficient tuple access.

PostgreSQL provides a unique OID for every row in the database.

... Record Identifiers

`RecordID` components are

- implemented as counters (table indexes) rather than absolute offsets
- to save space and to allow for 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`

Example `RecordId` Structure

Consider a DBMS like Oracle which uses a small number of large files.

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:

(Note: however you partition the bits, you can address at most 4 billion records)

... Example `RecordId` Structure

Consider a DBMS like MiniSQL, which uses one data file per relation.

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

(Under this scheme, there will be multiple records in the DB with the same `rid`)

Manipulating `RecordIds`

Functions for constructing/interrogating `RecordIds`:

```
typedef unsigned int RecordId;

RecordId makeRecordId(int file, int page, int slot) {
   return (file << 28) | (page << 8) | (slot);
}
int fileNo(RecordId rid) { return (rid >> 28) & 0xF; }

int pageNo(RecordId rid) { return (rid >> 8) & 0xFFFFF; }

int slotNo(RecordId rid) { return rid & 0xFF; }
```

... Manipulating `RecordIds`

Alternative implementation if details of file/page are hidden within `PageId`:

```
typedef unsigned int PageId; //only uses 24-bits
typedef unsigned int RecordId;

RecordId makeRecordId(PageId pid, int slot) {
    return (pid << 8) | (slot);
}

int pageId(RecordId rid) { return (rid >> 8) & 0xFFFFFF; }

int slotNo(RecordId rid) { return rid & 0xFF; }
```

Record Formats

Records are stored within fixed-length pages.

Records may be fixed-length:

- simplifies intra-block space management (i.e. implementation of insert/delete)
- may waste some (substantial) space

Records may be variable-length:

- complicates intra-block space management
- doesn't waste (as much) space

Fixed-length Records

Encoding scheme for fixed-length records:

- record format (length + offsets) stored in catalogue

- data values stored in fixed-size slots in data pages

Since record format is frequently consulted at query time, it should be memory-resident.

... Fixed-length Records

152/247

Advantages of fixed-length records:

- don't need slot directory in page (compute record offset as *number × size*)
- records are smaller (formatting info stored only once, outside data pages)
- intra-page memory management is simplified (as long as not data overflow)

Disadvantages of fixed-length records:

- need to allocate maximum likely space in *every* record slot
- leads to (potentially) considerable space wastage (e.g. 40% for string values)

Note: if all records were close to specified maximum size, this would be the most compact format.

... Fixed-length Records

153/247

Handling attempts to insert values larger than available fields:

- simply refuse (generate DBMS run-time error)
- place oversize data in an overflow page
  - field contains a reference to the "overflow page" instead of value
  - requires field to be at least as large as a `RecordId`

Alignment considerations (for numeric fields) may require:

- all records and all fields start on a 4-byte boundary
- thus, `varchar` fields may be rounded up to nearest 4-bytes

Variable-length Records

154/247

Some encoding schemes for variable-length records:

- Prefix each field by length

- Terminate fields by delimeter

- Array of offsets

... Variable-length Records

155/247

More encoding schemes for variable-length records:

- 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

---

... Variable-length Records

156/247

Advantages of variable-length records:

- minimal wasted space *within* records   (markers,lengths,delimeters)
- more flexibility in managing space within pages

Disadvantages of variable-length records:

- potential for free-space fragmentation within pages
- more complex intra-page space management algorithms

Spanned Records

How to handle record that does not fit into free space in page?

Two approaches:

- waste some space

- span the record between two pages

... Spanned Records

Advantages of spanned records:

- better storage utilisation   (i.e. less wasted space)
- ability to store arbitrarily large records

Disadvantages of spanned records:

- fetching a single record may require multiple page accesses

More common strategy than spanning:

- store large data values outside record in separate file

Converting Records to Tuples

A `Record`

- is an array of bytes (`byte[]`)
- representing the data values from a typed `Tuple`

The information on how to interpret the bytes

- may be contained in a schema in the DBMS catalogue
- may be stored the header for the data file
- may be stored partly in the record and partly in a DTD (for XML)

For variable-length records, further formatting information is stored in the record itself.

... Converting Records to Tuples

DBMSs typically define a fixed set of field types for use in schema.

E.g. `DATE`, `FLOAT`, `INTEGER`, `NUMBER(`*n*`)`, `VARCHAR(`*n*`)`, ...

This determines the primitive types to be handled in the implementation:

`DATE`	`time_t`
`FLOAT`	`float,double`
`INTEGER`	`int,long`

`NUMBER(`*n*`)`	`int[]`
`VARCHAR(`*n*`)`	`char[]`

Defining `Tuples`

161/247

To convert a `Record` to a `Tuple` we need to know:

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

This leads to two structs: `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;
```

... Defining `Tuples`

162/247

For the example relation:

```
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);
```

This defines the schema

- for fixed-length tuples, this describes all tuple instances
- for variable-length tuples, need to compute actual lengths and offsets

... Defining `Tuples`

163/247

A `Tuple` can be defined as

- a list of field descriptors for a record instance
- along with a reference to the `Record` data

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

... Defining `Tuples`

164/247

A `Tuple` is produced from a `Record` in the context of a `RelnDesc`.

It also necessary to know how the `Record` byte-string is structured.

Assume the following `Record` structure:

Assume also that lengths are 1-byte quantities (no field longer than 256-bytes).

... Defining `Tuples`

165/247

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;
}
```

PostgreSQL Tuples

Definitions: **`src/include/access/*tup*.h`**

Functions: **`src/backend/access/common/*tup*.c`**

PostgreSQL defines tuples via:

- a contiguous chunk of memory
- starting with a header giving e.g. #fields, nulls
- followed by the data values (as sequence of `Datum`)

... PostgreSQL Tuples

Tuple structure:

... PostgreSQL Tuples

Tuple-related data types:

```
// representation of a data value
// may be the actual value, or may be a pointer to it
typedef unitptr_t Datum;
```

The actual data value:

- may be stored in the `Datum` (e.g. `int`)
- may have a header with length (for varlen attributes)
- may be stored in a TOAST file

... PostgreSQL Tuples

Tuple-related data types: (cont)

```
typedef struct HeapTupleFields  // simplified
{
    TransactionId   t_xmin;       // inserting xact ID
    TransactionId   t_xmax;       // deleting or locking xact ID
    CommandId       t_cid;        // inserting/deleting command ID, or both
} HeapTupleFields;
typedef struct HeapTupleHeaderData // simplified
{
    HeapTupleFields t_heap;
```

```
    ItemPointerData t_ctid;       // current TID of this or newer tuple
    uint16          t_infomask2;  // number of attributes + flags
    uint16          t_infomask;   // flags e.g. has_null, has_varwidth
    uint8           t_hoff;       // sizeof header incl. bitmap+padding
    // above is fixed size (23 bytes) for all heap tuples
    bits8           t_bits[1];    // bitmap of NULLs, variable length
    // actual data follows at end of struct
} HeapTupleHeaderData;
```

... PostgreSQL Tuples


```
typedef struct tupleDesc
{
    int                  natts;      // number of attributes in the tuple
    Form_pg_attribute *attrs;      // array of pointers to attr descriptors
    TupleConstr       *constr;     // constraints, or NULL if none
    Oid                  tdtypeid;   // composite type ID for tuple type
    int32                tdtypmod;   // typmod for tuple type
    bool                 tdhasoid;   // tuple has oid attribute in its header
    int                  tdrefcount; // reference count, -1 if not counting
} *TupleDesc;
```

... PostgreSQL Tuples

Operations on Tuples:

```
// create Tuple from values
HeapTuple
heap_form_tuple(TupleDesc tupDesc, Datum *values, bool *isnull)

// return Datum given Tuple, attr and descriptor
//   sets isnull to true if value is NULL
#define heap_getattr(tup, attnum, tupleDesc, isnull) ...

// returns true if attribute has no value
bool heap_attisnull(HeapTuple tup, int attnum) ...

// produce a modified tuple from an existing one
HeapTuple
heap_modify_tuple(HeapTuple tuple, TupleDesc tupleDesc,
                  Datum *replValues, bool *replIsnull,
                  bool *doReplace)
```

Page Formats

Ultimately, a `Page` is simply an array of bytes (`byte[]`).

We want to interpret/manipulate it as a collection of `Records`.

Typical operations on `Pages`:

- `get(rid)` ... get a record via its `TupleId`
- `first()` ... get first record from `Page` (start scan)
- `next()` ... fetch next record during a `Page` scan
- `insert(rec)` ... add a new record into a `Page`
- `update(rid,rec)` ... update value of specified record
- `delete(rid)` ... remove a specified record from a `Page`

... Page Formats

Factors affecting `Page` formats:

- determined by record size flexibility   (fixed, variable)
- how free space within `Page` is managed
- whether some data is stored outside `Page`

- does `Page` have an associated overflow chain?
- are large data values stored elsewhere? (e.g. TOAST)
- can one tuple span multiple `Pages`?

Implementation of `Page` operations critically depends on format.

... Page Formats

174/247

For fixed-length records, use *record slots*.

Insertion: place new record in first available slot.

Deletion: two possibilities for handling free record slots:

... Page Formats

175/247

Problem with packed format and no slot directory

- records must move around, so *rid*s are not fixed

Could add a slot directory to overcome this, but wastes space.

Problem with unpacked/bitmap format

- records are not allowed to move (*rid*s use absolute offsets)
- using *rid*s to specify offset is more expensive than slot index
  (e.g. 4KB page requires 12-bit offset (10-bit if word-aligned), 256 slots requires 8-bit index)

... Page Formats

176/247

For variable-length records, use *slot directory*.

Possibilities for handling free-space within block:

- compacted (one region of free space)
- fragmented (distributed free space)

In practice, a combination is useful:

- normally fragmented (cheap to maintain)
- compacted when needed (e.g. record won't fit)

... Page Formats

177/247

Compacted free space:

Note: "pointers" are implemented as word offsets within block.

... Page Formats

178/247

Fragmented free space:

Storage Utilisation

179/247

How many records can fit in a page?   (How long is a piece of string?)

Depends on: page size, (avg) record size, slot directory, ...

For a typical DBMS application

- a record is 32..256 bytes, a page has 2K bytes

- so each page contains from 10..100 records

... Storage Utilisation

180/247

Example of determining space utilisation ...

Assumptions:

- 1024-byte (1KB) page size
- records of type `(integer,varchar(20),char(10),number(4))`
- variable-length records with 4 (1-byte) offsets at start of record
- `char(10)` field rounded up to 12-bytes to preserve alignment
- maximum size of second field is 20 bytes; average length is 16 bytes
- records start at 4-byte offsets ⇒ 8-bits per directory slot
- page has 4-byte overflow `PageId` (other header info?)

... Storage Utilisation

181/247

Max record size = 4(offsets) + 4 + 20 + 12 + 4 = 44 bytes

Minimum number of records = 1024/44 = 23 (assume all max size and no directory)

Average number of records = 1024/40 = 25 (assume no directory)

So, allow 32 directory slots (5-bit slot indexes), and 32 bytes for directory.

Number of records = N_r, where $44 \times N_r + 32+4 \leq 1024$

Aim to maximise N_r, so $N_r = 22$

Notes: because there are 32 slots, could have up to 32 (small) records

---

... Storage Utilisation

182/247

If we switched to 8KB pages, then

- directory slots need 11 bits each to address 4-byte-aligned records

Minimum number of records = 8192/44 = 186 (assume all max size and no directory)

So, allow 256 slots (8-bit slot indexes), and 352 bytes for directory (256*11bits)

Number of records = N_r, where $44 \times N_r + 352 \leq 8192$

Aim to maximise N_r, so $N_r = 178$

Could reduce size of directory to allow more records ... but only so far.

Note: 11-bit directory entries also means that it's costly to access them.

Overflows

183/247

Sometimes, it may not be possible to insert a record into a page:

1. no free-space fragment large enough
2. overall free-space is not large enough
3. the record is larger than the page
4. no more free directory slots in page

The first case can initially be handled by compacting the free-space.

If there is still insufficient space, we have one of the other cases.

... Overflows

184/247

How the other cases are handled depends on the file organisation:

- records may be inserted anywhere that there is free space
  - cases (2) and (4) can be handled by making a new page
  - case (3) requires either spanned records or "overflow file"
- record placement is determined by access method (e.g. hashed file)
  - case (2) requires an "overflow page"
  - case (3) requires an "overflow file"
  - case (4) is problematic, since the *rid* can only address N_r slots

---

... Overflows

185/247

Overflow files for very large records and BLOBs:

- abandon notion of slots and simply access record via offset

---

... Overflows

186/247

Page-based handling of overflows:

- add the `PageId` of the overflow page to the page header

Useful for scan-all-records type operations.

... Overflows

187/247

Record-based handling of overflows:

- store the *rid* of the overflow record instead of the record itself

Useful for locating specific record via *rid*.

PostgreSQL Page Representation

188/247

Functions: **`src/backend/storage/page/*.c`**

Definitions: **`src/include/storage/bufpage.h`**

Each page is 8KB (default `BLCKSZ`) and contains:

- header (free space pointers, flags, xact data)
- array of (offset,length) pairs for tuples in page
- free space region (between array and tuple data)
- actual tuples themselves (inserted from end towards start)
- (optionally) region for special data (e.g. index data)

Large data items are stored in separate (TOAST) files.

... PostgreSQL Page Representation

189/247

PostgreSQL tuple page layout:

... PostgreSQL Page Representation

190/247

Page-related data types:

```
// a Page is simply a pointer to start of buffer
typedef Pointer Page;
```

```
// indexes into the tuple directory
typedef unit16  LocationIndex;

// entries in tuple directory (line pointer array)
typedef struct ItemIdData
{
   unsigned   lp_off:15,    // tuple offset from start of page
              lp_flags:2,   // state of item pointer
              lp_len:15;    // byte length of tuple
} ItemIdData;
```

... PostgreSQL Page Representation

Page-related data types: (cont)

```
typedef struct PageHeaderData
{
   ...
   uint16        pd_flags;    // flag bits (e.g. free, full, ...
   LocationIndex pd_lower;    // offset to start of free space
   LocationIndex pd_upper;    // offset to end of free space
   LocationIndex pd_special;  // offset to start of special space
   uint16        pd_pagesize_version;
   ...
   ItemIdData    pd_linp[1];  // beginning of line pointer array
} PageHeaderData;

typedef PageHeaderData *PageHeader;
```

... PostgreSQL Page Representation

Operations on `Pages`:

`void PageInit(Page page, Size pageSize, ...)`

- initialize a `Page` buffer to empty page
- in particular, sets `pd_lower` and `pd_upper`

`OffsetNumber`
`PageAddItem(Page page, Item item, Size size, ...)`

- insert one tuple into a `Page`
- fails if: not enough free space, too many tuples

`void PageRepairFragmentation(Page page)`

- compact tuple storage to give on large free space region

... PostgreSQL Page Representation

PostgreSQL has two kinds of pages:

- *heap pages* which contain tuples
- *index pages* which contain index entries

Both kinds of page have the same page layout.

One important difference:

- index entries tend be a smaller than tuples
- can typically fit more index entries per page

Representing Database Objects

Database Objects

RDBMSs manage different kinds of objects

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

Many objects have names (and, in PostgreSQL, all have OIDs).

How are the different types of objects represented?

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

... Database Objects

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 definitions

- each schema is defined with a database/catalog
- used for name-space management (`Schema.Relation`)

tablespace ... collection of DB files

- files contain DB objects from multiple catalog/schemas
- used for file-space management (disk load sharing)

PostgreSQL also has *cluster*: a server managing a set of DBs.

... Database Objects

Consider what information the RDBMS needs about relations:

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

Similarly for other DBMS objects (e.g. views, functions, triggers, ...)

All of this information is stored in the *system catalog*.

(The "system catalog" is also called "data dictionary" or "system view")

In most RDBMSs, the catalog itself is also stored as tables.

... Database Objects

Standard for catalogs in SQL:2003: `INFORMATION_SCHEMA`.

```
Schemata(catalog_name, schema_name, schema_owner, ...)

Tables(table_catalog, table_schema, table_name, table_type, ...)

Columns(table_catalog, table_schema, table_name, column_name,
        ordinal_position, column_default, is_nullable, data_type, ...)

Views(table_catalog, table_schema, table_name, view_definition,
                       check_option, is_updatable, is_insertable_into)

Role_table_grants(grantor, grantee, privilege_type, is_grantable,
                         table_catalog, table_schema, table_name, ...)
etc. etc.
```

For complete details, see Section 37 of the PostgreSQL 11.3 documentation.

System Catalog

Most DBMSs also have their own internal catalog structure.

Would typically contain 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.
```

Standard SQL `INFORMATION_SCHEMA` is provided as a set of views on these tables.

... System Catalog

The catalog is manipulated by a range of SQL operations:

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

where *Object* is one of table, view, function, trigger, schema, ...

E.g. consider an SQL DDL operation such as:

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

... System Catalog

This would produce a set of catalog changes something 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, ...)
```

... System Catalog

In PostgreSQL, the system catalog is available to users via:

- special commands in the `psql` shell (e.g. `\d`)
- SQL standard `information_schema`
  (e.g. `select * from information_schema.tables;`)

The low-level representation is available to sysadmins via:

- a global schema called `pg_catalog`
- a set of tables/views in that schema (e.g. `pg_tables`)

PostgreSQL Catalog

The `\d?` special commands in `psql` are just wrappers around queries on the low-level catalog tables, e.g.

`\dt`	list information about tables
`\dv`	list information about views
`\df`	list information about functions
`\dp`	list table access privileges
`\dT`	list information about data types
`\dd`	shows comments attached to DB objects

... PostgreSQL Catalog

A PostgreSQL installation typically has several databases.

Some catalog information is global, e.g.

- databases, users, ...
- there is one copy of each such table for the whole PostgreSQL installation
- this copy is shared by all databases in the installation (lives in `PGDATA/pg_global`)

Other catalog information is local to each database, e.g

- schemas, tables, attributes, functions, types, ...
- there is a separate copy of each "local" table in each database
- a copy of many "global" tables is made when a new database is created

---

... PostgreSQL Catalog

Global installation data is recorded in shared tables

- users/groups: pg_authid (pg_shadow), pg_auth_members (pg_group)
- DBs/namespaces: pg_database, pg_namespace

Each kind of DB object has table(s) to describe it, e.g.

- tables: pg_class, pg_attr, pg_constraint, pg_attrdef
- functions: pg_proc, pg_operator, pg_aggregate
- indexes: pg_index, pg_am, pg_amop, pg_amproc

... PostgreSQL Catalog

PostgreSQL tuples contain

- owner-specified attributes (from `create table`)
- system-defined attributes

`oid`	unique identifying number for tuple (optional)
`tableoid`	which table this tuple belongs to
`xmin/xmax`	which transaction created/deleted tuple (for MVCC)

OIDs are used as primary keys in many of the catalog tables.

Representing Users/Groups

In version 8, PostgreSQL merged notions of users/groups into roles.

Represented by two base tables: `pg_authid`, `pg_auth_members`

View `pg_shadow` gives a more symbolic view of `pg_authid`.

View `pg_user` gives a copy of `pg_shadow` with passwords "hidden".

`CREATE|ALTER|DROP USER` statements modify `pg_authid` table.

`CREATE|ALTER|DROP GROUP` statements modify `pg_auth_members` table.

Both tables are global (shared across all DBs in a cluster).

... Representing Users/Groups

`pg_authid` table contains information about roles:

`oid`	unique integer key for this role
`rolname`	symbolic name for role (PostgreSQL identifier)
`rolpassword`	plain or md5-encrypted password
`rolcreatedb`	can create new databases
`rolsuper`	is a superuser (owns server process)
`rolcatupdate`	can update system catalogs

etc. etc.

... Representing Users/Groups

`pg_shadow` view contains information about users:

`usename`	symbolic user name (e.g. `'jas'`)
`usesysid`	integer key to reference user (`pg_authid.oid`)
`passwd`	plain or md5-encrypted password
`usecreatdb`	can create new databases
`usesuper`	is a superuser (owns server process)
`usecatupd`	can update system catalogs

etc. etc.

... Representing Users/Groups

`pg_group` view contains information about user groups:

`groname`	group name (e.g. `'developers'`)

`grosysid`	integer key to reference group
`grolist[]`	array containing group members (vector of refs to `pg_authid.oid`)

Note the use of multi-valued attribute (PostgreSQL extension)

Representing High-level Objects

Above the level of individual DB schemata, we have:

- *databases* ... represented by `pg_database`
- *schemas* ... represented by `pg_namespace`
- *table spaces* ... represented by `pg_tablespace`

These tables are global to each PostgreSQL cluster.

Keys are names (strings) and must be unique within cluster.

... Representing High-level Objects

`pg_database` contains information about databases:

`datname`	database name (e.g. `'mydb'`)
`datdba`	database owner (refs `pg_authid.oid`)
`datpath`	where files for database are stored (if not in the PGDATA directory)
`datacl[]`	access permissions
`datistemplate`	can be used to clone new databases (e.g. `template0`, `template1`)

etc. etc.

... Representing High-level Objects

Digression: access control lists (`acl`)

PostgreSQL represents access via an array of access elements.

Each access element contains:

```
UserName=Privileges/Grantor
group GroupName=Privileges/Grantor
```

where *Privileges* is a string enumerating privileges, e.g.

```
jas=arwdRxt/jas,fred=r/jas,joe=rwad/jas
```

... Representing High-level Objects

`pg_namespace` contains information about schemata:

`nspname`	namespace name (e.g. `'public'`)
`nspowner`	namespace owner (refs `pg_authid.oid`)
`nspacl[]`	access permissions

Note that nspname is a key and must be unique across cluster.

... Representing High-level Objects

pg_tablespace contains information about tablespaces:

spcname	tablespace name (e.g. 'disk5')
spcowner	tablespace owner (refs pg_authid.oid)
spclocation	full filepath to tablespace directory
spcacl[]	access permissions

Two pre-defined tablespaces:

- pg_default ... corresponds to PGDATA/base directory
- pg_global ... corresponds to PGDATA/global directory

Representing Tables

Entries in multiple catalog tables are required for each user-level table.

Due to O-O heritage, base table for tables is called pg_class.

The pg_class table also handles other "table-like" objects:

- views ... represents attributes/domains of view
- composite (tuple) types ... from CREATE TYPE AS
- "toast" tables ... for holding over-sized tuples

pg_class also handles sequences, indexes, and other "special" objects.

Tuples in pg_class have an OID, used as primary key.

... Representing Tables

pg_class contains information about tables:

relname	name of table (e.g. employee)
relnamespace	schema in which table defined (refs pg_namespace.oid)
reltype	data type corresponding to table (refs pg_type.oid)
relowner	owner (refs pg_authid.oid)
reltuples	# tuples in table
relacl	access permissions

... Representing Tables

pg_class also holds various flags/counters for each table:

relkind	what kind of object 'r' = ordinary table, 'i' = index, 'v' = view 'c' = composite type, 'S' = sequence, 's' = special

`relnatts`	# attributes in table (how many entries in `pg_attribute` table)
`relchecks`	# of constraints on table (how many entries in `pg_constraint` table)
`relhasindex`	table has/had an index?
`relhaspkey`	table has/had a primary key?

etc.

... Representing Tables

`pg_type` contains information about data types:

`typname`	name of type (e.g. `'integer'`)
`typnamespace`	schema in which type defined (refs `pg_namespace.oid`)
`typowner`	owner (refs `pg_authid.oid`)
`typtype`	what kind of data type 'b' = base type, 'c' = complex (row) type, ...

Note: a complex type is automatically created for each table
(defines "type" for each tuple in table; also, type for functions returning `SETOF`)

... Representing Tables

`pg_type` also contains storage-related information:

`typlen`	how much storage used for values (-1 for variable-length types, e.g. `text`)
`typalign`	memory alignment for values ('c' = byte-boundary, 'i' = 4-byte-boundary, ...)
`typrelid`	table associated with complex type (refs `pg_class.oid`)
`typstorage`	where/how values are stored ('p' = in-tuple, 'e' = in external table, compressed?)

(We discuss more details of the `pg_type` table later ...)

... Representing Tables

`pg_attribute` contains information about attributes:

`attname`	name of attribute (e.g. `'empname'`)
`attrelid`	table this attribute belongs to (refs `pg_class.oid`)
`attnum`	attribute position (1..n, sys attrs are -ve)
`atttypid`	data type of this attribute

(refs `pg_type.oid`)

`(attrelid,attnum)` is unique, and used as primary key.

... Representing Tables

`pg_attribute` also holds storage-related information:

`attlen`	storage space required by attribute (copy of `pg_type.typlen` for fixed-size values)
`atttypmod`	storage space for var-length attributes (e.g. 6+ATTR_HEADER_SIZE for `char(6)`)
`attalign`	memory-alignment info (copy of `pg_type.typalign`)
`attndims`	number of dimensions if attr is an array

... Representing Tables

`pg_attribute` also holds constraint/status information:

`attnotnull`	attribute may not be null?
`atthasdef`	attribute has a default values (value is held in `pg_attrdef` table)
`attisdropped`	attribute has been dropped from table

Also has notion of large data being stored in a separate table (so-called "TOAST" table).

... Representing Tables

An SQL DDL statement like

```
create table MyTable (
   a int unique not null,
   b char(6)
);
```

will cause entries to be made in the following tables:

- `pg_class` ... one tuple for the table as a whole
- `pg_attribute` ... one tuple for each attribute
- `pg_type` ... one tuple for the row-type

... Representing Tables

The example leads to a series of database changes like

```
rel_oid := new_oid(); user_id = current_user();
insert into
   pg_class(oid,name,owner,kind,pages,tuples,...)
   values (rel_oid, 'mytable', user_id, 'r', 0, 0, ...)
select oid,typlen into int_oid,int_len
from   pg_type where typname = 'int';
insert into
   pg_attribute(relid,name,typid,num,len,typmod,notnull...)
   values (rel_oid, 'a', int_oid, 1, int_len, -1, true, ...)
select oid,typlen into char_oid,char_len
from   pg_type where typname = 'char';
insert into
```

```
   pg_attribute(relid,name,typid,num,len,typmod,notnull...)
   values (rel_oid, 'b', char_oid, 2, -1, 6+4, false, ...)
insert into
   pg_type(name,owner,len,type,relid,align,...)
   values ('mytable', user_id, 4, 'c', rel_oid, 'i', ...)
```

... Representing Tables

`pg_attrdef` contains information about default values:

`adrelid`	table that column belongs to (refs `pg_class.oid`)
`adnum`	which column in the table (refs `pg_attribute.attnum`)
`adsrc`	readable representation of default value
`adbin`	internal representation of default value

... Representing Tables

`pg_constraint` contains information about constraints:

`conname`	name of constraint (not unique)
`connamespace`	schema containing this constraint
`contype`	kind of constraint 'c' = check, 'u' = unique, 'p' = primary key, 'f' = foreign key
`conrelid`	which table (refs `pg_class.oid`)
`conkey`	which attributes (vector of values from `pg_attribute.attnum`)
`consrc`	check constraint expression

(Names are automatically generated from context (`fkey,check`) if not supplied)

... Representing Tables

For foreign-key constraints, **`pg_constraint`** also contains:

`confrelid`	referenced table for foreign key
`confkey`	key attributes in foreign table
`conkey`	corresponding attributes in local table

Foreign keys also introduce triggers to perform checking.

For column-specific constraints:

`consrc`	readable check constraint expression
`conbin`	internal check constraint expression

... Representing Tables

An SQL DDL statement like

```
create table MyOtherTable (
   x int check (x > 0),
   y int references MyTable(a),
   z int default -1
);
```

will cause similar entries as before in catalogs, plus

- `pg_constraint` ... one tuple for `x` and `y`
- `pg_attrdef` ... one tuple for `z` default

... Representing Tables

The example leads to a series of database changes like

```
rel_oid := new_oid(); user_id = current_user();
insert into
   pg_class(oid,name,owner,kind,pages,tuples,...)
   values (rel_oid, 'myothertable', user_id, 'r', 0, 0, ...)
select oid,typlen into int_oid,int_len
from   pg_type where typname = 'int';
select oid into old_oid
from   pg_class where relname='mytable';
-- pg_attribute entries for attributes x=1, y=2, z=3
insert into
   pg_attrdef(relid,num,src,bin)
   values (rel_oid, 3, -1, {CONST :...})
insert into
   pg_constraint(type,relid,key,src,...)
   values ('c', rel_oid, {1}, '(x > 0)', ...)
insert into
   pg_constraint(type,relid,key,frelid,fkey,...)
   values ('f', rel_oid, {2}, old_oid, {1}, ...)
```

Representing Functions

Stored procedures (functions) are defined as

```
create function power(int x, int y) returns int
as $$
declare  i int;  product int := 1;
begin
   for i in 1..y loop
      product := product * x;
   end loop;
   return product;
end;
$$ language plpgsql;
```

Stored procedures are represented in the catalog via

- an entry in the `pg_proc` table
- with references to `pg_type` table for signature

... Representing Functions

`pg_proc` contains information about functions:

`proname`	name of function (e.g. `substr`)
`pronamespace`	schema in which function defined (refs `pg_namespace.oid`)

`proowner`	owner (refs `pg_authid.oid`)
`proacl[]`	access permissions

etc.

... Representing Functions

`pg_proc` also contains argument/usage information:

`pronargs`	how many arguments
`prorettype`	return type (refs `pg_type.oid`)
`proargtypes[]`	argument types (ref `pg_type.oid` vector)
`proreset`	returns set of values of `prorettype`
`proisagg`	is function an aggregate?
`proisstrict`	returns null if any arg is null
`provolatile`	return value depends on side-effects? ('i' = immutable, 's'= stable, 'v' = volatile)

... Representing Functions

`pg_proc` also contains implementation information:

`prolang`	what language function written in
`prosrc`	source code if interpreted (e.g. PLpgSQL)
`probin`	additional info on how to invoke function (interpretation is language-specific)

... Representing Functions

Consider two alternative ways of defining a x^2 function.

```
sq.c  int square_in_c(int x) { return x * x; }

create function square(int) returns int
as '/path/to/sq.o', 'square_in_c' language 'C';
```

or

```
create function square(int) returns int
as $$
begin
    return $1 * $1;
end;
$$ language plpgsql;
```

... Representing Functions

The above leads to a series of database changes like

```
user_id := current_user();
select oid,typlen into int_oid,int_len
from   pg_type where typname = 'int';
insert into
   pg_proc(name,owner,rettype,nargs,argtypes,
           prosrc,probin...)
   values ('square', user_id, int_oid, 1, {int_oid},
           'square_in_c', '/path/to/sq.o', ...)
-- or
insert into
   pg_proc(name,owner,rettype,nargs,argtypes,
           prosrc,probin...)
   values ('square', user_id, int_oid, 1, {int_oid},
           'begin return $1 * $1; end;', '-', ...)
```

... Representing Functions

Users can define their own *aggregate* functions (like `max()`).

Requires definition of three components:

- *state* to accumulate partial values during the scan
- *update function* to maintain state after each tuple
- *output function* to return the final acucmulated result

This information is stored in the **`pg_aggregate`** catalog.

The aggregate's name is stored in the **`pg_proc`** catalog.

... Representing Functions

Consider defining your own `average()` function

Need to define a new aggregate:

```
create aggregate average (
   basetype   = integer,
   sfunc      = int_avg_accum,
   stype      = int[],
   finalfunc  = int_avg_result,
   initcond   = '{0,0}'
);
```

and need to define functions to support aggregate ...

... Representing Functions


```
create function
       int_avg_accum(state int[], int) returns int[]
as $$
declare res int[2];
begin
   res[1] := state[1] + $2; res[2] := res[2] + 1;
   return res;
end;
$$ language plpgsql;

create function
       int_avg_result(state int[]) returns int
as $$
begin
   if (state[2] = 0) then return null; end if;
   return (state[1] / state[2]);
end;
$$ language plpgsql;
```

... Representing Functions

Users can define their own *operators* to use in expressions.

Operators are syntactic sugar for unary/binary functions.

Consider defining an operator for the `power(x,y)` function:

```
create operator ** (
   procedure = power, leftarg = int, rightarg = int
);

-- which can be used as
select 4 ** 3;
-- giving a result of 64
```

Operator definitions are stored in **pg_operator** catalog.

Representing Types

Users can also define new *data types*, which includes

- data structures for objects of the type
- type-specific functions, aggregates, operators
- type-specific indexing (access) methods

Consider defining a 3-dimensional point type for spatial data:

```
create type point3d (
   input = point3d_in,     -- function to parse values
   output = point3d_out,   -- function to display values
   internallength = 24,    -- space for three float8's
   alignment = double      -- align tuples properly
);
```

... Representing Types

pg_type additional fields for user-defined types:

`typinput`	text input conversion function
`typoutput`	text output conversion function
`typreceive`	binary input conversion function
`typsend`	binary output conversion function

All attributes are references to `pg_proc.oid`

... Representing Types

All data types need access methods for querying.

The following catalogs tables are involved in this:

- **pg_am** ... main definition of access method
- **pg_opclass** ... access operator classes
- **pg_amop** ... operators for indexed access
- **pg_amproc** ... support procedures for AM

... Representing Types

pg_am holds information about access methods:

`amname`	name of access method (e.g. `btree`)

`amowner`	owner (refs `pg_authid.oid`)
`amorder strategy`	operator for determining sort order (0 if unsorted)
`amcanunique`	does AM support unique indexes?
`ammulticol`	does AM support multicolumn indexes?
`amindexnulls`	does AM support NULL index entries?
`amconcurrent`	does AM support concurrent updates?

... Representing Types

`pg_am` also contains links to access functions:

`amgettuple`	"next valid tuple" function
`ambeginscan`	"start new scan" function
`amrescan`	"restart this scan" function
`amendscan`	"end this scan" function
`amcostestimate`	estimate cost of index scan

All attributes are references to `pg_proc.oid`

Functions drive the query evaluation process.

... Representing Types

`pg_am` also contains links to update functions:

`aminsert`	"insert this tuple" function
`ambuild`	"build new index" function
`ambulkdelete`	bulk delete function
`amvacuum cleanup`	post-vacuum cleanup function

All attributes are references to `pg_proc.oid`

Functions implement different aspects of updating data/index files.

... Representing Types

Built-in access methods:

- **`heap`** ... simple sequence of pages, sequential access
- **`btree`** ... ordered access by key, Lehman-Yao version
- **`hash`** ... assiocative access, Litwin's linear hashing
- **`rtree`** ... spatial data index, quadratic split version
- **`GiST`** ... generalised tree indexes (e.g. B-trees, R-trees)
- **`SP-GiST`** ... space-partitioned search trees (e.g. k-d trees)
- **`GIN`** ... generalised inverted index (e.g. (key,docs) pairs)

Some access methods introduce additional files (e.g. B-tree)

Produced: 22 Aug 2019