# CS186: Introduction to Database Systems

## Lecture 2 – Disks, Buffers, Storage

(Book Ch: 9.1,9.3 & 9.4)

Michael Franklin
Fall 2013

## Plan for today

- First, we'll finish up our quick overview of SQL, focusing on some simple queries.
- Then, we'll talk a bit about storage and memory hierarchies
- Finally, we'll cover buffer management and buffer replacement policies

## The SQL DML

- Single-table queries are straightforward.
- To find records for all 18 year old students with gpa's above 2.0, we can write:

```
SELECT *
  FROM Students S
 WHERE S.age=18
    AND S.gpa > 2.0
```

To get just names and logins, replace the first line:

```
SELECT S.name, S.login
```

## Basic SQL Queries

```
SELECT    [DISTINCT] target-list
FROM      relation-list
WHERE     qualification
```

- <u>*relation-list*</u> : A list of relation names
  – possibly with a *range-variable* after each name
- <u>*target-list*</u> : A list of attributes of tables in *relation-list*
- <u>*qualification*</u> : Comparisons combined using AND, OR and NOT.
  – Comparisons are Attr *op* const or Attr1 *op* Attr2, where *op* is one of =≠<>≤≥
- <u>*DISTINCT*</u>: (optional) indicates that the answer should have no duplicates.
  – In SQL SELECT, the default is that duplicates are <u>*not*</u> eliminated! (Result is called a "multiset")

## Querying Multiple Relations

```
SELECT S.name, E.cid
 FROM Students S, Enrolled E
 WHERE S.sid=E.sid AND E.grade='B'
```

- Can specify a join over two tables as follows:

| sid | cid | grade |
|---|---|---|
| 53831 | Carnatic101 | C |
| 53831 | Reggae203 | B |
| 53650 | Topology112 | A |
| 53666 | History105 | B |

| sid | name | login | age | gpa |
|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@ee | 18 | 3.2 |

Note: obviously no referential integrity constraints have been used here.

result =

| S.name | E.cid |
|---|---|
| Jones | History105 |

## Basic Query Semantics

The **<u>Semantics</u>** of a SQL query are defined in terms of the following conceptual evaluation strategy:

1. do FROM clause: compute <u>*cross-product*</u> of tables (e.g., Students and Enrolled).
2. do WHERE clause: Check conditions, discard tuples that fail.
3. do SELECT clause: Delete unwanted fields.
4. If DISTINCT specified, eliminate duplicate rows.

Probably the least efficient way to compute a query!

- A *query optimizer* will find more efficient strategies to get the *same answer*.

## Step 1 – Cross Product

| S.sid | S.name | S.login | S.age | S.gpa | E.sid | E.cid | E.grade |
|---|---|---|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 | 53831 | Carnatic101 | C |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53832 | Reggae203 | B |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53650 | Topology112 | A |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53666 | History105 | B |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53831 | Carnatic101 | C |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53831 | Reggae203 | B |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53650 | Topology112 | A |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53666 | History105 | B |

| sid | cid | grade |
|---|---|---|
| 53831 | Carnatic101 | C |
| 53831 | Reggae203 | B |
| 53650 | Topology112 | A |
| 53666 | History105 | B |

| sid | name | login | age | gpa |
|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@ee | 18 | 3.2 |

## Step 2) Discard tuples that fail predicate

| S.sid | S.name | S.login | S.age | S.gpa | E.sid | E.cid | E.grade |
|---|---|---|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 | 53831 | Carnatic101 | C |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53832 | Reggae203 | B |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53650 | Topology112 | A |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53666 | History105 | B |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53831 | Carnatic101 | C |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53831 | Reggae203 | B |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53650 | Topology112 | A |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53666 | History105 | B |

```
SELECT S.name, E.cid
 FROM Students S, Enrolled E
 WHERE S.sid=E.sid AND E.grade='B'
```

## Step 3) Discard Unwanted Columns

| S.sid | S.name | S.login | S.age | S.gpa | E.sid | E.cid | E.grade |
|---|---|---|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 | 53831 | Carnatic101 | C |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53832 | Reggae203 | B |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53650 | Topology112 | A |
| 53666 | Jones | jones@cs | 18 | 3.4 | 53666 | History105 | B |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53831 | Carnatic101 | C |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53831 | Reggae203 | B |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53650 | Topology112 | A |
| 53688 | Smith | smith@ee | 18 | 3.2 | 53666 | History105 | B |

```
SELECT S.name, E.cid
 FROM Students S, Enrolled E
 WHERE S.sid=E.sid AND E.grade='B'
```

## Aggregate Operators

For calculation and analytics

- COUNT (*)
- COUNT ([DISTINCT] A)
- SUM ([DISTINCT] A)
- AVG (A)
- MAX (A)
- MIN (A)

```
SELECT COUNT (*)
FROM Students;
```

```
SELECT AVG (S.gpa)
FROM Students S
WHERE S.age=18;
```

```
SELECT COUNT (DISTINCT S.age)
FROM Students S
WHERE S.name='Bob';
```

## GROUP BY and HAVING

- Sometimes, we want to apply aggs to each of several *groups* of tuples.
  - This query computes the average gpa per major (assume students have a "major" attribute)

```
SELECT S.major, AVG (S.gpa) as AvgGPA
FROM Students S
GROUP BY S.major ;
```

  - If you want to exclude "small" majors, use Having:

```
SELECT S.major, AVG (S.gpa) as AvgGPA
FROM Students S
GROUP BY S.major
HAVING COUNT (*) > 10 ;
```

## (Slightly) Less Basic SQL Queries

```
SELECT      [DISTINCT] target-list
FROM        relation-list
WHERE       qualification
GROUP BY    grouping-list
HAVING      group-qualification
```

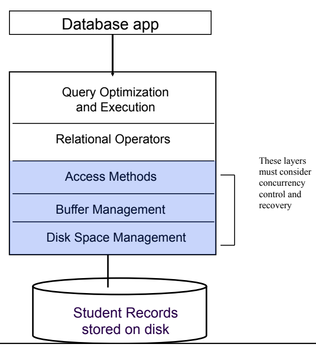
## Conceptual Evaluation

- The cross-product of *relation-list* is computed, tuples that fail *qualification* are discarded, `*unnecessary*' fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.
- One answer tuple is generated per qualifying group.

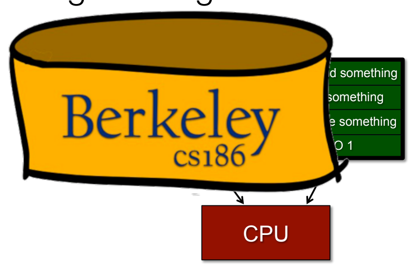
## Conceptual Evaluation (cont.)

- The *group-qualification* is then applied to eliminate some groups.
  - Expressions in *group-qualification* must have a *single value per group*!
  - That is, attributes in *group-qualification* must be arguments of an aggregate op or must also appear in the *grouping-list*.
- One answer tuple is generated per qualifying group.

## Okay: Let's start from the bottom up…




## Dealing with Big Data



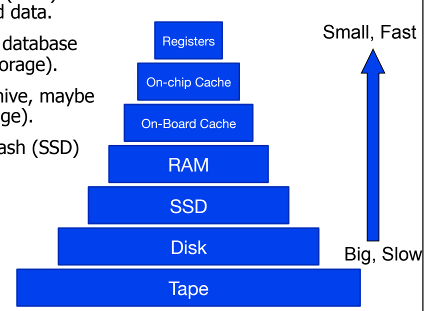

The Von Neumann machine

## Why Not Store It All in Main Memory?

- *Costs too much.* $100 will buy you either ~100 GB of RAM or around 2000 GB (2 TB) of disk today.
  - High-end Databases today can be in the Petabyte (1000TB) range.
  - Approx 60% of the cost of a production system is in the disks.
- *Main memory is volatile.* We want data to be saved between runs. (Obviously!)
- Note, some specialized systems do store entire database in main memory.
  - Vendors claim 10x speed up vs. traditional DBMS running in main memory.

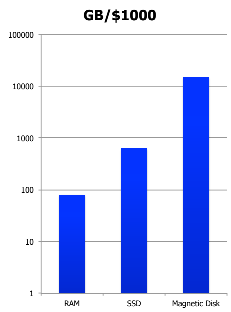
## The Storage Hierarchy

- Main memory (RAM) for currently used data.
- Disk for main database (secondary storage).
- Tapes for archive, maybe (tertiary storage).
- The role of Flash (SSD) still unclear

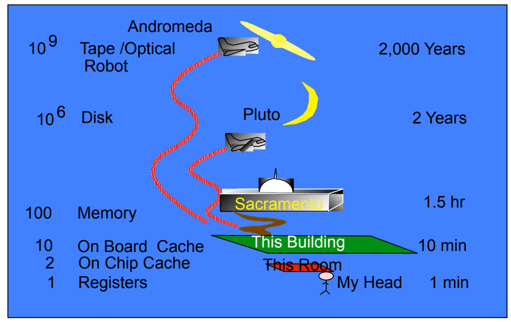


## Economics

For $1000, PCConnection offers:
- ~80GB of RAM
- ~1TB of Solid State Disk
- ~19TB of Magnetic Disk

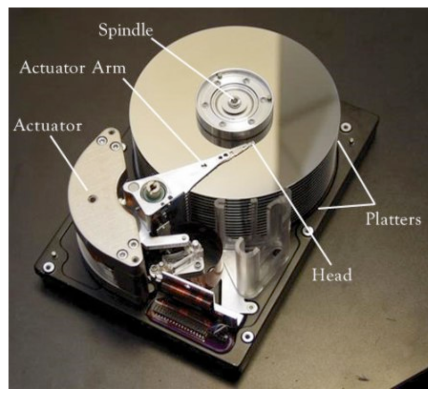



## Jim Gray's Latency Analogy: How Far Away is the Data?




## Disks and Files

- Today: Most data is stored on magnetic disks.
  - Disks are a mechanical anachronism!
- Major implications!
  - No "pointer derefs". Instead, an API:
    - READ: transfer "page" of data from disk to RAM.
    - WRITE: transfer "page" of data from RAM to disk.
  - Both API calls expensive
    - Plan carefully!
  - An explicit API can be a good thing
    - Minimizes the kind of pointer errors you see in C



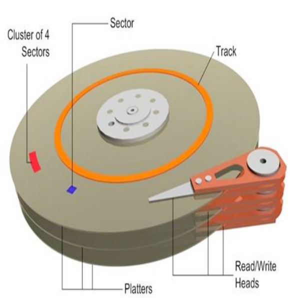

## Anatomy of a Disk

The platters spin

The arm assembly is moved in or out to position a head on a desired track. Tracks under heads make a *cylinder* (imaginary!).

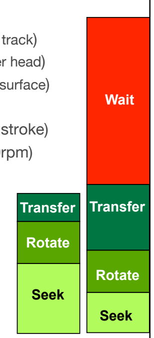
Only one head reads/writes at any one time.

- *Block size* is a multiple of *sector size* (which is fixed)




## Accessing a Disk Page

- Time to access (read/write) a disk block:
  - ***seek time*** (moving arms to position disk head on track)
  - ***rotational delay*** (waiting for block to rotate under head)
  - ***transfer time*** (actually moving data to/from disk surface)
- Seek time and rotational delay dominate.
  - Seek time varies from about 1 to 15msec (full stroke)
  - Rotational delay varies from 0 to 8msec (7200rpm)
  - Transfer rate is < 0.1msec per 8KB block
- Key to lower I/O cost:
  reduce seek/rotation delays!
  Hardware vs. software solutions?
- Also note: For **shared disks** most time spent waiting in queue for access to arm/controller



## Arranging Pages on Disk

- `Next' block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder
- Arrange file pages sequentially on disk
  - minimize seek and rotational delay.
- For a sequential scan, pre-fetch
  - several pages at a time!

## From the DB Administrator's View

Modern disk structures are so complex even industry experts refer to them as "black boxes". Today there is no alignment to physical disk sectors, no matter what we believe. Disks do not map sectors to physical regions in a way that we can understand from outside the box; the simplistic "geometry" reported by the device is an artifice.

from Microsoft's "Disk Partition Alignment Best Practices for SQL Server"

## Disk Space Management

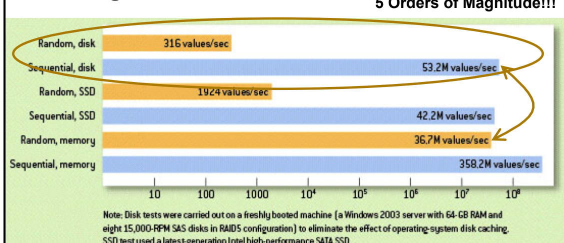
- Lowest layer of DBMS software manages space on disk (using OS file system or not?).
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Best if a request for a *sequence* of pages is satisfied by pages stored sequentially on disk! Higher levels don't need to know if/how this is done, or how free space is managed.

## Notes on Flash (SSD)

- Various technologies, we focus on NAND
  - suited for volume data storage
  - alternative: NOR Flash
- Read is random access and fast
  - E.g. 512 ***Bytes*** at a time
- Write is coarser grained and slower
  - E.g. 16-512 ***KBytes*** at a time.
  - Can get slower over time
- Some concern about write endurance
  - 100K cycle lifetimes?
- Still changing quickly

## Retrieval Costs – 4 Byte Integers

**5 Orders of Magnitude!!!**




From A. Jacobs, "The Pathologies of Big Data", ACM Queue Magazine, July 2009

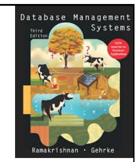
## Storage Pragmatics & Trends

- Many significant DBs are not that big.
  - Daily weather, round the globe, 1929-2009: 20GB
  - 2000 US Census: 200GB
  - 2009 English Wikipedia: 14GB
- But data sizes grow faster than Moore's Law
- What is the role of disk, flash, RAM?
  - The subject of much debate/concern!

## Bottom Line (for now!)

- Very Large DBs: relatively traditional
  - Disk still the best cost/MB by orders of magnitude
- Smaller DB story is changing quickly
  - Entry cost for disk is not cheap, so flash wins at the low end
  - Many interesting databases fit in RAM (e.g.,SAP HANA)
- Lots of change brewing on the HW storage tech side
- Lots of uncertainty on the SW/usage side
  - It's Big: Can generate and archive data cheaply and easily
  - It's Small: Many rich data sets have (small) fixed size
- Hmmm…!
- This semester, we'll mostly focus on traditional RAM and Disk.

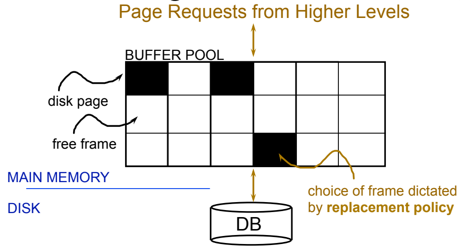
## Administrivia




- The web page is live:
  http://tinyurl.com/cs186fall2013
- Piazza site is up and active
- Bunny 1 and Assignment 1 released later this week (aiming for Wednesday)
- Please attend your assigned Section (until enrollments get sorted out)
- Book – read it

Database Management Systems **3rd Edition** by Ramakrishnan and Gehrke.

## Buffer Management in a DBMS




- *Data must be in RAM for DBMS to operate on it!*
  - *The query processor refers to data using virtual memory addresses.*
- *Buffer Mgr hides the fact that not all data is in RAM*

## Some Terminology…

- Disk Page – the unit of transfer between the disk and memory
  Typically set as a config parameter for the DBMS.
  Typical value between 4 KBytes to 32 KBytes.
- Frame – a unit of memory
  Typically the same size as the Disk Page Size
- Buffer Pool – a collection of frames used by the DBMS to temporarily keep data for use by the query processor.
  - note: We will sometime use the term "buffer" and "frame" synonymously.

Question: When would you use a larger page size rather than a smaller one?

## When a Page is Requested ...

- If requested page IS in the pool:
  - *Pin* the page and return its address.
- Else, if requested page IS NOT in the pool:
  - If a free frame exists, choose it, Else:
    - Choose a frame for *replacement (only un-pinned pages are candidates)*
    - If chosen frame is "dirty", write it to disk
  - Read requested page into chosen frame
  - *Pin* the page and return its address.

Q: What information about buffers and their contents must the system maintain?

## Buffer Control Blocks (BCBs):

***<frame#, pageid, pin_count, dirty>***

- A page may be requested many times, so
  - a *pin count* is used.
  - To pin a page, pin_count++
  - A page is a candidate for replacement iff *pin_count* == 0 (*"unpinned"*)
- Requestor of page must eventually unpin it.
  - pin_count--
- Must also indicate if page has been modified:
  - *dirty* bit is used for this.
  
  Q: Why is this important?

Q: How should BCB's be organized?

## Additional Buffer Mgr Notes

- BCB's are hash indexed by pageID
- Concurrency Control & Recovery may entail additional I/O when a frame is chosen for replacement.

  (*Write-Ahead Log* protocol; more later.)
- If requests can be predicted (e.g., sequential scans) pages can be pre-fetched several pages at a time.

## Buffer Replacement Policy

- Frame is chosen for replacement by a *replacement policy:*
  - Least-recently-used (LRU), MRU, Clock, etc.
- This policy can have big impact on the number of disk reads and writes.
  - Remember, these are slooooooooooow.
- **BIG IDEA** – throw out the page that you are least likely to need in the future.
  - Q: How do you predict the future?
- Efficacy depends on the *access pattern.*

## Random Quote

Making predictions is hard,
especially about the future.

Q: Who said this?
a) Yogi Berra
b) Neils Bohr
c) Mark Twain
d) A famous Statistician

http://www.larry.denenberg.com/predictions.html

## LRU Replacement Policy

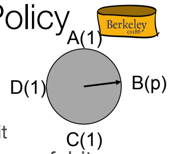
*Least Recently Used (LRU)*

1) for each page in buffer pool, keep track of time last *unpinned*

*What else might you keep track off?*
*How would that impact performance?*

2)Replace the frame that has the oldest (earliest) time
- Most common policy: intuitive and simple
  - Based on notion of "Temporal Locality"
  - Works well to keep "working set" in buffers.
- Implemented through doubly linked list of BCBs
  - Requires list manipulation on unpin

## "Clock" Replacement Policy




- An approximation of LRU
- Arrange frames into a cycle, store one *reference bit per frame*
  - Can think of this as the 2nd chance bit
- When pin count reduces to 0, turn on ref. bit
- When replacement necessary

```
do for each page in cycle {
    if (pincount == 0 && ref bit is on)
        turn off ref bit;
    else if (pincount == 0 && ref bit is off)
        choose this page for
            replacement;
} until a page is chosen;
```

Questions:
How like LRU?
Problems?

## Some issues with LRU

- *Problem: Sequential flooding*
  - LRU + repeated sequential scans.
  - # buffer frames < # pages in file means each page request causes an I/O. *MRU* much better in this situation (but not in all situations, of course).
- *Problem: "cold" pages can hang around a long time before they are replaced.*

## “2Q” Replacement Policy

- One Queue (A1) has pages that have been referenced only once.
  - new pages enter here
- A second, LRU Queue (Am) has pages that have been referenced (pinned) multiple times.
  - pages get promoted from A1 to here
- Replacement victims are usually taken from A1
  - Q: Why????

## DBMS vs. OS File System

OS does disk space & buffer mgmt: why not let OS manage these tasks?

- Some limitations, e.g., files can’t span disks.
  - Note, this is changing --- OS File systems are getting smarter (i.e., more like databases!)
- Buffer management in DBMS requires ability to:
  - pin a page in buffer pool, force a page to disk & order writes (important for implementing CC & recovery)
  - adjust *replacement policy,* and pre-fetch pages based on access patterns in typical DB operations.
- Q: Compare DBMS Buffer Mgmt to OS Virtual Memory? to Processor Cache?