Final Review

CS634

Slides based on "Database Management Systems" 3rd ed, Ramakrishnan and Gehrke

Coverage

- Text, chapters 8 through 18, 25 (hw1 – hw6)
- PKs, FKs, E-R to Relational: Text, Sec. 3.2-3.5, to pg. 77 inclusivel, hw1
- Basics of Disks and RAID
- Indexing: Hash Index, B+Tree, hw2, hw3
- Cloud VM, mysql DBA actions, hw3
- Query evaluation & optimization, chap 14-15. hw4

See MidtermReview for above.   Since midterm exam:

- Transactions, Concurrency Control, chap. 16-17, hw5
- Mysql DBA actions, hw5, hw6
- Crash Recovery, chap 18, hw6
- Data Warehousing and Decision Support, chap 25 to pg. 856, hw6
- Basics of Docker containers, hw6

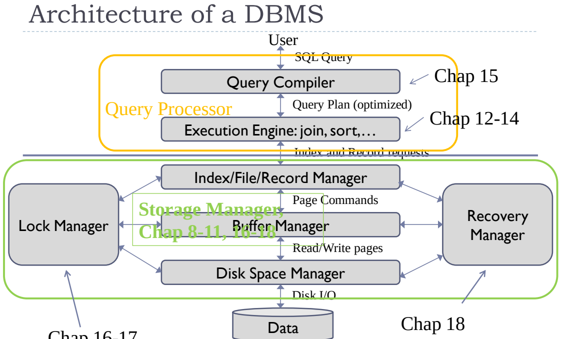
Highlights of before-midterm coverage

- Disks: idea of cylinders, LBNs running in "next" order
- RAID levels
- Concept of "File": sequence of pages, possibly on multiple disks, accessible by random access by page no.
  - Unordered "heap", records have RIDs for random access
  - Sorted (less common) by some record key
  - Clustered file (nearly sorted by some record key)
- Concept of an index File: has a key for lookup to its records
  - Itself can by a heap File or a clustered File (then a clustered index)
  - Its records are called "data entries", three formats listed on pg. 276
    - The whole data "row", which contains the key
    - (key, RID) where the data is found by the RID (in another File)
    - Book also lists (key, list of RIDs), but this is just a compression

Highlights of before-midterm coverage

- A Table is implemented by one or more Files
  - Heap file for data records plus 0 or more non-clustered indexes (themselves in heap files)
  - Clustered file for data records (Alt. 1) plus 0 or more non-clustered indexes (themselves in heap files)
  - Clustered file for data entries (Alt. 2) plus heap file in index-sorted order, plus 0 or more non-clustered indexes.
  - A table can have only one clustered index!
- Normally, only one index can be used at a time for access to table data by the storage engine (we saw this later), so see cases in Chap 8: heap file with unclustered tree index, heap file with clustered index, etc.
- Chap. 10: concentrate on B-tree case
- Chap. 11: concentrate on linear hashing
- Chap. 12: access path, index matching rules, selectivity, reduction factors, query plans, including use of indexes
- Chap. 13: external merge sort
- Chap. 14: More on matching indexes, projection by hashing, sorting, join methods
- Chap. 15: Evaluating alternative plans, incl. multiple-index plans, index-only evaluation.

Architecture of a DBMS



A first course in database systems, 3rd ed, Ullman and Widom

Single-table Plans With Indexes

- There are four cases:

1. Single-index access path
   - Each matching index offers an alternative access path
   - Choose one with lowest I/O cost
   - Non-primary conjuncts, projection, aggregates/grouping applied next
2. Multiple-index access path
   - Each of several indexes used to retrieve set of rids
   - Rid sets intersected, result sorted by page id
   - Retrieve each page only once
   - Non-primary conjuncts, projection, aggregates/grouping applied next

Plans With Indexes (contd.)

3. Tree-index access path: extra possible use…
   - If GROUP BY attributes prefix of tree index, retrieve tuples in order required by GROUP BY
   - Apply selection, projection for each retrieved tuple, then aggregate
   - Works well for clustered indexes

Example: With tree index on rating

```
SELECT count(*), max(age)
FROM Sailors S
GROUP BY rating
```

Plans With Indexes (contd.)

3. Index-only access path
   - If all attributes in query included in index, then there is no need to access data records: **index-only scan**
   - If index matches selection, even better: only part of index examined
   - Does not matter if index is clustered or not!
   - If GROUP BY attributes prefix of a tree index, no need to sort!
   - Example: With tree index on rating

```
SELECT max(rating),count(*)
FROM Sailors S
```

   - Note count(*) doesn't require access to row, just RID.

Example Schema

Sailors (<u>*sid*: integer</u>, *sname*: string, *rating*: integer, *age*: real)
Reserves (<u>*sid*: integer, *bid*: integer, *day*: dates</u>, *rname*: string)

- Similar to old schema; *rname* added
- Reserves:
  - 40 bytes long tuple, 100K records, 100 tuples per page, 1000 pages
- Sailors:
  - 50 bytes long tuple, 40K tuples, 80 tuples per page, 500 pages
- Assume index entry size 10% of data record size

Cost Estimates for Single-Relation Plans

- Sequential scan of file:
  - *NPages(R)*
- Index *I* on primary key matches selection
  - *Cost is Height(I)+1 for a B+ tree, about 1.2 for hash index*
- Clustered index *I* matching one or more selects:
  - *NPages(CI) * product of RF's of matching selects*

  *Quick estimate: Npages(CI) = 1.1*NPages(TableData)*

  i.e. 10% more for needed keys
- Non-clustered index *I* matching one or more selects:
  - *(NPages(I)+NTuples(R)) * product of RF's of matching selects*

  *Quick estimate: Npages(I) = .1*Npages(R) (10% of data size)*

Example

```
SELECT S.sid
FROM Sailors S
WHERE S.rating=8
```

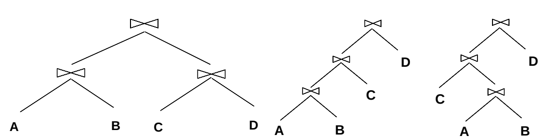
- File scan: retrieve all 500 pages
- Clustered Index *I* on *rating*

  (1/NKeys(I)) * (NPages(CI)) = (1/10) * (50+500) pages
- Unclustered Index *I* on *rating*

  (1/NKeys(I)) * (NPages(I)+NTuples(S)) = (1/10) * (50+40000) pages

Queries Over Multiple Relations

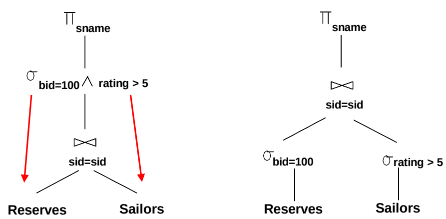
- In System R <u>*only left-deep join trees*</u> are considered
  - In order to restrict the search space
  - Left-deep trees allow us to generate all *fully pipelined* plans
    - Intermediate results not written to temporary files.
    - Not all left-deep trees are fully pipelined (e.g., sort-merge join)



Left-deep

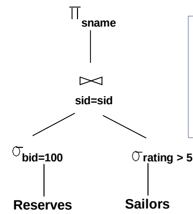
Example of push downs of selections

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND S.rating>5 AND R.bid=100
```



Push-down and pipelining

- But note that the right selection may not be best pushed-down: can't pipeline inner-table data for indexed NLJ



Can't be indexed NLJ here as it stands. Not left-deep. For NLJ, could materialize rating>5 result, with additional i/o. Or push rating condition back up. Then left-deep.

What are Transactions?

- So far, we looked at individual queries; in practice, a task consists of a sequence of actions
- E.g., "Transfer $1000 from account A to account B"
  - Subtract $1000 from account A
  - Subtract transfer fee from account A
  - Credit $1000 to account B
- A *transaction* is the DBMS's view of a user program:
  - Must be interpreted as "unit of work": either entire transaction executes, or no part of it executes/has any effect on DBMS
  - Two special final actions: COMMIT or ABORT

ACID Properties

Transaction Management must fulfill four requirements:

1. Atomicity: either all actions within a transaction are carried out, or none is
   - Only actions of committed transactions must be visible
2. Consistency: concurrent execution must leave DBMS in consistent state
3. Isolation: each transaction is protected from effects of other concurrent transactions
   - Net effect is that of **some sequential execution**
4. Durability: once a transaction commits, DBMS changes will persist
   - Conversely, if a transaction aborts/is aborted, there are no effects

Modeling Transactions

- User programs may carry out many operations …
  - Data-related computations
  - Prompting user for input, handling web requests
- … but the DBMS is only concerned about what data is read/written from/to the database
- A *transaction* is abstracted by a sequence of time-ordered read and write actions
  - e.g., R(X), R(Y), W(X), W(Y)
  - R=read, W=write, data element in parentheses
  - Each individual action is indivisible, or atomic
  - SQL UPDATE = R(X) W(X)

Concurrency: lost update anomaly

- Consider two transactions (in a really bad DB) where A = 100

T1:	A = A + 100
T2:	A = A + 100

- T1 & T2 are concurrent, running same transaction program
- T1& T2 both read old value, 100, add 100, store 200
- One of the updates has been lost!
- Consistency requirement: after execution, A should reflect all deposits (Money should not be created or destroyed)
- No guarantee that T1 will execute before T2 or vice-versa…
- … but the net effect must be equivalent to these two transactions running one-after-the-other in ***some*** order

Concurrency: lost update anomaly

- Consider two transactions (in a really bad DB) where A = 100
- T1 & T2 are concurrent, running same transaction program
- T1& T2 both read old value, 100, add 100, store 200
- One of the updates has been lost!
- Using R/W notation, marking conflicts: same data item, different transactions, at least one a write:

R1(A) R2(A)W2(A)C2W1(A)C1

- First arc says T1 → T2, second says T2→ T1, so there is a cycle in the dependency graph
- This execution is not allowed under 2PL

Strict Two-Phase Locking (Strict 2PL)

- Protocol steps
  - Each transaction must obtain a S *(shared)* lock on object before reading, and an X *(exclusive)* lock on object before writing.
  - All locks held are released when the transaction completes
    - (Non-strict) 2PL: Release locks anytime, but cannot acquire locks after releasing any lock.
- Strict 2PL allows only serializable schedules.
  - It simplifies transaction aborts
  - (Non-strict) 2PL also allows only serializable schedules, but involves more complex abort processing
- Strict 2PL prevents anomalies if the set of database items never changes: here insert and delete are excluded as not R or W. With insert/delete, need index locking.

Concurrency: lost update anomaly

R1(A) R2(A)W2(A)C2W1(A)C1

- First arc says T1 → T2, second says T2→ T3, so there is a cycle in the dependency graph
- This execution is not allowed under 2PL
- Run it under 2PL:

S1(A) R1(A) S2(A) R2(A) --shows sharing of lock

<X2(A) blocked> --so look for next non-T2 operation to do

<X1(A) blocked>-- DEADLOCK, abort T2 (say)

A2 <X1(A) unblocked>W1(A) C1

Concurrency: lost update anomaly

R1(A) R2(A)W2(A)C2W1(A)C1

- Run it under 2PL, but get X lock for R(A) W(A) sequence:

X1(A) R1(A)<X2(A)blocked> --so skip T2 ops…

W1(A)C1 <X2(A) unblocked> R2(A)W2(A)C2

Works better!

Aborting Transactions

- When *Ti* is aborted, all its actions have to be undone
  - if *Tj* reads an object last written by *Ti*, *Tj* must be aborted as well!
  - *cascading aborts* can be avoided by releasing locks only at commit
  - If *Ti* writes an object, *Tj* can read this only after *Ti* commits
- In Strict 2PL, cascading aborts are prevented
  - At the cost of decreased concurrency
  - No free lunch!
  - Increased parallelism leads to locking protocol complexity

Deadlock Detection

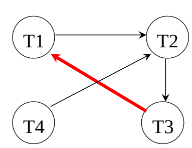
- Create a waits-for graph:
  - Nodes are transactions
  - Edge from *Ti* to *Tj* if *Ti* is waiting for *Tj* to release a lock

T1: S(A), R(A), S(B)

T2: X(B),W(B) X(C)

T3: S(C), R(C) X(A)

T4: X(B)



Dirty Reads

- Example: Reading Uncommitted Data (Dirty Reads)

T1:	R(A), W(A),	R(B), W(B)
T2:	R(A), W(A), R(B), W(B)	

$R_1(A)\ W_1(A)\ R_2(A)\ W_2(A)\ R_2(B)\ W_2(B)\ R_1(B)\ W_1(B)$

Note: commits are not involved in locating conflicts

T1→T2 T2→T1

- Again, this schedule can't happen under 2PL

Index Locking

- Needed for full serializability in face of inserts and deletes
- Example: assume index on the *rating* field using Alternative (2)
- Row locking is the industry standard now
- T1 should lock all the data entries with *rating* = 1
  - If there are no records with *rating* = 1, T1 must lock the entries adjacent to where data entry *would* be, if it existed!
  - e.g., lock the last entry with rating = 0 and beginning of rating=2
- If there is no suitable index, T1 must lock the table

Locking for B+ Trees (contd.)

- **Searches**
  - Higher levels only direct searches for leaf pages
- **Insertions**
  - Node on a path from root to modified leaf must be "locked" in X mode only if a split can propagate up to it
  - Similar point holds for deletions
- There are efficient locking protocols that keep the B-tree healthy under concurrent access, and support 2PL on rows, and provide index locking to avoid phantoms

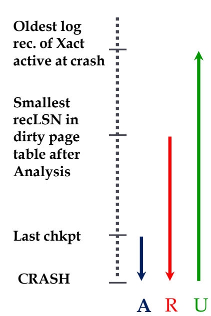
Isolation Levels in Practice

- Databases default to RC, read-committed, so many apps run that way, can have their read data changed, and phantoms
- Web apps (JEE, anyway) have a hard time overriding RC, so most are running at RC
- The 2PL locking scheme we studied was for RR, repeatable read: transaction takes long term read and write locks
- Long term = until commit of that transaction

Read Committed (RC) Isolation

- 2PL can be modified for RC: take long-term write locks but not long term read locks
- Reads are atomic as operations, but that's it
- Lost updates can happen in RC: system takes 2PC locks only for the write operations:
  R1(A)R2(A)W2(B)C2W1(B)C1
  R1(A)R2(A)X2(B)W2(B)C2X1(B)W1(B)C1 (RC isolation)
- Update statements are atomic, so that case of read-then-write is safe even at RC
- Update T set A = A + 100 (safe at RC isolation)
- Remember to use update when possible!

Crash Recovery: Big Picture



Start from a checkpoint (found in master record)

Three phases:

- **ANALYSIS**: Find which transactions committed or failed since checkpoint
- REDO ***all*** actions (repeat history)
- UNDO effects of failed transactions

Logging

- Essential function for recovery
  - Record REDO and UNDO information, for every update
  - Example: T1 updates A from 10 to 20
    - Undo: know how to change 20 back to 10 if find 20 in disk page and know T1 aborted
    - Redo: know how to change 10 to 20 if see 10 in the disk page and know T1 committed.
  - Updates include row inserts and deletes, but not emphasized here
  - Writes to log must be sequential, should be stored on a separate (mirrored) disk
  - Minimal information (summary of changes) written to log, since writing the log can be a performance problem

Write-Ahead Logging (WAL)

- The Write-Ahead Logging Protocol:
  1. Must force the log record for an update *before* the corresponding data page gets to disk
  2. Must write all log records for transaction *before commit returns*
  - Property 1 guarantees Atomicity
  - Property 2 guarantees Durability
- We focus on the ARIES algorithm
  - Algorithms for Recovery and Isolation Exploiting Semantics

The Analysis Phase

- Reconstruct state at checkpoint.
  - from end_checkpoint record
  - Fill in Transaction table, replace status = aborted/running with status U (needs undo)
  - Fill in DPT (dirty page table)
- Scan log forward from checkpoint, tracing transactions and dirty pages
- Finished: now all Transactions still marked U are "losers", DPT represents state at crash: which pages didn't get written to disk

The REDO Phase

- We *repeat history* to reconstruct state at crash:
  - Reapply *all* updates (even of aborted transactions), redo CLRs.
- Redo Update, basic case:
  - Read in page if not in buffer
  - Apply change to part of page (often a row)
  - Leave page in buffer, to be pushed out later (lazy again)
- Redo CLR:
  - Do same action as original UNDO:
  - Read in page if not in buffer, apply change, leave page in buffer
- But sometimes we don't need to do the redo, check conditions first…this is an optimization, skip for now.

The UNDO Phase, simple case, no rollbacks in progress at crash

In this case, losers have no CLRs in the old log

ToUndo = set of lastLSNs for "loser" transactions (ones active at crash)

Repeat:
  - Choose largest LSN among ToUndo
  - This LSN is an update. Undo the update, write a CLR, add prevLSN to ToUndo

Until ToUndo is empty
- **i.e. move backwards through update log records of all loser transactions, doing UNDOs**
- **End up with a bunch of CLRs in log to document what was done, so it doesn't have to be all repeated if this recovery crashes.**

Summary of Logging/Recovery

- Recovery Manager guarantees Atomicity & Durability.
- Use WAL to allow STEAL/NO-FORCE w/o sacrificing correctness.
- LSNs identify log records; linked into backwards chains per transaction (via prevLSN).
- pageLSN allows comparison of data page and log records.

Containers, e.g. Docker containers

Containers create a sandbox environment for a program to run in, isolating it from other programs and even the filesystem of the system it's running in, and its network.

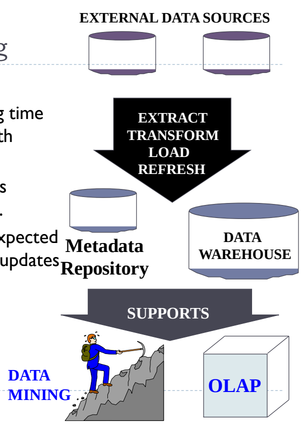
- It does use the OS kernel, originally only Linux.
- Needs to provide its own filesystem, since isolated from the shared one.
- Needs to have its own network, since isolated from the shared one.
- Usually a single process runs inside the container, but more are allowed.
- Note that an ordinary process isolates memory from other processes, but shares the filesystem, and network ports.

Docker Containers and Images

- Container: the executable object, like an executable file but holding a whole filesystem inside ready for the program.
- Docker Image: stored software in a format ready for use in a container. A container is built from one or more images. An image is something like a .class file, a template for building executables, not an executable itself.
  - Pre-built images are available from the Docker hub and elsewhere
  - You can build an image from your own software
  - Once you have an image, you can "run" it, passing various arguments. This will build and execute the container.
  - Once installed on a host system, Docker provides the a docker command, and a docker daemon (dockerd) to live on the host system and carry out the docker commands.
  - Docker commands: build, run, inspect, ps, kill, exec
  - Dockerfiles for building small Java programs (hw6)

Data Warehousing

- Integrated data spanning long time periods, often augmented with summary information.
- Several gigabytes to terabytes common, now petabytes too.
- Interactive response times expected for complex queries; ad-hoc updates uncommon.
- Read-mostly data



OLAP: Multidimensional data model

- Example: sales data in fact table
- Dimensions: Product, Location, Time
- A measure is a numeric value like sales we want to understand in terms of the dimensions. It's in the fact table.
- Example measure: dollar sales value "sales"
- Example data point (one row of fact/cube table):
  - Sales = 25 for pid=1, timeid=1, locid=1 is the sum of sales for that day, in that location, for that product
  - Pid=1: details in Product table
  - Locid = 1: details in Location table
- Note aggregation here for OLAP: sum of sales is most detailed data
  - Data warehouse fact table may have individual sales info: much bigger.
  - Need aggregation query to compute OLAP fact table from DW fact table.

OLAP Queries: cross-tabs

With relational DBs, we are used to tables with column names across the top, rows of data.

With OLAP, a spreadsheet-like representation is common,

Called a <u>cross-tabulation</u>:

- One dimension horizontally
- Another vertically
- Can "pivot" the table
- Can "drill down", "roll up"
- SQL queries for values

	WI	CA	Total
1995	63	81	144
1996	38	107	145
1997	75	35	110
Total	176	223	339

Topics FYI (not on final exam)

- Container tools other than docker itself
- Containerized mysql (too complex, not always a good idea anyway). Study containerized Java program examples.
- Materialized views
- NoSQL databases
- Data Lake idea (unstructured data, Hadoop)
- Big Data tools, like Apache Spark