CS 4604: Introduction to Database Management Systems

Final Review

Virginia Tech CS 4604 Sprint 2021

Instructor: Yinlin Chen

Today’s Topics

- Query Processing & Optimization
- FD’s & Normalization
- Security
- NoSQL
- Tx Management
- Logging and Recovery
- Data Warehousing
- Remember: Textbook exercise questions

Query Processing

- Estimating costs
  - What are you estimating? = #disk accesses
  - How to estimate?
    - Sorting
    - Different types of joins (NLJ, Block-NLJ, SMJ, HJ)
    - Don’t just memorize the formulae, understand how to use/apply them

Query Evaluation

- Types of joins: NL, index NL, sort-merge, hash
- Query cost – indexes and relations
- Query optimization –
  - good, not necessarily optimal

Query Optimization

- Algebraic manipulation
- Selectivity estimation
  - Many cases
  - How to use selectivities to get the output size

Query Optimization

```
SELECT travelers.name, cities.name
FROM travelers left outer join cities on city_id = dest_id
WHERE cities.name == ‘Berkeley’
ORDER BY cities.name;
```

- Many different orders to perform all these operations
- We use the System R optimizer (aka Selinger optimizer)
- Plan space: **only left-deep trees (important!), avoid cartesian products**
- Cost estimation: We’ll only use I/O cost for this class (exclude CPU)
- Search algorithm: dynamic programming

Selectivity Estimation

- To estimate the cost of a query, add up the estimated costs of each operator in the query
  - Need to know the size of the **intermediate relations** (generated from one operator and passed into another)
    - Need to know the **selectivity** of predicates - what % of tuples are selected by a predicate
- These are all estimates: if we don't know, we make up a value for it (selectivity = 1/10)
- System R assume uniform and indep. distribution of values

Selectivity Estimation - Equalities

Predicate	Selectivity	Assumption
c = v	1 / (number of distinct values of c in index)	We know |c|.
c = v	1 / 10	We don’t know |c|.
c1 = c2	1 / MAX(number of distinct values of c1, number of distinct values of c2)	We know |c1| and |c2|.
c1 = c2	1 / (number of distinct values of ci)	We know |ci| but not |other column|.
c1 = c2	1 / 10	We don't know |c1| or |c2|.

Included for completeness - don’t memorize, just put on your reference sheet

|column| = the number of distinct values for the column

Note: If you have an index on the column, you can assume you know |column|, max(c), and min(c)

Selectivity Estimation - Inequalities on Integers

Predicate	Selectivity	Assumption
c > v	(high key - v) / (high key - low key + 1)	We know max(c) and min(c).
c > v	1 / 10	We don't know max(c) and min(c).
c >= v	(high key - v) / (high key - low key + 1) + (1 / number of distinct values of c)	We know max(c) and min(c).
c >= v	1 / 10	We don't know max(c) and min(c).

Selectivity Estimation - Inequalities on Integers

Predicate	Selectivity	Assumption
c < v	(v - low key) / (high key - low key + 1)	We know max(c) and min(c).
c < v	1 / 10	We don’t know max(c) and min(c).
c <= v	(v - low key) / (high key - low key + 1) + (1 / number of distinct values of c)	We know max(c) and min(c).
c <= v	1 / 10	We don’t know max(c) and min(c).

Selectivity Estimation - Connectives

Predicate	Selectivity	Assumption
p1 AND p2	S(p1)S(p2)	Independent predicates
p1 OR p2	S(p1) + S(p2) - S(p1)S(p2)	
NOT p	1 - S(p)	

Query Optimization

- Pass 1: find minimum cost access method for each (relation, interesting order)
  - Index scan, full table scans
- Pass i (for $1 < i \leq n$): take in list of optimal plans for (i - 1 relations, interesting order) from Pass i-1, and compute minimum cost plan for (i relations, interesting orders) (every size i subset of the n relations)

Query Optimization

- For n relations joined, perform n passes
  - on the i-th path, output only the best plan for joining any i of the n relations
  - Also keep around plans that have higher cost but have an **interesting order**
- **This along with only considering left-deep plans forms the crux of most QO questions**

Query Optimization

- **Interesting orders** are orderings on intermediate relations that may help reduce the cost of later **joins**
  - ORDER BY attributes
  - GROUP BY attributes
  - *downstream* join attributes
    - For instance, sort merge join will produce a relation that can help with an ORDER BY clause

Security

- Concerned with secrecy, availability and integrity
- Granting privileges to users/roles
  - select, insert, delete, references
- SQL Injection & bind variables

Redundancy and Anomalies

- Potential problems with relations
    - **Redundancy**: repeated sets of dependent values
    - Anomalies that can result from redundancy
        - Ex. Rating determines Wage, so **Wage depends on Rating.**
        - **Update anomaly**: if we change wage for one person, we have to change it for everyone
        - **Insert anomaly**: if we want to insert a person with rating 10, we have to figure out the wage associated with it
        - **Delete anomaly**: if we delete all employees with rating 8, we no longer know the wage value corresponding to rating 8 (what if we add a rating 8 person later?)

FDs & Normalization

- Understand functional dependencies: A →B
- Understand normal forms and their definitions
  - Be able to tell what NF a given set of FDs are in and decompose into a higher level NF
- Attribute closures
- Minimal/Canonical cover

Functional Dependencies

- **functional dependency**: **X → Y** (X *determines* Y)
  - X, Y are *sets* of attributes
  - For every tuple in R, if attributes in X match, then attributes in Y must match
  - **Can *not* be inferred from the data: must come from outside of the data itself**
- **superkey**: X is a superkey of R if X → [all attributes of R]
- **candidate key**: the minimal superkey (smallest subset of attributes that is a superkey is itself; not necessarily smallest of all superkeys ever, but cannot be reduced further)

Functional Dependencies: Inference Rules

- Armstrong's Axioms
  - **Reflexivity**: If $Y \subseteq X$, then $X \rightarrow Y$
  - **Augmentation**: If $X \rightarrow Y$, then $XZ \rightarrow YZ$
    - $XZ \rightarrow YZ$ does NOT imply $X \rightarrow Y$
  - **Transitivity:** If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
- Union: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
- Decomposition: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
  - $XZ \rightarrow Y$ does NOT imply $X \rightarrow Y$ and $Z \rightarrow Y$

Functional Dependencies: Closure

- The **closure** of a set of FDs F is F^+, the set of all FDs implied by F
  - hard to find, exponential in # of attributes, so we use attribute closure instead
- The **attribute closure** of an attribute X given a set of FDs is X^+
  - set of all attributes A such that $X \rightarrow A$ is in F^+ (all attributes that can be determined by just X)
  - Algorithm:
    - Closure = X;
    - Repeat until there is no change
      - If there is an FD $U \rightarrow V$ in F s.t. $U \subseteq$ closure,
        - set closure = closure $\cup$ V

Normal Forms

Form	Requirement
1NF	Each attribute name must be unique. Each attribute value must be single. Each row must be unique.
2NF	1NF no non-key attribute is dependent on any proper subset of the key
3NF	2NF No transitive dependencies
BCNF	3NF All determinants are superkeys

Normalization: Boyce-Codd Normal Form (BCNF)

- R is in BCNF if:
  - for every FD $X \rightarrow A$ that holds over R, either $A \subseteq X$ or X is a superkey of R
  - $A \subseteq X$ means the FD is trivial.
- Redundancies removed in BCNF
  - every field of every tuple contains some information that *cannot* be inferred from the FDs
- Simpler to deal with than other normal forms

Normalization: BCNF Decomposition

- **We can decompose a relation R that is not in BCNF into multiple relations that are in BCNF**
- Algorithm:
  - Find FD $X \rightarrow Y$ that violates BCNF
  - Decompose R into $(R - X^+) \cup X$ and X^+
  - Repeat until no FDs violate BCNF
- Heuristic: for the violating FD, make Y as big as possible (i.e. replace with X+; helps avoid unnecessarily fine-grained decomposition)
- What relations you get depends on what order you go in
- two attribute relations are always in BCNF

Normalization: Lossiness

- **Lossiness**: we may not be able to reconstruct the original relation (doesn't actually lose data, it generates bad data)
    - BCNF is lossless (can still reconstruct original relation)
    - Decompose R into X and Y. Decomposition is lossless iff F^+ contains:
        - X INTERSECT Y → X or
        - X INTERSECT Y → Y
    - Alternatively, can attempt natural join between the two and manually check if the reconstruction works

Normalization: Dependency Preservation

- **Dependency preservation**: if we can enforce F+ individually on each table; this in turn enforces the FDs on the entire database
  - BCNF is not necessarily dependency preserving (enforce FDs on each decomposed relation independently)
  - Formalism: dependency preserving iff $F_x^+ \cup F_y^+ = F^+$ where F_x are the FDs we can enforce just in relation X
    - For example: imagine we decomposed R = ABC into X=AB, Y=BC. If we have an FD A→C this is not dependency preserving because we can't enforce the dependency on either relation

Decomposition

- Given Relation R(A,B,C,D,E) and functional dependencies
  F{ A->BCD, C->E}, decompose until in 3NF.
- Answer:
  - R1(A,B,C,D)
  - R2(C,E)

- (Also review 3NF Synthesis)

NoSQL Data Model: Key-Value Stores

- Data Model: (key, value) pairs
  - Key: typically a string/integer to uniquely identify the record
  - Value: can be anything (even a complex object)
- One of the most flexible data models (least-structured)
  - Data can be represented in many ways as (key, value) pairs
  - Best to choose the data model based on the desired use case
- Operations
  - get(key) and put(key, value)
  - Operations on value are not supported due to the flexible data model
- Distribution
  - Without replication: stored on one server
  - With replication: stored on multiple machines (updates need to be made on all servers)

NoSQL Data Model: Document Stores

- Document: semi-structured data format (like JSON)
- It can be beneficial to provide some structure to the “value” of (key, value) pairs
    - In this data model, the values are called documents
- One of the most structured data models

JSON

- Supported types:
  - **Object:** collection of (key, value) pairs
    - Keys: strings
    - Values: object, array, or atomic (any JSON type)
    - Denoted with “{” and “}”
  - **Array:** ordered list of values
    - Denoted with “[” and “]”
  - **Atomic:** a number, string, boolean, or null
- Can be interpreted as a tree due to its inherent nested structure
- **Self-describing:** each document can have its own schema

JSON vs. Relational Model

	JSON	Relational
Flexibility	Very flexible, can represent complex structures and nested data	Less flexible
Schema Enforcement	Self-describing; Each document can have unique structure	Schema is fixed
Representation	Text-based (easily parsed and manipulated by many languages)	Binary representation (designed for efficient storage and retrieval from disk)
	"Enforcing schema on read"	"Enforcing schema on write"

MQL

- Operates on collections
- Dot notation can be used to index into nested documents or arrays
  - Ex: “student_information.name” → name field within the student document
  - Must be used with quotes
- $ notation indicate the special keywords
  - Ex: $gt, $lte, $add
  - Used in the “field” part of “field:value” expression
- 3 main types of queries
  - **Retrieval**: essentially SELECT-WHERE-ORDER BY-LIMIT queries
  - **Aggregation**: in MQL this refers to a general pipeline of operations
  - **Updates**

Tx Management

- ACID – Atomic, Consistent, Isolated, Durable
- Problems with concurrency and Serializability concept
- Conflict-Serializability, how to detect
- Definitions:
  - Transaction
  - Schedule – serial, serializable
- Strict 2PL
- Transaction Logs, Aries Recovery Algorithm

Why Transactions?

- Usually have multiple users accessing the database concurrently
- Can cause these problems:
  - **Inconsistent Reads:** A user reads only part of what was updated (one user updates two tables, another user reads old version of one table and new version of the other table)
  - **Lost Update:** Two users try to update the same record so one of the updates gets lost
  - **Dirty Reads:** One user reads an update that was never committed (usually due to reading after abort but before rollback)

Transactions

- A sequence of multiple actions that should be executed as a single, logical, atomic unit. Abbreviated as “Xact”. Enforces these properties:
- **Atomicity**: A transaction ends in two ways: it either commits or aborts; either all actions in the Xact happen, or none happen.
- **Consistency**: If the DB starts out consistent (adhering to all rules), it ends up consistent at the end of the Xact.
- **Isolation**: Execution of each Xact is isolated from that of others; DBMS will ensure that each Xact executes as if it ran by itself, even with interleaved actions
- **Durability**: If a Xact commits, its effects persist; the effects of a committed Xact must survive failures.

Equivalence and Serializability

- Easiest way to enforce Isolation is to run transactions one at a time (a serial schedule), but this is inefficient
- Two schedules are **equivalent** if
    - They involve the same transactions
    - Each transaction has its operations in the same order
    - The final state after all the transactions is the same
- If a schedule is equivalent to a serial schedule, it is **serializable**
- Some schedules that interleave transaction actions are serializable, but it's hard to check.

Conflict Serializability

- Two operations in a schedule **conflict** if:
  - at least one operation is a write
  - they are on *different* transactions
  - they work on the *same* resource
- Conflicts are basically just pairs of operations that we need to be careful about

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

T2:   R(B) W(B)

Conflict Serializability

- If two schedules **order their conflicting pairs the same way**, they are **conflict equivalent** (and thus equivalent).
- If a schedule is **conflict equivalent** to a serial schedule, it is **conflict serializable**.
- Conflict serializability is a more **strict** condition than serializability (all conflict serializable schedules are serializable, but not all serializable schedules are conflict serializable). However, it's a lot easier to check.
- View equivalence/serializability falls in between them in terms of difficulty, but it's NP hard to check for.
    - Essentially, check same conditions as conflict serializability, except you can ignore blind writes (two writes without an interleaving read)

Conflict Serializability

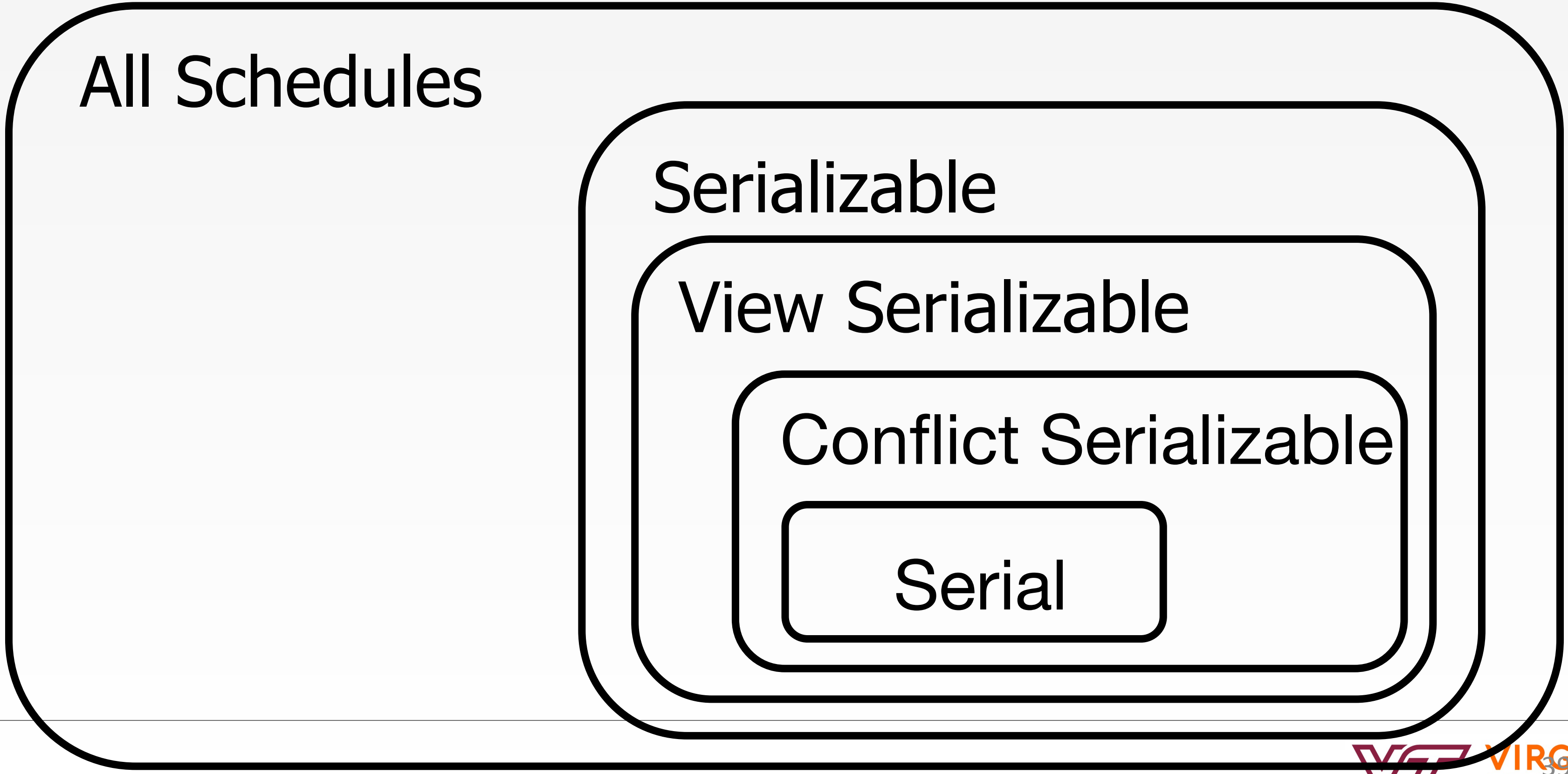
- How do we check for conflict equivalence/serializability?
  - We build a **dependency graph** (**precedence graph**)
    - If an operation in T_i conflicts with an operation in T_j, and the operation in T_i comes first, add an edge from T_i to T_j
    - Cycle → not conflict serializable

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

T2: R(B) W(B)

[T1] --------------> [T2]

Types of Serializability

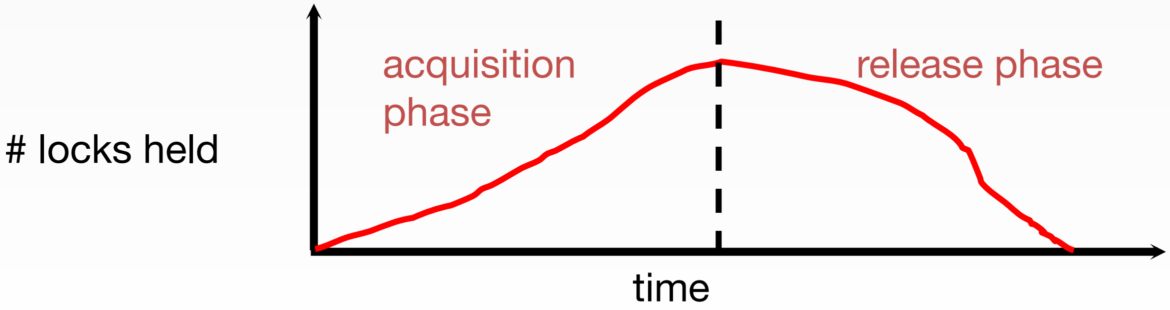


Locks

- Make sure that no other transaction is modifying the resource while you are using that resource
- Lock types: for a given resource A,
  - **S (Shared)** can read A and all descendants of A.
  - **X (Exclusive)** can read and write A and all descendants of A.

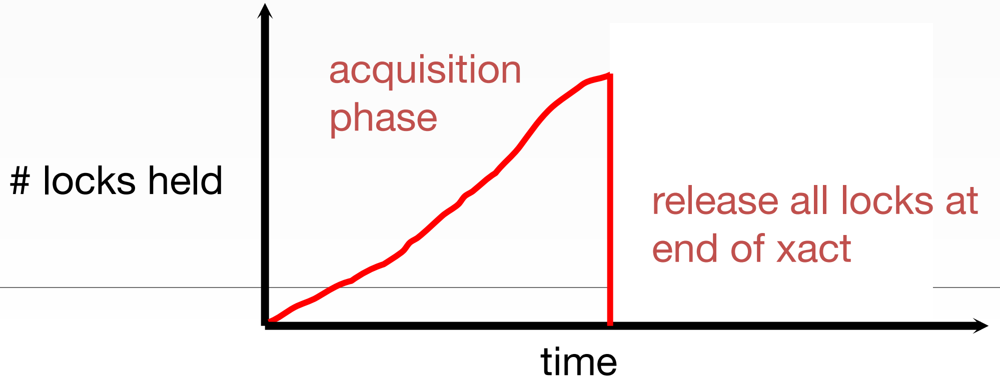
2-Phase Locking (2PL)

- One way to enforce conflict serializability
- In **2-phase locking**,
    - a transaction may not acquire a lock after it has released any lock
    - two “phases”
        - from start to until a lock is released, the transaction is just acquiring locks
        - then until the end of the transaction, it is just releasing locks



Strict 2-Phase Locking (Strict 2PL)

- The problem is that 2PL lets another transaction read new values before the transaction commits (since locks can be released long before commit)
- **Strict 2PL** avoids cascading aborts (and guarantees conflict serializability and recoverability)
  - Same as 2PL, except only allow releasing locks at end of transaction



Deadlock Detection

- We draw out a **"waits-for" graph**
  - One node for each transaction
  - If T_j *holds* a lock that conflicts with the lock that T_i wants, we add an edge from T_i to T_j
  - **A cycle indicates a deadlock** (between the transactions in the cycle) - we can abort one to end the deadlock

Deadlock Avoidance

- Typically assign priority based on start time (starting earlier means higher priority), but can use other methods (will specify on exams)
- Two approaches
  - **wait-die**: if a transaction T_i wants lock but T_j has conflicting lock
    - if T_i is higher priority, it waits for T_j to release conflicting lock
    - if T_i is lower priority, it aborts
    - transactions can only wait on lower priority transactions → cannot have deadlock (lowest priority transactions cannot wait)
  - **wound-wait**: if a transaction T_i wants lock but T_j has conflicting lock
    - if T_i is higher priority, it causes T_j to abort (“wound”)
    - if T_i is lower priority, it waits for T_j to finish
    - transactions can only wait on higher priority transactions → cannot have deadlock (highest priority transactions can’t wait)

Recovery Policies

- **Steal/No Force**
  - **Steal** - Uncommitted transactions can overwrite the most recent committed value of an object on disk
    - Necessitates UNDO for Atomicity (all or none of Xact's operations persist)
  - **No Force -** *Don't have to* write all pages modified by a transaction from the buffer cache to disk before committing the transaction
    - Necessitates REDO for Durability (not losing results of committed Xacts)
  - Harder to enforce atomicity and durability, but gives best performance
- **No Steal** locks buffer pages from optimal use, but keeps uncommitted changes away from disk (easy atomicity)
- **Force** necessitates extra writes on commit, but everything is guaranteed to be there (easy durability)

Write-Ahead Logging

1. **Log records must be on disk before the data page gets written to disk.**
   - How we achieve atomicity
   - Can’t undo an operation if data page written before log - don’t know operation happened
2. **All log records must be written to disk when a transaction commits.**
   - How we achieve durability
   - We know what operations to redo in case of crash

Undo Logging

- Write log records to ensure **atomicity** after a system crash:
  - **<START T>**: transaction T has begun
  - **<COMMIT T>**: T has committed
  - **<ABORT T>**: T has aborted
  - **<T,X,v>**: T has updated element X, and its **old** value was v
- If T commits, then **FLUSH(X)** must be written to disk before **<COMMIT T>**
  - **Force** – we can UNDO any modifications if a Xact crashes before COMMIT
- If T modifies X, then **<T,X,v>** log entry must be written to disk before **FLUSH(X)**
  - **Steal** – we can UNDO any modifications if a Xact crashes before FLUSH

Redo Logging

- Write log records to ensure **durability** after a system crash:
  - **<START T>**: transaction T has begun
  - **<COMMIT T>**: T has committed
  - **<ABORT T>**: T has aborted
  - <T,X,v>: T has updated element X, and its **new** value was v
- If T modifies X, then both **<T,X,v>** and **<COMMIT T>** must be written to disk before **FLUSH(X)**
  - **No-Steal, No-Force** – we can REDO any modifications if a Xact crashes before FLUSH

Undo/Redo Logging Summary

- Undo logging:
  - Uses Steal/Force policies
  - Undoes all updates for **running** transactions
- Redo logging:
  - Uses No Steal/No Force policies
  - Redoes all updates for **committed** transactions

Aries Recovery - LSNs

- **LSN (Log Sequence Number)**: stored in each log record. Unique, increasing, ordered identifier for each log record
- **flushedLSN**: stored in memory, keeps track of the most recent log record written to disk
- **pageLSN**: LSN of the last operation to update the page (in memory page may have a different pageLSN than the on disk page)
- **prevLSN**: stored in each log record, the LSN of the previous record written by the current record's transaction
- **lastLSN**: stored in the Xact Table, the LSN of the most recent log record written by the transaction
- **recLSN**: stored in the DPT, the log record that first dirtied the page since the last checkpoint
- **undoNextLSN**: stored in CLR records, the LSN of the next operation we need to undo for the current record's transaction

Recovery Structures

- **Transaction Table** - stores information on active transactions. Fields include
  - Xid (Transaction ID)
  - Status (Running, Committing, Aborting)
  - lastLSN
- **Dirty Page Table (DPT)** - tracks dirty pages (pages whose changes have not been flushed to disk)
  - pageID
  - recLSN

Record Types

- Records have LSN, common fields include xid (transaction ID), pageID (for modified page), type
- **UPDATE** - write operation (SQL insert/update/delete). Also includes fields for offset (where data change started), length (how much data was changed), old_data (old version of changed data - used for undos), new_data (updated version of data - used for redos)
- **COMMIT** - Xact is beginning committing process (ARIES: flush log up to and including COMMIT record)
- **ABORT** - Xact is beginning aborting process (ARIES: begin writing CLRs for undone UPDATEs)
  - **Compensation Log Record (CLR)** - indicates a given UPDATE has been undone
- **END** - Xact is finished (as in, finished committing or aborting)

Record Types (cont.)

- Checkpoint Records
  - Useful for ARIES analysis so we don't start from very beginning of log
  - Checkpoint serves as snapshot of Xact Table/DPT
  - Fuzzy checkpoints - Xacts operating during checkpoint; Xact
  - **BEGIN CHECKPOINT** - checkpoint start, earliest point Xact Table/DPT could represent
  - **END CHECKPOINT** - checkpoint end, holds Xact Table/DPT snapshot
- **Master Record** - stores location of most recent checkpoint for recovery purposes, usually LSN 0

ATT: Xact Table

lastLSN

status

Dirty Page Table

recLSN

flushedLSN

Buffer pool

Log tail

LogRecords

LSN

prevLSN

XID

type

pageID

length

offset

before-image

after-image

Data pages

each

with a

pageLSN

Master record

ARIES: Overview

ARIES: Analysis (Part 1)

- Reconstructing Xact Table and DPT
- Need to know which transactions started/committed/aborted, which pages dirtied
- Start from **begin** checkpoint log record (or start of log), go until end of log
- On any record that is not an END record:
  - Add the Xact to the Xact Table if not in table
  - Set the lastLSN of the transaction to the current operation's LSN
  - If the record is a COMMIT or an ABORT record, change the status of Xact to Committing/Aborting
- If the record is an UPDATE record and the page being updated is not in the DPT, add the page to the DPT and set recLSN equal to the LSN
- If the record is an END record, remove the transaction from the Xact table.

ARIES: Analysis (Part 2)

- After going through the log, clean up the Xact Table
- For each Xact in the Xact Table:
  - Write END records for committing Xacts. Because they're committing, they must be finished - preserve durability
  - For running Xacts, change status to aborting and write ABORT record - preserve atomicity since not finished

ARIES: Redo

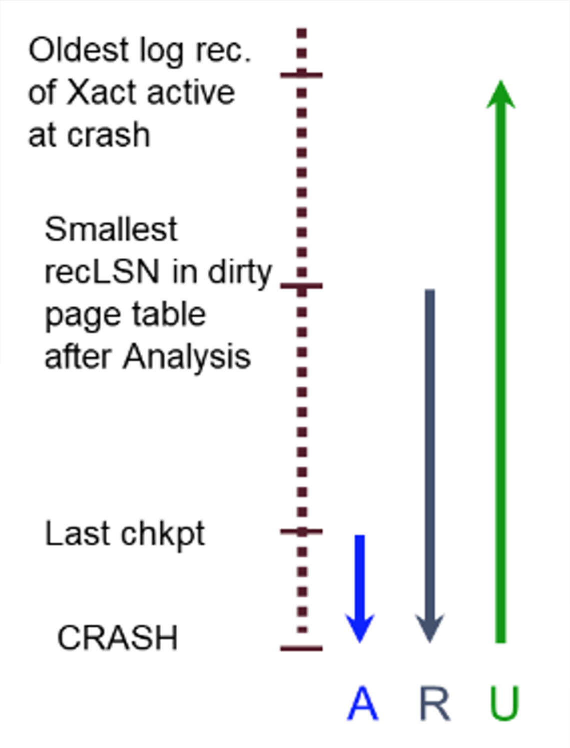
- Redo *updates and CLRs* from the earliest recLSN in the DPT to get back unflushed changes from before crash, unless:
  - page not in DPT
    - page on disk must be up to date, since we have no changes!
  - recLSN of page > LSN
    - no need to undo here: recLSN of page is *first* record that dirtied page, so this change must have been flushed
  - pageLSN (disk) >= LSN
    - page LSN for disk (LSN of last record with change written to disk) is the authoritative source for determining which changes have been applied in disk
  - Redo with after-image (redo state), update pageLSNs as you go

ARIES: Undo

- Undo each Xact in the Xact Table
  - Only UNDO updates (ignore CLRs)
- Start at end of log and work backwards to the beginning
- For every UPDATE the undo phase undoes, write a corresponding CLR to the log.
  - undoNextLSN stores the LSN of the next operation to be undone for that transaction (the prevLSN of the operation that you are undoing).
- Once you have undone all the operations for a transaction, write the END record for that transaction to the log.

ARIES: Overall

- Why does redo happen before undo?
  - If failure happens during redo or undo, next recovery can pick up what previous recovery has left and continue
    - E.g. Crash while writing CLRs in UNDO, we have to redo them
- When are transactions removed from the xact table?
  - END log record
- When is a page removed from the DPT?
  - When that page flushed to disk (pages aren't necessarily flushed to disk on commit - no force)



Logging and Recovery

- Make sure you know *exactly* how recovery takes place, and what is logged
  - Practice, practice
  - Check out problems in lectures, practice problems and hws
  - Be comfortable with small conceptual questions

Workloads

- **Online Transaction Processing** (OLTP)
  - Typically simple lookups with few joins or aggregations
  - Characterized by high number of transactions by a high number of users
  - Modern “Web 2.0” applications with lots of user-generated content and user interactions have OLTP workloads
- **Online Analytical Processing** (OLAP)
  - Read-only queries and typically involve many joins and aggregations
  - Used to support data-driven decision making
- OLTP and OLAP are served by separate databases
  - **Extract-transform-load** (ETL) migrates data from OLTP systems to OLAP systems

OLAP

- Prioritizes in summarizing and extracting insights from petabytes of data
- Performed on a **separate** data warehouse separate from OLTP's critical path
    - Data warehouse is periodically updated with OLTP using ETL (consolidate, clean, canonicalize data)
    - Ex: run a chron job to update the data warehouse at the end of each day

Next Week

Project Presentation

Good Luck!