Announcements

- HW wrap-up:
  - HW#8 in flight!
  - Due tomorrow at 5PM!
  - Remember: NoSQL has NoLateDay!

- Endterm exam:
  - In class on Friday, June 7, 5-5:50 PM
    - Cheat sheet allowed, as per usual
    - Non-cumulative (see Wiki syllabus for official scope)
  - Sample exam available (interpret it appropriately)
  - Will include indexing, physical design, NoSQL, JSON, and even transactions
**Midterm #2 Stats**

Review Grades for Midterm 2-A

- **Minimum:** 68.0
- **Maximum:** 67.63

Review Grades for Midterm 2-B

- **Minimum:** 28.0
- **Maximum:** 92.0

**Midterm #2 Deeper Dive**

Comparison of Version A and Version B

- **Try To SQLize That:** Version A: 54.8, Version B: 61.2
- **I'm Queryous:** Version A: 62.0, Version B: 62.0
- **Query This:** Version A: 65.6, Version B: 64.0
- **Never Ending SQL:** Version A: 87.6, Version B: 84.0
Transactions

- Concurrent execution of user programs is essential for good DBMS performance (and wait times).
  - Disk I/O’s are slow, so DBMS’s keep the CPU cores busy by running multiple concurrent requests.
- A program may perform many operations on data from the DB, but the DBMS only cares about what’s being read (R) and written (W) from/to the DB.
- A transaction is the DBMS’s view of a user program:
  - It is seen as a sequence of database R’s and W’s.
  - The targets of the R’s and W’s are records (or pages).

The ACID Properties

- **Atomicity**: Each transaction is all or nothing.
  - No worries about partial effects (if failures) and cleanup!
- **Consistency**: Each transaction moves the database from one consistent state to another one.
  - This is largely the application builder’s responsibility...
- **Isolation**: Each transaction can be written as if it’s the only transaction in existence.
  - No concurrency worries while building applications!
- **Durability**: Once a transaction has committed, its effects will not be lost.
  - Application code doesn’t have to worry about data loss!
Concurrency in a DBMS

- Users run transactions and can think of each one as executing all by itself.
  - Concurrency is handled by the DBMS, which allows the actions (R’s & W’s) of various transactions to interleave.
  - Each transaction must leave the DB in a consistent state if it was consistent when the transaction started.
    - The DBMS may enforce some ICs, depending on the constraints declared in CREATE TABLE statements. (CHECK, PK, FK, ...)
    - But the DBMS does not understand the semantics of the data! (It doesn’t know how interest on a bank account is computed.)
- **Issues**: Effects of **interleaving** and of **crashes**.

Atomicity of Transactions

- A transaction may commit after completing all of its actions, or it might abort (or might be aborted) after executing some of its actions.
  - Could violate a constraint, encounter some other error, be caught in a crash, or be picked to resolve a deadlock.
- The DBMS guarantees that transactions are **atomic**. A user can think of a Xact as doing all of its actions, in one step, or executing none of its actions.
  - The DBMS logs all actions so that it can undo the actions of any aborted transactions.
Example

- Consider two transactions (Xacts):
  
  \[
  \begin{align*}
  T1: & \quad \text{BEGIN} \quad A &= A + 100, \quad B = B - 100 \quad \text{END} \\
  T2: & \quad \text{BEGIN} \quad A &= 1.06 \times A, \quad B = 1.06 \times B \quad \text{END}
  \end{align*}
  \]

- E.g., T1 is transferring $100 from bank account A to account B, while T2 is crediting both with 6% interest.
- No guarantee that T1 will execute before T2, or vice-versa, if both arrive together. The net effect must be equivalent to running them serially in some (either!) order.

A Quick Aside on “A” & “B”

- What are these two transactions, really?
  
  \[
  \begin{align*}
  \text{T1:} & \quad \text{START TRANSACTION}; \quad \text{-- needed to couple the statements} \\
  & \quad \text{UPDATE Acct SET bal = bal + 100 WHERE acct_no = 101;} \\
  & \quad \text{UPDATE Acct SET bal = bal - 100 WHERE acct_no = 201;} \\
  & \quad \text{COMMIT;} \\
  \text{T2:} & \quad \text{START TRANSACTION}; \quad \text{-- not needed if just one statement} \\
  & \quad \text{UPDATE Acct SET bal = bal * 1.06 WHERE acct_type = ’SV’;} \\
  & \quad \text{COMMIT;}
  \end{align*}
  \]

- Again, T1 is transferring $100 from account B (201) to account A (101). T2 is giving all accounts their 6% interest payment.
Example (Cont’d.)

- Consider a possible interleaving (schedule):

<table>
<thead>
<tr>
<th>T1:</th>
<th>A = A + 100,</th>
<th>B = B - 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>A = 1.06 * A,</td>
<td>B = 1.06 * B</td>
</tr>
</tbody>
</table>

- This is OK. But what happens if:

<table>
<thead>
<tr>
<th>T1:</th>
<th>A = A + 100,</th>
<th>B = B - 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>A = 1.06 * A, B = 1.06 * B</td>
<td>Too much interest!</td>
</tr>
</tbody>
</table>

- The DBMSs view of the second schedule:

<table>
<thead>
<tr>
<th>T1:</th>
<th>R(A), W(A), R(B), W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>R(A), W(A), R(B), W(B)</td>
</tr>
</tbody>
</table>

Scheduling Transactions (Defn’s.)

- **Serial schedule**: Any schedule that does not interleave the actions of different transactions.
- **Equivalent schedules**: If for any database state, the effect (on the DB) of executing the first schedule is identical to the effect of the second schedule.

- **Serializable schedule**: A schedule that is equivalent to *some* (any!) serial execution of the transactions.

  ➔ If each transaction preserves consistency, then *every* serializable schedule preserves consistency!
Anomalies with Interleaved Execution

- Reading Uncommitted Data ("dirty reads"): 

  T3: \(R(A), W(A), R(B), W(B), \text{Abort}\)
  T4: \(R(A), W(A), C\)

- Unrepeateable Reads:

  T5: \(R(A), R(A), W(A), C\)
  T6: \(R(A), W(A), C\)

Anomalies (Continued)

- Overwriting Uncommitted Data:

  T7: \(W(A), W(B), C\)
  T8: \(W(A), W(B), C\)

  (Results are a “must have been concurrent!” mix of T7’s & T8’s writes – B from T7, and A from T8, yet both transactions wrote both A and B.)
Lock-Based Concurrency Control

- **Strict Two-phase Locking (2PL) Protocol:**
  - Each Xact acquires an *S (shared)* lock on an object before reading it, and an *X (exclusive)* lock on it before writing.
  - All locks held by a transaction are released only when the transaction completes.
  - Note: If a Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object – they must wait.

- Strict 2PL allows only serializable schedules.
  - And additionally, it simplifies transaction aborts!

2PL Prevents the Anomalies

- Reading Uncommitted Data (WR Conflicts, a.k.a. “dirty reads”):

  | T3: R(A), W(A), R(B), W(B), Abort |
  | T4: R(A), W(A), X |
  | T5: R(A), R(A), W(A), C |
  | T6: R(A), W(A), C |

- Unrepeatable Reads (RW Conflicts):

  | T3: R(A), W(A), R(B), W(B), Abort |
  | T4: R(A), W(A), X |
  | T5: R(A), R(A), W(A), C |
  | T6: R(A), W(A), C |
2PL & Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

  T7: \(W(A), W(B)\)
  T8: \(W(A), W(B), C\)

  (Now results will no longer be a “must have been concurrent!” intermingling of T1’s & T2’s writes…)

Aborting a Transaction

- If transaction \(Ti\) aborts, all its actions must be undone.
  - And, if some \(Tj\) already read a value last written by \(Ti\), \(Tj\) must also be aborted! (“If I tell you, I’ll have to kill you…” 😇)
- Most systems avoid such *cascading aborts* by releasing a transaction’s locks only at *commit time*.
  - If \(Ti\) writes an object, \(Tj\) can read it only after \(Ti\) commits.
- In order to *undo* the actions of an aborted transaction, the DBMS keeps a *log* where every write is recorded.
  - Also used to recover from system crashes: active Xacts at crash time are aborted when the DBMS comes back up.
The Transaction Log

- The following actions are recorded in the log:
  - *Ti writes an object:* record its old and new values.
    - Log record must go to disk before the changed page – hence the name write-ahead logging (WAL).
  - *Ti commits/aborts:* write a log record noting the outcome.
- All log related activities (and all concurrency-related activities, like locking) are *transparently* taken care of by the DBMS.

Reminder: Disks and Files

- DBMSs store all information on disk.
- This has major implications for DBMS design!
  - **READ:** transfer data from disk to main memory (RAM).
  - **WRITE:** transfer data from RAM to disk.
  - Both are high-cost operations, relative to in-memory operations, so must be considered carefully!
Recovering From a Crash

- A three-phase recovery algorithm (*Aries*):
  - **Analysis**: Scan log (starting from most recent checkpoint) to identify the Xacts that were active, and the pages that were “dirty” in the buffer pool, when the system crashed.
  - **Redo**: Redo any updates to dirty pages to ensure that all logged updates were carried out and made it to disk. (Establishes the state from which to recover.)
  - **Undo**: Undo the writes of all Xacts that were active at the crash (restoring the *before value* of each update from its log record), working backwards through the log, to abort any partially-completed transactions.

Support for Transactions in SQL-92

- A transaction is *automatically* started whenever a statement accesses or modifies the database
  - SELECT, UPDATE, CREATE TABLE, INSERT, ...
  - Multi-statement transactions also supported
- A transaction can be terminated by
  - A COMMIT statement
  - A ROLLBACK statement (SQL-speak for *abort*)
- Each transaction runs under a combination of an access mode and an isolation level
Transactions in SQL-92 (Cont’d.)

- Access mode – controls what the transaction can potentially do to the database:
  - READ ONLY: not permitted to modify the DB
  - READ WRITE (default): allowed to modify the DB

- Isolation level – controls the transaction’s exposure to other (concurrent) transactions:
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE

Which Isolation Level is for Me?

- An application-“controllable” tradeoff:
  - Consistency vs. performance (concurrency)
  - Warning: It will affect your programming model!

- Things to watch out for:
  - Default consistency level is DBMS engine-specific
  - Some engines may not support all levels
  - Default consistency level often not SERIALIZABLE

- You may also hear about “snapshot isolation”
  - DBMS keeps multiple versions of data
  - Transactions see versions as of when they started
Remember the **ACID** Properties!

- **Atomicity**: Each transaction is *all or nothing*.
  - No worries about partial effects (if failures) and cleanup.
- **Consistency**: Each transaction moves the database from one *consistent state* to another one.
  - This is largely the application builder’s responsibility.
- **Isolation**: Each transaction can be written as if it’s the *only transaction* in existence (*if so desired*).
  - Minimize concurrency worries when building applications.
- **Durability**: Once a transaction has committed, its *effects will not be lost*.
  - Application code needn’t worry about data loss.

---

**A Few Quick NoSQL Xact Notes**

- For transactions, NoSQL systems tend to be limited to *record-level* transactions (in order to *scale* on a cluster)
- As a result, one sometimes consider an application’s transactional needs when picking a schema (deciding what to "nest") for it
CS122a has just given you an “outside” view of database management systems.

CS122b is available to give you a “programmer’s” view – with an emphasis on data-centric web applications.

CS122c (a.k.a. CS222 lite) is available to give you an “insider’s” (engine developer’s) view of database systems.

CS223 is available for learning all about transactions.

CS190 (when offered – like Beyond SQL Data Management next Winter quarter: NoSQL, Graph DBs, Spark, …)

CS199 (independent project work) is also a possible avenue for gaining further experience.