Announcements

- HW and exam info:
  - HW#8 now in flight! (Due tomorrow at 5PM!)
  - Endterm is in class this Friday, June 8, 5-5:50 PM
    - Cheat sheet allowed, as per usual
    - Non-cumulative (see Wiki syllabus for official scope)
    - Sample exam available (but interpret it appropriately)
    - Will include NoSQL, JSON, and even transactions!
- This week’s material:
  - Today: Transactions, cont.
**Lock-Based Concurrency Control**

- *Strict Two-phase Locking (Strict 2PL) Protocol:*
  - Each Xact must get an *S* (shared) lock on an object before reading, and an *X* (exclusive) lock on it before writing.
  - All locks held by a transaction are released only when the transaction completes.
    - *(Non-strict) 2PL Variant:* Release locks anytime, but do not acquire any new locks after releasing *any* lock.
    - *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!
  - *(Non-strict) 2PL also allows only serializable schedules, but needs more complex abort processing (as you’ll see).*

**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: $R(A), W(A),$ $R(A), W(A),$ *Abort*  
  - T5: $R(A),$ $W(A),$ $R(A), W(A),$ *Abort*  

- Unrepeatable Reads (RW Conflicts):
  - T5: $R(A),$ $W(A),$ $R(A),$ $W(A),$ *Abort*  
  - T6: $R(A),$ $W(A),$ $R(A),$ *Abort*
2PL & Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

  T7: W(A), W(B), C
  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 the object’s old and new values.
    - Log record must go to disk *before* the changed page does – hence the name write-ahead logging (or WAL).
  - *Ti commits/aborts*: write a log record noting this outcome.

- Log records are back-chained together by Xact id, so it’s easy to undo a specific Xact if need be.
- Log is often *duplexed* and *archived* on stable storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, handling deadlocks etc.) 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 forward (from most recent *checkpoint*) to identify all Xacts that were active, and also all dirty pages in the buffer pool, as of the time of the crash.
  - **Redo**: Redo all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk. (*Establishes the state to recover from.*)
  - **Undo**: Undo writes of all Xacts that were active at the crash (restore the *before value* of each update, which is in the log record for the update), working backwards through the log. (*Note*: Care must be taken to account for the possibility of a subsequent crash *during* this recovery process...!)

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
  - An isolation level
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

Increasing isolation

Hey, what could possibly go wrong...? (©)
- Dirty Read (WR conflict): a transaction could read an object written by an uncommitted transaction.
- Unrepeatable Read (RW conflict): a transaction could overwrite an existing object that was read by an uncommitted transaction.
- Phantom: a transaction reads a collection of objects twice, and sees some different (newly inserted) objects on the second time through.
Isolation Levels

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Which Isolation Level is for Me?

- An application-“controllable” tradeoff:
  - Consistency vs. performance (concurrency)
  - Note that this 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 data versions as of their start timestamp
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 doesn’t have to worry about data loss.

And a Few Closing **NoSQL** Words

- For transactions, NoSQL systems tend to be limited to *record-level* transactions (in order to *scale* on a cluster)
- As a result, one might consider transactional desires when designing a schema (e.g., what to “nest”)
- In general, *my “rules”* for NoSQL schema design are:
  - Start with an E-R model – you’re still DB professionals!!
  - Strong entities: keep as “top-level objects” in most cases
  - Weak entities: probably nest them within their parent object
  - Relationships: all relational options still available, *plus* you could have a *set* of keys if you wanted (e.g., dept.emps)
  - 1NF: composite and set-valued attributes are fair game!
  - Size: keep object sizes reasonable and also “*fixed*”(-ish)
"But Wait!.... I Need More...!!!"

- **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** (if/when offered) and **CS199** (independent project work) are also possible avenues for gaining further info.