Lecture 8 - Isolation Levels Revisited

- Two papers:
  - Adya's Generalized Isolation Level Definitions
  - A critique of SQL Isolation Levels
- Goal: begin exploration of weaker isolation levels beyond the original degrees of consistency paper

To discuss in class:

- Q: How do folks feel about the project?
- Q: How are the readings treating you?
- Announcement: special seminar
- Q: How do we make up the lecture?

Prehistory and a Spooky Problem

- Original Gray proposal:
  - degree 0 - short duration write locks
  - degree 1 - long duration write locks
  - degree 2 - long duration write locks, short duration read locks
  - degree 3 = serializable - long duration write locks, long duration read locks
- Unfortunately, degree 3 doesn't actually guarantee serializability, except under a simplified model of locking of individual items
  - as opposed to for SQL
- Phantoms come from predicates/ranges, not individual items:
  - Suppose T1 reads all rows where Salary > 100K
  - Then T2 inserts a new row with Salary = 120K and commits
  - If T1 repeats the predicate, it sees an extra row: a *phantom*
- The paper that formalizes this came right after the original degrees of consistency paper: The Notions of Consistency and Predicate Locking
  - Basically, degree 3 is insufficient for serializability
  - Instead, proves that predicate locking + 2PL => serializability
  - What is predicate locking: basically "lock" corresponding to a predicate, e.g., Salary > 100K
    - T2's insertion here would not be allowed as it conflicts with the predicate
- Q: Easy enough, but how to enforce? Why is locking predicates hard?
  - Really difficult in practice:
    - General lock manager for predicates is not practical
    - How does know if Salary > 100K overlaps with Dept = EECS?
- Modern consensus on how to handle phantoms:
  - Restrict predicate locking to indexable ranges (e.g., B+tree intervals)
    - Lock ranges e.g., as in Lomet 1993
    - Operationally, easy to coordinate in a hierarchical index (Q: what does this remind you of? remember multi-granularity locking?) - details differ
  - Or: if no index exists, can use coarser grained locking (page/table)
  - Or: not a locking-based approach at all (e.g., SI - more next or SSI - will see next time)

ANSI Isolation Levels and Problems

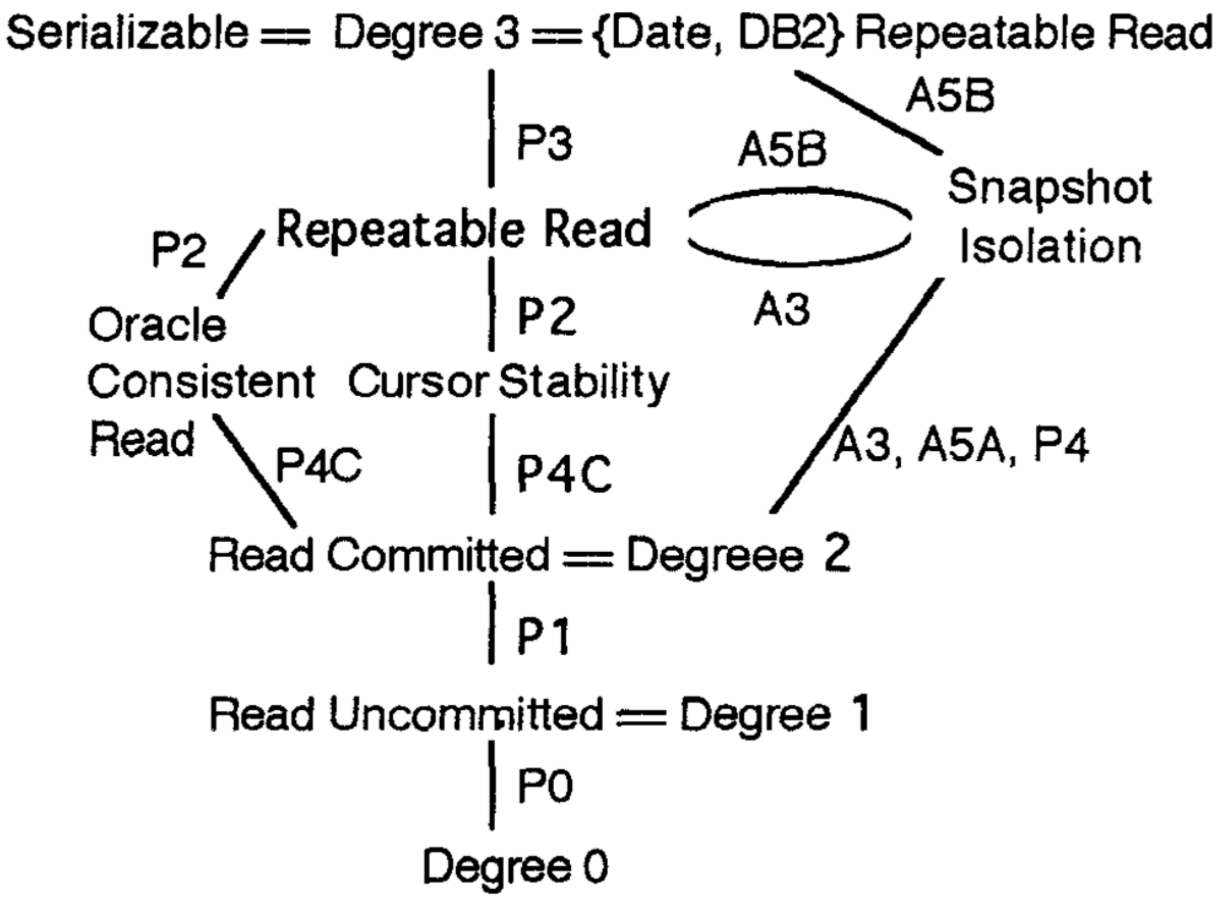
- This plus refinement by Date, led to ANSI isolation levels SQL 92

  - Goal - implementation independence
    - Actual impact: Definitions less constraining
  - Described *phenomena* that are proscribed, such as dirty read, non-repeatable read, phantoms, ...
  - And corresponding isolation levels, read uncommitted, read committed, repeatable read, serializable, etc.
    - We'll see what these are in a bit
- Berenson and co said:
  - These are ambiguous, and do not exclude some anomalous behavior
    - No clear mapping between locking isolation levels and the phenomena
  - Also doesn't identify other popular isolation levels
- So redefined the phenomena:
  - P0: w1[x] ... w2[x] ... (c1 or a1) - dirty write
    - Can't write an object if an active transaction has already written to it.
    - missing in the ANSI-SQL definition
  - P1: w1[x] ... r2[x] ... (c1 or a1) - dirty read
    - Can't read an object an active transaction has already written
  - P2: r1[x] ... w2[x] ... (c1 or a1) - non-repeatable read
    - Can't write an object that has been read by an active transaction
  - P3: r1[P] ... w2[y in P] ... (c1 or a1) - phantoms
    - Can't modify rows matching a predicate that an active transaction has read

Locking Isolation Level	**Proscribed Phenomena**	**Read Locks on Data Items and Phantoms (same unless noted)**	**Write Locks on Data Items and Phantoms (always the same)**
Degree 0	none	none	Short write locks
Degree 1 = Locking READ UNCOMMITTED	P0	none	Long write locks
Degree 2 = Locking READ COMMITTED	P0, P1	Short read locks (on both)	Long write locks
Locking REPEATABLE READ	P0, P1, P2	Long data-item read locks, Short phantom read locks	Long write locks
Degree 3 = Locking SERIALIZABLE	P0, P1, P2, P3	Long read locks (on both)	Long write locks

- Other phenomena: P4: r1[x]... w2[x] ... w1[x]... c1 - lost update
  - Note that P4 is already proscribed by P2 - a special case of P2
  - So helps identify isolation levels between READ COMMITTED and REPEATABLE READ
- Two new serializable levels, all between Degree 2 and Degree 3
  - Cursor Stability
    - Details are not important but prevents P4
    - Amounts to READ COMMITTED $\ll$ Cursor Stability $\ll$ REPEATABLE READ
  - Snapshot Isolation
    - MVCC approach not captured in ANSI Levels
    - Txns take and operate on copies, like OCC
    - Txn Ti are allowed to commit if no other txn in the interim wrote data that Ti wrote
      - "First committer wins" on ww conflicts
    - Like cursor stability, lost update (P4) can't happen because w2 above won't be able to commit
    - But not serializable:
      - r1[x=50] r1[y=50] r2[x=50] r2[y=50] w1[y=-40] w2[x=-40] c1 c2
      - Essentially: r1[x]...r2[y]...w1[y]...w2[x]...(c1 and c2 occur)
      - this is called *write skew* - there is an analogous one called *read skew*

        - We can show that READ COMMITTED $\ll$ Snapshot Isolation
        - But, interestingly REPEATABLE READ $\ll\gg$ Snapshot Isolation
            - For example, write skew is prohibited in REPEATABLE READ
            - Likewise, reading the same item cannot result in different values in SI
    - Read consistency
        - Omitted details (a bit hard to follow from Berenson)
        - READ COMMITTED $\ll$ Read Consistency $\ll$ REPEATABLE READ
        - and Read Consistency $\ll\gg$ Cursor Stability



Onto Adya

- But P0 is essentially a long-duration write lock!
    - P1 requires long-duration write locks, and readers to take short-duration read locks
    - P2 implies long-duration read and write locks
    - P3 involves predicate locks
- Problem: P0, P1, P2, P3 disallows many legal schedules
    - P0/P2 - can happen in optimistic implementations - many txns are editing local or reading copies simultaneously, but some will be forced to abort
- At its core:
    - Preventative approach expresses phenomena in terms of single-object histories
        - both OCC and MVCC-based approaches can't be modeled that way
    - Preventative approach also has same properties for running and committed txns
        - but OCC-based approaches have weak guarantees while running, and then strong consistency for committed txns
- What Adya hopes for: Implementation-independent spec of isolation levels
    - Both correct (rule out bad histories) and complete (include all conflict serializable schedules)

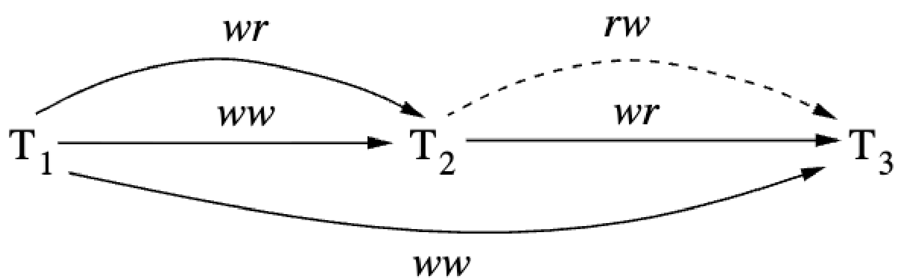
        - For lower isolation levels, more permissive than Berenson
- Done by proscribing (eliminating) different types of cycles in a serialization graph

Adya's Formalization

- $x_{i.1}, x_{i.2}, \ldots$ denotes kth modification of x by T_i; final modification is x_i.
- After committing T_i *installs* x_i.
- Each object starts in the x_{init} state (unborn), then becomes "visible", then becomes "dead"
    - If comes back again, essentially a new object
- $w_i(x_{i.m})$ - write of x by T_i.
- $r_j(x_{i.m})$ - read by T_j of the mth version written by T_i.
- A history is a partial order that:
    - preserves individual op order by transactions
    - for something to be read, it must have been written - $r_j(x_{i.m})$ must be preceded by a $w_i(x_{i.m})$
    - if a txn writes then reads immediately without an intervening event, then it must see its write, e.g., $r_i(x_j)$ after a $w_i(x_{i.m})$ should be $j = i.m$
    - must be complete: if you see any events for T_i it must be committed
        - note that we don't reason about uncommitted txns
            - i.e., there can be violations that "get fixed later"
        - one way to reason about partial histories is to add aborts for uncommitted txns
            - "worst-case behavior"
- In addition to events, we want a total order $\ll$ of versions of object created by committed txns (also called committed versions)
    - x_{init} ...(various visible versions) ... x_{dead}
    - if $r_j(x_i)$ is in a schedule, it is visible
- Schedule:
    - $w_1(x_1)w_2(x_2)w_2(y_2)c_1c_2r_3(x_1)w_3(x_3)w_4(y_4)a_4. \ldots x_2 \ll x_1$
    - x_2 serialized before x_1 even though commits are other way around
- Predicates
    - $VSet(P)$ set of versions of all tuples read by P
    - $r_i(P : VSet(P))r_i(x_j)r_i(y_k)\ldots$ - where x_j, y_k all versions in $VSet$ that match P and read by T_i
    - While most of the real-estate in 4.3 onwards is on predicates, we are going to ignore it and try to focus on the simpler setting without predicates

Direct Serialization Graphs

- Three types of edges in the graph:
    - direct write dependency ww : T_i installs x_i and T_j installs next version
    - direct read dependency wr : T_i installs x_i and T_j reads x_i or (predicate based version)
    - direct anti dependency rw : T_i reads x_h and T_j installs next version (or predicate based version)
- History : $w_1(z_1)w_1(x_1)w_1(y_1)w_3(x_3)c1r_2(x_1)w_2(y_2)c_2r_3(y_2)w_3(z_3)c_3$ plus
    - $x_1 \ll x_3, y_1 \ll y_2, z_1 \ll z_3$



-
  - Serializable as $T_1 \ll T_2 \ll T_3$
- Definitions:
  - *direct depends*: if there is a direct read or write
  - *depends*: if there is a path

Isolation level PL-1

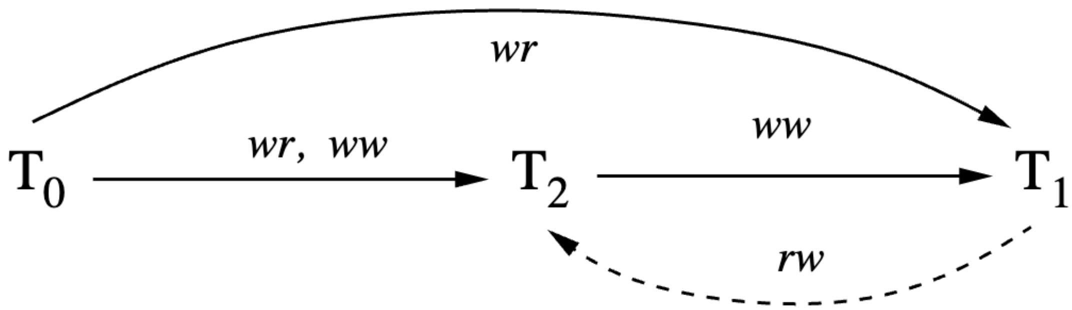
- Prohibiting something analogous to
  - P0: w1[x] ... w2[x] ... (c1 or a1) -- dirty write
- The generalizable phenomenon, G0 is *no write dependency cycles*
- $w_1(x_1, 2)w_2(x_2, 5)w_2(y_2, 5)c_2w_1(y_1, 8)c_1$ alongside
  - $x_1 \ll x_2, y_2 \ll y_1$
- T1 --- ww ---> T2 (and back ww edge)
- More permissive because same object can be concurrently modified unlike P0

Isolation level PL-2

- Prohibiting something analogous to
  - P1: w1[x] ... r2[x] ... (c1 or a1) -- dirty read
- This is a bit complicated (omitting predicate versions):
  - G1a - No *aborted reads*
    - $w_1(x_{1.i}) \dots r_2(x_{1.i}) \dots$ (a_1 and c_2 in any order)
    - Can't have a committed txn reading a version created by an aborted one
  - G1b - No *intermediate reads*
    - $w_1(x_{1.i}) \dots r_2(x_{1.i}) \dots w_1(x_{1.j}) \dots c_2$
    - Can't have a committed txn reading "intermediate" versions
  - Together G1a and G1b ensures committed txns only read objects that existed or will exist at some instant in the committed state
  - G1c - No *circular info flow*
    - Directed cycle of dependency edges (read or write edges)
    - Also includes G0 by definition
- G1 is weaker than P1 because G1 allows reads from uncommitted txns
- And lock-based impl (long write locks, short read locks) will guarantee G1
  - G1a and G1b can't happen because T_2 can't read before T_1 commits
  - G1c if T_1 read something T_2 wrote, then T_2 can't read/write something T_1 wrote

Isolation Level PL-3

- Prohibiting something analogous to:
  - P2: r1[x] ... w2[x] ... (c1 or a1) -- non repeatable read
- Simple enough: G2: No *cycles with one or more anti-dependency edges*.

- G2 + G1 together give something analogous to P2
- If we omit predicate centric dependency edges, we have the variant PL-2.99
- G2 also prevents lost update (just like P2 prevents P4): from Adya's Thesis
    - Lost update history : r1 (x0, 10) r2(x0 , 10) w2(x2 , 15) c2 w1(x1 , 14) c1
    - with $x_0 \ll x_2 \ll x_1$
    - 

More reading

- Adya's thesis also formalizes SI and Cursor Stability through the DSG, and identifies new isolation levels as well