Locking

- **Consistency requires system to appear as though all transactions happened sequentially**
- **Need locking for consistency in concurrent system**
- **First plan: Simple record locking**
  - If you read a record, acquire shared lock to end of xaction
  - If you write a record, get exclusive lock to end of xaction
  - Thus, no one will change a record you are accessing
  - May experience deadlock, but system can abort transaction
- **Is this good enough?**

Phantoms

- **Consider the following example:**
  - C_1 queries DB for all suspects with blue eyes, red hair
  - C_2 inserts new suspect with blue eyes & red hair
- **Problem: Can't lock a record that doesn't exist yet**
  - C_1 may lock all appropriate suspects
  - Won't prevent C_2 from inserting new suspect, violating consistency
- **How to prevent?**

Solution 1: Predicate locks

- **Idea: Instead of locking records, lock predicates**
  - E.g., lock all potential records with blue eyes/red hair
  - Two predicates p, p' compatible iff no record can satisfy p AND p'
  - Exclusive/shared locking based on compatible predicates
- **Problem: Expensive/impossible to implement**
  - Determining if predicates are compatible is NP-hard
  - "Compatibility" may be overly pessimistic—p and p' may not be compatible, but DB may have invariant C such that p AND p' AND C is impossible

Solution 2: Precision locks

- **Use sh/ex set-oriented locks + ex record locks**
  - When locking predicates, all record locks compared
  - When writing record, *both* old and new value checked against all locked predicates
  - When reading record, check against ex predicate locks
- **Drawbacks:**
  - Still fairly expensive to implement (lots of checks)
  - Very deadlock prone—lots of aborted xactions

Solution 3: Granular locks

- **Idea: Pick predicates, precompute dependencies**
- **Organize predicates as tree (or DAG), E.g.:**
  - Whole database (all records)
  - Site: New York, Boston, San Francisco
  - File (within site): Phone, email, . . .
  - Record (within file): ⟨Name, phone number⟩, . . .
- **Applications can lock at any granularity**
  - Small transactions don't need to lock whole DB
  - Big ones may need to
  - Because of hierarchy, easy to see how predicates conflict
- **Problem: Will cause lots of deadlocks**

Intent locks

- **Goal: Reduce incidence of deadlocks**
- **Have 3 types of locks: shared, exclusive, & *intent***
  - Intent locks declare intent to lock at finer granularity
  - Prevent shared & intent from existing at same granularity
- **Allowed locks given existing locking mode:**

	None	Intent	Shared	Exclusive
Intent	+	+		
Shared	+		+	
Exclusive	+			

Update locks

- **Updating Sh→Ex a major source of deadlocks**
  - Virtually all deadlocks in System R of this form
- **Solution: New type of lock, *update***
  - Like shared lock, but declares intention to upgrade to ex
- **Asymmetric lock compatibility:**
  - Can acquire Update lock on record that has shared lock
  - Cannot acquire shared lock on record with update lock
  - Can downgrade update to shared if no modificatio needed

Full locking matrix

	CURRENT LOCK MODE						
REQUEST	**None**	**IS**	**IX**	**S**	**SIX**	**Update**	**X**
IS	+	+	+	+	+		
IX	+	+	+				
S	+	+		+			
SIX	+	+					
U	+			+			
X	+						

Key-range locking

- **Useful for access to structures such as sorted lists**
  - 4 Operations: Read unique, read next, insert delete
  - Phantom records arise if insert before read next
- **Solution: break key space into ranges**
  - After "read unique X", can just lock ranges around X
  - Guarantees consistency for access to next element
  - But need to determine ranges ahead of time
- **Alternate: Dynamic key-range locks**
  - E.g., previous-key & next-key locking
  - Range depends on contents of DB