## COMP421 Crib Sheet Francis Piché

### Transactions

-A sequence of reads **r(x)** and writes **w(x)**

-**Atomic** (all or nothing)

-Keep *backup* of state before transaction

-Restore to this point in case of failure

-**Consistency** (preserve consistency)

-**Isolation** must have serial equivalent

-**Durability** must be permanent/fault tolerant

Transactions can be **aborted**

-*Global recovery:*

-Transactions committed before crash are in effect.

-Transactions aborted before crash are reversed

-Transactions active at time of crash are reversed

-Assume disk doesn't crash

**Logs:** are kept because holding back writes is insufficient. Limited number of buffer frames means transactions cannot all be atomic.

-Log writing is fast since logs are *append-only*. Save on seek time.

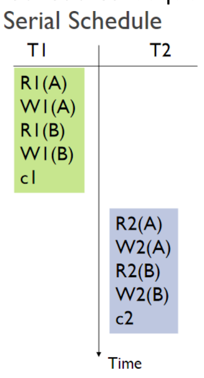
### Concurrency:

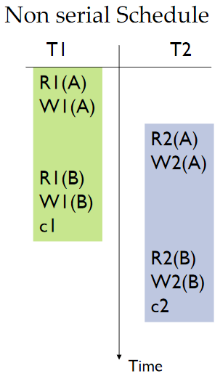
-Transactions need to run in isolation

-Must have concurrency control protocol to enforce this

-Ensure net effect of concurrent transactions is equivalent to some serial order

-**Schedules**: sequence of actions (reads/writes)





-**Unrepeatable read:** Two or more reads that give different results (another transaction changed the value in between).

-**Lost Update:** A write of T1 overwritten by the write of T2.

-**Dirty Read:** Read value that doesn't exist (was undone by an abort later)

-**Dirty Write:** (permanent damage)

-**Conflicting Operations:**

-Same object being accessed

- WW, WR(not RR)

-**Conflict Equivalent:** Schedule is conflict equiv if:

-Every pair of conflicting actions is ordered same way

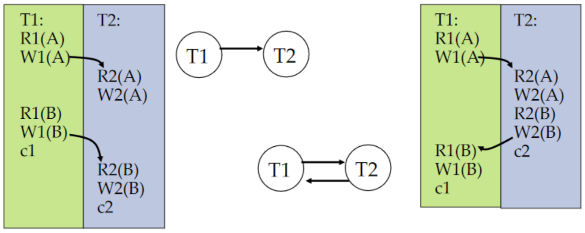
-Same actions of same committed transactions

-**Conflict Serializable*:** if:

-Equivalent to some serial schedule with actions of schedule

**Dependency Graphs:**

-Not serializable if there is a cycle in the dependency graph:



-Edge formed if first operation conflicts with later one (edges are always downward in time

-**Forming serial schedules:**

-Choose node with no incoming edges

-Put in in the schedule, delete it and all outgoing edges

-Repeat until no more nodes remaining

### Locking:

-Transactions must acquire shared lock S

-Exclusive lock X before writing

-X blocks all other operations

-S blocks only writes

| | S | X |
|---|---|---|
| S | ✓ | -- |
| X | -- | -- |

-**Phase 1**: *Acquire* locks when needed

-**Phase 2**: *Release* locks at end of transaction

-Two phase locking allows only serializable schedules

-No dirty reads/writes possible

-Transaction *cannot acquire same lock twice*

-No need to acquire S on resource if already have X for resource

-Locks are managed using **lock table**

-Entry for each resource that is locked

-Pointer to queue of locks granted

-Pointer to queue of lock requests (waiting)

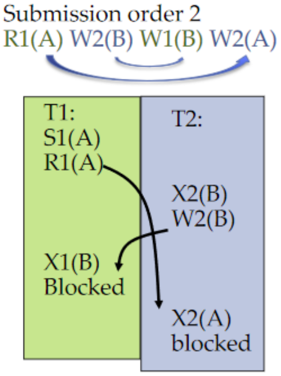
-A transaction has only one lock per object

-If T has S and requests X, S is upgraded to X.

-Keep track of type of lock held

-Pointer to list of locks held by each T

-Locking/Unlocking is atomic



**Deadlock:**

-Deadlock is possible with two phase locking. (SEE LEFT)

**Wait-For Graph:**

-Like dependency graph but arrows are BACKWARDS

-Edge from Ti to Tj if Ti waits for Tj to release lock.

-Cycles mean there is deadlock

-Avoid deadlock by breaking cycles.

-Can try timeout but how long should you wait?

-Can try to request all locks at beginning of transaction (loss of concurrency)

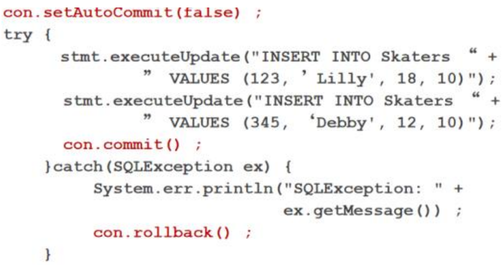
-Optimistic concurrency control: Try transaction (no locking), if conflict, abort.

**Snapshots:**

-Writers make new copy

-Readers use old copy

**Transactions in Java:**



```
con.setAutoCommit(false) ;
try {
      stmt.executeUpdate("INSERT INTO Skaters  “ +
               ”  VALUES (123, ’ Lilly', 18, 10)");
      stmt.executeUpdate("INSERT INTO Skaters  “ +
               ”  VALUES (345,  ‘Debby', 12, 10)");
      con.commit() ;
    }catch(SQLException ex) {
         System.err.println("SQLException: " +
                            ex.getMessage()) ;
         con.rollback() ;
    }
```

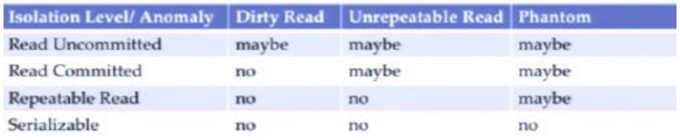
**Phantoms:**

-Can arise when new entries being added concurrently

-Locking can’t prevent

-If inserting while doing aggregation, aggregate gets weird values

**Isolation Levels:**



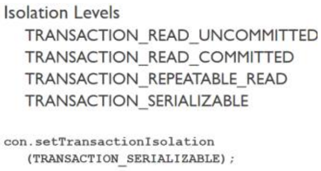
| Isolation Level/ Anomaly | Dirty Read | Unrepeatable Read | Phantom |
|---|---|---|---|
| Read Uncommitted | maybe | maybe | maybe |
| Read Committed | no | maybe | maybe |
| Repeatable Read | no | no | maybe |
| Serializable | no | no | no |

-Read uncommitted = Read operation does not set locks, can read uncommitted writes

-Read Committed = Do not read uncommitted writes. Release read lock immediately after reading.

-Repeatable reads = standard S locking on reads

-Serializable = lock entire relation



Isolation Levels

- TRANSACTION_READ_UNCOMMITTED
- TRANSACTION_READ_COMMITTED
- TRANSACTION_REPEATABLE_READ
- TRANSACTION_SERIALIZABLE

```
con.setTransactionIsolation
   (TRANSACTION_SERIALIZABLE);
```

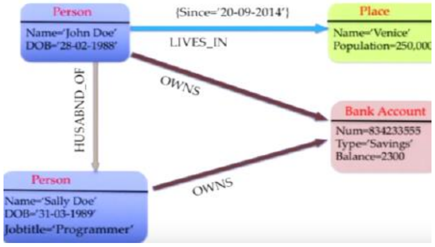
## <u>Graph Databases: (FLEXIBLE)</u>

-Each vertex has own properties

-Properties are K-V pair

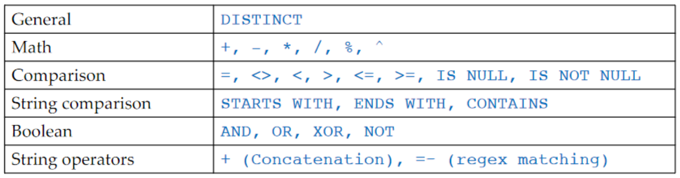
-Can easily be extended. No pre-planning required

-Edges can have properties too (are directional)



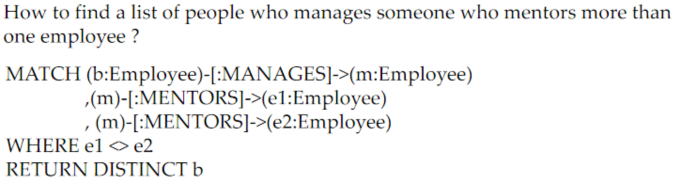
**<u>Cypher:</u>**

<u>TRAVERSALS:</u>



| General | DISTINCT |
|---|---|
| Math | +, -, *, /, %, ^ |
| Comparison | =, <>, <, >, <=, >=, IS NULL, IS NOT NULL |
| String comparison | STARTS WITH, ENDS WITH, CONTAINS |
| Boolean | AND, OR, XOR, NOT |
| String operators | + (Concatenation), =~ (regex matching) |

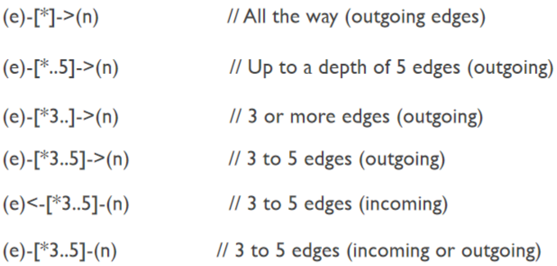
Can combine conditions by comma separating:



How to find a list of people who manages someone who mentors more than one employee ?

```
MATCH (b:Employee)-[:MANAGES]->(m:Employee)
          ,(m)-[:MENTORS]->(e1:Employee)
          , (m)-[:MENTORS]->(e2:Employee)
WHERE e1 <> e2
RETURN DISTINCT b
```

EACH EDGE IS TRAVERSED ONLY ONCE TO AVOID CYCLES



| | |
|---|---|
| (e)-[*]->(n) | // All the way (outgoing edges) |
| (e)-[*..5]->(n) | // Up to a depth of 5 edges (outgoing) |
| (e)-[*3..]->(n) | // 3 or more edges (outgoing) |
| (e)-[*3..5]->(n) | // 3 to 5 edges (outgoing) |
| (e)<-[*3..5]-(n) | // 3 to 5 edges (incoming) |
| (e)-[*3..5]-(n) | // 3 to 5 edges (incoming or outgoing) |

| | |
|---|---|
| SELECT * FROM Employees | MATCH(e:Employee) RETURN e; |
| SELECT email FROM Employees | MATCH(e:Employee) RETURN e.email; |
| …. ORDER BY email | … RETURN e ORDER BY e.email; |
| …WHERE name = ‘Janet’ | MATCH(e:Empl {ename: ‘Janet’} RETURN e; |
| … WHERE deptid IS NULL | … WHERE NOT (e)-[:WORKS_IN]-() … |
| | … WHERE e.job IS NULL …<br>(treat non-exist property as NULL) |
| INSERT INTO … | CREATE (e:Empl {name: ‘Jane’}-[:WORKS_IN]->(d:Depart {dname:’PR’} ); |
| New edge b/w existing nodes: | MATCH (n1: Empl {eid: 101}), (n2: …)<br>CREATE (n1)-[:MANAGES]->(n2); |
| DELETE FROM …<br>(Must delete relationships) | MATCH(e: …)-[r:WORKS_IN]->(d:Dep..)<br>DELETE e, r, d; |
| Delete all edges connected to this node | DETACH DELETE e; |