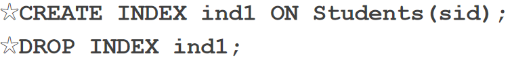
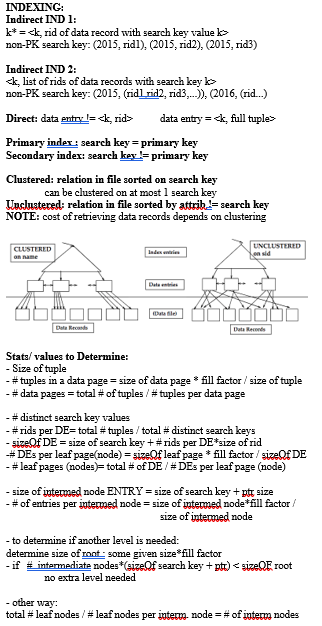
**COMP421 Crib Sheet Francis Piché**

DON’T RELY ON THIS CRIBSHEET, YOU WON’T HAVE TIME TO LOOK AT IT DURING THE EXAM!

**Indexing:**



**INDEXING FORMULAS TAKEN FROM**

**LEILA AE’S MIDTERM CRIBSHEET**

**Query Evaluation:**

-*Parser translates* query into internal expression

-*Query optimizer* turns internal expression into plan

-*Plan Executor* executes execution plan

-**Access Path:** The method used to get a set of tuples from a relation.

-**Cost model:** Metric used to compare different execution paths.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_USEFUL FORMULAS\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

-**Reduction factor:**



**Sort Costs:**

,

(N read + N write per pass)

*(Last pass doesn’t need to write if pipelining)*

**Join Cardinality**

*If Join Attrib is Primary Key for A*:

**Join Costs (A JOIN B):**

*Simple Nested Loop Join:*

*Page Nested Loop:*

*\*Block Nested Loop:*



Block size ^

-Best case: (outer fits in mem)

*Index Nested Loop:*

**INDEX NESTED LOOP FASTER THAN BLOCK NESTED WHEN:**

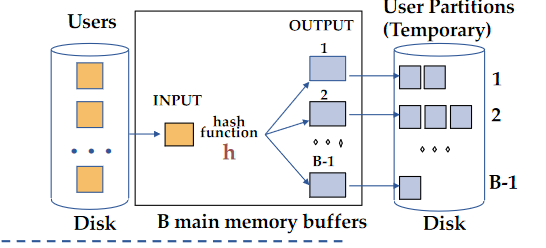
-PAGES(OUTER) > |OUTER| \* (MATCHING TUPLES INNER)

(When outer is result of selection that resulted in few tuples)

**Sort Merge Join:**

Assume already sorted, if not then sort and pipeline to Join

**Hash Join:**



-Send outer table to hash function.

-One buffer used for hashing, then B-1 partitions are made

-Repeat for inner table

-Join by loading each partition bin and finding matches (they will always be together since sorted by hashing)

-Assume # Pages < B-2

I/O:

**MERGE JOIN VS HASH JOIN**

-Hash join better if one relation very large (sort would require multiple passes)

-Merge join better if partitions don’t fit in memory

-Hash Join good for parallelization

**-SELECT costs**

-No index, arbitrary attribute: (full scan)

-No index, primary key: (on average scan half pages)

-Clustered B+ Tree: Read left leaf + all matching tuples until cond fails

-Unclustered:

1 Leaf + #Data pages = (worst case) #matching tuples

(potentially worse than scan) (index only useful on small redux factors)

-Selection with A AND B:

No Index: Read potentially all data pages containing match

Index on A:

-Read all pages for A

2 Indexes:

-Find RIDs for A, B

-Make intersection of RIDS

-Retrieve data pages through intersection

Multi-Attrib (A,B) index:

-Read all A data pages(potentially), then

-Selection with A OR B:

No Index:

-Read all data pages

Index on A:

-Not useful

2 Indexes:

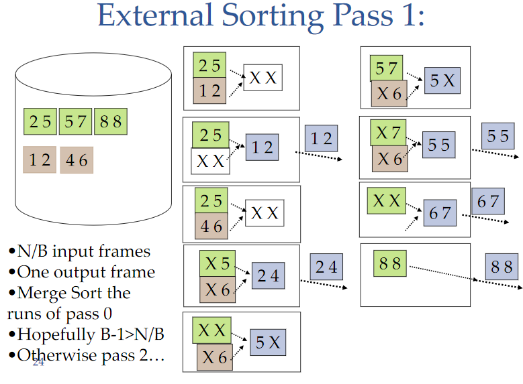
-Union of RIDS

(A,B) index:

-Read all leaf pages anyways

**External Sorting:**

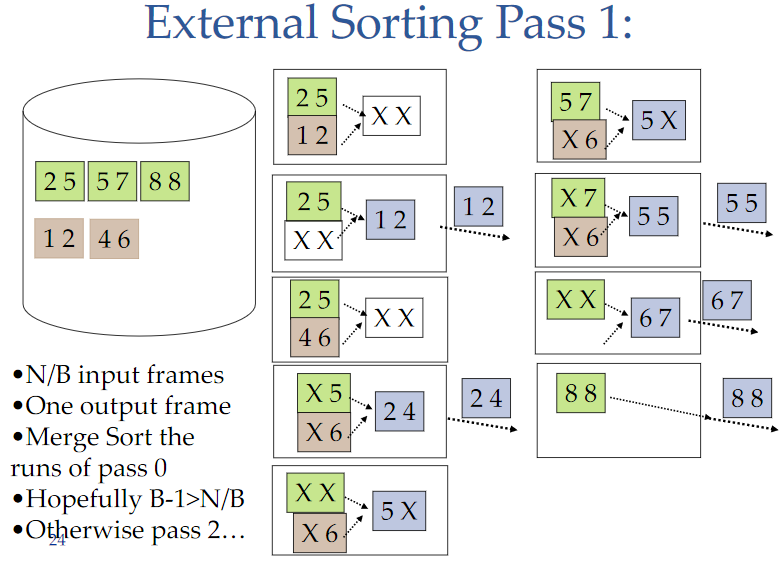
**-**Pass 0 just copies to temp file



**Union and Except:**

**-**Sort both relations on combo of all attributes

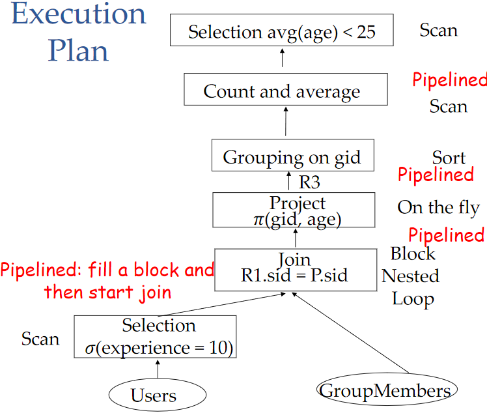
-Scan through sorted relations and merge

**Aggregations:**

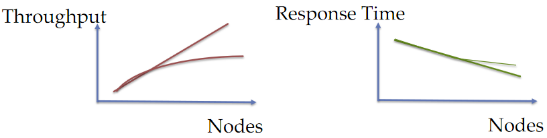
Without grouping: Scan entire relation

With Grouping: Sort on group attributes, scan relation (can combine)

Example Execution plan:



**Large-Scale Data Processing:**



- The more nodes we have, the more coordination overhead

- **Skew** is where its not possible to evenly distribute workload

-**Inter-Query:** Different queries running in parallel

-**Inter-operator:** Differnent operators within one query running in parallel (pipelining)

-**Intra-Operator:** Single operator running on many processors

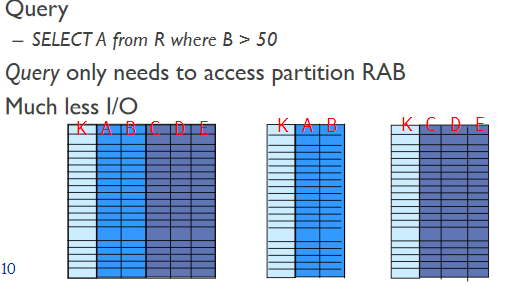
-**Horizontal Data Partitioning:**

**-**Cut table into chunks store each chunk on separate node. Usually done by hash funciton or range partition.

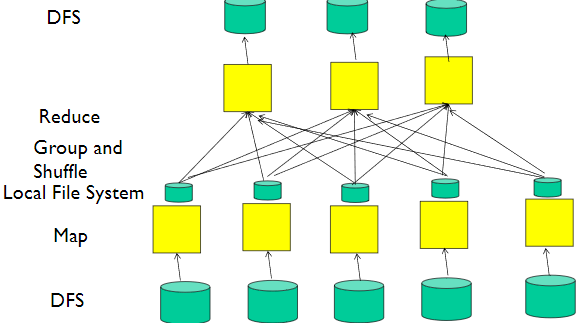
-Execute locally at each node, push result to coordinating node. Assemble result and return to user.

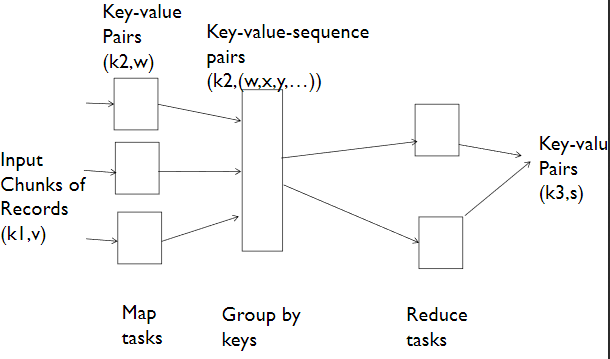
**-Vertical Data Partitioning:**

-Store each column on separate node.



*Note key is replicated* since must know how to find the data.

**Map-Reduce:**



-**Map task**: Extract data from records, outputnew data records

-Shuffle and sort (to send same keys to same reduce task)

-**Reduce task:** aggregate, summarize, filter



**Combining (optional):** Happens before the shuffle. Apply reduce function at each mapper on the partial result of mapper

-Only if reduce function is commutative and associative

**Failure Handling:**

-Detected by master node

*-Map Node Fails During Map Phase:*

*-*Move job to different node

-*Map Node Fails During Reduce:*

*-*Don’t care. Unless reduce node also fails, then bad. Handle by creating a redundant map job to replace

-*Reduce Node Fails During Reduce:*

*-*Move job to different node

-*Stragglers:*

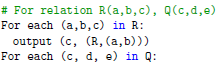
-Duplicate task and see if it finishes faster

**Selection with Map-Reduce:**

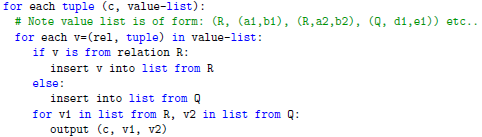
-Map: for each tuple, output (k, (a1,…,an)) if condition holds

-reduce: Identity function

**Join with Map-Reduce:**

Map:



Reduce:

**Projection with map reduce:**

Given R(A,B,C) project to keep only A, B:

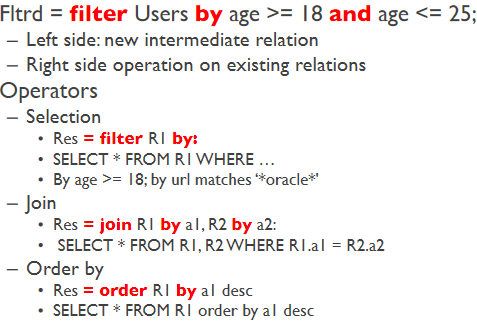
Map: -> ((a,b), 0) (removes duplicates)

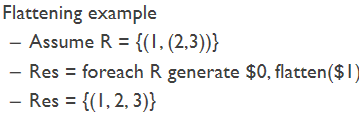
Reduce : ((a,b), (0,0,0,0,0…)) -> ((a,b), 0)

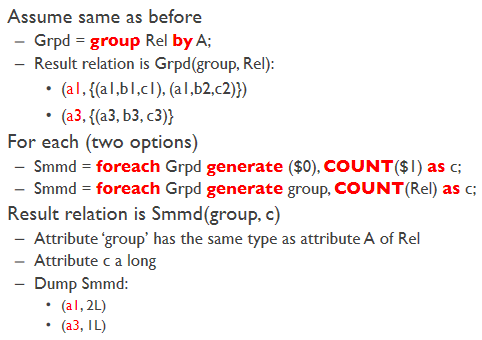
**Group By With Map Reduce:**

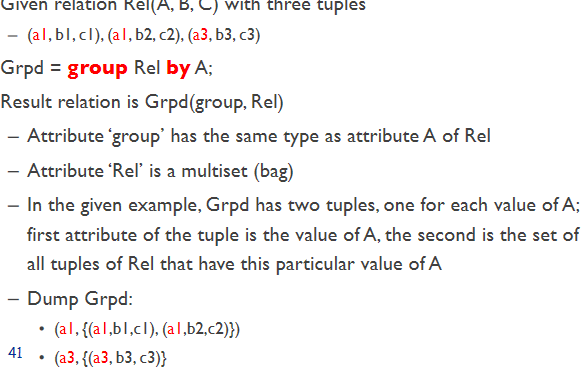
-Use group by attrib as key so that go to same reducer node.

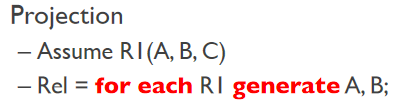
-Reducer just performs aggregation

**Pig Latin:**









**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.

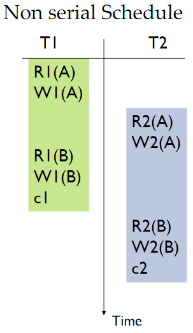
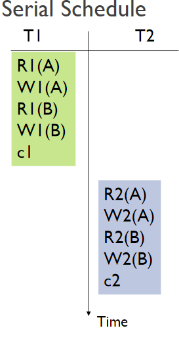
**Concurrency:**

-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:

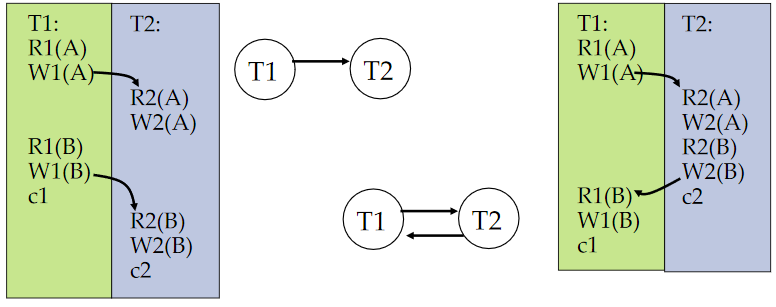
-Every pair of conflicting actions is ordered same way

-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

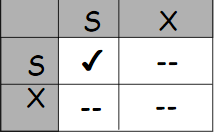
-**Forming serial schedules:**

-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

-**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.

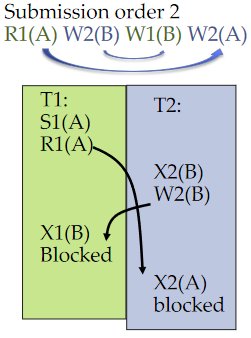
-Keep track of type of lock held

-Pointer to list of locks held by each T

-Locking/Unlocking is atomic

**Deadlock:**

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



**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)

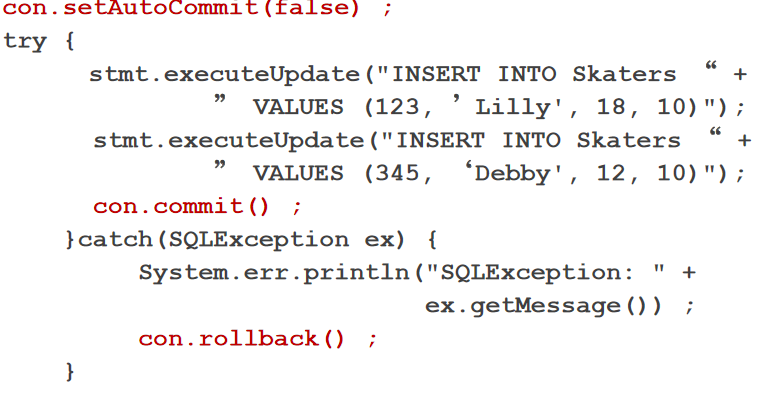
-Optimistic concurrency control: Try transaction (no locking), if conflict, abort.

**Snapshots:**

-Writers make new copy

-Readers use old copy

**Transactions in Java:**

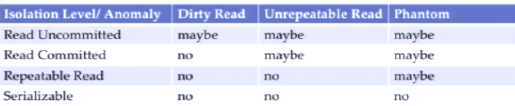


**Phantoms:**

-Can arise when new entries being added concurrently

-Locking can’t prevent

-If inserting while doing aggregation, aggregate gets weird values

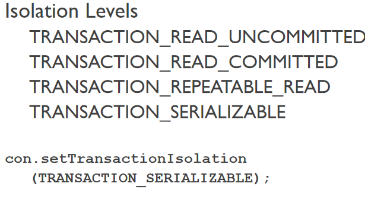
**Isolation Levels:**

-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.

|  |  |
| --- | --- |
| 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 …  (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: …)  CREATE (n1)-[:MANAGES]->(n2); |
| DELETE FROM …  (Must delete relationships) | MATCH(e: …)-[r:WORKS\_IN]->(d:Dep..) DELETE e, r, d; |
| Delete all edges connected to this node | DETACH DELETE e; |

-Repeatable reads = standard S locking on reads

-Serializable = lock entire relation

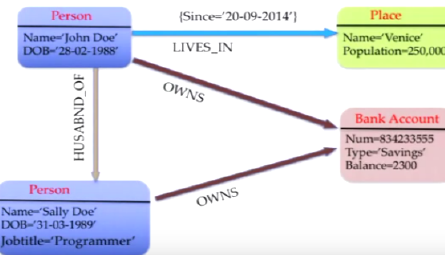
**Graph Databases: (FLEXIBLE)**

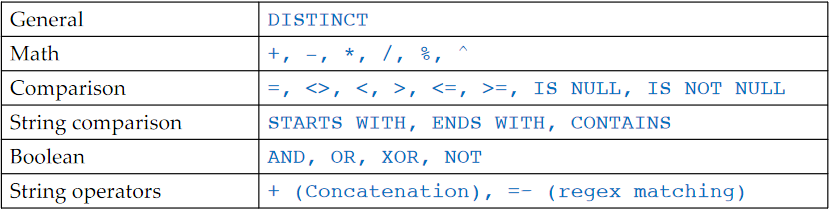
-Each vertex has own properties

-Properties are K-V pair

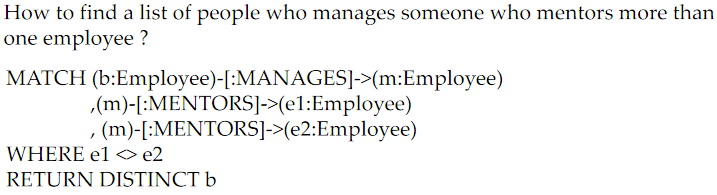
-Can easily be extended. No pre-planning required

-Edges can have properties too (are directional)



**Cypher:**

TRAVERSALS:

Can combine conditions by comma separating:

EACH EDGE IS TRAVERSED ONLY ONCE TO AVOID CYCLES

