COMP421 Crib Sheet Francis Piché

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

Indexing:

```
☆CREATE INDEX ind1 ON Students(sid);
☆DROP INDEX ind1;
```

INDEXING FORMULAS TAKEN FROM LEILA AE'S MIDTERM CRIBSHEET

INDEXING:

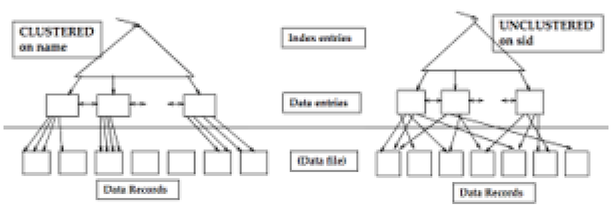
Indirect IND 1:
k* = <k, rid of data record with search key value k>
non-PK search key: (2015, rid1), (2015, rid2), (2015, rid3)

Indirect IND 2:
<k, list of rids of data records with search key k>
non-PK search key: (2015, (rid1,rid2, rid3,...)), (2016, (rid...)

Direct: data entry != <k, rid>   data entry = <k, full tuple>

Primary index : search key = primary key
Secondary index: search key != primary key

Clustered: relation in file sorted on search key
can be clustered on at most 1 search key
Unclustered: relation in file sorted by attrib != search key
NOTE: cost of retrieving data records depends on clustering



Stats/ values to Determine:
- Size of tuple
- # tuples in a data page = size of data page * fill factor / size of tuple
- # data pages = total # of tuples / # tuples per data page

- # distinct search key values
- # rids per DE= total # tuples / total # distinct search keys
- sizeOf DE = size of search key + # rids per DE*size of rid
-# DEs per leaf page(node) = sizeOf leaf page * fill factor / sizeOf DE
- # leaf pages (nodes)= total # of DE / # DEs per leaf page (node)

- size of intermed node ENTRY = size of search key + ptr size
- # of entries per intermed node = size of intermed node*fill factor / size of intermed node

- to determine if another level is needed:
determine size of root : some given size*fill factor
- if # intermediate nodes*(sizeOf search key + ptr) < sizeOF root
no extra level needed

- other way:
total # leaf nodes / # leaf nodes per interm. node = # of interm nodes

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:**

$$Red(\sigma_{condition}(R)) = \frac{|\sigma_{condition}(R)|}{|R|}$$

$$\#DATAPAGES = \frac{\#MATCHINGTUPLES}{TUPLES/PAGE}$$

Sort Costs:

$\#RUNS = \frac{N}{B}$, $\#Passes = 1 + \text{ceil}\left(\log_{B-1}\text{ceil}\left(\frac{N}{B}\right)\right)$

$SORTCOST = 2N * (\#\ PASSES)$ (N read + N write per pass)
(Last pass doesn't need to write if pipelining)

Join Cardinality

$|A \times B| = |A| * |B|$
If Join Attrib is Primary Key for A: $|A \bowtie B| = |B|$

Join Costs (A JOIN B):

Simple Nested Loop Join: $PAGES(A) + |A| * PAGES(B)$
Page Nested Loop: $PAGES(A) + PAGES(A) * PAGES(B)$
**Block Nested Loop:*

$$Pages(A) + \frac{Pages(A)}{|bp_a|} * Pages(B)$$

Block size ^

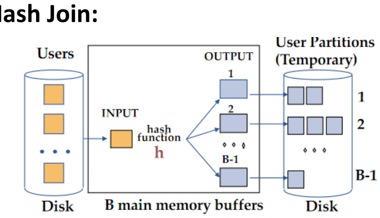
-Best case: $PAGES(A) + PAGES(B)$ (outer fits in mem)
Index Nested Loop:
$PAGES(A) + |A| * COST(lookupIndexOnB)$

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

$$\boldsymbol{PAGES(A) + PAGES(B)}$$

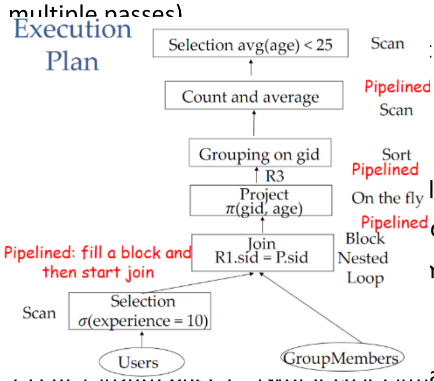
Hash Join:



-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: $3Pages(A) + 3Pages(B)$.

MERGE JOIN VS HASH JOIN

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



: in memory

ll scan)

can half pages)

natching tuples until

...atching 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:

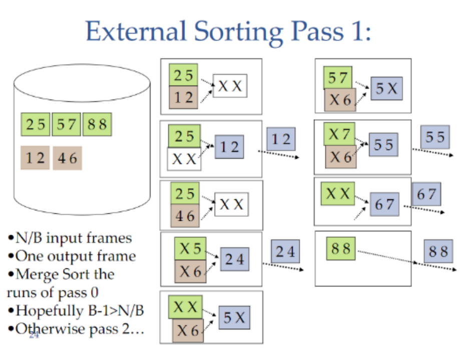
-Union of RIDS

(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

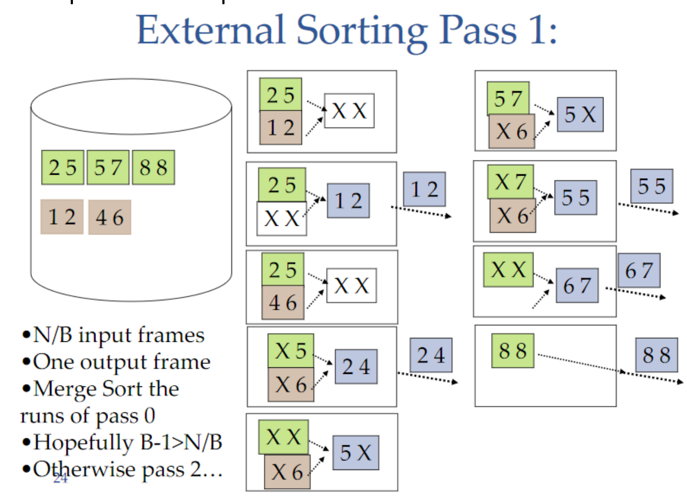
-Scan through sorted relations and merge

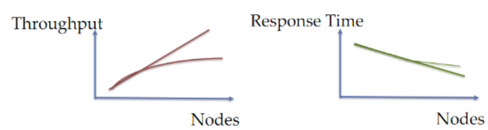
Aggregations:

Without grouping: Scan entire relation

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

Example Execution plan:





- 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

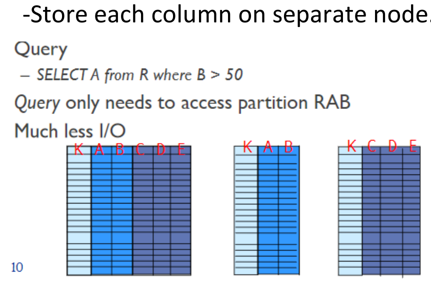
-**Horizontal Data Partitioning:**

-Cut table into chunks $C_1, \ldots, C_n$ 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.

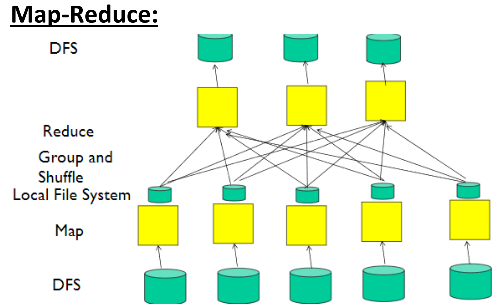
-**Vertical Data Partitioning:**

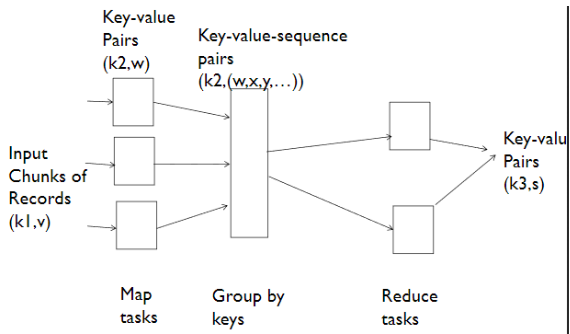
-Store each column on separate node.



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

<u>Map-Reduce:</u>





-**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

```
WordCountReduce:
  for each input key/value-list (k, v_1,..v_n))
    output (k, n)
```

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:

```
# For relation R(a,b,c), Q(c,d,e)
For each (a,b,c) in R:
  output (c, (R,(a,b)))
For each (c, d, e) in Q:
      output (c, (Q,(d,e)))
```

Reduce:

```
for each tuple (c, value-list):
  # Note value list is of form: (R, (a1,b1), (R,a2,b2), (Q, d1,e1)) etc..
  for each v=(rel, tuple) in value-list:
    if v is from relation R:
      insert v into list from R
    else:
      insert into list from Q
    for v1 in list from R, v2 in list from Q:
      output (c, v1, v2)
```

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

<u>Pig Latin:</u>

Fltrd = **filter** Users **by** age >= 18 **and** age <= 25;

– Left side: new intermediate relation

– Right side operation on existing relations

Operators

– Selection
  - Res **= filter** R1 **by:**
  - SELECT * FROM R1 WHERE …
  - By age >= 18; by url matches '*oracle*'

– Join
  - Res **= join** R1 **by** a1, R2 **by** a2:
  - SELECT * FROM R1, R2 WHERE R1.a1 = R2.a2

– Order by
  - Res **= order** R1 **by** a1 desc
  - SELECT * FROM R1 order by a1 desc

Flattening example

– Assume R = {(1, (2,3))}

– Res = foreach R generate $0, flatten($1)

– Res = {(1, 2, 3)}

Assume same as before

– Grpd = **group** Rel **by** A;

– Result relation is Grpd(group, Rel):
  - (a1, {(a1,b1,c1), (a1,b2,c2)})
  - (a3, {(a3, b3, c3)}

For each (two options)

– Smmd = **foreach** Grpd **generate** ($0), **COUNT**($1) **as** c;

– Smmd = **foreach** Grpd **generate** group, **COUNT**(Rel) **as** c;

Result relation is Smmd(group, c)

– Attribute 'group' has the same type as attribute A of Rel

– Attribute c a long

– Dump Smmd:
  - (a1, 2L)
  - (a3, 1L)

Given relation Rel(A, B, C) with three tuples

– (a1, b1, c1), (a1, b2, c2), (a3, b3, c3)

Grpd = **group** Rel **by** A;

Result relation is Grpd(group, Rel)

– Attribute 'group' has the same type as attribute A of Rel

– Attribute 'Rel' is a multiset (bag)

– In the given example, Grpd has two tuples, one for each value of A; first attribute of the tuple is the value of A, the second is the set of all tuples of Rel that have this particular value of A

– Dump Grpd:
  - (a1, {(a1,b1,c1), (a1,b2,c2)})
  - 41 (a3, {(a3, b3, c3)}

Projection

– Assume R1(A, B, C)

– Rel = **for each** R1 **generate** A, B;

<u>Transactions</u>

-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.

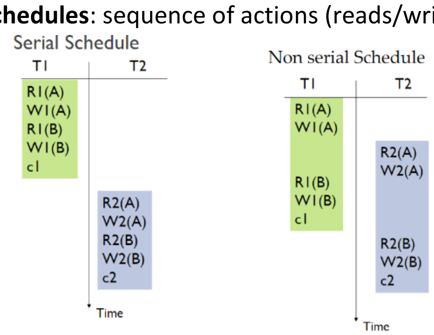
Concurrency:

-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

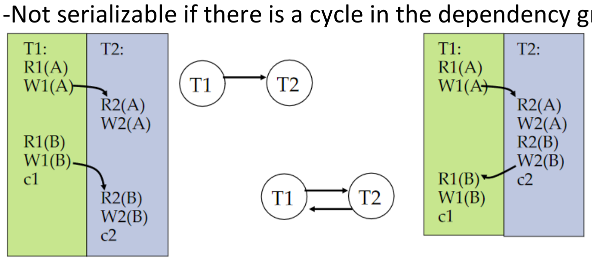
-Same actions of same committed transactions

-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 <u>must</u> 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.

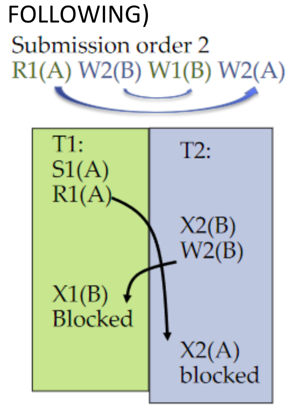
-Keep track of type of lock held

-Pointer to list of locks held by each T

-Locking/Unlocking is atomic

Deadlock:

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



-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);
```

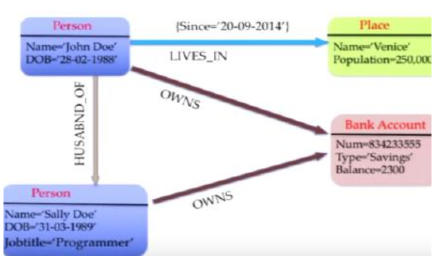
<u>Graph Databases: (FLEXIBLE)</u>

-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>

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)

<u>TRAVERSALS:</u>

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 ... (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;