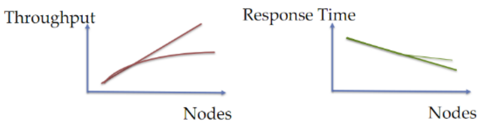
COMP421 Crib Sheet Francis Piché

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

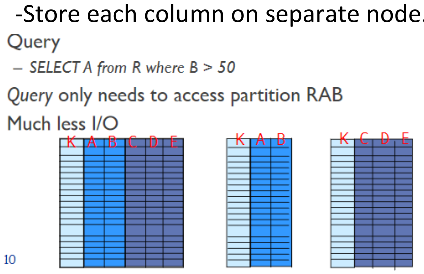
-**Horizontal Data Partitioning:**

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



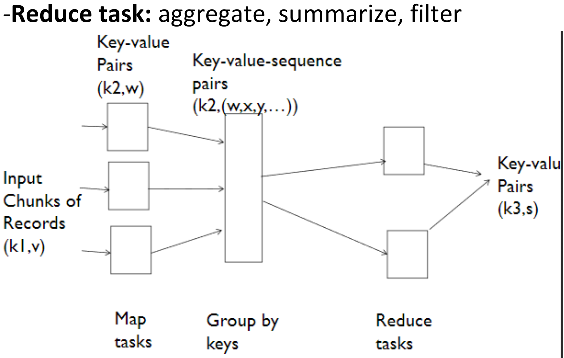
Note key is replicated since must know how to find the data.

Map-Reduce:

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

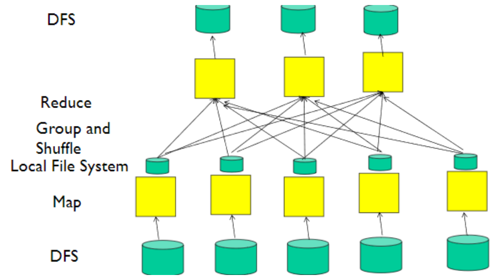
-Shuffle and sort (to send same keys to same reduce task)

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




```
WordCountMap:
  For each input k,v pair (dkey, dtext):
```

```
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

Pig Latin:

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

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)}

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)

Projection

- Assume R1(A, B, C)
- Rel = **for each** R1 **generate** A, B;

Flattening example

- Assume R = {(1, (2,3))}
- Res = foreach R generate $0, flatten($1)
- Res = {(1, 2, 3)}

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.

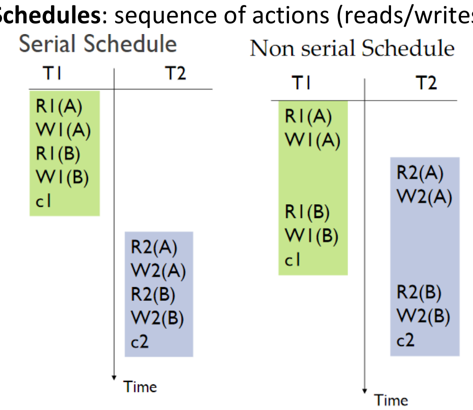
Concurrency:

-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

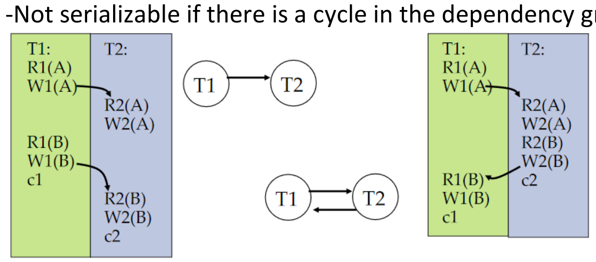
  -Same actions of same committed transactions

-**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

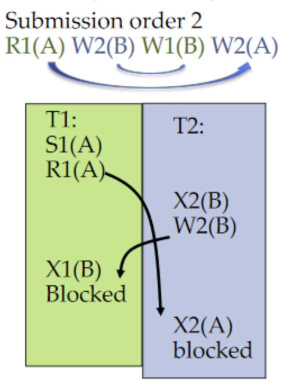
  -Put in in the schedule, delete it and all outgoing edges

  -Repeat until no more nodes remaining

Locking:

-Transactions <u>must</u> 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 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:

	S	X
S	✓	--
X	--	--

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

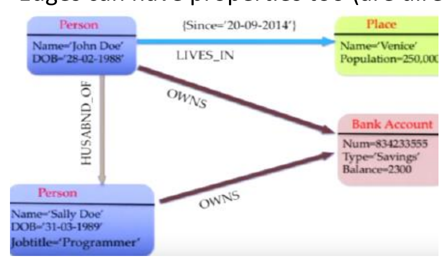
-Read uncommitted = Read operation does not set locks, can read uncommitted writes
-Read Committed = Do not read uncommitted writes. Release read lock immediately after reading.
-Repeatable reads = standard S locking on reads
-Serializable = lock entire relation

Isolation Levels
  TRANSACTION_READ_UNCOMMITTED
  TRANSACTION_READ_COMMITTED
  TRANSACTION_REPEATABLE_READ
  TRANSACTION_SERIALIZABLE

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

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:

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;