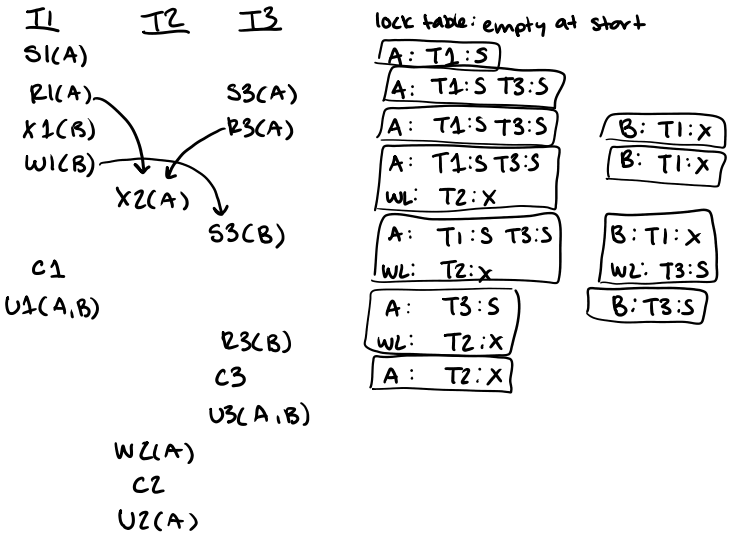
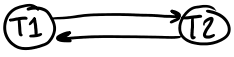
Concurrency control cont.

- 2PL example

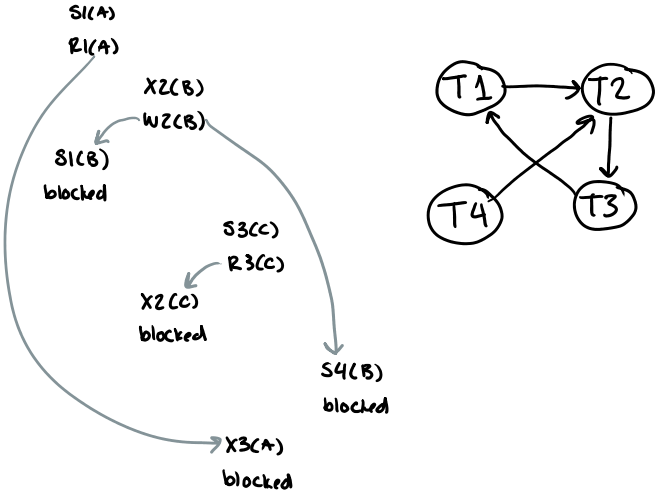
submission order: R1(A) R3(A) W1(B) W2(A) R3(B)



- 2PL details:
  - a transaction does not request the same lock twice
  - a transaction does not need to request an S lock on an object for which it holds an X lock
  - if a transaction has an S lock and needs an X lock, it must wait until all other S locks are released
- Deadlocks: cycle of transactions waiting for locks to be released by each other
  - Wait-for graph: nodes are transactions, there is an edge from T_i to T_j if T_i is waiting for T_j to release a lock
  - detection: cycles in wait for graph
  - e.g. T1: S1(A) R1(A) X1(B) Blocked
    T2: X2(B) W2(B) X2(A) Blocked



  - e.g. I1 I2: I3: I4:



Extra assignment Ex. 1/2:

S1

T1	T2	T3
		r3(a)
	r2(b)	
w1(c)		
	w2(b)	
		r3(c)
	r2(a)	
	c2	
r1(b)		
w1(b)		
c1		
		w3(b)
		c3

S2

T1	T2	T3
r1(a)		
		r3(c)
	w2(b)	
w1(a)		
	r2(c)	
	w2(c)	
	c2	
w1(c)		
c1		
		r3(a)
		c3

S3

T1	T2	T3
r1(a)		
	w2(a)	
		r3(a)
		w3(b)
		c3
	r2(b)	
w1(c)		
	w2(c)	
	c2	
c1		

Distinguish two variations of the locking scheme. (1) Upon a shared lock request $S_i(a)$ of transaction T_i if there are only shared locks active on a, then grant $S_i(a)$, else $S_i(a)$ must wait; (2) Upon a shared lock request $S_i(a)$ of transaction T_i, if there are only shared locks active on a and no lock is waiting on a, then grant $S_i(a)$, else $S_i(a)$ must wait.

S1

T1	T2	T3
		S3(a)
		r3(a)
	S2(b)	
	r2(b)	
X1(c)		
w1(c)		
	X2(b)	
	w2(b)	
		S3(c)
		blocked
	S2(a)	
	r2(a)	
	c2	
	U(a,b)	
S1(b)		
r1(b)		
X1(b)		
w1(b)		
c1		
U(b,c)		
		r3(c)
		X3(b)
		w3(b)
		c3

S2

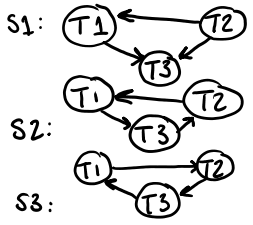
T1	T2	T3
	S1(a)	
	r1(a)	
		S3(c)
		r3(c)
	X2(b)	
	w2(b)	
X1(a)		
w1(a)		
	S2(c)	
	r2(c)	
	X2(c)	
	blocked	
X1(c)		
blocked		
		S3(a)
		deadlock
		abort
		U(c)
	w2(c)	
	c2	
	U(b,c)	
w1(c)		
c2		
U(a,c)		

S3 Variation 1

T1	T2	T3
S1(a)		
r1(a)		
	X2(a)	
		S3(a)
		r3(a)
		X3(b)
		w3(b)
		c3
		U(a,b)
X1(c)		
w1(c)		
c1		
U(a,c)		
	w2(a)	
	S2(b)	
	r2(b)	
	X2(c)	
	w2(c)	
	c2	
	U(a,b,c)	

S3 Variation 2

T1	T2	T3
S1(a)		
r1(a)		
	X2(a)	
		S3(a)
X1(c)		
w1(c)		
c1		
U(a,c)		
	w2(a)	
	S2(b)	
	r2(b)	
	X2(c)	
	w2(c)	
	c2	
	U(a,b,c)	
		r3(a)
		X3(b)
		w3(b)
		c3
		U(a,b)



no cycle ⇒ Serializable
alternative schedule: T2→T1→T3
cycle ⇒ not serializable
cycle ⇒ not serializable

Lost update eg:

```
T1: SELECT grade FROM Enrolled WHERE sid = 4711
T2: UPDATE Enrolled SET grade = grade + 5 WHERE cid = 'COMP-421'
T1: UPDATE Enrolled SET grade = 84 WHERE sid = 4711 AND cid = 'COMP-421'
T1: COMMIT
T2: COMMIT
```

Dirty read eg:

```
T1: UPDATE Enrolled SET grade = grade + 5 WHERE cid = 'MATH-240'
T2: UPDATE Enrolled SET grade = grade + 5 WHERE cid = 'COMP-421'
T3: SELECT cid, AVG(grade) FROM Enrolled GROUP BY cid
T2: ABORT
T3: ABORT
T1: COMMIT
```

unrepeatable read eg:

```
T1: SELECT cid, avg(GRADE) FROM Enrolled GROUP BY cid
T2: UPDATE Enrolled SET grade = grade + 5 WHERE cid = 'COMP-421'
T2: COMMIT
T1: SELECT sid, avg(GRADE) FROM Enrolled GROUP BY sid
T1: COMMIT
```

Upgrades from S to X are preferred over deadlocks:

T1: S1(a) r1(a)
T2: S2(a) r2(a)
T3: X3(a) ← this would cause deadlock

this upgrade gets to go first to avoid DL → X1(a)

Graph Databases

- Nodes/Vertices → entities
- edges → relationships
- properties are key-value pairs associated with a specific node or relationship
- ER vs Graph
  - ↳ nodes = entity set
  - ↳ edges = relationship sets
  - each row is a node/edge
- lots of joins ⇒ use a graph DB
- Schema flexibility: new entity types, properties, relationships ... can emerge w/o significant impact on existing data model
  - ↳ the model does not need to be completely developed ahead
- can accommodate the concept of relationship between entities more efficiently than a relational model
- Cypher – specifies what data to retrive, not how
  - ↳ "Select":
```
MATCH(e:Employee)
RETURN e
```
    - match any node node of type employee, return the entire node
  - ↳ "Project":
```
MATCH(e:Employee)
RETURN e.email, e.ephone
```
    - returns only specific fields
    - ↳ ordering output:
```
MATCH(e:Emplyee)
RETURN e.email,e.ephone
ORDER BY e.email
```
  - ↳ "Select" with a codition:
    - find emp info of janet:
```
MATCH(e:Employee)
WHERE e.ename='Janet'
RETURN e
```
    - search for multiple employees atonce:
```
MATCH(e:Employee)
WHERE e.ename in ['Janet','Steve']
RETURN e
```
    - pattern match names starting with S: `WHERE e.ename =~'S*'`
    - employees w/o a department:
```
MATCH(e:Employee)
WHERE NOT (e)-[:works_in]->()
RETURN e
```
    - employees not mentoring anyone:
```
MATCH(e:Employee)
WHERE NOT (e)-[:mentors]->()
RETURN e
```
    - employees not mentored by anyone:
```
MATCH(e:Employee)
WHERE NOT (e)<-[:mentors]-()
RETURN e
```
    - employees not mentoring or mentored:
```
MATCH(e:Employee)
WHERE NOT (e)-[:mentors]-()
RETURN e
```
  - ↳ NULL: a non existent property is treated as null:
```
MATCH(e:Employee)
WHERE e.job is NULL
RETURN e
```
  - ↳ "Insert":
```
- CREATE(d:Department{dname:'PR',did=12})
      <-[works_in]-(e:Employee{ename:'jane'})
or: CREATE (e:Employee{ename:'jane'})-[works_in]->
      (d:Department{dname:'PR',did=12})
```

- Cypher
  - in cypher, any query can return data
  - ↳ "Insert": add a new relationship between existing nodes:
```
MATCH(n1:Emp{eid='101}),(n2:Emp{eid:201})
CREATE (n1)-[:manages]->(n2);
```
  - ↳ Delete: delete relationships before/while deleting the node
```
MATCH(e:Emp{empid:201})-[r:WORKS_IN]->
(d:Department{deptid:12}) DELETE e,d,r
```
    - or:
```
MATCH(e:Employee{empid:201})
DETACH DELETE e
```
      - ↳ deletes edges as well as node
  - ↳ "Joins"/traversals:
    - find employees working in HR:
```
MATCH(e:Employee) -[w:WORKS_IN]->
(d:Department{dname:"HR"}) RETURN e
```
      - ↳ or:
```
MATCH(e:Employee) -[w:WORKS_IN]->
(d:Department) WHERE d.dname = "HR"
RETURN e
```
    - return info on paul + nodes he has relationships with:
```
MATCH(e:Emp)--(n)where e.ename = 'Paul' return e,n
match(e:Emp)-[]-(n)where e.ename = 'Paul' return e,n
```
    - return info on paul + nodes of outgoing relationships:
```
match(e:Emp)-[r]->(n)where e.name='Paul'return e,r,n
```
    - return names of all employees steve is managing:
```
MATCH(e:Employee) -[:MANAGES]-> (n:Employee)
WHERE e.ename = 'Steve'
RETURN n.ename
```
    - return info of people who are manged or mentored by katie:
```
MATCH(e:Employee) -[:MANAGES|MENTORS]-> (n)
WHERE e.ename = 'Katie' RETURN n
```
    - return info of people neither managed nor mentored by katie:
```
MATCH(e:Employee), (n:Employee)
WHERE e.ename = 'Katie' AND NOT (e) -
[:MANAGES|MENTORS]-> (n) RETURN n;
```
    - return info of people who report to managers managed by Steve
```
MATCH(e:Emp) -[:manages]->() -[:manages]->(n)
WHERE e.ename = 'Steve'
RETURN n
```
      - ↳ or:
```
MATCH(e:Employee) -[:MANAGES*2]->(n)
WHERE e.ename = 'Steve'
RETURN n
```
    - multi depth traversals:
      - ↳ all the way: (e)-[*]->(n)
      - ↳ up to a depth of 5: (e)-[*..5]->(n)
      - ↳ 3+ edges: (e)-[*3..]->(n)
      - ↳ 3-5 edges: (e)-[*3..5]->(n)
      - ↳ 3-5 edges: (e)<-[*3..5]-(n)
      - ↳ 3-5 edges: (e)-[*3..5]-(n)

      } outgoing
      } ingoing
      } either
    - employees under steve who are not managers:
```
MATCH(e:Emp) -[:manages]-> (e2:Emp)
WHERE e.name = 'Steve' AND NOT (e2)-[:manages]->()
RETURN e2;
```

```
MATCH(e:Employee) -[:MANAGES]->() -[:MENTORS]->(n)
WHERE e.ename = 'Steve'
RETURN n
```

```
MATCH(n:Employee)
WHERE NOT (:Employee{ename:'Steve'})
-[:MANAGES]->()-[:MENTORS]->(n) RETURN n
  MATCH(e:Employee), (n:Employee)
WHERE e.ename = 'Steve' AND NOT
(e)-[:MANAGES]->()-[:MENTORS]->(n) RETURN n;
```

```
(Janet)-[:FRIEND_OF]->(Sue)
        (Sue)-[:FRIEND_OF]->(Janet)
  MATCH (e1:Employee(ename:'Janet')
-[:FRIEND_OF*]->(e2:Employee) RETURN e1,e2
```

```
MATCH
(n:Employee{ename:'Janet'})-[:MANAGES]->
(e1:Employee)-[:MANAGES]->(e2:Employee) ,
(n)-[:FRIEND_OF]->(e2) RETURN n,e1, e2
```

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

```
CREATE CONSTRAINT ON (e: Employee)
ASSERT e.eid IS UNIQUE
CREATE CONSTRAINT ON (e: Employee)
ASSERT exists(e.ename)
CREATE CONSTRAINT ON ()-[m:MENTORS]-()
ASSERT exists(m.skill)
```

All the people who has a potential risk for Huntington but not Lebers:

```
MATCH(s:Subject)<-[:motherof|fatherof *]-(:Subject)-
[:has]->(d:Disorder {id:1}),(d2:Disorder {id:2})
WHERE NOT(s)<-[:mohter_of*]-(:Subject)-[:has]->(d2)
RETURN s
```

Create Siblings:

```
MATCH(s1:subject)<-[:fatherof]-(f1:subject), (s1)<-
[:motherof]-(m1:Subject),
(s2:Subject)<-[:fatherof]-(f1), (s2)<-[:motherof]-(m1)
WHERE S1.id < S2.id
CREATE (s1)-[:Sibling_of]->(s2)
```

friends of Sally who went to the same school as her:

```
MATCH(p:Person {name:'Sally'})-[:STUDIED_AT]->
(s:School),(p2:Person)-[:STUDIED_AT]->(s)
      ,(P)-[:FRIEND_OF]-(P2)
RETURN p2;
```

People who went to the same school and graduated the same year, but are not immediate friends:

```
MATCH(p1:Person)-[s1:STUDIED_AT]->(s:School)
            ,(p2:Person )-[s2:STUDIED_AT]->(s)
WHERE s1.grad_year = s2.grad_year AND NOT (p1)-
      [:FRIEND_OF]-(p2)
Return p1;
```

Movies for John that he has not yet liked but is of the same Genre as the other movies that he has liked

```
MATCH(p:Person {name:'John'})-[:LIKED]-(m:Movie)-
      [:BELONGS_TO]->(g:Genre)
      ,(m2: Movie)-[:BELONGS_TO]->(g)
WHERE NOT (p)-[:LIKES]->(m2)
RETURN m2;
```

Create fanclubs:

```
MATCH(b:Band)
WHERE b.name in {'Eagles', 'Enigma'}
CREATE (f1:Fanclub {city:'Montreal'})-
      [:ASSOCIATED_TO]->(b)
      ,(f2:Fanclub {city:'Ottawa'})-
      [:ASSOCIATED_TO]->(b);
MATCH(p:Person)-[:LIKES]->(b:Band), (f:Fanclub)-
      [:ASSOCIATED_TO]->(b)
WHERE p.city = f.city
CREATE (p)-[:MEMBER_OF]->(f);
```

Assignment 3

- A map-reduce workflow that takes a person and their friends, (P, (F1, F2, ...)) and gives common friends for each set of friends.
  - mapper: input (P, list of friends)
    - For each friend F in list of friends
      - If (P<F) output ((P,F), (list of friends)-F)
      - else output ((F,P), (list of friends)-F)
  - reducer: input is a pair of friends and corresponding list of friends: ((P1,P2),((list1),(list2)))
    - output ((P1,P2), common friends intersection)
- A map-reduce workflow that produces provinces w/ over 100 patients over 60 who are in the hospital and the total number of patients. [Hospital (Hname, province) Patient (HInsnum, age, Hname) Hname ref hospital]
  - mapper: For each tuple (HInsnum, age, Hname) of Patient
    - if (age > 60) (output (Hname, P))
    - For each tuple (Hname, province) of Hospital
      - output (Hname, CH, province))
  - reducer: (input (Hname, (P, ..., P), (H, province))
    - numpatients = #P
    - output (province from H tuple, numpatients)
  - mapper2: input (province, numpatients)
    - output identity : (province, numpatients)
  - reducer2: input (province, (numpatients1, ..., numpatientsn))
    - totalpatients = sum all numpatients
    - if (totalpatients > 100)
      - output (province, total patients)
- Q6: 10 movies w/ max # user ratings

```
Ratingsgrp = GROUP ratings BY movieid;
ratingspermovie = FOREACH ratingsgrp GENERATE group
    AS movieid, COUNT($1) AS numratings;
Ratingspermovieord = ORDER ratingspermovie
    BY numratings DESC;
top10movies = LIMIT ratinspermovieord 10;
top10moviesinfo = JOIN top10movies BY movieid,
    movies BY movieid;
Topmovies = FOREACH top10moviesinfo GENERATE title
    AS title, numratings AS numratings;
Topmoviesord = ORDER topmovies BY numratings DESC;
DUMP topmoviesord;
```

- #movies per genre 2015-2016

```
Movies20156 = FILTER movies BY year==2015 OR
    year==2016;
Moviegenres20156 = JOIN movies20156 BY movieid,
    Moviegenres BY movieid;
Geresandmovies = GROUP moviegenres20156 BY
    (genre,year);
Genremoviecount = FOREACH genresandmovies
    GENERATE FLATTEN(group), COUNT($1)
    AS nummovies;
Orderedgenres = ORDER genremoviecount BY genre, year;
DUMP orderedgenres
```

- years in which #movies were less than the previous yr

```
Moviesperyear = GROUP movies BY year
Yearcount = FOREACH moviesperyear GENERATE
    Group AS year, COUNT($1) AS nummovies;
Yearcount2 = FOREACH yearcount GENERATE
    Year+1 AS curryear, nummovies;
Joinyears = JOIN yearcount BY year, yearcount2
    BY curryear;
Moviecounts = FOREACH joinyears GENERATE
    Yearcount2::curryear, yearcount::nummovies
    AS currnummovies, yearcount2::nummovies
    AS prevnummovies;
Badyears = FILTER moviecounts
    BY currnummovies < prevnummovies;
DUMP badyears;
Orderbadyears = ORDER badyears BY curryear;
DUMP orderbadyears;
```

Quiz 5 - Trans. and CC.

It is safe for a program to crash without leaving inconsistent data in the DB if there are no changes to the DB or all changes are complete.

- crashes and aborts should not be present in a system after recovery.
- Before the DB sends a commit confirmation to the app it must record the commit from the transaction in its log, flush the log buffers (log tail) to the disk, and the app must initiate the transaction commit.
- Data buffers are flushed when the memory buffer pool needs to make room for new pages.
- A transaction T updates X then aborts, then the DBS crashes:
  - ↳ It is possible in some situations that it is not necessary for the DBS during the recovery to apply the before image of X.
  - ↳ writes made by T may still be in memory buffers, not disk. so it's like the update never happened which is good
- T updates X and commits, then the DBS crashes:
  - ↳ It is possible in some situations that it is not necessary for the DBS during recovery to apply the after image for X
  - ↳ the changes may already be updated in disk so no redo would be necessary.
- conflict equivalent schedules:
  - w2(b),r1(a),w1(a),r3(a),r2(a),r1(b),r3(a),c3,c2,c1
  - r1(a),w1(a),w2(b),r3(a),r2(a),r1(b),r3(a),c1,c2,c3
  - r1(a),w1(a),r3(a),r3(a),c3,w2(b),r1(b),r2(a),c1,c2,c3
- conflict equivalent serializable schedule
  - r3(a),w2(b),r1(b),w2(c),r4(c),r2(a),r1(b),w3(b),r4(b),c2,c1,c4,c3 => T2,T1,T3,T4 (flatten conflict graph)
- submission order → 2PL schedule
  - r2(a),w1(b),r1(c),r3(c),w2(c),c3,w2(b),c2,r1(a),c1
  - (r2(a),w1(b),r1(c),r3(c),c3,r1(a),c1,w2(c),w2(b),c2)

continue next page...

Quiz 5 - cont.

	T1	T2	T3
1	A=read1(A)		
2	A=A+100		
3			C=read3(C)
4			C=C-300
5	C=read1(C)		
6		A=read2(A)	
7			write3(C)
8			commit3
9	C=C-100		
10	write1(A)		
11	write1(C)		
12		B=read2(B)	
13		B=B-200	
14		write2(B)	
15	commit1		
16		A=read2(A)	
17		C=read2(C)	
18		A=A+200	
19		write2(A)	
20		commit2	

↳ lost update: write3(C)
un-repeatable read: read2(A)
no dirty reads
no dirty writes

Practice final

- we can enforce the constraint that rating can only be null or 1-10.
- If we create an index on psid of parent, a type II indirect index will not necessarily take less space than type I.
- A search for a sitter by a range on hourly rate and a >= condition on minpreffered age would not benefit from clustering the table on minpreffered age.
- The efficiency of a page nested loop join cannot be increased by adding more buffer frames (assuming the total memory buffer frames are still less than the total number of pages that either relation has.
- The height of a B+ tree index is dependent on the size of the data entries.
- Adding an extra attribute that is not part of the search key of an index will not reduce the number of page pointer entries that can fit into an intermediate node of an index structure.
- In a graph database, a node/vertex is similar to an entity in ER

SQL:

- name + # of sitters who have an entry in sitterCalendar for may - August 2017:

```
SELECT sname, sphone
FROM sitters
WHERE ssid IN (SELECT ssid FROM sitterCalendar WHERE
      availDate BETWEEN '2017-05-01' AND '2017-08-31')
```

- ssid, sname, sphone of sitters w/ at least 5 ratings >7:

```
SELECT ssid, sname, sphone
FROM sitters s, babySitting b
WHERE s.ssid =b.ssid AND b.rating >7
GROUP BY s.ssid, s.sname, s.sphone
HAVING COUNT(*)>=5
```

- psid of parents w/ more than one child, at least one is female:

```
SELECT psid FROM child WHERE psid IN
      (SELECT psid FROM child WHERE cgender='female')
GROUP BY psid HAVING COUNT(*) > 1
```

Cypher:

- find preferred sitters of people in melanies network

```
MATCH(p1:Parent {name='melanie'}) -[:knows*]->
      (p:parent), (p)-[:prefers}->(b:babysitter)
RETURN b
```

- add a knows rel between parents and preffered sitters

```
MATCH(p:parent)-[:prefers]->(b:babysitter)
CREATE (p)-[:knows]->(b), (b)-[:knows]->(p)
```

Pig:

- given a psid, get info on female sitters booke atleast 10x

```
femaleSitters = filter sitter by gender = 'female';
parentsBabySitting = filter babySitting by psid = 123456789;
jnd = join femaleSitters by ssid, parentsBabySitting by ssid;
grpd = group jnd by (ssid, name, phone);
smmd = foreach grpd generate($0), COUNT($1) as
bookingCount;
fltrsmmd = filter smmd by bookingCount >= 10;
srtd = order fltrsmmd by name;
dump srtd;
```

- Map reduce:
- count of # pages referencing a URL

```
Map: for each tuple (referencingURL, referencedURL_LIST)
      for each referencedURL in referencedURL_LIST
            If referencedURL != referencingURL
                  Output (referencedURL, referencingURL)
```

Group and shuffle will take in each (referencedURL, referencingURL) and output tuples (referencedURL, referencingURL_LIST)

```
Reduce: for each tuple(referencedURL,referencingURL_LIST)
            Aref= {}, counter  =0
            For each referencingURL in referencingURL_LIST
                  If referecnignURL not in aref
                        Add referecning url to aref
                        Counter++
            Ouput (referencedURL, counter)
```

- produce weighted output of previous result

```
Map: For each tuple(referencedURL, referenceCount)
      output(referencedURL, ('RefCounts',referenceCount)
      For each tuple(URL, popValue) of popularity
      output (URL, ('popularity', popValue))
Reduce: For each tuple (URL, tupleList)
            refTuple = NULL
            popTuple = NULL
            For each tuple t in tupleList
                  If t.rel == 'RefCounts'
                        refTuple = t;
                  Else If t.rel == 'popularity'
                        popTuple = t;
                  If popTuple == NULL
                      ouput(URL, refTuple.referenceCount)
                  Else ouput(URL, refTuple.referenceCount*
                  popTuple.popValue)
```

Practice final cont

Query evaluation:

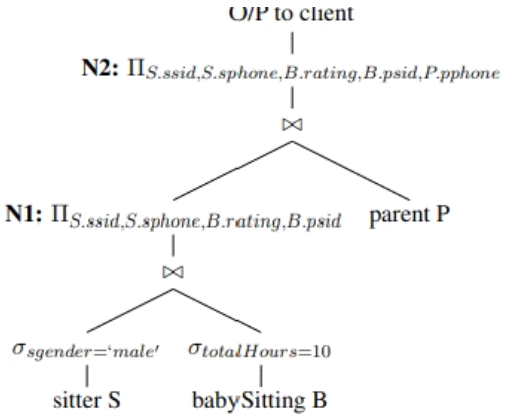
```
sitter(ssid:INT ,sname:VARCHAR(40) ,sphone:CHAR(12) ,sgender:CHAR(6)
,hourlyRate:INT ,preferredMinAge:INT)
parent(psid):INT, pname:VARCHAR(40), pphone:CHAR(12))
babySitting(reservationId:INT, ssid:INT ,availDate:DATE, psid:INT
,cname:VARCHAR(40) ,starthr:INT, totalhrs:INT, addnlChildren:INT, rating:INT)
⟶ (ssid ,availdate) references sitterCalendar
⟶ (psid ,cname) references child
```

INT takes 4 Bytes, FLOAT takes 8 bytes, all VARCHAR fields take half of the size specified. DATE field is 10 Bytes
All pages are 4K size (you can use 4000 for calculations)

`sitter` has 1,000 records spread across 20 data pages. 20% of the babysitters are *male*
`parent` has 20,000 records.
`babySitting` has 1,000,000 records spread across 20,000 pages. `availDate` has 10,000 distinct values. `totalhrs` have values in the range 1-20.

You can make the following assumptions. There are indexes on all primary keys (unclustered). Root and intermediate pages of B+-tree indices are always in main memory. No other page is assumed to be in main memory at the begin of a query execution. There are around 10 buffer frames available. (you can assume another couple of frames are available if it helps your computation to be easier)

```
SELECT s.ssid, s.sname,
       SUM(s.hourlyRate*b.totalhrs*(1+0.50*addnlChildren))
FROM sitter s, babySitting b
WHERE s.ssid = b.ssid
  AND b.availDate BETWEEN '2017-03-01' AND '2017-03-05'
GROUP BY s.ssid, s.sname
```



3. N1 processes about 10,000 records (20% of the 50,000 qualified from babySitting), both entering and leaving. Incoming tuple size is either 108B or 104B, leaving tuple size is 24B

   N2 processes about 10,000 records too (because only half of the parents will be needed), both entering and leaving. Incoming tuple size is either 60B or 56B and outgoing tuple size is 36B

↳process + I/O costs:

Steps

1. First read the 20,000 pages of babySitting, use the selection on availdate and pipeline it to projection and store only the required fields (ssid, totalhrs, addnlChildren) of the selected records in the memory buffers. The width of the output tuple is 4+4+4 = 12, and can be done into 2 frames ⇒ temp1. The number of records in temp1 will be 1,000,000 x 5/10,000 = 500 . Total IO is 20,000 reads.
2. Block Nested with temp1 as outer and sitter as the inner relation. Temp1 is in memory so join read cost = 20 (for sitter). Compute the amount earned for this babysitting reservation save only required fields (ssid, sname, amt), width = 4+20+8 = 32 … takes about 4 frames (no need to write can be held in memory). Total IO = 20 reads
3. Do an in memory sorting based on ssid, sname. This is then grouped (in memory) - pipelined to sum - and pipeline to project - and send to client directly.

Total IO is therefore only 20,000 + 20 = 20,020

decrease I/O with an index:

Create a clustered index on availDate of babySitting
For selection operator on availDate use this index and , read 1 leaf page + 10 data pages , project and store only required fields in memory (ssid, totalhrs, addnlChildren) (2 frames) -> temp1 = 11 reads

Rest of the steps same as before - results in 11 + 20 = 31 I/O

Lost update: read-write-write pattern

↳eg r1 → w2 → w1
or r2 → w1 → w2