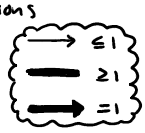
ER Model

- overlap constraint: can an entity be in more than one subclass?
- covering constraint: Must every entity of the superclass be in one of the subclasses?
- Key constraints
  - ↳ one-many ⟶ thin arrow
  - ↳ one-one ⟶ ⟵ key constraints in both directions
- Participation constraints
  - ↳ thick line, at least one
- combined key + participation: exactly once



Relational Model

- SQL
  - ↳ Create table X ( )
  - ↳ Insert into X Values ( )
  - ↳ Delete from X where ...
  - ↳ Update X set ... where ...
  - ↳ Select ... from X where ...
  - ↳ Drop Table X
  - ↳ Alter Table X ADD column ...
- Integrity Constraints
  - ↳ need to always be true
  - ↳ NOT NULL
- Primary key constraints
  - ↳ need a unique row identifier
- referential integrity
  - ↳ all foreign key constraints are enforced
  - ↳ foreign key (⟶) references X

ER-Relational Translation

Eg: DB2: CREATE TABLE Companies
(name VARCHAR(30),
Addr VARCHAR(50),
Empl INTEGER,
PRIMARY KEY (name))

⟶ Companies(<u>name</u>, address, empl)

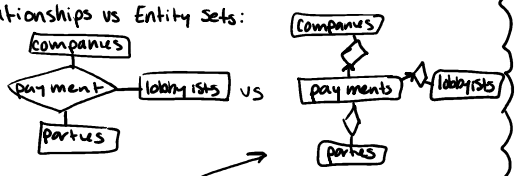
- ↳ many-many relationship translated to a table
- ↳ participation constraints can't usually be reflected besides NOT NULL
- ↳ weak entity set and identifying relationship are translated into a single table
- ↳ ISA Hierarchies: distribute info among relations OR subclasses have all attributes OR one big table with null values

Relational Algebra

- closed → can be composed and concatenated
- relations are sets (no two rows are identical)
- selection: σ (get rows)
- projection: Π (get cols, eliminate dups)
- renaming: ρ
- set difference: − (in first but not second)
- cross product: $\times$ (all w/ all)
- join: $\bowtie$ ($\times$ and σ)
- intersection: $\cap$
- Union: $\cup$
- $\sigma_{condition}(relation)$, $R_1 \bowtie_C R_2$
- e.g. all makers that don't make laptops: $\Pi_{maker}(computer) - \Pi_{maker}(\sigma_{category="laptop"}(computer))$
- e.g. names of skaters who participated in local or regional comps:
  $\rho(Temp, \sigma_{type=local \vee type=regional}(C) \bowtie P)$, $\Pi_{name}(Temp \bowtie S)$
- e.g. names of skaters who participated in local or regional comps:
  $\rho(locals, \Pi_{sid}(\sigma_{type=local}(C) \bowtie P))$, $\rho(Regionals, \Pi_{sid}(\sigma_{type=regional}(C) \bowtie P))$, $locals \cap regional$
- rules: $\Pi_L(\sigma_C(R)) = \sigma_C(\Pi_L(R))$ (if C considers attributes of L, $R1 \bowtie R2 = R2 \bowtie R1$,

Quiz 1: ER

- relationships vs Entity Sets:



payment as entity set allows several payments of the same company to the same party by same lobbyist (more flexibility) → can't really track a relation directly. Relationships can only have one instance amongst the exact same entities.

- Weak entity → when introducing an artificial key we should <u>not</u> make it a partial key – e.g. no weak entity

Quiz 2: Relational

- In a relational model: ① all rows stored must be distinct ② given a query, a database system can implement various ways to retrive results ③ Query languages built for relational models are data-centric
- candidate keys: all primary keys are candidate keys. If the combination of attributes (a1, a2) is a candidate key, then it's possible that 2 records have the same value for a1.
  - ↳ candidate keys have to be the <u>minimal</u> subset. e.g. if a1 is a candidate key alone, (a1, a2) is not one.
- foreign keys: a relation can have more than one foreign key to the same relation. Foreign keys must refer to something that exists. A foreign key <u>by itself</u> can also be a primary key but a foreign key cannot, by itself, be a primary key.
- can only reflect the NOT NULL participation constraint. (thick arrow) A one-one key constraint implies you can merge the entity and relation.
- In the relational translation of a weak entity set, the foreign key is strictly to the relation that it is immediately dependent on.
- "at most one" constraint (⟶) means that entity should be the primary key of relationship set table.
- many-many needs at least 2 tables

Quiz 3: Relational algebra

- In the union of 2 relations, the output may have same num rows as the input relations if the inputs had identical rows.
- selection and projection are commutative if the attributes in the selection are a subset of the attributes of the proj.
- be careful that projections don't eliminate necessary attributes
- non-natural join needs attribute to join on specified
- can't natural join with no common attribute.

$\Pi_{L2}(\Pi_{L1}(R)) = \Pi_{L2}(R)$ IF $L2 \subseteq L1$, $\sigma_{C2}(\sigma_{C1}(R)) = \sigma_{C1 \wedge C2}(R)$, $R1 \bowtie (R2 \bowtie R3) = (R1 \bowtie R2) \bowtie R3$

SQL

- σ ⇔ WHERE, Π ⇔ SELECT
- no elimination of duplicates (unless DISTINCT)
- Join, always indicate condition
- self join e.g.: (avoid the same pair)
```
SELECT p1.sid, p2.sid
FROM Players p1, Players p2
WHERE p1.cid=p2.cid AND p1.sid<p2.sid
```
- union (or) / intersect (and) / join need set-compatible relations
  - ↳ provide duplicate elimination (use UNION ALL to avoid)
- Nested Queries (IN, or NOT IN)
  - ↳ e.g. names of skaters who participated in Comp101

```
SELECT sname
FROM Skaters
WHERE sid IN (SELECT sid
              FROM Participates
              WHERE cid=101)
```

!= ↓ <>

- EXISTS - true iff the nested relation is not empty
- ANY, ALL - e.g. skaters with highest rating:

```
SELECT *
FROM Skaters
WHERE >= ALL (SELECT rating
              FROM Skaters)
```

- e.g. Names of skaters who participated in all competitions:

```
SELECT sname
FROM Skaters s
WHERE NOT EXISTS ( (SELECT c.cid
                    FROM Competition c)
                   EXCEPT
                   (SELECT p.cid
                    FROM Participates p
                    WHERE p.sid=s.sid) )
```

- COUNT, SUM, AVG, MAX, MIN → apply to single attribute/column
  - ↳ e.g. Names of skaters w/ highest rankings

```
SELECT sname
FROM skaters
WHERE rating = (
      SELECT MAX(rating)
      FROM skaters)
```

- If any aggregation is used then each element in the attribute list of the Select clause must be aggregated or appear in a group-by
- HAVING: eg- for each rating, find the min age, only consider rating levels w/ at least 2 skaters:

```
SELECT rating, MIN(age)
FROM skaters
GROUP BY rating
HAVING COUNT(*) >= 2
```

- Aggregate operations cannot be nested
  - ↳ View: CREATE VIEW name (a1,a2) AS SELECT ...
    - ↳ like an intermediate relation
- COUNT(*) counts NULLs
- Nulls → unknown logic value
  - ↳ NOT(U) = U, U AND T = U, U AND F = F, U AND U = U
    U OR T = T, U OR F = U, U OR U = U

- Inner Join (default)
  - ↳ no match in other relation ⇒ no output
- outer join
  - ↳ no match in other relation ⇒ dummy record (NULLs)
  - ↳ LEFT ⇒ every tuple in the left relation appears in the output no matter what
  - ↳ RIGHT ⇒ pad dangling tuples from right relation
  - ↳ FULL ⇒ pad dangling tuples from right and left relations
- If B has NOT NULL foreign key referencing A
  - ↳ A INNER JOIN B = A RIGHT OUTER JOIN B
- WITH (like a view for the scope of the statement

```
WITH partinfo(sid, sname, age, rank, cid) AS (
      SELECT S.sid, S.sname, S.age, P.rank, P.cid
      FROM Skaters S INNER JOIN participates P
      ON S.sid, P.pid
) SELECT sid, name, cid
FROM partinfo
WHERE age >7;
WITH A(...) AS
      (SELECT ... )
      ,B(...) AS
      (SELECT ...)
SELECT ...;
```

- Derived Table: temp table def within FROM

```
SELECT sid, sname, cid
FROM( SELECT s.sid, s.sname, s.age, p.rank, p.cid
      FROM skaters s INNER JOIN participates p
      ON s.sid=p.sid
)partinfo
WHERE age > 7;
```

- Insertion/Deletion:

```
INSERT INTO skaters VALUES (68, 'Jacky', 10, 10);
INSERT INTO skaters (sid,name) VALUES (68, 'Jacky');
INSERT INTO activeSkaters (
      SELECT skaters.sid, skaters.name
      FROM skaters, participates
      WHERE skaters.sid=participates.sid);
DELETE FROM competitions WHERE cid=103;
DELETE FROM competitions;
UPDATE skaters
SET ranking =10, age=age+1
WHERE name='debby' OR name='lily';
```

- Integrity Constraints – every instance of a relation must satisfy. → Checks: in create statement
  - ↳ check(constraint)
  - ↳ checks only happen with insert/update, not delete
  - ↳ ALTER TABLE matches
    ADD CONSTRAINT check_dates CHECK (
    mDate >= '2023-07-20' AND mDate <= '2023-08-20');

e.g.

```
CREATE VIEW PlayerInfo (name, number, natassoc)
AS SELECT p.name, p.number, t.natassoc
FROM players p, teams t
WHERE p.country=t.country;
```

SQL examples from P2 & A1

- Stadium names + locations where Christine Sinclair has scored ≥ 1 goal

```
FROM( SELECT matchId
        FROM goals
        WHERE playerId = ( SELECT playerId
                FROM players
                WHERE name ='Christine Sinclair')
        GROUP BY matchId
        HAVING count(*) >=1)goalcounts, matches m, stadiums s
WHERE goalcounts.matchId=m.matchId AND m.sName = s.sName;
```

- Info of players who played in all matches

```
SELECT p.name, p.number, p.country
FROM( SELECT playerId, COUNT(*) pcount
        FROM plays_in
        GROUP BY playerId)pc,
        (SELECT country, COUNT(*) tcount
        FROM team_plays
        GROUP BY country)tp, players p
WHERE p.playerId=pc.playerId AND p.country=tp.country AND
pc.pcount=tp.tcount;
```

- Country, # matches, # goals scored per team

```
SELECT t.country, COUNT(DISTINCT t.matchid) num_matches,
        COUNT(t.matchid) num_goals
FROM team_plays t LEFT OUTER JOIN goals g
        ON g.matchid = t.matchid AND t.country = g.country AND g.penalty
        = FALSE
GROUP BY t.country;
```

- total tickets and sold tickets per match

```
SELECT m.matchid, m.sname stadium, COUNT(t.ticketid) total_tickets,
          (SELECT COUNT(ticketid)
        FROM tickets
        WHERE tickets.matchid = m.matchid AND sold = TRUE) AS
        sold_tickets
FROM matches m LEFT JOIN tickets t ON m.matchid = t.matchid
GROUP BY m.matchid, m.sname
ORDER BY m.matchid;
```

- match, minute, scoring player and country for all goals scored in second half

```
SELECT g.matchid, g.minute, p.name, p.country
FROM goals g, players p
WHERE g.playerid = p.playerid AND g.minute >= '00:30:00' AND g.penalty
= FALSE
ORDER BY g.matchid;
```

- Topic Id and name of topics used by all URLs

```
SELECT topicid, name
FROM topics t
WHERE NOT EXISTS ((SELECT w.url
        FROM webpages w)
EXCEPT (SELECTl.url
         FROM links l
         WHERE l.topicid=t.topicid))
ORDER BY topicid;
```

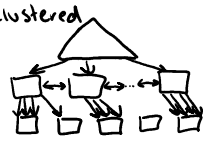
Note on views: DB does not allow insertion into complex views that have join conditions on the view definition

Buffers

- DBMS stores info persistently on disks
- Pin count = # upper layer programs using page
- record ID : rid

Indexing

- helpful for equality and range queries on search key
- B+ Tree: each node = one page
  - ↳ leaf contain data entries - one data entry = pointer to one tuple
  - ↳ assume leaf and data pages are not in main mem
  - ↳ inserting data - if not enough space, split node and redistribute (50%)
- Indirect Indexing I : $k^* = <k, \text{rid of data record w/ search key } k>$
  - ↳ gives several entries w/ same search key side by side (more space)
- Indirect Indexing II : <k, list of rids of data record w/ search key k>
  - ↳ gives a list for each search key
  - ↳ variable length data entries, can have large entries that span a page
- Direct Indexing: leaf pages contain data entries instead of pointers
  - ↳ <k, full record>
  - ↳ leaves represent sorted file of data records
- Clustered Index – relation in file sorted by the search key attributes of the index.
  - ↳ direct index is clustered
  - ↳ indirect clustered:



- Unclustered Index – relation in heap file or sorted by an attribute other than the search key
  - ↳ indirect unclustered:

- Primary Index : search key contains primary key
- Secondary Index : search key is not unique
- Unique Index : search key is primary key or unique attribute

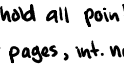
B+ tree cost example:

Given: R(int, int, char, char, char, char) (int = 8)
- int = 4 bytes, char[40] = 40 bytes
- Vals of B ~ Uniform across [0;19999]
- 200,000 tuples
- data page = 4000 bytes ↳ each 75% full
- rid = 10 bytes, ptr = 8
- indirect index II

CALCULATE: # Data pages in the heap file: - tuple size = 168,
- # tuples/page = $4000 \cdot 0.75 / 168 \approx 18$
- # pages = $200{,}000 / 18 \approx$ 11,111 pages

↳ # Leaf pages: # data entries: distinct vals of B = 20,000
- # records w/ same val / # rids in data entry: $200{,}000 / 20{,}000 = 10$ (records / val of B)
- size of data entry: size(B) + 10 rids · size(rid) = $4 + 10 \cdot 10 = 104$ (bytes / entry)
- # data entries per page: $4000 \cdot 0.75 / 104 \approx 29$
- # leaf pages: $20{,}000 / 29 =$ 690 leaf pages

↳ height of the tree: · size of index entry in intermediate node = 4+8 = 12 bytes
- max num of index entry per intermediate page: $4000 / 12 = 333$
  - ↳ single root can't hold all pointers ⇒ height > 1
- 690 pointers to leaf pages, int. nodes 75% full ⇒ 3 intermediate nods
- height = 2 (see pic ↗)



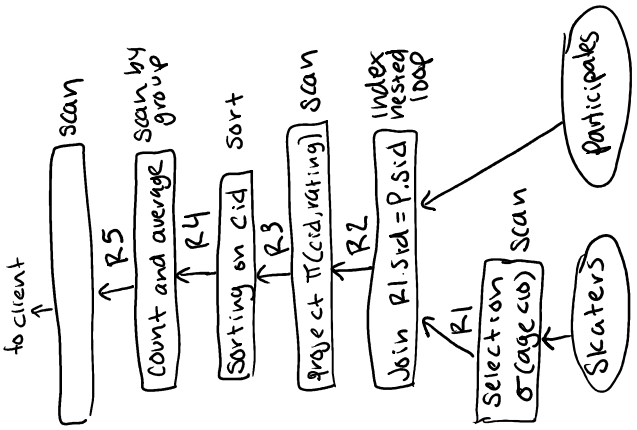
- DB2 : CREATE INDEX ind1 ON Students (startyear);
  DROP INDEX ind1;
  - ↳ referential integrity / uniqueness
    CREATE UNIQUE INDEX ind1 ON Students(email)
  - ↳ additional attributes:
    CREATE UNIQUE INDEX ind1 ON Students(sid)
    INCLUDE (name)
    Index on sid, entry contains sid, name, rid (doesn't need to access real data pages)
    SELECT name FROM Students WHERE sid = 100
  - ↳ clustered index:
    CREATE INDEX ind1 ON Students (name) CLUSTER
  - ↳ multi-attribute index:
    CREATE INDEX ind2 ON Students(email,sid)

Quiz 4 - Buff. Mgmt and Idx:

- If a frame is dirty, it means the content of the page that resides in the frame is different to the content of the page as stored on stable storage
- pin counter is dictated by how many requests are using a particular page as multiple requests could be accessing the same page.
- page = 4000 bytes, 2 bytes per pointer/trackers. records = 40 bytes, no empty slots
  - ↳ Directory: $100 \cdot 2 + 2 \cdot 2 = 204$ bytes ⇒ space = 4000 − 204 = 3796
    $3796 / 40 = 94$ records ⇒ $94 * 40 + 94 * 2 + 2 * 2 = 3950$ ⇒ space for +1
    ⇒ $95 \cdot 40 + 95 \cdot 2 + 2 \cdot 2 = 3994$ (95 records max per page)
- leaf node = 100 data entries, interm. node = 140 index entries
  - ↳ find max # records indexed by tree of height 2 : root holds 140, each interm holds 140, each leaf holds 100 ⇒ $140 \cdot 140 \cdot 100 =$ 1960000
- R(A,B,C) (each size = 4). Index on A w/ addl attribute B. page = 4000, rid = 10, ptr = 8
  - ↳ find # index entries per node (tree 75% full)
    $8 + 4 = 12$ bytes per index, $4000 \cdot 0.75 = 3000$, $3000 / 12 =$ 250
- Indirect Index II on A (size = 10). each val of A occurs 4 times. page = 4000, rid = 10, ptr = 8
  - ↳ find # data entries per leaf page (75% full)
    $4000 \cdot 0.75 = 3000$, size(entry) = 10+4·10 = 50, $3000/50 =$ 60
- Indirect Index II on A, B, A = 4, B = 6, 100000 records, 5 records per (A,B)
  - ↳ find # leaf pages: size(data entry) = 10+5·10 = 60, $4000 \cdot 0.75 / 60 = 50$ data entries
    $100000 / 5 = 20000$ data entries total. $20000 / 50 =$ 400 pages
- A multi-attribute B+ index on attributes A,B of a relation always has a larger space requirement than an index on A with additional attribute B
- You can have both an indirect, clustered index on A of relation R and an indirect non-clustered index on Attributes A,B of R.

- multi-attribute index - order matters → ordered into groups by one then ordered within groups by other. Supports queries on first attribute or both but is not useful for only second attribute (→ full scan)

Query Evaluation

- assume root and intermediate nodes are in main mem
- execution tree: leaf nodes are the relations, inner nodes are operators
  - ↳ e.g. 

Schema for examples:

Users(U): CARD(U) = 40,000 tuples
- 80 $\frac{\text{tuples}}{\text{data page}}$, 500 data pages
- nc index on uID has 170 leaf pages
- nc index on uname has 300 leaf pages

Group members (GM): card(GM) = 100,000
- 100 $\frac{\text{tuples}}{\text{data page}}$, 1,000 data pages, nc index on gid (500 diff groups)

- Selection
  - reduction factor: $\text{Red}(\sigma_{\text{condition}}(R)) = |\sigma_{\text{condition}}(R)| / |R|$
  - e.g. (assume 10,000 w/ exp 5) $\text{Red}(\sigma_{\text{exp}=5}(U)) = 10{,}000/40{,}000 = 0.25$
  - unknown # tuples for condition ⇒ assume uniform + independent distribution
    - ↳ e.g. $\text{Red}(\sigma_{\text{exp}=5}(U)) = \frac{1}{10} = 0.1$
  - red = size of selected range / total size of domain
  - e.g. $\text{Red}(\sigma_{\text{exp}=5 \text{ and age}<=16}(U)) = \text{Red}(\sigma_{\text{exp}=5}(U)) * \text{Red}(\sigma_{\text{age}<=16}(U))$
  - Size of result of selection = (# input tuples)(reduction factor)
    - ↳ eg. rf = 0.1, 40,000 tuples ⇒ |result| = 40,000 · 0.1 = 4000 tuples
  - Simple selections:
    - ↳ no index: search on arbitrary attribute: scan relation (cost = # data pages)
      - ↳ search on primary key attribute: Scan half the relation (cost = $\frac{\#\text{data pages}}{2}$)
    - ↳ index on selection attributes: cost = # leaf pages w/ matches + num data pages w/ match
      - ↳ clustered B+ Tree: cost = 1 leaf page + (# matching tuples)/(tuples per page)
        - e.g. SELECT * FROM Users WHERE uid=123
          cost = 1 leaf page + 1 data page
        - e.g. SELECT * FROM Users WHERE uname LIKE 'B%'
          cost = 1 leaf page + 2 data pages (100 tuples match)
        - e.g. SELECT * FROM Users WHERE uname<'F'
          cost = 1 leaf page + 125 data pages (10,000 tuples match)
      - ↳ non-clustered B+ Tree: worst case: # data pages = # matching tuples
        - e.g. SELECT * FROM Users WHERE uid=123
          cost = 1 leaf page + 1 data page (point query)
        - e.g. SELECT * FROM Users WHERE uname LIKE 'B%'
          cost = 1-2 leaf pages + 100 data pages (loc data not ordered, still better than scan)
        - e.g. SELECT * FROM Users WHERE uname<'F'
          cost = 75 leaf pages + 10,000 pages (worse than scan)
        - non-clustered only useful with very small reduction factors
        - if all desired leaf pages fit in mem then can sort to speed up
  - Selections on multiple attributes
    - ↳ e.g. A = 100 AND B = 50 (300 pages)
      - no index ⇒ scan, cost = 500
      - Index on A ⇒ cost = query for A = 100 (get A, check value of B)
      - one index per attribute ⇒ Intersection based: find rids w/ A=100 (A index), find rids w/ B = 50 (index B), intersect rids. (good if all fit into mem)
        - ↳ use one index: use index on A if A is unique or has small rf
      - one index with both attributes = good
    - ↳ eg. A = 100 OR B = 50
      - no index ⇒ Scan, cost = 500 · 2 indexes ⇒ union of rids

- Sorting: N = # pages, B = # blocks
  - ↳ # passes = $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$
  - ↳ cost = 2N(# passes)
- Joins
  - cardinality estimations
    - ↳ $|U \bowtie GM| = |GM|$
      - Join attribute is a primary key for U, foreign key for GM ⇒ each GM matches with one U
    - ↳ $|R1 \times R2| = |R1| * |R2|$
    - ↳ Joins with conditions → reduction factor of condition * cardinality of join
  - Simple Nested Loop Join: for each tuple in the outer relation, scan the entire inner relation
    - ↳ cost = # outer pages + |outer| * # inner pages
  - page nested loop join: for each page P_o of the outer relation, get each page P_i of the inner relation
    - ↳ cost = # outer pages + (# outer pages)(# inner pages)
  - block nested loop join: For each block of pages bp_o of the outer relation, get each page P_i of the inner relation
    - ↳ bp_o and one P_i must fit in main memory
    - ↳ cost = # outer pages + # outer pages / $|bp_o|$ (# pages size) * # inner pages
  - Index nested loop Join: for each tuple in the outer relation, find the matching tuples of the inner relation through an index (inner must have an index on join att)
    - ↳ cost = # Outer pages + |outer relation| * cost finding matching
  - Block vs Index Nested Loop
    - best case for block: cost = # outer pages + # inner pages
      - ↳ index better - # inner pages > |outer| * cost of matching
  - Sort-merge join: sort relations on join column, then scan to merge. Outer is scanned once, each inner is scanned once per matching outer
    - ↳ if relations are already sorted:
      - cost = # Outer pages + # inner pages
    - ↳ if relations need to be sorted:
      - cost = (2(num passes)+1)(# outer pages) + (2(num passes)+1)(# inner pages)
    - ↳ sort by pipelining:
      - cost = (2(num passes - 1) + 1)(# outer pages) + (2(num passes - 1)+1)(# inner pages)

- no index ⇒ Scan, cost = 500
- Index on A ⇒ Not useful
- 2 indexes ⇒ Union of rids
- 1 index w/ both ⇒ may or may not be faster

Query Evaluation cont.

- Projections: done on the fly with another operation
- Union: sort → scan → merge
- Aggregate operations: usually done last
  - ↳ without grouping ⇒ scan
  - ↳ with grouping → sort on group-by attribute then scan relation and compute aggregate for each group
- Pipelining: as soon as a tuple is determined, it is forwarded to next operation
  - ↳ parallel execution of operators
  - ↳ no materialization of intermediate relations if it can be avoided
- Algebraic optimization (use a tree)
  - ↳ perform operations that eliminate a lot of tuples
    - push down projections and selections (careful with projections)
  - ↳ consider # tuples that flow between operators
  - ↳ e.g. $\Pi_{uname,exp}(\sigma_{gid=G1 \wedge age>30}(U \bowtie GM))$
    $\Rightarrow \Pi_{uname,exp}(\sigma_{age>30}(U) \bowtie \Pi_{uid}(\sigma_{gid=G1}(GM)))$
- Cost-based optimization
  - ↳ alternative plans created by algebraic optimization
  - ↳ considers how operators are executed
  - ↳ process:
    - Pass 1: Find best 1-relation plan for each relation
    - Pass 2: Find best way to join results of each 1-relation to another relation (consider inner and outer)
    - Pass N: Find best way to join results of (N-1)-relation plan to the N^{th} relation
    - prune high cost alternatives
    - bottom-up calculation each possible alternative, estimate the cost

optimization example:

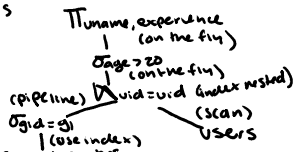
SELECT u.uname, u.experience
FROM Users u, Groupmembets g
WHERE u.uid = g.uid AND g.gid='G1'
AND u.age>20

Pass 1:

- Users: Scan is cheapest since index is non-clustered and age>20 has high RF
  - ↳ cost = 500, output = 20,000 tuples
- Groupmembers: index on gid matches gid = g1 ⇒ $\frac{100000 \text{ records}}{500 \text{ groups}} = 200$ tuples
  - ↳ worst case cost = (1-2 leaf pages) + 200 data pages ≈ 201-202
  - Scan → cost = 10000
  - ↳ output 200 tuples = 2 pages (leave in mem)

Pass 2:

- Users as outer:
  - merge sort join: Sort 200 tuples of GM (in main mem), Sort 500 pages of U and combine merge join w/ Pass 1 of sort: 2 passes ⇒ 3·500 = 1500
  - Block nested loop join: inner relation = 200 tuples = 2 pages in main mem
    read in U once: 500 pages
  - Index nested loop join on uid not possible
- Group member as outer:
  - Index nested loop: cost = 200·(1 leaf + 1 data) = 400 pages ← best
  - Block nested loop: read in U once: 500 pages
  - Merge sort same as above
- Selection + projection on the fly



Assignment 2

- # Data Pages required for a relation:
  - ↳ # tuples/page = (data page size) * (fill) / (size of tuple)
  - ↳ # pages = (total # tuples) / (# tuples per page)
- # leaf pages for an index (indirect II)
  - ↳ #data entries/page = (data page size) * (fill) / (data entry size) (data entry size = size of search key + #rids · size of rid)
  - ↳ # leaf pages = (total # data entries) / (# data entries per page)
- I/O for a conditional selection w/ index ...
  - ↳ non-clustered indirect type II index on desired attribute:
    - (# leaf pages)(% matching tuples) + (total # tuples)(% matching) (worst case)
  - ↳ non-clustered indirect (II) index on 2 attributes
    - (# leaf pages matching attr 1) + (# leaf pages matching Attr 2) + (# tuples matching overall)
- Group by / Selection query
  - ↳ no index: Scan tuples, only desired attributes (tuple size · # tuples / page size)
    (if it all fits in main mem), sort, group by, select: I/O = # tuples (otherwise write out and read next batch)
  - ↳ index on group by attribute: Scan all leaf pages + select (I/O = # leaf pages)
- Index nested loop join (clustered) (clustered ⇒ 1 leaf + 1 data)
  - ↳ I/O = #outer pages + |outer relation| * (cost of finding matching)
- Block nested loop join
  - ↳ use smaller relation as outer.
    - I/O = #outer pages + (#outer pages / |block size|) (# inner pages)
- Sort merge join

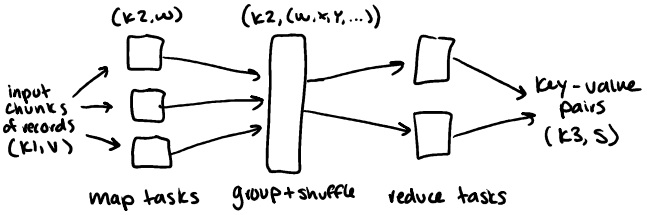
group members

Large Scale Data Processing

- throughput: transactions/queries per time unit
- response time: time for execution of a single transaction
- OLTP (online transaction processing): workload of short, update intensive queries
- OLAP (online analytical processing): workload of complex queries
- inter-query parallelism: different queries run in parallel on different processors
- inter-operator parallelism: different operators w/in same execution tree run on different processors. Pipelining leads to parallelism
- intra-operator parallelism: single operator runs on many processors (scan, join)
- Data partitioning: partition into chunks of records stored at nodes
  - ↳ hash function or range partition
- Execution path: push operator to nodes w/ chunks
  - execute locally at each node in parallel
  - send results to coordinating node
  - merge results
  - minimize I/O and communication costs
- vertical partitioning / column stores ⇒ less I/O

Map-Reduce

1) Distribute data (key-value pairs)
2) Map tasks: extract info and output new key-value pair
3) Shuffle + sort
4) Reduce tasks: aggregate, summarize, filter
5) Write results



- input/output are key/value pairs
- map + reduce are written by us
- group + shuffle are done by the system
- example: word count, DS(k, doc)

```
wordcount map:
    For each input key/value pair (dkey, dtext)
    For each word w of dtext
        output key-value pair (w, 1)
System groups eg (k,(v1, v2, v3 ..., vn))
wordcount Reduce:
    For each input key/value-list pair (k, (v1, ..., vn))
        output (k, n)
```

- Phases:
  - ↳ map tasks: record reader → map function → write to local file
  - ↳ group + shuffle: group keys and aggregate value-lists → copy from map to reduce
  - ↳ reduce tasks: reduce → write to file system
- Implementation:
  - ↳ master node controls execution, partitions file into m parts, assigns workers to m map tasks
  - ↳ map workers execute map tasks and write to local disk
  - ↳ master assigns workers to r reduce tasks
  - ↳ reduce workers implement group and shuffle and execute reduce tasks
- Selection w/ MapReduce
  - ↳ R(A,B,C), selection w/ condition c on R
    - map: for each tuple t of R for which c holds, output (A, (B, C))
    - Reduce: identity (output (A, (B, C))
- Join w/ Map-reduce
  - ↳ natural join R(A,B,C) with Q(C,D,E)
    - map: For each tuple (a,b,c) of R, output (c, (R,(a,b)))
      For each tuple (c,d,e) of Q, output (c, (Q,(d,e)))
    - group + shuffle aggregates key-value pairs
    - reduce: For each tuple (c, value-list)
      $R_t = Q_t$ = empty
      for each v = (rel, tuple) in value-list
      if v.rel = R; insert tuple into R_t
      else insert tuple into Q_t
      for V1 in R_t, for V2 in Q_t, output (c, (V1, V2))
- Projection with map-reduce
  - ↳ projection on B, C of R
    - map: for each tuple t = (a, (b, c)) of R, let t' = (b, c)
      output (t', 0)
    - group + shuffle
    - reduce: for each tuple (t', (0, 0, 0...)), output (t', 0)
- Group by w/ Map-reduce
  - ↳ grouping: SELECT b, max(c) GROUPBY b
    - map: for each tuple (a, (b, c)) of R, output (b, c)
    - group + shuffle produces key-value list
    - reduce: for each (b, (c1, c2, ...)) perform aggregation

Declarative Languages

- pig latin - run complex queries that require several map-reduce jobs
- E.g.:

```
Users = load 'users' as (name, age);
Fltrd = filter Users by age>=18 and age<=15;
Pages = load 'pages' as (uname, url);
Jnd = join Fltrd by name, Pages by uname;
Grpd = group Jnd by url;
Smmd = foreach Grpd generate ($0),
            COUNT($1) as clicks;
Srtd = order Smmd by clicks desc;
Top5 = limit Srtd 5; store Top5 into 'top5sites';
```

- load: read info from a file into a temp relation
- store: write relation into file
- selection: Res = filter R1 by ...
- Join: Res = join R1 by a1, R2 by a2
- order by: Res = order R1 by a1 desc
- group by: Grpd = group Rel by A
- For each:

```
Smmd = foreach Grpd generate ($0), COUNT($1) as c
Smmd = foreach Grpd generate group, Count(Rel)
```

- projection: Rel = foreach R1 generate A,B;
- flattening: Res = foreach R generate $0, flatten($1)
  - ↳ make $1 appear as part of list, not nested
- Dump to screen

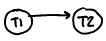
- master handles failures (assigns new workers)

Transactions

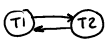
- Database transaction: series of reads and writes on objects
- OLTP: update intensive (15-50% of SQL statements are updates)
- ACID:
  - Atomicity - a transaction is atomic if it is executed all or none
  - Consistency - transaction preserves the consistency of the data
  - Isolation - in case that transactions are executed concurrently, the effect must be the same as if one at a time
  - Durability - changes made by a transaction must be permanent
- Atomicity: - after a commit, all changes are installed in the DB.
  - after an abort → local recovery, eliminate partial results / undo all modifications so far
- Durability: there must be a guarantee that the changes will last - i.e. survive failures / crashes. persistance.
  - e.g.: T_1 —c T_2 —c T_3 — T_4 — T_5 —a (crash)
    - T1, T2, T3 are dumbie
    - T4, T5 are aborted
- System crash recovery
  - restart server and perform global recovery:
    - transactions that committed before crash: changes in DB
    - transactions that aborted: no changes in DB
    - transactions that were active: no changes in DB
- Each update changes the records corresponding page
  - dirty pages are written to disk at buffer manager's descretion
- To undo/redo in case of crash - write log info a log disk
- Write ahead logging (WAL):
  - for each write(x), append log record with before and after image of x to log tail. Undo changes of active transactions with before image. Redo changes of commited transactions w/ after image
  - flush log to disk before flushing the page. (so you can undo in case of abort)
  - flush log to disk befor commit (so you can redo when something commited)
  - log is an append; flush means all of log is flushed in one shot
- Recovery
  - Undo: for each transaction that did not commit, find log records and install before images.
  - Redo: for each transaction that commited, find all log records and install after images.

Isolation and Concurrency Control

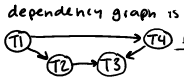
- Isolation: transactions execute concurrently but each runs in isolation - and do not affect each other.
  - enforced by concurrency control
- Concurrency control provides serializable executions: affects of concurrent execution is the same as running one after the other.
- Schedule: sequence of actions (w, r, c, a) from a set of transactions
- complete schedule: contains commit/abort for each transaction
- Serial Schedule: transactions executed fully one after the other
- Dirty reads: T2 reads from Ti before Ti commits (might read inconsistent data)
  - e.g. T1: R1(A) W1(A) R1(B) W1(B) C
    T2: R2(A) R3(B) C3
    - like T2 executes after T1 (reads what T1 wrote)
    - like T2 executes before T1 (reads B before T1 wrote B)
    > conflict
- Unrepeatable reads
  - T1 reads an item twice but T2 changes the value inbetwen
  - e.g. T1 R1(A) R1(A) C1
    T2 W2(A) C2
    - T1 conceptually executes before and after T2
- Lost update
  - e.g. T1 R1(A) W1(A) C1
    T2 R2(A) W2(A) C2
    - as if T2 never happened
- Conflicting operations: 2 operations (from different transaction) if they access the same object and both operations are writes, or one is write and one is read
- conflict serializable schedules
  - two schedules are conflict equivalent if they ① involve the same actions of the same committed transactions and ② every pair of conflicting actions of committed transactions is ordered the same way
  - a schedule is conflict serializable if it is conflict equivalent to some serial schedule
  - e.g: T1 r1(x) w1(x) w1(y) c1
    T2 r2(z) r2(y) w2(x) c2
    is conflict serializable with
    T1 r1(x) w1(x) w1(y) c1
    T2 r2(z) r2(y) w2(x) c2
- Serializability + conflict graphs:
  - Let S be a shedule $(T, O, <)$
    - each transaction T_i in T is represented by a node
    - there is an edge from T_i to T_j if an operation of T_i proceeds and conflicts with an operation of T_j
    - e.g. T1: R1(A) W1(A) R1(B) W1(B) C1
      T2: R2(A) W2(A) R2(B) W2(B) C2

      T1: R1(A) W1(A) R1(B) W1(B) C
      T2: R2(A) W2(A) R2(B) W2(B) C2

- Dependency Graphs: Schedule is conflict serializable iff its dependency graph is acyclic

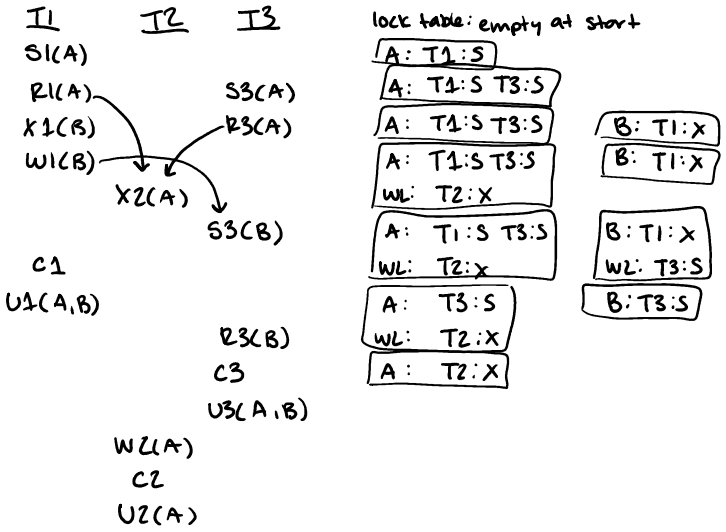
  

- concurrency control: during execution, take measures such that a non-serializable execution can never happen
- locking: each transaction must obtain an S (shared) lock on object before reading and an X (exclusive) lock on object before writing. If an X lock is granted to an object, no other lock can be granted on the object at the same time. If an S lock is granted, no X lock can be granted on the same object at the same time. If a conflicting lock is active, the transaction must wait until the lock is released at the end of the other transaction. ← 2PL
- Phase 1: growing + locking, Phase 2: shrinking + unlocking at end

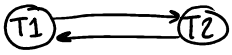
Concurrency control cont.

- 2PL example

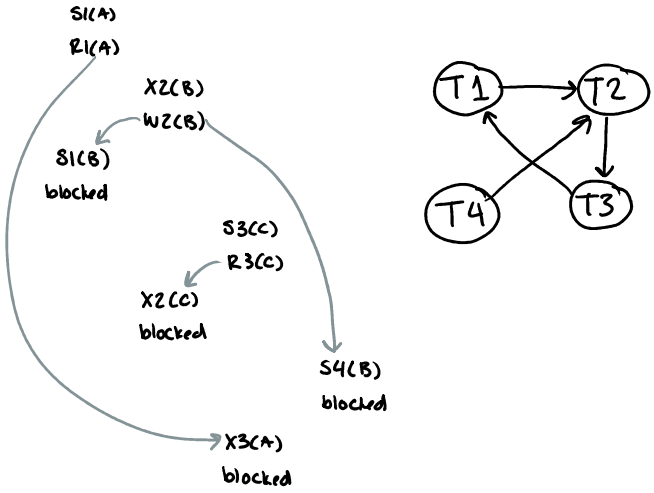
Submission order: R1(A) R3(A) W1(B) W2(A) R3(B)



- 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. T1 T2: T3: T4:

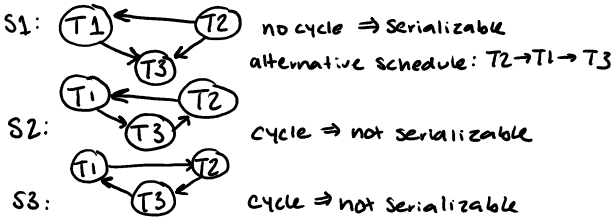


Extra assignment Ex.1/2:

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

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			S2			S3 Variation 1			S3 Variation 2		
T1	T2	T3	T1	T2	T3	T1	T2	T3	T1	T2	T3
		S3(a)	S1(a)			S1(a)			S1(a)		
		r3(a)	r1(a)			r1(a)			r1(a)		
	S2(b)				S3(c)		X2(a)			X2(a)	
	r2(b)				r3(c)			S3(a)			S3(a)
X1(c)				X2(b)				r3(a)	X1(c)		
w1(c)				w2(b)				X3(b)	w1(c)		
	X2(b)		X1(a)					w3(b)	c1		
	w2(b)		w1(a)					c3	U(a,c)		
		S3(c)		S2(c)				U(a,b)		w2(a)	
		blocked		r2(c)		X1(c)				S2(b)	
	S2(a)			X2(c)		w1(c)				r2(b)	
	r2(a)			blocked		c1				X2(c)	
	c2		X1(c)			U(a,c)				w2(c)	
	U(a,b)		blocked				w2(a)			c2	
S1(b)					S3(a)		S2(b)			U(a,b,c)	
r1(b)					deadlock		r2(b)				r3(a)
X1(b)					abort		X2(c)				X3(b)
w1(b)					U(c)		w2(c)				w3(b)
c1				w2(c)			c2				c3
U(b,c)				c2			U(a,b,c)				U(a,b)
		r3(c)		U(b,c)							
		X3(b)	w1(c)								
		w3(b)	c2								
		c3	U(a,c)								



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 to avoid DL →

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

V2(a)

X3(a) ← this would cause deadlock

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)

      } outgoing
      - ↳ 3-5 edges: (e)<-[*3..5]-(n) } ingoing
      - ↳ 3-5 edges: (e)-[*3..5]-(n) } 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)
- P16: 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

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 booked 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;
```

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')
```

- 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)
```

```
GROUP BY psid HAVING COUNT(*) > 1
```

Practice final cont

Query evaluation:

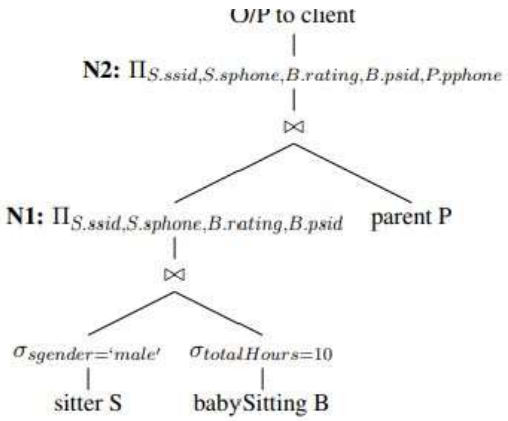
```
sitter(ssid:INT ,sname:VARCHAR(40) ,sphone:CHAR(12) ,sgender:CHAR(6)
,hourlyRate:INT ,preferredMinAge:INT)
parent((psid):INT, pname:VARCHAR(40), pphone:CHAR(12))
babySitting(reservationId:INT, ssid:INT ,availDate:DATE, psid:INT
,cname:VARCHAR(40) ,starthr:INT, totalhrs:INT, addnlChildren:INT, rating:INT)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