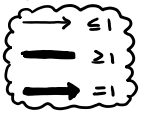
ER Model

- 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)

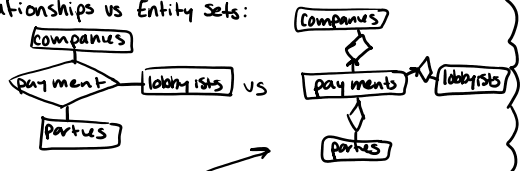
- ↳ many-many relationship translated to a table
- ↳ participation constraints can't usually be reflected besides NOT NULL
- ↳ weak entity set and identifying relationship are translated into a single table
- ↳ ISA hierarchies: distribute info among relations OR subclasses have all attributes OR one 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: $\cup$
- $\sigma_{comparison}(relation)$, $R_1 \bowtie_C R_2$
- 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, $R1 \bowtie R2 = R2 \bowtie R1$,
  $\Pi_{L2}(\Pi_{L1}(R)) = \Pi_{L2}(R)$ IF $L2 \subseteq L1$, $\sigma_{C2}(\sigma_{C1}(R)) = \sigma_{C1 \wedge C2}(R)$, $R1 \bowtie (R2 \bowtie R3) = (R1 \bowtie R2) \bowtie R3$

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.

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
```
(or) (and)
- Union/Intersect/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;
```

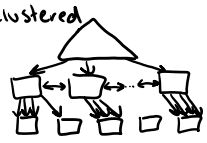
Note on views: DB does not allow insertion into complex views that have join conditions on the view definition

Buffers

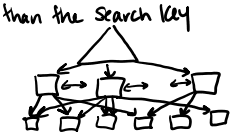
- DBMS stores info persistently on disks
- Pin count = # upper layer programs using page
- record ID : rid

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

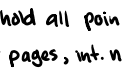
B+tree cost example:

Given: R(int, int^{8}, char, char, char, char)
- 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 = $200{,}000 / 18 \approx$ 11111 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 ↗)



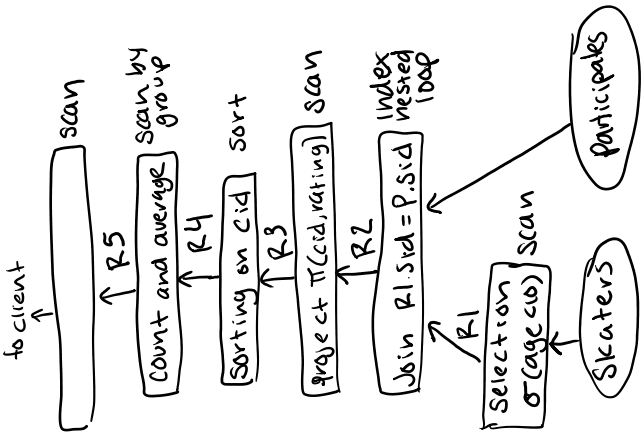
- DB2: CREATE INDEX ind1 ON Students (startyear);
  DROP INDEX ind1;
  - ↳ referential integrity / uniqueness
    CREATE UNIQUE INDEX ind1 ON Students(email)
    - ↳ additional attributes:
      CREATE UNIQUE INDEX ind1 ON Students(sid)
      INCLUDE (name)
      Index on sid, entry contains sid, name, 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) = $10 + 4 \cdot 10 = 50$, $3000/50 =$ 60
- Indirect Index II on A,B, A=4, B=6, 100000 records, 5 records per (A,B) data entries
  - ↳ find # leaf pages: size(data entry) = $10 + 5 \cdot 10 = 60$, $4000 \cdot 0.75/60 = 50$ 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) $\text{Red}(\sigma_{exp=5}(U)) = 10{,}000/40{,}000 = 0.25$
  - unknown # tuples for condition ⇒ assume uniform + independent distribution
    - e.g. $\text{Red}(\sigma_{exp=5}(U)) = \frac{1}{10} = 0.1$
  - red = size of selected range / total size of domain
  - e.g. $\text{Red}(\sigma_{exp=5 \text{ and } age<=16}(U)) = \text{Red}(\sigma_{exp=5}(U)) * \text{Red}(\sigma_{age<=16}(U))$
  - Size of result of selection = (# input tuples)(reduction factor)
    - e.g. 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 (bc 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 (500 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
    - e.g. A = 100 OR B = 50
      - 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

- 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 / $|bp_o|$ (block 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 attr)
    - 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)

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. $\Pi_{uname,exp}(\sigma_{gid=G1 \wedge age>50}(U \bowtie GM))$
    $\Rightarrow \Pi_{uname,exp}(\sigma_{age>50}(U) \bowtie \Pi_{uid}(\sigma_{gid=G1}(GM)))$
- 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:

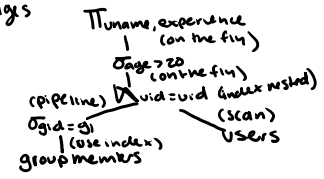
SELECT u.uname, u.experience
FROM Users u, Groupmembets g
WHERE u.uid = g.uid AND g.gid='G1'
AND u.age>20

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

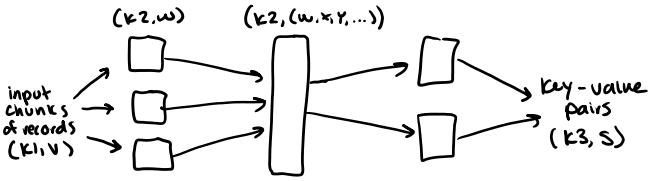
- # Data Pages required for a relation:
  - ↳ # tuples/page = (data page size) * (fill) / (size of tuple)
  - ↳ # pages = (total # tuples) / (# tuples per page)
- # leaf pages for an index (indirect II) (attr size + #rids · sizeof(rid))
  - ↳ # data entries/page = (data page size) * (fill) / (data entry size)
  - ↳ # 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: (worst case)
    - (# leaf pages)(% matching tuples) + (total # tuples)(% matching)
  - ↳ non-clustered indirect (II) index on 2 attributes
    - (# leaf pages matching attr 1) + (# leaf pages matching Attr2) + (# tuples matching all)
- 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

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
  - master handles failures (assigns new workers)
- 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';
```

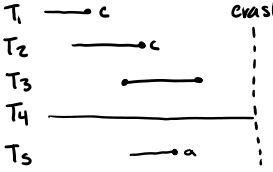
- Load: read info from a file into a temp relation
- store: write relation into file
- selection: Res = filter R1 by ...
- Join: Res = join R1 by a1, R2 by a2
- order by: Res = order R1 by a1 desc
- group by: Grpd = group Rel by A
- For each:

```
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

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.:

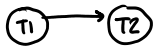


  - T1, T2, T3 are durable
  - T4, T5 are aborted
- System crash recovery
  - restart server and perform global recovery:
    - transactions that committed before crash: changes in DB
    - transactions that aborted: no changes in DB
    - transactions that were active: no changes in DB
- Each update changes 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 instal all before images.
  - Redo: for each transaction that commited, find all log records and instal 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 T1 before T1 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 inbetween
  - 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 transactions) 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