Comp 5311 Database Management Systems

13. Query Processing and Optimization

Complex Joins

- Join with a conjunctive condition:

  $r\ JOIN_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}\ s$

  1. Either use nested loops/block nested loops, or
  2. Compute the result of one of the simpler joins $r\ JOIN_{\theta i}\ s$
     - final result comprises those tuples in the intermediate result that satisfy the remaining conditions

       $\theta_1 \wedge \ldots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \ldots \wedge \theta_n$

- Join with a disjunctive condition

  $r\ JOIN_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}\ s$

  1. Either use nested loops/block nested loops, or
  2. Compute as the union of the records in individual joins*:*

     $(r\ JOIN_{\theta 1}\ s) \cup (r\ JOIN_{\theta 2}\ s) \cup \ldots \cup (r\ JOIN_{\theta n}\ s)$

     useful only if all conditions are restrictive (selective)

The Projection Operation

```
SELECT   DISTINCT R.bid
FROM     Reserves R
```

- An approach based on external sorting for duplicate elimination:
  - Modify Pass 0 of external sort to eliminate unwanted fields. Thus, sorted runs contain smaller records. (Size ratio depends on # and size of fields that are dropped.)
  - Modify merging passes to eliminate duplicates. Thus, number of result tuples smaller than input. (Difference depends on # of duplicates.)
  - Cost: In Pass 0, read original relation, write out same number of smaller tuples. In merging passes, fewer tuples written out in each pass.

Projection Based on Hashing

- *Partitioning phase*: Read R using one input buffer. For each tuple, discard unwanted fields, apply hash function *h1* to choose one of M-1 output buffers (M is the number available main memory pages).
  - Result is M-1 partitions (of tuples with no unwanted fields). 2 tuples from different partitions guaranteed to be distinct.
- *Duplicate elimination phase*: For each partition, read it and build an in-memory hash table, using hash function *h2* (<> *h1*) on all fields, while discarding duplicates.
- Cost: For partitioning, read R, write out each tuple, but with fewer fields. This is read in next phase.

Discussion of Projection

- Sort-based approach is the standard; better handling of skew and result is sorted.
- If an index on the relation contains all wanted attributes in its search key, can do *index-only* scan.
  - Apply projection techniques to data entries (much smaller!)
- If an ordered (i.e., tree) index contains all wanted attributes as *prefix* of search key, can do even better:
  - Retrieve data entries in order (index-only scan), discard unwanted fields, compare adjacent tuples to check for duplicates.

Set Operations

Set operations can be handled by join algorithms. Should also remove duplicates.

- Sorting based approach:
  - Sort both relations (on the same attribute).
  - The merging phase depends on the operation.
    - **Intersection**: report a tuple only if it belongs to both files
    - **Union**: report all tuples except for the ones that belong to both files
    - **Set difference**: report the tuples that belong to the first file but not the second one
- Hash based approach:
  - Partition files *r* and *s* using hash function *h* (on all attributes). *s* is the build input.
  - For each *s*-partition, build in-memory hash table (using *h2*), scan corresponding *r*-partition (page-by-page) and for each tuple t of R
    - **Intersection** : report t only if it also belongs to *s*
    - **Union**: report t if it does not belong to *s*. At the end report all tuples of *s*
    - **Set difference** : report t only if it does not belong to *s* (computes R-S; how to compute S-R)

Aggregate Operations (AVG, MIN, etc.)

- Without grouping:
  - In general, requires scanning the relation.
  - Given index whose search key includes all attributes in the SELECT or WHERE clauses, can do index-only scan (e.g., "find the average age of all sailors given an index on age").
- With grouping (assuming that grouping attribute values do not fit in memory):
  - Sort on group-by attributes, then scan relation and compute aggregate for each group. (E.g., "compute the average rating for each age value"; what about "compute the average age for each rating value")
  - Similar approach based on hashing on group-by attributes.
  - Given tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, can do index-only scan; if group-by attributes form prefix of search key, can retrieve data entries/tuples in group-by order.

Evaluation of Expressions

- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression tree
  - **Materialization**: generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
  - **Pipelining**: pass on tuples to parent operations even as an operation is being executed

Materialization

- Materialized evaluation is always applicable
- Cost of writing intermediate results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing final results to disk, so
    - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk

Pipelining

- **Pipelined evaluation :** evaluate several operations simultaneously, passing the results of one operation on to the next.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.
- Pipelines can be executed in two ways: **demand driven** and **producer driven**

Pipelining (Cont.)

- In **demand driven** or **lazy** evaluation
  - system repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain "**state**" so it knows what to return next
- In **produce-driven** or **eager** pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples

Evaluation Algorithms for Pipelining

- Some algorithms are not able to output results even as they get input tuples. They are called blocking.
  - E.g. sort merge join, or hash join
  - These leads to intermediate results being written to disk and then read back always
- Algorithm variants are possible to generate (at least some) results on the fly, as input tuples are read in
  - E.g. hybrid hash join generates output tuples even as probe relation tuples in the in-memory partition (partition 0) are read
  - **Pipelined join technique**: Hybrid hash join, modified to buffer partition 0 tuples of both relations in-memory, reading them as they become available, and output results of any matches between partition 0 tuples
    - When a new r_0 tuple is found, match it with existing s_0 tuples, output matches, and save it in r_0
    - Symmetrically for s_0 tuples

Query Optimization - Motivation

- Consider the relations R1(<u>A</u>,B,*C*), R2(<u>C</u>,D,*E*), and R3(<u>E</u>,F). Primary keys are underlined and foreign keys in italics. Foreign keys are not NULL.
- Assume that :
  - ✓ R1 has 1000 tuples
  - ✓ R2 has 10000 tuples
  - ✓ R3 has 100000 tuples

✓ What is the best way to join R_1, R_2, and R_3?

(R_1 JOIN$_C$ R_2) JOIN$_E$ R_3 or R_1 JOIN$_C$ (R_2 JOIN$_E$ R_3)

- What is the size (number of records) in intermediate result R_1 JOIN$_C$ R_2?
- What is the size (number of records) in intermediate result R_2 JOIN$_E$ R_3?
- What is the size (number of records) in the final result?

Cost difference between a good and a bad way of evaluating a query can be enormous

- How the optimizer can choose the best evaluation plan for processing the query?
- Different plans for a given query involve
  - Different but equivalent algebra expressions
  - Different algorithms for each operation

Query Optimization Approaches

- Practical query optimizers incorporate elements of the following two broad approaches:
  - Search all the plans and choose the best plan in a cost-based fashion – COST BASED OPTIMIZATION

    GENERAL IDEA:

    1] Generate possible evaluation plans

    2] Estimate the cost of each plan

    3] Execute the plan with the minimum expected cost
  - Use heuristic to choose a plan – HEURISTIC OPTIMIZATION

    GENERAL IDEA:

    1] Perform the cheap operations first (i.e., selections before joins)

    2] Try to utilize existing indexes

    3] Remove the useless attributes early

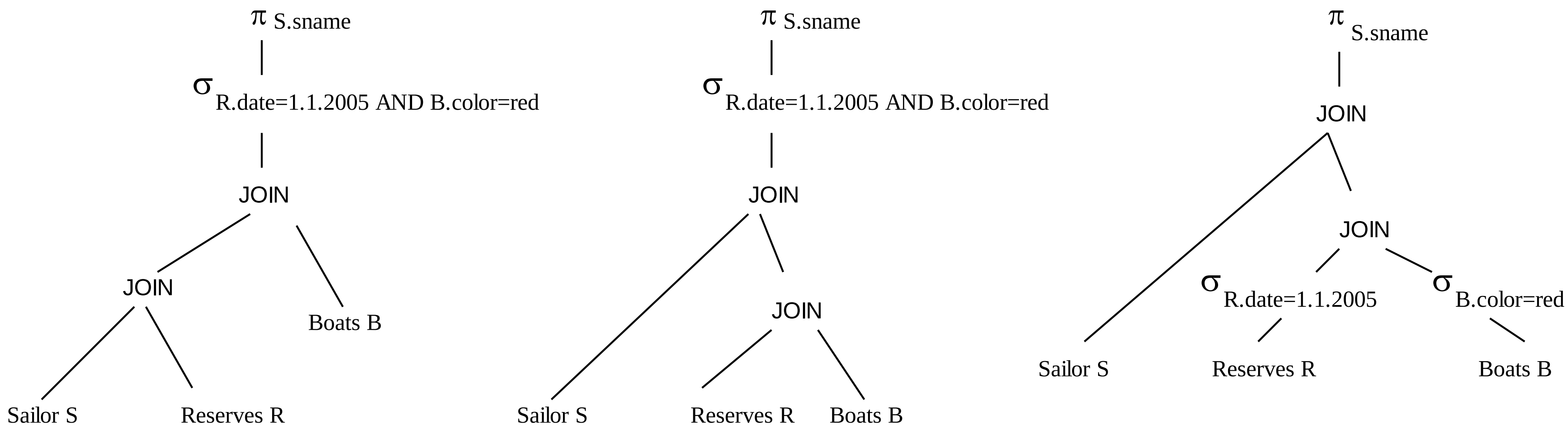
Different Algebra Expressions

- Given a query, the optimizer will first generate an algebra expression (tree)

  SELECT sname

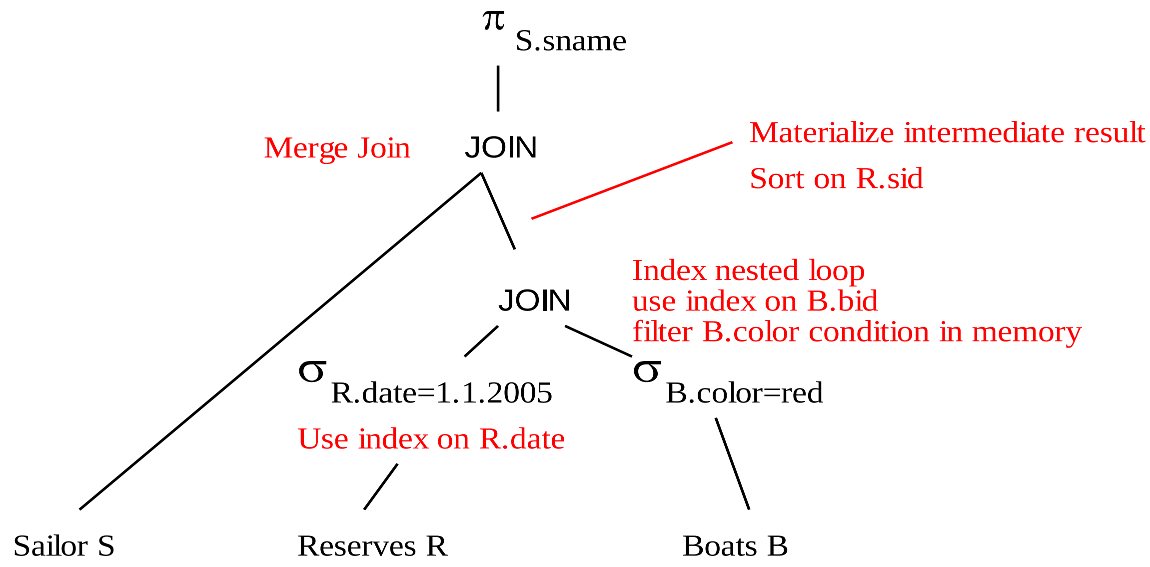
  FROM Sailor S, Reserves R, Boats B

  WHERE S.sid=R.sid and R.bid = B.bid and R.date=1.1.2005 and B.color=red



Evaluation Plan

- An evaluation plan defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.
- Example assuming that Sailor is sorted on sid



Choice of Evaluation Plans

- Must consider the interaction of evaluation techniques when choosing evaluation plans: choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. E.g.
  - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost of a subsequent operation (e.g., duplicate elimination).
  - nested-loop join may provide opportunity for pipelining
- Need to estimate the cost of operations
  - Depends critically on statistical information about relations which the database must maintain
    - E.g. number of tuples, number of distinct values for join attributes, etc.
  - Need to estimate statistics for intermediate results to compute cost of complex expressions

Catalog Information for Statistics Estimation

Every database system has a **system catalog** (or otherwise called **data dictionary**) that stores **metadata**. Metadata include statistics about the stored tables. Specifically, for each relation *R* it stores:

- n_R: number of tuples in *R.*
- b_R: number of blocks containing tuples of *R.*

additional info:

- f_R: blocking factor of *R* — i.e., the number of tuples of *R* that fit into one page.
- size of each attribute
- *V(A, R):* number of distinct values that appear in *R* for attribute *A;* same as the size of $\prod_A(R)$.

Catalog Information about Indices

- HT_i: number of levels in index i — i.e., the height of i.
  - For a balanced tree index (such as B+-tree) on attribute A of relation R, $HT_i = \lceil \log_{f_i}(V(A,R)) \rceil$.
  - For a hash index, HT_i is 1, or 1.2 if we assume the existence of overflow buckets.
- Additional Info:
  - f_i: average fan-out of internal nodes of index i, for tree-structured indices such as B+-trees.
  - LB_i: number of lowest-level index blocks in i — i.e, the number of blocks at the leaf level of the index.

Selection Size Estimation

The output size of an operation determines (i) the cost of the operation and (ii) the cost of subsequent operations. Therefore its accurate estimation is important for optimization.

- **Equality selection** $\sigma_{A=v}(R)$
  Example: $\sigma_{rating=8}(SAILORS)$
  $SC(A, R)$: **s**election **c**ardinality of attribute A of relation R; average number of records that satisfy equality on A.
    - $SC(A, R) = n_R / V(A, R)$

- $\lceil SC(A, R)/f_R \rceil$ — number of blocks that these records will occupy if these records are ordered on attribute A.
- If the records are not ordered on A, each record may reside in a different page

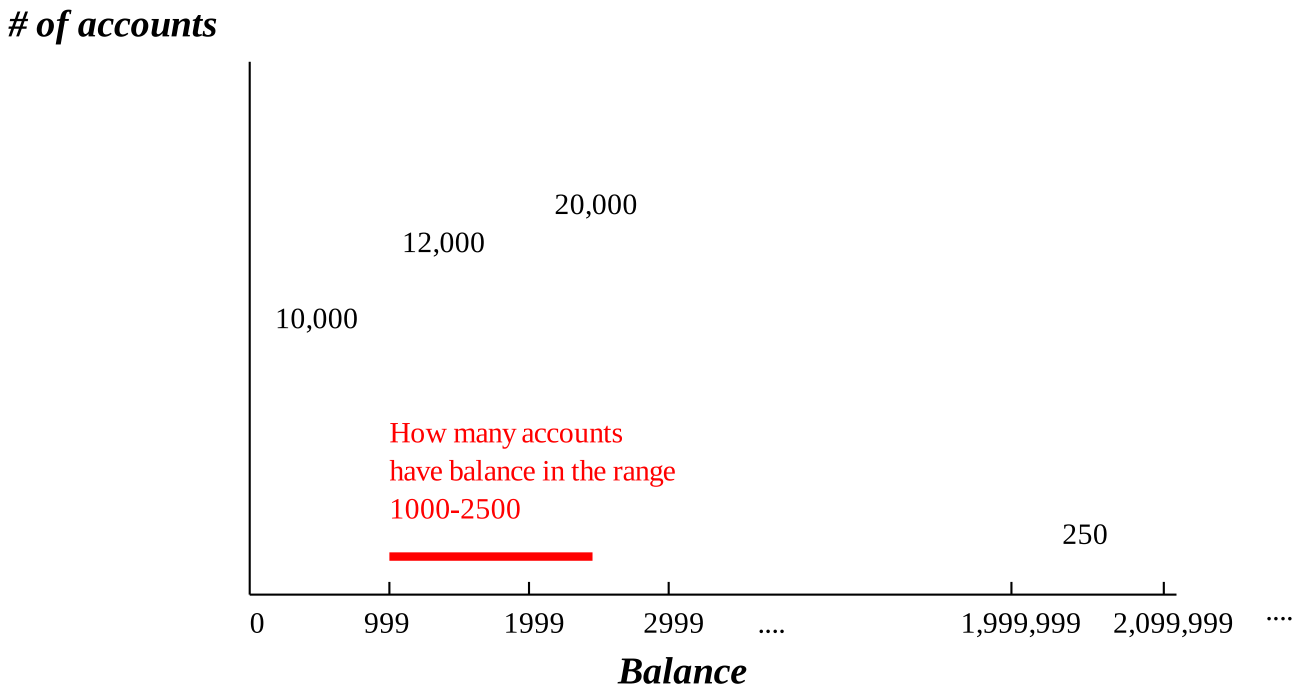
- Equality condition on a key attribute: $SC(A,R) = 1$

Selections Involving Comparisons

- Selections of the form $\sigma_{A<v}(R)$ (case of $\sigma_{A>v}(R)$ is symmetric)
- Let C denote the estimated number of tuples satisfying the condition.
  - min(A,*R*) and max(A,*R*) are available in catalog
    - C = 0 if v < min(A,*R*)
    - C = $n_R \cdot \dfrac{v - \min(A,R)}{\max(A,R) - \min(A,R) + 1}$
  - Example: $\sigma_{rating<2}(SAILORS)$ = # records in sailors * (2-1)/(10-1+1)=# records in sailors /10
  - Again: more accurate estimation using histograms

Histograms

- The previous estimates are based on the assumption that each value of *A* has the same probability.
- This uniformity assumption rarely holds in practice.
- Commercial systems use histograms.
- In histograms, we assume local uniformity within each bucket (but not global uniformity).



Implementation of Complex Selections

- The **selectivity** of a condition θ_i is the probability that a tuple in the relation R satisfies θ_i. If s_i is the number of satisfying tuples in R, the selectivity of θ_i is given by s_i / n_R.
- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(R)$. The estimate for number of tuples in the result is: $n_R * \frac{s_1 * s_2 * \ldots * s_n}{n_R^n}$
- **Disjunction:** $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}(R)$. Estimated number of tuples:

$$n_R * \left(1 - (1 - \frac{s_1}{n_R}) * (1 - \frac{s_2}{n_R}) * \ldots * (1 - \frac{s_n}{n_R})\right)$$

based on the logical equivalence: $\theta_1 \vee .. \vee \theta_n = \overline{\overline{\theta_1} \wedge .. \wedge \overline{\theta_n}}$

Attribute Independence

- The previous estimates are based on the assumption that values of attributes are independent.
- This attribute independence assumption rarely holds.
- Consider for instance the sailor table, and assume that the rating of a sailor increases with his experience. We have histograms on both the rating and the age rating. Furthermore, the number of sailors with age 20 and 50 are equal.

- The two queries below are estimated to have the same SC

```
Select *
From Sailors
Where Rating = 10 and Age = 20

Select *
From Sailors
Where Rating = 10 and Age = 50
```

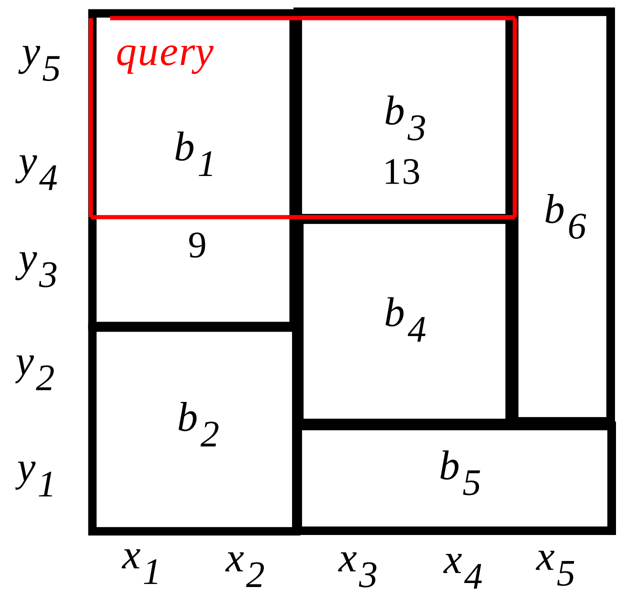
- Which query is expected to retrieve more record?
- Solution: Multidimensional histograms

Multi-dimensional histograms

- Main idea:
  - Divide the space (e.g., age-rating) in buckets, so that data in each bucket are almost uniform.
  - Keep in memory the bucket extents and the number of records per bucket
  - Use this information to estimate the number of objects in query window

MINSKEW example

y_5	1	2	3	3	5
y_4	2	2	3	4	5
y_3	1	1	9	11	5
y_2	4	5	10	9	6
y_1	5	6	1	1	1
	x_1	x_2	x_3	x_4	x_5



Estimation of the Size of Joins

- The Cartesian product $R \times S$ contains $n_R . n_S$ tuples; each tuple occupies $s_R + s_S$ bytes.
  - If $R \cap S = \emptyset$, then *R* JOIN *S* is the same as $R \times S$.
- If $R \cap S$ is a key for *R*, then a tuple of *S* will join with at most one tuple from *R*
  - therefore, the number of tuples in *R* JOIN *S* is no greater than the number of tuples in *S*.
- If $R \cap S$ *in S* is a (not null) foreign key referencing *R*, then the number of tuples in *R* JOIN *S* is exactly the same as the number of tuples in *S*.
  - The case for $R \cap S$ being a foreign key referencing *S* is symmetric.
- In the example query *sailor* JOIN *reserves, sid* in *reserves* is a foreign key of *sailor*
  - hence, the result has exactly $n_{reserves}$ tuples.

Estimation of the Size of Joins (Cont.)

- If $R \cap S = \{A\}$ is not a key for R or S.
  If we assume that every tuple r in R produces $n_S / V(A,S)$ tuples in R *JOIN* S, the number of tuples in R JOIN S is estimated to be:

  $$\frac{n_R * n_S}{V(A,S)}$$

  If the reverse is true, the estimate obtained will be:

  $$\frac{n_R * n_S}{V(A,R)}$$

  The lower of these two estimates is probably the more accurate one.

Estimation of the Size of Joins (Cont.)

- Example two tables with information about *sailors* and *reserves*. The join attribute is the sailor id, i.e., *sid* in *reserves* is a foreign key on *sailor.*
- Catalog information for join examples: n_{sailor} = 10,000, $n_{reserves}$ = 5000, *V(sid, reserves)* = 2500, which implies that only 2500 sailors have boat reservations.
- Compute the size estimates for *reserves* $JOIN_{sid}$ *sailor* without using information about foreign keys:
  - *V(sid, reserves) =* 2500, and
    V(sid, sailor) = 10000
  - The two estimates are 5000 * 10000/2500 = 20,000 and 5000 * 10000/10000 = 5000
  - We choose the lower estimate, which in this case, is the same as our earlier computation using foreign keys.

Size Estimation for Other Operations

- Projection: estimated size of $\prod_A(R) = V(A,R)$
- Aggregation : estimated size of *Group-by A* = $V(A,R)$
- Set operations
  - For unions/intersections of selections on the same relation: rewrite and use size estimate for selections
    - E.g. $\sigma_{\theta 1}(R) \cup \sigma_{\theta 2}(R)$ can be rewritten as $\sigma_{\theta 1 \vee \theta 2}(R)$
  - For operations on different relations:
    - estimated size of $R \cup S$ = size of R + size of S.
    - estimated size of $R \cap S$ = minimum (size of R, size of S).
    - estimated size of $R - S$ = R.
    - All the three estimates may be quite inaccurate, but provide upper bounds on the sizes.