

# Query Processing & Optimization

Michalis Vazirgiannis

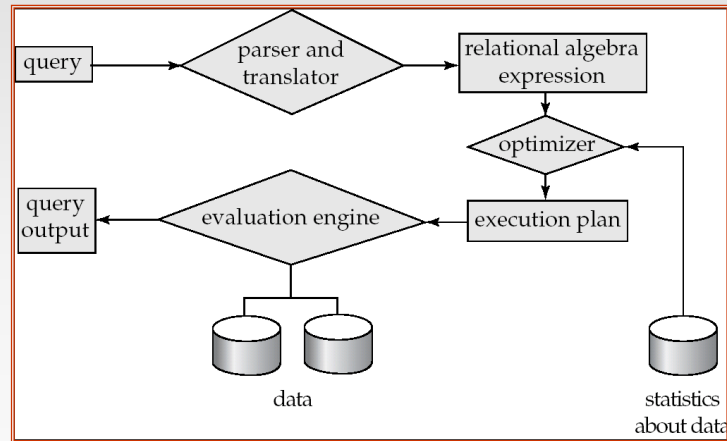
Based on slides from the book  
Database System Concepts, 5th Ed.

## Query Processing

- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions

## Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



13.3

## Basic Steps in Query Processing (Cont.)

- Parsing and translation
  - translate the query into its internal form. This is then translated into relational algebra.
  - Parser checks syntax, verifies relations
- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

13.4

## Basic Steps in Query Processing : Optimization

- A relational algebra expression may have many equivalent expressions
  - E.g.,  $\sigma_{balance < 2500}(\Pi_{balance}(account))$  is equivalent to  $\Pi_{balance}(\sigma_{balance < 2500}(account))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.
  - E.g., can use an index on *balance* to find accounts with balance < 2500,
  - or can perform complete relation scan and discard accounts with balance  $\geq 2500$

13.5

## Basic Steps: Optimization (Cont.)

- **Query Optimization:** Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - e.g. number of tuples in each relation, size of tuples, etc.
- Here we study
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression
- Remains
  - how to optimize queries, that is, how to find an evaluation plan with lowest estimated cost

13.6

## Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU, or even network communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
  - Number of seeks \* average-seek-cost
  - Number of blocks read \* average-block-read-cost
  - Number of blocks written \* average-block-write-cost
    - Cost to write a block is greater than cost to read a block
      - data is read back after being written to ensure that the write was successful

13.7

## Measures of Query Cost (Cont.)

- For simplicity we just use the *number of block transfers from disk and the number of seeks* as the cost measures
  - $t_T$  – time to transfer one block
  - $t_S$  – time for one seek
  - Cost for b block transfers plus S seeks  
 $b * t_T + S * t_S$
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
- We do not include cost to writing output to disk in our cost formulae
- Several algorithms can reduce disk IO by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
    - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available
- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation

13.8

## Selection Operation

- **File scan** – search algorithms that locate and retrieve records that fulfill a selection condition.
- **Algorithm A1 (linear search)**. Scan each file block and test all records to see whether they satisfy the selection condition.
  - Cost estimate =  $b_r$  block transfers + 1 seek
    - ▶  $b_r$  denotes number of blocks containing records from relation  $r$
  - If selection is on a key attribute, can stop on finding record
    - ▶ cost =  $(b_r/2)$  block transfers + 1 seek
  - Linear search can be applied regardless of
    - ▶ selection condition or
    - ▶ ordering of records in the file, or
    - ▶ availability of indices

13.9

## Selection Operation (Cont.)

- **A2 (binary search)**. Applicable if selection is an equality comparison on the attribute on which file is ordered.
  - Assume that the blocks of a relation are stored contiguously
  - Cost estimate (number of disk blocks to be scanned):
    - ▶ cost of locating the first tuple by a binary search on the blocks
      - $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
    - ▶ If there are multiple records satisfying selection
      - Add transfer cost of the number of blocks containing records that satisfy selection condition
      - Will see how to estimate this cost further

13.10

## Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.
- **A3** (*primary index on candidate key, equality*). Retrieve a single record that satisfies the corresponding equality condition
  - $Cost = (h_i + 1) * (t_T + t_S)$
- **A4** (*primary index on nonkey, equality*) Retrieve multiple records.
  - Records will be on consecutive blocks
    - ▶ Let  $b$  = number of blocks containing matching records
  - $Cost = h_i * (t_T + t_S) + t_S + t_T * b$
- **A5** (*equality on search-key of secondary index*).
  - Retrieve a single record if the search-key is a candidate key
    - ▶  $Cost = (h_i + 1) * (t_T + t_S)$
  - Retrieve multiple records if search-key is not a candidate key
    - ▶ each of  $n$  matching records may be on a different block
    - ▶  $Cost = (h_i + n) * (t_T + t_S)$ 
      - Can be very expensive!

13.11

## Selections Involving Comparisons

- Can implement selections of the form  $\sigma_{A \leq v}(r)$  or  $\sigma_{A \geq v}(r)$  by using
  - a linear file scan or binary search,
  - or by using indices in the following ways:
- **A6** (*primary index, comparison*). (Relation is sorted on A)
  - ▶ For  $\sigma_{A \geq v}(r)$  use index to find first tuple  $\geq v$  and scan relation sequentially from there
  - ▶ For  $\sigma_{A \leq v}(r)$  just scan relation sequentially till first tuple  $> v$ ; do not use index
- **A7** (*secondary index, comparison*).
  - ▶ For  $\sigma_{A \geq v}(r)$  use index to find first index entry  $\geq v$  and scan index sequentially from there, to find pointers to records.
  - ▶ For  $\sigma_{A \leq v}(r)$  just scan leaf pages of index finding pointers to records, till first entry  $> v$
  - ▶ In either case, retrieve records that are pointed to
    - requires an I/O for each record
    - Linear file scan may be cheaper

13.12

## Implementation of Complex Selections

- **Conjunction:**  $\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$
- **A8** (*conjunctive selection using one index*).
  - Select a combination of  $\theta_i$  and algorithms A1 through A7 that results in the least cost for  $\sigma_{\theta_i}(r)$ .
  - Test other conditions on tuple after fetching it into memory buffer.
- **A9** (*conjunctive selection using multiple-key index*).
  - Use appropriate composite (multiple-key) index if available.
- **A10** (*conjunctive selection by intersection of identifiers*).
  - Requires indices with record pointers.
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
  - Then fetch records from file
  - If some conditions do not have appropriate indices, apply test in memory.

13.13

## Algorithms for Complex Selections

- **Disjunction:**  $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$ .
- **A11** (*disjunctive selection by union of identifiers*).
  - Applicable if *all* conditions have available indices.
    - Otherwise use linear scan.
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - Then fetch records from file
- **Negation:**  $\sigma_{\neg\theta}(r)$ 
  - Use linear scan on file
  - If very few records satisfy  $\neg\theta$ , and an index is applicable to  $\theta$ 
    - Find satisfying records using index and fetch from file

13.14

## Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Examples use the following information
  - Number of records of *customer*: 10,000    *depositor*: 5000
  - Number of blocks of *customer*: 400    *depositor*: 100

13.15

## Nested-Loop Join

- To compute the theta join  $r \bowtie_{\theta} s$ 
  - for each tuple  $t_r$  in  $r$  do begin**
  - for each tuple  $t_s$  in  $s$  do begin**
  - test pair  $(t_r, t_s)$  to see if they satisfy the join condition  $\theta$
  - if they do, add  $t_r \bullet t_s$  to the result.
  - end**
  - end**
- $r$  is called the **outer relation** and  $s$  the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.

13.16

## Nested-Loop Join (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is

$$n_r * b_s + b_r$$

block transfers, plus

$$n_r + b_r$$

seeks

- If the smaller relation fits entirely in memory, use that as the inner relation.
  - Reduces cost to  $b_r + b_s$  block transfers and 2 seeks
- Assuming worst case memory availability cost estimate is
  - with *depositor* as outer relation:
    - ▶  $5000 * 400 + 100 = 2,000,100$  block transfers,
    - ▶  $5000 + 100 = 5100$  seeks
  - with *customer* as the outer relation
    - ▶  $10000 * 100 + 400 = 1,000,400$  block transfers and 10,400 seeks
- If smaller relation (*depositor*) fits entirely in memory, the cost estimate will be 500 block transfers.
- Block nested-loops algorithm (next slide) is preferable.

13.17

## Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```
for each block  $B_r$  of  $r$  do begin
  for each block  $B_s$  of  $s$  do begin
    for each tuple  $t_r$  in  $B_r$  do begin
      for each tuple  $t_s$  in  $B_s$  do begin
        Check if  $(t_r, t_s)$  satisfy the join condition
        if they do, add  $t_r \bullet t_s$  to the result.
      end
    end
  end
end
```

13.18

## Block Nested-Loop Join (Cont.)

- Worst case estimate:  $b_r * b_s + b_r$  block transfers +  $2 * b_r$  seeks
  - Each block in the inner relation  $s$  is read once for each *block* in the outer relation (instead of once for each tuple in the outer relation)
- Best case:  $b_r + b_s$  block transfers + 2 seeks.
- Improvements to nested loop and block nested loop algorithms:
  - In block nested-loop, use  $M - 2$  disk blocks as blocking unit for outer relations, where  $M$  = memory size in blocks; use remaining two blocks to buffer inner relation and output
    - ▶ Cost =  $\lceil b_r / (M-2) \rceil * b_s + b_r$  block transfers +  $2 \lceil b_r / (M-2) \rceil$  seeks
  - If equi-join attribute forms a key or inner relation, stop inner loop on first match
  - Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement)
  - Use index on inner relation if available (next slide)

13.19

## Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
    - ▶ Can construct an index just to compute a join.
- For each tuple  $t_r$  in the outer relation  $r$ , use the index to look up tuples in  $s$  that satisfy the join condition with tuple  $t_r$ .
- Worst case: buffer has space for only one page of  $r$ , and, for each tuple in  $r$ , we perform an index lookup on  $s$ .
- Cost of the join:  $b_r (t_r + t_s) + n_r * c$ 
  - Where  $c$  is the cost of traversing index and fetching all matching  $s$  tuples for one tuple or  $r$
  - $c$  can be estimated as cost of a single selection on  $s$  using the join condition.
- If indices are available on join attributes of both  $r$  and  $s$ , use the relation with fewer tuples as the outer relation.

13.20

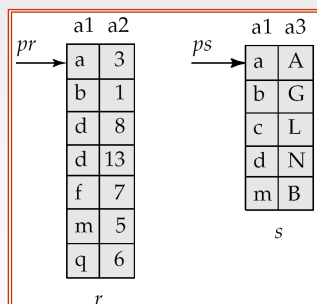
## Example of Nested-Loop Join Costs

- Compute  $depositor \bowtie customer$ , with  $depositor$  as the outer relation.
- Let  $customer$  have a primary B<sup>+</sup>-tree index on the join attribute  $customer-name$ , which contains 20 entries in each index node.
- Since  $customer$  has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
- $depositor$  has 5000 tuples
- Cost of block nested loops join
  - $400 * 100 + 100 = 40,100$  block transfers +  $2 * 100 = 200$  seeks
    - ▶ assuming worst case memory
    - ▶ may be significantly less with more memory
- Cost of indexed nested loops join
  - $100 + 5000 * 5 = 25,100$  block transfers and seeks.
  - CPU cost likely to be less than that for block nested loops join

13.21

## Merge-Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them
  1. Join step is similar to the merge stage of the sort-merge algorithm.
  2. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
  3. Detailed algorithm in book



13.22

## Merge-Join (Cont.)

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory)
- Thus the cost of merge join is:
  - $b_r + b_s$  block transfers +  $\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil$  seeks
  - + the cost of sorting if relations are unsorted.
- **hybrid merge-join:** If one relation is sorted, and the other has a secondary B<sup>+</sup>-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B<sup>+</sup>-tree .
  - Sort the result on the addresses of the unsorted relation's tuples
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
    - ▶ Sequential scan more efficient than random lookup

13.23

## Complex Joins

- Join with a conjunctive condition:

$$r \bowtie_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n} s$$

- Either use nested loops/block nested loops, or
- Compute the result of one of the simpler joins  $r \bowtie_{\theta_i} s$ 
  - ▶ final result comprises those tuples in the intermediate result that satisfy the remaining conditions

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

- Join with a disjunctive condition

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n} s$$

- Either use nested loops/block nested loops, or
- Compute as the union of the records in individual joins  $r \bowtie_{\theta_i} s$ :

$$(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \dots \cup (r \bowtie_{\theta_n} s)$$

13.24

## Other Operations

- **Duplicate elimination** can be implemented via hashing or sorting.
  - On sorting duplicates will come adjacent to each other, and all but one set of duplicates can be deleted.
  - *Optimization*: duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge.
  - Hashing is similar – duplicates will come into the same bucket.
- **Projection**:
  - perform projection on each tuple
  - followed by duplicate elimination.

13.25

## Other Operations : Aggregation

- **Aggregation** can be implemented in a manner similar to duplicate elimination.
  - Sorting or hashing can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
  - *Optimization*: combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values
    - ▶ For count, min, max, sum: keep aggregate values on tuples found so far in the group.
      - When combining partial aggregate for count, add up the aggregates
    - ▶ For avg, keep sum and count, and divide sum by count at the end

13.26

## Other Operations : Set Operations

- **Set operations** ( $\cup$ ,  $\cap$  and  $-$ ): can either use variant of merge-join after sorting, or variant of hash-join.
- E.g., Set operations using hashing:
  1. Partition both relations using the same hash function
  2. Process each partition  $i$  as follows.
    1. Using a different hashing function, build an in-memory hash index on  $r_i$ .
    2. Process  $s_i$  as follows
      - $r \cup s$ :
        1. Add tuples in  $s_i$  to the hash index if they are not already in it.
        2. At end of  $s_i$ , add the tuples in the hash index to the result.
      - $r \cap s$ :
        1. output tuples in  $s_i$  to the result if they are already there in the hash index
      - $r - s$ :
        1. for each tuple in  $s_i$ , if it is there in the hash index, delete it from the index.
        2. At end of  $s_i$ , add remaining tuples in the hash index to the result.

13.27

## 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
- We study above alternatives in more detail

13.28

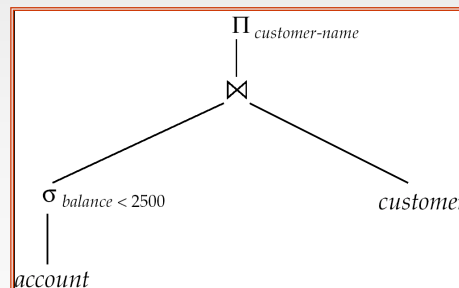
## Materialization

- **Materialized evaluation:** evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.

- E.g., in figure below, compute and store

$$\sigma_{balance < 2500}(account)$$

then compute the store its join with *customer*, and finally compute the projections on *customer-name*.



13.29

## Materialization (Cont.)

- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk, so
    - ▶ Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- **Double buffering:** use two output buffers for each operation, when one is full write it to disk while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time

13.30

## Pipelining

- **Pipelined evaluation** : evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of  $\sigma_{balance < 2500}(account)$ 
  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort, hash-join.
- 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**

13.31

## 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 **producer-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
  - System schedules operations that have space in output buffer and can process more input tuples
- Alternative name: **pull** and **push** models of pipelining

13.32

## Pipelining (Cont.)

- Implementation of demand-driven pipelining
  - Each operation is implemented as an **iterator** implementing the following operations
    - ▶ open()
      - E.g. file scan: initialize file scan
        - » state: pointer to beginning of file
      - E.g. merge join: sort relations;
        - » state: pointers to beginning of sorted relations
    - ▶ next()
      - E.g. for file scan: Output next tuple, and advance and store file pointer
      - E.g. for merge join: continue with merge from earlier state till next output tuple is found. Save pointers as iterator state.
    - ▶ close()

13.33

## Evaluation Algorithms for Pipelining

- Some algorithms are not able to output results even as they get input tuples
  - E.g. merge join, or hash join
  - intermediate results written to disk and then read back
- Algorithm variants 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 in
  - **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

13.34

## Complex Joins

- Join involving three relations:  $loan \bowtie depositor \bowtie customer$
- **Strategy 1.** Compute  $depositor \bowtie customer$ ; use result to compute  $loan \bowtie (depositor \bowtie customer)$
- **Strategy 2.** Compute  $loan \bowtie depositor$  first, and then join the result with  $customer$ .
- **Strategy 3.** Perform the pair of joins at once. Build and index on  $loan$  for  $loan-number$ , and on  $customer$  for  $customer-name$ .
  - For each tuple  $t$  in  $depositor$ , look up the corresponding tuples in  $customer$  and the corresponding tuples in  $loan$ .
  - Each tuple of  $depositor$  is examined exactly once.
- Strategy 3 combines two operations into one special-purpose operation that is more efficient than implementing two joins of two relations.

13.35

## Query Optimization

Based on slides from the book  
Database System Concepts, 5th Ed.

## Query Planning/Optimization

- Generation of query-evaluation plans for an expression involves several steps:
  1. Generating logically equivalent expressions using **equivalence rules**.
  2. Annotating resultant expressions to get alternative query plans
  3. Choosing the cheapest plan based on **estimated cost**
- The overall process is called **cost based optimization**.

13.37

## Transformation of Relational Expressions

- Two relational algebra expressions are said to be **equivalent** if on every legal database instance the two expressions generate the same set of tuples
  - Note: order of tuples is irrelevant

13.38

## Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are commutative.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted.

$$\Pi_{L_1}(\Pi_{L_2}(\dots(\Pi_{L_n}(E))\dots)) = \Pi_{L_1}(E)$$

4. Selections can be combined with Cartesian products and theta joins.

- a.  $\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$

- b.  $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

13.39

## Equivalence Rules (Cont.)

5. Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$$

6. (a) Natural join operations are associative:

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

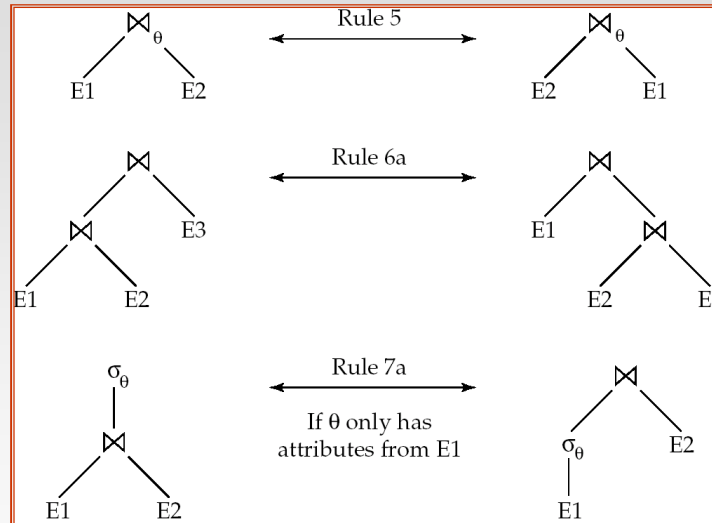
- (b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

where  $\theta_2$  involves attributes from only  $E_2$  and  $E_3$ .

13.40

## Pictorial Depiction of Equivalence Rules



13.41

## Equivalence Rules (Cont.)

7. The selection operation distributes over the theta join operation under the following two conditions:

- (a) When all the attributes in  $\theta_0$  involve only the attributes of one of the expressions ( $E_1$ ) being joined.

$$\sigma_{\theta_0}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_{\theta} E_2$$

- (b) When  $\theta_1$  involves only the attributes of  $E_1$  and  $\theta_2$  involves only the attributes of  $E_2$ .

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta} (\sigma_{\theta_2}(E_2))$$

13.42

## Equivalence Rules (Cont.)

8. The projections operation distributes over the theta join operation as follows:

(a) if  $\Pi$  involves only attributes from  $L_1 \cup L_2$ :

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) = (\Pi_{L_1}(E_1)) \bowtie_{\theta} (\Pi_{L_2}(E_2))$$

(b) Consider a join  $E_1 \bowtie_{\theta} E_2$ .

- Let  $L_1$  and  $L_2$  be sets of attributes from  $E_1$  and  $E_2$ , respectively.
- Let  $L_3$  be attributes of  $E_1$  that are involved in join condition  $\theta$ , but are not in  $L_1 \cup L_2$ , and
- let  $L_4$  be attributes of  $E_2$  that are involved in join condition  $\theta$ , but are not in  $L_1 \cup L_2$ .

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_{\theta} (\Pi_{L_2 \cup L_4}(E_2)))$$

13.43

## Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative

$$\begin{aligned} E_1 \cup E_2 &= E_2 \cup E_1 \\ E_1 \cap E_2 &= E_2 \cap E_1 \end{aligned}$$

- (set difference is not commutative).

10. Set union and intersection are associative.

$$\begin{aligned} (E_1 \cup E_2) \cup E_3 &= E_1 \cup (E_2 \cup E_3) \\ (E_1 \cap E_2) \cap E_3 &= E_1 \cap (E_2 \cap E_3) \end{aligned}$$

11. The selection operation distributes over  $\cup$ ,  $\cap$  and  $-$ .

$$\sigma_{\theta}(E_1 - E_2) = \sigma_{\theta}(E_1) - \sigma_{\theta}(E_2)$$

and similarly for  $\cup$  and  $\cap$  in place of  $-$

Also: 
$$\sigma_{\theta}(E_1 - E_2) = \sigma_{\theta}(E_1) - E_2$$

and similarly for  $\cap$  in place of  $-$ , but not for  $\cup$

12. The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

13.44

## Transformation Example

- Query: Find the names of all customers who have an account at some branch located in Brooklyn.

$$\Pi_{customer\_name}(\sigma_{branch\_city='Brooklyn'}(branch \bowtie (account \bowtie depositor)))$$

- Transformation using rule 7a.

$$\Pi_{customer\_name}((\sigma_{branch\_city='Brooklyn'}(branch)) \bowtie (account \bowtie depositor))$$

- Performing the selection as early as possible reduces the size of the relation to be joined.

13.45

## Example with Multiple Transformations

- Query: Find the names of all customers with an account at a Brooklyn branch whose account balance is over \$1000.

$$\Pi_{customer\_name}(\sigma_{branch\_city='Brooklyn' \wedge balance > 1000}(branch \bowtie (account \bowtie depositor)))$$

- Transformation using join associatively (Rule 6a):

$$\Pi_{customer\_name}((\sigma_{branch\_city='Brooklyn' \wedge balance > 1000}(branch \bowtie account)) \bowtie depositor)$$

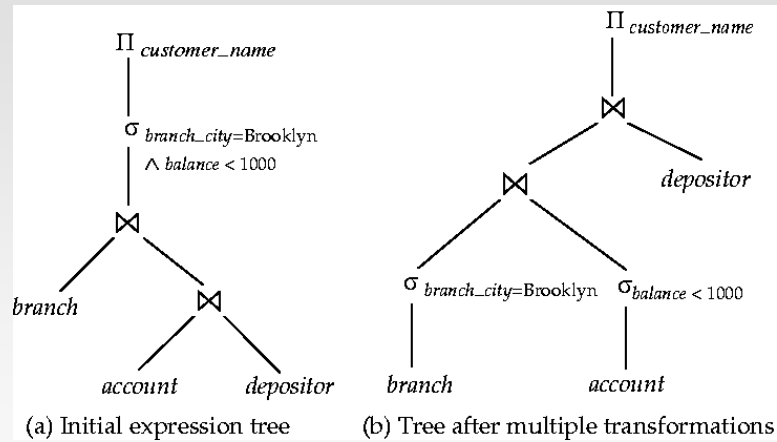
- Second form provides an opportunity to apply the “perform selections early” rule, resulting in the subexpression

$$\sigma_{branch\_city='Brooklyn'}(branch) \bowtie \sigma_{balance > 1000}(account)$$

- Thus a sequence of transformations can be useful

13.46

## Multiple Transformations (Cont.)



13.47

## Join Ordering Example

- For all relations  $r_1$ ,  $r_2$ , and  $r_3$ ,

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

- If  $r_2 \bowtie r_3$  is quite large and  $r_1 \bowtie r_2$  is small, we choose

$$(r_1 \bowtie r_2) \bowtie r_3$$

so that we compute and store a smaller temporary relation.

13.48

## Cost Estimation

- Cost of each operator computer as described in Chapter 13
  - Need statistics of input relations
    - E.g. number of tuples, sizes of tuples
- Inputs can be results of sub-expressions
  - Need to estimate statistics of expression results
  - To do so, we require additional statistics
    - E.g. number of distinct values for an attribute
- More on cost estimation later

13.49

## Statistical Information for Cost Estimation

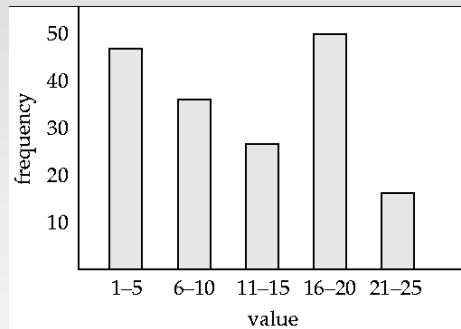
- $n_r$ : number of tuples in a relation  $r$ .
- $b_r$ : number of blocks containing tuples of  $r$ .
- $l_r$ : size of a tuple of  $r$ .
- $f_r$ : blocking factor of  $r$  — i.e., the number of tuples of  $r$  that fit into one block.
- $V(A, r)$ : number of distinct values that appear in  $r$  for attribute  $A$ ; same as the size of  $\Pi_A(r)$ .
- If tuples of  $r$  are stored together physically in a file, then:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

13.50

## Histograms

- Histogram on attribute *age* of relation *person*



- Equi-width histograms
- Equi-depth histograms

13.51

## Selection Size Estimation

- $\sigma_{A=v}(r)$ 
  - ▶  $n_r / V(A,r)$  : number of records that will satisfy the selection
  - ▶ Equality condition on a key attribute: *size estimate* = 1
- $\sigma_{A \leq v}(r)$  (case of  $\sigma_{A \geq v}(r)$  is symmetric)
  - Let  $c$  denote the estimated number of tuples satisfying the condition.
  - If  $\min(A,r)$  and  $\max(A,r)$  are available in catalog
    - ▶  $c = 0$  if  $v < \min(A,r)$
    - ▶  $c = n_r \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$
  - If histograms available, can refine above estimate
  - In absence of statistical information  $c$  is assumed to be  $n_r/2$ .

13.52

## Size Estimation of Complex Selections

- The **selectivity** of a condition  $\theta_i$  is the probability that a tuple in the relation  $r$  satisfies  $\theta_i$ .
  - If  $s_i$  is the number of satisfying tuples in  $r$ , the selectivity of  $\theta_i$  is given by  $s_i/n_r$ .
- **Conjunction:**  $\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$ . Assuming independence, estimate of tuples in the result is:

$$n_r * \frac{s_1 * s_2 * \dots * s_n}{n_r^n}$$

- **Disjunction:**  $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$ . Estimated number of tuples:

$$n_r * \left( 1 - \left( 1 - \frac{s_1}{n_r} \right) * \left( 1 - \frac{s_2}{n_r} \right) * \dots * \left( 1 - \frac{s_n}{n_r} \right) \right)$$

- **Negation:**  $\sigma_{\neg \theta}(r)$ . Estimated number of tuples:  $n_r - \text{size}(\sigma_{\theta}(r))$

13.53

## Join Operation: Running Example

Running example:  
 $depositor \bowtie customer$

Catalog information for join examples:

- $n_{customer} = 10,000$ .
- $f_{customer} = 25$ , which implies that  $b_{customer} = 10000/25 = 400$ .
- $n_{depositor} = 5000$ .
- $f_{depositor} = 50$ , which implies that  $b_{depositor} = 5000/50 = 100$ .
- $V(customer\_name, depositor) = 2500$ , which implies that, on average, each customer has two accounts.
  - Also assume that  $customer\_name$  in  $depositor$  is a foreign key on  $customer$ .
  - $V(customer\_name, customer) = 10000$  (primary key!)

13.54

## Estimation of the Size of Joins

- The Cartesian product  $r \times s$  contains  $n_r \cdot n_s$  tuples; each tuple occupies  $s_r + s_s$  bytes.
- If  $R \cap S = \emptyset$ , then  $r \bowtie s$  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  $r \bowtie s$  is no greater than the number of tuples in  $s$ .
- If  $R \cap S$  in  $S$  is a foreign key in  $S$  referencing  $R$ , then the number of tuples in  $r \bowtie s$  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  $depositor \bowtie customer$ ,  $customer\_name$  in  $depositor$  is a foreign key of  $customer$ 
  - hence, the result has exactly  $n_{depositor}$  tuples, which is 5000

13.55

## 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  $t$  in  $R$  produces tuples in  $R \bowtie S$ , the number of tuples in  $R \bowtie S$  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.

- Can improve on above if histograms are available
  - Use formula similar to above, for each cell of histograms on the two relations

13.56

## Estimation of the Size of Joins (Cont.)

- Compute the size estimates for  $depositor \bowtie customer$  without using information about foreign keys:
  - $V(customer\_name, depositor) = 2500$ , and  $V(customer\_name, customer) = 10000$
  - 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.

13.57

## Size Estimation for Other Operations

- Projection: estimated size of  $\Pi_A(r) = V(A, r)$
- Aggregation: estimated size of  $\mathcal{A}_F(r) = V(A, r)$
- 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  $\sigma_{\theta_1} \sigma_{\theta_2}(r)$
  - For operations on different relations:
    - ▶ estimated size of  $r \cup s = \text{size of } r + \text{size of } s$ .
    - ▶ estimated size of  $r \cap s = \text{minimum size of } r \text{ and size of } s$ .
    - ▶ estimated size of  $r - s = r$ .
    - ▶ All the three estimates may be quite inaccurate, but provide upper bounds on the sizes.

13.58

## Size Estimation (Cont.)

- Outer join:
  - Estimated size of  $r \bowtie s = \text{size of } r \bowtie s + \text{size of } r$ 
    - ▶ Case of right outer join is symmetric
  - Estimated size of  $r \bowtie \sqcup s = \text{size of } r \bowtie s + \text{size of } r + \text{size of } s$

13.59

## Estimation of Number of Distinct Values

Selections:  $\sigma_\theta(r)$

- If  $\theta$  forces  $A$  to take a specified value:  $V(A, \sigma_\theta(r)) = 1$ .
  - ▶ e.g.,  $A = 3$
- If  $\theta$  forces  $A$  to take on one of a specified set of values:
  - $V(A, \sigma_\theta(r)) = \text{number of specified values}$ .
  - ▶ (e.g.,  $(A = 1 \vee A = 3 \vee A = 4)$ ),
- If the selection condition  $\theta$  is of the form  $A \text{ op } r$ 
  - estimated  $V(A, \sigma_\theta(r)) = V(A.r) * s$
  - ▶ where  $s$  is the selectivity of the selection.
- In all the other cases: use approximate estimate of
  - $\min(V(A,r), n_{\sigma_\theta(r)})$
  - More accurate estimate can be got using probability theory, but this one works fine generally

13.60

## Estimation of Distinct Values (Cont.)

Joins:  $r \bowtie s$

- If all attributes in  $A$  are from  $r$   
estimated  $V(A, r \bowtie s) = \min(V(A, r), n_{r \bowtie s})$
- If  $A$  contains attributes  $A1$  from  $r$  and  $A2$  from  $s$ , then estimated  $V(A, r \bowtie s) =$   
 $\min(V(A1, r) * V(A2 - A1, s), V(A1 - A2, r) * V(A2, s), n_{r \bowtie s})$ 
  - More accurate estimate can be got using probability theory, but this one works fine generally

13.61

## Estimation of Distinct Values (Cont.)

- Estimation of distinct values are straightforward for projections.
  - They are the same in  $\Pi_A(r)$  as in  $r$ .
- The same holds for grouping attributes of aggregation.
- For aggregated values
  - For  $\min(A)$  and  $\max(A)$ , the number of distinct values can be estimated as  $\min(V(A, r), V(G, r))$  where  $G$  denotes grouping attributes
  - For other aggregates, assume all values are distinct, and use  $V(G, r)$

13.62

## Searching for the best plan

- Option 1:
  - Enumerate all equivalent expressions for the original query expression
    - ▶ Using the rules outlined earlier
  - Estimate cost for each and choose the lowest
  
- Too expensive !
  - Consider finding the best join-order for  $r_1 \bowtie r_2 \bowtie \dots \bowtie r_n$ .
  - With  $n = 7$ , the number is 665280, with  $n = 10$ , the number is greater than 176 billion!

13.63

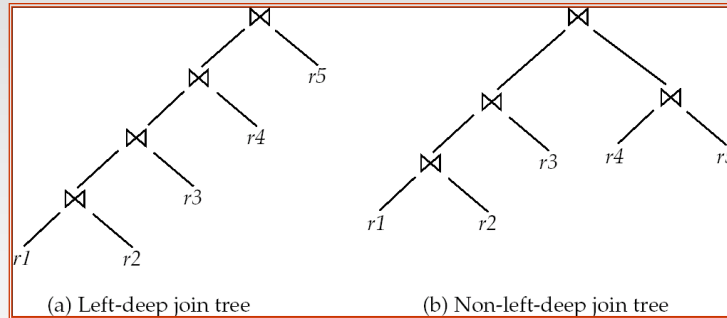
## Searching for the best plan

- Option 2:
  - Dynamic programming
    - ▶ There is too much commonality between the plans
    - ▶ Also, costs are additive
      - Caveat: Sort orders (also called “interesting orders”)
      - E.g. if a child operator to a sort-merge join produces results in the required sorted order, the cost of sort-merge join is lower
  - Reduces the cost down to  $O(n^3n)$  or  $O(n^2n)$  in most cases
    - ▶ Interesting orders increase this a little bit
  - Considered acceptable
    - ▶ Typically  $n < 10$ .
  - Switch to heuristic if not acceptable.

13.64

## Left Deep Join Trees

- In **left-deep join trees**, the right-hand-side input for each join is a relation, not the result of an intermediate join.



13.65

## Heuristic Optimization

- Cost-based optimization is expensive, even with dynamic programming.
- Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
- Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations before other similar operations.
  - Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

13.66

## Steps in Typical Heuristic Optimization

1. Deconstruct conjunctive selections into a sequence of single selection operations (Equiv. rule 1.).
2. Move selection operations down the query tree for the earliest possible execution (Equiv. rules 2, 7a, 7b, 11).
3. Execute first those selection and join operations that will produce the smallest relations (Equiv. rule 6).
4. Replace Cartesian product operations that are followed by a selection condition by join operations (Equiv. rule 4a).
5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed (Equiv. rules 3, 8a, 8b, 12).
6. Identify those subtrees whose operations can be pipelined, and execute them using pipelining).

13.67

## Structure of Query Optimizers

- The System R/Starburst optimizer considers only left-deep join orders. This reduces optimization complexity and generates plans amenable to pipelined evaluation. System R/Starburst also uses heuristics to push selections and projections down the query tree.
- Heuristic optimization used in some versions of Oracle:
  - Repeatedly pick “best” relation to join next
    - Starting from each of n starting points. Pick best among these.
- For scans using secondary indices, some optimizers take into account the probability that the page containing the tuple is in the buffer.
- Intricacies of SQL complicate query optimization
  - E.g. nested subqueries

13.68

## Structure of Query Optimizers (Cont.)

- Some query optimizers integrate heuristic selection and the generation of alternative access plans.
  - System R and Starburst use a hierarchical procedure based on the nested-block concept of SQL: heuristic rewriting followed by cost-based join-order optimization.
- Even with the use of heuristics, cost-based query optimization imposes a substantial overhead.
- This expense is usually more than offset by savings at query-execution time, **particularly by reducing the number of slow disk accesses.**