Database Systems
Spring 2013

Query Processing and Query Optimization
SL08

- Query Processing
  - Sorting, Partitioning
  - Selection, Join
- Query Optimization
  - Cost estimation
  - Rewriting of relational algebra expressions
  - Rule- and cost-based query optimization

Literature and Acknowledgments

Reading List for SL08:

- Database Systems, Chapter 18, Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Education, 2010.

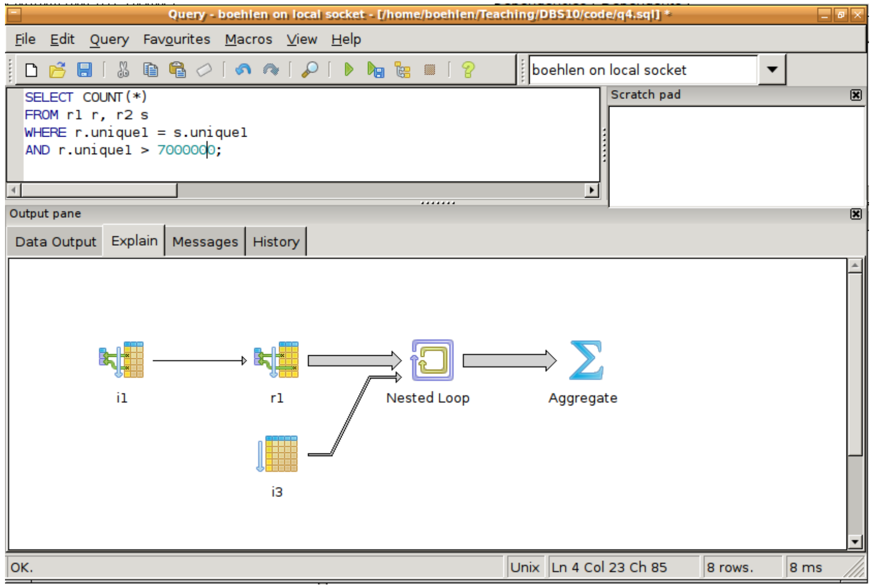
These slides were developed by:

- Michael Böhlen, University of Zürich, Switzerland
- Johann Gamper, Free University of Bozen-Bolzano, Italy

The slides are based on the following text books and associated material:

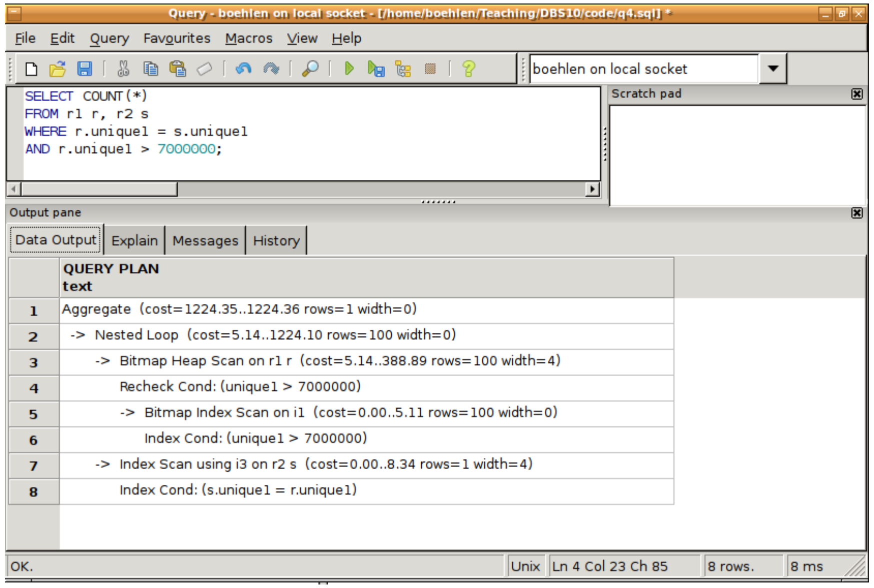
- Database Systems, Sixth Edition, Ramez Elmasri and Shamkant B. Navathe, Pearson Education, 2010.
- A. Silberschatz, H. Korth, and S. Sudarshan: Database System Concepts, McGraw Hill, 2006.

PostgreSQL Example/1



```
SELECT COUNT(*)
FROM r1 r, r2 s
WHERE r.unique1 = s.unique1
AND r.unique1 > 7000000;
```

PostgreSQL Example/2

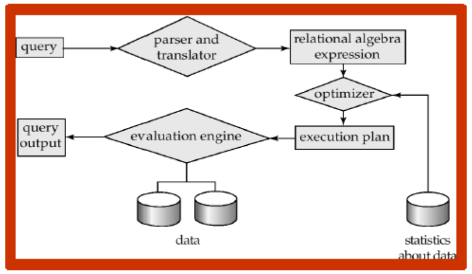

Query - boehlen on local socket - [/home/boehlen/Teaching/DBS10/code/q4.sql] *
File Edit Query Favourites Macros View Help
boehlen on local socket
Scratch pad
Output pane
Data Output
Explain
Messages
History
OK.
Unix
Ln 4 Col 23 Ch 85
8 rows.
8 ms

```
SELECT COUNT(*)
FROM r1 r, r2 s
WHERE r.unique1 = s.unique1
AND r.unique1 > 7000000;
```

```
   QUERY PLAN
   text
1  Aggregate  (cost=1224.35..1224.36 rows=1 width=0)
2    -> Nested Loop  (cost=5.14..1224.10 rows=100 width=0)
3         -> Bitmap Heap Scan on r1 r  (cost=5.14..388.89 rows=100 width=4)
4              Recheck Cond: (unique1 > 7000000)
5              -> Bitmap Index Scan on i1  (cost=0.00..5.11 rows=100 width=0)
6                   Index Cond: (unique1 > 7000000)
7         -> Index Scan using i3 on r2 s  (cost=0.00..8.34 rows=1 width=4)
8              Index Cond: (s.unique1 = r.unique1)
```

Query Processing and Optimization

- One of the most important tasks of a DBMS is to figure out an efficient **evaluation plan** (also termed **execution plan** or **access plan**) for high level statements.
  - It is particularly important to have evaluation strategies for:
  - Selections (search conditions)
  - Joins (combining information in relational database)
- Query processing is a 3-step process:
  1. Parsing and translation (from SQL to RA)
  2. Optimization (refine RA expression)
  3. Evaluation (exec RA operators)



Query Processing

- Measuring the query costs
- Sorting
- Optimizing selections
- Optimizing joins

Measuring the Query Costs/1

- **Query cost** is generally measured as the **total elapsed time** for answering a query.
- Many factors contribute to time cost and are considered in real DBMS, including
  - CPU cost and network communication
  - Disk access
    - Difference between sequential and random I/O
  - Buffer Size
    - Having more memory reduces need for disk access
    - Amount of real memory available for buffers depends on other concurrent OS procsesses, and is difficult to determine ahead of actual execution.
    - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available

Measures of Query Cost/2

- Typically **disk access is the predominant cost**, which is relatively easy to estimate. The cost of disk accesses is 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, since data is read back after being written to ensure that the write was successful
- For simplicity
  - we just use **number of block transfers from disk** as the cost measure, and
  - we do not include cost of **writing output to disk**

Sorting

- **Sorting** is important for for several reasons:
  - SQL queries can specify that the output is sorted
  - Several relational operations can be implemented efficiently if the input relations are first sorted, e.g., joins
  - Often sorting is a crucial first step for efficient algorithms
- We may build an index on the relation, and then use the index to read the relation in sorted order.
  - With an index sorting is only logical and not physical. This might lead to one disk block access for each tuple (can be very expensive)
    - It may be desirable/necessary to order the records physically.
- Relation fits in memory: Use techniques like **quicksort**
- Relation does not fit in main memory: Use external sorting, e.g., **external sort-merge** is a good choice

External Sort-Merge/1

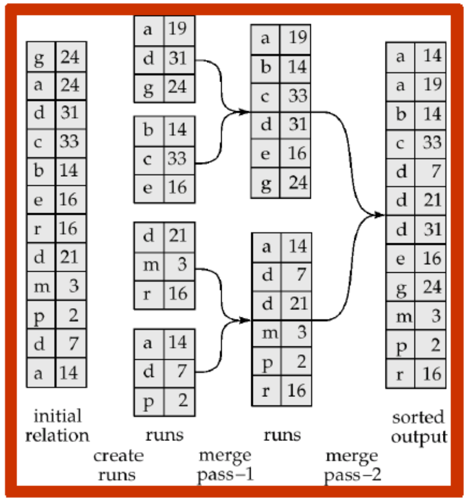
- **Step 1: Create N sorted runs** (M is # blocks in buffer)
  1. Let i be 0 initially.
  2. Repeatedly do the following until the end of the relation
     2.1 Read M blocks of the relation (or the rest) into memory
     2.2 Sort the in-memory blocks
     2.3 Write sorted data to run file R_i;
     2.4 Increment i.
- **Step 2: Merge runs (N-way merge)** (assume N < M)
  (Use N blocks in memory to buffer input runs, and 1 block to buffer output)
  1. Read the first block of each run R_i into its buffer page
  2. **Repeat until** all input buffer pages are empty
     2.1 Select the first record (in sort order) among all buffer pages
     2.2 Write the record to the output buffer. If the output buffer is full write it to disk.
     2.3 Delete the record from its input buffer page.
     2.4 **If** the buffer page becomes empty **then**
         read the next block (if any) of the run into the buffer

External Sort-Merge/2

- If $N \geq M$, **several merge passes** (step 2) are required:
  - In each pass, contiguous groups of $M - 1$ runs are merged
  - A pass reduces the number of runs by a factor of $M - 1$, and creates runs longer by the same factor.
    - E.g. If $M = 11$, and there are 90 runs, one pass reduces the number of runs to 9, each run being 10 times the size of the initial runs
  - Repeated passes are performed until all runs have been merged into one.

External Sort-Merge/3

- **Example**: M = 3, 1 block = 1 tuple


g 24, a 24, d 31, c 33, b 14, e 16, r 16, d 21, m 3, p 2, d 7, a 14
a 19, d 31, g 24
b 14, c 33, e 16
d 21, m 3, r 16
a 14, d 7, p 2
a 19, b 14, c 33, d 31, e 16, g 24
a 14, d 7, d 21, m 3, p 2, r 16
a 14, a 19, b 14, c 33, d 7, d 21, d 31, e 16, g 24, m 3, p 2, r 16
initial relation
runs
runs
sorted output
create runs
merge pass–1
merge pass–2

External Sort-Merge/4

- **Cost analysis**
  - b_r = number of blocks in r
  - Initial number of runs: b_r/M
  - Total number of merge passes required: $\lceil log_{M-1}(b_r/M) \rceil$
    - The number of runs decreases by a factor of *M-1* in each merge pass
  - Disk accesses for initial run creation and in each pass is $2b_r$
    - Exception: For final pass there is no write cost
  - Thus total number of disk accesses for external sorting:
    $Cost = b_r \ (2 \lceil log_{M-1}(b_r/M) \rceil + 1)$
- **Example:** Cost analysis of previous example
  - 12 (2 * 2 + 1) = 60 disk block transfers

Selection Evaluation Strategies/1

- The selection operator:
  - **select** * **from** r **where** θ
  - $\sigma_\theta(r)$

  is used to retrieve those records that satisfy the selection condition
- The strategy/algorithm for the evaluation of the selection operator depends
  - on the type of the selection condition
  - on the available index structures

Review 8.1

Assume a B+ tree index on (BrName, BrCity). What would be the best way to evaluate the query:

$$\sigma_{BrCity<'Brighton' \ \wedge \ Assets<5000 \ \wedge \ BrName='Downtown'}(branch)$$

Selection Evaluation Strategies/2

Types of selection conditions:

- **Equality queries**: $\sigma_{a=v}(r)$
- **Range queries**: $\sigma_{a\leq v}(r)$ or $\sigma_{a\geq v}(r)$
  - Can be implemented by using
    - linear file scan
    - binary search
    - using indices
- **Conjunctive selection**: $\sigma_{\theta_1\wedge\theta_2\cdots\wedge\theta_n}$(r)
- **Disjunctive selection**: $\sigma_{\theta_1\vee\theta_2\cdots\vee\theta_n}$(r)

Selection Evaluation Strategies/3

Basic search methods for selection operator:

- **File scan**
  - Class of search algorithms that **read the file line by line** to locate and retrieve records that fulfill a selection condition, i.e., σ_θ(r)
  - Lowest-level operator to access data
- **Index scan**
  - Class of search algorithms that **use an index**
  - Assume B+ tree index and equality conditions, i.e., $\sigma_{a=v}$(r)

Selection Evaluation Strategies/4

- **A1 Linear search:** Scan each file block and test all records to see whether they satisfy the selection condition.
  - Fairly expensive, but always applicable (regardless of indexes, ordering, selection condition, etc)
  - Fetching a contiguous range of blocks from disk has been optimized by disk manufacturers and is cheap in terms of seek time and rotational delay (pre-fetching)
  - Cost estimate (b_r = number of blocks in file):
    - Worst case: $Cost = b_r$
    - If the selection is on a key attribute: $Average\ cost = b_r/2$ (stop when finding record)

Selection Evaluation Strategies/5

- **A2 Binary search:** Apply binary search to locate records that satisfy selection condition.
  - Only applicable if
    - the blocks of a relation are stored contiguously (very rare), and
    - the selection condition is a comparison on the attribute on which the file is ordered
  - Cost estimate for $\sigma_{a=v}(r)$:
    - $\lceil log_2(b_r) \rceil$ — cost of locating the first tuple by a binary search on the blocks
    - Plus number of blocks containing records that satisfy selection condition

Selection Evaluation Strategies/6

- **A3 Primary index + equality on candidate key**
  - Retrieve a single record that satisfies the equality condition
  - $Cost = HT_i + 1$ (height of B+ tree + 1 data block)
- **A4 Primary index + equality on non-candidate key**
  - Retrieve multiple records, where records are on consecutive blocks
  - $Cost = HT_i + \#$ blocks with records with given search key
- **A5 Secondary index + equality on search-key**
  - Retrieve a single record if the search-key is a candidate key
    - $Cost = HT_i + 1$
  - Retrieve multiple records if search-key is not a candidate key
    - $Cost = HT_i + \#$ buckets with search-key value $+ \#$ retrieved records
    - Can be very expensive, since each record may be on a different block
    - Linear file scan may be cheaper if many records have to be fetched

Selection Evaluation Strategies/7

- **A6 Primary index on A + comparison condition**
  - $\sigma_{a \geq v}$: Use index to find first tuple $\geq v$; then scan relation sequentially
  - $\sigma_{a \leq v}$: Scan relation sequentially until first tuple $> v$; do not use index.
- **A7 Secondary index on A + comparison cond.**
  - $\sigma_{a \geq v}$: Use index to find first index entry $\geq v$; scan index sequentially from there, to find pointers to records.
  - $\sigma_{a \leq v}$: Scan leaf pages of index finding record pointers until first entry $> v$
  - Requires in the worst case one I/O for each record; linear file scan may be cheaper if many records are to be fetched

Review 8.2

Consider relations $r1(\underline{A}, B, C)$, $r2(\underline{C}, D, E)$, $r3(\underline{E}, F)$ with keys underlined and cardinalities $|r1| = 1000$, $|r2| = 1500$, $|r3| = 750$.

- Estimate the size of $r1 \bowtie r2 \bowtie r3$
- Give an efficient strategy for computing the result and compute its cost

Join Evaluation Strategies

- There exist several different algorithms for the evaluation of join operations:
  - Nested loop join
  - Block nested loop join
  - Indexed nested loop join
  - Merge join
  - Hash join
- Choice based on cost estimate
- Examples use the following relations:
  - customer = (CustName, CustStreet, CustCity)
    - Number of records: $n_c = 10'000$
    - Number of blocks: $b_c = 400$
  - depositor = (CustName, AccNumber)
    - Number of records: $n_d = 5'000$
    - Number of blocks: $b_d = 100$

Nested Loop Join/1

- Compute the theta join: r $\bowtie_\theta$ s

```
for each tuple $t_r$ in $r$ do
    for each tuple $t_s$ in $s$ do
        test pair $(t_r, t_s)$ to see if they satisfy the join condition $\theta$
        if they do, add $t_r \circ t_s$ to the result.
    end
end
```

- r is called the outer relation, s the inner relation of the join.
- Always applicable. Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples.

Nested Loop Join/2

- Order of r and s important: Relation r is read once, relation s is read up to $|r|$ times
  - **Worst case:** Only one block of each relation fits in main memory
    $Cost = n_r * b_s + b_r$
  - If the smaller relation fits entirely in memory, use that as the inner relation.
    $Cost = b_s + b_r$
- **Example:**
  - Depositor as outer relation:
    5'000 * 400 + 100 = 2'000'100 block accesses
  - Customer as outer relation:
    10'000 * 100 + 400 = 1'000'400 block accesses
  - Smaller relation *(depositor)* fits into memory:
    400 + 100 = 500 blocks

Block Nested Loop Join/1

- Simple nested loop algorithm is not used directly since it is not block-based.
- Variant of nested loop join in which every block of the inner relation is paired with every block of the outer relation.

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

Block Nested Loop Join/2

- Worst case: $Cost = b_r * b_s + b_r$
  - 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: $Cost = b_s + b_r$
- **Example:** Compute depositor ⋈ customer, with depositor as the outer relation.
  - Block nested loop join:
    Cost = 100 * 400 + 100 = 40'100 blocks (worst case)

Block Nested Loop Join/3

- Improvements to nested loop and block nested loop algorithms (M is the number of main memory blocks):
  - Block nested loop: Use M-2 disk blocks for outer relation and two blocks to buffer inner relation and output; join each block of the inner relation with M-2 blocks of the outer relation.
    - $Cost = \lceil b_r/(M-2) \rceil * b_s + b_r$
  - If equi-join attribute forms a key on 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).

Indexed Nested Loop Join/1

- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - index is available on the inner relation's join attribute
  - index can be constructed 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 perform an index lookup on s.
  - $Cost = n_r * c + b_r$
    - c is the cost of traversing the index and fetching all matching s tuples for one tuple of r
    - c can be estimated as cost of a single selection on s using the join condition.
- If indexes are available on join attributes of both r and s, use relation with fewer tuples as the outer relation.

Indexed Nested Loop Join/2

- **Example:** Compute depositor x customer, with depositor as the outer relation.
  - Let customer have a primary B+ tree index on the join attribute CustName, 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 5'000 tuples and 100 blocks
  - Indexed nested loops join:
    Cost = 5'000 * 5 + 100 = 25'100 disk accesses.

Review 8.3

Consider $E \bowtie_{SSN=MgrSSN} D$ with $r_D = 50$ (number of tuples in relation D), $r_E = 5000$, $b_D = 10$ (number of blocks for relation D), $b_E = 2000$, $n_B = 6$ (number of available buffer blocks).
Compute the number of IOs for the following evaluation strategies:

1. Block NL, $E \bowtie D$, 4 blocks for E (1 block for D, 1 block for result)

2. Block NL, $E \bowtie D$, 4 blocks for D

Review 8.3

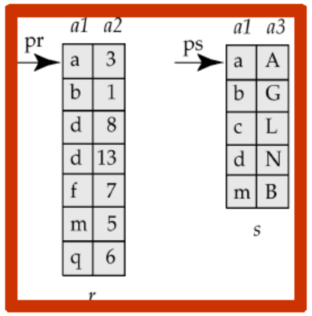
3. Block NL, $D \bowtie E$, 4 blocks for D

4. Indexed NL, $E \bowtie D$

5. Indexed NL, $D \bowtie E$

Merge Join/1

- Basic idea of **merge join**: Use two pointers *pr* and *ps* that are initialized to the first tuple in *r* and *s* and move in a synchronized way through the sorted relations.
- Algorithm
  1. Sort both relations on their join attributes (if not already sorted on the join attribute).
  2. Scan *r* and *s* in sort order and return matching tuples.
  3. Move the tuple pointer of the relation that is less far advanced in sort order (more complicated if the join attributes are not unique - every pair with same value on join attribute must be matched).



a1	a2
a	3
b	1
d	8
d	13
f	7
m	5
q	6

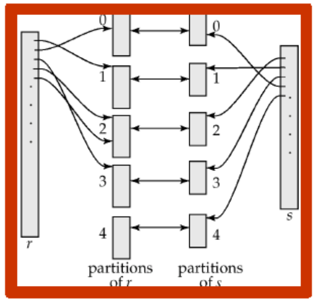
a1	a3
a	A
b	G
c	L
d	N
m	B

Merge Join/2

- Applicable for equi-joins and natural joins only
- If all tuples for any given value of the join attributes fit in memory
  - One file scan of *r* and *s* is enough
  - Cost $= b_r + b_s$ (+ the cost of sorting if relations are not sorted)
- Otherwise, a block nested loop join must be performed between the tuples with the same attributes
- If the relation are not sorted appropriately we first have to sort them. The combined operator is called a **sort-merge join**.

Hash Join/1

- Applicable for equi-joins and natural joins only.
- Partition tuples of r and s using the same hash function *h*, which maps the values of the join attributes to the set 0, 1, ..., n
  - Partitions of r-tuples: $r_0, r_1, ..., r_n$
    - All $t_r \in$ r with h(t_r*[JoinAttrs]*) $= i$ are put in r_i
  - Partitions of s-tuples: $s_0, s_1, .., s_n$
    - All $t_s \in$ s with h(t_s*[JoinAttrs]*) $= i$ are put in s_i
- *r*-tuples in r_i need only to be compared with *s*-tuples in s_i
  - an *r*-tuples and *s*-tuples that satisfy the join condition have the same hash value *i*, and are mapped to r_i and s_i, respectively.



Hash Join/2

- **Algorithm** for the hash join of *r* and *s*
  1. Partition the relation s using hash function h. (When partitioning a relation, one block of memory is reserved as the output buffer for each partition.)
  2. Partition *r* similarly.
  3. For each *i*:
     - 3.1 Load s_i into memory and **build** an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one *h*.
     - 3.2 Read the tuples in r_i from the disk (block by block). For each tuple t_r **probe** (locate) each matching tuple t_s in s_i using the in-memory hash index. Output the concatenation of their attributes as result tuple.
- Relation *s* is called the **build input** and *r* is called the **probe input**.

Hash Join/3

- **Cost analysis** of hash join
  - Partitioning of the two relations: 2 * $(b_r + b_s)$
    - Complete reading of the two relations plus writing back
  - The build and probe phases read each of the partitions once: $b_r + b_s$
  - Cost $= 3 * (b_r + b_s)$
- **Example:** *customer* ⋈ *depositor*
  - Assume that memory size is 20 blocks
  - $b_d = 100$ and $b_c = 400$.
  - *depositor* is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass.
  - Similarly, partition customer into five partitions, each of size 80. This is also done in one pass.
    - Partition size of probe relation needs not to fit into main memory!
  - Therefore total cost = 3 * (100 + 400) = 1500 block transfers
    - Ignores cost of writing partially filled blocks

Review 8.4

Consider $b_C = 400$, $n_C = 10'000$, $b_D = 100$, $n_D = 5'000$, disk IO time = 10 msec, memory access time = 60 nsec. Compare the execution times for NL (best case) and sort merge.

Query Optimization

- Cost estimation
- Transformation of relational algebra expressions (rewrite rules)
- Rule-based (aka heuristic) query optimization
- Cost-based query optimization

Query Optimization/1

- Alternative ways of evaluating a query because of
  - Equivalent expressions
  - Different algorithms for each operation
- A **query evaluation plan** (query plan) is an annotated RA expression that specifies for each operator how to evaluate it.
- The cost difference between a good and a bad query evaluation plan can be enormous
  - e.g., performing $r \times s$ followed by a selection r.A = s.B is much slower than performing a join on the same condition
- The query optimizer needs to estimate the cost of operations
  - Depends critically on statistical information about relations
  - Estimates statistics for intermediate results to compute cost of complex expressions

Query Optimization/2

- **Step 1: Parsing and translation**
  - Translate the query into its internal form (query tree)
  - The query tree corresponds to a **relational algebra (RA)** expression
  - Each RA expression can be written as a **tree** where the algebra operator is the root and the argument relations are the children.
- **Example:**
  - SQL query: **select** balance **from** account **where** balance $<$ 2500
  - RA expression: $\sigma_{balance<2500}(\pi_{balance}(\text{account}))$
  - Tree:

$$\begin{tikzcd}
\sigma_{balance<2500} \arrow[d, dash] \\
\pi_{balance} \arrow[d, dash] \\
account
\end{tikzcd}$$

Query Optimization/3

- **Step 2: Optimization**
  - An RA expression may have many (semantically) equivalent expressions
  - The following two RA expressions are equivalent:
    - $\sigma_{balance<2500}(\pi_{balance}(\text{account}))$
    - $\pi_{balance}(\sigma_{balance<2500}(\text{account}))$
  - Each RA operation can be evaluated using one of several different algorithms.
  - Thus, an RA expression can be evaluated in many ways.

Query Optimization/4

- **Step 2: Optimization**
  - **Evaluation plan:** Annotated RA expression that specifies for each operator detailed instructions on how to evaluate it.
    - use index on balance to find accounts with balance $<$ 2500
    - can perform complete relation scan and discard accounts with balance $\geq$ 2500

$$\boxed{\begin{tikzcd}
\pi_{balance} \arrow[d, dash] \\
\sigma_{balance<2500};\ \text{use index 1} \arrow[d, dash] \\
account
\end{tikzcd}}$$

  - **Goal of query optimization:** Among 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.
- **Step 3: Evaluation**
  - The query-execution engine takes an evaluation plan, executes that plan, and returns the answers.

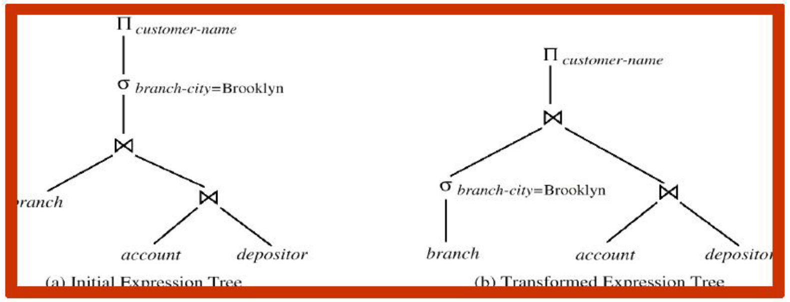
Review 8.5

Display the trees that correspond to the following algebra expressions:

- $RA1 = \pi_A(R1 \bowtie \sigma_{X=Y}(R2 \bowtie \pi_{B,C}(R3 - R4) \bowtie R5))$
- $RA2 = \pi_A(R1) \cup \sigma_{X>5}(R2)$

Query Optimization/5

- **Example:** Find the names of all customers who have an account at any branch located in Brooklyn.
  - $\pi_{CustName}(\sigma_{BranchCity='Brooklyn'}(branch \bowtie (account \bowtie depositor)))$
    - Produces a large intermediate relation
    - Transformation into a more efficient expression
      $\pi_{CustName}(\sigma_{BranchCity='Brooklyn'}(branch) \bowtie (account \bowtie depositor))$



Query Optimization/6

- **Goal of query optimizer:** Find the most efficient query evaluation plan for a given query.
- **Cost-based** optimization:
  1. Generate logically equivalent expressions by using equivalence rules to rewrite an expression into an equivalent one
  2. Annotate resulting expressions with information about algorithms/indexes for each operator
  3. Choose the cheapest plan based on **estimated cost**
- **Rule-based/heuristic** optimization:
  1. Generate logically equivalent expressions, controlled by a set of heuristic query optimization rules
- In general, it is not possible to identify the optimal query tree since there are too many. Instead, a reasonably efficient one is chosen.

Statistical Information/1

- The cost of an operation depends on the size and other statistics of its inputs, which is partially stored in the database catalog and can be used to estimate statistics on the results of various operations.
  - n_r: number of tuples in a relation r.
  - b_r: number of blocks containing tuples of r.
  - s_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 $\pi_A(r)$.
  - SC(A, r): selection cardinality of attribute A of relation r; average number of records that satisfy equality on A.

Statistical Information/2

- f_i: average fan-out of internal nodes of index i, for tree-structured indexes such as B+ trees.
- HT_i: number of levels in index i, i.e., the height of i.
  - For a 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.
  - LB_i: number of lowest-level index blocks in i, i.e, the number of blocks at the leaf level of the index.
- For accurate statistics, the catalog information has to be updated every time a relation is modified.
  - Many systems update statistics only during periods of light system load (or when requested explicitly), thus statistics is not completely accurate.
  - Plan with lowest estimated cost might not be the cheapest
  - PostgreSQL: run ANALYZE once a day

Rewriting Relational Algebra Expressions

- Two relational algebra expressions are **equivalent** if on every legal database instance the two expressions generate the same set of tuples
  - Note: order of tuples is irrelevant
- Two expressions in the multiset version of the relational algebra are said to be equivalent if on every legal database instance the two expressions generate the same multiset of tuples
- An equivalence rule states that two different expressions are equivalent and can replace each other

Equivalence Rules/1

- $E, E_1, \dots$ = RA expressions
  $\theta, \theta_1, \dots$ = predicates/conditions
  - **ER1** Conjunctive selection operations can be deconstructed into a sequence of individual selections.
    $$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$
  - **ER2** Selection operations are commutative.
    $$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$
  - **ER3** Only the last in a sequence of projections is needed, the others can be omitted (Li are lists of attributes).
    $$\pi_{L1}(\pi_{L2}(\dots(\pi_{Ln}(E))\dots)) = \pi_{L1}(E)$$
  - **ER4** Selections can be combined with Cartesian product and theta joins
    (a) $\sigma_\theta(E_1 \; x \; E_2) = E_1 \bowtie_\theta E_2$
    (b) $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

Equivalence Rules/2

- **ER5** Theta joins (and natural joins) are commutative.
  $$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$
- **ER6** Associativity
  (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 way:
  $$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$
  where θ_2 involves attributes from only E_2 and E_3.
  Any of these conditions might be empty, hence, the Cartesian product operation is also associative
- Commutativity and associativity of join operations are important for join reordering.

Equivalence Rules/3

- **ER7** The selection operation distributes over the theta join operation under the following conditions:
  - (a) When all attributes in θ_o involve only the attributes of one of the expressions (E1) being joined:
    $$\sigma_{\theta_o}(E_1 \bowtie_\theta E_2) = \sigma_{\theta_o}(E_1) \bowtie_\theta E_2$$
  - (b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2:
    $$\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_\theta E_2) = \sigma_{\theta 1}(E_1) \bowtie_\theta \sigma_{\theta 2}(E_2)$$

Equivalence Rules/4

- **ER8** The projection operation distributes over the theta join operation as follows:
  - Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.
  - (a) if θ 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_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
    - Let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
    $\pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \pi_{L_1 \cup L_2}(\pi_{L_1 \cup L_3}(E_1) \bowtie_\theta \pi_{L_2 \cup L_4}(E_2))$

Equivalence Rules/5

- **ER9** The set operations union and intersection are commutative
  $E_1 \cup E_2 = E_2 \cup E_1$
  $E_1 \cap E_2 = E_2 \cap E_1$
  Set difference is not commutative
- **ER10** Set union and intersection are associative.
  $(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$
  $(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$

Equivalence Rules/6

- **ER11** The selection operation distributes over $\cup, \cap$ and $-$.
  $\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - \sigma_\theta(E_2)$
  $\sigma_\theta(E_1 \cup E_2) = \sigma_\theta(E_1) \cup \sigma_\theta(E_2)$
  $\sigma_\theta(E_1 \cap E_2) = \sigma_\theta(E_1) \cap \sigma_\theta(E_2)$

  Also $\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - E_2$
  and similarly for $\cap$ in place of $-$, but not for $\cup$
- **ER12** The projection operation distributes over union
  $\pi_L(E_1 \cup E_2) = \pi_L(E) \cup \pi_L(E_2)$

Review 8.6

Determine the equivalences that hold. Give counterexamples for the false ones.

1. $\sigma_\theta({}_X\vartheta_F(A)) = {}_X\vartheta_F(\sigma_\theta(A))$, $attr(\theta) \subseteq attr(X)$
2. $\pi_X(A - B) = \pi_X(A) - \pi_X(B)$
3. $A \leftouterjoin (B \leftouterjoin C) = (A \leftouterjoin B) \leftouterjoin C$
4. $A \cap B = A \cup B - (A - B) - (B - A)$

Rewrite Examples/1

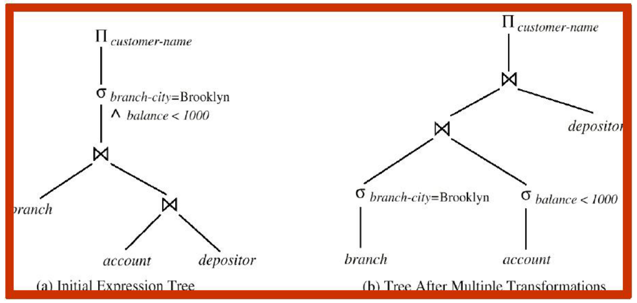
- **Example 1:** Bank database
  - *branch = (BranchName, BranchCity, Assets)*
  - *account = (AccNumber, BranchName, Balance)*
  - *depositor = (CustName, AccNumber)*
- **Query:** Find the names of all customers who have an account at some branch located in Brooklyn.
  $\pi_{CustName}(\sigma_{BranchCity='Brooklyn'}(branch \bowtie (account \bowtie depositor)))$
- Transformation using rule **ER7**(a):
  $\pi_{CustName}$
  $(\sigma_{BranchCity='Brooklyn'}(branch) \bowtie (account \bowtie depositor))$
- Performing the selection as early as possible reduces the size of the intermediate relation to be joined.

Rewrite Examples/2

- **Example 2:** Multiple transformations are often needed
  - **Query:** Find the names of all customers with an account at Brooklyn whose balance is below $1000.
    $\pi_{CustName}(\sigma_{BranchCity='Brooklyn' \wedge balance<1000}$
    $(branch \bowtie (account \bowtie depositor)))$
- Rewrite using rule **ER6**(a) (join associativity):
  $\pi_{CustName}(\sigma_{BranchCity='Brooklyn' \wedge balance<1000}$
  $(branch \bowtie account) \bowtie depositor)$
- Rewrite using rule **ER7**(b) (perform selection early)
  $\sigma_{BranchCity='Brooklyn'}(branch) \bowtie \sigma_{balance<1000}(account)$

Rewrite Examples/3

- **Example 2** (continued)
  - Tree representation after multiple transformations



$\Pi_{customer\text{-}name}$
$\sigma_{branch\text{-}city=\text{Brooklyn} \wedge balance < 1000}$
$\bowtie$
branch
$\bowtie$
account
depositor
$\Pi_{customer\text{-}name}$
$\bowtie$
$\bowtie$
depositor
$\sigma_{branch\text{-}city=\text{Brooklyn}}$
$\sigma_{balance < 1000}$
branch
account

(a) Initial Expression Tree (b) Tree After Multiple Transformations

Rewrite Examples/4

- **Example 3:** Projection operation
  - Query:
    $\pi_{CustName}((\sigma_{BranchCity='Brooklyn'}(branch) \bowtie account) \bowtie depositor)$
- When we compute
  $\sigma_{BranchCity='Brooklyn'}(branch) \bowtie account$
  we obtain an intermediate relation with schema
  (BranchName, BranchCity, Assets, AccNumber, Balance)
- Push projections using equivalence rules **ER8**(a) and **ER8**(b); thus, eliminate unneeded attributes from intermediate results:
  $\pi_{CustName}$
  $(\pi_{AccNumber}(\sigma_{BranchCity='Brooklyn'}(branch) \bowtie account)$
  $\bowtie depositor)$

Rewrite Examples/5

- **Example 4:** Join ordering
- 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.

Rewrite Examples/6

- **Example 5:** Join ordering
- Consider the expression
  $\pi_{CustName}(\sigma_{BranchCity='Brooklyn'}(branch) \bowtie account \bowtie depositor)$
- Could compute *account* $\bowtie$ *depositor* first, and join result with
  $\sigma_{BranchCity='Brooklyn'}(branch)$
  but *account* $\bowtie$ *depositor* is likely to be a large relation.
- Since it is more likely that only a small fraction of the bank's customers have accounts in branches located in Brooklyn, it is better to compute first
  $\sigma_{BranchCity='Brooklyn'}(branch) \bowtie account$

Review 8.7

Show how to rewrite and optimize the following SQL query:

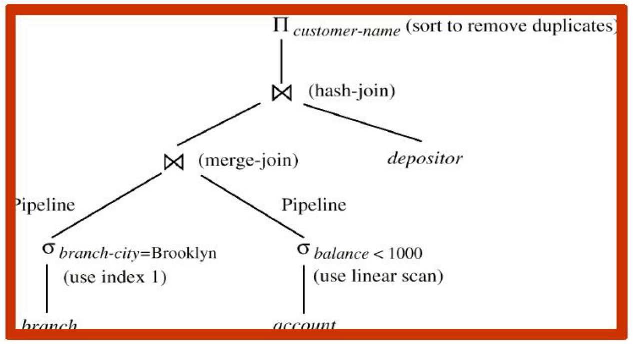
```
select E.LName
from Employee E, WorksOn W, Project P
where P.PName = 'A'
and P.PNum = W.PNo
and W.ESSN = E.SSN
and E.BDate = '31.12.1957'
```

Enumeration of Equivalent Expressions

- **Query optimizers** use the equivalence rules to systematically generate expressions that are equivalent to the given expression
- **repeat**
  For each expression found so far, use all applicable equivalence rules, and add newly generated expressions to the set of expressions found so far
  until no more expressions can be found
- This approach is very expensive in space and time
- Reduce space requirements by sharing common subexpressions:
  - When E1 is generated from E2 by an equivalence rule, usually only the top level of the two are different, subtrees below are the same and can be shared (e.g. when applying join associativity)
- Time requirements are reduced by not generating all expressions (e.g. take cost estimates into account)

Evaluation Plan

- **Evaluation plan (query plan/query tree):** Defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.



Choosing Evaluation Plans

- When choosing the best evaluation plan, the query optimizer must consider the interaction of evaluation techniques:
  - 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 for an outer level aggregation.
    - nested loop join may provide opportunity for pipelining
  - Practical query optimizers combine elements of the following two broad approaches:
    1. **Cost-based optimization:** Search all plans and choose the best plan in a cost-based fashion.
    2. **Rule-based optimization:** Uses heuristics to choose a plan.

Heuristic Optimization/1

- Heuristic optimization transforms the query-tree by using a set of heuristic rules that typically (but not in all cases) improve execution performance.
- Overall goal of heuristic rules:
  - Try to **reduce the size of (intermediate) relations** as early as possible
- Heuristic rules
  - 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 (old) systems use only heuristics
- Modern database systems combine heuristics (consider some plans only) with cost-based optimization (determine database specific cost of each plan).

Heuristic Optimization/2

- Example: Consider the expression $\sigma_\theta(r \bowtie s)$, where θ is on attributes in s only.
  - Selection early rule would push down the selection operator, producing $r \bowtie \sigma_\theta(s)$.
  - This is not necessarily the best plan if
    - relation r is extremely small compared to s,
    - and there is an index on the join attributes of s,
    - but there is no index on the attributes used by θ.
  - The early select would require a scan of all tuples in s, which is probably more expensive than the join

Heuristic Optimization/3

- Steps in typical heuristic optimization
  1. Break up conjunctive selections into a sequence of single selection operations (rule **ER1**).
  2. Move selection operations down the query tree for the earliest possible execution (rules **ER2**, **ER7**(a), **ER7**(b), **ER11**).
  3. Execute first those selection and join operations that will produce the smallest relations (rule **ER6**).
  4. Replace Cartesian product operations that are followed by a selection condition by join operations (rule **ER4**(a)).
  5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed (rules **ER3**, **ER8**(a), **ER8**(b), **ER12**).
  6. Identify those subtrees whose operations can be pipelined, and execute them using pipelining.

Cost-Based Optimization/1

Basic working of a cost-based query optimizer:

- **Algorithm**
  1. Use transformations (equivalence rules) to generate multiple candidate evaluation plans from the original evaluation plan.
  2. Cost formulas estimate the cost of executing each operation in each candidate evaluation plan.
     - Cost formulas are parameterized by
       - statistics of the input relations;
       - dependent on the specific algorithm used by the operator;
       - CPU time, I/O time, communication time, main memory usage, or a combination.
  3. The candidate evaluation plan with the **least total cost** is selected for execution.

Cost-Based Optimization/2

- Cost-based optimization can be used to determine the best join order.
- A good ordering of joins is important for reducing the size of temporary results ($|r|, ..., |r|^n$).
- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie ...r_m$
- There are $(2(m-1))!/(m-1)!$ different join orders for above expression.
  - With $m = 3$, the number is 12
  - With $m = 7$, the number is 665,280
  - With $m = 10$, the number is greater than 17.6 billion

Cost-Based Optimization/3

- Cost-based optimization is expensive, but worthwhile for queries on large datasets
- Typical queries have a small number m of operations; generally m<10
- With dynamic programming time complexity of optimization with bushy trees is $O(3^m)$.
  - With $m = 10$, this number is 59000 instead of 17.6 billion!
- Space complexity is $O(2^m)$

Review 8.8

Consider a DB with the following characteristics:

- $|r1(A,B,C)| = 1000, V(C,r1) = 900$
- $|r2(c,D,E)| = 1500, V(C,r2) = 1100, V(E,r2) = 50$
- $|r3(E,F)| = 750, V(E,r3) = 100$

Estimate the size of $r1 \bowtie r2 \bowtie r3$ and determine an efficient evaluation strategy.

Cost-Based Optimization Example/1

- **Example:** $\sigma_{SSN=0810643773}(Emp)$
- Statistics:
  - $|Emp|$ = 10'000 tuples
  - 5 tuples per block
  - Secondary B^+-tree index of depth 4 on SSN
  - SSN is primary key
- Plan p1: full table scan
  - cost(p1) = (10'000/5)/2 = 1'000 blocks
- Plan p2: B^+-tree lookup
  - cost(p2) = 4 + 1 = 5 blocks

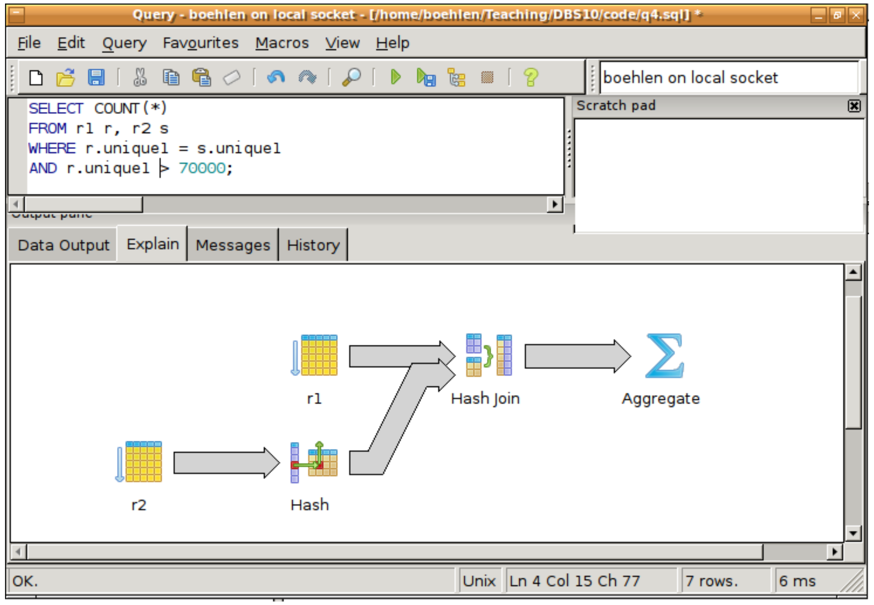
Cost-Based Optimization Example/2

- Example: $\sigma_{DNo>15}(Emp)$
- Statistics:
  - $|Emp|$ = 10'000 tuples
  - 5 tuples per block
  - Primary index on DNo of depth 2
  - 50 different departments
- Plan p1: full table scan
  - cost (p1) = 10'000/5 = 2'000 blocks
- Plan p2: index search
  - cost(p2) = 2 + (50-15)/50*(10'000/5) = 1'400 blocks

Cost-Based Optimization Example/3

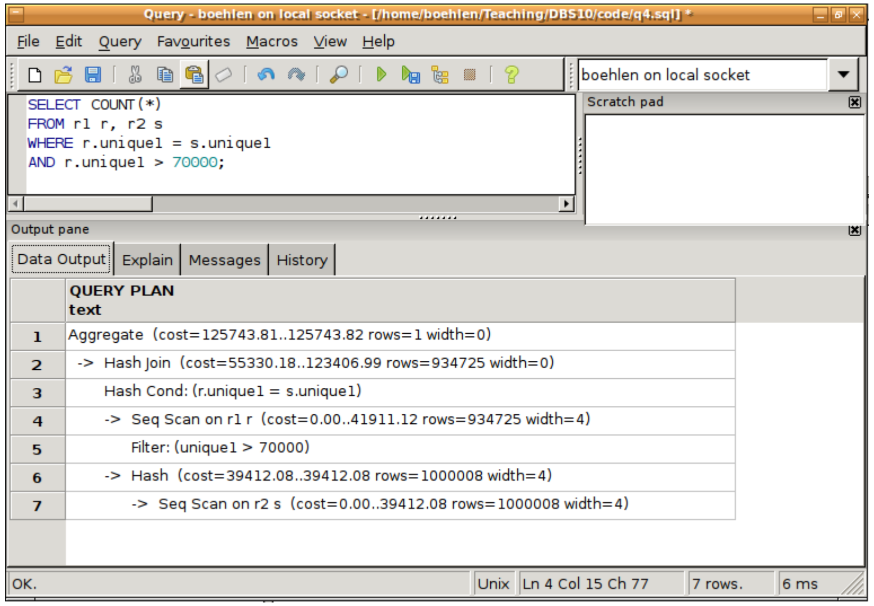
- Emp $\bowtie_{DNo=DNum}$ Dept
- Statistics:
  - $|Emp|$ = 10'000 tuples ; 5 Emp tuples per block
  - $|Dept|$ = 125; 10 Dept tuples per block
  - Hash index on Emp(DNo)
  - 4 EmpDept result tuples per block
- Plan p1: Block nested loop with Emp as outer loop
  - cost(p1) = (10'000/5) + (10'000/5) * (125/10) + (10'000/4)
    = 30'500 IOs
  - (10.000/4 is cost of writing final output)
- Plan p2: Indexed nested loop with Dept as outer loop and hashed lookup in Emp
  - cost(p2) = (125/10) + 125 * (10'000/125/5) + (10'000/4)
    = 4'513 IOs
  - 10'000/125/5 is the average number of blocks/department

PostgreSQL Query Optimization/1



```
SELECT  COUNT(*)
FROM r1 r, r2 s
WHERE r.unique1 = s.unique1
AND r.unique1 > 70000;
```

PostgreSQL Query Optimization/2

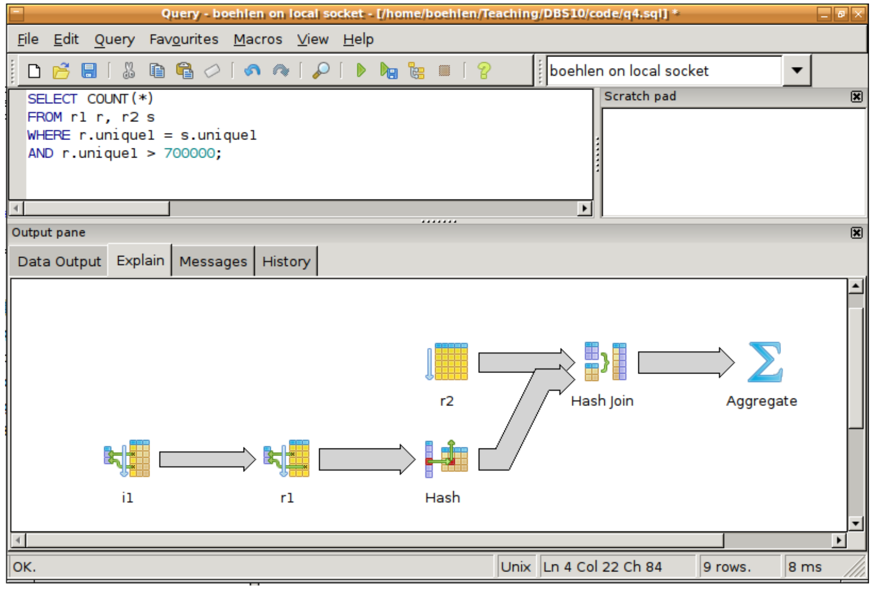


```
SELECT  COUNT(*)
FROM r1 r, r2 s
WHERE r.unique1 = s.unique1
AND r.unique1 > 70000;
```

	QUERY PLAN text
1	Aggregate (cost=125743.81..125743.82 rows=1 width=0)
2	-> Hash Join (cost=55330.18..123406.99 rows=934725 width=0)
3	Hash Cond: (r.unique1 = s.unique1)
4	-> Seq Scan on r1 r (cost=0.00..41911.12 rows=934725 width=4)
5	Filter: (unique1 > 70000)
6	-> Hash (cost=39412.08..39412.08 rows=1000008 width=4)
7	-> Seq Scan on r2 s (cost=0.00..39412.08 rows=1000008 width=4)

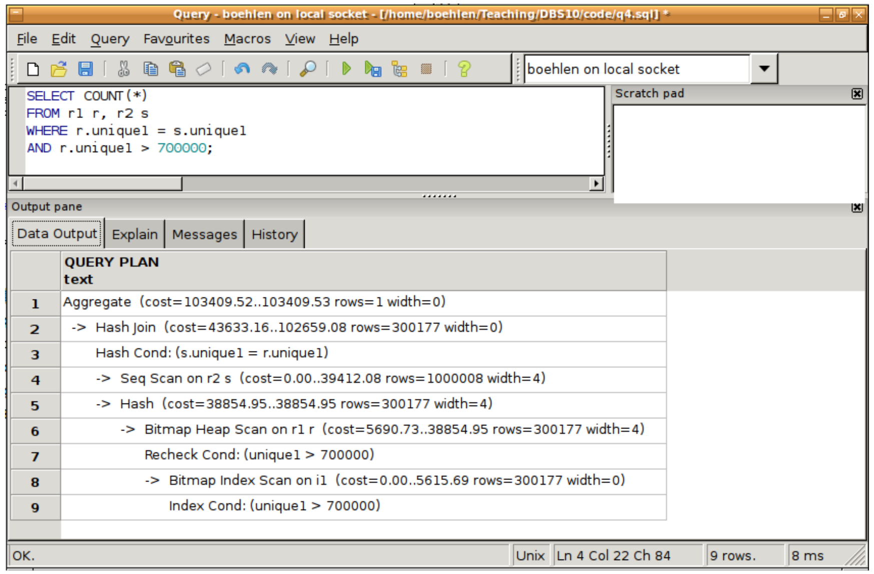
Query - boehlen on local socket - [/home/boehlen/Teaching/DBS10/code/q4.sql] *
File Edit Query Favourites Macros View Help
boehlen on local socket
Scratch pad
Output pane
Data Output
Explain
Messages
History
OK.
Unix
Ln 4 Col 15 Ch 77
7 rows.
6 ms

PostgreSQL Query Optimization/3



```
SELECT  COUNT(*)
FROM r1 r, r2 s
WHERE r.unique1 = s.unique1
AND r.unique1 > 700000;
```

PostgreSQL Query Optimization/4



```
SELECT  COUNT(*)
FROM r1 r, r2 s
WHERE r.unique1 = s.unique1
AND r.unique1 > 700000;
```

	QUERY PLAN text
1	Aggregate (cost=103409.52..103409.53 rows=1 width=0)
2	-> Hash Join (cost=43633.16..102659.08 rows=300177 width=0)
3	Hash Cond: (s.unique1 = r.unique1)
4	-> Seq Scan on r2 s (cost=0.00..39412.08 rows=1000008 width=4)
5	-> Hash (cost=38854.95..38854.95 rows=300177 width=4)
6	-> Bitmap Heap Scan on r1 r (cost=5690.73..38854.95 rows=300177 width=4)
7	Recheck Cond: (unique1 > 700000)
8	-> Bitmap Index Scan on i1 (cost=0.00..5615.69 rows=300177 width=0)
9	Index Cond: (unique1 > 700000)

Query - boehlen on local socket - [/home/boehlen/Teaching/DBS10/code/q4.sql] *
File Edit Query Favourites Macros View Help
boehlen on local socket
Scratch pad
Output pane
Data Output
Explain
Messages
History
OK.
Unix
Ln 4 Col 22 Ch 84
9 rows.
8 ms

Summary/1

- Query evaluation techniques:
  - Physical sorting:
    - Physical sorting is a basic and important technique
    - The same sort order should be useful to many operators and not just one (global optimization versus local optimization)
  - Evaluation techniques for selections:
    - Use primary index if available; secondary index is much worse
    - Equality conditions are selective and should be optimized
    - Linear scan with sequential IO is the base line for selections
  - Evaluation techniques for joins:
    - nested loop: base line; avoid whenever possible
    - sort merge: robust and fast
    - hash join: fastest; only for equality

Summary/2

- Query optimization techniques
  - **Equivalence rules** for relational algebra expressions (must hold for multisets)
  - **Rule-based query optimization** is based on heuristics (usually the goal is to keep intermediate results as small as possible)
  - **Cost-based query optimization** uses statistical information to find the cheapest (or reasonably cheap) plan