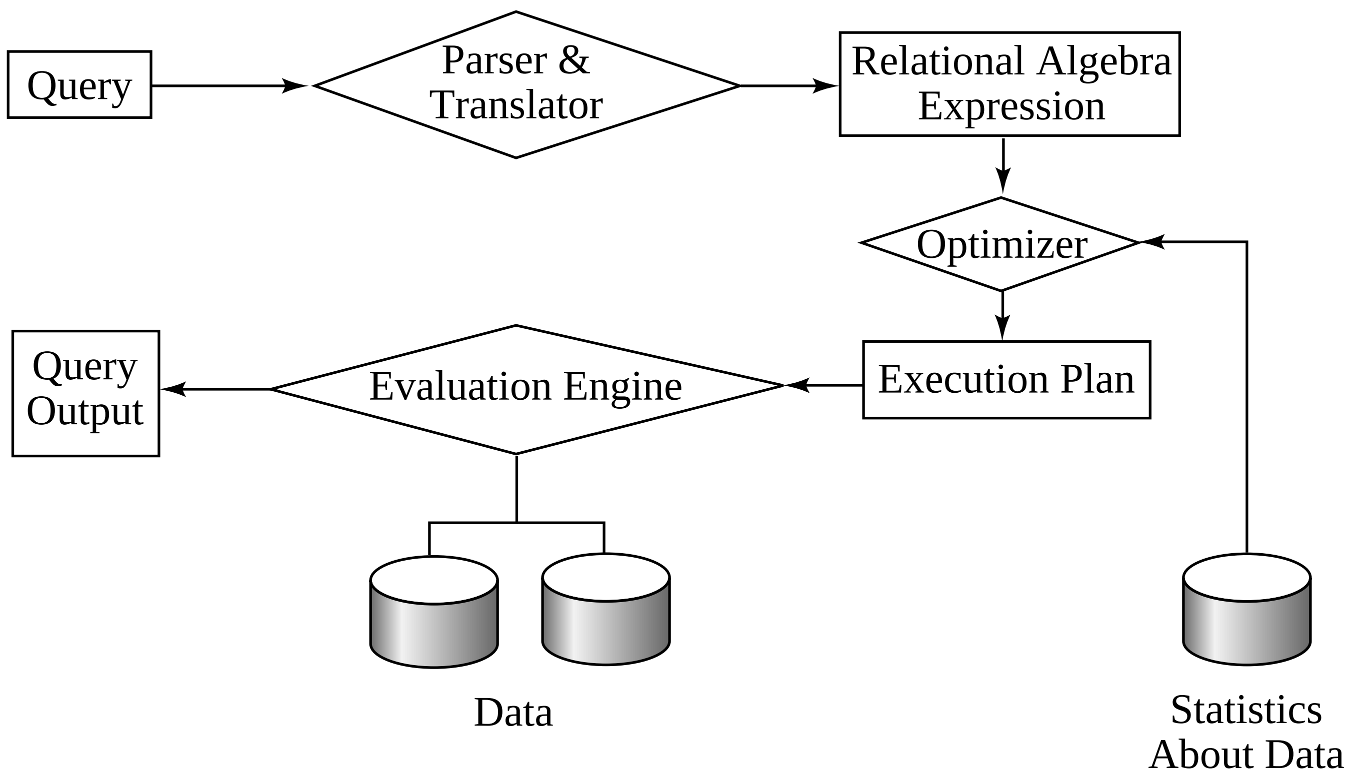
Chapter 12: Query Processing

- Overview
- Catalog Information for Cost Estimation
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions
- Transformation of Relational Expressions
- Choice of Evaluation Plans

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



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.

Basic Steps in Query Processing

Optimization – finding the cheapest evaluation plan for a query.

- Given relational algebra expression may have many equivalent expressions

  E.g. $\sigma_{balance<2500}(\pi_{balance}(account))$ is equivalent to $\pi_{balance}(\sigma_{balance<2500}(account))$

- Any 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 ≥ 2500

- Amongst all equivalent expressions, try to choose the one with cheapest possible evaluation-plan. Cost estimate of a plan based on *statistical information* in the DBMS catalog.

Catalog Information for Cost Estimation

- n_r: number of tuples in relation r.
- b_r: number of blocks containing tuples of r.
- s_r: size of a tuple of r in bytes.
- 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.
- If tuples of r are stored together physically in a file, then:

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

Catalog Information about Indices

- f_i: average fan-out of internal nodes of index i, for tree-structured indices such as B+-trees.
- HT_i: number of levels in index i — i.e., the height of i.
  - For a balanced tree index (such as 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.

Measures of Query Cost

- Many possible ways to estimate cost, for instance *disk accesses*, *CPU time*, or even *communication overhead* in a distributed or parallel system.
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Therefore *number of block transfers from disk* is used as a measure of the actual cost of evaluation. It is assumed that all transfers of blocks have the same cost.
- Costs of algorithms depend on the size of the buffer in main memory, as having more memory reduces need for disk access. Thus memory size should be a parameter while estimating cost; often use worst case estimates.
- We refer to the cost estimate of algorithm A as E_A. We do not include cost of writing output to disk.

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 (number of disk blocks scanned) $E_{A1} = b_r$
  - If selection is on a key attribute, $E_{A1} = (b_r/2)$ (stop on finding record)
  - Linear search can be applied regardless of
    - selection condition, or
    - ordering of records in the file, or
    - availability of indices

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):

$$E_{A2} = \lceil \log_2(b_r) \rceil + \left\lceil \frac{SC(A,r)}{f_r} \right\rceil - 1$$

    - $\lceil \log_2(b_r) \rceil$ — cost of locating the first tuple by a binary search on the blocks
    - $SC(A,r)$ — number of records that will satisfy the selection
    - $\lceil SC(A,r)/f_r \rceil$ — number of blocks that these records will occupy
  - Equality condition on a key attribute: $SC(A,r) = 1$; estimate reduces to $E_{A2} = \lceil \log_2(b_r) \rceil$

Statistical Information for Examples

- $f_{account} = 20$ (20 tuples of *account* fit in one block)
- $V(branch\text{-}name, account) = 50$ (50 branches)
- $V(balance, account) = 500$ (500 different *balance* values)
- $n_{account} = 10000$ (*account* has 10,000 tuples)
- Assume the following indices exist on *account*:
  - A primary, B^+-tree index for attribute *branch-name*
  - A secondary, B^+-tree index for attribute *balance*

Selection Cost Estimate Example

$$\sigma_{branch\text{-}name=\text{``Perryridge''}}(account)$$

- Number of blocks is $b_{account} = 500$: $10,000$ tuples in the relation; each block holds 20 tuples.
- Assume *account* is sorted on *branch-name*.
  - $V(branch\text{-}name, account)$ is 50
  - $10000/50 = 200$ tuples of the *account* relation pertain to Perryridge branch
  - $200/20 = 10$ blocks for these tuples
  - A binary search to find the first record would take $\lceil log_2(500) \rceil = 9$ block accesses
- Total cost of binary search is $9 + 10 - 1 = 18$ block accesses (versus 500 for linear scan)

Selections Using Indices

- **Index scan** – search algorithms that use an index; condition is on search-key of index.
- **A3** (*primary index on candidate key, equality*). Retrieve a single record that satisfies the corresponding equality condition. $E_{A3} = HT_i + 1$
- **A4** (*primary index on nonkey, equality*) Retrieve multiple records. Let the search-key attribute be A.
  $E_{A4} = HT_i + \left\lceil \frac{SC(A,r)}{f_r} \right\rceil$
- **A5** (*equality on search-key of secondary index*).
  - Retrieve a single record if the search-key is a candidate key
    $E_{A5} = HT_i + 1$
  - Retrieve multiple records (each may be on a different block) if the search-key is not a candidate key.
    $E_{A5} = HT_i + SC(A, r)$

Cost Estimate Example (Indices)

Consider the query is $\sigma_{branch\text{-}name=\text{“Perryridge”}}(account)$, with the primary index on *branch-name*.

- Since $V(branch\text{-}name,\ account) = 50$, we expect that $10000/50 = 200$ tuples of the *account* relation pertain to the Perryridge branch.
- Since the index is a clustering index, $200/20 = 10$ block reads are required to read the *account* tuples
- Several index blocks must also be read. If B^+-tree index stores 20 pointers per node, then the B^+-tree index must have between 3 and 5 leaf nodes and the entire tree has a depth of 2. Therefore, 2 index blocks must be read.
- This strategy requires 12 total block reads.

Selections Involving Comparisons

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*). The cost estimate is:

$$E_{A6} = HT_i + \left\lceil \frac{c}{f_r} \right\rceil$$

  where c is the estimated number of tuples satisfying the condition. In absence of statistical information c is assumed to be $n_r/2$.

- **A7** (*secondary index, comparison*). The cost estimate is:

$$E_{A7} = HT_i + \frac{LB_i \cdot c}{n_r} + c$$

  where c is defined as before. (Linear file scan may be cheaper if c is large!)

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, θ_i's selectivity 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:

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

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

$$n_r - size(\sigma_\theta(r))$$

Algorithms for Complex Selections

- **A8** (*conjunctive selection using one index*). Select a combination of θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta_i}(r)$. Test other conditions in 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 read file. If some conditions did not have appropriate indices, apply test in memory.
- **A11** (*disjunctive selection by union of identifiers*). Applicable if *all* conditions have available indices. Otherwise use linear scan.

Example of Cost Estimate for Complex Selection

- Consider a selection on *account* with the following condition:
  where *branch-name* = "Perryridge" **and** *balance* = 1200
- Consider using algorithm A8:
  - The *branch-name* index is clustering, and if we use it the cost estimate is 12 block reads (as we saw before).
  - The *balance* index is non-clustering, and $V(balance, account) = 500$, so the selection would retrieve $10,000/500 = 20$ accounts. Adding the index block reads, gives a cost estimate of 22 block reads.
  - Thus using *branch-name* index is preferable, even though its condition is less selective.
  - If both indices were non-clustering, it would be preferable to use the *balance* index.

Example (contd.)

- Consider using algorithm A10:
  - Use the index on *balance* to retrieve set S_1 of pointers to records with $balance = 1200$.
  - Use index on *branch-name* to retrieve set S_2 of pointers to records with $branch\text{-}name =$ "Perryridge".
  - $S_1 \cap S_2 =$ set of pointers to records with $branch\text{-}name =$ "Perryridge" and $balance = 1200$.
  - The number of pointers retrieved (20 and 200) fit into a single leaf page; we read four index blocks to retrieve the two sets of pointers and compute their intersection.
  - Estimate that one tuple in $50 * 500$ meets both conditions. Since $n_{account} = 10000$, conservatively overestimate that $S_1 \cap S_2$ contains one pointer.
  - The total estimated cost of this strategy is five block reads.

Sorting

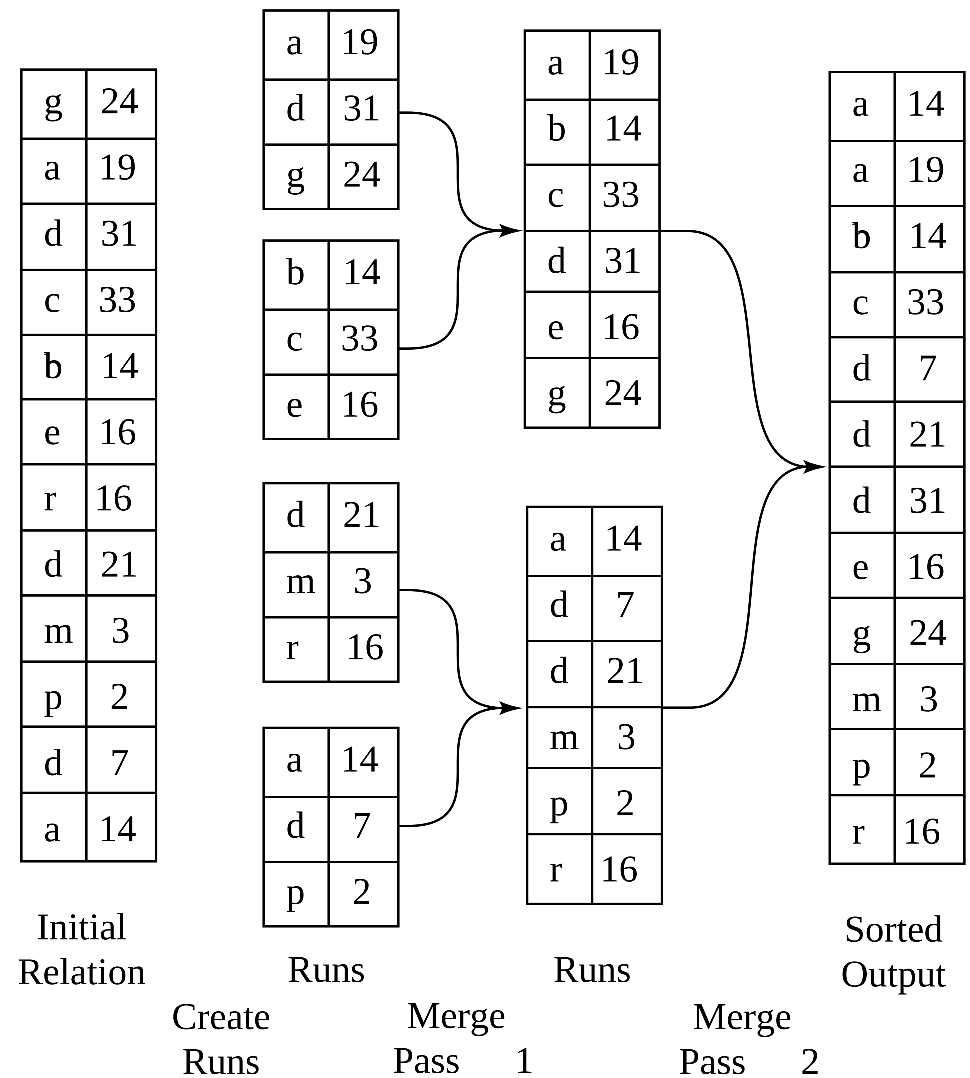
- We may build an index on the relation, and then use the index to read the relation in sorted order. May lead to one disk block access for each tuple.
- For relations that fit in memory, techniques like quicksort can be used. For relations that don't fit in memory, **external sort-merge** is a good choice.

External Sort–Merge

Let M denote memory size (in pages).

1. Create sorted **runs** as follows. Let i be 0 initially. Repeatedly do the following till the end of the relation:
   (a) Read M blocks of relation into memory
   (b) Sort the in-memory blocks
   (c) Write sorted data to run R_i; increment i.
2. Merge the runs; suppose for now that $i < M$. In a single merge step, use i blocks of memory to buffer input runs, and 1 block to buffer output. Repeatedly do the following until all input buffer pages are empty:
   (a) Select the first record in sort order from each of the buffers
   (b) Write the record to the output
   (c) Delete the record from the buffer page; if the buffer page is empty, read the next block (if any) of the run into the buffer.

Example: External Sorting Using Sort–Merge



External Sort–Merge (Cont.)

- If $i \geq M$, several merge *passes* 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.
  - Repeated passes are performed till all runs have been merged into one.
- Cost analysis:
  - Disk accesses for initial run creation as well as in each pass is $2b_r$ (except for final pass, which doesn't write out results)
  - Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$.

  Thus total number of disk accesses for external sorting:

$$b_r(2\lceil \log_{M-1}(b_r/M) \rceil + 1)$$

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
- Join size estimates required, particularly for cost estimates for outer-level operations in a relational-algebra expression.

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
  $b_{depositor} = 5000/50 = 100$.
- $V(customer\text{-}name, depositor) = 2500$, which implies that, on average, each customer has two accounts.

Also assume that *customer-name* in *depositor* is a foreign key on *customer*.

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 $r \bowtie s$ 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.

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 $R \bowtie S$, 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.

Estimation of the Size of Joins (Cont.)

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

Nested-Loop Join

- 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 θ
        if they do, add t_r · 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. If the smaller relation fits entirely in main memory, use that relation as the inner relation.

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 $n_r * b_s + b_r$ disk accesses.
- If the smaller relation fits entirely in memory, use that as the inner relation. This reduces the cost estimate to $b_r + b_s$ disk accesses.
- Assuming the worst case memory availability scenario, cost estimate will be $5000 * 400 + 100 = 2,000,100$ disk accesses with *depositor* as outer relation, and $10000 * 100 + 400 = 1,000,400$ disk accesses with *customer* as the outer relation.
- If the smaller relation (*depositor*) fits entirely in memory, the cost estimate will be 500 disk accesses.
- Block nested-loops algorithm (next slide) is preferable.

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
                test pair (t_r, t_s) for satisfying the join condition
                if they do, add t_r · t_s to the result.
            end
        end
    end
end
```

- Worst case: each block in the inner relation s is read only once for each *block* in the outer relation (instead of once for each *tuple* in the outer relation)

Block Nested-Loop Join (Cont.)

- Worst case estimate: $b_r * b_s + b_r$ block accesses. Best case: $b_r + b_s$ block accesses.
- Improvements to nested-loop and block nested loop algorithms:
  - If equi-join attribute forms a key on inner relation, stop inner loop with first match
  - In block nested-loop, use $M - 2$ disk blocks as blocking unit for outer relation, where M = memory size in blocks; use remaining two blocks to buffer inner relation and output. Reduces number of scans of inner relation greatly.
  - Scan inner loop forward and backward alternately, to make use of blocks remaining in buffer (with LRU replacement)
  - Use index on inner relation if available

Indexed Nested-Loop Join

- If an index is available on the inner loop's join attribute and join is an equi-join or natural join, more efficient index lookups can replace file scans.
- 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 one page of the index.
  - b_r disk accesses are needed to read relation r, and, for each tuple in r, we perform an index lookup on s.
  - Cost of the join: $b_r + n_r * c$, where c is the cost of a single selection on s using the join condition.
- If indices are available on both r and s, use the one with fewer tuples as the outer relation.

Example of Index Nested-Loop Join

- Compute $depositor \bowtie customer$, with *depositor* as the outer relation.
- Let *customer* have a primary B^+-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.
- Since $n_{depositor}$ is 5000, the total cost is $100 + 5000 * 5 = 25,100$ disk accesses.
- This cost is lower than the $40,100$ accesses needed for a block nested-loop join.

Merge–Join

1. First sort both relations on their join attribute (if not already sorted on the join attributes).

2. Join step is similar to the merge stage of the sort-merge algorithm. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched

pr

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

r

ps

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

s

Merge–Join (Cont.)

- Each tuple needs to be read only once, and as a result, each block is also read only once. Thus number of block accesses is $b_r + b_s$, plus the cost of sorting if relations are unsorted.
- Can be used only for equi-joins and natural joins
- If one relation is sorted, and the other has a secondary B$^+$-tree index on the join attribute, **hybrid merge-joins** are possible. The sorted relation is merged with the leaf entries of the B$^+$-tree. The result is sorted on the addresses of the unsorted relation's tuples, and then the addresses can be replaced by the actual tuples efficiently.

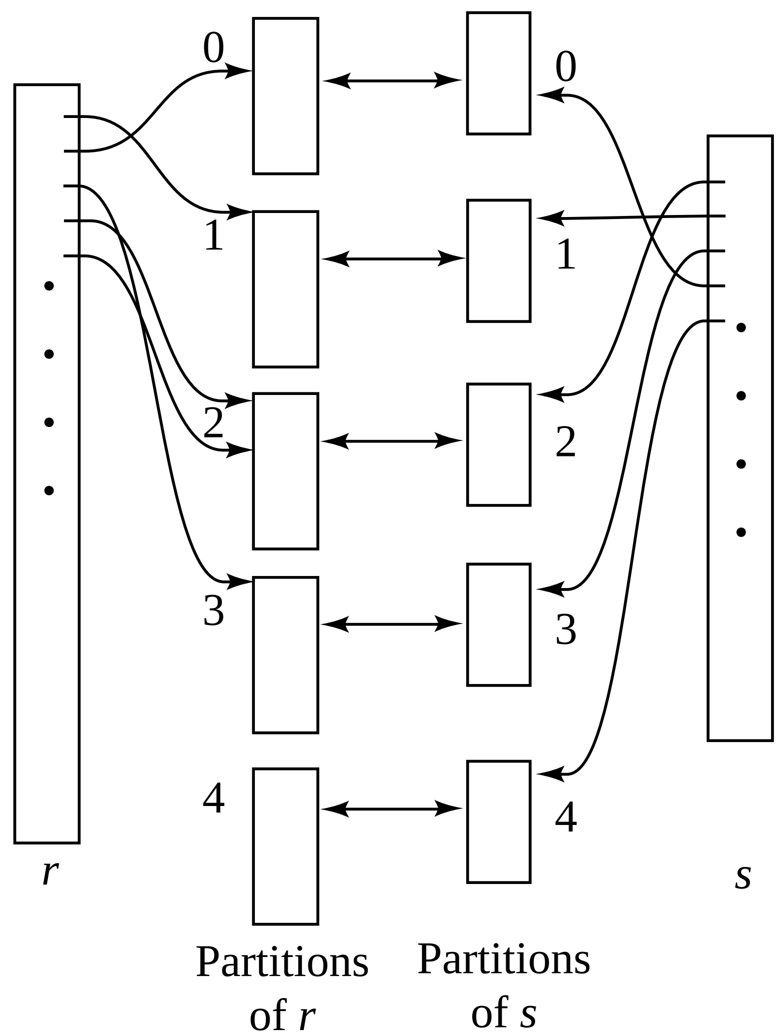
Hash–Join

- Applicable for equi-joins and natural joins.
- A hash function h is used to partition tuples of both relations into sets that have the same hash value on the join attributes, as follows:
  - h maps $JoinAttrs$ values to $\{0, 1, \ldots, max\}$, where $JoinAttrs$ denotes the common attributes of r and s used in the natural join.
  - H_{r_0}, $H_{r_1}, \ldots, H_{r_{max}}$ denote partitions of r tuples, each initially empty. Each tuple $t_r \in r$ is put in partition H_{r_i}, where $i = h(t_r[JoinAttrs])$.
  - H_{s_0}, $H_{s_1}, ..., H_{s_{max}}$ denote partitions of s tuples, each initially empty. Each tuple $t_s \in s$ is put in partition H_{s_i}, where $i = h(t_s[JoinAttrs])$.

Hash–Join (Cont.)

- r tuples in H_{r_i} need only to be compared with s tuples in H_{s_i}; they do not need to be compared with s tuples in any other partition, since:
  - An r tuple and an s tuple that satisfy the join condition will have the same value for the join attributes.
  - If that value is hashed to some value i, the r tuple has to be in H_{r_i} and the s tuple in H_{s_i}.

Hash–Join (Cont.)



Hash–Join algorithm

The hash-join of r and s is computed as follows.

1. Partition the relations s using hashing 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:
   (a) Load H_{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.
   (b) Read the tuples in H_{r_i} from disk one by one. For each tuple t_r locate each matching tuple t_s in H_{s_i} using the in-memory hash index. Output the concatenation of their attributes.

Relation s is called the **build input** and r is called the **probe input**.

Hash–Join algorithm (Cont.)

- The value max and the hash function h is chosen such that each H_{s_i} should fit in memory.
- **Recursive partitioning** required if number of partitions max is greater than number of pages M of memory.
  - Instead of partitioning max ways, partition s $M-1$ ways;
  - Further partition the $M-1$ partitions using a different hash function
  - Use same partitioning method on r
  - Rarely required: e.g., recursive partitioning not needed for relations of 1GB or less with memory size of 2MB, with block size of 4KB.
- **Hash-table overflow** occurs in partition H_{s_i} if H_{s_i} does not fit in memory. Can resolve by further partitioning H_{s_i} using different hash function. H_{r_i} must be similarly partitioned.

Cost of Hash–Join

- If recursive partitioning is not required: $3(b_r + b_s) + 2 * max$
- If recursive partitioning is required, number of passes required for partitioning s is $\lceil log_{M-1}(b_s) - 1 \rceil$. This is because each final partition of s should fit in memory.
- The number of partitions of probe relation r is the same as that for build relation s; the number of passes for partitioning of r is also the same as for s. Therefore it is best to choose the smaller relation as the build relation.
- Total cost estimate is:

$$2(b_r + b_s)\lceil log_{M-1}(b_s) - 1 \rceil + b_r + b_s$$

- If the entire build input can be kept in main memory, max can be set to 0 and the algorithm does not partition the relations into temporary files. Cost estimate goes down to $b_r + b_s$.

Example of Cost of Hash–Join

$$customer \bowtie depositor$$

- Assume that memory size is 20 blocks.
- $b_{depositor} = 100$ and $b_{customer} = 400$.
- *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.
- Therefore total cost: $3(100 + 400) = 1500$ block transfers

  (Ignores cost of writing partially filled blocks).

Hybrid Hash–Join

- Useful when memory sizes are relatively large, and the build input is bigger than memory.
- With a memory size of 25 blocks, *depositor* can be partitioned into five partitions, each of size 20 blocks.
- Keep the first of the partitions of the build relation in memory. It occupies 20 blocks; one block is used for input, and one block each is used for buffering the other four partitions.
- *customer* is similarly partitioned into five partitions each of size 80; the first is used right away for probing, instead of being written out and read back in.
- Ignoring the cost of writing partially filled blocks, the cost is $3(80 + 320) + 20 + 80 = 1300$ block transfers with hybrid hash-join, instead of 1500 with plain hash-join.
- Hybrid hash-join most useful if $M >> \sqrt{b_s}$.

Complex Joins

- Join with a conjunctive condition:

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

  - 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

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

  - Test these conditions as tuples in $r \bowtie_{\theta_i} s$ are generated.

- Join with a disjunctive condition:

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

  Compute as the union of the records in individual joins $r \bowtie_{\theta_i} s$:

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

Complex Joins (Cont.)

- 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 an 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 *deposit* 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.

Other Operations

- **Duplicate elimination** can be implemented via hashing or sorting.
  - On sorting duplicates will come adjacent to each other, and all but one of a 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** is implemented by performing projection on each tuple followed by duplicate elimination.

Other Operations (Cont.)

- **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.
- **Set operations** ($\cup$, $\cap$ and $-$): can either use variant of merge-join after sorting, or variant of hash-join.

Other Operations (Cont.)

- E.g., Set operations using hashing:

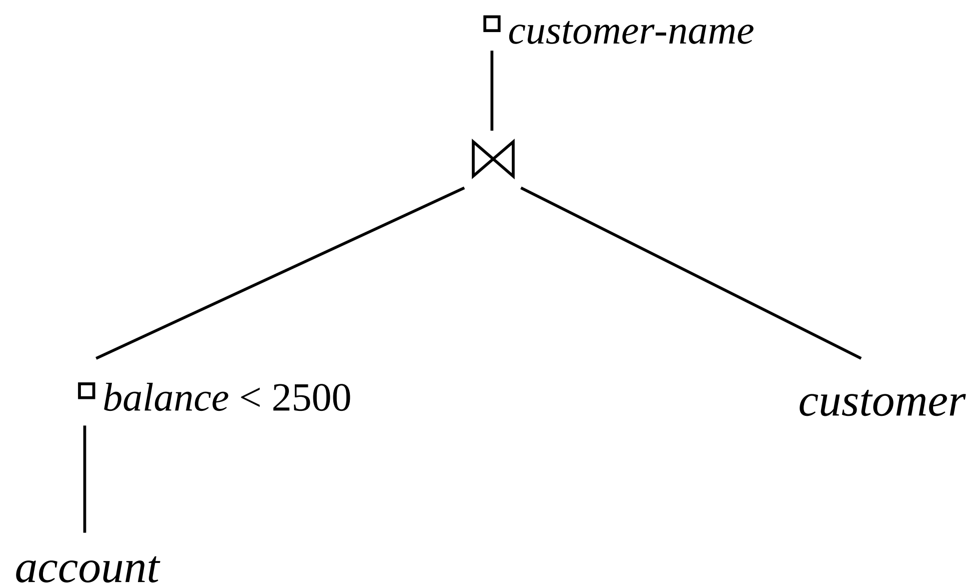
1. Partition both relations using the same hash function, thereby creating $H_{r_0}, \ldots, H_{r_{max}}$, and $H_{s_0}, \ldots, H_{s_{max}}$.
2. Process each partition i as follows. Using a different hashing function, build an in-memory hash index on H_{r_i} after it is brought into memory.
3.
   - $r \cup s$: Add tuples in H_{s_i} to the hash index if they are not already in it. Then add the tuples in the hash index to the result.
   - $r \cap s$: output tuples in H_{s_i} to the result if they are already there in the hash index.
   - $r - s$: for each tuple in H_{s_i}, if it is there in the hash index, delete it from the index. Add remaining tuples in the hash index to the result.

Other Operations (Cont.)

- **Outer join** can be computed either as
  - A join followed by addition of null-padded non-participating tuples.
  - by modifying the join algorithms.
- Example:
  - In $r ⟕ s$, non participating tuples are those in $r - \Pi_R(r \bowtie s)$
  - Modify merge-join to compute $r ⟕ s$: During merging, for every tuple t_r from r that do not match any tuple in s, output t_r padded with nulls.
  - Right outer-join and full outer-join can be computed similarly.

Evaluation of Expressions

- **Materialization**: 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 and store its join with *customer*, and finally compute the projection on *customer-name*.



Evaluation of Expressions (Cont.)

- **Pipelining**: evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in expression in previous slide, 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**.

Transformation of Relational Expressions

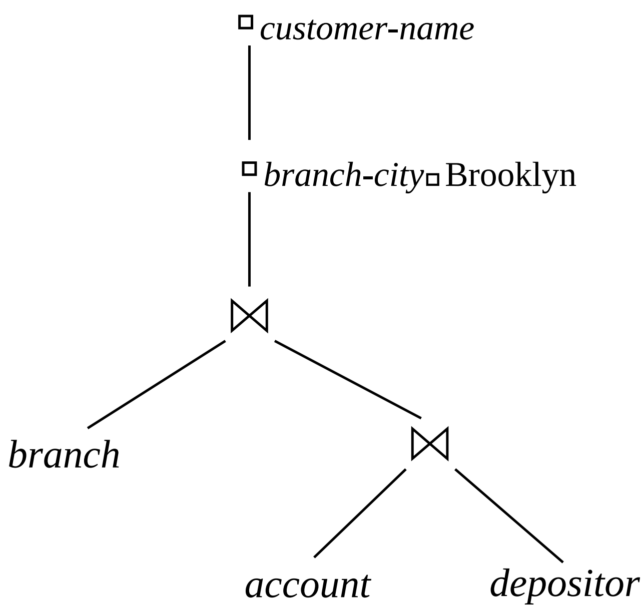
- Generation of query-evaluation plans for an expression involves two steps:

  1. generating logically equivalent expressions
  2. annotating resultant expressions to get alternative query plans

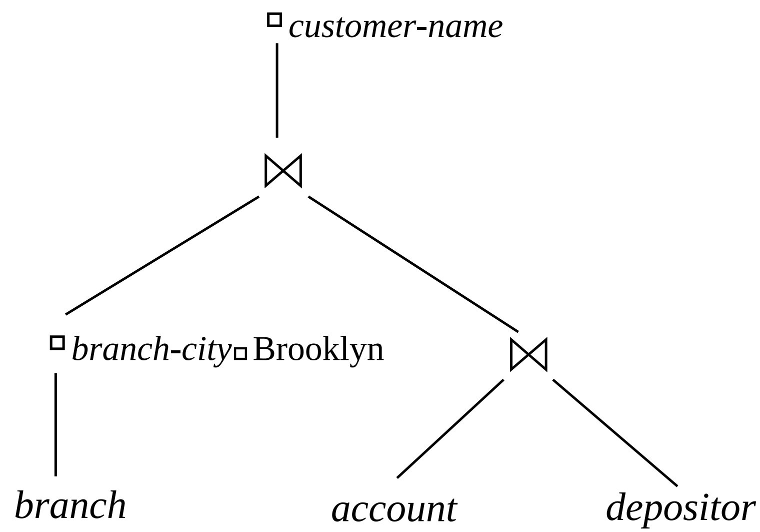
- Use **equivalence rules** to transform an expression into an equivalent one.
- Based on **estimated cost**, the cheapest plan is selected. The process is called **cost based optimization**.

Equivalence of Expressions

Relations generated by two equivalent expressions have the same set of attributes and contain the same set of tuples, although their attributes may be ordered differently.



(a) Initial Expression Tree



(b) Transformed Expression Tree

Equivalent expressions

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.

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

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

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 θ_2 involves attributes from only E_2 and E_3.

Equivalence Rules (Cont.)

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

   (a) When all the attributes in θ_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 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 (Cont.)

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

   (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_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 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 $L_1 \cup L_2$.

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

Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative (set difference is not commutative).

$$E_1 \cup E_2 = E_2 \cup E_1$$

$$E_1 \cap E_2 = E_2 \cap E_1$$

10. Set union and intersection are associative.
11. The selection operation distributes over $\cup, \cap$ and $-$. E.g.:

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - \sigma_P(E_2)$$

For difference and intersection, we also have:

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - E_2$$

12. The projection operation distributes over the union operation.

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

Selection Operation Example

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

  $$\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch \bowtie (account \bowtie depositor)))$$

- Transformation using rule 7a.

  $$\Pi_{customer\text{-}name}\ ((\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch)) \bowtie (account \bowtie depositor))$$

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

Selection Operation Example (Cont.)

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

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

- Transformation using join associativity (Rule 6a):

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

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

  $$\sigma_{branch\text{-}city = \text{"Brooklyn"}}\ (branch) \bowtie\ \sigma_{balance > 1000}\ (account)$$

- Thus a sequence of transformations can be useful

Projection Operation Example

$$\Pi_{customer\text{-}name}((\sigma_{branch\text{-}city = \text{“Brooklyn”}} (branch) \\ \bowtie \; account) \bowtie \; depositor)$$

- When we compute

$$(\sigma_{branch\text{-}city = \text{“Brooklyn”}} (branch) \bowtie \; account)$$

  we obtain a relation whose schema is:

  (*branch-name, branch-city, assets, account-number, balance*)

- Push projections using equivalence rules 8a and 8b; eliminate unneeded attributes from intermediate results to get:

$$\Pi_{customer\text{-}name}((\Pi_{account\text{-}number} (\\ (\sigma_{branch\text{-}city = \text{“Brooklyn”}} (branch)) \; \bowtie \; account)) \bowtie \; depositor)$$

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.

Join Ordering Example (Cont.)

- Consider the expression

$$\Pi_{customer\text{-}name}\ ((\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch)) \\ \bowtie\ account \bowtie\ depositor)$$

- Could compute $account \bowtie\ depositor$ first, and join result with

$$\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch)$$

  but $account \bowtie\ depositor$ is likely to be a large relation.

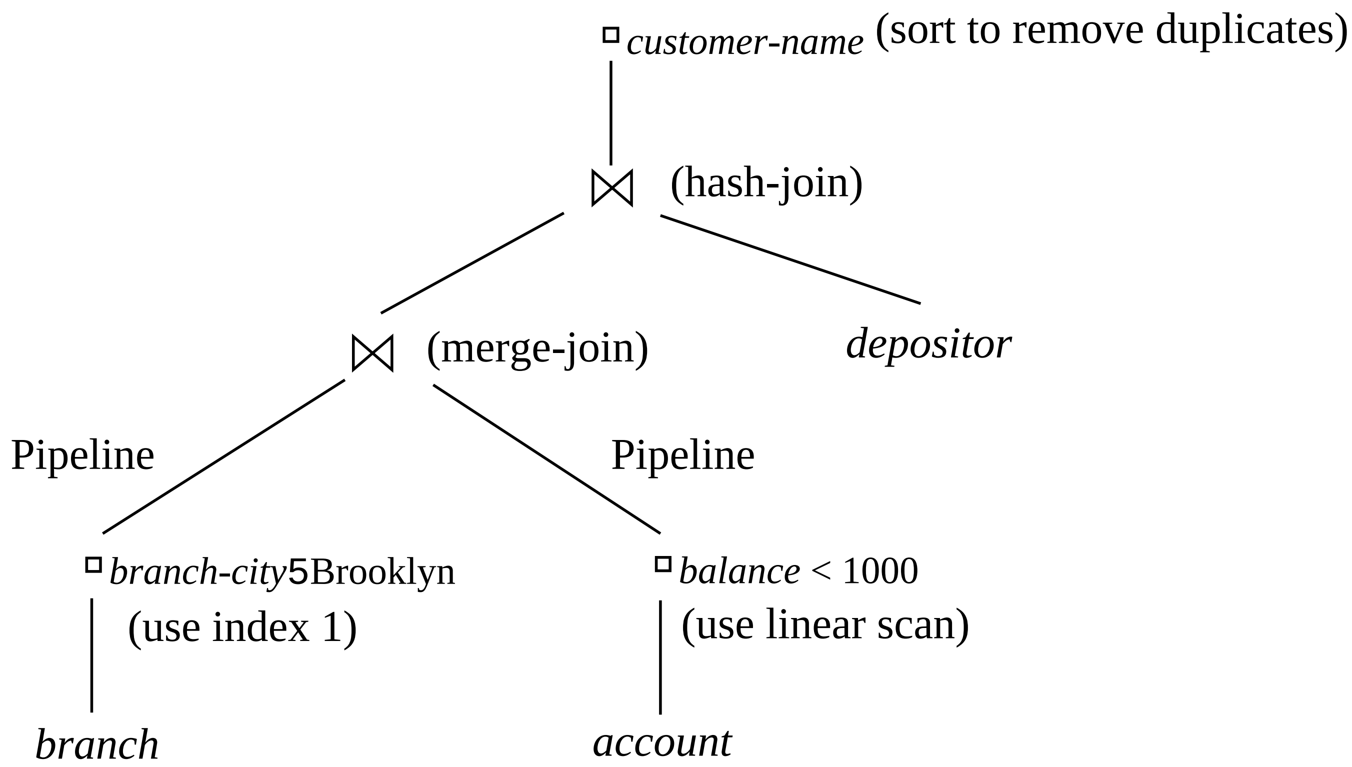
- Since it is more likely that only a small fraction of the bank's customers have accounts in branches located in Brooklyn, it is better to compute

$$\sigma_{branch\text{-}city = \text{“Brooklyn”}}\ (branch) \bowtie\ account$$

  first.

Evaluation Plan

An evaluation plan defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.



An evaluation plan

Choice of Evaluation Plans

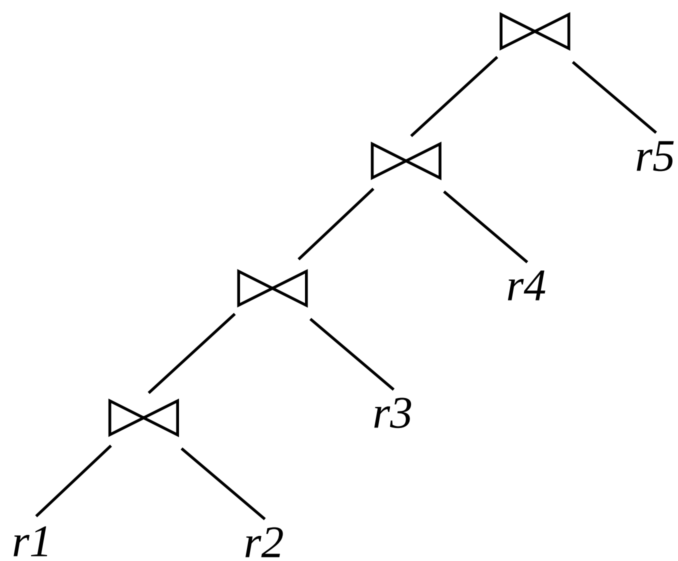
- Must consider the interaction of evaluation techniques when choosing evaluation plans: choosing the cheapest algorithm for each operation independently may not yield the best overall algorithm. E.g.
  - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation.
  - nested-loop join may provide opportunity for pipelining
- Practical query optimizers incorporate elements of the following two broad approaches:
  1. Search all the plans and choose the best plan in a cost-based fashion.
  2. Use heuristics to choose a plan.

Cost-Based Optimization

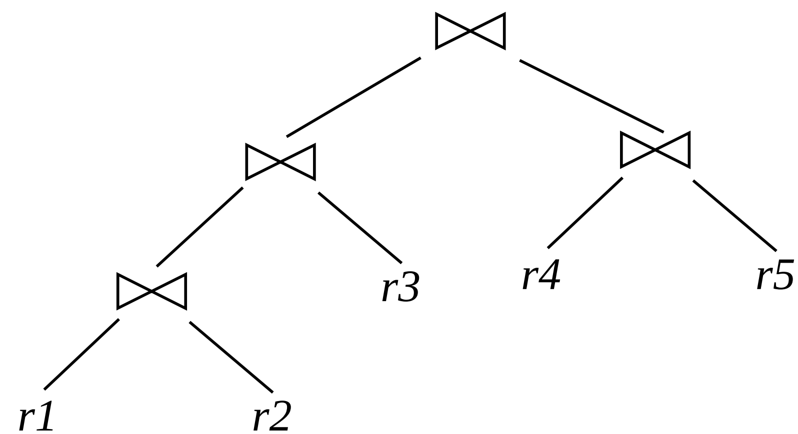
- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots r_n$.
- There are $(2(n-1))!/(n-1)!$ different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!
- No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of $\{r_1, r_2, \ldots, r_n\}$ is computed only once and stored for future use.
- This reduces time complexity to around $O(3^n)$. With $n = 10$, this number is 59000.

Cost-Based Optimization (Cont.)

- In **left-deep join trees**, the right-hand-side input for each join is a relation, not the result of an intermediate join.
- If only left-deep join trees are considered, cost of finding best join order becomes $O(2^n)$.



(a) Left-deep Join Tree



(b) Non-left-deep Join Tree

Dynamic Programming in Optimization

- To find best left-deep join tree for a set of n relations:
  - Consider n alternatives with one relation as right-hand-side input and the other relations as left-hand-side input.
  - Using (recursively computed and stored) least-cost join order for each alternative on left-hand-side, choose the cheapest of the n alternatives.
- To find best join tree for a set of n relations:
  - To find best plan for a set S of n relations, consider all possible plans of the form: $S_1 \bowtie (S - S_1)$ where S_1 is any non-empty subset of S.
  - As before, use recursively computed and stored costs for subsets of S to find the cost of each plan. Choose the cheapest of the $2^n - 1$ alternatives.

Interesting Orders in Cost-Based Optimization

- Consider the expression $(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$
- An **interesting sort order** is a particular sort order of tuples that could be useful for a later operation.
  - Generating the result of $r_1 \bowtie r_2 \bowtie r_3$ sorted on the attributes common with r_4 or r_5 may be useful, but generating it sorted on the attributes common to only r_1 and r_2 is not useful.
  - Using merge–join to compute $r_1 \bowtie r_2 \bowtie r_3$ may be costlier, but may provide an output sorted in an interesting order.
- Not sufficient to find the best join order for each subset of the set of n given relations; must find the best join order for each subset, for each interesting sort order of the join result for that subset. Simple extension of earlier dynamic programming algorithms.

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.

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.

Structure of Query Optimizers

- The System R optimizer considers only left-deep join orders. This reduces optimization complexity and generates plans amenable to pipelined evaluation.

  System R also uses heuristics to push selections and projections down the query tree.

- For scans using secondary indices, the Sybase optimizer takes into account the probability that the page containing the tuple is in the buffer.

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.
  - The Oracle7 optimizer supports a heuristic based on available access paths.
- 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.