Chapter 12: Query Processing

- Choice of Evaluation Plans
- Transformation of Relational Expressions
- Evaluation of Expressions
- Other Operations
- Join Operation
- Sort Time
- Selection Operation
- Measures of Query Cost
- Catalog Information for Cost Estimation
- Overview
Basic steps in Query Processing:

1. Parsing and translation
2. Optimization
3. Evaluation

Data

About Data Statistics

Query Output

Execution Engine

Query

Parser & Translator

Relational Algebra

Expression

Optimizer

Execution Plan

Data

Query
The query-execution engine takes a query-evolution plan, executes that plan, and returns the answers to the query.

**Evaluation**

- Parser checks syntax, verifies relations
- Translated into relational algebra
- Translated the query into its internal form

**Parsing and Translation**

Basic Steps in Query Processing (Cont.)
based on statistical information in the DBMS catalog.

- In an optimal evaluation plan, try to choose the one with the smallest cost:

  - Amongst all equivalent expressions, try to choose the one with the smallest cost.

- To optimize the query, we can use an index on balance to find accounts with balance > 2500.

Examples of expressions and their evaluation:

- \( \text{balance} > 2500 \): This expression can be evaluated in many different ways, depending on the available indexes.

- Given a relational algebra expression, equivalent expressions may have many equivalent variations.

**Optimization:** Finding the cheapest evaluation plan for a query.
\[
\left[ \frac{df}{du} \right] = q
\]

If tuples of \( r \) are stored together physically in a file, then:

- average number of records that satisfy equality on \( A \).
- \( SC(A) \) - selection cardinality of attribute \( A \) of relation \( r \).
- same as the size of \( \Pi A(r) \).
- \( V(A,r) \) - number of distinct values that appear in \( r \) for \( A \).
- \( f_r \) - blocking factor of \( r \), i.e., the number of tuples of \( r \) that fit into one block.
- \( s_r \) - size of a tuple of \( r \) in bytes.
- \( q_r \) - number of blocks containing tuples of \( r \).
- \( n_r \) - number of tuples in relation \( r \).
number of blocks at the leaf level of the index.

$B_I^2$ \# number of lowest-level index blocks in $I$, i.e., the

For a hash index, $LH$, is $I$.

$\lfloor \log \left( \forall \Lambda \right) \rfloor = \#LH$

A of relation $\mathcal{R}$ on attribute

For a balanced tree index (such as $B+$-tree) on attribute

For a balanced tree index such as $B+$-trees.

$\average$ fan-out of internal nodes of index $I$, for

Catalog Information about Indices
include cost of writing output to disk. We refer to the cost estimate of algorithm A as $E_A$. We do not often use worst case estimates.

Thus memory size should be a parameter while estimating cost; memory, as having more memory reduces need for disk access.

Costs of algorithms depend on the size of the buffer in main

It is assumed that all transfers of blocks have the same cost.

Therefore easy to estimate. Therefore number of block transfers relatively easy to estimate. Therefore number of block transfers

Typically disk access is the predominant cost, and is also

parallel system.

CPU time, or even communication overhead in a distributed or

Many possible ways to estimate cost, for instance disk accesses,

Measures of Query Cost
availability of indices  
* ordering of records in the file, or  
* selection condition, or  

- Linear search can be applied regardless of (finding record) on.

If selection is on a key attribute, \( A_1 \) \( \sum (z/q) = A_1 \) \( q = A_1 \) \( q = A_1 \).

- Cost estimate (number of disk blocks scanned) and test all records to see whether they satisfy the selection condition.

Algorithm A1 (linear search), scan each file block and test all that fulfill a selection condition.

File scan – search algorithms that locate and retrieve records

Selection Operation
estimate reduces to \[ \forall \mathcal{E} \]

Equality condition on a key attribute: \( \forall \mathcal{A}, (T, I) = 1 \).

- occupy

number of blocks that these records will

\[ \left\lfloor \frac{\sum_f}{(\forall \mathcal{A}) \mathcal{C}} \right\rfloor \]

* selection

number of records that will satisfy the

search on the blocks

estimate (number of disk blocks to be scanned):

\[ 1 - \left\lfloor \frac{\sum_f}{(\forall \mathcal{A}) \mathcal{C}} \right\rfloor + \left\lfloor (\forall q) \frac{\log_2 \mathcal{E}}{I} \right\rfloor = \forall \mathcal{E} \]

- Cost estimate (number of blocks of a relation are stored contiguously)

- Assume that the blocks of a relation are stored contiguously.

\( \forall \mathcal{E} \) (binary search) - Applicable if selection is an equality comparison on the attribute on which file is ordered.

Selection Operation (Cont.)
A secondary, B+ tree index for attribute `balance`

A primary, B+ tree index for attribute `branch-name`.

Assume the following indices exist on `account`:

1. `account` has 10,000 tuples.
2. `balance` account has 500 different balance values.
3. `branch-name` account has 50 branches.
4. 20 tuples of account fit in one block.

---

**Statistical Information for Examples**
(versus 500 for linear scan)

Total cost of binary search is 6 + 10 = 18 block accesses

\[
\log_2(500) = 9 \text{ block accesses}
\]

- A binary search to find the first record would take

- 200/20 = 10 blocks for these tuples

- Periphery branch

- Assume account is sorted on branch-name.

- relation: each block holds 20 tuples.

- Number of blocks is \(\text{blocks} = 500: 10,000 \text{ tuples in the}

\(\text{periphery}\text{-name}=\text{branch-name}\) account

• Selection Cost Estimate Example
If the search-key is not a candidate key,

- Retrieve multiple records (each may be on a different block)

\[
\forall \exists \text{SC} \left( \forall' \right) + 2^{\frac{\text{I}}{\text{I}}} H = 2^{\forall' \exists} \forall' \\
\text{if the search-key is not a candidate key.}
\]

- Retrieve a single record if the search-key is a candidate key

\[
\forall 9 \text{ (equality on search-key of secondary index)}.
\]

\[
\begin{pmatrix}
\forall 9
\end{pmatrix} + 2^{\frac{\text{I}}{\text{I}}} H = 2^{\forall' \forall} \forall
\]

records. Let the search-key attribute be \( A \).

- Retrieve multiple records (nonkey, equality)

\[
\forall 4 \text{ (primary index on nonkey, equality)}.
\]

\[
\begin{pmatrix}
\forall
\end{pmatrix} + 2^{\frac{\text{I}}{\text{I}}} H = 2^{\forall' \forall} \forall
\]

condition. The search-key's corresponding equality

- Retrieve a single record that satisfies the corresponding equality

\[
\forall 3 \text{ (primary index on candidate key, equality)}.
\]

- \( \forall \text{ on search-key of index}. \)

- \( \forall \text{ scan} \) – search algorithms that use an index; condition is

\[
\text{Selections Using Indices}
\]
This strategy requires 12 total block reads.

Therefore, 2 index blocks must be read. Between 3 and 5 leaf nodes and the entire tree has a depth of 2. 20 pointers per node, then the B+ tree index must have several index blocks must also be read. If B+ tree index stores several index blocks must also be read.

Several index blocks must also be read. If B+ tree index stores several index blocks must also be read.

Since the index is a clustering index, 200/20 = 10 block reads.

Since the index is a clustering index, 200/20 = 10 block reads.

Since the index is a clustering index, 200/20 = 10 block reads.

Since the index is a clustering index, 200/20 = 10 block reads.

10000/50 = 200 tuples of the account relation pertain to the branch primary index on branch-name.

Since the query is Q(branch-name = "primary-name", "account") with the primary index on branch-name.

Consider the query is Q(branch-name = "primary-name", "account") with the primary index on branch-name.

Cost Estimate Example (Indices)
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!)
\[
(\mathcal{D}_\theta)_{\theta = \theta_0} - \mu 
\]

**Negation:** \(\neg (\mathcal{D}_\theta)_{\theta = \theta_0} \cdot \theta^{\mu} \cdot \text{Estimated number of tuples:} \)

\[
\left(\frac{\mu}{u_\theta} - 1\right) \cdot \ldots \cdot \left(\frac{\mu}{u_\theta} - 1\right) \cdot \left(\frac{\mu}{u_\theta} - 1\right) \cdot \mu 
\]

**Disjunction:** \(\bigwedge_\theta (\mathcal{D}_\theta)_{\theta = \theta_0} \cdot \theta^{\mu} \cdot \text{Estimated number of tuples:} \)

\[
\frac{\mu}{u_\theta} \cdot \ldots \cdot \frac{\mu}{u_\theta} \cdot \mu 
\]

**Implication:** The tuples in the result is:

\[
\mathcal{D}_\theta (\mathcal{D}_\theta)_{\theta = \theta_0} \cdot \theta^{\mu} \cdot \text{Estimated for number of tuples in }\mathcal{D}_\theta \cdot \text{satisfy is given by } s^{\mu} 
\]

\[
\text{satisfy is the number of tuples in the relation } \mathcal{R} \cdot \text{satisfy } \theta \text{ is the number of tuples that a tuple } \in \mathcal{R} \cdot \text{satisfy } \theta \text{ is the probability that a tuple } \in \mathcal{R} \cdot \text{satisfy } \theta \text{ is the probability that a tuple } \in \mathcal{R} \cdot \text{satisfy } \theta \text{ is the probability that a tuple } \in \mathcal{R} \cdot \text{satisfy } \theta \text{ is the probability that a tuple } \in \mathcal{R} 
\]
scan.

if all conditions have available indices, otherwise use linear

\textit{11} (disjunctive selection by union of identifiers).

• have appropriate indices, apply test in memory.

\textit{10} (concurrent selection by intersection of identifiers).

• requires indices with record pointers, use corresponding index

\textit{9} (concurrent selection using multiple-key index).

• use buffer.

the least cost for \( \Theta \) (\( \text{Test other conditions in memory} \)). Test other conditions and algorithms \( \text{A7 through A7} \) that results in

\textit{8} (concurrent selection using one index).

Select a

\textbf{Algorithms for Complex Selections}
use the balance index.
- If both indices were non-clustered, it would be preferable to
  condition is less selective.

- Thus using branch-name index is preferable, even though its
  gives a cost estimate of 22 block reads.
10,000/500 = 20 accounts. Adding the index block reads, 
\( V(\text{balance, account}) = 500 \), so the selection would retrieve
- The balance index is non-clustered, and
- The branch-name index is clusterized, and if we use it the

Consider using Algorithm A8:

**where branch-name = „Perlybridge“ and balance = 1200**

Consider a selection on account with the following condition:

**Example of Cost Estimate for Complex Selection**
- The total estimated cost of this strategy is five block reads.

 - $S_1 \cup S_2$ contains one pointer.

 Since $n_{accoun} = 10000$, conservatively overestimate that

 - Estimate that one tuple in 50 * 500 meets both conditions.

 - Two or three sets of pointers and compute their intersection.

 Since each page, we read four index blocks to retrieve the

 - The number of pointers retrieved (20 and 200) fit into a

 - "PerfIndex" and balance = 1200.

 = $S_1 \cup S_2 = \text{set of pointers to records with branch-name} = \text{"PerfIndex".}$

 - Use index on branch-name to retrieve set $S_2$ of pointers to

   - Use the index on balance to retrieve set $S_1$ of pointers to

   Consider using algorithm A10:

   - Example (cont'd)
sort-merge is a good choice.

- For relations that don’t fit in memory, external sort can be used. For relations that fit in memory, techniques like quicksort can be used.

- Access for each tuple.

- To read the relation in sorted order, we may lead to one disk block.

- We may build an index on the relation, and then use the index.
buffer. (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. (b) Write the record to the output.

Select the first record in sort order from each of the buffers: buffer pages are empty:

Repeat output. Repeatedly do the following until all input to buffer output.

Step, use $\theta$ blocks of memory to buffer input runs, and 1 block.

2. Merge the runs: Suppose for now that $M > \frac{\theta}{M}$. In a single merge:

(c) Write sorted data to run $R_i$; increment $i$.
(b) Sort the in-memory blocks.

(a) Read $M$ blocks of relation into memory.

Do the following till the end of the relation:

I. Create sorted $R_i$ as follows. Let $i$ be 0 initially. Repeatedly:

Let $W$ denote memory size (in pages).

External Sort-Merge
Example: External Sorting Using Sort-Merge
Thus total number of disk accesses for external sort time:

\[ (W/q) \log_{W/q} |W| q \]

Total number of merge passes required:

\[ \left\lfloor (W/q) \log_{W/q} W \right\rfloor \]

– Total number of merge passes required: \( \left\lfloor \log_{W/q} W \right\rfloor \) is 2q except for final pass which doesn’t write out results

– Disk accesses for initial run creation as well as in each pass

Cost analysis:

merged into one.

– Repeated passes are performed till all runs have been

and creates runs longer by the same factor.

– A pass reduces the number of runs by a factor of \( W - 1 \),

– In each pass, contiguous groups of \( W \) runs are merged.

If \( W > q \), several merge passes are required.

External Sort—Merge (Cont.)
outer-level operations in a relational-algebra expression.

Join size estimates required, particularly for cost estimates for choice based on cost estimate

- Hash-Join
- Merge-Join
- Indexed-nested-Loop Join
- Block-nested-Loop Join
- Nested-Loop Join

Several different algorithms to implement joins

Join Operation
Illegal assume that customer-name in deposition is a foreign key on
an average, each customer has two accounts.

\[ \forall (\text{customer-name}, \text{deposition}) = 2000, \text{which implies that} \]

\[ q_{\text{deposition}} = 5000/50 = 100. \]

\[ f_{\text{deposition}} = 50, \text{which implies that} \]

\[ n_{\text{deposition}} = 5000. \]

\[ q_{\text{customer}} = 10000/25 = 400. \]

\[ f_{\text{customer}} = 25, \text{which implies that} \]

\[ n_{\text{customer}} = 10,000. \]

Catalog information for join example:

deposition \times customer

Join operation: Running Example
exactly \texttt{depositor} tuples, which is 5000.

depositor is a foreign key of customer, hence, the result has
in the example query depositor \texttt{customer} customer-name in

\bullet \text{symmetric:}

\text{If } H \cup S \text{ is a foreign key in } S \text{ referencing } H \text{ then the
tuples in } s, \text{ number of tuples in } r \times s \text{ is exactly the same as the number of
greater than the number of tuples in } s.

\text{If } H \cup S \text{ is a key for } H, \text{ then a tuple of } s \text{ will join with at most
one tuple from } r; \text{ therefore, the number of tuples in } r \times s \text{ is no
occupies } s^t + s^g \text{ bytes.}

\text{The Cartesian product } r \times s \text{ contains } n^t \times n^g \text{ tuples; each tuple

\textbf{Estimation of the Size of Joins}
The lower of these two estimates is probably the more accurate:

\[
\frac{(s, \forall) \Lambda}{s \cdot s}
\]

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

\[
\frac{(u, \forall) \Lambda}{s \cdot u}
\]

number of tuples in \( \Sigma \) is estimated to be:

If we assume that every tuple in \( R \) produces tuples in \( \Sigma \),

\[
\text{If } R \cup S \neq S \text{ } \forall \text{ is not a key for } R \text{ or } S.
\]
same as our earlier computation using foreign keys.

We choose the lower estimate, which, in this case, is the

\[ 5000 \times 1000/1000 = 5000 \]

and

- \( \Lambda(\text{customer-name, customer}) = 0000 \)
- \( \Lambda(\text{customer-name, deposit}) = 2500 \)

\( \Lambda(\text{customer}) \)

using information about foreign keys:

- Compute the size estimates for \( \text{depositor} \times \text{customer} \) without

Estimation of the size of joins (cont.)
use that relation as the inner relation. If the smaller relation fits entirely in main memory,

- expensive since it examines every pair of tuples in the two relations. It examines every pair of tuples in the two relations. It requires no indices and can be used with any kind of join.

\[ r \text{ is called the outer relation and } s \text{ the inner relation of the} \]

\[ \text{end} \]

if they do, add \( t_r \cdot t_s \) to the result.

\[ \theta \text{ test pair } (t_r, t_s) \text{ to see if they satisfy the join condition} \]

for each tuple \( t_r \) in \( r \) do begin

for each tuple \( t_s \) in \( s \) do begin

compute the theta join, \( r \, \land \, \theta \, s \)
Block nested-loops algorithm (next slide) is preferable.

- Cost estimate will be 500 disk accesses.
- If the smaller relation (depositor) fits entirely in memory, the disk access with customer as the outer relation: 
  \[ \text{cost estimate will be } 500 \times 400 + 100 = 1,000,400 \]
- If the smaller relation fits entirely in memory, the disk access with depositor as outer relation and 10000 * 100 = 2,000,100 disk accesses with
  \[ \text{cost estimate will be } 5000 \times 400 + 100 = 2,000,400 \]

Assuming the worst case memory availability scenario, cost accesses.

- Inner relation. This reduces the cost estimate to \( q^* + q^* \text{ block of each relation, the estimated cost is } u^* q^* \text{ disk access} \)
- If there is enough memory only to hold one

**Nested-Loop Join (Cont.)**
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 \( B' \) of \( B \) do
  For each block \( B' \) of \( B \) do
    For each tuple \( t \) of \( B \) do
      For each tuple \( t' \) of \( B \) do
        If they do, add \( t \cdot t' \) to the result.

Worst case: each block in the inner relation \( s \) is read only once

Tuple in the outer relation (instead of once for each
Use index on inner relation if available

(use of blocks remaining in buffer (with TRW replacement)

Scan inner loop forward and backward alternately, to make

reduces number of scans of inner relation greatly.

remaining two blocks to buffer inner relation and output.

for outer relation, where \( W = \) memory size in blocks, use

in block nested-loop, use \( W = 2 \) disk blocks as blocking unit

inner loop with first match

if equal join attribute forms a key on inner relation, stop

Improvements to nested-loop and block nested loop algorithms:

- \( q' + q^s \) block accesses.
- Worst case estimate: \( q' q^s + \) block accesses. Best case:

Block Nested-Loop Join (cont.)
tuples as the outer relation.

If indices are available on both \( r \) and \( s \), use the one with fewer

selection on \( s \) using the join condition.

- Cost of the join: \( q' \) + \( u' \) * \( c' \), where \( c \) is the cost of a single
  tuple in \( r \). For each tuple in \( r \), we perform an index lookup on \( s \).
- \( q' \) disk accesses are needed to read relation \( r \), and, for each
  page of the index.

Worst case: buffer has space for only one page of \( r \) and one

up tuples in \( s \) that satisfy the join condition with tuple \( t' \).

For each tuple \( t' \) in the outer relation \( r \), use the index to look

can construct an index just to compute a join.

can replace the scans.

join is an equi-join or natural join, more efficient index lookups

If an index is available on the inner loop's join attribute and

Indexed Nested-Loop Join
nested-loop join. This cost is lower than the 40, 100 accesses needed for a block. Since 
100 + 5000 * 5 = 25, 100 disk accesses. Since 
index node. Since customer has 10,000 tuples, the height of the tree is 4, 
attribute customer-name, which contains 20 entries in each 
join relation. Compute department ∩ customer, with department as the outer
must be matched

1. Join step is similar to the merge stage of the sort-merge
sorted on the join attributes.
2. Join algorithm. Main difference is handling of duplicate values in

merge—join
actual tuples efficiently. If one relation is sorted, and the other has a secondary B+-tree.

- Can be used only for equi-joins and natural joins.
- \( q^1 + q^2 \) plus the cost of sorting if relations are unsorted.
- Each tuple needs to be read only once, and as a result, each

```merge-join (cont.)```
$\forall i \in \text{Join_attrs}[t] \forall \mathcal{H} \in \text{partition}_{t}$

Initially empty. Each tuple $t \in \mathcal{H}$ is put in partition $\mathcal{H}$ 

denote partitions of $t$ tuples, each

$\forall i \in \text{Join_attrs}[t] \forall \mathcal{H} \in \text{partition}_{t}$

Initially empty. Each tuple $t \in \mathcal{H}$ is put in partition $\mathcal{H}$ 

denote partitions of $t$ tuples, each

the natural join.

Join attrs denotes the common attributes of $t$ and $s$ used in

$\forall i \in \text{Join_attrs}$ values to $0, 1, \ldots, \text{max}$, where

$\forall i \in \text{Join_attrs}$ values to $0, 1, \ldots, \text{max}$, where

as follows:

into sets that have the same hash value on the Join attributes.

A hash function $h$ is used to partition tuples of both relations.

Applicable for equi-joins and natural joins.

\textbf{Hash-Join}
in \( H \) and the \( s \) tuple in \( H \)

- If that value is hashed to some value \( j \), the \( r \) tuple has to be
  - have the same value for the join attributes.

An \( r \) tuple and an \( s \) tuple that satisfy the join condition will

partition since:

they do not need to be compared with \( s \) tuples in any other

\( H \) \( r \) tuples in \( H \) need only to be compared with \( s \) tuples in

Hash-Join (cont.)
Partitions of $r$

Partitions of $s$

Hash-Join (Cont.)
Input: Relation s is called the build input and t is called the probe.

Output the concatenation of their attributes.

1. Read the tuples in $H$ from disk one by one. For each tuple, locate each matching tuple $t$ in using the in-memory hash index.
2. For each partition, output buffer for each partition.
3. Partition and build an in-memory hash index.

When partitioning a relation, one block of memory is reserved as the partitioning key. When partitioning the relations $s$ using hashing function $h$, $s$ is partitioned according to the hash algorithm.
different hash function. $H^2$ must be similarly partitioned.

Hash-table overflow occurs in partition if $H^2$ does not fit in memory. Can resolve by further partitioning using $H^2$.

Block size of 4 KB.

Relations of 1 GB or less with memory size of 2 MB:

- Rarely required, e.g., recursive partitioning not needed.
- Use same partitioning method on $r$.
- Further partition the $W-1$ partitions using a different hash.
- $W$-partition $s-1$ ways.
- Instead of partitioning max ways, partition $s-1$ ways.
- Recursive partitioning required if number of partitions max is greater than number of pages of memory.
- Each $H^2$ should fit in memory.

The value $\text{max}$ and the hash function $h$ is chosen such that

Hash-Join algorithm (cont.)
Total cost estimate is:

\[ s^q + q + \lceil 1 - (s^q)^{\log_q -1} \rceil \cdot (s^q + \lfloor q \rfloor) \cdot \max \]

Cost of Hash-Join
(Ignore cost of writing partially filled blocks).

Therefore total cost: \(3(100 + 400) = 1500\) block transfers

Therefore total cost: \(3(100 + 400) = 1500\) block transfers

80. This is also done in one pass.

Similarly, partition customer into five partitions, each of size

in one pass.

partitions, each of size 20 blocks. This partitioning can be done

depositor is to be used as build input. Partition it into five

\[ \text{depositor} = 100 \mathrm{and} \text{depositor} = 400. \]

Assume that memory size is 20 blocks.

\[ \text{customer} \times \text{depositor} \]

Example of Cost of Hash Join
Hybrid hash-join most useful if $M^{\delta}$.

Hybrid hash-join, instead of 1500 with plain hash-join, 3(80 + 320) + 20 = 1300 block transfers with hybrid hash-join, instead of being written out and read back in, the cost is used right away for probing, instead of being written partially. All blocks, the cost is.

Ignoring the cost of writing partitions each of size 80; the first is used right away for probing, instead of being.

customer is similarly partitioned into five partitions each of which is used for buffering the other four partitions. If occupies 20 blocks, one block is used for input, and one block keeps the first of the partitions of the build relation in memory.

Keep the first of the partitions of the build relation in memory. With a memory size of 20 blocks, decomposition can be partitioned into five partitions, each of size 20 blocks.

Input is bigger than memory.

Useful when memory sizes are relatively large, and the build

Hybrid Hash-Join
\[(s^u \bowtie J) \cap \cdots \cap (s^z \bowtie J) \cap (s^1 \bowtie J)\]

Compute as the union of the records in individual joins \( J \):

\[s^u \lor \cdots \lor s^z \lor s^1\]

Join with a disjunctive condition:

- Test these conditions as tuples in \( J \) are generated.
- If these conditions are satisfied then tuples in the intermediate result \( s^u \bowtie J \lor \cdots \lor s^z \bowtie J \lor s^1 \bowtie J \) are included in the final result.
- Compute the result of one of the simpler joins \( J \):

Join with a conjunctive condition:

**Complex Joins**
two relations.

operation that is more efficient than implementing two joins of

- Strategy 3 combines two operations into one special-purpose

  - Each tuple of deposit is examined exactly once.

    - Tuple in customer and the corresponding tuples in loan,

    - For each tuple t in deposit, look up the corresponding

  - Strategy 3. Perform the pair of joins at once. Build an index

on loan for loan-number and on customer for customer-name.

- Strategy 2. Compute loan X deposit first, and then join the

  - Compute loan X (deposit X customer)

  - Compute deposit X customer; use result to

- Strategy 1. Compute deposit X (deposit X customer)

  - Join involving three relations: loan X customer

Complex Joins (cont.)
tuple followed by duplicate elimination.

projection is implemented by performing projection on each bucket.

Hashing is similar — duplicates will come into the same generation as well as all intermediate merge steps in external sort.

Optimization: duplicates can be deleted during run.

On sorting duplicates will come adjacent to each other, and sorting can be implemented via hashing or

Other Operations
merge-join after sorting, or variant of hash-join.

Set operations (\( \cup \) and \(-\)) can either use variant of

aggregate values.

Generation and intermediate merges, by combining partial

optimization: combine tuples in the same group during run

applied on each group.

Group together, and then the aggregate functions can be

aggregation can be implemented in a manner similar to

Other Operations (cont.)
the hash index to the result.
    Index, delete it from the index. Add remaining tuples in
    index, already there in the hash index. $H^s \setminus r$ if it is there in the hash
    result. If they are already there in the hash index.

$H \cup r$ : output tuples in $H^s$ to the result if they are
not already in it. Then add the tuples in the hash index to
bring it into memory.

1. Partition both relations using the same hash function,

2. Process each partition, as follows. Using a different hashing

3. $H \cap r$ : Add tuples in $H^s \setminus r$ to the hash index if they are not

Function, build an in-memory hash index on $H$ after it is

E.g. Set operations using hashing:

Other Operations (Cont.)
Similarly:
- Right outer-join and full outer-join can be computed output padded with nulls.
  every tuple from \( t \) that do not match any tuple in \( s \).
- Modify merge-join to compute \( t \) \( s \) during merging, for

\[
J \left( t \cap s \right) \cap \Pi_{t}(s)
\]
- In \( t \), non-participating tuples are those in

\[
\begin{align*}
\text{Example:} \\
\text{by modifying the join algorithms.}
\end{align*}
\]
- A join followed by addition of null-padded non-participating tuples
  can be computed either as

\[
\text{Outer Join (cont.)}
\]
compute the projection on customer-name.
then compute and store its join with customer', and finally

E.g., in Figure below, compute and store \( \text{balance} > 2500 \) (account);

temporary relations to evaluate next-level operations.
Use intermediate results materialized into
the lowest-level. Use intermediate results materialized into

Materialization: evaluate one operation at a time, starting at

Evaluation of Expressions
Producer-driven. Pipelines can be executed in two ways: demand-driven and the operation. Generate output tuples even as tuples are received for inputs to the operation. For pipeline reading to be effective, use evaluation algorithms that temporarily relation to disk. Pipelining may not always be possible — e.g., sort, hash-join.

• Much cheaper than materialization: no need to store a temporary relation.

• Pipeline: evaluate several operations simultaneously.

• E.g., in expression in previous slice, don’t store result of the next. Pass the results of one operation on to the next. Evaluation of Expressions (Cont.)
process is called cost based optimization. Based on estimated cost, the cheapest plan is selected. The equivalent one.

- Use equivalence rules to transform an expression into an equivalent one.

2. annotating resultant expressions to get alternative query

1. Generating logically equivalent expressions

Two steps:

- Generation of query-evaluation plans for an expression involves

Transformation of Relational Expressions
Equivalent expressions

(a) Initial Expression Tree
(b) Transformed Expression Tree

Relationships generated by two equivalent expressions have the same
attributes may be ordered differently.

\[ (\mathcal{A})^\mathcal{T} = (\cdots ((\mathcal{A})^u)^v \cdots)^z \mathcal{T} \mathcal{II} \]

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

\[ (\mathcal{A} \theta \mathcal{V} \mathcal{I}) \mathcal{II} = (\mathcal{A} \mathcal{I} \mathcal{II} \mathcal{I}) \mathcal{II} \theta \mathcal{O} \]

3. Only the last in a sequence of projection operations is needed.

\[ ((\mathcal{A})^\mathcal{I} \mathcal{O})^z \mathcal{O} = ((\mathcal{A})^\mathcal{Z} \mathcal{O})^\mathcal{I} \mathcal{O} \]

2. Selection operations are commutative.

\[ ((\mathcal{A} \mathcal{Z} \mathcal{O}) \mathcal{I} \mathcal{O} = (\mathcal{A} \mathcal{Z} \mathcal{O})^\mathcal{W} \mathcal{I} \mathcal{O} \]

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

Equivalence Rules

---

**Equivalence Rules**
where involves attributes from only \( \theta \) and \( \exists \).

\[
(\exists \theta \times \exists \theta^\prime) \times (\exists \theta^\prime \times \exists \theta) = (\exists \theta \times \exists \theta) \times (\exists \theta \times \exists \theta^\prime)
\]

(b) \( \Pi \) \( \Lambda \) \( \theta \) \( \Lambda \) \( \Pi \) \( \theta \)

\[
(\exists \theta \times \exists \theta) \times \exists \theta = \exists \theta \times (\exists \theta \times \exists \theta)
\]

6. (a) \( \Pi \) \( \Lambda \) \( \theta \) \( \Lambda \) \( \Pi \) \( \theta \)

5. \( \Pi \) \( \Lambda \) \( \theta \) \( \Lambda \) \( \Pi \) \( \theta \)

Equivalence Rules (Cont.)
\[(\mathcal{E}^\theta \theta \mathcal{E}) \theta \mathcal{E} \mathcal{E}^\theta \theta \mathcal{E}) = (\mathcal{E}^\theta \mathcal{E} \mathcal{E}^\theta \mathcal{E})^\theta \theta \mathcal{E}
\]

only the attributes of \( \mathcal{E} \).

When \( \mathcal{E}^\theta \theta \mathcal{E} \) involves only the attributes of \( \mathcal{E} \) and involves \( \mathcal{E} \).

\[
\mathcal{E} \theta \mathcal{E} \theta \mathcal{E}^\theta \theta \mathcal{E} = (\mathcal{E} \theta \mathcal{E} \mathcal{E}^\theta \theta \mathcal{E})^\theta \theta \mathcal{E}
\]

one of the expressions \( \mathcal{E} \theta \mathcal{E} \theta \mathcal{E} \) being joined.

When all the attributes in \( \theta \) involve only the attributes of \( \mathcal{E} \).
\[(\varepsilon \theta \varepsilon)(\varepsilon \theta \varepsilon) = (\varepsilon \theta \varepsilon)(\varepsilon \theta \varepsilon)\]

Involved in join condition, \(\theta\) that are not in \(\varepsilon\) and \(\varepsilon\) that are not in \(\varepsilon\), and let \(\varepsilon\) be attributes of \(\varepsilon\) that are involved in join condition, \(\theta\) that are not in \(\varepsilon\) and \(\varepsilon\) that are not in \(\varepsilon\), and let \(\varepsilon\) be attributes of \(\varepsilon\) and \(\varepsilon\) respectively. Let \(\varepsilon\) and \(\varepsilon\) be sets of attributes from \(\varepsilon\) and \(\varepsilon\), respectively. Let \(\varepsilon\) and \(\varepsilon\) be sets of attributes from \(\varepsilon\) and \(\varepsilon\), respectively. Let \(\varepsilon\) be attributes of \(\varepsilon\) that are involved in join condition, \(\theta\) that are not in \(\varepsilon\) and \(\varepsilon\) that are not in \(\varepsilon\), and let \(\varepsilon\) be attributes of \(\varepsilon\) and \(\varepsilon\) respectively.

Consider a join \(\varepsilon \theta \varepsilon\) and \(\varepsilon \theta \varepsilon\) where \(\theta\) involves only attributes from \(\varepsilon\) that are involved in join condition, \(\theta\) that are not in \(\varepsilon\) and \(\varepsilon\) that are not in \(\varepsilon\), and let \(\varepsilon\) be attributes of \(\varepsilon\) and \(\varepsilon\) respectively.

The projection operation distributively over the theta join operation as follows:

Equivalence Rules (Cont.)
\[(\mathcal{Z} \cap \mathcal{Y}) \cap (\mathcal{Z} \cap \mathcal{Y}) = (\mathcal{Z} \cap \mathcal{Y}) \cap \mathcal{Y}\]

12. The projection operation distributes over the union operation.

\[\mathcal{Z} \cap (\mathcal{Y} \cap \mathcal{Y}) = (\mathcal{Z} \cap \mathcal{Y}) \cap \mathcal{Y}\]

For difference and intersection, we also have:

\[(\mathcal{Z} \setminus (\mathcal{Y} \setminus \mathcal{Y})) \cap (\mathcal{Z} \setminus (\mathcal{Y} \setminus \mathcal{Y})) = (\mathcal{Z} \setminus \mathcal{Y}) \setminus (\mathcal{Y} \setminus \mathcal{Y})\]

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

10. Set union and intersection are associative.

\[\mathcal{Y} \cup \mathcal{Z} = \mathcal{Z} \cup \mathcal{Y}\]

\[\mathcal{Y} \cap \mathcal{Z} = \mathcal{Z} \cap \mathcal{Y}\]

9. The set operations union and intersection are commutative (set equivalence rules (cont.).
The relation to be joined.

- Performing the selection as early as possible reduces the size of
  \[ (\text{account} \times \text{depositor}) \times (\text{branch}_{\text{branch-city} = \text{Brooklyn}}) \]

\[ \text{customer-name} \]

- Transformation using Rule 7a.

\[ (\text{branch} \times \text{account} \times \text{depositor}) \times (\text{branch}_{\text{branch-city} = \text{Brooklyn}}) \]

\[ \text{customer-name} \]

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

**Example**
Thus a sequence of transformations can be useful:

\[
\text{branch} \quad \text{city} = \text{Brooklyn} \quad \text{branch} \quad \text{name} = \text{John} \quad \text{account} \quad \text{balance} > 1000
\]

Second form provides an opportunity to apply the „partition“ selection early, resulting in the subexpression:

\[
\text{branch} \quad \text{name} = \text{John} \quad \text{account} \quad \text{balance} > 1000
\]

Transformation using join associativity (Rule 6a):

\[
((\text{branch} \quad \text{name}) \quad \text{account} \quad \text{balance} > 1000)
\]

Brooklyn branch whose account balance is over $1000.

Query: Find the names of all customers with an account at a
Push projections using equivalence rules 8a and 8b; eliminate unnecessary attributes from intermediate results to get:

\[ \text{branch-name, branch-city, assets, account-number, balance} \]

We obtain a relation whose schema is:

\[ \text{account \ \land \ \text{branch-name = "Brooklyn"} \ \text{branch-city, account-number, depositor}} \]

When we compute:

\[ \text{account \ \land \ \text{branch-name = "Brooklyn"} \ \text{branch-city, account-number, depositor}} \]

Example Operation
so that we compute and store a smaller temporary relation,

\[ \varepsilon.t \Join (\tau.t \Join I.t) \]

If \( \varepsilon.t \Join I.t \) is quite large and \( \varepsilon.t \Join \tau.t \) is small, we choose

\[ (\varepsilon.t \Join \tau.t) \Join I.t = \varepsilon.t \Join (\tau.t \Join I.t) \]

For all relations \( \tau.t \), \( I.t \), and \( \varepsilon.t \), and \( \varepsilon.t \).
First,

\[ \text{branch} \cap \text{(branch)} \text{ \( \text{branch-city} = \text{Brooklyn} \)} \]

better to compute

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

but account \( \text{depositor} \cap \text{account} \) is likely to be a large relation.

\[ \text{branch} \cap \text{(branch)} \text{ \( \text{branch-city} = \text{Brooklyn} \)} \]

Could compute account \( \text{depositor} \cap \text{account} \) first, and join result with

\[ \text{customer-name} \cap \text{(branch)} \text{ \( \text{branch-city} = \text{Brooklyn} \)} \]

Consider the expression

• Join \( \text{Orders Example} \) (Cont.)
An evaluation plan defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.
2. Use heuristics to choose a plan.
   cost-based fashion.

1. Search all the plans and choose the best plan in a
   following two broad approaches:

   Practical query optimizers incorporate elements of the
   - nested-loop join may provide opportunity for pipelining
     aggregation.

   a sorted output which reduces the cost for an outer level
   - merge-join may be costlier than hash-join, but may provide
     algorithm.

   E.g. each operation independently may not yield the best overall

   choosing evaluation plans: choosing the cheapest algorithm for
   Must consider the interaction of evaluation techniques when

Choice of Evaluation Plans
This number is 59000.

This reduces the complexity to around $O(3^n)$ with $n = 10$.

The number is computed only once and stored for future use.

No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of $\{u_1, u_2, \ldots, u_t\}$ is found.

The number is greater than 176 billion.

Expression with $n = 7$, the number is 66,7280, with $n = 10$.

There are $(2^7 - 1)(2^7 - 2)/2$ different join orders for above.

Consider finding the best join order for $u_t \land u_{t-1} \land \ldots \land u_1$.
(a) Non-left-deep Join Tree

(b) Left-deep Join Tree

If only left-deep join trees are considered, cost of finding best join order becomes \(O(n^2)\).

In left-deep join trees, the right-hand-side input for each relation is a relation, not the result of an intermediate join.
cheapest of the \(2^n-1\) alternatives.

Choose the non-empty subset of \(S\). As before, use recursively computed and stored costs for any possible plans of the form: \(S \times S'\) where \(S'\) is any possible plan for a set \(S'\) of \(n\) relations. Consider all possible plans of the form: \(S \times S'\) where \(S'\) is any possible plan for a set \(S'\) of \(n\) relations. Consider all possible plans of the form: \(S \times S'\) where \(S'\) is any possible plan for a set \(S'\) of \(n\) relations.

To find best join tree for a set of \(n\) relations: •

cheapest of the \(n\) alternatives. Choose the order for each alternative on left-hand-side. Using (recursively computed and stored) least-cost join input and the other relations as left-hand-side input. Consider \(n\) alternatives with one relation as right-hand-side.

To find best left-join tree for a set of \(n\) relations: •

Dynamic Programming in Optimization
algorithms.

Subsets or super-extensions. Simple extension of earlier dynamic programming subset for each interesting sort order of the join result for that subset, for each interesting sort order of the \textit{given} relations. must find the best join order for each set of \textit{given} relations. Not sufficient to find the best join order for each subset of the

but may provide an output sorted in an interesting order.

– Using merge-join to compute \textit{t} \textit{t} \textit{t} 3 may be costlier,

and \textit{t} 2 is not useful.

– Generalizing \textit{t} 1 3 2 \textit{t} 2 \textit{t} 3 sorted on the attributes common to only \textit{t} 1

attributes common with \textit{t} 4 or \textit{t} 5 may be useful, but

– Generalizing the result of \textit{t} \textit{t} 1 2 \textit{t} 3 \textit{t} 3 that could be useful for a later operation.

An interesting sort order is a particular sort order of tuples

Consider the expression (t 1 \textit{t} 1 \textit{t} 2 \textit{t} \textit{t} 3)
With partial cost-based optimization:

- Some systems use only heuristics, others combine heuristics and other similar operations.

  - Perform most restrictive selection and join operations before projection early (reduces the number of attributes)
  - Perform selection early (reduces the number of tuples)

  Performance:

    - Heuristic optimization transforms the query tree by using a set of rules (but not in all cases) to improve execution
    - Cost-based optimization is expensive, even with dynamic programming

[Heuristic Optimization]
execute them using pipelining.

6. Identify those subqueries whose operations can be pipelined, and

(Equality rules 3, 8a, 8b, 12).

projection attributes, creating new projections where needed.

5. Deconstruct and move as far down the tree as possible. Lists of

selection condition by join operations (Equality rule 4a).

4. Replace Cartesian product operators that are followed by a

produce the smallest relations (Equality rule 6).

3. Execute first those selection and join operations that will

possible execution (Equality rules 2, 7a, 7b, 11).

2. Move selection operations down the query tree for the earliest

1. Deconstruct consecutive selections into a sequence of single

Steps in Typical Heuristic Optimization
is in the buffer.

For scans using secondary indices, the database optimizer takes
down the query tree.

System R also uses heuristics to push selections and projections
amenable to pipelined evaluation.

This reduces optimization complexity and generates plans
that System R optimizer considers only if they join orders.
Slow disk accesses.

Query-execution time, particularly by reducing the number of

This expense is usually more than offset by savings at

imposes a substantial overhead.

Even with the use of heuristics, cost-based query optimization

available access paths.

The Oracle optimizer supports a heuristic-based on

followed by cost-based join-order optimization.

on the nested-block concept of SQL: heuristic rewriting

System R and Starburst use a hierarchically procedure-based

generation of alternative access plans.

Some query optimizers integrate heuristic selection and the