Overview of Query Optimization

- **Input**: SQL query
- **Output**: Query Plan: Tree of Relational algebra operators, with choice of algorithm for each operator

Main issues:
- For a given query, what plans are generated/considered?
  - Algorithm to search plan space for cheapest (estimated) plan.
- How is the cost of a plan estimated?
  - Using the cost formulas studied so far + assumptions

Ideally: Want to find best plan.
Practically: Avoid worst plans!
We will study the System R approach.
Why System R Optimizer

- Most widely used currently; works well for < 10 joins.
- **Cost estimation:** Approximate art at best.
  - Statistics, maintained in system catalogs, are used to estimate cost of operations and result sizes.
  - Considers combination of CPU and I/O costs.
- **Plan Space:** Too large, must be pruned.
  - Only the space of *left-deep plans* is considered.
    - Left-deep plans allow output of each operator to be pipelined into the next operator without storing it in a temporary relation.
  - Cartesian products avoided.

Types of Qeps

![Figure 4. Left-Deep, Bushy, and Right-Deep Plans.](image)
**Schema for Examples**

Sailors \((sid: \text{integer}, \ sname: \text{string}, \ \text{rating}: \text{integer}, \ \text{age}: \text{real})\)

Reserves \((sid: \text{integer}, \ bid: \text{integer}, \ \text{day}: \text{dates}, \ \text{rname}: \text{string})\)

- Similar to old schema; \(rname\) added for variations.
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

**Motivating Example**

- Cost: 500+500*1000 or 500500 I/Os
- By no means the worst plan!
- Misses several opportunities: selections could have been `pushed’ earlier, no use is made of any available indexes, etc.
- Goal of optimization: To find more efficient plans that compute the same answer.
Relational Algebra Equivalences

- Allow us to choose different join orders and to `push’ selections and projections ahead of joins.

  - **Selections:** \( \sigma_{c_1 \land \ldots \land c_n}(R) \equiv \sigma_{c_1}(\ldots(\sigma_{c_n}(R)) \) (Cascade)
    \[
    \sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))
    \] (Commute)

  - **Projections:** \( \pi_{a_1}(R) \equiv \pi_{a_1}(\ldots(\pi_{a_n}(R)) \) (Cascade)

  - **Joins:** \( R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \) (Associative)
    \[
    (R \bowtie S) \equiv (S \bowtie R)
    \] (Commute)

Show that: \( R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S \)

Alternative Plans 1
(No Indexes)

- **Main difference:** push selects.

- With 5 buffers, cost of plan:
  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, under uniform distribution).
  - Scan Sailors (500) + write temp T2 (250 pages, if we have 10 ratings).
  - Sort T1 (2*10), sort T2 (2*250), merge (10+250)
  - Total: 4060 page I/Os.

- If we used BNL join, join cost = 10+4*250, total cost = 2770.

- If we `push’ projections, T1 has only \( sid \), T2 only \( sid \) and \( sname \):
  - T1 fits in 3 pages, cost of BNL drops to under 250 pages, total < 2000.
Alternative Plans 2
With Indexes

- With clustered index on bid of Reserves, we get 100,000/100 = 1000 tuples on 1000/100 = 10 pages.
- INL with pipelining (outer is not materialized).
  - Projecting out unnecessary fields from outer doesn’t help.
- Join column sid is a key for Sailors.
  - At most one matching tuple, unclustered index on sid OK.
- Decision not to push rating>5 before the join is based on availability of sid index on Sailors.
- Cost: Selection of Reserves tuples (10 I/Os); for each, must get matching Sailors tuple (1000*1.2); total 1210 I/Os.

Query Blocks: Units of Optimization

- An SQL query is parsed into a collection of query blocks, and these are optimized one block at a time.
- Nested blocks are usually treated as calls to a subroutine, made once per outer tuple. (This is an oversimplification, but serves for now.)
- For each block, the plans considered are:
  - All available access methods, for each reln in FROM clause.
  - All left-deep join trees (i.e., all ways to join the relations one-at-a-time, with the inner reln in the FROM clause, considering all reln permutations and join methods.)
Optimization – SQL to a Plan

- There are 2 major phases in optimization [Freytag 89, Info. Processing]
  - Query modification (high level or logical optimization)
    - view substitution
    - equivalence transformations
      - syntactic, semantic
    - Access control predicate substitution
  - Query evaluation plan (QEP) generation (low level or physical level optimization)
    - the QEP generator
    - the search strategy
    - the cost function

Optimization

- Parsing
- Query validation
- View resolution
- optimization
- Plan compilation

- Generalization [Vldb 88]
  - There may be more than 2 levels or phases.
  - The process of optimization can be viewed in terms of:
**Optimization**

- Levels of abstraction
  - semantic transformations
  - view transformations
  - query rewriting
  - physical access plan level
  - merging physical access plans
- Transformation from one level of abstraction to the next level
  - transformations
  - heuristics or actual costs for determining pruning/retaining a subset

---

**Physical vs. logical algebra**

- Physical and logical algebra
  Physical algebra is equivalent to but quite different from the logical algebra of the data model of the database system
- Relational algebra is a logical algebra
- Specific algorithms and therefore cost functions are associated with physical operators, not with logical algebra operators.
- For lack of algorithmic specs, logical operators are not directly executable
**Optimization contd.**

 Intersection

 Set A  Set B

 Merge-join (intersection)

 Sort  sort

 File scan A  File scan B

 Logical operator  algorithm

---

**Types of QEP’s**

![Diagram of different QEP types]

*Figure 4. Left-Deep, Bushy, and Right-Deep Plans.*
Optimization Strategies

- Static [used by System R, and by most commercial DBMSs]
  - done prior to the execution of the query
  - relies on the accuracy of statistical information and the estimate for the size of intermediate relations
  - compile and store for later or repeated execution; some changes to the database structure may invalidate the QEP and force recompilation
  - can use exhaustive strategy space generation (as the cost can be amortized over a number of executions of the query)

Optimization Strategies contd.

- Dynamic
  - is done at run time
  - can use exact information on the size of intermediate results. If coupled with exhaustive approach, can produce an optimal plan
  - useful where the database is volatile, queries are ad hoc, and are not executed repeatedly
  - the cost of optimization cannot be amortized over multiple executions of the same query
  - this approach is immune to changes in the structure of the database
**Optimization Strategies contd.**

- **Hybrid**
  - Usually starts with a static QEP but changes the sequence based upon the size of the intermediate results (or the difference between estimated and actual sizes)
  - Tuning the QEP at run time as you go along, and if there is a need

- dynamic approach is lazy evaluation
- static approach is eager evaluation

**State Space Generation Strategies**

- **Exhaustive**
  - depth first
  - breadth first (used by System R with dynamic programming)

- **Others**
  - greedy (University Ingres)
  - A* (used in AI, by Sellis in MQO)
  - Branch and Bound (used by Grant and Minker)

- **Intermediate Results**
  - Streaming Vs. creating temporary relations
Optimization contd.

- Synchronization and data transfer
  - Temporary files
  - Interprocess communication
  - Iterative algorithm (may be rule-based)
    - Best; but assumes that ALL computations can be cast into this form
    - operators schedule each other within a single process
    - Need a notion of granule (typically a record) to iterate over all granules

How to implement iterators

- Separate functions
  - prepare an operator for producing data
  - to produce an item
  - to perform final housekeeping
- Example: File Scan
  - Open
  - Next
  - data
Examples of iterator functions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Open</th>
<th>Next</th>
<th>Close</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan</td>
<td>Open file</td>
<td>Read next item</td>
<td>Close file</td>
</tr>
<tr>
<td>Print</td>
<td>Open input</td>
<td>Call next on input; format the item on screen</td>
<td>Close input</td>
</tr>
<tr>
<td>Select</td>
<td>Open input</td>
<td>Call next on input until an item qualifies</td>
<td>Close input</td>
</tr>
<tr>
<td>Hash join (without overflow resolution)</td>
<td>Allocate hash directory; open left &quot;build&quot; input; build hash table calling next on input; close build input; open right &quot;probe&quot; input</td>
<td>Call next on probe input until a match is found</td>
<td>Close probe input, deallocate hash directory</td>
</tr>
<tr>
<td>Merge join</td>
<td>Open both inputs</td>
<td>Get next item from input with smaller key until a match is found</td>
<td>Close both inputs</td>
</tr>
<tr>
<td>Sort</td>
<td>Open input; build all initial run files calling next on input and quicksort or replacement selection; close input; merge run files until only one merge step is left; open the remaining run files</td>
<td>Determine next output item from read new item from the correct run file</td>
<td>Destroy remaining run files</td>
</tr>
</tbody>
</table>

Figure 5. Two Operators in a Volcano Query Plan.
**System R Optimization**

- **Strategy**
  - Exhaustive search of the solution space where the best execution plan is selected among (almost) all possible plans.

- **Static optimization using statistical information**

- **3 factors**
  - consider I/O and CPU cost
  - instead of using only “thumb rules”, use statistical information (in contrast to Ingres optimizer)
  - considers ordering of the result tuples produced by an execution plan

---

**System R Optimization contd.**

- Minimizes the weighted sum of predicted I/Os and cpu cost in processing an SQL statement

\[
\text{Cost} = \text{page fetches} + W \times \text{cpu cost (RSS calls)}
\]

where page fetches represent I/O costs

\( W \) indicate whether the system is I/O or CPU bound (typically 0.1 to 0.3);

RSS calls represents CPU cost
**System R Optimization contd**

- **File scan** touches each non-empty page in the file (called segment scan in system R).

- An **index scan** touches each leaf index page only once (for retrieving the entire relation). However, data pages may be retrieved more than once.

- For a **clustered index scan**, not only each leaf index page, but also each data page containing a tuple from that relation is touched only once.

---

**Scans**

- **File scan**
Scan using a Clustered B+ Tree

Index (Directs search)

Data Entries ("Sequence set")

Data Records

Always better than external sorting!

Scan using a Unclustered B+ Tree

Index (Directs search)

Data Entries ("Sequence set")

Data Records
Cost Estimation

- For each plan considered, must estimate cost:
  - Must estimate cost of each operation in plan tree.
    - Depends on input cardinalities.
    - We’ve already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
  - Must estimate size of result for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.

- We’ll discuss the System R cost estimation approach.
  - Very inexact, but works ok in practice.
  - More sophisticated techniques known now.

Cost parameters

- Selectivity
  - defined as the ratio of the number of records (tuples) that satisfy the condition to the total number of records (tuples) in the file (relation)

- Join selectivity
  - estimates the percentage of records in a file that will be joined with records in other file (depends on both relations as well as join attributes and predicates)

- Cardinality of relations: size
- other: Number of unique values in an attribute domain
Statistics and Catalogs

- Need information about the relations and indexes involved. **Catalogs** typically contain at least:
  - # tuples (NTuples) and # pages (NPages) for each relation.
  - # distinct key values (NKeys) and NPages for each index.
  - Index height, low/high key values (Low/High) for each tree index.

- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.

- More detailed information (e.g., histograms of the values in some field) are sometimes stored.

Size Estimation and Reduction Factors

- Consider a query block:
  - Maximum # tuples in result is the product of the cardinalities of relations in the FROM clause.

  **Reduction factor (RF)** associated with each term reflects the impact of the term in reducing result size. **Result cardinality** = Max # tuples * product of all RF’s.
  - Implicit assumption that terms are independent! And uniform distribution
  - Term \( \text{col} = \text{value} \) has RF \( 1/\text{NKeys(I)} \), given index I on \( \text{col} \)
  - Term \( \text{col1} = \text{col2} \) has RF \( 1/\text{MAX(NKeys(I1), NKeys(I2))} \)
  - Term \( \text{col} \rangle \text{value} \) has RF \( (\text{High(I)}-\text{value})/(\text{High(I)}-\text{Low(I)}) \)
Statistics maintained contd.
- Term Col BETWEEN value1 and value2
- Term Col IN (list of values)
- Term Col IN subquery
- Term (pred expression 1) OR (Pred expression 2)
  \[ F = F(\text{pred1}) + F(\text{pred2}) - F(\text{pred1}) \times F(\text{pred2}) \]
- (pred expression 1) AND (Pred expression 2)
  \[ F = ? \]
- NOT pred
  \[ F = ? \]

Relational Algebra Equivalences
- Allow us to choose different join orders and to `push` selections and projections ahead of joins.
- **Selections**: \( \sigma_{c_1 \land \cdots \land c_n}(R) \equiv \sigma_{c_1}(\cdots \sigma_{c_n}(R)) \) (Cascade)
  \( \sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R)) \) (Commute)
- **Projections**: \( \pi_{a_1}(R) \equiv \pi_{a_1}(\cdots(\pi_{a_n}(R))) \) (Cascade)
- **Joins**: \( R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \) (Associative)
  \( (R \bowtie S) \equiv (S \bowtie R) \) (Commute)

Show that: \( R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S \)
More Equivalences

- A projection commutes with a selection that only uses attributes retained by the projection.
- Selection between attributes of the two arguments of a cross-product converts cross-product to a join.
- A selection on just attributes of R commutes with $R \bowtie S$. (i.e., $\sigma (R \bowtie S) \equiv \sigma (R) \bowtie S$)
- Similarly, if a projection follows a join $R \bowtie S$, we can `push’ it by retaining only attributes of R (and S) that are needed for the join or are kept by the projection.