Query Processing: Goal

- Go through tables to find the right tuples
  - Efficiently
- Challenges
  - Selection
    - Use of indices
  - Projection
    - Duplicate elimination

- Cartesian Product
  - Ouch
  - $|R1| \times |R2|$ tuples...
- Join processing
  - Combining Cartesian product and selection can be much more efficient

- Set operations
  - Union, Intersection, Difference
Example

Select B,D
From R,S
Where R.A = “c” \( \land \) S.E = 2 \( \land \) R.C=S.C
How do we execute query?

One idea

- Do Cartesian product
- Select tuples
- Do projection

<table>
<thead>
<tr>
<th>RXS</th>
<th>R.A</th>
<th>R.B</th>
<th>R.C</th>
<th>S.C</th>
<th>S.D</th>
<th>S.E</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td>20</td>
<td>y</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Bingo! Got one...
Relational Algebra - can be used to describe plans...

**Ex: Plan I**

\[ \Pi_{B,D} \]

\[ \sigma_{R.A=\text{“c”} \land S.E=2 \land R.C=S.C} \]

\[ R \bowtie S \]

**OR: \[ \Pi_{B,D} [ \sigma_{R.A=\text{“c”} \land S.E=2 \land R.C=S.C} (R \times S)] \]

**Another idea:**

**Plan II**

\[ \sigma_{R.A = \text{“c”}} \]

\[ \sigma_{S.E = 2} \]

\[ R \bowtie S \]

natural join
Plan III

Use R.A and S.C Indexes

(1) Use R.A index to select R tuples with R.A = “c”
(2) For each R.C value found, use S.C index to find matching tuples
(3) Eliminate S tuples S.E ≠ 2
(4) Join matching R,S tuples, project B,D attributes and place in result
Chapter 15: Query Processing

- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions
Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation

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

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{\text{salary} < 75000} (\Pi_{\text{salary}} (\text{instructor}))$ is equivalent to $\Pi_{\text{salary}} (\sigma_{\text{salary} < 75000} (\text{instructor}))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an evaluation plan. E.g.,:
  - Use an index on salary to find instructors with salary $< 75000$, 
  - Or perform complete relation scan and discard instructors with salary $\geq 75000$

Basic Steps: Optimization (Cont.)

- **Query Optimization**: Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - E.g., number of tuples in each relation, size of tuples, etc.
- In this chapter we study
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression
- In Chapter 16
  - We study how to optimize queries, that is, how to find an evaluation plan with lowest estimated cost
Basic Steps in Query Processing

1. Parsing and translation – use standard compiler techniques (CS35200)
2. Optimization – choose from different ways of getting the same result
3. Evaluation – Today…

SQL query

\[ \downarrow \]

parse

\[ \downarrow \]

parse tree

convert

\[ \downarrow \]

logical query plan

apply laws

“improved” l.q.p

\[ \downarrow \]

estimate result sizes

l.q.p. + sizes

\[ \downarrow \]

consider physical plans

\{P_1,P_2,\ldots\}

\[ \uparrow \]

answer

\[ \uparrow \]

execute

\[ \uparrow \]

Pi

\{P_1,C_1),(P_2,C_2)\ldots\}

\[ \uparrow \]

pick best

\[ \uparrow \]

estimate costs
Example: SQL query

```
SELECT title
FROM StarsIn
WHERE starName IN (  
    SELECT name
    FROM MovieStar
    WHERE birthdate LIKE '1960'
  );
```

(Find the movies with stars born in 1960)
Example: Generating Relational Algebra

\[ \Pi_{\text{title}} \sigma_{\star \text{name}} \left( \text{StarsIn} \right) \text{IN } \Pi_{\text{name}} \sigma_{\text{birthdate LIKE '1960'}} \left( \text{starName} \right) \text{MovieStar} \]

Example: Logical Query Plan

\[ \Pi_{\text{title}} \sigma_{\star \text{name} = \text{name}} \times \left( \text{StarsIn} \times \Pi_{\text{name}} \right) \sigma_{\text{birthdate LIKE '1960'}} \text{MovieStar} \]
Example: Improved Logical Query Plan

$$\Pi_{\text{title}}$$
$$\times$$
$$\Pi_{\text{name}}$$
$$\sigma_{\text{birthdate LIKE '1960'}}$$
StarsIn

Question: Push project to StarsIn?

Example: Estimate Result Sizes

Need expected size
StarsIn
$$\Pi_{\text{name}}$$
$$\sigma$$
MovieStar
Example: One Physical Plan

Example: Estimate costs
Evaluation of Expressions

- Alternatives for evaluating an entire expression tree
  - **Materialization**: generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
  - **Pipelining**: pass on tuples to parent operations even as an operation is being executed
- We study above alternatives in more detail
Materialization

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

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

\[ \sigma_{\text{building} = "\text{Watson}"}(\text{department}) \]

then compute the store its join with instructor, and finally compute the projection on name.

![Diagram](image)

Materialization (Cont.)

- Materialized evaluation is always applicable

- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk, so
    - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk

- **Double buffering**: use two output buffers for each operation, when one is full write it to disk while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time
Pipelining

- **Pipelined evaluation**: evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don’t store result of $\sigma_{\text{building}=\text{Watson}}(\text{department})$
  - 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

Pipelining (Cont.)

- In demand driven or lazy evaluation
  - system repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain “state” so it knows what to return next
- In producer-driven or eager pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples
  - Alternative name: pull and push models of pipelining
Pipelining (Cont.)

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

Blocking Operations

- **Blocking operations**: cannot generate any output until all input is consumed
  - E.g., sorting, aggregation, …
- But can often consume inputs from a pipeline, or produce outputs to a pipeline
- Key idea: blocking operations often have two suboperations
  - E.g., for sort: run generation and merge
  - For hash join: partitioning and build-probe
- Treat them as separate operations

(a) Logical Query

(b) Pipelined Plan
Pipeline Stages

- **Pipeline stages:**
  - All operations in a stage run concurrently
  - A stage can start only after preceding stages have completed execution

(a) Logical Query

(b) Pipelined Plan

Pipelining for Continuous-Stream Data

- **Data streams**
  - Data entering database in a continuous manner
  - E.g., Sensor networks, user clicks, ...

- **Continuous queries**
  - Results get updated as streaming data enters the database
  - Aggregation on windows is often used
    - E.g., **tumbling windows** divide time into units, e.g., hours, minutes
  - Need to use pipelined processing algorithms
    - **Punctuations** used to infer when all data for a window has been received
Measures of Query Cost

- Many factors contribute to time cost
  - *disk access, CPU, and network communication*
- Cost can be measured based on
  - *response time*, i.e., total elapsed time for answering query, or
  - total *resource consumption*
- We use total resource consumption as cost metric
  - Response time harder to estimate, and minimizing resource consumption is a good idea in a shared database
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
  - Network costs must be considered for parallel systems
- We describe how estimate the cost of each operation
  - We do not include cost to writing output to disk

Disk cost can be estimated as:
- Number of seeks * average-seek-cost
- Number of blocks read * average-block-read-cost
- Number of blocks written * average-block-write-cost

For simplicity we just use the number of block transfers from disk and the number of seeks as the cost measures
- $t_T$ – time to transfer one block
  - Assuming for simplicity that write cost is same as read cost
- $t_S$ – time for one seek
- Cost for b block transfers plus S seeks
  \[ b \times t_T + S \times t_S \]

$t_S$ and $t_T$ depend on where data is stored; with 4 KB blocks:
- High end magnetic disk: $t_S = 4$ msec and $t_T = 0.1$ msec
- SSD: $t_S = 20-90$ microsec and $t_T = 2-10$ microsec for 4KB
Measures of Query Cost (Cont.)

- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation
- Several algorithms can reduce disk I/O by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
- Worst case estimates assume that no data is initially in buffer and only the minimum amount of memory needed for the operation is available
  - But more optimistic estimates are used in practice