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…
Example: SQL query

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

(Find the movies with stars born in 1960)
```
Example: Parse Tree

$$\begin{align*}
\text{SELECT} & \quad \text{FROM} \\
\text{title} & \quad \text{StarsIn} \\
\text{starName} & \quad \text{<SFW>} \\
\text{name} & \quad \text{MovieStar} \\
\text{birthdate} & \quad \text{LIKE} \\
\text{birthdate} & \quad \text{LIKE} \quad \text{\textquote{\textquote{'\textquote{1960}}}'} \\
\text{starName} & \quad \text{<SFW>} \\
\text{<SFW>} & \quad \text{IN} \\
\text{<Query>} & \quad \text{<Query>}
\end{align*}$$

Example: Generating Relational Algebra

$$\Pi_{\text{title}} \quad \sigma_{\text{birthdate LIKE \textquote{\textquote{'1960'}}}} \quad \text{<tuple>} \quad \text{IN} \quad \Pi_{\text{name}} \quad \text{<condition> \quad \text{<attribute>}}$$
Example: Logical Query Plan

```
Π title
  σ starName=name
   ⨉
   StarsIn π name
     σ birthdate LIKE '%1960'
     MovieStar
```

Example: Improved Logical Query Plan

```
Π title
  ⬇️
  starName=name
  ⨉
  StarsIn π name
    σ birthdate LIKE '%1960'
    MovieStar
```

Question: Push project to StarsIn?
Example: Estimate Result Sizes

```
<table>
<thead>
<tr>
<th>Need expected size</th>
</tr>
</thead>
<tbody>
<tr>
<td>StarsIn</td>
</tr>
<tr>
<td>σ</td>
</tr>
<tr>
<td>π</td>
</tr>
<tr>
<td>MovieStar</td>
</tr>
</tbody>
</table>
```

Example: One Physical Plan

```
<table>
<thead>
<tr>
<th>Hash join</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters: join order, memory size, project attributes,...</td>
</tr>
<tr>
<td>SEQ scan</td>
</tr>
<tr>
<td>index scan</td>
</tr>
<tr>
<td>Parameters: Select Condition,...</td>
</tr>
<tr>
<td>StarsIn</td>
</tr>
<tr>
<td>MovieStar</td>
</tr>
</tbody>
</table>
```
Example: Estimate costs

L.Q.P

- P1
  - C1
  - P2
  - C2
  - ...
  - Pn
  - Cn

Pick best!

SQL query
- Parse
  - Parse tree
- Convert
  - Logical query plan
- Apply laws
- "Improved" L.Q.P
- Estimate result sizes
- L.Q.P. + sizes
- Consider physical plans

{P1,P2,.....}

Answer
- Execute
- Pick best
- Estimate costs
- Statistics

{(P1,C1),(P2,C2).....}
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
  \[ \sigma_{building = “Watson”}(department) \]
  then compute the store its join with **instructor**, and finally compute the projection on **name**.
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
  \( \sigma_{\text{building} = “Watson”}(\text{department}) \)
  - 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

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

![Pipeline Stages](image)

### Pipeline Stages

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

- 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
Selection Operation

- **File scan**
  - Algorithm **A1** (linear search). Scan each file block and test all records to see whether they satisfy the selection condition.
    - Cost estimate = \( b_r \) block transfers + 1 seek
    - \( b_r \) denotes number of blocks containing records from relation \( r \)
    - If selection is on a key attribute, can stop on finding record
      - cost = \( (b_r/2) \) block transfers + 1 seek
    - Linear search can be applied regardless of
      - selection condition or
      - ordering of records in the file, or
      - availability of indices
  - Note: binary search generally does not make sense since data is not stored consecutively
    - except when there is an index available,
    - and binary search requires more seeks than index search

Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.

- **A2** (clustering index, equality on key). Retrieve a single record that satisfies the corresponding equality condition
  - Cost = \( (h_i + 1) \times (t_T + t_S) \)

- **A3** (clustering index, equality on nonkey) Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let \( b \) = number of blocks containing matching records
  - Cost = \( h_i \times (t_T + t_S) + t_S + t_T \times b \)
Selections Using Indices

- **A4** (secondary index, equality on key/non-key).
  - Retrieve a single record if the search-key is a candidate key
    - Cost = \((h_i + 1) \times (t_T + t_S)\)
  - Retrieve multiple records if search-key is not a candidate key
    - each of \(n\) matching records may be on a different block
    - Cost = \((h_i + n) \times (t_T + t_S)\)
    - Can be very expensive!

Selections Involving Comparisons

- Can implement selections of the form \(\sigma_{A < V}(r)\) or \(\sigma_{A \geq V}(r)\) by using
  - a linear file scan,
  - or by using indices in the following ways:
    - **A5** (clustering index, comparison). (Relation is sorted on A)
      - For \(\sigma_{A \geq V}(r)\) use index to find first tuple \(\geq V\) and scan relation sequentially from there
      - For \(\sigma_{A < V}(r)\) just scan relation sequentially till first tuple \(> V\); do not use index
    - **A6** (clustering index, comparison).
      - For \(\sigma_{A \geq V}(r)\) use index to find first index entry \(\geq V\) and scan index sequentially from there, to find pointers to records.
      - For \(\sigma_{A < V}(r)\) just scan leaf pages of index finding pointers to records, till first entry \(> V\)
      - In either case, retrieve records that are pointed to
      - requires an I/O per record; Linear file scan may be cheaper!
Implementation of Complex Selections

- **Conjunction:** $\sigma_{\theta_1 \land \theta_2 \land \ldots \land \theta_n}(r)$
  - A7 *(conjunctive selection using one index)*.
    - Select a combination of $\theta_i$ and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta_i}(r)$.
    - Test other conditions on tuple after fetching it into memory buffer.
  - A8 *(conjunctive selection using composite index)*.
    - Use appropriate composite (multiple-key) index if available.
  - A9 *(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 fetch records from file
    - If some conditions do not have appropriate indices, apply test in memory.

Algorithms for Complex Selections

- **Disjunction:** $\sigma_{\theta_1 \lor \theta_2 \lor \ldots \lor \theta_n}(r)$
  - A10 *(disjunctive selection by union of identifiers)*.
    - Applicable if all conditions have available indices.
      - Otherwise use linear scan.
    - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
    - Then fetch records from file
  - **Negation:** $\sigma_{\neg \theta}(r)$
    - Use linear scan on file
    - If very few records satisfy $\neg \theta$, and an index is applicable to $\theta$
      - Find satisfying records using index and fetch from file
Bitmap Index Scan

- The **bitmap index scan** algorithm of PostgreSQL
  - Bridges gap between secondary index scan and linear file scan when number of matching records is not known before execution
  - Bitmap with 1 bit per page in relation
  - Steps:
    - Index scan used to find record ids, and set bit of corresponding page in bitmap
    - Linear file scan fetching only pages with bit set to 1
  - Performance
    - Similar to index scan when only a few bits are set
    - Similar to linear file scan when most bits are set
    - Never behaves very badly compared to best alternative