Selection Size Estimation

- $\sigma_{A=\mathit{v}}(r)$
  - $n_r / V(A, r)$: number of records that will satisfy the selection
  - equality condition on a key attribute:
    - size estimate = 1

- $\sigma_{A\geq\mathit{v}}(r)$ (case of $\sigma_{A\leq\mathit{v}}(r)$ is symmetric)
  - Let $c$ denote the estimated number of tuples satisfying the condition.
  - If min($A, r$) and max($A, r$) are available in catalog
    - $c = 0$ if $v < \text{min}(A, r)$
    - $c = n_r$ if $v >= \text{max}(A, r)$
    - $c = \frac{n_r}{\text{max}(A, r) - \text{min}(A, r)}$ $v - \text{min}(A, r)$
  - If histograms available, can refine above estimate
  - In absence of statistical information $c$ is assumed to be $n_r / 2$. 
Size Estimation of Complex Selections

- **selectivity**($\theta_i$) = the probability that a tuple in $r$ satisfies $\theta_i$.

  - If $s_i$ is the number of satisfying tuples in $r$, then selectivity ($\theta_i$) = $s_i/n_r$.

- **conjunction:** $\sigma_{\theta_1 \land \theta_2 \land \ldots \land \theta_n}(r)$. **Assuming independence**, estimate of tuples in the result is:
  
  $$\frac{n_r \cdot s_1 \cdot s_2 \cdot \ldots \cdot s_n}{n_r^n}$$

- **disjunction:** $\sigma_{\theta_1 \lor \theta_2 \lor \ldots \lor \theta_n}(r)$. Estimated number of tuples:
  
  $$n_r \cdot \left( 1 - \left(1 - \frac{s_1}{n_r}\right) \cdot \left(1 - \frac{s_2}{n_r}\right) \cdot \ldots \cdot \left(1 - \frac{s_n}{n_r}\right) \right)$$

- **negation:** $\sigma_{\neg \theta}(r)$. Estimated number of tuples: $n_r - \text{size}(\sigma_{\neg \theta}(r))$

*we assume all predicates are independent*

Size Estimation Examples

- **Assume:**
  
  - $n_r = 1000$
  - $\theta_1$ ("balance > $1000"") = 0.9
  - $\theta_2$ ("age < 40") = 0.4
  - ("balance > $1000"") AND ("age < 40")
    
    $$= n_r \cdot \theta_1 \cdot \theta_2$$
    
    $$= 1000 \cdot 0.9 \cdot 0.4 = 360$$
  
  - ("balance > $1000"") OR ("age < 40")
    
    $$= n_r \cdot (1 - (1 - \theta_1) \cdot (1 - \theta_2))$$
    
    $$= 1000 \cdot (1 - 0.1 \cdot 0.6)$$
    
    $$= 1000 \cdot 0.94$$
    
    $$= 940$$

*we assume all predicates are independent*
Joins

- R join S: R.a = S.a
  - |R| = 10,000; |S| = 5000

- CASE 1: a is key for S
  - Worst case: each tuple of R joins with exactly one tuple of S
  - So: |R join S| = |R| = 10,000

- CASE 2: a is key for R
  - So: |R join S| = |S| = 5,000

- CASE 3: a is not a key for either
  - Reason with the distributions on a
  - Assume domain of a (the number of distinct values a can take) as follows:
    - V(A, R) = V(A, S) = 100
  - assuming uniformity
    - For each value of a:
      - We have 10,000/100 = 100 tuples of R with that value of a
      - We have 5000/100 = 50 tuples of S with that value of a
      - All of these will join with each other, and produce 100 * 50 = 5000
    - So total number of results in the join:
      - 5000 * 100 (distinct values) = 500,000
  - We can improve the accuracy if we know the distributions on a better
    - Histogram
Other Operations

- **Projection**: $\Pi_A(R)$
  - If no duplicate elimination, THEN $|\Pi_A(R)| = |R|$
  - If *distinct* used (duplicate elimination performed): $|\Pi_A(R)| = V(A, R)$

- **Set operations**: (heuristic upper bounds)
  - Union ALL: $|R \cup S| = |R| + |S|$
  - Intersect ALL: $|R \cap S| = \min\{|R|, |S|\}$
  - Except ALL: $|R - S| = |R|$
  - Union, Intersection, Except (with duplicate elimination)
    - Somewhat more complex reasoning based on the frequency distributions etc…

- Note that we are being conservative.
- And so on …

Query Optimization

- **Introduction**
- **Statistics Estimation**
- **Transformation of Relational Expressions**
- **Optimization Algorithms**
Equivalence of Expressions

- Two relational expressions equivalent iff:
  - Their result is identical on all legal databases
- Equivalence rules:
  - Allow replacing one expression with another
- Equivalence rules:
  1. \( \sigma_{\theta_1 \land \theta_2} (E) = \sigma_{\theta_1} (\sigma_{\theta_2} (E)) \)
  2. Selections are commutative
     \( \sigma_{\theta_1} (\sigma_{\theta_2} (E)) = \sigma_{\theta_2} (\sigma_{\theta_1} (E)) \)
  3. \( \Pi_{l_1} (\Pi_{l_2} (\ldots (\Pi_{l_n} (E))\ldots)) = \Pi_{l_1} (E) \)
  5. \( E_1 \bowtie_0 E_2 = E_2 \bowtie_0 E_1 \) \hspace{1cm} \text{[although attribute orders differ]}
  7(a). If \( \theta_0 \) only involves attributes from \( E_1 \)
      \( \sigma_{\theta_0} (E_1 \bowtie_0 E_2) = (\sigma_{\theta_0} (E_1)) \bowtie_0 E_2 \)
- And so on…
  - Many rules of this type
Example

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

\[ \Pi_{\text{customer name}} (\sigma_{\text{branch city} = \text{Brooklyn}} \land \text{balance} > 1000) \]

\[ (\text{branch} \times (\text{account} \times \text{depositor})) \]

- Apply the rules one by one

\[ \Pi_{\text{customer name}} (\sigma_{\text{branch city} = \text{Brooklyn}} \land \text{balance} > 1000) \]

\[ (\text{branch} \times (\text{account})) \times \text{depositor} \]

\[ \Pi_{\text{customer name}} ((\sigma_{\text{branch city} = \text{Brooklyn}} (\text{branch})) \times (\sigma_{\text{balance} > 1000} (\text{account})) \times \text{depositor}) \]
Example

Equivalence of Expressions

- The rules give us a way to enumerate all equivalent expressions
  - Note that the expressions don’t contain physical access methods, join methods etc…

- Simple Algorithm:
  - Start with the original expression
  - Apply all possible applicable rules to get a new set of expressions
  - Repeat with this new set of expressions
  - Till no new expressions are generated
Equivalence of Expressions

- Works, but is not feasible
- Consider a simple case:
  - \( R_1 \bowtie (R_2 \bowtie (R_3 \bowtie \ldots \bowtie R_n)) \ldots \)

- Just join commutativity and associativity will give us:
  - At least:
    - \( n^2 \times 2^n \)
  - At worst:
    - \( n! \times 2^n \)
  - Typically enumeration combined with the search process

Evaluation Plans

- We still need to choose the join methods etc..
  - Option 1: Choose for each operation separately
    - Usually okay, but sometimes the operators interact
    - Consider joining three relations on the same attribute:
      - \( R_1 \bowtie_a (R_2 \bowtie_a R_3) \)
      - Best option for R2 join R3 might be hash-join
        - But if \( R_1 \) is sorted on \( a \), then sort-merge join is preferable
          - Because it produces the result in sorted order by \( a \)
  - Also, pipelining or materialization
  - Such issues typically arise when doing the optimization
Query Optimization

- Introduction
- Example of a Simple Type of Query
- Transformation of Relational Expressions
- Statistics Estimation
- Optimization Algorithms

Optimization Algorithms

- Two types:
  - Exhaustive: attempt to find the best plan
  - Heuristic: simpler, but not guaranteed to find the optimal plan

- Consider a simple case
  - Join of the relations $R_1, \ldots, R_n$
  - No selections, no projections
  - Still very large plan space
Searching for the best plan

- **Option 1:**
  - Enumerate all equivalent expressions for the original query
    - Using the rules outlined earlier
  - Estimate cost for each and choose the lowest

- **Too expensive!**
  - Consider finding the best join-order for $r_1 \times r_2 \times \ldots r_n$.
  - $(2(n - 1))/(n - 1)!$ different join orders for above expression.
    - With $n = 7$, the number is 665280
    - With $n = 10$, the number is greater than 176 billion!

Searching for the best plan

- **Option 2:**
  - Dynamic programming
    - There is too much commonality between the plans
    - Also, costs are additive
      - Caveat: Sort orders (also called “interesting orders”)
  - Reduces costs to $O(3^n)$ or $O(2^n)$ in most cases
    - Interesting orders increase this a little bit
  - Considered acceptable
    - Typically $n < 10$.
  - Switch to heuristic if not acceptable
Left Deep Join Trees

- In left-deep join trees, the right-hand-side input for each join is a relation, not the result of an intermediate join
- Early systems only considered these types of plans
  - Easier to pipeline

![Left-deep join tree](image1)

Heuristic Optimization

- Dynamic programming is expensive
- Use heuristics to reduce the number of choices
- Typically rule-based:
  - 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).

Query Optimization

- Introduction
- Example of a Simple Type of Query
- Transformation of Relational Expressions
- Optimization Algorithms
- Statistics Estimation
- Summary
Query Optimization

- Integral component of query processing
  - Why?
- One of the most complex pieces of code in a database
- Active area of research
  - E.g. XML Query Optimization?
  - What if you don’t know anything about the statistics
  - Better statistics
  - Etc …

Transactions
Databases

- Data Models
  - Conceptual representation of the data
- Data Retrieval
  - How to ask questions of the database
  - How to answer those questions
- Data Storage
  - How/where to store data, how to access it
- Data Integrity
  - Manage crashes, concurrency
  - Manage semantic inconsistencies

Overview

- **Transaction**: A sequence of database actions enclosed within special tags
- Properties:
  - **Atomicity**: Entire transaction or nothing
  - **Consistency**: Transaction, executed completely, takes database from one consistent state to another
  - **Isolation**: Concurrent transactions appear to run in isolation
  - **Durability**: Effects of committed transactions are not lost
- Consistency: Programmer needs to guarantee this
  - DBMS can do a few things, e.g., enforce constraints on the data
- Rest: DBMS guarantees
How does this relate to queries that we discussed?

- Queries don't update data, so **durability** and **consistency** not relevant
- Would want **concurrency**
  - Consider a query computing balance at the end of the day
- Would want **isolation**
  - What if somebody makes a *transfer* while we are computing the balance
  - Typically not guaranteed for such long-running queries

- **TPC-C vs TPC-H**
  - data entry vs decision support

Assumptions and Goals

- **Assumptions:**
  - The system can crash at any time
  - Similarly, the power can go out at any point
    - Contents of the main memory won’t survive a crash, or power outage
  - BUT… **disks are durable. They might stop, but data is not lost.**
    - For now.
  - Disks only guarantee *atomic sector writes*, nothing more
  - Transactions are by themselves consistent

- **Goals:**
  - Guaranteed durability, atomicity
  - As much concurrency as possible, while not compromising isolation and/or consistency
    - Two transactions updating the same account balance… NO
    - Two transactions updating different account balances… YES
Transaction States

- **active** – initial state, while executing
- **partially committed** – after final statement
- **failed** – after discover that can not proceed
- **aborted** – after rolled back and DB restored
- **committed** – after successful completion

Next...

- **Concurrency control schemes**
  - A CC scheme is used to guarantee that concurrency does not lead to problems
  - For simplicity, we will ignore durability during this section
    - So no crashes
    - Though transactions may still abort

- **Schedules**

- **When is concurrency okay?**
  - Serial schedules
  - Serializability
A Schedule

Transactions:
T1: transfers $50 from A to B
T2: transfers 10% of A to B

Database constraint: A + B is constant (*checking+saving accts*).

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Effect: Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
<td>A</td>
<td>100</td>
</tr>
<tr>
<td>A = A - 50</td>
<td>tmp = A*0.1</td>
<td>B</td>
<td>50</td>
</tr>
<tr>
<td>write(A)</td>
<td>A = A - tmp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td>write(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B = B + 50</td>
<td>read(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td>B = B + tmp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each transaction obeys the constraint.
The schedule does too.

Schedules

- *A schedule* is simply a (possibly interleaved) execution sequence of transaction instructions.

- **Serial Schedule**: A schedule in which transactions appear one after the other
  - i.e., No interleaving

- Serial schedules satisfy isolation and consistency
  - Since each transaction by itself does not introduce inconsistency
### Another serial schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Effect: Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td></td>
<td>A  100</td>
<td>40</td>
</tr>
<tr>
<td>A = A - 50</td>
<td>tmp = A * 0.1</td>
<td>B  50</td>
<td>110</td>
</tr>
<tr>
<td>write(A)</td>
<td>A = A - tmp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td>write(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B = B + 50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td>Consistent?</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Constraint is satisfied.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Since each Xion is consistent, any serial schedule must be consistent.

### Another schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td></td>
</tr>
<tr>
<td>A = A - 50</td>
<td>tmp = A * 0.1</td>
</tr>
<tr>
<td>write(A)</td>
<td>A = A - tmp</td>
</tr>
<tr>
<td>read(B)</td>
<td>write(A)</td>
</tr>
<tr>
<td>B = B + 50</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td>Is this schedule okay?</td>
</tr>
</tbody>
</table>

Lets look at the final effect...

<table>
<thead>
<tr>
<th>Effect:</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
<td>45</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
<td>105</td>
</tr>
</tbody>
</table>

Consistent.

So this schedule is okay too.
Another schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>A = A -50</td>
<td>tmp = A*0.1</td>
</tr>
<tr>
<td>write(A)</td>
<td>A = A - tmp</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>B = B + 50</td>
<td>B = B + tmp</td>
</tr>
<tr>
<td>write(B)</td>
<td>write(B)</td>
</tr>
</tbody>
</table>

Is this schedule okay?

Let's look at the final effect...

<table>
<thead>
<tr>
<th>Effect: Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>50</td>
</tr>
</tbody>
</table>

Further, the effect same as the serial schedule 1.

Called **serializable**

Example Schedules (Cont.)

A “bad” schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>A = A -50</td>
<td>tmp = A*0.1</td>
</tr>
<tr>
<td>write(A)</td>
<td>A = A - tmp</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
</tr>
<tr>
<td></td>
<td>read(B)</td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
</tr>
<tr>
<td>B = B + 50</td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td>B = B + tmp</td>
</tr>
<tr>
<td></td>
<td>write(B)</td>
</tr>
</tbody>
</table>

Effect: Before After
A 100 50
B 50 60

Not consistent
Serializability

- A schedule is called *serializable* if:
  - *its final effect is the same as that of a serial schedule*

- Serializability $\rightarrow$ database remains consistent
  - Since serial schedules are fine

- Non-serializable schedules are unlikely to result in consistent databases

- We will ensure serializability
  - *Though typically relaxed in real high-throughput environments...*