Query Optimization

- Overview
- Statistics Estimation
- Transformation of Relational Expressions
- Optimization Algorithms

Estimating Output Sizes: Joins

- \( R \) JOIN \( S \): \( R.a = S.a \)
  - \( |R| = 10,000; |S| = 5000 \)

- CASE 3: \( a \) is not a key for either
  - Reason with the distributions on \( a \)
  - Say: the domain of \( a \): \( V(A, R) = V(A, S) = 100 \)
  - THEN, \textit{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 \( a \)
      - So each distinct value will produce \( 100 \times 50 = 5000 \) tuples
        - this assumes that the distinct values in \( R,S \) are the same
    - So total number of results in the join:
      - \( 5000 \) per distinctValue \(* 100 \) distinct values = \( 500,000 \) total tuples
  - We can improve the accuracy if we know the distributions on \( a \) better
    - Say using a histogram
Estimating Output Sizes: Other Ops

- 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: $|R \setminus S| = |R|$
  - Union, Intersection, Except (with duplicate elimination)
    - Somewhat more complex reasoning based on the frequency distributions etc…

- And so on …

Equivalence of Expressions

- Two relational expressions equivalent iff:
  - Their result is identical on all legal databases
- Equivalence rules (Section 13.2.1):
  - Allow replacing one expression with another
- Examples:
  1. $\sigma_{\theta_1 \land \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
  2. Selections are commutative
     $\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
Equivalence Rules

- **Examples:**
  3. $\Pi_{L_1} (\Pi_{L_2} (\ldots (\Pi_{L_n}(E))\ldots)) = \Pi_{L_1}(E)$
  5. $E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$

7(a). If $\theta$ only involves attributes from $E_1$:

$$\sigma_{\theta}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta}(E_1)) \bowtie_{\theta} E_2$$

- **And so on...**
  - Many rules of this type

---

**Pictorial Depiction**

- **Rule 5**: $E_1 \bowtie_{\theta} E_2$ \(\xRightarrow{\text{Rule 5}}\) $E_2 \bowtie_{\theta} E_1$
- **Rule 6a**: $E_1 \bowtie E_2 \bowtie E_3$ \(\xRightarrow{\text{Rule 6a}}\) $E_1 \bowtie E_2 \bowtie E_3$
- **Rule 7a**: $\sigma_{\theta}(E_1 \bowtie E_2)$ \(\xRightarrow{\text{Rule 7a}}\) $\sigma_{\theta}(E_1 \bowtie E_2)$

*Assuming projection on output*

*Natural joins associative*

*If $\theta$ only has attributes from $E_1$*
Example

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

\[ \Pi_{\text{customer}}(\sigma_{\text{branch} = \text{Brooklyn} \land \text{balance} > 1000}(\text{branch} \Join (\text{account \Join \text{depositor}))}) \]

- Apply the rules one by one

\[ \Pi_{\text{customer}}((\sigma_{\text{branch} = \text{Brooklyn} \land \text{balance} > 1000}(\text{branch} \Join (\text{account \Join \text{depositor})))) \]

inner joins associative

\[ \Pi_{\text{customer}}((\sigma_{\text{branch} = \text{Brooklyn} \land \text{balance} > 1000}(\text{branch} \Join (\text{account \Join \text{depositor})))) \]

first predicate on branch

second predicate on account
Equivalence of Expressions

- The rules allow enumeration of 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:
  - $R_1 \Join (R_2 \Join (R_3 \Join (\ldots \Join R_n))\ldots)$

- 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 a 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:
      - $R_1 \Join_{a} (R_2 \Join_{a} R_3)$
    - Best option for $R_2$ join $R_3$ 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
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 $R1, \ldots, Rn$
  - 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 $r_1 \bowtie r_2 \bowtie \ldots r_n$.
  - There are $(2(n - 1))!/(n - 1)!$ 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 much commonality between the plans
    - 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
  - only $O(n!)$ to search all options (this is much better than prior approaches)
  - $O(n2^n)$ if we use dynamic programming techniques

Heuristic Optimization

- Dynamic programming is still 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 system
- 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

Transaction Concept

- **A transaction** is a unit of program execution that accesses and possibly updates various data items.
- E.g. transaction to transfer $50 from account A to account B:

  ```
  begin
  read(A)
  A := A - 50
  write(A)
  read(B)
  B := B + 50
  write(B)
  end
  ```

- **Two main issues to deal with:**
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions
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

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:</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A = A -50</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B = B + 50</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>time</th>
<th>read(A)</th>
<th>tmp = A*0.1</th>
<th>A = A – tmp</th>
<th>write(A)</th>
<th>read(B)</th>
<th>B = B + tmp</th>
<th>write(B)</th>
</tr>
</thead>
</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>tmp = A*0.1</td>
<td>A</td>
<td>100</td>
</tr>
<tr>
<td>A = A – tmp</td>
<td></td>
<td>B</td>
<td>50</td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B = B+ tmp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Consistent?
Constraint is satisfied.

Since each Xion is consistent, any serial schedule is also consistent

Another schedule

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

Effect: Before | After |
A    100 | 45 |
B    50 | 105 |

Consistent.
So this schedule is okay too.
Another 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>read(A)</td>
<td>A 100</td>
<td>45</td>
</tr>
<tr>
<td>A = A -50</td>
<td>tmp = A*0.1</td>
<td>B 50</td>
<td>105</td>
</tr>
<tr>
<td>write(A)</td>
<td>A = A – tmp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td>read(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B=B+50</td>
<td>B = B+ tmp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(B)</td>
<td>write(B)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is this schedule okay?

Lets look at the final effect...

**Effect:**

<table>
<thead>
<tr>
<th></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>

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>
<th>Effect: Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
<td>A 100</td>
<td>50</td>
</tr>
<tr>
<td>A = A -50</td>
<td>tmp = A*0.1</td>
<td>B 50</td>
<td>60</td>
</tr>
<tr>
<td>write(A)</td>
<td>A = A – tmp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td>read(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td>write(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td>B = B+ tmp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B=B+50</td>
<td>write(B)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Not consistent**
Serializability

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

- **Serializability** → 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...*