Query Processing

Estimating Output Sizes: 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: \(|R \text{ JOIN } S| = |R| = 10,000\)

- **CASE 2: a is key for R**
  - Each S tuple can match with only a single R tuple.
  - So: \(|R \text{ JOIN } S| = |S| = 5,000\)

Equijoins simplify things.
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 $a$: $V(A, R) = 100$ (the number of distinct values $a$ can take)
  - THEN, *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 $100 \times 50 = 5000$
  - So total number of results in the join:
    - $5000 \times 100$ (distinct values) = 500,000
  - 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: $|R - S| = |R|$
  - Union, Intersection, Except (with duplicate elimination)
    - Somewhat more complex reasoning based on the frequency distributions etc…

- And so on …
Index-ANDing

Q5
3 Points
Consider a query with a conjunctive predicate:

\[
\text{select } * \text{ from } R \text{ where } a = 10 \text{ and } b = 20.
\]

- R occupies 1 million blocks on disk, and
- there are secondary indexes of height 4 on both R.a and R.b.
- Assume number of tuples in R with R.a = 10 is 1000, with R.b = 20 is 3000, and with both R.a = 10 & R.b = 20 is 200.

For all the indexes, assume the number of pointers in each leaf (to the actual records) is 500, and number of records of R per block is 100.

Q5.1
1 Point
How many blocks are transferred when using the index on R.a to fetch tuples matching R.a = 10, and then checking the condition in memory.

\[
1005
\]

EXPLANATION
4 for tree (including leaf), 1 for extra leaf (1000 ptrs needed, 500 per leaf), 1000 for tuple blocks
Index-ANDing

Q5
3 Points
Consider a query with a conjunctive predicate:

```
select * from R where a = 10 and b = 20.
```

- R occupies 1 million blocks on disk, and
- there are secondary indexes of height 4 on both R.a and R.b.
- Assume number of tuples in R with R.a = 10 is 1000, with R.b = 20 is 3000, and with both R.a = 10 & R.b = 20 is 200.

For all the indexes, assume the number of pointers in each leaf (to the actual records) is 500, and number of records of R per block is 100.

Q5.2
1 Point
The same as the above but using the index on R.b:

```
3009
```

EXPLANATION
4 for tree, 5 for extra leaves, 3000 for tuple blocks

Index-ANDing

Q5
3 Points
Consider a query with a conjunctive predicate:

```
select * from R where a = 10 and b = 20.
```

- R occupies 1 million blocks on disk, and
- there are secondary indexes of height 4 on both R.a and R.b.
- Assume number of tuples in R with R.a = 10 is 1000, with R.b = 20 is 3000, and with both R.a = 10 & R.b = 20 is 200.

For all the indexes, assume the number of pointers in each leaf (to the actual records) is 500, and number of records of R per block is 100.

Q5.3
1 Point
Same as above, but instead using "index-ending" to identify tuples w/o reading them, and then reading only those that match the entire predicate:

```
214
```

EXPLANATION
4 for tree, 1 extra leaf for a, 4 for tree, 5 extra leaves for b, 200 tuple blocks
Index-ORing?

**Q6**
0.2 Points
Consider a query with a disjunctive predicate:

```sql
select * from R where a = 10 OR b = 20
```

- R occupies 1 million blocks on disk
- secondary indexes of height 4 on both R.a and R.b
- 1000 tuples match R.a = 10, 1500 match R.b = 20, 2000 tuples match the entire predicate.

What are the total number of disk I/Os (i.e., # blocks transferred) required to identify all matching tuples?

For all the indexes, assume the number of pointers on the leaf level (to the actual records) is 500 per block, and number of records of R per block is 100.

**Q6.1**
0.1 Points
Specify the minimal number of disk I/Os (i.e., # blocks transferred) required to identify all matching tuples.

**EXPLANATION**
4 for tree and 1 extra leaf for a, 4 for tree and 2 extra leaves for b = 11

11

Index-ORing?

**Q6**
0.2 Points
Consider a query with a disjunctive predicate:

```sql
select * from R where a = 10 OR b = 20
```

- R occupies 1 million blocks on disk
- secondary indexes of height 4 on both R.a and R.b
- 1000 tuples match R.a = 10, 1500 match R.b = 20, 2000 tuples match the entire predicate.

What are the total number of disk I/Os (i.e., # blocks transferred) required to identify all matching tuples?

For all the indexes, assume the number of pointers on the leaf level (to the actual records) is 500 per block, and number of records of R per block is 100.

**Q6.2**
0.1 Points
Using this approach (OR-ing the ptr sets), what would then be the total number of disk accesses to identify and load all matching tuples?

2011

**EXPLANATION**
Just the above, plus 2000 blocks for the matching tuples = 2011
Query Optimization

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

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. $\sigma_{\theta_1,\theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$

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 $\theta_0$ only involves attributes from $E_1$:

$$\sigma_{\theta_0}(E_1 \bowtie_0 E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$

- And so on…
  - Many rules of this type

Pictorial Depiction

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.

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

- Apply the rules one by one

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

Example

(a) Initial expression tree

(b) Tree after multiple transformations
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:
  - $R1 \Join (R2 \Join (R3 \Join (\ldots \Join Rn)))\ldots$
- Just join commutativity and associativity will give us:
  - At least:
    - $n^2 \cdot 2^n$
  - At worst:
    - $n! \cdot 2^n$
- 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:
      - \( R1 \bowtie_a (R2 \bowtie_a R3) \)
    - Best option for \( R2 \) join \( R3 \) might be hash-join
      - But if \( R1 \) 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
- Optimization Algorithms
- Statistics Estimation
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 $r_1 \Join r_2 \Join \ldots \Join r_n$
  - 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 $O(n3^n)$ or $O(n2^n)$ 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

![Join Trees](image)

(a) Left-deep join tree  
(b) Non-left-deep join tree

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 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