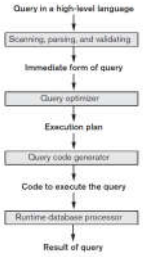
Query Processing and Query Optimization

CO527 Advanced Database Systems

Query Processing



Translating SQL queries into Relational Algebra

- SQL query is first translated into an equivalent extended relational algebra expression
- It represented as a query tree data structure (or query graph)
- Then query is optimized by choosing suitable query execution strategy
- SQL queries are decomposed into **query blocks**
- A query block contains a single SELECT-FROM-WHERE clause
- Therefore, nested queries should be identified as separate blocks.

Query Tree

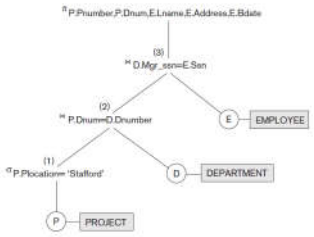
- Query tree includes the relational algebra operations being executed and is used as a possible data structure for the internal representation of the query in an RDBMS.

Example of a query tree

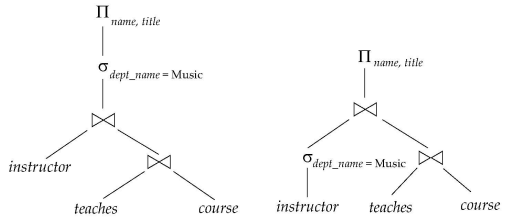
- *For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.*

$$\pi_{Pnumber,\ Dnum,\ Lname,\ Address,\ Bdate}(((\sigma_{Plocation='Stafford'}(PROJECT)) \bowtie_{Dnum=Dnumber}(DEPARTMENT)) \bowtie_{Mgr\_ssn=Ssn}(EMPLOYEE))$$

Example of a query tree

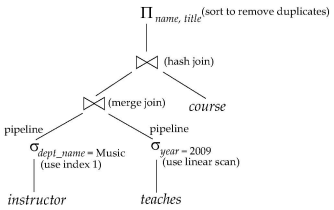


Alternative ways to evaluate a query



Execution plan

- Execution plan defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.



General Approaches to Optimization

- **Heuristics in query optimization**
  - Given a query perform selection and projections as early as possible.
  - Eliminate duplicate computations.
- **Cost-based query optimization**
  - Estimate the cost of different equivalent query expressions and choose the execution plan with the lowers cost estimation.

Using Heuristics in Query Optimization

- **Process for heuristics optimization**
  1. The parser of a high-level query generates an *initial internal representation*;
  2. Apply heuristics rules to optimize the internal representation.
  3. A query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.
- The **main heuristic** is to apply first the operations that reduce the size of intermediate results.
  - E.g., Apply SELECT and PROJECT operations before applying the JOIN or other binary operations.

Using Heuristics in Query Optimization

- **Query tree**: a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as *leaf nodes* of the tree, and represents the relational algebra operations as *internal nodes*.
- An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation.
- **Query graph**: a graph data structure that corresponds to a relational calculus expression. It does **not** indicate an order on which operations to perform first. There is only a **single** graph corresponding to each query.

Using Heuristics in Query Optimization

- **Example:**

  For every project located in 'Stafford', retrieve the project number, the controlling department number and the department manager's last name, address and birthdate.

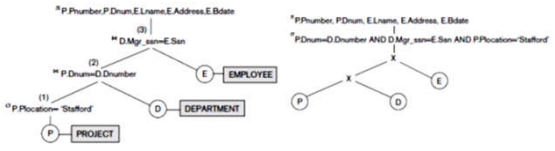
  Relation algebra:

  $\pi_{PNUMBER,\ DNUM,\ LNAME,\ ADDRESS,\ BDATE}\ (((\sigma_{PLOCATION='STAFFORD'}(PROJECT)) \bowtie_{DNUM=DNUMBER} (DEPARTMENT)) \bowtie_{MGRSSN=SSN} (EMPLOYEE))$

  SQL query:

```
Q2: SELECT  P.NUMBER,P.DNUM,E.LNAME, E.ADDRESS, E.BDATE
      FROM    PROJECT AS P,DEPARTMENT AS D, EMPLOYEE AS E
      WHERE  P.DNUM=D.DNUMBER AND D.MGRSSN=E.SSN AND
                P.PLOCATION='STAFFORD';
```

Query Trees



Using Heuristics in Query Optimization

Heuristic Optimization of Query Trees:

- The same query could correspond to many different relational algebra expressions — and hence many different query trees.
- The task of heuristic optimization of query trees is to find a **final query tree** that is efficient to execute.
- **Example**:

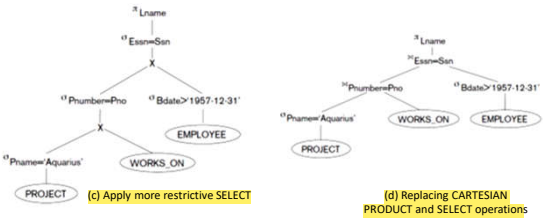
Q: SELECT LNAME
FROM EMPLOYEE, WORKS_ON, PROJECT
WHERE PNAME = 'AQUARIUS' AND PNMUBER=PNO
AND ESSN=SSN AND BDATE > '1957-12-31';

Using Heuristics in Query Optimization

(a) Initial query tree

(b) Moving SELECT operations down the query tree

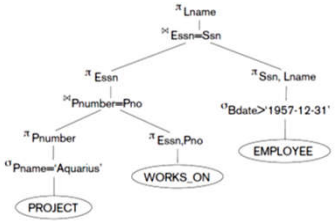
Using Heuristics in Query Optimization



(c) Apply more restrictive SELECT

(d) Replacing CARTESIAN PRODUCT and SELECT operations

Using Heuristics in Query Optimization



(e) Moving PROJECT operations down the query tree

Using Heuristics in Query Optimization

General Transformation Rules for Relational Algebra Operations:

1. Cascade of σ : A conjunctive selection condition can be broken up into a cascade (sequence) of individual s operations:

   $\sigma_{c_1 AND c_2 \ AND \ \ldots \ AND \ c_n}(\mathrm{R}) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(\mathrm{R}))\ldots))$

2. Commutativity of σ : The σ operation is commutative

   $\sigma_{c_1}(\sigma_{c_2}(\mathrm{R})) \equiv \sigma_{c_2}(\sigma_{c_1}(\mathrm{R}))$

3. Cascade of π : In a cascade (sequence) of π operations, all but the last one can be ignored:

   $\pi_{List_1}(\pi_{List_2}(\ldots(\pi_{List_n}(\mathrm{R}))\ldots)) \equiv \pi_{List_1}(\mathrm{R})$

4. Commuting σ with π : If the selection condition c involves only the attributes A1, ..., An in the projection list, the two operations can be commuted:

   $\pi_{A_1,A_2,\ldots,A_n}(\sigma_c(\mathrm{R})) \equiv \sigma_c(\pi_{A_1,A_2,\ldots,A_n}(\mathrm{R}))$

Using Heuristics in Query Optimization

General Transformation Rules for Relational Algebra Operations (cont.)

5. Commutativity of ⋈ (and x): The ⋈ operation is commutative as is the x operation:

   $R \bowtie_C S \equiv S \bowtie_C R$; $R \times S \equiv S \times R$

6. Commuting σ with ⋈ (or x): If all the attributes in the selection condition c involve only the attributes of one of the relations being joined—say, R—the two operations can be commuted as follows :

   $\sigma_c (R \bowtie S) \equiv (\sigma_c (R)) \bowtie S$

   Alternatively, if the selection condition c can be written as (c_1 and c_2), where condition c_1 involves only the attributes of R and condition c_2 involves only the attributes of S, the operations commute as follows:

   $\sigma_c (R \bowtie S) \equiv (\sigma_{c_1}(R)) \bowtie (\sigma_{c_2}(S))$

Using Heuristics in Query Optimization

General Transformation Rules for Relational Algebra Operations (cont.):

7. Commuting π with ⋈ (or x): Suppose that the projection list is L = {A_1, ..., A_n, B_1, ..., B_m}, where A_1, ..., A_n are attributes of R and B_1, ..., B_m are attributes of S. If the join condition c involves only attributes in L, the two operations can be commuted as follows:

   $\pi_L (R \bowtie_C S) \equiv (\pi_{A_1,...,A_n}(R)) \bowtie_C (\pi_{B_1,...,B_n}(S))$

   If the join condition c contains additional attributes not in L, these must be added to the projection list, and a final π operation is needed.

Using Heuristics in Query Optimization

General Transformation Rules for Relational Algebra Operations (cont.):

8. Commutativity of set operations: The set operations ∪ and ∩ are commutative but − is not.
9. Associativity of ⋈, ×, ∪, and ∩ : These four operations are individually associative; that is, if θ stands for any one of these four operations (throughout the expression), we have

   $(R \theta S) \theta T \equiv R \theta (S \theta T)$

10. Commuting s with set operations: The s operation commutes with ∪ , ∩ , and –. If q stands for any one of these three operations, we have

    $\sigma_c (R \theta S) \equiv (\sigma_c (R)) \theta (\sigma_c (S))$

Using Heuristics in Query Optimization

General Transformation Rules for Relational Algebra Operations (cont.):

11. The π operation commutes with ∪.

    $\pi_L (R \cup S) = (\pi_L (R)) \cup (\pi_L (S))$

12. Converting a (σ, ×) sequence into ⋈: If the condition c of a σ that follows a × Corresponds to a join condition, convert the (σ, ×) sequence into a ⋈ as follows:

    $(\sigma_C (R \times S)) = (R \bowtie_C S)$

13. Other transformations

Using Heuristics in Query Optimization

Outline of a Heuristic Algebraic Optimization Algorithm:

1. Using rule 1, break up any select operations with conjunctive conditions into a cascade of select operations.
2. Using rules 2, 4, 6, and 10 concerning the commutativity of select with other operations, move each select operation as far down the query tree as is permitted by the attributes involved in the select condition.
3. Using rule 9 concerning associativity of binary operations, rearrange the leaf nodes of the tree so that the leaf node relations with the most restrictive select operations are executed first in the query tree representation.
4. Using Rule 12, combine a Cartesian product operation with a subsequent select operation in the tree into a join operation.

Using Heuristics in Query Optimization

5. Using rules 3, 4, 7, and 11 concerning the cascading of project and the commuting of project with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new project operations as needed.
6. Identify subtrees that represent groups of operations that can be executed by a single algorithm.

Using Heuristics in Query Optimization

Summary of Heuristics for Algebraic Optimization:

1. The main heuristic is to apply first the **operations that reduce the size of intermediate results**.
2. Perform **select operations as early as possible** to reduce the number of tuples and perform **project operations as early as possible** to reduce the number of attributes. (This is done by moving select and project operations as far down the tree as possible.)
3. The **select and join operations that are most restrictive should be executed before other similar operations**. (This is done by reordering the leaf nodes of the tree among themselves and adjusting the rest of the tree appropriately.)

Using Heuristics in Query Optimization

Query Execution Plans

- An execution plan for a relational algebra query consists of a combination of the **relational algebra query tree** and information about the **access methods** to be used for each relation as well as the methods to be used in computing the relational operators stored in the tree.
- **Materialized evaluation:** the result of an operation is stored as a temporary relation.
- **Pipelined evaluation:** as the result of an operator is produced, it is forwarded to the next operator in sequence.

Using Selectivity and Cost Estimates in Query Optimization

Cost-based query optimization

- **Cost-based query optimization:** Estimate and compare the costs of executing a query using different execution strategies and choose the strategy with the lowest cost estimate.

  (Compare to heuristic query optimization)

- **Issues**
  - **Cost function**
  - **Number of execution strategies to be considered**

Cost components for query execution

1. Access cost to secondary storage
2. Disk storage cost
3. Computation cost
4. Memory usage cost
5. Communication cost

Note: Different database systems may focus on different cost components.

Catalog Information Used in Cost Functions

- Information about the size of a file
  - **number of records (tuples) (r),**
  - **record size (R),**
  - **number of blocks (b)**
  - **blocking factor (bfr)**
- Information about indexes and indexing attributes of a file
  - **Number of levels (x)** of each multilevel index
  - **Number of first-level index blocks (b_{I1})**
  - **Number of distinct values (d)** of an attribute
  - **Selectivity (sl)** of an attribute
  - **Selection cardinality (s)** of an attribute. (s = sl * r)

Physical Database Design and Tuning

Factors that Influence Physical Database Design

A. Analyzing the database queries and transactions
   - Queries and transactions expected to run on the database

B. Analyzing the expected frequency of invocation of queries and transactions
   - Expected frequency of use for all queries and transactions.
   - 80-20% rule: 80% of the processing happens with 20% of queries and transactions.

C. Analyzing the time constraints of queries and transactions
   - Any timing constraints. E.g. Transaction should terminate within 5 seconds on 95% of the time. It should never take more than 20 seconds.

D. Analyzing the expected frequencies of update operations
   - Minimum number of access paths (e.g. indexes) should be specified if a file is frequently updated.

E. Analyzing the uniqueness constraints on attributes.
   - Access paths should be specified on all candidate key attributes or set of attributes that are either the primary key of a file or unique attributes.

Physical Database Design Decisions

1. **Design decisions about indexing**
   - Whether to index an attribute?
   - What attribute or attributes to index on?
   - Whether to set up a clustered index?
   - Whether to use a hash index over a tree index?
   - Whether to use dynamic hashing for the file?
2. **Denormalization as a design decision for speeding up queries**

Database tuning

- The process of continuing to revise/adjust the physical database design by monitoring resource utilization as well as internal DBMS processing to reveal bottlenecks such as contention for the same data or devices.

- Goals
  - To make application run faster
  - To lower the response time of queries/transactions
  - To improve the overall throughput of transactions

Inputs to the tuning process

- **Statistics internally collected in DBMSs**
  - Size of individual tables
  - Number of distinct values in a column
  - The number of times a particular query or transaction is submitted/executed in an interval of time
  - The times required for different phases of query and transaction processing
- **Statistics obtained from monitoring**
  - Storage statistics
  - I/O and device performance statistics
  - Query/transaction processing statistics
  - Locking/logging related statistics
  - Index statistics

Problems to be considered in tuning

- How to avoid excessive lock contention?
- How to minimize overhead of logging and unnecessary dumping of data?
- How to optimize buffer size and scheduling of processes?
- How to allocate resources such as disks, RAM and processes for most efficient utilization?

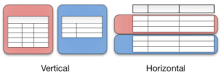
Tuning Indexes

- **Reasons to tuning indexes**
  - Certain queries may take too long to run for lack of an index;
  - Certain indexes may not get utilized at all;
  - Certain indexes may be causing excessive overhead because the index is on an attribute that undergoes frequent changes
- **Options to tuning indexes**
  - Drop or/and build new indexes
  - Change a non-clustered index to a clustered index (and vice versa)
  - Rebuilding the index

Tuning the Database Design

- **Dynamically changed processing requirements** need to be addressed by making changes to the conceptual schema if necessary and to reflect those changes into the logical schema and physical design.
- **Possible changes to the database design**
  - Existing tables may be joined (denormalized) because certain attributes from two or more tables are frequently needed together. This reduced the normalization level from BCNF to 3NF, 2NF, or 1NF.
  - For the given set of tables, there may be alternative design choices, all of which achieve 3NF or BCNF. One may be replaced by the other.

Tuning the Database Design



- **Possible changes to the database design**
  - A relation of the form R(K, A, B, C, D, …) that is in BCNF can be stored into multiple tables that are also in BCNF by replicating the key K in each table. This is called **vertical partitioning**.
    - E.g. EMPLOYEE(SSN, Name, Phone, Grade, Salary) may be split into EMP1(SSN, Name, Phone) and EMP2(SSN, Grade, Salary)
  - Attribute(s) from one table may be repeated in another even though this creates redundancy and potential anomalies.
    - E.g. Part_name may appear wherever the Part # appears. However, you may have a one master file such as PART_MASTER(Part#, Part_name, …)
  - **Horizontal partitioning** takes horizontal slices of a table and stores them as distinct tables.
    - E.g. product sales data may be separated in to different product lines. Each table has same attributes but contains a distinct set of products.

Tuning Queries

- **Indications for tuning queries**
  - A query issues too many disk accesses
  - The query plan shows that relevant indexes are not being used.
- **Typical instances for query tuning**
  1. Many query optimizers do not use indexes in the presence of arithmetic expressions (e.g. Salary/365 > 10.50), numerical comparisons of attributes of different sizes and precision (e.g. comparing INTEGER with SMALLINTEGER type attributes), NULL comparisons (e.g. Bdate IS NULL), and sub-string comparisons (e.g. LIKE '%mann').
  2. Indexes are often not used for nested queries using IN;

Tuning Queries (cont.)

- **Typical instances for query tuning (cont.)**
  3. Some *DISTINCT*s may be redundant and can be avoided without changing the result.
  4. Unnecessary use of temporary result tables can be avoided by collapsing multiple queries into a single query unless the temporary relation is needed for some intermediate processing.
  5. In some situations involving using of correlated queries, temporaries are useful.

Tuning Queries (cont.)

- **Typical instances for query tuning (cont.)**
  6. If multiple options for join condition are possible, choose one that uses a clustering index and avoid those that contain string comparisons.
     - E.g. Use EMPLOYEE.SSN = STUDENT.SSN join condition compared to EMPLOYEE.Name=STUDENT.Name
  7. The order of tables in the *FROM* clause may affect the join processing.
  8. Some query optimizers perform worse on nested queries compared to their equivalent un-nested counterparts.
  9. Many applications are based on views that define the data of interest to those applications. Sometimes these views become an overkill.

Additional Query Tuning Guidelines

1. A query with multiple selection conditions that are connected via *OR* may not be prompting the query optimizer to use any index. Such a query may be split up and expressed as a union of queries, each with a condition on an attribute that causes an index to be used.

```
SELECT Fname, Lname, Salary, Age
FROM EMPLOYEE
WHERE Age>45 OR Salary<50000;
```

```
SELECT Fname, Lname, Salary, Age
FROM EMPLOYEE
WHERE Age>45
UNION
SELECT Fname, Lname, Salary, Age
FROM EMPLOYEE
WHERE Salary<50000;
```

Additional Query Tuning Guidelines

2. Apply the following transformations
   - *NOT* condition may be transformed into a positive expression.
   - Embedded *SELECT* blocks (e.g. IN, ALL, SOME) may be replaced by joins.
   - If an equality join is set up between two tables, the range predicate on the joining attribute set up in one table may be repeated for the other table
3. *WHERE* conditions may be rewritten to utilize the indexes on multiple columns.

Index Region# (a) vs composite index on (Region#, Prod_type) (b)

```
(a) SELECT Region#, Prod_type, Month, Sales
    FROM SALES_STATISTICS
WHERE Region# = 3 AND ((Prod_type BETWEEN 1 AND 3) OR (Prod_type BETWEEN 8 AND 10));

(b) SELECT Region#, Prod_type, Month, Sales
    FROM SALES_STATISTICS
WHERE (Region# = 3 AND (Prod_type BETWEEN 1 AND 3)) OR (Region# = 3 AND (Prod_type BETWEEN 8 AND 10));
```

Query optimization in MySQL

- The EXPLAIN statement in MySQL provides information about how MySQL executes statements.
- When EXPLAIN is used with a statement, MySQL displays information from the optimizer about the statement execution plan.
- MySQL Workbench provides a graphical representation for EXPLAIN (called Visual Explain).

Explain Output

- **id** the sequential number of the table(s)
- **select_type** the type of SELECT
- **table** the name of the table or alias
- **type** the type of join for the query
- **possible_keys** which indexes MySQL could use
- **keys** which indexes MySQL will use
- **key_len** the length of keys used
- **ref** any columns used with the key to retrieve results
- **rows** estimated number of rows returned
- **extra** any additional information

Explain tells you

- In which order the tables are read
- What types of read operations that are made
- Which indexes could have been used
- Which indexes are used
- How the tables refer to each other
- How many rows the optimizer estimates to retrieve from each table

Visual Explain / Execution Plan in MySQL Workbench