

Advanced Database Management System (CoSc3052)

Query Processing and Optimization

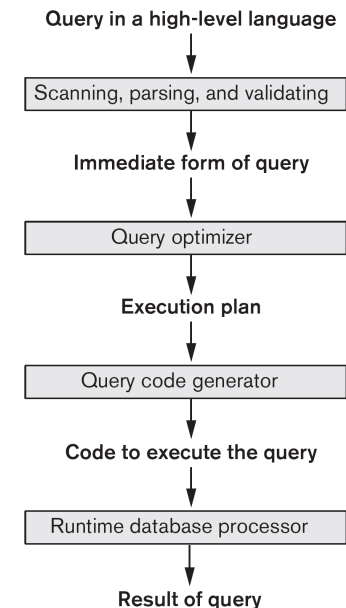
- **Scanner:** The scanner identifies the language tokens such as SQL Keywords, attribute names, and relation names in the text of the query.
- **Parser:** The parser checks the query syntax to determine whether it is formulated according to the syntax rules of the query language.
- **Validation:** The query must be validated by checking that all attributes and relation names are valid and semantically meaningful names in the schema of the particular database being queried.

Introduction to Query Processing

- Query Processing
 - The process by which the query results are retrieved from a high-level query such as SQL or OQL.
- Query Optimization:
 - The process of choosing a suitable execution strategy for processing a query
- Two internal representations of a query:
 - Query Tree
 - Query Graph

Query Processing ...

Introduction to Query Processing



Code can be:

Executed directly (interpreted mode)
Stored and executed later whenever needed (compiled mode)

Query Processing ...

- **Query Optimization:** the process of choosing a suitable execution strategy for processing a query.
- **Query Code Generator:** It generates the code to execute the plan.
- **Runtime Database Processor:** It has the task of running the query either in compiled or interpreted mode
 - If a runtime error results, an error message is generated by the runtime database processor.

Query Processing ...

- Chosen plan is just reasonably optimal
 - Finding the optimal
 - Too time consuming
 - May require detailed information
 - =>Planning a **good execution strategy** seems accurate than query optimization
- RDBMSs must systematically evaluate alternative query execution strategies to choose **reasonably efficient strategy**

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- Relational algebra is a basic set of operations for the relational model
- These operations enable a user to specify **basic retrieval requests**
- The result of an operation is a *new relation*, which may have been formed from one or more *input relations*
 - This property makes the algebra “closed” (all objects in relational algebra are relations)

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- Why Relational Algebra?
 - Provides a formal foundation for relational model operations
 - Used as a basis for implementing and optimizing queries in the query processing and optimization modules
 - Some of its concepts are incorporated into the SQL
 - Core operations and functions in the internal modules of most relational systems are based on relational algebra operations

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- Relational Algebra consists of several groups of operations
 - Unary Relational Operations
 - SELECT (symbol: σ (sigma))
 - PROJECT (symbol: π (pi))
 - RENAME (symbol: ρ (rho))
 - Relational Algebra Operations From Set Theory
 - UNION (\cup), INTERSECTION (\cap), DIFFERENCE (MINUS, $-$)
 - CARTESIAN PRODUCT (\times)

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- Relational Algebra consists of several groups of operations
 - Binary Relational Operations
 - JOIN (several variations of JOIN exist)
 - DIVISION
 - Additional Relational Operations
 - OUTER JOINS, OUTER UNION
 - AGGREGATE FUNCTIONS (Compute summary of information: SUM, COUNT, AVG, MIN, MAX)

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- The SELECT operation (σ (sigma))
 - used to select a *subset* of the tuples from a relation based on a **selection condition**.
 - The selection condition acts as a **filter**
 - Tuples satisfying the condition are *selected* whereas the other tuples are discarded (*filtered out*)
 - horizontal partition of the relation into two sets of tuples

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- Examples:
 - Select the EMPLOYEE tuples whose department number is 4:
 - $\sigma_{DNO = 4}$ (EMPLOYEE)
 - Select the employee tuples whose salary is greater than \$30,000:
 - $\sigma_{SALARY > 30,000}$ (EMPLOYEE)

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- In general, the *select* operation is denoted by $\sigma_{\langle \text{selection condition} \rangle}(R)$ where
 - the symbol σ (sigma) is used to denote the *select* operator
 - the selection condition is a Boolean (conditional) expression specified on the attributes of relation R
 - tuples that make the condition **true** are selected
 - appear in the result of the operation
 - tuples that make the condition **false** are filtered out
 - discarded from the result of the operation

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- SELECT Operation Properties
 - Unary
 - applied to each tuple individually
 - selection conditions cannot involve more than one tuple
 - The SELECT operation $\sigma_{\langle \text{selection condition} \rangle}(R)$ produces a relation S that has the same schema (same attributes) as R
 - Degree of the resulting relation is the same as degree of R
 - SELECT is commutative:
 - $\sigma_{\langle \text{condition1} \rangle}(\sigma_{\langle \text{condition2} \rangle}(R)) = \sigma_{\langle \text{condition2} \rangle}(\sigma_{\langle \text{condition1} \rangle}(R))$

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- SELECT Operation Properties
 - Because of commutativity property, a cascade (sequence) of SELECT operations may be applied in any order:
 - $\sigma_{\langle \text{cond1} \rangle}(\sigma_{\langle \text{cond2} \rangle}(\sigma_{\langle \text{cond3} \rangle}(R))) = \sigma_{\langle \text{cond2} \rangle}(\sigma_{\langle \text{cond3} \rangle}(\sigma_{\langle \text{cond1} \rangle}(R)))$
 - A cascade of SELECT operations may be replaced by a single selection with a conjunction of all the conditions:
 - $\sigma_{\langle \text{cond1} \rangle}(\sigma_{\langle \text{cond2} \rangle}(\sigma_{\langle \text{cond3} \rangle}(R))) = \sigma_{\langle \text{cond1} \rangle \text{ AND } \langle \text{cond2} \rangle \text{ AND } \langle \text{cond3} \rangle}(R)$
 - $|\sigma_c(R)| \leq |R|$
 - Fraction of tuples selected: **selectivity** of the condition

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- PROJECT Operation is denoted by π (pi)
- This operation keeps certain *columns* (attributes) from a relation and discards the other columns.
 - PROJECT creates a vertical partitioning
 - The list of specified columns (attributes) is kept in each tuple
 - The other attributes in each tuple are discarded
- Example: To list each employee's first and last name and salary, the following is used:
 - $\pi_{\text{LNAME, FNAME, SALARY}}(\text{EMPLOYEE})$

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- General form of the *project* operation: $\pi_{\langle \text{attribute list} \rangle}(R)$
 - π (pi): the symbol used to represent the *project* operation
 - $\langle \text{attribute list} \rangle$: desired list of attributes from relation R
 - degree is equal to the number of attributes in $\langle \text{attribute list} \rangle$
 - Removes any duplicate tuples
 - This is because the result of the *project* operation must be a *set of tuples*
 - Mathematical sets *do not allow* duplicate elements.

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- PROJECT Operation Properties
 - The number of tuples in the result of projection $\pi_{\langle \text{list} \rangle}(R)$ is always less or equal to the number of tuples in R
 - If the list of attributes includes a *key* of R, then the number of tuples in the result of PROJECT is *equal* to the number of tuples in R
 - PROJECT is *not* commutative
 - $\pi_{\langle \text{list1} \rangle}(\pi_{\langle \text{list2} \rangle}(R)) = \pi_{\langle \text{list1} \rangle}(R)$ as long as $\langle \text{list2} \rangle$ contains the attributes in $\langle \text{list1} \rangle$

Translating SQL Queries into Relational Algebra

Relational Algebra Overview: PROJECT Example

$\pi_{\text{Lname, Fname, Salary}}(\text{EMPLOYEE})$ $\pi_{\text{Sex, Salary}}(\text{EMPLOYEE})$

Lname	Fname	Salary
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

Sex	Salary
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- Sequences of Operations and RENAME
 - for most queries, we need to apply several relational algebra operations one after the other
 - write them as a single relational algebra expression by nesting
 - $\pi_{\text{Fname, Lname, Salary}}(\sigma_{\text{Dno}=5}(\text{EMPLOYEE}))$
 - apply one operation at a time and create intermediate result relations
 - $\text{DEP5_EMPS} \leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE})$
 - $\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Salary}}(\text{DEP5_EMPS})$

Translating SQL Queries into Relational Algebra

Relational Algebra Overview: \cup , \cap and $-$ Example

STUDENT		INSTRUCTOR		ST - IN		ST \cup IN	
Fn	Ln	Fname	Lname	Fn	Ln	Fn	Ln
Susan	Yao	John	Smith	Johnny	Kohler	Susan	Yao
Ramesh	Shah	Ricardo	Browne	Barbara	Jones	Ramesh	Shah
Johnny	Kohler	Susan	Yao	Amy	Ford	Johnny	Kohler
Barbara	Jones	Francis	Johnson	Jimmy	Wang	Barbara	Jones
Amy	Ford	Ramesh	Shah	Ernest	Gilbert	Amy	Ford
Jimmy	Wang					Jimmy	Wang
Ernest	Gilbert					Ernest	Gilbert

IN \cap ST		IN - ST	
Fn	Ln	Fname	Lname
Susan	Yao	John	Smith
Ramesh	Shah	Ricardo	Browne
		Francis	Johnson

ADBMS: Query Processing and Optimization

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- CARTESIAN PRODUCT
 - CROSS PRODUCT or CROSS JOIN
 - Denoted by \times
 - Binary set operation
 - Relations do not have to be union compatible
 - Useful when followed by a selection that matches values of attributes

ADBMS: Query Processing and Optimization

26

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- CARTESIAN PRODUCT: Example - Retrieve a list of names of each female employee's dependents
 - $\text{FEMALE_EMPS} \leftarrow \sigma_{\text{Sex}='F'}(\text{EMPLOYEE})$
 - $\text{EMP_NAMES} \leftarrow \pi_{\text{Fname, Lname, Ssn}}(\text{FEMALE_EMPS})$
 - $\text{EMP_DEPEND} \leftarrow \text{EMP_NAMES} \times \text{DEPENDENT}$
 - $\text{ACTUAL_DEPEND} \leftarrow \sigma_{\text{Ssn}=\text{Essn}}(\text{EMP_DEPEND})$
 - $\text{RES} \leftarrow \pi_{\text{Fname, Lname, Dependent_name}}(\text{ACTUAL_DEPEND})$

ADBMS: Query Processing and Optimization

27

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- The JOIN Operation
 - Denoted by \bowtie
 - Combine related tuples from two relations into single "longer" tuples
 - General join condition of the form $\langle \text{condition} \rangle \text{ AND } \langle \text{condition} \rangle \text{ AND } \dots \text{ AND } \langle \text{condition} \rangle$
 - Example:

$\text{DEPT_MGR} \leftarrow \text{DEPARTMENT} \bowtie_{\text{Mgr_ssn}=\text{Ssn}} \text{EMPLOYEE}$
 $\text{RESULT} \leftarrow \pi_{\text{Dname, Lname, Fname}}(\text{DEPT_MGR})$

ADBMS: Query Processing and Optimization

28

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- **THETA JOIN**
 - A general join condition is of the form
 - $\langle \text{condition} \rangle \text{ AND } \langle \text{condition} \rangle \text{ AND } \dots \text{ AND } \langle \text{condition} \rangle$
 - Each $\langle \text{condition} \rangle$ of the form $A_i \theta B_j$
 - A_i is an attribute of R and B_j is an attribute of S
 - A_i and B_j have the same domain
 - θ (theta) is one of the comparison operators:
 - $\{=, <, \leq, >, \geq, \neq\}$

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- **EQUIJOIN**
 - Only $=$ comparison operator used
 - Always have one or more pairs of attributes that have identical values in every tuple
- **NATURAL JOIN**
 - Denoted by $*$
 - Removes second (superfluous) attribute in an EQUIJOIN condition

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- Join selectivity
 - Expected size of join result divided by the maximum size $n_R * n_S$
- Inner joins
 - Type of match and combine operation
 - Defined formally as a combination of CARTESIAN PRODUCT and SELECTION

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- Set of relational algebra operations $\{\sigma, \pi, \cup, \rho, -, \times\}$ is a **complete set**
 - Any relational algebra operation can be expressed as a sequence of operations from this set
 - Eg: $R \cap S = ((R \cup S) - (R - S)) - (S - R)$

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- **DIVISION**
 - Denoted by \div
 - Example:
 - retrieve the names of employees who work on all the projects that 'John Smith' works on
 - $R(Z) \div S(X)$

SSN_PNOS

Essn	Pno
123456789	1
123456789	2
666884444	3
453453453	1
453453453	2
333445555	2
333445555	3
333445555	10
333445555	20
999887777	30
999887777	10
987987987	10
987987987	30
987654321	30
987654321	20
888665555	20

SMITH_PNOS

Pno
1
2

SSNS

Ssn
123456789
453453453

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- **Aggregate functions and grouping**
 - Common functions applied to collections of numeric values
 - Include SUM, AVERAGE, MAXIMUM, and MINIMUM
 - Group tuples by the value of some of their attributes
 - Apply aggregate function independently to each group
 - $\langle \text{grouping attributes} \rangle \mathcal{S} \langle \text{function list} \rangle (R)$

Translating SQL Queries into Relational Algebra

Relational Algebra Overview

- **Aggregate functions and grouping**
 - Examples:
 - $\rho_{R(\text{Dno}, \text{No_of_emp}, \text{Average_sal})}(\text{Dno } \mathcal{S} \text{ COUNT Ssn, AVERAGE Salary}(\text{EMPLOYEE}))$
 - $\text{Dno } \mathcal{S} \text{ COUNT Ssn, AVERAGE Salary}(\text{EMPLOYEE})$
 - $\mathcal{S} \text{ COUNT Ssn, AVERAGE Salary}(\text{EMPLOYEE})$

Translating SQL Queries into Relational Algebra

Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation R .	$\sigma_{\langle \text{selection condition} \rangle}(R)$
PROJECT	Produces a new relation with only some of the attributes of R , and removes duplicate tuples.	$\pi_{\langle \text{attribute list} \rangle}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{\langle \text{join condition} \rangle} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{\langle \text{join condition} \rangle} R_2$, OR $R_1 \bowtie_{(\langle \text{join attributes 1} \rangle), (\langle \text{join attributes 2} \rangle)} R_2$
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of R_2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R_1 \star_{\langle \text{join condition} \rangle} R_2$, OR $R_1 \star_{(\langle \text{join attributes 1} \rangle), (\langle \text{join attributes 2} \rangle)} R_2$, OR $R_1 \star R_2$

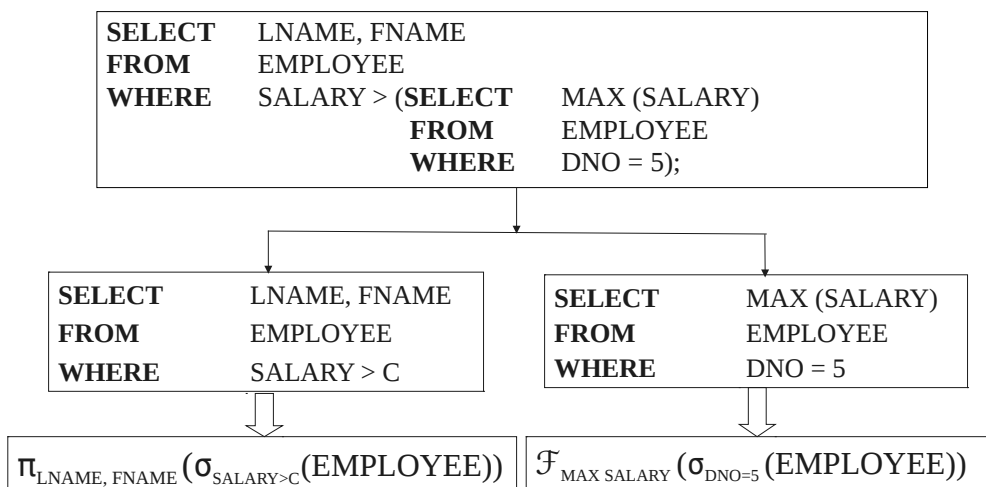
Translating SQL Queries into Relational Algebra

UNION	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2 ; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2 ; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2 ; R_1 and R_2 must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2 .	$R_1 \times R_2$
DIVISION	Produces a relation $R(X)$ that includes all tuples $t[X]$ in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$, where $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

Translating SQL Queries into Relational Algebra

- **Query block:**
 - The basic unit that can be translated into the algebraic operators and optimized
- A query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clause if these are part of the block
- **Nested queries** within a query are identified as separate query blocks
- Aggregate operators in SQL must be included in the extended algebra

Translating SQL Queries into Relational Algebra



Translating SQL Queries into Relational Algebra

EMPLOYEE									
Filename	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

DEPARTMENT			
Dname	Dnumber	Mgr_ssn	Mgr_start_date

DEPT_LOCATIONS	
Dnumber	Dlocation

PROJECT			
Pname	Pnumber	Plocation	Dnum

WORKS_ON		
Essn	Pno	Hours

DEPENDENT				
Essn	Dependent_name	Sex	Bdate	Relationship

Exercises depend on this COMPANY schema and check your textbook for answers.

Translating SQL Queries into Relational Algebra

- Examples:
 - Retrieve the name and address of all employees who work for the 'Research' department.
 - QUERY???
 - RELATIONAL ALGEBRA???
 - 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.
 - QUERY???
 - RELATIONAL ALGEBRA???

Translating SQL Queries into Relational Algebra

- Examples:
 - Find the names of employees who work on all the projects controlled by department number 5.
 - QUERY???
 - RELATIONAL ALGEBRA???
 - Make a list of project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.
 - QUERY???
 - RELATIONAL ALGEBRA???

Translating SQL Queries into Relational Algebra

- Examples:
 - List the names of all employees with two or more dependents.
 - QUERY???
 - RELATIONAL ALGEBRA???
 - List the names of managers who have at least one dependent.
 - QUERY???
 - RELATIONAL ALGEBRA???
 - Retrieve the names of employees who have no dependents.
 - QUERY???
 - RELATIONAL ALGEBRA???

Algorithms for External Sorting

- Why Sorting?
 - A classic problem in computer science!
 - A precursor to other algorithms like search and merge
 - Important utility in DBMS:
 - Data requested in sorted order (e.g., ORDER BY)
 - e.g., find students in increasing gpa order
 - Sorting useful for eliminating duplicate copies in a collection of records (e.g., SELECT DISTINCT)
 - Sort-merge join algorithm involves sorting.
 - Sorting is first step in bulk loading B+ tree index.

Algorithms for External Sorting

- External Sorting:
 - Refers to sorting algorithms that are suitable for large files of records stored on disk that do not fit entirely in main memory, such as most database files
- Sort-Merge Strategy:
 - Starts by sorting small subfiles (runs) of the main file and then merges the sorted runs, creating larger sorted subfiles that are merged in turn.
 - Sorting phase: $n_R = \lceil (b/n_B) \rceil$
 - Merging phase: $d_M = \text{Min}(n_B - 1, n_R)$; $n_P = \lceil (\log_{d_M}(n_R)) \rceil$
 - n_R : number of initial runs; b : number of file blocks;
 - n_B : available buffer space; d_M : degree of merging;
 - n_P : number of passes.

Algorithms for External Sorting

Sort the following 16 numbers

- The system's main memory is capable of storing 4 numbers

19	11	03	05	02	07	31	17	13	29	23	41	37	51	47	43
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

19	11	03	05	02	07	31	17	13	29	23	41	37	51	47	43
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Algorithms for External Sorting

19	11	03	05	INTERNAL SORTING	03	05	11	19
02	07	31	17	INTERNAL SORTING	02	07	17	31
13	29	23	41	INTERNAL SORTING	13	23	29	41
37	51	47	43	INTERNAL SORTING	37	43	47	51

Algorithms for External Sorting

03	05	11	19	03	05	11	19	03	05	11	19	...	03	05	11	19
02	07	17	31	02	07	17	31	02	07	17	31	...	02	07	17	31
13	23	29	41	13	23	29	41	13	23	29	41	...	13	23	29	41
37	43	47	51	37	43	47	51	37	43	47	51	...	37	43	47	51
EXTERNAL MERGING				EXTERNAL MERGING				EXTERNAL MERGING				...	EXTERNAL MERGING			
02				02	03			02	03	05		...	02	03	05	07
													11	13	17	19
													23	29	31	37
													41	43	47	51

Algorithms for External Sorting

- Calculate cost in terms of the number of disk block reads and writes [For a file of size b blocks]???

```

set
  i ← 1;
  j ← b; {size of the file in blocks}
  k ← n_B; {size of buffer in blocks}
  m ← ⌈(j/k)⌉;

{Sorting Phase}
while (i ≤ m)
do {
  read next k blocks of the file into the buffer or if there are less than k blocks
  remaining, then read in the remaining blocks;
  sort the records in the buffer and write as a temporary subfile;
  i ← i + 1;
}

{Merging Phase: merge subfiles until only 1 remains}
set
  i ← 1;
  p ← ⌈log_{k-1} m⌉; {p is the number of passes for the merging phase}
  j ← m;
while (i ≤ p)
do {
  n ← 1;
  q ← ⌈j/(k-1)⌉; {number of subfiles to write in this pass}
  while (n ≤ q)
  do {
    read next k-1 subfiles or remaining subfiles (from previous pass)
    one block at a time;
    merge and write as new subfile one block at a time;
    n ← n + 1;
  }
  j ← q;
  i ← i + 1;
}

```

ADBMS: Query Processing and Optimi

Outline of the sort-merge algorithm for external sorting.

Algorithms for SELECT and JOIN Operations

- Implementing the SELECT Operation
- Examples:
 - (OP1): $\sigma_{SSN='123456789'}(EMPLOYEE)$
 - (OP2): $\sigma_{DNUMBER>5}(DEPARTMENT)$
 - (OP3): $\sigma_{DNO=5}(EMPLOYEE)$
 - (OP4): $\sigma_{DNO=5 \text{ AND } SALARY>30000 \text{ AND } SEX=F}(EMPLOYEE)$
 - (OP5): $\sigma_{ESSN=123456789 \text{ AND } PNO=10}(WORKS_ON)$

ADBMS: Query Processing and Optimization

50

Algorithms for SELECT and JOIN Operations

- Search Methods for Simple Selection:
 - S1 Linear search (brute force):**
 - Retrieve every record in the file, and test whether its attribute values satisfy the selection condition.
 - S2 Binary search:**
 - If the selection condition involves an equality comparison on a key attribute on which the file is ordered, binary search can be used.
 - S3 Using a primary index or hash key to retrieve a single record:**
 - If the selection condition involves an equality comparison on a key attribute with a primary index (or a hash key), use the primary index (or the hash key) to retrieve the record

ADBMS: Query Processing and Optimization

51

Algorithms for SELECT and JOIN Operations

- Search Methods for Simple Selection:
 - S4 Using a primary index to retrieve multiple records:**
 - If the comparison condition is $>$, \geq , $<$, or \leq on a key field with a primary index, use the index to find the record satisfying the corresponding equality condition, then retrieve all subsequent records in the (ordered) file.
 - S5 Using a clustering index to retrieve multiple records:**
 - If the selection condition involves an equality comparison on a non-key attribute with a clustering index, use the clustering index to retrieve all the records satisfying the selection condition.
 - S6 Using a secondary (B+-tree) index:**
 - On an equality comparison, this search method can be used to retrieve a single record if the indexing field has unique values (is a key) or to retrieve multiple records if the indexing field is not a key.
 - In addition, it can be used to retrieve records on conditions involving $>$, \geq , $<$, or \leq

ADBMS: Query Processing and Optimization

52

Algorithms for SELECT and JOIN Operations

- Search Methods for Simple Selection:
 - **S7 Conjunctive selection:**
 - If an attribute involved in any single simple condition in the conjunctive condition has an access path that permits the use of one of the methods S2 to S6, use that condition to retrieve the records and then check whether each retrieved record satisfies the remaining simple conditions in the conjunctive condition.
 - **S8 Conjunctive selection using a composite index**
 - If two or more attributes are involved in equality conditions in the conjunctive condition and a composite index (or hash structure) exists on the combined field, we can use the index directly: $\sigma_{\text{ESSN}=123456789 \text{ AND } \text{PNO}=10}(\text{WORKS_ON})$

ADBMS: Query Processing and Optimization

53

Algorithms for SELECT and JOIN Operations

- Search Methods for Complex Selection:
 - **S9 Conjunctive selection by intersection of record pointers:**
 - This method is possible if secondary indexes are available on all (or some of) the fields involved in equality comparison conditions in the conjunctive condition and if the indexes include record pointers (rather than block pointers).
 - Each index can be used to retrieve the record pointers that satisfy the individual condition.
 - The intersection of these sets of record pointers gives the record pointers that satisfy the conjunctive condition, which are then used to retrieve those records directly.
 - If only some of the conditions have secondary indexes, each retrieved record is further tested to determine whether it satisfies the remaining conditions.

ADBMS: Query Processing and Optimization

54

Algorithms for SELECT and JOIN Operations

- Whenever a *single condition specifies the selection*, we can only check whether an access path exists on the attribute involved in that condition.
 - If an access path exists, the method corresponding to that access path is used; otherwise, the “brute force” linear search approach of method S1 is used. (See OP1, OP2 and OP3)
- For *conjunctive selection conditions*, whenever more than one of the attributes involved in the conditions have an access path, query optimization should be done to choose the access path that *retrieves the fewest records* in the most efficient way.
- Disjunctive selection conditions
 - $\sigma_{\text{Dno}=5 \text{ OR } \text{Salary} > 30000 \text{ OR } \text{Sex}='F'}(\text{EMPLOYEE})$

ADBMS: Query Processing and Optimization

55

Algorithms for SELECT and JOIN Operations

- Implementing the JOIN Operation:
 - Join (EQUIJOIN, NATURAL JOIN)
 - two-way join: a join on two files
 - e.g. $R \bowtie_{A=B} S$
 - multi-way joins: joins involving more than two files.
 - e.g. $R \bowtie_{A=B} S \bowtie_{C=D} T$
 - Examples
 - (OP6): $\text{EMPLOYEE} \bowtie_{\text{DNO}=\text{DNUMBER}} \text{DEPARTMENT}$
 - (OP7): $\text{DEPARTMENT} \bowtie_{\text{MGRSSN}=\text{SSN}} \text{EMPLOYEE}$

ADBMS: Query Processing and Optimization

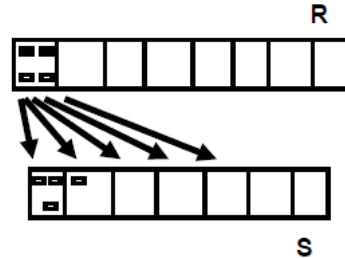
56

Algorithms for SELECT and JOIN Operations

- Methods for implementing joins:
 - J1 Nested-loop join (brute force):**
 - For each record t in R (outer loop), retrieve every record s from S (inner loop) and test whether the two records satisfy the join condition $t[A] = s[B]$.

```

FOR EACH  $r$  IN  $R$ 
  FOR EACH  $s$  IN  $S$ 
    IF ( $r[A] == s[B]$ )
      #Add tuple to result
    
```



Algorithms for SELECT and JOIN Operations

- Methods for implementing joins:
 - J2 Single-loop join** (Using an access structure to retrieve the matching records):
 - If an index (or hash key) exists for one of the two join attributes — say, B of S — retrieve each record t in R , one at a time, and then use the access structure to retrieve directly all matching records s from S that satisfy $s[B] = t[A]$.

Algorithms for SELECT and JOIN Operations

- Methods for implementing joins:
 - J3 Sort-merge join:**
 - If the records of R and S are *physically sorted (ordered)* by value of the join attributes A and B , respectively, we can implement the join in the most efficient way possible.
 - Both files are scanned in order of the join attributes, matching the records that have the same values for A and B .
 - In this method, the records of each file are scanned only once each for matching with the other file—unless both A and B are non-key attributes, in which case the method needs to be modified slightly.

Implementing the operation $T \leftarrow R \bowtie_{A=B} S$; where R has n tuples and S has m tuples
Using sort-merge

```

sort the tuples in  $R$  on attribute  $A$ ; (* assume  $R$  has  $n$  tuples (records) *)
sort the tuples in  $S$  on attribute  $B$ ; (* assume  $S$  has  $m$  tuples (records) *)
set  $i \leftarrow 1, j \leftarrow 1$ ;
while ( $i \leq n$ ) and ( $j \leq m$ )
do {
  if  $R(i)[A] > S(j)[B]$ 
    then set  $j \leftarrow j + 1$ 
  elseif  $R(i)[A] < S(j)[B]$ 
    then set  $i \leftarrow i + 1$ 
  else { (*  $R(i)[A] = S(j)[B]$ , so we output a matched tuple *)
    output the combined tuple  $\langle R(i), S(j) \rangle$  to  $T$ ;

    (* output other tuples that match  $R(i)$ , if any *)
    set  $l \leftarrow j + 1$ ;
    while ( $l \leq m$ ) and ( $R(i)[A] = S(l)[B]$ )
    do { output the combined tuple  $\langle R(i), S(l) \rangle$  to  $T$ ;
        set  $l \leftarrow l + 1$ 
      }

    (* output other tuples that match  $S(j)$ , if any *)
    set  $k \leftarrow i + 1$ ;
    while ( $k \leq n$ ) and ( $R(k)[A] = S(j)[B]$ )
    do { output the combined tuple  $\langle R(k), S(j) \rangle$  to  $T$ ;
        set  $k \leftarrow k + 1$ 
      }
    set  $i \leftarrow k, j \leftarrow l$ 
  }
}
    
```

Algorithms for SELECT and JOIN Operations


Algorithms for SELECT and JOIN Operations

- Methods for implementing joins:
 - **J4 Partition Hash-join:**
 - The records of files R and S are both hashed to the *same hash file*, using the *same hashing function* on the join attributes A of R and B of S as hash keys.
 - A single pass through the file with fewer records (say, R) hashes its records to the hash file buckets.
 - A single pass through the other file (S) then hashes each of its records to the appropriate bucket, where the record is combined with all matching records from R.

Algorithms for SELECT and JOIN Operations

- Factors affecting JOIN performance
 - Available buffer space
 - advantageous to read as many blocks as possible at a time into memory from the file whose records are used for the outer loop ($n_B - 2$)
 - Reduced the total number of block accesses

Algorithms for SELECT and JOIN Operations

- Factors affecting JOIN performance
 - Available buffer space
 - Join selection factor
 - Fraction of records in one file that will be joined with records in the other file
 - DEPARTMENT  EMPLOYEE with secondary index on MGRSSN (level 2) and SSN (level 4)
 - $n_B = 7$ Blocks
 - $r_D = 50$ records stored in $b_D = 10$ disk blocks
 - $r_E = 6000$ records stored in $b_E = 2000$ disk blocks

Algorithms for SELECT and JOIN Operations

- Factors affecting JOIN performance
 - Available buffer space
 - Join selection factor
 - Two options to implement J2
 - Retrieve all EMP record, use the index on MGR_SSN of DEP
 - Join selection factor:
 - Number of block accesses:
 - Retrieve all DEP record, use the index on SSN of EMP
 - Join selection factor:
 - Number of block accesses:

Algorithms for SELECT and JOIN Operations

- Factors affecting JOIN performance
 - Available buffer space
 - Join selection factor
 - Choice of inner VS outer relation
 - $\text{EMPLOYEE} \bowtie_{\text{DNO=DNUMBER}} \text{DEPARTMENT}$
 - $n_B = 7$ Blocks
 - $r_D = 50$ records stored in $b_D = 10$ disk blocks
 - $r_E = 6000$ records stored in $b_E = 2000$ disk blocks

Algorithms for SELECT and JOIN Operations

- Factors affecting JOIN performance
 - Available buffer space
 - Join selection factor
 - Choice of inner VS outer relation
 - Assume: $\text{EMPLOYEE} \bowtie_{\text{DNO=DNUMBER}} \text{DEPARTMENT}$
 - For nested-loop join, if EMPLOYEE is used as outer loop:
 - Total number of blocks accessed for outer loop:
 - Number of times $n_B - 2$ blocks of outer file are loaded to memory:
 - Number of files accessed for inner loop file:
 - Total number of block read access:

Algorithms for SELECT and JOIN Operations

Other types of JOIN algorithms

- General form of Partition hash join
 - Partitioning phase:
 - Each file (R and S) is first partitioned into M partitions using a partitioning hash function on the join attributes:
 - $R_1, R_2, R_3, \dots, R_m$ and $S_1, S_2, S_3, \dots, S_m$
 - Minimum number of in-memory buffers needed for the partitioning phase: $M+1$.
 - A disk sub-file is created per partition to store the tuples for that partition.
 - Joining or probing phase:
 - Involves M iterations, one per partitioned file.
 - Iteration i involves joining partitions R_i and S_i .

Algorithms for SELECT and JOIN Operations

- Partitioned Hash Join Procedure:
 - Assume R_i is smaller than S_i .
 1. Copy records from R_i into memory buffers.
 2. Read all blocks from S_i , one at a time and each record from S_i is used to *probe* for a matching record(s) from partition S_i .
 3. Write matching record from R_i after joining to the record from S_i into the result file.

Algorithms for SELECT and JOIN Operations

- Cost analysis of partition hash join:
 1. Reading and writing each record from R and S during the partitioning phase:
 $(b_R + b_S), (b_R + b_S)$
 2. Reading each record during the joining phase:
 $(b_R + b_S)$
 3. Writing the result of join:
 b_{RES}
- Total Cost:
 - $3 * (b_R + b_S) + b_{RES}$

Algorithms for SELECT and JOIN Operations

Implementing the JOIN Operation

- Hybrid hash join:
 - Same as partitioned hash join except:
- Joining phase of one of the partitions is included during the partitioning phase.
 - Partitioning phase:
 - Allocate buffers for smaller relation- one block for each of the M-1 partitions, remaining blocks to partition 1.
 - Repeat for the larger relation in the pass through S.)
 - Joining phase:
 - M-1 iterations are needed for the partitions R2 , R3 , R4 ,Rm and S2 , S3 , S4 ,Sm. R1 and S1 are joined during the partitioning of S1, and results of joining R1 and S1 are already written to the disk by the end of partitioning phase.

Algorithms for PROJECT and SET Operations

- Algorithm for PROJECT operations $\pi_{\langle \text{attribute list} \rangle}(R)$
 1. If $\langle \text{attribute list} \rangle$ has a key of relation R, extract all tuples from R with only the values for the attributes in $\langle \text{attribute list} \rangle$.
 2. If $\langle \text{attribute list} \rangle$ does NOT include a key of relation R, duplicated tuples must be removed from the results.
- Methods to remove duplicate tuples
 1. Sorting
 2. Hashing

Algorithms for PROJECT and SET Operations

Implementing the operation $T \leftarrow \pi_{\langle \text{attribute list} \rangle}(R)$

```

create a tuple  $t[\langle \text{attribute list} \rangle]$  in  $T'$  for each tuple  $t$  in R;
(*  $T'$  contains the projection results before duplicate elimination *)
if  $\langle \text{attribute list} \rangle$  includes a key of R
then  $T \leftarrow T'$ 
else { sort the tuples in  $T'$ ;
      set  $i \leftarrow 1, j \leftarrow 2$ ;
      while  $i \leq n$ 
      do { output the tuple  $T'[i]$  to  $T$ ;
          while  $T'[i] = T'[j]$  and  $j \leq n$  do  $j \leftarrow j + 1$ ;
           $i \leftarrow j; j \leftarrow i + 1$ 
          }
      }
(*  $T$  contains the projection result after duplicate elimination *)
  
```

Algorithms for PROJECT and SET Operations

Algorithm for SET operations

- Set operations:
 - UNION, INTERSECTION, SET DIFFERENCE and CARTESIAN PRODUCT
- CARTESIAN PRODUCT of relations R and S include all possible combinations of records from R and S. The attribute of the result include all attributes of R and S.
- Cost analysis of CARTESIAN PRODUCT
 - If R has n records and j attributes and S has m records and k attributes, the result relation will have n*m records and j+k attributes.
- CARTESIAN PRODUCT operation is very expensive and should be avoided if possible.

Algorithms for PROJECT and SET Operations

Algorithm for SET operations

- **UNION**
 - Sort the two relations on the same attributes.
 - Scan and merge both sorted files concurrently, whenever the same tuple exists in both relations, only one is kept in the merged results.
- **INTERSECTION**
 - Sort the two relations on the same attributes.
 - Scan and merge both sorted files concurrently, keep in the merged results only those tuples that appear in both relations.
- **SET DIFFERENCE R-S**
 - Keep in the merged results only those tuples that appear in relation R but not in relation S.

Algorithms for PROJECT and SET Operations

Implementing the operation $T \leftarrow R \cup S$

sort the tuples in R and S using the same unique sort attributes;

set $i \leftarrow 1, j \leftarrow 1$;

while $(i \leq n)$ and $(j \leq m)$

do { if $R(i) > S(j)$

then { output $S(j)$ to T ;

set $j \leftarrow j + 1$

}

elseif $R(i) < S(j)$

then { output $R(i)$ to T ;

set $i \leftarrow i + 1$

}

else set $j \leftarrow j + 1$

(* $R(i)=S(j)$), so we skip one of the duplicate tuples *)

}

if $(i \leq n)$ then add tuples $R(i)$ to $R(n)$ to T ;

if $(j \leq m)$ then add tuples $S(j)$ to $S(m)$ to T ;

Algorithms for PROJECT and SET Operations

Implementing the operation $T \leftarrow R \cap S$

sort the tuples in R and S using the same unique sort attributes;

set $i \leftarrow 1, j \leftarrow 1$;

while $(i \leq n)$ and $(j \leq m)$

do { if $R(i) > S(j)$

then set $j \leftarrow j + 1$

elseif $R(i) < S(j)$

then set $i \leftarrow i + 1$

else { output $R(j)$ to T ;

(* $R(i)=S(j)$), so we output the tuple *)

set $i \leftarrow i + 1, j \leftarrow j + 1$

}

}

Algorithms for PROJECT and SET Operations

Implementing the operation $T \leftarrow R - S$

```
sort the tuples in  $R$  and  $S$  using the same unique sort attributes;
set  $i \leftarrow 1, j \leftarrow 1$ ;
while ( $i \leq n$ ) and ( $j \leq m$ )
do { if  $R(i) > S(j)$ 
    then set  $j \leftarrow j + 1$ 
    elseif  $R(i) < S(j)$ 
    then { output  $R(i)$  to  $T$ ;          (*  $R(i)$  has no matching  $S(j)$ , so output  $R(i)$  *)
        set  $i \leftarrow i + 1$ 
    }
    else set  $i \leftarrow i + 1, j \leftarrow j + 1$ 
}
if ( $i \leq n$ ) then add tuples  $R(i)$  to  $R(n)$  to  $T$ ;
```

Implementing Aggregate Operations and Outer Joins

Implementing Aggregate Operations

- Aggregate operators:
 - MIN, MAX, SUM, COUNT and AVG
- Options to implement aggregate operators:
 - Table Scan
 - Index

Implementing Aggregate Operations and Outer Joins

Implementing Aggregate Operations

- Example
 - SELECT MAX (SALARY)
 - FROM EMPLOYEE;
- If an (ascending) index on SALARY exists for the employee relation, then the optimizer could decide on traversing the index for the largest value, which would entail following the right most pointer in each index node from the root to a leaf.

Implementing Aggregate Operations and Outer Joins

Implementing Aggregate Operations

- SUM, COUNT and AVG
- For a dense index (each record has one index entry):
 - Apply the associated computation to the values in the index.
- For a non-dense index:
 - Actual number of records associated with each index entry must be accounted for

Implementing Aggregate Operations and Outer Joins

Implementing Aggregate Operations

- With GROUP BY: the aggregate operator must be applied separately to each group of tuples.
 - Use sorting or hashing on the group attributes to partition the file into the appropriate groups;
 - Compute the aggregate function for the tuples in each group.
- What if we have Clustering index on the grouping attributes?

Implementing Aggregate Operations and Outer Joins

Implementing Outer Join

- Outer Join Operators:
 - LEFT OUTER JOIN
 - RIGHT OUTER JOIN
 - FULL OUTER JOIN
- The full outer join produces a result which is equivalent to the union of the results of the left and right outer joins

Implementing Aggregate Operations and Outer Joins

Implementing Outer Join

- Example:

```
SELECT FNAME, DNAME
FROM   (EMPLOYEE LEFT OUTER JOIN DEPARTMENT
        ON DNO = DNUMBER);
```
- Note: The result of this query is a table of employee names and their associated departments. It is similar to a regular join result, with the exception that if an employee does not have an associated department, the employee's name will still appear in the resulting table, although the department name would be indicated as null

Implementing Aggregate Operations and Outer Joins

Implementing Outer Join

- Modifying Join Algorithms:
 - Nested Loop or Sort-Merge joins can be modified to implement outer join. E.g.,
 - For left outer join, use the left relation as outer relation and construct result from every tuple in the left relation.
 - If there is a match, the concatenated tuple is saved in the result.
 - However, if an outer tuple does not match, then the tuple is still included in the result but is padded with a null value(s).

Implementing Aggregate Operations and Outer Joins

Implementing Outer Join

- Executing a combination of relational algebra operators.
- Implement the previous left outer join example
 - {Compute the JOIN of EMPLOYEE and DEPARTMENT tables}
 - $TEMP1 \leftarrow \pi_{FNAME,DNAME}(EMPLOYEE \bowtie_{DNO=DNUMBER} DEPARTMENT)$
 - {Find the EMPLOYEES that do not appear in the JOIN}
 - $TEMP2 \leftarrow \pi_{FNAME}(EMPLOYEE) - \pi_{FNAME}(Temp1)$

Implementing Aggregate Operations and Outer Joins

Implementing Outer Join

- {Pad each tuple in TEMP2 with a null DNAME field}
 - $TEMP2 \leftarrow TEMP2 \times NULL$
- {UNION the temporary tables to produce the LEFT OUTER JOIN}
 - $RESULT \leftarrow TEMP1 \cup TEMP2$
- The cost of the outer join, as computed above, would include the cost of the associated steps (i.e., join, projections and union).

Combining Operations using Pipelining

- Motivation
 - A query is mapped into a sequence of operations.
 - Each execution of an operation produces a temporary result.
 - Generating and saving temporary files on disk is time consuming and expensive.
- Alternative:
 - Avoid constructing temporary results as much as possible.
 - Pipeline the data through multiple operations - pass the result of a previous operator to the next without waiting to complete the previous operation.

Combining Operations using Pipelining

- Example:
 - For a 2-way join, combine the 2 selections on the input and one projection on the output with the Join.
- Dynamic generation of code to allow for multiple operations to be pipelined.
- Results of a select operation are fed in a "Pipeline" to the join algorithm.
- Also known as stream-based processing.

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
 - 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
 - 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:

$\pi_{\text{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}}$

$((((\sigma_{\text{PLOCATION}='STAFFORD'}(\text{PROJECT})))$

$\bowtie_{\text{DNUM=DNUMBER}} (\text{DEPARTMENT}))$

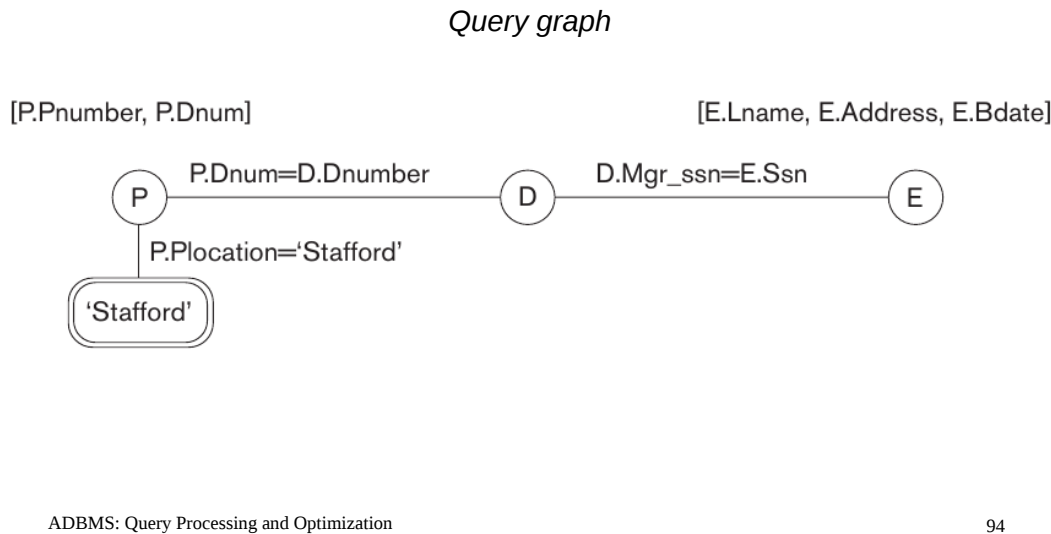
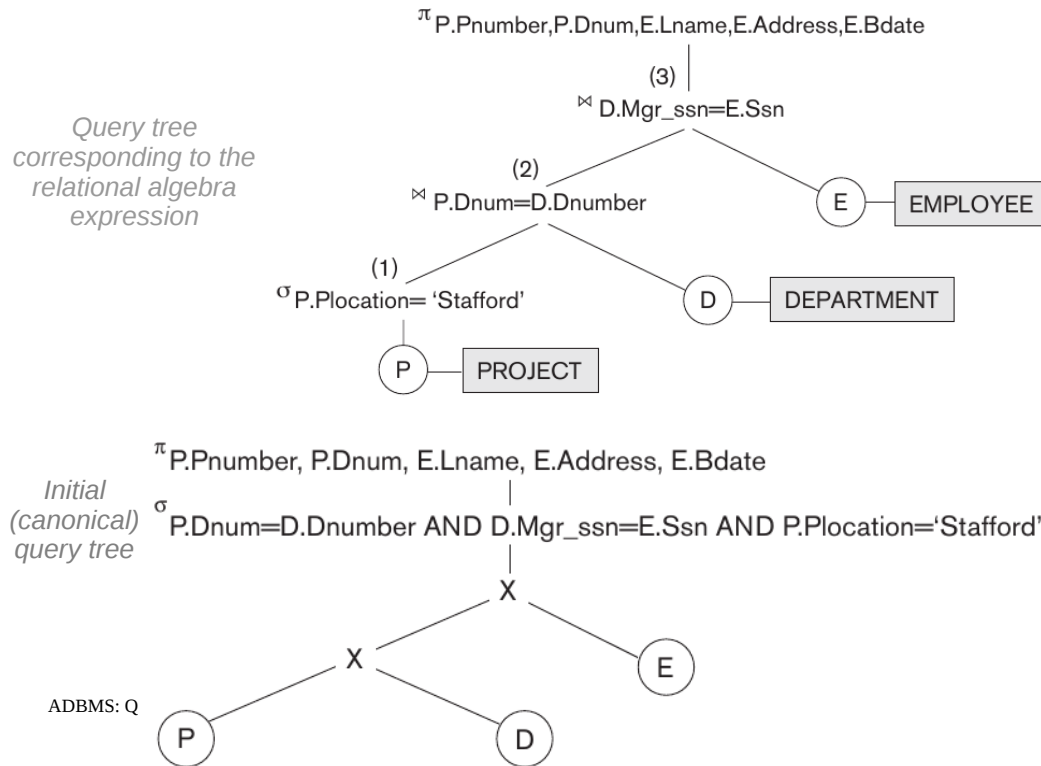
$\bowtie_{\text{MGRSSN=SSN}} (\text{EMPLOYEE}))$

Using Heuristics in Query Optimization

- SQL query:

```
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';
```

Using Heuristics in Query Optimization



94

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:


```

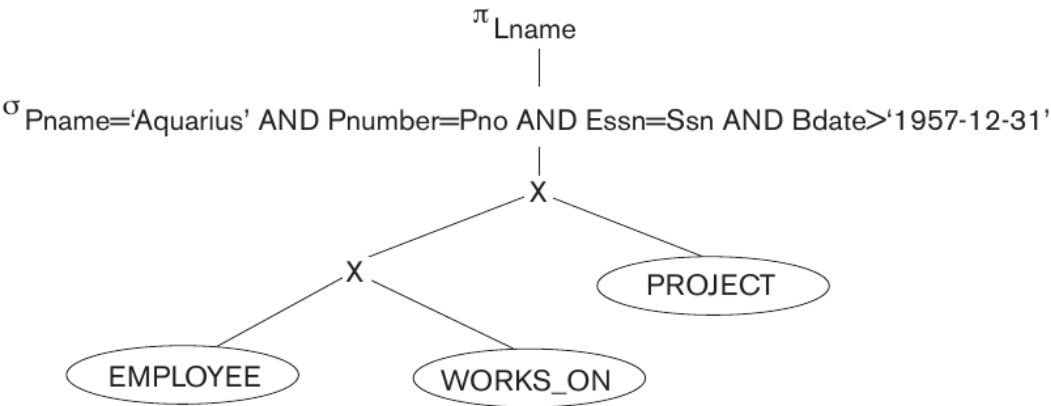
SELECT  LNAME
FROM    EMPLOYEE, WORKS_ON, PROJECT
WHERE   PNAME = 'AQUARIUS'
        AND  PNMUBER=PNO
        AND  ESSN=SSN
        AND  BDATE > '1957-12-31';
      
```

Using Heuristics in Query Optimization

- Steps in converting a query tree during heuristic optimization:
 - Initial (canonical) query tree for SQL query Q
 - Moving SELECT operations down the query tree
 - Applying the more restrictive SELECT operation first
 - Replacing CARTESIAN PRODUCT and SELECT with JOIN operations
 - Moving PROJECT operations down the query tree

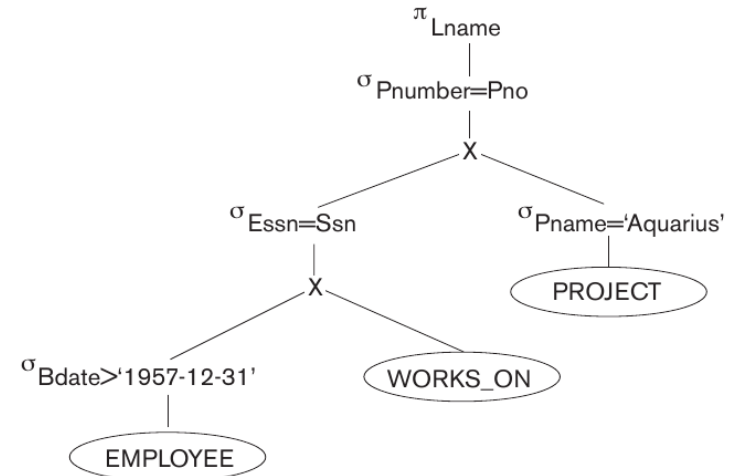
Using Heuristics in Query Optimization

Initial (canonical) query tree



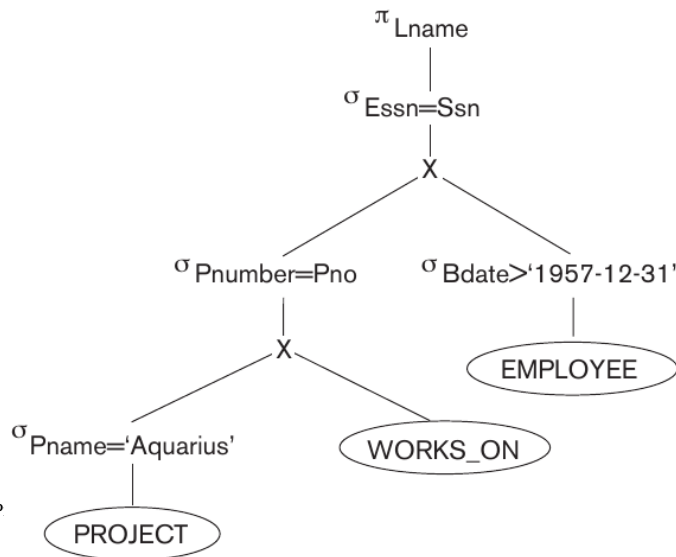
Using Heuristics in Query Optimization

Moving SELECT operations down the query tree



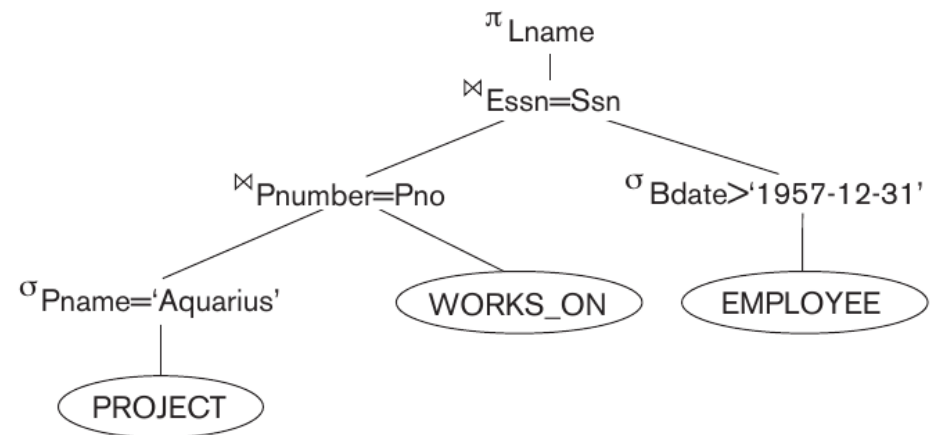
Using Heuristics in Query Optimization

Applying the more restrictive SELECT operation first



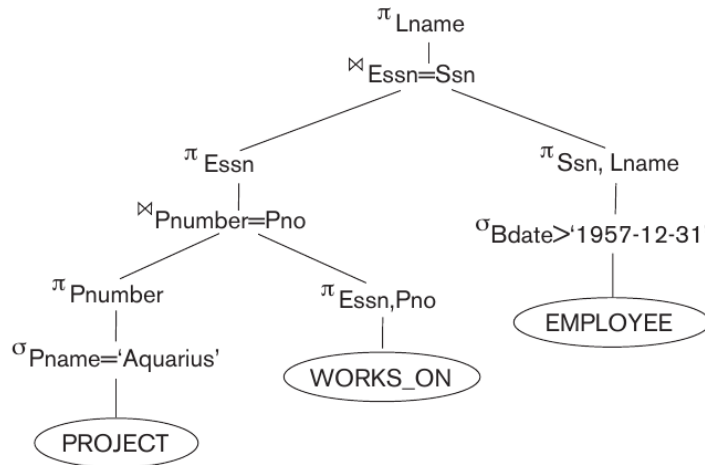
Using Heuristics in Query Optimization

Replacing CARTESIAN PRODUCT and SELECT with JOIN operations



Using Heuristics in Query Optimization

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 σ operations:

$$- \sigma_{c_1 \text{ AND } c_2 \text{ AND } \dots \text{ AND } c_n}(R) = \sigma_{c_1}(\sigma_{c_2}(\dots(\sigma_{c_n}(R))\dots))$$

2. Commutativity of σ : The σ operation is commutative:

$$- \sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_2}(\sigma_{c_1}(R))$$

Using Heuristics in Query Optimization

General Transformation Rules for Relational Algebra Operations:

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

$$- \pi_{List1}(\pi_{List2}(\dots(\pi_{Listn}(R))\dots)) = \pi_{List1}(R)$$

4. Commuting σ with π : If the selection condition c involves only the attributes A_1, \dots, A_n in the projection list, the two operations can be commuted:

$$- \pi_{A_1, A_2, \dots, A_n}(\sigma_c(R)) = \sigma_c(\pi_{A_1, A_2, \dots, A_n}(R))$$

Using Heuristics in Query Optimization

General Transformation Rules for Relational Algebra Operations:

5. Commutativity of \bowtie (and \times): The \bowtie operation is commutative as is the \times operation:

$$- R \bowtie_c S = S \bowtie_c R; R \times S = S \times R$$

6. Commuting σ with \bowtie (or \times): 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:

$$- \sigma_c(R \bowtie S) = (\sigma_c(R)) \bowtie S$$

Using Heuristics in Query Optimization

General Transformation Rules for Relational Algebra Operations

- Alternatively, if the selection condition c can be written as $(c1 \text{ and } c2)$, where condition $c1$ involves only the attributes of R and condition $c2$ involves only the attributes of S , the operations commute as follows:
 - $\sigma_c (R \bowtie S) = (\sigma_{c1} (R)) \bowtie (\sigma_{c2} (S))$

Using Heuristics in Query Optimization

General Transformation Rules for Relational Algebra Operations

- 7. Commuting π with \bowtie (or \times):** Suppose that the projection list is $L = \{A1, \dots, An, B1, \dots, Bm\}$, where $A1, \dots, An$ are attributes of R and $B1, \dots, Bm$ are attributes of S . If the join condition c involves only attributes in L , the two operations can be commuted as follows:
 - $\pi_L (R \bowtie_C S) = (\pi_{A1, \dots, An} (R)) \bowtie_C (\pi_{B1, \dots, Bm} (S))$
- 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

- 8. Commutativity of set operations:** The set operations \cup and \cap are commutative but “ $-$ ” is not.
- 9. Associativity of \bowtie , \times , \cup , and \cap :** These four operations are individually associative; that is, if θ stands for any one of these four operations (throughout the expression), we have
 - $(R \theta S) \theta T = R \theta (S \theta T)$
- 10. Commuting σ with set operations:** The σ operation commutes with \cup , \cap , and “ $-$ ”. If θ stands for any one of these three operations, we have
 - $\sigma_c (R \theta S) = (\sigma_c (R)) \theta (\sigma_c (S))$

Using Heuristics in Query Optimization

General Transformation Rules for Relational Algebra Operations

- 11. The π operation commutes with \cup**
 - $\pi_L (R \cup S) = (\pi_L (R)) \cup (\pi_L (S))$
- 12. Converting a (σ, \times) sequence into \bowtie :** If the condition c of a σ that follows a \times corresponds to a join condition, convert the (σ, \times) sequence into a \bowtie as follows:
 - $(\sigma_c (R \times S)) = (R \bowtie_c S)$
- 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.

Using Heuristics in Query Optimization

Outline of a Heuristic Algebraic Optimization Algorithm

4. Using Rule 12, combine a Cartesian product operation with a subsequent select operation in the tree into a join operation.
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:**
 - 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

Using Selectivity and Cost Estimates in Query Optimization

- **Cost Components for Query Execution**
 1. Access cost to secondary storage
 2. Storage cost
 3. Computation cost
 4. Memory usage cost
 5. Communication cost
- **Note:** Different database systems may focus on different cost components.

Using Selectivity and Cost Estimates in Query Optimization

- **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)

Using Selectivity and Cost Estimates in Query Optimization

- **Catalog Information Used in Cost Functions**
 - Information about indexes and indexing attributes of a file
 - Number of levels (x) of each multilevel index
 - Number of first-level index blocks (bl_1)
 - Number of distinct values (d) of an attribute
 - Selectivity (sl) of an attribute
 - Selection cardinality (s) of an attribute. ($s = sl * r$)

Using Selectivity and Cost Estimates in Query Optimization

Examples of Cost Functions for SELECT

- S1. Linear search (brute force) approach
 - $C_{S1a} = b$;
 - For an equality condition on a key, $C_{S1a} = (b/2)$ if the record is found; otherwise $C_{S1a} = b$.
- S2. Binary search:
 - $C_{S2} = \log_2 b + (s/bfr) - 1$
 - For an equality condition on a unique (key) attribute, $C_{S2} = \log_2 b$
- S3. Using a primary index (S3a) or hash key (S3b) to retrieve a single record
 - $C_{S3a} = x + 1$; $C_{S3b} = 1$ for static or linear hashing;
 - $C_{S3b} = 1$ for extendible hashing;

Using Selectivity and Cost Estimates in Query Optimization

Examples of Cost Functions for SELECT

- S4. Using an ordering index to retrieve multiple records:
 - For the comparison condition on a key field with an ordering index, $C_{S4} = x + (b/2)$
- S5. Using a clustering index to retrieve multiple records:
 - $C_{S5} = x + \lceil (s/bfr) \rceil$
- S6. Using a secondary (B+-tree) index:
 - For an equality comparison, $C_{S6a} = x + s$;
 - For a comparison condition such as $>$, $<$, $>=$, or $<=$,
 - $C_{S6a} = x + (b_{l1}/2) + (r/2)$

Using Selectivity and Cost Estimates in Query Optimization

Examples of Cost Functions for SELECT

- S7. Conjunctive selection:
 - Use either S1 or one of the methods S2 to S6 to solve.
 - For the latter case, use one condition to retrieve the records and then check in the memory buffer whether each retrieved record satisfies the remaining conditions in the conjunction.
- S8. Conjunctive selection using a composite index:
 - Same as S3a, S5 or S6a, depending on the type of index.
- Examples of using the cost functions.

Using Selectivity and Cost Estimates in Query Optimization

Examples of Cost Functions for JOIN

- Join selectivity (js)
 - $js = | (R \bowtie_c S) | / | R \times S | = | (R \bowtie_c S) | / (|R| * |S|)$
 - If condition C does not exist, $js = 1$;
 - If no tuples from the relations satisfy condition C , $js = 0$;
 - Usually, $0 \leq js \leq 1$;
- Size of the result file after join operation
 - $| (R \bowtie_c S) | = js * |R| * |S|$

Using Selectivity and Cost Estimates in Query Optimization

Examples of Cost Functions for JOIN

- J1. Nested-loop join:
 - $C_{J1} = b_R + (b_R * b_S) + ((js * |R| * |S|)/bfr_{RS})$
 - (Use R for outer loop)
- J2. Single-loop join (using an access structure to retrieve the matching record(s))
 - If an index exists for the join attribute B of S with index levels x_B , we can retrieve each record s in R and then use the index to retrieve all the matching records t from S that satisfy $t[B] = s[A]$.
 - The cost depends on the type of index.

Using Selectivity and Cost Estimates in Query Optimization

Examples of Cost Functions for JOIN

- J2. Single-loop join (contd.)
 - For a secondary index,
 - $C_{J2a} = b_R + (|R| * (x_B + s_B)) + ((js * |R| * |S|)/bfr_{RS});$
 - For a clustering index,
 - $C_{J2b} = b_R + (|R| * (x_B + (s_B/bfr_B))) + ((js * |R| * |S|)/bfr_{RS});$
 - For a primary index,
 - $C_{J2c} = b_R + (|R| * (x_B + 1)) + ((js * |R| * |S|)/bfr_{RS});$
 - If a hash key exists for one of the two join attributes — B of S
 - $C_{J2d} = b_R + (|R| * h) + ((js * |R| * |S|)/bfr_{RS});$
- J3. Sort-merge join:
 - $C_{J3a} = C_S + b_R + b_S + ((js * |R| * |S|)/bfr_{RS});$
 - (CS: Cost for sorting files)

Using Selectivity and Cost Estimates in Query Optimization

- **Multiple Relation Queries and Join Ordering**
 - A query joining n relations will have n-1 join operations, and hence can have a large number of different join orders when we apply the algebraic transformation rules.
 - Current query optimizers typically limit the structure of a (join) query tree to that of left-deep (or right-deep) trees.
- **Left-deep tree:**
 - A binary tree where the right child of each non-leaf node is always a base relation.
 - Amenable to pipelining
 - Could utilize any access paths on the base relation (the right child) when executing the join.

Overview of Query Optimization in Oracle

- Oracle DBMS V8
 - **Rule-based query optimization:** the optimizer chooses execution plans based on heuristically ranked operations.
 - (Currently it is being phased out)
 - **Cost-based query optimization:** the optimizer examines alternative access paths and operator algorithms and chooses the execution plan with lowest estimate cost.
 - The query cost is calculated based on the estimated usage of resources such as I/O, CPU and memory needed.
 - Application developers could specify hints to the ORACLE query optimizer.
 - The idea is that an application developer might know more information about the data.

Semantic Query Optimization

- **Semantic Query Optimization:**

- Uses constraints specified on the database schema in order to modify one query into another query that is more efficient to execute.

- Consider the following SQL query,

```
SELECT      E.LNAME, M.LNAME
FROM        EMPLOYEE E M
WHERE       E.SUPERSSN=M.SSN AND E.SALARY>M.SALARY
```

- Explanation:

- Suppose that we had a constraint on the database schema that stated that no employee can earn more than his or her direct supervisor. If the semantic query optimizer checks for the existence of this constraint, it need not execute the query at all because it knows that the result of the query will be empty. Techniques known as theorem proving can be used for this purpose.