CHAPTER 2
ABSTRACT DATABASE QUERIES

Section 1. Introduction

A request for data from a database is known as a ***query***. In this chapter we discuss the two principal abstract approaches underlying the design of languages for querying a database - an algebraic language approach (or ***relational algebra***) and a predicate calculus language approach (or ***relational calculus***).

In relational algebra, queries are expressed by applying a sequence of specialized operators to relations, while predicate calculus languages uses predicate logic to describe conditions (or predicates) that tuples in a relation must satisfy, without specifying how the system is to go about retrieving the specified data. The predicate-based approach are further divided into two classes -- ***tuple calculus*** and ***domain calculus*** -- depending on whether the primitive objects of the predicate are tuples or elements of the domain of some attributes. We shall only consider tuple calculus.

In Chapter 3 we shall discuss the query language SQL (for "Structured Query Language" -- pronounced "S-Q-L" by some and "sequel" by others). SQL is still the dominant language for extracting data from a relational database. While one can jump directly into SQL without the need for understanding either of the more abstract languages, one will see some of the terminology used in both relational algebra and tuple calculus appearing in SQL, and one should be better able to understand what is going on in an SQL query by relating that query to a relational algebra query, or tuple calculus query, or sometimes both.

Examples of Queries:

A. Consider the **`Salesperson-Customer-Orders`** database from Chapter 1. The following are examples of some of the types of information one might want from this database.

   1. Get full details all salespeople.
   2. Get only the names and ages of all salespeople.
   3. Get the names and ages of salespeople who have an order with customer ***Abernathy***.
   4. Get the names and salaries of the salespeople who have an order with a customer from ***Charlotte***.
   5. Get the names and salaries of the salespeople who have an order with a customer from ***Charlotte***, but also give the name of the customer.
   6. List the salespeople in descending order by age.
   7. List the total sales for each salesperson.
   8. List the total sales from all orders with customer ***Abernathy***.

B. Consider the **`DreamHome`** database from Chapter 1. The following are examples of some of the types of information one might want from this database.

   1. Get full details of all staff with a salary greater than 10,000
   2. Get the just name and date-of-birth of all staff.
   3. Get the name, salary, and branch number of all staff with a salary greater than 10,000
   4. List the names of all cities where there is a branch office but no properties for rent.
   5. List the names and comments of all clients who have viewed a property for rent.
   6. Get the client numbers of everyone who has viewed all properties with three rooms.

The abstract query languages we describe in the following sections are not implemented in any DBMS exactly as we describe them, but ours serve as a foundation for learning and understanding existing query languages, especially SQL. We begin with a discussion of the basic operations and operands of relational algebras.

Section 2. Relational Algebra

In mathematics, an *algebraic system* consists of a set or collection of sets, an assortment of operators on those sets, and fundamental rules (or axioms) those operators must satisfy. If we take as our collection of sets the relations in a relational database, then these together with the operators we shortly define are what we mean as a ***relational algebra***.

The idea behind the relational algebra approach to querying a database is to express the query using only the operators allowed by the relational algebra. Any query expressed in relational algebraic indicates the data that are to be read, processed, and returned in terms of relational algebraic expressions. These expressions would trigger the data access mechanism of the DBMS to process and return the prescribed data in a way which should be opaque to the user (that is, the user is not aware of how the DBMS goes about actually processing the query). Thus, the process of navigating around the internal level of the database to locate the desired data is done automatically by the system, not (normally) by the user.

Basic Operations of a Relational Algebra

E.F. Codd introduced eight basic relational algebra operations for query languages. Each operation is either unary (that is, has one operand) or binary(has two operands) and returns a new relation as its result.

{one or two relations}→ relational algebra operation → result relation

These eight operations can be divided into two groups of four each:

1. ***Traditional set-theoretic operations***: *union*, *intersection*, *difference* and (Cartesian) *product*.

2. ***Special relational operations***: *projection*, *selection*, *joins* (natural join, equijoin, and outer join) and *divide*.

Special Relational Operations-Part I

Before defining our next operation, we introduce the notion of a projection operation for a tuple over an attribute.

Given a relation $R(R_1,...,R_n)$ and an element $r = (r_1,...,r_n) \in R$, we define the R_jth projection of r, $\Pi_{R_j}(r)$, by

$$\Pi_{R_j}(r) = r_j$$

A. *Projection Operation* (denoted Π): Given a relation $R(R_1, ..., R_n)$, we define the **projection of R over $R_{j_1},...,R_{j_p}$**,
where $1 <= j_1 <...< j_p <= n$, as follows

$$\Pi_{R_{j_1}, ..., R_{j_p}}(R) = \{(r_{j_1}, ..., r_{j_p}): \text{ for some } r \in R \text{ with } \Pi_{R_{j_i}}(r) = r_{j_i},\ i=1,...,p\}$$

Example: See Example 5.2 on page 122 of your textbook.

Note, if we relax the condition $1 <= j_1 <...< j_p <= n$, to be merely $1 <= j_1,..., j_p <= n$ then the projection operation can also be used to produce a relation that is the same as a given relation except the order of the attributes has been changed. This is useful for making two relations union-compatible.

Before defining our next operations -- selection -- we again need a preliminary definition.

Let $R(R_1,...,R_n)$ and $S(S_1,...,S_m)$ be two relations. R and S need not be distinct. A *simple comparison* is an expression in either of the following forms:

i. $A\ \theta$ *constant*, where $A \in \{R_1, ..., S_m\}$ and $\theta \in \{=, <, >, <=, >=, \neq\}$.

ii. $A_1\ \theta\ A_2$, where $\{A_1, A_2\} \subseteq \{R_1, ..., R_n, S_1, ..., S_m\}$, $\theta \in \{=, <, >, <=, >=, \neq\}$, and domain($A_1$) and domain($A_2$) are compatible.

A *θ-predicate* is a Boolean expression formed using simple comparisons and any of the Boolean operators: ~ (not), ∧ (conjunction/and}, ∨ (disjunction/or).

We now use the concept of a θ-predicate in the following definitions of the selection and the join operations.

B. *Selection* (σ): Given a relation $R(R_1, ..., R_n)$, let θ be a θ-predicate whose simple comparisons involve only attributes from R or constants. We define the selection operator σ_θ on R with respect to θ as follows:

$$\sigma_\theta = \{r \in R: \sigma_\theta(r) \text{ is true}\}$$

where $\sigma_\theta(r)$ denotes the value of the Boolean expression θ when evaluated using the attribute values of r.

Example: See Example 5.1 on page 122 of your textbook.

Traditional Set-Theoretic Operations

These should be familiar to you from discrete mathematics. All four of these set-theoretic operations are binary operations. The first three of these operations described below further require that their operands be *union-compatible*. That is, given relations $P(P_1, ..., P_n)$ and $Q(Q_1, ..., Q_m)$ with attributes named P_i (i=1,...,n) and Q_j (j=1,...,m) respectively, then P and Q are ***union-compatible*** if and only if

a. n = m

b. dom(P_i) = dom(Q_i), where dom(A) represents the domain of attribute A

Note, it is not necessary that the names of attribute match in order to have union-compatibility; it is only necessary that their associated domains (or perhaps types) match[1].

C. ***Union*** (∪): Given two union-compatible relations P and Q, then the union of P and Q, P ∪ Q, is defined as

$$P \cup Q = \{t: (t \in P) \vee (t \in Q)\}$$

Duplicate tuples are removed.

D. ***Intersection*** (∩): Given two union-compatible relations P and Q, then the intersection of P and Q, P ∩ Q, is defined as

$$P \cap Q = \{t: (t \in P) \wedge (t \in Q)\}$$

E. ***Difference*** (–): Given two union-compatible relations P and Q, then the difference of P and Q, P - Q, is defined as

$$P - Q = \{t: (t \in P) \wedge (t \notin Q)\}$$

F. ***(Cartesian) Product*** (×): Given two relations $P(P_1, ..., P_n)$ and $Q(Q_1, ..., Q_m)$, then the product of P and Q, written P × Q, is defined as

$$P \times Q = \{(p_1, ..., p_n, q_1, ..., q_m): (p_1, ..., p_n) \in P \wedge (q_1, ..., q_m) \in Q\}$$

Examples:See Examples 5.3, 5.4, 5.5, and 5.6 on pages 123-125 of your textbook.

[1] Where domains are not enforced, type compatibility would be required instead.

Special Relational Operations -- Part II

G. *Joins* : Informally, a join operation is way of concatenating (joining) two relations (possibly the same) by forming the Cartesian product of these relations and then following it with a selection on this product. In fact, we shall define several types of join operations. All of them come down to being a Cartesian product followed by a selection, however.

1. *Theta Join* ($|\times|_\theta$): Let $R(R_1, ..., R_n)$ and $S(S_1, ..., S_m)$ be relations and let θ be a θ-predicate in which the only Boolean operations involved are conjunctions ($\wedge$s). We define the *theta join* of R and S over θ, denoted $R\ |\times|_\theta\ S$ as follows:

$$R\ |x|_\theta\ S = \sigma_\theta(R \times S)$$

Example: Consider the relations

EMPLOYEE

ID	Name
101	Jones
103	Smith
104	Lalonde
107	Evans

SALARY

EID	SALARY
101	67
103	55
104	75
107	80

Then the theta join EMPLOYEE $|\times|_{SALARY>70}$ SALARY yields the relation

ID	NAME	EID	SALARY
101	Jones	104	75
101	Jones	107	80
103	Smith	104	75
103	Smith	107	80
104	Lalonde	104	75
104	Lalonde	107	80
107	Evans	104	75
107	Evans	107	80

One may wonder why we are interested in concatenating rows of EMPLOYEE with rows of SALARY that have different values for ID. Usually we are not interested in doing this. Consequently, a more useful special case of the theta join in this situation is the ***equijoin*** in which the only comparison operation permitted is for equality.

2. *Equijoin* ($|\times|_E$): The most commonly used theta join involves join conditions E that only have equality comparisons. Such a join is called an *equijoin*.

Example: Given the EMPLOYEE and SALARY relations from before,

EMPLOYEE $|x|_{ID=EID}$ SALARY

ID	NAME	EID	SALARY
101	Jones	101	67
103	Smith	103	55
104	Lalonde	104	75
107	Evans	107	80

Note, in this example the ID and EID columns are identical. This will arise with every equijoin, although it seems unnecessary to include both columns. The following form of the join operation, known as the *natural join*, removes the redundancy, but requires the redundant attributes to have the same name.

Example: See Example 5.7 on page 127 of your textbook.

3. *Natural join* (*): In the case where two relations have attributes with the same name that represent the same type of attribute, the natural join eliminates the second (redundant) attribute from each such pair that would otherwise result from an equijoin operation. The general specification of a natural join is R*S. There is no need to specify any attributes since it is understood that they will be joining on common names.

   - Normally, in using the term "join" it is the natural join that is intended.

   Example: See Example 5.8 on page 127 of your textbook.

4. *Left outer join* ( ]×|), *right outer join* (|×[ ) and *full outer join*( ]×[ ): In the join operations described above, tuples that do not have a matching "related" value would be eliminated from the result as would any tuple that had a *null* value in any of the attributes being related would be eliminated from the result. If we want to include all tuples in R, or in S, or in both, whether or not they have matching tuples in the other relation, we use an outer join.

   a. The *left outer join* of R and S, R]×| S, keeps every tuple in R. If no matching tuple is found in S then the attributes of S in the result are filled with null values.

   b. The *right outer join* of R and S, R |×[S, keeps every tuple in S. If no matching tuple is found in R then the attributes of R in the result are filled with null values.

   c. The *full outer join* of R and S, R]×[S, keeps every tuple in both R and S. When no matching tuplea are found then the attributes of R or S in the result are filled with null values as necessary.

   Examples: See Example 5.9 on page 128 of your textbook.

H. *Division* (÷) (or *Quotient*): Division is a binary operation that can be defined on two relations where the **entire structure** of one (the divisor) is a portion of the structure of the other (the dividend). It tells us which values in the rows of the dividend appear with all rows of the divisor. Let R and S be relations of degree n and m respectively, where $n > m > 0$. Assume further that $R_i = S_i$ and domain(R_i) = domain(S_i), for i=1,...,m. We define the quotient of R and S, denoted R ÷ S, as follows:

$$R \div S(R_{m+1},...,R_n)=\{(r_{m+1},...,r_n): \text{for every } (s_1,...,s_m)\in S, (s_1,...,s_m,r_{m+1},...,r_n)\in R\}$$

- Note, we have assumed here that the attributes common to both relations appeared in the first m positions of R. For those situations where this is not the case, one need only reorder the positions of the attributes in R to make it so.

Example: Consider the relations

TAKEN

CourseID
101
323
565

CLASS

STID	CourseID	Grade
1000	101	A
1000	214	B
2001	101	C
2001	323	D
3030	101	F
3030	214	A
1000	456	B
3030	323	F
2001	565	D
1000	323	A

1000	565	A
3030	565	F
4444	101	C
4444	323	D

Now without changing the nature of the CLASS relation, we can reorder its attributes to CLASS(CourseID,STID,GRADE). We now determine that

CLASS÷TAKEN

STID	GRADE
1000	A
3030	F

Note, if we had another relation TAKEN2(COURSENO) with the following tuples:

TAKEN2

CourseID
323
565

Then,

CLASS ÷ TAKEN2

STID	GRADE
1000	A
2001	D
3030	F

Example: See **Example** 5.11 on page 130 of your textbook.

More Examples of Queries in Relational Algebra

We complete our discussion of relational algebras by giving some additional examples that show how a number of queries can be expressed using the operations available in a relational algebra. All of these examples will be done in class.

Assume the following set of relations (or relational schema).

```
STUDENT(STID,Name,Major,GradeLevel,Age))
CLASS(Name,Time,Room)
ENROLLMENT(STID,ClassName,Semester,Year,Grade)
```

1. What are the names of all students?
2. Give the details of all students who are mathematics (MATH) majors.
3. Give the names and IDs of all mathematics majors who are juniors.
4. Give the IDs of all students and the courses they took in Spring 2018.
5. Give the names and IDs of all students and the courses they took in Spring 2018.
6. What are the student numbers of all students not yet enrolled in a class?
7. For each student who took a course give the student's id, name, course taken, and grade in that course.
8. What are the names and IDs of students enrolled in the class "BD445"?
9. What are the names and meeting times of the student "PARKS" classes?
10. What are the names and IDs of any students who have taken every class?
11. What are the names and IDs of any students who have taken every MWF8 class?