4. Formal Relational Query Language

Objectives:

1. Relational Algebra
2. Tuple Relational Calculus
3. Domain Relational Calculus

Relational Algebra

The relational algebra is a *procedural* query language. It consists of a set of operations that take one or two relations as input and produce a new relation as their result. The fundamental operations in the relational algebra are *select, project*, *union*, *set difference*, *Cartesian product,* and *rename*. In addition to the fundamental operations, there are several other operations—namely, *set intersection*, *natural join*, and *assignment*.

Select Operation

A **generalized selection** is a unary operation written as $\sigma_{\varphi}(R)$ where φ is a propositional formula that consists of atoms as allowed in the normal selection and the logical operators (and), (or) and (negation). This selection selects all those tuples in R for which φ holds. We allow comparisons using =, _=, <, ≤, >, and ≥ in the selection predicate.

SQL: SELECT * FROM EMPLOYEE WHERE ID=101;
Relational Algebra: $\sigma_{ID=101}(Employee)$

Project Operation

The project operation is a unary operation that returns its argument relation, with certain attributes left out. Since a relation is a set, any duplicate rows are eliminated. Projection is denoted by the uppercase Greek letter pi (∏). We list those attributes that we wish to appear in the result as a subscript to ∏. The argument relation follows in parentheses.
Format: $\Pi_{attributes}(relation)$

SQL: SELECT NAME, DEPT FROM EMPLOYEE WHERE ID=101;
Relational Algebra: $\Pi_{NAME,DEPT}(\sigma_{ID=101}(Employee))$

Rename Operation

A **rename** is a unary operation written $\rho_{a/b}(Relation)$ as where the result is identical to R except that the b attribute in all tuples is renamed to an a attribute. This is simply used to rename the attribute of a relation or the relation itself.

To rename the 'isFriend' attribute to 'isBusinessContact' in a relation, we use $\rho_{isBusinessContact/isFriend}(Relation)$.

Union Operation

Consider a query to find the set of all courses taught in the Fall 2009 semester, the Spring 2010 semester, or both. We need the **union** of two sets; that is, we need all section IDs that appear in either or both of the two relations. We find these data by the binary operation union, denoted, as in set theory, by ∪.
Format: $(Relation1) \cup (Relation2)$

For a union operation $r \cup s$ to be valid, we require that two conditions hold:

1. The relations r and s must be of the same arity. That is, they must have the same number of attributes.
2. The domains of the ith attribute of r and the ith attribute of s must be the same, for all i.

Note that r and s can be either database relations or temporary relations that are the result of relational-algebra expressions.

Intersection Operation

Consider a query to find the set of all courses taught in both the Fall 2009 semester and the Spring 2010 semester. We need the **intersection** of two sets; that is, we need all section IDs that appear in both of the two relation. For a valid intersection operation, conditions are same as union operator.

Format: $(Relation1) \cap (Relation2)$

Set Difference Operation

Consider a query to find the set of all courses taught in the Fall 2009 semester but not in the Spring 2010 semester. We need the **set difference** of two sets; that is, we need all section IDs that appear in only the first relation of the two relation. For a valid set difference operation, conditions are same as union operator.

Format: $(Relation1) - (Relation2)$

Cartesian Product (Cross-Join) Operation

The **Cartesian-product** operation, denoted by a cross (×), allows us to combine information from any two relations. We write the Cartesian product of relations r and s as $r \times s$. The Cartesian product of a set of n-tuples with a set of m-tuples yields a set of "flattened" $(n + m)$-tuples.

$$R \times S := \{(r_1, r_2, \ldots, r_n, s_1, s_2, \ldots, s_m) | (r_1, r_2, \ldots, r_n) \in R, (s_1, s_2, \ldots, s_m) \in S\}$$

The cardinality of the Cartesian product is the product of the cardinalities of its factors, that is, $|R \times S| = |R| \times |S|$.

Natural Join Operation

Natural join ($\bowtie$) is a binary operator that is written as ($R \bowtie S$) where R and S are relations. The result of the natural join is the set of all combinations of tuples in R and S that are equal on their common attribute names. For an example consider the tables *Employee* and *Dept* and their natural join:

Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Mary	1257	Human Resources

Dept

DeptName	Manager
Finance	George
Sales	Harriet
Production	Charles

Employee ⋈ Dept

Name	EmpId	DeptName	Manager
Harry	3415	Finance	George
Sally	2241	Sales	Harriet
George	3401	Finance	George
Harriet	2202	Sales	Harriet

$$R \bowtie S = \{r \cup s \mid r \in R \ \wedge \ s \in S \ \wedge \ Fun(r \cup s)\}$$

Where Fun(t) is a predicate that is true for a relation t (in the mathematical sense) iff t is a function.

It is usually required that R and S must have at least one common attribute, but if this constraint is omitted, and R and S have no common attributes, then the natural join becomes exactly the Cartesian product.

The natural join is arguably one of the most important operators since it is the relational counterpart of logical AND operator. Note that if the same variable appears in each of two predicates that are connected by AND, then that variable stands for the same thing and both appearances must always be substituted by the same value (this is a consequence of the idempotence of the logical AND). In particular, natural join allows the combination of relations that are associated by a foreign key.

Outer Join

Whereas the result of a join (or inner join) consists of tuples formed by combining matching tuples in the two operands, an outer join contains those tuples and additionally some tuples formed by extending an unmatched tuple in one of the operands by "fill" values for each of the attributes of the other operand. Outer joins are not considered part of the classical relational algebra discussed so far.

The operators defined below assume the existence of a *null* value, ω, which we do not define, to be used for the fill values. Three outer join operators are defined: left outer join, right outer join, and full outer join. (The word "outer" is sometimes omitted.)

Left Outer Join

The left outer join is written as $R \bowtie S$ where R and S are relations. The result of the left outer join is the set of all combinations of tuples in R and S that are equal on their common attribute names, in addition (loosely speaking) to tuples in R that have no matching tuples in S.

Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Tim	1123	Executive

Dept

DeptName	Manager
Sales	Harriet
Production	Charles

Employee ⋈ Dept

Name	EmpId	DeptName	Manager
Harry	3415	Finance	ω
Sally	2241	Sales	Harriet
George	3401	Finance	ω
Harriet	2202	Sales	Harriet
Tim	1123	Executive	ω

$R \bowtie S$ in terms of natural join can be defined as follow:

$$(R \bowtie S) \cup ((R - \pi_{r_1, r_2, \ldots, r_n}(R \bowtie S)) \times \{(\omega, \ldots \omega)\})$$

Right Outer Join

The right outer join behaves almost identically to the left outer join, but the roles of the tables are switched.

The right outer join of relations R and S is written as $R \bowtie S$. The result of the right outer join is the set of all combinations of tuples in R and S that are equal on their common attribute names, in addition to tuples in S that have no matching tuples in R.

$R \bowtie S$ in terms of natural join:

$$(R \bowtie S) \cup (\{(\omega, \ldots, \omega)\} \times (S - \pi_{s_1, s_2, \ldots, s_n}(R \bowtie S)))$$

Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Tim	1123	Executive

Dept

DeptName	Manager
Sales	Harriet
Production	Charles

Employee ⋈ Dept

Name	EmpId	DeptName	Manager
Sally	2241	Sales	Harriet
Harriet	2202	Sales	Harriet
ω	ω	Production	Charles

Full Outer Join

The **outer join** or **full outer join** in effect combines the results of the left and right outer joins.

The full outer join is written as $R \bowtie S$ where R and S are relations. The result of the full outer join is the set of all combinations of tuples in R and S that are equal on their common attribute names, in addition to tuples in S that have no matching tuples in R and tuples in R that have no matching tuples in S in their common attribute names.

The full outer join can be simulated using the left and right outer joins (and hence the natural join and set union) as follows:

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

Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Sales
Tim	1123	Executive

Dept

DeptName	Manager
Sales	Harriet
Production	Charles

Employee ⋈ Dept

Name	EmpId	DeptName	Manager
Harry	3415	Finance	ω
Sally	2241	Sales	Harriet
George	3401	Finance	ω
Harriet	2202	Sales	Harriet
Tim	1123	Executive	ω
ω	ω	Production	Charles

Aggregation

The second extended relational-algebra operation is the aggregate operation $\mathcal{G}$, which permits the use of aggregate functions such as min or average, on sets of values.
Aggregate functions take a collection of values and return a single value as a result. For example, the aggregate function **sum** takes a collection of values and returns the sum of the values.

Format: $_{group\ by}\mathcal{G}_{function(parameters)}(Relation)$

Example: Find the average salary in each department
$_{department\_name}\mathcal{G}_{average(salary)}(instructor)$

General Form:

$$_{G_1,G_2,\ldots,G_n}\mathcal{G}_{F_1(A_1),\ F_2(A_2),\ldots,\ F_m(A_m)}(E)$$

Tuple Relational Calculus (TRC)

When we write a relational-algebra expression, we provide a sequence of procedures that generates the answer to our query. The tuple relational calculus, by contrast, is a **nonprocedural** query language. It describes the desired information without giving a specific procedure for obtaining that information.
A query in the tuple relational calculus is expressed as: $\{t \mid P(t)\}$,
where t = resulting tuples,
P(t) = known as Predicate and these are the conditions that are used to fetch t
Thus, it generates set of all tuples t, such that Predicate P(t) is true for t.

P(t) may have various conditions logically combined with OR (∨), AND (∧), NOT (¬).

It also uses quantifiers:
∃ t ∈ r (Q(t)) = "there exists" a tuple in t in relation r such that predicate Q(t) is true.
∀ t ∈ r (Q(t)) = Q(t) is true "for all" tuples in relation r.

Following our earlier notation, we use $t[A]$ to denote the value of tuple t on attribute A, and we use $t \in r$ to denote that tuple t is in relation r.

Examples

1. Fetch all data from instructors whose salary is greater than $80,000:
   $\{t \mid t \in instructor \wedge t[salary] > 80000\}$

2. Suppose that we want only the *ID* attribute, rather than all attributes of the *instructor* relation. To write this query in the tuple relational calculus, we need to write an expression for a relation on the schema (*ID*). We need those tuples on (*ID*) such that there is a tuple in *instructor* with the *salary* attribute > 80000. To express this request, we need the construct "there exists" from mathematical logic. The notation:

   $\exists\ t \in r\ (Q(t))$ means "there exists a tuple t in relation r such that predicate $Q(t)$ is true."

   "Find the instructor *ID* for each instructor with a salary greater than $80,000" as:
   $\{t \mid \exists\ s \in instructor\ (t[ID] = s[ID] \wedge s[salary] > 80000)\}$

3. Consider the query "Find the names of all instructors whose department is in the Watson building." This query is slightly more complex than the previous queries, since it involves two relations: *instructor* and *department*. As we shall see, however, all it requires is that we have two "there exists" clauses in our tuple-relational-calculus expression, connected by *and* (∧). We write the query as follows:

   $\{t \mid \exists\ s \in instructor\ (t[name] = s[name] \wedge \exists\ u \in department\ (u[dept\ name] = s[dept\ name] \wedge u[building] = \text{"Watson"}))\}$

4. To find the set of all courses taught in the Fall 2009 semester, the Spring 2010 semester, or both, we used the union operation in the relational algebra. In the tuple relational calculus, we shall need two "there exists" clauses, connected by *or* (∨):

   $\{t \mid \exists\ s \in section\ (t[course\ id] = s[course\ id]) \wedge s[semester] = \text{"Fall"} \wedge s[year] = 2009)$
   $\vee\ \exists\ u \in section\ (u[course\ id] = t[course\ id]) \wedge u[semester] = \text{"Spring"} \wedge u[year] = 2010)\}$

5. Now consider the query "Find all the courses taught in the Fall 2009 semester but not in Spring 2010 semester." The tuple-relational-calculus expression for this query is similar to the expressions that we have just seen, except for the use of the *not* ($\neg$) symbol:

   $\{t \mid \exists\, s \in section\ (t[course\ id\ ] = s[course\ id\ ]) \wedge s[semester] = \text{"Fall"} \wedge s[year] = 2009)$
   $\wedge\ \neg \exists\, u \in section\ (u[course\ id\ ] = t[course\ id\ ]) \wedge u[semester] = \text{"Spring"} \wedge u[year] = 2010)\}$

6. Consider the query that "Find all students who have taken all courses offered in the Biology department." To write this query in the tuple relational calculus, we introduce the "for all" construct, denoted by $\forall$. The notation: $\forall\ t \in r\ (Q(t))$
   means "Q is true for all tuples t in relation r."

   We write the expression for our query as follows:
   $\{t \mid \exists\, r \in student\ (r\ [ID] = t[ID]) \wedge (\forall\ u \in course\ (u[dept\ name] = \text{" Biology"} \Rightarrow \exists\, s \in takes\ (t[ID] = s[ID] \wedge s[course\ id\ ] = u[course\ id\ ]))\}$

Formal Definition

A tuple-relational-calculus expression is of the form: $\{t|P(t)\}$
Where P is a *formula*. Several tuple variables may appear in a formula. A tuple variable is said to be a *free variable* unless it is quantified by a $\exists$ or $\forall$.

Thus, in: $t \in instructor \wedge \exists s \in department(t[dept\ name] = s[dept\ name])$, t is a free variable. Tuple variable s is said to be a *bound* variable.

A tuple-relational-calculus formula is built up out of *atoms*. An atom has one of the following forms:

- $s \in r$, where s is a tuple variable and r is a relation (we do not allow use of the / $\in$ operator).
- $s[x]\ \Theta\ u[y]$, where s and u are tuple variables, x is an attribute on which s is defined, y is an attribute on which u is defined, and Θ is a comparison operator ($<, \leq, =, \neq, >, \geq$); we require that attributes x and y have domains whose members can be compared by Θ.
- $s[x]\ \Theta\ c$, where s is a tuple variable, x is an attribute on which s is defined, Θ is a comparison operator, and c is a constant in the domain of attribute x.

We build up formulae from atoms by using the following rules:

- An atom is a formula.
- If $P1$ is a formula, then so are $\neg P1$ and $(P1)$.
- If $P1$ and $P2$ are formulae, then so are $P1 \vee P2$, $P1 \wedge P2$, and $P1 \Rightarrow P2$.
- If $P1(s)$ is a formula containing a free tuple variable s, and r is a relation, then $\exists\, s \in r\ (P1(s))$ and $\forall\ s \in r\ (P1(s))$ are also formulae.

Safety of Expressions

A tuple-relational-calculus expression may generate an infinite relation. Suppose that we write the expression: $\{t \mid \neg\ (t \in instructor)\}$

There are infinitely many tuples that are not in *instructor*. Most of these tuples contain values that do not even appear in the database! Clearly, we do not wish to allow such expressions. To

help us define a restriction of the tuple relational calculus, we introduce the concept of the domain of a tuple relational formula, P. Intuitively, the domain of P, denoted $dom(P)$, is the set of all values referenced by P. They include values mentioned in P itself, as well as values that appear in a tuple of a relation mentioned in P. Thus, the domain of P is the set of all values that appear explicitly in P or that appear in one or more relations whose names appear in P.

We say that an expression $\{t \mid P(t)\}$ is *safe* if all values that appear in the result are values from $dom(P)$. The expression $\{t \mid \neg (t \in instructor)\}$ is not safe.

Domain Relational Calculus (DRC)

Domain Relational Calculus is a non-procedural query language equivalent in power to Tuple Relational Calculus. Domain Relational Calculus provides only the description of the query but it does not provide the methods to solve it.

It uses *domain* variables that take on values from an attribute domain, rather than values for an entire tuple. The domain relational calculus, however, is closely related to the tuple relational calculus. Domain relational calculus serves as the theoretical basis of the widely used QBE language, just as relational algebra serves as the basis for the SQL language.

Formal Definition

An expression in the domain relational calculus is of the form

$$\{< x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n)\}$$

where $x_1, x_2, \ldots, x_n$ represent domain variables. P represents a formula composed of atoms, as was the case in the tuple relational calculus.

An atom in the domain relational calculus has one of the following forms:

- $< x_1, x_2, \ldots, x_n > \in r$, where r is a relation on n attributes and $x_1, x_2, \ldots, x_n$ are domain variables or domain constants.
- $x \; \Theta \; y$, where x and y are domain variables and Θ is a comparison operator ($<$, $\leq$, $=$, $\neq$, $>$, $\geq$). We require that attributes x and y have domains that can be compared by Θ.
- $x \; \Theta \; c$, where x is a domain variable, Θ is a comparison operator, and c is a constant in the domain of the attribute for which x is a domain variable.

We build up formulae from atoms by using the following rules:

- An atom is a formula.
- If $P1$ is a formula, then so are $\neg P1$ and $(P1)$.
- If $P1$ and $P2$ are formulae, then so are $P1 \vee P2$, $P1 \wedge P2$, and $P1 \Rightarrow P2$.
- If $P1(x)$ is a formula in x, where x is a free domain variable, then $\exists \; x \; (P1(x))$ and $\forall \; x \; (P1(x))$ are also formulae.

As a notational shorthand, we write $\exists \; a, b, c \; (P(a, b, c))$ for $\exists \; a \; (\exists \; b \; (\exists \; c \; (P(a, b, c))))$.

Note: The domain variables those will be in resulting relation must appear before | within $\prec$ and $\succ$ and all the domain variables must appear in which order they are in original relation or table.

Examples

1. Find the names of all instructors in the Physics department together with the *course id* of all courses they teach:
   $\{< n, c > \mid \exists\, i, a\ (< i, c, a, s, y > \in teaches \wedge \exists\, d, s\ (< i, n, d, s > \in instructor \wedge d =$ "Physics"))$\}$

2. Find the set of all courses taught in the Fall 2009 semester, the Spring 2010 semester, or both:
   $\{< c > \mid \exists\, s\ (< c, a, s, y, b, r, t > \in section \wedge s =$ "Fall" $\wedge\ y =$ "2009" $\vee \exists u\ (< c, a, s, y, b, r, t > \in section \wedge s =$ "Spring" $\wedge\ y =$ "2010"

3. Find all students who have taken all courses offered in the Biology department:
   $\{< i > \mid \exists\, n, d, t\ (< i, n, d, t > \in student) \wedge \forall\, x, y, z,w\ (< x, y, z,w > \in course \wedge z =$ "Biology" $\Rightarrow \exists\, a, b\ (< a, x, b, r, p, q > \in takes \wedge < c, a > \in depositor\ ))\}$

Safety of Expressions

In the tuple relational calculus, we restricted any existentially quantified variable to range over a specific relation. Since we did not do so in the domain calculus, we add rules to the definition of safety to deal with cases like our example. We say that an expression $\{< x_1, x_2, \ldots, x_n > \mid P\ (x_1, x_2, \ldots, x_n)\}$ is safe if all of the following hold:

1. All values that appear in tuples of the expression are values from $dom(P)$.
2. For every "there exists" sub-formula of the form $\exists\, x\ (P1(x))$, the sub-formula is true if and only if there is a value x in $dom(P1)$ such that $P1(x)$ is true.
3. For every "for all" sub-formula of the form $\forall x\ (P1(x))$, the sub-formula is true if and only if $P1(x)$ is true for all values x from $dom(P1)$.

The purpose of the additional rules is to ensure that we can test "for all" and "there exists" sub-formulae without having to test infinitely many possibilities.

The basic relational algebra is a procedural language that is equivalent in power to both forms of the relational calculus when they are restricted to safe expressions.