Relational Model

6 Relational algebra

40. What is and why do we study relational algebra. Recall that Codd's maxim was to represent all data in the form of tables, and that querying a relational database can be seen as a process which computes an output table from some given tables. As a language for specifying such a computation we have introduced SQL. This remains the most widely used formalism for this purpose but it is not the most convenient one from either an application programmer's point of view, nor from the point of view of a person who wants to make general statements about the shape and manipulation of tables.

What we are seeking is a language of a few *basic operators*, which can be combined in the style of ordinary algebra, and which are sufficient to express all queries. Algebra for numbers is our guiding example, and it indeed combines conciseness with expressivity in a beautiful way. For example, we can write

$$y = \sqrt{a^2 + b^2}$$

in Mathematics, or

```
y = sqrt(a*a + b*b);
```

in Java, to express the operation which first squares a and b, adds together the results and takes the square root. Relational algebra provides the analogue of this for tables.

41. Relation schemas Tables (or relations) form the domain of interest for relational algebra, just like numbers form the domain of interest for high school algebra. Unlike numbers, tables come in various shapes. To distinguish them, we use the notation of *schemas*. A schema for a relation is simply its name followed by the names of all the attributes stored in columns, written using the notation:

$$R(A, B, C, D, \ldots)$$

For example, the schemas for the "`fundamentals`" database are written as follows:

```
staff(sid, title, firstname, lastname, email, office, phone)
courses(level, cid, name, credits, semester, bc)
period(semester, description)
lecturing(cid, sid, year, numbers)
allmarks03(student, mark, bc)
```

The shape of the possible tables are constrained by the schemas. For instance, any "`staff`" table must have exactly 7 columns and these columns must contain values for the specified attributes exactly in that order.

42. The basic relational algebra operators. For our purposes it is (more than) sufficient to consider the following four operators:

Selection This is the process which filters from a table those rows which match a given condition. If we let C stand for the condition under consideration, and T for the input table, then the syntax for selecting all those rows of T which satisfy C is

$$\sigma_C(T) \ .$$

(The Greek character used here, σ, is called "sigma", similar to "s" in our alphabet.) In SQL the condition for selection is placed in the "`WHERE`" part.

Projection This is the process of erasing all columns from the table except those explicitly mentioned in the projection operator. If we let X stand for a subset of attributes and again T for the input table, then the syntax for "narrowing down" the table to the attributes mentioned in X is

$$\pi_X(T) \ .$$

(The Greek character used for this, π, is pronounced "pi", similar to "p" in our alphabet.) As we have seen in the examples as at the beginning of the course, projecting a table can result in many rows being identical (e.g., projecting on "`title`" in the table "`staff`"). It is an assumption of relational algebra that rows are never repeated, in other words, duplicates are silently removed. In SQL projection happens as a result of mentioning the attributes contained in X in the "`SELECT`" part.

Renaming This is the process which allows us to give new names the attributes stored in its columns. Suppose T is a table with attributes $A_1, \ldots, A_n$. To rename its attributes to new names $B_1, \ldots, B_n$, we write

$$\rho_{(B_1,\ldots,B_n)}(T)$$

(The Greek character used here, ρ, is pronounced "rho", similar to "r" in our alphabet.) In SQL, there is no explicit renaming operator for attributes, but we have it available implicitly via the dot notation. When we write "`lecturing.sid`" for the "`sid`" attribute of the "`lecturing`" table, we have essentially renamed "`sid`" to the composite name "`lecturing.sid`".

Natural join This is the process which combines two tables into one, along the common attributes. Suppose we have two tables T_1 and T_2. We denote their natural join as:

$$T_1 \bowtie T_2 \ .$$

(We pronounce the symbol $\bowtie$ as just "join".) The resulting table will have columns for each attribute of T_1 and each attribute of T_2. Attributes which are common to both are *mentioned only once* in $T_1 \bowtie T_2$. The rows of the resulting table are composed of a row of T_1 together with a row of T_2, provided that they have the same value in all common attributes.

Basic SQL does not have a natural join operation but the operation can be simulated by listing T_1 and T_2 in the "`FROM`" part, and by including the conditions "`T_1.x=T_2.x`" for every common attribute x in the "`WHERE`" part.[1]

Other operators It is easy to add to this collection the usual set-theoretic operators $\cup$ (meaning "union"), $\cap$ (meaning "intersection"), and $-$ (meaning "set difference"). They are only applicable if the tables involved share the same set of attributes. In order to describe the process of aggregation one needs yet further operators. Although these are being used to represent queries inside a database management system, we will not pursue this topic further.

If you are interested in how PostgreSQL executes a query, then precede it with the keyword "`EXPLAIN`". Instead of actually running the query, PostgreSQL will output a verbose form of the corresponding relational algebra expression.

43. Some example queries expressed in relational algebra. Consider

```
SELECT lastname
FROM staff
WHERE title='Prof';
```

In relational algebra this would be expressed as $\pi_{\texttt{lastname}}(\sigma_{\texttt{title='Prof'}}(\texttt{staff}))$.
The order of evaluation is the same as in ordinary algebra: first select the rows where the `title` field contains the value `'Prof'`, then narrow the table down to the `lastname` field alone. The query

[1]NATURAL JOIN has been added to the SQL standard in 2011. However, it may not be supported on all implementations of SQL.

```
SELECT s.name, l.cid
FROM staff AS s, lecturing AS l
WHERE s.sid = l.sid;
```

becomes $\pi_{\texttt{name,cid}}(\texttt{staff} \bowtie \texttt{lecturing})$.
As another example, consider

```
SELECT s.lastname, c.name
FROM staff AS s, lecturing AS l, courses AS c
WHERE s.sid = l.sid AND l.cid = c.cid;
```

This becomes simply $\pi_{\texttt{lastname,name}}(\texttt{staff} \bowtie \texttt{lecturing} \bowtie \texttt{courses})$.
As a last example, consider

```
SELECT l2.cid
FROM lecturing AS l1, lecturing AS l2
WHERE l1.cid = l2.cid AND
      l1.year = 2002 AND l2.year = 2003 AND
      l1.sid <> l2.sid;
```

By renaming "`lecturing`" to "`l1`" in SQL, we are effectively renaming its attributes to "`l1.cid, l1.sid, l1.year, l1.numbers`". Similarly for the renaming of the second copy of "`lecturing`" to "`l2`". But we want to join the two copies of "`lecturing`" on the column "`cid`". To achieve this effect via the natural join operator, we refrain from renaming the "`cid`" column. Notice this expression:

$$L = \rho_{(\texttt{cid,l1.sid,l1.year,l1.numbers})}(\texttt{lecturing}) \bowtie \rho_{(\texttt{cid,l2.sid,l2.year,l2.numbers})}(\texttt{lecturing})$$

Since the only common column in the renamed tables is "`cid`", the natural join matches the rows on this attribute. From this table, we can obtain the desired result by selection and projection:

$$\pi_{\texttt{cid}}(\sigma_{\texttt{l1.year}=2002 \wedge \texttt{l2.year}=2003 \wedge \texttt{l1.sid} \neq \texttt{l2.sid}}(L))$$

44. More on natural join. The third example illustrates an important property of the natural join operation, namely, that it is *associative*. In other words, it is true that

$$(T_1 \bowtie T_2) \bowtie T_3 = T_1 \bowtie (T_2 \bowtie T_3) .$$

In this way, natural join behaves like addition and multiplication for numbers. (As an aside, there are, of course, also non-associative operations in arithmetic; just think of subtraction or division.)
If we join together two tables which have no attributes in common, then *every row* of the first table will be combined with every row of the second table. The result will be very big; indeed, if the first table has n rows, and the second has m rows, then the result will have $n \times m$ rows. (This result is also referred to as the "cross product" or "cross join" of the two tables.) In practice, such a natural join is rarely needed; usually, the common attribute is the primary key of one of the tables. In that case, the result will have at most as many rows as the table where the attribute plays the role of foreign key.

Pause. Do you believe this?

The other extreme is the case where T_1 and T_2 have exactly the same attributes, and so by the join condition we can only bring rows together which agree in *all* columns. In this case the natural join degenerates to a simple intersection. This fact is of no practical importance.

45. Exercises.

a. Explain each of the following relational algebra expressions in plain English, and translate them into SQL.

   i. $\pi_{\texttt{lastname}}(\sigma_{\texttt{firstname}='\text{John}'}(\texttt{staff}))$
   ii. $\pi_{\texttt{lastname}}(\sigma_{\texttt{numbers}>100}(\texttt{staff} \bowtie \texttt{lecturing}))$

b. Translate the following SQL queries into relational algebra.

   i.
```
SELECT c.name
FROM courses c, staff s, lecturing l
WHERE l.sid=s.sid AND l.cid=c.cid AND
      (l.year=1999 OR l.year=2000)
      AND s.lastname='Jung';
```

ii.
```
SELECT c.name FROM lecturing l, courses c
WHERE l.cid=c.cid and l.year=2001
EXCEPT
SELECT c.name FROM lecturing l, courses c
WHERE l.cid=c.cid AND (l.year=2000 OR l.year=1999);
```

c. Like Exercise a:

i. $\pi_{\mathtt{name}}(\sigma_{\mathtt{numbers}>100}(\mathtt{lecturing} \bowtie \mathtt{courses})) - \pi_{\mathtt{name}}(\sigma_{\mathtt{level}=1}(\mathtt{courses}))$

ii. $\pi_{\{\mathtt{lastname},\mathtt{name}\}}(\sigma_{\mathtt{year}=1999}(\mathtt{staff} \bowtie \mathtt{lecturing} \bowtie \mathtt{courses})) \cap$
$\pi_{\{\mathtt{lastname},\mathtt{name}\}}(\sigma_{\mathtt{level}=2}(\mathtt{staff} \bowtie \mathtt{lecturing} \bowtie \mathtt{courses}))$

d. Like Exercise b:

i.
```
SELECT c.name
FROM lecturing l1, lecturing l2, courses c
WHERE l1.cid=l2.cid AND l1.cid=c.cid AND
      l1.sid=l2.sid AND
      l1.year=1999 AND l2.year=2000;
```

ii.
```
SELECT s.lastname
FROM staff s
WHERE s.sid NOT IN (SELECT l.sid
                    FROM lecturing l);
```

7 Join operators in SQL

46. Natural Join in FROM clauses. SQL allows us to write a natural join expression involving one or more tables in the "FROM" clause of a "SELECT" statement. For example, the query

```
SELECT lecturing.cid
FROM staff NATURAL JOIN lecturing
WHERE staff.firstname = 'Achim' AND
      staff.lastname = 'Jung';
```

Since "sid" is the only common column in the "staff" and "lecturing" tables, this query has exactly the same effect as the following variant:

```
SELECT lecturing.cid
FROM staff, lecturing
WHERE staff.firstname = 'Achim' AND
      staff.lastname = 'Jung' AND
      staff.sid = lecturing.sid;
```

So, using the "NATURAL JOIN" in "SELECT" clauses is merely for convenience. It does not allow you to write any new queries that you could not have written without them. Consider the following example, where two JOIN operators are combined:

```
SELECT courses.name
FROM staff NATURAL JOIN lecturing NATURAL JOIN courses
WHERE staff.firstname = 'Achim' AND
      staff.lastname = 'Jung';
```

47. Outer Joins. The normal join operator is called "inner join". Another join operator called "outer join" differs from it in a subtle way, and is found to be quite useful.

Inner join picks up rows from the two tables that have *matching values* of the common attributes. If a row in one table fails to match any row in the other table on the common attributes, then that row does not appear in the result table. We call the row that do not match any row in the other table as "dangling rows". In some situations, dangling rows can produce anomalous results.

Suppose we want a query to list all the lecturers along with the number of courses taught by them in a particular year, say 2004. We might consider writing

```
SELECT staff.firstname, staff.lastname, count(cid)
FROM staff NATURAL JOIN lecturing
WHERE year = 2004
GROUP BY staff.sid, staff.firstname, staff.lastname;
```

If a lecturer did not teach any course in 2004, e.g., Gavin Brown, we might expect that Gavin Brown's name would appear in the output with a count of 0. However, his name does not appear at all. The reason is that, since Gavin Brown did not teach any course in 2004, there is no tuple in the "`lecturing`" table with his "`sid`" for the year 2004. So his "`staff`" tuple becomes a dangling tuple in the inner join.
The "outer join" operator has been introduced to solve this problem. If we modify the query to:

```
SELECT staff.firstname, staff.lastname, count(cid)
FROM staff NATURAL LEFT OUTER JOIN lecturing
WHERE year = 2004
GROUP BY staff.sid, staff.firstname, staff.lastname;
```

then all the rows of the "`staff`" table that would otherwise be dangling are added to the result table along with null values for the remaining attributes. In particular, for a lecturer that did not teach any course in 2004, there would be a tuple with a null value for the "`cid`" attribute. So, the count of the "`cid`" values produces 0 for that lecturer.
Similarly, "`RIGHT OUTER JOIN`" adds result rows for all the dangling rows of the right hand side argument, and "`FULL OUTER JOIN`" adds result rows for dangling rows of either argument.

48. Non-natural joins. The term "natural" in joins refers to the fact that all the common columns of the two tables are matched during the join. SQL also provides syntax for "non-natural" joins, where

- *fewer* than the common columns may be matched, and
- *more* than the common columns may be matched.

The case of "fewer columns" means that some, but not all, of the common columns may be matched. This case arises often when we join two copies of the same table because both the copies have the same set of columns and, so, all the columns would be common. You can specify the columns that you want to match in the join via a "`USING`" modifier. Here is an example query that asks which courses changed lecturers from 2003 to 2004 (assuming there were no multiple lecturers):

```
SELECT courses.name
FROM lecturing AS lect2003 JOIN lecturing AS lect2004 USING (cid)
     NATURAL JOIN courses
WHERE lect2003.sid <> lect2004.sid
```

By specifying `USING (cid)` we are asking SQL to match the `lect2003` and `lect2004` tables only using the `cid` column. All the other columns, even though they are common to both the copies of `lecturing`, will *not* be matched.
The case of "more columns" means that columns that are not common, going by their column names, may be used for matching. For example, the `allmarks` tables use the name `student` for the registration numbers of students. Suppose we have a `students` table, where the registration number is given by the column `regno`. Then we can join them using an `ON` condition along with a `JOIN` operator.

```
SELECT students.firstname, students.lastname
FROM allmarks JOIN students ON allmarks.student = students.regno
WHERE ...
```

Without the `ON` condition, the join would reduce to a cross product because there are no common columns between the `allmarks` and `students` tables.

49. Exercise. Use outer joins to solve the problem of finding the percentage of first class marks in the year 2003. How does this differ from the solution using inner joins?

50. Exercise. Use outer joins to solve this problem: Show, for every course in 2003, the average mark and the average mark of the students who achieved a pass mark.