RELATIONAL DATABASES
2. Relational algebra

Carmelo Vaccaro

University of Paris-Saclay

Master 1 - AI
2022/23: first semester

The content of these slides is taken from the book
Database Systems - The Complete Book,
by Hector Garcia-Molina, Jeffrey D. Ullman and Jennifer Widom,
published by Pearson, 2014.

1. Relational algebra

Relational algebra 1/2

Relational algebra consists of some simple but powerful ways to construct new relations from given relations. When the given relations are stored data, then the constructed relations can be answers to queries about this data.

Relational algebra is not used today as a query language in commercial DBMS's (some of the early prototypes used this algebra directly).

But SQL incorporates relational algebra at its center, and when a DBMS processes queries, the first thing that happens to a SQL query is that it gets translated into relational algebra or a very similar internal representation.

Relational algebra 2/2

Computing with relational algebra is less powerful than computing with a programming language like Java or C, thus there are some computations that can be done with a usual programming language but not with relation algebra.

But relation algebra has two advantages: it is simpler than programming languages (making the same computation with relation algebra demands less effort to the programmer than with Java or C) and produces highly optimized code.

Since the size of data to process tends to become bigger and bigger it is crucial to have a tool that makes calculations in an efficient way.

Operators and operands of a general algebra

An algebra consists of operators and atomic operands and allows to build *expressions* by applying operators to atomic operands and/or other expressions of the algebra.

For example in arithmetic we have expressions such as *(x+y) * z* or *(x + 7)/(y - 3) + x*. The operators are +, -, *, / and the atomic operands are numeric constants and variables.

Operators and operands of relational algebra 1/2

For relational algebra the atomic operands are variables that stand for relations and constants, which are finite relations.

The operations of relational algebra fall into four broad classes:

Operators and operands of relational algebra 2/2

1. The usual set operations — union, intersection, and difference — applied to relations.
2. Operations that remove parts of a relation: "selection" eliminates some rows (tuples), and "projection" eliminates some columns.
3. Operations that combine the tuples of two relations, including "Cartesian product," which pairs the tuples of two relations in all possible ways, and various kinds of "join" operations, which selectively pair tuples from two relations.
4. An operation called "renaming" that does not affect the tuples of a relation, but changes the relation schema, i.e., the names of the attributes and/or the name of the relation itself.

Set operations on relations

Let R and S be two relations. We can apply set operations to them only if the following two conditions are verified:

1. R and S must have schemas with identical sets of attributes, and the types (domains) for each attribute must be the same in R and S.
2. the columns of R and S must be ordered so that the order of attributes is the same for both relations.

In order to apply set operations to relations that have the same number of attributes, with corresponding domains, but that use different names for their attributes we may use the renaming operator to be discussed later.

Set operations on relations: example

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

Figure: The relation R

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

Figure: The relation S

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88
Harrison Ford	789 Palm Dr., Beverly Hills	M	7/7/77

Figure: The relation $R \cup S$

name	*address*	*gender*	*birthdate*
Carrie Fisher	123 Maple St., Hollywood	F	9/9/99

Figure: The relation $R \cap S$

name	*address*	*gender*	*birthdate*
Mark Hamill	456 Oak Rd., Brentwood	M	8/8/88

Figure: The relation $R \setminus S$

Projection

The *projection* operator is used to produce from a relation R a new relation that has only some of R's columns.

The projection operator is denoted π. If $A_1, A_2, \cdots, A_n$ are attributes of R then the value of the expression $\pi_{A_1,A_2,\cdots,A_n}(R)$ is a relation that has only the columns for attributes $A_1, A_2, \cdots, A_n$.

Therefore applying the projection operator to a relation R produces a relation with a schema different than that of R.

Projection: example

Suppose that the relation *Movies* is the following:

title	*year*	*length*	*genre*	*studioName*	*producerC#*
Star Wars	1977	124	sciFi	Fox	12345
Galaxy Quest	1999	104	comedy	DreamWorks	67890
Wayne's World	1992	95	comedy	Paramount	99999

Then $\pi_{title,year,length}(Movies)$ is

title	*year*	*length*
Star Wars	1977	124
Galaxy Quest	1999	104
Wayne's World	1992	95

and $\pi_{genre}(Movies)$ is

genre
sciFi
comedy

Selection

The *selection* operator, applied to a relation R, produces a new relation with a subset of R's tuples.

The selection operator is denoted σ. If C is a condition involving the attributes of R, then $\sigma_C(R)$ is the subset of tuples of R that satisfy C.

The schema for $\sigma_C(R)$ is the same as that of R.

Selection: example

Let the relation *Movies* be as in the previous example. Then $\sigma_{length \geq 100}(R)$ is the following

title	*year*	*length*	*genre*	*studioName*	*producerC#*
Star Wars	1977	124	sciFi	Fox	12345
Galaxy Quest	1999	104	comedy	DreamWorks	67890

and $\sigma_{length \geq 100 \ AND \ studioName = 'Fox'}(R)$ is the following

title	*year*	*length*	*genre*	*studioName*	*producerC#*
Star Wars	1977	124	sciFi	Fox	12345

Cartesian product 1/2

The *Cartesian product* (or *cross-product*) of two sets R and S is the set of pairs that can be formed by choosing the first element of the pair to be any element of R and the second any element of S. This product is denoted $R \times S$.

If R and S are relations, the result of pairing a tuple from R with a tuple from S is a longer tuple, with one component for each of the components of the constituent tuples.

By convention, the components from R (the left operand) precede the components from S in the attribute order for the result.

Cartesian product 2/2

The relation schema for $R \times S$ is the union of the schemas for R and S.

However, if R and S have some attributes with the same name, then we need to invent new names for at least one of each pair of these attributes.

To disambiguate an attribute A that is in the schemas of both R and S, we use $R.A$ for the attribute from R and $S.A$ for the attribute from S.

Cartesian product: example

Let R and S be the following relations:

A	B
1	2
3	4

(a) Relation R

B	C	D
2	5	6
4	7	8
9	10	11

(b) Relation S

Then $R \times S$ is the following

A	$R.B$	$S.B$	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

We have used $R.B$ and $S.B$ for B.

Natural joins

The *natural join* of two relations R and S, denoted $R \bowtie S$, is the relation obtained by pairing only those tuples from R and S that agree in whatever attributes are common to the schemas of R and S.

More precisely, let $A_1, A_2, \cdots, A_n$ be all the attributes that are in both the schema of R and the schema of S. Then a tuple r from R and a tuple s from S are successfully paired if and only if r and s agree on each of the attributes $A_1, A_2, \cdots, A_n$.

Natural joins: example 1

Let R and S be the following relations:

A	B
1	2
3	4

(a) Relation R

B	C	D
2	5	6
4	7	8
9	10	11

(b) Relation S

Then $R \bowtie S$ is

A	B	C	D
1	2	5	6
3	4	7	8

Natural joins: example 2

Let U and V be the following relations:

A	B	C
1	2	3
6	7	8
9	7	8

(a) Relation U

B	C	D
2	3	4
2	3	5
7	8	10

(b) Relation V

Then $U \bowtie V$ is

A	B	C	D
1	2	3	4
1	2	3	5
6	7	8	10
9	7	8	10

(c) Result $U \bowtie V$

Theta-joins

The *theta-join* pairs tuples from two relations based on some condition C and is denoted $R \bowtie_C S$.

The result of theta-join is constructed as follows:

1. take $R \times S$, the Cartesian product of R and S;
2. select from $R \times S$ only those tuples that satisfy the condition C.

A theta-join is the composition of Cartesian product with selection. The schema for the theta-join is the same as that of the Cartesian product.

Theta-joins: example

Let U and V be the following relations:

A	B	C
1	2	3
6	7	8
9	7	8

(a) Relation U

B	C	D
2	3	4
2	3	5
7	8	10

(b) Relation V

Then $U \bowtie_{A<D} V$ is

A	$U.B$	$U.C$	$V.B$	$V.C$	D
1	2	3	2	3	4
1	2	3	2	3	5
1	2	3	7	8	10
6	7	8	7	8	10
9	7	8	7	8	10

and $U \bowtie_{(A<D \text{ AND } U.B \neq V.B)} V$ is

A	$U.B$	$U.C$	$V.B$	$V.C$	D
1	2	3	7	8	10

Combining operations to form queries

Relational algebra allows us to form expressions of arbitrary complexity by applying operations to the result of other operations.

We can construct expressions of relational algebra by applying operators to subexpressions, using parentheses when necessary to indicate grouping of operands.

Combining operations to form queries: example

Example. Suppose we want to know for the *Movies* relation, "What are the titles and years of movies made by Fox that are at least 100 minutes long?" We could answer the query by doing the following:

1. select those *Movies* tuples that have length ≥ 100;
2. select those *Movies* tuples that have *studioName* = *'Fox'*;
3. compute the intersection of (1) and (2);
4. project the relation from (3) onto attributes *title* and *year*.

The same expression in conventional notation is

$$\pi_{title,\, year}\left(\sigma_{length \geq 100}(Movies) \cap \sigma_{studioName = 'Fox'}(Movies)\right)$$

and is equivalent to the simpler

$$\pi_{title,\, year}\left(\sigma_{(length \geq 100 \ AND \ studioName = 'Fox')}(Movies)\right).$$

Naming and renaming

The renaming operator, denoted ρ allows to change the name of a relation and/or of the attributes.

The operation to rename a relation named R into S is denoted $\rho_S(R)$. If R has n attributes, the operation to rename it into S and its attributes into $A_1, \cdots, A_n$ is denoted $\rho_{S(A_1,\cdots,A_n)}(R)$

2. Constraints on relations

Referential integrity constraints

A referential integrity constraint asserts that a value appearing in one context also appears in another, related context.

For example, in the movies database, we expect that a person appearing in the *starName* component appears as the name of some star in the *MovieStar* relation.

Referential integrity constraints: example 1

Consider the two relations from the movie database:

$$Movies(title, year, length, genre, studioName, producerC\#)$$
$$MovieExec(name, address, cert\#, netWorth)$$

We expect that *producerC#* component of each *Movies* tuple must also appear in the *cert#* component of some *MovieExec* tuple.

We can express this constraint by the set-containment

$$\pi_{producerC\#}(Movies) \subset \pi_{cert\#}(MovieExec).$$

Referential integrity constraints: example 2

Consider the two relations from the movie database:

$$StarsIn(movieTitle, movieYear, starName)$$
$$Movies(title, year, length, genre, studioName, producerC\#)$$

We expect that any movie mentioned in the relation *StarsIn* also appears in the relation *Movies*. Since movies are represented in both relations by title-year pairs, then we have the constraint expressed by

$$\pi_{(movieTitle,\, movieYear)}(StarsIn) \subset \pi_{(title,\, year)}(Movies).$$

Expressing key constraints: example 1/2

We now see how to use the same notation seen before for expressing key constraints.

The attribute *name* is the key for the relation

MovieStar(name, address, gender, birthdate)

that is, no two tuples agree on the *name* component. This means that if two tuples agree on *name*, then they must also agree on *address*, *gender* and *birthdate*.

Expressing key constraints: example 2/2

Let us express that constraint for *address* (for the other two attributes it is the same procedure).

We make the Cartesian product of *MovieStar* with itself, renaming the first factor *MS1* and the second factor *MS2*. Then the key constraint means that there do not exist one tuple of *MS1* and one of *MS2* agreeing on *name* and not agreeing on *address*,

$$\sigma_{(MS1.name=MS2.name\ AND\ MS1.address\neq MS2.address)}(MS1 \times MS2) = \emptyset.$$

Other constraints: example 1

To specify that the only legal values for the *gender* attribute of *MovieStar* are 'F' and 'M' we use the expression

$$\sigma_{(gender \neq `F' \ AND \ gender \neq `M')}(MovieStar) = \emptyset.$$

We could express this constraint also as

$$\pi_{gender}(MovieStar) \subset \{F, M\}.$$

Other constraints: example 2

Suppose that we want to require that one must have a net worth of at least \$10,000,000 to be the president of a movie studio. We proceed this way.

First we theta-join the two relations

$$MovieExec(name, address, cert\#, netWorth)$$
$$Studio(name, address, presC\#)$$

using the condition that *presC#* from *Studio* and *cert#* from *MovieExec* are equal. This join combines pairs of tuples consisting of a studio and an executive, such that the executive is the president of the studio.

If we select from this relation those tuples where the net worth is less than ten million, we have a set that, according to our constraint, must be empty,

$$\sigma_{(netWorth<10000000}(Studio \bowtie_{presC\#=cert\#} MovieExec) = \emptyset.$$

Other constraints: example 2, another solution

An alternative way to express the same constraint is to compare the set of certificates that represent studio presidents with the set of certificates that represent executives with a net worth of at least \$10,000,000; the former must be a subset of the latter.

This is expressed by the containment

$$\pi_{presC\#}(Studio) \subset \pi_{cert\#}\left(\sigma_{netWorth \geq 10000000}(MovieExec)\right).$$

The end.