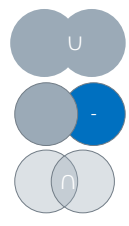



In this module we'll learn about relational algebra, which has been used as the basis of the SQL data manipulation language for relational databases.

π

Basic Operations

- Selection (σ)
 - › Selects a **subset of rows** from a relation (horizontal split).
- Projection (π)
 - › Specifies the **attributes** to include (vertical split).
- Cartesian Product (\times)
 - › **Combines** every element of a set (relation) with every element of another set.
- Union (\cup)
 - › Includes **all** tuples from two or more relations.
- Set-difference ($-$)
 - › Includes tuples from relation A, **but not the ones that are also in B**.
- Intersection (\cap)
 - › Includes tuples that are **both** in relation A and B.





2

Relational algebra is a theoretical language with operations that work on one or more relations to define a new relation without changing the original relation/relations. Relational algebra illustrates how the relational model is based on set theory. The practical implementation of relational algebra is SQL. As always, there are several variations of syntax, even in the theoretical science of relational algebra. This lecture just shows you the most common symbols.

There are five fundamental operations in relational algebra.

Selection works on a single relation and extracts only those tuples that satisfy a specific condition. The result is a new relation with a subset of tuples.

Projection works on a single relation and extracts only the desired attributes. The new relation contains a vertical subset of the content of the original relation.

The Cartesian product needs at least two sets. Every item in each set is combined with every item in every other set. If you're still awake you might be given to wonder how this is useful. Most entries in one set of items, you would think, are specific to one or a few items in another set. You're right, you need to know about Cartesian products mostly so you know how to avoid them!

Unions contain tuples from two tables or tuples from one table that fulfil different conditions. Union is just a combination of different sets of tuples.

Set-difference finds all tuples from a relation that are not found in the other.

Intersection finds all the tuples that two sets have in common.

If this sounds very theoretical, that's because this is a theory!

π

Selection

> Example: Find all staff with a salary greater than 10000.
 $\sigma_{\text{salary} > 10000}(\text{Staff})$

Staff

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003

In this example, we look into selecting all tuples for those of our staff who satisfy the condition of having a salary above 10000 ($\text{salary} > 10000$). The notation for selection uses the sigma sign and the restriction is shown as the subscript. The relation affected is mentioned in parentheses.

Looking at the staff relation (table), you can see that there are four tuples that satisfy the condition. The new relation is the result of the query. It shows only the four tuples that match the restriction.

π

Projection (π)

- Example: Find only staffNo, fName, lName and salary of all staff.
- $\pi_{\text{staffNo}, \text{fName}, \text{lName}, \text{salary}}(\text{Staff})$

Staff

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

→

staffNo	fName	lName	salary
SL21	John	White	30000
SG37	Ann	Beech	12000
SG14	David	Ford	18000
SA9	Mary	Howe	9000
SG5	Susan	Brand	24000
SL41	Julie	Lee	9000

4

A projection is a selection of attributes. The notation is a pi and the columns we want to include are in the subscript.

So we look at the staff table and pick the attributes we want. The result is a new table with just those attributes.

π

Combining Selection and Restriction

› Example: Find the propertyNo of all properties that have three rooms.

- Select all rows where rooms = 3
- Project to return the propertyNo column only

– $\pi_{\text{PropertyNo}}(\sigma_{\text{rooms}=3}(\text{PropertyForRent}))$

PropertyForRent

propertyNo	street	city	postcode	type	rooms	rent	ownerNo	staffNo	branchNo
PA14	16 Holthead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
PL94	6 Aspyll St	London	NW2	Flat	4	400	CO87	SL41	B005
PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40		B003
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93	SG37	B003
PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003

→

propertyNo
PG4
PG36

5

In many actual queries, we need both restriction and projection. This is because many relations have lots of attributes, and showing them all is only confusing. If we combine selection and projection, we work in a ‘set of a set’ kind-of-fashion. We extract a set and then give it to the next operator as a relation. So here we execute the selection in the parenthesis first, then give it to the projection. In the parentheses, where we used to have the name of the relation, we now have the outcome of the selection.

So the selection of the properties with three rooms happens first, and from that result we pick only the attribute that has the propertyNo, which is a unique identifier of the properties. We could, of course, use other attributes as well if we wanted.

If we couldn’t combine operators, the possibilities of relational operators would be pretty limited.

π Cartesian Product (X)

- Example: List the names and comments of all clients who have viewed a property for rent.
- $\pi_{clientNo, fName, lName}(Client) \times \pi_{clientNo, propertyNo, comment}(Viewing)$

Client

clientNo	fName	lName	telNo	prefType	maxRent
CR76	John	Kay	0207-774-5632	Flat	425
CR56	Aline	Stewart	0141-848-1825	Flat	350
CR74	Mike	Ritchie	01475-392178	House	750
CR62	Mary	Tregear	01224-196720	Flat	600

Viewing

clientNo	propertyNo	viewDate	comment
CR56	PA14	24-May-04	too small
CR76	PG4	20-Apr-04	too remote
CR56	PG4	26-May-04	
CR62	PA14	14-May-04	no dining room
CR56	PG36	28-Apr-04	

Result of Cartesian Product (X):

clientNo	fName	lName	Viewing.clientNo	propertyNo	comment
CR76	John	Kay	CR56	PA14	too small
CR76	John	Kay	CR76	PG4	too remote
CR76	John	Kay	CR62	PA14	no dining room
CR76	John	Kay	CR56	PG36	
CR56	Aline	Stewart	CR56	PA14	too small
CR56	Aline	Stewart	CR76	PG4	too remote
CR56	Aline	Stewart	CR62	PA14	no dining room
CR56	Aline	Stewart	CR56	PG36	
CR74	Mike	Ritchie	CR56	PA14	too small
CR74	Mike	Ritchie	CR76	PG4	too remote
CR74	Mike	Ritchie	CR62	PA14	no dining room
CR74	Mike	Ritchie	CR56	PG36	
CR62	Mary	Tregear	CR56	PA14	too small
CR62	Mary	Tregear	CR76	PG4	too remote
CR62	Mary	Tregear	CR62	PA14	no dining room
CR62	Mary	Tregear	CR56	PG36	

Oh for heaven's sake.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Back to the basic operations of relational algebra.

Cartesian product is a concept from set theory. It means take two sets and make all pairs of items where one item is from set A and the other from set B. So all combinations of items from two sets. If you think of it from a database point of view, this really makes no sense.

If we want to find the names and comments of all clients on each of the property, what we logically have to do is match the tuples on the foreign key – primary key relationship. So the clientNo is the column that links the tables. But Cartesian product means we aren't using the column. We are combining rows willy-nilly. If you think this is daft, you are absolutely correct. The only reason why you need to know about Cartesian products in relational databases is to avoid them. Students often get Cartesian products by mistake, because they are forgetting to tell the DBMS to link the tables by the foreign key – primary key link. If a database task says you should get 10 rows as a result and you're getting 200, you've likely created the Cartesian product.

It's important to realise this because the result of the query is nonsense. We can see that the result here has two clientNo fields, and they do not match. What this tells us is that John Kay didn't make the comment 'too small', because it was made by client CR56 and John is client CR76. If you didn't include the clientNo field twice, you might not even notice this.

For what it's worth, the slide shows you the syntax for Cartesian product, using two projections. There is no reason why you can't use a selection as well. You still get

a Cartesian product even when you use a subset of the tuples of two tables, because the two subsets are still combined in every possible manner.

π

Union (\cup)

› Example: Find all cities that have either a branch office or a property for rent

– $\pi_{city}(Branch) \cup \pi_{city}(PropertyForRent)$

Branch

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

\longrightarrow

PropertyForRent

propertyNo	street	city	postcode	type	rooms	rent	ownerNo	staffNo	branchNo
PA14	16 Holthead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
PL94	6 Argyll St	London	NW2	Flat	4	400	CO87	SL41	B005
PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40		B003
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93		B003
PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003

city
London
Aberdeen
Glasgow
Bristol

Union is another operation from set theory. It exists as an operator in relational databases, but it isn't used very often. What is worth noting, though, is that UNION eliminates duplicates. If you remember, sets have neither an ordering nor duplicates. So according to the rules of set theory, UNION eliminates duplicates. What you've also noticed is that you need to have compatible columns when you make a UNION of two relations because the resulting tuples are shown in the same column or columns.

If you think about including the street, this would be perfectly fine. In set theory, because the streets are all different, we no longer have duplicates and all entries should be shown – meaning Glasgow and London would appear three times and Aberdeen twice.

In practice, relational DBMSs have a UNION ALL command that return duplicates just in case you want them.

Regarding the syntax, you can see that we project the same column from Branch and PropertyForRent, then union them.

π

Set Difference

› Example: Find all cities where there is a branch office but no property for rent.

– $\pi_{city}(Branch)$ – $\pi_{city}(PropertyForRent)$

Branch

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

→

city
Bristol

PropertyForRent

propertyNo	street	city	postcode	type	rooms	rent	ownerNo	staffNo	branchNo
PA14	16 Holthead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
PL94	6 Argyll St	London	NW2	Flat	4	400	CO87	SL41	B005
PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40		B003
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93		B003
PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003

Set difference is useful in practice. You can see how this example may be motivated by a CEO getting worried about some branch offices not pulling their weight – why would a city have a branch office if there were no rental properties there?

As you can see, the syntax is quite similar to the UNION one, it just replaces UNION with a minus. Whereas with UNION, you could obviously swap the projections around, in this case it is important which projection comes first. If you swap them, you'll get all rental properties in cities where there are no branch offices. Not quite the intended meaning.

Again, because it's set theory, you won't have duplicates in the result. Not that there would be any in this case anyway.

π

Intersection (\cap)

- Example: Find all cities where there is both a branch office and at least one property for rent.
- $\pi_{city}(Branch) \cap \pi_{city}(PropertyForRent)$

Branch

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

PropertyForRent

propertyNo	street	city	postcode	type	rooms	rent	ownerNo	staffNo	branchNo
PA14	16 Holthead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
PL94	6 Argyll St	London	NW2	Flat	4	400	CO87	SL41	B005
PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40		B003
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93		B003
PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003

→

city
Aberdeen
London
Glasgow

Intersection is a set operation where we identify the items that are in both sets. This is a practical operation that you're likely to use in relational databases as well. Interestingly, the INTERSECT operator exists in very few relational database products, so you have to work with restrictions instead.

In this example, we need to create a relation that lists all cities which have a branch office and at least one property for rent.

Like in the case of UNION, it makes no difference in which order you list the projections. Both projections are carried out first and then we discard the ones that are only in one of the relations.

Again, no duplicates are included.

π

Complex Operations and Temporary Sets

- Result \leftarrow operation
- $(\pi_{clientNo, fName, lName}(Client)) \times (\pi_{clientNo, propertyNo, comment}(Viewing))$


```

TempClient  $\leftarrow$   $(\pi_{clientNo, fName, lName}(Client))$ 
TempViewing  $\leftarrow$   $(\pi_{clientNo, propertyNo, comment}(Viewing))$ 

Result  $\leftarrow$  TempViewing X TempClient
          
```

same

Result(clientNo, fName, lName, vclientNo, propertyNo, comment) \leftarrow TempViewing X TempClient


10

Relational algebra can be very complex, so we need a way to subdivide the steps into smaller ones. We can then read the operation like an algorithm. This way, our heads don't explode when we try to find out what is going on.

Using the assignment operation denoted by an arrow, we can assign an operation to a specific set variable. In this example we have two temporary sets and a final result set.

In this example we are making another Cartesian product of two projections. We can execute the projection operations first and put each result in its own set. First we get the result of the projection from Client and assign it to TempClient. Then we get the result of the projection from Viewing and assign it to TempViewing.

Now we can create the cartesian product of TempViewing and TempClient and assign them to Result. When we denote the result variable, we can either mention all column names or omit them, whichever way we prefer.



The slide is titled "JOIN Operations" and features a list of four join types on the left, each with a corresponding symbol or expression in a grey box on the right. The symbols are: π for Equijoin, $=$ for Natural join, $a.clientNo = b.clientNo$ for Outer join, and π for Semijoin. A small logo for "SWINBURNE UNIVERSITY OF TECHNOLOGY" is visible in the bottom left corner of the slide frame.

Join Type	Symbol/Expression
Equijoin	$=$
Natural join	$a.clientNo = b.clientNo$
Outer join	π
Semijoin	π

JOIN operations are the answers to our predicament with the Cartesian product. JOINS actually use a meaningful column to join tuples from different tables on. So this kind of operation you'll need quite a lot when working with relational databases.

Equijoin means the join operator is an equal sign. This is probably what you expected, and it is by far the most common operator on which to link two tables, but it isn't the only one. You could use less than or greater than just as well.

A Natural join is an equijoin that joins the entries of two tables based on the columns that have the same name in both tables. It exists in many database products.

Natural joins are a concept you should know about at the time you are modelling your data. If you use different names for columns in different tables, or worse still, the same names for columns that don't mean the same, you can't use natural joins.

By default, joins are inner joins, meaning that if we join rows from two relations, if a tuple has no matching tuple in the other table, it is excluded. Sometimes that's not what you want. If you need to include tuples that have no match in the other relation, you have to specify an outer join.

A semijoin is a join where only attributes from one of the relations are included. This can be very practical – essentially it means you are picking rows from a table by a criterion that is specified in another table.

π Join Operation - Natural join (\bowtie)

- Example: Find the names and comments of all clients who have viewed a property for rent.
- $(\pi_{clientNo, fName, lName}(Client)) \bowtie (\pi_{clientNo, propertyNo, comment}(Viewing))$

Client

clientNo	fName	lName	telNo	prefType	maxRent
CR76	John	Kay	0207-774-5632	Flat	425
CR56	Aline	Stewart	0141-848-1825	Flat	350
CR74	Mike	Ritchie	01475-392178	House	750
CR62	Mary	Tregear	01224-196720	Flat	600

Viewing

clientNo	propertyNo	viewDate	comment
CR56	PA14	24-May-04	too small
CR76	PG4	20-Apr-04	too remote
CR56	PG4	26-May-04	too remote
CR62	PA14	14-May-04	no dining room
CR56	PG36	28-Apr-04	

Result of Natural Join:

clientNo	fName	lName	propertyNo	comment
CR76	John	Kay	PG4	too remote
CR56	Aline	Stewart	PA14	too small
CR56	Aline	Stewart	PG4	
CR56	Aline	Stewart	PG36	
CR62	Mary	Tregear	PA14	no dining room

12

Natural joins use columns that have the same name to join the records. This is something you should remember – most people make the mistake of thinking natural join means using the foreign-key – primary key relationship. That's not true, although in many cases the foreign key in a table has the same name as the primary key column it references. So you can forgive people for making that mistake. But as you can guess, making this mistake can have some unpleasant consequences.

This problem also comes down to how conscientiously you work with your naming conventions. If you have two columns that mean totally different things and give them the same name, you only have yourself to blame. Of course you may not be the one who created the database in the first place. This is why you have to be aware of the natural join.

In this example, we pick up the names and comments of all clients who have viewed a property for rent. The only column that has the same name in both tables is clientNo. This column also happens to define the foreign key relationship. So the model is neat and tidy and the result is what we would expect: A meaningful combination of the rows in both tables.

And since this is a one-to-many relationship, the names of the clients repeat as many times as they have been viewing properties.

π Join Operation – Outer join (\bowtie)

- Example: Find currently available properties and their viewings
- $(\pi_{\text{propertyNo}, \text{street}, \text{city}}(\text{PropertyForRent}))$
- $\bowtie \text{propertyNo} = \text{propertyNo}(\pi_{\text{clientNo}, \text{viewDate}, \text{comment}}(\text{Viewing}))$

Viewing

clientNo	propertyNo	viewDate	comment
CR56	PA14	24-May-04	too small
CR76	PG4	20-Apr-04	too remote
CR56	PG4	26-May-04	
CR62	PA14	14-May-04	no dining room
CR56	PG36	28-Apr-04	

PropertyForRent

propertyNo	street	city	postcode	type	rooms	rent	ownerNo	staffNo	branchNo
PA14	16 Holthead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
PL94	6 Argyll St	London	NW2	Flat	4	400	CO87	SL41	B005
PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40		B003
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93	SG37	B003
PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003

Result of Outer Join:

propertyNo	street	city	clientNo	viewDate	comment
PA14	16 Holthead	Aberdeen	CR56	24-May-04	too small
PA14	16 Holthead	Aberdeen	CR62	14-May-04	no dining room
PL94	6 Argyll St	London	null	null	null
PG4	6 Lawrence St	Glasgow	CR76	20-Apr-04	too remote
PG4	6 Lawrence St	Glasgow	CR56	26-May-04	
PG36	2 Manor Rd	Glasgow	CR56	28-Apr-04	
PG21	18 Dale Rd	Glasgow	null	null	null
PG16	5 Novar Dr	Glasgow	null	null	null

SWINBURNE

13

Outer joins are joins that include tuples from one table that don't have matching tuples in the other table. You can have left outer joins and right outer joins.

Here we are producing a report that shows all currently available properties for rent and their possible viewings. If we used the usual equijoin, which is an inner join, we wouldn't have any entries for current properties that haven't been viewed. Which is probably not what we want.

If we specified a natural join for this case, we would find that the properties PL94, PG21 and PG16 wouldn't be included, because inner joins don't include values that don't have a match for the join. The left outer join includes them and shows null values for the attributes from the Viewing table.

π

Aggregation Operation

- Aggregate functions:
 - > COUNT > MIN
 - > SUM > MAX
 - > AVG
- Example: Find out how many properties cost more than 350.

$\rho_R(propCount) COUNT\ propertyNo(\sigma_{rent > 350}(PropertyForRent))$

Used to rename a column

PropertyForRent									
propertyNo	street	city	postcode	type	rooms	rent	ownerNo	staffNo	branchNo
PA14	16 Holthead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
PL94	6 Argyll St	London	NW2	Flat	4	400	CO87	SL41	B005
PG4	6 Lawrence St	Glasgow	G11 9QX	Flat	3	350	CO40		B003
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93	SG37	B003
PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003

→

propCount

5

14

Sometimes, when retrieving information from relations, we get interested in summary data about the relation. For example, a lecturer might need to count the number of students who have passed a unit, or maybe find out the average mark among all students or the highest mark achieved.

To perform such an operation, we can use aggregate functions. The functions defined in relational algebra have been implemented in all relevant database products, so this isn't just a nice theory.

The COUNT function counts how many values in a specific attribute (column). Essentially, it is a row count, but if your attribute has nulls the nulls may be excluded, so you may have to be careful there. Nulls are always trouble.

The SUM operation returns the sum of all values in an attribute.

AVG computes the average over the specified values.

MIN returns the smallest value in an attribute

MAX returns the largest value in an attribute

If we want to find out how many properties cost over 350 pounds a month, we first apply a selection to the PropertyForRent table. Then we apply the aggregate to the resulting set. The aggregate first specifies the column name for the display of the result. Since this is an aggregate, it is an entirely new value, and we can't reuse the column names from the old table. So propCount is a new attribute name for the result. After this we specify the particular function we want – a COUNT function and the column whose entries we are counting. Not surprisingly, we find five entries, because only one property of the six has a rent of exactly 350, not over.

π

Grouping Operation

- Example: find the number of staff working in each branch and the sum of their salaries
- $\rho_R(\text{branchNo}, \text{myCount}, \text{mySum})_{\text{branchNo COUNT staffNo, SUM salary}}(\text{Staff})$

Staff

staffNo	rName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

→

branchNo	myCount	mySum
B003	3	54000
B005	2	39000
B007	1	9000

15

Rather than finding a single number that describes a property of the entire table, such as the number of rows that have more than 350 in the rent column, we can find out properties of groups of values in the table.



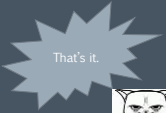
To get an idea how this works, let's look at this example, which requires us to find how many staff are working in each branch office and the sum of their salaries.

The table shows that the branch column is not unique in the staff table – this is expected because we know that more than one staff is working in each branch. If we want to know how many staff there are and how much they earn, we have to count them by branch and sum up their salaries by branch. So we group the staff by branchNo.

The rho function does just that. It specifies the columns to project (branchNo) or create (myCount, mySum) for the result. Then it states that it will use branchNo to form the groups. The count uses the staffNo column. For the Sum function we have to use the salary column. All this is done using the staff relation, that's why it is mentioned in the parentheses.

This type of operation has been implemented in all relational database products, so you will be able to try this out in practice later.

Summary



- › Relational algebra is a theoretical language with operations that work on one or more relations to define another relation without changing the original relation/relations.
- › It consist of many operations that perform basic tasks such as selection, projection, Cartesian product, union, intersection and set difference.
- › It can also be used to join two relations with the different join operations.
- › Aggregated and grouping operations assist in producing results that are of interest which contains specific values based on the relations.
- › Relational algebra is the set-theory underpinning of SQL queries. If you understand the basics of relational algebra, SQL will be easy.

16

Here are the most important points discussed in this module. You may want to stop the recording to have a read through them. When you are ready, start the quiz about this module.