Querying a Relational Database

So far we have seen how to design a relational database and now we turn to mechanisms to interrogate a database

We wish to extract from the database a subset of the information that answers some question. Here are some typical questions we might ask:

- What are the department names?
- Tell me all the data held about male employees.
- What are the names of the employees in the R&D department?

The questions are called **queries** and consists of programs built out of:

(i) retrieving data as a subset of some relation

and (ii) combining two relations together in a meaningful way

We will see two kinds of language for describing queries

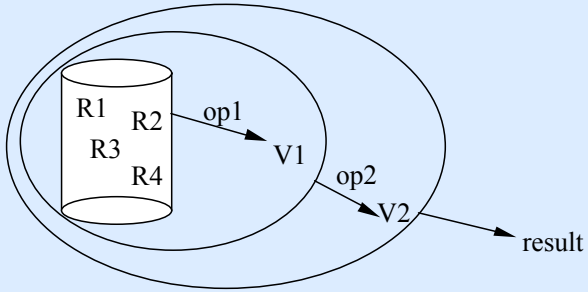
- **procedural** languages describe step-by-step what is to be done
- **declarative** language only describe what is to be returned

Procedural Languages

The **procedural** approach builds programs as sequences of sub-setting and combining operations;

- The base tables in the database are **relations**
- Each of the operators returns a **relation** which is **derived** from the base tables (i.e. a **view** not a **copy)**
- the result of one operator can be fed into another

The **Relational Algebra** is a formal (i.e. not one you can use in practice) procedural language which is what the computer uses inside



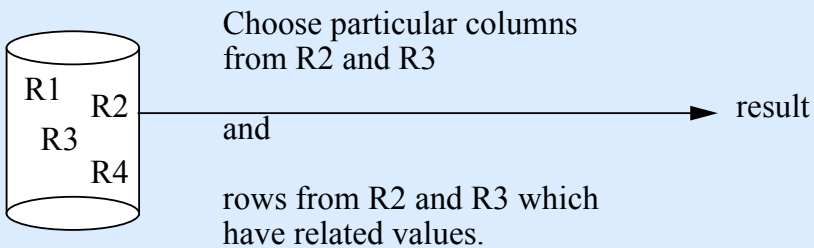
Declarative Languages

The **declarative** approach describes the desired results and lets the DBMS work out the best sequence of operations

The **Relational Calculus** (not covered) is a formal declarative language

SQL is a concrete language (i.e. you can use it) which is declarative and is the standard language with which databases are communicated

- both by humans and by other programs

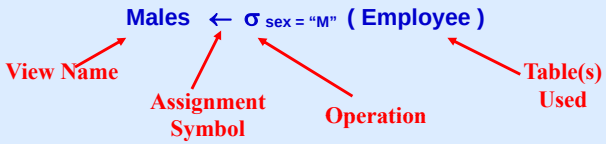


The Relational Algebra

is a formal language for querying relations

- an internal language, NOT a language for the user
- has the basic operations required of any Data Manipulation Language
- all DBMS should provide these capabilities
- they can be used as the basis of a usable Data Manipulation Language
- they are used in query implementation and optimization
  - SQL is turned into these operations
- relations can be manipulated in limited but useful ways
- many query languages also allow additional operations such as calculations, summary and ordering

A query in the relational algebra is a series of assignments to variables which hold views, e.g.:



SUMMARY : Relational Algebra Operations

Key Slide

The principal relational operations are:

- σ **select*** - pick rows from a relation by some condition
- Π **project*** - pick columns by name
- ⋈ **join** - connect two relations usually by a Foreign Key

The main set operations include:

- ∪ **union*** - make the table containing all the rows of two relations
- ∩ **intersection** - pick the rows which are common to two relations
- – **difference*** - pick the rows which are in one table but not another
- X **Cartesian product*** - pair off each of the tuples in one relation with those in another - creating a double sized row for each pair

All the other operations can be defined in terms of the five marked with a star

All of the operations return **relations**

Selection (or Restriction) σ

Key Slide

Selection extracts the tuples of a relation which satisfy some condition on the **values** of their rows and return these as a relation

Example: return all the employees who work in the city of Glasgow

i.e. get all of the records from the Employee table for which the city column holds the value "Glasgow"

Locals ← σ city = "Glasgow" (Employee)

– *Locals* is then the name of the query which could be used as a view in subsequent queries

The condition is similar to a Java boolean expression except that it uses the words NOT, OR and AND instead of !, || and && and it contain:

- – Literals – i.e. constants
- – column names
- – comparison operators (=, >, etc.)
- – boolean operators (and, not, or)

YoungOrNear ← σ city = "Glasgow" OR (city = "Stirling" AND bDate > '1/1/80') (Employee)

Projection Π

Key Slide

Projection extracts some of the fields from a relation, by giving the names of the fields

GenderSalary ← Π gender, salary (Employee)

In the result:

- – No attribute may occur more than once
- – Duplicate entries will be removed
- – Thus if we want to retain the number of times each is used, we must include the Primary Key

Projection and selection can be combined.

- – It is usually more appropriate to reduce the number of rows first before reducing the columns
- – For instance, to determine the names of employees working in Glasgow

Answer ← Π name (σ city = "Glasgow" (Employee))

This does a selection followed by a projection

- – the inner operation is performed first

Union ∪

The union operator produces a relation which combines two relations by containing all of the tuples from each - **removing duplicates**

- – See examples on next slide

People ← Students ∪ Staff

The two relations must be "union compatible"

- – i.e. have the same number of attributes drawn from the same domains (but maybe having different names)

If attribute names differ, the names from the first one are taken

Two Union Compatible Relations

Key Slide

STUDENT

FN	LN
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

STAFF

FN	LN
John	Smith
Ricardo	Browne
Susan	Yao
Francis	Johns
Ramesh	Shah

STUDENT ∪ STAFF

FN	LN
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johns

STUDENT ∩ STAFF

FN	LN
Susan	Yao
Ramesh	Shah

STAFF - STUDENT

FN	LN
John	Smith
Ricardo	Browne
Francis	Johns

Example

- We would like a list of names and emails of everyone living in Glasgow
- We can use union if we reduce the staff and student tables to a common set of columns

Staff (ni#, sname, city,…, phone, email, room)
Student (ni#, name, city, ……, email, course, year)

GStaff ← Π sname, email (σ city = "Glasgow" (Staff))
GStud ← Π name, email (σ city = "Glasgow" (Student))

GlasgowEmails ← GStudents ∪ GStaff

Intersection ∩

This is similar to union but returns tuples that are in **both** relations

- Example – the female students living in Glasgow:

FemalesInGlasgow ← σ city = "Glasgow" (Employee)
∩ σ gender = "F" (Employee)

- *How else can this particular query be written?*
- *Why can it be rewritten?*

Difference -

Similar to union but returns tuples that are in the first relation but not the second

- Example non-local employees:

NonLocals ← Employee - Locals

Intersection and difference both require union compatibility

Both use column names from the first relation

Only operations based on the same relation can be rewritten

Cartesian Product X

Every row from the first relation is paired with every row in the second relation – **rarely** useful on its own

Employee

name	edNum	ni#
Jim Grey	5	444
Jo White	5	555

Dependent

eni#	dpdName	relship
444	Tim	Child
444	Tom	Father
555	June	Child

Employee x Dependent

	name	edNum	ni#	eni#	Name	relship
*	Jim Grey	5	444	444	Tim	Child
*	Jim Grey	5	444	444	Tom	Father
	Jim Grey	5	444	555	June	Child
	Jo White	5	555	444	Tim	Child
	Jo White	5	555	444	Tom	Father
*	Jo White	5	555	555	June	Child

Defining the Cartesian Product

The **Cartesian Product** of two relations A and B, which have attributes $A_1 \ldots A_m$ and $B_1 \ldots B_n$, is the relation with $m + n$ attributes containing a row for every pair of rows, one from A and one from B

Thus if A has *a* tuples and B has *b* tuples then the result has *a x b* tuples

Equi-Join

The rows marked with a star on Slide 349 are the ones we really want

This could be created with the following selection, which essentially makes use of the foreign key

$$\sigma_{ni\# = eni\#}\ (\text{Employee} \times \text{Dependent})$$

Cartesian product followed by this kind of selection is called a **JOIN** because it joins together two relations

There are a wide variety of join operators, as we shall see

This one is called an **equi-join**

- It has the weakness that it keeps both of the columns which are now identical

Key Slide

Natural or Inner Join ⋈

In its simplest form, the join of relations A and B pairs off the tuples of A and B so that named attributes from the relations have the same value

Now we have two columns holding the same value, so we eliminate the duplicated column to form the **natural** or **inner** join

$$\text{Employee} \bowtie_{ni\# = eni\#} \text{Dependent}$$

	name	edNum	ni#	dpdName	relship
*	Jim Grey	5	444	Tim	Child
*	Jim Grey	5	444	Tom	Father
*	Jo White	5	555	June	Child

When we use the term "join" without further comment, this is what is meant!

Employee

ni#	Name	edNum	etc
222	Joe Brown	4	….
333	Kay Lee	4	….
555	Tom Low	5	…..

Department

dNum	dName	mgrni#	etc
4	R & D	222	….
5	Admin	555	….
6	Finance		…..

These relations can be joined in two ways

- Here they are joined on department number, so that we have the department details for an employee
- Unmatched tuples disappear (no employees in Finance)

EmpAndDept ← Employee ⋈ edNum = dNum Department

ni#	Name	edNum	etc	dName	mgrni#	etc
222	Joe Brown	4	….	R & D	222	….
333	Kay Lee	4	….	R & D	222	….
555	Tom Low	5	…..	Admin	555	…..

They can also be joined by matching the employee national insurance number with the manager national insurance number, to give the department details together with the employee who manages it

Unmatched tuples disappear (no manager for Finance, and Kay Lee is not a manager)

MgrAndDept ← Employee ⋈ ni# = mgrni# Department

ni#	Name	edNum	etc	dNum	dName	etc
222	Joe Brown	4	….	4	R & D	….
555	Tom Low	5	….	5	Admin	….

Theta Join θ

The most general form of join is called a **theta-join.** This allows the connection of tuples by other comparison components

The operator, θ, can be any comparison operator

- e.g. < > >= <=
- if it is = this is an equi-join

Employee X ni# θ mgrni# Department

Outer Join

This includes all of the unmatched data so as to preserve all the data

Employee outerJoin ni# = mgrni# Department

ni#	Name	edNum	etc	dNum	dName	etc
222	Joe Brown	4	….	4	R & D	….
555	Tom Low	5	….	5	Admin	….
333	Kay Lee	4	…..	null	null	null
null	null	null	null	6	Finance	…..

Right and Left Outer Joins

Left and **right outer joins** include unmatched data from only one of the two relations

- To get a list of all departments and their employees, and to include the departments without employees – a right outer join

Employee rightOuterJoin ni# = mgrni# Department

ni#	Name	edNum	etc	dNum	dName	etc
222	Joe Brown	4	….	4	R & D	….
555	Tom Low	5	….	5	Admin	….
null	null	null	null	6	Finance	…..

Semi Join ▷

A semi-join is one in which only the columns of one of the two tables is returned :

- E.g. to get the details of the managers, a left semi-join version of the join on the manager number would give:
- Because the only employees in the result are those which join with departments on the manager foreign key, all non-managers are eliminated

ni#	Name	edNum	etc
222	Joe Brown	4	….
555	Tom Low	5	….

Division ÷

"Which employees work on **all** projects that John Smith works on"

- These are the most difficult kinds of query to describe in relational algebra (and in SQL)

Start by finding all the numbers of the projects that John Smith works on (maybe these are 3 & 4)

EMPWO ← Emp $\bowtie_{ni\#=wni\#}$ WorksOn
JSPNos ← $\sigma_{eName="JS"}$ (EMPWO)
JSPNos ← $\prod_{pNum}$ (JSproj)

Then make a list of who works on which project

ProjEmps ← $\prod_{wni\#, wpNum}$ (WorksOn)

"Dividing" the *JSPNos* into *ProjEmps* gives us the employee numbers of anyone who works on all the entries in the *JSPNos* relation:

Result ← ProjEmps ÷ JSPNos

JSPNos

pNum
3
4

ProjEmps

wni#	wpNum
123	1
145	3
145	4
269	1
169	3
172	2
172	3
172	4

Result

wni#
145
172

Some Examples

Note the words in the question which alert us to a division query are **all** and **every**

1. Give me the name and salaries of all employees who work for the R & D department List the project name, controlling department, and the department manager's name for every project in Stafford

ResDept ← σ Dname="R&D"(DEPARTMENT)
ResDeptEmps ← Emp $\bowtie_{dNum=dept}$ ResDept
Result ← $\prod_{name, salary}$ (ResDeptEmps)

Common Pattern: select – join - project

2. List the project name, controlling department, and the department manager's name for every project in Stafford

StaffordProjects ← σ Location="Stafford"(PROJECT)
StaffProjDepts ← StaffordProjects $\bowtie_{conDept = dNum}$ Department
StaffProjMgrs ← StaffProjDepts $\bowtie_{mgrni\# = ni\#}$ Employee
Result ← $\prod_{Pname,Dname, name}$ (StaffProjMgrs)

3. List the names of employees who work on **all** the projects controlled by department 5.

D5projects ← σ conDept=5(Project)
D5projNums ← $\prod_{pNum}$ (D5Projects) // just the numbers
EmpProj ← $\prod_{emp, project}$ (WorksOn) // get rid of hours
EmpNIs ← EmpProj ÷ D5ProjNums // get emp numbers
EmpsWanted ← EmpNIs $\bowtie_{emp=ni\#}$ Employee
Result ← $\prod_{name}$ (EmpsWanted)

General Strategy: Identify which tables you need. Usually start by reducing number of rows. Join. Then reduce number of columns to what you are asked for.