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

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

Choose particular columns from R2 and R3

and

rows from R2 and R3 which have related values.

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.
SUMMARY: Relational Algebra Operations

The principal relational operations are:

- \( \sigma \) select*: pick rows from a relation by some condition
- \( \Pi \) project*: pick columns by name
- \( \bowtie \) join: connect two relations usually by a Foreign Key

The main set operations include:

- \( \cup \) union*: make the table containing all the rows of two relations
- \( \cap \) intersection: pick the rows which are common to two relations
- \( - \) difference*: pick the rows which are in one table but not another
- \( \times \) 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) \( \sigma \)

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

\[ \text{Locals} \leftarrow \sigma \text{city} = \text{"Glasgow"} \ (\text{Employee}) \]

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
- compare operators ( =, >, etc.)
- column names
- boolean operators ( and, not, or )

YoungOrNear \( \leftarrow \sigma \text{city} = \text{"Glasgow"} \ OR \ (\text{city} = \text{"Stirling"} \ AND \ \text{bDate} > '1/1/80') \ (\text{Employee}) \)

Projection \( \Pi \)

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

\[ \text{GenderSalary} \leftarrow \Pi \text{gender}, \text{salary} (\text{Employee}) \]

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

\[ \text{Answer} \leftarrow \Pi \text{name} (\sigma \text{city} = \text{"Glasgow"} \ (\text{Employee})) \]

This does a selection followed by a projection

- the inner operation is performed first

Union \( \cup \)

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

\[ \text{People} \leftarrow \text{Students} \cup \text{Staff} \]

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

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>STAFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>FN LN</td>
<td>FN LN</td>
</tr>
<tr>
<td>Susan Yao</td>
<td>John Smith</td>
</tr>
<tr>
<td>Ramesh Shah</td>
<td>Ricardo Browne</td>
</tr>
<tr>
<td>Johnny Kohler</td>
<td>Susan Yao</td>
</tr>
<tr>
<td>Barbara Jones</td>
<td>Johnny Kohler</td>
</tr>
<tr>
<td>Amy Ford</td>
<td>Barbara Jones</td>
</tr>
<tr>
<td>Jimmy Wang</td>
<td>Amy Ford</td>
</tr>
<tr>
<td>Ernest Gilbert</td>
<td>Jimmy Wang</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STUDENT ∪ STAFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>FN LN</td>
</tr>
<tr>
<td>Susan Yao</td>
</tr>
<tr>
<td>Ramesh Shah</td>
</tr>
<tr>
<td>Johnny Kohler</td>
</tr>
<tr>
<td>Barbara Jones</td>
</tr>
<tr>
<td>Amy Ford</td>
</tr>
<tr>
<td>Jimmy Wang</td>
</tr>
<tr>
<td>Ernest Gilbert</td>
</tr>
</tbody>
</table>

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 ( n#, sname, city, ..., phone, email, room )
Student ( n#, name, city, ..., email, course, year )

GStaff ← ∪ sname, email ( σ city = "Glasgow" ( Staff ) )
GStud ← ∪ 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
Defining the Cartesian Product

The **Cartesian Product** of two relations A and B, which have attributes A₁ ... Aₘ and B₁ ... Bₙ, 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.

---

**Employee x Dependent**

<table>
<thead>
<tr>
<th>name</th>
<th>edNum</th>
<th>ni#</th>
<th>eni#</th>
<th>dpdName</th>
<th>relship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim Grey</td>
<td>5</td>
<td>444</td>
<td>444</td>
<td>Tim</td>
<td>Child</td>
</tr>
<tr>
<td>Jo White</td>
<td>5</td>
<td>555</td>
<td>555</td>
<td>June</td>
<td>Child</td>
</tr>
</tbody>
</table>

---

**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

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

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**.

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

<table>
<thead>
<tr>
<th>name</th>
<th>edNum</th>
<th>ni#</th>
<th>eni#</th>
<th>dpdName</th>
<th>relship</th>
</tr>
</thead>
<tbody>
<tr>
<td>* Jim Grey</td>
<td>5</td>
<td>444</td>
<td>444</td>
<td>Tim</td>
<td>Child</td>
</tr>
<tr>
<td>* Jim Grey</td>
<td>5</td>
<td>444</td>
<td>444</td>
<td>Tom</td>
<td>Father</td>
</tr>
<tr>
<td>* Jo White</td>
<td>5</td>
<td>555</td>
<td>555</td>
<td>June</td>
<td>Child</td>
</tr>
</tbody>
</table>

When we use the term “join” without further comment, this is what is meant!
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 \( \bowtie \) edNum = dNum Department

Theta Join \( \theta \)

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

The operator, \( \theta \), can be any comparison operator
- e.g. \(<, >, \geq, \leq\)
- if it is = this is an equi-join

Employee \( \bowtie \) \( \theta \) ni# = mgrni# Department

Outer Join

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

Employee \( \bowtie \) \( \bowtie \) ni# = mgrni# Department

Right and Left Outer Joins

\textit{Left} and \textit{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 \( \bowtie \) ni# = mgrni# Department

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 \( \bowtie \) ni# = mgrni# Department

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 \( \bowtie \) edNum = dNum Department
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

<table>
<thead>
<tr>
<th>ni#</th>
<th>Name</th>
<th>edNum</th>
<th>etc</th>
</tr>
</thead>
<tbody>
<tr>
<td>222</td>
<td>Joe Brown</td>
<td>4</td>
<td>.....</td>
</tr>
<tr>
<td>555</td>
<td>Tom Low</td>
<td>5</td>
<td>.....</td>
</tr>
</tbody>
</table>

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)

\[
\text{ProjEmps} \leftarrow \Pi_{\text{wni#}, \text{wpNum}} (\text{WorksOn})
\]

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

\[
\text{Result} \leftarrow \text{ProjEmps} \div \text{JSPNos}
\]

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

\[
\begin{align*}
\text{ResDept} & \leftarrow \sigma_{\text{Dname}='\text{R&D'}\text{'}(\text{DEPARTMENT }) \\
\text{ResDeptEmps} & \leftarrow \text{Emp} \bowtie \text{dNum}\rightarrow \text{dept ResDept} \\
\text{Result} & \leftarrow \Pi_{\text{name}, \text{salary}} (\text{ResDeptEmps})
\end{align*}
\]

Common Pattern: select – join - project

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

\[
\begin{align*}
\text{StaffordProjects} & \leftarrow \sigma_{\text{Location}='\text{Stafford'}\text{'}(\text{PROJECT }) \\
\text{StaffProjDepts} & \leftarrow \text{StaffordProjects} \bowtie \text{conDept} \rightarrow \text{dNum, Department} \\
\text{StaffProjMgrs} & \leftarrow \text{StaffProjDepts} \bowtie \text{mgrni#} \rightarrow \text{ni# Employee} \\
\text{Result} & \leftarrow \Pi_{\text{name, Dname, name (StaffProjMgrs)}}
\end{align*}
\]

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