Ian Kenny

November 28, 2017
Introductory Databases
Relational Algebra
Introduction

In this lecture we will cover Relational Algebra.

Relational Algebra is the foundation upon which SQL is built and is used for query optimisation in DBMSs.

As its name suggests, relational algebra is an algebra for relations ... Thus it has operators that operate on relations. (Compare with the more familiar algebra of the real numbers, which has operations like +, -, * which produce real numbers as results.)

Relational Algebra is closed: operations on relations produce relations. This means that relational algebra expressions can be nested.

Relational algebra is a procedural language. Expressions imply an order of processing for a query.
Set theory background

The term *relation* is from set theory. Consider two sets $R$ and $S$

$$R = \{1, 2\}, S = \{2, 3, 4\}$$

The *cross product* of $R$ and $S$, $R \times S$, is the set of all ordered pairs such that the first element is a member of $R$ and the second element is a member of $S$.

$$R \times S = \{(1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4)\}$$

Each element of this set is a *tuple*: an ordered list of values. Any subset of $R \times S$ is a *relation*.

The cross product is not limited to two sets, and can be extended to any number of sets. The cross product of three sets, for example, will result in a set of ordered triples. In general, the cross product operation will produce *$n$-tuples*: tuples with $n$ elements.
Set-builder notation

We can use *set builder notation* to define a set. For example, to specify the cross product of two sets $R$ and $S$

$$T = \{(x, y) : x \in R, y \in S\}$$

We can also specify other conditions for the construction of the set, where some condition is satisfied. For example, we can specify the relation $T$ is a subset of $R \times S$ such that the second component of each pair is equal to 4

$$T = \{(x, y) : x \in R, y \in S, \text{ and } y = 4\}$$
Database relations

A database table can also be called a relation. We will see why.

We call the columns of a database table the attributes.

We call the set of values from which any given attribute can be taken, the domain of those attributes.

Any particular set of values for all attributes in a relation is called a tuple. A tuple is a row of a table.

The degree of a relation is the number of attributes the relation has.

The cardinality of a relation is the number of tuples in the relation.
Database relations

If we have attributes $A_1, A_2, \ldots, A_n$ with domains $D_1, D_2, \ldots, D_n$ then the set

$$\{A_1 : D_1, A_2 : D_2, \ldots, A_n : D_n\} \text{ is a relation schema.}$$

A relation schema is effectively the definition of a relation in a database. It specifies the attributes and the values they may take.

If the value of each attribute is $d_i$ where $i$ is the attribute then the relation itself is a set of n-tuples

$$\{(A_1 : d_1, A_2 : d_2, \ldots, A_n : d_n) : d_1 \in D_1, d_2 \in D_2, d_n \in D_n\}$$

Where $d_1$ is the value of attribute $A_1$, $d_2$ is the value of attribute of $A_2$, $D_1$ is the domain of attribute $A_1$ and $D_2$ is the domain of attribute $A_2$, etc.

A relation is a particular instance of a relation schema with specific values associated with each attribute.
Database relations

We can ignore the attributes and the domains if we know which order they are in. If we consider each tuple as simply the set of values (i.e. we ignore the attributes) then we have, for $n$ attributes

$$(d_1, d_2, ..., d_n)$$

This is a n-tuple of values, each value from the corresponding domain.

If we consider each domain as a set of possible values, then a relation can be seen as a subset of the cross product of the domains.
Database relations

Consider a relation with two attributes $X$ and $Y$ which have domains $D_1$ and $D_2$, respectively.

\[
D_1 = \{A, B, C\} \\
D_2 = \{1, 2, 3, 4\} \\
D_1 \times D_2 =
\]

<table>
<thead>
<tr>
<th>$X$</th>
<th>$Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
</tbody>
</table>
Database relations

Possible subsets (relations) of $D_1 \times D_2$ include

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
</tbody>
</table>
Basic relational algebra operators

Now we have given a basic definition of a relation, we can look at some relational algebra operators and how they operate on relations.

The basic operators in relational algebra are Select, Project and Rename. We will now consider each of these.

We will use the sample set of database tables shown on the next slide.
### Employees database

#### Employees table

<table>
<thead>
<tr>
<th>empid</th>
<th>lname</th>
<th>location</th>
<th>salary</th>
<th>manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Johnson</td>
<td>London</td>
<td>30000</td>
<td>Khan</td>
</tr>
<tr>
<td>1</td>
<td>Khan</td>
<td>London</td>
<td>45000</td>
<td>Davis</td>
</tr>
<tr>
<td>2</td>
<td>Wilson</td>
<td>Oxford</td>
<td>28000</td>
<td>Peters</td>
</tr>
<tr>
<td>3</td>
<td>Cuthbert</td>
<td>Oxford</td>
<td>21000</td>
<td>Peters</td>
</tr>
<tr>
<td>4</td>
<td>Peters</td>
<td>St Albans</td>
<td>50000</td>
<td>Davis</td>
</tr>
<tr>
<td>5</td>
<td>Davis</td>
<td>London</td>
<td>75000</td>
<td></td>
</tr>
</tbody>
</table>

(6 rows)

#### Projects table

<table>
<thead>
<tr>
<th>projectid</th>
<th>projectname</th>
<th>projectleader</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Vanity</td>
<td>Peters</td>
<td>100000</td>
</tr>
<tr>
<td>1</td>
<td>White Elephant</td>
<td>Davis</td>
<td>50000</td>
</tr>
<tr>
<td>2</td>
<td>Infeasible</td>
<td>Peters</td>
<td>125000</td>
</tr>
</tbody>
</table>

(3 rows)

#### Project-temps table

<table>
<thead>
<tr>
<th>projectid</th>
<th>empid</th>
<th>contract</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

(7 rows)
Note that the Select operator in relational algebra does not perform the same operation as the SELECT operation in SQL.

The Select operator returns a relation with the same relational schema as its operand but with a subset of the tuples. The subset selected is determined by a condition applied to the tuples in the operand. Those meeting the condition are added to the result.

This means that the Select operator performs the same operation as the WHERE clause in SQL.

The symbol for Select is the Greek letter sigma $\sigma$.

The Select operator is applied to a relation expression which may be an existing relation or the result of another query.
Select

The form of a Select expression is

$\sigma_{condition}(R)$

The result is a new relation containing the subset of tuples from $R$ that meet the condition.
Assignment statements

It can be helpful to use assignment statements in relational algebra, which are analogous to the setting of variable values in many programming languages. An assignment statement simply sets the value of an identifier to the value of an expression. For example, we could have the following statement

\[ S := \sigma_{condition}(R) \]

\( S \) is a relation containing the subset of tuples from \( R \) that meet the condition.
Select examples

With reference to the employees database, the following selects those employees whose salary exceeds 30000

$$\sigma_{\text{salary} > 30000}(\text{Employee})$$

In SQL this would be

```
SELECT *
FROM Employee
WHERE salary > 30000;
```
Select examples

The following expression selects those employees who are managed by Khan or Davis and live in London

$$\sigma_{\text{manager}=\text{`Khan'} \lor \text{manager}=\text{`Davis'} \land \text{location}=\text{`London'}(\text{Employee})$$

In SQL this would be

SELECT *
FROM Employee
WHERE manager='Khan' or manager='Davis' and location='London';
Project

The *Project* operator returns a relation with a subset of the attributes of the operand but containing the same tuples. Hence, it produces a relation that has a *different* schema to its operand.

The symbol for *Project* is the Greek letter $\Pi$.

The *Project* operator performs the same function as the SELECT clause in SQL.

The *Project* operator is applied to a relation expression which may be an existing relation or the result of another query.
The form of a *Project* expression is

$$\Pi_{(A_1, \ldots, A_n)}(R)$$

Which evaluates to the relation which is the result of projecting attributes $A_1, \ldots, A_n$ on relation $R$. 
The following expression projects the attributes empID and manager on the Employee relation

$$\Pi_{(empID,\text{manager})}(Employee)$$

In SQL this would be

```sql
SELECT empID, manager
FROM Employee;
```
Using Select and Project

It is usually the case that we need to nest *Select* and *Project* operations because we usually want to select tuples based on some condition and also select only a subset of the attributes.

Depending on the result we seek, we can nest operators however we need to. This expression, for example, applies the *Select* operator and then applies the *Project* operator

\[ \Pi_{(A_1, ..., A_n)}(\sigma_{\text{condition}}(R)) \]

The result of this query is a relation with the subset of tuples in \( R \) meeting the condition, and with the set of attributes \((A_1, ..., A_n)\).

Note that the expressions are evaluated in the expected order: the *Select* operation is applied first and then *Project* operation is applied to the result.
Using Select and Project: example

The following query finds the empID and manager of any employee located in London

$$\Pi_{(\text{emplID,manager})}(\sigma_{\text{location}=\text{`London'}}(\text{Employee}))$$
Duplicates

Note that in relational algebra duplicates are eliminated from results. The tuples of a relation form a set: no duplicates.

In SQL, results can contain duplicates, hence the use of the keyword DISTINCT to eliminate them, where desired.
Operations with multiple relations

As with SQL, it is often the case that our queries will need to involve multiple relations, including multiple instances of the same relation. There are various operations that allow relations to be combined, called *joins*. We can also combine results using set operators. Firstly, we will look at joins in relational algebra.
Cross Product

The cross product of two relations is a relation containing all of the tuples in the first relation paired with all of the tuples in the second relation. The cross product of two relations $R$ and $S$ is denoted $R \times S$.

The set of attributes in the result of the cross product is the union of the sets of attributes in the operands. The set of tuples is the cross product of the set of tuples in the operands. If there are $n$ tuples in $R$ and $m$ tuples in $S$ there will be $n \times m$ tuples in the result. No selection or projection occurs.

As with SQL, the cross product is only of limited use by itself.
Cross Product

In SQL the cross product operation is achieved by simply listing the relations to be joined rather than specifying a particular join.

For example

SELECT A,B
FROM R,S
Where A > 6;

This SQL command performs the cross product of $R$ and $S$. 
Cross Product: example

Consider the following relations

\[
\begin{array}{ccc}
A_1 & A_2 & A_3 \\
1 & 2.5 & 'ABB' \\
4 & 4.5 & 'CDK'
\end{array}
\]

\[
\begin{array}{cc}
B_1 & B_2 \\
2.5 & 15 \\
4.5 & 30
\end{array}
\]

Their cross product \( R \times S \) is

\[
\begin{array}{cccccc}
A_1 & A_2 & A_3 & B_1 & B_2 \\
1 & 2.5 & 'ABB' & 2.5 & 15 \\
1 & 2.5 & 'ABB' & 4.5 & 30 \\
4 & 4.5 & 'CDK' & 4.5 & 30 \\
4 & 4.5 & 'CDK' & 2.5 & 15
\end{array}
\]

\( R \times S \)
Cross Product

As with SQL, we can restrict the results of the cross product operation to those tuples that meet a condition, and restrict the attributes in the result, using selection and projection.

\[ \Pi_{(A_1,\ldots,A_n)}(\sigma_{condition}(R \times S)) \]

This expression performs the cross product of \( R \) and \( S \) and then selects only those tuples that meet the condition and projects only the attributes listed.
For example, to find the name of the employees and the IDs of the projects they are working on, apart from those working on the project with ID = 0, we can use the following expression

\[ \Pi_{\text{Employee}.\text{lname}, \text{ProjectEmps}.\text{projectID}} \left( \sigma_{\text{Employee}.\text{empID} = \text{ProjectEmps}.\text{empID} \land \text{ProjectEmps}.\text{projectID} \neq 0} (\text{Employee} \times \text{ProjectEmps}) \right) \]
Assignment statements

As seen earlier, we can assign the result of an expression to an identifier. This can help with the readability of complex expressions. The query on the previous slide can be written with assignment statements, for example

\[
S := Employee \times ProjectEmps
\]
\[
R := \sigma_{Employee.emplID=ProjectEmps.emplID \land ProjectEmps.projectID \neq 0}(S)
\]
\[
T := \Pi_{Employee.lname,ProjectEmps.projectID}(R)
\]
Natural join

The natural join operator takes two relations and joins them on common attributes.

The natural join allows for less cumbersome syntax, as it does in SQL.

The symbol for natural join is $\Join$. 

The form of a natural join expression is

$$R \Join S$$

This expression evaluates to the join of $R$ and $S$ where they are equal on their common attributes. Each tuple in the result must be equal on all common attributes to be included. Common attributes simply have the same name and domain. If they do not have the same name and domain, they are not common.
Natural join: example

This expression creates the same result as the previous cross product example.

\[ \Pi_{Employee.lname, ProjectEmps.projectID} 
\left( \sigma_{ProjectEmps.projectID \neq 0} 
\left( Employee \bowtie ProjectEmps \right) \right) \]

There is no longer a need to express the same join conditions. In particular, we do not need to explicitly specify the equality of empid in the two relations since that is automatically dealt with by the natural join.
Natural join: example

Note that duplicate attributes are removed from the result. In the previous example, the column *empID* was present in both the Employee and ProjectEmps relations. Since, by definition, these columns will be identical following a natural join, only one copy of the attribute is retained.
Theta join

The theta join is a join in which the join condition can be more general than with the natural join (which joins on common attributes only). The theta join allows the specification of the join condition rather than having a default join condition.

The form of a theta join expression is

\[ R \bowtie_\theta S \]

This expression evaluates to the natural join of relations \( R \) and \( S \) on the \( \theta \) condition, i.e. the relations are joined on (same-named) common attributes and then the condition \( \theta \) is applied.
Theta join

The theta join is also equivalent to a selection on the result of a cross product in which the condition also includes the common attribute join, i.e.

\[ R \bowtie_\theta S \equiv \sigma_\theta(R \times S) \]

In this case, \( \theta \) must include the explicit join on the common attributes (unlike the natural join which does this by default).

In the special case where the condition is an equality, the theta join is also called an equijoin.
The Rename operator

Attribute names are important when combining tables. For the natural join, attribute names must be the same for them to participate in the join condition. It may be the case that we wish to join relations on attributes that have the same semantics (or where the comparison is meaningful, at least) but the attributes have different names. For example, in some domain, ownerID and petownerID might mean the same thing.

This naming restriction also applies to the set operators (which we are coming to). For relations to be union compatible they must have the same schema. The rename operator can be used to create modified relations that are union compatible.

Furthermore, it is sometimes necessary to rename a relation. This occurs, for example, when we need to join a table to itself. Ambiguity will arise if we don’t have the facility to give alternative names to relations.
The symbol for the *Rename* operator is rho $\rho$.

The general form of a *Rename* expression is

$$ \rho_{S(A_1,\ldots,A_n)}(R) $$

Where $n$ is number of attributes of $R$.

Which evaluates to the relation which is the result of renaming the relation $R$ to $S$ and renaming the attributes of $R$ to $(A_1,\ldots,A_n)$. 
**Rename**

We may need to specify which attributes are being renamed (if it is not the entire schema, for example). In which case we can specify the renamed attributes as follows

\[
\rho_S(B_1/A_1,\ldots,B_n/A_n)(R)
\]

Where \( B_1 \) is the new attribute name for \( A_1 \), etc. (and, as before \( S \) is the new name for the relation \( R \)).

If we are only renaming the attributes we can omit the rename of the relation

\[
\rho(A_1,\ldots,A_n)(R)
\]

If we are only renaming the relation we can write

\[
\rho_S(R)
\]
Rename: examples

This expression renames the Employee relation to E1

\[ \rho_{E_1}(Employee) \]

This expression renames some of the attributes of the Employee relation

\[ \rho_{(employeeID/empID, city/location)}(Employee) \]

This expression renames the Employee relation and the lname attribute

\[ \rho_{EMP(surname/lname)}(Employee) \]
The standard set operators can be used in relational algebra. Relations must be *union compatible* before they can be used as the operands of the set operators. This means they must have the same schema at the point at which the set operation is applied. Thus, relations can be made to be union compatible with projection and with rename.

The set operators we will use are *union*, *intersection*, and *difference*. 
Union

The union of two sets $R$ and $S$ is the set including all distinct elements in $R$ or $S$. This is the standard definition of the union operation, for example

$$\{1, 2, 3\} \cup \{1, 5, 7\} = \{1, 2, 3, 5, 7\}$$

The form of a union operation is

$$R \cup S$$

Where $R$ and $S$ are union-compatible relations (including more complex expressions that evaluate to relations).
Consider an example query in which we wish to find all of the names of entities in the employees database. This would include the names of employees and projects.\(^1\) The natural way to find this would be to combine together the names from the Employee relation and the names from the Project relation into one relation that contains only names. We can use the \textit{union} operator for this but there is a problem: the names of the attributes are not the same. The relations are not union compatible.

In order to overcome this we can use the \textit{rename} operator to ensure that the attribute names are the same.

\(^1\)This may not be considered a very useful query.
We can formulate this query as follows

\[ \rho_{name}(\Pi_{lname}(Employee)) \cup \rho_{name}(\Pi_{projectName}(Project)) \]
Intersection

The intersection of two sets $R$ and $S$ is the set containing all elements that are in both $R$ and $S$. This is the standard definition of the intersection operation, for example

$\{1, 3, 6, 8\} \cap \{3, 5, 2, 8\} = \{3, 8\}$

The form of an intersection operation is

$R \cap S$

Where $R$ and $S$ are union-compatible relations (including more complex expressions that evaluate to relations).
Intersection: example

This query lists the employee IDs of all employees who are involved in a project. Note that no rename is required since the relations resulting from the Project operations are already union compatible.

\[ \Pi_{\text{empID}}(\text{Employee}) \cap \Pi_{\text{empID}}(\text{ProjectEmps}) \]
Difference

The difference of two sets $R$ and $S$ is the set including all members of $R$ that are not in $S$, for example.

\[
\{1, 3, 6, 8\} \setminus \{3, 5, 2, 8\} = \{1, 6\}
\]

The form of a difference operation is

\[R \setminus S\]

Where $R$ and $S$ are union-compatible relations (including more complex expressions that evaluate to relations).
This query lists the employee IDs of all employees who are not involved in a project. Note that no rename is required since the relations resulting from the Project operations are already union compatible.

\[ \Pi_{\text{emplID}}(Employee) \setminus \Pi_{\text{emplID}}(ProjectEmps) \]
Uses of relational algebra

Relational algebra allows us to formulate expressions without consideration for particular database implementations, SQL syntax, etc.

A key use of relational algebra is query optimisation, which is the next topic we will consider.