Set Theory

Set theory is a branch of mathematics that studies sets. Sets are a collection of objects.

Often, all members of a set have similar properties, such as odd numbers less than 10 or students in a tutorial group.

Objects in a set are called *elements* or *members* of a set.

A set is said to *contain its elements.*

Describing Sets

List all the members between braces. For example, **{a, b, c, d}**

e.g The set V of all vowels in the alphabet

V = {a,e,i,o,u}

| | denotes the cardinality of a set. For example **|V| = 5**

Set Equality

Two sets are equal if and only if they have the same elements. Order doesn't matter and repetition doesn't matter.

e.g {1,3,5} = {1,5,3} = {3,1,5} = …

e.g {1,2} = {1,1,2} = {1,1,1,1,2,2,2,2,2,2} = …

Sets

Sets usually group together elements with associated properties, but seemingly unrelated properties can also be listed as a set. For example, {2, e, Fred, Paris} is also a set, we just don't know how they are related.

It is sometimes inconvenient or impossible to describe a set by listing its elements, like the set of all integers less than 1 million. To do this, we use *Set Builder Notation*.

The set O of all positive integers less than 10 in set builder notation is:

O = { X | X is an odd integer less than 10} or **$O = \{X \mid X \in \mathbb{N} \land x < 10 \land x \% 2 == 1\}$**

Predicate

A predicate is sometimes used to indicate *set membership*.

A predicate **F(x)** will be true or false, depending on whether x belongs to a set.

An example:

> **{ x | x is a positive integer less than 4 } is the set {1,2,3}**
>
> If t is an element of the set {x | F(x)} then the statement F(t) is **true**

So if **F(x)** says **x % 2 = 0 {x | F(x) }** contains the set of all even numbers.

Here, **F(x)** is referred to as the *predicate*, and x the subject of the preposition.

Sometimes **F(x)** is also called a propositional function, as each choice of x produces a proposition.

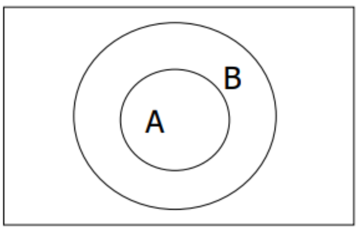
Set Notation

a ∈ A – a is an element of set A

a ∉ A – a is not an element of set A

∅ – the empty or null set, also represented by { }

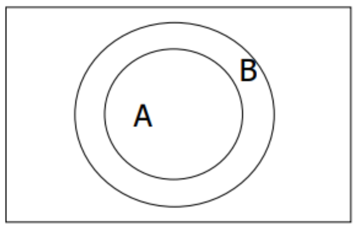
Subsets



A is a subset of B

$A \subset B$

A test that returns true iff $A \subset B$



A is a subset or equal to B

$A \subseteq B$

A test that returns true iff $A \subseteq B$

The Power Set

Given a set S, the *power set* is the set of all subsets of the set S. It is denoted by P(S).

The power set of **{0,1,2}** is **{∅, {0}, {1}, {2}, {0,1}, {0,2}, {1,2}, {0,1,2} }**

If a set has *n* elements, its power set has 2^n elements.

The power set contains sets, not numbers.

N-Tuples

n-tuples are not sets. The order of elements in a collection is sometimes important. But sets are unordered, so a different structure is needed. This is provided by ordered n-tuples.

E.g **<2,1,5>** is a *3-tuple.* It can also be noted as **(2,1,5)**

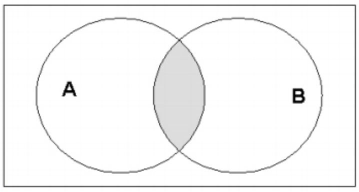
Two ordered n-tuples are equal if and only if each corresponding pair of their elements is equal.

{1, 3, 5} = {3, 1, 5} = *TRUE* for SETS

(1, 3, 5) = (3, 1, 5) = *FALSE* for N-TUPLES

We can use <> or () to denote tuples, but not {}

Set Operations

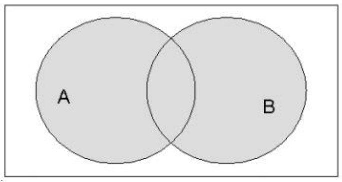


The **intersection** of A and B

$A \cap B$

Symbol like aNd

$A \cap B = \{x \mid x \in A \wedge x \in B\}$



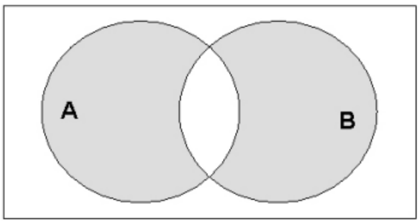
The union of A and B

Symbol like Union

$A \cup B$

$A \cup B = \{x \mid x \in A \vee x \in B\}$

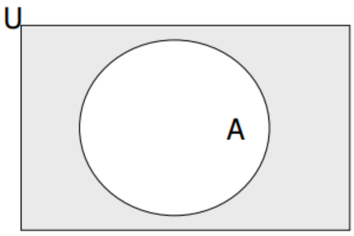
The set that contains those elements that are either in A, B, or in both



The symmetric difference of A and B

$A \oplus B = (A - B) \cup (B - A)$

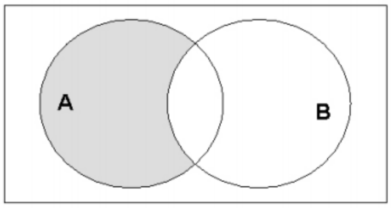
$A \oplus B = \{x \mid (x \in A \wedge x \notin B) \vee (x \in B \wedge x \notin A)\}$



The complement of A

$\overline{A}$

$\overline{A} = \{x \mid x \notin A\}$



The **difference** of A and B

$A - B$

$A - B = \{x \mid x \in A \wedge x \notin B\}$

List of operators:
- $\cup$ **union**
- $\cap$ **intersection**
- **– difference**
- $\overline{A}$ **complement**
- $\oplus$ **symmetric difference**

Cartesian Product

Let A and B be sets.

The *cartesian product* of A and B (A x B) is the set of all ordered pairs (tuples):

<a,b> where **a ∈ A** and **b ∈ B**

A = {0,1}, **B = {a,b,c}**

A x B = {<0,a>,<0,b>,<0,c>,<1,a>,<1,b>,<1,c>}

Relations

Relationships between elements of sets are represented using a structure called a *relation*.

A relation R is a subset of the cartesian product of the domains that define R.

Relations are the fundamental data structure used to store information in databases.

Properties of Relations

A and B are sets.

A binary relation R from A to B is a subset of the cartesian product A x B.

Notation:

- a R b denotes <a,b> ∈ R
- a is related to b by R

Representing Relations

Forename = {Angus, Alex, Matthew, Luke}

Surname = {McHaggis, Ackermann, Firix, Skywalker}

Names = {<Angus,McHaggis>,<Alex,Ackermann>,<Matthew,Firix>,<Luke,Skywalker>}

Querying Languages

There are two ways of querying a database:

- procedural (relation algebra)
    - based on set theory
    - sequence of operations
    - the output of each operation is the input to the next
- declarative (SQL)
    - describes the desired results (in terms of conditions)
    - the DBMS works out the operations

Relational Assignment

A query is made up of a sequence of operations of the form:

newRelation := UnaryOperation $_{\text{parameter}}$ (inputRelation)

Or (for a binary operator):

newRelation := inputRelation$_1$ Operator $_{\text{parameter}}$ inputRelation$_2$

Relation Operations

Select – pick rows from a relation by some condition

Project – pick columns by name

Join – connect two relations (usually by a foreign key)

Set Operations

Union – make a relation containing all the rows of two relations

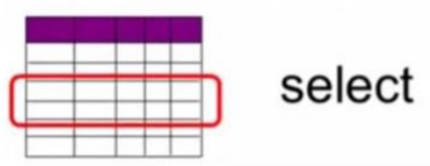
Intersection – pick the tuples which are common to two relations

Difference – pick the tuples which are in one relation but not another

Cartesian Product – pair off each of the tuple in one relation with those in another – creating a double sized row for each pair

Selection (σ)

Extract the tuples (rows) of a relation (table) which satisfy some condition on the value of their rows and return these as a relation (table view)



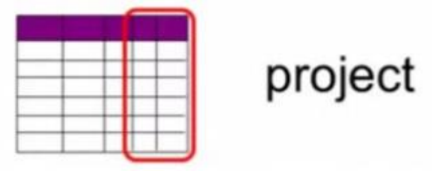
Syntax: $\sigma_{\text{condition}}$ (RelationName)

Example: Locals := $\sigma_{\text{city = "Glasgow"}}$ (Employee)

Would return all the employees that live in Glasgow in a table named “Locals”

Projection (Π)

Extracts the columns from the relation that match the given name. No attribute may occur more than once and duplicates will be removed.



Example: GenderSalary := $\Pi_{\text{(Gender, salary)}}$ (Employee)

Projection and selection can be combined:

$\Pi_{\text{(house, street)}}$ ($\sigma_{\text{city = "Glasgow"}}$ (Employee))

This would return the *house* and *street* of employees in Glasgow.

Union (∪)

Produces a relation which combines two relations into a new relation containing all of the tuples from each (removing duplicates)

The two relations must be "union compatible" i.e. have the same number of attributes drawn from the same domain.

Essentially works like an *OR* operator.

Union Compatibility

Tables must have the same columns if you wish to union them.

Intersection (∩)

Like union but returns tuples that are in both relations. Essentially works like the *AND* operator.

Requires union compatibility

Difference (-)

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

NonLocals := Employee – Locals

Requires union compatibility

Cartesian Product (X)

Cartesian Product A X B 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

Equi-join

$\sigma_{NI\# = ENI\#}$ (Employee x Dependent)

Cartesian Product followed by a selection is called a join (⋈) because it joins together two relations.

$$\text{Relation}_1 \bowtie_{A = B} \text{Relation}_2$$

$$\Leftrightarrow$$

$$\sigma_{A = B} (\text{Relation}_1 \times \text{Relation}_2)$$

Natural Join (⋈)

In its simplest form, the join of relations A and B pairs off the tuples of A and B so that identically named attributes from the relations have the same value. We now have two columns holding the same value, so we eliminate the duplicated common attributes to form the natural join or inner join.

Natural join is written as:

$$\text{Relation1} \bowtie \text{Relation2}$$

Summary of Operators

Applying to one relation:

Projection(attributes), selection(conditions)

Applying to two relations of identical structure:

Union, intersection, difference (no conditions)

Applying to two relations of different structure:

Cartesian product (no conditions), joins (conditions)

SQL (pronounced sequel)

SELECT – retrieves required columns from a relation

FROM – select the data from this table

WHERE – *retrieves rows which meet the* condition

In terms of relational algebra:

Project => **SELECT**

Select => **WHERE**

Selecting Unique Rows

```
SELECT DISTINCT [rows]
```

Will remove duplicates in a selection.

SQL Union

```
(SELECT name FROM Person WHERE age=15) UNION (SELECT name FROM Person
WHERE age=20);
```

Will combine the results.

SQL Cartesian Product

```
SELECT name, age, person.houseNum, aname, type, animal.houseNum, fedBy
FROM Person, Animal;
```

A cartesian product can be achieved by using the dot operator when selecting columns.

SQL Equi-Join

An equi-join is the cartesian product followed by a selection.

```
SELECT Person.name
FROM Person, Animal
WHERE Person.houseNum = Animal.houseNum;
```

Natural Join

The natural join is the product and a condition that all attributes of the same name are equated, then only one column for each pair of equated attributes is projected out.

Renaming

Renaming is needed when you have to use the same table twice in the same query.

The word *AS* can be used to define aliases for attributes or relations:

Relations:

```
SELECT p.name, age, p.houseNum,
       a.aname, type, a.houseNum, fedBy
FROM   Person AS p, Animal AS a
WHERE  p.houseNum = a.houseNum;
```

Attributes:

```
SELECT p.name AS pn, age, p.houseNum AS ph
       a.aname AS an, type, a.houseNum AS ah, fedBy
FROM   Person AS p, Animal AS a
WHERE  ph = ah;
```

WHERE Options

```
WHERE age > 21
WHERE age <> 18                      -- not equal to
WHERE age = 20 AND houseNum IS NOT NULL
WHERE houseNum IS NULL
WHERE age BETWEEN 20 AND 30
WHERE (type='dog') OR (type='cat')
WHERE name LIKE 'J%'                 -- names beginning with J
WHERE name LIKE 'J_ _ _ _'           -- 4 letter names beginning with J
WHERE name LIKE '_ _ M%'             -- names with M as the third letter
```

Operator Precedence

Different operators have higher precedence, they go in this order:

/, *, +, -

AND before OR

Ordering

Rows in a relation have no order (being a set). SQL can order the rows by specifying the order criteria.

```
SELECT *
FROM Person
ORDER BY age;
```

More examples of ORDER BY:

```
ORDER BY age DESC, name
ORDER BY age ASC
ORDER BY age, name ASC
```

Etc…

If you'd like to get better at SQL: https://www.w3schools.com/sql/

Design Patterns

Unfinished, but reusable designs for commonly occurring problem types.

SQL Design Patterns help you select the appropriate form of a query, such as basic query, equi-join or self-join.

Basic Query Pattern

"What are the names of the pets at house number 42?"

Basic Query pattern is used when all the data are in one table and rows need to be filtered based on a simple static condition.

```
SELECT aname
FROM Pets
WHERE houseNum = 42
```

Equi-Join Pattern

"List all information of employees and their department"

Equi-join pattern is used when all the data are in more that one table and rows need to be filtered based on data in other rows in the tables.

```
SELECT Employee_ID, Department_ID, Department_Name
FROM Employees as E, Departments as D
WHERE E.Department_ID = D.Department_ID
```

OR, IF THE ATTRIBUTE NAMES ARE NAMED THE SAME IN E AND D…

```
SELECT Employee_ID, Department_ID, Department_Name
FROM Employees NATURAL JOIN Departments
```

Self-Join Pattern

"Give the names of employees and their managers in dept 5"

Self-join pattern is used when all the data are in one table and rows need to be filtered based on data in other rows in the same table.

```
SELECT E.name, M.name
FROM Employee as E, Employee as M
WHERE E.supervisor = M.NI# AND E.deptno = 5
```

Here's a confusing table to remember it all:

Data to project is	Condition is	Pattern is
… in one table, on the same row	(…select only some rows)	Basic-query
…in two tables	Anything (but remember you need a join condition)	Equi-Join
…in one table, on different rows	"	Self-join

What are the names of all of the dogs living at house 42?

What are the names of the dogs living at the same house as (person) Jim?

What are the names of the dogs that live at the same house as (animal) Red?

Aggregate Functions

SELECT clause can contain expressions calculating data from the columns.

For example:

```
SELECT AVG(Salary) FROM Employee
```

Or

```
SELECT COUNT(DISTINCT Supervisor) FROM Employee
```

Here are a list of some useful aggregate functions:

- SUM (sum of values)
- MAX (maximum)
- MIN (minimum)
- AVG (average of values)
- COUNT (number of values)

They all return a value derived from all the values in a column, resulting in a table containing a single record summarising the Employee table.

Grouping

The previous aggregations selected the rows according to the WHERE condition, and produced a single answer.

GROUP BY allows the aggregations to be applied to groups of rows, according to the grouping of values in a column. It produces as many answers as there are identified groups.

Example:

```
SELECT type, AVG(age) AS ageAve

FROM Animal

GROUP BY type
```

Produces the average age for each animal type.

Grouping Pattern

Grouping pattern is used when you are looking for a description of a group of data in a relation, such as a count, maximum, minimum, average, etc and only one value per group is required.