Databases (LIX022B05)
SQL (2)

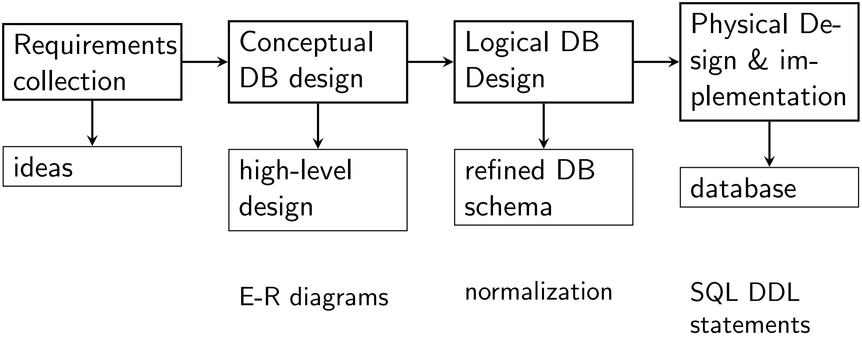
Instructor: Çağrı Çöltekin

c.coltekin@rug.nl

Information science/Informatiekunde

2012-10-01

Database Design Process



SQL basics: summary (1)

- SQL includes a data definition language (DDL) and data manipulation language (DML) statements as well as being a database query language.
- The DDL statements include **create table**, **alter table** and **drop table**
- The DML statements include **insert into**, **update** and **delete from** statements.
- The SQL query language is closely related to formal query language relational algebra.
- Relational algebra operations include, selection (σ), projection (π), Cartesian product ($\times$), natural join ($\bowtie$), other join operations such as outer joins, and set operations union, intersection, set difference.

SQL basics: summary (2)

- Basic form of SQL queries is:

  select attribute$_1$, $\cdots$, attribute$_N$
  from table_name$_1$, $\cdots$, table_name$_M$
  where condition;

- **from** clause lists the tables used in the query.
- **where** statement picks the rows we are interested in using predicates containing
  - comparisons: $=$, $<>$ $>$, $<$, $>=$ and $<=$.
  - sub-strings match operator **like**.
  - logical operators **and**, **or** and **not**.
- **select** clause picks the columns we are interested in.
- **select** and **where** may include arithmetic operations, and string operations, **upper**, **lower**, and **concat**

SQL basics: summary (3)

SQL queries can include

- We can sort the output of an SQL query by adding an **order by** clause at the end of our queries.
- A set of aggregate functions, **count**, **sum**, **avg**, **max** and **min** can be used to gather statistics about certain column(s) of a query.
- The results of aggregate functions can be grouped together by **group by** clause.
- Set operations **union**, intersection and difference (expect), can be used to combine the results of two queries.
- Sub-queries can be used in the **from** clause, or as an argument to **in**.

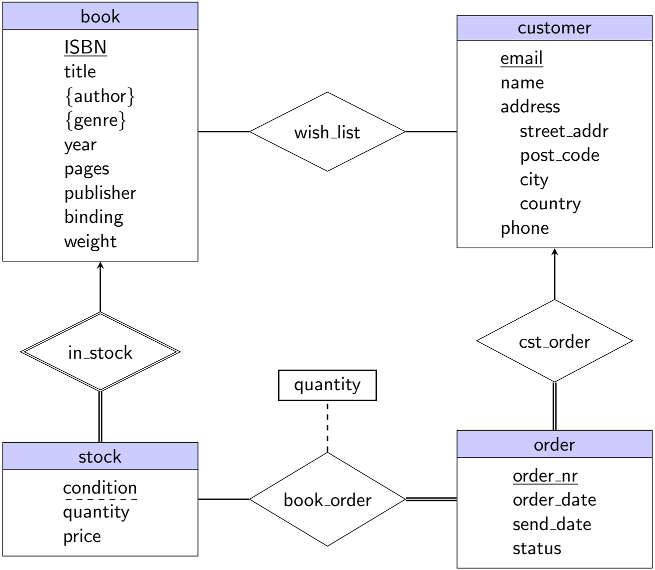
Homework 1: common problems

The most common problem is confusing entity sets with tables with eventual database tables.

E-R design is not the physical design of the database, you should try to describe the world with an abstraction consisting of the following components:

- Entity sets represent 'things', such as a person, or a book. Entity sets are typically represented with rectangles. Entity sets are **not** tables.
- Relationship sets represent relations between entities, such as person reads books. Relationship sets are not unnecessary details, only way two entities in an E-R model to be related is through a relationship set.
- Attributes are features of entities we are interested in such as name of a person. Relationship sets can also get attributes.

E-R design: a better solution



Homework 2: common problems

- A primary key (a key in general) can be a combination of multiple attributes (columns). For example, it is perfectly fine to define a (primary) key like (street, number, city, country).

Homework 2: common problems

- A primary key (a key in general) can be a combination of multiple attributes (columns). For example, it is perfectly fine to define a (primary) key like (street, number, city, country).
- 1NF is only about the values a particular column can take. Duplication of rows has nothing (directly) to do with 1NF. Having first and last names in a single column violates 1NF, but repeating these fields for each email address does not.

Homework 2: common problems

- A primary key (a key in general) can be a combination of multiple attributes (columns). For example, it is perfectly fine to define a (primary) key like
  (street, number, city, country).
- 1NF is only about the values a particular column can take. Duplication of rows has nothing (directly) to do with 1NF. Having first and last names in a single column violates 1NF, but repeating these fields for each email address does not.
- Functional dependencies are about the 'real world' you are modeling, they are not about a certain data set in a particular table.

Homework 2: common problems

- A primary key (a key in general) can be a combination of multiple attributes (columns). For example, it is perfectly fine to define a (primary) key like
(street, number, city, country).
- 1NF is only about the values a particular column can take. Duplication of rows has nothing (directly) to do with 1NF. Having first and last names in a single column violates 1NF, but repeating these fields for each email address does not.
- Functional dependencies are about the 'real world' you are modeling, they are not about a certain data set in a particular table.
- To be a foreign key, a set of attributes first needs to be a key. Defining a foreign key that references to non-key attributes on another table is wrong. (Even if your DBMS system allows it).

Homework 2: common problems

- A primary key (a key in general) can be a combination of multiple attributes (columns). For example, it is perfectly fine to define a (primary) key like (street, number, city, country).
- 1NF is only about the values a particular column can take. Duplication of rows has nothing (directly) to do with 1NF. Having first and last names in a single column violates 1NF, but repeating these fields for each email address does not.
- Functional dependencies are about the 'real world' you are modeling, they are not about a certain data set in a particular table.
- To be a foreign key, a set of attributes first needs to be a key. Defining a foreign key that references to non-key attributes on another table is wrong. (Even if your DBMS system allows it).
- A recommendation: try to write your SQL statements directly. You will learn better.

Rest of today...

- Some more reminders with additions: set operations and aggregate functions.
- More on joins.
- Null values.
- Indexes.
- Views.
- Access control.

Distinct values in SQL queries

genre	
title	genre
The Godfather	crime
The Godfather	drama
Seven Samurai	drama

What is the result of the query
select genre **from** genre;?

genre
crime
drama

or

genre
crime
drama
drama

Distinct values in SQL queries

genre	
title	genre
The Godfather	crime
The Godfather	drama
Seven Samurai	drama

What is the result of the query
select genre **from** genre;?

genre
crime
drama

or

genre
crime
drama
drama

- The left table (relation) is the theoretically correct answer, but SQL's answer is the right one.
- The reason is efficiency: reducing duplicates are an expensive process.
- But we can get the left table by adding **distinct** keyword to select clause. For example,
  select distinct genre **from** genre;

Set operations and distinct values

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010

genre	
title	genre
The Godfather	crime
The Godfather	drama
Seven Samurai	drama

- Set operations (**union**, **intersect**, **except**) always eliminate the duplicates.
- For example,

```
(select title    from movie)
union
(select title    as from genre);
```

will return:

title
The Godfather
Seven Samurai
Inception

Set comparison operators

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010

genre	
title	genre
The Godfather	crime
The Godfather	drama
Seven Samurai	drama

- We have already seen that we can test for set membership by **in** (or **not in**).
- For example, to find all movie titles without a genre assignment,

```
select *
from movie
where title not in (select title from genre);
```

- We can also use the following comparisons on sets:
  - **some**: the condition is true for at least one of the members.
  - **all** the condition true for all members.
  - **exists**: true if the set is not empty.

The operators also work on non-set (non-distinct) sub-queries.

Set comparison (examples)

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010

genre	
title	genre
The Godfather	crime
The Godfather	drama
Seven Samurai	drama

```
select * from movie
where year >= all (select year from movie);
```

Set comparison (examples)

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010

genre	
title	genre
The Godfather	crime
The Godfather	drama
Seven Samurai	drama

```
select * from movie
where year >= all (select year from movie);
```

Inception	2010

Set comparison (examples)

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010

genre	
title	genre
The Godfather	crime
The Godfather	drama
Seven Samurai	drama

```
select * from movie
where year >= all (select year from movie);
```

Inception	2010

```
select * from movie
where year > some (select year from movie);
```

Set comparison (examples)

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010

genre	
title	genre
The Godfather	crime
The Godfather	drama
Seven Samurai	drama

```
select * from movie
where year >= all (select year from movie);
```

Inception	2010

```
select * from movie
where year > some (select year from movie);
```

The Godfather	1972
Inception	2010

Set comparison (examples)

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010

genre	
title	genre
The Godfather	crime
The Godfather	drama
Seven Samurai	drama

```
select * from movie
where year >= all (select year from movie);
```

Inception	2010

```
select * from movie
where year > some (select year from movie);
```

The Godfather	1972
Inception	2010

```
select * from movie
where not exists (select * from genre where movie.title = genre.title)
```

Set comparison (examples)

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010

genre	
title	genre
The Godfather	crime
The Godfather	drama
Seven Samurai	drama

```
select * from movie
where year >= all (select year from movie);
```

Inception	2010

```
select * from movie
where year > some (select year from movie);
```

The Godfather	1972
Inception	2010

```
select * from movie
where not exists (select * from genre where movie.title = genre.title)
```

Inception	2010

Aggregate functions and **having** clause

genre	
title	genre
The Godfather	crime
The Godfather	drama
Seven Samurai	drama

We use **group by** to group the output of the aggregate functions(**count**, **sum**, **avg**, **max**, **min**).
For example

```
select genre, count(title) as count from genre group by genre;
```

Aggregate functions and **having** clause

genre	
title	genre
The Godfather	crime
The Godfather	drama
Seven Samurai	drama

We use **group by** to group the output of the aggregate functions(**count**, **sum**, **avg**, **max**, **min**).
For example

```
select genre, count(title) as count from genre group by genre;
```

genre	**count**
drama	2
crime	1

Aggregate functions and **having** clause

genre	
title	genre
The Godfather	crime
The Godfather	drama
Seven Samurai	drama

We use **group by** to group the output of the aggregate functions(**count**, **sum**, **avg**, **max**, **min**).
For example
select genre, **count**(title) **as count from** genre **group by** genre;

genre	**count**
drama	2
crime	1

Sometimes we want to restrict the groups, this can be done by **having** clause.
select genre, **count**(title) **as count from** student **group by year**
having count(title) >= 2;

Aggregate functions and **having** clause

genre	
title	genre
The Godfather	crime
The Godfather	drama
Seven Samurai	drama

We use **group by** to group the output of the aggregate functions(**count**, **sum**, **avg**, **max**, **min**).
For example

select genre, **count**(title) **as count from** genre **group by** genre;

genre	count
drama	2
crime	1

Sometimes we want to restrict the groups, this can be done by **having** clause.

select genre, **count**(title) **as count from** student **group by year**
having count(title) >= 2;

year	count
drama	2

Reasoning with **null** values

Null values create a number of difficult cases in relational database theory.

- Arithmetic expressions involving **null** are **null** ($1 +$ **null** $=$ **null**).
- Any comparison (like $1 =$ **null**, $1 <$ **null**) involving nulls results in a third truth value: unknown.
- This includes the comparison **null** $=$ **null**, except for set operations and for **distinct**.
- Expressions is **null** or is **not null** can be used to test if a value is null or not.
- Logical operations with unknown values:

true **and** unknown =
false **and** unknown =
true **or** unknown =
false **or** unknown =
not unknown =

Reasoning with **null** values

Null values create a number of difficult cases in relational database theory.

- Arithmetic expressions involving **null** are **null** ($1 + \textbf{null} = \textbf{null}$).
- Any comparison (like $1 = \textbf{null}$, $1 < \textbf{null}$) involving nulls results in a third truth value: unknown.
- This includes the comparison **null** = **null**, except for set operations and for **distinct**.
- Expressions is **null** or is **not null** can be used to test if a value is null or not.
- Logical operations with unknown values:

true **and** unknown	=	**unknown**
false **and** unknown	=	
true **or** unknown	=	
false **or** unknown	=	
not unknown	=	

Reasoning with **null** values

Null values create a number of difficult cases in relational database theory.

- Arithmetic expressions involving **null** are **null** ($1 +$ **null** $=$ **null**).
- Any comparison (like $1 =$ **null**, $1 <$ **null**) involving nulls results in a third truth value: unknown.
- This includes the comparison **null** $=$ **null**, except for set operations and for **distinct**.
- Expressions is **null** or is **not null** can be used to test if a value is null or not.
- Logical operations with unknown values:

true **and** unknown	=	**unknown**
false **and** unknown	=	**false**
true **or** unknown	=	
false **or** unknown	=	
not unknown	=	

Reasoning with **null** values

Null values create a number of difficult cases in relational database theory.

- Arithmetic expressions involving **null** are **null** ($1 + \textbf{null} = \textbf{null}$).
- Any comparison (like $1 = \textbf{null}$, $1 < \textbf{null}$) involving nulls results in a third truth value: unknown.
- This includes the comparison $\textbf{null} = \textbf{null}$, except for set operations and for **distinct**.
- Expressions is **null** or is **not null** can be used to test if a value is null or not.
- Logical operations with unknown values:

$$
\begin{aligned}
\text{true } \textbf{and} \text{ unknown} &= \textbf{unknown} \\
\text{false } \textbf{and} \text{ unknown} &= \textbf{false} \\
\text{true } \textbf{or} \text{ unknown} &= \textbf{true} \\
\text{false } \textbf{or} \text{ unknown} &= \\
\textbf{not} \text{ unknown} &=
\end{aligned}
$$

Reasoning with **null** values

Null values create a number of difficult cases in relational database theory.

- Arithmetic expressions involving **null** are **null** ($1 +$ **null** $=$ **null**).
- Any comparison (like $1 =$ **null**, $1 <$ **null**) involving nulls results in a third truth value: unknown.
- This includes the comparison **null** $=$ **null**, except for set operations and for **distinct**.
- Expressions is **null** or is **not null** can be used to test if a value is null or not.
- Logical operations with unknown values:

true **and** unknown	=	**unknown**
false **and** unknown	=	**false**
true **or** unknown	=	**true**
false **or** unknown	=	**unknown**
not unknown	=	

Reasoning with **null** values

Null values create a number of difficult cases in relational database theory.

- Arithmetic expressions involving **null** are **null** ($1 +$ **null** $=$ **null**).
- Any comparison (like $1 =$ **null**, $1 <$ **null**) involving nulls results in a third truth value: unknown.
- This includes the comparison **null** $=$ **null**, except for set operations and for **distinct**.
- Expressions is **null** or is **not null** can be used to test if a value is null or not.
- Logical operations with unknown values:

true **and** unknown	=	**unknown**
false **and** unknown	=	**false**
true **or** unknown	=	**true**
false **or** unknown	=	**unknown**
not unknown	=	**unknown**

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010
The Hobbit	null

null values and aggregate functions

All aggregate functions ignore
the null values (**count**($*$) is an exception). Examples:

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010
The Hobbit	null

null values and aggregate functions

All aggregate functions ignore
the null values (**count**($*$) is an exception). Examples:

- **count**($*$):
  select count($*$) movie; $\Rightarrow$

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010
The Hobbit	null

null values and aggregate functions

All aggregate functions ignore
the null values (**count**(∗) is an exception). Examples:

- **count**(∗):
  select count(∗) movie; ⇒ 4

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010
The Hobbit	null

null values and aggregate functions

All aggregate functions ignore
the null values (**count**(∗) is an exception). Examples:

- **count**(∗):
  select count(∗) movie; ⇒ 4
- **count**(): count
  select count(**year**) **from** movie; ⇒

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010
The Hobbit	null

null values and aggregate functions

All aggregate functions ignore
the null values (**count**(∗) is an exception). Examples:

- **count**(∗):
  select count(∗) movie; ⇒ 4
- **count**(): count
  select count(**year**) **from** movie; ⇒ 3

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010
The Hobbit	null

null values and aggregate functions

All aggregate functions ignore
the null values (**count**(∗) is an exception). Examples:

- **count**(∗):
  select count(∗) movie; ⇒ 4
- **count**(): count
  select count(**year**) **from** movie; ⇒ 3
- **sum**: total
  select sum(**year**) **from** movie; ⇒

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010
The Hobbit	null

null values and aggregate functions

All aggregate functions ignore
the null values (**count**(∗) is an exception). Examples:

- **count**(∗):
  select count(∗) movie; ⇒ 4
- **count**(): count
  select count(**year**) **from** movie; ⇒ 3
- **sum**: total
  select sum(**year**) **from** movie; ⇒ 5936

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010
The Hobbit	null

null values and aggregate functions

All aggregate functions ignore
the null values (**count**(*) is an exception). Examples:

- **count**(*):
  select count(*) movie; ⇒ 4
- **count**(): count
  select count(**year**) **from** movie; ⇒ 3
- **sum**: total
  select sum(**year**) **from** movie; ⇒ 5936
- **avg**: average
  select average(**year**) **from** movie; ⇒

null values and aggregate functions

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010
The Hobbit	null

All aggregate functions ignore the null values (**count**(∗) is an exception). Examples:

- **count**(∗):
  select count(∗) movie; ⇒ 4
- **count**(): count
  select count(**year**) **from** movie; ⇒ 3
- **sum**: total
  select sum(**year**) **from** movie; ⇒ 5936
- **avg**: average
  select average(**year**) **from** movie; ⇒ 1978.67

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010
The Hobbit	null

null values and aggregate functions

All aggregate functions ignore
the null values (**count**(∗) is an exception). Examples:

- **count**(∗):
  select count(∗) movie; ⇒ 4
- **count**(): count
  select count(**year**) **from** movie; ⇒ 3
- **sum**: total
  select sum(**year**) **from** movie; ⇒ 5936
- **avg**: average
  select average(**year**) **from** movie; ⇒ 1978.67
- **min**: minimum
  select min(**year**) **from** movie; ⇒

null values and aggregate functions

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010
The Hobbit	null

All aggregate functions ignore
the null values (**count**(∗) is an exception). Examples:

- **count**(∗):
  select count(∗) movie; ⇒ 4
- **count**(): count
  select count(**year**) **from** movie; ⇒ 3
- **sum**: total
  select sum(**year**) **from** movie; ⇒ 5936
- **avg**: average
  select average(**year**) **from** movie; ⇒ 1978.67
- **min**: minimum
  select min(**year**) **from** movie; ⇒ 1972

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010
The Hobbit	null

null values and aggregate functions

All aggregate functions ignore
the null values (**count**(∗) is an exception). Examples:

- **count**(∗):
  select count(∗) movie; ⇒ 4
- **count**(): count
  select count(**year**) **from** movie; ⇒ 3
- **sum**: total
  select sum(**year**) **from** movie; ⇒ 5936
- **avg**: average
  select average(**year**) **from** movie; ⇒ 1978.67
- **min**: minimum
  select min(**year**) **from** movie; ⇒ 1972
- **max**: maximum
  select max(**year**) **from** movie; ⇒

movie	
title	year
The Godfather	1972
Seven Samurai	1954
Inception	2010
The Hobbit	null

null values and aggregate functions

All aggregate functions ignore
the null values (**count**(∗) is an exception). Examples:

- **count**(∗):
  select count(∗) movie; ⇒ 4
- **count**(): count
  select count(**year**) **from** movie; ⇒ 3
- **sum**: total
  select sum(**year**) **from** movie; ⇒ 5936
- **avg**: average
  select average(**year**) **from** movie; ⇒ 1978.67
- **min**: minimum
  select min(**year**) **from** movie; ⇒ 1972
- **max**: maximum
  select max(**year**) **from** movie; ⇒ 2010

Queries on multiple tables

book		
bID	**pages**	**year**
1	130	1995
2	544	1995
3	213	2005
4	210	2012

orders		
cID	**bID**	**qty**
1	1	1
1	2	1
3	1	3
4	3	1

```
select book.bID, book.year, orders.cID, orders.qty
from book, orders
where book.bID = orders.bID;
```

Queries on multiple tables

book		
bID	**pages**	**year**
1	130	1995
2	544	1995
3	213	2005
4	210	2012

orders		
cID	**bID**	**qty**
1	1	1
1	2	1
3	1	3
4	3	1

```
select book.bID, book.year, orders.cID, orders.qty
from book, orders
where book.bID = orders.bID;
```

bID	year	cID	qty
1	1995	1	1
2	1995	1	1
1	1995	3	3
3	2005	4	1

Queries on multiple tables

book		
bID	**pages**	**year**
1	130	1995
2	544	1995
3	213	2005
4	210	2012

orders		
cID	**bID**	**qty**
1	1	1
1	2	1
3	1	3
4	3	1

select book.bID, book.**year**, orders.cID, orders.qty
from book, orders
where book.bID = orders.bID;

bID	year	cID	qty
1	1995	1	1
2	1995	1	1
1	1995	3	3
3	2005	4	1

Note: if you do not specify a **where** clause, you get the Cartesian product.

Natural join

book		
bID	**pages**	**year**
1	130	1995
2	544	1995
3	213	2005
4	210	2012

orders		
cID	**bID**	**qty**
1	1	1
1	2	1
3	1	3
4	3	1

The previous example was doing a natural join implicitly, we can get the same effect with **natural join** expression.

Natural join

book		
bID	**pages**	**year**
1	130	1995
2	544	1995
3	213	2005
4	210	2012

orders		
cID	**bID**	**qty**
1	1	1
1	2	1
3	1	3
4	3	1

The previous example was doing a natural join implicitly, we can get the same effect with **natural join** expression.

```
select bID, year, cID, qty
from book natural join orders;
```

Natural join

book		
bID	**pages**	**year**
1	130	1995
2	544	1995
3	213	2005
4	210	2012

orders		
cID	**bID**	**qty**
1	1	1
1	2	1
3	1	3
4	3	1

The previous example was doing a natural join implicitly, we can get the same effect with **natural join** expression.

```
select bID, year, cID, qty
from book natural join orders;
```

Result is (again) the same

bID	**year**	**cID**	**qty**
1	1995	1	1
2	1995	1	1
1	1995	3	3
3	2005	4	1

Natural join

book		
bID	**pages**	**year**
1	130	1995
2	544	1995
3	213	2005
4	210	2012

orders		
cID	**bID**	**qty**
1	1	1
1	2	1
3	1	3
4	3	1

The previous example was doing a natural join implicitly, we can get the same effect with **natural join** expression.

```
select bID, year, cID, qty
from book natural join orders;
```

Result is (again) the same

bID	year	cID	qty
1	1995	1	1
2	1995	1	1
1	1995	3	3
3	2005	4	1

You can join more than two tables using the same syntax:
t_1 **natural join** t_2 **natural join** t_3 ...

Natural join: accidental column match

student			
sID	**dept**	**year**	**aID**
1	IK	1	1
2	CIW	2	2
3	IK	3	1
4	CIW	2	3

advisor		
aID	**dept**	**phone**
1	CIW	1111
2	IK	2222
3	IK	3333
4	CIW	4444

Natural join: accidental column match

student			
sID	**dept**	**year**	**aID**
1	IK	1	1
2	CIW	2	2
3	IK	3	1
4	CIW	2	3

advisor		
aID	**dept**	**phone**
1	CIW	1111
2	IK	2222
3	IK	3333
4	CIW	4444

```
select sID, student.dept, aID, phone
from student natural join advisor;
```

sID	student.dept	aID	phone
3	IK	1	1111
4	CIW	3	3333

Natural join: accidental column match

student			
sID	**dept**	**year**	**aID**
1	IK	1	1
2	CIW	2	2
3	IK	3	1
4	CIW	2	3

advisor		
aID	**dept**	**phone**
1	CIW	1111
2	IK	2222
3	IK	3333
4	CIW	4444

```
select sID, student.dept, aID, phone
from student join advisor using (aID);
```

sID	student.dept	aID	phone
1	IK	1	1111
2	CIW	2	2222
3	IK	1	1111
4	CIW	3	5555

Natural join: arbitrary column expressions

student			
ID	**s_dept**	**year**	**aID**
1	IK	1	1
2	CIW	2	2
3	IK	3	1
4	CIW	2	3

advisor		
ID	**a_dept**	**phone**
1	CIW	1111
2	IK	2222
3	IK	3333
4	CIW	4444

Natural join: arbitrary column expressions

student			
ID	**s_dept**	**year**	**aID**
1	IK	1	1
2	CIW	2	2
3	IK	3	1
4	CIW	2	3

advisor		
ID	**a_dept**	**phone**
1	CIW	1111
2	IK	2222
3	IK	3333
4	CIW	4444

```
select student.ID, s_dept, advisor.ID, phone
from student natural join advisor;
```

studetn.ID	s_dept	advisor.ID	phone
1	IK	1	1111
2	CIW	2	2222
3	IK	3	3333
4	CIW	4	4444

Natural join: arbitrary column expressions

student			
ID	**s_dept**	**year**	**aID**
1	IK	1	1
2	CIW	2	2
3	IK	3	1
4	CIW	2	3

advisor		
ID	**a_dept**	**phone**
1	CIW	1111
2	IK	2222
3	IK	3333
4	CIW	4444

```
select student.ID, s_dept, advisor.ID, phone
from student join advisor on student.aID = advisor.ID;
```

studetn.ID	s_dept	advisor.ID	phone
1	IK	1	1111
2	CIW	2	2222
3	IK	1	1111
4	CIW	3	3333

Natural join: arbitrary column expressions

student			
ID	s_dept	year	aID
1	IK	1	1
2	CIW	2	2
3	IK	3	1
4	CIW	2	3

advisor		
ID	a_dept	phone
1	CIW	1111
2	IK	2222
3	IK	3333
4	CIW	4444

```
select student.ID, s_dept, advisor.ID, phone
from student join advisor on student.aID = advisor.ID;
```

studetn.ID	s_dept	advisor.ID	phone
1	IK	1	1111
2	CIW	2	2222
3	IK	1	1111
4	CIW	3	3333

on clause can take any expression allowed in a **where** clause.

Join conditions with an equation are sometimes called equi-join and join conditions with arbitrary comparisons are called θ-join (theta-join).

Joins so far...

- The join expressions we saw so far are called inner joins (the SQL **join** statements can also be optionally prepended by **inner** to make this explicit).
- The usual (inner) joins do not include rows that do not meet the join condition. For example, the advisor ID = 4, never showed up in our join examples.
- In cases where this is not desirable, outer joins can be used. (We will discuss outer joins in a few minutes).

Outer join

student			
sID	**s_dept**	**year**	**aID**
1	IK	1	1
2	CIW	2	2
3	IK	3	1
4	CIW	2	null

advisor		
aID	**a_dept**	**phone**
1	CIW	1111
2	IK	2222
3	IK	3333
4	CIW	4444

- The tuples without a matching join attribute are not included in inner joins. For the (modified) example, the advisors with ID 3 and 4, and the student with ID 4 will not show up in a inner join.
- There are cases where we may want to list,
  - all students (including the ones without an assigned advisor)
  - all advisors (including the ones who do not advise a student at the moment),
  - both

  in the joined result.
- Outer join operation allows preserving all tuples from one or both sides by filling **null** values for the missing attributes.

Outer join types

Outer joins in SQL are specified with prepending **join** keyword with **outer**. The join expression becomes

- t_1 **left outer join** t_2 preserves all tuples from the table specified on the left (t_1).
- t_1 **right outer join** t_2 preserves all tuples from the table specified on the right (t_2).
- t_1 **full outer join** t_2 preserves all tuples from the tables on both sides (both t_1 and t_2).
- using the **natural** keyword before join condition joins using the attributes with matching names on both tables.
- as with inner joins, we can use **using** or **on** to specify the attributes to use in the join operation.

Left outer join

student			
sID	**s_dept**	**year**	**aID**
1	IK	1	1
2	CIW	2	2
3	IK	3	1
4	CIW	2	null

advisor		
aID	**a_dept**	**phone**
1	CIW	1111
2	IK	2222
3	IK	3333
4	CIW	4444

```
select * from student natural join advisor;
```

sID	s_dept	year	aID	a_dept	phone
1	IK	1	1	CIW	1111
2	CIW	2	2	IK	2222
3	IK	3	1	CIW	1111

Left outer join

student			
sID	**s_dept**	**year**	**aID**
1	IK	1	1
2	CIW	2	2
3	IK	3	1
4	CIW	2	null

advisor		
aID	**a_dept**	**phone**
1	CIW	1111
2	IK	2222
3	IK	3333
4	CIW	4444

```
select * from student natural left outer join advisor;
```

sID	s_dept	year	aID	a_dept	phone
1	IK	1	1	CIW	1111
2	CIW	2	2	IK	2222
3	IK	3	1	CIW	1111
4	CIW	2	null	null	null

Right outer join

student			
sID	**s_dept**	**year**	**aID**
1	IK	1	1
2	CIW	2	2
3	IK	3	1
4	CIW	2	null

advisor		
aID	**a_dept**	**phone**
1	CIW	1111
2	IK	2222
3	IK	3333
4	CIW	4444

select ∗ **from** student **join** advisor **using** (aID);

sID	s_dept	year	aID	a_dept	phone
1	IK	1	1	CIW	1111
2	CIW	2	2	IK	2222
3	IK	3	1	CIW	1111

Right outer join

student			
sID	**s_dept**	**year**	**aID**
1	IK	1	1
2	CIW	2	2
3	IK	3	1
4	CIW	2	null

advisor		
aID	**a_dept**	**phone**
1	CIW	1111
2	IK	2222
3	IK	3333
4	CIW	4444

select * **from** student **right outer join** advisor **using** (aID);

sID	s_dept	year	aID	a_dept	phone
1	IK	1	1	CIW	1111
2	CIW	2	2	IK	2222
3	IK	3	1	CIW	1111
null	null	null	3	IK	3333
null	null	null	4	CIW	4444

Full outer join

student			
sID	**s_dept**	**year**	**aID**
1	IK	1	1
2	CIW	2	2
3	IK	3	1
4	CIW	2	null

advisor		
aID	**a_dept**	**phone**
1	CIW	1111
2	IK	2222
3	IK	3333
4	CIW	4444

select * **from** student **join** advisor **on** student.aID = advisor.aID;

sID	s_dept	year	aID	a_dept	phone
1	IK	1	1	CIW	1111
2	CIW	2	2	IK	2222
3	IK	3	1	CIW	1111

Full outer join

student			
sID	**s_dept**	**year**	**aID**
1	IK	1	1
2	CIW	2	2
3	IK	3	1
4	CIW	2	null

advisor		
aID	**a_dept**	**phone**
1	CIW	1111
2	IK	2222
3	IK	3333
4	CIW	4444

```
select * from student full outer join advisor on student.aID = advisor.aID;
```

sID	s_dept	year	aID	a_dept	phone
1	IK	1	1	CIW	1111
2	CIW	2	2	IK	2222
3	IK	3	1	CIW	1111
4	CIW	2	null	null	null
null	null	null	3	IK	3333
null	null	null	4	CIW	4444

MySQL note: MySQL does not support **full outer join**. Typical trick is to take the union of left and right outer joins. For example:

```
(select * from student natural left outer join advisor)
union
(select * from student natural right outer join advisor);
```

Joins: summary

- A join is a combination of rows from multiple tables according to one or more related columns on each table.
- In SQL a join can either be specified implicitly in **where** clause, or explicitly in **from** clause.
- Inner joins are join operations which select only the tuples that meet the join condition.
- Outer joins allow tuples that do not meet the join condition to be included from one or both tables being joined.
- A natural join uses matching attribute names from each table.
- Joins can be restricted to certain columns with a **using** clause, or full join conditions can be specified using **on**.

Summary

Today we have discussed:

- A bit of database design, using homework 1 as a case study.
- More on set operations and aggregation.
- Null values and the problems associated with them.
- Joining tables: inner/outer joins.

What is next?

- Now: discussion of homework 2.
- Access control.
- Indexes.
- Triggers.
- Stored functions/procedures.
- Reading for next week: Intermediate SQL (Chapter 4, if you haven't) and Sections 5.2 and 5.3 (on stored procedures and triggers).
- Lab/Homework: more SQL exercises, will be posted today, due next week Thursday, 2012-10-11 13:00.