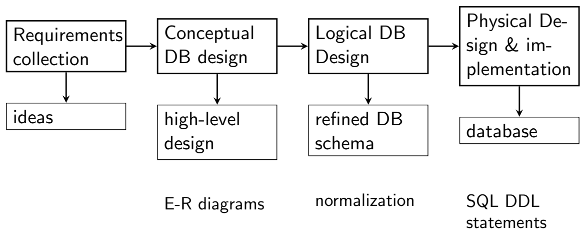
Database Management Systems (LIX022B05)

Instructor: Çağrı Çöltekin
c.coltekin@rug.nl

Information science/Informatiekunde

September 30, 2013

Database Design Process



SQL basics: summary

- 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: query statements

- 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: more on queries

- 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**.

This week

- Distinct rows in SQL queries.
- Set comparison operators.
- More on aggregate functions.
- SQL and null values.
- SQL statements using multiple tables: joins.
- Views.
- Indexes

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 correct answer according to theory, 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 **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);
```

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;
```

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** yield **null** (1 + **null** = **null**).
- Any comparison (like 1 = **null**, 1 < **null**) involving nulls results in a third truth value: unknown (or **null**).
- This includes the comparison **null** = **null**, except for set operations and for **distinct**.
- For comparisons involving **null** use is **null** or is **not null**.
- Logical operations with unknown values:

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

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:

- **select count**(*) movie; ⇒ 4
- **select count**(**year**) **from** movie; ⇒ 3
- **select sum**(**year**) **from** movie; ⇒ 5936
- **select avg**(**year**) **from** movie; ⇒ 1978.67
- **select min**(**year**) **from** movie; ⇒ 1954
- **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;
```

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

```
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				advisor		
ID	s_dept	year	aID	ID	a_dept	phone
1	IK	1	1	1	CIW	1111
2	CIW	2	2	2	IK	2222
3	IK	3	1	3	IK	3333
4	CIW	2	3	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 called equi-join and join with arbitrary comparisons are called θ-join (theta-join).

Outline SQL: distinct SQL: set comparisons SQL: having Null values SQL: joins Views Indexes Wrapping up

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).

Outline SQL: distinct SQL: set comparisons SQL: having Null values SQL: joins Views Indexes Wrapping up

Outer join

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

Outline SQL: distinct SQL: set comparisons SQL: having Null values SQL: joins Views Indexes Wrapping up

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.

Outline SQL: distinct SQL: set comparisons SQL: having Null values SQL: joins Views Indexes Wrapping up

Left outer join

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

Outline SQL: distinct SQL: set comparisons SQL: having Null values SQL: joins Views Indexes Wrapping up

Right outer join

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

Outline SQL: distinct SQL: set comparisons SQL: having Null values SQL: joins Views Indexes Wrapping up

Full outer join

student				advisor		
sID	s_dept	year	aID	aID	a_dept	phone
1	IK	1	1	1	CIW	1111
2	CIW	2	2	2	IK	2222
3	IK	3	1	3	IK	3333
4	CIW	2	null	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);
```


Outline SQL: distinct SQL: set comparisons SQL: having Null values SQL: joins Views Indexes Wrapping up

Joins: a summary

- **inner** joins join two tables where only rows match are included.
- join condition can be
  - **natural**, where matching column names are matched for equality.
  - specified with keyword **using** to specify column names that are expected to match.
  - specified with keyword **on**, where one can specify arbitrary conditions.
- inner joins join can also be specified using conditions in **where** clause.
- **outer** joins allow non-matching rows from one or both tables to be included in the result.
  - **left outer join** preserves all rows from the left table.
  - **right outer join** preserves all rows from the right table.
  - **full outer join** preserves all rows from both tables.

Views

Outcome of a query can be presented by a DBMS as a virtual table, called a view.

- A views content is calculated each time it is accessed.
- Views are simply result of a query. They can be a subset of a table, or a join over multiple tables.
- Views can be used to present derived attributes, such as aggregated values in a virtual table.
- Views can provide a limited 'view' of the underlying data.
- They can be used to set access restrictions.
- So-called 'materialized views' can be used to speed up frequent queries.

Operations on views

- To create a view:
  create view view_name **as** sql_query; For example:
  ```
  create view ik_students as
    select *
    from student
    where dept = 'IK';
  ```
- To drop a view: **drop view** view_name;
- To modify a view: **alter view** view_name ...

Updating a view

Views can be used just like tables, except updates on views are constrained. The views with an underlying query that meet the following conditions are updatable.

- The **from** clause should reference only one table.
- If a column does not have a **default** value and not allowed to take null values, then it has to be listed in **select** clause.
- The **select** clause should only list the column names: no arithmetic operations, no aggregation.
- No **group by** or **order by** clause is allowed.

Some DBMSes are slightly more liberal on updating views, but the above rules cover the common base.

Indexes: motivation

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

How would a DBMS find the results of the following query?

```
select * from book where bID = 100;
```

If we had 2,000,000 books with no bID 100, we would need 2,000,000 comparisons, 1M comparisons on average.
How about this one?

```
select * from book natural join orders;
```

Assuming we have 2,000 orders, we need 2,000 × 2,000,000 comparisons (worst case, 2,000 × 1,000,000 on average).

The problem is not only the CPU time for the comparisons, but also many disk reads, not necessarily in sequential disk blocks.

Indexes

Indexes allow fast retrieval of a single records, or a range of records.

- Indexes can be created for any set of attributes in a table.
- You can require index key to be unique.
- Typically, DBMSs will create a unique index for the primary key of a table automatically.
- And some DBMSs (e.g., MySQL) will automatically create indexes for foreign keys as well.

Index types

Common indexing mechanism are based on hash indexes or B-trees (or B+-trees).

- Indexes speed up the retrieval considerably.
- With a hash index, access time is constant (typically single disk read).
- Access time with B-trees are proportional to logarithm of the number of records (This translates to approximately 15 comparisons for 20,000 records, 26 for 40,000,000).
- A hash index only allows you retrieve a single value.
- With a B-tree index ranges of values can also be retrieved efficiently.
- Some DBMSs will allow you to choose the type of index (read: some will not).

Creating indexes

Index creation is not part of the SQL standard, but all DBMSs provide a way to create them.

- Indexes can be created during the table creation,
  ```
  create table orders (oID int,
                       cID int,
                       bID int,
                       qty int,
                       prinry key (oID),
                       index(bID));
  ```
- or, later using the syntax
  create unique index bid_index **on** orders(bid);
- and, you can delete an index using,
  drop index bid_index;
  or using **alter table** if index is unnamed.

Indexes: an example

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

Which index would speed up this query?

```
select * from book where bID = 100;
```

- **create index** book_id_index **on** book (bID);

Do we need more indexes for the following query?

```
select * from book natural join orders;
```

- It is tempting to create an index for orders.bID but it will not improve the query time unless we have more rows in orders table than in the book table. Why?

Indexes: summary

- Indexes are used to speed access to particular rows.
- Indexes will speed up the queries, as well as update statements that refer to particular rows.
- Indexes can use one or more column values as keys.
- Why not creating indexes for every column?
  - unnecessary indexes waste storage
  - for updates and inserts indexes create an additional overhead: as well as the data, the relevant indexes must be updated.

Summary

Summing up all of today:

- Distinct rows in SQL queries.
- Set comparison operators.
- More on aggregate functions.
- SQL and null values.
- SQL statements using multiple tables: joins.
- Views.

What is next?

- Access control.
- Stored functions/procedures.
- Triggers.
- Reading for next week: Intermediate SQL (Chapter 4, if you haven't) and Sections 5.2 and 5.3 (on stored procedures and triggers).