Database Management Systems (LIX022B05)

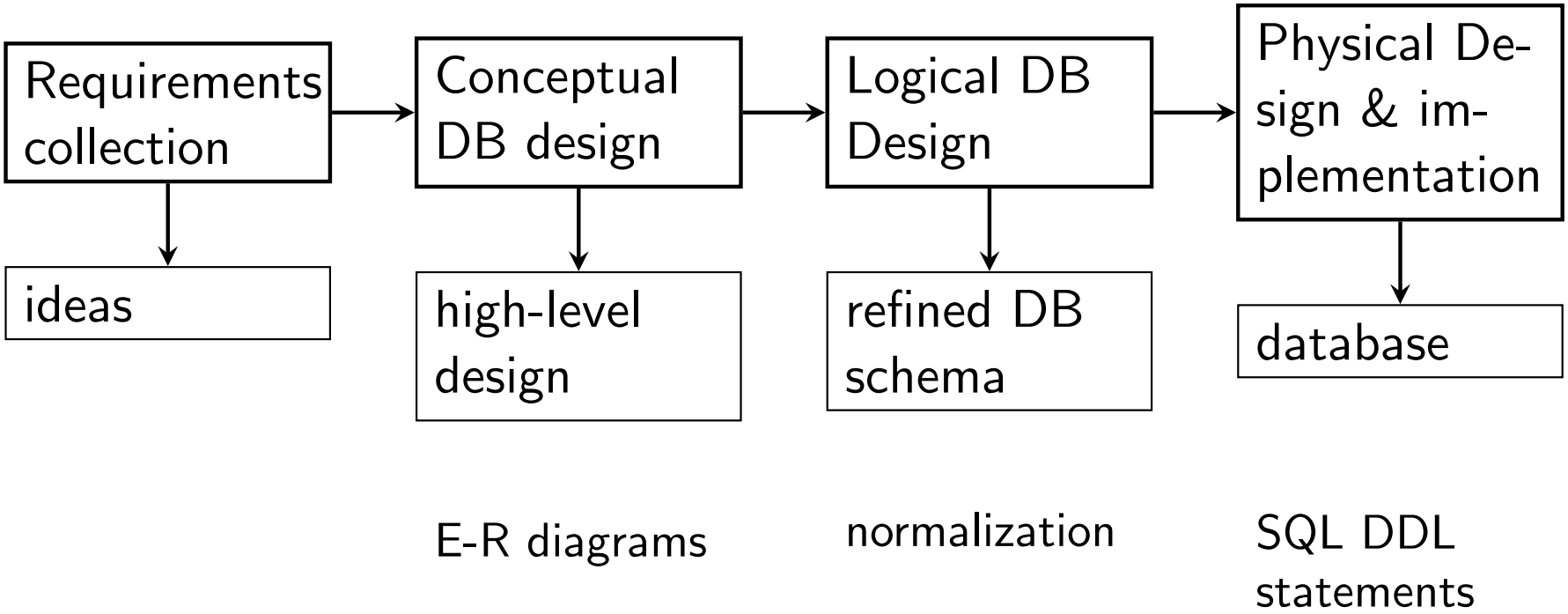
Instructor: Çağrı Çöltekin

c.coltekin@rug.nl

Information science/Informatiekunde

October 7, 2013

Database Design Process



Conceptual (E-R) design: things to remember

- Entity / entity set
- Relationship / relationship set
- Attribute
  - Simple
  - Composite
  - Multi-valued
- Weak entity
- one-to-one, one-to-many, many-to-one relationships
- total or partial participation
- binary or n-ary relationship sets
- recursive relationship sets
- Primary keys, foreign keys
- Converting E-R diagrams to table schemas and SQL statements

DB schema refinement/normalization: things to remember

- Database anomalies
  - Insertion anomaly
  - Deletion anomaly
  - Update anomaly
- First normal form (1NF)
- Functional dependencies
- Third normal form (3NF)
- Boyce-Codd normal form (BCNF)
- Decomposition
  - Lossless-join
  - Dependency preserving
- Multi valued dependencies
- Fourth normal form (4NF)

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**

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.

Indexes

- Indexes are used to speed up queries.
- 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.

Today's plan

- A short note on date/time processing in SQL
- Access control
- Stored procedures
- Triggers
- Transaction processing

SQL and date/time

date represents a calendar date

SQL and date/time

date represents a calendar date

time time of day (hours, minutes, seconds)

SQL and date/time

date	represents a calendar date
time	time of day (hours, minutes, seconds)
timestamp	date and time together.

SQL and date/time

date	represents a calendar date
time	time of day (hours, minutes, seconds)
timestamp	date and time together.

- you can add **with** timezone for storing timezone information (e.g., order_date **timestamp with** timezone).

SQL and date/time

date	represents a calendar date
time	time of day (hours, minutes, seconds)
timestamp	date and time together.

- you can add **with** timezone for storing timezone information (e.g., order_date **timestamp with** timezone).
- functions such as **year**(), **hour**() pick out a particular value.

SQL and date/time

date	represents a calendar date
time	time of day (hours, minutes, seconds)
timestamp	date and time together.

- you can add **with** timezone for storing timezone information (e.g., order_date **timestamp with** timezone).
- functions such as **year**(), **hour**() pick out a particular value.
- functions **current_date**(), **current_time**(), **current_timestamp**() can be used to retrieve the current date and time.

SQL and date/time

date	represents a calendar date
time	time of day (hours, minutes, seconds)
timestamp	date and time together.

- you can add **with** timezone for storing timezone information (e.g., order_date **timestamp with** timezone).
- functions such as **year**(), **hour**() pick out a particular value.
- functions **current_date**(), **current_time**(), **current_timestamp**() can be used to retrieve the current date and time.
- there are quite a few vendor extensions, or vendor specific behavior. But the basics should work for all.

Access control in relational databases

Database management systems allow a set of restrictions to be applied to control who can access the data. We can typically control who can

- read data
- insert new data
- update data
- delete data

Depending on the DBMS in use, access rights may be controlled on database, table or view and columns.

Granting access in SQL

The command **grant** is used for allowing certain actions. It takes the form:

grant privilege_list
on table_name
to user;

- A list of commonly used privileges are, **select**, **update**, **insert**, **delete** and **all** (for all privileges).
- A certain privilege can follow column names in parentheses to indicate which columns the privilege will affect.
- **on** follows a table or view name.
- **to** follows a database user name, or a role.

The privileges and the levels they can be applied are DBMS dependent.

SQL grant examples

SQL **grant** examples

- To allow user 'user1' to select from and update the table 'student':
  grant select, **update on** student **to** user1;

SQL grant examples

SQL **grant** examples

- To allow user 'user1' to select from and update the table 'student':
  grant select, **update on** student **to** user1;
- To allow user 'user2' to update and insert only to columns 'ID' and 'address' on 'student' table:
  grant update(ID,address), **insert**(ID,address)
  on student **to** user2;

SQL grant examples

SQL **grant** examples

- To allow user 'user1' to select from and update the table 'student':
  grant select, **update on** student **to** user1;
- To allow user 'user2' to update and insert only to columns 'ID' and 'address' on 'student' table:
  grant update(ID,address), **insert**(ID,address)
  on student **to** user2;
- Grant all privileges to 'user4' on student table:
  grant all on student **to** user3;

SQL grant examples

SQL **grant** examples

- To allow user 'user1' to select from and update the table 'student':
  grant select, **update on** student **to** user1;
- To allow user 'user2' to update and insert only to columns 'ID' and 'address' on 'student' table:
  grant update(ID,address), **insert**(ID,address)
  on student **to** user2;
- Grant all privileges to 'user4' on student table:
  grant all on student **to** user3;
- Grant all privileges to 'user4' on student table, including transfer of the rights:
  grant all on student **to** user4 **with grant option**;

Restricting access to certain rows

We can use views to grant to rights for only some rows of a table.

- To allow user 'user1' to only see data about students in department 'IK';

```
create view ik_students as
select * from student where dept = 'IK';
grant select on ik_students to user1;
```

Restricting access to certain rows

We can use views to grant to rights for only some rows of a table.

- To allow user 'user1' to only see data about students in department 'IK';

  ```
  create view ik_students as
  select * from student where dept = 'IK';
  grant select on ik_students to user1;
  ```

- We can still limit access to certain columns either by

  ```
  grant select (ID, address) on ik_students to user1;
  ```

  or restricting view to only include these columns.

Revoking access in SQL

The command **grant** is used for allowing certain actions. It takes the form:

revoke privilege_list
on table_name
from user;

Examples:

- **revoke select on** student **from** user1;
- **revoke update**(ID) **on** student **from** user2;

Stored procedures

Stored procedures are general purpose programming procedures on a DBMS.

- Stored procedures support all typical general purpose programming constructs (variables, conditional execution, loops, . . .)
- They are database objects, and stored in the database.

```
create procedure get_books()
begin
    select * from book;
end
```

```
call get_books;
```

Why (not) use stored procedures?

+ You put all your 'business logic' into one place.
+ They are (typically) faster than individual SQL queries.
+ They may reduce network traffic.
+ They may provide convenient ways of control access, and may be useful to prevent some security problems.
– Syntax is incompatible between different DBMSes.
– Typically SPs are more difficult to debug.
– Puts a bigger burden on DBMS.

Why (not) use stored procedures?

+ You put all your 'business logic' into one place.
+ They are (typically) faster than individual SQL queries.
+ They may reduce network traffic.
+ They may provide convenient ways of control access, and may be useful to prevent some security problems.
– Syntax is incompatible between different DBMSes.
– Typically SPs are more difficult to debug.
– Puts a bigger burden on DBMS.

Note: The issue of stored procedures vs. inline SQL code may easily get into a heated discussion. Use the one that makes sense for the particular case.

Stored procedure implementations

- ANSI standard for stored procedure language is called SQL/PSM.
- Many vendors implemented their own languages, e.g., Oracle PL/SQL.
- Even when a DBMS system implements the standard, the level standard compliance tends to be varied.
- Many DBMS systems support stored procedures written in more common programming languages as well: Java, C, perl, . . . , even PHP (PostgreSQL).
- We will go through basics of SQL/PSM as implemented by MySQL (version 5+).

Stored procedures in MySQL

```
delimiter //
create procedure get_books()
begin
    select * from book;
end //
delimiter ;
```

- call get_books(); calls the procedure.
- show procedure status; lists the stored procedures in the database.
- show **create** procedure get_books(); lists the procedure code.
- **drop** procedure get_books; drops it.
- Change of delimiter is a trick to be able to use multiple statements with the default statement delimiter ';'.

Variables in stored procedures

- You can use local variables in a stored procedure.
- You have to declare all local variables before the actual code starts. For example:

```
declare customer_id int;
```

- The keyword **set** is used for variable assignments.

```
set customer_id = 10;
```

- You can define or use so-called session variables which are accessible throughout the same database connection. Session variables start with a '@'.

```
set @update_status = 'success';
...
select @update_status;
```

Arguments of stored procedures

- Stored procedures can take arguments,

```
create procedure
    confirm_order(in cid int, out status varchar(10))
```

- The arguments are defined to be one of

  - **in** arguments are read-only.
  - **out** arguments are set inside the procedure, they do not have to be defined before.
  - **inout** arguments are read, and modified by the stored procedure.

```
call confirm_order(10, @status);
select @status;
```

Control structures in stored procedures

Stored procedures support basic control structures.

- if-then-else:

```
if x = 0 then
    set @status = 'x = 0';
elsif x < 10 then
    set @status = '0 < x < 10';
else
    set @status = 'x > 10';
end if;
```

- case

```
case x
    when 0 then set @status = 'x = 0';
    when 1 then set @status = 'x = 1';
    else set @status = 'not 0 or 1';
end case;
```

Loops

- while

```
while <condition> do
    ...
end while;
```

- repeat

```
repeat
...
until <condition>
end repeat;
```

- loop

```
<loop_label>: loop
    ...
    if <condition> then
        leave <loop_label>:
    end if;
end loop <loop_label>;
```

Cursors

- A cursor is a pointer to a row of a table, or a query result.
- Like local variables, you need to declare the cursor before using it:

```
declare cur cursor for select * from book;
```

- To start using it, you need to use the statement open.
- fetch reads the row, and moves the cursor to the next row,

```
fetch cur into isbn, author, title;
```

(assuming isbn, author and title are previously defined variables)

MySQL stored procedure an example

```
drop procedure if exists confirm_order;
delimiter //
create procedure confirm_order(in cust_id int, out nitems int)
begin
    declare isbn_tmp varchar(13) default null;
    declare customer, quantity int;
    declare more_rows bool default true;
    declare cur cursor for
        select cID, ISBN, qty from basket where cID = cust_id;
    declare continue handler for not found set more_rows = false;
    set nitems = 0;
    open cur;
    fetch cur into customer, isbn_tmp, quantity;
    while more_rows do
        set nitems = nitems + quantity;
        insert into orders (cID, ISBN, qty, order_date, status)
            values (customer, isbn_tmp, quantity, now(), 'N');
        fetch cur into customer, isbn_tmp, quantity;
    end while;
end //
delimiter ;
```

```
call confirm_order(10, @nbooks);
select @nbooks;
```

Stored procedures: access control

- Stored procedures can be used to restrict direct access to database tables.
- The stored procedures are run with the database user who created them.
- The other users can execute a stored procedure even if they have no rights to access the tables used by the stored procedures.
- The rights are granted (and taken away) as in any other database object, using **grant** and **revoke** SQL statements.

Stored procedures: closing notes

- Similar to stored procedures, users can also define stored functions which act like standard SQL functions, such as **upper**() or **year**().
- Stored procedures created, removed just like other database objects.
- SPs allow full procedural language constructs to be used with databases.
- SPs are just another tool available to software developer. Use them when it makes sense.

Triggers

- Triggers are pieces of conditional code that runs on DBMS when a certain event happens.
- Triggers are similar to stored procedures, except they are not called by user code, but executed automatically.
- Triggers can be used for doing arbitrary checks in case a certain event occurs. For example, they can be used to simulate check constraints.
- Triggers can be used to ensure data integrity, or automatically duplicate the data.
- Trigger syntax and support varies among different DBMSes.

Triggers in SQL

```
create trigger <trigger_name> <when> <action>
    on <table>
    for each row
begin
    /* trigger body */
end
```

- <**action**> is either **insert**, **update** or **delete**.
- <**when**> is either before or after
- If for each row is specified, the trigger is run for each row. Otherwise, it is run once.
- Trigger body is similar to stored procedures.
- Two special variables: new contains the new values (to be inserted), old contains the previous value (to be changed or discarded).
- The trigger can be removed using **drop trigger** <trigger_name>.

Trigger example (MySQL)

```
drop trigger if exists test_year;
delimiter //
create trigger test_year before insert on book
    for each row
begin
    declare msg varchar(255);
    if new.year > year(now()) then
        set msg = concat('Invalid year in book table: ',
                         cast(new.year as char));
        signal sqlstate '45000' set message_text = msg;
    end if;
end //
delimiter ;
```

Note: this works only in MySQL 5.5+

Transactions

Consider a customer ordering a book in an online bookshop.

1. Customer finds the book s/he is interested in stock.
2. Customer orders the book: we add it to orders and remove it from the stock.

Transactions

Consider a customer ordering a book in an online bookshop.

1. Customer finds the book s/he is interested in stock.
2. Customer orders the book: we add it to orders and remove it from the stock.

How about:

Customer 1	**Customer 2**
Finds the book in stock	
	Finds the book in stock
Orders the last book	
	Orders the book

Transactions

Consider a customer ordering a book in an online bookshop.

1. Customer finds the book s/he is interested in stock.
2. Customer orders the book: we add it to orders and remove it from the stock.

How about:

Customer 1	**Customer 2**
Finds the book in stock	
	Finds the book in stock
Orders the last book	
	Orders the book

We want the operations 'check availability' and 'update stock/order' to be atomic.

Database transactions

The solution to these problem by the databases are transactions.

A. Transactions are atomic: either all parts are executed or none.

C. Transactions are required to preserve consistency when run alone.

I. The DBMS executes transactions such that they appear to run in isolation.

D. The effects of the transactions are required to be durable: after transaction is finished, the effect persists even in case of system failures.

These properties are often called the ACID properties.

Transactions in SQL

- The statement start **transaction** starts a transaction.
- The transaction ends with either **commit** or **rollback**.
- If some hardware/network failure caused transaction to be interrupted, the system rolls back by default.
- Normally every SQL DDL or DML statements is committed automatically.

Transactions in SQL

- The statement start **transaction** starts a transaction.
- The transaction ends with either **commit** or **rollback**.
- If some hardware/network failure caused transaction to be interrupted, the system rolls back by default.
- Normally every SQL DDL or DML statements is committed automatically.

```
set autocommit = 0;
start transaction;
select qty from stock where isbn = @isbn;
-- rollback & exit here if qty < 1;
update stock set qty= qty - 1 where isbn = @isbn;
commit;
```

Transactions in SQL

- The statement start **transaction** starts a transaction.
- The transaction ends with either **commit** or **rollback**.
- If some hardware/network failure caused transaction to be interrupted, the system rolls back by default.
- Normally every SQL DDL or DML statements is committed automatically.

```
set autocommit = 0;
start transaction;
select qty from stock where isbn = @isbn;
-- rollback & exit here if qty < 1;
update stock set qty= qty - 1 where isbn = @isbn;
commit;
```

MySQL Note: transactions are available in some storage engines.

Transaction isolation levels

Transaction isolation levels can be set using **set transaction isolation level <level>** command.
Where **<level>** is one of:

SERIALIZABLE transactions execute in complete isolation. DBMS may run multiple transactions concurrently only if they do not interfere with the others.

REPEATABLE READ is similar to serializable, but inserts are allowed in the range the transaction may be reading. The same query should return the same values during the transaction (except so-called 'phantom reads').

READ COMMITTED The transaction does not read the uncommitted data, but two queries may return different results during the transaction.

READ UNCOMMITTED 'Dirty reads' are allowed. The data transaction reads may be rolled back.

Summary of today

- Access control: database management systems provide access control through users and roles.
- DB-side programming: stored procedures and stored functions are server-side programming constructs in database management systems.
- Triggers allow execution of a piece of code on the database side when certain conditions are met.
- Transaction processing: DBMSs provide mechanisms for preventing database inconsistency during concurrent access to the databases.

Summary of today

- Access control: database management systems provide access control through users and roles.
- DB-side programming: stored procedures and stored functions are server-side programming constructs in database management systems.
- Triggers allow execution of a piece of code on the database side when certain conditions are met.
- Transaction processing: DBMSs provide mechanisms for preventing database inconsistency during concurrent access to the databases.

You are not expected to be able to write stored procedures with this short introduction. You should know what they are for, and be able to learn the details when you need it

What is next?

- Next week: an overall summary. Bring your questions.
- Assignments: both assignment 5 and 6 will be posted today.
- Exam: scheduled Friday Nov 1, 10:00.
  - An example exam with solutions will be posted today.
  - You can bring a 'cheat sheet':
    - containing anything that you think may be useful during the exam.
    - no more than a single A4.
    - both sides can be used.
    - the text on the sheet should be legible with bare eye.