Database Technology
Lectures 2014/15

Per Holm

Per.Holm@cs.lth.se

What Is a Database?

A database is a collection of data, which is:

- persistent,
- structured, and the database contains a description of the structure (metadata).

A database management system (DBMS):

- manages the data (very large amounts of them, usually),
- allows users to specify the logical structure of the database (the database "schema") and to enter, alter, and delete data,
- allows users to query the data via a query language (SQL most common),
- supports concurrent access to the data,
- handles backup, crash recovery, ...

Why Not Simple Files Instead?

Consider:

- The file structure is determined by the application program:
  - Difficult to change the logical structure of the data.
  - A new kind of query ⇒ a new program must be written.
- Difficult to implement efficient queries and updates.
- Difficult to handle concurrent access, crash recovery, ...

Developing a Database Application

Several steps:

- *Model* the data. Entity-Relationship (E/R) modeling (also called conceptual modeling, semantic modeling, ...). Very similar to object-oriented modeling (static model).
- Translate the E/R model into a *relational* model.
- Implement the relational model in a DBMS.
- Write the necessary programs to implement the desired queries and updates.

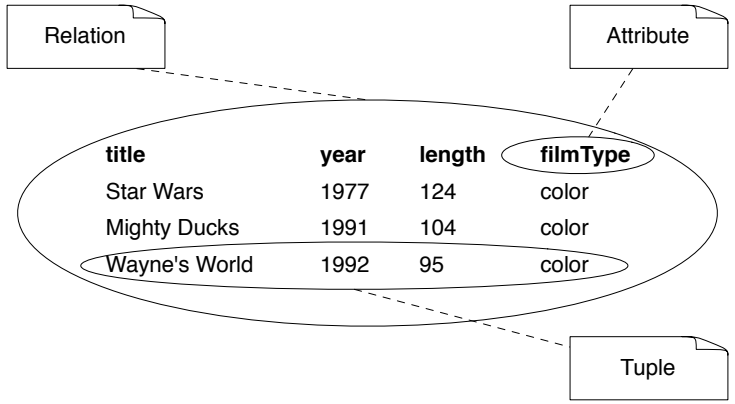
The Relational Data Model

Suggested in 1970 by E.F. Codd. Today used in almost all DBMS's.

- A relational database consists of *relations* (tables).
- A relation has *attributes* (columns in the table).
- A relation is a set of *tuples* (rows in the table).

Earlier, hierarchical or network databases were used.

The Relational Model Illustrated



Sample Relation

An instance of a relation `Accounts`, which records bank accounts with their balance, type, and owner:

```
+-----------+---------+----------+-------------+
| accountNo | balance | type     | ownerPNo    |
+-----------+---------+----------+-------------+
|     12345 |    1000 | savings  | 730401-2334 |
|     67890 | 2846.92 | checking | 790614-2921 |
+-----------+---------+----------+-------------+
```

Four attributes: `accountNo`, `balance`, `type`, and `ownerPNo`. Two tuples: the first says that account number 12345 has a balance of one thousand dollars, and is a savings account owned by a person with the person number 730401-2334.

Relation schemas are usually described in a more concise form. The relation name is given first, followed by the attribute names inside parentheses: `Accounts(accountNo, balance, type, ownerPNo)`.

SQL — Structured Query Language

- SQL History
- Simple Queries
- Joins
- Subqueries
- Aggregation Operators
- Grouping
- Database Modifications
- Defining Relations
- Primary Keys and Foreign Keys

SQL

- SQL stands for "Structured Query Language", pronounced "ess-queue-ell"
- Current standard SQL3, 1999
- Declarative — just tell SQL what should be done; it is up to the SQL compiler to figure out details about how to do it
- Two major parts: a DDL (data definition language), a DML (data manipulation language)
- Free format, reserved words, case-insensitive

SQL Usage

SQL can be used in different ways:

- Write SQL statements directly at a terminal or in a file:

```
select accountNo from Accounts where balance > 1000;
```

  This is for developers, not for end users.

- Write an application program (here in Java) and pass the SQL statements to the DBMS:

```
ResultSet rs = stmt.executeQuery("select accountNo from " +
        "Accounts where balance > 1000");
/* present the results to the user */
```

  This is the most common usage, and the way that most people come into contact with databases.

In either case, the application developer has to know SQL.

Different DBMS's

Huge, commercial:

- Oracle, MS SQL Server, DB2, Sybase, Informix, ...

Smaller, free:

- MySQL, PostgreSQL, mSQL, SQLite, ...

We use MySQL in the course.

- Advantages with MySQL: free, easy to install (you can do it on your own computer), fast, supports most of the SQL standard (from version 5), enormous user community, ...
- Disadvantages: we found none last year ...
- Differences between MySQL and standard SQL are noted on some slides.

Simple SQL Queries

Find the balance of account 67890:

```
select balance
from Accounts
where accountNo = 67890;
```

Find all savings accounts with negative balances:

```
select accountNo
from Accounts
where type = 'savings' and balance < 0;
```

Typical structure:

```
select SOMETHING
from SOMEWHERE
where CERTAIN CONDITIONS HOLD;
```

SQL From Book

The following examples are similar, but not always identical, to the examples in the book.

A movie database: a movie has a title, a production year, a length, is in color or black-and-white, is produced by a studio, and the producer has a certificate number. Schema:

```
Movies(title, year, length, filmType, studioName, prodNbr)
```

Example relation instance:

```
+---------------+------+--------+----------+------------+---------+
| title         | year | length | filmType | studioName | prodNbr |
+---------------+------+--------+----------+------------+---------+
| Pretty Woman  | 1990 |    119 | color    | Disney     |   99912 |
| Star Wars     | 1977 |    124 | color    | Fox        |   12345 |
| Wayne's World | 1992 |     95 | color    | Paramount  |   34129 |
+---------------+------+--------+----------+------------+---------+
```


SQL Example

Find all movies produced by Disney in 1990:

```
select *
from Movies
where studioName = 'Disney' and year = 1990;
```

Sample result:

```
+--------------+------+--------+----------+------------+---------+
| title        | year | length | filmType | studioName | prodNbr |
+--------------+------+--------+----------+------------+---------+
| Pretty Woman | 1990 |    119 | color    | Disney     |   99912 |
+--------------+------+--------+----------+------------+---------+
```

MySQL: string comparisons are not case sensitive.

Projection (`select` Clause)

Select only certain attributes:

```
select title, length
from Movies
where studioName = 'Disney' and year = 1990;
+--------------+--------+
| title        | length |
+--------------+--------+
| Pretty Woman |    119 |
+--------------+--------+
```

The same with new names for the attributes (as is optional):

```
select title as name, length as duration
from Movies
where studioName = 'Disney' and year = 1990;
+--------------+----------+
| name         | duration |
+--------------+----------+
| Pretty Woman |      119 |
+--------------+----------+
```


Selection (`where` Clause)

Expressions in the `where` clause (note single quotes around string literals):

```
select title
from Movies
where year > 1970 and filmType <> 'color';
```

Wildcards:

```
select title
from Movies
where title like 'Star%';
```

`%` matches 0–many characters, `_` matches exactly one character.

Ordering the Output

Sort the output, first by length, then by title:

```
select *
from Movies
order by length, title;
```

May (naturally) be combined with `where`:

```
select *
from Movies
where studioName = 'Disney' and year = 1990
order by length, title;
```


More Than One Relation

A relation describing movie studio executives who have a name, address, certificate number, and a net worth:

```
MovieExecs(name, address, certNbr, netWorth)
```

Find the producer of Star Wars:

```
select prodNbr
from Movies
where title = 'Star Wars';
```

(Output: 12345, remember this)

```
select name
from MovieExecs
where certNbr = 12345;
```

Can be done in a better way, see next slide.

Joining Several Relations in One Query

The better way to find the producer of Star Wars:

```
select name
from Movies, MovieExecs
where title = 'Star Wars' and
      prodNbr = certNbr;
```

The relations mentioned in the `from` clause are joined according to the condition in the `where` clause.

In the `select` and `where` clauses, all attributes in all the relations mentioned in the `from` clause are available.

If attribute names are not unique, they must be prefixed with the relation name and a dot.

Join Example II

The banking database (see slide 7, "Sample Relation") also keeps track of bank customers.

- A customer is described by his person number (Swedish civic registration number), his name, and his address.
- Each customer can own many accounts.
- Each account is owned by exactly one customer.

This is described by two relations:

```
Customers(persNo, name, address)
Accounts(accountNo, balance, type, ownerPNo)
```

The attribute `ownerPNo` in `Accounts` refers to the attribute `persNo` in one of the `Customers` tuples.

Join Example II, cont'd

Find the name of the customer who owns the account with number 12345:

```
select name
from Accounts, Customers
where accountNo = 12345 and
      ownerPNo = persNo;
```

accountNo	balance	type	ownerPNo	persNo	name	address
67890	2846.92	checking	790614-2921	730401-2334	Bo Ek	Malmö
12345	1000.00	savings	730401-2334	801206-4321	Eva Alm	Lund
...	...	...	...	...	...	...

Join Example III

Two relations (describe movies and that stars appear in movies):

```
Movies(title, year, length, filmType, studioName, prodNbr)
StarsIn(title, year, starName)
```

Find all movies with all stars who appear in them:

```
select Movies.title, Movies.year,
       length, filmType, studioName, starName
from Movies, StarsIn
where Movies.title = StarsIn.title and
      Movies.year = StarsIn.year;
```

The resulting table is shown on the next slide.

Join Example III, cont'd

title	year	length	filmType	studioName
Star Wars	1977	124	color	Fox
Mighty Ducks	1991	104	color	Disney
Wayne's World	1992	95	color	Paramount

title	year	starName
Star Wars	1977	Carrie Fisher
Star Wars	1977	Mark Hamill
Star Wars	1977	Harrison Ford
Mighty Ducks	1991	Emilio Estevez
Wayne's World	1992	Dana Carver
Wayne's World	1992	Mike Myers

title	year	length	filmType	studioName	starName
Star Wars	1977	124	color	Fox	Carrie Fisher
Star Wars	1977	124	color	Fox	Mark Hamill
Star Wars	1977	124	color	Fox	Harrison Ford
Mighty Ducks	1991	104	color	Disney	Emilio Estevez
Wayne's World	1992	95	color	Paramount	Dana Carvey
Wayne's World	1992	95	color	Paramount	Mike Myers

Set Operations, Union

The common set operations, union, intersection, and difference, are available in SQL.

The relation operands must be compatible in the sense that they have the same attributes (same data types) in the same order.

The relations `Movies` and `StarsIn` again:

```
Movies(title, year, length, filmType, studioName, prodNbr)
StarsIn(title, year, starName)
```

Find all movies mentioned in either relation:

```
select title, year from Movie
  union
select title, year from StarsIn;
```

Intersection, Difference

```
Stars(name, address, gender, birthDate)
MovieExecs(name, address, certNbr, netWorth)
```

Female movie stars who are also movie executives and who are rich:

```
select name, address
from Stars
where gender = 'F'
  intersect
select name, address
from MovieExecs
where netWorth > 10000000;
```

Movie stars who are not also movie executives:

```
select name, address from Stars
  except
select name, address from MovieExecs;
```

MySQL doesn't have `intersect` or `except`.

Subqueries

Find the producer of Star Wars (same as before):

```
select name
from Movies, MovieExecs
where title = 'Star Wars' and prodNbr = certNbr;
```

Another way to do this, using a *subquery:*

```
select name
from MovieExecs
where certNbr =
    (select prodNbr
     from Movies
     where title = 'Star Wars');
```

The subquery in parentheses produces a single tuple like `(12345)`. It would be an error if it produced more than one tuple (then you cannot test with =).

Conditions Involving Relations

Other tests on relations:

- `exists R`
- `s in R`
- `s > all R` (or <, <=, =, <>, >=)
- `s > any R` (or <, ...)

Prefix with `not` to get the negated condition.

Conditions Involving Tuples

```
Movies(title, year, length, filmType, studioName, prodNbr)
StarsIn(title, year, starName)
MovieExecs(name, address, certNbr, netWorth)
```

Find all producers of movies where Harrison Ford stars:

```
select name
from MovieExecs
where certNbr in
    (select prodNbr
    from Movies
    where (title, year) in
        (select title, year
         from StarsIn
         where starName = 'Harrison Ford')
    );
```

Note that you can explicitly construct tuples like `certNbr` or `(title, year)`.

Eliminating Subqueries

Many queries that use subqueries can be rewritten to use joins. A simpler way to find producers of Harrison Ford movies:

```
select name
from MovieExecs, Movies, StarsIn
where certNbr = prodNbr and
      Movies.title = StarsIn.title and
      Movies.year = Starsin.year and
      starName = 'Harrison Ford';
```

That is, join three relations.

Testing For Membership

Find the producers who haven't produced any movies. This *cannot* be written as a join:

```
select name
from MovieExecs, Movies
where certNbr <> prodNbr; -- WRONG
```

Must do it this way:

```
select name
from MovieExecs
where certNbr not in (select prodNbr from Movies);
```


Correlated Subqueries

Find titles that have been used for two or more movies (produced in different years). Using a correlated subquery:

```
select title, year
from Movies Old -- "rename" the relation
where year <> any
    (select year
    from Movies
    where title = Old.title);
```

With a join:

```
select Old.title, Old.year
from Movies Old, Movies New
where Old.title = New.title and
      Old.year <> New.year;
```

This will list the same title more than once. Use `select distinct` to avoid this.

Cross Join (Cartesian Product)

Two tables, `R` and `S`:

```
+------+------+    +------+------+------+
| a    | b    |    | b    | c    | d    |
+------+------+    +------+------+------+
|    1 |    2 |    |    2 |    5 |    6 |
|    3 |    4 |    |    4 |    7 |    8 |
+------+------+    |    9 |   10 |   11 |
                   +------+------+------+
```

`select * from R, S;` (or `select * from R cross join S;`)

```
+------+------+------+------+------+
| a    | b    | b    | c    | d    |
+------+------+------+------+------+
|    1 |    2 |    2 |    5 |    6 |
|    3 |    4 |    2 |    5 |    6 |
|    1 |    2 |    4 |    7 |    8 |
|    3 |    4 |    4 |    7 |    8 |
|    1 |    2 |    9 |   10 |   11 |
|    3 |    4 |    9 |   10 |   11 |
+------+------+------+------+------+
```

Natural ("normal") Join

```
select * from R, S where R.b = S.b;
+------+------+------+------+------+
| a    | b    | b    | c    | d    |
+------+------+------+------+------+
|    1 |    2 |    2 |    5 |    6 |
|    3 |    4 |    4 |    7 |    8 |
+------+------+------+------+------+
```

This is called "equijoin" (equality join). A *natural* join, where attributes with the same names are joined, is almost the same:

```
select * from R natural join S;
+------+------+------+------+
| b    | a    | c    | d    |
+------+------+------+------+
|    2 |    1 |    5 |    6 |
|    4 |    3 |    7 |    8 |
+------+------+------+------+
```


Variations on Join Syntax

Joining two tables with a condition in the `where` clause is common. The following syntaxes are equivalent:

```
select *
from R, S
where <some-condition>;

select *
from R inner join S
     on <some-condition>;
```

The two forms can be combined, often like this:

```
select *
from R inner join S
     on R.b = S.b
where a = 1;
```


Outer Joins

Sometimes, you wish to join but also include tuples from one table which do not match with any tuple in the other table. A left outer join includes tuples from the "left" table (mentioned first), a right outer join from the "right" table.

```
select *
from R right outer join S
     on R.b = S.b;
+------+------+------+------+------+
| a    | b    | b    | c    | d    |
+------+------+------+------+------+
|    1 |    2 |    2 |    5 |    6 |
|    3 |    4 |    4 |    7 |    8 |
| NULL | NULL |    9 |   10 |   11 |
+------+------+------+------+------+
```


Aggregation Operators

Five operators that can be applied to a column name:

```
SUM, AVG, MIN, MAX, COUNT
```

Examples:

```
select avg(netWorth)
from MovieExecs;

select count(starName) -- does not count nulls
from StarsIn;

select count(*) -- counts nulls
from StarsIn;
```

The aggregation operators may not be used in `where` clauses — they operate on a whole relation *after* tuples have been selected with `where`.

Grouping

It is possible to first group tuples from a query with `group by` and then apply an operator to the tuples of each group. Examples:

Find the sum of the lengths of all movies for each studio:

```
select studioName, sum(length)
from Movies
group by studioName;
```

Find the total length of all movies produced by each producer:

```
select name, sum(length)
from MovieExecs, Movies
where prodNbr = certNbr
group by name;
```


Grouping Illustrated

Schema: `Operas(composer, opera)`

```
select composer, count(*)
from Operas
group by composer;
```

composer	count(*)
Bellini	2
Puccini	3
Rossini	1
Verdi	4

composer	opera
Verdi	Nabucco
Verdi	La Traviata
Bellini	Norma
Puccini	La Bohème
Verdi	Rigoletto
Puccini	Madama Butterfly
Rossini	Guglielmo Tell
Puccini	Il Trittico
Verdi	Otello
Bellini	Il Pirata

composer	opera
Bellini	Norma
Bellini	Il Pirata
Puccini	La Bohème
Puccini	Madama Butterfly
Puccini	Il Trittico
Rossini	Guglielmo Tell
Verdi	Nabucco
Verdi	La Traviata
Verdi	Rigoletto
Verdi	Otello

`having` Clauses

You can control which groups that should be present in the output by introducing a condition about the group.

Find the composers, except Verdi, who have written more than two operas:

```
select composer, count(*)
from Operas
where composer <> 'Verdi'
group by composer
having count(*) > 2:
```

Note the order of selection:

1. first `where` (determine which tuples to include),
2. then `group by`,
3. last `having` (determine which groups to include).

Database Modifications — Insertion

Insert a new tuple into the relation `StarsIn(title, year, starName)`:

```
insert into StarsIn(title, year, starName)
values ('The Maltese Falcon', 1942, 'Sidney Greenstreet');
```

Since we here give new values for all the attributes we may instead write:

```
insert into StarsIn
values ('The Maltese Falcon', 1942, 'Sidney Greenstreet');
```

When this form is used, the attribute values must be in the same order as in the definition of the relation schema. This is sensitive to changes in the relation schema, so it is better to use the first form.

Deletion

Delete Sidney Greenstreet as a star in The Maltese Falcon:

```
delete from StarsIn
where title = 'The Maltese Falcon' and
      year = 1942 and
      starName = 'Sidney Greenstreet';
```

Note that the tuple must be described precisely. This would delete all occurrences of Sidney Greenstreet as a star:

```
delete from StarsIn
where starName = 'Sidney Greenstreet';
```

And this would delete all tuples from the relation:

```
delete from StarsIn;
```


Updates

Update the relation `MovieExecs(name, address, certNbr, netWorth)`. Double the net worth of all executives:

```
update MovieExecs
set netWorth = 2 * netWorth;
```

Attach the title `'Pres. '` in front of the name of every movie executive who is the president of a studio (described by the relation `Studios(name, address, presCNbr)`):

```
update MovieExecs
set name = 'Pres. ' || name
where certNbr in (select presCNbr from Studios);
```

`||` is string concatenation.

MySQL: use `concat(string1, string2)` to concatenate strings.

Defining a Relation Schema

A table is created in SQL with the `create table` statement. You list all attributes with their data types.

Example:

```
create table Stars (
    name      char(20),
    address   varchar(256),
    gender    char(1) default '?',
    birthdate date
);
```

See next slide for possible data types.

Data Types

`char(n)`	A character string of fixed length, `n` characters, padded with blanks.
`varchar(n)`	A character string of varying length, at most `n` characters.
`boolean`	`True` or `false` or `unknown` (!). In MySQL, `boolean` is a synonym for `tinyint` and `true` is `1`, `false` is `0`, `unknown` is `null`.
`int`, `integer`	Integers.
`float`, `real`	Floating-point numbers.
`decimal(n,d)`	Real numbers, `n` positions, `d` decimals.
`date`, `time`	Date and time.
`blob`	"Binary Large Objects"

Modifying Relation Schemas

Delete a table:

```
drop table Stars;
```

Add a column:

```
alter table Stars add phone char(16);
```

Delete a column:

```
alter table Stars drop birthdate;
```

Note: adding and deleting columns is common — databases are long-lived and changed many times during their lifetime.

Indexes

An *index* on an attribute of a relation is a data structure that makes it efficient to find those tuples that have a fixed value for the attribute.

It would speed up queries like the following if we could find the movies made in a specific year quickly:

```
select *
from Movies
where studioName = 'Disney' and year = 1990;
```

An index on the `year` attribute is created with:

```
create index YearIndex on Movies(year);
```

and deleted with:

```
drop index YearIndex on Movies;
```


Index Selection

Naturally:

- An index on an attribute speeds up queries where a value for the attribute is specified.
- An index makes insertions, deletions, and updates more time consuming.

Index selection is a difficult task!

Usually, the DBMS automatically creates an index on the *primary key* attributes of a relation. More about keys later.

View Definitions

Tables created with `create table` actually exist in the database. Another class of relations, *views*, do not exist physically. They are created with `create view` as the answer to a query and are *materialized* when they are accessed. Views can be queried as other relations, and sometimes also modified. You cannot have an index on a view.

Example:

```
create view ParamountMovies as
select title, year
from Movie
where studioName = 'Paramount';

select title
from ParamountMovies
where year = 1976;
```

There are rules for modification of relations via views.

View Advantages

Views are good for several things:

- to restrict what data a user sees. You can create a view and give a user access rights to this view only, not to the underlying tables,
- to simplify queries where another query is used more than once.

Materialized Views

Some DBMS's (not MySQL) support *materialized* views:

- like a normal ("virtual") view, but the values are stored in the database (like a table), not recomputed every time the view is accessed,
- efficient if the view is used frequently,
- but the view must be recomputed by the DBMS when one of the underlying base tables changes (like an index).

Constraints

The data in a database should be correct ...

Data can be checked by the application program when it is entered, but some of these checks can be automatically performed by the DBMS.
Integrity constraints in SQL:

- Key constraints
- Foreign-key constraints
- Assertions (not in MySQL)
- Triggers

Key constraints and foreign-key constraints are covered here, triggers later.

Primary Key Constraints

A *key* on a relation is an attribute, or a set of attributes, that uniquely identifies each tuple in a relation.

Example: `Persons(persNo, name, address)`. `persNo` is a key, `name` is not a key, `address` is not a key, `name+address` is not a key.

SQL (two equivalent variants):

```
create table Persons (
    persNo  char(11),
    name    varchar(40),
    address varchar(60),
    primary key (persNo)
);

create table Persons (
    persNo  char(11) primary key,
    name    varchar(40),
    address varchar(60)
);
```

Invented ("Synthesized") Keys

For many relations, the selection of attributes that are to be used as a primary key is straightforward. In the `Persons` example, the person number is an obvious key (actually, person numbers were invented for this purpose.)

But consider the `Movies` relation that we have used:

```
Movies(title, year, filmType, studioName, prodNbr)
```

We will show later that {`title, year`} is a key to the `Movies` relation.

But it may be costly to use a string and an integer as a key, and also impractical if you later find that there actually exist two movies with the same name that are made in the same year.

In practice, you would probably invent a "movie identification number" to use as a key for the relation. It would be advantageous if this number could be accepted as a standard by the whole movie industry.

Invented Keys in MySQL

In MySQL, you usually use an *auto increment* column when you need an invented key. Example:

```
create table T (
    x integer auto_increment,
    y varchar(10),
    primary key (x)
);

insert into T(y) values('first');
insert into T values(0, 'second');
insert into T values(null, 'third');
```

```
select * from T;
+---+--------+
| x | y      |
+---+--------+
| 1 | first  |
| 2 | second |
| 3 | third  |
+---+--------+

select last_insert_id();
+------------------+
| last_insert_id() |
+------------------+
|                3 |
+------------------+
```


Primary Key Consequences

If you declare an attribute (or a set of attributes) `S` as a primary key, you state that:

- two tuples cannot agree on all of the attributes in `S`,
- attributes in `S` cannot have `null` as a value.

Any insertion or update that violates one of these conditions will be rejected by the DBMS.

Declaring Keys with `unique`

Keys may be declared with `unique` instead of with `primary key`. This means almost the same, but:

- there may be any number of `unique` declarations, but only one `primary key`,
- `unique` permits nulls in the attribute values, and nulls are always considered unique.

Foreign Keys

A foreign key is an attribute in a relation that references a key (primary key or unique) in another relation. Example:

```
Studios(name, address, presCNbr)
MovieExecs(name, address, certNbr, netWorth)
```

We expect that each studio president is also a movie executive (or null, if the studio doesn't have a president at the present time). This is enforced by specifying presCNbr as a foreign key.

Declaring Foreign Keys

The Studios relation on the previous slide can be written:

```
create table Studios (
    name     char(30),
    address  varchar(255),
    presCNbr int,
    primary key (name),
    foreign key (presCNbr) references MovieExecs(certNbr)
);
```

or, alternatively, in standard SQL but NOT in MySQL:

```
create table Studios (
    name     char(30) primary key,
    address  varchar(255),
    presCNbr int references MovieExecs(certNbr)
);
```

certNbr must be a key (primary or unique) in MovieExecs. MySQL: only the InnoDB storage engine supports foreign keys.

Foreign Key Checks

We wish to state that Steven Spielberg, certNbr = 12345, is the president of Fox studios (he isn't). The following is ok:

```
insert into Studios values('Fox', 'Hollywood', 12345);
```

This will fail, since there is no movie executive with certNbr = 99999:

```
insert into Studios values('Fox', 'Hollywood', 99999);
```

The following updates fail — foreign keys are always checked.

```
update Studios set presCNbr = 99999 where name = 'Fox';
update MovieExecs set certNbr = 99999 where certNbr = 12345;
delete from MovieExecs where certNbr = 12345;
```

More on next slide.

More on Deletes and Updates

There are alternatives on what to do on deletes and updates. Suppose that Steven Spielberg retires — then we must set presCNbr = null in the Fox studio before we delete Steven's row in the MovieExecs table. This is simpler:

```
foreign key (presCNbr) references MovieExecs(certNbr)
on delete set null
```

Or we can delete the studio if the president retires (not realistic):

```
foreign key (presCNbr) references MovieExecs(certNbr)
on delete cascade
```

Similar with updates — what if Steven's certificate number is changed? Do it like this:

```
foreign key (presCNbr) references MovieExecs(certNbr)
on delete set null
on update cascade
```

Constraints on Attributes

You can disallow tuples in which an attribute is null by declaring the attribute as `not null`:

```
presCNbr int not null,
```

You can also make simple checks on attribute values:

```
presCNbr int check (presCNbr > 100000),
```

The following constraint is correct, but it cannot replace the foreign-key declaration:

```
presCNbr int check
(presCNbr in (select certNbr from MovieExecs)),
```

since the constraint isn't checked when a tuple from the `MovieExecs` relation is deleted or updated.

MySQL: `check` on attributes is not supported (`not null` is).

Database Design — E/R Modeling

- Database Design
- Entity Sets, Relationships
- UML Notation
- Finding Entity Sets and Relationships
- Constraints
- Weak Entity Sets

E/R Modeling, Introduction

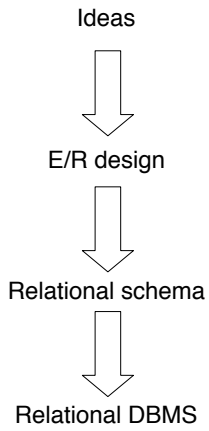
What information must the database hold? What are the relationships between the information components?

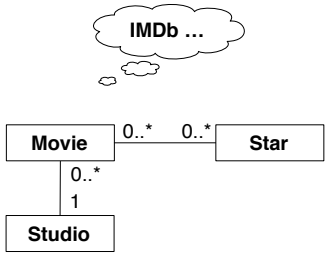
These questions are answered during the analysis phase of database development, when an analysis model is developed. This is called Entity-Relationship (E/R) modeling. Entities are "information pieces", "things" (compare with objects in object-oriented modeling).

After the analysis, the E/R model is translated into a relational model with tables, attributes, keys, etc. (compare with the design phase in object-oriented modeling).

Finally, the relational model is expressed in a data definition language and the queries in a data manipulation language (compare with the implementation phase in object-oriented modeling).

Database Design, Overview






```
Movies(title, year, ...)
Stars(name, ...)
...

 create table Movies (
   title varchar(40),
   year  int,
   ...
 );
```

Elements of the E/R Model

Element types in the E/R model:

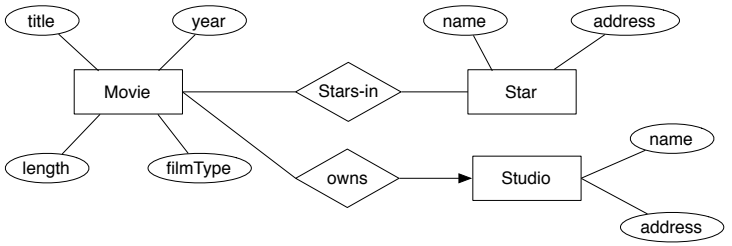
Entity sets Similar to (object-oriented) classes but contain only data, no operations. An entity set is a collection of entities (objects).
Note: I use singular names for entity sets (as for classes), corresponding plural names for relations.

Attributes The data in the entity sets. Attributes are atomic, i.e. "simple values" (ints, chars, strings, but not arrays, structs, ...).

Relationships Connections between entity sets (associations in object-oriented modeling).

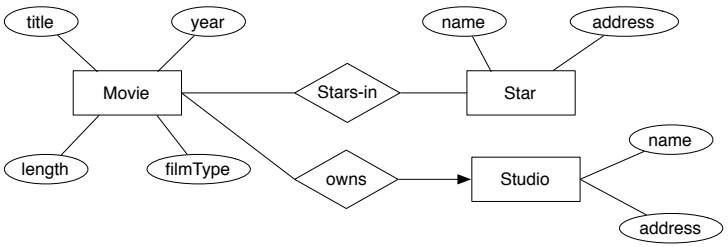
Entity-Relationship Diagrams

An E/R model is expressed in diagram form. There are several notations available, none of which is standardized. The book uses boxes for entity sets, ovals for attributes, and diamonds for relationships. Multiplicity of relationships is expressed with different kinds of arrow heads.



The First Diagram Explained

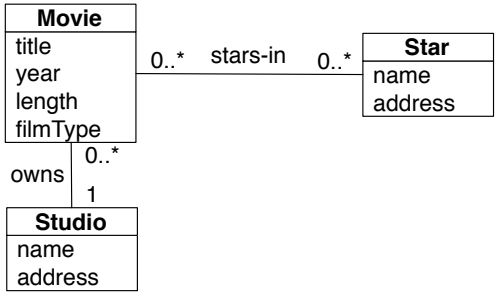
- Entity sets `Movie`, `Star`, and `Studio`.
- Movies have a title, a production year, a length, and a type, "color" or "B/W". Stars have a name and an address. Studios have a name and an address.
- A movie has several stars, a star can star in several movies. A movie is owned by one studio (the arrow head means "one"), a studio can own several movies.



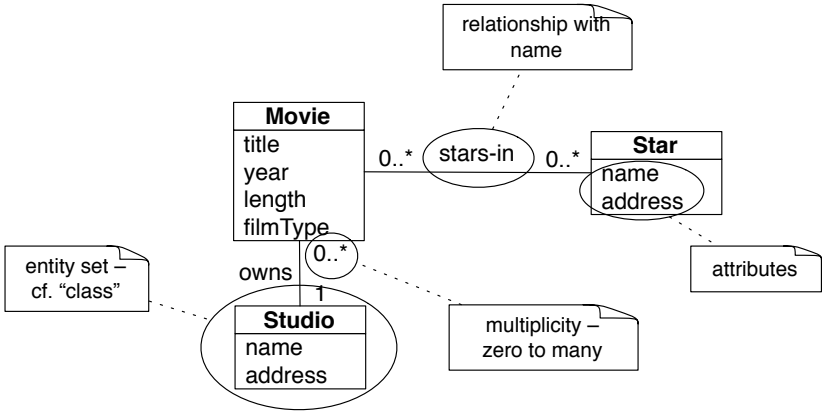
UML Notation

We will use UML (Unified Modeling Language) instead of the conventional E/R notation (this is becoming common in the database world).
Advantages with UML:

- standardized,
- compact,
- multiplicity is explicit.



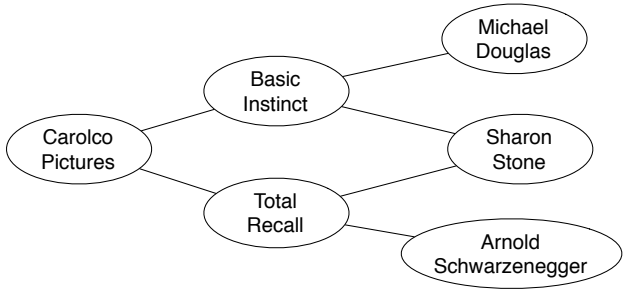
UML Notation Details



Instances of an E/R Diagram

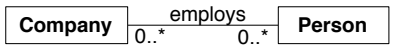
In object-oriented modeling, a class diagram can be instantiated to show objects instead of classes. Similarly, an E/R diagram can be instantiated to show entities instead of entity sets.

Example (not UML notation, and most attribute values have been omitted):

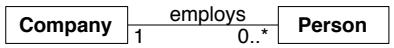


Multiplicity

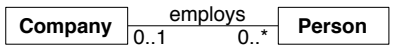
A company has many employees; each employee may work in several companies.



A company has many employees; each employee works in one company.

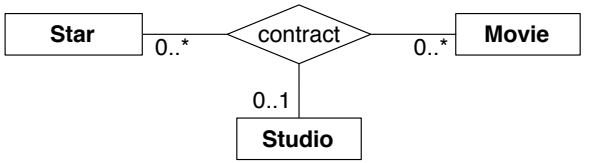


A company has many employees; each employee works in one company or is unemployed.

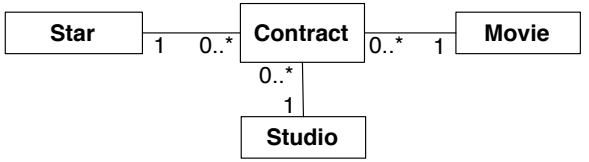


Multiway Relationships

A star signs a contract with a studio for a movie:

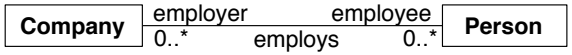


It's not easy to figure out the multiplicities in such relationships. Easier to promote the relationship to an entity set:

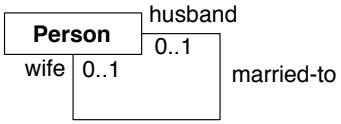


Roles

In addition to naming a relationship, you can specify the roles that the entities play in the relationship. A role name is written at the end of the relationship line.



Roles are especially useful when an entity set has a relationship to itself:

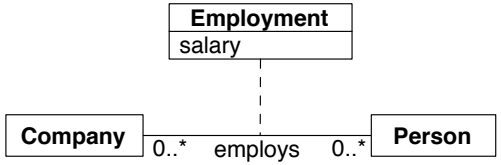


Attributes on Relationships

Reconsider the model with the company that has employees. Employees are paid salaries. Where does the salary attribute belong?

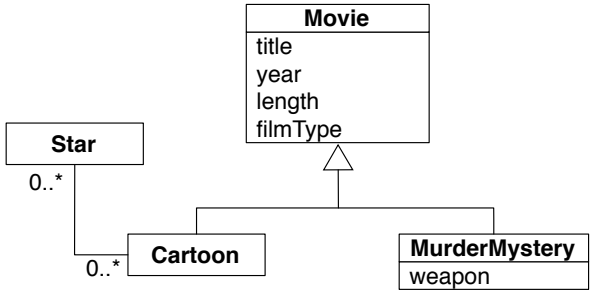
- Not in `Company`, since the company has many employees with different salaries.
- Not in `Person`, since a person may be employed by many companies and thus have many different salaries, or be unemployed and have no salary.

The attribute belongs to the *relationship*, employs. In UML:



Subclasses

There are special kinds of movies: cartoons and murder mysteries. Expressed with the "is-a" relationship (generalization/specialization):



This is similar to inheritance in object-oriented modeling.

Finding Entity Sets and Relationships

We have not considered how we go about finding the entity sets and relationships in a system, just said that they should "reflect reality".

In object-oriented modeling one common technique to find classes is to pick out the nouns from the requirements specification. This will result in a long list of names, and then there are rules for determining which of the names that are good classes. Many associations can also be found in the requirements specification.

This technique works also in database modeling. Some of the nouns in the list will become entities, some will become attributes, some are irrelevant, etc.

Design Principles

If you are used to object-oriented modeling, you have already met most of the principles of good design.

Faithfulness:

- The entity sets and their attributes should reflect reality.

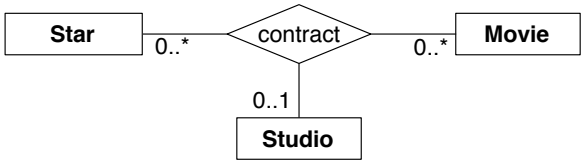
In order to meet this requirement you have to study the current system in detail.

Example, where we have entity sets `Course` and `Instructor`, with a relationship `teaches` between them. What is the multiplicity of this relationship?

- Do instructors teach more than one course? Are there instructors that do not teach?
- Can courses be taught by more than one instructor?

The answers to these and similar questions may vary depending on the policies and needs of the institution that is being modeled.

Choosing Relationships

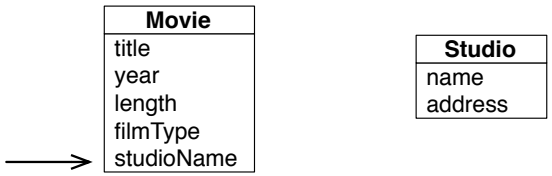


We have the relationship `contract` — do we still need the relationships `stars-in` (`Star` $\longleftrightarrow$ `Movie`) and `owns` (`Studio` $\longleftrightarrow$ `Movie`)?

- If a star can appear in a movie only if there's a contract we don't need `stars-in`.
- If all movies have at least one star under contract we don't need `owns`.

Expressing Relationships

This is *not* a correct model:



The attribute `studioName` expresses the same thing as the relationship `owns` that we had earlier, but in the wrong way:

- one of the main purposes of the E/R model is to make relationships clear and visible,
- so relationships must `not` be hidden in attributes.

Later, when the E/R model is translated into relations, the relation `Movies` will (maybe) contain an attribute `studioName` as a foreign key, but that is an implementation detail.

The Right Kind of Element

Why is `Studio` an entity set? Couldn't we put the name and address of the studio as attributes in `Movie` instead?

The answer is no, since

- studios are probably important real-world entities that deserve their own description, and
- to do so would lead to *redundancy*: we would have to repeat the studio address for each movie.

The case would be different if we did not record the address of studios, but even then it would probably be a good idea to keep the entity set `Studio` during analysis, and maybe remove it later, during design.

Modeling Constraints

Constraints give additional information on a model. Different kinds:

- Key constraints
- Single-value constraints
- Referential integrity constraints
- Domain constraints
- General constraints

Constraints are part of the database schema and may be enforced by a DBMS.

Key Constraints

A *key* is an attribute, or a set of attributes, that uniquely identifies an entity.

- A key can consist of more than one attribute. For instance, (title + year) uniquely identifies a movie. (Not just title by itself, since there may be several movies with the same title, but hopefully made in different years.)
- There can be more than one possible key. Pick one and use it as the *primary* key. In a generalization hierarchy, the key must be contained in the "root" entity set.

NOTE: in database modeling, keys are essential! Not so in object-oriented modeling, where each object has a unique identity.

Keys in the E/R Model

To show keys in a UML diagram, you underline the attributes belonging to a key for an entity set:

Referential Integrity

Referential integrity means that an entity surely exists at "the other end" of a relationship. Like this, where the multiplicity "1" tells us that every movie is owned by a studio:

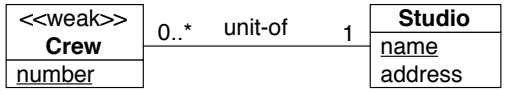
Note that if a studio is deleted, all movies owned by that studio must also be deleted.

The following is another case — there may be movies that are currently not owned by any studio.

Weak Entity Sets

An entity set which does not contain enough information to form a key is called a *weak* entity set. Some attributes from another (related) entity set must be used to form a key.

Example: a movie studio has several film crews. The crews are numbered 1, 2, ... Other studios may use the same numbering, so to identify a crew we must first identify the studio (using the name as a key), then we can use the number to identify the crew.



Weak Entity Sets, cont'd

We use the *stereotype* <<weak>> to designate a weak entity set. (This is not a standard stereotype, but UML allows you to invent your own stereotypes.)

The relationship to the entity set whose key is used to form the key for the weak entity set is called a *supporting* relationship. We don't have any notation to express this.

The example on the previous slide shows a common source of weak entity sets, where an entity is existent-dependent on another entity.

The Relational Data Model

The Relational Data Model

Suggested in 1970 by E.F. Codd. Today used in almost all DBMS's.

- A relational database consists of *relations* (tables).
- A relation has *attributes* (columns in the table).
- A row in a table is called a *tuple*.

The relational model is very well understood, and high level, very efficient, query languages (e.g., SQL) are supported.

During the analysis phase, however, it is better to use a model (E/R, for instance), that is richer and more expressive. After analysis, the E/R model is translated into relations.

Before, and After, Relations

Hierarchical databases (1965–1985)

- Data is structured in trees.
- Operations: tree traversal and node manipulation.

Network databases (1965–1985)

- Data is structured in graphs.
- Operations: graph traversal and node manipulation.

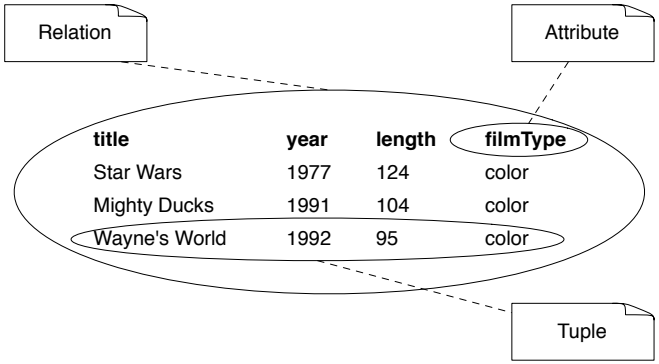
Relational databases (1975–?)

- Data is structured in tables.
- Operations: create new tables, modify tables, ...

Object-oriented databases (1990–?)

- Data is structured in objects, with references.
- Operations: methods in the objects.

Relational Model Basics



- Relation schema: `Movies(title, year, length, filmType)`.
- Relations are sets (order between tuples is immaterial).
- Order between attributes is also immaterial.

E/R Model → Relational Model

In short:

- Each entity set becomes a relation with the same attributes as the entity set.
- Each relationship becomes a relation whose attributes are the keys for the connected entity sets, plus the attributes of the relationship.

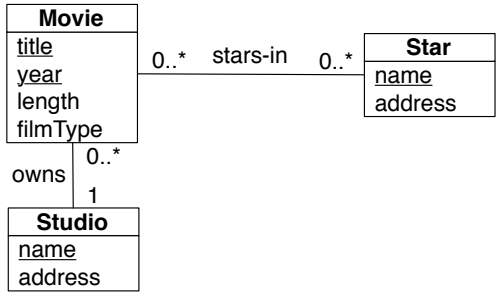
But:

- Special cases for relationships with multiplicity 1 (or 0..1).
- Special treatment of weak entity sets.
- Careful treatment of generalization hierarchies ("is-a").

And:

- Relations must be *normalized* to avoid redundancy.

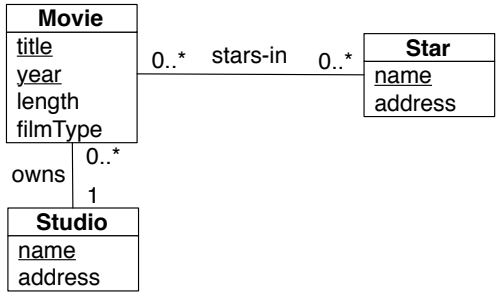
E/R Entity Sets → Relations



⇒

```
Movies(title, year, length, filmType)
Stars(name, address)
Studios(name, address)
```

E/R Relationships → Relations



$\Rightarrow$

```
Owns(title, year, studioName)
StarsIn(title, year, starName)
```

The relationship `owns` usually isn't implemented as a separate relation — see slide 96. Notice the keys in the relations — they are related to the multiplicities in the E/R model.

Foreign Keys

Foreign keys in relations that come from relationships:

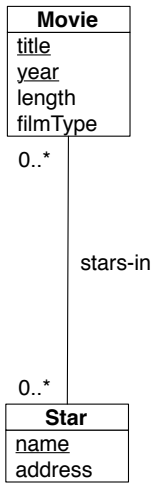
```
Owns(title, year, studioName)
```

In SQL, the corresponding table will look like this:

```
create table Owns (
    title      varchar(20),
    year       int,
    studioName varchar(20) not null,
    primary key (title, year),
    foreign key (title, year) references Movies(title, year),
    foreign key (studioName) references Studios(name)
);
```

Note `not null`: this is because there is a "1" on the studio side of the relationship. Had the multiplicity been "0..1" we wouldn't have specified `not null`.

Example Relations



title	year	length	filmType
Star Wars	1977	124	color
Mighty Ducks	1991	104	color
Wayne's World	1992	95	color

title	year	starName
Star Wars	1977	Carrie Fisher
Star Wars	1977	Mark Hamill
Star Wars	1977	Harrison Ford
Mighty Ducks	1991	Emilio Estevez
Wayne's World	1992	Dana Carver
Wayne's World	1992	Mike Myers

name	address
Carrie Fisher	123 Maple St., Hollywood
Mark Hamill	456 Oak Rd., Brentwood
Harrison Ford	789 Palm Dr., Beverly Hills
...	...

Combining Relations

Relations that arise from relationships with multiplicity 1 (or 0..1) can be deleted by modifying the relation on the "many" side.

Earlier, we used the schema:

```
Movies(title, year, length, filmType)
Owns(title, year, studioName)
Studios(name, address)
```

The studio is determined uniquely by the key for `Movies`, so we can use the following, simpler, schema instead:

```
Movies(title, year, length, filmType, studioName)
Studios(name, address)
```

Nulls

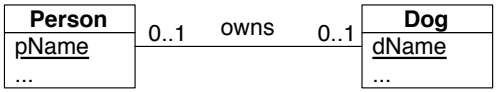
The combining of relations on the previous slides works also for relationships with multiplicity "0..1" instead of "1":

Results in:

```
Movies(title, year, length, filmType, studioName)
Studios(name, address)
```

In this case, `studioName` will be `null` if a movie doesn't have an owning studio.

Several Alternatives



Persons(pName, ...), Dogs(dName, ...), Owns(*pName*, *dName*) Good if there are lots of persons who don't own a dog, and lots of dogs without an owner, but introduces a new table.

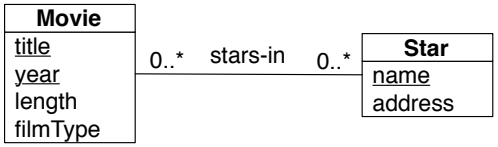
Persons(pName, ..., *dName*), Dogs(dName, ...) Good if all (most) persons own a dog.

Persons(pName, ...), Dogs(dName, ..., *pName*) Good if all (most) dogs have an owner.

PersonsDogs(pName, ..., dName, ...) Good if all (most) persons own a dog and all (most) dogs have an owner.

When Not to Combine Relations

Many-many (0..* – 0..*) relationships may *not* be combined. Example:



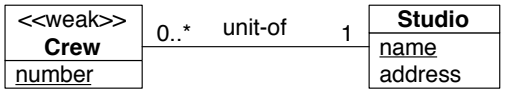
If we combined the relations we would have to repeat all the information about a movie (`length` and `filmType`) for each star!

title	year	length	filmType	starName
Star Wars	1977	124	color	Carrie Fisher
Star Wars	1977	124	color	Mark Hamill
Star Wars	1977	124	color	Harrison Ford
Mighty Ducks	1991	104	color	Emilio Estevez
Wayne's World	1992	95	color	Dana Carver
Wayne's World	1992	95	color	Mike Myers

Handling Weak Entity Sets

Recall: A weak entity set does not contain enough information to form a key.

- The relation for a weak entity set must include also the key attributes of the other (supporting) entity sets that help form the key.
- A supporting relationship need not be converted to a relation.

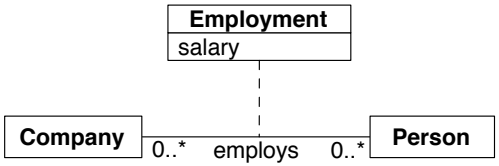


This becomes:

```
Studios(name, address)
Crews(number, studioName)
```

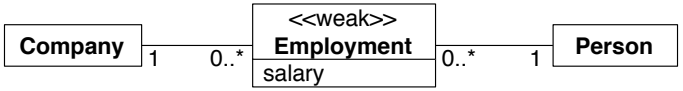
Relationships with Attributes

Create relations from the following model:

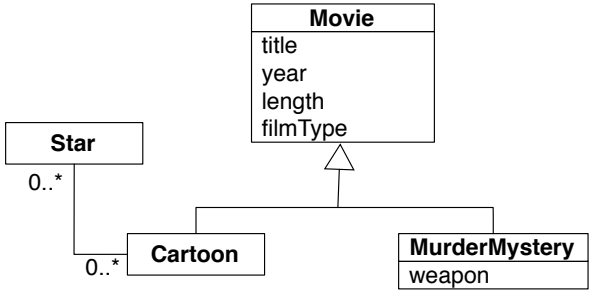



```
Company(cName, ...)
Person(pNbr, ...)
Employment(cName, pNbr, salary)
```

We get exactly the same result if we promote the `employs` relationship to a weak entity set:



Subclass Structures → Relations



Different options:

- **E/R** create a relation for each entity set.
- **Object-oriented** create a relation for each possible "concrete" entity set.
- **Null values** create one relation containing all attributes, use null where an attribute is not applicable.

Subclass Examples

None of the following alternatives is the best in all respects. The E/R method is the most commonly used.

E/R:

```
Movies(title, year, length, filmType)
MurderMysteries(title, year, weapon)
Cartoons(title, year)
```

Object-Oriented:

```
Movies(title, year, length, filmType)
MurderMysteries(title, year, length, filmType, weapon)
Cartoons(title, year, length, filmType)
```

Nulls:

```
Movies(title, year, length, filmType, weapon)
```

where `weapon` is `null` for regular movies and cartoons. Often necessary to introduce a type-attribute (like here, to differentiate between regular movies and cartoons).

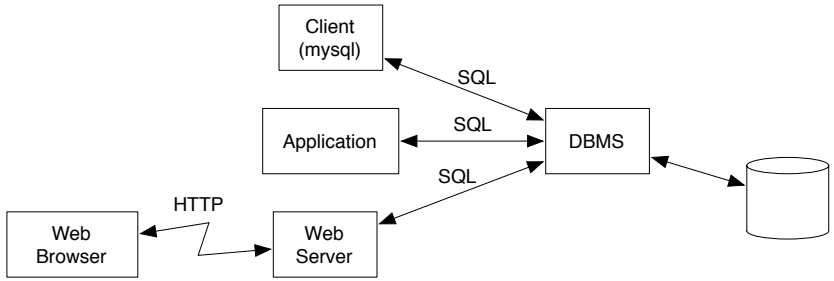
JDBC, Transactions

- SQL in Programs
- Embedded SQL and Dynamic SQL
- JDBC
- Drivers, Connections, Statements, Prepared Statements
- Updates, Queries, Result Sets
- Transactions

SQL in Programs

We have started by using SQL interactively, i.e., by writing SQL statements in a client and watching the results. All DBMS's contain a facility for doing this, but it is not intended for end users.

Instead, the SQL statements are specified within programs.



Advantages and Problems

The most important advantage with SQL in programs:

- You get access to a powerful programming language with advanced program and data structures, graphics, etc.

But there are also problems. SQL uses relations as the only data structure, "normal" programming languages use other structures: classes, arrays, lists, ... There must be a way to overcome this mismatch:

- How are values passed from the program into SQL commands?
- How are results of SQL commands returned into program variables?
- How do we deal with relation-valued data?

Embedded SQL

One way to interface to a DBMS is to embed the SQL statements in the program. Example (C language, Java's SQLJ is similar):

```
void createStudio() {
    EXEC SQL BEGIN DECLARE SECTION;
    char studioName[80], studioAddr[256];
    char SQLSTATE[6];
    EXEC SQL END DECLARE SECTION;

    /* read studioName and studioAddr from terminal */

    EXEC SQL INSERT INTO Studios(name, address)
            VALUES (:studioName, :studioAddr);
}
```

This code must be preprocessed to produce a normal C program with special DBMS function calls. The C program is then compiled and linked with a DBMS library.

Dynamic SQL

Another way to interface to a DBMS is to let the program assemble the SQL statements at runtime, as strings, and use the strings as parameters to the library function calls (CLI, Call Level Interface). Then, the preprocessing step can be skipped.

Example (Java, JDBC):

```
void createStudio() {
    String studioName, studioAddr;
    /* read studioName and studioAddr from terminal */
    String sql = "insert into Studios(name, address) "
            + "values (?, ?)";
    PreparedStatement ps = conn.prepareStatement(sql);
    ps.setString(1, studioName);
    ps.setString(2, studioAddr);
    ps.executeUpdate();
}
```

JDBC

JDBC (just a name, sometimes said to mean Java Database Connectivity) is a call level interface that allows Java programs to access SQL databases. JDBC is a part of the Java 2 Platform, Standard Edition (packages `java.sql`, basics, and `javax.sql`, advanced and new features).

In addition to the Java classes in J2SE you need a vendor-specific driver class for your DBMS.

ODBC is an earlier standard, not only for Java.

Loading the Database Driver

To load a database driver (in this example, a MySQL driver), you use the following statement:

```
try {
    Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException e) {
    // ... if the class cannot be found
}
```

You must load the driver only once in the program.

Establishing a Connection

When a driver has been loaded you connect to the database with a statement of the following form:

```
Connection conn = DriverManager.getConnection(
        url, username, password);
```

`Connection`	is a class in package `java.sql`,
`url`	is a vendor- and installation-specific string,
`username`	is your database login name,
`password`	is your database login password.

Example (for our local installation):

```
try {
    Connection conn = DriverManager.getConnection(
            "jdbc:mysql://puccini.cs.lth.se/" + "db01",
            "db01", "abc123de");
} catch (SQLException e) { ... }
```


A Note on `SQLExceptions`

All JDBC calls can throw `SQLExceptions` (package `java.sql`). In the following, we will not show the necessary `try catch` structure that is necessary to handle these exceptions:

```
try {
    // ... JDBC calls
} catch (SQLException e) {
    // ... do something to handle the error
} finally {
    // ... cleanup
}
```

Do *not* leave the `catch` block empty. As a minimum, write something like this:

```
} catch (SQLException e) {
    e.printStackTrace();
}
```

JDBC Statement Objects

To send an SQL statement to the DBMS, you use a JDBC `Statement` object or (better) a `PreparedStatement` object (package `java.sql`). An active connection is needed to create a statement object:

```
Statement stmt = conn.createStatement();
```

At this point `stmt` exists, but it does not have an SQL statement to pass on to the DBMS. You must supply that, as a string, to the method that is used to execute the statement. Example:

```
String sql = "update BankAccounts "
        + "set balance = balance + 1000.00 "
        + "where accntNbr = 'SA-223344'";
stmt.executeUpdate(sql);
```


SQL Injection

Be careful to check all input that you request from a user. Do not use unchecked input as part of an SQL string. Example:

```
String accntNbr = requestAccntNbrFromUser();
double amount = requestAmountFromUser();
String sql = "update BankAccounts"
        + "set balance = balance + " + amount + " "
        + "where accntNbr = '" + accntNbr + "'";
stmt.executeUpdate(sql);
```

A user who knows SQL could enter the following string when requested to enter the account number (this is called "SQL injection"):

```
' or 'x' = 'x
```

The resulting condition always evaluates to `true`! The problem doesn't occur if you use `PreparedStatements` (next slide).

Prepared Statements

In a `PreparedStatement`, you use JDBC calls to insert parameter values into an SQL statement. All strings are properly escaped, and the correct delimiters are supplied, so there is no danger of SQL injection.

```
String sql = "update BankAccounts "
        + "set balance = balance + ? "
        + "where accntNbr = ?";
PreparedStatement ps = conn.prepareStatement(sql);
```

The question marks are placeholders for the parameters. These are plugged in as follows:

```
ps.setDouble(1, amount);
ps.setString(2, accntNbr);
```

Now the statement is ready for execution:

```
ps.executeUpdate();
```


Exceptions and Closing of Statements

In the previous examples, we didn't close the statement object if an exception occurred. This we must do, and it is best done in the `finally` block:

```
try {
    // ... JDBC calls
} catch (SQLException e) {
    // ... do something to handle the error
} finally {
    try {
        ps.close();
    } catch (SQLException e2) {
        // ... can do nothing if things go wrong here
    }
}
```

The code in the `finally` block is *always* executed, regardless of what has happened before.

Creating Tables

Different kinds of SQL statements must be executed with different JDBC calls:

`executeQuery` `select` statements.

`executeUpdate` `insert`, `update`, `delete`, ..., (all modifying statements).

In the examples that follow, we will assume a database with the following schema (bars sell beers at different prices):

```
Bars(name, address)
Sells(barName, beerName, price)
```

It is possible to create these tables from JDBC, but database tables are usually not created or deleted in application programs (instead, in `mysql` or a similar database client).

Insert/Update Statements

Insert some data into the beer database, then raise the price of Urquell at Bishop's Arms:

```
String sql = "insert into Bars "
        + "values ('Bishop''s Arms', 'Lund')";
PreparedStatement ps = conn.prepareStatement(sql);
ps.executeUpdate();

sql = "update Sells "
        + "set price = price + 2.00 "
        + "where barName = 'Bishop''s Arms' "
        + "and beerName = 'Urquell')";
ps = conn.prepareStatement(sql);
int n = ps.executeUpdate();
```

Comments on next slide.

Comments, Insert/Update

Notice:

- No semicolon after the SQL statement string.
- Two consecutive '-s inside an SQL string become one ' (`'Bishop''s Arms'`).
- `executeUpdate` returns the number of affected tuples when an `update`, `insert` or `delete` statement is executed. It returns zero on `create` and `drop` statements.

Update Example

A more general method to raise the price of a beer at a specific bar.

```
boolean raiseBeerPrice(String bar, String beer, double amount) {
    try {
        String sql = "update Sells "
                + "set price = price + ? "
                + "where barName = ? and beerName = ?";
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setDouble(1, amount);
        ps.setString(2, bar);
        ps.setString(3, beer);
        int n = ps.executeUpdate();
        if (n != 1) {
            return false;
        }
    } catch (SQLException e) {
        return false;
    } finally { ... }
    return true;
}
```

`select` Statements

To issue a `select` query you execute the statement with `executeQuery`. This method returns an object of class `ResultSet` that functions as an iterator for the returned bag of tuples. A tuple is fetched with the `next()` method, and attributes of a tuple are fetched with `getXXX()` methods.

Example:

```
String sql = "select * from Sells";
PreparedStatement ps = conn.prepareStatement(sql);
ResultSet rs = ps.executeQuery();
while (rs.next()) {
    String bar = rs.getString("barName");
    String beer = rs.getString("beerName");
    double price = rs.getDouble("price");
    System.out.println(bar + " sells " + beer
            + " for " + price + " kr.");
}
```

`next()` returns `false` if there are no more tuples.

Result Sets

Notice: there is at most *one* `ResultSet` object associated with each statement object. So the following sequence of statements is wrong:

```
ResultSet rs = ps.executeQuery();
...
ps.close();
String beer = rs.getString("beerName");
// the statement is closed, the result set no longer exists
```

This is also wrong:

```
ps = conn.prepareStatement("select * from Sells where barName=?");
ps.setString(1, "Bishop's Arms'");
ResultSet rs1 = ps.executeQuery();
ps.setString(1, "John Bull");
ResultSet rs2 = ps.executeQuery();
...
String beer = rs1.getString("beerName");
// rs1 no longer exists, it has been replaced by rs2
```


Different get Methods

There are overloaded versions of the get methods that access a column not by name but by ordinal number. The following gives the same result as the get-s on slide 121:

```
String bar = rs.getString(1);
String beer = rs.getString(2);
double price = rs.getDouble(3);
```

This form is not recommended, since it presumes knowledge of the column order.

There are several different get methods: `getByte`, `getShort`, `getInt`, `getLong`, `getFloat`, `getDouble`, `getBoolean`, `getString`, ...

Let the DBMS Do the Work — 1

It may seem natural to write code like the following:

```
String sql = "select * from Sells";
PreparedStatement ps = conn.prepareStatement(sql);
ResultSet rs = ps.executeQuery();
while (rs.next()) {
    if (rs.getDouble("price") < 40.00 &&
            rs.getString("beerName").equals("Urquell")) {
        System.out.println("Cheap Urquell at: "
                + rs.getString("barName"));
    }
}
```

After all, this is "normal" programming.

Let the DBMS Do the Work — 2

You should let the DBMS do the work it is designed to do:

```
String sql = "select barName " +
        + "from Sells "
        + "where price < 40.00 and beerName = 'Urquell'";
PreparedStatement ps = conn.prepareStatement(sql);
ResultSet rs = ps.executeQuery();
while (rs.next()) {
    System.out.println("Cheap Urquell at: " +
            rs.getString("barName"));
}
```


Communication Java ↔ Database

Guidelines:

- The database communication should be in one class. Do not spread SQL statements all over your program!
- Write methods in the class that return results of SQL queries, or perform updates.
- Do not return `ResultSet` objects or anything JDBC-related.
- `select address from Bars where barName = ?;`
  ⇒ `public String getAddress(String barName) { ... }`
- `select barName from Bars;`
  ⇒ `public ArrayList<String> getBarNames() { ... }`
- `select * from Bars where barName = ?;`
  ⇒ `public Bar getBar(String barName) { ... }`
  and write a class `Bar` with the same attributes as the relation.

Transactions

Normally, databases are used by several clients simultaneously, and the DBMS executes the code for the clients in parallel (one thread for each client). The DBMS must ensure that actions performed by different clients do not interfere with each other.

The client code is grouped into *transactions*. A transaction is a sequence of actions that is performed as a "unit". The DBMS guarantees that a transaction is ACID:

- **Atomic** either performed in its entirety or not performed at all,
- **Consistent** transforms the database from one consistent state to another consistent state,
- **Isolated** executes independently of other transactions, so the partial effects of an incomplete transaction is invisible,
- **Durable** the effects of a successfully completed transaction are permanently recorded in the database.

Commit and Rollback

Normally, a client executes in "auto-commit" mode. This means that each SQL statement is its own transaction — the changes performed by the statement are immediately committed (written to the database).

A transaction is started with a command (START TRANSACTION) and ended with another command (COMMIT to save changes, or ROLLBACK to undo all changes).

```
delete from A;
start transaction;
insert into A values (1);
insert into A values (2);
commit;
-- A contains 1 and 2
start transaction;
insert into A values (3);
rollback;
-- A still contains 1 and 2
```

Problem: Lost Update

Two transactions T1 and T2 (x could be a bank account balance):

```
T1: read x; x = x + 100; write x; commit.
T2: read x; x = x - 100; write x; commit.
```

There will be problems (a "lost update") if T1's and T2's actions are interleaved:

```
      T1              T2          x
read x                            1000
                read x
x = 1000+100
                x = 1000-100
write x                           1100
commit
                write x           900
                commit
```

Here, T1 *must* execute before T2, or T2 before T1.

Problem: Dirty Read

Another problem is called "dirty read". It may occur if a transaction performs a rollback after it has written a data item, and that item has been read by another transaction.

```
      T1              T2          x
read x                            1000
x = 1000+100
write x                           1100
                read x            "dirty read"
                x = 1100-100
rollback                          1000
                write x           1000
                commit
```

Other problems: unrepeatable read, where a transaction reads the same data item twice and receives different answers because another transaction has changed the item; and the phantom problem, where a transaction receives different answers to a query because another transaction has modified a table.

Using Locks to Control Transactions, Deadlocks

One way of solving problems like the ones described is to use *locks*. A transaction may request a lock on a data item. If another transaction requests a lock for the same item it will have to wait until the first transaction has released the lock.

Locks may be of different granularity. No problems will surely occur if each transaction starts by locking the entire database, but since that would prevent all concurrency it is not acceptable. InnoDB and several other DBMS's lock rows in tables, others lock entire tables.

Two transactions can become stuck waiting for each other to release a lock. This is called *deadlock* and must be detected (or prevented) by the DBMS. Servers respond to deadlock by aborting at least one of the deadlocked transactions and releasing its locks.

Different Kinds of Locks

Most DBMS's support two kinds of locks:

Read (Shared) Lock A transaction which intends to read an object needs a *read lock* on the object. The lock is granted if there are no locks, or only other read locks, on the object (otherwise the transaction must wait for the other locks to be released).

Write (Exclusive) Lock A transaction which intends to write an object needs a *write lock* on the object. The lock is granted only if there are no read or write locks on the object (otherwise the transaction must wait for the other locks to be released)

Transactions that are read only (only hold read locks) can never block each other.

Locks may be explicit (explicitly requested by a transaction) or implicit (automatically requested by a transaction as a side effect of executing an SQL statement).

Serializable Transaction Schedules, Two-Phase Locking

The schedule of two (or more) transactions is an ordering of their actions. A schedule is *serial* if there is no interleaving of the actions of the different transactions (i.e., T1 executes in its entirety before T2, ...). To require serial schedules is not acceptable, since they forbid concurrency.

What we need is schedules that have concurrent execution but behave like serial schedules. Such schedules are called *serializable*. There is a surprisingly simple condition, called *two-phase locking* (or 2PL), under which we can guarantee that a schedule is serializable:

> *In every transaction, all lock actions precede all unlock actions.*

This condition can be enforced by the DBMS — normally by not providing any "unlock" action, but instead releasing all locks at commit (or rollback).

Transaction Isolation Levels

By setting the transaction *isolation level* clients can control what kind of changes the transaction is allowed to see:

READ UNCOMMITTED Can see modifications even before they are committed. Dirty, nonrepeatable, and phantom reads can occur.

READ COMMITTED Can only see committed modifications. Dirty reads are prevented, but nonrepeatable and phantom reads can occur.

REPEATABLE READ If a transaction issues the same query twice, the results are identical. Dirty and nonrepeatable reads are prevented, but phantom reads can occur. This is the default level in InnoDB.

SERIALIZABLE Rows examined by one transaction cannot be modified by other transactions. Dirty, nonrepeatable, and phantom reads are prevented.

Transactions in MySQL

MySQL supports several storage engines. The default engine, MyISAM, is not transaction safe. It only supports the LOCK/UNLOCK TABLES commands, which can be used in some cases to emulate transactions.

The InnoDB engine supports transactions. It performs row-level locking. If rows are accessed via an index, only the index records are locked. If there is no index and a full table scan is necessary, every row of the table becomes locked, which in turn blocks all modifications by other clients. So good indexes are important.

Consistent Non-Locking Read in InnoDB

A read (SQL `select`) in InnoDB sets no locks, not even a read lock. Instead, multi-version concurrency control (MVCC) is used to create a snapshot of the database when the transaction starts. The query sees the changes made by those transactions that committed before that point of time, and no changes made by later or uncommitted transactions (but it sees changes made by the same transaction). This is called "consistent read" in InnoDB.

Consistent reads has the advantage that read-only transactions never are blocked, not even by writers.

MVCC Implementation

Consistent read is implemented in the following way:

- Before an update, the affected rows are copied to a "rollback segment".
- There can be several rollback segments, identified by transaction number and timestamp.
- Consistent reads are made from the appropriate rollback segment.
- When the transaction commits, the rollback segment is discarded.
- If the transaction aborts, the information from the rollback segment is written to the database.

Locking in InnoDB

`select ...` sets no locks, consistent reads are used.

`update ...` (and `insert` and `delete`) sets write locks.

If a select is followed by a later update of the same item, it is necessary to explicitly set a lock on the item. This is done in the following way:

Read lock `select ... lock in share mode`

Write lock `select ... for update`

Transaction Example

A table `Flights` with flight number (`flight`) and number of available seats (`free`). Book a ticket on flight `A1`:

```
start transaction;
select free from Flights where flight = 'A1';
if (free == 0) rollback;
update Flights set free = free - 1 where flight = 'A1';
insert into Tickets ticket-information;
commit;
```

Here, two simultaneous transactions may find that one seat (the last) is available, and both may book that seat. This must naturally be prevented (one of the transactions must be rolled back); see next slide for alternatives.

Solutions

1. Write lock: `select free ... for update`. This will set a write lock on `A1` which will not be released until `commit`. Another transaction will block on the `select` statement and find that `free` has become 0.
2. Constraint check. In the table definition, specify `check (free >= 0)`. If this constraint is violated an exception will occur, which can be caught and the transaction aborted. (This does not work in MySQL.)
3. Explicit test. Before `commit`, select `free` again and rollback if it has become < 0.
4. Actually, `select free ... lock in share mode` also works. Both transactions will be granted a read lock. When the table is to be updated, both locks must be upgraded to write locks. This results in a deadlock, and one of the transactions will be aborted.

Transactions in JDBC

Transaction control in JDBC is performed by the `Connection` object. When a connection is created, by default it is in auto-commit mode.

Turn off auto-commit mode (start a transaction) with:

```
conn.setAutoCommit(false);
// ... a series of statements treated as a transaction
```

The series of statements can then be committed with:

```
conn.commit();
```

or rolled back with:

```
conn.rollback();
```

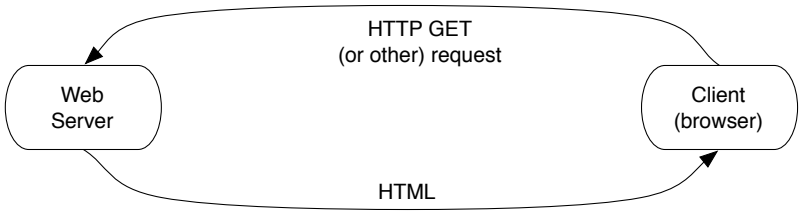
After a transaction, auto-commit mode must be turned on again.

PHP

- Web Servers
- Dynamic Web Pages
- HTML Forms
- The HTTP Protocol
- CGI
- PHP Overview
- PHP and HTML
- Sessions
- PHP and MySQL

Web Servers

A web server services requests from web clients ("browsers"). Servers must be able to handle several clients simultaneously.



Most HTML pages are static. Then, the server's task is easy: find the requested file, return it unchanged.

Dynamic Web Pages

There is a need for web pages with dynamic content:

- answers to database queries,
- animated web pages,
- user dialogs,
- checking user input,
- etc.

Dynamic content may be handled by the client (Java applets, JavaScript, VBScript, Flash, Shockwave, ...) or by the server (next slide).

Server-Side Technologies

Many systems of plug-in modules or scripting languages:

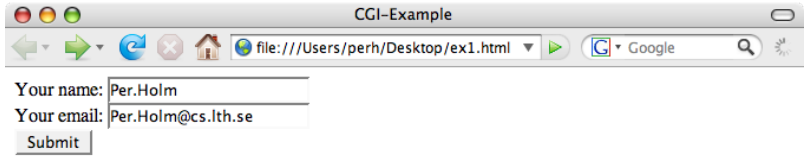
- CGI — Common Gateway Interface
- ASP — Active Server Pages
- PHP — PHP Hypertext Preprocessor (earlier Personal Home Page)
- Java Servlets
- Java Server Pages

When the server handles dynamic requests from a client, parameters are transmitted to the server as part of the URL or as part of the message.

Supplying Parameters, HTML Forms

This is a static HTML page containing a form:

```
<html><head><title>CGI-Example</title></head>
<body>
<form method = "get" action = "/cgi-bin/storeaddress.pl">
    Your name: <input name = "name" type = "text">
    <br>
    Your email: <input name = "email" type = "text">
    <br>
    <input type = "submit" value = "Submit">
</form>
</body></html>
```



Parameters in HTTP Requests

The form on the previous slide generates the following HTTP request:

```
GET /cgi-bin/storeaddress.pl?name=Per+Holm&email=
Per.Holm%40cs.lth.se&Submit=Submit HTTP 1.0
```

GET encodes the parameters in the URL. POST sends the parameters as part of the message. If POST had been used, the following request would have been generated:

```
POST /cgi-bin/storeaddress.pl HTTP 1.0
Content-type: application/x-www-form-urlencoded
Content-length: 54
name=Per+Holm&email=Per.Holm%40cs.lth.se&Submit=Submit
```

Usually, you use POST to transfer parameters from HTML forms. It is necessary if you have many long parameters.

CGI — Common Gateway Interface

1. The web server receives a request for a web page with a special URL (`/cgi-bin/program-name`).
2. The "CGI script" is started in a new process. The script may be written in any programming language.
3. The script reads the parameters from *stdin*.
4. The script writes output (HTML-code) to *stdout*.
5. The script terminates.

There are many problems with this. It is:

- expensive to start a new process for each request,
- difficult to save state between program executions,
- difficult to synchronize between processes.

Reply to a HTTP Request

A reply to a HTTP request contains:

1. the MIME type, typically `Content-type: text/html`,
2. a blank line,
3. HTML code.

Example:

```
Content-type: text/html

<html>
<head><title>Registration completed</title></head>
<body>
<h1>Registration completed</h1>
Per Holm (Per.Holm@cs.lth.se) has been added to
the user database.
</body>
</html>
```


PHP

PHP is an interpreted language. You can use PHP at the console, as a "normal" programming language, but it is more usual to embed PHP code in HTML pages. This code can be executed as a CGI program, but the interpreter can also live "inside" the web server.

When the PHP code is executed by the server, there is no overhead for process creation and destruction, and it is possible to save state between executions.

Java Server Pages and Active Server Pages have the same advantages.

The following slides contain an introduction to PHP – but only what's necessary to do lab 4.

Embedding PHP in HTML Files

In a HTML file, HTML code and PHP code may be freely mixed. The PHP code must be inside <?php and ?> tags. Example:

```
<html>
<title>Date</title>
<body>
Today is
<?php
    date_default_timezone_set('Europe/Copenhagen');
    $format = 'l, Y-m-d'; // Weekday, yyyy-mm-dd
    print date($format);  // Thursday, 2013-11-07
?>
</body>
</html>
```

The output from the PHP code is inserted into the dynamic HTML page.

Data Types and Variables

There are the usual data types: integers, reals, booleans, strings. You don't declare variables — PHP uses *dynamic typing:* an assignment to a variable determines its type. Variable names start with a $.

Examples:

```
$i = 0;
$x = 3.5;
$i_less = $i < $x;
$name1 = "Bob"; // "parsed" string
$name2 = 'Bob'; // "unparsed" string
```

Difference between parsed and unparsed strings:

```
$sum = 123;
print "The sum is $sum"; // prints The sum is 123
print 'The sum is $sum'; // prints The sum is $sum
```

Expressions

Nothing really strange, but two special operators:

- `.` String concatenation (like + in Java). Also `.=` (like += in Java).
- `===` Like == but both operands must have the same type.

```
$x = 123;
if ($x == "123") ...  // true!
if ($x === "123") ... // false
```

There is a set of "normal" standard functions like `sqrt`, `strcmp`, ...

Control Structures, Functions

`if`, `switch`, `while`, `for` as in Java. `elseif` instead of `else if`. `case` labels in a `switch` statement may have arbitrary type.

Functions:

```
function sqr($x) {
    return $x * $x;
}

print sqr(2);
```

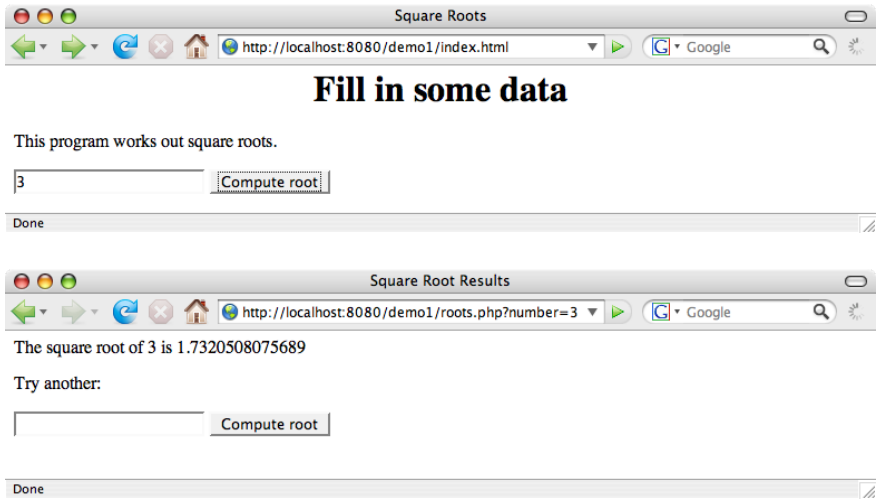
Functions can be written in separate *php*-files and `include`-d (or `require`-d).

Arrays

Arrays are *associative:* they consist of a number of key/value pairs. Keys and values may be of arbitrary types. Example:

```
$arr = array(); // not really necessary
$arr['a'] = 1;
$arr['xyz'] = "abc";
foreach ($arr as $key=>$value) {
    print "$key , $value<br>";
}
```


PHP and HTML, Example



HTML for First Page (*index.html*)

```
<html>
<head><title>Square Roots</title></head>
<body>
<h1 align = "center">Fill in some data</h1>

This program works out square roots.
<p>

<form method = "get" action = "roots.php">
    <input type = "text" name = "number">
    <input type = "submit" value = "Compute root">
</form>
</body>
</html>
```


PHP Code (*roots.php*), 1

```
<html>
<head><title>Square Root Results</title><head>
<body>

<?php
    $number = $_REQUEST['number'];
    if (is_numeric($number)) {
        if ($number >= 0) {
            print "The square root of $number is ";
            print sqrt($number);
        } else {
            print "The number must be >= 0.";
        }
    } else {
        print "$number isn't a number.";
    }
?>
```

Continued on next page.

PHP Code (*roots.php*), 2

The remainder of the file is pure HTML:

```
<p>
Try another:
<p>

<form method = "get" action = "roots.php">
    <input type = "text" name = "number">
    <input type = "submit" value = "Compute root">
</form>
</body>
</html>
```


Notes

- Note the URL in the call to *roots.php:* `.../roots.php?number=3`. GET is used, so the parameter is coded into the URL.
- The parameter is available in the array `$_REQUEST`. You could have used `$_GET` instead (or `$_POST` if the method had been POST).
- The standard function `is_numeric` checks both that the parameter is set and that it is numeric.

Sessions

Often, you need to save user data between calls to different PHP programs. For this the `$_SESSION` array is used. You may save only "serializable" data: numbers, strings, etc., but *not* "resource" data like database connections.

It must also be possible to determine which session data that belongs to which user. PHP uses a "session id" for this purpose. Usually, the session id is saved in a cookie in the client.

To start or restore a session the function `session_start()` is called, *before* anything else is sent to the client.

Sessions, Example

The following variant of the `roots` program remembers and prints the number of root computations:

```
<?php
    session_start();
    $_SESSION['computationNbr']++;
?>

<html>
<head><title>Square Root Results</title><head>
<body>
<?php
    $computationNbr = $_SESSION['computationNbr'];
    print "Root computation number $computationNbr<p>";
    $number = $_REQUEST['number'];
    ... same as before
```


Object-Oriented Programming in PHP

There are classes in PHP:

```
class BankAccount {
    private $balance;

    public function __construct() {
        $this->balance = 0;
    }

    public function getBalance() {
        return $this->balance;
    }

    public function deposit($amount) {
        $this->balance += $amount;
    }
}
```

`$this->` must be used to access attributes.

Creating and Accessing Objects

Suppose that the definition of the `BankAccount` class is in the file *bankaccount.inc.php*.

```
<?php
    require_once('bankaccount.inc.php');

    $myAccount = new BankAccount();
    $myAccount->deposit(150);
    print $myAccount->getBalance();
?>
```

In PHP 5, all objects are accessed via references, like in Java.

More Object-Oriented Programming

Other object-oriented constructs:

Destructors	`__destruct()`. PHP calls destructors during the "script shutdown phase," which is typically right before the execution of the PHP script finishes.
Inheritance	Like in Java, `class Subclass extends Superclass`.
Interfaces	Like in Java.
Type hints	The type of parameters can be specified, so the compiler can check method availability.

```
function clearAccount(BankAccount $account) {
    $account->deposit(- $account->getBalance());
}
```


PHP, MySQL and PDO

In PHP there are `mysqli` functions to access a MySQL database, `OCI` functions for an Oracle database, `sqlite` functions for an SQLite database, etc. These functions mostly do the same things but they have different names. PDO (PHP Data Objects) is an abstraction layer which provides a common API for many different DBMS's (like JDBC for Java). There are many similar packages in PHP (DB, DB2, MDB2, Zend, ...).

Opening and Closing a Connection

```
$host = "puccini.cs.lth.se";
$username = "db01";
$password = "abc123de";
$database = "db01";

$conn = new PDO("mysql:host=$host;dbname=$database",
        $username, $password);
$conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
...
$conn = null;
```

Errors can be caught with

```
try { ... } catch (PDOException $e) { ... }
```

The script is terminated with an error message if an exception is not caught.

Performing a Query

```
$sql = "select * from PersonPhones order by name";
$stmt = $conn->prepare($sql);
$stmt->execute();
$result = $stmt->fetchAll();
```

The result of `fetchAll` is an array of rows (like a JDBC `ResultSet`). A row is an array of attributes:

```
foreach ($result as $row) {
    foreach ($row as $attr) {
        ...
    }
}
```

The array of rows is both associative (with attribute names as keys) and indexed. Can be changed with `PDO::FETCH_ASSOC` or `PDO::FETCH_NUM` as parameter to `fetchAll`.

Counting Rows, `lastInsertId`

When an insert/update/delete statement is executed, `rowCount` returns the number of affected rows:

```
$sql = "insert ...";
$stmt = $conn->prepare($sql);
$stmt->execute($param);
$result = $stmt->fetchAll();
$count = $stmt->rowCount();
$id = $conn->lastInsertId();
```

For a select statement the rows must be counted:

```
$sql = "select ...";
$stmt = $conn->prepare($sql);
$stmt->execute($param);
$result = $stmt->fetchAll();
$count = count($result);
```


Parameters to Prepared Statements

The parameters to a prepared statement are given in an array:

```
$sql = "select * from Movies ".
       "where studioName = ? and year = ?";
$stmt = $conn->prepare($sql, array('Disney', 1900));
$stmt->execute();
```


Transactions

Transactions are handled the same way as in JDBC:

```
$conn->beginTransaction();
...
$conn->commit();
// ... or
$conn->rollback();
```


Things We Haven't Covered

The following are chapter titles from the book "PHP 5 Unleashed" by John Coggeshall. We haven't mentioned anything about these subjects:

- Regular Expressions
- PEAR
- Debugging and Optimization
- Data Encryption
- Writing Email in PHP
- SOAP and PHP
- Working with the File System
- Accessing the Underlying OS
- PHP's dba Functions
- Printable Document Generation
- Using Templates
- XSLT and Other XML Concerns
- User Authentication
- Working with HTML/XHTML Using Tidy
- Using PHP for Console Scripting
- Building WAP-enabled Websites
- Network I/O
- Using SQLite with PHP
- Working with Images

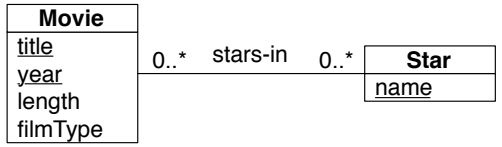
So there's more to learn if you want to be a professional PHP programmer ...

Normalization

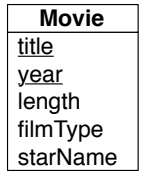
- Anomalies
- Functional Dependencies
- Closures
- Key Computation
- Projecting Relations
- BCNF
- Reconstructing Information
- Other Normal Forms

Normalization, Background

This "feels right":



This doesn't:



What happens if we convert the second model into a relation with the schema `Movies(title, year, length, filmType, starName)`? *Anomalies* occur — see next slide.

Anomalies

title	year	length	filmType	starName
Star Wars	1977	124	color	Carrie Fisher
Star Wars	1977	124	color	Mark Hamill
Star Wars	1977	124	color	Harrison Ford
Mighty Ducks	1991	104	color	Emilio Estevez
Wayne's World	1992	95	color	Dana Carver
Wayne's World	1992	95	color	Mike Myers

- Redundancy — information is repeated in several tuples (`length`, `filmType`).
- Update anomalies — you must be careful to change every occurrence of a value (the length of Star Wars must be changed in three places).
- Deletion anomalies — if a set of values becomes empty we may lose other information (delete Estevez ⇒ all information about Mighty Ducks is lost).

Normalization

If we have a relation, e.g., `Movies` on the previous slide, there exists a formal procedure to:

1. discover anomalies, and
2. decompose ("split") the relation into two (or more) relations without anomalies.

This procedure is called *normalization*. Normalization builds on the theory of *functional dependencies*.

(In the example on the previous slides, common sense would have led us right — it seems "unnatural" to put the `starName` attribute in the `Movies` entity set. However, there are other examples where common sense may not suffice.)

Functional Dependencies

A functional dependency, FD, on a relation R is a statement of the form:

Definition

If two tuples of R agree on the attributes $A_1, A_2, \ldots, A_n$, then they must also agree on another attribute, B. Notation:

$$A_1 A_2 \ldots A_n \rightarrow B$$

A functional dependency is a constraint on a schema and must hold for all instances of a schema. Example:

```
Persons(persNo, name, address)

persNo → name
persNo → address
```

or, shorter:

```
persNo → name address
```


FD Example

Relation `Movies(title, year, length, filmType, studioName, starName)`. The following FD's hold:

```
title year → length
title year → filmType
title year → studioName
```

or:

```
title year → length filmType studioName
```

but the following FD does *not* hold:

```
title year → starName // wrong!
```

This is because there may be several stars in a movie.

FD's Are About the Schema

Notice again that you cannot find FD's by looking at one specific instance of a relation. FD's are semantic properties that concern the *meaning* of the attributes.

Example: by looking at the instance of the `Movies` schema below, you might be led to believe that the FD `title` → `filmType` holds. This is not true (e.g., there are three versions of King Kong, two in color and one in black-and-white).

title	year	length	filmType	starName
Star Wars	1977	124	color	Carrie Fisher
Star Wars	1977	124	color	Mark Hamill
Star Wars	1977	124	color	Harrison Ford
Mighty Ducks	1991	104	color	Emilio Estevez
Wayne's World	1992	95	color	Dana Carver
Wayne's World	1992	95	color	Mike Myers

Keys of Relations

We have already defined the concept of a key for a relation informally. A formal definition:

Definition

A set of one or more attributes $A_1, A_2, \ldots, A_n$ is a key for a relation R if:

1. Those attributes functionally determine all other attributes of R.
2. No proper subset of $A_1, A_2, \ldots, A_n$ functionally determines all other attributes of R.

The last point means that a key must be minimal.

Example: {`title, year, starName`} is a key for the relation `Movies`.

A relation may have more than one key. In that case, one of the keys is chosen as the *primary key*. A set of attributes that contains a key is called a superkey ("superset of key").

Simple Rules About FD's

1. Transitivity (the most important rule):

$$A \to B \wedge B \to C \Rightarrow A \to C$$

2. Splitting/combining (just a convenient notation):

$$A \to B \wedge A \to C \Leftrightarrow A \to BC$$

3. Trivial dependencies that always hold, where the right side of an FD is a subset of the left side:

$$AB \to A$$

Closure of a Set of Attributes

Definition

The closure of a set of attributes, $\{A_1, A_2, \ldots, A_n\}$, under a set S of FD's is the set of attributes B such that $A_1A_2 \ldots A_n \to B$. That is, $A_1A_2 \ldots A_n \to B$ follows from the FD's in S.

The closure of $\{A_1, A_2, \ldots, A_n\}$ is denoted $\{A_1, A_2, \ldots, A_n\}^+$.

Closures are used for finding keys (slide 184). The algorithm for computing closures works by adding attributes on the right side of FD's to an initial set.

Example on next slide.

Closure Computation, Example

Relation $R(A, B, C, D, E, F)$.

FD's: $AB \to C$, $BC \to AD$, $D \to E$, $CF \to B$.

Compute $X = \{A, B\}^+$:

1. Start with $X = \{A, B\}$
2. $AB \to C$, so C can be added, $X = \{A, B, C\}$
3. $BC \to AD$, so AD can be added, $X = \{A, B, C, D\}$
4. $D \to E$, so E can be added, $X = \{A, B, C, D, E\}$
5. Nothing more can be added, so $\{A, B\}^+ = \{A, B, C, D, E\}$

From this we can infer that $AB \to D$ and $AB \to E$ follows from the initial set of FD's. (But not $AB \to F$, so $\{A, B\}$ is not a key for R.)

Key Computation I

The same relation and FD's as in the previous example:

$R(A, B, C, D, E, F)$

FD1. $AB \to C$
FD2. $BC \to AD$
FD3. $D \to E$
FD4. $CF \to B$

What are the keys in this relation? To answer this you have to compute the closures of all subsets of attributes. The keys are the subsets whose closure contains all five attributes.

Initial observation: F is not on the right-hand side of any FD. This means that all keys must contain F.

One-attribute subsets containing F:

$\{F\}^+ = \{F\}$

Key Computation II

Two-attribute subsets containing F:

$\{AF\}^+ \Rightarrow \{AF\}$
$\{BF\}^+ \Rightarrow \{BF\}$
$\{CF\}^+ \Rightarrow \{CF\} \overset{FD4}{\Rightarrow} \{CFB\} \overset{FD2}{\Rightarrow} \{CFBAD\} \overset{FD3}{\Rightarrow} \{CFBADE\}$
$\{DF\}^+ \Rightarrow \{DF\} \overset{FD3}{\Rightarrow} \{DFE\}$
$\{EF\}^+ \Rightarrow \{EF\}$

This shows that $\{CF\}$ is a key. So, when we examine three-attribute subsets we don't have to consider subsets that contain $\{CF\}$ (such subsets would be superkeys).

Key Computation III

Three-attribute subsets containing F but not C:

$\{ABF\}^+ \Rightarrow \{ABFCDE\}$
$\{ADF\}^+ \Rightarrow \{ADFE\}$
$\{AEF\}^+ \Rightarrow \{AEF\}$
$\{BDF\}^+ \Rightarrow \{BDFE\}$
$\{BEF\}^+ \Rightarrow \{BEF\}$
$\{DEF\}^+ \Rightarrow \{DEF\}$

$\{ABF\}$ is a key. The four-attribute subsets $\{ADEF\}$ and $\{BDEF\}$ are not keys, so the keys are $\{CF\}$ and $\{ABF\}$.

Projecting Relations

An unnormalized relation is normalized by splitting the relation in two (or more) relations. This is done by eliminating certain attributes from the relation schema. It is called *projection*.

Question: what FD's hold in the projected relation? Example:

$R(A, B, C, D)$ with FD's $A \to B$, $B \to C$, $C \to D$.

B is removed from R. What FD's hold in the new relation $S(A, C, D)$?

$\{A\}^+ = \{A, B, C, D\}$	gives $A \to C$, $A \to D$
$\{C\}^+ = \{C, D\}$	gives $C \to D$
$\{D\}^+ = \{D\}$	gives nothing
$\{C, D\}^+ = \{C, D\}$	gives nothing

Observe that $A \to C$ and $A \to D$ hold in the projected relation.

An Important Note on Projecting

Remember that FD's are semantic statements about the data in the schema. FD's hold regardless of the decomposition of data into relations and cannot disappear just because data items are split over several relations.

The previous example again:

$R(A, B, C, D)$ with FD's $A \to B$, $B \to C$, $C \to D$.

B is removed from R. What FD's hold in the new relation $S(A, C, D)$?

Wrong reasoning

"$A \to B$ cannot hold since B is not in S, $B \to C$ cannot hold since B is not in S, so $C \to D$ is the only FD that holds in S."

You must take dependencies that have been derived from the transitive rule into account!

Normalization again

From slide 176. There exists a formal procedure to:

1. discover anomalies, and
2. decompose ("split") the relation into two (or more) relations without anomalies.

This procedure is called *normalization*.

There are several "normal forms" (1st, 2nd, 3rd, Boyce-Codd, 4th, ...). The most important is Boyce-Codd normal form (BCNF).

Most books start by first defining the first normal form, then the second, ... Our book defines BCNF directly. Advantages with this:

- BCNF removes most anomalies, and
- it is easy to check if a relation is in BCNF.

Anomalies again

From slide 175:

title	year	length	filmType	starName
Star Wars	1977	124	color	Carrie Fisher
Star Wars	1977	124	color	Mark Hamill
Star Wars	1977	124	color	Harrison Ford
Mighty Ducks	1991	104	color	Emilio Estevez
Wayne's World	1992	95	color	Dana Carver
Wayne's World	1992	95	color	Mike Myers

- Redundancy — information is repeated in several tuples (`length`, `filmType`).
- Update anomalies — you must be careful to change every occurrence of a value (the length of Star Wars must be changed in three places).
- Deletion anomalies — if a set of values becomes empty we may lose other information (delete Estevez ⇒ all information about Mighty Ducks is lost).

BCNF, Boyce-Codd Normal Form

Definition

A relation R is in BCNF if and only if: whenever there is a nontrivial FD $A_1A_2 \ldots A_n \rightarrow B$ for R, it is the case that $A_1, A_2, \ldots, A_n$ is a superkey for R.

Example. The relation `Movies(title, year, length, filmType, starName)` has the FD's:

```
title year starName → length filmType
title year → length filmType
```

{`title, year, starName`} is the key.

{`title, year`} is not a superkey, i.e., it is not a superset of {`title, year, starName`}.

`Movies` is not in BCNF.

A Relation in BCNF

Consider the relation `Movies1(title, year, length, filmType)`, i.e., `Movies` without the `starName`. This relation has the only FD:

```
title year → length filmType
```

{`title, year`} is the key. There are no other non-trivial FD's, so `Movies1` is in BCNF.

Decomposition into BCNF

By repeatedly applying suitable decompositions, we can split any relation into smaller relations that are in BCNF.

Important: it must be possible to reconstruct the original relation instance exactly by joining the decomposed relation instances. The reconstruction will be shown later.

The following decomposition algorithm meets this goal:

BCNF decomposition

1. Start with a BCNF-violating FD, $A_1 A_2 \ldots A_n \rightarrow B_1 B_2 \ldots B_m$. Optionally expand the right-hand side as much as possible (closure).
2. Create a new relation with all the attributes of the FD, i.e., all the A's and all the B's.
3. Create a new relation with the left-hand side of the FD, i.e., all the A's, plus all the attributes *not* involved in the FD.
4. Repeat if necessary.

BCNF Decomposition, Example

```
Movies(title, year, length, filmType, starName).
```

BCNF-violating FD:	`title year` → `length filmType`
New relation:	`Movies1(title, year, length, filmType)`
New relation:	`Movies2(title, year, starName)`

We have already checked that `Movies1` is in BCNF. `Movies2` is also in BCNF — it contains no FD's at all, so the key is all three attributes.

Another BCNF Decomposition

Consider the relation

```
MovieStudios(title, year, length, filmType, studioName,
             studioAddr)
```

The only key is {`title, year`}.

An obvious FD is `studioName` → `studioAddr`, and `studioName` is not a superkey. Decomposition:

BCNF-violating FD:	`studioName` → `studioAddr`
New relation:	`MovieStudios1(studioName, studioAddr)`
New relation:	`MovieStudios2(title, year, length, filmType, studioName)`

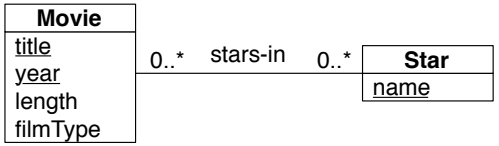
What Causes non-BCNF?

The causes of BCNF-violations:

- Putting a many-many relationship in one relation (the first example, with `Movies` containing star names).
- Transitive dependencies (the second example, where `studioName` → `studioAddr`).

Common Sense I

We started our discussion of normalization by saying that the following E/R diagram "feels right", but then we did not take it as a basis for the relations:



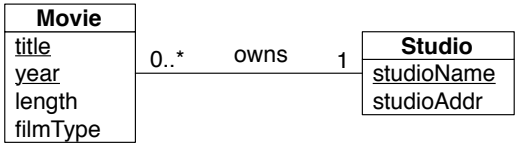
If we had followed the rules for translating an E/R model into relations we would have ended up with the following relations:

```
Movies(title, year, length, filmType)
Stars(starName) [superfluous if every star is in a movie]
StarsIn(title, year, starName)
```

i.e., exactly the same relations as `Movies1` and `Movies2` on slide 194.

Common Sense II

The example with movies and their owning studios should be modeled like this:



By the rules, this would have resulted in the following relations:

```
Movies(title, year, length, filmType, studioName)
Studios(studioName, studioAddr)
```

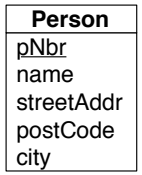
i.e., exactly the same relations as `MovieStudios2` and `MovieStudios1` on slide 195.

Common Sense III

Conclusions:

- Careful E/R modeling is essential for proper understanding of a problem.
- But careful E/R modeling also results in relations that are better normalized than if you start with relations directly.

This is not to say that careful E/R modeling always results in normalized relations. Consider the following model, which "feels right" but gives a relation that isn't in BCNF:



Reconstructing Information

We earlier said that "we must be able to reconstruct the original relation instance exactly from the decomposed relation instances". Such decompositions are called "lossless".

The reconstruction is performed by joining two relations. Two tuples can be joined if they agree on the values of an attribute (or values of sets of attributes).

The decomposition algorithm presented earlier, which is based on FD's, yields relations that may be reconstructed by joining on the attributes on the left-hand sides of the FD.

Other "ad hoc" algorithms may yield relations that, when joined, contain spurious (false) tuples ("lossy" decompositions).

Testing for Lossless Join

In the book the *chase algorithm*, which tests for lossless join, is described. Not part of the course.

A simpler test, which can be used only for testing decomposition into *two* relations, is the following:

Lossless Test

A decomposition of a relation in two relations $R1$ and $R2$ is lossless if:

- $R1 \cap R2$ is a superkey of $R1$, or
- $R1 \cap R2$ is a superkey of $R2$.

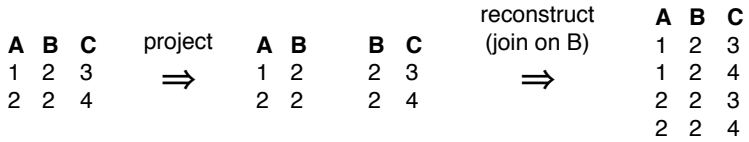
A Reconstruction Example

A relation $R(A, B, C)$ with only one FD:

FD1. $A \rightarrow B$

R is not in BCNF: $\{A, C\}$ is the only key, FD1 violates the BCNF condition. A decomposition of R that is not done according to the BCNF rules is $R1(A, B)$ and $R2(B, C)$. Both $R1$ and $R2$ are in BCNF.

Example projection and reconstruction:



The tuples $(1, 2, 4)$ and $(2, 2, 3)$ were not in the original relation and are spurious. Exercise: decompose according to the BCNF rules and then check the same example.

More Reconstruction

The same example as on the previous slide, with better names for the attributes. People have names and own cars. A person may own many cars, and a car may be owned by many persons. This is described by the relation `R(pNo, name, carNo)`, with the FD:

`FD1. pNo` $\rightarrow$ `name`

{`pNo, carNo`} is the only key, FD1 violates the BCNF condition. A decomposition of `R` that is not done according to the BCNF rules is `R1(pNo, name)` and `R2(name, carNo)`. Both `R1` and `R2` are in BCNF, since they have only two attributes each.

With these attributes, it is obvious that the decomposition in `R1` and `R2` is stupid. If we instead decompose according to the BCNF rules, we get the relations `S1(pNo, name)` and `S2(pNo, carNo)`, which can be joined on `pNo`.

Other Normal Forms, Motivation

Sometimes, BCNF is "too strong", in the sense that we may lose important FD's if we decompose into BCNF.

Example: the relation `Bookings(title, theater, city)` describes that a movie plays in a theater in a city. FD's:

`FD1. theater` $\rightarrow$ `city`
`FD2. title city` $\rightarrow$ `theater`

FD1 says (unrealistically) that all theaters have different names. FD2 says (unrealistically) that two theaters in the same city never show the same movie.

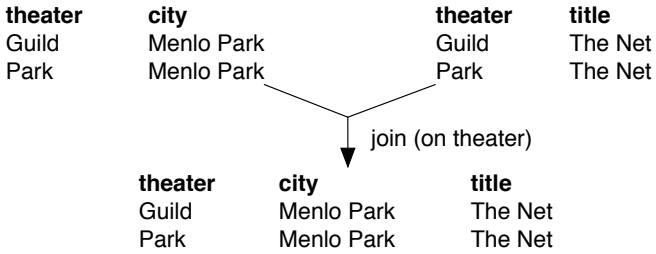
The keys are {`title, city`} and {`theater, title`}. So, FD1 is a BCNF violation. Attempt at decomposition on next slide.

Other Normal Forms, cont'd

If we decompose the relation `Bookings` into BCNF we get the relations:

```
R1(theater, city)
R2(theater, title)
```

These relations can be updated independently of each other. But when we do that, we cannot check that FD2, `title city` $\rightarrow$ `theater`, holds. I.e., when we join the relations (on `theater`) we may get tuples for which FD2 does not hold.



3NF, Third Normal Form

Third normal form is a relaxation of BCNF. Definition:

Definition

A relation R is in third normal form if and only if: whenever there is a nontrivial FD $A_1A_2 \ldots A_n \rightarrow B$ for R, it is the case that $A_1, A_2, \ldots, A_n$ is a superkey for R, or B is a member of some key.

The definition is the same as for BCNF with the addition "or B is a member of some key".

In the relation `Bookings(title, theater, city)` this allows the BCNF-violating FD:

```
theater → city
```

Since `city` is a member of the key {`title, city`}, `Bookings` is in 3NF.

Another Example

The following relation is not in BCNF:

```
Persons(persNo, name, street, postCode, city)
```

The problem is with the postal code. FD's:

```
FD1. persNo → name street postCode city
FD2. postCode → city
FD3. street city → postCode
```

FD2 and FD3 are BCNF violations. Notice that the relation is not even in 3NF, since neither `city` nor `postCode` is a member of a key.

Decomposition 1

We can decompose the `Persons` relation into BCNF relations. If we start with FD2, we get:

```
R1(postCode, city)
R2(persNo, name, street, postCode)
```

Both `R1` and `R2` are in BCNF.

This decomposition eliminates the redundancy that the city has to be mentioned several times for each postal code. And if we change the city for a postal code, we only have to perform the change in one place.

In the decomposition we have lost the possibility to check FD3, `street city` $\rightarrow$ `postCode`.

Decomposition 1 Comments

In practice, you probably wouldn't bother to decompose the `Persons` relation:

- How often are postal codes changed? Probably never or very seldom.
- How costly is the extra storage requirement? Probably not very costly.
- How often do you wish to check FD3? Probably never.

There is also an advantage to keeping the original relation. If you decompose, you have to perform a join on two tables every time you wish to access a person's full address.

Decomposition 2

If we instead start decomposing with FD3 we get:

```
R1(street, city, postCode)
R2(persNo, name, street, city)
```

`R1` is in 3NF, `R2` is in BCNF. We do not wish to decompose `R1` further — we would need to join *three* tables in order to access a person's full address.

Conclusions:

- Sometimes, unnormalized relations and the resulting redundancy are acceptable.
- But you must be able to motivate why you choose to use unnormalized relations (and to be able to realize that you are using them . . .).

More Redundancy

Some redundancy is not detected by the BCNF condition.

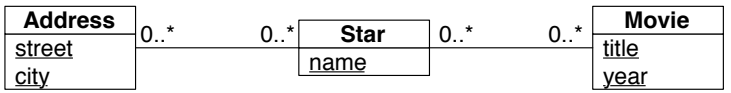
Consider the relation `StarMovie(name, street, city, title, year)`: a star has a name, may have several addresses, and may star in several movies. Example instance:

name	street	city	title	year
C. Fisher	123 Maple St.	Hollywood	Star Wars	1977
C. Fisher	5 Locust Ln.	Malibu	Star Wars	1977
C. Fisher	123 Maple St.	Hollywood	Empire Strikes Back	1980
C. Fisher	5 Locust Ln.	Malibu	Empire Strikes Back	1980
C. Fisher	123 Maple St.	Hollywood	Return of the Jedi	1983
C. Fisher	5 Locust Ln.	Malibu	Return of the Jedi	1983

There is obvious redundancy in this relation, but the relation is still in BCNF (it has no FD's at all).

Common Sense Again

In the relation `StarMovie` we have put *two* many–many relationships in one relation:



Naturally, this is not a reasonable thing to do, but it illustrates the need for yet another normal form.

Multivalued Dependencies

In the relation `StarMovie` there are no FD's at all. For example, `name` $\rightarrow$ `street city` does not hold, since a star may have several addresses. However, for each star there is a well-defined *set* of addresses. Also, for each star there is a well-defined *set* of movies. Furthermore, these sets are independent of each other.

This is called a *multivalued dependency* (MVD) and is denoted (note the two-headed arrows):

```
name ↠ street city
name ↠ title year
```

MVD's always come in pairs.

A Formal Definition of MVD

Definition

In a relation R, the MVD

$$A_1 A_2 \ldots A_n \twoheadrightarrow B_1 B_2 \ldots B_m$$

holds if: for each pair of tuples t and u of relation R that agree on all the A's, we can find in R some tuple v that agrees:

1. With both t and u on the A's,
2. With t on the B's,
3. With u on all attributes of R that are not among the A's or the B's.

Note that v may be $= t$ or $= u$.

t	a1	b1	c1
v	a1	b1	c2
u	a1	b2	c2

Reasoning About MVD's

Some rules about MVD's are similar to the FD rules.

- Trivial dependencies have the same definition.
- The transitive rule is the same.
- The splitting rule for FD's does not hold for MVD's.
- Every FD is a MVD.

4NF, Fourth Normal Form

Definition

A relation R is in fourth normal form if and only if: whenever there is a nontrivial MVD $A_1 A_2 \ldots A_n \twoheadrightarrow B$ for R, it is the case that $A_1, A_2, \ldots, A_n$ is a superkey for R.

I.e., the same definition as BCNF but FD is changed to MVD.

The decomposition into 4NF is analogous to BCNF decomposition. When the `StarMovie` relation is decomposed into 4NF relations, the relations become:

```
R1(name, street, city)
R2(name, title, year)
```

Normal Forms Hierarchy

Relations in 4NF $\subset$ BCNF $\subset$ 3NF ($\subset$ 2NF $\subset$ 1NF).

Property	3NF	BCNF	4NF
Eliminates redundancy due to FD's	Most	Yes	Yes
Eliminates redundancy due to MVD's	No	No	Yes
Preserves FD's	Yes	Maybe	Maybe
Preserves MVD's	Maybe	Maybe	Maybe

You should aim for at least BCNF for all relations. In some cases, 3NF is acceptable.

Again: you have to know what normal form your relations are in, and to know why if you choose a lower form than BCNF.

SQL — Stored Programs

- You Can Not Do Everything in SQL
- SQL/PSM
- Cursors
- Recursion
- Triggers

Stored Programs

SQL is not "Turing complete" so there are things that you cannot express in SQL — for example `if` statements and `while` statements.

In most practical situations such constructs are necessary. Alternative solutions:

1. Write a "normal program" in Java or some other programming language, call SQL from this language.

   This can result in a lot of data being shuffled between the database and the application program, and this is expensive.
2. Write the program and store the program in the database. This is called SQL/PSM, Persistent Stored Modules.

   In PSM, you can mix "normal" programming language statements with SQL.

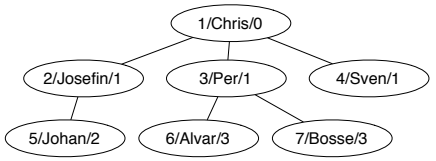
Example

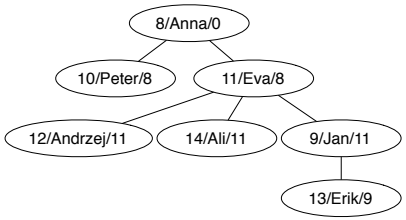
Employees in a company are described by the following relation:

```
Employees(nbr, name, bossNbr)
```

`nbr` is the employee number, `name` is the employee's (unique) name. `bossNbr` is the employee number of the employee's boss. A boss may have a boss, ... A top-level boss has `0` as `bossNbr`. There may be many top-level bosses.

Example:





Questions

- Who are the top-level bosses?

```
select name from Employees where bossNbr = 0;
```

- Who is Jan's immediate boss?

```
select e1.name
from Employees e1, Employees e2
where e2.name = 'Jan' and
      e2.bossNbr = e1.nbr;
```

- Who is Jan's top-level boss?

```
... cannot answer in SQL
```

(In the latest SQL standard there is a provision for recursive queries which would solve this problem. This is not available in MySQL.)

JDBC Solution

The answer to the last question is easy to obtain if you use Java (details regarding statement creation and deletion, and exceptions, are omitted):

```
public String getTopLevelBossName(String empName) {
    ResultSet rs = stmt.executeQuery
            ("select bossNbr from Employees where name = '" +
            empName + "'");
    String bName = empName;
    rs.next(); int bNbr = rs.getInt("bossNbr");
    while (bNbr != 0) {
        rs = stmt.executeQuery
            ("select name, bossNbr from Employees where nbr = " +
            bNbr);
        rs.next();
        bName = rs.getString("name");
        bNbr = rs.getInt("bossNbr");
    }
    return bName;
}
```


PSM Solution

```
create function getTopLevelBossName(in empName varchar(10))
    return varchar(10)
begin
    declare bNbr int;
    declare bName varchar(10);

    set bName = empName;
    select bossNbr into bNbr from Employees where name = empName;
    while bNbr <> 0 do
        select name, bossNbr
        into bName, bNbr
        from Employees
        where nbr = bNbr;
    end while;
    return bName;
end;
```


Comments

Note the following:

- Many syntactic differences from other programming languages (`declare` to declare local variables, `set` to assign a value, no parentheses around the condition in the `while` loop, etc.).
- You can mix PSM statements with SQL statements, and you use `select into` to assign the result of an SQL query to a PSM variable.
- The data types are the usual SQL types.
- You cannot have relations as parameters (the relation `Employees` in the example is "global" and must exist when the function is defined).

Running in MySQL

This you must try at home; you don't have the privileges to create stored functions at the LTH installation.

`mysql` and PSM both use a semicolon as delimiter. You must change the `mysql` delimiter before you define the function, and restore it afterwards.

```
delimiter //
create function getTopLevelBossName(empName varchar(10))
    return varchar(10)
begin
    ...
end;
//
delimiter ;
```

Call the function:

```
select getTopLevelBossName('Jan');
```


Cursors

When you wish to examine all tuples in a relation, you use a *cursor*. A cursor is a variable that runs through the tuples of a relation. Compare with `ResultSet` objects in JDBC and the `next()` function.

- Cursors must be declared as local variables.
- Cursors must be opened and closed.
- A tuple is fetched with the `fetch` statement.
- To detect that there are no more tuples, you declare a "continue handler" that checks for a specific SQL error code (`SQLSTATE`). Similar to an exception handler.

The procedure `copyToStaff` on the next slide copies all employee names from `Employees` to the `Staff` table. It is an example only, you could have copied like this in pure SQL:

```
insert into Staff select name from Employees;
```


A Cursor Example

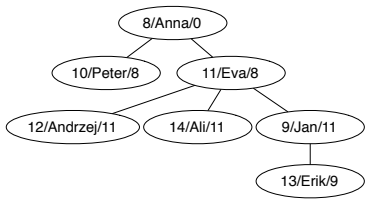
```
create procedure copyToStaff()
begin
    declare done int default 0;
    declare eName varchar(10);
    declare empCursor cursor for
        select name from Employees;
    declare continue handler for SQLSTATE '02000' set done = 1;

    delete from Staff;
    open empCursor;
    fetch empCursor into eName;
    while not done do
        insert into Staff values(eName);
        fetch empCursor into eName;
    end while;
    close empCursor;
end;
```


Recursive Procedures

PSM allows recursive procedures.

Example: Find all staff that are managed by a boss. Store the staff names in a relation `Staff(name)`.




```
call findStaff('Eva');
    => Staff = {'Andrzej', 'Ali', 'Jan', 'Erik'}
call findStaff('Jan');
    => Staff = {'Erik'}
call findStaff('Peter');
    => Staff = {}
```

Solution, 1

Do the following in mysql:

1. Create the `Staff` table (once):

```
create table Staff (
    name varchar(10)
);
```

2. Create the procedures `findStaff` and `findStaffRecursive` (once, see following slides).
3. Execute the procedure `findStaff`:

```
call findStaff('Eva');
```

4. Study the result:

```
select * from Staff;
```


Solution, 2

Set up for the recursion (clear the `Staff` table and call the recursive procedure):

```
create procedure findStaff(in bName varchar(10))
begin
    declare bNbr int;
    delete from Staff;
    select nbr into bNbr from Employees where name = bName;
    call findStaffRecursive(bNbr);
end;
```


Solution, 3

```
create procedure findStaffRecursive(in bNbr int)
begin
    declare done int default 0;
    declare eName varchar(10);
    declare eNbr int;
    declare empCursor cursor for
        select nbr, name from Employees where bossNbr = bNbr;
    declare continue handler for SQLSTATE '02000' set done = 1;

    open empCursor;
    fetch empCursor into eNbr, eName;
    while not done do
        insert into Staff values (eName);
        call findStaffRecursive(eNbr);
        fetch empCursor into eNbr, eName;
    end while;
    close empCursor;
end;
```


PSM Types and Statements

Types Same as the SQL data types. Local variables must be declared with `declare`.

Assignment

```
set empName = 'Bosse';

select name into empName
from Employees where nbr = 10;
```

If statement

```
if <condition> then
    <statements>
else
    <statements>
end if;
```

Use `elseif` for `else if`.

PSM Loops

Loop

```
[label:] loop
    ...
    if ... leave label;
    ...
end loop;
```

While loop

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

For loop Not implemented in MySQL.

Debugging Stored Procedures

It is not easy to debug stored procedures. It may be difficult even to find compilation errors, since the error messages are not very informative (usually "you have an error in your SQL syntax").

There is no "stored procedure debugger" where you can follow the execution of a stored procedure. What you can do is to insert "normal" `select` statements in a procedure (not `select into`). The results from such statements are sent directly to the database client (usually `mysql`).

Why Use PSM?

Some reasons for using stored procedures instead of client code:

- Essential to have the "business logic" in one place, instead of spread out over client programs. (For instance, banks often don't give applications access to database tables directly, they must perform all database actions via stored procedures.)
- More efficient.
- Clients in different languages on different platforms can perform the same database actions.

Triggers

A trigger is an active database element that is executed whenever a triggering event occurs. The triggering event can be an insertion, deletion or modification of a tuple in a relation.

Typical uses of triggers:

- make sure that an attribute contains a reasonable value (you can also use `check` on an attribute value when a table is defined, but only for simple cases),
- insert an audit tuple in another table when something is modified,
- perform checks so new information is consistent, and if it is not, roll back the transaction.

Trigger Example

A table which records texts with author and creation/modification date:

```
create table Comments (
    author   varchar(10),
    text     varchar(1000),
    modified date
);
```

A trigger that ensures that the `modified` field always contains the date of the modification:

```
create trigger commentsModified
    before insert or update on Comments
for each row
begin
    new.modified := now();
end;
```


MySQL Trigger Syntax

```
create trigger <trigger-name>
    [ before | after ] [ insert | update | delete ]
        on <table-name>
    for each row
    <trigger body in PSM>
```

Notes:

- `for each row` means "for each *modified* row". The trigger body is executed for each modified tuple. The new and old tuples are accessed with `new` and `old`.
- Note that the syntax `before insert or update` is not available in MySQL. You have to write one `insert` trigger and one `update` trigger. If the trigger body is complex, write a stored procedure and call it from both triggers.
- MySQL trigger support is still rudimentary. Standard SQL provides more possibilities.

Object-Oriented Databases, NoSQL

Object-Oriented Databases

Object-oriented databases are better than relational databases at handling complex data that arise in many applications:

- Images, video, sound, ... (multimedia in general)
- Spatial data (GIS, Geographical Information Systems)
- Biological data (DNA strings)
- CAD data (Computer Aided Design)
- Virtual worlds, games, ...

Object-oriented databases provide *persistent storage* for objects. The ODBMS and the application programming language are integrated.

Object-oriented databases *may* provide a query language, indexing, transaction support, distributed objects, ...

Shortcomings of RDBMS's

In a relational database system:

- Everything must be a relation — the logical model must be "flattened". E.g., a many-many relationship becomes a relation.
- There are no complex objects, apart from BLOB's (Binary Large Objects). BLOB's cannot be type checked.
- There is no inheritance.
- There is a mismatch between the data access language (SQL) and the host language (Java, C++, ...). You need a lot of time-consuming code to convert from tuples to objects, and vice-versa.

In an object-oriented database system the objects are moved "unchanged" between the program and the database.

Future of ODBMS's

- When should you use an ODBMS?
  When you have a need for high performance on complex data.
- Are ODBMS's ready for production applications?
  Yes, many new applications use ODBMS's.
- Are there any commercial ODBMS products?
  Objectivity, Gemstone, POET, Jasmine, ... All are very small compared to Oracle or IBM ...
- Will ODBMS's replace RDBMS's?
  Certainly not. ODBMS's are mainly used in new development.
- Can ODBMS's and RDBMS's coexist?
  Certainly, and this is what you usually see. OO is good for many things, relational also but for different things.

Standards?

There is not much standardization in the object-oriented database world. Most vendors provide their own solutions to different problems.

One standards body:

> ODMG, Object Data Management Group. Started in 1990, died in 2001. Standardized ODL (Object Definition Language) and OQL (Object Query Language).

Java standards:

> JDO (Java Data Objects). Has the goal to standardize data store access in Java, but is broader in scope than the ODMG attempts and does not follow ODL or OQL standards.
>
> JDBC and SQLJ will continue to exist but are limited to relational databases with SQL as the query language.

Object-Relational Databases

The latest SQL standard has extended SQL with object-oriented features. Relations are still the basis, but in the new standard:

- there are objects with data *and* methods,
- the objects may contain explicit references to other objects,
- an attribute in a tuple may contain an object.

MySQL does not support objects.

Persistence

When you use a programming language together with an ODBMS the objects you create are (may be) persistent, i.e., they outlive the execution of a program.

In an object-oriented database the objects are stored in "object format", instead of being stored as tuples in a relation, or even worse spread out over several relations.

Objects are loaded from the database when they are accessed by the program. Pointers (references) are automatically translated ("swizzled") back and forth between two representations: memory address or disk address.

Approaches to Persistence

Class based: persistent objects must be of a class that inherits from a `Persistent` class.

Object based: any object may be marked as persistent.

Reachability based: one "root" object is marked as persistent, all objects that can be reached from this object are also persistent.

In JDO (Java Data Objects, see for example `db.apache.org/jdo/`) reachability-based persistence is implemented.

Persistent Objects in Java

Different approaches:

Serialization: save/restore objects with explicit commands.

JDBC, SQLJ: interface to a relational database using standard SQL.

JDO: transparent persistence. Automatic persistence, persistent objects are treated the same as transient objects. The underlying data store may be a file system, a spreadsheet, a relational DBMS, an object-oriented DBMS, ...

Java Data Objects (JDO)

The primary participants in the JDO model are:

PersistenceCapable classes: the actual objects that are stored and fetched.

PersistenceManager: negotiates accesses, stores, transactions, and queries between applications and the underlying data store.

Transaction: handles ACID transactions.

Query: handles language-independent queries.

PersistenceCapable Classes

All user-defined classes can be made persistent. Some system classes are persistent, e.g., the `java.util.Collection` classes.

Persistent classes must implement the `PersistenceCapable` interface, but this is not visible in the user code. Instead:

- the class author provides an XML file with details about the class, e.g. which of the attributes that are persistent,
- a Class Enhancer tool processes the class file.

During runtime, objects of `PersistenceCapable` classes can be made persistent by calling the `PersistenceManager`. Each persistent object has its own unique identity in the data store and hence can be shared between different applications concurrently.

Managing Persistence

The `PersistenceManager` class contains the following methods to make objects persistent:

```
void makePersistent(Object pc);
void makePersistent(Object[] pcs);
void makePersistent(Collection pcs);
```

And methods to find and fetch persistent objects:

```
Object getObjectById(Object oid_or_pc, boolean validate);
Object getObjectId(Object pc);
```

And methods to delete persistent objects and to make persistent objects transient:

```
void deletePersistent(Object pc);
void makeTransient(Object pc);
```


Transactions

Transactions are handled by the class `Transaction`. Methods:

```
void begin();
void commit();
void rollback();
```

Almost the same as in SQL.

Queries

Queries in JDO are handled by the `Query` class. The methods for specifying `select`, `from`, and `where` are language independent (SQL, OQL, ...). Example:

```
class Employee {
    String name;
    Integer salary;
    Employee manager;
}

Collection extent = persistMgr.getExtent
        (Class.forName("Employee"), false);
Query q = persistMgr.newQuery(
        Class.forName("Employee"), // class
        extent,                    // candidates
        "salary > 50000");         // filter
Collection resultSet = q.execute();
```

Compare: `select * from Employees where salary > 50000;`

More Complex Queries

SQL query:

```
select e1.name
from Employees e1, Employees e2
where e1.salary > ? and
      e1.manager.name = e2.name and
      e2.salary > ?;
```

JDO query:

```
q.declareParameters("int sal");
q.setFilter("salary > sal and manager.salary > sal");
resultSet = query.execute(new Integer(50000)); // sal = 50000
```

Notice that joins are handled transparently. It is up to JDO to translate the filter specification into SQL (or OQL, ...).

NoSQL Overview

- Relational databases can be used to solve all kinds of problems.
- But are maybe not the right solution to all problems.
- New applications (often web-centric) have new requirements.
  - Huge amounts of data (terabytes or petabytes)
  - Simple data structure (often)
  - Must scale well
- NoSQL = "Not only SQL". A better name would be "Not only relational".
- A mixture of ideas, concepts, tools, products, ...

Examples, Lots of Data

Twitter 95 million tweets per day (1100 per second) must be stored. Only simple queries (based on primary key, no joins). Used MySQL earlier, now Cassandra (and more).

Facebook 500 million active users, half of them log in every day. Each user has 130 friends (on average). 30 billion pieces of content (links, texts, blog posts, photo albums) accessed every day. (Cassandra)

LinkedIn More than 90 million members, one new member every second. Two billion people searches per year. (Voldemort)

(The figures are from 2009-2010, may have grown ...)

Buy a Bigger Computer Instead?

- Big computers can store lots of data ...
- Big computers are expensive
- And you have to pay big license fees for a big Oracle installation
- Even big computers can fail
- Better to use a lot of cheap commodity PC-s
- And replicate data so one or a few failing nodes don't matter
- Design the storage system so it can be expanded (during uptime) by adding PC-s

Is It New?

- Yes: the term NoSQL is from 2009.
- But NoSQL databases have been around longer than that.
- And before anything NoSQL there were object-oriented databases, hierarchical databases, network databases, ...

Different Types of Data Stores

Key–Value A distributed hash table. Arbitrary key type; the value is a "blob". The application program must be aware of the structure of the value. (Amazon Dynamo)

Document As key–value, but the value is a document, and the DBMS knows that. (MongoDB, CouchDB)

Columns The value is a set of columns, like in a relational database, but they do not necessarily follow a schema. (Google BigTable, Cassandra)

Graph The database is a set of nodes with properties, and a set of connections between the nodes (with properties). (Neo4J)

The CAP Theorem

The CAP Theorem says that you cannot have all three of Consistency, Availability. and Partition tolerance.

- Strong Consistency: all clients see the same version of the data, even on updates to the dataset – e.g. by means of the two-phase commit protocol,
- High Availability: all clients can always find at least one copy of the requested data, even if some of the machines in a cluster is down,
- Partition-tolerance: the total system keeps its characteristic even when being deployed on different servers, transparent to the client.

Many NoSQL systems sacrifice consistency and go for BASE (next slide).

No ACID, BASE Instead

Transactions are no longer guaranteed to be ACID: atomic, consistent, isolated, durable). BASE is almost the opposite: basically available, soft state, eventually consistent.

BASE is optimistic and accepts that the database consistency is in a state of flux. "Eventual consistency" (actually more like durability) means that inaccurate reads are permitted just as long as the data is synchronized "eventually." (Compare with DNS, it takes time for changes to propagate.)

Amazon Dynamo

Dynamo was developed by Amazon.

- First used for the shopping cart, now also for other applications.
- Goal: always available, writes never fail.
- Key–value store. Records are replicated on several computers.
- Read & write: only single records.
- Operations: `get(key)` returns a value or a list of several versions of a value. The application must solve problems with inconsistencies.
- `put(key,value)` writes a value. The key is hashed, the hash code determines on which nodes the value should be stored ("consistent hashing").

Cassandra

First developed by Facebook, now a top-level Apache project.

- Key–value & replication like in Dynamo.
- But the value has structure: it contains columns (which are stored in column families which may be stored in super columns). A column has a name, a value, and a timestamp. Columns may be sorted on value or on timestamp.
- Inbox search at Facebook: 50+ TB of data stored on 150 machines.
  - Term search: the user id is the key. Words in messages are the super columns, message id's become the columns.
  - Interaction search: the user id is the key. Recipient id's are the super columns, message id's become the columns.

Computing Model

Not only storage should be distributed, but also computing. It is difficult to write parallel programs ... MapReduce is a new programming model.

- All data is treated as sets of key–value pairs. The key is a string, the value is a blob.
- All programs are sequences of alternating `map` and `reduce` functions.
- The map function processes a key–value pair and generates one or more intermediate key–value pairs.
- The reduce function merges all intermediate values associated with the same intermediate key.
- Map functions run in parallel on many computers, as do reduce functions

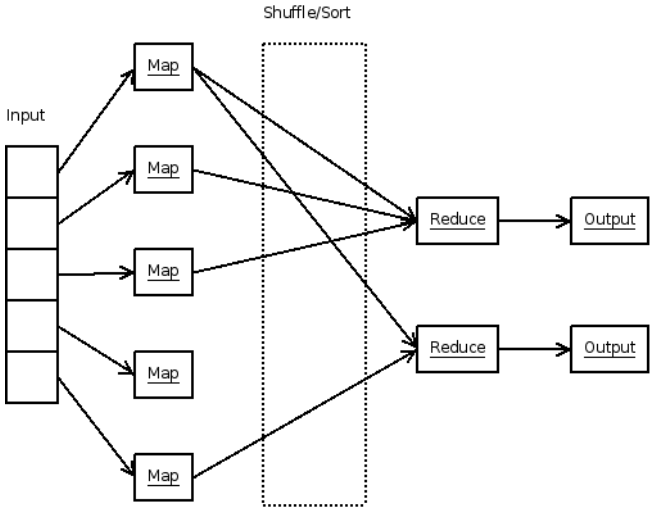
MapReduce Example

Compute word counts within a set of documents.

```
map(key, value):
    // key: document name
    // value: document text
    for each word w in value:
        emitIntermediate(w, 1)

reduce(key, values)
    // key: a word
    // values: a list of counts
    result = 0
    for each v in values:
        result += v
    emit(result)
```

MapReduce Data Flow



MapReduce Figures (From Google)

Execution on a cluster of 1800 machines, 2 × 2GHz processors, 4GB memory, 320GB disk, Gigabit Ethernet. The figures are from the original MapReduce paper, 2004.

Grep Scan through 10^{10} 100-byte records, searching for a three-character pattern. 150 seconds, including 60 seconds startup overhead.

Sort Sort 10^{10} 100-byte records. 15 minutes.

Google Google web search uses an index which is created with MapReduce.

MapReduce vs Traditional Databases

- Data has no explicit schema
  - The map and reduce functions must "understand" the data format.
  - Users have to write procedural code to interpret and process the data.
  - A step backwards?
  - Higher-level programming languages for MapReduce: PIG, Hive.
- Data is stored in files in a distributed file system.
- All processing is sort based — makes the programming easier, but may be a performance concern.

More Information

- `http://en.wikipedia.org/wiki/Nosql`
- `http://nosql-databases.org/`
- `http://nosql.mypopescu.com/`
- `http://www.vineetgupta.com/2010/01/nosql-databases-part-1-landscape/`

Logical Query Languages

- Datalog
- Predicates
- Datalog Rules
- SQL and Datalog
- Recursive Queries

Logical Query Languages

SQL is an "implementation" of an abstract programming language for relations called *relational algebra*. We will study relational algebra later (page 309).

In relational algebra (and in SQL) there are operations to manipulate sets (or bags) of tuples (selection, projection, joins, ...). There is a close relationship between set theory and logic, so the necessary operations can also be expressed in logic.

We will describe (parts of) the logical query language *Datalog*. Datalog is a subset of the logical programming language Prolog.

Predicates, Atoms, Facts

These are facts (or "atoms"):

```
parent_of(bill, mary) // the parent of bill is mary
parent_of(mary, john) // the parent of mary is john
```

`parent_of` is a *predicate*, a boolean valued function which returns true for the arguments in the example. For any other combination of arguments it returns false. This can be seen as a relation:

child	motherorfather
bill	mary
mary	john

An *arithmetic atom* is a comparison, for example $x < y$ or $x = 1$.

Datalog Rules

A movie relation (or predicate):

```
Movies(title, year, length, genre, studioName, producerC#)
```

A Datalog rule which defines a new predicate containing long movies (like an SQL view):

```
LongMovie(t,y) ← Movies(t,y,l,g,s,p) AND l ≥ 100
```

The body (right-hand side) of a rule consists of subgoals (atoms) connected by `AND`. Each subgoal may be negated with `NOT`.

Variables that occur only once in a rule may be replaced by anonymous variables:

```
LongMovie(t,y) ← Movies(t,y,l,_,_,_) AND l ≥ 100
```

Safe Rules

A Datalog rule must produce a finite relation. These rules are not correct:

```
Large(l) ← l ≥ 100
P(x,y) ← Q(x)
R(x) ← NOT S(x)
```

Definition (Safety Condition)

Every variable that appears anywhere in the rule must appear in some nonnegated, relational subgoal of the body.

The following rule is safe but forbidden by the condition (doesn't matter, the rule wouldn't be used in practice):

```
P(x) ← x = 1
```


Extensional and Intensional Predicates

Two kinds of predicates:

- *Extensional* predicates, which are predicates whose relations are stored in a database, and
- *Intensional* predicates, whose relations are computed by applying one or more Datalog rules.

The extensional predicates define the current instance of a relation.

SQL and Datalog

The basic SQL operations (actually relational algebra operations) can all be expressed in Datalog. For example the set operations union, intersection, and difference. Two relations, `R(A,B,C)` and `S(A,B,C)`:

```
R union S          U(x,y,z) ← R(x,y,z)
                   U(x,y,z) ← S(x,y,z)

R intersect S      I(x,y,z) ← R(x,y,z) AND S(x,y,z)

R except S         D(x,y,z) ← R(x,y,z) AND NOT S(x,y,z)
```


Projection and Selection

Projection:

```
select title, year      P(t,y) ← Movies(t,y,_,_,_,_)
from Movies
```

Selection:

```
select title, year      S(t,y) ← Movies(t,y,l,_,_,_) AND l ≥ 100
from Movies
where length >= 100
```

Joins

Two relations, `R(A,B)` and `S(B,C,D)`. Natural join:

```
select *                J(a,b,c,d) ← R(a,b) AND S(b,c,d)
from R natural join S
```

Theta join:

```
select *                J(a,rb,sb,c,d) ← R(a,rb) AND S(sb,c,d)
from R inner join S                        AND a < d
    on A < D
```


Recursive Queries

Datalog has one big advantage over SQL (relational algebra), namely that it is easy to express recursive queries (SQL-99 also has recursive queries, but this is not implemented in most DBMS's). Example:

```
// facts (define a tree)
parent_of(bill, mary)
parent_of(mary, john)
parent_of(ann, john)
parent_of(bob, mary)

// rules
ancestor_of(x,y) ← parent_of(x,y)
ancestor_of(x,y) ← parent_of(x,z) AND ancestor_of(z,y)
descendant_of(x,y) ← ancestor_of(y,x)
```


XML

- Overview
- Semistructured Data
- Valid XML and Well-Formed XML
- DTD's
- XML Parsers
- XPath, XSLT
- XML and Databases

XML

XML (eXtensible Markup Language) is a World-Wide Web Consortium (`www.w3.org`) standard for defining the structure and meaning of data stored in text documents. It's still under development.

Some Google search results:

XML	461 million hits
"XML Tutorial"	131 000 (!)
XML & Database	13 million

`java.sun.com` search:

XML	35000

Digesting the Alphabet Soup

The names of the following important XML standards and technologies have been fetched from the XML tutorial at `java.sun.com`:

> XML, SAX, DOM, JDOM, dom4j, DTD, XSL, XSLT, XPath, XML Schema, RELAX NG, TREX, SOX, XML Linking, XML Base, XPointer, XHTML, RDF, RDF Schema, XTM, XLink, XPointer, SMIL, MathML, SVG, DrawML, ICE, ebXML, cxml, CBL, UBL.

So there is a lot to learn . . . This is an introduction to get a feeling of what's it all about.

The Structure of Data

In the relational, object-oriented and object-relational data models data is structured according to a schema (or class, . . .). This makes searchable databases possible and is important for efficiency.

In the real world data often is unstructured. It can be of any type and it doesn't necessarily follow any organized format or sequence.

You sometimes need to handle unstructured data, but your programs must know *something* of the data to be able to handle it.

A new invention is *semistructured* data.

Semistructured Data

Semistructured data is organized enough to be predictable:

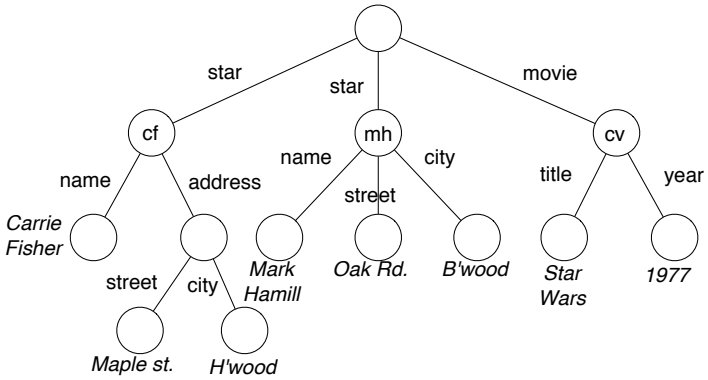
- Data is organized in semantic entities.
- Similar entities are grouped together.

But:

- Entities in the same group do not necessarily have the same attributes.
- The order of the attributes is not necessarily important.
- The presence of some attributes may not always be required.
- The type and size of attributes of entities in the same group may not be the same.

An HTML document is an example of semistructured data.

Semistructured Data, Example



This is a tree. You may introduce links (with attribute id's and idref's) to express a graph.

XML Basics

XML:

- is a language, like HTML, for markup of data,
- the markup is, unlike HTML, for the semantic meaning of data, not just for presentation,
- has no predefined tags,
- but extensible tags can be defined and extended based on applications and needs,
- and tags can have attributes.

Rules for XML Documents

Rules:

- An XML document must have *one* root.
- XML is case sensitive.
- All tags must be terminated: `<FirstName>Jennifer</FirstName>`
- Tags must be properly nested: `<Author> <name>Widom</name> </Author>`

Two Modes of XML

Well-formed XML:

Allows you to invent your own tags. Entirely schemaless.

Valid XML:

Involves a Document Type Definition (DTD) that specifies the allowable tags and how they may be nested. That is, a schema, but more flexible than a relational schema.

Well-Formed XML Example

```
<?xml version = "1.0" standalone = "yes" ?>
<Star-Movie-Data>
    <Star> <Name>Carrie Fisher</Name>
    <Address> <Street>123 Maple St.</Street>
        <City>Hollywood</City>
    </Address>
    </Star>

    <Star> <Name>Mark Hamill</Name>
        <Street>456 Oak Rd.</Street> <City>Brentwood</City>
    </Star>

    <Movie> <Title>Star Wars</Title> <Year>1977</Year>
    </Movie>
</Star-Movie-Data>
```

Document Type Definition (DTD)

A valid XML document follows a DTD, which is a "grammar" for XML documents. Example:

```
<!DOCTYPE Stars [
    <!ELEMENT Stars (Star*)>
    <!ELEMENT Star (Name, Address+, Movies)>
    <!ELEMENT Name (#PCDATA)>
    <!ELEMENT Address (#PCDATA | (Street, City))>
    <!ELEMENT Street (#PCDATA)>
    <!ELEMENT City (#PCDATA)>
    <!ELEMENT Movies (Movie*)>
    <!ELEMENT Movie (Title, Year)>
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT Year (#PCDATA)>
]>
```

`*` means 0–many times, `+` means 1–many times, `|` means "or", `PCDATA` is character data.

A Document Following a DTD

```
<?xml version = "1.0" standalone = "no" ?>
<!DOCTYPE Stars SYSTEM "star.dtd">
<Stars>
    <Star> <Name>Carrie Fisher</Name>
        <Address> <Street>123 Maple St.</Street>
            <City>Hollywood</City>
        </Address>
        <Address>5 Locust Ln. Malibu</Address>
        <Movies>
            <Movie><Title>Star Wars</Title>
                <Year>1977</Year> </Movie>
            <Movie><Title>Empire Strikes Back</Title>
                <Year>1980</Year> </Movie>
        </Movies>
    </Star>
    ...
</Stars>
```


More About DTD's

- An XML document is validated against a DTD by an XML parser.
- Any element or attribute not defined in the DTD generates an error.
- `SYSTEM` indicates that the DTD is intended for private use, `PUBLIC` references a public DTD.
- Importing external DTD's:

```
<!DOCTYPE rss PUBLIC
"-//Netscape Communications//DTD RSS 0.91//EN"
"http://my.netscape.com/publish/formats/rss-0.91.dtd">
```

(Rich Site Summary, XML format for sharing headlines and other web content.)

Attribute Lists

It should be clear that well-formed XML can describe a tree of data. However, it is possible to label XML nodes with attributes (ID's), and to use other attributes (IDREF's) to link to these nodes. In this way, an arbitrary graph may be described. A DTD with ID's and IDREF's:

```
<!DOCTYPE Stars-Movies [
    <!ELEMENT Stars-movies (Star* Movie*)>
    <!ELEMENT Star (Name, Address+)>
        <!ATTLIST Star
            starId ID
            starredIn IDREFS>
    <!ELEMENT Movie (Title, Year)>
        <!ATTLIST Movie
            movieId ID
            starsOf IDREFS>
    <!-- elements Name, ... as before -->
]>
```

A Document with ID's and IDREF's

```
<Stars-Movies>
    <Star starId = "cf" starredIn = "sw, esb, rj">
        <Name>Carrie Fisher</Name>
        <Address> <Street>123 Maple St.</Street>
            <City>Hollywood</City> </Address>
    </Star>
    <!-- similar for Mark Hamill, "mh" -->
    <Movie movieId = "sw" starsOf = "cf, mh">
        <Title>Star Wars</Title>
        <Year>1977</Year>
    </Movie>
    <!-- similar for Empire Strikes Back, "esb", and
    Return of the Jedi, "rj" -->
</Stars-Movies>
```


XML Parsers

XML documents are readable for humans but are intended to be handled by programs. In a program that processes an XML document you need to convert the XML text into program data structures. There are several ways to do this (in Java and in other languages that have implemented the standards). Two examples:

SAX (Simple API for XML):

> A "serial access" protocol. Event-driven: you register a handler with a SAX parser, and the parser invokes your callback methods whenever it sees a new XML tag, or encounters an error, or wants to tell you anything else.

DOM (Document Object Model):

> Converts an XML document into a tree of objects in your program. You can then manipulate the data in any way that makes sense: modify the data, remove it, or insert new data.

SAX

The SAX parser calls methods from the `ContentHandler` interface, which you must implement in your program (similar to a listener interface in AWT or Swing). Example:

```
<priceList>  [parser calls startElement]
    <coffee>  [parser calls startElement]
        <name>  [parser calls startElement]
        Mocha Java  [parser calls characters]
        </name>  [parser calls endElement]
        <price>11.95</price>  [parser calls startElement,
            characters, and endElement]
    </coffee>  [parser calls endElement]
    ...
```

Notice: with SAX, you can only read an XML document, not modify it in any way. The parser can check that an XML document is valid (follows a given DTD).

DOM

DOM defines a standard structure of XML documents in memory. The structure is a tree, with nodes for the different kinds of XML elements.

DOM contains:

- a parser, so you can parse an existing XML document and build a DOM representation in memory. Many DOM parsers use SAX parsers internally,
- an API to manipulate the DOM tree,
- an API to create a new XML document from a DOM tree.

Naturally, you can also create a DOM tree in your program and programmatically create an XML document.

Locating Data in a Document

Suppose that you have XML documents containing interesting data, and you want only specific parts of that data. You could write a program that parses a document, builds a DOM tree, and then searches that tree using the DOM API. But this is often not flexible enough.

Another example: in order to write a program that processes different parts of an XML data structure in different ways, you need to be able to specify the part of the structure you are talking about at any given time.

The XML Path Language, XPath, provides a syntax for locating specific parts of an XML document.

XPath

XPath is an addressing mechanism that lets you specify a path to an element so that, for example, `<article><title>` can be distinguished from `<person><title>`. That way, you can describe different kinds of translations for the different `<title>` elements. Examples:

`/h1/h2` select all `h2` elements under a `h1` tag,

`/h1[4]/h2[5]` select the fourth `h1` element, then the fifth `h2` element under that,

`/books/book/translation[.='Japanese']/../title` select the `title` element node for each book that has a Japanese translation.

As you can see from the examples, XPath expressions look like search paths in a tree structured file system. (There is much more to XPath than this . . .)

Transforming XML Documents

XML specifies how to identify data, but you often need to transform the data in predefined ways.

Examples:

- Present the data in a readable form, e.g., in HTML, XHTML, plain text, . . . Note that XML is text, but it is not intended to be read.
- Create another XML document, maybe in a different format.

Naturally, you can do this programmatically, but more often you use XSLT, Extensible Stylesheet Language for Transformations. XSLT uses XPath to match nodes.

XSLT is the first part of XSL, Extensible Stylesheet Language. The second part is XSL formatting objects.

An XSLT Example

One common use for XSLT is to transform XML documents into HTML. A stylesheet specifies which transformations that should be applied. A simple template stylesheet (*intro.xsl*):

```
<?xml version = "1.0"?>

<xsl:stylesheet version = "1.0"
    xmlns:xsl = "http://www.w3.org/1999/XSL/Transform">

    <xsl:template match = "myMessage">
        <html>
            <body><xsl:value-of select = "message"/></body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

XPath is used to find the `myMessage` element, then the `message` element. The value of the `message` element is placed inside HTML `<body>` tags.

A Stylesheet Applied

An XML document that uses this stylesheet:

```
<?xml version = "1.0"?>

<?xml stylesheet type = "text/xsl" href = "intro.xsl"?>

<myMessage>
    <message>Welcome to XSLT!</message>
</myMessage>
```

This is a normal XML document, with the addition of the `xml stylesheet` tag. More in XSLT:

- variables,
- iteration, sorting,
- conditional processing.

XSLT Processors

The work necessary for transforming an XML document is performed by an XSLT processor. Some alternatives:

- Write your own processor. There are standard classes in Java (JAXP) and PHP (XSL extension) to do this.
- If all you want is HTML presented in a web browser, use Internet Explorer and the processor `msxml` from Microsoft.

Custom Markup Languages

XML is used as a markup language in many application areas. Examples:

MathML	mathematical formulas (like LaTeX, but processor independent)
CML	chemical formulas, molecules, ...
SMIL	multimedia presentation
SVG	scalable vector graphics. Lines, curves, etc.
XBRL	business information

XML and Databases

An XML document is a collection of data, so in the strictest sense it is a database. It has some advantages as a database:

- self-describing, portable, can define data in trees or graphs
- flexible schemas (DTD's, XML Schema)
- query languages (XPath, XQuery, ...)
- programming interfaces (SAX, DOM, ...)

Also disadvantages:

- verbose, access to data is slow (needs parsing)
- lacks indexes, security, transactions, multi-user access, triggers, queries across multiple documents, ...

XML for Transporting Data

One important use of XML in database systems is not to store data, but to transport data extracted from a (relational) database.

In commercial database systems there are nowadays applications for this "serializing" of data.

Example, e-commerce: Use a relational database to store information about products, customers, etc. Use XML documents to transport this information. Use an XSL stylesheet to convert the information for presentation.

Extracting XML from a Database

An example of an XSLT template for a query returning XML from a relational database:

```
<?xml version="1.0"?>
<FlightInfo>
    <Intro>The following flights are available:</Intro>
    <Select>SELECT airline, flightNumber, depart, arrive
        FROM Flights</Select>
    <Flight>
        <Airline>$airline</Airline>
        <FltNumber>$flightNumber</FltNumber>
        <Depart>$depart</Depart>
        <Arrive>$arrive</Arrive>
    </Flight>
</FlightInfo>
```

When the template is processed, the query is executed and an XML document with the appropriate format is produced.

XML for Storing Data

If your data is not structured in a way so it can be conveniently described by a relational (or object-oriented) schema, you can use a native XML database.

As an example, suppose you have a Web site built from a number of XML documents, and you would like to provide a way for users to search the contents of the site. In this case, you could use a native XML database and execute queries in an XML query language.

Native XML Databases

One possible definition of a native XML database is that it:

- defines a logical model for an XML document, and stores and retrieves documents according to that model,
- has an XML document as its fundamental unit of logical storage, just as a relational database has a tuple in a relation as its fundamental unit of logical storage,
- is not required to have any particular physical storage model. For example, it can be built on a relational database, or an object-oriented database, or use a normal file system.

Note that it is not required that XML documents be stored as text. They may equally well be stored in some other format, such as the DOM model.

Relational Algebra

- Basics
- Set Operators
- Relational Operators (π, σ, $\times$, $\bowtie$)

Relational Algebra

SQL and other query languages have a theoretical basis. This basis is called relational algebra, "computing with relations".

Relational algebra is an algebra that operates on sets of tuples. It must be modified somewhat to handles bags (multivalued sets), which are used in commercial DBMS's.

Relational algebra is good for:

- understanding what queries that can be expressed,
- expressing queries non-ambiguously and compactly,
- reasoning about queries, e.g., which queries that are equivalent to each other,
- planning and optimizing query execution (only of interest for query language implementers).

Sets or Bags?

SQL handles *bags* instead of sets. In a bag, each value may appear several times. The motivation behind this is that bags are more efficient.

Consider the "union" operation:

- Take the union of two sets — you have to check each tuple in the result so it only appears once.
- Take the union of two bags — you just have to concatenate the bags.

Basics of Relational Algebra

The operands in a relational algebra operation are relations. Basic operations:

- Set operations — union, intersection, difference.
- Operations that remove parts of a relation: "selection" (remove tuples) and "projection" (remove attributes).
- Operations that combine the tuples of two relations: Cartesian product, different join operations.
- A renaming operation that changes the name of a relation or the names of attributes.

Set Operations

The set operations operate on two relations R and S:

$R \cup S$	Union
$R \cap S$	Intersection
$R - S$	Difference

Naturally, R and S must be compatible in the sense that they have the same attributes with the same types in the same order.

In SQL:

```
R union S
R intersect S
R except S
```


Projection

Project: remove attributes. Operator π ("pi" for "project").

title	year	length	inColor	studioName	prodCNbr
Star Wars	1977	124	true	Fox	12345
Mighty Ducks	1991	104	true	Disney	67890
Wayne's World	1992	95	true	Paramount	99999

$\pi_{title,year,length}(Movie)$

title	year	length
Star Wars	1977	124
Mighty Ducks	1991	104
Wayne's World	1992	95

$\pi_{inColor}(Movie)$

inColor
true

Projection in SQL

The projection operator corresponds to the "select list" of an SQL `select` statement:

```
select title, year, length
from Movie;

select inColor
from Movie;
```

But note that the second SQL statement produces a bag with three tuples instead of a set:

inColor
true
true
true

`select distinct` produces a set.

Selection

Select: choose tuples based on some condition. Operator σ ("sigma" for "select").

title	year	length	inColor	studioName	prodCNbr
Star Wars	1977	124	true	Fox	12345
Mighty Ducks	1991	104	true	Disney	67890
Wayne's World	1992	95	true	Paramount	99999

$\sigma_{length \geq 100}(Movie)$

title	year	length	inColor	studioName	prodCNbr
Star Wars	1977	124	true	Fox	12345
Mighty Ducks	1991	104	true	Disney	67890

Selection in SQL

The selection operator corresponds to the "where condition" of an SQL statement:

```
select *
from Movie
where length >= 100;
```

Note that the selection operator of relational algebra has nothing to do with the `select` clause of SQL ...

Cartesian Product

Cartesian product, "cross product": combine every tuple from a relation with every tuple from another relation. Operator $\times$.

$R =$

A	B
1	2
3	4

$S =$

B	C	D
2	5	6
4	7	8
9	10	11

$R \times S =$

A	R.B	S.B	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

Cartesian Product in SQL

In SQL, when selection from two relations with a `*` in the `select` clause and no `where` clause, is performed, the result is the Cartesian product between the relations:

```
select *
from R, S;
```

Alternatively:

```
select *
from R cross join S;
```

There is usually not much use for the "unrestricted" Cartesian product. When you restrict the product with conditions in the `where` clause, you get a join instead.

Natural Join

Natural join: take the Cartesian product of two relations, but keep only the tuples whose values match on attributes with the same name. Operator $\bowtie$.

$R =$

A	B
1	2
3	4

$S =$

B	C	D
2	5	6
4	7	8
9	10	11

$R \bowtie S =$

A	B	C	D
1	2	5	6
3	4	7	8

Natural Join in SQL

In SQL, a natural join is performed when you select from two relations, give a `where` condition that expresses equality, and restrict the `select` list:

```
select A, R.B, C, D
from R, S
where R.B = S.B;
```

or

```
select *
from R natural join S;
```


Another Natural Join Example

More than one attribute can participate in a natural join:

$U =$

A	B	C
1	2	3
6	7	8
9	7	8

$V =$

B	C	D
2	3	4
2	3	5
7	8	10

$U \bowtie V =$

A	B	C	D
1	2	3	4
1	2	3	5
6	7	8	10
9	7	8	10

Theta Join

Theta join: join relations on an arbitrary condition. Operator $\bowtie_C$ (the condition is C).

$U =$

A	B	C
1	2	3
6	7	8
9	7	8

$V =$

B	C	D
2	3	4
2	3	5
7	8	10

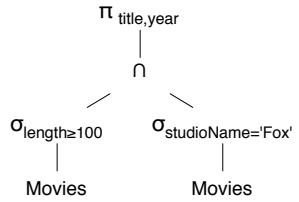
$U \underset{A<D}{\bowtie} V =$

A	U.B	U.C	V.B	V.C	D
1	2	3	2	3	4
1	2	3	2	3	5
1	2	3	7	8	10
6	7	8	7	8	10
9	7	8	7	8	10

Combining Operations

Relation `Movies(title, year, length, filmType, studioName)`. What are the titles and years of movies made by Fox that are at least 100 minutes long?

As an expression tree:



In linear form:

$$\pi_{title,year}(\sigma_{length \geq 100}(Movies) \cap \sigma_{studioName='Fox'}(Movies))$$

Equivalent Queries

$$\pi_{title,year}(\sigma_{length>=100}(Movies) \cap \sigma_{studioName='Fox'}(Movies))$$
$$\Longleftrightarrow \pi_{title,year}(\sigma_{length>=100 \wedge studioName='Fox'}(Movies))$$

It is the task of the *query optimizer* to rewrite queries in the most efficient form. To do this, it uses:

- relational algebra rules,
- ad hoc rules, e.g., perform selections as early as possible,
- usage statistics collected by the DBMS.

Join Example

Two relations:

```
Movies1(title, year, length, filmType, studioName)
Movies2(title, year, starName)
```

Find the stars of movies that are at least 100 minutes long.

$$\pi_{starName}(\sigma_{length \geq 100}(Movies1 \bowtie Movies2))$$

Independent Operations

The following operations are "independent", i.e., basic:

- union
- difference
- selection
- projection
- Cartesian product
- renaming (operator ρ, renames a relation or an attribute)

The other operations can be expressed in terms of the basic operations. Examples:

$$\begin{aligned} R \cap S &= R - (R - S) \\ R \underset{C}{\bowtie} S &= \sigma_C(R \times S) \end{aligned}$$

Outerjoins

As described earlier, the purpose of the join operation is to match tuples from two relations that agree on the values of two attributes.

There are cases when you wish to include "dangling" tuples in the output, i.e., tuples that fail to match with a tuple in the other relation. The missing attributes are padded with `null`.

These cases are handled by different kinds of outerjoins, operator $\overset{\circ}{\bowtie}$.

In practice, you normally consider one of the relations as the "basis", the tuples of which you want included in the output even if they have no match in the other relation. Then, you use a *left* or *right* outerjoin, indicated with L or R on the operator.

Outerjoin Example

Two relations and the result of their natural left outer join:

$U =$

A	B	C
1	2	3
4	5	6
7	8	9

$V =$

B	C	D
2	3	10
2	3	11
6	7	12

$U \overset{\circ}{\bowtie}_L V =$

A	B	C	D
1	2	3	10
1	2	3	11
4	5	6	null
7	8	9	null

Implementation of DBMS's

- Write Your Own DBMS
- Disk Storage
- Indexes
- B-trees
- Query Compilation
- Query Execution
- Algorithms

Write Your Own DBMS

Students take courses:

```
Students(pNbr, name)
TakenCourses(pNbr, courseCode, grade)
```

Design your own DBMS for handling these data:

- Schema information in one file:

```
Students#pNbr#STR#name#STR
TakenCourses#pNbr#STR#courseCode#STR#grade#INT
```

- Each relation in one file. The student file:

```
790101-1234#Bo Ek
770403-4321#Eva Alm
```


Process a Query

```
select courseCode, grade
from Students, TakenCourses
where name = 'Eva Alm' and
      Students.pNbr = TakenCourses.pNbr;
```

Your own DBMS must do the following:

1. Read the schema file to determine the attributes and their types.
2. Check that the where-condition is valid for the relations.
3. Read the relation files into memory.
4. Perform the join:

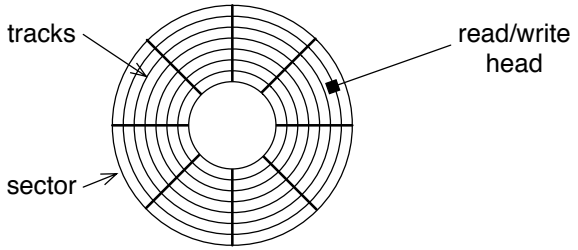
```
for each tuple s in Students
    for each tuple t in TakenCourses
        if where-condition is true
            output the course code and the grade
```

Your Own DBMS is No Good

Writing your own DBMS is not a good idea:

- The tuple layout on disk is inadequate, with no flexibility when the database is modified.
- Searching is very expensive — you always have to read an entire relation.
- Much better ways of doing joins are available.
- There is no concurrency control.
- There is no reliability.

Disk Storage



This disk: one surface, 8 tracks, each track 8 sectors.

Typical: 16 surfaces, 16384 tracks, each 128 sectors (avg.), 4 kB per sector, 128 GB. Average disk latency (time to read/write a block): 10 ms.

The number of disk accesses must be minimized — the CPU can execute millions of instructions in 10 ms.

Storing Relations

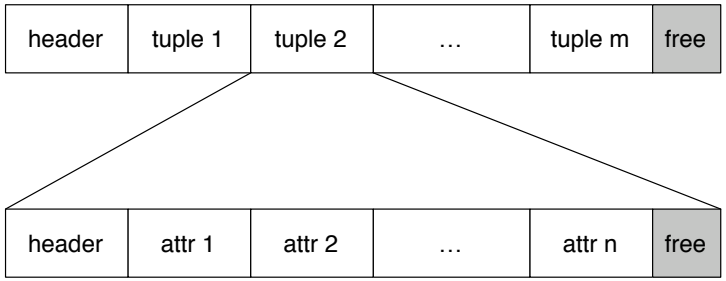
The smallest unit that can be read from or written to a disk is a disk block (a disk sector). A common block size is a few kilobytes, e.g., 4kB.

Most relations are much larger than 4 kB, so a relation must be stored in several blocks. Some DBMS's rely on the underlying operating system (file system) to handle such issues, but most take over the block handling themselves.

It is advantageous if the blocks that a relation occupies are "close" to each other. Preferably the blocks should be stored in the same "cylinder" (same track on many surfaces), in order to minimize head movement.

Storing Tuples

The tuples of a relation need to be packed into disk blocks. Example layout for one block:



A header may contain links to the table schema, length, timestamp, checksum, ...

Data Modifications

Change contents of a tuple:

- find the tuple, modify it, write it back.

Insert a tuple:

- unsorted relation: find space for the tuple, write it.
- sorted relation: find space in the current block, write it. If no space in the current block, create overflow block.

Delete a tuple:

- delete the tuple. May be possible to reclaim a block or to do away with an overflow block.

Note: it may be advantageous not to fill blocks as much as possible, to prepare for future insertions.

Indexes

As noted before, the DBMS often automatically creates an index on the primary key of a relation:

```
create table Stars (
    name      varchar(20) primary key,
    birthDate date
);
```

Now the following kind of query can be done quickly:

```
select * from Stars where name = 'Carrie Fisher';
```

Indexes also help in joins. We can also create an index on an arbitrary attribute:

```
create index BirthDateIndex on Stars(birthDate);
```

This makes queries of the following kind efficient:

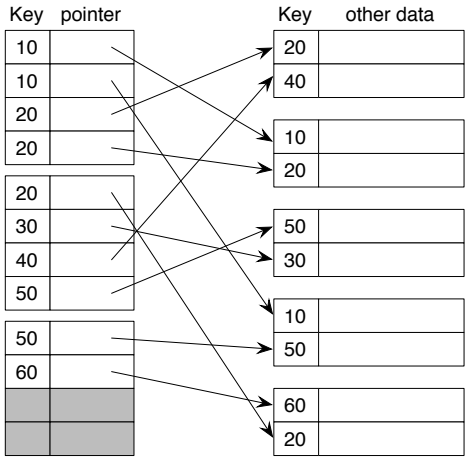
```
select * from Stars where birthDate = '1952-01-01';
```


An Idea for an Index

The index file (left, arrows are disk addresses):

- small, may perhaps be kept in main memory,
- sorted on the key, may use binary search to find a key.

"Key" is not necessarily the primary key; it is any attribute that has an index.



Multi-level Indexes

If the index fits into main memory, you need only one disk access to retrieve a tuple with a specific value.

If the index does not fit into main memory and you intend to use binary search, you first have to read the middle block, then read the middle of the next half, ... Then, you need many disk accesses.

To help in such situations, you can create a multi-level index. In principle, you have a sparse index for the index file. The most common type of multi-level index is a B-tree.

A different strategy is to use a hash table for the index. The in-memory versions of hash tables must be modified for use with secondary memory.

B-trees

A B-tree is a data structure that:

- automatically maintains as many levels of index as is appropriate (normally three levels),
- manages the space on the blocks so that every block is between half used and completely full. No overflow blocks are needed,
- automatically balances the tree.

A B-tree is like a balanced binary search tree:

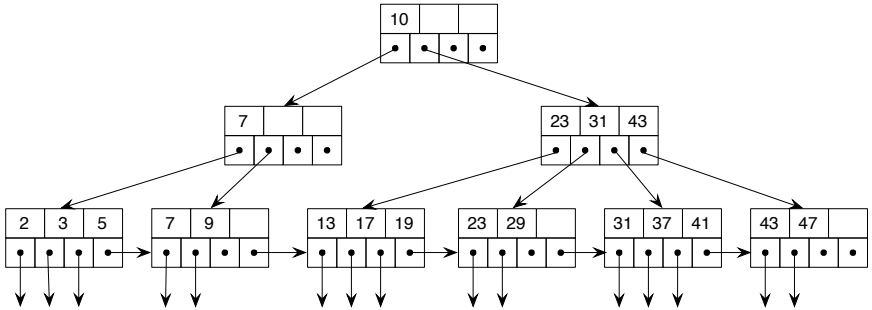
- you look for key values by traversing the tree, to the left if the key is less than the node value.

But also unlike a binary tree:

- packs n key values and $n+1$ pointers in each node,
- typically, a node is a disk block with space for many (hundreds) key/pointer pairs.

A B-tree Example

A B-tree with integer keys, $n = 3$:



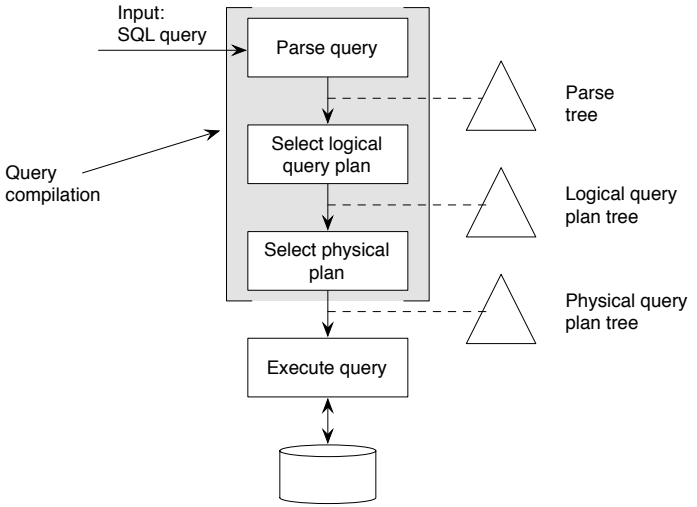
The first n pointers in a leaf node point at data blocks (or are `null` if the node isn't filled). The last pointer points at the next leaf node.

B-tree Observations

Some things to observe regarding B-trees:

- B-trees supports fast lookup of a specific key. At most three disk accesses, and normally less since the root node may be kept in primary memory.
- B-trees support range queries on the key, `where low<=key and key<high`. Find the leaf node containing the lower limit, then use the pointer to the next leaf node, etc., until the upper limit is found.
- As an extreme, you can get all the keys in sorted order by starting in the leftmost leaf node.
- Insertion is easy. If a leaf node overflows a new leaf node must be created, and the parent node updated (and maybe split).
- Deletion is easy. Some DBMS's don't bother to fix up the node structure on deletion, on the assumption that most relations grow in size.

Query Compilation and Execution



The Need for Optimization

Consider this query (similar to a previous example, but `Students` and `TakenCourses` have been renamed to `S` and `TC`):

```
select *
from S, TC
where name = 'Eva Alm' and
      S.pNbr = TC.pNbr;
```

There are many ways to evaluate the query. Alternatives, expressed in relational algebra:

1. $\sigma_{name='EvaAlm' \wedge S.pNbr=TC.pNbr}(S \times TC)$
2. $\sigma_{name='EvaAlm'}(S \bowtie TC)$
3. $\sigma_{name='EvaAlm'}(S) \bowtie TC$

These alternatives have different efficiency, and the query compiler has to choose the most efficient.

Estimates of Costs

Suppose that the `S` relation has 500 tuples. Each student has taken 20 courses, so the `TC` relation has 10,000 tuples. Further suppose that no indexes are present and that all intermediate results are written to disk.

1. Read `S` and `TC`, $500 + 10,000$ disk accesses.
   Take the product, write it, $500 \cdot 10,000$.
   Read again to check the condition, $500 \cdot 10,000$.
   Total $10,010,500$ disk accesses.
2. Read `S` and `TC`, $500 + 10,000$.
   Join, write, $500 \cdot 20$.
   Read and select, 10,000.
   Total $30,500$ disk accesses.
3. Read `S`, 500.
   Select and write, 1.
   Read `TC` and result of select, then join, $10,000 + 1$.
   Total 10,502 disk accesses.

Parsing a Query

The first phase of query compilation is parsing, i.e., checking that the SQL query is syntactically correct. Parsing is performed by all compilers, regardless of language. The result is a parse tree ("syntax tree").

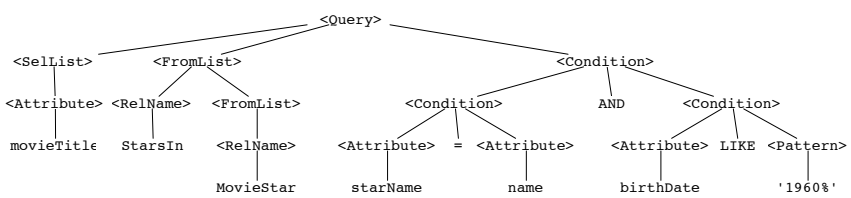
SQL syntax is described by a grammar. The parser checks the query against the grammar.

Example grammar (much simplified):

```
<Query>    ::= SELECT <SelList> FROM <FromList>
               WHERE <Condition>
<SelList>  ::= <Attribute> , <SelList> | <Attribute>
<FromList> ::= <RelName> , <FromList> | <RelName>
```

...and so on

The Parse Tree




```
select movieTitle
from StarsIn, MovieStar
where starName = name and
      birthDate like '1960%';
```

Semantic Checks

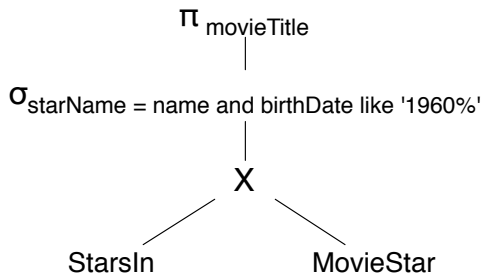
The parser checks that the SQL query is syntactically correct, i.e., that the "form" of the query is correct. But other things must be checked as well, for example:

- Relation uses: every relation mentioned in the query must exist in the database schema.
- Attribute uses: every attribute must be defined in the relation schemas.
- Types: all attributes must be of the correct type for the expression in which they occur.

Logical Query Plans

The next step in query compilation is to transform the parse tree into an equivalent logical query plan, expressed as an "algebraic expression tree". In this tree, the nodes are relational-algebra operators.

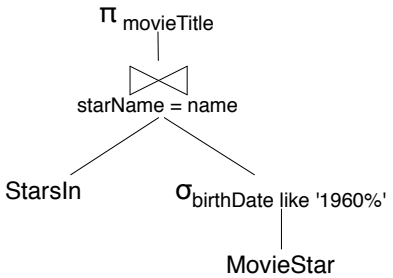
Example (query on a previous slide):



Query Rewriting

The process of transforming the parse tree into a logical query plan is mechanical, but the result probably isn't the most efficient query plan. The plan needs to be rewritten using different algebraic laws and heuristic techniques.

Example rewrite:



Laws for Rewriting

There are rules that can be applied for query rewriting. For instance, commutative and associative rules:

$$R \cup S = S \cup R$$
$$(R \cup S) \cup T = R \cup (S \cup T)$$

$$R \bowtie S = S \bowtie R$$
$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

There is a lot of similar laws.

One of the most important rules for optimization is "pushing selection", i.e., performing selection as early as possible. The intuitive motivation behind this is that all other operators will perform better if their operands are smaller relations.

One such rule:

$$\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$$

Physical Query Plans

The final step in query compilation is to transform the preferred logical query plan into a physical query plan.

The input is the algebraic tree that corresponds to the logical query plan. The output is a similar tree where the operators are physical operators.

Some physical operators have a direct correspondence in relational algebra, e.g., select, project, and join.

Other operators are necessary as "helpers":

- Scan — read an entire relation, or the tuples that match a simple criterion.
- Sort-scan — as scan, but produce a sorted result. That the result is sorted is used by some other algorithms, e.g., some join algorithms.

Physical Query-Plan Operators

There are many variants of the physical operators. Some work by reading the data from disk only once, some read the data twice or more times.

The DBMS determines which variant to use, by examining the sizes of the operands, the presence of indexes, and the amount of available primary memory.

As an example, consider sorting a relation (`order by` in SQL).

Small relation that fits in memory:

- read the entire relation, sort it with a good in-memory algorithm (Quicksort, ...).

Large relation:

- use merge-sort (following slide).

Merge-Sort

Merge-Sort usually is preferred for external sorting:

1. Repeat until the entire relation has been read:
   1. Fill all available memory with blocks from the original relation to be sorted.
   2. Sort the records that are in main memory.
   3. Write the sorted records onto new blocks of secondary memory, forming one sorted sublist.
2. Merge all the sorted sublists into a single sorted list.

Join Algorithms

We assume that we shall join two relations `R(X,Y)` and `S(Y,Z)` on the attribute (set of attributes) `Y`, i.e., natural join.

Several join algorithms are possible:

- Nested loop (two `for` statements and a test, as in "your own DBMS", slide 332). Only suitable for small relations, but may be used as a subroutine by other algorithms.
- One relation in memory. Used when one of the relations fits in memory and there are no indexes.
- Sort-based join. Used with large relations without indexes.
- Sort-based index join. For large relations with indexes.

One Relation in Memory

Join R(X,Y) and S(Y,Z). At least one of the relations, say S, fits in main memory. Neither R nor S has an index on Y.

The join can be performed in this way:

1. Read all the tuples of S and form them into a main-memory search structure (hash table, balanced binary tree, ...) with Y as the search key.
2. Read each block of R. For each tuple of R, find the tuples of S that match. Join the tuples, output them.

Sort-Based Join

Join R(X,Y) and S(Y,Z). R and S are large, and there still isn't an index on the common attribute Y. Then, you can join as follows:

1. Sort R, using merge sort.
2. Sort S similarly.
3. "Merge-join" the sorted R and S. Use one buffer for the current block of R, one buffer for the current block of S. Repeat:
   - Find the smallest value y that is at the front of the blocks for R and S.
   - If y appears only in one of the relations, drop all tuples with value y.
   - Otherwise, find all tuples from both relations having this value. If necessary, read blocks from R and/or S, until it is certain that there are no more y's in either relation.
   - Output tuples that can be formed by joining tuples from R and S with a common y value.

Index-Based Join

When there is a B-tree index on a relation we can obtain the tuples of the relation in sorted order from the index.

To perform the join between R(X,Y) and S(Y,Z) when there is an index on one or both of the Y's, we use the sort-join algorithm but we can skip one or two of the initial sorting steps of the algorithm.

Note that we don't have to read the entire relations, only the tuples that actually join.

Choosing a Physical Query Plan

The logical query plan must be transformed to a physical query plan. Normally, this is done by considering many different plans and choosing the one with the least estimated cost.

When enumerating possible physical plans, we select for each plan:

- An order and grouping for associative and commutative operations.
- An algorithm for each operation.
- Additional operators (scanning, sorting, ...) that are needed for the physical plan.