

# Crash Course Relational Databases

hajo@chalmers.se

# A Scenario...

Assume you're the manager...

Managing consults and projects and...

There are no computers...!

Many consults, many projects...(who's working where and when?)

How would you handle it?...

# Using a Table

Probably using some kind of table

<b>Consult</b>	<b>Phone</b>	...	...	<b>Project</b>	<b>Account</b>
Sven	070712345			Nightmare	09-34245-12
Olle	...			Kamikaze	...
Fia	...			NoSurvivors	...

Hmm, ...this will cause problems (anomalies)....!

# Anomalies

## Problems

- If Sven will work on 2 projects → another row with much common data (duplicate data)
- If so, if Sven moves, have to update many rows (possible inconsistency)
- If only one consult on a project, and consult quits, where to put the project? Half row empty...
- Add a new project. Where to put it if we don't know who will work on it
- ... have duplicated data (bad as we know)

Think OO: We mixed two objects (bad analysis)!

# Solving Anomalies

Better to use one table for consults and one for projects

Consult	Phone	...
Sven	070712345	...
Olle	...	...
Fia	...	...

Project	Account
Nighmare	09-34245-12
Kamikaze	
NoSurvivors	

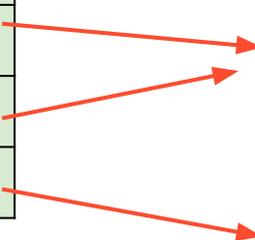
Data in tables independent (many anomalies solved)

But how to know which consult on which project??

# Relationships

To connect consults with projects we use **unique** id's as references between tables. This is called a **relationship**

Id	Consult	Phone	...	ProjId
1	Sven	070712345	...	1
2	Olle	...	...	1
3	Fia	...	...	3



Id	Project	Account
1	Nighmare	09-34245-12
2	Kamikaze	...
3	NoSurvivors	...

Sven and Olle works on Nightmare, Fia on NoSurvivors. No one is assigned to Kamikaze

# Primary and Foreign Key

The id's used to connect tables are known as **keys**

Primary keys (for Consults).  
Must be unique for every  
row



This is a join column (joining  
two tables)



Id	Consult	Phone	...	ProjId
1	Sven	070712345	...	1
2	Olle	...	...	1
3	Fia	...	...	3



Foreign keys (primary key  
for projects (or other table),  
possible non unique)

# Relationship Multiplicity

Depending on the rules for the company we possibly have different multiplicity of relationships

- One consult always works on one project (1:1, one to one)
- One consult possible works on many projects (1:N or 1:\*, one to many)
- Many consults possible work on same project (N:1 or \*:1, many to one)
- There can be many consults working on many projects (M:N, \*\*:\*, many to many)

We must be able to handle all these

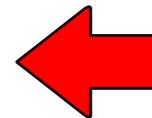
- We have seen N:1 so far
- 1:1, join column id's must be unique
- 1:N, put join column in project table
- But M:N...???

# Many to Many Relationships

Id	Consult	Phone	...
1	Sven	070712345	...
2	Olle	...	...
3	Fia	...	...

Id	Project	Account
1	Nighmare	09-34245-12
2	Kamikaze	...
3	NoSurvivors	...

ConsultId	ProjectId
1	2
1	3
2	3
3	1



Many to many relationships need an extra table, a jointable. Jointable has joincolumns (only)

- A consult, Sven, works on many projects (2,3)
- Many consults (1,2) works on one project, NoSurvivors

# Summary Multiplicity

## Assume tables A and B

Consider the primary keys for A (a column)

- If all keys appears in one and only one location (row) in B (as foreign key), then we have 1:1
- If any key appears in more locations (rows) we have a 1:N

Consider primary key for B

- Same as above

If any of A's key appears in multiple times in B and any of B's key appears multiple times in A, we have M:N

# The Relational Model

What we accomplished so far can be formalized to "**the relational model**"

*"The relational model for database management is a database model based on first-order predicate logic, first formulated and proposed in 1969 by E.F. Codd."*

- Tables (relations) are in fact sets of tuples (rows)
- Tuples have attributes (columns) with **primitive** values (no objects or lists,...)...
- ..or possible **NULL** (unknown or missing value)
- Attribute values have types (similar to Java, String = VARCHAR (20))
- Note: No ordering of rows (a set is unordered)

# Relational Database Management Systems, RDBMS

The software implementation of the relational model will show up as a RDBMS

- Handling collections of tables and much more
- Normally run as a **database server** (on a dedicated machines)
- For a collection of tables we just say a **database**
- Operations to create/delete(drop) database, create/delete (drop) tables, **create/read/update/delete** rows (**CRUD**), and more...
- Advanced and very efficient methods for searching

# JavaDB (Derby)

We'll use the Derby RDBMS in this course

- Bundled with NetBeans, see Services tab
- Will run on same machine

Possible to **create/drop a database** from inside Netbeans

When database created possible to **create/drop tables** (all tables should belong to a "schema" APP (similar to Java package))

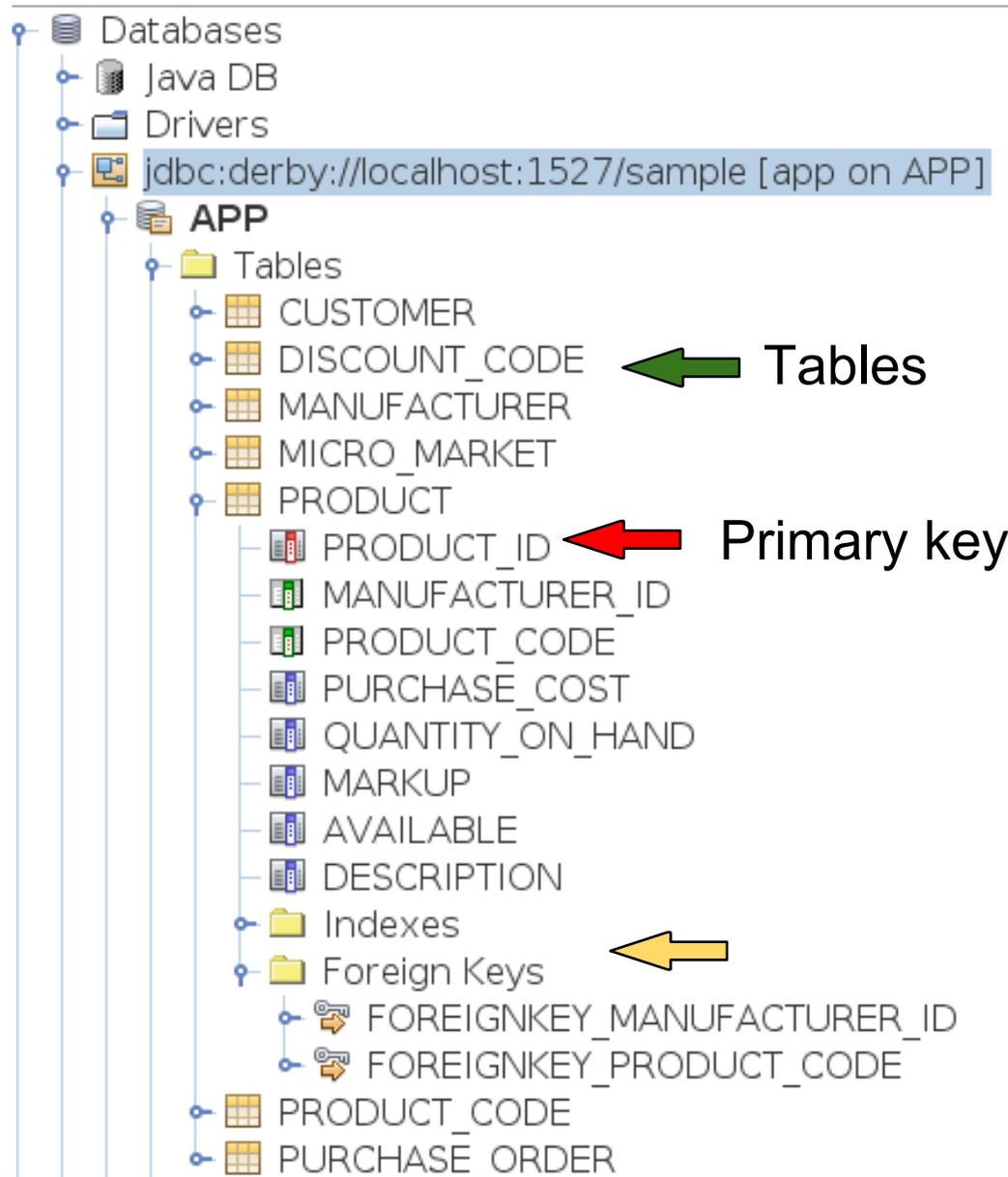
Possible to do the **CRUD** (row) operation directly in NetBeans

All above also possible from within a Java program

Databases stored as files in ~/.netbeans-derby directory

- Possible to delete database by erasing files

# The Sample Database



Derby has a sample database (kind of ordering system)

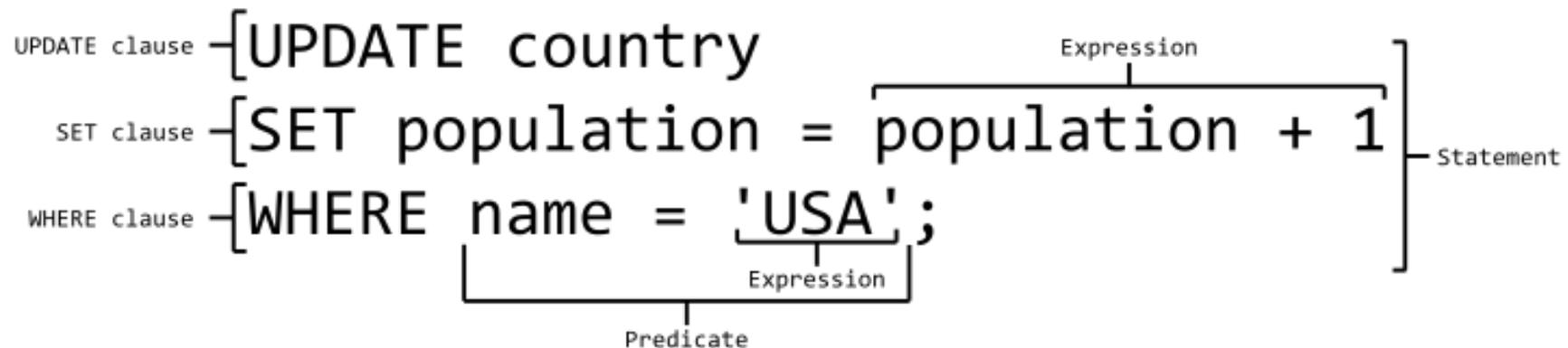
# Structured Query Language, SQL

Assume we have created the database, the tables and the relationships

To manage the data in the database we use SQL

- A declarative language (compare XSLT)
- Written as (looooooong) strings, verbose...
- Standardization efforts but many different dialects
- Normally not possible to move SQL "programs" from one database to another (from different vendors)
- Possible to execute SQL from inside Java programs (as embedded SQL strings in Java)

# SQL Elements



- **Clauses**, part of statements and queries
- **Expressions**, represents scalar values or tables
- **Predicates**, boolean values
- **Statements**, a **write** operation will alter data. Ending in ';' (and more...)
- **Queries**, a **read** statement This is the most important element of **SQL**.
- **White space** ignored in SQL statements and queries.
- Strings uses ' (single quote)
- Keywords normally case insensitive, table and column names varies(!?)

# Create

```
/* Insert a row into table PRODUCT_CODE */
```

```
INSERT INTO  
PRODUCT_CODE (PROD_CODE, DISCOUNT_CODE, DESCRIPTION)  
VALUES ('XX', 'M', 'Junk');
```

Tuple must "match", same components, types must match (here strings), ... if not exception!

Note: We always use a **single numeric value as primary key**. This makes it possible for Derby to automatically assign each row an primary key (ascending sequence). **Never specify primary key** when inserting

# Update and Delete

```
/* Update PROD_CODE from XX to YY */  
UPDATE PRODUCT_CODE  
SET PROD_CODE = 'YY'  
WHERE PROD_CODE = 'XX';
```

```
/* Deleting row with prod code YY */  
DELETE FROM  
PRODUCT_CODE  
WHERE  
PROD_CODE = 'YY';
```

# Queries (Reads)

We say **query**. We query the database to collect information

## Query single table

```
/* Everything from product code table */  
select * from PRODUCT_CODE;
```

```
/* All discount codes (a column) */  
select DISCOUNT_CODE from PRODUCT_CODE;
```

```
/* Product codes for software (0-many rows)*/  
select * from PRODUCT_CODE where DESCRIPTION = 'Software'
```

```
/* All orders delivered by Poney Express with quantity < 10 */  
select * from PURCHASE_ORDER where QUANTITY < 20 AND  
FREIGHT_COMPANY = 'Poney Express'
```

# Join

To make it possible to select data from more tables we **join** the tables

Id	Consult	Phone	...	ProjId
1	Sven	070712345	...	1
2	Olle	...	...	1
3	Fia	...	...	3

Id	Project	Account
1	Nighmare	09-34245-12
2	Kamikaze	...
3	NoSurvivors	...

```
SELECT Id, Consult, Phone, Account
FROM Consult INNER JOIN Project
ON Consult.ProjId = Project.id;
```

Id	Consult	Phone	Account
1	Sven	070712345	09-34245-12
2	Olle	...	09-34245-12

Join result. Matching (by id) rows from both tables. Many types of joins, here INNER

# Join and NULL's

If a Consult has no project a NULL in ProjectId column.

If joining tables NULL values will never match i.e. row not in result

Possible to specify that **all** rows, even if null, (from left or right table) should be included in join result using `LEFT OUTER JOIN` or `RIGHT OUTER JOIN`

- `INNER JOIN` will **not** include consults with no projects, `LEFT OUTER` will (ProjectId value is NULL)

# Orderings

- Possible to order result (sorting done by database, very efficient)

```
SELECT column_name(s)  
FROM table_name  
ORDER BY column_name(s) ASC|DESC
```

# Aggregate Functions

Functions for simplify server side aggregate calculation.

- [Average\(\)](#)
- [Count\(\)](#)
- [Maximum\(\)](#)
- [Median\(\)](#)
- [Minimum\(\)](#)
- [Mode\(\)](#)
- [Sum\(\)](#)
- ...

```
SELECT avg(QUANTITY) from PURCHASE_ORDER;
```

Returns single value (NULL's eliminated)

# Constraints

RDBMS will (can) enforce many constraints (if fail, exception!)

## Default constraints

- Primary key must be unique and not NULL
- Can't add non existing foreign key
- Can't delete a consult if he/she has a project. Must delete project first. Possible: Cascading deletes

## Specifying constraints

- Specifying “**Unique**” on attribute prevents duplicate values in column, necessary for 1:1 relationships
- Specifying attribute must not be NULL, RDBMS will check!

# Transactions

Assume transferring \$1000 from one account to another

A two step operation in the computer world

- Withdraw \$1000 from account A
- Insert \$1000 on account B

But what if a crash in middle!!???

- Money lost...!!

Solution: Transactions

# ACID Property for Transaction

## Atomicity

- A transaction must be seen as a single atomic operation

## Consistency

- A transaction must not violated any constraints, keys, etc.

## Isolation

- Other transactions should have limited access to data involved in the transaction

## Durability

- When transaction finish data shall be permanent

# Commit and Rollback

Inside the transaction data isn't really written to store

Final write operation at transaction end = database **commit**

- After commit data is **persistent**

If transaction fails

- Previous state is restored (i.e. nothing changed) = **rollback**

In programming we sometimes have to handle transactions  
(write code to)

- Java API
- If transaction failed (exception) do a rollback

# Help

W3Schools <http://www.w3schools.com/sql/default.asp>