

# IS5102

# Database Management Systems

## Lecture 1: Introduction

Alexander Konovalov

[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

(with thanks to Susmit Sarkar)

2021



## Course Information – CS Student Handbook

<https://info.cs.st-andrews.ac.uk/student-handbook/modules/IS5102.html>

## Lecturers

- ▶ Dr Alexander Konovalov
- ▶ Dr Michael Young (support)

## Resources – accessible directly or through the Module Management System (MMS)

- ▶ Module directory on **Studres** <http://studres.cs.st-andrews.ac.uk/IS5102>
- ▶ MMS <https://mms.st-andrews.ac.uk/>
  - ▶ IS5102 team in Microsoft Teams
  - ▶ Lecture recordings on **Panopto** (<https://st-andrews.cloud.panopto.eu/>)
  - ▶ Also link to the Studres module directory

## Meeting Logistics

- ▶ 2 lectures per week via Microsoft Teams:
  - ▶ **1pm** on **Mondays** (1 hour)
  - ▶ **1pm** on **Thursdays** (1 hour)
- ▶ 1 exercise session per week via Microsoft Teams:
  - ▶ **1pm** on **Tuesdays** (1 hour)  
Selected material available online by the end of the preceding week
- ▶ 1 on-line drop-in surgery session **each week**
  - ▶ **1pm** on **Wednesdays** via Microsoft Teams
- ▶ 1 in-person drop-in surgery session **fortnightly**
  - ▶ **1pm** on **Mondays** even weeks, details TBC
- ▶ Lecture slides (including this one) & other materials on **Studies**
- ▶ Lectures (including this one) and exercise classes recordings on **Panopto**

## How we will use Teams

- ▶ I will record the meeting and upload the video on Panopto afterwards
- ▶ What will be recorded:
  - ▶ my application window (e.g. slides, database browser, terminal, etc.)
  - ▶ my video and my voice
- ▶ What will **not** be recorded:
  - ▶ your video and your voice
  - ▶ conversations in the meeting chat
- ▶ Teams etiquette and illusion of presence:
  - ▶ please mute your microphone when not speaking
  - ▶ type your questions in the meeting chat
  - ▶ raise hand if you want to talk
  - ▶ **only** if you are comfortable: turn on your camera (it will not be recorded)
- ▶ Not necessary, but may be helpful: earphones and an external screen

## Logistics: Assignments

- ▶ **Three** assignments, submitted via MMS (dates TBA)
  - ▶ (20 %) Data Modelling
  - ▶ (20 %) SQL
  - ▶ (20 %) Group Research
- ▶ One 48-hour exam replacement assessment in December (date TBA), worth 40 %
  - ▶ examinable material:  
lectures, exercises, coursework, relevant chapters of textbooks
  - ▶ we will practise at the end of the semester by looking at past papers

- ▶ **Database Design** by A. Watt & N. Eng
  - ▶ 2nd Ed., e-book (CC BY 4.0), Victoria, B.C. : <https://opentextbc.ca/dbdesign01/>
- ▶ **Database Systems** by T. Connolly & C.E. Begg
  - ▶ 6th Ed., Pearson Education, e-book:  
[https://encore.st-andrews.ac.uk/iii/encore/record/C\\_\\_Rb2379911](https://encore.st-andrews.ac.uk/iii/encore/record/C__Rb2379911)
  - ▶ 6th Ed., Pearson Education, printed:  
[https://encore.st-andrews.ac.uk/iii/encore/record/C\\_\\_Rb2379725](https://encore.st-andrews.ac.uk/iii/encore/record/C__Rb2379725)
  - ▶ 5th Ed., Addison-Wesley, printed:  
[https://encore.st-andrews.ac.uk/iii/encore/record/C\\_\\_Rb1668114](https://encore.st-andrews.ac.uk/iii/encore/record/C__Rb1668114)

► **Database System Concepts** by A. Silberschatz, H. Korth & S. Sudarshan

Web page with supplementary materials, including slides for each chapter:  
<https://www.db-book.com/>

- 7th Ed., McGraw-Hill, printed:  
[https://encore.st-andrews.ac.uk/iii/encore/record/C\\_\\_Rb2696468](https://encore.st-andrews.ac.uk/iii/encore/record/C__Rb2696468)
- 6th Ed., McGraw-Hill, printed:  
[https://encore.st-andrews.ac.uk/iii/encore/record/C\\_\\_Rb1854246](https://encore.st-andrews.ac.uk/iii/encore/record/C__Rb1854246)
- 5th Ed., McGraw-Hill, printed:  
[https://encore.st-andrews.ac.uk/iii/encore/record/C\\_\\_Rb1527546](https://encore.st-andrews.ac.uk/iii/encore/record/C__Rb1527546)
- 4th Ed., McGraw-Hill, printed:  
[https://encore.st-andrews.ac.uk/iii/encore/record/C\\_\\_Rb1394850](https://encore.st-andrews.ac.uk/iii/encore/record/C__Rb1394850)

- ▶ You are assumed to be familiar with the whole [student handbook](#)
- ▶ Read the [Good Academic Practice policy](#)
- ▶ Check that coursework submitted to MMS has been received successfully, and that it is the right piece of coursework
- ▶ Coursework submitted after deadline is subject to automatic penalty
- ▶ Any special circumstances must be documented immediately through the self-certification system, and followed up with coordinator if you are seeking any allowance
- ▶ You must be available for the entire exam period
- ▶ Familiarise yourself with the [School](#) and [University](#) health & safety guidance

## Key points from student handbook:

<https://info.cs.st-andrews.ac.uk/student-handbook/key-points.html>

*The database is now the underlying framework of the information system*

*Connolly and Begg*

- ▶ Collection of inter-related data
- ▶ A logically coherent collection of data with some inherent meaning
- ▶ A representation of some aspect of the real world
- ▶ A fundamental part of most large software systems

- ▶ A collection of related data (database)
- ▶ A set of programs to access, manipulate and present the data
- ▶ Primary goal
  - ▶ A means to store and retrieve data
  - ▶ In a convenient and efficient way

- ▶ Banking: transactions
- ▶ Airlines: reservations, schedules
- ▶ Universities: registration, grades
- ▶ Sales: customers, products, purchases
- ▶ Online retailers: order tracking, customized recommendations
- ▶ Manufacturing: production, inventory, orders, supply chain
- ▶ Human resources: employee records, salaries, tax deductions
- ▶ ...

- ▶ Application program examples
  - ▶ Add new students, instructors and courses
  - ▶ Enrol students on courses, and generate class lists
  - ▶ Assign grades, compute grade point averages and generate transcripts
  - ▶ ...

- ▶ Earlier, database applications were built directly on top of file systems
  - ▶ Supported by OS
  - ▶ Ad-hoc programs
- ▶ Ever tried to sort your Documents folder?

- ▶ Data redundancy and inconsistency
  - ▶ Multiple file formats, duplication of information in different files
- ▶ Difficulty in accessing data
  - ▶ New programs need to be written to carry out new tasks
- ▶ Data isolation
  - ▶ Multiple files and formats
- ▶ Integrity problems
  - ▶ Integrity constraints (e.g., account balance  $\geq £20$ ) become hidden in program code rather than being stated explicitly
  - ▶ Hard to add new constraints or change existing ones
- ▶ Security problems
  - ▶ Hard to provide access to some, not all, data
- ▶ Lack of concurrency
  - ▶ When an application opens a file, it is locked, and nobody else can access it at the same time

Database systems provide a centralised, well-controlled data repository

- ▶ Can deal with **enormous** amounts of persistent data
- ▶ And do so **efficiently** and **flexibly**
- ▶ Can design with **security** in mind
- ▶ Can deal with **multiple** concurrent accesses

## Database management system (DBMS)

A complex set of applications that enables users to create and maintain a database.

Main features of modern DBMS:

- ▶ Provide facilities to store, retrieve and update data
- ▶ Provide a user-accessible catalog which gives descriptions of stored data items and is accessible to users
- ▶ Provide mechanisms to support authorisation
- ▶ Transaction Support – ensures that either all the updates corresponding to a given transaction are made or that none of them are made
- ▶ Concurrency control – allows shared access

- ▶ Reading
  - ▶ Chapters 1-3, Database design
  - ▶ Chapter 1, Database System Concepts
- ▶ Practice
  - ▶ To prepare for the exercise session tomorrow, install **DB Browser for SQLite**:  
<https://sqlitebrowser.org/>

# IS5102

# Database Management Systems

## Lecture 2: E-R Diagrams

Alexander Konovalov

[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

(with thanks to Susmit Sarkar)

2021



## Physical Level

Describe how a data record (e.g. student) is stored

## Logical Level

Describe the data and relationships between data

## View Level

Describe selected aspects (**views**) of the data

Many views of same data possible

Also important to **hide** data in views (think security)

Analogous to types and values in programming languages

- ▶ Schema

- ▶ The overall design of the database
- ▶ **Physical** schema – database design at the physical
- ▶ **Logical** schema – database design at the logical level level
- ▶ Changes are infrequent

- ▶ Instance

- ▶ Content of the database at a particular point in time
- ▶ Changes may be frequent

A collection of conceptual tools for describing

- ▶ Data
- ▶ Data relationships
- ▶ Data semantics
- ▶ Data constraints

**Physical** data model – geared towards implementation

**Logical/conceptual** data model – more abstract

Examples:

- ▶ **Entity-Relationship (E-R)** data model (mainly for database design)
- ▶ **Relational** model (lower level, later)
- ▶ Object-based data models (Object-oriented and Object-relational)
- ▶ Semistructured data model (XML)
- ▶ Other older models:
  - ▶ Network model
  - ▶ Hierarchical model

A **database** can be modeled as:

- ▶ a collection of **entities**,
- ▶ **relationship** among entities.

- ▶ An **entity** is an object that exists and is distinguishable from other objects.
  - ▶ Example: specific person, company, event, plant
- ▶ Entities have **attributes**
  - ▶ Example: people have names and addresses
- ▶ An **entity set** is a set of entities of the same type that share the same properties.
  - ▶ Example: set of all persons, companies, trees, holidays

- ▶ An entity is represented by a set of **attributes**, that is descriptive properties possessed by all members of an entity set.

Example:

```
instructor = (ID, name, street, city, salary)  
course = (course_id, title, credits)
```

- ▶ **Domain** – the set of permitted values for each attribute

- ▶ Attribute types:
  - ▶ **Simple** and **composite** attributes
    - ▶ Example of a composite attribute: `address`
  - ▶ **Single-valued** and **multivalued** attributes
    - ▶ Example of a multivalued attribute: `phone_numbers`
  - ▶ **Derived** attributes
    - ▶ Can be computed from other attributes
    - ▶ Example: `age`, if given `date_of_birth`

- ▶ A **relationship** is an association among several entities

Example:

44553 (Student X) advisor 22222 (Instructor Y)  
student entity → relationship set → instructor entity

- ▶ A **relationship set** is a mathematical relation among  $n \geq 2$  entities, each taken from corresponding entity sets

$$\{ (e_1, e_2, \dots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n \}$$

where  $(e_1, e_2, \dots, e_n)$  is a relationship

Example:  $(44553, 22222) \in \text{advisor}$

- ▶ An attribute can also be property of a relationship set
- ▶ For instance, the `advisor` relationship set may have the attribute `date` which tracks when the student started being associated with the advisor

- ▶ **Binary** relationship
  - ▶ involves two entity sets (i.e. has **degree** two).
  - ▶ most relationship sets in a database system are binary.
- ▶ Relationships between more than two entity sets are less common, but also occur
  - ▶ Example: students work on research projects under the guidance of an instructor relationship `proj_guide` is a ternary relationship between instructor, student, and project

## Participation Constraints

determined by the **minimum** number of times entity participates in relationship

- if zero, then **partial** participation
- if more than zero, then **total** participation

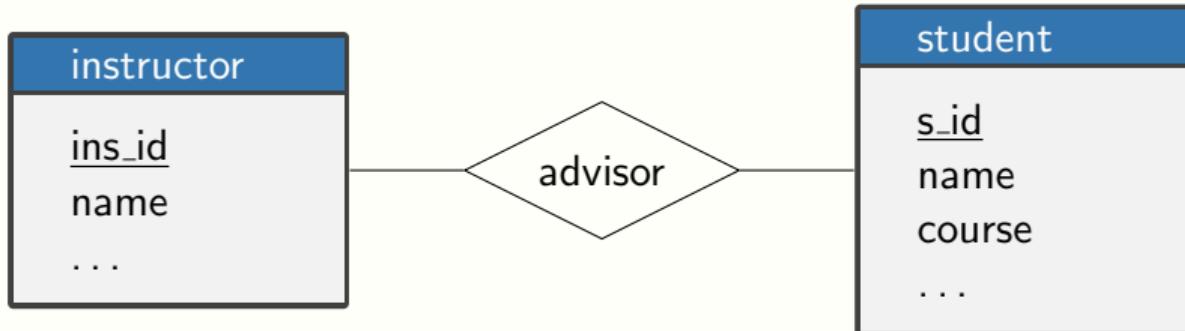
## Cardinality Constraints

**maximum** number of times entity participates in relationship

- ▶ Express the **number** of entities to which another entity can be **associated** via a relationship set
- ▶ Most useful in describing binary relationship sets
- ▶ For a binary relationship set the mapping cardinality must be one of the following types:
  - ▶ **One to one**
  - ▶ **One to many**
  - ▶ **Many to one**
  - ▶ **Many to many**

- ▶ A **super key** of an entity set is a set of one or more attributes whose values uniquely determine each entity.
- ▶ A **candidate key** of an entity set is a minimal super key
  - ▶ ID is candidate key of instructor
  - ▶ course\_id is candidate key of course
- ▶ Although several candidate keys may exist, one of the candidate keys is selected to be the **primary key**.

- ▶ The **combination of primary keys** of the participating entity sets forms a super key of a relationship set.  
(s\_id, i\_id) is the super key of advisor
- ▶ Must consider the **mapping cardinality** of the relationship set when deciding what are the candidate keys



- ▶ Rectangles represent **entity sets**.
- ▶ Diamonds represent **relationship sets**.
- ▶ **Attributes** listed inside entity rectangle
- ▶ Underline indicates **primary key attributes**

We express cardinality constraints by drawing:

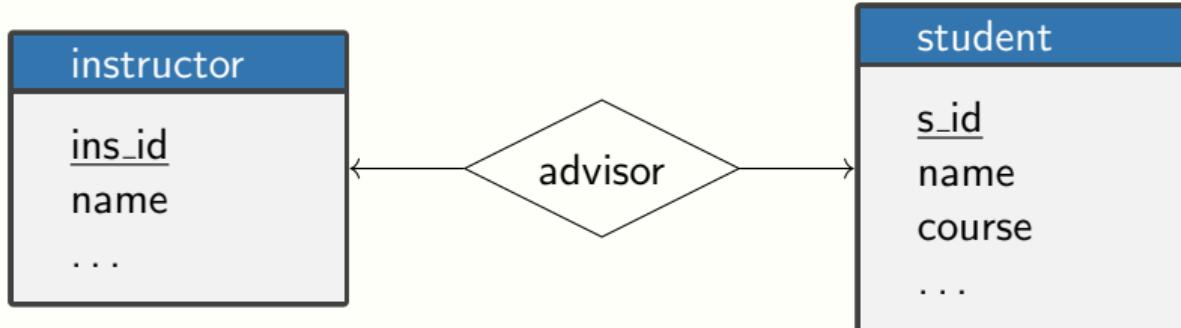
either a directed line ( $\rightarrow$ ), signifying “**one**”

or an undirected line ( $\_$ ), signifying “**many**”

between the relationship set and the entity set

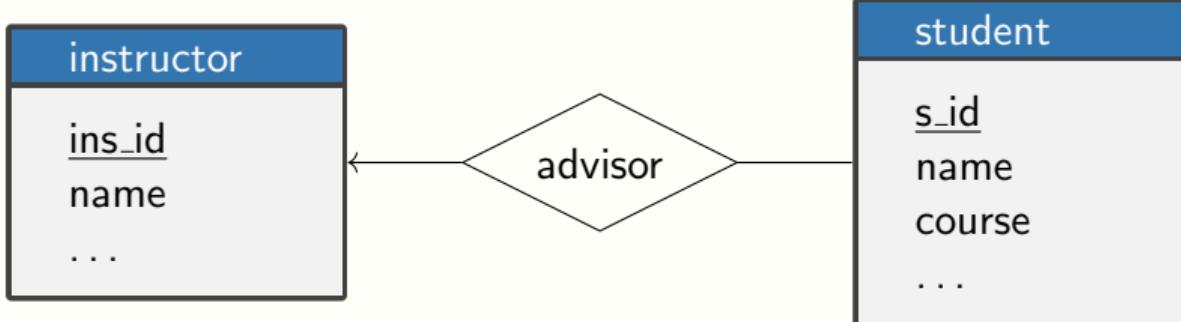
**One-to-one** relationship between an instructor and a student

- ▶ an instructor is associated with at most one student via advisor
- ▶ and a student is associated with at most one instructor via advisor



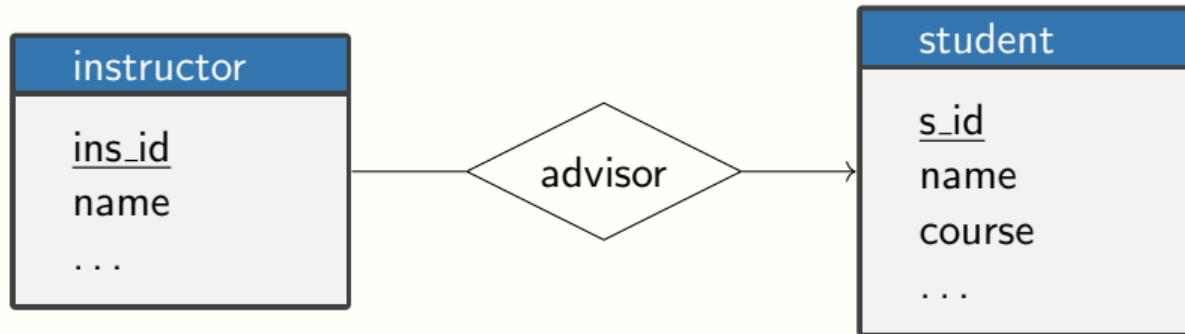
**One-to-many** relationship between an instructor and a student

- ▶ an instructor is associated with several (including 0) students via advisor
- ▶ a student is associated with at most one instructor via advisor



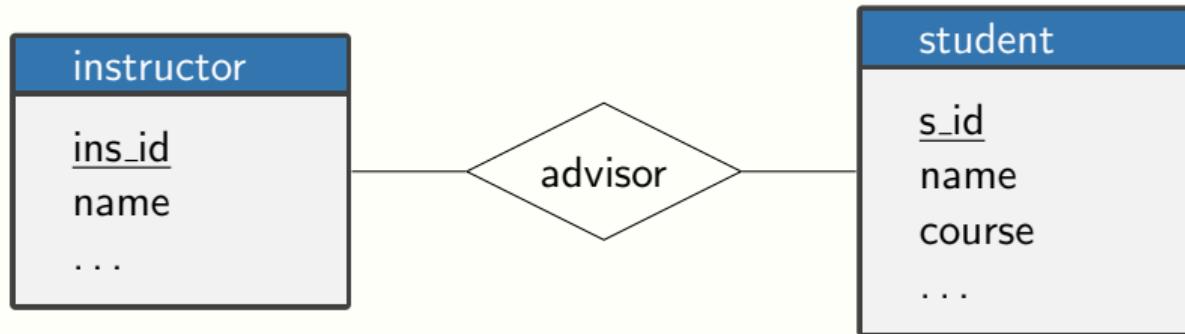
## Many-to-one relationship between an instructor and a student

- ▶ an instructor is associated with at most one student via advisor,
- ▶ and a student is associated with several (including 0) instructors via advisor



## Many-to-many relationship between an instructor and a student

- ▶ An instructor is associated with several (possibly 0) students via advisor
- ▶ A student is associated with several (possibly 0) instructors via advisor



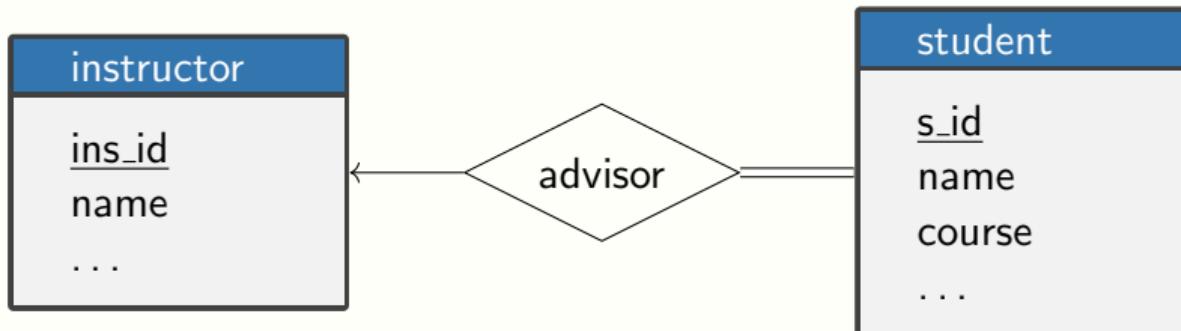
# Participation of an Entity Set in a Relationship Set

- ▶ **Total participation** (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set

Example: participation of student in advisor is total  
every student must have an associated advisor

- ▶ **Partial participation**: some entities may not participate in any relationship in the relationship set

Example: participation of instructor in advisor is partial



- ▶ Data models:
  - ▶ Chapters 4-5, Database Design
  - ▶ Chapter 1, Database System Concepts
- ▶ E-R models:
  - ▶ Chapter 8, Database Design
  - ▶ Chapter 7, Database System Concepts

# IS5102

# Database Management Systems

## Lecture 3: E-R Diagrams

Alexander Konovalov

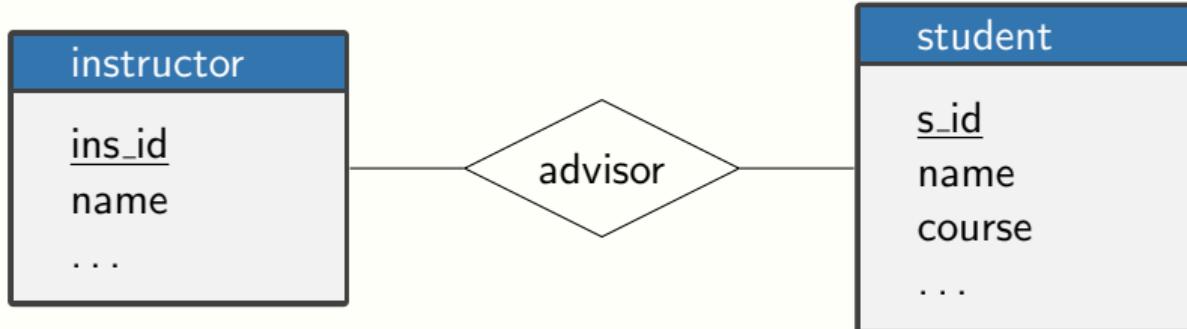
[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

(with thanks to Susmit Sarkar)

2021



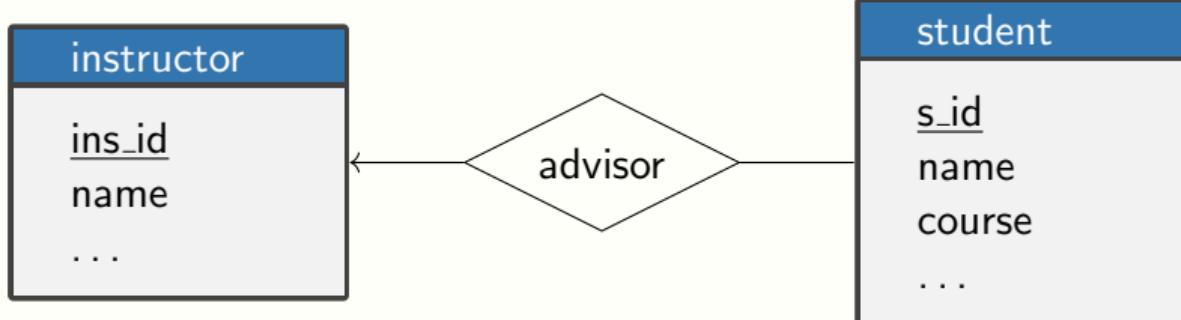
- ▶ Entity Relationship Modeling
- ▶ Entities, Relationships, Attributes
- ▶ Drawing E-R models



- ▶ Rectangles represent entity sets.
- ▶ Diamonds represent relationship sets.
- ▶ Attributes listed inside entity rectangle
- ▶ Underline indicates primary key attributes

one-to-many relationship between an instructor and a student

- ▶ an instructor is associated with several (including 0) students via advisor
- ▶ a student is associated with at most one instructor via advisor

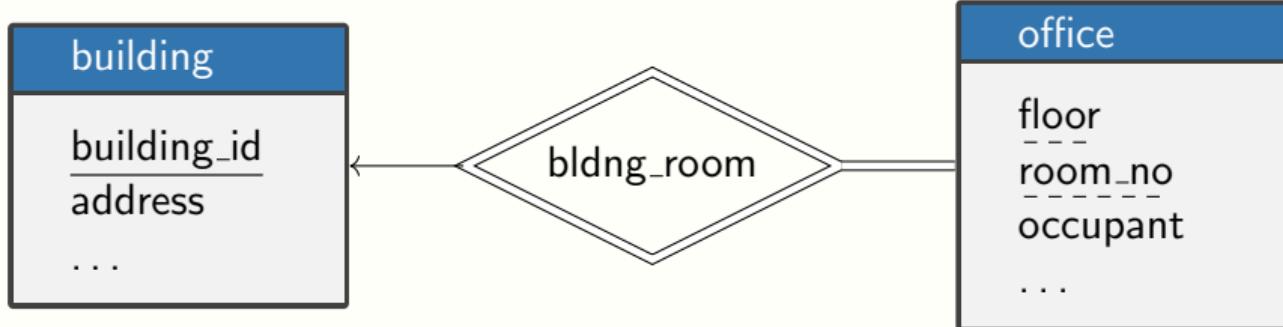


- ▶ Consolidating E-R Models
- ▶ Refinements to Weak Entities
- ▶ Specialisation, Generalisation
- ▶ Some common pitfalls

- ▶ An entity set with a primary key is called a **strong entity set**
- ▶ An entity set without a primary key is called a **weak entity set**
- ▶ For example, consider entity set of offices in company's buildings: room 0.11 might exist in Jack Cole Building and in the School of Mathematics and Statistics
- ▶ The existence of a weak entity set depends on the existence of a **identifying entity set** (also called an **owner entity set**)
  - ▶ It must relate to the identifying entity set via a **total, many-to-one** relationship set from the identifying to the weak entity set
  - ▶ This relationship set is called an **identifying relationship** and is depicted using a **double diamond**

- ▶ In our example, the identifying identity set is building
- ▶ The **discriminator** (or **partial key**) of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entity set that depend on one particular strong entity
- ▶ The **primary key** of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent,  
**plus** the weak entity set's discriminator

- ▶ We put the **identifying relationship** of a weak entity in a double diamond.
- ▶ We underline the **discriminator** of a weak entity set with a dashed line.
- ▶ Primary key for office — (building\_id, floor, room\_no).
- ▶ Note the double line (total participation) and the arrow (one-to-many).



- ▶ Note: the primary key of the strong entity set is not explicitly stored with the weak entity set, since it is implicit in the identifying relationship
- ▶ If `building_id` were explicitly stored, office could be made a strong entity . . .  
 . . . but then the relationship between office and building would be duplicated by an implicit relationship defined by the attribute `building_id` common to building and office

## Use of entity sets vs multivalued composite attributes

- ▶ A weak entity set could be more appropriate as an attribute if it participates only in its identifying relationship, and has a few attributes
- ▶ A phone type (office, home, mobile) and number could be a multivalued attribute for a person
- ▶ Use of phone as an entity allows extra information about phone numbers, other relationships e.g. between phone numbers and offices, etc.

## Use of entity sets vs relationship sets

- ▶ Possible guideline is to designate a relationship set to describe an **action** that occurs between entities

- ▶ There are some relationships that are naturally non-binary  
Example: proj\_guide
- ▶ Sometimes an  $n$ -ary relationship set shows more clearly that several entities participate in a single relationship.
- ▶ In general, any non-binary relationship can be represented using binary relationships by creating an artificial entity set.
- ▶ Placement of relationship attributes e.g., attribute date as attribute of advisor or as attribute of student

# Converting Non-Binary Relationships to Binary Form

In general, any non-binary relationship can be represented using binary relationships by creating an artificial entity set.

- ▶ Replace  $R$  between entity sets  $A$ ,  $B$  and  $C$  by an entity set  $E$ , and three relationship sets:
  1.  $RA$ , relating  $E$  and  $A$
  2.  $RB$ , relating  $E$  and  $B$
  3.  $RC$ , relating  $E$  and  $C$
- ▶ Create a special identifying attribute for  $E$
- ▶ Add any attributes of  $R$  to  $E$
- ▶ For each relationship  $(a_i, b_i, c_i) \in R$ , create
  1. a new entity  $e_i$  in the entity set  $E$
  2. add  $(e_i, a_i)$  to  $RA$
  3. add  $(e_i, b_i)$  to  $RB$
  4. add  $(e_i, c_i)$  to  $RC$

- ▶ Chapter 8, Database Design
- ▶ Chapter 7, Database System Concepts

# IS5102

# Database Management Systems

## Lecture 4: E-R Diagrams

Alexander Konovalov

[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

(with thanks to Susmit Sarkar)

2021



- ▶ Consolidating E-R Models
- ▶ Refinements to Weak Entities
- ▶ Specialisation, Generalisation
- ▶ Some common pitfalls

Designate subgroups within an entity set

Lower-level entities **inherit** all attributes and relationships of higher-level entities

Additionally, **lower-level** entity sets that have (additional) attributes or participate in relationships not applicable to **higher-level** entity set

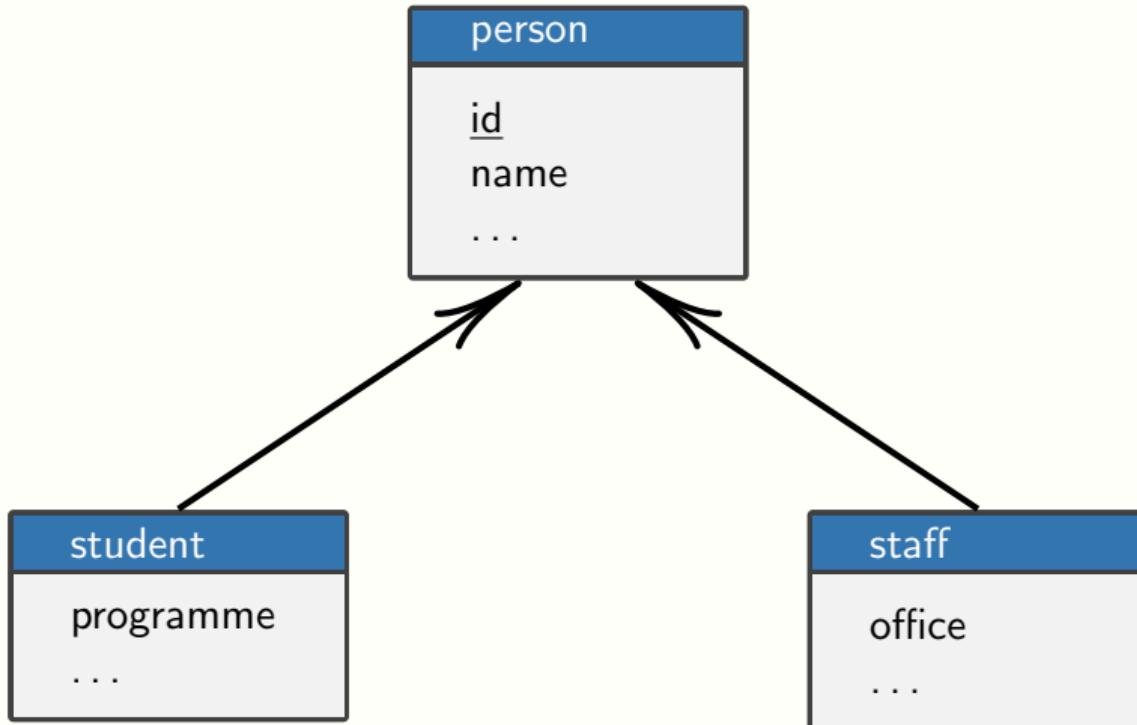
**lower-level** and **higher-level** entity sets are also called **subclass** and **superclass**

A lower-level entity set may have several higher-level entity sets (**multiple inheritance**)

Combine a number of entity sets sharing features into higher-level entity set

**Specialisation** and **Generalisation** are inverses of each other

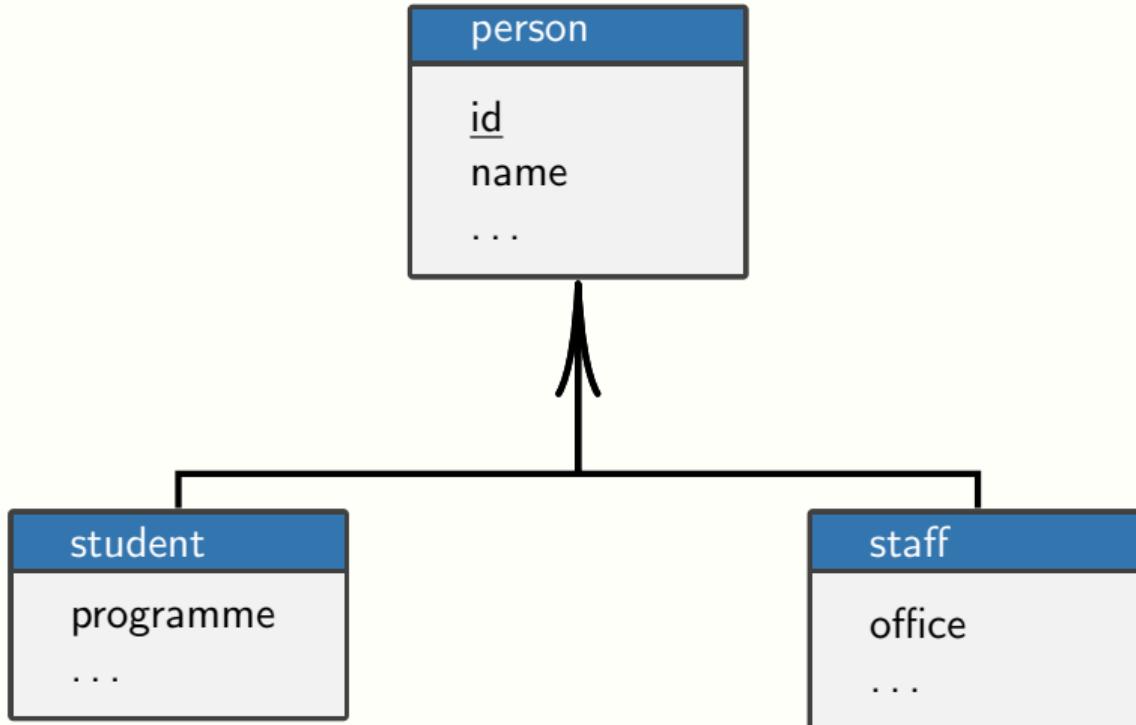
# Specialisation/Generalisation Example



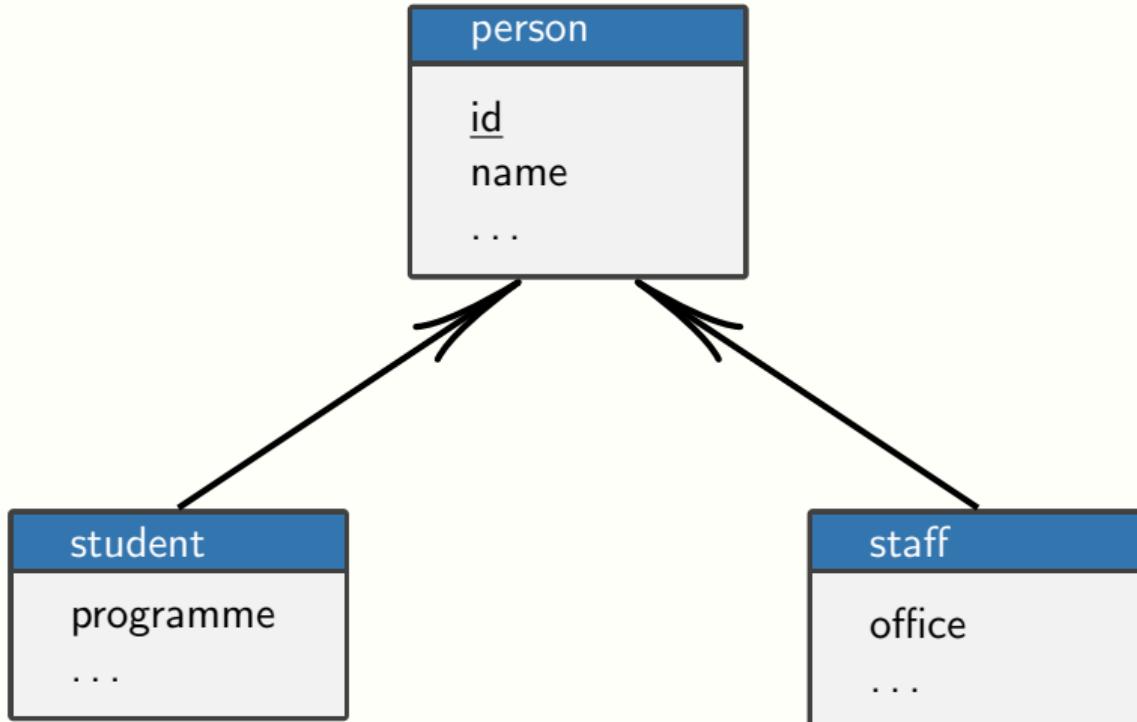
**Overlapping** constraint: An entity can belong to more than one lower-level entity set

**Disjoint** constraint: An entity can belong to only one lower-level entity set

# Disjoint Specialisation/Generalisation Example



# Overlapping Specialisation/Generalisation Example



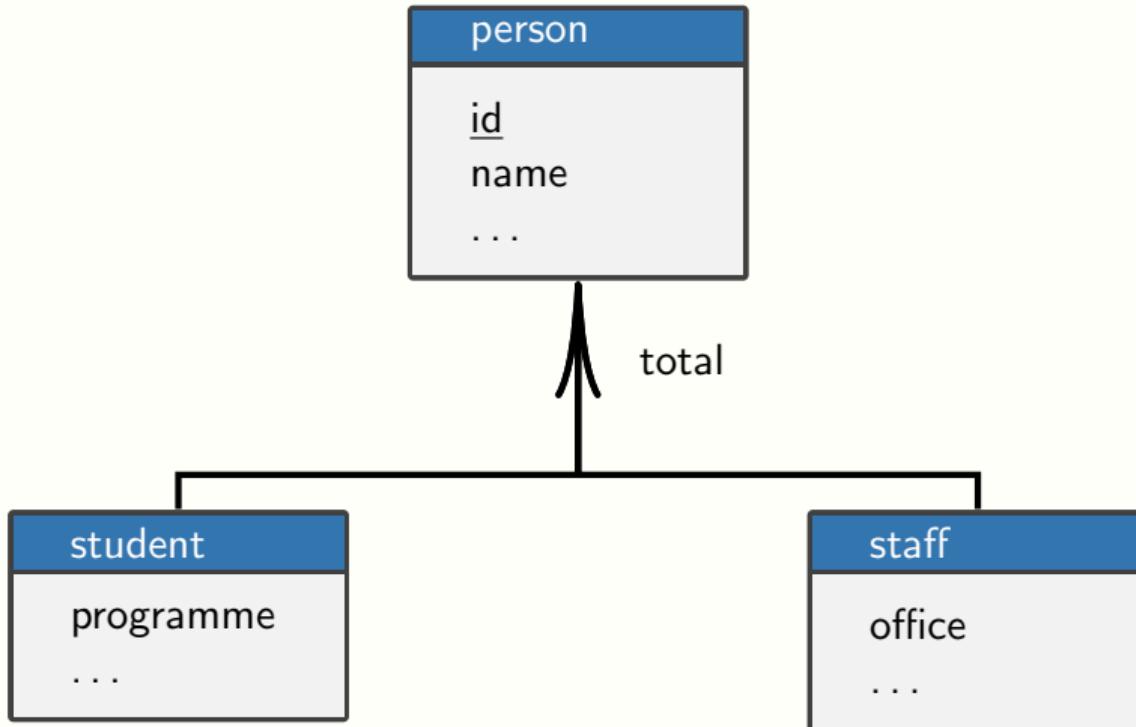
Note that there can be combinations (two specialisations are disjoint, but another one can overlap with both of those)

**Completeness**: specifying whether or not an entity in higher-level entity set must belong to at least one of lower-level entity sets

**Partial specialisation/generalisation**: an entity need not belong to any lower-level entity set

**Total specialisation/generalisation**: an entity must belong to at least one lower-level entity set

# Total Disjoint Specialisation/Generalisation Example



Another type of constraint is related to determining which entities can be members of a given lower-level entity set

**Condition-defined**: membership depends on an explicitly stated condition

for example, **attribute-defined**

**User-defined**: a user makes a decision to assign an entity to a lower-level entity set

- ▶ Database Planning
- ▶ Requirement Collection and Analysis
- ▶ Database Design
- ▶ Database Selection
- ▶ Application Design
- ▶ Implementation
- ▶ Testing
- ▶ Management

## Requirements Collection and Analysis

- ▶ **What** data is to be used;
- ▶ **How** that data is to be used

Often helpful to think of types of users

Each will have their own requirements on data

Can be used to create user views

Integration of user views subsequently

Typically, the objects (or nouns) are entities

... or sometimes attributes!

What are the natural groups?

Many different kinds of relationships (actions, subject-object, etc)

**How** do we identify our entities? (Keys)

Naturally occurring identifiers?

**When** does an entity (identify!) exist?

**Scenario:** An online shop

## Questions

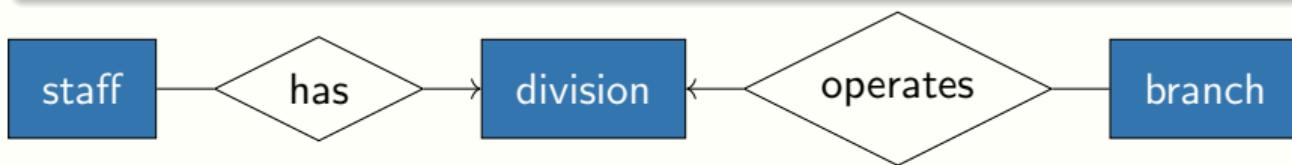
Who are the users of the database?

What example queries will they run?

What data needs to be recorded to answer those queries?

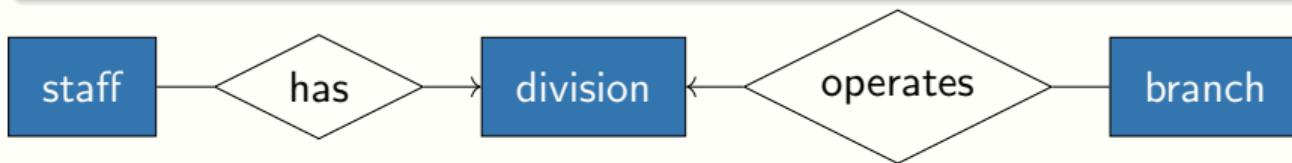
## Fan Trap

When a model represents a relationship, but the pathway between entity occurrences is ambiguous.

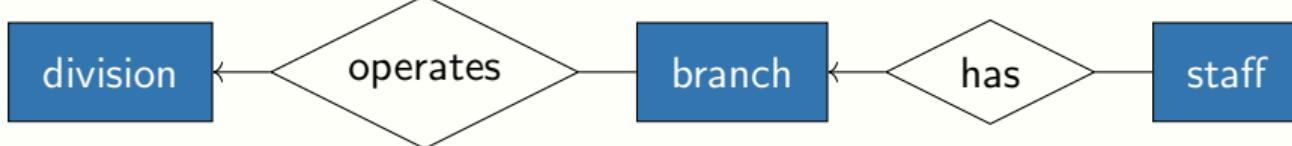


## Fan Trap

When a model represents a relationship, but the pathway between entity occurrences is ambiguous.

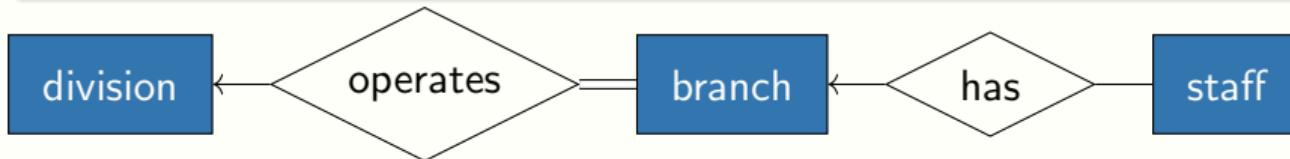


## Solution:



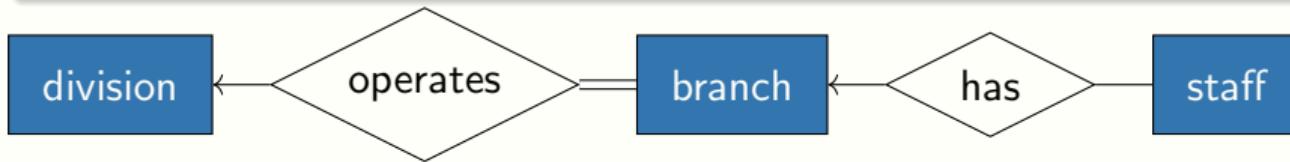
## Chasm Trap

When a model suggests a relationship exists, but there is no pathway between certain entity occurrences.

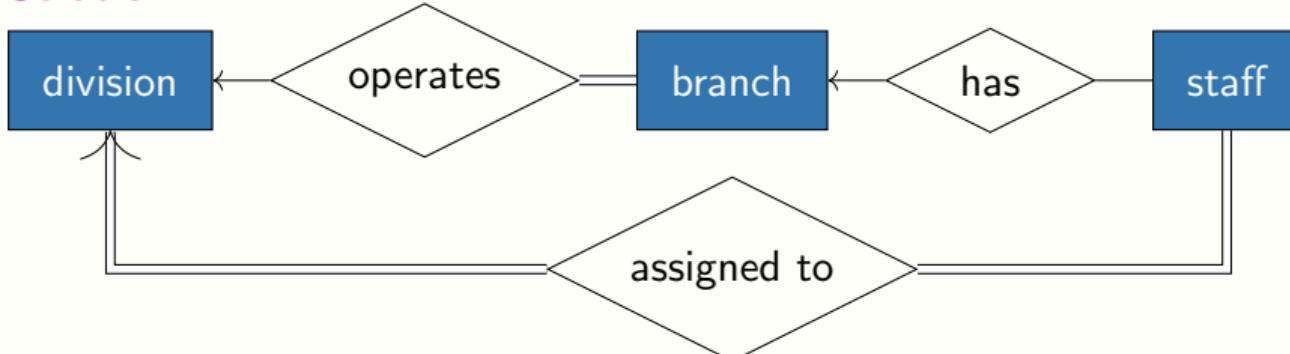


## Chasm Trap

When a model suggests a relationship exists, but there is no pathway between certain entity occurrences.



## Solution:



- ▶ Reading
  - ▶ Chapter 8, Database Design, 2nd Ed. Watt & Eng
  - ▶ Chapter 7, Database System Concepts, 6th Ed. Silberschatz, Korth & Sudarshan
  - ▶ Chapters 11-12: Database Systems, 6th Ed. Connolly & Begg
- ▶ Next Time: Relational Calculus and Algebra

# IS5102

# Database Management Systems

## Lecture 5: Relational Model

Alexander Konovalov

[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

(with thanks to Susmit Sarkar)

2021



- ▶ Data models
- ▶ Entity–Relationship data models
- ▶ E-R diagrams and graphical notation

- ▶ Lower level (logical) data models
- ▶ Relational data models and translations
- ▶ Formal query analysis

- ▶ A relational database consists of
  - ▶ a collection of tables each with a unique name
    - ▶ similar to a spreadsheet document containing a number of worksheets
- ▶ Each row in a table represents a relationship among a set of values
  - ▶ similar to mathematical notion of a **relation**
    - ▶ from which the model gets its name
- ▶ Each column in a table represents a distinct kind of value

- ▶ Table with a number of rows and columns
- ▶ Terms
  - ▶ **Relation** is the same as table
  - ▶ **Tuple** is the same as row or record
  - ▶ **Attribute** is the same as column or field

- ▶ No two rows with same values in all column positions
- ▶ All the values within a column in a relation have the same type
  - ▶ simple types only
  - ▶ no complex values such as sets or other rows

- ▶ Attributes (fields or columns)
  - ▶ a field is used to store an individual item of data
- ▶ Key Attributes
  - ▶ **Candidate key** – candidate key is an attribute or combination of attributes that uniquely identifies rows in a table
  - ▶ **Primary key** – there can be more than one candidate from which the primary key is chosen
  - ▶ **Foreign Key** – an attribute or combination of attributes that match attribute(s) in another table.
- ▶ Record (or tuple)
  - ▶ a record is a group of related attributes
  - ▶ identifiable by its primary key (or any candidate key)

- ▶ Data Types:
  - ▶ the fields are set to accept a particular data type
  - ▶ helps check for the wrong type of data being entered
  - ▶ stores data as efficiently as possible
  - ▶ sorts data correctly
- ▶ Examples:
  - ▶ Integer number
  - ▶ Floating point number
  - ▶ String of given maximum length
  - ▶ Date and time
  - ▶ ...

- ▶ Proposed by E. F. Codd in 1970
- ▶ A means of storing information in tables called relations
  - ▶ Each table has multiple columns each with a unique name
- ▶ Use of data structures to access information quickly
  - ▶ E.g. indexes can help perform operations quickly
- ▶ High level means of expressing queries
  - ▶ Powerful means of expressing queries using relational algebra
  - ▶ Expressions can be optimised for faster evaluation

The Relational Model is based on the mathematical concept of a relation

- ▶ **Relation** – table
  - ▶ represented by a table with columns and rows
- ▶ **Tuple** – row
  - ▶ order of tuples (rows) is not important
- ▶ **Attributes** – named column headers
  - ▶ All values in a column have the same type
  - ▶ Attributes can have only simple types

- ▶ **Domain** of an attribute
  - ▶ Set of permitted values for that attribute
- ▶ **Degree** of a relation
  - ▶ Number of attributes it contains
- ▶ **Cardinality** of a relation
  - ▶ Number of tuples it contains
  - ▶ Changes as tuples are added or deleted

# Example of a Relation (instructor)

Attributes (or columns)

ID	Name	Department	Salary
10101	Soros	Finance	78500
10210	Einstein	Physics	56000
15675	Mozart	Music	62800
28675	Turing	Computer Science	49750
31822	Curie	Physics	67000
33821	Johnson	Mathematics	81000
45893	Franklin	Biology	48250
45910	Ramanujan	Mathematics	51900
57264	Porter	Management	92000
67450	Fleming	Medicine	64520

Tuples (or rows)

- ▶ The set of allowed values for each attribute is called the domain of the attribute
- ▶ Attribute values are (normally) required to be atomic; that is, indivisible
- ▶ The special value **null** is a member of every domain
- ▶ The null value causes complications in the definition of many operations

Attribute	Meaning	Domain Definition
ID	The set of all possible instructor IDs	character: size 5, range 00000 – 99999
name	The set of all possible names	character : size 50
dept_name	The set of all possible department names within the university	character: size 30
salary	Possible values of the instructor salaries	currency: 6 digits, range 10000 - 150000

- ▶  $A_1, A_2, \dots, A_n$  are attributes
- ▶  $R = (A_1, A_2, \dots, A_n)$  is a relation schema

Example:

```
instructor = (ID, name, dept_name, salary)
```

- ▶ Formally, given sets  $D_1, D_2, \dots, D_n$   
a relation  $r$  is a subset of  $D_1 \times D_2 \times \dots \times D_n$

Thus, a relation is a set of n-tuples  $(a_1, a_2, \dots, a_n)$   
where each  $a_i \in D_i$

- ▶ The current values (relation instance) of a relation are specified by a table
- ▶ An element  $t$  of  $r$  is a **tuple**, represented by a row in a table

Order of tuples is irrelevant (tuples may be stored in an arbitrary order)

Example: instructor relation with unordered tuples

ID	Name	Department	Salary
10101	Soros	Finance	78500
31822	Curie	Physics	67000
45910	Ramanujan	Mathematics	51900
15675	Mozart	Music	62800
33821	Johnson	Mathematics	81000
67450	Fleming	Medicine	64520
10210	Einstein	Physics	56000
45893	Franklin	Biology	48250
57264	Porter	Management	92000
28675	Turing	Computer Science	49750

- ▶ A database often consists of multiple relations
- ▶ Information about an enterprise is broken up into parts:
  - instructor
  - student
  - advisor
- ▶ Bad design:

```
univ (instructor_id, name, dept_name, salary, student_id, ...)
```

Results in

- ▶ repetition of information (e.g., two students have the same instructor)
- ▶ the need for null values (e.g., represent an student with no advisor)
- ▶ Normalization theory (later) deals with how to design “good” relational schemas

- ▶ Chapter 7, Database Design
- ▶ Chapter 2, Database System Concepts
- ▶ Chapter 4 & 5.1, Database Systems

# IS5102

# Database Management Systems

## Lecture 6: Relational Model

Alexander Konovalov

[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

(with thanks to Susmit Sarkar)

2021



- ▶ Lower level (logical) data models
- ▶ Relational data models
- ▶ Translations
- ▶ Formal query analysis

- ▶ Let  $A_1, A_2, \dots, A_n$  are attributes
- ▶ Let  $R = (A_1, A_2, \dots, A_n)$  is a relation schema
- ▶ Let  $K \subseteq \{A_1, A_2, \dots, A_n\}$
- ▶  $K$  is a **superkey** of  $R$  if values for  $K$  are sufficient to identify a unique tuple of each possible relation  $r(R)$ 
  - Example:  $\{\text{ID}\}$  and  $\{\text{ID}, \text{ name}\}$  are both superkeys of `instructor`
- ▶ Superkey  $K$  is a **candidate key** if  $K$  is minimal
  - Example:  $\{\text{ID}\}$  is a candidate key for `instructor`
- ▶ One of the candidate keys is selected to be the **primary key**
  - ▶ which one?

- ▶ Need to define a primary key for each table
- ▶ Sometimes a suitable set of attributes may already be present in data model
  - ▶ e.g. consider the relation Branch with attributes {branch\_name, assets, branch\_city}
- ▶ Sometimes they will not ...
  - ▶ e.g. relation Person with {name, age}
  - ▶ In such cases, need to invent one or more artificial attributes which are designed to be unique
    - ▶ examples are NI number, passport number, driving licence number, NHS number, clubcard number, etc.

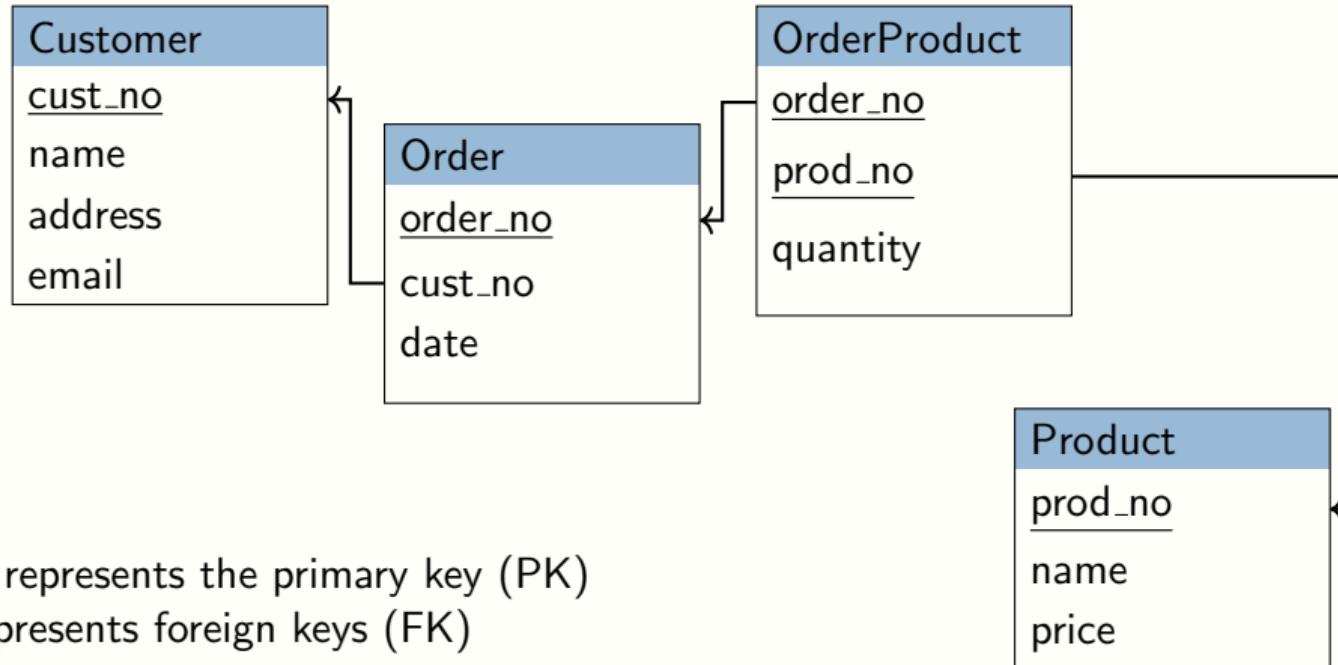
**Foreign key** constraint: Value in one relation must appear in another

A **foreign key** constraint from attribute  $A$  of relation  $R_1$  to the primary key  $B$  of relation  $R_2$ :

for every tuple in  $R_1$ , the value of  $A$  must also be the value of some tuple in  $R_2$ :

$$\forall v \in R_1 \quad \exists w \in R_2 : v.A = w.B$$

- ▶  $A$  – **foreign key** from  $R_1$  referencing  $R_2$
- ▶  $R_1$  – **referencing** relation
- ▶  $R_2$  – **referenced** relation



Underline represents the primary key (PK)

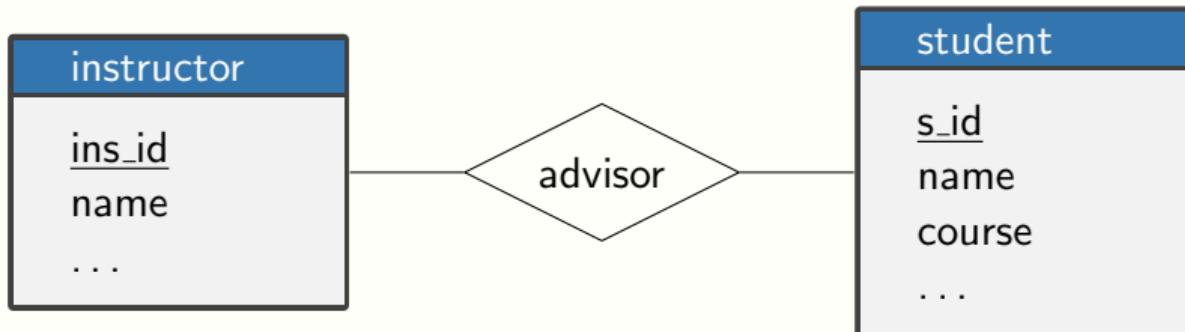
Arrows represents foreign keys (FK)

**Exercise:** how can the foreign key constraints be violated?

- ▶ Entity sets and relationship sets can be expressed uniformly as **relation schemas** that represent the contents of the database.
- ▶ A database which conforms to an E-R diagram can be represented by a **collection of schemas**.
- ▶ For each entity set and relationship set there is a unique schema that is assigned the name of the corresponding entity set or relationship set.
- ▶ Each schema has a number of columns (generally corresponding to **attributes**), which have unique names.

- ▶ A strong entity set reduces to a schema with the same attributes  
`student(ID, name, credits)`
- ▶ A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set  
`(building_id, floor, room_no,occupant)`

- ▶ A many-to-many relationship set is represented as a schema with attributes for the primary keys of the two participating entity sets, and any descriptive attributes of the relationship set.
- ▶ Example: schema for relationship set advisor  
 $\text{advisor} = (\text{s\_id}, \text{ins\_id})$



Many-to-one and one-to-many relationship sets that are **total** on the many-side:

Can be represented by adding an **extra attribute** to the “many” side, containing the primary key of the “one” side

Example: Instead of creating a schema for relationship set `inst_dept`:

add an attribute `dept_name` to the schema arising from entity set `instructor`

Composite attributes are **flattened out** by creating a separate attribute for each component attribute

Example: given entity set `instructor` with composite attribute `name` with component attributes `first_name`, `middle_initial` and `last_name`

The schema corresponding to the entity set `instructor` has the attributes `first_name`, `middle_initial` and `last_name`

A multivalued attribute M of an entity E is represented by a separate schema EM

- ▶ Schema EM has attributes corresponding to the primary key of E and an attribute corresponding to multivalued attribute M
- ▶ Example: Multivalued attribute `phone_number` of `instructor` is represented by a schema:

```
inst_phone = (ID, phone_number)
```

- ▶ Each value of the multivalued attribute maps to a separate tuple of the relation on schema EM

- ▶ Abstract query language
- ▶ Defines a set of operations on relations
- ▶ Operations take a relation(s) as input and produce a relation as output
- ▶ They form the basis for the SQL language

Relational algebra operators include:

Selection	$\sigma$	(unary)
Projection	$\Pi$	(unary)
Cartesian Product	$\times$	(binary)
Natural Join	$\bowtie$	(binary)
Union	$\cup$	(binary)
Intersection	$\cap$	(binary)
Set difference	$-$	(binary)

Relation  $r$ :

A	B	C	D
$\alpha$	$\alpha$	1	7
$\alpha$	$\beta$	5	7
$\beta$	$\beta$	12	3
$\beta$	$\beta$	23	10

$\sigma_{A = B \text{ and } D > 5}(r)$ :

A	B	C	D
$\alpha$	$\alpha$	1	7
$\beta$	$\beta$	23	10

Relation  $r$ :

A	B	C	D
$\alpha$	$\alpha$	1	7
$\alpha$	$\beta$	5	7
$\beta$	$\beta$	12	3
$\beta$	$\beta$	23	10

$\Pi_{A,D}(r)$ :

A	D
$\alpha$	7
$\beta$	3
$\beta$	10

# Set operations: union, intersection, difference

Relation  $r$

A	B
$\alpha$	1
$\alpha$	2
$\beta$	1

Relation  $s$

A	B
$\alpha$	2
$\beta$	3

$r \cup s$

A	B
$\alpha$	1
$\alpha$	2
$\beta$	1
$\beta$	3

$r \cap s$

A	B
$\alpha$	2

$r - s$

A	B
$\alpha$	1
$\beta$	1

Relation  $r$ 

A	B
$\alpha$	1
$\beta$	2

Relation  $s$ 

C	D	E
$\alpha$	10	a
$\alpha$	20	a
$\beta$	10	b

 $r \times s$ :

A	B	C	D	E
$\alpha$	1	$\alpha$	10	a
$\alpha$	1	$\alpha$	20	a
$\alpha$	1	$\beta$	10	b
$\beta$	2	$\alpha$	10	a
$\beta$	2	$\alpha$	20	a
$\beta$	2	$\beta$	10	b

Relation  $r$ 

A	B	C	D
$\alpha$	1	$\alpha$	a
$\beta$	2	$\gamma$	a
$\gamma$	4	$\beta$	b
$\alpha$	1	$\gamma$	a
$\delta$	2	$\beta$	b

Relation  $s$ 

B	D	E
1	a	$\alpha$
3	a	$\beta$
1	a	$\gamma$
2	b	$\delta$
3	b	$\epsilon$

 $r \bowtie s$ :

A	B	C	D	E
$\alpha$	1	$\alpha$	a	$\alpha$
$\alpha$	1	$\alpha$	a	$\gamma$
$\alpha$	1	$\gamma$	a	$\alpha$
$\alpha$	1	$\gamma$	a	$\gamma$
$\delta$	2	$\beta$	b	$\delta$

Symbol	Name	Result
$\sigma$	Selection	Returns rows of the input relation that satisfy the predicate
$\Pi$	Projection	Returns the specified attributes from all rows of the input relation. Duplicate rows removed
$\times$	Cartesian product	Output all combinations of rows from the two input relations
$\bowtie$	Natural Join	Output all combinations of rows from the two input relations that are equal on their common attribute names
$\cup$	Union	Output all rows that are in the two similarly structured input relations or in both. Duplicate rows are eliminated
$\cap$	Intersection	Output all rows that are in both the two similarly structured input relations
$-$	Difference	Output all rows that are the first input relation but are not in the second

- ▶ Consolidation
  - ▶ Chapter 7, Database Design, 2nd Ed., Watt and Eng
  - ▶ Chapter 2, Database System Concepts, 6th Ed., Silberschatz, Korth and Sudarshan
  - ▶ Chapter 4 & 5.1, Database Systems, 6th Ed., Connolly, Begg
- ▶ Next few weeks: SQL
  - ▶ Chapters 15-16, Database Design, 2nd Ed., Watt and Eng
  - ▶ Chapters 3-5, Database System Concepts, 6th Ed., Silberschatz, Korth and Sudarshan
  - ▶ Chapters 6-8, Database Systems, 6th Ed., Connolly, Begg

oin

# IS5102

# Database Management Systems

## Lecture 7: Introduction to SQL

Alexander Konovalov

[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

(with thanks to Susmit Sarkar)

2021



- ▶ Data Modeling
- ▶ ER models and Relational Models
- ▶ Relational algebra

- ▶ Overview of the SQL Query Language
- ▶ Data Definition
- ▶ Basic Query Structure

- ▶ (early 1970s) IBM Sequel Language as part of System R project
- ▶ (early 1980s) Evolved and renamed to Structured Query Language (SQL)
- ▶ (1986) ANSI and ISO published an SQL standard
  - ▶ SQL-86; SQL-89; SQL-92
  - ▶ SQL:1999
  - ▶ SQL:2003; SQL:2006; SQL:2008; SQL:2011; SQL:2016

Not all features supported in all systems

SQLite documentation: see <https://www.sqlite.org/docs.html>  
in particular, “About”, “Distinctive features” and “Quirks” under “Overview Documents”

You have most likely used SQLite today: <https://www.sqlite.org/famous.html>

- ▶ Data Definition Language (**DDL**)  
define, modify and delete relations
- ▶ Data Manipulation Language (**DML**)  
insert, modify and delete tuples; perform **queries**
- ▶ Integrity Constraints  
enforced by forbidding updates violating them
- ▶ Data Control Language (**DCL**)  
Transactions, authorisation, ...

Allow the specification of **information about relations**, e.g.:

- ▶ Schema of each relation
- ▶ Domain of values for each attribute
- ▶ Integrity constraints

Also allow additional information, e.g.:

- ▶ Indices for each relation
- ▶ Security and authorization information for relation
- ▶ Physical storage structure of relation on disk

- ▶ **CHAR(n)**: fixed length character string, length  $n$
- ▶ **VARCHAR(n)**: variable length character string, max length  $n$
- ▶ **INT** (or **INTEGER**): integer (machine-dependent size)
- ▶ **SMALLINT**: “small” integer (machine-dependent size)
- ▶ **NUMERIC(p,d)**: fixed-point number with user-specified precision  
( $p$  digits of which  $d$  are to the right of the decimal point)
- ▶ **REAL** and **DOUBLE**: floating point numbers (machine-dependent precision)
- ▶ **FLOAT(n)**: floating point number (at least  $n$  digit precision)

## Temporal data

- ▶ **DATE**: 4 year digits (yyyy) + 2 month digits (mm) + 2 day digits (dd).  
Format 'yyyy-mm-dd' e.g. 2012-10-04
- ▶ **TIME**: 2 hour digits (hh) + 2 minute digits (mm) + 2 second digits (ss) in 24 hour notation: 'hh:mm:ss'
- ▶ **TIMESTAMP**: Combination of the above
- ▶ More on this later...

## Practical implementations:

- ▶ “rigidly typed”:  
MySQL et al.: [http://www.w3schools.com/sql/sql\\_datatypes.asp](http://www.w3schools.com/sql/sql_datatypes.asp)
- ▶ “flexibly typed”:  
SQLite: <https://www.sqlite.org/datatype3.html>

An SQL relation is defined using the `CREATE TABLE` command:

```
CREATE TABLE r (
    A1 D1,
    A2 D2,
    ...,
    An Dn,
    (integrity-constraint1),
    ...,
    (integrity-constraintk)
);
```

- ▶  $r$  is the name of the relation
- ▶  $A_i$  is an attribute name in the schema of relation  $r$
- ▶  $D_i$  is the data type of values in domain of attribute  $A_i$

```
CREATE TABLE department (
    dept_id      CHAR(5),
    dept_name    VARCHAR(20),
    building     VARCHAR(15),
    budget       NUMERIC(12,2)
);
```

We are following the **SQL Style Guide** by Simon Holywell:

<https://www.sqlstyle.guide>

SQLite documentation entry for **CREATE TABLE**:

[https://www.sqlite.org/lang\\_createtable.html](https://www.sqlite.org/lang_createtable.html)

- ▶ Add tuples to a relation

```
INSERT INTO department
VALUES ('CS', 'Computer Science', 'Jack Cole', 1500000),
       ('MATH', 'Mathematics and Statistics', 'Maths', 900000),
       ('PHYS', 'Physics and Astronomy', 'Physics', 1500000);
```

- ▶ Remove all tuples from a relation

```
DELETE FROM department;
```

- ▶ Remove a relation from a database

```
DROP TABLE department;
```

Obligatory XKCD: <https://xkcd.com/327/>

Queries are performed using the **SELECT** command, which lists the attributes requested as a result of the query.

A typical SQL query has the form:

```
SELECT A1, A2, ..., An  
FROM r1, r2, ..., rm  
WHERE P;
```

In this query:

- ▶  $A_i$  is an attribute
- ▶  $r_i$  is a relation
- ▶  $P$  is a predicate

The result of an SQL query is a relation.

An asterisk \* is a wildcard to denote all attributes in a relation. It can be used in

```
SELECT * FROM department;
```

to inspect the content of the department table.

```
CREATE TABLE department (
    dept_id      CHAR(5),
    dept_name    VARCHAR(20),
    building     VARCHAR(15),
    budget       NUMERIC(12,2)
);
```

```
INSERT INTO department
VALUES ('CS', 'Computer Science', 'Jack Cole', 1500000),
       ('MATH', 'Mathematics and Statistics', 'Maths', 900000),
       ('PHYS', 'Physics and Astronomy', 'Physics', 1500000);
```

```
SELECT * FROM department;
```

- ▶ Put the code from the example of an SQL script into the text file called `deps.sql`
- ▶ Open the terminal in the directory containing the file `deps.sql`, and execute it with:
  - ▶ `sqlite3 --init deps.sql` to connect to a transient in-memory database, or
  - ▶ `sqlite3 uni.db --init deps.sql` to connect to a database `uni.db`  
(new database will be created, if it does not exist)
- ▶ This will run commands from the `deps.sql` script and keep SQLite shell open for further exploration. You can enter, for example:
  - ▶ `.tables` to check that it outputs department
  - ▶ `.schema` to show corresponding CREATE commands
  - ▶ `.dump` to dump the database in an SQL text format
- ▶ Then enter `.quit` (or `.q`) to exit SQLite shell
- ▶ Add lines

```
.mode column  
.headers on
```

on top of the script to change output formatting, and try this procedure again

- ▶ You have now established a repeatable and reproducible workflow for further SQL practice
  - ▶ It recreates tables each time, so you can easily modify their schemas and test changes
  - ▶ You can also rerun the script entering `.read depts.sql` in the SQLite shell
  - ▶ Repeat this process several times. What goes wrong?
  - ▶ Fix this by adding the line `DROP table department;` on top of the script
    - ▶ In this case, the next restart will cause `DROP` to report “no such table” errors
    - ▶ That clearly happens because the database is empty - can be ignored in this case

This is the equivalent of the previous workflow for the DB Browser for SQLite

- ▶ Put the code from the example of an SQL script into the text file called `depts.sql` and open it from the “Execute SQL” tab (alternatively, copy and paste it into the code editor in the “Execute SQL” tab)
  - ▶ Customise DB Browser: use monospace font (e.g. Monaco) for indentation, close some tabs
- ▶ Use “New Database” button to create the new database. Select “Cancel” in the form for editing table definition (later use “Open Database” to connect to an existing database)
- ▶ Run SQL code using “Execute all” button or the keyboard shortcut
- ▶ Inspect “Database Structure” and “Browse Data” tabs

- ▶ Run the same SQL code again. What goes wrong?
- ▶ Fix this by adding the line `DROP table department;` on top of the SQL script
- ▶ You can now modify SQL code and run it again
  - ▶ Resetting tables is more cumbersome though:
    - ▶ Change to a new database (or delete an old one and pick the same name for the new one)
    - ▶ Comment out `DROP` command(s) for the first run of the script

- ▶ SQLite
  - ▶ <https://sqlite.org/docs.html>
- ▶ Command Line Shell For SQLite
  - ▶ <https://sqlite.org/cli.html>
- ▶ DB Browser for SQLite
  - ▶ <https://github.com/sqlitebrowser/sqlitebrowser/wiki>

# IS5102

# Database Management Systems

## Lecture 8: Introduction to SQL

Alexander Konovalov

[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

(with thanks to Susmit Sarkar)

2021



- ▶ Overview of the SQL Query Language
- ▶ Data Definition
- ▶ Basic Query Structure

- ▶ NOT NULL
- ▶ PRIMARY KEY (A<sub>1</sub>, ..., A<sub>n</sub>)  
primary key declaration on an attribute automatically ensures not null
- ▶ FOREIGN KEY (A<sub>m</sub>, ..., A<sub>n</sub>) REFERENCES r

Example: updated declaration of department from the previous lecture:

```
CREATE TABLE department (
    dept_id    CHAR(5),
    dept_name  VARCHAR(20) NOT NULL,
    building   VARCHAR(15),
    budget     NUMERIC(12,2),
    PRIMARY KEY (dept_id)
);
```

## Integrity Constraints in CREATE TABLE

Example: Declare instr\_id as the primary key for instructor and dept\_id a foreign key

```
CREATE TABLE instructor (
    instr_id    CHAR (5),
    instr_name  VARCHAR(20) NOT NULL,
    dept_id    VARCHAR(5),
    salary      NUMERIC (8,2),
    PRIMARY KEY (instr_id),
    FOREIGN KEY (dept_id) REFERENCES department);
```

```
INSERT INTO instructor
VALUES ('45797', 'Bob', 'CS', 28000),
       ('12355', 'Petro', 'MATH', 32000),
       ('23456', 'Alice', 'PHYS', 29500),
       ('45638', 'Sana', 'PHYS', 31500);
```

**Warning 1:** In SQLite, we have to use

```
PRAGMA foreign_keys = TRUE;
```

to enforce foreign key constraints

**Warning 2:** In MariaDB, have to write

```
FOREIGN KEY (dept_id) REFERENCES department(dept_id)
```

```
CREATE TABLE student (
    stud_id    CHAR(5),
    name        VARCHAR(20) NOT NULL,
    dept_id    VARCHAR(20),
    tot_cred   NUMERIC(3,0),
    PRIMARY KEY (stud_id),
    FOREIGN KEY (dept_id) REFERENCES department);
```

```
INSERT INTO student
VALUES ('64545', 'Abdul', 'MATH', 180),
       ('79879', 'Tom', 'CS', 90),
       ('89675', 'Eilidh', 'PHYS', 120),
       ('96544', 'Sarah', 'PHYS', 180);
```

```
CREATE TABLE course (
    course_id  VARCHAR(8),
    title       VARCHAR(50),
    dept_id    VARCHAR(20),
    credits     NUMERIC(2,0),
    PRIMARY KEY (course_id),
    FOREIGN KEY (dept_id) REFERENCES department);
```

```
INSERT INTO course
VALUES ('CS1234', 'Python', 'CS', 15),
       ('CS2234', 'Haskell', 'CS', 15),
       ('MT4665', 'Algebra', 'MATH', 15),
       ('PH3457', 'Photonics', 'PHYS', 30);
```

Altering a table: used to add or delete attributes from an existing relation

Syntax:

```
ALTER TABLE table r ADD A D
```

- ▶ where A is the name of the attribute to be added to relation r and D is the domain of A
- ▶ All tuples in the relation are assigned **NULL** as the value for the new attribute

```
ALTER TABLE r DROP A
```

- ▶ where A is the name of an attribute of relation r
- ▶ Warning: Dropping of attributes not supported by many systems

- ▶ The **SELECT** clause list the attributes desired in the result of a query
  - ▶ corresponds to the projection operation of the relational algebra
- ▶ Example: find the names of all instructors:

```
SELECT instr_name FROM instructor
```

- ▶ NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - ▶ E.g., **Select** = **SELECT** = **select**
  - ▶ We follow the SQL Style Guide: <https://www.sqlstyle.guide/>

- ▶ SQL allows duplicates in relations as well as in query results.
- ▶ To force the elimination of duplicates, insert the keyword **DISTINCT** after **SELECT**
- ▶ Find the department names of all instructors, and remove duplicates

```
SELECT DISTINCT dept_id  
  FROM instructor;
```

- ▶ The keyword **all** specifies that duplicates not be removed.

```
SELECT ALL dept_id  
  FROM instructor;
```

- ▶ An asterisk in the `SELECT` clause denotes “all attributes”

```
SELECT *
  FROM instructor;
```

- ▶ The `SELECT` clause can contain arithmetic expressions involving the operations `+`, `-`, `*`, and `/`, and operating on constants or attributes of tuples.
- ▶ The query:

```
SELECT instr_id, instr_name, salary/12
  FROM instructor;
```

would return a relation that is the same as the `instructor` relation, except that the value of the attribute `salary` is divided by 12.

- ▶ The WHERE clause specifies conditions that the result must satisfy
  - ▶ Corresponds to the selection predicate of the relational algebra.
- ▶ To find all instructors in Physics dept with salary > 30000

```
SELECT instr_name
  FROM instructor
 WHERE dept_id = 'PHYS' AND salary > 30000;
```

- ▶ Comparison results can be combined using the logical connectives AND, OR, and NOT.
- ▶ Comparisons can be applied to results of arithmetic expressions.

- ▶ The **FROM** clause lists the relations involved in the query
  - ▶ Corresponds to the Cartesian product operation of the relational algebra.
- ▶ Find the Cartesian product **instructor**  $\times$  **teaches**

```
SELECT *
  FROM instructor, teaches;
```

- ▶ generates every possible **instructor** – **teaches** pair, with all attributes from both relations.
- ▶ Cartesian product not very useful directly, but useful combined with **WHERE** clause condition (selection operation in relational algebra).

- ▶ For all instructors who have taught courses, find their names and the course ID of the courses they taught.

```
SELECT instr_name, course_id
  FROM instructor, teaches
 WHERE instructor.instr_id = teaches.instr_id;
```

`NATURAL JOIN` matches tuples with the same values for all common attributes, and retains only one copy of each common column

```
SELECT * FROM instructor NATURAL JOIN teaches;
```

Compare this with

```
SELECT *
  FROM instructor, teaches
 WHERE instructor.instr_id = teaches.instr_id;
```

Also compare

```
SELECT instr_name, course_id
  FROM instructor NATURAL JOIN teaches;
```

with

```
SELECT instr_name, course_id
  FROM instructor, teaches
 WHERE instructor.instr_id = teaches.instr_id;
```

for listing the names of instructors along with the course ID of the courses that they taught.

{Are these equivalent? }

- ▶ The SQL allows renaming relations and attributes using the **AS** clause:

```
old-name AS new-name
```

E.g.,

```
SELECT instr_id, instr_name, salary/12 AS monthly_salary
  FROM instructor;
```

- ▶ Find the names of all instructors who have a higher salary than some instructor in Physics:

```
SELECT DISTINCT T.instr_name
  FROM instructor AS T, instructor AS S
 WHERE T.salary > S.salary
   AND S.dept_id = 'PHYS';
```

- ▶ Keyword **AS** is optional and may be omitted

```
instructor AS T = instructor T
```

- ▶ Add a new tuple to course

```
INSERT INTO course
VALUES ('IS5102', 'DBMS', 'CS', 15);
```

- ▶ or equivalently

```
INSERT INTO course (course_id, title, dept_id, credits)
VALUES ('IS5102', 'DBMS', 'CS', 15);
```

- ▶ or with a different order of attributes

```
INSERT INTO course (course_id, title, credits, dept_id)
VALUES ('IS5040', 'HCI', 15, 'CS');
```

- ▶ Add new tuples to student with tot\_creds set to **NULL** in two ways:

```
INSERT INTO student
VALUES ('65467', 'Emma', 'CS', NULL);
INSERT INTO student (stud_id, name, dept_id)
VALUES ('83456', 'Nick', 'MATH' );
```

- ▶ Chapter 3, Database System Concepts, Silberschatz, Korth and Sudarshan
- ▶ Chapter 6, Database Systems, Connolly and Begg
- ▶ Chapters 15-16, Database Design, Watt and Eng
- ▶ Useful sites
  - ▶ <http://www.w3schools.com/sql/>
  - ▶ [http://sqlzoo.net/wiki/Main\\_Page](http://sqlzoo.net/wiki/Main_Page)

# IS5102

# Database Management Systems

## Lecture 9: Introduction to SQL

Alexander Konovalov

[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

(with thanks to Susmit Sarkar)

2021



## Basic SQL

- ▶ Defining Table Structure
- ▶ Querying Tables

- ▶ Modifying Data
- ▶ Orderings and Aggregates
- ▶ Integrity Constraints
- ▶ Views and Authorisation

- ▶ Insertion of new tuples into a given relation
- ▶ Deletion of tuples from a given relation
- ▶ Updating of values in some tuples in a given relation

- ▶ Delete all instructors

```
DELETE FROM instructor;
```

- ▶ Delete all instructors from the Finance department

```
DELETE FROM instructor
WHERE dept_id = 'FIN';
```

- ▶ Delete all tuples in the instructor relation for those instructors associated with a department located in the Bute building.

```
DELETE FROM instructor
WHERE dept_id IN
  (SELECT dept_id
   FROM department
   WHERE building = 'Bute');
```

- ▶ Add a new tuple to course

```
INSERT INTO course
VALUES ('IS5102', 'DBMS', 'CS', 15);
```

- ▶ or equivalently

```
INSERT INTO course (course_id, title, dept_id, credits)
VALUES ('IS5102', 'DBMS', 'CS', 15);
```

- ▶ Add a new tuple to student with tot\_creds set to `null`

```
INSERT INTO student
VALUES ('65467', 'Emma', 'CS', NULL);
```

- ▶ Add all instructors to the student relation with tot\_creds set to 0

```
INSERT INTO student
SELECT instr_id, instr_name, dept_id, 0
FROM instructor;
```

- ▶ The **SELECT FROM WHERE** statement is evaluated fully before any of its results are inserted into the relation. Otherwise queries like

```
INSERT INTO table1 SELECT * FROM table1;
```

would cause problems

**Exercise:** revert the previous update of student, relying on the ID of students and instructors being non-overlapping

Increase salaries of instructors whose salary is over £30,000 by 3%, and all others receive a 5% raise

Choice 1: Write two update statements:

```
UPDATE instructor
  SET salary = salary * 1.03
 WHERE salary > 30000;
```

```
UPDATE instructor
  SET salary = salary * 1.05
 WHERE salary <= 30000;
```

Choice 2: Same query as before but with case statement

```
UPDATE instructor
  SET salary =
    CASE
      WHEN salary <= 30000 THEN
        salary * 1.05
      ELSE
        salary * 1.03
    END;
```

NOTE: In MariaDB use

```
END CASE
```

instead of

```
END
```

SQL includes a string-matching operator for comparisons on character strings. The operator like uses patterns that are described using two special characters:

- ▶ percent (%): The % character matches any substring
- ▶ underscore (\_): The \_ character matches any character

- ▶ Find the names of all instructors whose name **includes the substring** “PH”.

```
SELECT instr_name
      FROM instructor
     WHERE dept_id LIKE '%PH%';
```

- ▶ **Exercise:** check if the match is case-sensitive
- ▶ Match the string “100%”

```
LIKE '100\%' escape '\'
```

in that above we use backslash (\) as the escape character

- ▶ Pattern matching examples:
  - ▶ '`Intro%`' matches any string beginning with "Intro".
  - ▶ '`%Comp%`' matches any string containing "Comp" as a substring.
  - ▶ '`___`' matches any string of exactly three characters.
  - ▶ '`___%`' matches any string of at least three characters.
- ▶ SQL supports a variety of string operations such as
  - ▶ concatenation
  - ▶ converting from upper to lower case (and vice versa)
  - ▶ finding string length, extracting substrings, etc.
- ▶ But standard syntax is not always supported!

- ▶ List in alphabetic order the names of all instructors

```
SELECT DISTINCT instr_name
  FROM instructor
 ORDER BY instr_name;
```

- ▶ We may specify **DESC** for descending order or **ASC** for ascending order, for each attribute; ascending order is the default

```
SELECT DISTINCT instr_name, salary
  FROM instructor
 ORDER BY salary DESC;
```

- ▶ Can sort on multiple attributes

```
ORDER BY dept_name, name
```

- ▶ It is possible for tuples to have a null value, denoted by **NULL**, for some of their attributes
- ▶ **NULL** signifies an unknown value or that a value does not exist.
- ▶ The result of any arithmetic expression involving **NULL** is **NULL**  
Example:  $5 + \text{NULL}$  returns **NULL**
- ▶ Any comparison with **NULL** returns unknown  
Example:  $5 < \text{NULL}$  or  $\text{NULL} <> \text{NULL}$  or  $\text{NULL} = \text{NULL}$
- ▶ The predicate **IS NULL** can be used to check for null values  
Example: Find all students with `tot_cred` is null.

```
SELECT name
      FROM student
     WHERE tot_cred IS NULL;
```

These functions operate on the multi-set of values of a column of a relation, and return a value

AVG: average value

MIN: minimum value

MAX: maximum value

SUM: sum of values

COUNT: number of values

- ▶ Find the average salary of instructors in the Physics department

```
SELECT AVG (salary)
      FROM instructor
     WHERE dept_id = 'PHYS';
```

- ▶ Find the total number of instructors who teach at least one course

```
SELECT COUNT (DISTINCT instr_id)
      FROM teaches;
```

- ▶ Find the number of tuples in the course relation

```
SELECT COUNT (*)
      FROM course;
```

Find the average salary of instructors in each department

```
SELECT dept_id, AVG (salary) AS "Average Salary"  
  FROM instructor  
 GROUP BY dept_id;
```

**Exercise:** use NATURAL JOIN to output department name instead of ID

Attributes in select clause outside of aggregate functions must appear in group by list

The following does not produce meaningful result

```
SELECT dept_id, instr_name, AVG (salary)
      FROM instructor
 GROUP BY dept_id;
```

Find the names and average salaries of all departments whose average salary is greater than 30000

```
SELECT dept_id, AVG (salary)
  FROM instructor
 GROUP BY dept_id
 HAVING AVG (salary) > 30000;
```

**Note:** predicates in the **HAVING** clause are applied **after** the formation of groups whereas predicates in the **WHERE** clause are applied **before** forming groups

Find the total sum of all annual salaries

```
SELECT SUM (salary)  
FROM instructor;
```

Above statement ignores **NULL** amounts

Result is **NULL** if there is no non-null amount

**Exercise:** find the total amount of salaries to be paid in a month

- ▶ **[DBSC]** Chapters 4-5, Database System Concepts, Silberschatz, Korth and Sudarshan
- ▶ **[DBS]** Chapter 7, Database Systems, Connolly and Begg
- ▶ **[DBD]** Chapters 15-16, Database Design, Watt and Eng
- ▶ Useful sites
  - ▶ <http://www.w3schools.com/sql/>
  - ▶ [http://sqlzoo.net/wiki/Main\\_Page](http://sqlzoo.net/wiki/Main_Page)

# IS5102

# Database Management Systems

## Lecture 10: Intermediate SQL

Alexander Konovalov

[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

(with thanks to Susmit Sarkar)

2021



- ▶ Modifying Data
- ▶ Orderings and Aggregates
- ▶ Set Operations
- ▶ Integrity Constraints
- ▶ Views

- ▶ Set operations **UNION**, **INTERSECT**, and **EXCEPT**
  - ▶ Each of the above operations automatically eliminates duplicates
- ▶ To retain all duplicates use the corresponding multi-set versions
  - UNION ALL**,
  - INTERSECT ALL**
  - EXCEPT ALL**
- ▶ Suppose a tuple occurs  $m$  times in  $r$  and  $n$  times in  $s$ , then, it occurs:
  - ▶  $m + n$  times in  $r \text{ UNION ALL } s$
  - ▶  $\min(m, n)$  times in  $r \text{ INTERSECT ALL } s$
  - ▶  $\max(0, m - n)$  times in  $r \text{ EXCEPT ALL } s$

**Remark:** SQLite only supports **UNION ALL**, but not the other two multi-set versions

- ▶ Find courses that ran in Semester 2 of 2019 **or** in Semester 1 of 2020

```
SELECT course_id
  FROM course_runs
 WHERE semester = 2
   AND year = 2019
UNION
SELECT course_id
  FROM course_runs
 WHERE semester = 1
   AND year = 2020;
```

- ▶ Find courses that ran in Semester 2 of 2019 **and** in Semester 2 of 2020

```
SELECT course_id
  FROM course_runs
 WHERE semester = 2
   AND year = 2019
```

INTERSECT

```
SELECT course_id
  FROM course_runs
 WHERE semester = 2
   AND year = 2020;
```

- ▶ Find courses that ran in Semester 1 of 2019 **but not** in Semester 1 of 2020

```
SELECT course_id
  FROM course_runs
 WHERE semester = 1
   AND year = 2019
EXCEPT
SELECT course_id
  FROM course_runs
 WHERE semester = 1
   AND year = 2020;
```

We can allow a **default value** to be specified

Example:

```
CREATE TABLE student (
    stud_id    CHAR(5),
    name       VARCHAR(20) NOT NULL,
    dept_id    VARCHAR(20),
    tot_cred   NUMERIC(3,0) DEFAULT 0,
    PRIMARY KEY (stud_id),
    FOREIGN KEY (dept_id) REFERENCES department);
```

- ▶ The default value of tot\_cred is set to 0
- ▶ When a tuple is inserted if no value is provided its value is set to 0

Ensuring that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.

- ▶ Example: If MATH is a department ID appearing in one of the tuples in the instructor relation, then MATH also appears in some tuple in the department relation.

### **Formal Definition:**

Let  $A$  be a set of attributes. Let  $R$  and  $S$  be two relations that contain attributes  $A$  and where  $A$  is the primary key of  $S$ .  $A$  is said to be a foreign key of  $R$  if for any values of  $A$  appearing in  $R$  these values also appear in  $S$

```
CREATE TABLE course (
    course_id  VARCHAR(8),
    title       VARCHAR(50),
    dept_id    VARCHAR(20),
    credits     NUMERIC(2,0),
    PRIMARY KEY (course_id),
    FOREIGN KEY (dept_id) REFERENCES department
);


```

```
CREATE TABLE course (
    course_id  VARCHAR(8),
    title       VARCHAR(50),
    dept_id    VARCHAR(20),
    credits     NUMERIC(2,0),
    PRIMARY KEY (course_id),
    FOREIGN KEY (dept_id) REFERENCES department
    ON DELETE CASCADE
    ON UPDATE CASCADE
);
```

Alternative actions to cascade: `SET NULL`, `SET DEFAULT`

**Demo:** see `cascading.sql`

- ▶ In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- ▶ Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

```
SELECT instr_id, instr_name, dept_id  
      FROM instructor;
```

- ▶ A view provides a mechanism to hide certain data from the view of certain users.
- ▶ Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.
- ▶ A view is defined using the **CREATE VIEW** statement which has the form

**CREATE VIEW v AS < query expression >**

where **<query expression>** is any legal SQL expression. The view name is represented by v.

- ▶ Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- ▶ View definition is not the same as creating a new relation by evaluating the query expression.
- ▶ Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

A view of instructors without their salary

```
CREATE VIEW faculty AS
SELECT instr_id, instr_name, dept_id
  FROM instructor;
```

Create a view of department salary totals

```
CREATE VIEW departments_total_salary(dept_code, total_salary) AS
SELECT dept_id, SUM (salary)
  FROM instructor
 GROUP BY dept_id;
```

```
CREATE VIEW acad_year_2020 AS
SELECT semester, course_id, dept_id, title
  FROM course_runs NATURAL JOIN course
 WHERE year  = 2020;
```

```
CREATE VIEW cs_acad_year_2020 AS
SELECT semester, course_id, title
  FROM acad_year_2020
 WHERE dept_id= 'CS';
```

Add a new tuple to faculty view which we defined earlier

```
INSERT INTO faculty VALUES ('30765', 'James', 'CHEM');
```

This insertion must be represented by the insertion of the tuple

```
('30765', 'James', 'CHEM', NULL)
```

into the instructor relation.

**WARNING:** this feature is not supported in SQLite:

<https://www.sqlite.org/omitted.html>

Some updates cannot be translated uniquely:

```
CREATE VIEW instructor_info AS
SELECT instr_id, instr_name, building
  FROM instructor, department
 WHERE instructor.dept_id = department.dept_id;
```

```
INSERT INTO instructor_info
VALUES ('69987', 'Raul', 'Bute');
```

Which department, if there are multiple departments in Bute?

What if no department is in Bute?

Most SQL implementations allow updates only on simple views

- ▶ The **FROM** clause has only one database relation.
- ▶ The **SELECT** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or distinct specification.
- ▶ Any attribute not listed in the **SELECT** clause can be set to **NULL**
- ▶ The query does not have a **GROUP BY** or **HAVING** clause.

- ▶ When defining a view, simply create a physical table representing the view at the time of creation.
- ▶ Can be a cheaper option.
- ▶ How are updates handled to the “base” relations on which the view was defined?

Advantages	Disadvantages
Data independence Improved security Reduced complexity Convenience Customisation Data Integrity	Update restriction Performance

- ▶ **[DBSC]** Chapters 4-5, Database System Concepts, Silberschatz, Korth and Sudarshan
- ▶ **[DBS]** Chapter 7, Database Systems, Connolly and Begg
- ▶ **[DBD]** Chapters 15-16, Database Design, Watt and Eng
- ▶ Useful sites
  - ▶ <http://www.w3schools.com/sql/>
  - ▶ [http://sqlzoo.net/wiki/Main\\_Page](http://sqlzoo.net/wiki/Main_Page)

# IS5102

# Database Management Systems

## Lecture 11: Advanced SQL

Alexander Konovalov

[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

(with thanks to Susmit Sarkar)

2021



- ▶ Defining Table Structure
- ▶ Querying Tables
- ▶ Modifying Data
- ▶ Orderings and Aggregates
- ▶ Integrity Constraints
- ▶ Views

- ▶ Nested queries (subqueries)
- ▶ Join Expressions
- ▶ Authorisation
- ▶ Functions
- ▶ Triggers

- ▶ SQL provides a mechanism for the **nesting** of subqueries
- ▶ A subquery is a **SELECT-FROM-WHERE** expression that is nested within another query
- ▶ A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality

Find names of instructors with salary greater than that of **some** (at least one) instructor in the Physics and Astronomy department

This is an example demonstrated earlier:

```
SELECT DISTINCT T.instr_name
  FROM instructor AS T, instructor AS S
 WHERE T.salary > S.salary
   AND S.dept_id = 'PHYS';
```

In MySQL (but not in SQLite) the same result could be achieved using SOME clause

```
SELECT instr_name
  FROM instructor
 WHERE salary > SOME(SELECT salary
                        FROM instructor
                       WHERE dept_id = 'PHYS');
```

Find names of all instructors whose salary is greater than the salary of **all** instructors in the Physics and Astronomy department

```
SELECT instr_name
  FROM instructor
 WHERE salary > ALL(SELECT salary
                      FROM instructor
                     WHERE dept_id = 'PHYS');
```

SQL also allows a subquery expression to be used in the **FROM** clause

Example: Find the average instructors' salaries of those departments where the average salary is greater than £31,000.

```
SELECT dept_id, avg_salary
  FROM (SELECT dept_id, avg(salary) as avg_salary
          FROM instructor
         GROUP BY dept_id)
 WHERE avg_salary > 31000;
```

Note that we do not need to use the **HAVING** clause

Scalar subquery is one which is used where a single value is expected

```
SELECT dept_id,  
       (SELECT COUNT(*)  
            FROM instructor  
           WHERE department.dept_id = instructor.dept_id)  
      AS num_instructors  
   FROM department;
```

Runtime error if subquery returns more than one result tuple

- ▶ **Join condition** – defines **which tuples** in the two relations match, and **what attributes** are present in the result of the join.
- ▶ **Join type** – defines **how** tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Course

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>
MT5753	Statistical Modelling	Statistics	20
CS5012	Language & Computation	Comp.Sci	15
CS5010	Artificial Intelligence	Comp.Sci	15

Prereq

<i>course_id</i>	<i>prereq_id</i>
CS5012	CS5010
MT5753	MT5700
IS5120	IS5102

Observe:

Prereq information missing for CS5010

Course information missing for IS5120

```
SELECT *
  FROM course NATURAL JOIN prereq
```

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>
MT5753	Statistical Modelling	Statistics	20	MT5700
CS5012	Language & Computation	Comp.Sci	15	CS5010

- ▶ An **extension** of the join operation that avoids loss of information.
- ▶ Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- ▶ Uses **NULL** values.

```
SELECT *
  FROM course NATURAL LEFT OUTER JOIN prereq
```

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>
MT5753	Statistical Modelling	Statistics	20	MT5700
CS5012	Language & Computation	Comp.Sci	15	CS5010
CS5010	Artificial Intelligence	Comp.Sci	15	NULL

```
SELECT *
  FROM course NATURAL RIGHT OUTER JOIN prereq
```

course_id	title	dept_name	credits	prereq_id
MT5753	Statistical Modelling	Statistics	20	MT5700
CS5012	Language & Computation	Comp. Sci	15	CS5010
IS5120	NULL	NULL	NULL	IS5102

**Note:** Right Outer Join is not supported by SQLite

```
SELECT *
  FROM course NATURAL FULL OUTER JOIN prereq
```

course_id	title	dept_name	credits	prereq_id
MT5753	Statistical Modelling	Statistics	20	MT5700
CS5012	Language & Computation	Comp.Sci	15	CS5010
CS5010	Artificial Intelligence	Comp.Sci	15	NULL
IS5120	NULL	NULL	NULL	IS5102

**Note:** Full Outer Join is not supported by SQLite

- ▶ Some SQL implementations (but not SQLite) support **Discretionary Access Control**
  - ▶ User given access rights on database objects
  - ▶ Users gain certain privileges when they create an object and can pass these rights on at their discretion
- ▶ Mechanisms based on authorisation identifiers and ownership

Levels of authorization on parts of the database:

- ▶ **Read** – allows reading, but not modification of data.
- ▶ **Insert** – allows insertion of new data, but not modification of existing data.
- ▶ **Update** – allows modification, but not deletion of data.
- ▶ **Delete** – allows deletion of data.

- ▶ The **GRANT** statement is used to confer authorization

```
GRANT <privilege_list>
  ON <relation name or view name>
  TO <user_list>
```

- ▶ **<user\_list>** can be one of:
  - ▶ a user-id
  - ▶ public, which allows all valid users the privilege granted
  - ▶ a role (more on this later)
- ▶ Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- ▶ The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

- ▶ **SELECT**: allows read access to relation, or the ability to query using the view  
Example: grant users U1, U2, and U3 the select authorization on the instructor relation:

```
GRANT SELECT
```

```
  ON instructor  
  TO U1, U2, U3
```

- ▶ **INSERT**: the ability to insert tuples
- ▶ **UPDATE**: the ability to update using the SQL update statement
- ▶ **DELETE**: the ability to delete tuples.
- ▶ **ALL PRIVILEGES**: used as a short form for all the allowable privileges

Give the user with authorisation identifier Manager all privileges on the Staff table and allow their delegation

```
GRANT ALL PRIVILEGES
  ON Staff
  TO Manager
  WITH GRANT OPTION
```

Give users Personnel and Director the privileges of **SELECT** and **UPDATE** on the column salary of the Staff table.

```
GRANT SELECT, UPDATE (salary)  
  ON Staff  
  TO Personnel, Director
```

The **REVOKE** statement is used to revoke authorization.

```
REVOKE <privilege_list>
  ON <relation name or view name>
  FROM <user_list>
```

Example:

```
REVOKE SELECT
  ON branch
  FROM U1, U2, U3
```

<privilege-list> may be **ALL** to revoke all privileges the revoker may hold.

- ▶ Creating Roles

```
CREATE ROLE instructor;  
GRANT instructor TO Alexander;
```

- ▶ Privileges can be granted to roles:

```
GRANT SELECT ON takes_course TO instructor;
```

- ▶ Roles can be granted to users, as well as to other roles

```
CREATE ROLE teaching_assistant;  
GRANT teaching_assistant TO instructor;  
instructor inherits all privileges of teaching_assistant
```

- ▶ Chain of Roles

```
CREATE ROLE head;  
GRANT instructor TO head;  
GRANT head TO Ian;
```

- ▶ Chapter 4 and 5, Database System Concepts, 6th Ed. Silberschatz, Korth and Sudarshan
- ▶ Chapter 7, Database Systems, Connolly Begg

Useful URLs:

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

# IS5102

# Database Management Systems

## Lecture 12: Advanced SQL

Alexander Konovalov

[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

(with thanks to Susmit Sarkar)

2021



- ▶ Nested queries (subqueries)
- ▶ Join Expressions
- ▶ Authorisation
- ▶ Functions
- ▶ Triggers

Built-in functions include the following:

- ▶ Mathematical functions
- ▶ Date and time functions
- ▶ System functions
- ▶ ... (**many** more)

We have seen some SQLite **aggregate** functions already: AVG, COUNT, MAX, MIN, SUM

For a full list and documentation, see [https://sqlite.org/lang\\_aggfunc.html](https://sqlite.org/lang_aggfunc.html)

SQLite also supports a number of built-in **scalar** functions. E.g. the following query:

```
SELECT RANDOM() AS 'RANDOM',
       ABS(-42) AS 'ABS',
       LENGTH('SQL') AS 'LENGTH';
```

produces

RANDOM	ABS	LENGTH
-44132881353127140	42	3

For a full list and documentation, see [https://sqlite.org/lang\\_corefunc.html](https://sqlite.org/lang_corefunc.html)

MySql supports a range of mathematical functions

E.g. the output of the following query:

```
SELECT RAND() AS 'Random number',  
       PI(),  
       ROUND(PI(),4) AS 'Pi to four d.p.'
```

is

Random number	pi()	Pi to four d.p.
0.8797049102184409	3.141593	3.1416

The `round()` function rounds off a number to a specified number of places. The `ceil()` and `floor()` functions round off up/down to an integer value.

For a full list see:

<https://dev.mysql.com/doc/refman/8.0/en/mathematical-functions.html>

```
SELECT DATETIME('NOW') AS 'Current date/time',  
    DATE('NOW') AS 'Date',  
    TIME('NOW') AS 'Time',  
    JULIANDAY('2021-12-06') - JULIANDAY('NOW')  
        AS 'Days until exams';
```

For Date and Time functions available in SQLite, see:

[https://sqlite.org/lang\\_datefunc.html](https://sqlite.org/lang_datefunc.html)

# Date and Time Functions (MySQL example)

```
SELECT NOW()      AS 'Current date/time',
       CURTIME() AS 'Time',
       CURDATE() AS 'Date',
       DATEDIFF('2021-12-06', CURDATE())
              AS 'Days until exams';
```

For Date and Time functions available in MySQL, see:

<https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html>

- ▶ A **trigger** is a statement that is executed **automatically** by the system as a **side effect** of a modification to the database.
- ▶ An SQL trigger is executed each time an insert, update, or delete operation is performed.
- ▶ Triggers can be activated either **before** or **after** the event.
- ▶ For a trigger:
  - ▶ Specify the **conditions** under which the trigger is to be executed.
  - ▶ Specify the **actions** to be taken when the trigger executes.
  - ▶ This is referred to as the **Event-Condition-Action model**

Enforcing naming conventions for courses depending on the department

```
CREATE TRIGGER validate_course
  BEFORE INSERT ON course
BEGIN
  SELECT
  CASE
    WHEN NEW.dept_id = 'CS' AND
        (NEW.course_id NOT LIKE 'CS____' AND
         NEW.course_id NOT LIKE 'IS____') THEN
      RAISE (ABORT,'CS courses ID should start with CS or IS')
    WHEN NEW.dept_id = 'MATH' AND
        NEW.course_id NOT LIKE 'MT____' THEN
      RAISE (ABORT,'MATH courses ID should start with MT')
  END;
END;
```

Recording updates of student credits

```
CREATE TABLE credit_logs (
    id INTEGER PRIMARY KEY,
    stud_id TEXT,
    old_credits NUMERIC,
    new_credits NUMERIC,
    updated_at TEXT);
```

```
CREATE TRIGGER log_credits_update
AFTER UPDATE ON student
WHEN OLD.tot_cred <> NEW.tot_cred
BEGIN
    INSERT INTO credit_logs (stud_id, old_credits, new_credits, updated_at)
        VALUES (OLD.stud_id, OLD.tot_cred, NEW.tot_cred, DATETIME('NOW'));
END;
```

- ▶ Triggering event can be insert, delete or update
- ▶ In MySQL, values of attributes before and after an update can be referenced as follows:
  - ▶ `REFERENCING OLD ROW AS`: for deletes and updates
  - ▶ `REFERENCING NEW ROW AS`: for inserts and updates

## Triggering Events and Actions - example in MySQL

Triggers can be activated before an event, which can serve as extra constraints.

E.g. convert blank grades to **NULL**.

```
CREATE TRIGGER setnull_trigger
  BEFORE UPDATE OF takes
  REFERENCING NEW ROW AS nrow
  FOR EACH ROW
  WHEN (nrow.grade = ' ')
BEGIN
  SET nrow.grade = NULL;
END;
```

- ▶ Triggers were used earlier for tasks such as
  - ▶ Maintaining summary data (e.g., total salary of each department)
  - ▶ Replicating databases by recording changes to special relations (called change or delta relations) and having a separate process that applies the changes over to a replica
- ▶ There are **better** ways of doing these now:
  - ▶ Databases today provide built in materialized view facilities to maintain summary data
  - ▶ Databases provide built-in support for replication

Risk of unintended execution of triggers, for example, when

- ▶ loading data from a backup copy
- ▶ replicating updates at a remote site

Trigger execution can be disabled before such actions.

- ▶ SQL standard supports functions and procedures
  - ▶ Functions/procedures can be written in SQL itself, or in an external programming language.
  - ▶ Functions are particularly useful with specialized data types such as images and geometric objects.
- ▶ Example: functions to check if polygons overlap.
- ▶ SQL standard also supports a rich set of imperative constructs, including
  - ▶ Loops, if-then-else, assignment

- ▶ Can write own functions
- ▶ Stored in the database
- ▶ Functions mainly used where a value is returned
- ▶ Procedures may or may not return any value or may return more than one value
- ▶ Functions may be used in calculations and procedure in “business logic”

# Defining SQL Function – MySQL Example

Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
DELIMITER //
CREATE FUNCTION dept_count (dept_id CHAR(5))
RETURNS INTEGER
BEGIN
    DECLARE d_count INTEGER;
    SELECT COUNT(*) INTO d_count
        FROM instructor
        WHERE instructor.dept_id = dept_id;
    RETURN d_count;
END //
DELIMITER ;
```

## Using defined SQL Function – MySQL Example

Find the department name and budget of all departments with less than 3 instructors.

```
SELECT dept_id, budget
  FROM department
 WHERE dept_count(dept_id) < 3
```

The dept\_count function could instead be written as procedure:

```
CREATE PROCEDURE
    dept_count_proc (IN dept_id  VARCHAR(20),
                      OUT d_count  INTEGER)
BEGIN
    SELECT COUNT(*) INTO d_count
        FROM instructor
        WHERE instructor.dept_id = dept_count_proc.dept_id
END;
```

Procedures can be invoked either from an SQL procedure or from embedded SQL, using the `CALL` statement.

```
DECLARE d_count INTEGER;
CALL dept_count_proc('PHYS', d_count);
```

- ▶ Chapter 4 and 5, Database System Concepts, 6th Ed. Silberschatz, Korth & Sudarshan
- ▶ Chapter 7, Database Systems, Connolly & Begg

Useful URLs:

<https://www.w3schools.com/sql/default.asp>

[https://sqlite.org/lang\\_createtrigger.html](https://sqlite.org/lang_createtrigger.html)

<https://www.sqlitetutorial.net/sqlite-trigger/>

<https://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx>

<https://www.mysqltutorial.org/create-the-first-trigger-in-mysql.aspx>

# IS5102

# Database Management Systems

## Lecture 13: Relational Database Design

Alexander Konovalov

[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

(with thanks to Susmit Sarkar)

2021



- ▶ Features of Good Relational Design
- ▶ Update Anomalies
- ▶ Functional Dependencies
- ▶ Normalisation
- ▶ Problems of Normalisation

ID	Name	Department	Salary	Building	Head
10210	Einstein	Physics	56000	Allen	Curie
45893	Franklin	Biology	48250	Purdie	Watson
31822	Curie	Physics	67000	Allen	Curie
33821	Brandt	Computer Science	81000	Cole	Knuth
45910	Tao	Psychology	51900	Bute	Piaget
28675	Turing	Computer Science	49750	Cole	Knuth
67450	Crick	Biology	64500	Purdie	Watson
57264	Porter	Management	92000	Gateway	Porter
10101	Wu	Finance	78500	Gateway	Black
15675	Pavlov	Psychology	62800	Bute	Piaget

- ▶ Schema on previous slide has **redundant** data; the details of a department are repeated for every instructor.
- ▶ Relations that contain redundant information may potentially suffer from update anomalies.
- ▶ Types of update anomalies include
  - ▶ Insertion
  - ▶ Deletion
  - ▶ Modification

## Normalisation

Normalisation is a technique for producing a set of suitable relations that support the data requirements in a way that minimises redundancy and improves integrity of a database.

Functional dependency describes relationship between attributes.

## Functional Dependency

If  $A$  and  $B$  are attributes of relation  $R$ ,  $B$  is functionally dependent on  $A$  (denoted  $A \rightarrow B$ ), if each value of  $A$  in  $R$  is associated with exactly one value of  $B$  in  $R$ .

$A \rightarrow B$  is read as “ $B$  functionally dependent on  $A$ ” or “ $A$  functionally determines  $B$ ”

- ▶ Property of the meaning or semantics of the attributes in a relation.
- ▶ Can be represented diagrammatically.
- ▶ The **determinant** of a functional dependency refers to the attribute or group of attributes on the left-hand side of the arrow.

## Example: Functional Dependency

ID	Name	Department	Salary	Building	Head
10210	Einstein	Physics	56000	Allen	Curie
45893	Franklin	Biology	48250	Purdie	Watson
31822	Curie	Physics	67000	Allen	Curie
33821	Brandt	Computer Science	81000	Cole	Knuth
45910	Tao	Psychology	51900	Bute	Piaget
28675	Turing	Computer Science	49750	Cole	Knuth
67450	Crick	Biology	64500	Purdie	Watson
57264	Porter	Management	92000	Gateway	Porter
10101	Wu	Finance	78500	Gateway	Black
15675	Pavlov	Psychology	62800	Bute	Piaget

ID functionally determines Name

$ID \rightarrow Name$

ID functionally determines Department

$ID \rightarrow Department$

Name functionally determines Department

$Name \rightarrow Department$

But: Department does not functionally determine Name

$Department \not\rightarrow Name$

## Example: Functional Dependency that holds for all time

Consider the values shown in ID and name attributes of the schema on the previous slide.

Based on sample data, the following functional dependencies appear to hold.

$ID \rightarrow name$

$name \rightarrow ID$

However, the only functional dependency that remains true for all possible values for the ID and name attributes of the instructor relation is:

$ID \rightarrow name$

Determinants should have the minimal number of attributes necessary to maintain the functional dependency with the attribute(s) on the right hand-side.

This requirement is called **full functional dependency**.

## Full Functional Dependency

Let A and B are attributes of a relation. B is **fully functionally dependent** on A, if B is functionally dependent on A, but not on any proper subset of A.

$ID, name \rightarrow dept\_name$

This is indeed a functional dependency:

each value of  $(ID, name)$  is associated with a single value of  $dept\_name$ .

However,  $dept\_name$  is also functionally dependent on a subset of  $(ID, name)$ , namely  $ID$ .

Example above is a **partial dependency**.

Important to recognise a **transitive dependency** because its existence in a relation can potentially cause update anomalies.

## Transitive Dependency

Let  $A$ ,  $B$ , and  $C$  are attributes of a relation. If  $A \rightarrow B$  and  $B \rightarrow C$  then  $C$  is said to be **transitively dependent** on  $A$  via  $B$ .

Consider functional dependencies in the `instructor_and_department` relation

`instructor_and_department(name, salary, department, building, head)`

$ID \rightarrow name, salary, department, building, head$

$department \rightarrow building, head$

Transitive dependencies (both via `department`):

$ID \rightarrow building$

$ID \rightarrow head$

- ▶ Identifying all functional dependencies between a set of attributes is relatively simple if the meaning of each attribute and the relationships between the attributes are well understood.
- ▶ This information should be obtained via discussions with users and/or documentation such as the user requirements specification.
- ▶ However, if the users are unavailable for consultation and/or the documentation is incomplete then depending on the database application it may be necessary for the database designer to use their common sense and/or experience to provide the missing information.
- ▶ Otherwise look at sample. Important to establish that sample data values shown in relation are representative of all possible values that can be held by attributes.

## Identifying the Primary Key for a Relation using Functional Dependencies

- ▶ Main purpose of identifying a set of functional dependencies for a relation is to specify the set of integrity constraints that must hold on a relation.
- ▶ An important integrity constraint to consider first is the identification of candidate keys, one of which is selected to be the primary key for the relation.

$ID \rightarrow name, salary, department, building, head$

$department \rightarrow building, head$

- ▶ The **determinants** are ID, department
- ▶ Task: Identify all candidate key(s), identify the attribute (or group of attributes) that uniquely identifies each tuple in this relation.

$ID \rightarrow \text{name, salary, department, building, head}$

$\text{department} \rightarrow \text{building, head}$

- ▶ The **determinants** are ID, department
- ▶ Task: Identify all candidate key(s), identify the attribute (or group of attributes) that uniquely identifies each tuple in this relation.
- ▶ **All** attributes that are not part of a candidate key should be functionally dependent on the key.
- ▶ The only candidate key and therefore primary key for `instructor_and_department` relation, is ID as all other attributes of the relation are functionally dependent on ID.

- ▶ Formal technique for **analysing** a relation based on its primary key and the functional dependencies between the attributes of that relation.
- ▶ Often executed as a series of steps. Each step corresponds to a specific normal form, which has known properties.
- ▶ As normalisation proceeds, the relations become progressively more restricted (stronger) in format and also less vulnerable to update anomalies.

Chapters 14, 15 – Database Systems, Connolly and Begg

Chapter 8 – Database System Concepts, 6th Ed. Silberschatz, Korth and Sudarshan

Chapter 11 – Database Design, Watt and Eng

# IS5102

# Database Management Systems

## Lecture 14: Normalisation

Alexander Konovalov

[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

(with thanks to Susmit Sarkar)

2021



- ▶ Features of Good Relational Design
- ▶ Update Anomalies
- ▶ Functional Dependencies
- ▶ Normalisation
- ▶ Problems of Normalisation

- ▶ Formal technique for **analysing** a relation based on its primary key and the functional dependencies between the attributes of that relation.
- ▶ Often executed as a series of steps. Each step corresponds to a specific normal form, which has known properties.
- ▶ As normalisation proceeds, the relations become progressively more restricted (stronger) in format and also less vulnerable to update anomalies.

## Unnormalised form (UNF)

A table that may contain one or more repeating groups.

- ▶ For example, created when someone transformed the data from an information source (e.g. from a form used to collect the data) into table format with columns and rows
- ▶ Starting point for analysis

Proj Code	Proj Title	Proj Manager	Proj Bud- get	Emp No	Emp Name	Dept No	Dept Name	Hours
UNF data	1	Starship	Johnston	120000	12	Spratt	2	IT 15
					15	Smith	3	HR 10
					22	Brown	2	IT 44
	2	Enterprise	Williamson	250000	23	Black	3	HR 10
					12	Spratt	2	IT 48
					22	Brown	2	IT 16

### Useful reading:

Karl W. Broman & Kara H. Woo (2018), **Data Organization in Spreadsheets**, The American Statistician, 72:1, 2-10, <https://doi.org/10.1080/00031305.2017.1375989>

**Data Organization in Spreadsheets for Ecologists.** Data Carpentry lesson.  
<https://datacarpentry.org/spreadsheet-ecology-lesson/>

## First Normal Form (1NF)

A relation in which the intersection of each row and column contains one and only one value.

- ▶ Nominate an attribute or group of attributes to act as the key for the unnormalised table.
- ▶ Identify the repeating group(s) in the unnormalised table which repeats for the key attribute(s).
- ▶ Remove the repeating group by
  - ▶ Entering appropriate data into the empty columns of rows containing the repeating data ('flattening' the table).
  - Or by
  - ▶ Placing the repeating data along with a copy of the original key attribute(s) into a separate relation.

- ▶ Based on the concept of full functional dependency.
- ▶ Recall: Full functional dependency indicates that if  $A$  and  $B$  are attributes of a relation,  $B$  is fully dependent on  $A$  if  $B$  is functionally dependent on  $A$  but not on any proper subset of  $A$ .

## Second normal form (2NF)

A relation that is in 1NF and every non-primary-key attribute is fully functionally dependent on the primary key.

- ▶ Identify the primary key for the 1NF relation.
- ▶ Identify the functional dependencies in the relation.
- ▶ If partial dependencies exist on the primary key remove them by placing them in a new relation along with a copy of their determinant.

- ▶ Based on the concept of transitive dependency.
- ▶ Recall: Transitive Dependency is a condition where  $A$ ,  $B$  and  $C$  are attributes of a relation such that if  $A \rightarrow B$  and  $B \rightarrow C$ , then  $C$  is transitively dependent on  $A$  through  $B$ .

## Third Normal Form (3NF)

A relation that is in 1NF and 2NF and in which no non-primary-key attribute is transitively dependent on the primary key.

- ▶ Identify the primary key in the 2NF relation.
- ▶ Identify functional dependencies in the relation.
- ▶ If transitive dependencies exist on the primary key remove them by placing them in a new relation along with a copy of their determinant.

## Second normal form (2NF)

A relation that is in first normal form and every non-primary-key attribute is fully functionally dependent on *any candidate key*.

## Third normal form (3NF)

A relation that is in first and second normal form and in which no non-primary-key attribute is transitively dependent on *any candidate key*.

MatricNo	Name	ModID	Module	EnrolDate	Credits	Grade
1200001	Brown	IS5102	DBMS	02/09/20	15	13
1200001	Brown	CS4160	Operating Systems	02/09/20	15	15
1200002	Sharma	IS5102	DBMS	02/09/20	15	13
1200002	Sharma	CS4160	Operating Systems	02/09/20	15	15
1200003	Liu	CS5111	System Analysis	05/09/20	15	12
1200002	Sharma	CS5200	Project	05/09/20	60	14
1200004	Lopez	CS4160	Operating Systems	05/09/20	15	15

The table is subject to update anomalies.

For example, suppose we change the name of student with matriculation number 1200001 from Brown to Black. We have to change it identically in two rows, otherwise there is an update anomaly.

If we delete the row of CS5200, we lose all information about that module in a delete anomaly.

1. Identify minimal set of functional dependencies:

$\text{MatricNo} \rightarrow \text{Name}$

$\text{ModID} \rightarrow \text{Module, Credits}$

$\text{MatricNo, ModID} \rightarrow \text{EnrolDate, Grade}$

2. Identify primary key:  $(\text{MatricNo, ModID})$

3. The table is already in 1NF.

4. Remove partial functional dependencies of name, module and credits on the primary key  $(\text{MatricNo, ModID})$  by splitting into tables  $(\text{MatricNo, Name})$ ,  $(\text{ModID, Module, Credits})$ , and  $(\text{MatricNo, ModID, EnrolDate, Grade})$ . This makes the three tables in 2NF.

5. These three tables are already in 3NF, so we stop here.

Note that it does not satisfy the general definition of 3NF (no transitive dependencies on any candidate key) for the  $(\text{ModID, Module, Credits})$  relation as  $\text{Module}$  is a candidate key as well. So we can split that relation into  $(\text{ModID, Module})$  and  $(\text{ModID, Credits})$  to get 3NF in the general sense.

Normalisation is **definitely** not a cure-all solution

Bad database design can occur nevertheless

- ▶ Normalisation typically leads to **more** (smaller) tables
- ▶ To retrieve information needs **joins**
- ▶ Joins: expensive operations

- ▶ Very common queries might suggest denormalised tables
- ▶ Leads to faster lookups
- ▶ Leads to slower updates
- ▶ Leads to extra storage space
- ▶ Leads to possibility of programmer error
- ▶ Denormalise **after** normalising (and careful consideration)

## Some Examples:

- ▶ Materialised views (handled by database)
- ▶ Precalculated summaries (rollup, cube) (Data Analytics)
- ▶ Star schemas (Data Warehouses)

- ▶ Normalisation in general: use constraints
- ▶ Create relational structure that will not violate constraints on update
- ▶ Several kinds of constraints not considered in this module
- ▶ Leads to higher normal forms: 4NF, 5NF, ...

- ▶ In legacy (non DBMS) systems, common to have hierarchical data
- ▶ Often easiest to translate directly to UNF
- ▶ Can apply normalisation to create nice normalised relations
- ▶ Can go back and validate if relations make semantic sense

Chapters 14, 15 – Database Systems, Connolly and Begg

Chapter 8 – Database System Concepts, 6th Ed. Silberschatz, Korth and Sudarshan

Chapter 11 – Database Design, Watt and Eng

# IS5102

# Database Management Systems

## Lecture 15: Data Warehouses

Alexander Konovalov

[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

(with thanks to Susmit Sarkar)

2021



- ▶ Relational Model and Analytics
- ▶ Data Warehouse and Dimensionality Modeling
- ▶ The rise of the term NoSQL
- ▶ Different non-relational data management systems
- ▶ Graph databases
- ▶ Document-oriented databases

- ▶ Growing amounts of data in **operational databases**  
(RDBMS has been around since late 1970s)
- ▶ Not directly suited for **decision support**
- ▶ Typically: numerous operational systems with overlapping and sometimes contradictory definitions

Decision making require access to data from multiple sources

- ▶ Multiple queries to individual sources?
- ▶ Do they have historic data?

Need a **data warehouse** to store data acquired from multiple sources, under a unified schema, in a single repository

## Definition

A subject-oriented, integrated, time-variant, and non-volatile collection of data in support of management decision-making process (Inmon, 1993).

Organized around the major **subjects** of the enterprise (e.g. customers, products, and sales)

Rather than major **application areas** (e.g. customer invoicing, stock control, and product sales).

**Integrated** application-oriented data from different source systems,  
... often including **inconsistent** data

Must be made consistent to present a unified view of the data

Data in the warehouse is **only** accurate and valid at **some point in time** or over some time interval.

Time-variance is also shown in the extended time that the data is held, the implicit or explicit association of time with all data, and the fact that the data represents a series of snapshots

Data in the warehouse is not normally updated in real-time (RT) but is refreshed from operational systems on a regular basis. (However, emerging trend is towards RT or near RT DWs).

New data is always added as a **supplement** to the database, rather than a replacement.

- ▶ DBMS
- ▶ Data loaders

Data loaders retrieve information from various data sources

DBMS are used by various tools to perform queries and analyse information

When and how to gather data?

- ▶ **source-driven architecture**: data sources submit new information, continuously or periodically
- ▶ **destination-driven architecture**: data periodically requested by the warehouse

Which schema to use?

- ▶ integrate data to a warehouse schema
- ▶ may involve **data cleansing, deduplication** (also known as **merge & purge**), **householding**, changing units of measurements, etc.
- ▶ warehouse as materialised view

- ▶ How to propagate updates (view maintenance)
- ▶ Which data to summarise?

**ETL**: extract – transform – load model

**ELT**: extract – load – transform model

For decision-support queries:

**Query profile** generated for each user, group of users, or the data warehouse  
... based on information that describes the **characteristics** of the queries such as  
frequency,  
target table(s),  
and size of results set

A logical design technique that aims to present the data in a standard, intuitive form that allows for high-performance access

Every dimensional model (DM) is composed of one table with a composite primary key, called the **fact table**, and a set of smaller tables called **dimension tables**

Attributes in the fact table classified as **measure attributes** or **dimension attributes**

**Multidimensional data:** data that can be modelled using dimension and measure attributes

Each dimension table has a simple (non-composite) primary key that corresponds exactly to one of the components of the composite key in the fact table.

Forms “star-like” structure, which is called a star schema or star join.

Star schema is a logical structure that has a **fact table** (containing factual data) in the center, surrounded by **denormalized dimension tables** (containing reference data).

Facts are generated by events that occurred in the past, and are unlikely to change, regardless of how they are analysed

Star schemas can be used to speed up query performance by denormalizing reference information into a single dimension table.

Complex data warehouse designs may have more than one fact table, and also multiple levels of dimension tables, leading to **snowflake schemas**

All natural keys are replaced with **surrogate keys**

Means that every join between fact and dimension tables is based on surrogate keys, not natural keys

Surrogate keys allows the data in the warehouse to have some **independence** from the data used and produced by the OLTP (Online Transactional Processing) systems.

**Data lake** - a repository where data can be stored in multiple formats

No up-front efforts to preprocess data . . .

. . . at a cost of more efforts and flexibility needed when creating queries

Chapter 11, *Database System Concepts*, Silberschatz, Korth and Sudarshan

Chapter 33, *Database Systems*, Connolly and Begg

# IS5102

# Database Management Systems

## Lecture 16: Beyond SQL

Alexander Konovalov

[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

(with thanks to Susmit Sarkar)

2021



- ▶ The rise of the term NoSQL
- ▶ Different non-relational data management systems
- ▶ Graph databases
- ▶ Document-oriented databases

- ▶ Object-oriented
- ▶ Semistructured (XML)
- ▶ Graph databases
- ▶ Key-Value databases
- ▶ Document-oriented databases

Relational Databases have a long, successful history

Used in a wide variety of operational contexts

SQL is a standard (relational) query language

Useful when data is tabular

When data is non-tabular, it is less clear

This has **always** been true

New rebranding as **NoSQL** to emphasise additional capabilities

Programming models are often **object-oriented**

Very structured data

- ▶ Encapsulation
- ▶ Inheritance

Not a good fit for SQL data models

Object-oriented databases (e.g. Versant, db4o) with Object Query Language

Object-relational mapping (e.g. Hibernate from Java)

## EXtended Markup Language (XML)

- ▶ Tree structured (nested)
- ▶ Data definition can be changed
- ▶ Common interchange format

Some databases can store XML

Several more can produce (and sometimes consume) XML

Useful for web services (and other service-oriented architectures)

Query languages can be defined (XPath, XQuery)

Much data is now in how things are connected

Social networks are prime example

*Value in Relationships*

Querying these are hard in relational DBMS

Graph databases are structured as nodes (like entities) and relationships

Designed for fast querying of **relatedness**-information

**Follow** the links along

Neo4J most widely used

<http://neo4j.com/>

Nodes can have different types (like schemas)

So can relationships

And they can all have properties

```
(:Person) -[:LIVES_IN]-> (:City) -[:PART_OF]-> (:Country)
```

Useful in writing queries

```
MATCH (s:Person {name: 'Alexander Konovalov'}) -[:LIVES_IN]-> (e:City)
      <-[:IS_IN] - (r:Restaurant)
RETURN r.name
```

Often there is minimal formal structure

But huge amounts of data

Associations of **keys** to **values**

One approach: store these associations natively

Allow several indices, ad-hoc queries

Examples: Riak, Apache Cassandra

<http://basho.com/products/#riak>

<http://cassandra.apache.org/>

Store documents

... and their **metadata**

Metadata tends to be key-value associations

Examples: CouchDB, MongoDB

<http://couchdb.apache.org/>

<https://www.mongodb.org/>

**Documents**: MongoDB analogue for what we call **tuples**

**Collections**: MongoDB analogue for what we call **schema**

Can be (and usually is) nested. Also can be (and usually is) denormalised

Example:

```
{ 'project name': 'Starship',
  'project code': '1',
  'manager' : { 'name' : 'Johnston',
    'staff id': '120',
    'phone': '42371' },
  'employees' : [
    { 'name': 'Brown', 'staff id': '108', 'hours': 12 },
    { 'name': 'Brown', 'staff id': '108', 'hours': 20 }
  ]
}
```

Umbrella term for graph, key-value, document (and other?) non-relational DBMS

Emphasise different query models

Analytics driving many of these models

Chapter 11, *Database System Concepts*, Silberschatz, Korth and Sudarshan

Chapter 33, *Database Systems*, Connolly and Begg

# IS5102

# Database Management Systems

## Lecture 17: Data Analytics

Alexander Konovalov

[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

(with thanks to Susmit Sarkar)

2021



Umbrella term for graph, key-value, document (and other) non-relational DBMS

Emphasise different query models

Analytics driving many of these models

## What's in a name?

- ▶ Data Mining
- ▶ Data Science
- ▶ Data Analytics
- ▶ Machine Learning

What's in a name?

- ▶ Data Mining
- ▶ Data Science
- ▶ Data Analytics
- ▶ Machine Learning

Fuzzy distinctions ...

Practically, aim at the same thing

## Data Mining

The process of extracting valid, previously unknown, comprehensible, and actionable information from large databases and using it to make crucial business decisions.  
(Simoudis,1996).

Involves the analysis of data and the use of software techniques for finding hidden and unexpected patterns and relationships in sets of data.

## Qualitative data

- ▶ Attributes, properties, category, ...
- ▶ For instance Weather Descriptors (foggy, misty, cold, ...)
- ▶ Sometimes numbers as labels (e.g. Likert Scale)

## Quantitative data

- ▶ Can be Continuous (e.g. temperature)
- ▶ Can be Discrete (e.g. number of things, persons)

Aims to establish links (associations) between records, or sets of records, in a database.

There are three specializations

- ▶ Associations discovery
- ▶ Sequential pattern discovery
- ▶ Similar time sequence discovery

Applications include product affinity analysis, direct marketing, and stock price movement.

**Associations Discovery:** Finds items that imply the presence of other items in the same event.

e.g. 'When a customer rents property for more than 2 years and is more than 25 years old, in 40% of cases, the customer will buy a property.'

**Sequential Pattern Discovery:** Finds patterns between events such that the presence of one set of items is followed by another set of items in a database of events over a period of time.

e.g. Used to understand long term customer buying behavior.

**Similar Time Sequence Discovery:** Finds links between two sets of data that are time-dependent, and is based on the degree of similarity between the patterns that both time series demonstrate.

e.g. 'Within three months of buying property, new home owners will purchase goods such as cookers, freezers, and washing machines'.

Databases have traditionally been very concerned with **consistency**

An issue in many concurrent accesses

Seemingly at odds with replicated and sharded data for analysis

... no longer so clear (if writes are infrequent)

Strengths lie in processing Big Data

Analytics, Web Applications

Designed for fast, flexible, complex queries

Designed for high availability

Often (not always) give up Strong Consistency

Interaction of Consistency, Availability, Partition Tolerance (Reliability) **CAP**

RDBMS are not going away

Neither are NoSQL databases

Lessons that transfer:

- ▶ high-scalability
- ▶ consistency

New branding: NewSQL (e.g. Google Spanner)

Chapter 11, *Database System Concepts*, Silberschatz, Korth and Sudarshan

Chapter 33, *Database Systems*, Connolly and Begg

# pymongo demonstration

Based on examples from <https://www.mongodb.com/blog/post/getting-started-with-python-and-mongodb> (<https://www.mongodb.com/blog/post/getting-started-with-python-and-mongodb>).

In [1]:

```
import sys
```

In [2]:

```
!{sys.executable} -m pip install pymongo --user
```

```
Requirement already satisfied: pymongo in /Users/alexk/Library/Python/3.9/lib/python/site-packages (3.12.1)
WARNING: Value for scheme.headers does not match. Please report this to <https://github.com/pypa/pip/issues/10151>
distutils: /Users/alexk/Library/Python/3.9/include/python3.9/UNKNOWN
sysconfig: /Users/alexk/Library/Python/3.9/include/UNKNOWN
WARNING: Additional context:
user = True
home = None
root = None
prefix = None
WARNING: You are using pip version 21.2.1; however, version 21.3.1 is available.
You should consider upgrading via the '/Library/Frameworks/Python.framework/Versions/3.9/bin/python3.9 -m pip install --upgrade pip' command.
```

In [3]:

```
# pprint library is used to make the output look more pretty
from pprint import pprint

# randint library needed to generate some randomise content of the database
from random import randint
```

## To establish a connection to MongoDB with PyMongo you use the MongoClient class.

Connect to MongoDB, change the URL to reflect your own connection string

Installation and running: <https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/> (<https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/>).

If you don't want/need a background service you can just run:

```
mongod --config /usr/local/etc/mongod.conf
```

and terminate it after the demo

In [4]:

```
from pymongo import MongoClient
```

In [5]:

```
#Step 1: Connect to MongoDB - Note: Change connection string as needed
client = MongoClient(port=27017)
db=client.business
```

In [6]:

```
#Step 2: Create sample data
names = ['Kitchen', 'Animal', 'State', 'Tastey', 'Big', 'City', 'Fish', 'Pizza', 'Goat',
company_type = ['LLC', 'Inc', 'Company', 'Corporation']
company_cuisine = ['Pizza', 'Bar Food', 'Fast Food', 'Italian', 'Mexican', 'American'
for x in range(1, 501):
    business = {
        'name' : names[randint(0, (len(names)-1))] + ' ' + names[randint(0, (len(names)-1))],
        'rating' : randint(1, 5),
        'cuisine' : company_cuisine[randint(0, (len(company_cuisine)-1))]
    }
    #Step 3: Insert business object directly into MongoDB via insert_one
    result=db.reviews.insert_one(business)
    #Step 4: Print to the console the ObjectId of the new document
    print('Created {0} of 500 as {1}'.format(x,result.inserted_id))
#Step 5: Tell us that you are done
print('finished creating 500 business reviews')
```

Created 1 of 500 as 61965e71e18af34b722e9815  
Created 2 of 500 as 61965e71e18af34b722e9816  
Created 3 of 500 as 61965e71e18af34b722e9817  
Created 4 of 500 as 61965e71e18af34b722e9818  
Created 5 of 500 as 61965e71e18af34b722e9819  
Created 6 of 500 as 61965e71e18af34b722e981a  
Created 7 of 500 as 61965e71e18af34b722e981b  
Created 8 of 500 as 61965e71e18af34b722e981c  
Created 9 of 500 as 61965e71e18af34b722e981d  
Created 10 of 500 as 61965e71e18af34b722e981e  
Created 11 of 500 as 61965e71e18af34b722e981f  
Created 12 of 500 as 61965e71e18af34b722e9820  
Created 13 of 500 as 61965e71e18af34b722e9821  
Created 14 of 500 as 61965e71e18af34b722e9822  
Created 15 of 500 as 61965e71e18af34b722e9823  
Created 16 of 500 as 61965e71e18af34b722e9824  
Created 17 of 500 as 61965e71e18af34b722e9825  
Created 18 of 500 as 61965e71e18af34b722e9826  
Created 19 of 500 as 61965e71e18af34b722e9827

In [7]:

```
fivestar = db.reviews.find_one({'rating': 4})  
print(fivestar)
```

```
{ '_id': ObjectId('618d40c5ceb4d06ed8f151d8'), 'name': 'Pizza Salty Corporation', 'rating': 4, 'cuisine': 'Pizza'}
```

In [8]:

```
fivestar = db.reviews.find_one({'cuisine': 'Pizza'})
print(fivestar)
```

```
{'_id': ObjectId('618d40c5ceb4d06ed8f151d8'), 'name': 'Pizza Salty Corporation', 'rating': 4, 'cuisine': 'Pizza'}
```

In [9]:

```
rating = 5
fivestarcnt = db.reviews.find({'rating': rating})
print(fivestarcnt.collection.count_documents({'rating': rating}))
```

354

In [10]:

```
# Now let's use the aggregation framework to sum the occurrence of each rating across
print('\nThe sum of each rating occurrence across all data grouped by rating ')
stargroup=db.reviews.aggregate(
# The Aggregation Pipeline is defined as an array of different operations
[
# The first stage in this pipe is to group data
{ '$group':
  { '_id': "$rating",
    "count" :
      { '$sum' :1 }
  }
},
# The second stage in this pipe is to sort the data
{ "$sort": { "_id":1}}
}
# Close the array with the ] tag
]
# Print the result
for group in stargroup:
  print(group)
```

The sum of each rating occurrence across all data grouped by rating

```
{'_id': 1, 'count': 362}
{'_id': 2, 'count': 370}
{'_id': 3, 'count': 368}
{'_id': 4, 'count': 346}
{'_id': 5, 'count': 354}
```

## Updating data with PyMongo

In [11]:

```
ASingleReview = db.reviews.find_one({"cuisine": "Vegetarian"})
print('A sample document:')
pprint(ASingleReview)

result = db.reviews.update_one({'_id' : ASingleReview.get('_id')}, {'$inc': {'likes': 1}})
print('Number of documents modified : ' + str(result.modified_count))

UpdatedDocument = db.reviews.find_one({'_id': ASingleReview.get('_id')})
print('The updated document:')
pprint(UpdatedDocument)
```

```
A sample document:
{'_id': ObjectId('618d40c5ceb4d06ed8f151d7'),
 'cuisine': 'Vegetarian',
 'likes': 3,
 'name': 'Fish Tastey Corporation',
 'rating': 5}
Number of documents modified : 1
The updated document:
{'_id': ObjectId('618d40c5ceb4d06ed8f151d7'),
 'cuisine': 'Vegetarian',
 'likes': 4,
 'name': 'Fish Tastey Corporation',
 'rating': 5}
```

## Deleting documents

In [12]:

```
result = db.reviews.delete_many({"cuisine": "Bar Food"})
```

In [13]:

```
result
```

Out[13]:

```
<pymongo.results.DeleteResult at 0x7fe9a0757bc0>
```

In [14]:

```
result.deleted_count
```

Out[14]:

```
67
```

## Administrative interface

In [15]:

```
client = MongoClient("mongodb://localhost:27017/")
db=client.admin
```

In [16]:

```
# Issue the serverStatus command and print the results
serverStatusResult=db.command("serverStatus")
pprint(serverStatusResult)

{'asserts': {'msg': 0,
             'regular': 0,
             'rollovers': 0,
             'tripwire': 0,
             'user': 7,
             'warning': 0},
 'catalogStats': {'capped': 0,
                  'collections': 1,
                  'internalCollections': 3,
                  'internalViews': 0,
                  'timeseries': 0,
                  'views': 0},
 'connections': {'active': 3,
                 'available': 2042,
                 'awaitingTopologyChanges': 2,
                 'current': 6,
                 'exhaustHello': 0,
                 'exhaustIsMaster': 0,
                 'threaded': 6,
                 'total': 3}}
```

# Working with SQLite in Python

Based on <https://docs.python.org/3/library/sqlite3.html> (<https://docs.python.org/3/library/sqlite3.html>).

In [1]:

```
import sqlite3
```

In [2]:

```
con = sqlite3.connect(":memory:")
```

In [3]:

```
?sqlite3.connect
```

This is a Connection object that represents the database.

`:memory:` is a special name to create a temporary database.

In [4]:

```
con
```

Out[4]:

```
<sqlite3.Connection at 0x7f815049eb70>
```

In [5]:

```
# cursor object, which has `execute()` method to perform SQL commands
cur = con.cursor()
```

In [6]:

```
cur
```

Out[6]:

```
<sqlite3.Cursor at 0x7f81405db7a0>
```

In [7]:

```
cur.execute('''
CREATE TABLE department (
    dept_id      CHAR(5),
    dept_name    VARCHAR(20) NOT NULL,
    building     VARCHAR(15),
    budget       NUMERIC(12,2),
    PRIMARY KEY (dept_id)
);
'''')
```

Out[7]:

```
<sqlite3.Cursor at 0x7f81405db7a0>
```

In [8]:

```
cur.execute(''  
    INSERT INTO department  
VALUES ('CS', 'Computer Science', 'Jack Cole', 1500000.00),  
           ('CHEM', 'Chemistry', 'Purdie', 200000.00),  
           ('MATH', 'Maths and Stats', 'Maths', 900000.00),  
           ('PHYS', 'Phys and Astro', 'Physics', 1500000.00);  
'')
```

Out[8]:

```
<sqlite3.Cursor at 0x7f81405db7a0>
```

In [9]:

```
for dept in cur.execute('SELECT * FROM department'):  
    print(dept)
```

```
('CS', 'Computer Science', 'Jack Cole', 1500000)  
('CHEM', 'Chemistry', 'Purdie', 200000)  
('MATH', 'Maths and Stats', 'Maths', 900000)  
('PHYS', 'Phys and Astro', 'Physics', 1500000)
```

In [10]:

```
a=cur.execute('SELECT * FROM department')
```

In [11]:

```
dept=[x for x in a]
```

In [12]:

```
dept
```

Out[12]:

```
[('CS', 'Computer Science', 'Jack Cole', 1500000),  
 ('CHEM', 'Chemistry', 'Purdie', 200000),  
 ('MATH', 'Maths and Stats', 'Maths', 900000),  
 ('PHYS', 'Phys and Astro', 'Physics', 1500000)]
```

In [13]:

```
dept[0]
```

Out[13]:

```
('CS', 'Computer Science', 'Jack Cole', 1500000)
```

In [14]:

```
2*dept[0][3]
```

Out[14]:

```
3000000
```

In [15]:

```
type(depts[1][3])
```

Out[15]:

```
int
```

Try now to have budget with some non-zero digits after the decimal point and re-run the notebook. What happens?

Also, try to execute some SQL command twice. What happens?

In [16]:

```
cur.execute('''
CREATE TABLE instructor (
    instr_id    CHAR (5),
    instr_name  VARCHAR(20) NOT NULL,
    dept_id    VARCHAR(5),
    salary      NUMERIC (8,2),
    PRIMARY KEY (instr_id),
    FOREIGN KEY (dept_id) REFERENCES department);
'''')
```

Out[16]:

```
<sqlite3.Cursor at 0x7f81405db7a0>
```

In [17]:

```
cur.execute('''
CREATE TABLE student (
    stud_id    CHAR(5),
    name       VARCHAR(20) NOT NULL,
    dept_id   VARCHAR(20),
    tot_cred   NUMERIC(3,0) DEFAULT 0,
    PRIMARY KEY (stud_id),
    FOREIGN KEY (dept_id) REFERENCES department);
'''')
```

Out[17]:

```
<sqlite3.Cursor at 0x7f81405db7a0>
```

In [18]:

```
cur.execute(''  
INSERT INTO instructor  
VALUES ('45797', 'Bob', 'CS', 28000),  
        ('23541', 'Javier', 'CS', 33600),  
        ('22418', 'Karolina', 'CS', 27000),  
        ('34123', 'Layla', 'MATH', 27000),  
        ('12355', 'Petro', 'MATH', 32000),  
        ('52412', 'Jan', 'MATH', 29300),  
        ('21357', 'Isaac', 'CHEM', 37500),  
        ('13842', 'Ali', 'CHEM', 34900),  
        ('23456', 'Alice', 'PHYS', 29500),  
        ('45638', 'Sana', 'PHYS', 31500);  
        '')
```

Out[18]:

```
<sqlite3.Cursor at 0x7f81405db7a0>
```

In [19]:

```
cur.execute(''  
INSERT INTO student  
VALUES ('64545', 'Abdul', 'MATH', 180),  
        ('78778', 'Martha', 'MATH', 90),  
        ('99680', 'Eliot', 'CHEM', 90),  
        ('78621', 'Bartosz', 'CHEM', 90),  
        ('67868', 'Elias', 'CS', 90),  
        ('87690', 'Joao', 'CS', 90),  
        ('79879', 'Robert', 'CS', 90),  
        ('90780', 'Julia', 'CS', 120),  
        ('89675', 'Eilidh', 'PHYS', 120),  
        ('96544', 'Sarah', 'PHYS', 180);  
        '')
```

Out[19]:

```
<sqlite3.Cursor at 0x7f81405db7a0>
```

Obligatory XKCD: <https://xkcd.com/327/> (<https://xkcd.com/327/>)

In [20]:

```
for row in cur.execute('SELECT * FROM student'):  
    print(row)
```

```
('64545', 'Abdul', 'MATH', 180)  
('78778', 'Martha', 'MATH', 90)  
('99680', 'Eliot', 'CHEM', 90)  
('78621', 'Bartosz', 'CHEM', 90)  
('67868', 'Elias', 'CS', 90)  
('87690', 'Joao', 'CS', 90)  
('79879', 'Robert', 'CS', 90)  
('90780', 'Julia', 'CS', 120)  
('89675', 'Eilidh', 'PHYS', 120)  
('96544', 'Sarah', 'PHYS', 180)
```

In [21]:

```
name = "Robert'; DROP TABLE student;--"
```

In [22]:

```
command = "SELECT * FROM student WHERE name = '%s'" % name
```

In [23]:

```
command
```

Out[23]:

```
"SELECT * FROM student WHERE name = 'Robert'; DROP TABLE student;--'"
```

In [24]:

```
cur.executescript(command)
```

Out[24]:

```
<sqlite3.Cursor at 0x7f81405db7a0>
```

In [25]:

```
for row in cur.execute('SELECT * FROM student'):
    print(row)
```

```
-----
-----
OperationalError                                Traceback (most recent call
last)
<ipython-input-25-e35e6177d8e6> in <module>
----> 1 for row in cur.execute('SELECT * FROM student'):
      2     print(row)
```

```
OperationalError: no such table: student
```

Oops! Lets restore the table before we continue.

In [26]:

```
cur.execute('''
CREATE TABLE student (
    stud_id    CHAR(5),
    name       VARCHAR(20) NOT NULL,
    dept_id   VARCHAR(20),
    tot_cred   NUMERIC(3,0) DEFAULT 0,
    PRIMARY KEY (stud_id),
    FOREIGN KEY (dept_id) REFERENCES department);
''' )
```

Out[26]:

```
<sqlite3.Cursor at 0x7f81405db7a0>
```

In [27]:

```
cur.execute('''
INSERT INTO student
VALUES ('64545', 'Abdul', 'MATH', 180),
       ('78778', 'Martha', 'MATH', 90),
       ('99680', 'Eliot', 'CHEM', 90),
       ('78621', 'Bartosz', 'CHEM', 90),
       ('67868', 'Elias', 'CS', 90),
       ('87690', 'Joao', 'CS', 90),
       ('79879', 'Robert', 'CS', 90),
       ('90780', 'Julia', 'CS', 120),
       ('89675', 'Eilidh', 'PHYS', 120),
       ('96544', 'Sarah', 'PHYS', 180);
''')
```

Out[27]:

&lt;sqlite3.Cursor at 0x7f81405db7a0&gt;

This is how to do it using parameter substitution

- qmark style

In [28]:

```
cur.execute("INSERT INTO student VALUES (?, ?, ?, ?, ?)", ('87650', 'Naomi', 'CS', 90))
```

Out[28]:

&lt;sqlite3.Cursor at 0x7f81405db7a0&gt;

- named style

In [29]:

```
cur.execute("SELECT * FROM student WHERE dept_id=:dept", {"dept": "CS"})
```

Out[29]:

&lt;sqlite3.Cursor at 0x7f81405db7a0&gt;

In [30]:

```
cur.fetchall()
```

Out[30]:

```
[('67868', 'Elias', 'CS', 90),
 ('87690', 'Joao', 'CS', 90),
 ('79879', 'Robert', 'CS', 90),
 ('90780', 'Julia', 'CS', 120),
 ('87650', 'Naomi', 'CS', 90)]
```

Further reading: <https://doi.org/10.1371/journal.pcbi.1007007> (<https://doi.org/10.1371/journal.pcbi.1007007>)



# Data Analytics

IS5102, Lecture 19

22 November 2021

Alexander Konovalov [alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk) (<mailto:alexander.konovalov@st-andrews.ac.uk>)

*Thanks to: Vinodh Rajan and Susmit Sarkar*

## NoSQL

- Umbrella term for graph, key-value, document (and other) non-relational DBMS
- Emphasise different query models
- Analytics driving many of these models

## Overview

- Introduction
- Steps
- Operations
- Techniques
- Handling multivariate data

## Data Mining: What's in a name?

- Data Mining
- KDD (Knowledge Discovery and Data)
- Data Science
- Data Analytics
- Machine Learning

- The difference between these are very fuzzy and there is indeed a lot of overlap
- You can give proper individual text-book definitions for each of these.
- But practically, they more or less aim at the same thing

## Data Mining: What is it?

- The process of extracting valid, previously unknown, comprehensible, and actionable information from large databases and using it to make crucial business decisions, (Simoudis, 1996)
- Involves the analysis of data and the use of software techniques for finding hidden and unexpected patterns and relationships in sets of data
- Reveals information that is hidden and unexpected, as little value in finding patterns and relationships that are already intuitive
- Patterns and relationships are identified by examining the underlying rules and features in the data

## Data Mining: Why we need it?

- Existence of large amount of data waiting to be analyzed
- The ability to harness past experience/decisions to shape future decisions
- The obsessive need to predict something
- Discover patterns/relationships that are not very obvious
- Ability to model user/market/anything behavior

## Types of Data

- Qualitative Data
  - Describes “attributes” “labels” “properties” “category” etc.
  - For instance Weather descriptors
    - “foggy”, “misty”, “rainy”, “cold”, “sultry”, “dreich” so on and so forth
  - Sometimes numbers can also be used as labels
    - Likert Scale (1 – Strong Disagree to 5 – Strong Agree)
- Quantitative Data
  - Continuous Data
    - These can take fractional value such as temperature, distance etc,
  - Discrete Data
    - Can only take whole numbers such as number of students

## Data Mining: Steps

- Data Cleaning
- Data Integration
- Data Transformation/Normalization
- Data Reduction/Selection

## Data Mining: Steps: Data Cleaning

- Real data are rarely clean
  - They are noisy: Smoothen your data, remove outliers
  - They are incomplete: Fill in the missing values/ignore them
  - They are inconsistent: Ensure consistency
- You must make sure the data you work with is clean
- Remember: Garbage in, garbage out
  - This is true for data as well.

## Data Mining: Steps: Data Integration

- Your data is probably split across multiple databases/files
- They are also most probably heterogeneous
- They must be integrated properly into a coherent entity to which you can apply data mining operations
- This is also means (again) ensuring consistency between different entities

## Data Mining: Steps: Data Transformation/Normalization

- More often than not data is not in a directly usable form
- You may have to apply transformation/aggregations to make sensible use of data
  - You cannot compare prices from 1800's to 2000's without accounting for inflation
- You may also want to normalize data if the attributes are scale dependent to transform to the same scale
  - You may have two attributes for rating, where one is scored from 1 to 5 and the other is scored from 1 to 10

## Data Mining: Steps: Data Reduction/Selection

- You probably have to deal with a huge number of attributes in your data
- At least some of them are probably useless
- So you need to select the useful attributes and disregard the others
- In some cases, you may want to reduce the number of attributes by combining some of them

## Data Mining: Operations

- Predictive Modeling
- Link Analysis
- Deviation Detection

## Data Mining: Techniques

- Techniques are specific implementations of the data mining operations
- Each operation has its own strengths and weaknesses

## Data Mining: Techniques

- Predictive modeling techniques:
  - classification
  - value prediction
- Link analysis techniques:
  - association discovery
  - sequential pattern discovery
  - similar time sequence discovery
- Deviation detection techniques:
  - statistical analysis
  - visualisation

## Predictive Modeling

- Similar to the human learning experience

- uses observations to form a model of the important characteristics of some phenomenon
- Uses generalizations of 'real world' and ability to fit new data into a general framework

## Predictive Modeling

- Model is developed using a supervised learning approach, which has two phases: training and testing
  - Training builds a model using a large sample of historical data called a training set
  - Testing involves trying out the model on new, previously unseen data to determine its accuracy and physical performance characteristics

## Predictive Modeling

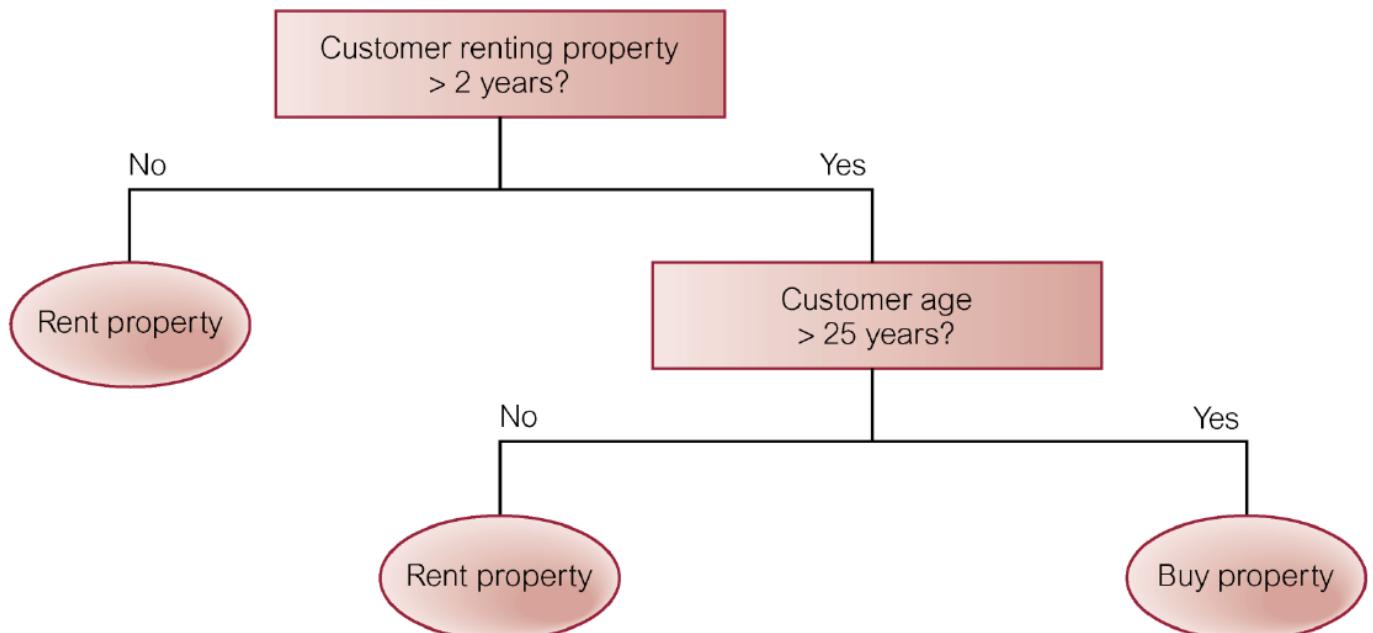
$$X \rightarrow f(X) \rightarrow Y$$

- $f(X)$  is the function approximation that is constructed to map a given  $X$  to  $Y$
- We attempt to learn  $f(X)$  through a training set of known  $(X, Y)$  pairs
- We use this to predict an unseen  $X$ , based on the above mapping

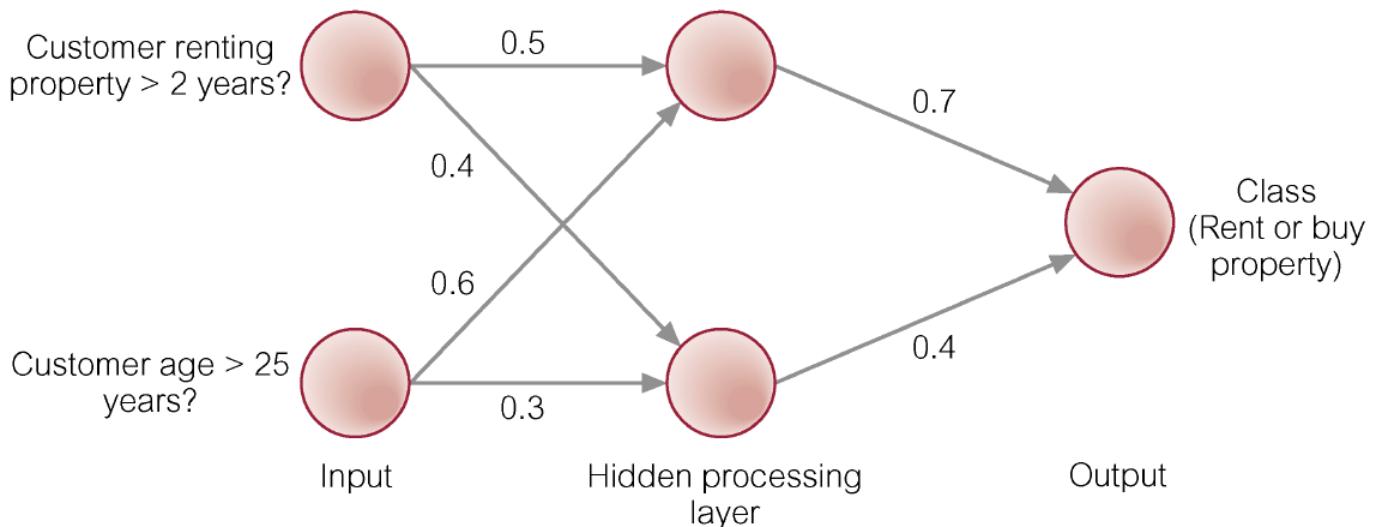
## Predictive Modeling: Classification

- Used to establish a specific predetermined class for each record in a database from a finite set of possible, class values
- Two specializations of classification: tree induction and neural induction

## Induction tree



## Neural network



## Predictive Modeling: Value Prediction

- Used to estimate a continuous numeric value that is associated with a database record
- Uses the traditional statistical techniques
- Relatively easy-to-use and understand

## Regression

- Explaining one variable in terms of another
- It models the relationship between two variables
- $y$  the dependent variable is modelled in terms of  $x$ , which is the independent variable
- It is used to predict/forecast or interpolate  $y$  given  $x$

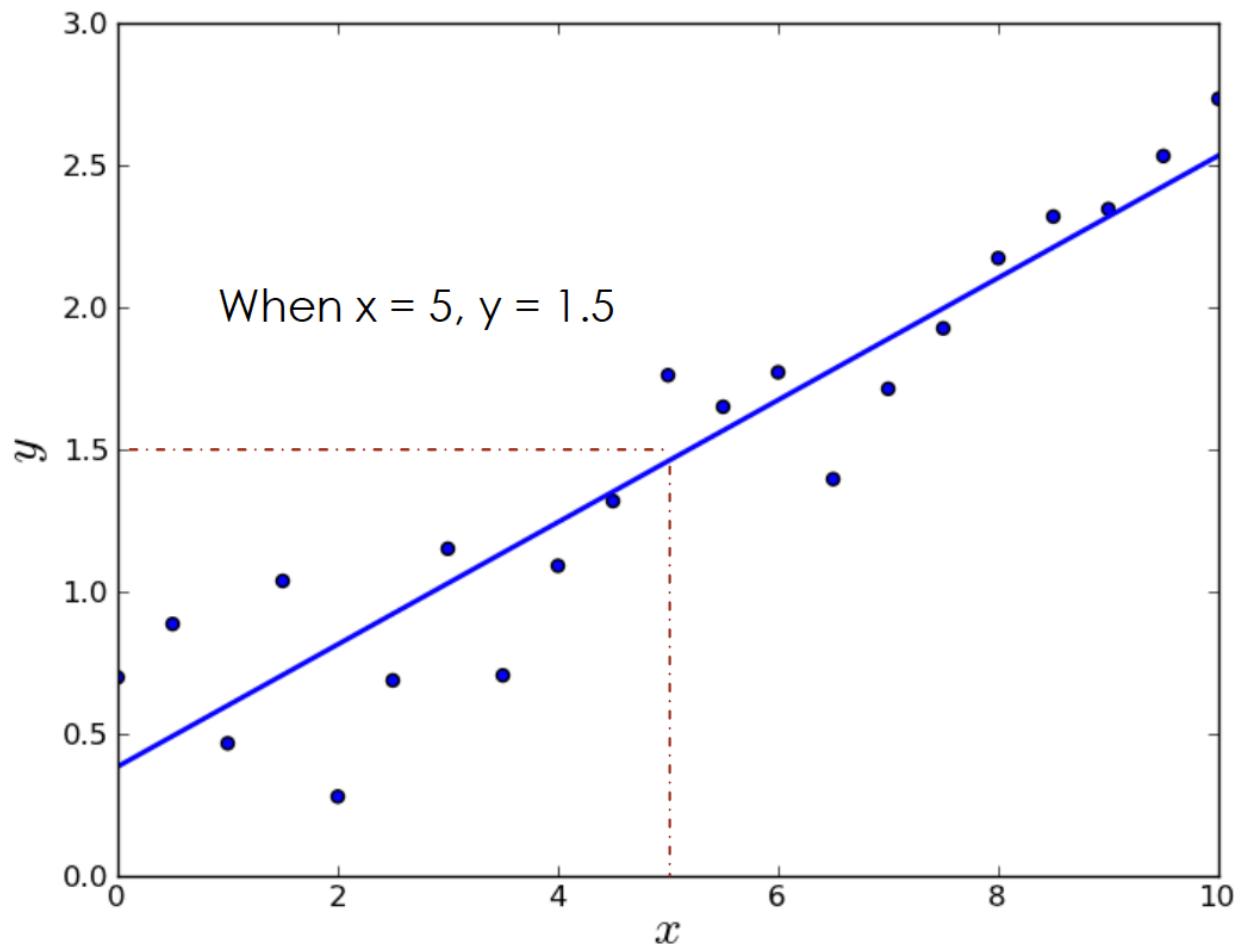
## Simple Linear Regression

- Linear regression fits the data in the form of a straight line:
  - $Y = aX + b$
  - $b$  = intercept on the  $y$ -axis
  - $a$  = slope
  - The fit is measured in terms of  $R^2$
  - Called the co-efficient of determination
  - $R^2 = 1$  means a perfect fit
  - Shows how much of variation in  $Y$  is explained in terms of  $X$

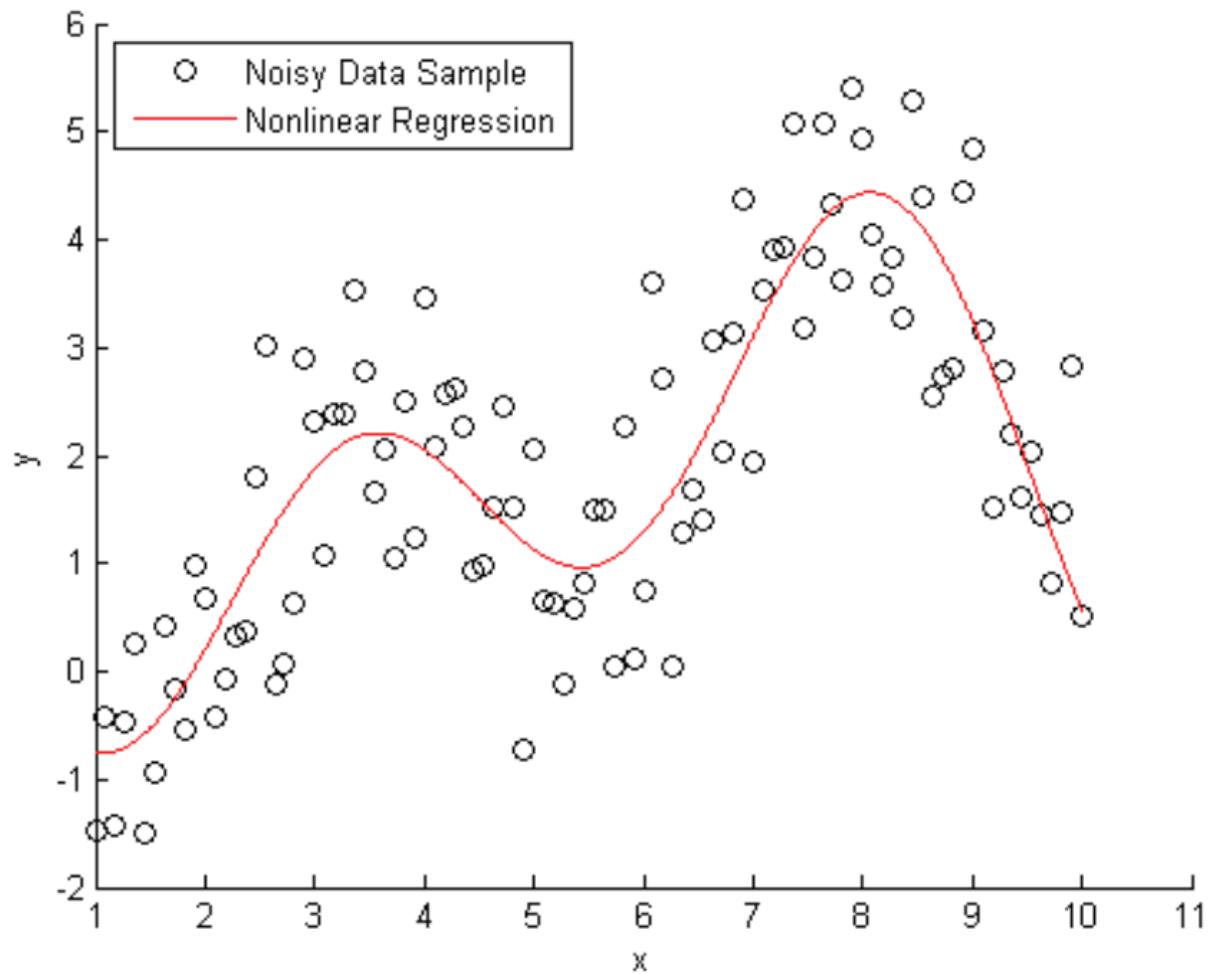
## Simple Linear Regression

- The relationship is not always linear
- If a linear model doesn't fit properly, try using a nonlinear model to fit the data
- Do not extrapolate/predict what would happen outside the range of the data. We just don't know

## Linear regression



## Non-linear regression

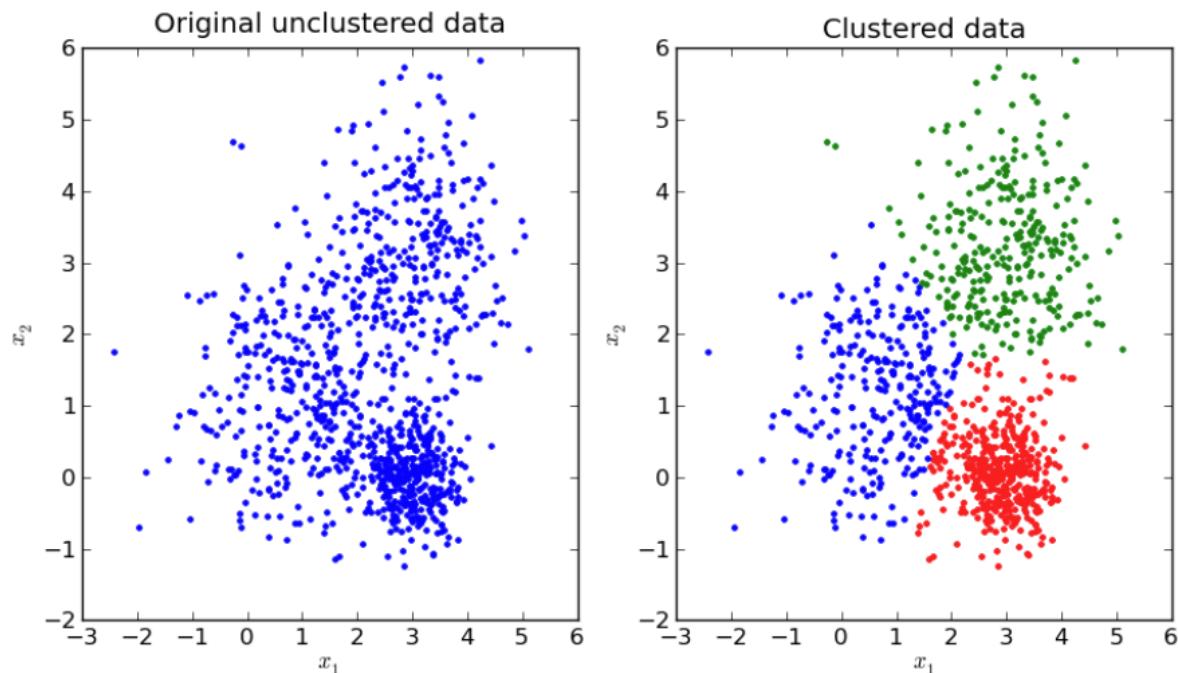


## Clustering

- The earlier techniques we saw (predictive modeling) are part of supervised learning
- This requires the existence of a training set. However, very frequently we come across data that we have no prior information (unsupervised learning)
- Clustering allows us to group related data without any prior information on how to do it
- Group related people based on their demographic attributes

## Clustering: *k*Means

- *k*Means is a very commonly used technique to perform clustering
- Divides  $n$  observations into  $k$  clusters ( $k \leq n$ ) clustered around a mean



## Link Analysis

- Aims to establish links (associations) between records, or sets of records, in a database
- There are three specializations
  - Associations discovery
  - Sequential pattern discovery
  - Similar time sequence discovery
- Applications include product affinity analysis, direct marketing, and stock price movement

## Link Analysis: Association Discovery

- Finds items that imply the presence of other items in the same event
- Affinities between items are represented by association rules
  - e.g. 'When a customer rents property for more than 2 years and is more than 25 years old, in 40% of cases, the customer will buy a property. This association happens in 35% of all customers who rent properties'

## Link Analysis: Sequential Pattern Discovery

- Finds patterns between events such that the presence of one set of items is followed by another set of items in a database of events over a period of time
- e.g. used to understand long term customer buying behavior

## Link Analysis: Similar Time Sequence Discovery

- Finds links between two sets of data that are time dependent, and is based on the degree of similarity between the patterns that both time series demonstrate

- e.g. Within three months of buying property, new home owners will purchase goods such as cookers, freezers, and washing machines

## Deviation Detection

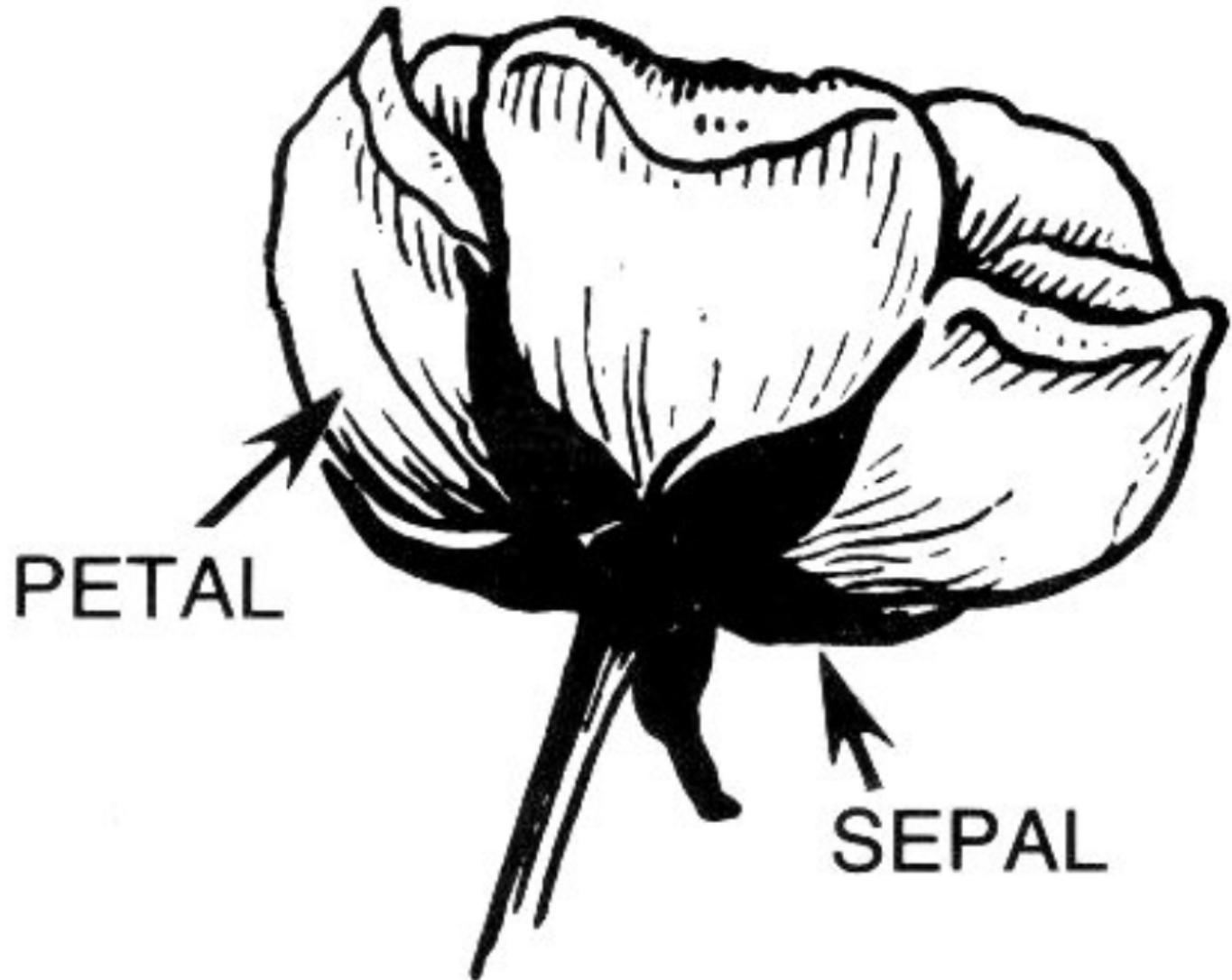
- Can be performed using statistics and visualization techniques or as a by-product of data mining
- Often a source of true discovery because it identifies outliers, which express deviation
- Applications include fraud detection, quality control, and defects tracing

## Multivariate Data

- Real datasets consists of multiple variables/attributes in tens or even hundreds
- For instance, house prices may depends on city, neighborhood, area, number of bedrooms and other factors
- The attributes are usually related to each other
- Visualizing and analyzing these data sets is comparatively complex

## Multivariate Data: Iris Flower Data Set

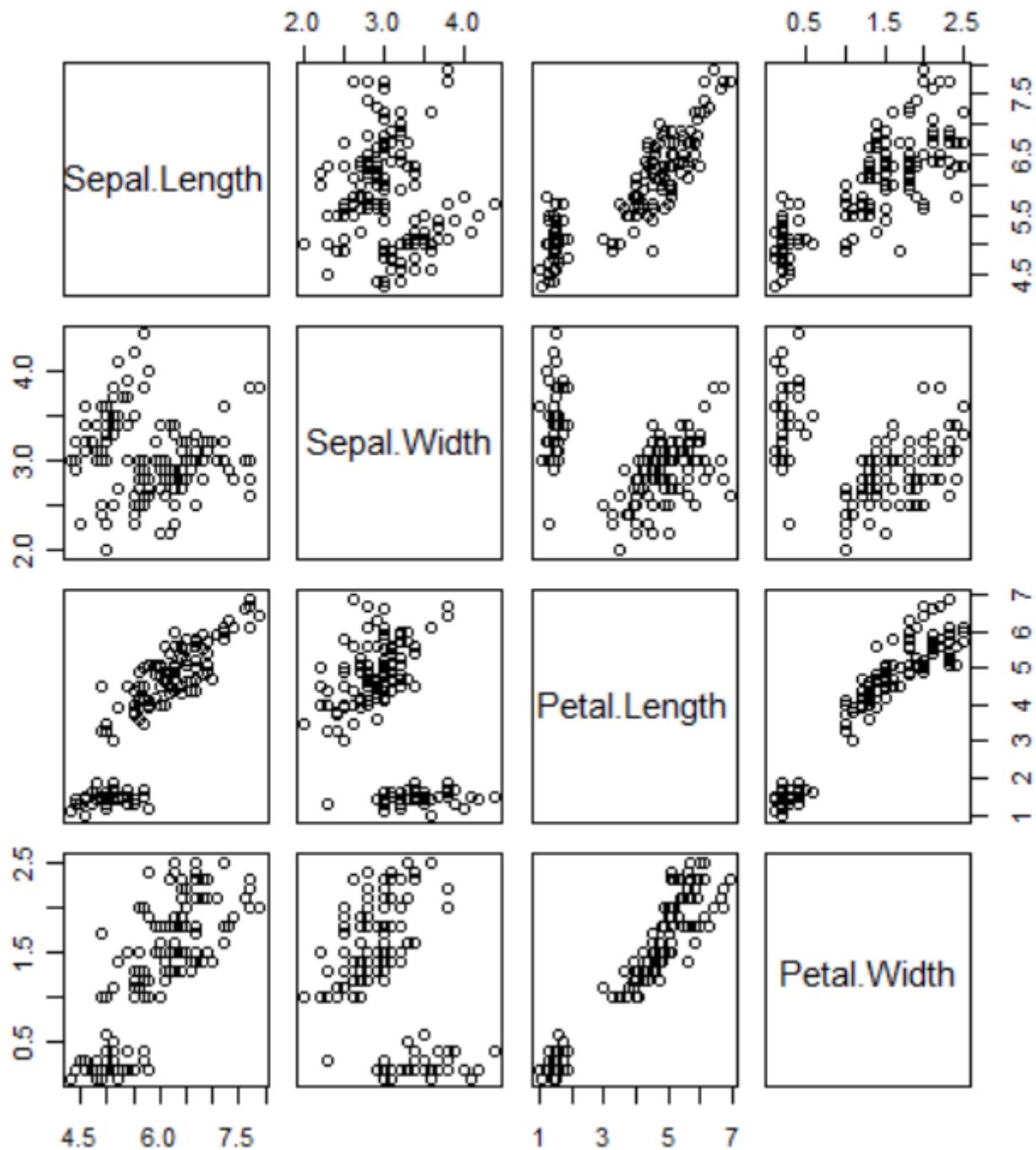
- The iris data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor).
- Has 150 samples in total
- Iris dataset has the attributes: Sepal Length ( $S_L$ ), Sepal Width( $S_W$ ), Petal Length ( $P_L$ ), Petal Width ( $P_W$ )



#### Visualizing Multivariate Data: Scatterplot Matrices

- 3D visualization is difficult to interpret by humans
- Visualizing more than 3 dimensions at the same becomes exponentially very difficult
- The easiest approach is to do a pairwise scatterplot and arrange them in a matrix to see the pairwise interaction between variables

## Scatterplot matrix

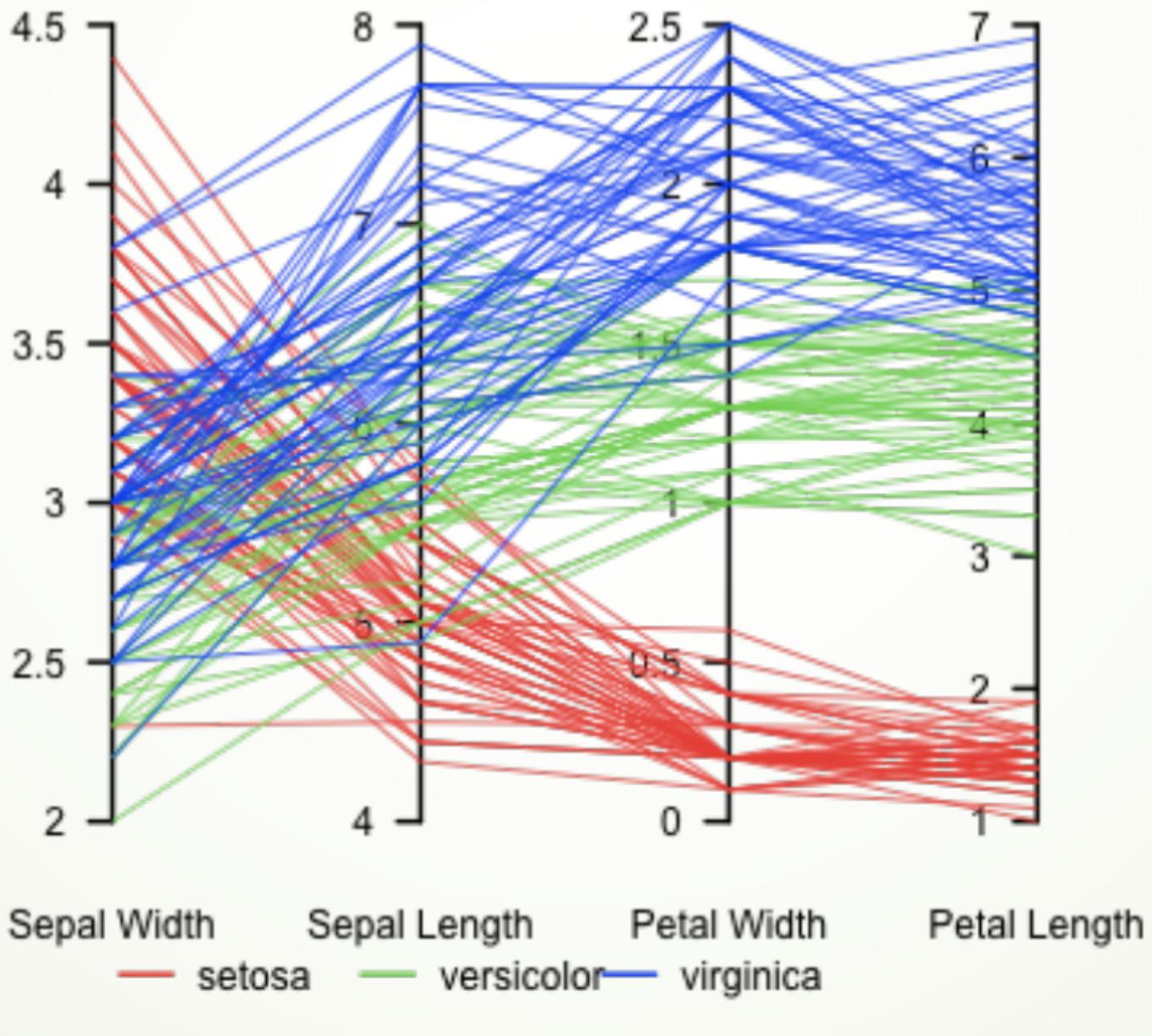


## Visualizing Multivariate Data: Parallel Coordinates Plots

- Allows to visualize all data attributes at the same time
- Consists of a number of axes arranged in parallel
- A data item is a line that moves across all the axes
- Can visualize interactions between multiple attributes at the same time
- Can detect patterns easily

## Parallel coordinates plot

## Parallel coordinate plot, Fisher's Iris data



## Analyzing Multivariate Data: Dimensionality Reduction

- Assuming, your data has 30 attributes – essentially it means an item has 30 dimensions.
  - You'd need  $435$  scatterplots!
- The analysis can be simplified if we are to reduce the dimensions (a.k.a attributes) to say 5
  - Now, you'd need only  $5C2 = 10$  scatterplots
- This is done by projecting data items from a higher dimensional space to a lower dimensional space

## Dimensionality Reduction: Principal Component Analysis (PCA)

- When there are numerous attributes present, they can be linearly combined to form aggregate attributes called Principal Components (PC)

- PCs helps us to bring out the structure of the data and relationships between the attributes
- Suppose, we have a dataset consisting of attributes:
  - $a_1$  to  $a_n$
- It's possible to create new attributes  $PC_1$  to  $PC_m$  where  $m$  is significantly less than  $n$ :

$$PC_1 = c_{11}a_1 + c_{12}a_2 + \dots + c_{1n}a_n$$

$$PC_2 = c_{21}a_1 + c_{22}a_2 + \dots + c_{2n}a_n$$

$$\dots$$

$$PC_m = c_{m1}a_1 + c_{m2}a_2 + \dots + c_{mn}a_n$$

## Dimensionality Reduction: Principal Component Analysis (PCA)

- Example:
  - For instance, for the iris data set we can possibly derive the following Principal Components reducing the set to 2 Dimensions that might look like this:
    - $PC_1 = 0.4S_W + 0.6P_W$  – We could call this new feature (Flower Width)
    - $PC_2 = 0.5S_L + 0.5P_L$  – We could call this new feature (Flower Length)
- Essentially, we have created two new aggregate attributes that can replace the original 4 attributes

## Multivariate Linear Regression

- Similar to simple (bivariate) linear regression, but now the regression is dependent on multiple variables
  - $Y = a + b_1X_1 + b_2X_2 + b_3X_3 + \dots + b_nX_n$
- You have single dependent variable regressed upon several independent variables
- Can be useful when several factors affect the dependent variable

## Multivariate Linear Regression

- Assume the example of the house price:
- Given a relevant dataset it can be possible to create a multiple linear regression model which might look like

$$Price = a + b \times area + c \times bedrooms + d \times floors$$

## Conclusions

- Significant investment in collecting/storing relevant data now pays off
- Data analyst jobs are now widespread
- Tools and how to learn them
- Ethical data science

## Obligatory XKCD

- Correlation: <https://xkcd.com/552/> (<https://xkcd.com/552/>)
- Linear regression: <https://www.xkcd.com/1725/> (<https://www.xkcd.com/1725/>)
- Curve fitting: <https://xkcd.com/2048/> (<https://xkcd.com/2048/>)



# IS5102

# Database Management Systems

## Lecture 20: Knowledge Review

Alexander Konovalov

[alexander.konovalov@st-andrews.ac.uk](mailto:alexander.konovalov@st-andrews.ac.uk)

2021



**Logistic arrangements will be announced by the Exams officer and/or level coordinator**

Answer **3** out of **3** exam-style questions as you would in an exam (citations of sources are not expected; answers should be from your own memory and understanding; significant stretches of text should not be taken verbatim from sources).

Any illustrations or diagrams you include should be original (hand or computer drawn).

You may typeset your answers, or write by hand and scan them.

In either case, please return your answers as a **single PDF**.

If you handwrite, make sure the pages are legible, the right way up and in the right order.

Minor syntax errors in **SQL code** will not be penalised, provided it is clear what you meant.

Your submission should be **your own unaided work** (while you are encouraged to work with your peers to understand the content of the course while revising, once you have seen the questions you should avoid any further discussion until you have submitted your results).

You must submit your completed assessment within the given time window.

Assuming you have revised the module contents beforehand, answering the questions should take no more than **three hours**.

**Good Luck!**

- ▶ Lectures
- ▶ Exercises
- ▶ Coursework

## Lecture 1: Why Database Management Systems?

## Lecture 2: Database Design:

- ▶ Conceptual models
- ▶ Logical models

## Lectures 2-4:

Entities, relations, attributes

Cardinality and participation constraints

Primary keys, super keys, candidate keys

Weak entities, partial keys (discriminators)

Specialisation and generalisation (overlapping/disjoint, partial/total)

Drawing conventions

## Lectures 5-6:

Relations, tuples, attributes

Keys: candidate, primary, foreign

Domains and Attribute Types

Relation schema

## Lecture 6:

Translating E-R diagram to schema diagram

Relational algebra and its operators

## Lectures 7-12:

CREATE/ALTER/DROP TABLE statements and datatypes

INSERT and UPDATE

CASE and LIKE

PRIMARY KEY and FOREIGN KEY

SELECT queries, with WHERE, ORDER BY, GROUP BY, HAVING

Aggregation: AVG, MIN, MAX, SUM, COUNT

## Lectures 7-12:

Cascading actions

Creating and using views

Joins: natural, outer

Functions, procedures, triggers

Authorisation: **GRANT** and **REVOKE**

## Lectures 13-14:

Update Anomalies

Functional Dependencies

Full Functional Dependencies and Transitive Functional Dependencies

1st–3rd Normal Forms and Processes of Normalisation/Denormalisation

## Lectures 15-20:

Data Warehousing: Why and How

Dimensional Modeling

The rise of NoSQL

Graph databases, Key-Value databases, Document-oriented databases

Uses in data mining/analytics

Good luck!