Introduction to Databases
Lectures 1 - 4

David J. Greaves

(with grateful thanks to Timothy G. Griffin)

Computer Laboratory
University of Cambridge, UK

Michaelmas Term, 2022-23

Lecture 1

- What is a Database Management System (DBMS)?
- In other words: what do we need beyond storing some data?
- We'll concentrate on the service provided - no implementation details.
- The diverse landscape of database systems.
  - Traditional SQL-based systems
  - Recent development of "NoSQL" systems.
- Three data models covered in this course
  - Relational,
  - Document-oriented,
  - Graph-oriented.
- Trade-offs imply that no one model ideally solves all problems.

Fields, records and CSV Data

Punched cards were used for weaving control in the Jacuqard Loom and were an inspiration for Hollerith in the 1890 US census, leading to the 80-column punched card.

Fixed-field record

```
Adam           Jonathan    Alexander    Hawkes          M20051969
David          James                    Greaves         M28111962
Peter          James                    Greaves         M28111932
Elizabeth      Jane        Yeti         Goosecreature   F02041965
```

Fixed-field used widely on punched cards and remains efficient for gender and DoB etc..

Comma/character-separated value record

```
Adam,Jonathan,Alexander,Hawkes,M,20,05,1969
David,James,,Greaves,M,28,11,1962
Peter,James,,Greaves,M,28,11,1932
Elizabeth,Jane,Yeti,Goosecreature,F,2,4,1965
```

But how to store Charles Philip Arthur George Mountbatten-Windsor?

A simple, in-core associative store (dictionary/collection)

Implementation in ML – Irrelevant (and not lectured yet!)

```
let m_stored:((string * string) list ref) = ref []   // The internal representation

let store (k, v) = m_stored := (k, v) :: !m_stored   // Function to store a value under
                                                     // a given key.

let retrieve k =                                     // Function to find the value stored
   let rec scan = function                           // under a given key or else
   | []          -> None                             // return 'None'.
   | (h, v)::tt -> if h=k then Some v else scan tt
   in scan !m_stored
```

API formal specification – Relevant to this course.

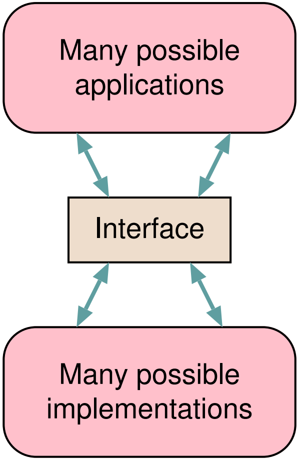
```
store     :  string * string -> unit
retrieve :  string -> string option
```

- The application program interface (API) is defined by its two methods/functions.
- They may be freely called in any order, so no invocation ordering constraints exist (unlike, eg. 'open . (read|write)* . close').

Further Database Jargon

Value:	often just a character string, but could be a number, date, or even a polygon in a spatial database.
Field:	a place to hold a value, also known as an attribute or column in an RDB (relational database).
Record:	a sequence of fields, also known as a row or a tuple in an RDB.
Schema:	the specification of how data is to be arranged, specifying table and field names and types and some rules of consistency (eg. air pressure field cannot be negative).
Key:	the field or concatenation of fields normally used to locate a record.
Index:	a derived structure providing quick means to find relevant records.
Query:	a retrieve or lookup function, often requiring automated planning.
Update:	a modification of the data, preserving consistency and often implemented as a transaction.
Transaction:	an atomic change of a set of fields with further ACID properties.

Abstractions, interfaces, and implementations



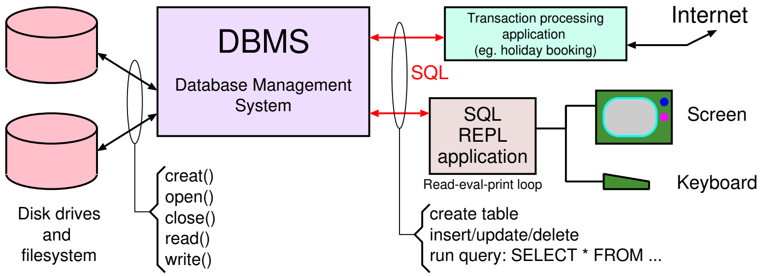
- An interface liberates application writers from low-level details.
- An interface represents an abstraction of resources/services used by applications.
- In a perfect world, implementations can change without requiring changes to applications.
- Performance concerns often challenge this idealised picture.
- ‘Mission-creep’ and specification change typically ruin things too (software misengineering!).

Narrow waist model.

Standard interfaces are everywhere, for example

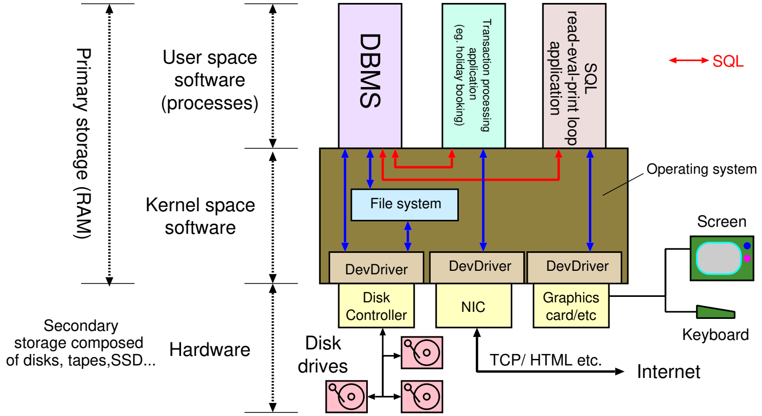
- a national electricity network,
- a landline telephone that's 100 years old can still be plugged in today,
- even money can be thought of as an interface.

Typical Database Logical Arrangement


DBMS
Database Management System
Transaction processing application (eg. holiday booking)
Internet
SQL
SQL REPL application
Read-eval-print loop
Screen
Keyboard
Disk drives and filesystem
creat()
open()
close()
read()
write()
create table
insert/update/delete
run query: SELECT * FROM ...

- The DBMS provides an abstraction over the secondary storage (disks/tapes [web:Video 3b]).
- It hides data storage detail using a narrow, standardised interface (eg. SQL) shared by concurrent applications.

O/S View of the Logical Arrangement



This set-up is covered in the *operating systems* course later in the year, so you need take no notice of this slide today.
In many simple scenarios, the application is in the same process as the DBMS.

A partial specification of computer memory

```
Primary storage (main RAM, typically volatile):
  type address_t = integer 0 to 2^16 - 1
  type word_t = integer 0 to 255
  method write : address_t * word_t -> unit
  method read  : address_t -> word_t

Secondary storage (disk/tape/SSD/USB-stick):
  type blkaddress_t = integer  0 to 2^19-1
  type block_t = array [0..4095] of integer 0 to 255
  method write : blkaddress_t * block_t -> unit
  method read  : blkaddress_t -> block_t option
  method trim (*forget*) : blkaddress_t -> unit
  method sync (*synchronise*)  : unit -> unit
```

Of course, this interface specification says nothing about the semantics of memory, which are basically what you write should be what you read back again! Such a specification needs to take time into account and whether reboot happened in the meantime.

Variations on the previous set-up and otherwise.

Where is the data stored?

- In primary store (in core, on the heap),
- or in secondary store,
- or distributed.

When in-core (in primary/main storage)

- Ephemeral – data lost when program exits,
- Persistent – data serialised to/from the O/S filesystem,
- Persistent – DBMS directly makes access to secondary storage devices.

Data size

- **Big data** – too big to fit in primary store,
- **In-core** – it all fits in (*NB*: ‘core’ is a historic term; today DRAM).

Variations continued ...

Amount of writing

- Read-optimised (data never or rarely changes),
- Transaction-optimised (many concurrent queries and updates),
- Append-only journal (new data always added at the end, ledger style).

Consistency Model – Lecture 5

- Atomic updates (ACID transactions),
- Eventual consistency (BASE).

Data Arrangement

- Relational organisation (tables),
- Semi-structured document (Lecture 6),
- Graph (Lecture 8), or others...

Consistency

Foreign key referential integrity:

Q1: “Mr Sartre, we have your GP down as Dr. Yeti Goosecreature, but we can’t find him/her on our database – do we have the correct spelling of their name?”

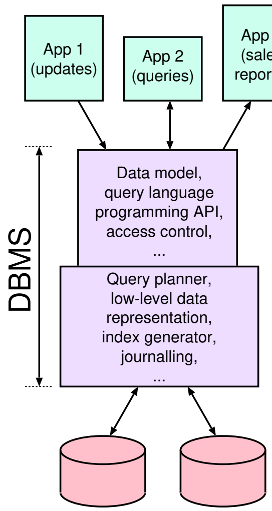
Value range check:

Q2: “Dr. Greaves, we have your weight recorded as minus fifteen kilograms – surely that’s not correct?”

Entity Integrity:

Q3: “Dr. Griffin, we seem to have two home addresses recorded for you – can you clarify?”

This course and the DBMS.



App 1 (updates)
App 2 (queries)
App 3 (sales reports)
DBMS
Data model, query language programming API, access control, ...
Query planner, low-level data representation, index generator, journalling, ...

Physical storage media.

- **This course will present databases from an application writer's point of view. It will stress data models and query languages.**
- We cover how a DBMS can provide a tidy interface to the stored data.
- We will not cover programming APIs or network APIs,
- or cover low-level implementation details,
- or cover how a query engine plans how to service each query.

DBMS operations

CRUD operations:

Create:	Insert new **data** items into the database,
Read:	Query the database,
Update:	Modify objects in the database,
Delete:	Remove data from the database.

Management operations - mostly beyond the scope of this course:

- Create schema (we might do some of this),
- Change schema (Yuck!) (*eg*. add a table or an attribute),
- Create view (*eg.* for access control) (we will be using some views),
- Physical re-organisation of data layout or re-index,
- Backup, stats generation, paying Oracle, etc. ...

This course looks at three data models

Relational Model: Data is stored in tables. SQL is the main query language. Optimised for high throughput of many concurrent updates.

Document-oriented Model: Also called aggregate-oriented database. Optimised for read-oriented databases with few updates and using semi-structured data.

Graph-oriented Model: Much of the data is graph nodes and edges, with extensive support for standard graph techniques. Query languages tend to have ‘path-oriented’ capabilities.

- The relational model has been the industry mainstay for the last 46 years.
- The other two models are representatives of a stuttering revolution in database systems often described under the “NoSQL” banner (Lectures 6&8).
- All three primarily hold discrete data. Lent term course *‘ML & real-world data’* deals with soft/continuous decision making.

This course uses three database systems

HyperSQL	A Java-based relational DBMS. Query language is SQL.
DOCtor Who	A bespoke **doc**ument-oriented collection of data. We'll just use some serialised python dictionaries containing JSON data!
Neo4j	A Java-based graph-oriented DBMS (if we can get it to work). Query language is Cypher (named after a character in The Matrix).

Relational Databases

A relational database consists of a number of 2-D tables. Here is one:

First name	Surname	Weight	GP	GP's age
David	Greaves	-15	Dr Luna	36
Jean-Paul	Sartre	94	Dr Yeti Goosecreature	<null>
Timothy	Griffin	105	Dr Luna	36

- For each table, there is one row per record, technically known as a tuple.
- Each record has a number of fields, technically known as attributes.
- Each table may also have a schema, indicating the field names, allowable data formats/ranges and which column(s) comprise the **key** (underlined).
- The ordering of columns (fields) is unimportant and often so for rows.

[*NB*: A table is called a relation in some textbooks, but we shall see tables represent entities too, so that is a confusing name.].

Distributed databases

Database held over multiple machines or over multiple datacentres.

Why distribute data?

- **Scalability**: The data set or the workload can be too large for a single machine.
- **Fault tolerance**: The service can survive the failure of some machines.
- **Lower Latency**: Data can be located closer to widely distributed users.

Downside of distributed data: consistency

- After an update, there is a massive overhead in providing a consistent view.
- There's a multitude of successively-relaxed consistency models (e.g. all viewers see all updates in the same order or not).
- (Exactly the same problem arise within a single chip for today's multi-core processors.)

Distributed databases pose difficult challenges

CAP concepts

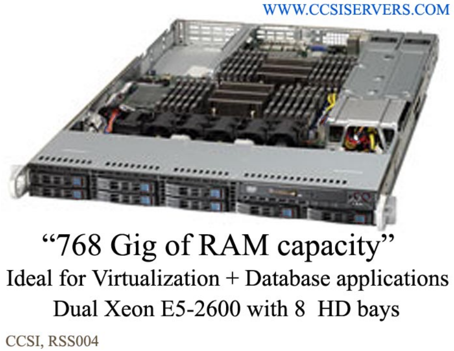
- **Consistency.** All reads return data that is up-to-date.
- **Availability.** All clients can find some replica of the data.
- **Partition tolerance.** The system continues to operate despite arbitrary message loss or failure of part of the system.

- It is impossible, with current (pre-quantum) technology, to achieve the CAP trio in a distributed database.
- Approximating CAP is the subject of the second half of *Ib Concurrency and Distributed Systems* lecture course.
- Alternatively, do not invest much effort. Instead, offer a BASE system with **eventual consistency**: if update activity ceases, then the system will eventually reach a consistent state.

Trade-offs often change as technology changes

Expect more dramatic changes in the coming decades ...

5 megabytes of RAM in 1956


WWW.CCSISERVERS.COM
"768 Gig of RAM capacity"
Ideal for Virtualization + Database applications
Dual Xeon E5-2600 with 8 HD bays
CCSI, RSS004

A modern server

IMDb: Our example data source

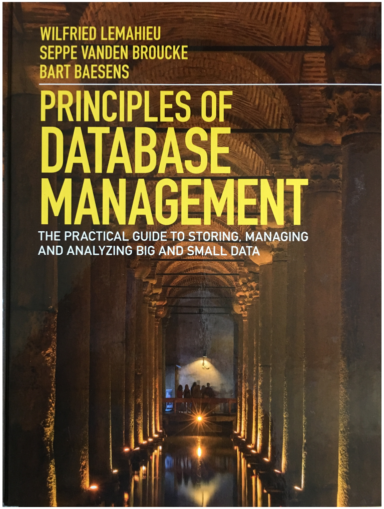
- Raw data available from IMDb plain text data files at `http://www.imdb.com/interfaces`.
- Extracted from this: 1480 movies made between 2000 and 2021 together with 7583 associated people (actors, directors, etc).
- The same data set was used to generate three database instances: relational, graph, and document-oriented.

Course Structure and Timetable

	date	topics
1	11/10	L1 What is a Database Management System (DBMS)?
2	18/10	L2 Entity-Relationship (ER) diagrams
3	25/10	L3 Relational Databases ...
4	1/11	L4 ... and SQL
5	8/11	L5 Redundancy, Consistency & Throughput
6	15/11	L6 Document-oriented Database
	16/11	**Relational DB Help and Tick Session (1)**
7	22/11	L7 Further SQL
	23/11	**Document DB Help and Tick Session (2)**
8	29/11	L8 Graph Database

Get started on the practicals straight after L1.

Recommended Text


WILFRIED LEMAHIEU
SEPPE VANDEN BROUCKE
BART BAESENS
PRINCIPLES OF DATABASE MANAGEMENT
THE PRACTICAL GUIDE TO STORING, MANAGING AND ANALYZING BIG AND SMALL DATA

Lemahieu, W., Broucke, S. van den, and Baesens, B. Principles of database management. Cambridge University Press. (2018)

Guide to relevant material in textbook

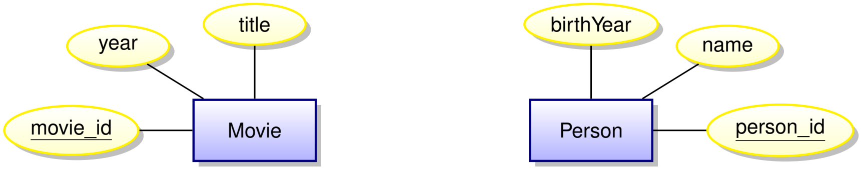
1. What is a Database Management System (DBMS)?
   - Chapter 2
2. Entity-Relationship (ER) diagrams
   - Sections 3.1 and 3.2
3. Relational Databases ...
   - Sections 6.1, 6.2.1, 6.2.2, and 6.3
4. ... and SQL
   - Sections 7.2 – 7.4
5. Indexes. Some limitations of SQL ...
   - 7.5,
6. ... that can be solved with Graph Database
   - Sections 11.1 and 11.5
7. Document-oriented Database
   - Chapter 10

Lecture 2 : Conceptual modelling with Entity-Relationship (ER) diagrams

Peter Chen

- It is very useful to have a **implementation independent** technique to describe the data that we store in a database.
- There are many formalisms for this, and we will use a popular one — Entity-Relationship (ER), due to Peter Chen (1976).
- The ER technique grew up around relational databases systems but it can help document and clarify design issues for any data model.

Entities (should) model **things** in the real world.



- **Entities** (squares) represent the nouns of our model
- **Attributes** (ovals) represent properties
- A **key** is an attribute whose value uniquely identifies an entity instance (here underlined)
- The **scope** of the model is limited — among the vast number of possible attributes that could be associated with a person, we are implicitly declaring that our model is concerned with only three.
- Very abstract, independent of implementation.

Entity Sets (instances)

Instances of the Movie entity

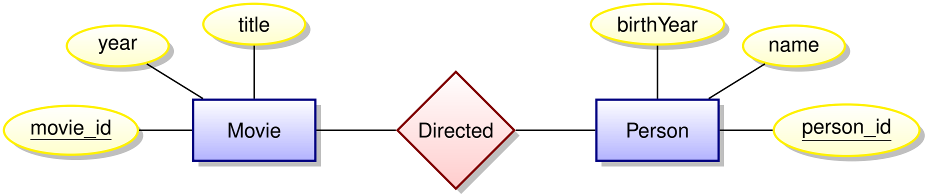
movie_id	title	year
tt1454468	Gravity	2013
tt0440963	The Bourne Ultimatum	2007

Instances of the Person entity

person_id	name	birthYear
nm2225369	Jennifer Lawrence	1990
nm0000354	Matt Damon	1970

- Keys must be unique.
- They might be formed from some algorithm, like your CRSID. Q: Might some domains have **natural keys** (National Insurance ID)? A: Beware of using keys that are out of your control.
- In the real-world, the only safe thing to use as a key is something that is automatically generated in the database and only has meaning within that database.

Relationships



- Relationships (diamonds) represent the verbs of our domain.
- Relationships are between entities.

Relationship instances

Instances of the **Directed** relationship (ignoring entity attributes)

- Kathryn Bigelow directed The Hurt Locker
- Kathryn Bigelow directed Zero Dark Thirty
- Paul Greengrass directed The Bourne Ultimatum
- Steve McQueen directed 12 Years a Slave
- Karen Harley directed Waste Land
- Lucy Walker directed Waste Land
- João Jardim directed Waste Land

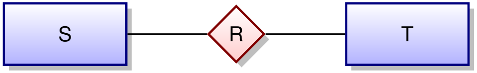
Relationship Cardinality

Directed is an example of a many-to-many relationship.

- Every person can direct multiple movies and every movie can have multiple directors.

A many-to-many relationship

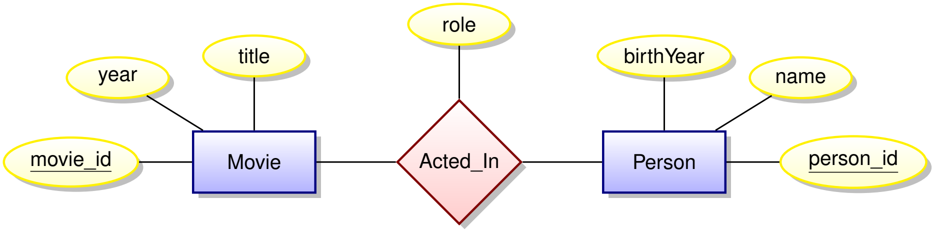
No arrows:



- Any *S* can be related to zero or more *T*'s,
- Any *T* can be related to zero or more *S*'s.
- The relation can also be symmetric and/or relate an entity domain to itself (eg. is_sibling), but these terms have slightly different meanings compared with a mathematical relation.

Crow's foot etc.: There are numerous arrowheads and other diagram annotations for denoting non-symmetric relations and the allowable cardinalities of a relationship. We can mostly leave them out when designing a model since we know what makes sense.

Relationships can also have attributes



Attribute **role** indicates the role played by a person in the movie.

Instances of the relationship **Acted_In**

(ignoring entity attributes)

- Ben Affleck played Tony Mendez in Argo
- Julie Deply played Celine in Before Midnight
- Bradley Cooper played Pat in Silver Linings Playbook
- Jennifer Lawrence played Tiffany in Silver Linings Playbook
- Tim Allan played Buzz Lightyear in Toy Story 3

Have we made a modelling mistake?

- Attributes exist at-most once for any entity or relation.
- So our model is restrictive in that an actor plays a single role in every movie. *This may not always be the case!*

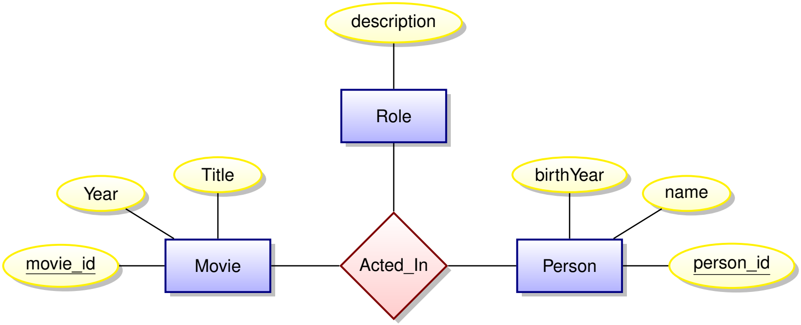
- Jennifer Lawrence played Raven in X-Men: First Class
- Jennifer Lawrence played Mystique in X-Men: First Class
- Scarlett Johansson played Black Widow in The Avengers
- Scarlett Johansson played Natasha Romanoff in The Avengers

So could we allow the role to be a comma-separated list of roles — a **multi-valued attribute** (but not a **composite attribute**)?

- More-than-likely we'll need to break up that list at some point in the future.
- Perhaps fair enough to do this in an E/R design model,
- But when stored in a real database, text processing at that level is an unspeakable data modelleing sin (it violates the rule of **value atomicity**).

Acted_In can be modelled as a Ternary Relationship

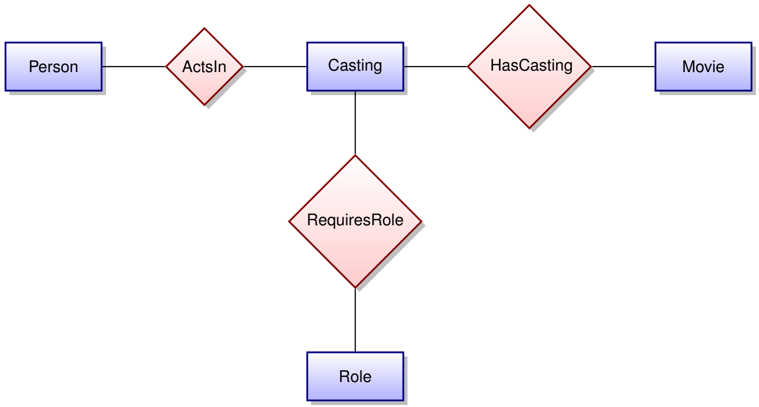
Let’s consider having ‘role’ as an entity.



Acted_In is now a ternary relationship, but

- is a role a real-world entity in its own right,
- and are ternary relations sensible?

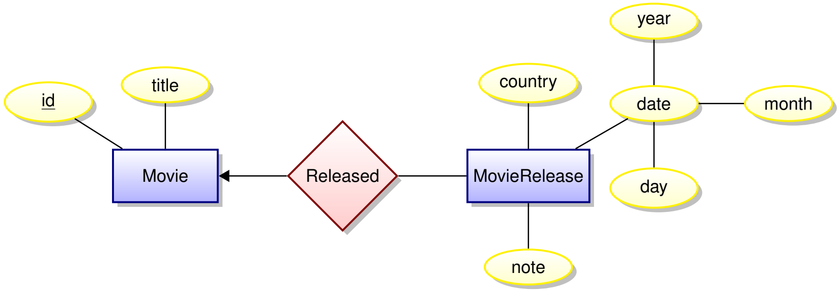
Can a ternary relationship be modelled with multiple binary relationships?



Yes, but is the **Casting** entity too artificial? [Let's hold a referendum.]

[*NB*: See textbook 3.2.6 (pen example) consequent data loss.]

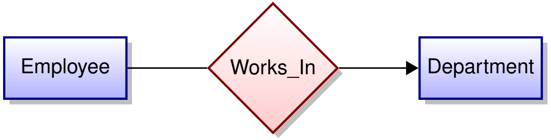
Attribute or entity with new relationship?



- Should the release date be a **composite attribute** or an entity?
- The answer may depend on the **scope** of your data model.
- If all movies within your scope have at most one release date, then an attribute will work well.
- However, if you scope is global, then a movie can have different release dates in different countries.
- Is the **MovieRelease** entity too artificial?

Many-to-one relationships

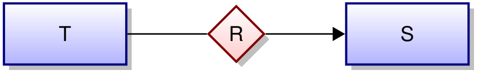
Suppose that every employee is related to at most one department. We are going to denote with an arrow:



- Does our movie database have any many-to-one relationships?
- Do we need some annotation to indicate that every employee must be assigned to a department?

One-to-many, many-to-one and one-to-one.

Suppose every member of *T* is related to at most one member of *S*. We will draw this as

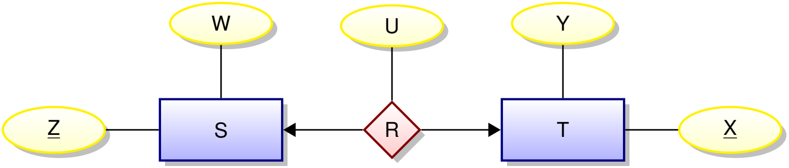


The relation *R* is **many-to-one** between *T* and *S*

The relation *R* is **one-to-many** between *S* and *T*

On the other hand, if *R* is both **many-to-one** between *S* and *T* and **one-to-many** between *S* and *T*, then it is **one-to-one** between *S* and *T*. We'll see two arrows. (These seldom occur in reality – why?)

A “one-to-one cardinality” does not mean a "1-to-1 correspondence”



This database instance is OK

S

Z	**W**
z_1	w_1
z_2	w_2
z_3	w_3

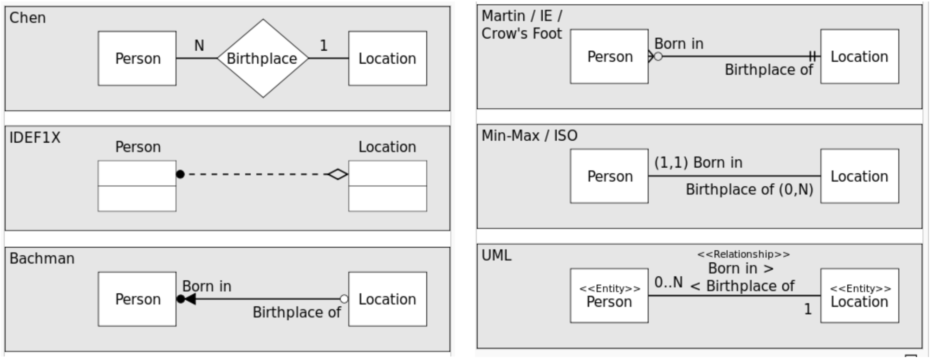
R

Z	**X**	U
z_1	x_2	u_1

T

X	**Y**
x_1	y_1
x_2	y_2
x_3	y_3
x_4	y_4

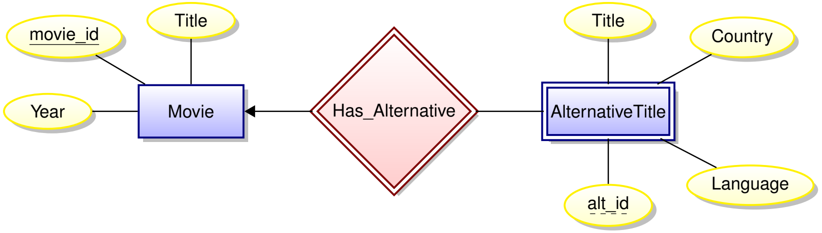
Diagrams can be annotated with cardinalities in many strange and wonderful ways ...



Various diagrammatic notations used to indicate a one-to-many relationship [Wikipedia: E/R model].

[*NB*: None of these detailed notations is examinable, but the concept of a relationship's cardinality is important.]

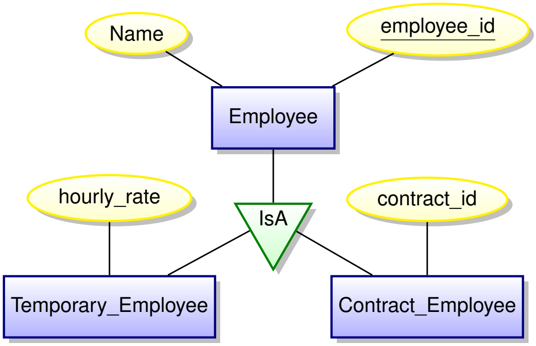
Weak entities



- AlternativeTitle is an example of a **weak entity**
- The attribute alt_id is called a **discriminator**.
- The existence of a weak entity depends on the existence of another entity. In this case, an AlternativeTitle exists only in relation to an existing movie. (This is what makes **MovieRelease** special!)
- Discriminators are not keys. To uniquely identify an AlternativeTitle, we need both a **movie_id** and an **alt_id**.

Entity hierarchy (OO-like)

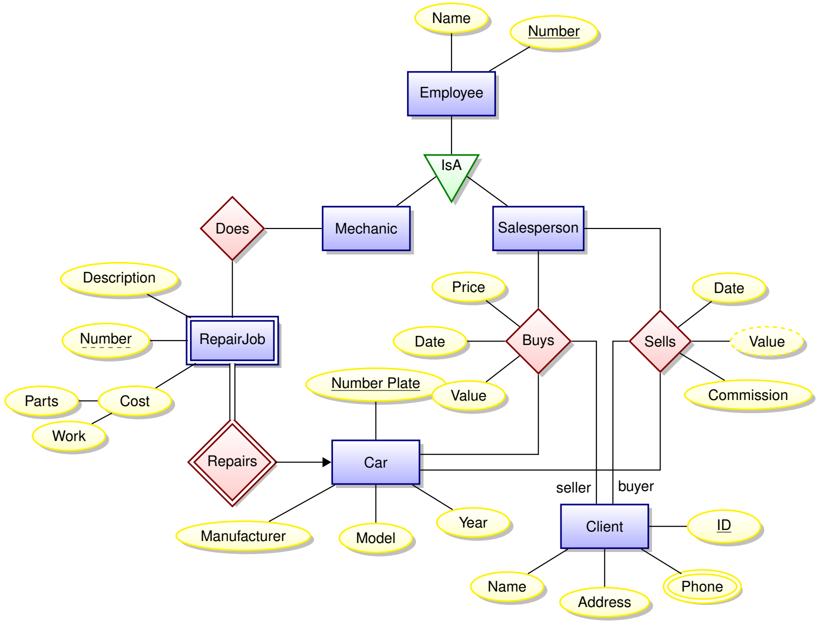
Sometimes an entity can have “sub-entities”. Here is an example:



Sub-entities inherit the attributes (including keys) and relationships of the parent entity. [Multiple inheritance is also possible.]

E/R Diagram Summary

- Forces you to think clearly about the model you want to implement in a database without going into database-specific details.
- Simple diagrammatic documentation.
- Easy to learn.
- Can teach it to techno-phobic clients in less than an hour.
- **Very valuable in developing a model in collaboration with clients who know nothing about database implementation details.**
- With the following slide, imagine you are a data modeller working with a car sales/repair company. The diagram represents your current draft data model. What questions might you ask your client in order to refine this model?



Example due to Pável Calado, author of the tikz-er2.sty package.

Lectures 3 and 4 - The Relational Database

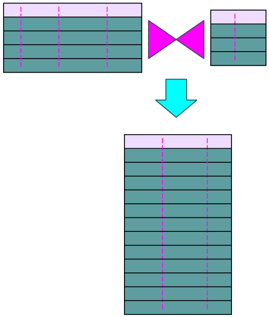
Lecture 3

- The relational model,
- SQL and the relational algebra (RA).

Lecture 4

- Representing an E/R model,
- Update anomalies,
- Avoid redundancy.

The dominant approach: Relational DBMSs



- In the 1970s you could not write a database application without knowing a great deal about the data's low-level representation.
- Codd's radical idea: give users a model of data and a language for manipulating that data which is completely independent of the details of its representation/implementation. That model is based on **mathematical relations**.
- This decouples development of the DBMS from the development of database applications.

Let's start with mathematical relations

Suppose that S and T are sets. The Cartesian product, $S \times T$, is the set

$$S \times T = \{(s,\ t) \mid s \in S,\ t \in T\}$$

EG: $\{A, B\} \times \{3, 4, 5\} = \{(A, 3), (A, 4), (A, 5), (B, 3), (B, 4), (B, 5)\}$

A (binary) relation over $S \times T$ is any set R with

$$R \subseteq S \times T.$$

Database parlance

- S and T are referred to as **domains.**
- We are interested in **finite relations** R that are explicitly stored.
- (*ie*. We shall not be solving integer linear programming puzzles or the like.)

n-ary relations

If we have *n* sets (domains),

$$S_1,\ S_2,\ \ldots,S_n,$$

then an *n*-ary relation R is a set

$$R \subseteq S_1 \times S_2 \times \cdots \times S_n = \{(s_1,\ s_2,\ \ldots,s_n) \mid s_i \in S_i\}$$

Tabular presentation

1	2	$\cdots$	n
x	y	$\cdots$	w
u	v	$\cdots$	s
$\vdots$	$\vdots$		$\vdots$
n	m	$\cdots$	k

All data in a relational database is stored in **tables**. However, referring to columns by number can quickly become tedious!

Mathematical vs. database relations

Use named columns

- Associate a name, A_i (called an attribute name) with each domain S_i.
- Instead of tuples, use records — sets of pairs each associating an attribute name A_i with a value in domain S_i.

Column order does not matter

A database relation R is a finite set

$$R \subseteq \{\{(A_1,\ s_1),\ (A_2,\ s_2),\ \ldots,(A_n,\ s_n)\} \mid s_i \in S_i\}$$

We specify R's **schema** as $R(A_1 : S_1,\ A_2 : S_2,\ \cdots\ A_n : S_n)$.

NB: We'll often say 'field name' instead of 'attribute name', Row order often does not matter but sometimes we will sort using **order by**.

Example: One table (a relational instance).

The relational schema for the table:

Students(**name**: string, **sid**: string, **age** : integer)

An instance of this schema:

$$\textbf{Students} = \{ \{(\textbf{sid}, \text{fm21}), (\textbf{name}, \text{Fatima}), (\textbf{age}, 20)\}, \{(\textbf{name}, \text{Eva}), (\textbf{sid}, \text{ev77}), (\textbf{age}, 18)\}, \{(\textbf{age}, 19), (\textbf{name}, \text{James}), (\textbf{sid}, \text{jj25})\} \}$$

Two equivalent renderings of the table:

name	**sid**	**age**
Fatima	fm21	20
Eva	ev77	18
James	jj25	19

sid	**name**	**age**
fm21	Fatima	20
ev77	Eva	18
jj25	James	19

What is a (relational) database query language?

Input : a collection of relation instances

Output : a single relation instance

$$R_1, R_2, \cdots, R_k \implies Q(R_1, R_2, \cdots, R_k)$$

How can we express Q?

In order to meet Codd's goals we want a query language that is high-level and independent of physical data representation.

There are many possibilities ...

[*NB*: RA is primarily used for queries. SQL suports other CRUD aspects that we'll hardly mention.]

The Relational Algebra (RA) abstract syntax

$$
\begin{array}{rclr}
Q & ::= & R & \text{base relation} \\
 & | & \sigma_p(Q) & \text{selection} \\
 & | & \pi_{\mathbf{X}}(Q) & \text{projection} \\
 & | & Q \times Q & \text{product} \\
 & | & Q - Q & \text{difference} \\
 & | & Q \cup Q & \text{union} \\
 & | & Q \cap Q & \text{intersection} \\
 & | & \rho_M(Q) & \text{renaming}
\end{array}
$$

- p is a [simple] boolean predicate over attributes values.
- $\mathbf{X} = \{A_1,\ A_2,\ \ldots,\ A_k\}$ is a set of attributes.
- $M = \{A_1 \mapsto B_1,\ \ A_2 \mapsto B_2,\ \ldots,\ A_k \mapsto B_k\}$ is a renaming map.
- A query Q must be **well-formed**: all column names of result are distinct. So in $Q_1 \times Q_2$, the two sub-queries cannot share any column names while in in $Q_1 \cup Q_2$, the two sub-queries must share all column names.

SQL: a **vast** and **evolving** language

- Origins at IBM in early 1970's.
- SQL has grown and grown through many rounds of standardization :
  - ANSI: SQL-86
  - ANSI and ISO : SQL-89, SQL-92, SQL:1999, SQL:2003, SQL:2006, SQL:2008, SQL:2008
- SQL is made up of many sub-languages, including
  - Query Language
  - Data Definition Language
  - System Administration Language
- SQL will inevitably absorb many "NoSQL" features ...

Why talk about the Relational Algebra?

- Due to the RA's simple syntax and semantics, it can often help us better understand complex queries.
- Tradition.
- (The RA lends itself to endlessly amusing Tripos questions.)

Selection operator (σ)

R

A	B	C	D
20	10	0	55
11	10	0	7
4	99	17	2
77	25	4	0

$\Longrightarrow$

$Q(R)$

A	B	C	D
20	10	0	55
77	25	4	0

Q

RA $\sigma_{A>12}(R)$

SQL `SELECT DISTINCT * FROM R WHERE R.A > 12`

[*NB*: Asterisk denotes all fields, so no projection going on.]

Projection operator (π)

R

A	B	C	D
20	10	0	55
11	10	0	7
4	99	17	2
77	25	4	0

$\Longrightarrow$

$Q(R)$

B	C
10	0
99	17
25	4

Q

RA $\pi_{B,C}(R)$

SQL `SELECT DISTINCT B, C FROM R`

[*NB*: No 'where' clause, so no selection going on, despite the 'SELECT'.]

Renaming operator (ρ)

R

A	B	C	D
20	10	0	55
11	10	0	7
4	99	17	2
77	25	4	0

$\Longrightarrow$

$Q(R)$

A	E	C	F
20	10	0	55
11	10	0	7
4	99	17	2
77	25	4	0

Q

RA $\rho_{\{B \mapsto E,\ D \mapsto F\}}(R)$

SQL
```
SELECT A, B AS E, C, D AS F FROM R
```

[*NB*: SQL implements renaming with the 'AS' keyword.]

Union operator (∪)

R

A	B
20	10
11	10
4	99

S

A	B
20	10
77	1000

$\Longrightarrow$

$Q(R, S)$

A	B
20	10
11	10
4	99
77	1000

Q

RA $R \cup S$

SQL `(SELECT * FROM R) UNION (SELECT * FROM S)`

[*NB*: This is union of records. We'll also use/abuse ∪ for field concatenation in another slide.]

Intersection operator ($\cap$)

R

A	B
20	10
11	10
4	99

S

A	B
20	10
77	1000

$\Longrightarrow$

$Q(R)$

A	B
20	10

Q

RA $R \cap S$

SQL `(SELECT * FROM R) INTERSECT (SELECT * FROM S)`

Difference operator (-)

R

A	B
20	10
11	10
4	99

S

A	B
20	10
77	1000

$\Longrightarrow$

$Q(R)$

A	B
11	10
4	99

Q

RA $R - S$

SQL `(SELECT * FROM R) EXCEPT (SELECT * FROM S)`

Product operator ($\times$)

R

A	B
20	10
11	10
4	99

S

C	D
14	99
77	100

$\Longrightarrow$

$Q(R,\ S)$

A	B	C	D
20	10	14	99
20	10	77	100
11	10	14	99
11	10	77	100
4	99	14	99
4	99	77	100

Q

RA $R \times S$

SQL `SELECT A, B, C, D FROM R CROSS JOIN S`

SQL `SELECT A, B, C, D FROM R, S`

[*NB*: The RA product is not precisely the mathematical Cartesian product which would return pairs of tuples.]

Natural Join (augmented $\times$)

First, some bits of notation:

- We will often ignore domain types and write a relational schema as $R(\mathbf{A})$, where $\mathbf{A} = \{A_1,\ A_2,\ \cdots,\ A_n\}$ is a set of attribute names.
- When we write $R(\mathbf{A},\ \mathbf{B})$ we mean $R(\mathbf{A} \cup \mathbf{B})$ and implicitly assume that $\mathbf{A} \cap \mathbf{B} = \phi$ (*ie.* disjoint fields).
- $u.[\mathbf{A}] = v.[\mathbf{A}]$ abbreviates $u.A_1 = v.A_1 \wedge \cdots \wedge u.A_n = v.A_n$.

Natural Join (SQL replace `CROSS` with `NATURAL`):

Given $R(\mathbf{A},\ \mathbf{B})$ and $S(\mathbf{B},\ \mathbf{C})$, we define the natural join, denoted $R \bowtie S$, as a relation over attributes $\mathbf{A}, \mathbf{B}, \mathbf{C}$ defined as

$$R \bowtie S \equiv \{t \mid \exists u \in R,\ v \in S,\ u.[\mathbf{B}] = v.[\mathbf{B}] \wedge t = u.[\mathbf{A}] \cup u.[\mathbf{B}] \cup v.[\mathbf{C}]\}$$

In the Relational Algebra:

$$R \bowtie S = \pi_{\mathbf{A},\mathbf{B},\mathbf{C}}(\sigma_{\mathbf{B}=\mathbf{B}'}(R \times \rho_{\vec{\mathbf{B}} \mapsto \vec{\mathbf{B}}'}(S)))$$

Natural join example

Students

name	sid	cid
Fatima	fm21	cl
Eva	ev77	k
James	jj25	cl

Colleges

cid	cname
k	King's
cl	Clare
q	Queens'

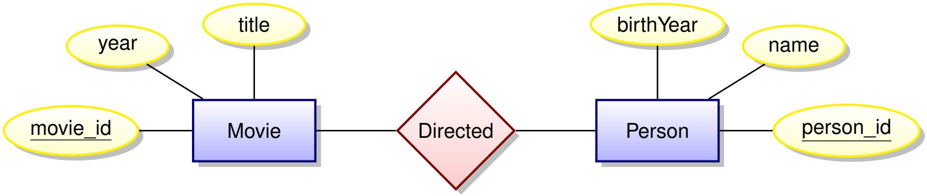
$\Longrightarrow$

Students $\bowtie$ Colleges

name	sid	cid	cname
Fatima	fm21	cl	Clare
Eva	ev77	k	King's
James	jj25	cl	Clare

- Explicit join predicates are commonly used: replace `NATURAL`(=equality) with a `WHERE` clause.
- When NULL values exist, there are further join variations you should know (left/right/inner/outer), but not taught in these slides (Lemahieu 7.3.1.5).

Lecture 4: How can we implement an E/R model relationally?



- The ER model does not dictate implementation.
- There are many options.
- We will discuss some of the trade-offs involved.

Remember, we only have tables to work with!

How about one big table?

DirectedComplete

```
MOVIE_ID    TITLE                          YEAR PERSON_ID NAME             BIRTHYEAR
----------  -----------------------------  ---- --------- ---------------- ---------
tt0126029   Shrek                          2001 nm0011470 Andrew Adamson   1966
tt0126029   Shrek                          2001 nm0421776 Vicky Jenson
tt0181689   Minority Report                2002 nm0000229 Steven Spielberg 1946
tt0212720   A.I. Artificial Intelligence   2001 nm0000229 Steven Spielberg 1946
tt0983193   The Adventures of Tintin       2011 nm0000229 Steven Spielberg 1946
tt4975722   Moonlight                      2016 nm1503575 Barry Jenkins    1979
tt5012394   Maigret Sets a Trap            2016 nm0668887 Ashley Pearce
tt5013056   Dunkirk                        2017 nm0634240 Christopher Nolan 1970
tt5017060   Maigret's Dead Man             2016 nm1113890 Jon East
tt5052448   Get Out                        2017 nm1443502 Jordan Peele     1979
tt5052474   Sicario: Day of the Soldado    2018 nm1356588 Stefano Sollima  1966
.....       .....                          .... .....                      ....
```

What's wrong with this approach?

[Later we'll be asking ourselves, 'What is the key to this table and does all the data stored in it naturally depend on the key?']

Problems with data redundancy

Data consistency anomalies:

Insertion:	How can we tell if a newly-inserted record is consistent with existing records? We may want to insert a person without knowing if they are a director. We might want to insert a movie without knowing its director(s).
Deletion:	We lose information about a Director if we delete all of their films from the table.
Update:	What if a director's name is mis-spelled? We may update it correctly for one film, but not for another.

Performance issue:

- A transaction implementing a conceptually simple update has a lot of work to do,
- it could even end up locking the entire table.

Lesson: In a database supporting many concurrent updates, we see that data redundancy can lead to complex transactions and low write throughput.

A better idea: break tables down in order to reduce redundancy (1)

movies

```
MOVIE_ID     TITLE                                 YEAR
----------   ----------------------------------    ----
tt0126029    Shrek                                 2001
tt0181689    Minority Report                       2002
tt0212720    A.I. Artificial Intelligence          2001
tt0983193    The Adventures of Tintin              2011
tt4975722    Moonlight                             2016
tt5012394    Maigret Sets a Trap                   2016
tt5013056    Dunkirk                               2017
tt5017060    Maigret's Dead Man                    2016
tt5052448    Get Out                               2017
tt5052474    Sicario: Day of the Soldado           2018
.....        .....                                 ....
```

A better idea: break tables down in order to reduce redundancy (2)

people

```
PERSON_ID  NAME                BIRTHYEAR
---------  ----------------    ---------
nm0011470  Andrew Adamson      1966
nm0421776  Vicky Jenson
nm0000229  Steven Spielberg    1946
nm1503575  Barry Jenkins       1979
nm0668887  Ashley Pearce
nm0634240  Christopher Nolan   1970
nm1113890  Jon East
nm1443502  Jordan Peele        1979
nm1356588  Stefano Sollima     1966
.....          .....           ....
```

[Later we'll again ask, 'What is are the keys for out new tables and does all the data stored in a table naturally depend on its key?']

Now use a third table to hold the relationship.

Directed

```
MOVIE_ID     PERSON_ID
----------   ---------
tt0126029    nm0011470
tt0126029    nm0421776
tt0181689    nm0000229
tt0212720    nm0000229
tt0983193    nm0000229
tt4975722    nm1503575
tt5012394    nm0668887
tt5013056    nm0634240
tt5017060    nm1113890
tt5052448    nm1443502
tt5052474    nm1356588
.....        .....
```

What is the key to this table? Is it ‘all key’? Can films now have multiple directors?

Computing DirectedComplete with SQL

```
SELECT movie_id, title, year,
       person_id, name, birthYear
FROM movies
join directed on directed.movie_id = movies_id
join people on people.person_id = person_id
```

Note: the relation **directed** does not exist in our database (more on that later). We have to write something like this:

```
SELECT movie_id, title, year,
       person_id, name, birthyear
FROM movies as m
join has_position as hp on hp.movie_id = m.movie_id
join people as p on p.person_id = hp.person_id
WHERE hp.position = 'director';
```

We can recover all information for the plays_role relation

The SQL query

```
SELECT movies.movie_id AS mid, title, year,
       people.person_id AS pid, name, role
FROM movies
JOIN plays_role ON movies.movie_id = plays_role.movie_id
JOIN people ON people.person_id = plays_role.person_id;
```

might return something like

```
MID         TITLE                YEAR  PID         NAME                 ROLE
----------  -------------------- ----  ----------  -------------------  ------------
tt0118694   In the Mood for Love 2000  nm0504897   Tony Chiu-Wai Leung  Chow Mo-wan
tt0118694   In the Mood for Love 2000  nm0803310   Siu Ping-Lam         Ah Ping
tt0120630   Chicken Run          2000  nm0000154   Mel Gibson           Rocky
tt0120630   Chicken Run          2000  nm0200057   Phil Daniels         Fetcher
tt0120630   Chicken Run          2000  nm0272521   Lynn Ferguson        Mac
tt0120630   Chicken Run          2000  nm0768018   Julia Sawalha        Ginger
tt0120679   Frida                2002  nm0000161   Salma Hayek          Frida Kahlo
tt0120679   Frida                2002  nm0000547   Alfred Molina        Diego Rivera
...         ...                  ...   ...         ...                  ...
```

Observations

- Both ER entities and ER relationships are implemented as tables.
- We call them tables rather than relations to avoid confusion!
- Good: we avoid many update anomalies by breaking tables into smaller tables.
- Bad: we have to work hard to combine information in tables (joins) to produce interesting results.

What about consistency/integrity of our relational implementation?

How can we ensure that the table representing an ER relation really implements a relationship? Answer : we use **keys** and **foreign keys**.

Key: conceptual and formal definitions.

One aspect of a key should already be conceptually clear: a unique handle on a record (table row).

Relational key – a definition from set theory:

Suppose $R(\mathbf{X})$ is a relational schema with $\mathbf{Z} \subseteq \mathbf{X}$. If for any records u and v in any instance of R we have

$$u.[\mathbf{Z}] = v.[\mathbf{Z}] \Longrightarrow u.[\mathbf{X}] = v.[\mathbf{X}],$$

then $\mathbf{Z}$ is a superkey for R. If no proper subset of $\mathbf{Z}$ is a superkey, then $\mathbf{Z}$ is a key for R. We write $R(\underline{\mathbf{Z}}, \mathbf{Y})$ to indicate that $\mathbf{Z}$ is a key for $R(\mathbf{Z} \cup \mathbf{Y})$.

The other aspect (we'll study in L5) is that, in a normalised schema, all row data **semantically depends** on the key.
[*NB*: A table/relation can have multiple keys, in either sense.]

Foreign keys and Referential integrity

Foreign key

Suppose we have $R(\underline{\mathbf{Z}},\ \mathbf{Y})$. Furthermore, let $S(\mathbf{W})$ be a relational schema with $\mathbf{Z} \subseteq \mathbf{W}$. We say that $\mathbf{Z}$ represents a Foreign Key in S for R if for any instance we have $\pi_{\mathbf{Z}}(S) \subseteq \pi_{\mathbf{Z}}(R)$. Think of these as (logical) pointers!

Referential integrity

A database is said to have referential integrity when all foreign key constraints are satisfied.

Q1: "Mr Sartre, we have your GP down as Dr. Yeti Goosecreature, but we can't find him/her on our database – do we have the correct spelling of their name?"

Referential integrity example.

The schema/table

$$Has\_Genre(\underline{movie\_id}, \underline{genre\_id})$$

will have referential integrity constraints

$$\pi_{movie\_id}(Has\_Genre) \subseteq \pi_{movie\_id}(Movies)$$

$$\pi_{genre\_id}(Has\_Genre) \subseteq \pi_{genre\_id}(Genres)$$

[*NB*: **Has_Genre** is said to be 'all key', which is quite common for schemas/tables representing relations.]

Schema and key definitions in SQL.

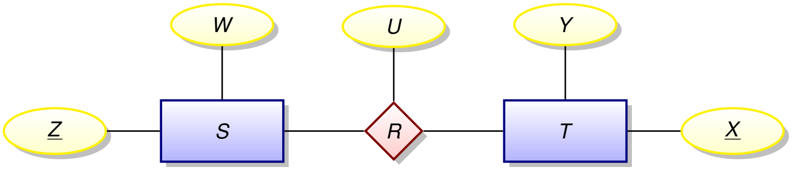
A schema with a simple key:

```
create table genres (
   genre_id integer NOT NULL,
   genre varchar(100) NOT NULL,
   PRIMARY KEY (genre_id));
```

A schema that is all-key and that has two foreign keys:

```
create table has_genre (
   movie_id varchar(16) NOT NULL
      REFERENCES movies (movie_id),
   genre_id integer NOT NULL
      REFERENCES genres (genre_id),
   PRIMARY KEY (movie_id, genre_id));
```

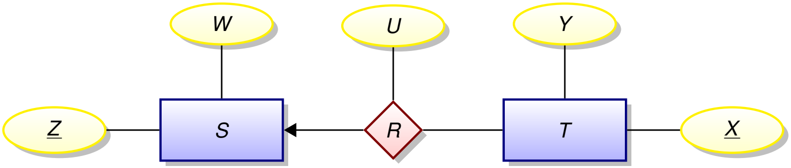
Relationships in tables (the “clean” approach).



Relation R is	Schema
many to many ($M : N$)	$R(\underline{X,\ Z},\ U)$
one to many ($1 : M$)	$R(\underline{X},\ Z,\ U)$
many to one ($M : 1$)	$R(X,\ \underline{Z},\ U)$

[*NB*. Copy out three times and add arrows if you are eager.]

Implementation can differ from the “clean” approach



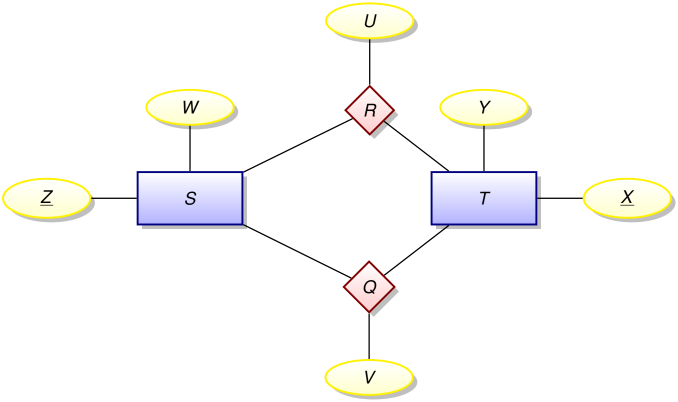
Suppose *R* is one-to-many (reading left to right)

Rather than implementing a new table $R(\underline{X}, Z, U)$ we could expand table $T(\underline{X}, Y)$ to $T(\underline{X}, Y, Z, U)$ and allow the Z and U columns to be NULL for those rows in T not participating in the relationship.

Pros and cons?

Implementing multiple relationships with a single table?

Suppose we have two many-to-many relationships:



Our two relationships are called R and Q.

Implementing multiple relationships with one table is possible.

Rather than using two tables

$$R(\underline{X,\ Z},\ U)$$
$$Q(\underline{X,\ Z},\ V)$$

we might squash them into a single table

$$RQ(\underline{X,\ Z,\ type},\ U,\ V)$$

using a tag $domain(type) = \{\mathbf{r}, \mathbf{q}\}$ (for some constant values r and q).

- represent an R-record (x, z, u) as an RQ-record $(x, z, \mathbf{r}, u, NULL)$
- represent an Q-record (x, z, v) as an RQ-record $(x, z, \mathbf{q}, NULL, v)$

Redundancy alert!

If we know the value of the *type* column, we can compute the value of either the U column or the V column (one must be NULL).

We have stuffed 5 relationships into the `has_position` table!

```
SELECT position, COUNT(*) as total
FROM has_position
GROUP BY position
ORDER BY total DESC;
```

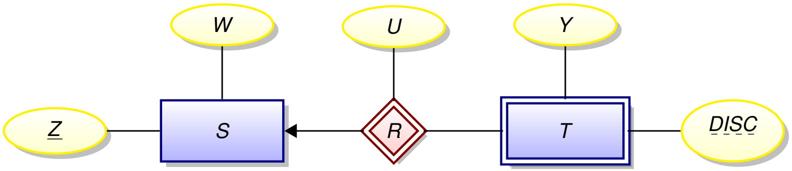
Using our database, this query produces the output

```
POSITION   TOTAL
--------   -----
actor       4950
producer    2300
writer      2215
director    1422
self         293
```

Was this a good idea?

Discuss!

Implementing weak entities.



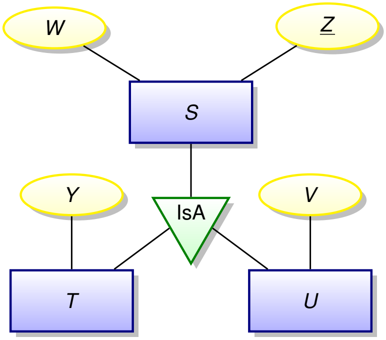
One (clean) approach:

- $S(\underline{Z},\ W)$
- $R(\underline{Z,\ DISC},\ U)$ with $\pi_Z(R) \subseteq \pi_Z(S)$
- $T(\underline{Z,\ DISC},\ Y)$ with $\pi_Z(T) \subseteq \pi_Z(S)$

A more concise (clean) approach:

- $S(\underline{Z},\ W)$
- $R(\underline{Z,\ DISC},\ U,\ Y)$ with $\pi_Z(R) \subseteq \pi_Z(S)$
- This is how **Has_Alternative** is implemented.

A 3-table implementation of entity hierarchy.

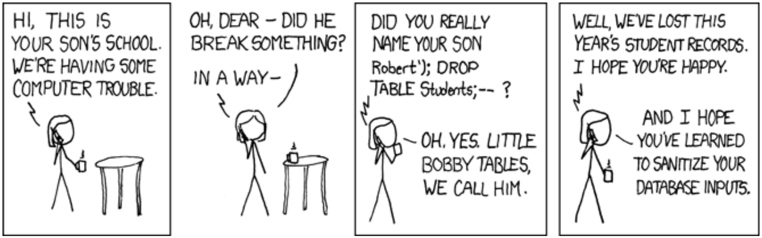


One (clean) approach:

- $S(\underline{Z},\ W)$
- $T(\underline{Z},\ Y)$ with $\pi_Z(T) \subseteq \pi_Z(S)$
- $U(\underline{Z},\ V)$ with $\pi_Z(U) \subseteq \pi_Z(S)$

Could we combine these tables into one with type tags? Yes but unclean. Try it yourself.

End of the first half (L5-8 are a separate file).


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

(`http://xkcd.com/327`)