Introduction to Databases
Lectures 1 - 8

David J. Greaves

with grateful thanks to James Sharkey and Tim Griffin

Computer Laboratory
University of Cambridge, UK

Michaelmas Term, 2023-24

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.

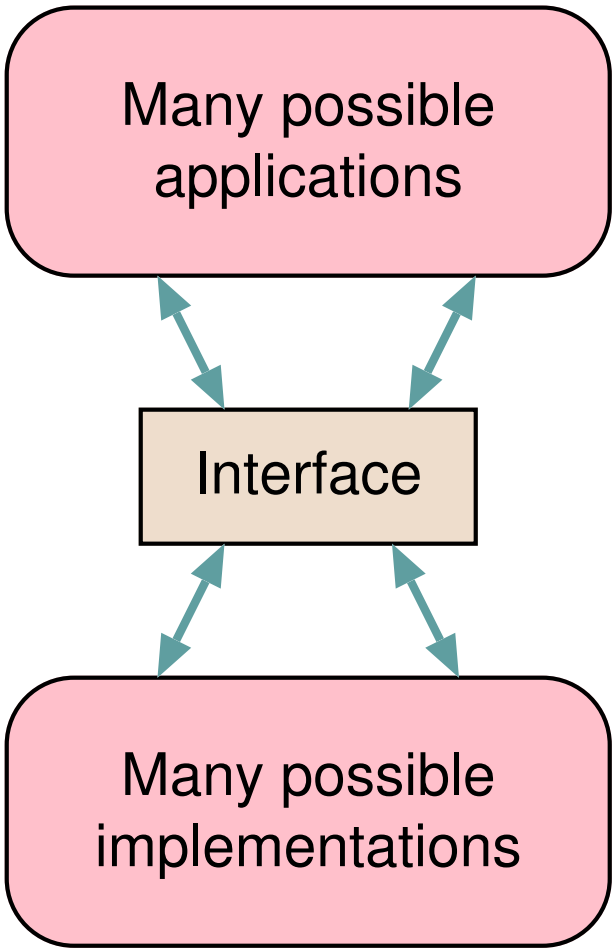
Index: a derived structure providing quick means to find relevant records.

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.

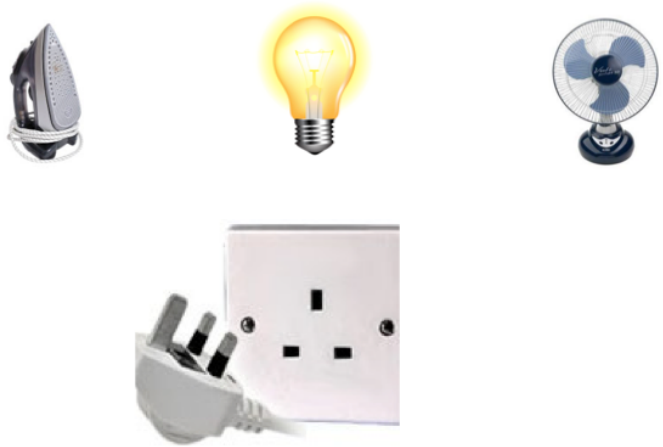
Abstractions, interfaces, and implementations



- 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!).

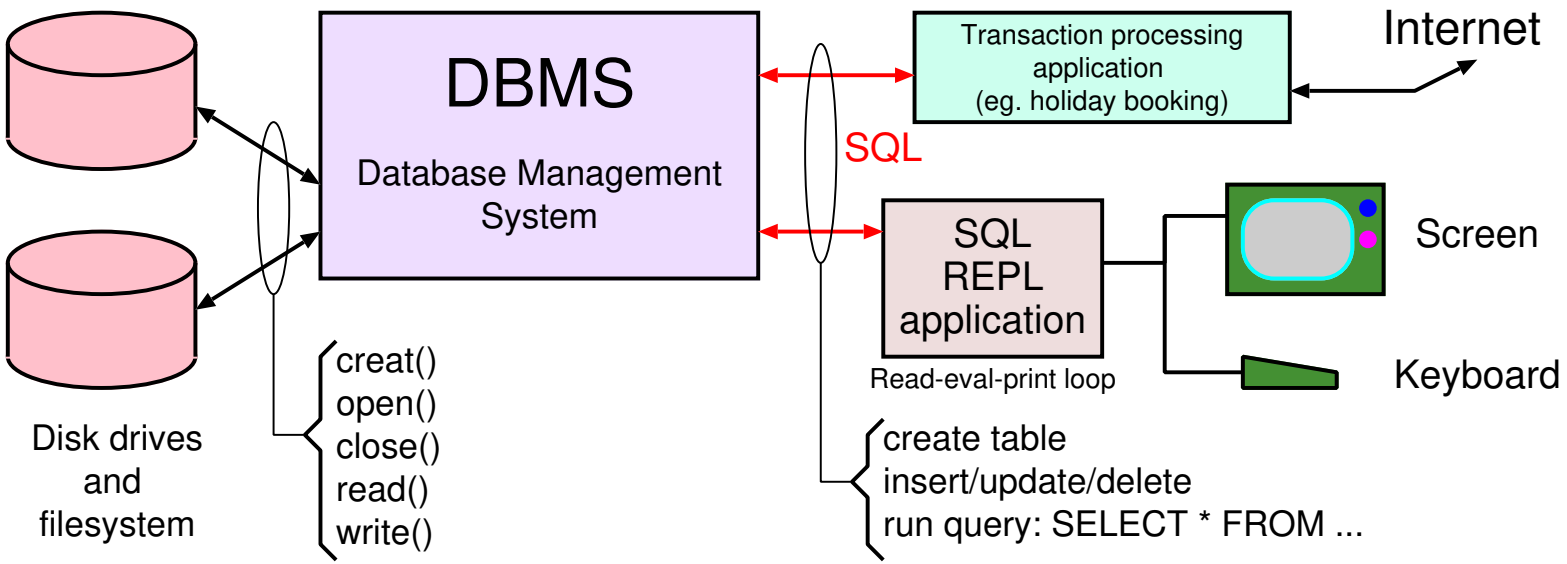
Narrow waist model.

Standard interfaces are everywhere, for example



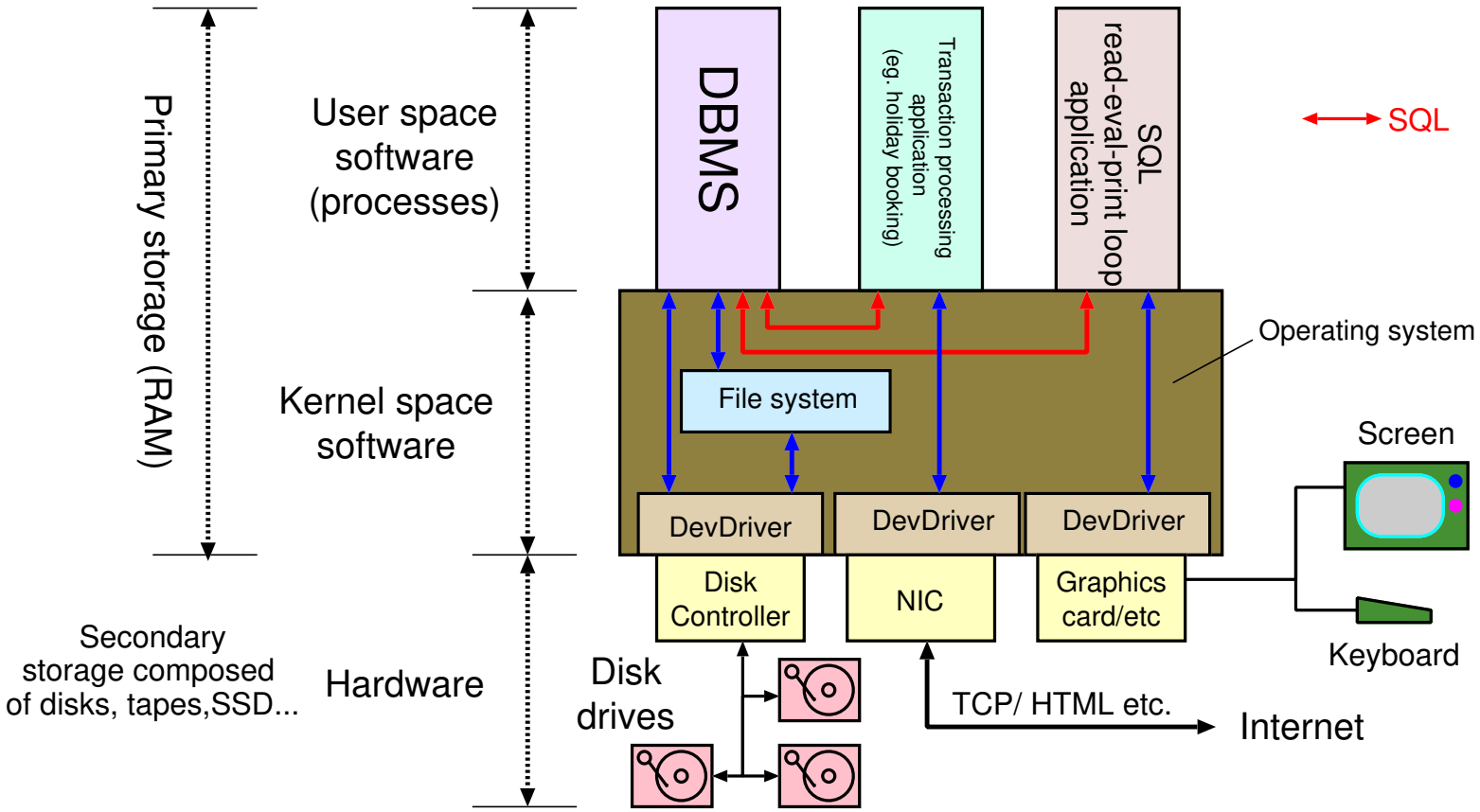
- a national electricity network,
- a landline telephone that's 100 years old can still be plugged in today,
- even money can be thought of as an interface.

Typical Database Logical Arrangement



- 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 little 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 (ie. 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

Value range check:

Q1: "Dr. Greaves, we have your weight recorded as minus fifteen kilograms – surely that's not correct?"

Foreign key referential integrity:

Q2: "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?"

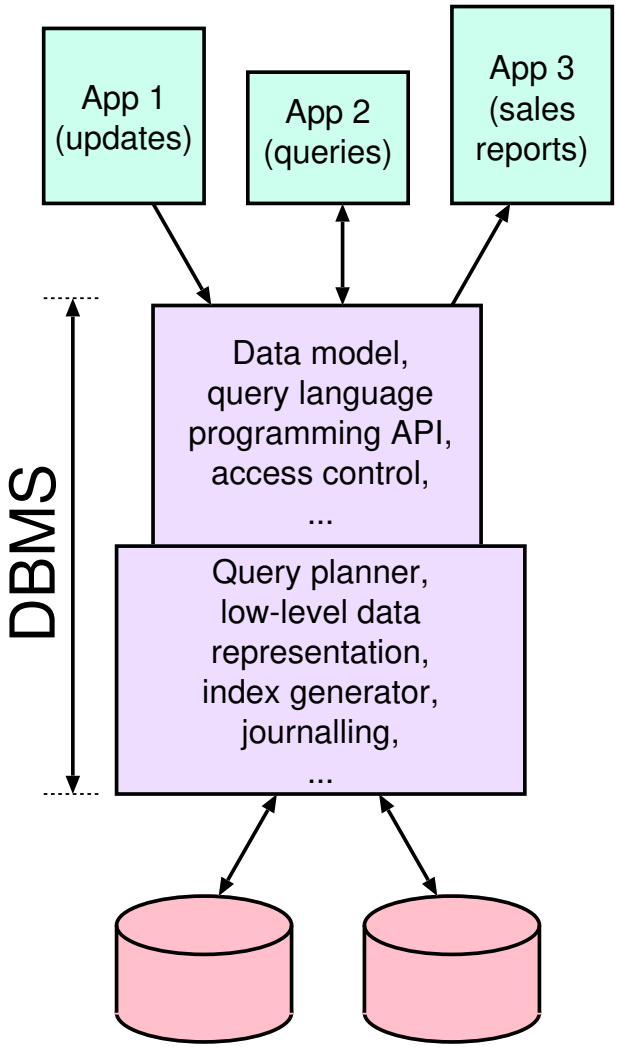
Value Atomicity:

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

Entity Integrity:

Q4: "Ms. du Pré, the flight is ready to board, but your cello has no passport number, so I'm afraid it cannot take its seat.

This course and the DBMS.



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 detail 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 47 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

SQLite An open-source, simple relational DBMS. Query language is SQL.

TinyDB An open-source, document-oriented DBMS coded and queried in Python.

Neo4j A Java-based graph-oriented DBMS — the query language is Cypher (named after a character in The Matrix).

For examination purposes you are expected to learn everything in this slide deck (unless specifically marked unexaminable). Also, you must learn in-detail, a core subset of SQL. For tick 2 you will use TinyDB.

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

A modern server

IMDb: Our example data source

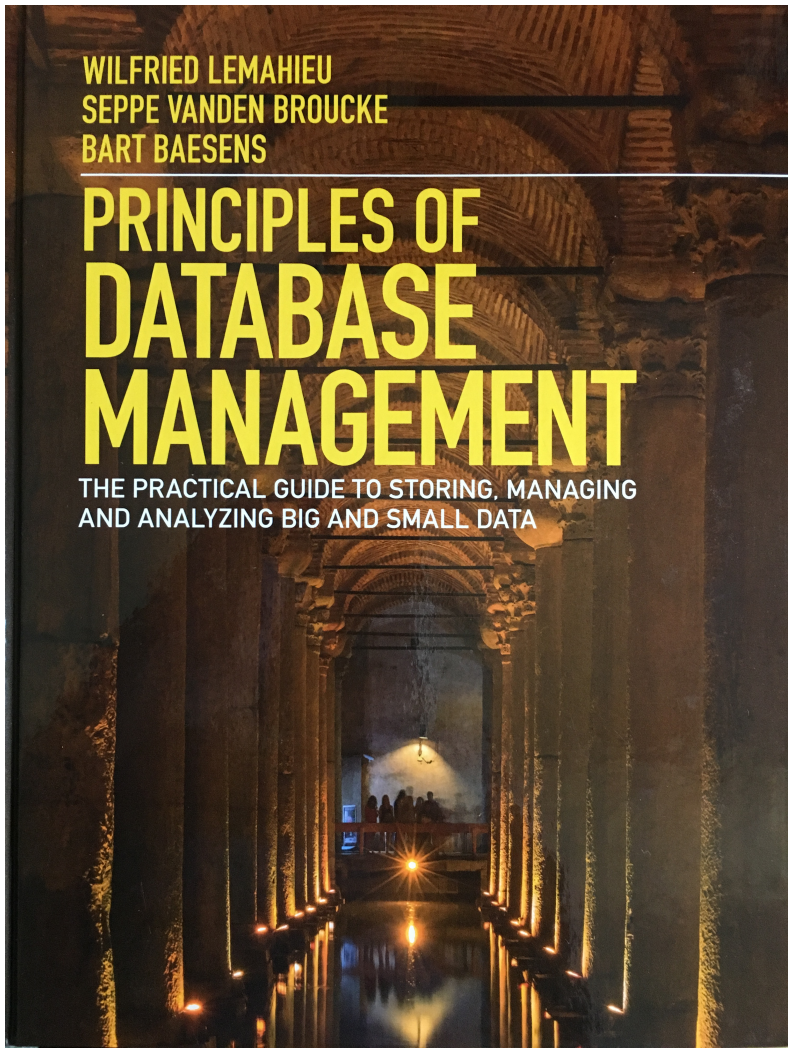
- Raw data available from IMDb plain text data files at `http://www.imdb.com/interfaces`.
- Extracted from this: 1489 movies made between 1921 and 2023 together with 7348 associated people.
- The same data set was used to generate three database instances: relational, graph, and document-oriented.

Course Structure and Timetable 2023

	date	topics
1	10/10	L1 What is a Database Management System (DBMS)?
2	17/10	L2 Entity-Relationship (ER) diagrams
3	24/10	L3 Relational Databases ...
4	31/10	L4 ... and SQL
5	7/11	L5 Redundancy, Consistency & Throughput
6	14/11	L6 Document-oriented Database
7	21/11	L7 Further SQL
8	28/11	L8 Graph Database

Tick deadlines are 16th and 23d Nov 2023. Help sessions may be organised a few days before each deadline. Note lecture rooms and times do change this year. Get started on the practicals straight after L1.

Recommended Text



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 and Secion 11.3

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

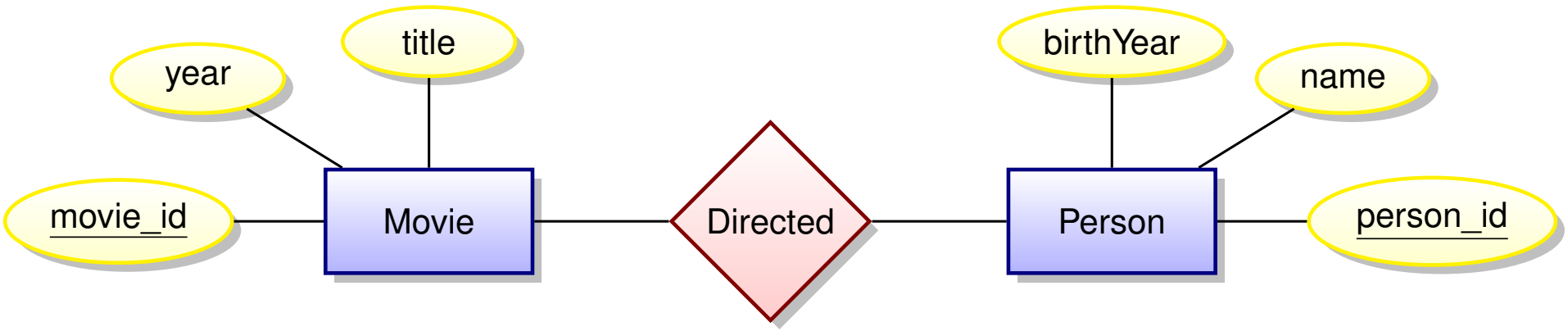
movie_id	title	year
tt1454468	Gravity	2013
tt0440963	The Bourne Ultimatum	2007

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 a **synthetic key** 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

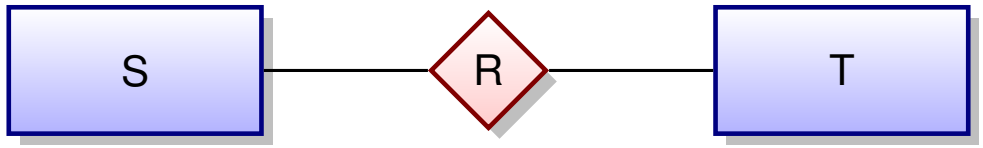
Relationship Cardinality

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

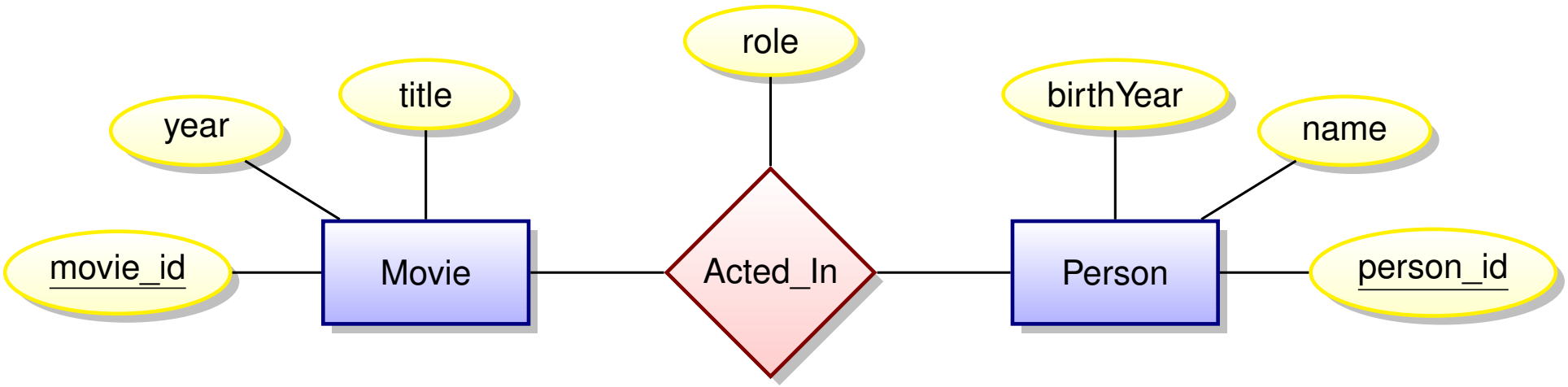
No arrows:



- 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!*

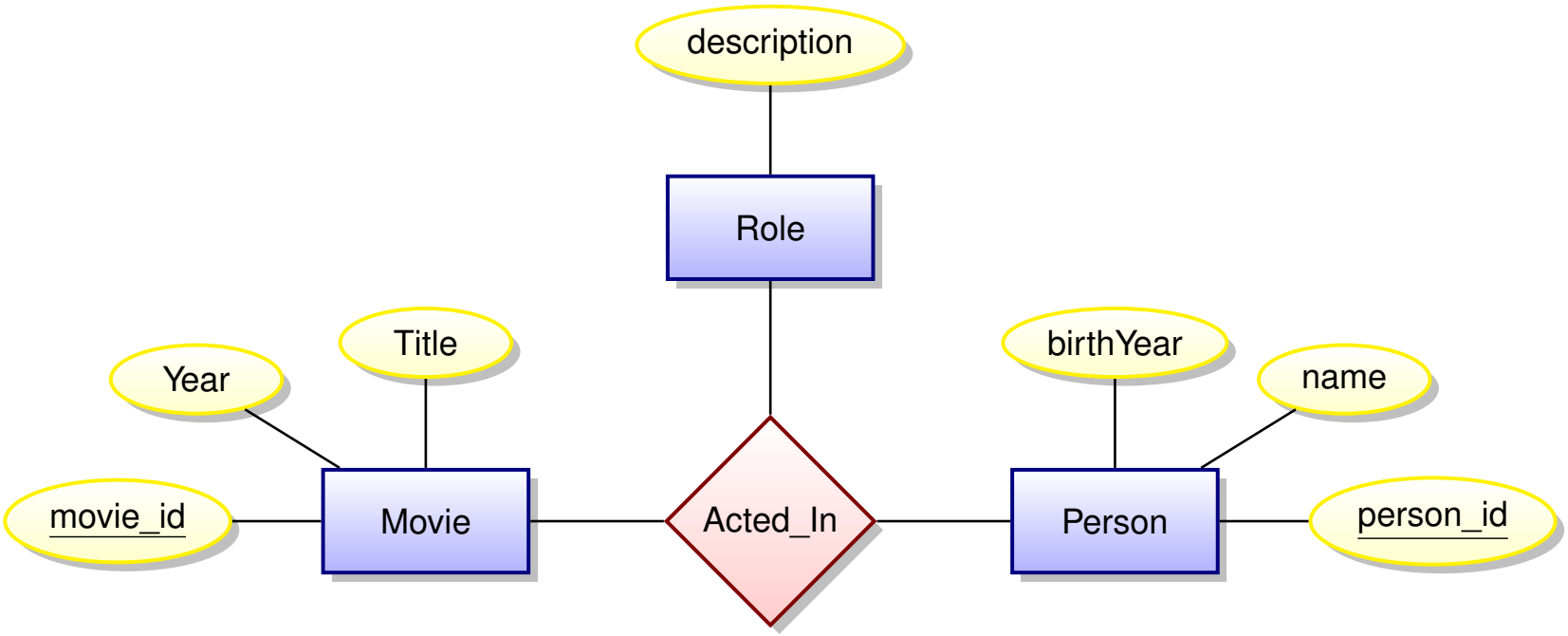
- Jennifer Lawrence played Raven in X-Men: First Class
- Jennifer Lawrence played Mystique in X-Men: First Class
- Scarlett Johansson played Black Widow in The Avengers
- Scarlett Johansson played Natasha Romanoff in The Avengers

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 modelling sin (it violates the rule of **value atomicity**).

Acted_In can be modelled as a Ternary Relationship

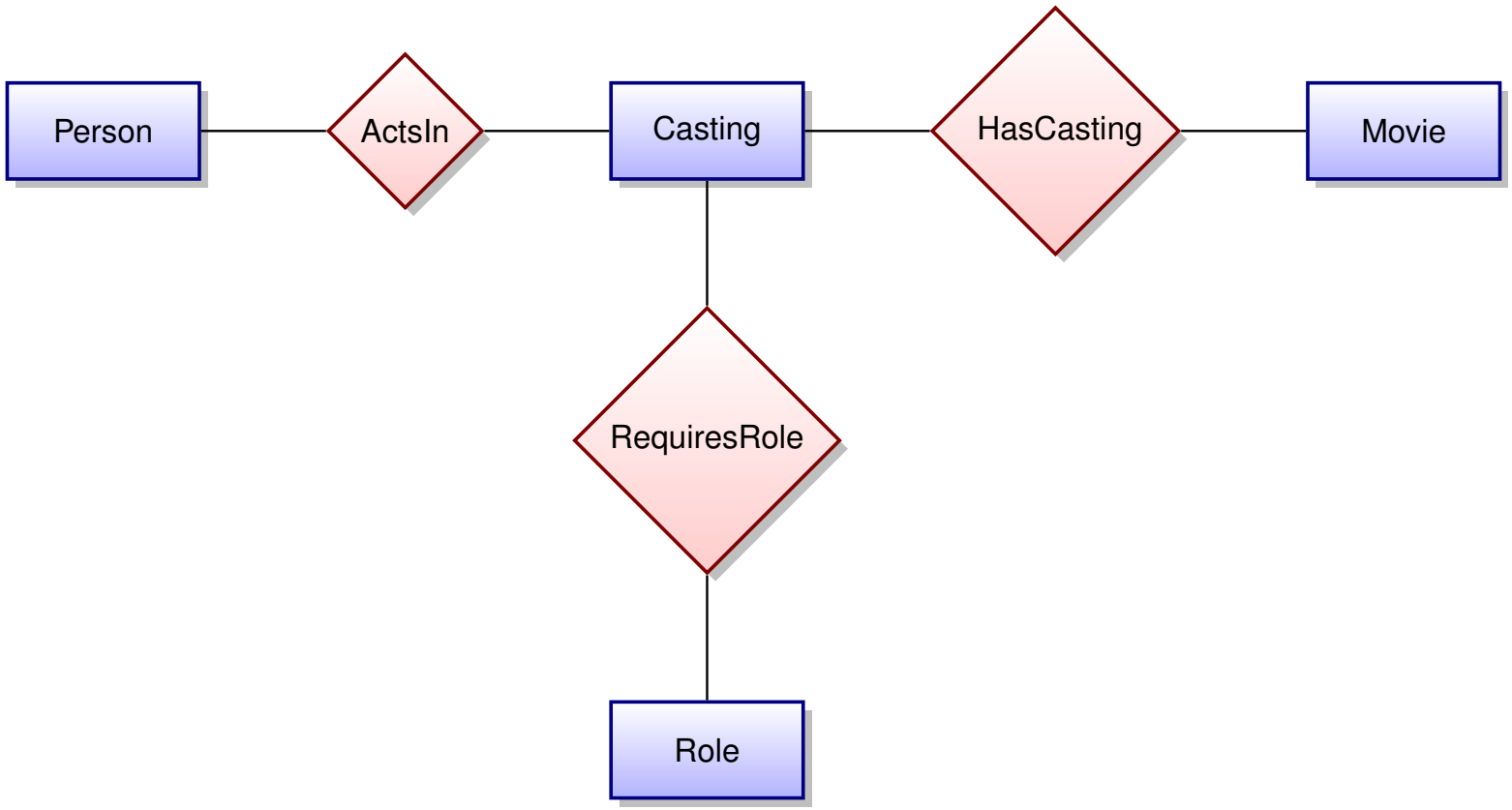
Let's consider having 'role' as an entity.



Acted_In is now a ternary relationship, but

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

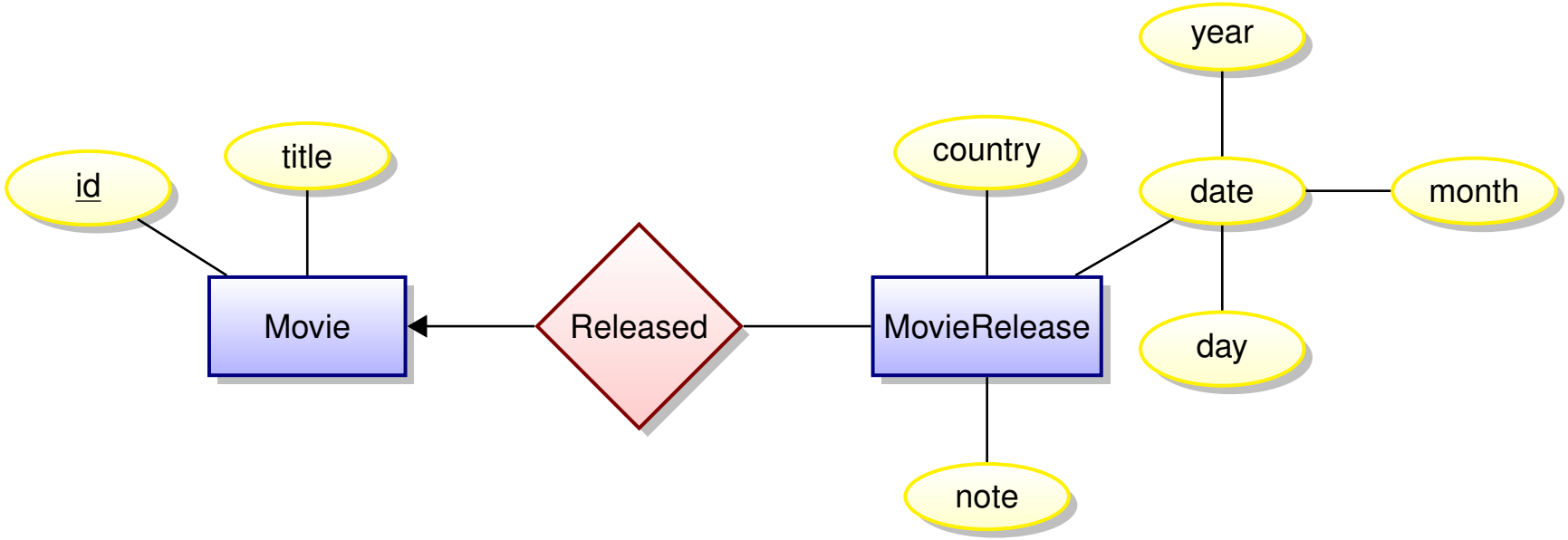
Can a ternary relationship be modelled with multiple binary relationships?



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

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

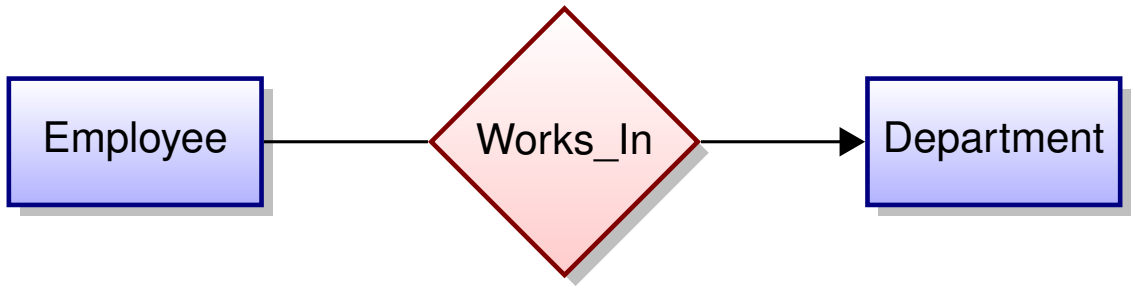
Attribute or entity with new relationship?



- 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

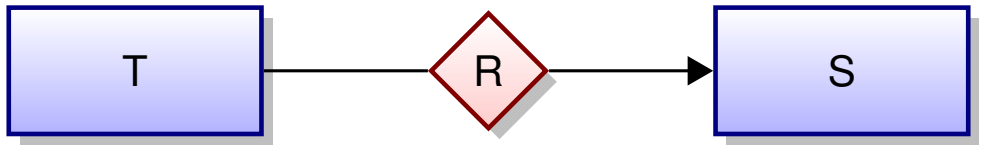
Suppose that every employee is related to at most one department. We are going to denote with an arrow:



- 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

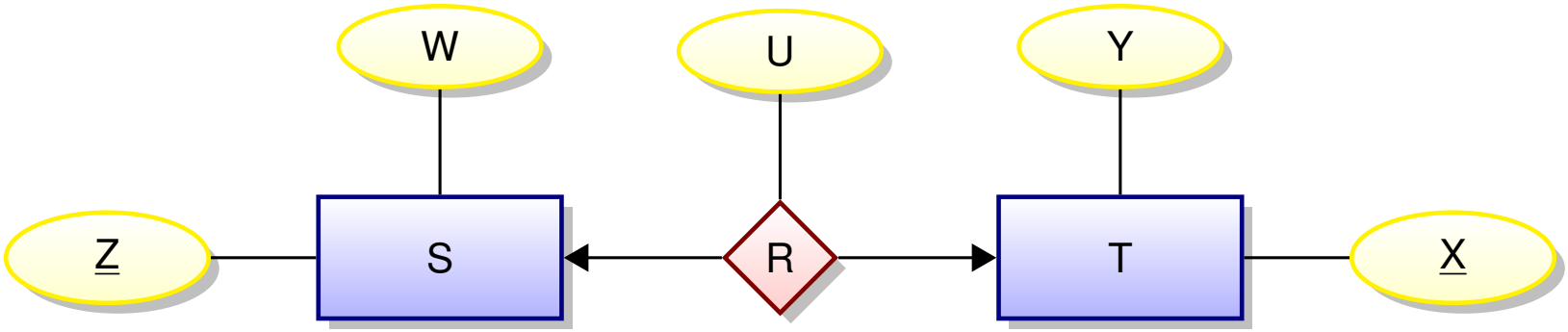


The relation R is **many-to-one** between T and S

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

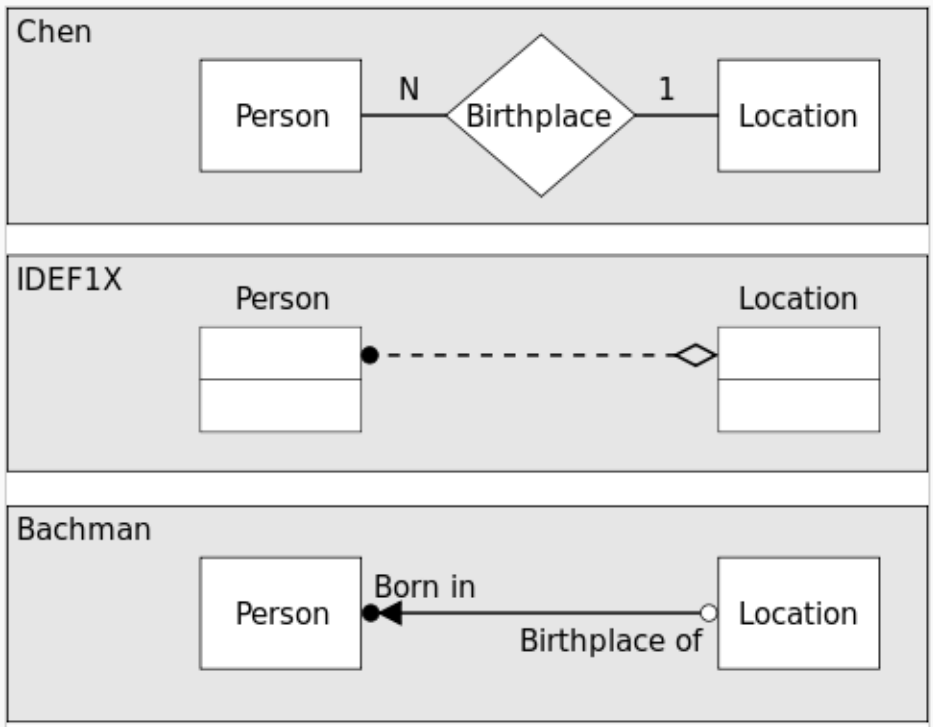
R

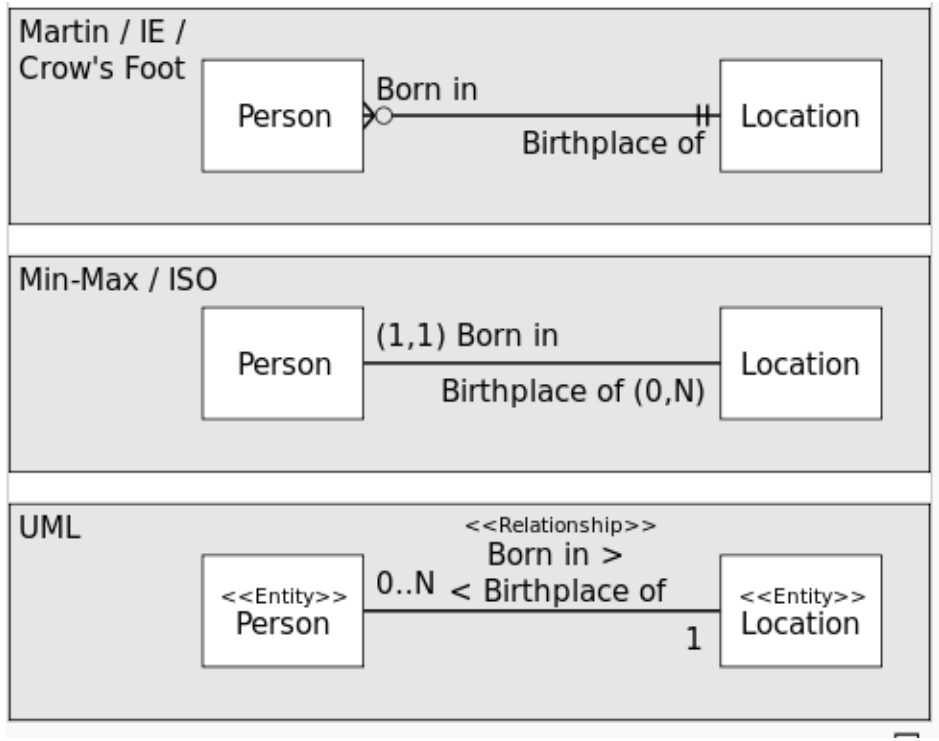
Z	**X**	U
z_1	x_2	u_1

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 ...

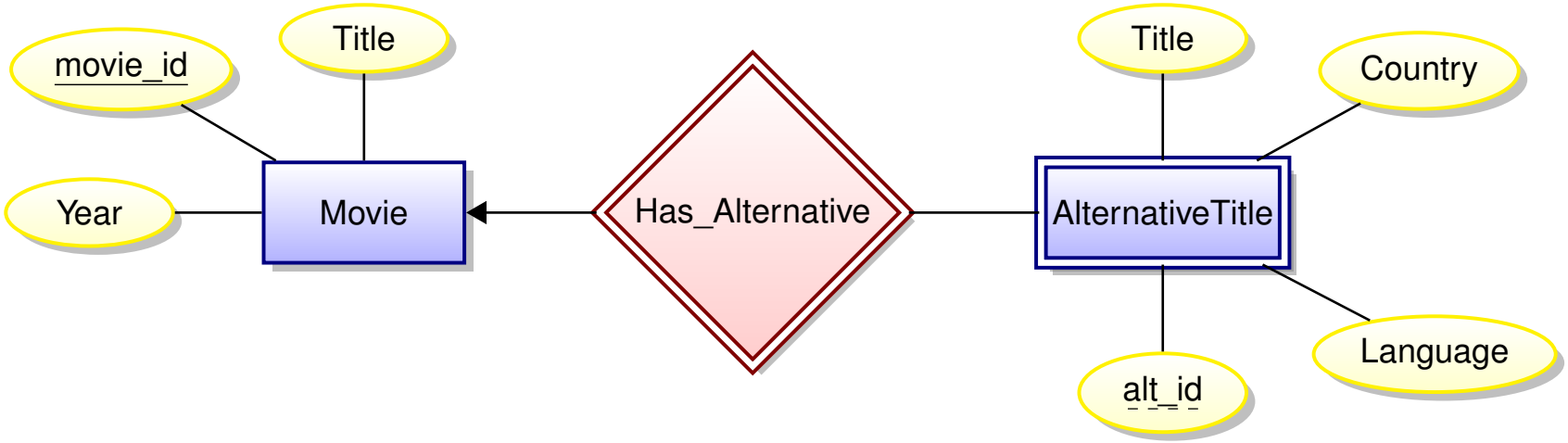




Various diagrammatic notations used to indicate a one-to-many relationship [Wikipedia: E/R model].

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

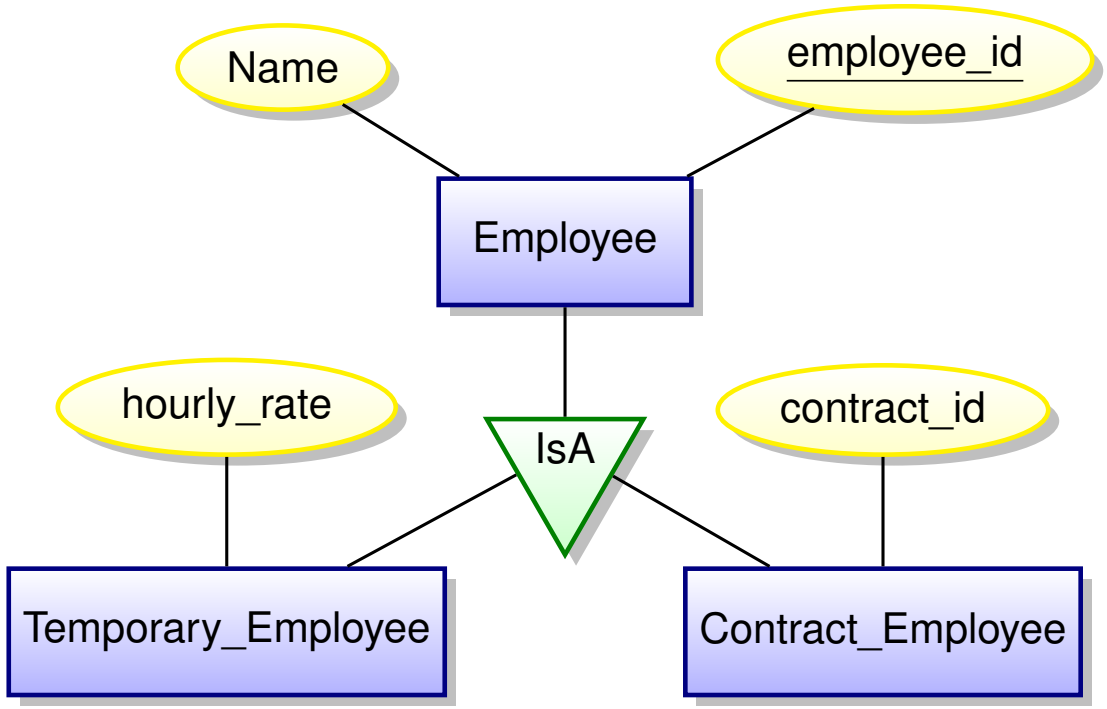
Weak entities



- 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)

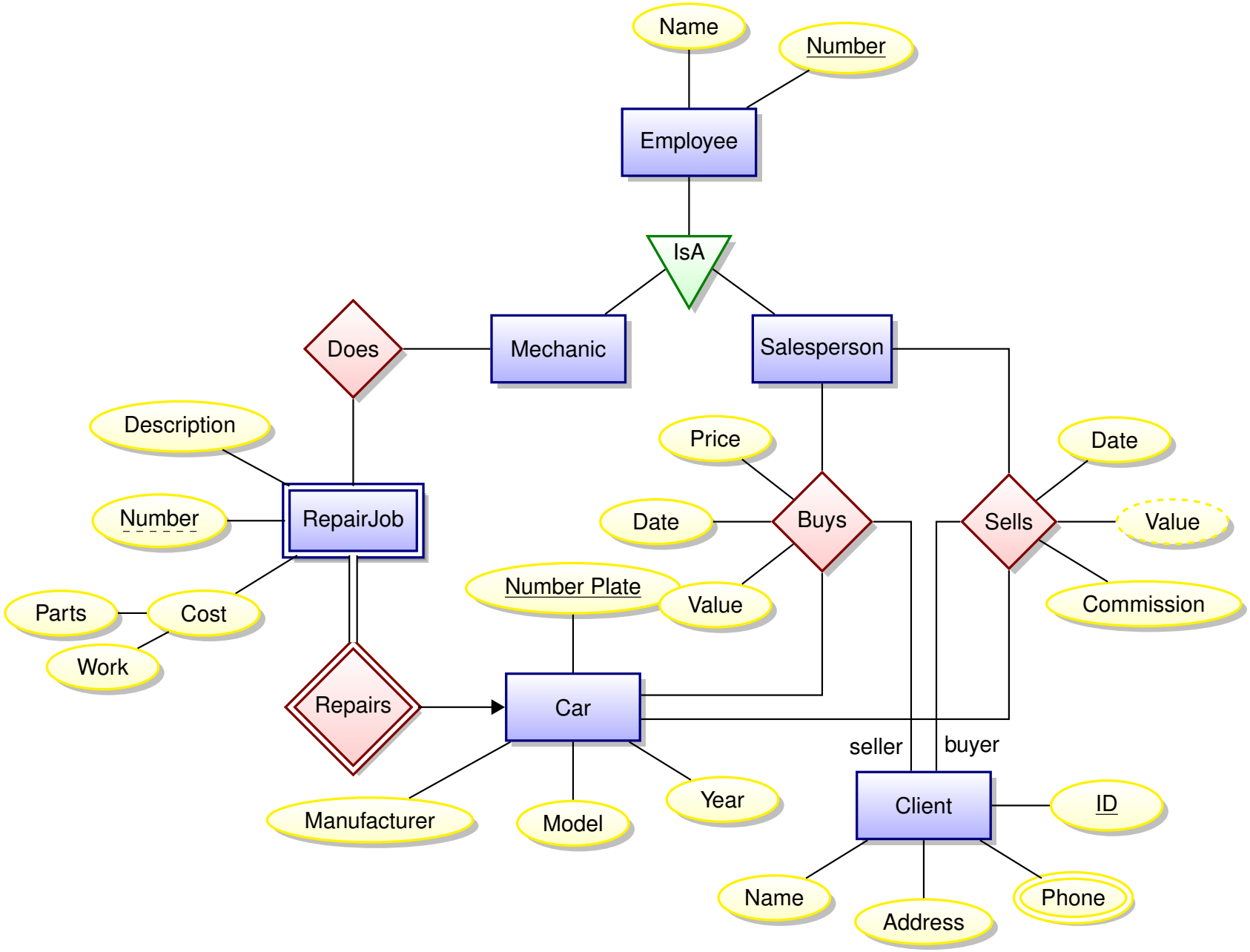
Sometimes an entity can have “sub-entities”. Here is an example:



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

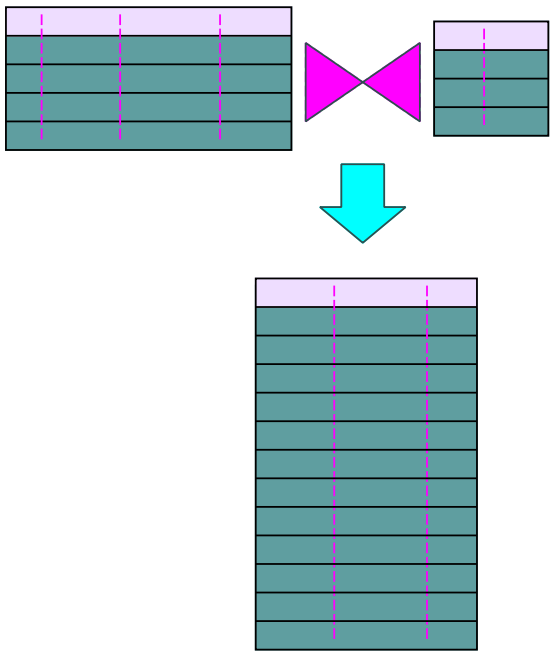
Lecture 3

- 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

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

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

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:

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

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

$$R_1,\ R_2,\ \cdots,\ R_k \quad \Longrightarrow \quad Q(R_1,\ R_2,\ \cdots,\ R_k)$$

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

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

- 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 ($\cup$)

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 $\cup$ 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'

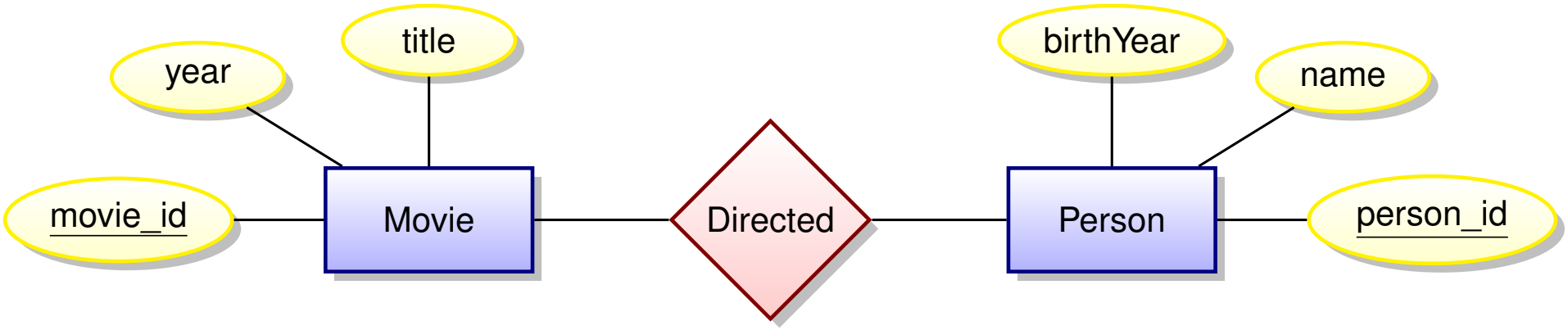
$\Longrightarrow$

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 E/R 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
tt9603212	Mission: Impossible - Dead Rec	2023	nm0003160	Christopher McQuarrie	1968
tt4873118	The Covenant	2023	nm0005363	Guy Ritchie	1968
tt15398776	Oppenheimer	2023	nm0634240	Christopher Nolan	1970
tt5971474	The Little Mermaid	2023	nm0551128	Rob Marshall	1960
tt6791350	Guardians of the Galaxy Vol. 3	2023	nm0348181	James Gunn	1966
tt0439572	The Flash	2023	nm0615592	Andy Muschietti	1973
tt2906216	Dungeons & Dragons: Honor Amon	2023	nm0197855	John Francis Daley	1985
tt2906216	Dungeons & Dragons: Honor Amon	2023	nm0326246	Jonathan Goldstein	1968
tt10366206	John Wick: Chapter 4	2023	nm0821432	Chad Stahelski	1968
tt12263384	Extraction II	2023	nm1092087	Sam Hargrave	
tt12758060	Tetris	2023	nm1580671	Jon S. Baird	1972
tt1517268	Barbie	2023	nm1950086	Greta Gerwig	1983
.....	.....	....	.....		....

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 (lecture 5) 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 our 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
tt0012349	The Kid	1921	nm0088471	B.F. Blinn	His Assistant
tt0012349	The Kid	1921	nm0000122	Charles Chaplin	A Tramp
tt0015864	The Gold Rush	1925	nm0000122	Charles Chaplin	The Lone Prospector
tt0021749	City Lights	1931	nm0000122	Charles Chaplin	A Tramp
tt0027977	Modern Times	1936	nm0000122	Charles Chaplin	A Factory Worker
tt0032553	The Great Dictator	1940	nm0000122	Charles Chaplin	Hynkel - Dictator of Tomania
tt0032553	The Great Dictator	1940	nm0000122	Charles Chaplin	A Jewish Barber
tt0012349	The Kid	1921	nm0701012	Edna Purviance	The Woman
tt0012349	The Kid	1921	nm0001067	Jackie Coogan	The Child
tt0012349	The Kid	1921	nm0588033	Carl Miller	The Man
...	...	...	...	...	...

Observations

- Both E/R entities and E/R 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?

Q. How can we ensure that the table representing an E/R relation really implements a relationship? A. 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.

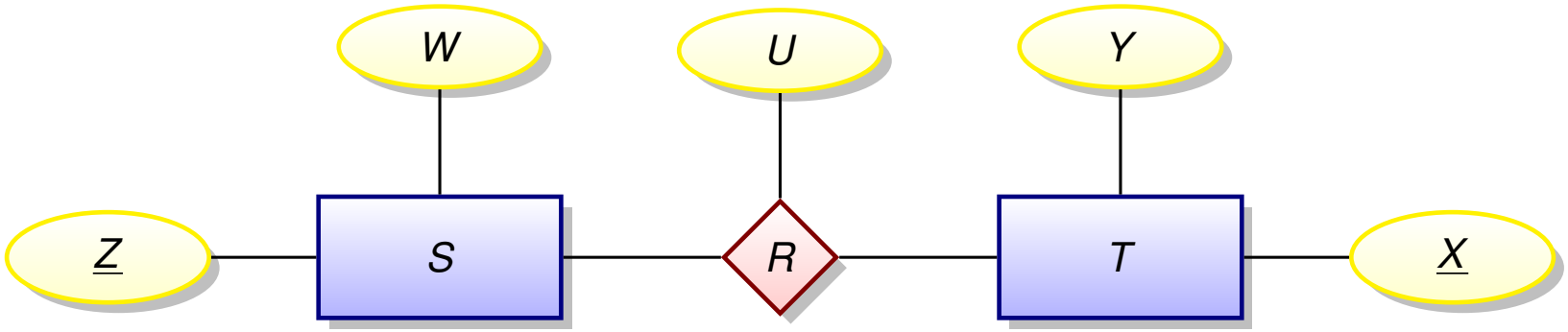
A schema with a simple key:

```
CREATE TABLE genres (
   genre_id integer NOT NULL,
   genre TEXT 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   -- up to 16 charact
      REFERENCES movies (movie_id),
   genre_id integer NOT NULL
      REFERENCES genres (genre_id),
   PRIMARY KEY (movie_id, genre_id));
```

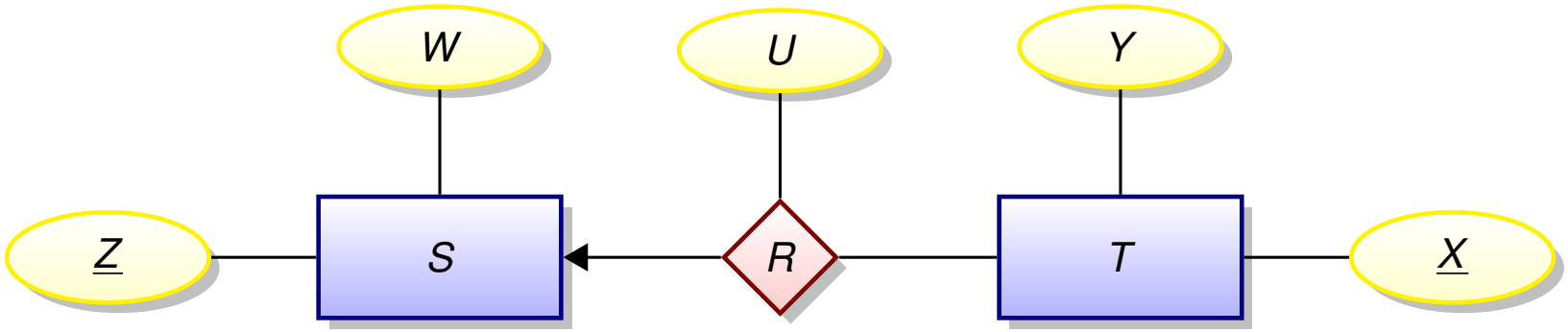
Relationships in tables (the “clean” approach).



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



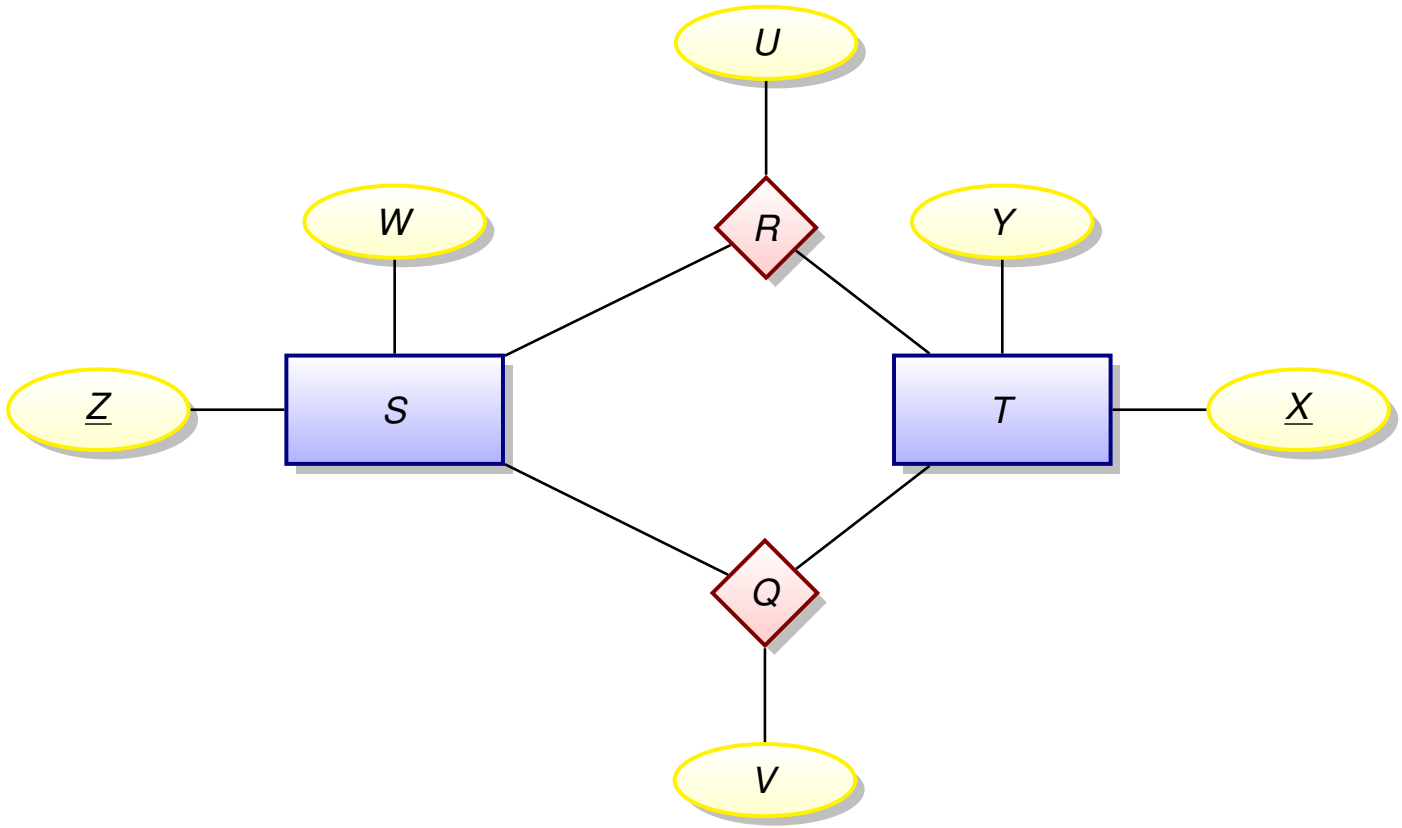
Suppose R is one-to-many (reading left to right)

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;
```

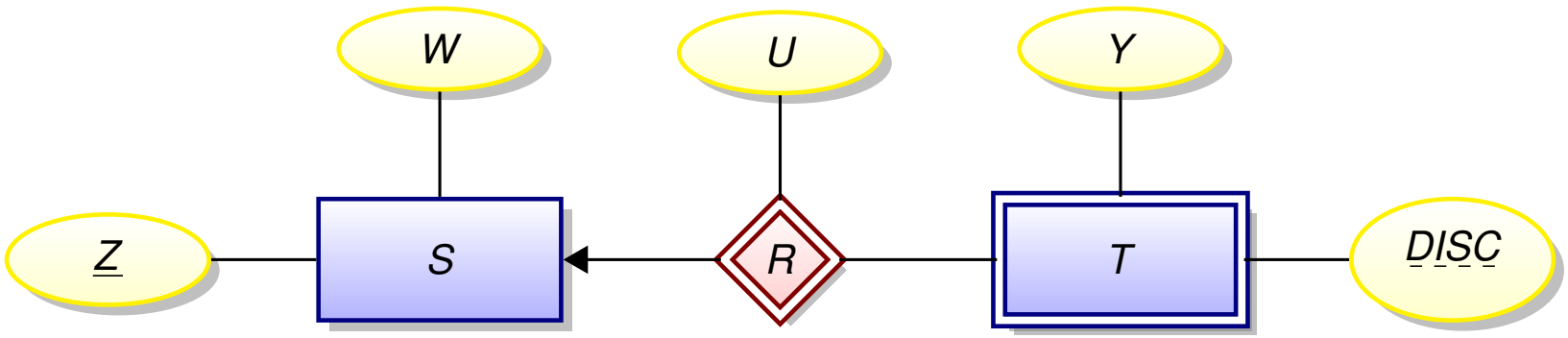
Using our database, this query produces the output

```
position  total
--------  -----
actor     5955
writer    2907
producer  2509
director  1588
composer  848
```

Was this a good idea?

Discuss!

Implementing weak entities.



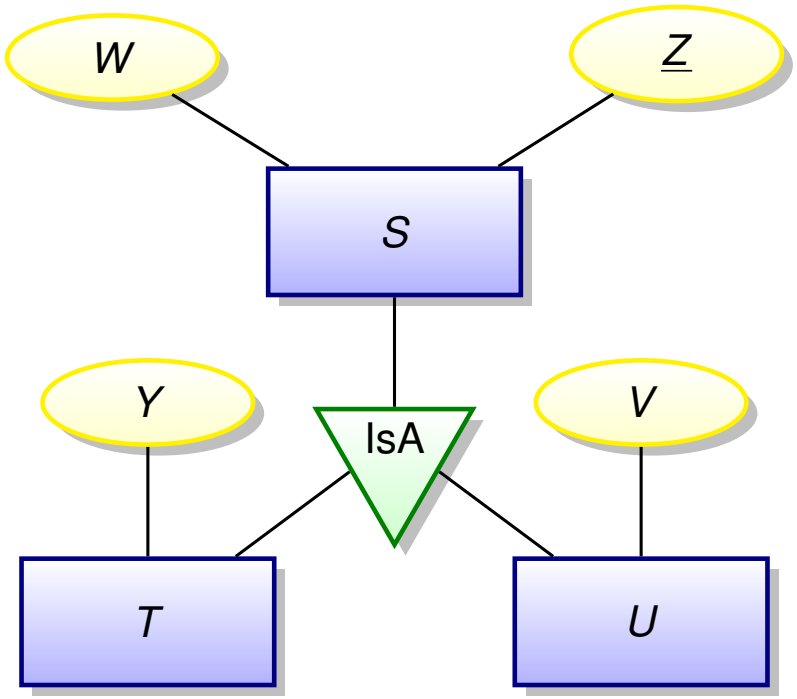
One (clean) approach:

- $S(\underline{Z},\ W)$
- $R(\underline{Z,\ DISC},\ U)$ 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.

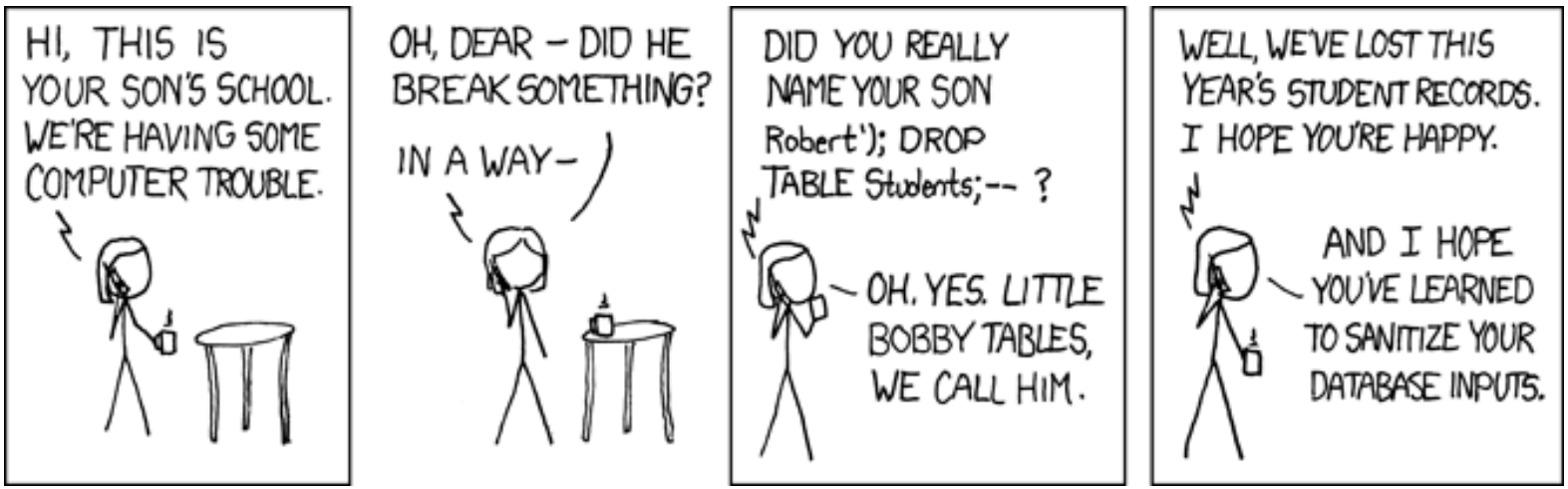


One (clean) approach:

- $S(\underline{Z},\ W)$
- $T(\underline{Z},\ Y)$ 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 of the course.



(http://xkcd.com/327)

Lecture 5 - Transactions, Reliability, Throughput & Consistency.

- What is a transaction?
- Locks and their effect on transaction rate (throughput).
- Data redundancy and update anomalies.
- Relational normalisation to reduce/eliminate redundancy.
- Normalisation vs. transaction throughput.
  - Databases can be designed to maximise the number of concurrent users executing update transactions.
- But what if your applications never or rarely update data?
  - Read-oriented vs. update-oriented databases.

Transaction Processing

A **transaction** on a database is a series of queries and changes that externally appear to be atomic.

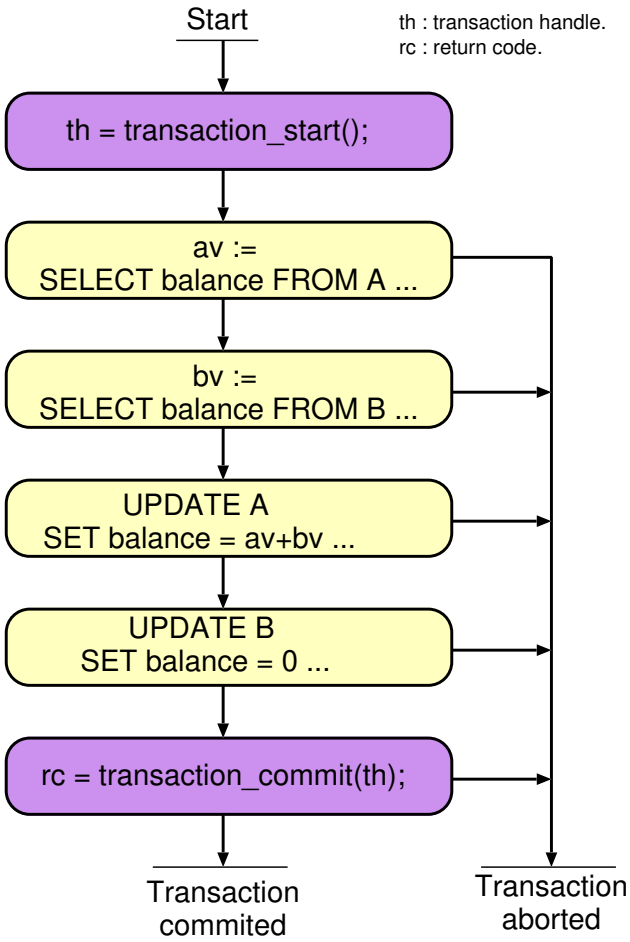
Internal transactions:

- Some number of values are read, perhaps more values conditionally read, and then various values are changed based on the values read.
- All of the values read or written are inside the same database.

External ‘transactions’ (do not really exist):

- Some of the values changed or other side effects (like sending an SMS acknowledgement) are external to the DBMS.
- The DBMS cannot help make these atomic. Instead the system designers have to think carefully about undoing them (e.g. “The flight booking we just confirmed has now been cancelled since it turns out you are broke.”).
- [Many DBMS systems allow the application to **abort** a transaction before it is **committed**, but this is a topic for Part Ib *Concurrent Systems*.]

Transaction client flow.



- Transaction ‘start’ and ‘commit’ calls bracket the body.
- The body consists of any number of queries and updates in any order.
- The client may chose to abort at any time: all updates are then undone by the DBMS.
- In some (optimistic) systems, the updates or commit may also abort and the client is forced to restart the transaction.
- DBMSs support **concurrent** transactions.

[*NB*: This slide’s contents are not examinable on this course; they form part of Part Ib CDS.

ACID transaction properties

Atomicity: All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are. For example, in an application that transfers funds from one account to another, the atomicity property ensures that, if a debit is made successfully from one account, the corresponding credit is made to the other account.

Consistency: Every transaction applied to a consistent database leaves it in a consistent state. For example, in an application that transfers funds from one account to another, the consistency property (*invariant*) is conservation of money: the total value of funds held over all accounts remains constant.

Isolation: The intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be (*serialized*). For example, in an application that transfers funds from one account to another, the isolation property ensures that another concurrent transaction sees the transferred funds in one account or the other, but not in both, nor in neither.

Durability: After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure. For example, in an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed.

[*NB*: Implementing ACID transactions is one topic covered *Ib Concurrent and Distributed Systems* [web: IBM definition].

ACID vs BASE

As we'll see next lecture, many NoSQL systems weaken ACID properties. The result is often called BASE transactions (pun intended).

BA: Basically Available,

S: Soft state,

E: Eventual consistency.

Exactly what this means varies from system to system. This is an area of ongoing research. It's certainly ideal for some applications, but some proponents have lost their faith and fallen back to a relational system.

[Wikipedia: BASE].

Implementing ACID transactions requires locking data

A **lock** is a special software or hardware primitive that provides **mutual exclusion**. A resource (section of code, data or file) can be locked for exclusive access by one concurrent application which must unlock it again after use. Other contending applications have to **wait**, which slows systems down.

- Locks are acquired and released by transactions.
- Locks can be placed along a spectrum of granularity from very coarse-grained (lock the entire database!) to very fine-grained (lock a single data value).
- How locks are used to implement ACID is not part of any DBMS API. Rather, this is part of the “secret sauce” implemented by each vendor.
- **Observation:** If transactions lock large amounts of data, or lock frequently used data, fewer concurrent updates can be supported, degrading **throughput**.

What is redundant data? Is it bad?

Our definition:

Data in a database is **redundant** if it can be deleted and then reconstructed from the data remaining in the database.

Why is redundant data problematic?

- If data is held in more than once place, copies can disagree.
- In a database supporting a high rate of update transactions, high levels of data redundancy imply that **correct** transactions may have to acquire many locks to consistently update redundant copies.

Redundant data goody:

- If updates are rare, having multiple copies can increase read bandwidth and speed up lookup.

[*NB*: Time-stamped, journalled or backup copies are used to provided durability, but this is not what we mean by redundancy here.]

‘Closure’ — a widely used term in Computer Science.

Closure: an iteration is repeated until there are no further changes (a fixed-point is found).

Least F/P iteration example: division.

```
let divider(num, den, quot) =  // Non-recursive!
    if den * quot >= num then (num, den, quot)
    else (num, den, quot+1)
```

- The least fixed-point of a function is the first argument value that is also its return value (intersects y=x).
- To divide, say 100 by 9 we ask for the LFP of `divider`(100, 8, 0) which will be (100, 8, 12).
- We’ll talk about transitive closure in Lecture 7, adding further edges to a graph until no further are needed for all paths to be achievable in one step.
- Normal-form conversion is also a closure iteration.

[*NB:* This slide is mostly an aside to discuss general principles.]

Normal form representation.

- For many forms of data, a unique normal form for that information can be defined.
- To achieve it, information-preserving, reorganisation/rewriting rules (transforms) are applied until closure.
- A typical rule might be: swap a commutative operator's arguments over if lexographical ordering of the arguments is not observed.
- For example $(x + 2)(x + y + x)$ might be normalised as $2x^2 + 2x + xy + 2y$ based on multiplying out, sorting terms in order of power and then sorting alphabetically.

[*NB:* Independently rewriting both the l.h.s. and r.h.s. of an equation until both are in normal form and then checking for equality (textual identity) is one standard approach to mathematical proof.]

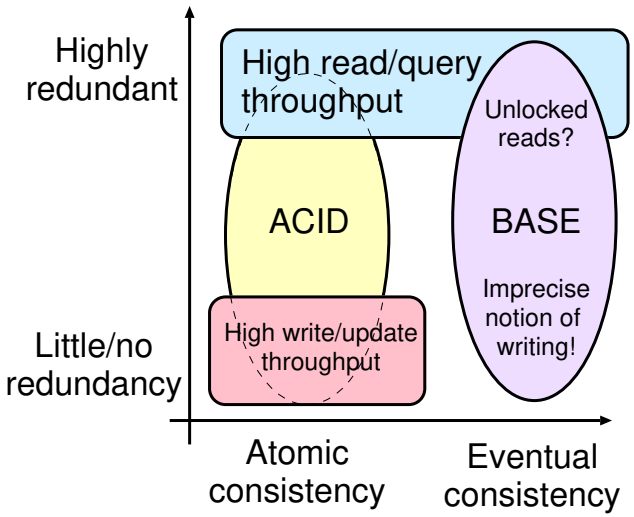
Normal form database schemas.

A normalised database is essentially one that has little or no redundant data.

- Typically, redundant relational databases have tables with too many attributes.
- A good rule is that all table data should either be key or semantically depend on the key.
- If you can spot data that does not directly depend on the key (recall GP's age field), that part of the table should be split off into a separate table. This procedure is then repeated on the new tables until closure.
- 'Splitting off' is essentially a division transform (ie. information-preserving rewrite) that can be reversed using a join, which behaves like a multiplication.
- Automated procedures have been mooted to convert databases into such normal forms (3^{rd} normal form or Boyce-Codd* normal form etc.).
- But computers cannot really understand what 'semantically depends' means so **doing a good job of Entity-Relationship modelling in the first place**, or manual decomposition, is generally preferable.
- Reducing redundancy facilitates higher update throughput.

*You only need to know that data should ideally be 'functionally' or 'semantically' dependent on the primary key. The subtleties of 3NF vs. BCNF etc. are off the syllabus.

Redundancy/Consistency/Throughput trade off.



- Low redundancy gives good update throughput (need only lock a few data items).
- High redundancy gives good query times (fewer files/blocks need be accessed).

- Data redundancy can lead to stored data inconsistency if updates are not thorough.
- Unlocked reading can give the impression of inconsistent data stored (*eg.* packet tracked as at depot and on van).
- Precomputing answers to common queries (either fully or partially) can greatly speed up query response time: introduces redundancy, but useful for some read-intensive applications. This is an approach common in aggregate-oriented databases.

[*NB*: DBMS design is multi-dimensional and no 2-D projection defines the whole space.

eg. Suppose only one updater?]

Throughput: Why read-oriented databases?

A fundamental tradeoff

Introducing data redundancy can speed up read-oriented transactions at the expense of slowing down write-oriented transactions.

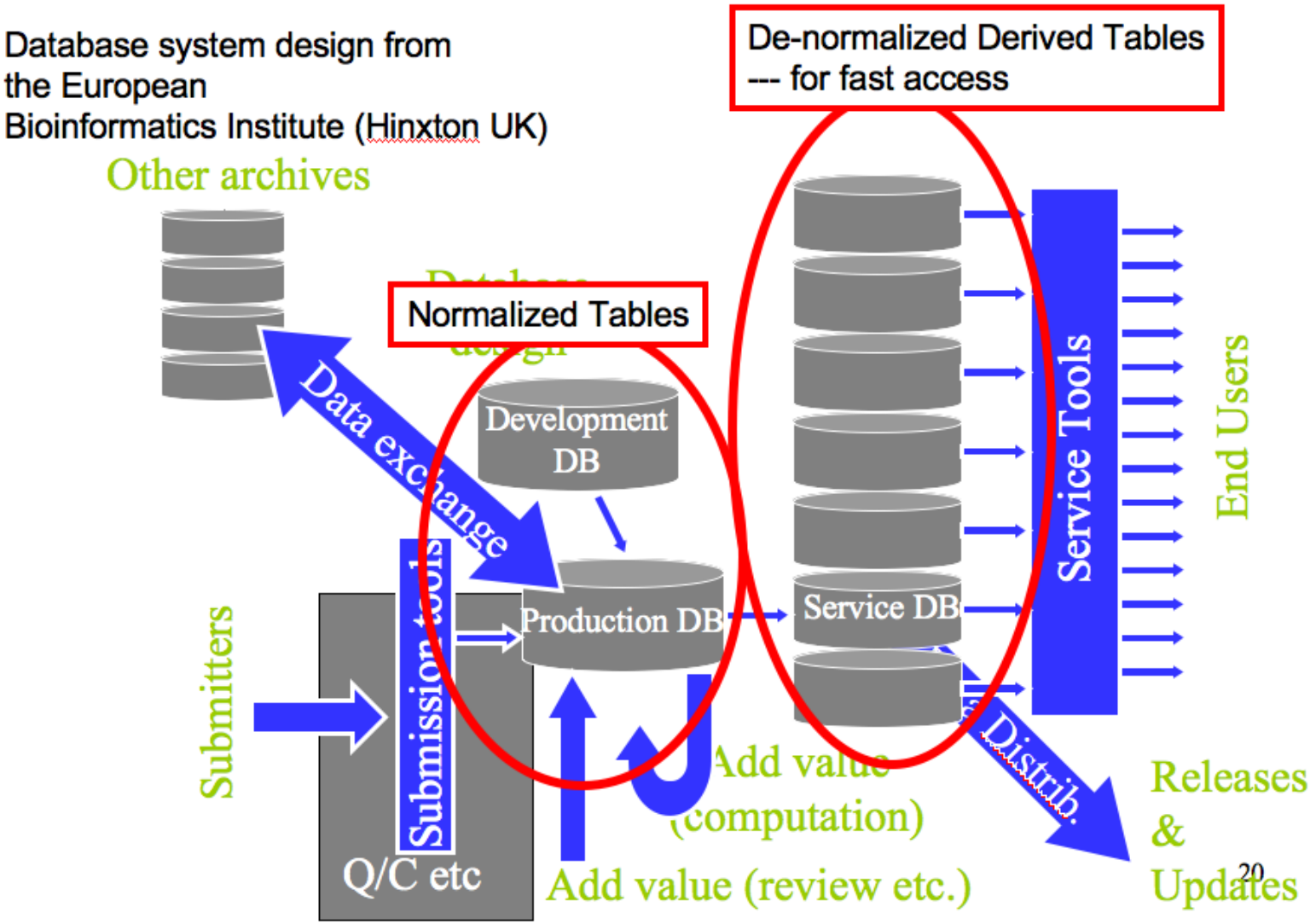
Something to ponder

How do database indexes demonstrate this point?

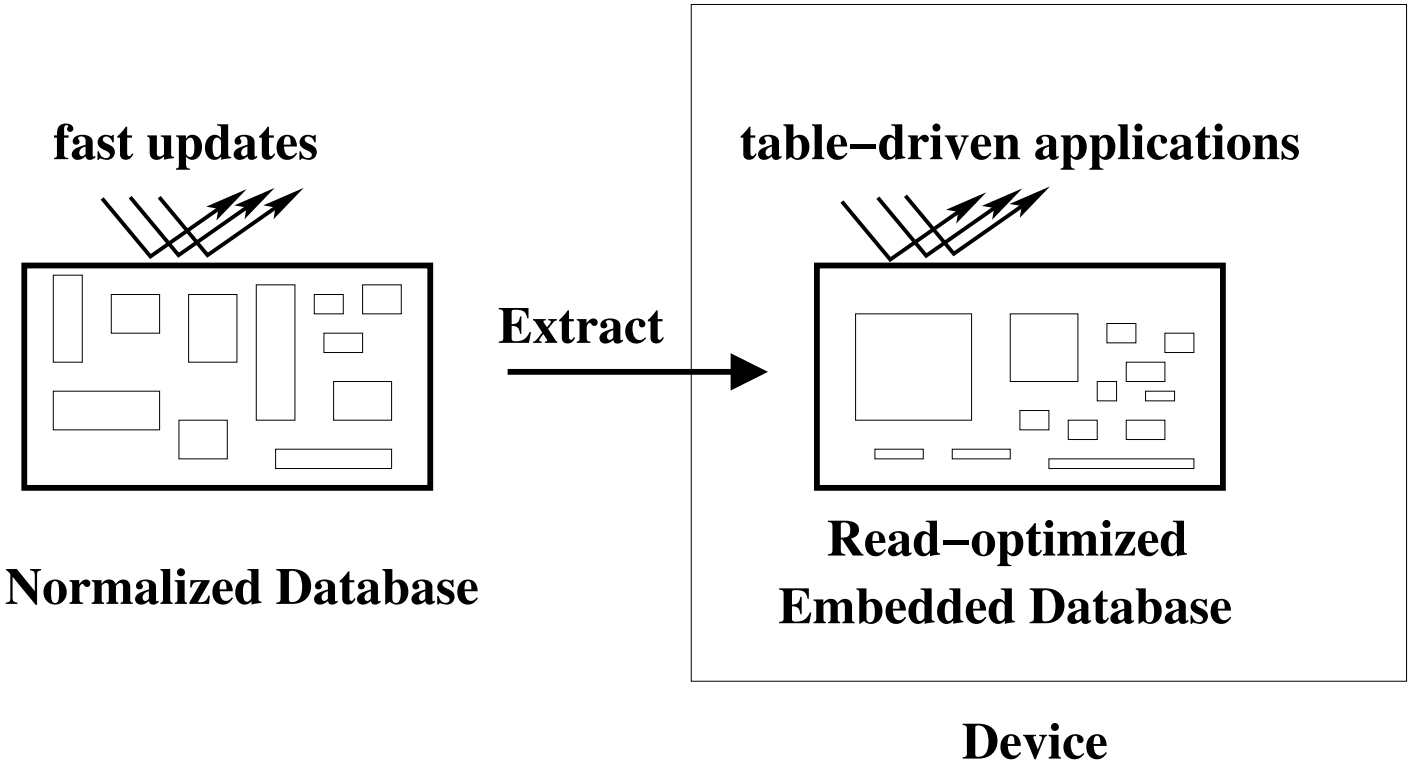
Situations where we might want a read-oriented database

1. Your data is seldom updated, but very often read.
2. Your reads can afford to be mildly out-of-synch with the write-oriented database. Then consider periodically extracting read-oriented snapshots and storing them in a database system optimised for reading. The following two slides illustrate examples of this situation.

Example : Hinxton Bio-informatics



Example: Embedded databases



An embedded database system is a database management system which is tightly integrated with an application software; it is embedded in the application — web: Wikipedia.

For instance: a different SELECT from the main staff table might be held in each electronic door lock.

FIDO = Fetch Intensive Data Organisation

OLAP vs. OLTP.

OLAP — Online Analytical Processing

- Write once or journal/ledger updates.
- Commonly associated with terms like Decision Support, Data Warehousing, etc..

OLTP — Online Transaction Processing

- A rich mix of queries and updates to live data.

	OLAP	**OLTP**
Supports	analysis	day-to-day operations
Data is	historical	current
Transactions mostly	reads	updates
optimised for	reads	updates
data redundancy	high	low
database size	humongous	large

OLAP vs. OLTP (continued).

Processing power:

- Historically, available computing power motiviated a clear distinction between OLAP and OTAP. Bridge using **E**xtract from OTAP, **T**ransform, **L**oad into OLAP).
- Today, both OLAP and OTAP applications often are supported by one DBMS [web: IBM].

Update history:

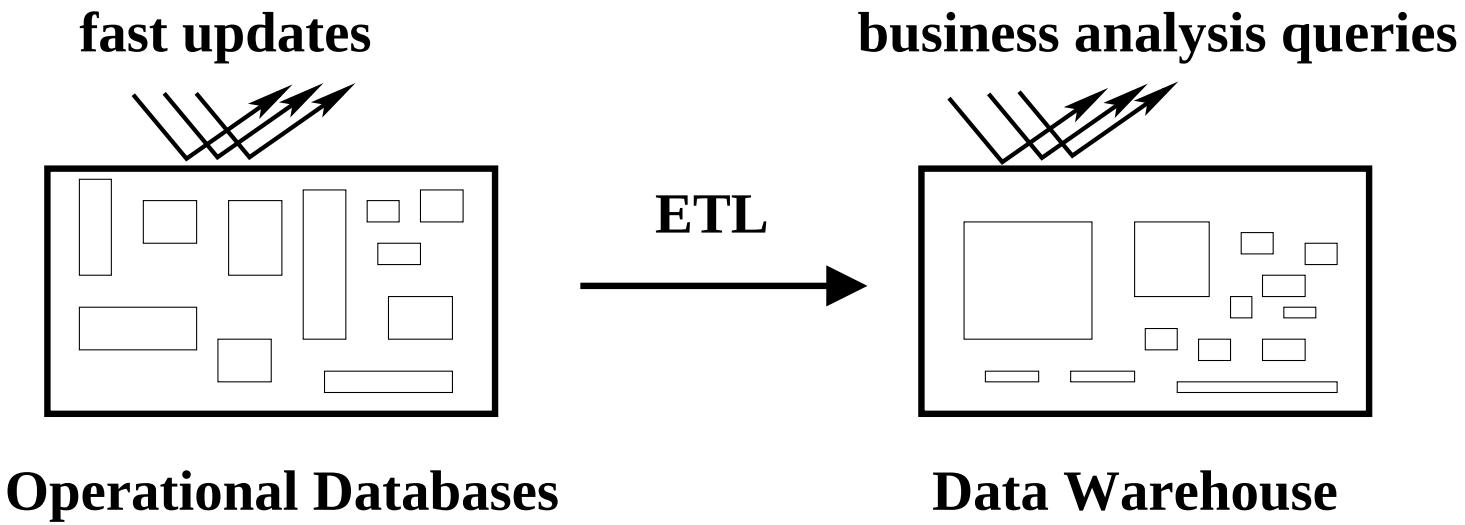
- An update to a relational database occludes the previous value of a field.
- A revision control system (*eg*. git) stores the update history — an additional **dimension** to the stored data/documents.
- Even for OLTP, an update history within a limited time horizon is always stored for ACID durability.

Further dimensions*:

- Looking at historic versions of a 2-D table makes it a **cube**.
- The data (hyper-)cube model* adds further dimensions where the indivdual contributions to a value in a table (*eg*. a total of something) can be seen.
- Summing (group-by then scalar reduction) in different dimensions gives the same result (*eg*. summing by region, salesperson or paint colour).

* = no longer on the syllabus or examinable.

Example: Data Warehouse (Decision support)



ETL = Extract, Transform, and Load

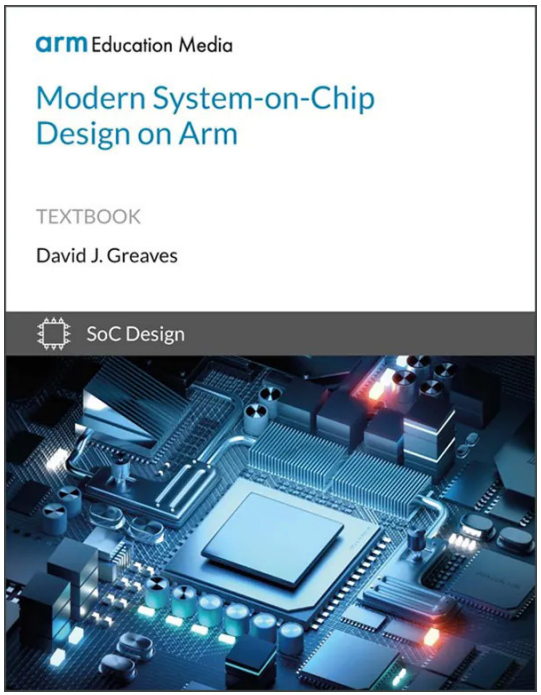
- [This looks very similar to slide 98!]
- Slide 98 stored data optimised for *a priori* known queries. Size would be an issue for embedded use.
- Here data is pre-processed in many/every conceivable way for visualisation and exploration by (typically) human agents.

Lecture 6 - Semi-structured Document Databases

- Semi-structured data.
- NoSQL movement.
- Document-oriented databases.
- Denormal and BASE possible advantages.
- An example database: TinyDB 

.
- Path query languages and *ad hoc* HLL access.

Semi-structured data.



[web: ONLINE]

A textbook such as the one illustrated is a document written in natural language (English) but it has some structure:

- There are chapters with names that contain numbered sections and sub-sections.
- There are figures and diagrams that have their own numbering system.
- There are extensive cross references between one section and another, etc..
- But it would be far too much work to manually index every word of text: a task unlikely to be useful and also poorly-defined.

What can sensibly or usefully be stored in a database?

Two approaches

Either

Store in two parts:

- Keep the document in its native form (LaTeX, Word, PDF...),
- Store the indexable features in relational tables.

or

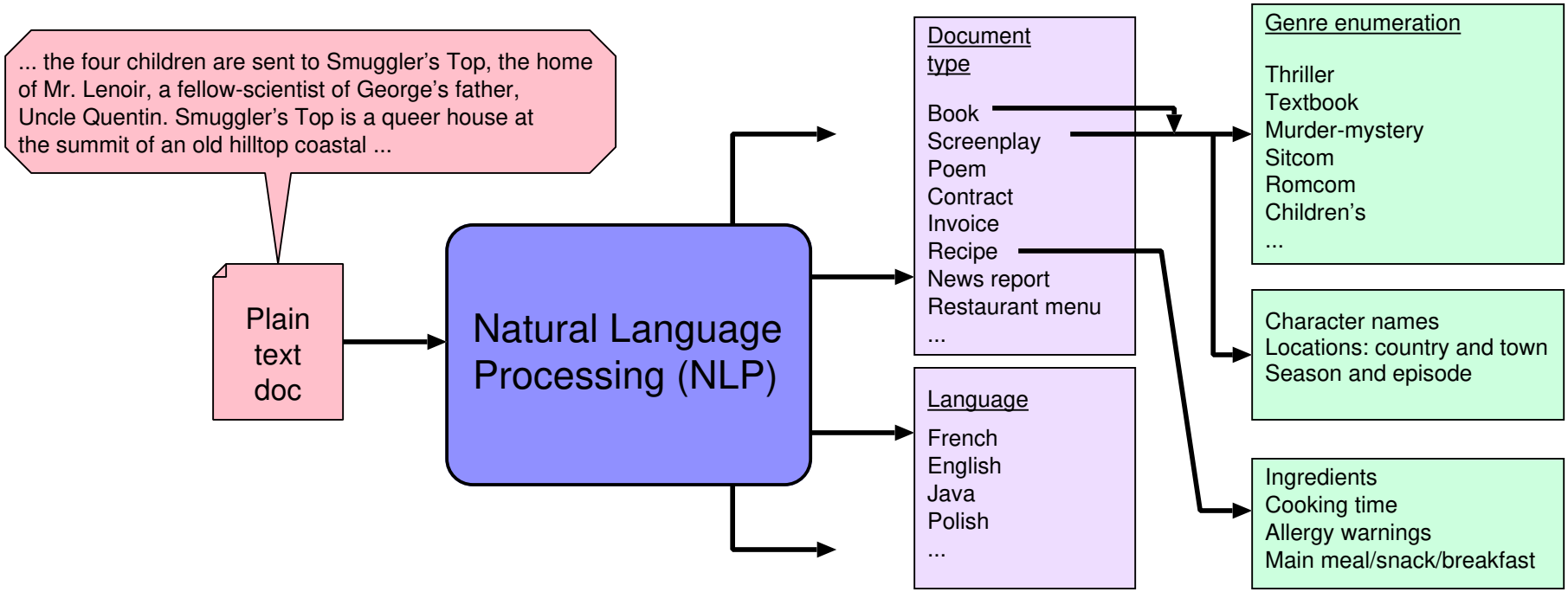
Your Ultimate Guide to the
Non-Relational Universe!

Store just once, perhaps **shredded**, and use something instead of SQL for queries:

- Keep the document largely in native form (especially XML, JSON),
- Develop database tools that can navigate semi-structured data. These must return best-effort query answers, given that 'schema' violations could be frequent.

Adding Structure to Unstructured Documents

Real-world data is often analogue and/or noisy



- Processing tools or humans can remove noise, discard spurious data, index and classify, correct spellings *etc.*.
- The document is carved up and marked up for storage.
- Advanced NLP or a simple keyword-based analysis.
- The original can be **unshredded**, as and when necessary.

[*NB*: Such NLP techniques are not examinable for this course.]

BASE - Soft state & Eventual consistency

Orthogonal aspects:

- Tables vs. Documents.
- Distributed vs. centralised (monolithic).
- ACID vs. BASE.

- Despite orthogonality, document databases are typically designed to be easy to distribute and to not support ACID transactions.
- Any or all ACID properties are relaxed, giving BASE:
  - **BAse**: Basically available: availability promoted over consistency. Any change in data made at one point is promulgated to all the different nodes.
  - **Soft State**: stored values may change without any application intervention owing to eventual consistency updates or network partition.
  - **Eventual Consistency**: all readers throughout the system will eventually see the same state as each other.

Key/Value Store

Recall the associative store (dictionary) from Lecture 1: the values stored could be generalised from strings to **blob**s, which are just sequences of bytes.

- Any structure inside the blobs is opaque to the key/value store.
- Many implementations are distributed, spreading the data randomly over all participating machines as **shards**.
- Opaqueness implies the DBMS knows nothing about what is stored – it would not mind if values were encrypted and it never saw the encryption keys.
- Distribution provides redundancy* and load balancing (*eg*. by a hash of the key).
- Implementations can range between ACID and BASE semantics.

[* The redundancy here is to help provide ACID durability and is nothing to do with schema redundancy.]

Serialising (marshalling or pickling) an object.

Serialising: converting a data structure into a series of bytes for transfer over a network or storing in a file.

- JSON was originally designed for serialising data.
- XML was designed for serialising and marking up a human-readable document so different parts could be located or processed in different ways.
- Both are frequently used for transferring data between databases or apps (CSV also commonly used).
- But NoSQL may use them as the primary form of a document to be stored.

Abstract Syntax (formal spec) of XML and JSON

Formal specifications using ML-like concrete syntax where ulist is the same as list except the order is unimportant and keys cannot be repeated (ie a dictionary).

Examples

```
XML:  <PERSON name="Greaves"><DOB month="May" year="1902"/></PERSON>
JSON: "person":{"name":"Greaves","dob":{"month":"May","year":"1902"}}
```

Slightly simplified abstract syntaxes (grammars):

```
type xml_t = // XML stands for eXtensible Markup Language
  | ELEMENT of string * (string * string) ulist * xml_t list
  | LEAF of string

type json_t = // JSON stands for JavaScript Object Notation
  | LEAF_S of string
  | LEAF_N of integer
  | ARRAY of json_t list
  | OBJECT of (string * json_t) ulist
  | NULL
```

Important fact: they both contain tree-structured text with named nodes and hence are broadly similar.

XML – Structured or Unstructured?

Structure spectrum:

1. All data in one large element,
2. Semi-structured: some elements contain a lot of text (**clob** ?), others contain an atomic value (as per RDBMS),
3. Every atomic value in its own element (unrealistic).

XML documents may associated with a (DTD or W3C [not exminable]) schema:

Schema rigorousness spectrum:

1. A schema, named with a URL exists. The schema dictates precisely the element names and which elements may be allowed inside which others along with occurrence limits, Allowable attributes are also named.
2. The schema is relaxed: *eg*. the order of elements inside a parent element is unimportant,
3. Other attribute or elements, beyond those in the schema are also allowed (*eg*. application-specific extensions),
4. There's no schema at all.

Document-oriented database systems

- A **document-oriented database** stores data in the form of *semi-structured objects*. Such database systems are also called **aggregate-oriented databases**.

Un-structured data:

- The key/value DBMS just mentioned could store unstructured documents.
- In any application, there is likely to be some application-level structure within the blobs,
- but this cannot be exploited by the DBMS.

- Query of a distributed database encounters a *round-trip time*.
- Denormalised data is not directly semantically-related to the key it is stored under (as we hinted for rDBMS).
- A denormal DBMS enables us to rapidly pull much or all of the data likely to be needed using one key.
- One or two fetches of denormal data should enable all sorts of fast, local operations (select, join etc.) in an application-specific way.

Document query languages

All sorts of queries are possible:

- Query unstructured text (*eg*. How many words? What is the FOG factor? Does it mention Kevin Bacon?)
- Query tags (*eg*. What are the 'eye-colour' attributes to each of the 'Vizier' elements under the second 'Chapter' element?)
- Application-specific compositions of these.

- So although there are standards such as Xpath [web], instead using general high-level languages to formulate queries is common.
- Ideally write queries in a declarative language since imperative programming defeats future automated query optimisation.
- The 'database' itself may support a variety of inverted indices or re-normalised data (example shortly).

Typical document query languages: *eg.* XPath

We need to navigate a semi-structured tree, aggregating various bits:

```
type pathexp_t =  // Typical query abstract syntax
  | SelectRoot                                  // Whole thing
  | SelectAttribute of pathexp_t * string       // v in string="v"
  | SelectElement of pathexp_t * predicate      // <EL> ... </EL>
  | NextElement of pathexp_t * int              // Fwd or back by n
  | SelectData of pathexp_t * ranges            // Chunks of raw text
  | Concatenate of pathexp_t * pathexp_t        // Aggregation
  | ...
```

If we have more than one tree, something equivalent to a join is also needed.

What is the return type of a query? SelectRoot clearly gives a whole tree whereas SelectAttribute just gives one string...

Some say *"Shucks, who needs types!"*, but algebraic data types can help [Part Ib *Concepts* Course]. We'll use TinyDB .

[*NB*: pathexp_t details not examinable.]

NoSQL Movement (1)

NOSQL DEFINITION: Next Generation Database Management Systems mostly addressing some of the points: being **non-relational, distributed, open-source** and **horizontally scalable.**

The original intention has been **modern web-scale database management systems**. The movement began early 2009 and is growing rapidly. Often more characteristics apply such as: **schema-free, easy replication support, simple API, eventually consistent / BASE** (not ACID), a **huge amount of data** and more. So the misleading term "nosql" (the community now translates it mostly with "**not only sql**") should be seen as an alias to something like the definition above. [based on 7 sources, 15 constructive feedback emails (thanks!) and 1 disliking comment. Agree / Disagree? Tell us so! By the way: this is a strong definition and it is out there here since 2009!]

- 'Horizontally scalable' — expand by adding further machines (not upgrading existing machines).
- [Is there a typo in their last line?]
- Can there really be schema-free, typeless programming?
- "There's a sketch on the whiteboard in Fred's office. It is slightly wrong because every tenth item in the list is actually a height and not a pointer to a wombat. Oh dear, I didn't know building management had installed new whiteboards over the summer!"

Different key nestings of (semi-)structured data.

- Here is some relation data [web] with composite key A B.
- To support rapid retrieval of all likely related data using different keys, we precompute and store several of them.
- This replication factor multiplies with any replication arising from the data being denormal.

Here the "A" value is unique and at the top of tree.

```
{ "A": a1, "X": x1,
  "R": [{"B": b1, "Z": z1, "Y": y1},
        {"B": b2, "Z": z2, "Y": y2},
        {"B": b3, "Z": z3, "Y": y3}],
  "Q": [{"B": b4, "Z": z4, "W": w1}]
}

{ "A": a2, "X": x2,
  "R": [{"B": b1, "Z": z1, "Y": y4},
        {"B": b3, "Z": z3, "Y": y5}],
  "Q": []
}

{ "A": a3, "X": x3,
  "R": [],
  "Q": [{"B": b2, "Z": z2, "W": w2},
        {"B": b3, "Z": z3, "W": w3}]
}
```

Same data, "B" value is now above "A" in the tree.

```
{ "B": b1, "Z": z1,
  "R": [{"A": a1, "X": x1, "Y": y2},
        {"A": a2, "X": x2, "Y": y4}],
  "Q": [] }

{ "B": b2, "Z": z2,
  "R": [{"A": a1, "X": x1, "Y": y2}],
  "Q": [{"A": a3, "X": x3, "Y": w2}] }

{ "B": b3, "Z": z3,
  "R": [{"A": a1, "X": x1, "Y": y3},
        {"A": a2, "X": x2, "Y": y5}],
  "Q": [{"A": a3, "X": x3, "Y": w3}]}

{ "B": b4, "Z": z4, "R": [],
  "Q": [{"A": a1, "X": x1, "Y": w1}] }
```

TinyDB database IMDB snapshot

TinyDB

This will be used for the 2^{nd} Assessed Exercise (tick).

- In-core, using JSON (not XML) and queried using Python.
- No support for transactions, hence easy(?) to implement a distributed/sharded version (we won't).
- Two primary, denormal tables (Movies and People).
- Unstructured text for Goofs, Trivia, Quotes *etc*. (now present).
- Data needs to indexed on various keys (keys must still be unique).
- Some fields are foreign keys, so key integrity is still expected, but it is not enforced.

Note: the database wouldn't stop you or even notice if you decided to put a person in the Movie table/collection; they are just names to help the programmer and provide logical separation, unlike SQL tables that enforce structure.

TinyDB : Example person record.

person_id nm0031976 maps to

```
{ 'person_id': 'nm0031976',
  'name': 'Judd Apatow',
  'birthYear': '1967',
  'acted_in': [
      {'movie_id': 'tt7860890', 'roles': ['Himself'],
       'title': 'The Zen Diaries of Garry Shandling', 'year': '2018'} ],
   'directed': [
      {'movie_id': 'tt0405422',
       'title': 'The 40-Year-Old Virgin', 'year': '2005'}],
   'produced': [
      {'movie_id': 'tt0357413',
       'title': 'Anchorman: The Legend of Ron Burgundy', 'year': '2004'},
      {'movie_id': 'tt5462602',
       'title': 'The Big Sick', 'year': '2017'},
      {'movie_id': 'tt0829482', 'title': 'Superbad', 'year': '2007'},
      {'movie_id': 'tt0800039',
       'title': 'Forgetting Sarah Marshall', 'year': '2008'},
      {'movie_id': 'tt1980929', 'title': 'Begin Again', 'year': '2013'}],
   'was_self': [
      {'movie_id': 'tt7860890',
       'title': 'The Zen Diaries of Garry Shandling', 'year': '2018'}],
   'wrote': [
      {'movie_id': 'tt0910936',
       'title': 'Pineapple Express', 'year': '2008'}]
}
```

TinyDB : Example movie record.

movie_id tt1045658 maps to

```
{ 'movie_id': 'tt1045658',
  'title': 'Silver Linings Playbook',
  'type': 'movie',
  'rating': '7.7',
  'votes': '651782',
  'minutes': '122',
  'year': '2012',
  'genres': ['Comedy', 'Drama', 'Romance'],
  'actors': [
     {'name': 'Robert De Niro', 'person_id': 'nm0000134',
      'roles': ['Pat Sr.']},
     {'name': 'Jennifer Lawrence', 'person_id': 'nm2225369',
      'roles': ['Tiffany']},
     {'name': 'Jacki Weaver', 'person_id': 'nm0915865',
      'roles': ['Dolores']},
     {'name': 'Bradley Cooper', 'person_id': 'nm0177896',
      'roles': ['Pat']}],
  'directors': [
     {'name': 'David O. Russell', 'person_id': 'nm0751102'}],
  'producers': [
     {'name': 'Jonathan Gordon', 'person_id': 'nm0330335'},
     {'name': 'Donna Gigliotti', 'person_id': 'nm0317642'},
     {'name': 'Bruce Cohen', 'person_id': 'nm0169260'}],
  'writers': [{'name': 'Matthew Quick', 'person_id': 'nm2683048'}]
}
```

But how do we query **TinyDB** ...?

... write python code:

```
>tdb_people.get(Query().person_id == 'nm0000002')
{'person_id': 'nm0000002',
 'name': 'Lauren Bacall',
 'birthyear': 1924,
 'deathyear': 2014,
 'acted_in': [{'movie_id': 'tt0276919',
               'title': 'Dogville',
               'year': 2003,
               'roles': ['Ma Ginger']}]}
```

This is a Python dictionary representing a person JSON document. It's not quite JSON: note the single-quotes.

Quick, in-lecture 

Things to think about (for Tick 2):

- When we write our Python we're doing query planning. What did we take into account? Did we make an index first?
- Imagine an actor's name has been systematically misspelled. What is the cost of correcting it in a document database? Should it even be corrected?
- An RDBMS query involves 3 joins. What affects the cost of the same query in TinyDB ?
- What sort of checks should be associated with inserting new data?
- Which of the ACID properties might be relatively easy to implement? [You'll be better placed to answer this after the Part Ib *CCDS* course.]

Branded types – an opposite to semi-structured.

- Databases hold a lot of strings and numbers.
- Many are members of enumerations: *eg*. colour, gender ...
- Many are units of measure (UoM): *eg*. date, weight_kg, weight_lbs ...
- Should we make types overt?

```
type velocity_t = branded float;
val speed_of_light:velocity_tlsh = 2.998e8;

type distance_t = branded float;
val bognor_to_romsey:distance_t = 45.2;
val romsey_to_paris:distance_t = 212.4;
val bognor_to_paris = bognor_to_romsey + romsey_to_paris;

val journey_time = bognor_to_paris / speed_of_light;
(* All ok so far *)

val nonsense_value = journey_time + bognor_to_romsey;
*** Error: dimensionally-unsound expression input!
```

- Many silly operations on data can be prevented.
- Being the key to another table is a sort of type.

[*NB*: I've used a made-up language that is not examinable.]

The NoSQL schema-free ideal (grail) ?

- "No schema" really means "not stored as part of the database or checked during update".
- For most activities, there will inevitably still be a schema - perhaps on a whiteboard, scrap of paper or stored in somebody's head.
- New joiners to a software project have to learn the schema somehow. The DBMS does not help.
- Poor education? — "Typeless languages don't use a keyboard to type them in" [web: Have the tables turned on NoSQL?].

Commercial success(?) of Javascript, Ruby, Python, PHP, and other dynamically-typed languages:

- Javascript is often just a compilation target and is being displaced by WASM.
- Python types are now being used *de rigueur* (pioneered by J Lehtosalo of this department).

Semi-structured, Aggregate and NoSQL Summary

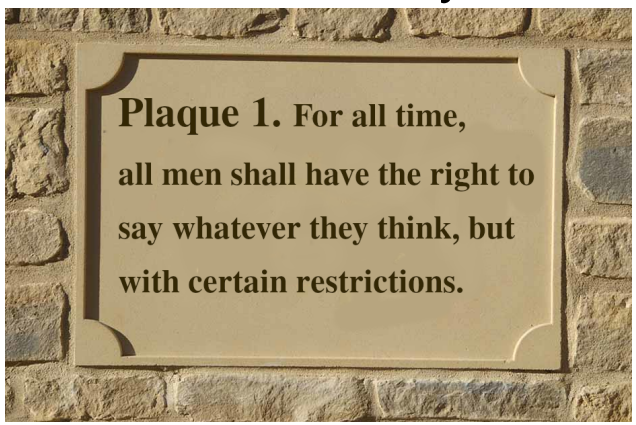
There has been a lot of churn in this area:

+ Lemahieu, Broucke & Baesens pp. 275 notes Xpath's ability to return items at different levels requires recursive SQL to express (next lecture).
+ In the noughties, a large number of new, XML- and web-related standards were defined, *eg*. RDF, OWL, YAML, SOAP, XMLRPC...
- Although computing power and network bandwidth were becoming cheaper, the move to human-readable representations has lead to an order-of-magnitude inflation in data size and parsing overhead compared with binary data exchange.
- Many traditional SQL-based systems were extended with NoSQL features. Likewise, many NoSQL systems were extended with traditional SQL features.
- Is ChatGPT a database?

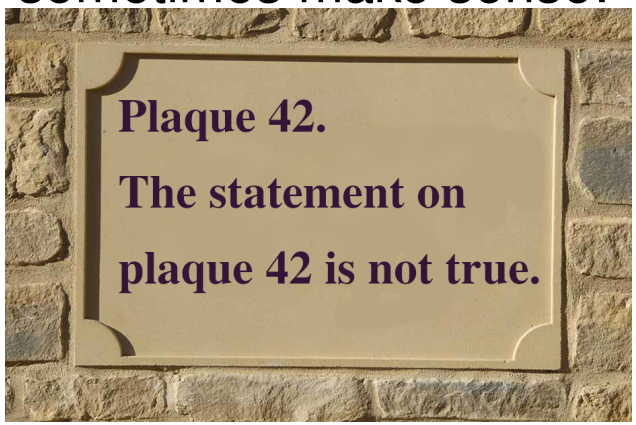
NB: For document database Tripos questions, a well-argued answer can garner full credit, even if completely disagreeing with the expected answer.

Lecture 7 - Further SQL

Declarations always hold:



Recursive declarations sometimes make sense:



(Hmm, no fixed point.)

Another look at SQL

- Complexity of join.
- What is a database index?
- Two complications for SQL semantics
  - Multi-sets (bags)
  - NULL values
- Transitive computations: Erdős (Kevin Bacon) numbers.
- Recursive SQL.

Complexity of a Join?

Given tables $R(\mathbf{A}, \mathbf{B})$ and $S(\mathbf{B}, \mathbf{C})$, how much work is required to compute the join $R \bowtie S$?

```
// Brute force appaoch:
// scan R
for each (a, b) in R {
    // scan S
    for each (b', c) in S {
        if b = b' then create (a, b, c) ...
    }
}
```

Worst case: requires on the order of $|R| \times |S|$ steps. But note that on each iteration over R, there may be only a very small number of matching records in S — only one if R's B is a foreign key into S.

We have already spoken of a table having an index.

An **index** is a data structure — created and maintained within a database system — that can greatly reduce the time needed to locate records.

```
// scan R
for each (a, b) in R {
   // don't scan S, use an index
   for each s in S-INDEX-ON-B(b) {
      create (a, b, s.c) ...
   }
```

- *Ia Algorithms* presents useful data structures for implementing database indices (search trees, hash tables and so on).
- The foreign key lookup can be performed in $\propto \log |S|$ instructions instead of $\propto |S|$ (linear).

Remarks

Typical SQL commands for creating and deleting an index:

```
CREATE INDEX index_name on S(B)

DROP INDEX index_name
```

- There are many types of database indices and the commands for creating them can be complex.
- Index creation is not defined in the SQL standards. It can sometimes be done by a specialist team or automated.
- **While an index can speed up reads, it will slow down updates.** This is one more illustration of a fundamental database tradeoff.
- The tuning of database performance using indices is a fine art.
- In some cases it is better to store read-oriented data in a separate database optimised for that purpose.

Why the `distinct` in the SQL?

The SQL query

```
select B, C from R
```

will produce a bag (multiset)!

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	$\star\star\star$
10	0	$\star\star\star$
99	17	
25	4	

SQL is actually based on multisets, not sets.

Why Multisets?

Duplicates are important for aggregate functions (min, max, ave, count, and so on). These are typically used with the **GROUP BY** construct.

sid	course	mark
ev77	databases	92
ev77	spelling	99
tgg22	spelling	3
tgg22	databases	100
fm21	databases	92
fm21	spelling	100
jj25	databases	88
jj25	spelling	92

group by $\Longrightarrow$

course	mark
spelling	99
spelling	3
spelling	100
spelling	92

course	mark
databases	92
databases	100
databases	92
databases	88

Visualizing the aggregate function **min**

course	**mark**
spelling	99
spelling	3
spelling	100
spelling	92

course	**mark**
databases	92
databases	100
databases	92
databases	88

$\min(\textbf{mark}) \Longrightarrow$

course	$\min(\textbf{mark})$
spelling	3
databases	88

Looking at this in SQL

```
select course,
       min(mark),
       max(mark),
       avg(mark)
from marks
group by course;
```

```
+-----------+-----------+-----------+-----------+
| course    | min(mark) | max(mark) | avg(mark) |
+-----------+-----------+-----------+-----------+
| databases |        88 |       100 |   93.0000 |
| spelling  |         3 |       100 |   73.5000 |
+-----------+-----------+-----------+-----------+
```

What is NULL?

- NULL is not the empty string "".
- NULL is a place-holder, not a value!
- NULL is not a member of any domain (type),
- This means we need three-valued logic.

Let $\perp$ represent **we don't know!**

$\wedge$	**T**	**F**	$\perp$
T	**T**	**F**	$\perp$
F	**F**	**F**	**F**
$\perp$	$\perp$	**F**	$\perp$

$\vee$	**T**	**F**	$\perp$
T	**T**	**T**	**T**
F	**T**	**F**	$\perp$
$\perp$	**T**	$\perp$	$\perp$

v	$\neg v$
T	**F**
F	**T**
$\perp$	$\perp$

[*NB:* Similar logic systems and lattices are used in many areas of computer science, such as digital logic simulation (Part Ib Verilog) or checking whether an expression is constant (Part II Optimising Compilers).]

NULL can lead to unexpected results

```
select * from students;
+------+--------+------+
| sid  | name   | age  |
+------+--------+------+
| ev77 | Eva    |   18 |
| fm21 | Fatima |   20 |
| jj25 | James  |   19 |
| ks87 | Kim    | NULL |
+------+--------+------+

select * from students where age <> 19;
+------+--------+------+
| sid  | name   | age  |
+------+--------+------+
| ev77 | Eva    |   18 |
| fm21 | Fatima |   20 |
+------+--------+------+
```

The ambiguity of NULL

Possible interpretations of NULL

- There is a value, but we don't know what it is.
- No value is applicable.
- The value is known, but you are not allowed to see it.
- ...

A great deal of semantic muddle is created by conflating all of these interpretations into one non-value.

> *"I don't have a sister, and nor does my friend. If "NULL = NULL" then we have a common sister, and are therefore related!" — Matt Hamilton, 2009.*

Avoided by SQL equality definition: 'NULL is not equal (=) to anything — not even to another NULL.'

On the other hand, introducing distinct NULLs for each possible interpretation leads to very complex logics ...

SQL's `NULL` has generated endless controversy

C. J. Date [D2004], Chapter 19

"Before we go any further, we should make it very clear that in our opinion (and in that of many other writers too, we hasten to add), NULLs and 3VL are and always were a serious mistake and have no place in the relational model."

In defense of Nulls, by Fesperman

"[...] nulls have an important role in relational databases. To remove them from the currently **flawed** SQL implementations would be throwing out the baby with the bath water. On the other hand, the **flaws** in SQL should be repaired immediately" [web: Are Nulls Evil?].

How can we select on null then?

With our small database, the query

```
SELECT note FROM credits WHERE note IS NULL;
```

returns 4892 records of `NULL`.

The SQL 'IS NULL' predicate:

Being a predicate, the expression '`foo IS NULL`' is either true or false

- true when `foo` is the `NULL` value,
- false otherwise.

[*NB:* There is also the 'IS NOT NULL' predicate in SQL, which returns the opposite value (negated answer).]

Flaws? One example of SQL's inconsistency.

Furthermore, the query

```
SELECT note, count(*) AS total
FROM credits
WHERE note IS NULL GROUP BY note;
```

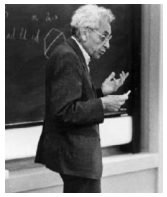
returns a single record

```
note total
---- -----
NULL 4892
```

We have one group. This seems to mean that `NULL` is equal to `NULL`. But we have defined that NULL is not equal to NULL!

[*NB:* Infact, 'NULL = NULL' returns 'NULL'.]

[Erdős or] Bacon Number




P. Erdős (maths) and K. Bacon (acting) are the origins. We'll ignore maths.

- Kevin Bacon has Bacon number 0.
- Anyone acting in a movie with Kevin Bacon has Bacon number 1.
- For any other actor, their Bacon number is calculated as follows. Look at all of the movies the actor acts in. Among all of the associated co-actors, find the smallest Bacon number k. Then the actor has Bacon number $k + 1$.

Let's try to calculate Bacon numbers using SQL.

First, what is Kevin Bacon's person_id?

```
select person_id from people where name = 'Kevin Bacon';
```

Result is "nm0000102".

Function composition and relation composition

Function composition operator:

Given two functions, f and g,

- If $f(g(x)) = y$ then $(f \circ g)(x) = y$ (mathematics definition).
- `let compose (f, g) = fun x -> f(g x)` (ML definition).

Relation composition operator:

Given two binary relations

$$R \subseteq S \times T$$
$$Q \subseteq T \times U$$

their composition is $Q \circ R \subseteq S \times U$ where

$$Q \circ R \equiv \{(s,\ u) \mid \exists t \in T.(s,\ t) \in R \wedge (t,\ u) \in Q\}$$

[*Aside:* In some ML dialects, the circle operator is built in, for example 'o' in standard ML and '»' in F#.]

Partial functions as relations

Functions of one argument are special cases of relations:

- A relation R where, if $(s,\ t_1) \in R$ and $(s,\ t_2) \in R$ implies that $t_1 = t_2$, defines a **function** (could be total or partial).

- Hence, the composition of functions is a special case of the composition of relations.
- The definition of $\circ$ for relations and functions is equivalent for relations that represent functions.

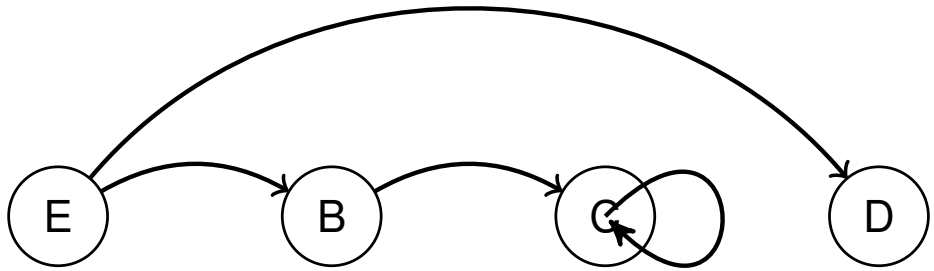
If we write $Q \circ R$ as $R \bowtie_{2=1} Q$ we see that **joins are a generalisation of function composition**; generalised in that they cope with relations and not just functions.

[*NB:* When mathematicians speak of 'functions' they mean total functions: those which give a single result for every value in their domain. A partial function, on the other hand, may not be defined for some input values. A relation can give multiple 'answers' for the same 'input'.]

Directed Graphs

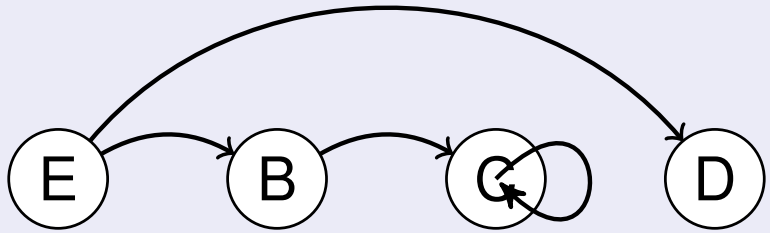
- $G = (V,\ A)$ is a **directed graph**, where
- V a finite set of **vertices** (also called **nodes**).
- A is a binary relation over V. That is $A \subseteq V \times V$.
- If $(u,\ v) \in A$, then we have an **arc** from u to v.
- The arc $(u,\ v) \in A$ is also called a directed edge, or a **relationship of u to v**.

$$V = \{E, B, C, D\}$$
$$A = \{(E,\ B),\ (E,\ D),\ (B,\ C),\ (C,\ C)\}$$

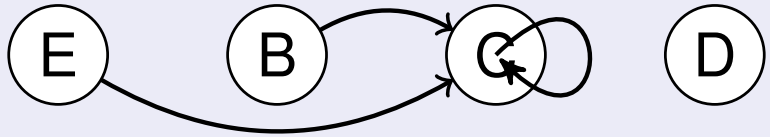


Composition example

A = {(E, B), (E, D), (B, C), (C, C)}



$A \circ A = \{(E,\ C),\ (B,\ C),\ (C,\ C)\}$



Elements of $A \circ A$ represent paths of length 2

- $(E,\ C) \in A \circ A$ by the path $E \to B \to C$
- $(B,\ C) \in A \circ A$ by the path $B \to C \to C$
- $(C,\ C) \in A \circ A$ by the path $C \to C \to C$

Iterated composition and paths.

Suppose R is a binary relation over S, $R \subseteq S \times S$. Define **iterated composition** as

$$\begin{aligned} R^1 &\equiv R \\ R^{n+1} &\equiv R \circ R^n \end{aligned}$$

Let $G = (V,\ A)$ be a directed graph. Suppose $v_1,\ v_2, \cdots v_{k+1}$ is a sequence of vertices. Then this sequence represents a **path in** G **of length** k when $(v_i,\ v_{i+1}) \in A$, for $i \in \{1, 2, \cdots k\}$. We will often write this as

$$v_1 \rightarrow v_2 \rightarrow \cdots v_k$$

Observation

If $G = (V,\ A)$ is a directed graph, and $(u,\ v) \in A^k$, then there is at least one path in G from u to v of length k. Such paths may contain loops.

Shortest path

Definition of R-distance (hop count)

Suppose $s_0 \in \pi_1(R)$ (*ie.* there is a pair $(s_0,\ s_1) \in R$).

- The distance from s_0 to s_0 is defined as 0.
- If $(s_0,\ s_1) \in R$, then the distance from s_0 to s_1 is 1.
- For any other $s' \in \pi_2(R)$, the distance from s_0 to s' is the least n such that $(s_0,\ s') \in R^n$.

We will think of the Bacon number as an R-distance where s_0 is Kevin Bacon. But what is R?

[*NB:* By π_1 we mean extracting the first field, since π_k is the k^{th} projection function.]

[*NB:* This is the 'single-source' shortest path problem. *Algorithms Ia* also considers all-sources shortest path problem.]

Let *R* be the co-actor relation

```
DROP VIEW  IFEXISTS coactors;

CREATE VIEW coactors AS
  SELECT DISTINCT p1.person_id AS pid1,
                  p2.person_id AS pid2
  FROM plays_role AS p1
  JOIN plays_role AS p2 ON p2.movie_id = p1.movie_id
;
```

On our database, this relation contains 18,252 rows. Note that this relation is **reflexive** and **symmetric**.

[*NB:* Recall the DISTINCT keyword eliminates duplicates from the default multi-set.]

SQL: Bacon number 1

```
DROP VIEW IF EXISTS bacon_number_1;

CREATE VIEW bacon_number_1 AS
 SELECT DISTINCT pid2 AS pid,
                 1 AS bacon_number
 FROM coactors
 WHERE pid1 = 'nm0000102' AND pid1 <> pid2;
```

Remember Kevin Bacon's person_id is nm0000102.

SQL: Bacon number 2

```
DROP VIEW IF EXISTS bacon_number_2;

CREATE VIEW BACON_number_2 AS
 SELECT DISTINCT ca.pid2 AS pid,
                 2 AS bacon_number
 FROM bacon_number_1 AS bn1
 JOIN coactors AS ca ON ca.pid1 = bn1.pid
 WHERE ca.pid2 <> 'nm0000102' AND
 NOT(ca.pid2 IN (SELECT pid FROM bacon_number_1));
```

SQL: Bacon number 3

```
DROP VIEW IF EXISTS bacon_number_3;

CREATE VIEW bacon_number_3 AS
 SELECT DISTINCT ca.pid2 AS pid,
                 3 AS bacon_number
 FROM bacon_number_2 AS bn2
 JOIN coactors AS ca ON ca.pid1 = bn2.pid
 WHERE ca.pid2 <> 'nm0000102' AND
   NOT(ca.pid2 IN (SELECT pid FROM bacon_number_1))
 AND
   NOT(ca.pid2 IN (SELECT pid FROM bacon_number_2));
```

You get the idea...
Let's do this all the way up to bacon_number_9.

SQL: Bacon number 9

```
DROP VIEW IF EXISTS bacon_number_9;

CREATE VIEW bacon_number_9 AS
 SELECT DISTINCT ca.pid2 AS pid,
                 9 AS bacon_number
 FROM bacon_number_8 AS bn8
 JOIN coactors AS ca ON ca.pid1 = bn8.pid
 WHERE ca.pid2 <> 'nm0000102'
 AND NOT(ca.pid2 in (SELECT pid FROM bacon_number_1))
 AND NOT(ca.pid2 in (SELECT pid FROM bacon_number_2))
 AND NOT(ca.pid2 in (SELECT pid FROM bacon_number_3))
 AND NOT(ca.pid2 in (SELECT pid FROM bacon_number_4))
 AND NOT(ca.pid2 in (SELECT pid FROM bacon_number_5))
 AND NOT(ca.pid2 in (SELECT pid FROM bacon_number_6))
 AND NOT(ca.pid2 in (SELECT pid FROM bacon_number_7))
 AND NOT(ca.pid2 in (SELECT pid FROM bacon_number_8));
```

SQL: Bacon numbers

```
DROP VIEW IF EXISTS bacon_numbers;

CREATE VIEW bacon_numbers AS
  SELECT * FROM bacon_number_1
  UNION
  SELECT * FROM bacon_number_2
  UNION
  SELECT * FROM bacon_number_3
  UNION
  SELECT * FROM bacon_number_4
  UNION
  SELECT * FROM bacon_number_5
  UNION
  SELECT * FROM bacon_number_6
  UNION
  SELECT * FROM bacon_number_7
  UNION
  SELECT * FROM bacon_number_8
  UNION
  SELECT * from bacon_number_9 ;
```

Bacon Numbers, counted

```
SELECT bacon_number, count(*) AS total
FROM bacon_numbers
GROUP BY bacon_number
ORDER BY bacon_number;
```

Results

BACON_NUMBER	TOTAL
1	12
2	110
3	614
4	922
5	381
6	123
7	86
8	16

bacon_number_9 is empty!

Transitive closure

Suppose R is a binary relation over S, $R \subseteq S \times S$. The **transitive closure of** R, denoted R^+, is the smallest binary relation on S such that $R \subseteq R^+$ and $R+$ is **transitive**. R^+ being transitive means:

$$(x,\ y) \in R^+ \wedge (y,\ z) \in R^+ \implies (x,\ z) \in R^+.$$

Then

$$R^+ = \bigcup_{n \in \{1,\ 2,\ \cdots\}} R^n.$$

- Happily, all of our relations are **finite**, so there must be some k with
$$R^+ = R \cup R^2 \cup \cdots \cup R^k.$$
- Sadly, k will depend on the contents of R!
- Conclude: we **cannot** compute transitive closure in the Relational Algebra (or SQL without recursion).

A ‘let rec’ for SQL enables recursion.

The WITH keyword in SQL allows a recursive declaration:

Does this have a least-fixed-point?

```
WITH R AS (SELECT 1 AS n)
SELECT n + 1 FROM R;
```

How about this one?

```
WITH countUp AS (SELECT 1 AS n
        UNION ALL SELECT n + 1 FROM countUp WHERE n<3)
SELECT * FROM countUp;
```

[Recusive SQL not examinable in 22/23]. [web:SWLH]).

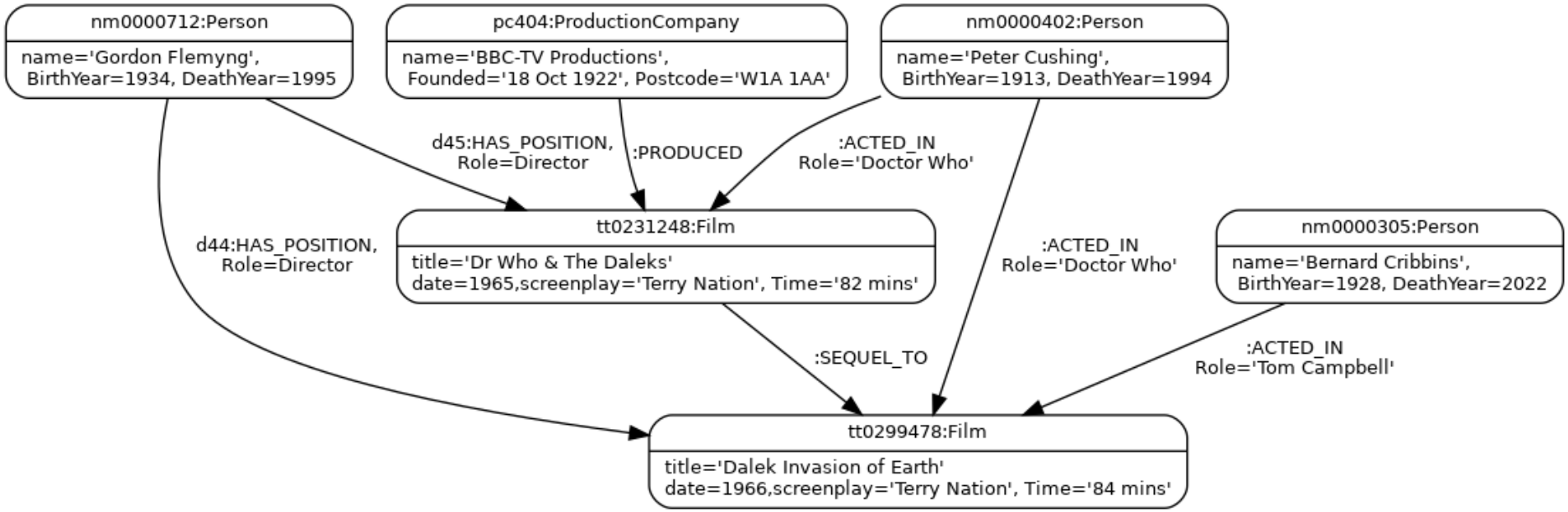
Recursive Bacon SQL query

A fine student answer from jp2002 (22nd Nov 2022):

```
WITH RECURSIVE bacon(n,pid) AS
  (SELECT 0 AS n, pid2 AS pid FROM coactors
            WHERE pid1='nm0000102' AND pid1=pid2
   UNION
   SELECT n+1 AS n, c.pid2 AS pid FROM bacon
   JOIN coactors AS c ON c.pid1 = pid WHERE
      NOT(c.pid2 IN (SELECT pid FROM bacon)) AND n < 20
   ) SELECT n, COUNT(*)
  FROM (SELECT min(n) AS n, pid FROM bacon GROUP BY pid)
  GROUP BY n;
```

Boggle! Efficiency? This will be **much** easier in a graph database.

Lecture 8: Graph-oriented Databases



Typically one big graph is stored (instance of an E/R diagram?)

- Nodes have a type, a unique label (or several in Neo4J) and properties.
- Edges are directed between two nodes. They have a type, optional label and properties.
- Can collate by type to convert to rDBMS tables.

We could simply store graphs in relational tables?

NODES

Town	Country
Rome	Italy
Verona	Italy
Bognor	UK
Paris	France
Romsey	UK

EDGES

EID	V1	V2	Form	Distance
L1	Rome	Verona	Bus	12
L2	Verona	Paris	Plane	181
L3	Bognor	Romsey	Bus	33
L4	Romsey	Paris	Teleport	Null
L5	Paris	Bognor	Plane	125

This is a unary relation: the schema range and domain type are both towns.

This is a small example. Think of a million nodes and considerably more edges.

- One table for nodes and one for edges?
- Need to name the edges (EID often artificial?).
- Inefficient:
  - All edges must be scanned to find the neighbour of a node.
  - The ends are interchangable for **undirected** searches, so two fields to examine.
  - Queries involving many hops are painful in SQL (especially Kleene star [Part Ia *Algorithms*]).
  - Will typically need to store two inverted indexes to the edges relation.

Binary and higher relations: one rDBMS table per node type?

- To avoid an EID, here the edges table is **all-key**.
- rDBMS is not ideal for enormous, many-to-many relations.
- For OLAP, a denormal representation would probably be used.
- This binary relation is **bipartite**: two types of node; all edges go from one type to the other.

TOWNS

Town	Population
Rome	343
Verona	33
Bognor	2
Paris	312
Brussels	201

Edges relation

OFFICIAL_LANGUAGE

Town	Language
Rome	Italian
Bognor	English
Paris	French
Brussels	French
Brussels	Flemish
Brussels	German

LANGUAGES

Language	Core Vocab	Genders
Italian	500,000	2
English	1,600,000	3
French	135,000	2
Flemish	300,000	2.5
German	200,000	3

Modelling ternary relations?

- Edges have two ends.
- Earlier we stored Terry Nation as an attribute value.
- Was the screenplay author a person? An attribute value may be a **foreign key**.
  - Is this a good schema? Edges from edge attributes?

Neo4j: Cypher immediate data entry.

Data is typically imported from external sources, but ...

Immediate Node Data:

```
CREATE (nm0000102:Person {name: 'Kevin Bacon', birthyear:1958, deathyear:Null})
CREATE (nm0002002:Person {name: 'Sean Connery', birthyear:1954, deathyear:2007})
CREATE (nm0012032:Person {name: 'Roger Moore', birthyear:1927, deathyear:2017})
CREATE (tt0299478:Movie {title:'Dr No', screenplay='Richard Maibaum', Time='109 mins'})
CREATE (tt0299479:Movie {title:'Thunderball', screenplay='Richard Maibaum', Time='130 mins'})
```

Immediate Edge Data:

```
CREATE (nm0002002)-[:ACTED_IN {Role:'James Bond'}]->(tt0299478)
CREATE (nm0002002)-[:ACTED_IN {Role:'James Bond'}]->(tt0299479)
```

- Edges and nodes have <primary name>:<type> and then key/value properties.
- All edges have a direction as stored.

Graph data normalisation.

Do we want the role name to be the arc name?

```
(nm0000084)-['Su Li-zhen':PLAYS_ROLE]->(tt0212712)
(nm0000090)-['Semyon':PLAYS_ROLE]->(tt0765443)
(nm0000093)-['Mickey O'Neil':PLAYS_ROLE]->(tt0208092)
```

Hmm!

- Arc names must be unique.
- Modelling mistake since the same role name will appear in remakes between different actors and movies.

Better:

```
(nm0000084)-[:PLAYS_ROLE {role:'Su Li-zhen'}]->(tt0212712)
```

Databases and Graph Databases

General points:

- An arc type essentially models an E-R binary (or unary) relation.
- Pattern matching on paths is supported.
- Transitive closure is free ...
- ... many other common graph algorithms supported ...

Neo4J specific:

- Edges, when created, need have no identifiers, so create is not idempotent?
- Edges, as entered, are directed, but queries can treat them as un-directed.
- Queries can be expressed as reusable functions with formal parameters (equally possible for rDBMS).
- Suffered some serious security vulnerabilities last year (equally possible for rDBMS).
- Regular expressions on values violate value atomicity (yes, widely done in SQL too!).

(**Idempotent** operation) $\iff$ (repeating it has no effect).

Neo4j — example pattern-matching queries:

Path patterns contain constants and/or bind local variables `a`, `b` ...

`(a)-->(b)` All pairs of nodes with an edge from one to the other.	`(a:Person)-->(b:Movie)` Any type of edge between any person and any film.	`(*)-[:ACTED_IN]->(b)` Nodes at the end of any edge of type `ACTED_IN`.
`(a)--(b)` Any pair of nodes with an edge between them in either direction.	`(a)--(b)--(c)-->(d)` Four (distinct? a=c?) connected nodes.	`(a)-[:ACTED_IN]->(b)` All pairs related by `ACTED_IN`.
`(*)-->(a)<--(*)` Any node with two or more incoming edges.	`(a:Person {name:'Madonna'})-->(*:Movie {title:t})` Node attribute matching and binding.	`(a:Person)-[:ACTED_IN*]->(b)` Transitive matching.

The Kleene star matches a path of any length. Further syntax upper and/or lower bounds the path length: *eg*. `(a)-[*3..5]->(b)`.

Pattern matching. Example 1:

```
MATCH
      (john {name: 'John'})-[:FRIEND]->()-[:FRIEND]->(fof)
RETURN john.name, fof.name
```

Resulting in:

```
+----------------------+
| john.name | fof.name |
+----------------------+
| "John"    | "Maria"  |
| "John"    | "Steve"  |
+----------------------+
2 rows
```

Friendship should surely be symmetric; shouldn't John be his own FOF?

[neo4j.com/docs/cypher-manual/current/introduction].

Pattern matching. Example 2: co-actors

Get all co-actors with

```
MATCH (p1:Person) -[:ACTED_IN]-> (m:Movie),
      (p2:Person) -[:ACTED_IN]-> (m:Movie)
WHERE p1.person_id <> p2.person_id
RETURN p1.name AS name1, p2.name AS name2, count(*) AS TOTAL
ORDER BY total desc, name1, name2
LIMIT 10;
```

OR

```
MATCH (p1:Person) -[:ACTED_IN]-> (m:Movie) <-[:ACTED_IN]- (p2:Person)
WHERE ...
```

OR

```
MATCH (p1:Person) -[:ACTED_IN*2]- (p2:Person)
WHERE ...
```

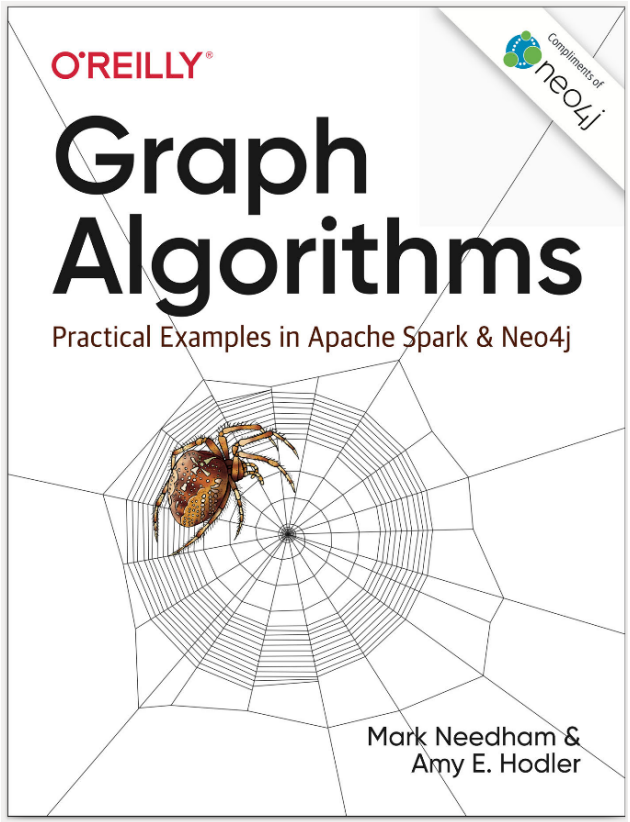
Resulting in:

```
+-------------------------------------------------+
| name1                | name2                | total |
+-------------------------------------------------+
| "Daniel Radcliffe" | "Rupert Grint"     | 8     |
| "Kohl Sudduth"     | "Tom Selleck"      | 8     |
| "Rupert Grint"     | "Daniel Radcliffe" | 8     |
| "Tom Selleck"      | "Kohl Sudduth"     | 8     |
```

Graph Algorithms (are important)

Desire efficient support for a large number of graph algorithms and metrics.

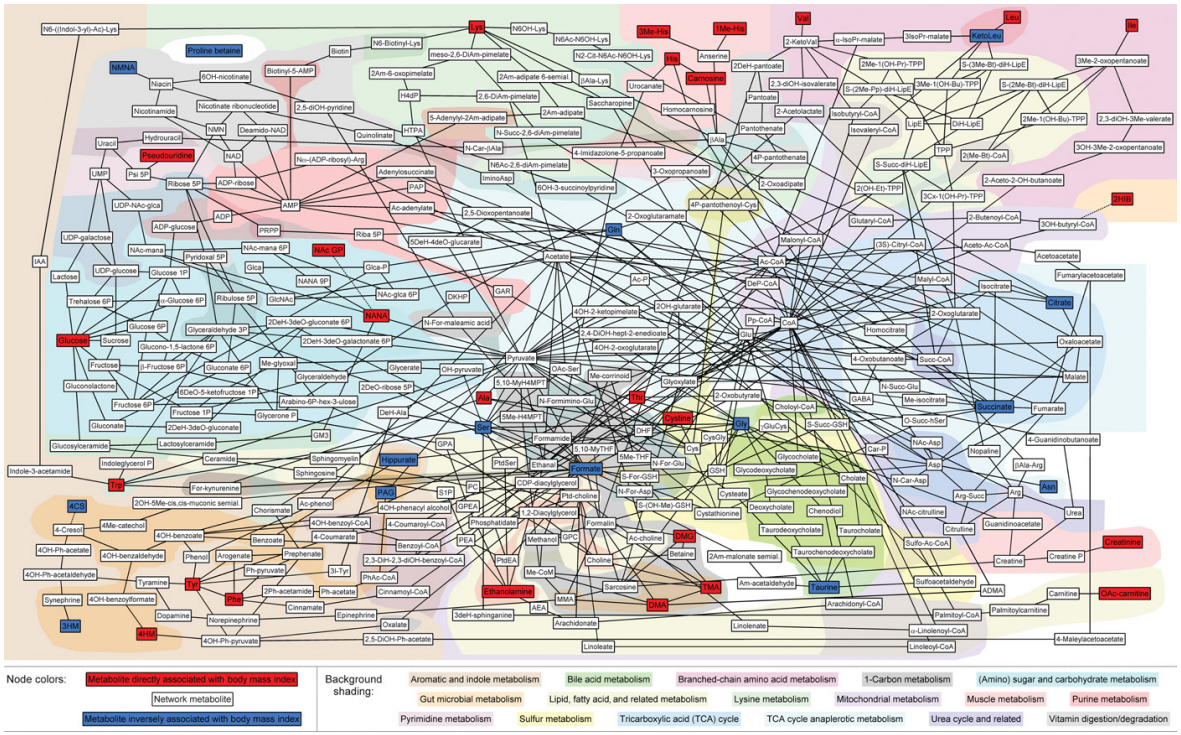
- Breadth-first search, depth-first, shortest path, Page Rank, spanning trees, articulation point, strongly-connected components, cliques, max flow ...



Metrics:

- **Community:** Edge/node ratio, diameter, how are nodes clustered, tree count...
- **Centrality:** How important is each node or link to the structure of the entire graph.
- **Similarity:** How alike are two or more nodes?
- **Prediction:** How likely is it that a new arc will be formed between two nodes?
- **Path finding:** What is the “best” path between two nodes?

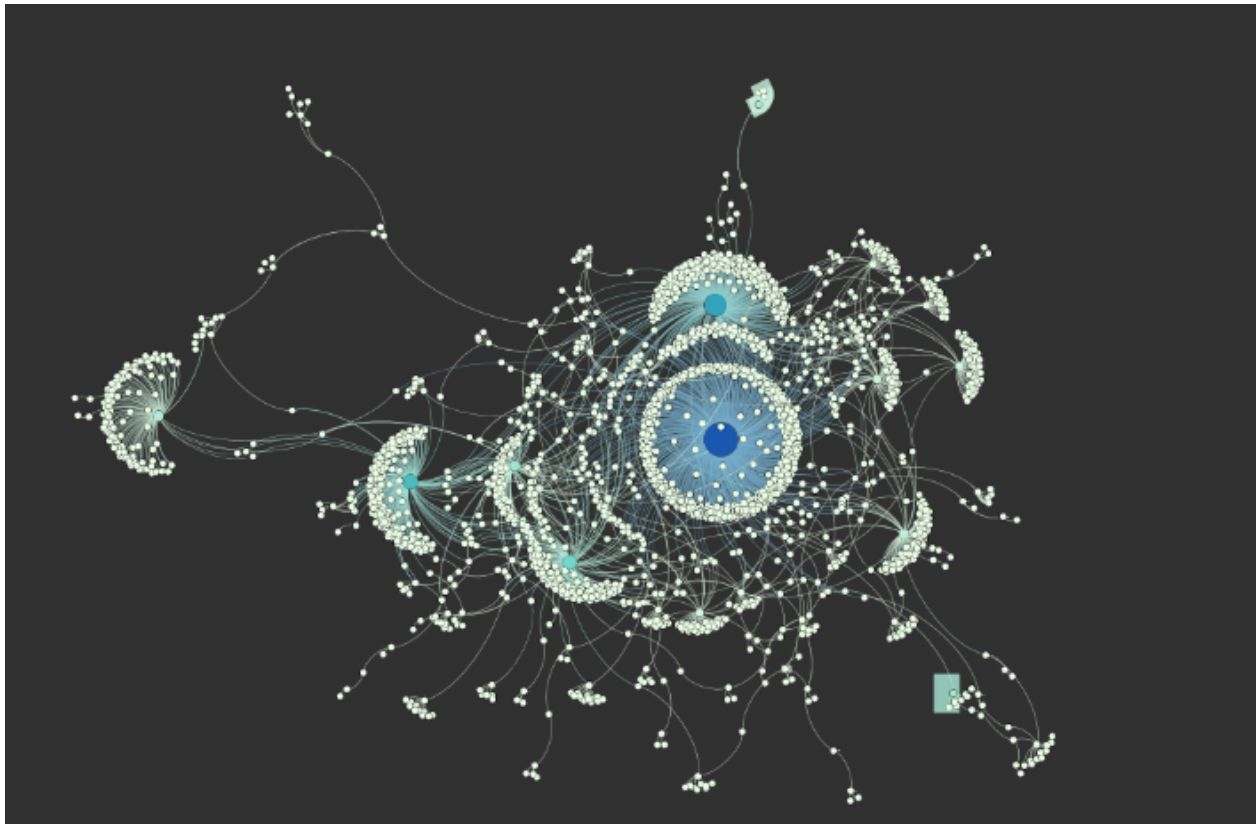
Graph DBMS optimised for Big Data (will not fit in core*)“Data Science” queries.



- This is a **small** metabolic network from **Urinary metabolic signatures of human adiposity (2015)** [web].
- Many biological networks derived from experiments have millions of nodes and edges.
- Biologist interested in drug development “query” such graphs to find important structures.

* = An historic term for data being entirely stored in primary memory.

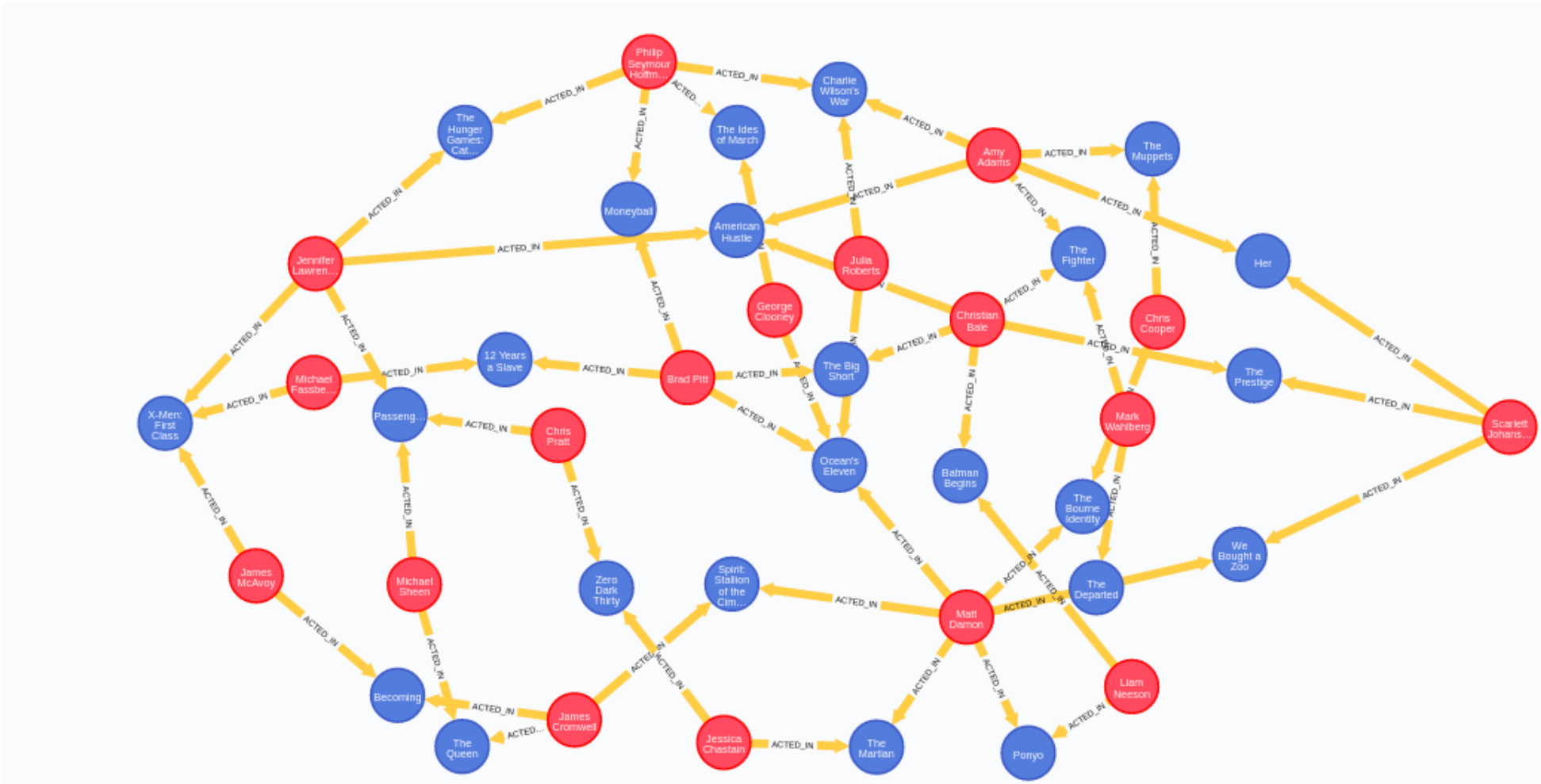
Social networks



- From **Building Social Network Visualizations** [web:sfm-ui].
- Graph algorithms are used to recommend new friend links.

Neo4j: Example of path-oriented query in Cypher

```
MATCH path=allshortestpaths((m:Person {name : 'Jennifer Lawrence'} )
                            -[:ACTED_IN*]-
                            (n:Person {name : 'Matt Damon'}))
RETURN path
```



Let's count Bacon numbers with Neo4j/Cypher

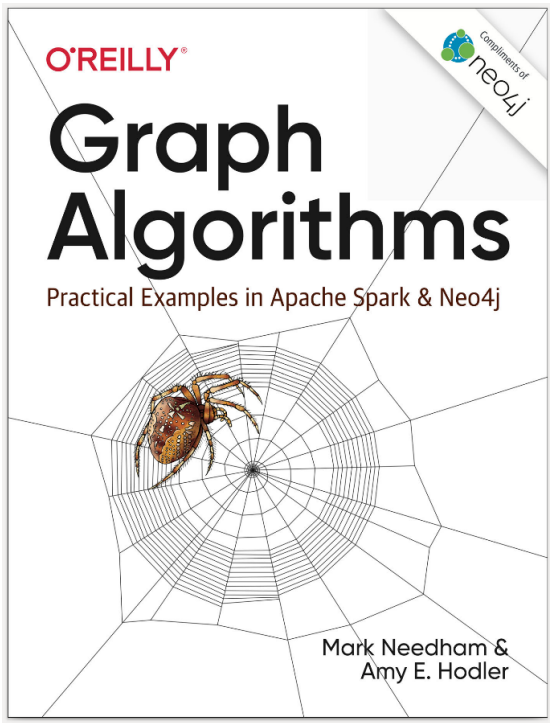
```
MATCH paths=allshortestpaths(
      (m:Person {name : "Kevin Bacon"} )
         -[:ACTED_IN*]-
      (n:Person))
WHERE n.person_id <> m.person_id
RETURN length(paths)/2 AS bacon_number,
       COUNT(distinct n.person_id) AS total
ORDER BY bacon_number;
```

bacon_number	total
1	15
2	314
3	977
4	685
5	145
6	60
7	20
8	28
9	6

Graph-oriented DBMS optimisations:

- In-core* databases can use pointers to implement referential links.
- Big-data implementations will stream the edges past processing elements (Pregel).

Convergence: Many SQL systems are optimising in-core table sets in the same similar ways (fighting back) and users typically want SQL-like access to node data.



[*NB:* This 'Graph Algorithms' book is available via the course web site. Many algorithms overlap with *Ia Algorithms*, but most content is irrelevant for this course.]

Last Slide!

What have we learned?

- Having a conceptual model of data is very useful, no matter which implementation technology is employed.
- Investment in data model planning pays off well.
- There is a trade-off between fast reads and fast writes.
- There is no database system that satisfies all possible requirements.
- Staging between a principle storage model used for updates and optimised views, clones or other alternatives for rapid query is commonly used.
- It is best to understand pros and cons of each approach and develop integrated solutions where each component database is dedicated to doing what it does best.
- The future will see enormous churn and creative activity in the database field!