Databases

Timothy G. Griffin

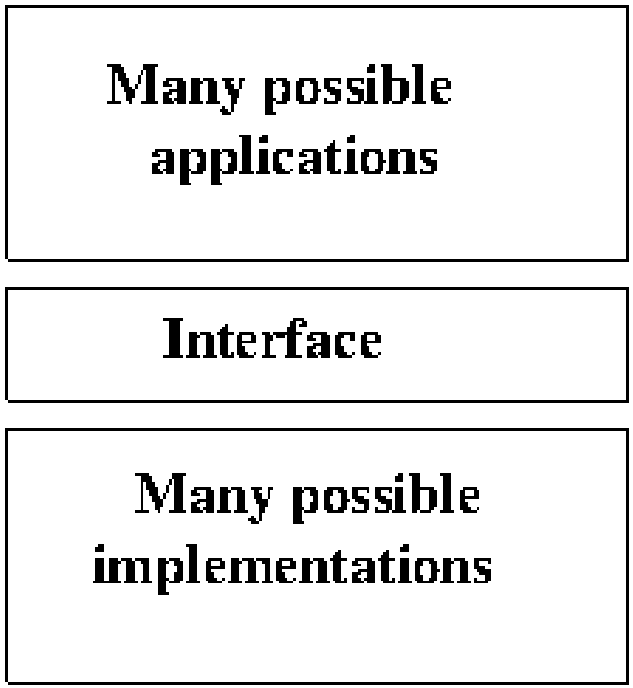
Computer Laboratory
University of Cambridge, UK

Michaelmas 2020

Lecture 1

- What is a Database Management System (DBMS)?
- 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/DBMS can solve all problems.

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 present a serious challenge to this idealised picture!

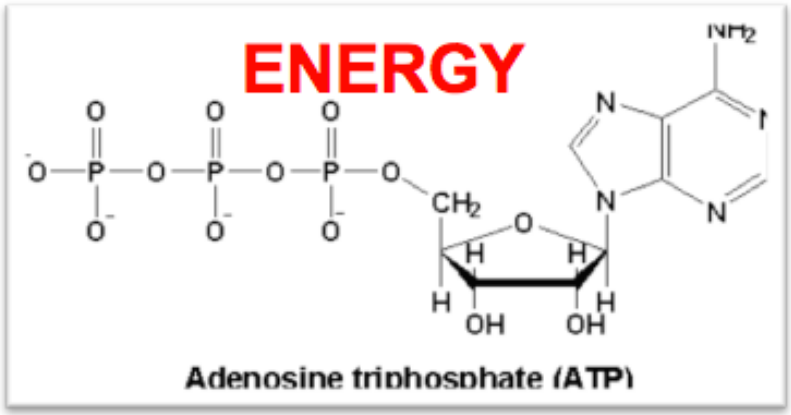
This is found everywhere, not just in computing ...





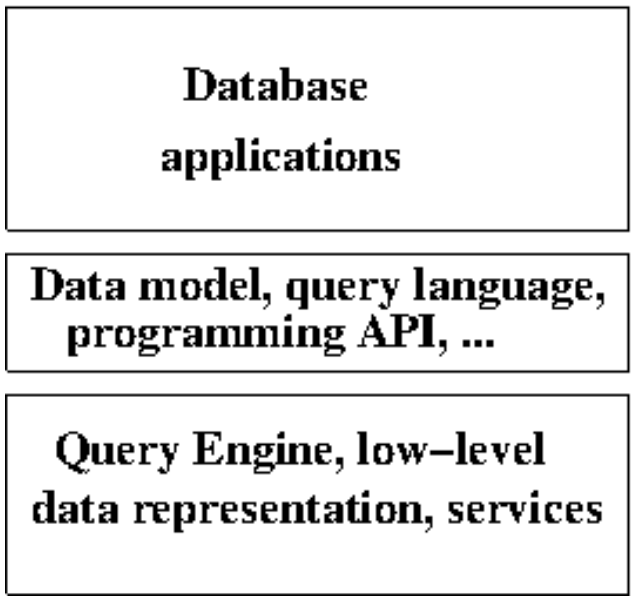


Evolution worked it out long long ago!



Photosynthesis ■ ■ ■ ■ ■ Cellular oxidation

What is a Database Management System (DBMS)?



- **This course will present databases from an application writer's point of view. It will stress data models and query languages.**
- We will not cover programming APIs, network APIs, or low-level implementation details.
- A query engine includes an optimiser that knows about low-level details hidden by the interface(s).
- The services typically implemented by a DBMS:
  - CRUD operations,
  - ACID (or BASE?) transactions.

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.

ACID transactions for concurrent updates

Atomicity: Either all actions of a transaction are carried out, or none are (even if the system crashes in the middle of a transaction).

Consistency: Every transaction applied to a consistent database leave it in a consistent state.

Isolation: Transactions are isolated, or protected, from the effects of other concurrently executed transactions.

Durability: If a transactions completes successfully, then its effects persist.

Implementing ACID transactions is one topic covered in Concurrent and Distributed Systems (1B).

Traditional Relational Databases

Based on SQL standards and ACID transactions. Data normally resides in secondary storage.

Commercial

- Oracle, IBM, and Microsoft together have over 85% of the commercial market.

Open source or free-ware

- HyperSql
- MySQL
- SQLite
- PostgreSQL

https://hostingdata.co.uk/nosql-database lists over 250 “NoSQL” Systems

Your Ultimate Guide to the Non-Relational Universe!

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!]

LIST OF NOSQL DATABASE MANAGEMENT SYSTEMS [currently >225]

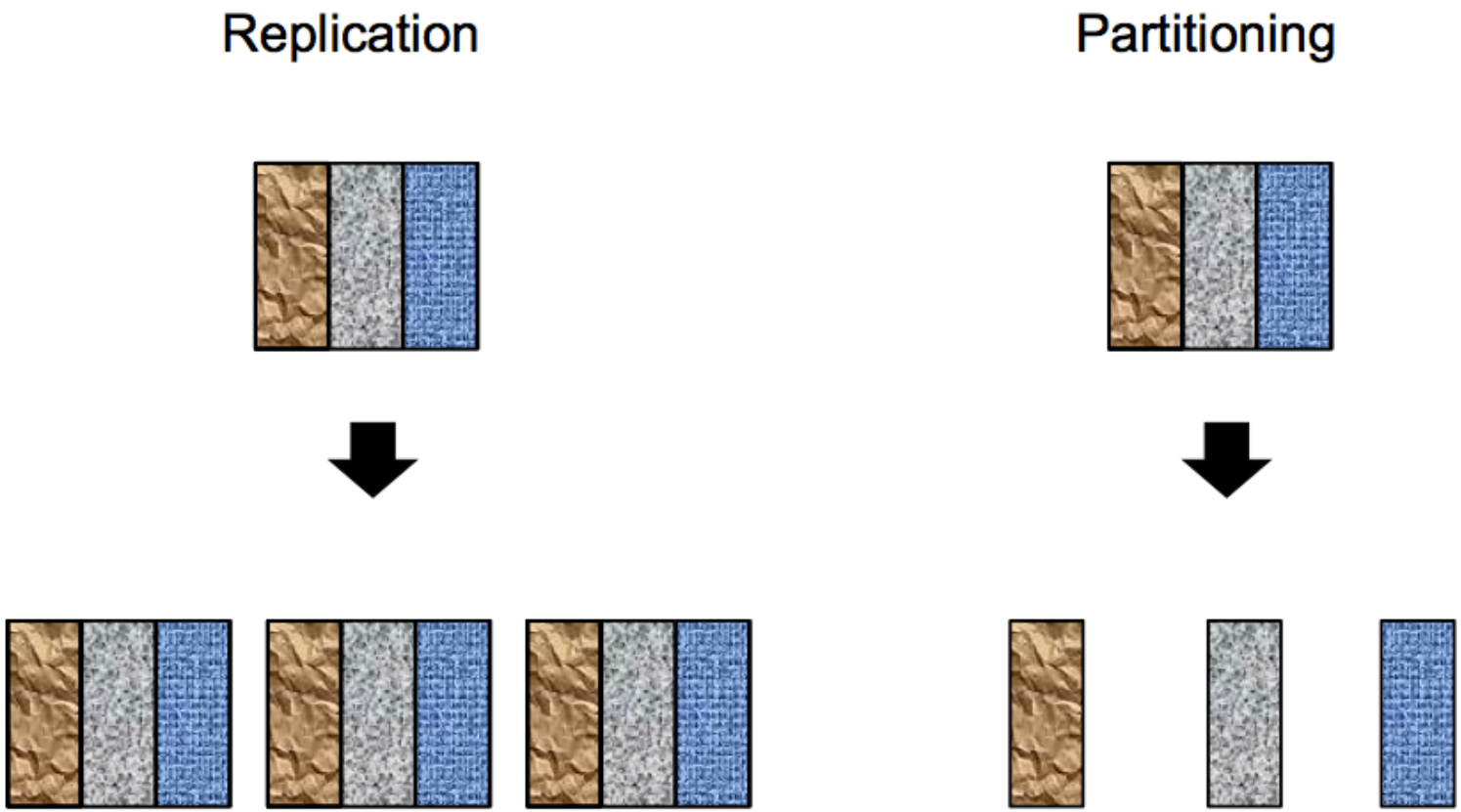
NoSQL systems are typically distributed databases

Why distribute data?

- **Scalability**. The data set or the workload can be too large for a single machine. Data often resides in RAM, rather than secondary storage.
- **Fault tolerance**. The service can survive the failure of some machines.
- **Lower Latency**. Data can be located closer to widely distributed users.

Distributed databases are an important technology supporting cloud computing.

How do we distribute the data?



Note: partitions themselves are often replicated.

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 very hard (impossible?) to achieve all three in a highly distributed database.

CAP principle

In a highly distributed system:

- Assume that network partitions and other connectivity problems will occur.
- Implementing ACID transactions is **very** difficult and slow.
- You are left engineering a **trade-off between availability and consistency**.

This gives rise to the notion of **eventual consistency**: if update activity ceases, then the system will eventually reach a consistent state.

ACID vs BASE

Many NoSQL systems weaken ACID properties. The result is often called BASE transactions (pun intended).

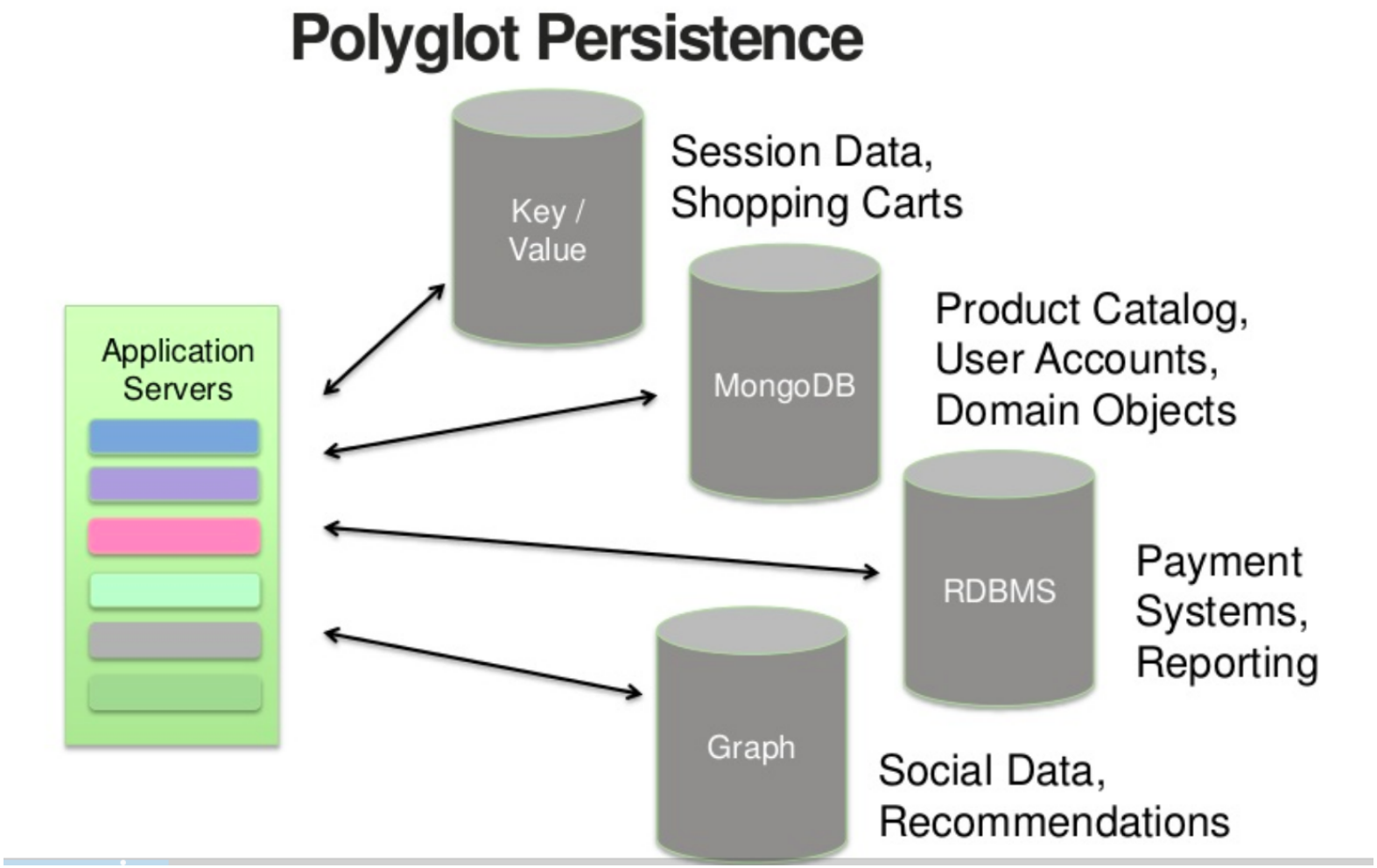
BA: Basically Available,

S: Soft state,

E: Eventual consistency.

Exactly what this means may vary from system to system. This is an area of ongoing research.

Moving away from monolithic databases?



https://www.slideshare.net/mongodb/webinar-mongodb-and-polyglot-persistence-architecture

Watch this space ...

- There is a lot of churn in this area.
- Many traditional SQL-based systems are being extended with NoSQL features.
- Many NoSQL systems are being extended with traditional SQL features.
- Think of the current 250+ NoSQL systems as representing attempts to explore a vast space of trade-offs...

Trade-offs often change as technology changes

Expect more dramatic changes in the coming decades ...

5 megabytes of RAM in 1956

A modern server

This course looks at 3 data models

3 models

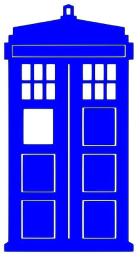
Relational Model: Data is stored in tables. SQL is the main query language. Optimised for high throughput of many concurrent updates.

Aggregate-oriented Model: Also called document-oriented database. Optimised for read-oriented databases with few updates.

Graph-oriented Model: Data is stored as a graph (nodes and edges). Query languages tend to have "path-oriented" capabilities.

The relational model has been the industry mainstay for the last 45 years. The other two models are representatives of an ongoing revolution in database systems often described under the "NoSQL" banner.

This course uses 3 database systems



HyperSQL A Java-based relational DBMS. Query language is SQL.

DOCtor Who A bespoke **doc**ument-oriented collection of data. Not really a DBMS, just some stored python dictionaries containing JSON data! Let's pretend it is a DBMS!

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

IMDb : Our data source

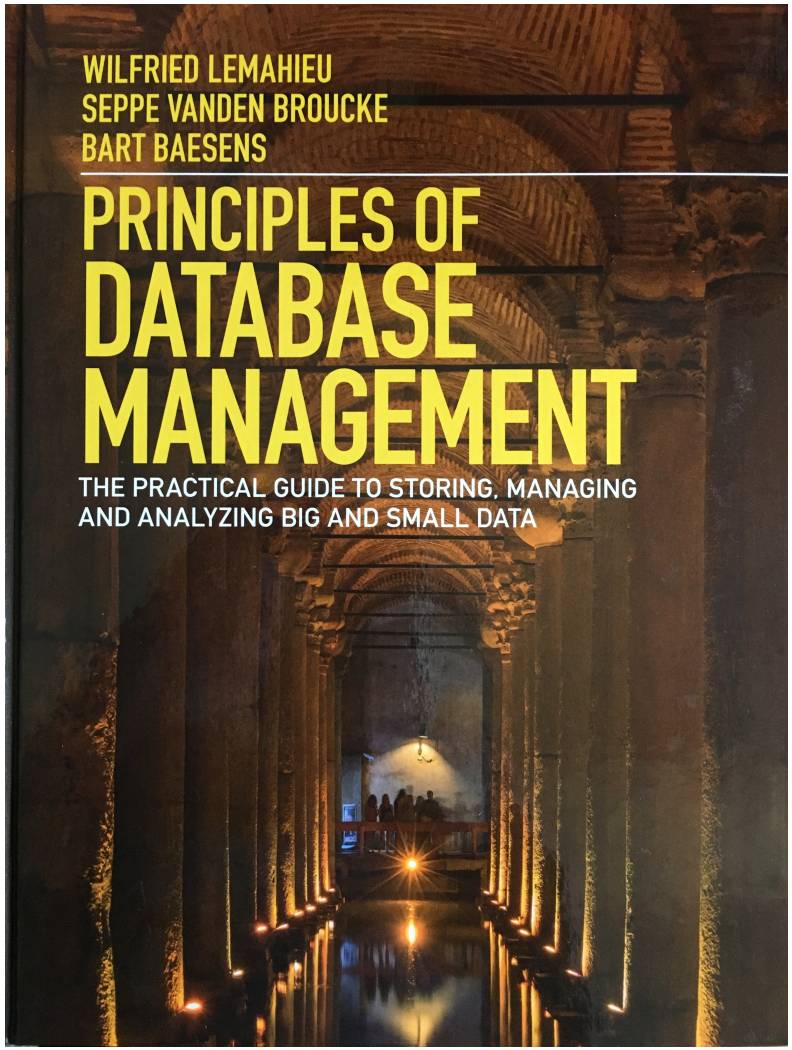
- Raw data available from IMDb plain text data files at `http://www.imdb.com/interfaces` (snapshot of 16/09/2020).
- Extracted from this: 1324 movies made between 2000 and 2021 together with 6745 associated people (actors, directors, etc).
- The same data set was used to generate three database instances: relational, graph, and document-oriented.

Outline

	date	topics
1	08/10	What is a Database Management System (DBMS)?
2	13/10	Entity-Relationship (ER) diagrams
3	15/10	Relational Databases ...
4	20/10	... and SQL
	21/10	**Relational DB practical due (“tick 1”)**
5	22/10	Some limitations of SQL ...
6	27/10	Document-oriented Database
	28/10	**Document DB practical due (“tick 2”)**
7	29/11	Graph Database
8	03/11	Graph Database continued
	04/11	**Graph DB practical due (“tick 3”)**

Get started **NOW** on the practicals!

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

Lecture 2 : Conceptual modeling 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 capture things of interest

- **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

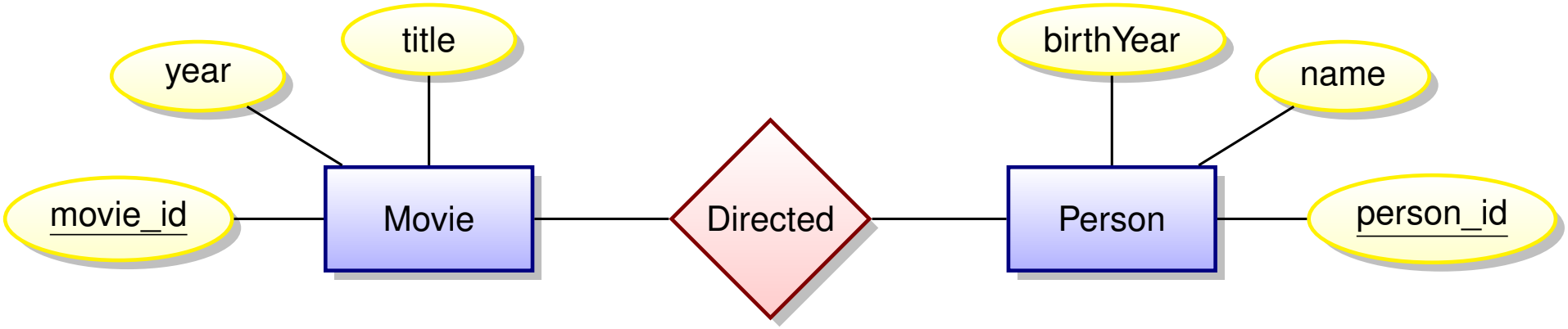
movie_id	title	year
tt1454468	Gravity	2013
tt0440963	The Bourne Ultimatum	2007

Instances of the Person entity

person_id	name	birthYear
nm2225369	Jennifer Lawrence	1990
nm0000354	Matt Damon	1970

Keys are often automatically generated to be unique. Or 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. The only safe thing to use as a key is something that is automatically generated in the database and only has meaning within that database.

Relationships



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

Relationship instances

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

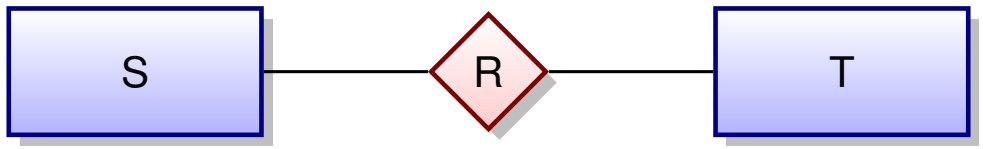
- Kathryn Bigelow directed The Hurt Locker
- Kathryn Bigelow directed Zero Dark Thirty
- Paul Greengrass directed The Bourne Ultimatum
- Steve McQueen directed 12 Years a Slave
- Karen Harley directed Waste Land
- Lucy Walker directed Waste Land
- João Jardim directed Waste Land

Relationship "Cardinality"

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

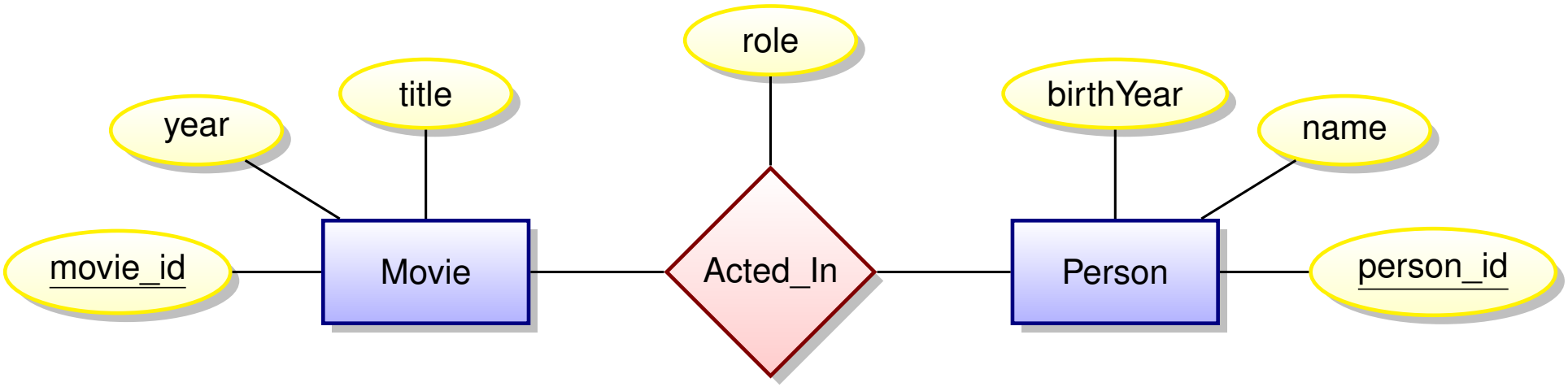
- Every person can direct multiple movies and every movie can have multiple directors

A many-to-many relationship



- Any *S* can be related to zero or more *T*'s
- Any *T* can be related to zero or more *S*'s

Relationships can have attributes



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

Instances of the relationship **Acted_In**

(ignoring entity attributes)

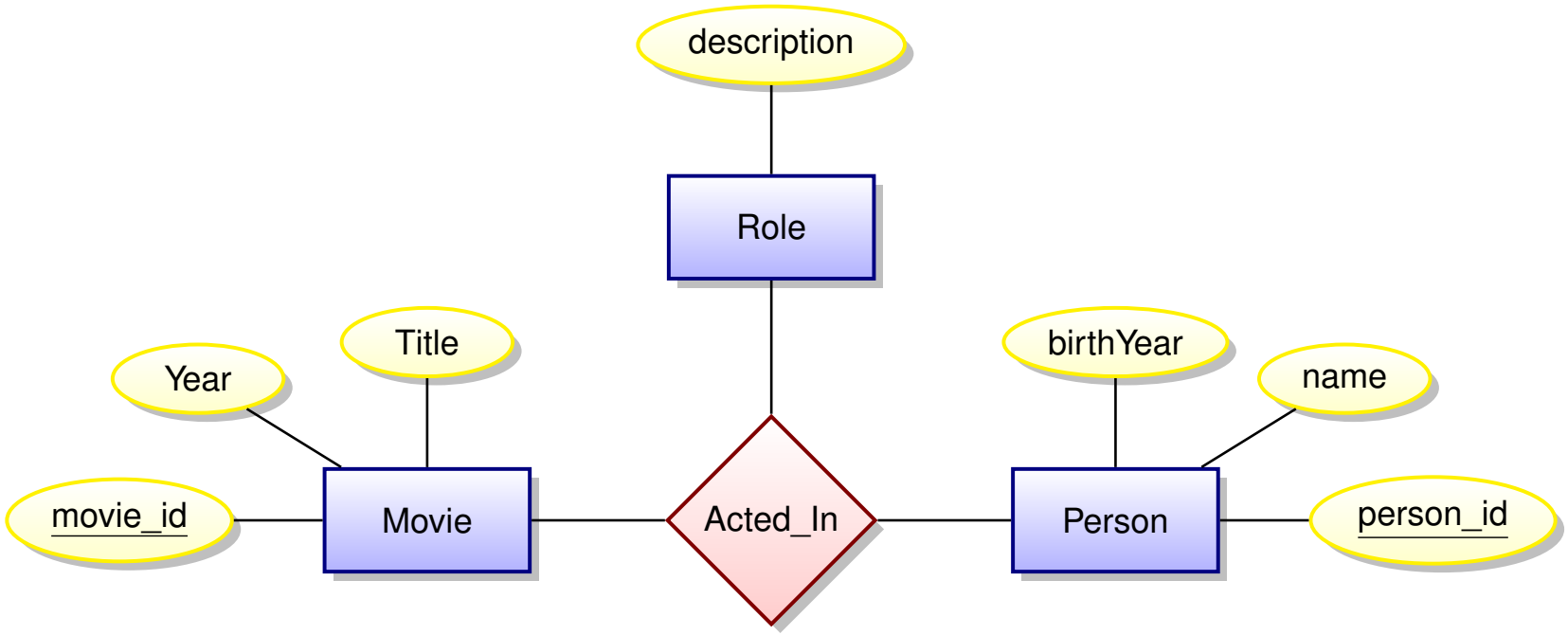
- Ben Affleck played Tony Mendez in Argo
- Julie Deply played Celine in Before Midnight
- Bradley Cooper played Pat in Silver Linings Playbook
- Jennifer Lawrence played Tiffany in Silver Linings Playbook
- Tim Allan played Buzz Lightyear in Toy Story 3

Have we made a mistake?

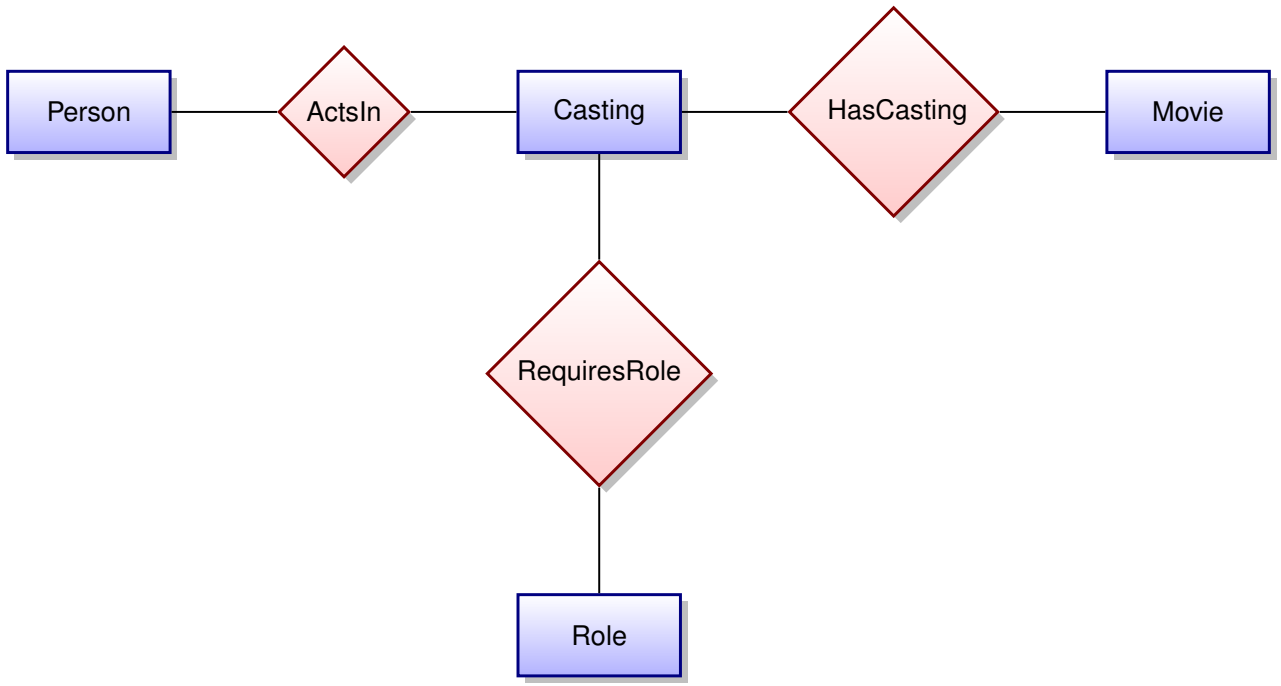
Note that this model assumes that an actor plays a single role in every movie. This may not be the case!

- 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

Acted_In can be modeled as a Ternary Relationship

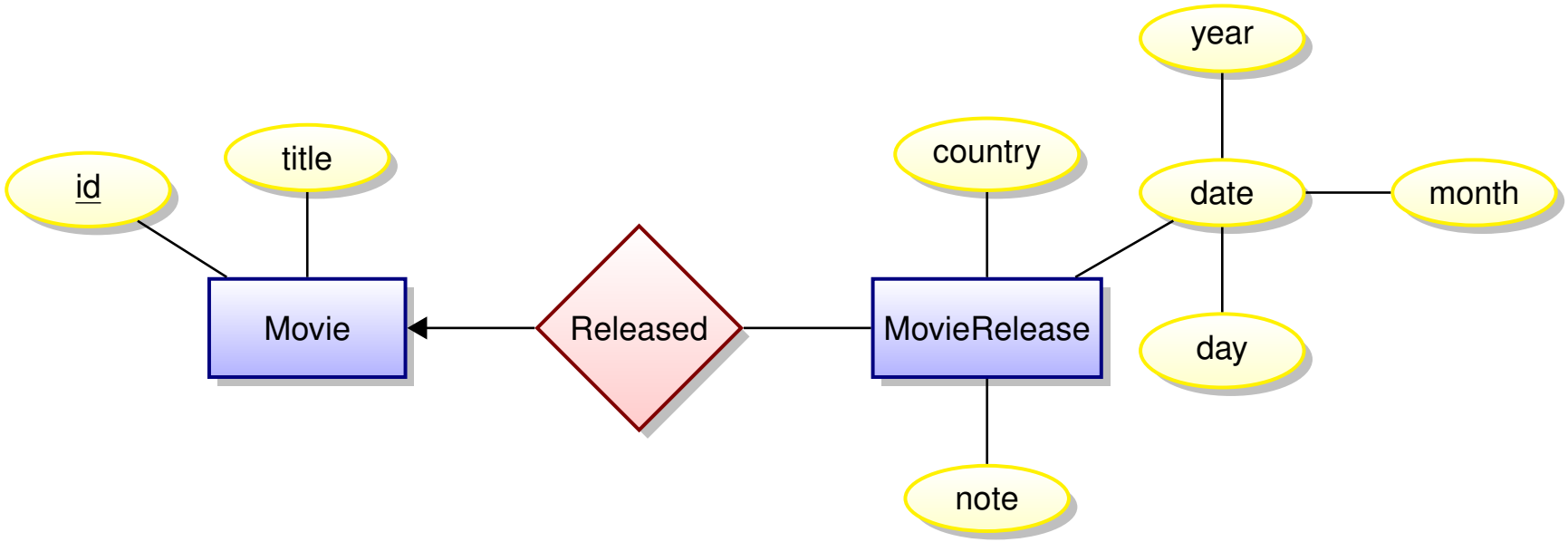


Can a ternary relationship be modeled with multiple binary relationships?



Is the **Casting** entity too artificial?

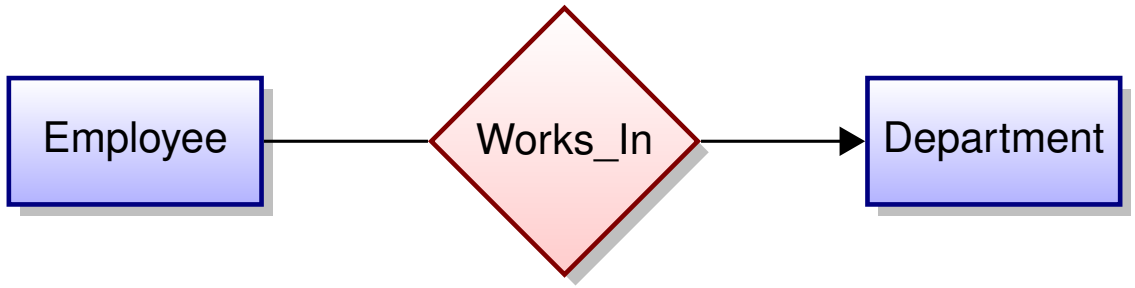
Attribute or entity with new relationship?



- Should the release year be an 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 there something special about the MovieRelease?**

many-to-one relationships

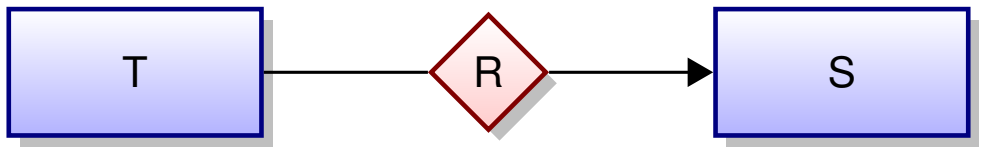
Suppose that every employee is related to at most one department. We will draw with an arrow:



- 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 and many-to-one relationships, abstractly

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

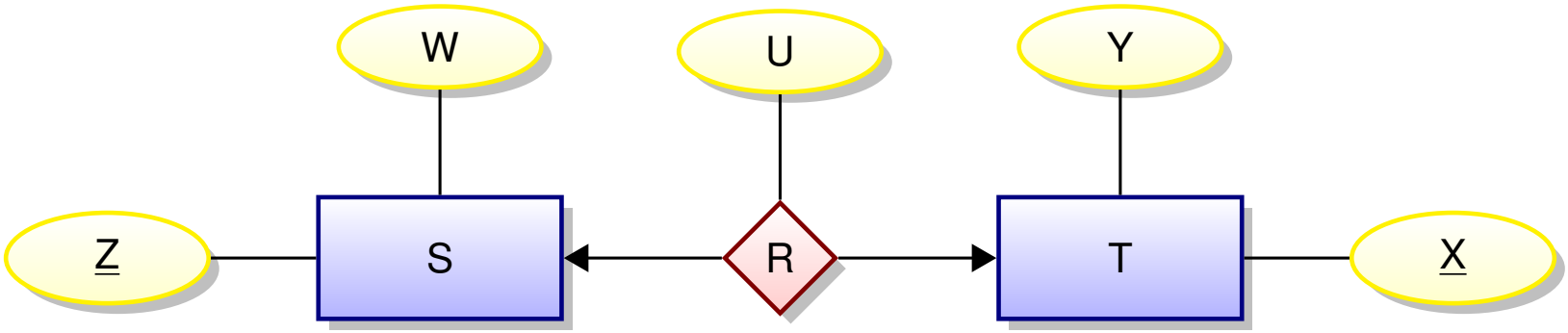


The relation R is **many-to-one** between T and S

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

If R is both **many-to-one** between T and S and **one-to-many** between S and T, then it is **one-to-one** between T and S

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

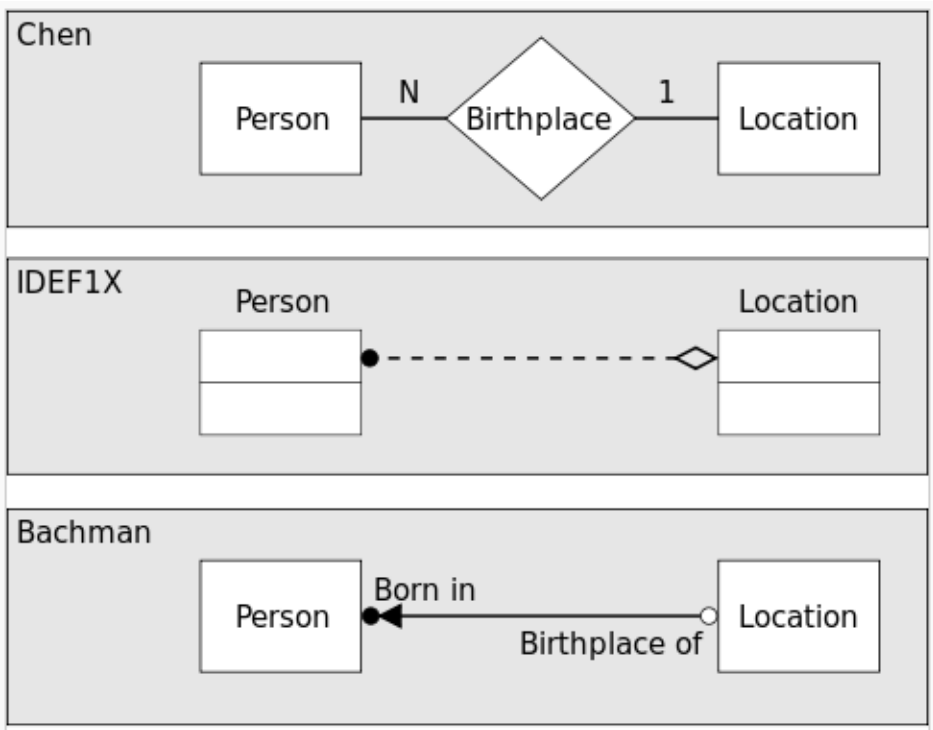
R

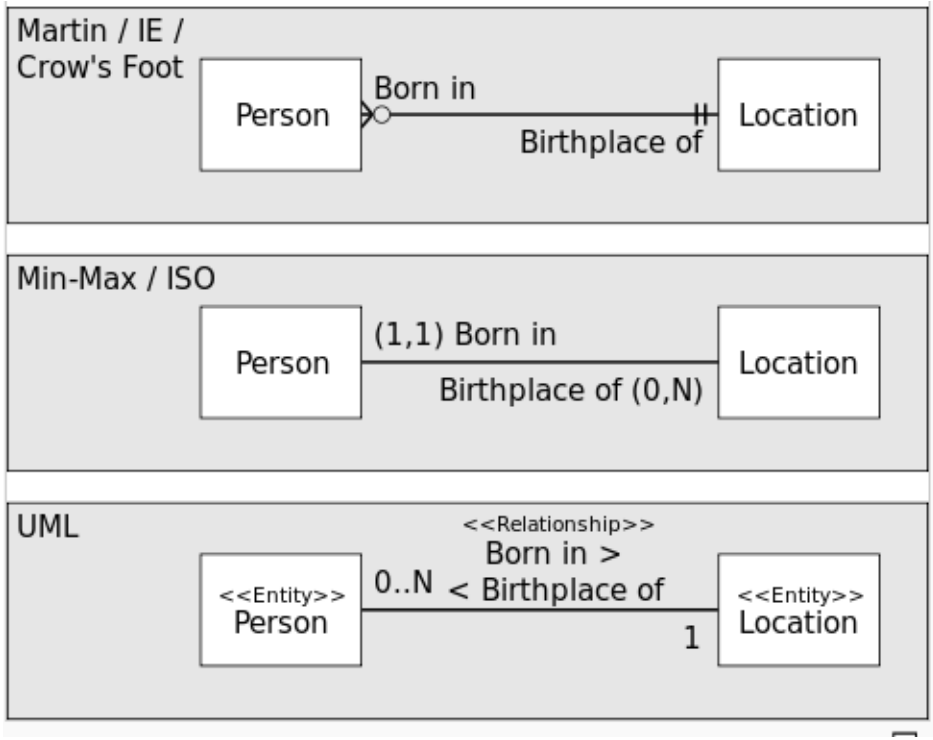
Z	**X**	U
z_1	x_2	u_1

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 ...

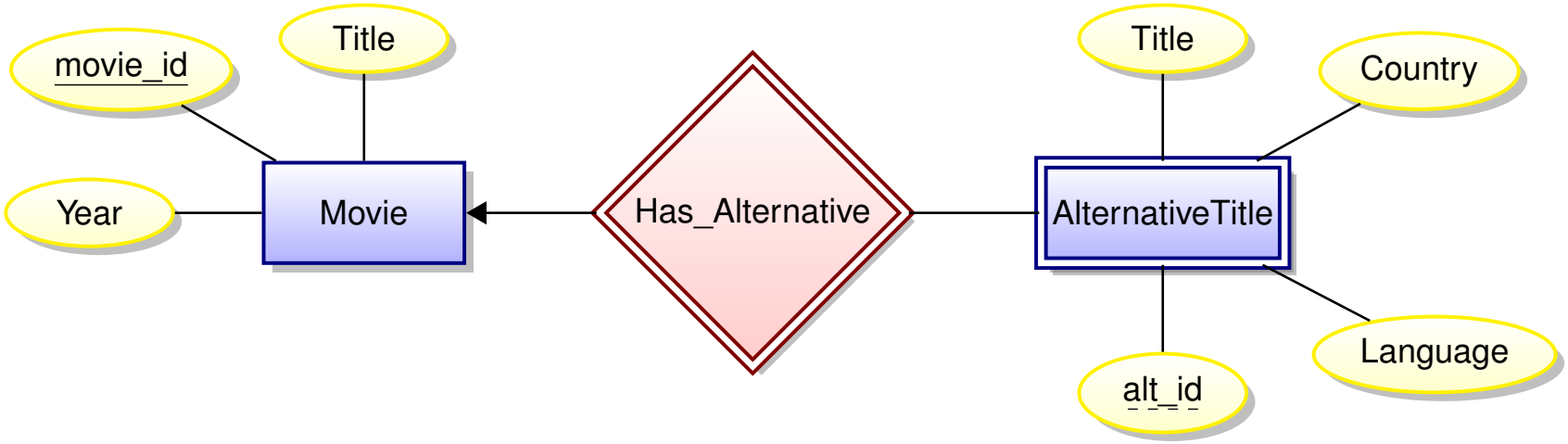




Various diagrammatic notations used to indicate a one-to-many relationship (`https://en.wikipedia.org/wiki/Entity-relationship_model`).

Note: We will not bother with these notations, but the concept of a relationship's cardinality is an important one.

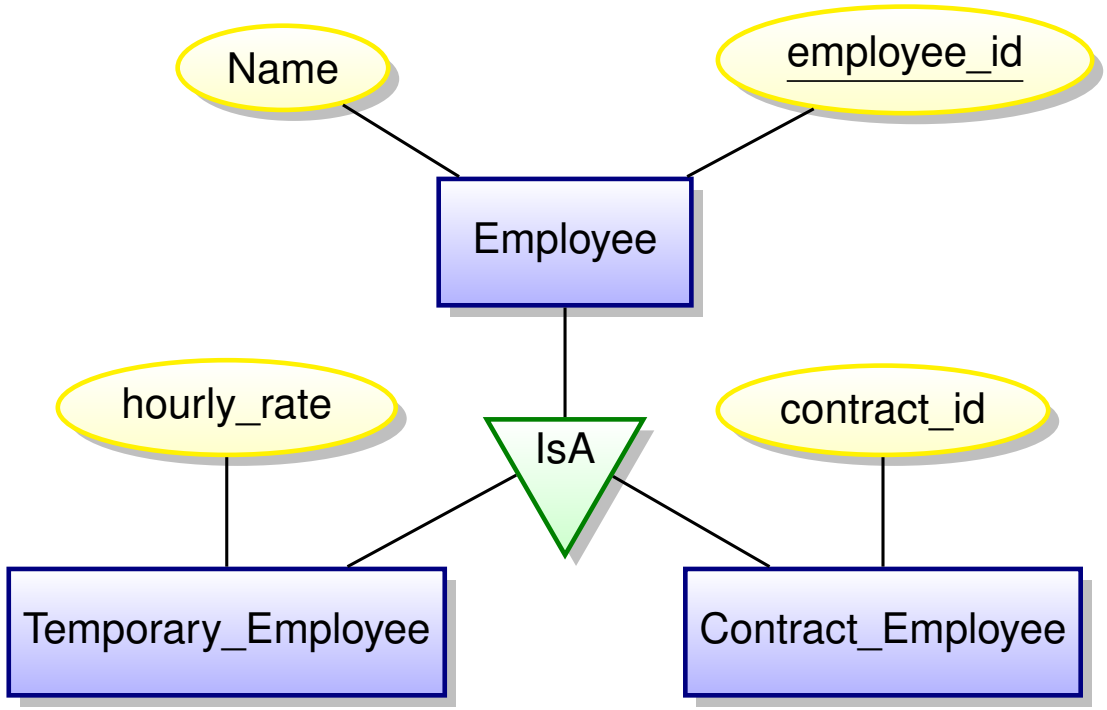
Weak entities



- 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

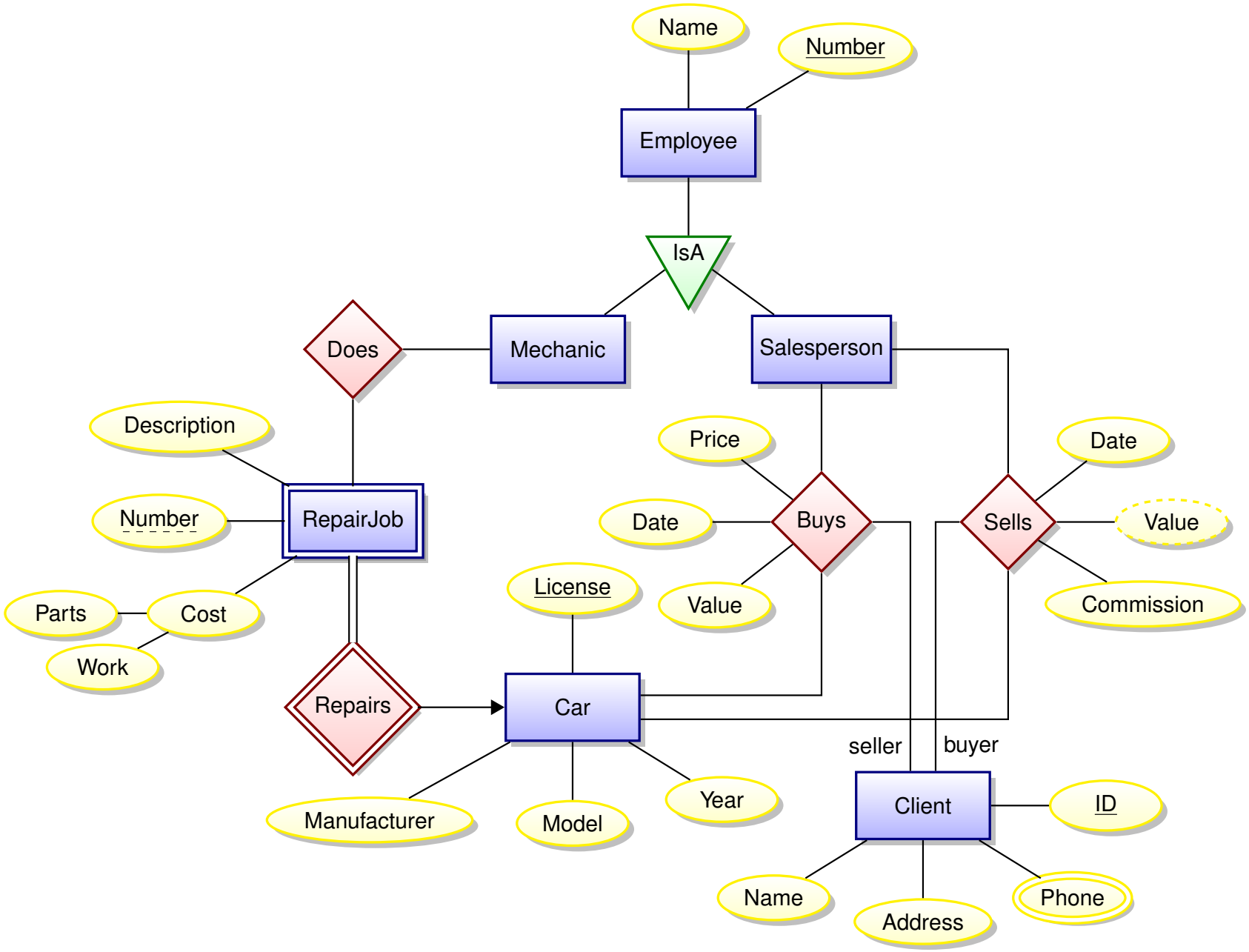
Sometimes an entity can have “sub-entities”. Here is an example:



Sub-entities inherit the attributes (including keys) and relationships of the parent entity.

Entity-Relationship Diagrams

- 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 modeler 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?

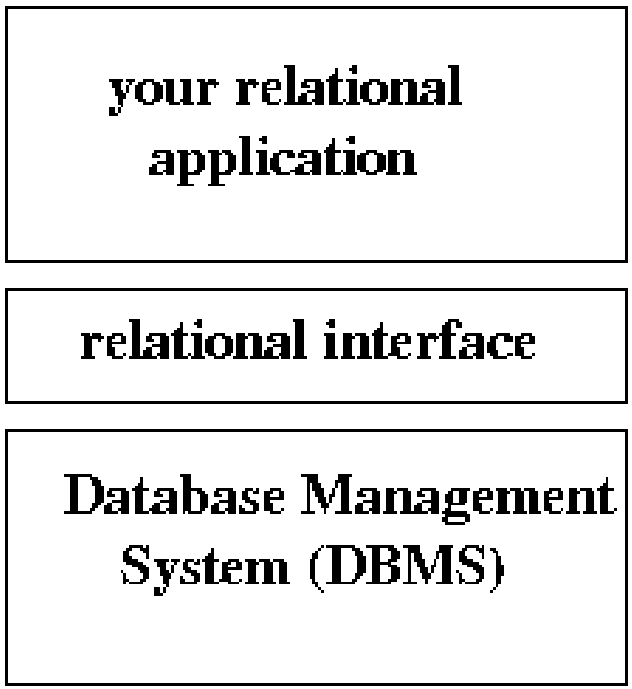


Example due to Pável Calado, author of the tikz-er2.sty package.

Lecture 3

- The relational Model
- SQL and the Relational Algebra (RA)
- Update anomalies
- Avoid redundancy!

Still 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

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

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 can be stored!

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 $R(A_1 : S_1,\ A_2 : S_2,\ \cdots A_n : S_n)$.

Example

A relational schema

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

A relational instance of this schema

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

Two equivalent tabular presentations

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 ...

The Relational Algebra (RA)

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

- 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

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`

Projection

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`

Renaming

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`

Union

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)`

Intersection

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

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

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`

Note that the RA product is not exactly the Cartesian product suggested by this notation!

Natural Join

First, a bit 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 $\mathbf{A} \cap \mathbf{B} = \phi$.
- $u.[\mathbf{A}] = v.[\mathbf{A}]$ abbreviates $u.A_1 = v.A_1 \wedge \cdots \wedge u.A_n = v.A_n$.

Natural Join

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:

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

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'

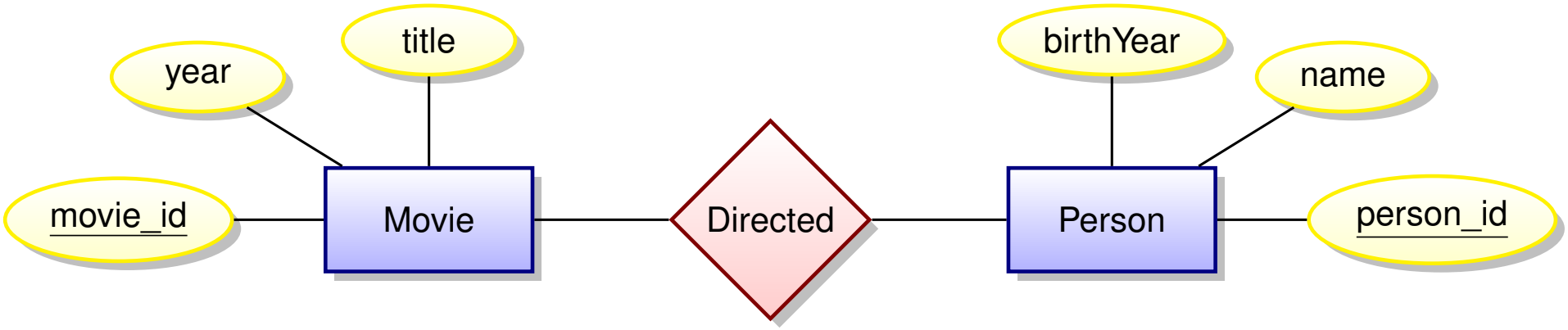
$\Longrightarrow$

Students $\bowtie$ Colleges

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

(See online tutorial for joins in SQL.)

Lecture 4: How can we implement an ER model relationally?



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

Remember, we only have tables to work with!

How about one big table?

DirectedComplete

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

What's wrong with this approach?

Anomalies caused by data redundancy

Insertion anomalies: How can we tell if a newly inserted record is consistent all other records 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 anomalies: We will wipe out information about people when last record is deleted from this table.

Update anomalies: What if an director's name is mis-spelled? We may update it correctly for one movie but not for another.

- A transaction implementing a conceptually simple update but containing checks to guarantee correctness may end up locking the entire table.
- Lesson: In a database supporting many concurrent updates we see that data redundancy can lead to complex transactions and low write throughput.

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

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

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
.....	.....	....

What about 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
.....	.....

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 actually exist in our database (more on that later). We would 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 ActsIn relation

The SQL query

```
select movies.id as mid, title, year,
       people.id as pid, name, character, position
from movies
join actsin on movie_id = movies.id
join people on people.id = person_id
```

might return something like

MID	TITLE	YEAR	PID	NAME	CHARACTER	POSITION
2544956	12 Years a Slave (2013)	2013	146271	Batt, Bryan	Judge Turner	4
2544956	12 Years a Slave (2013)	2013	2460265	Bennett, Liza J.	Mistress Ford	32
2544956	12 Years a Slave (2013)	2013	173652	Bentley, Tony (I)	Mr. Moon	9
2544956	12 Years a Slave (2013)	2013	477824	Dano, Paul	Tibeats	35
2544956	12 Years a Slave (2013)	2013	256114	Bright, Gregory	Edward	42
2544956	12 Years a Slave (2013)	2013	2826281	Haley, Emily D.	Tea Seller	NULL
...	...	...	...	...	...	...

Observations

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

What about consistency/integrity of our relational implementation?

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

Key Concepts

Relational Key

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 $R(\mathbf{Z} \cup \mathbf{Y})$.

Note that this is a semantic assertion, and that a relation can have multiple keys.

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.

A representation using tables

A relational schema

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

With referential integrity constraints

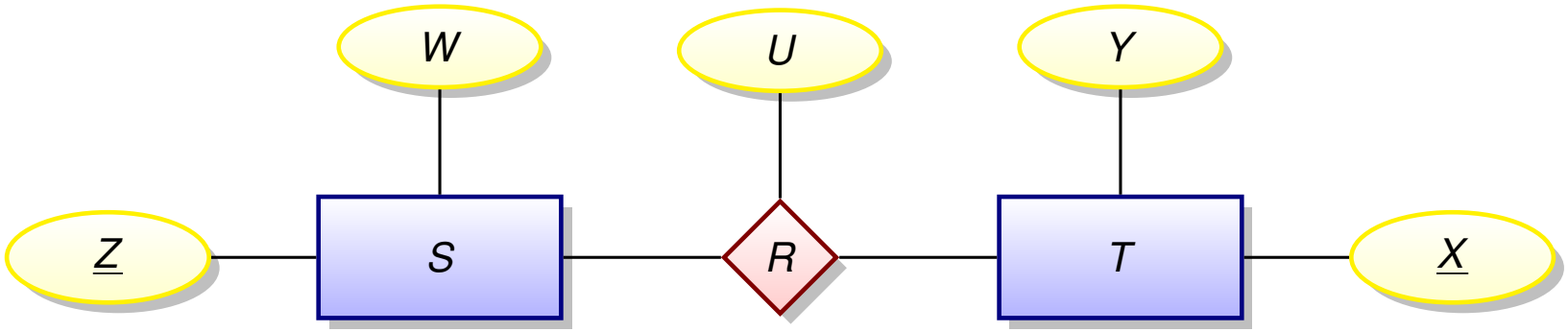
$$\pi_{movie\_id}(Has\_Genre) \subseteq \pi_{movie\_id}(Movies)$$

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

Foreign Keys in SQL

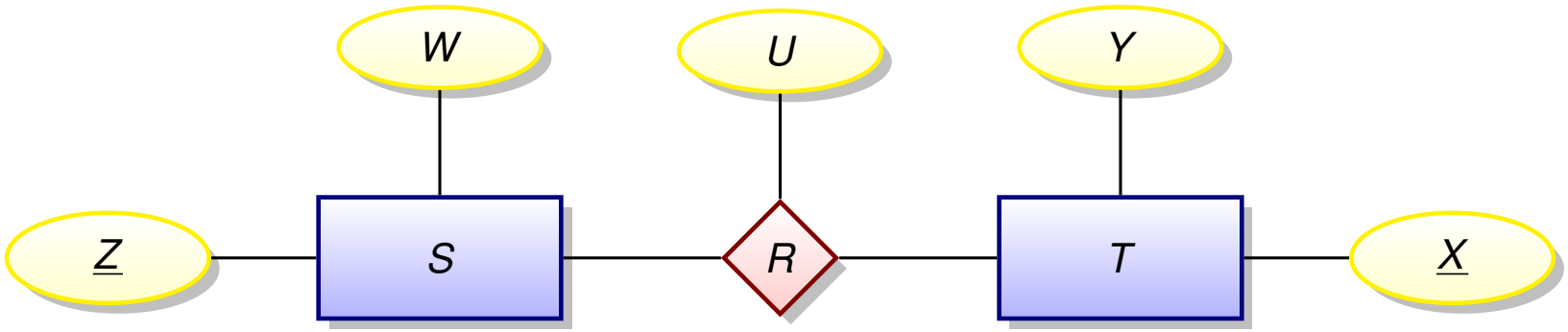
```
create table genres (
    genre_id integer NOT NULL,
    genre varchar(100) NOT NULL,
    PRIMARY KEY (genre_id));

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

Relationships to 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 $(M : 1)$	$R(X, \underline{Z}, U)$

Implementation can differ from the “clean” approach

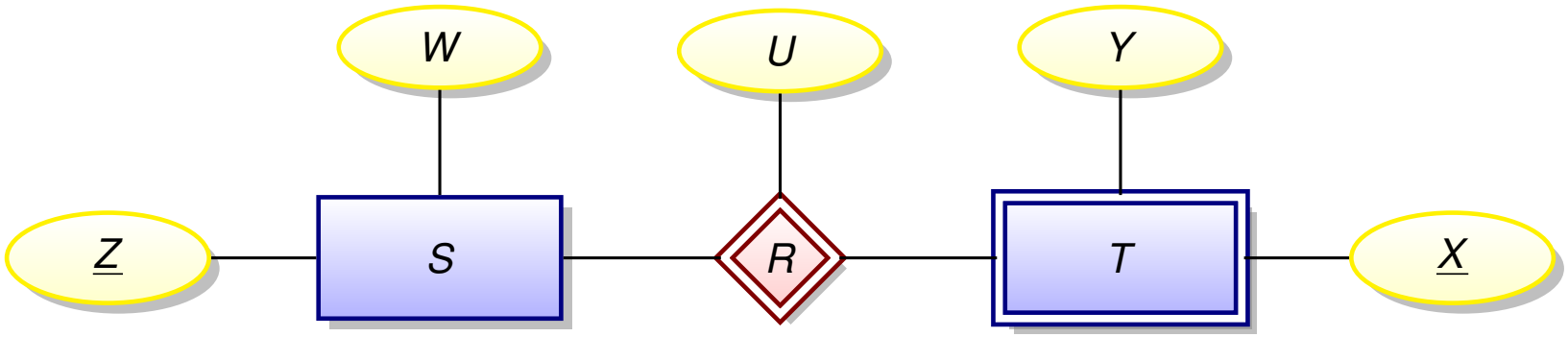


Suppose R is one to many

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 weak entities

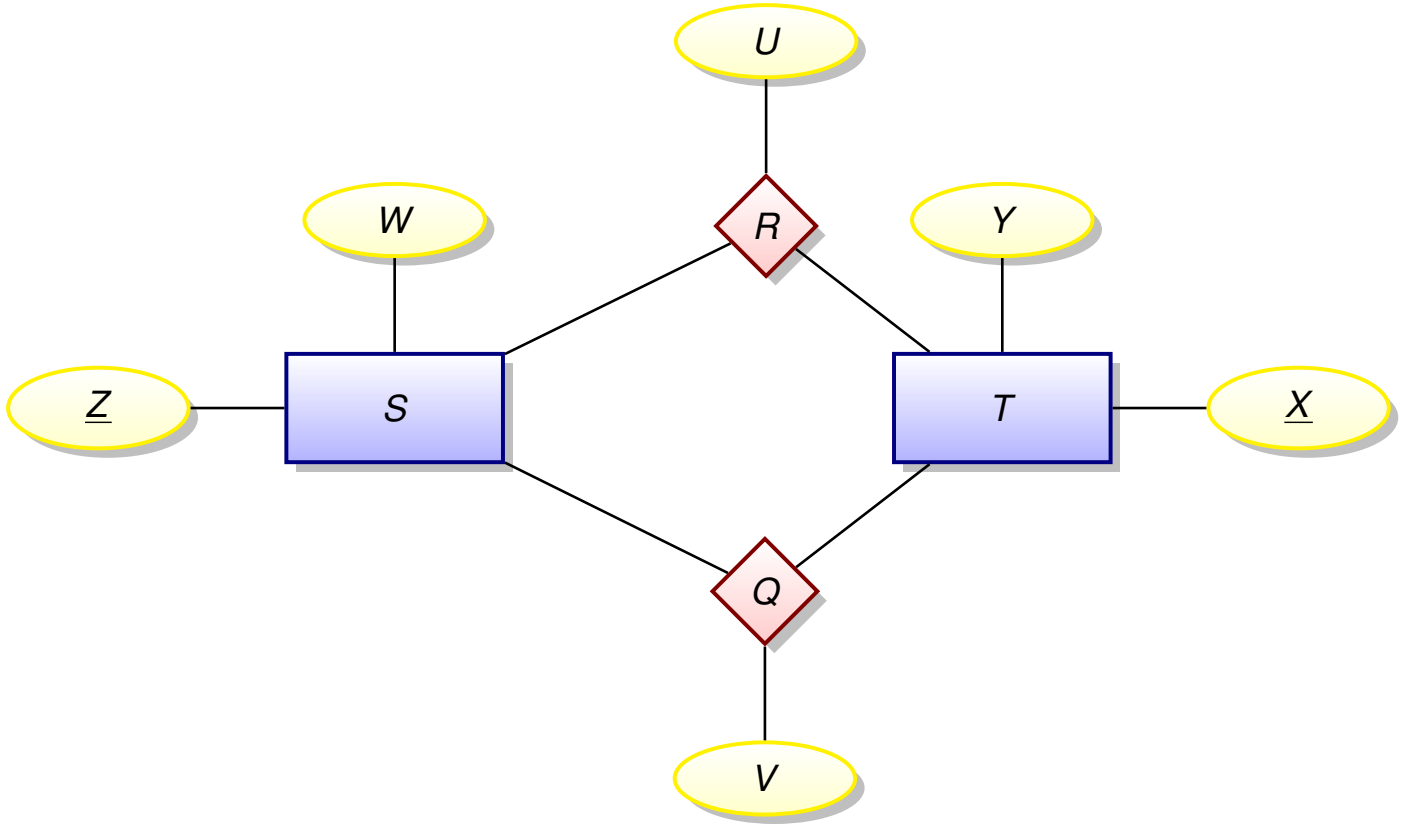


This is always a one to many relationship!

- Notice that **all** rows of T must participate in the relationship.
- The expanded $T(\underline{X},\ Y,\ Z,\ U)$ is even more compelling.
- We might drop the keys $\underline{X}$ from T resulting in $T(\underline{Y},\ \underline{Z},\ \underline{U})$.

Implementing multiple relationships into a single table?

Suppose we have two many-to-many relationships:



Implementing multiple relationships into a single table?

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 now the value of the *type* column, we can compute the value of either the U column or the V column!

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

```
select position, count(*) as total
from has_position
group by position
order by total desc;
```

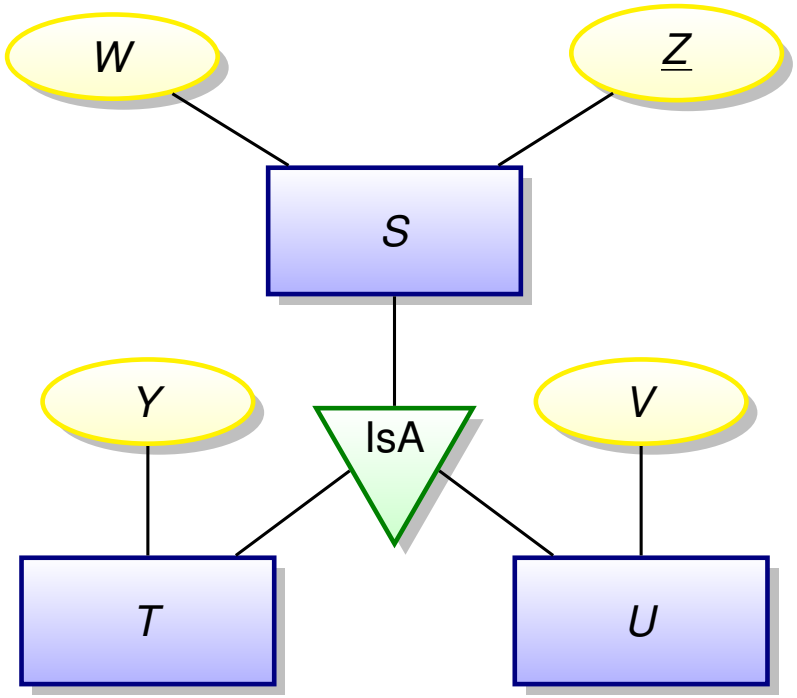
Using our database this query produces the output

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

Was this a good idea?

Discuss!

Implementation of entity hierarchy

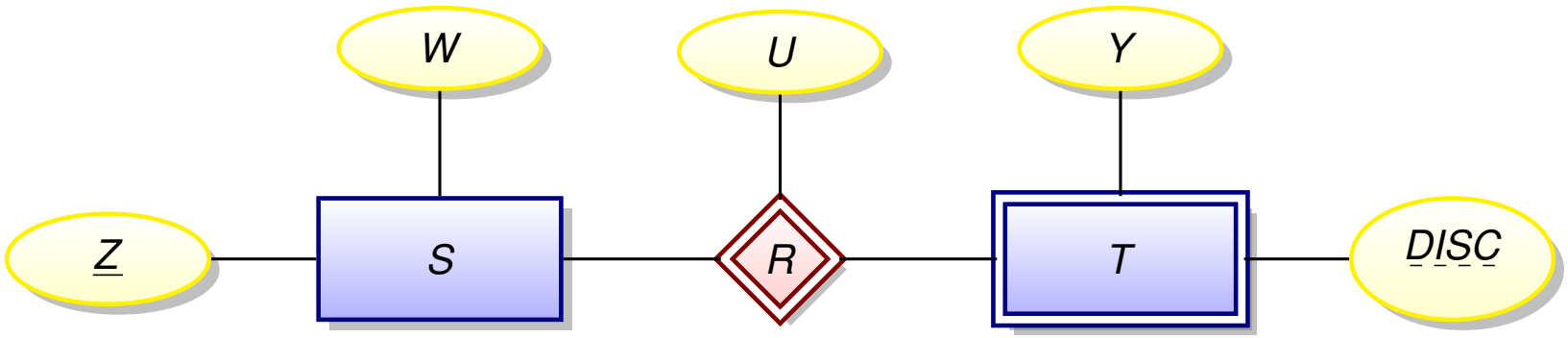


One approach:

- $S(\underline{Z},\ W)$
- $T(\underline{Z},\ Y)$ 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?

Implementing weak entities



One approach:

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

Another 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.

Lecture 5

- Data redundancy and update anomalies.
- Relational normalisation attempts to eliminate redundancy.
- Normalisation and transaction throughput.
  - Relation databases are 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.

Implementing ACID transactions requies locking data

- As will be discussed in IB Concurrent and Distributed Systems, implementing ACID transaction requires **locks** on data.
- Locks are acquired and released by transactions.
- Once a lock is acquired a transaction has exclusive access to the locked data.
- Locking mechanisms can be placed along a spectrum of granulariy from very coarse-grained (lock the entire database!) to very fine-grained (lock a single data value).
- Implementation details of implementing ACID transactions are not a part of SQL standards. Rather, this is part of the “secret sauce” implemented by every vendor.
- **Observation:** As more and more data is locked by typical transactions, fewer and fewer concurrent updates can be supported within a given time period.

What is redundant data? Why 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?

In a database supporting high-throughput update transactions, high levels of data redundancy imply that **correct** transactions may have to acquire many locks to consistently update copies of redundant data.

Update anomalies

This is a general term for transactions that do no correctly update redundant data.

Database normalisation

- Since reducing redundancy results in higher throughput many techniques have been developed for eleminating redundancy from schema designs.
- This is called database normalisation.
- A normalised database is one that has little or no redundant data.
- We will not cover the theoretical details of normal forms (3rd normal form, Boyce-Codd normal form, fourth normal form, fifth normal form, and so on).
- **Why? In practice, if you have done a good job of Entity-Relationship modeling, then your database should be fairly well normalised.**

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

- Your data is seldom updated, but very often read.
- 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.

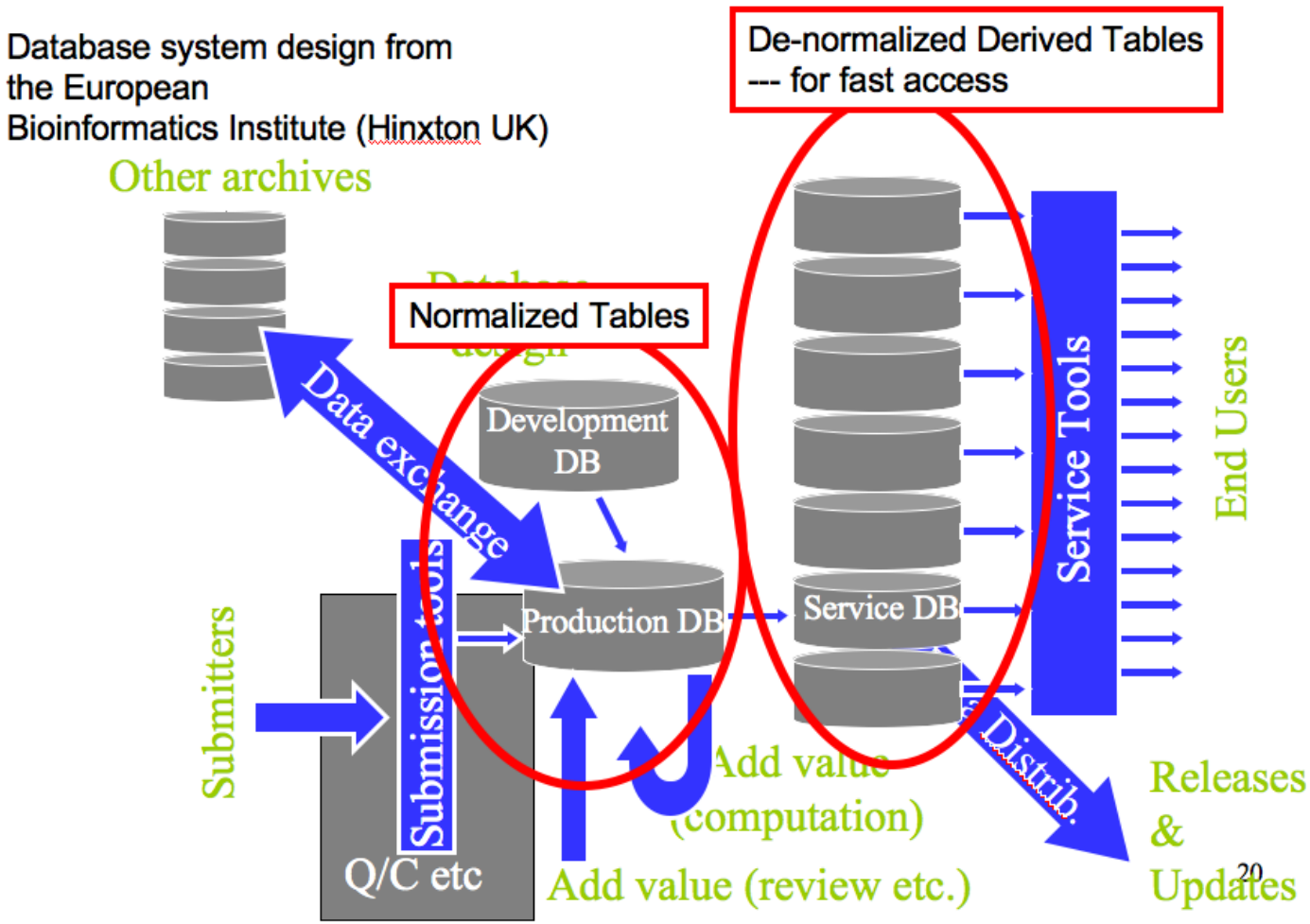
Data redundancy is problematic for some applications

If a database supports many concurrent updates, then data redundancy leads to many problems, discussed in Lecture 4. If a database has little redundancy, then update throughput is typically better since transactions need only lock a few data items. This has been the traditional approach in the relational database world.

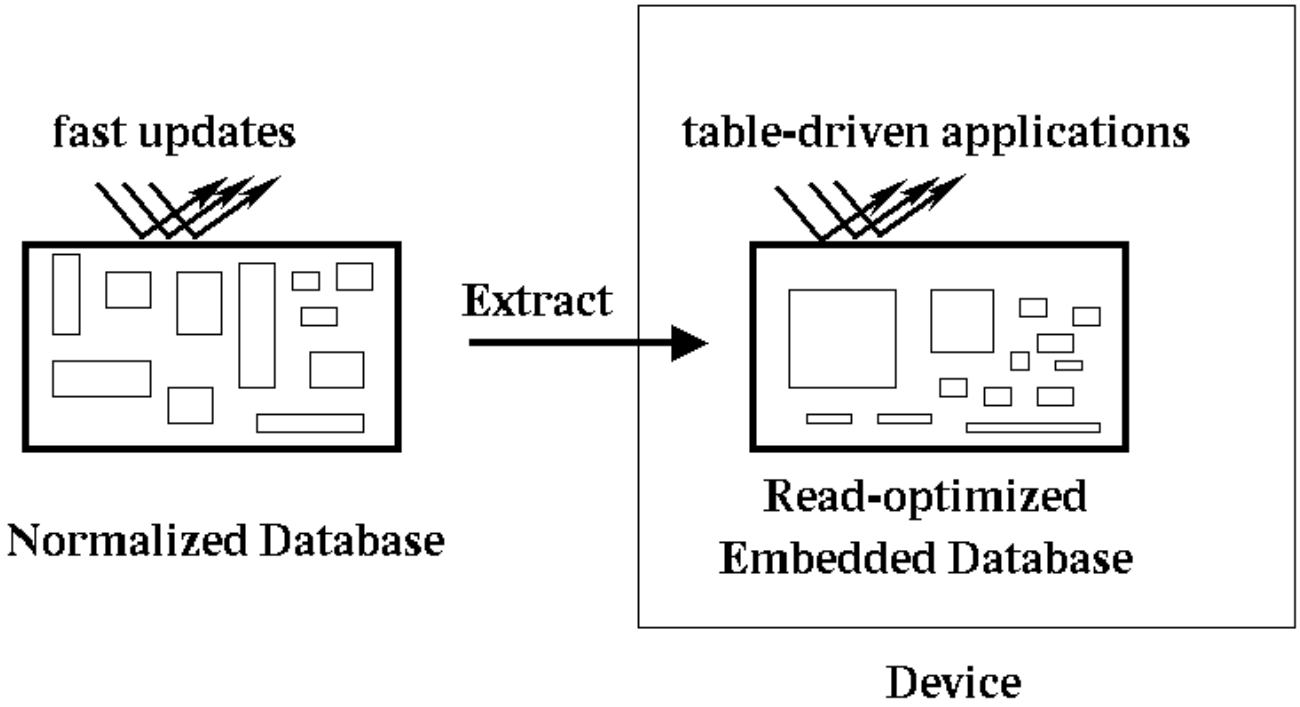
Data redundancy is highly desirable for some applications

In a low redundancy database, evaluation of complex queries can be very slow and require large amounts of computing power. Precomputing answers to common queries (either fully or partially) can greatly speed up query response time. This introduces redundancy, but it may be appropriate for databases supporting applications that are read-intensive, with few or no data modifications. This is an approach common in aggregate-oriented databases.

Example : Hinxton Bio-informatics



Example : Embedded databases



FIDO = Fetch Intensive Data Organization

Key-value stores

- One of the simplest types of database systems is the **key-value store** that simply maps a key to a block of bytes.
- The retrieved block of bytes is typically opaque to the databases system.
- Interpretation of such data is left to applications.

OLAP vs. OLTP

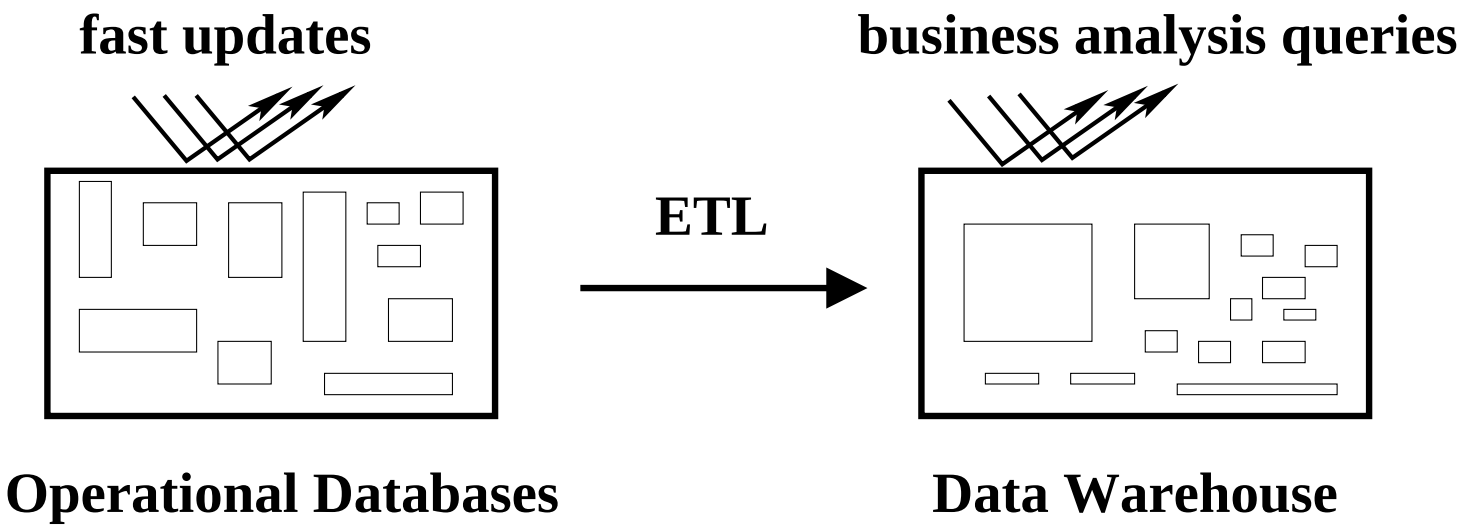
OLTP Online Transaction Processing

OLAP Online Analytical Processing

- Commonly associated with terms like Decision Support, Data Warehousing, etc.

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

Example : Data Warehouse (Decision support)



ETL = Extract, Transform, and Load

- This looks very similar to slide 91!
- But there must be differences that are not illustrated.
- What are these differences?

Lecture 6

- Optimise for reading data?
- Document-oriented databases
- Semi-structured data
- Our bespoke database: DoctorWho
- Using python as a query language

Semi-structured data : JSON

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

From `http://json.org/example.html`.

Semi-structured data : XML

```
<menu id="file" value="File">
  <popup>
    <menuitem value="New" onclick="CreateNewDoc()" />
    <menuitem value="Open" onclick="OpenDoc()" />
    <menuitem value="Close" onclick="CloseDoc()" />
  </popup>
</menu>
```

From `http://json.org/example.html`.

Document-oriented database systems

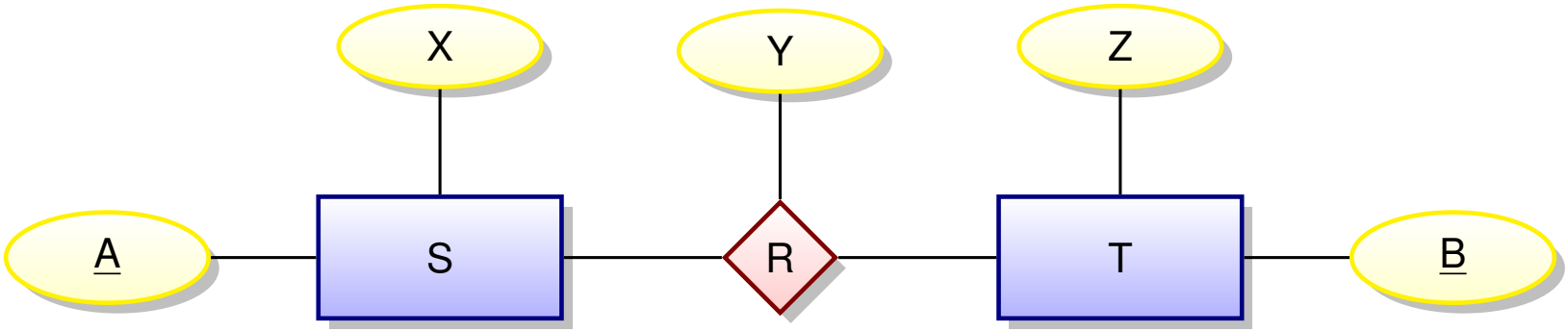
Our working definition

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

Why Semi-structured data?

- Let's do **a thought experiment**.
- In the next few slides imagine that we intend to use a relational database to store read-optimised tables generated from a a set of write-optimised tables (that is, having little redundancy).
- We will encounter some problems that can be solved by representing our data as semi-structured objects.

Start with a simple relationship ...



A database instance

S

A	X
a1	x1
a2	x2
a3	x3

R

A	B	Y
a1	b1	y1
a1	b2	y2
a1	b3	y3
a2	b1	y4
a2	b3	y5

T

B	Z
b1	z1
b2	z2
b3	z3
b4	z4

Imagine that our read-oriented applications can't afford to do joins!

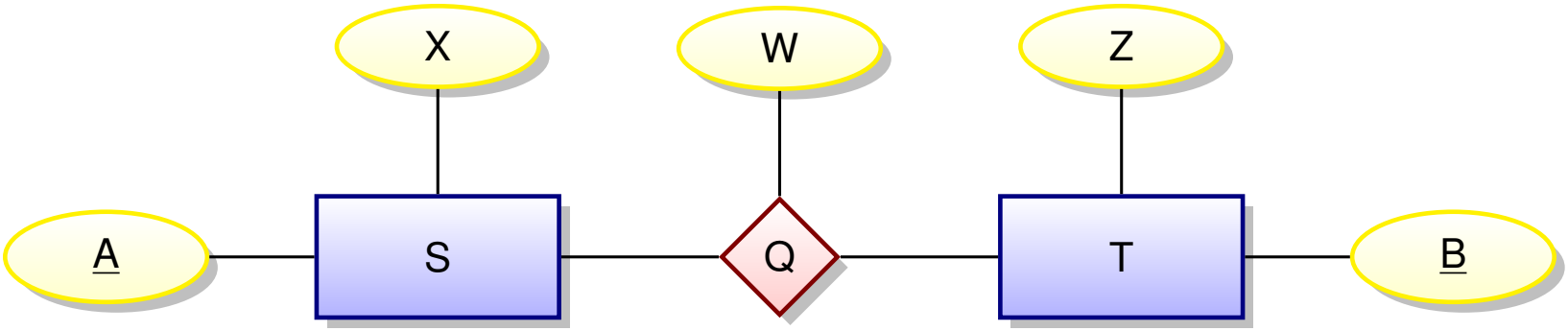
Implement the relationship as one big table?

BigTableOne: An outer join of *S*, *R*, and *T*

A	X	B	Z	Y
a1	x1	b1	z1	y1
a1	x1	b2	z2	y2
a1	x1	b3	z3	y3
a2	x2	b1	z1	y4
a2	x2	b3	z3	y5
a3	x3			
		b4	z4	

Since we don't update this date we will not encounter the problems associated with redundancy.

However, we might have many more relationships ...



A database instance

S

A	X
a1	x1
a2	x2
a3	x3

Q

A	B	W
a1	b4	w1
a3	b2	w2
a3	b3	w3

T

B	Z
b1	z1
b2	z2
b3	z3
b4	z4

Implement with another big table?

BigTableTwo: An outer join of *S*, *Q*, and *T*

A	X	B	Z	W
a1	x1	b4	z4	w1
a3	x3	b2	z2	w2
a3	x3	b3	z3	w3
a2	x2			
		b1	z1	

Having two tables makes reading a bit more difficult!

Combine into one big table?

BigTable: Derived from *S*, *R*, *Q*, and *T*

A	X	B	Z	Y	W
a1	x1	b1	z1	y1	
a1	x1	b2	z2	y2	
a1	x1	b3	z3	y3	
a2	x2	b1	z1	y4	
a2	x2	b3	z3	y5	
a1	x1	b4	z4		w1
a3	x3	b2	z2		w2
a3	x3	b3	z3		w3

Problems with BigTable

- We could store BigTable and speed up some queries.
- But suppose that our applications typically access data using either S's key or T's key.
- Creating indices on the A and B columns could speed things up, but our applications may still be forced to gather information from many rows in order to collect all information related to a given key of S or a given key of T.
- It would be better to access all data associated with a given key of S or a given key of T using only **a single database lookup**.

Potential Solution

Represent the data using semi-structured objects.

Use (*S*-oriented) documents ("A" value is unique id)

```
{ "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}]
}
```

Use (*T*-oriented) documents ("B" value is id)

```
{ "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}] }
```

DOCtor Who

Our **DOCtor Who** “database” is simply made up of two key-value stores – one for movies and one for people. Here database is in quotes since we will just be using python dictionaries to implement mappings from keys to JSON objects. In other words, no ACID transactions, no auxiliary indices, no JSON-specific (XPath-like) query language, ...

Could be implemented on top of transactional key-value store.
(If anyone finds a JSON-based database that is easy to install, please let me know....)

DOCtor Who: person_id nm2225369 maps to


```
{ 'person_id': 'nm2225369',
  'name': 'Jennifer Lawrence',
  'birthYear': '1990',
  'acted_in': [
    {'movie_id': 'tt1355644', 'roles': ['Aurora Lane'],
     'title': 'Passengers', 'year': '2016'},
    {'movie_id': 'tt1045658', 'roles': ['Tiffany'],
     'title': 'Silver Linings Playbook', 'year': '2012'},
    {'movie_id': 'tt1392170', 'roles': ['Katniss Everdeen'],
     'title': 'The Hunger Games', 'year': '2012'},
    {'movie_id': 'tt1800241', 'roles': ['Rosalyn Rosenfeld'],
     'title': 'American Hustle', 'year': '2013'},
    {'movie_id': 'tt1951264', 'roles': ['Katniss Everdeen'],
     'title': 'The Hunger Games: Catching Fire', 'year': '2013'},
    {'movie_id': 'tt1270798', 'roles': ['Raven', 'Mystique'],
     'title': 'X-Men: First Class', 'year': '2011'},
    {'movie_id': 'tt1399683', 'roles': ['Ree'],
     'title': "Winter's Bone",'year': '2010'}
  ]
}
```

DOCtor Who: 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'}]
}
```

DOCtor Who: 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 **DOCtor Who**? We write python code!

```
import sys      # operating system interface
import os.path # manipulate paths to files, directories
import pickle  # read/write pickled python dictionaries
import pprint  # pretty print JSON

# first command-line argument = directory of pickled data files
data_dir = sys.argv[1]

# use os.path.join so that path works on both Windows and Unix
movies_path = os.path.join(data_dir, 'movies.pickled')
people_path = os.path.join(data_dir, 'people.pickled')

# open the files and un-pickle them
moviesFile = open(movies_path, mode= "rb")
movies = pickle.load(moviesFile)
peopleFile = open(people_path, mode= "rb")
people = pickle.load(peopleFile)

    ... YOUR QUERY CODE HERE...
```

Thought experiment

Imagine you discover that an actor's name has been misspelled. Now you want to correct it in the database.

Compare the complexity of doing this in our relational database compared to our document database.

Further, imagine that our document database had to support ACID transaction.

Lecture 7

Another look at SQL

- What is a database index?
- Two complications for SQL semantics
  - Multi-sets (bags)
  - NULL values
- **Kevin Bacon!**
- Transitive closure of a relation
- Problems computing a transitive closure in relational databases

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.

What is a database 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) ...
   }
```

In 1A Algorithms you will see a few of the data structures used to implement database indices (search trees, hash tables, and so on).

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.
- **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(**mark**)
spelling	3
databases	88

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 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**
$\perp$	$\perp$

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.

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." (See http://www.firstsql.com/idefend.htm.)

Flaws? One example of SQL's inconsistency

With our small database, the query

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

returns 4892 records of `NULL`.

The expression `note IS NULL` is either true or false — true when `note` is the `NULL` value, false otherwise.

Flaws? One example of SQL's inconsistency (cont.)

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
```

This seems to mean that `NULL` is equal to `NULL`. But recall that `NULL = NULL` returns `NULL`!

Bacon Number

- 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 Bac
```

Result is "nm0000102".

Mathematical relations, again

Given two binary relations

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

we can define their **composition** $Q \circ R \subseteq S \times U$ as

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

Partial functions as relations

- A (partial) function $f \in S \to T$ can be thought of as a binary relations where $(s,\ t) \in f$ if and only if $t = f(s)$.
- Suppose R is a relation where if $(s,\ t_1) \in R$ and $(s,\ t_2) \in R$, then it follows that $t_1 = t_2$. In this case R represents a (partial) function.
- Given (partial) functions $f \in S \to T$ and $g \in T \to U$ their **composition** $g \circ f \in S \to U$ is defined by $(g \circ f)(s) = g(f(s))$.
- Note that the definition of $\circ$ for relations and functions is equivalent for relations representing functions.

Since we could write $Q \circ R$ as $R \bowtie_{2=1} Q$ we can see that **joins are a generalisation of function composition**!

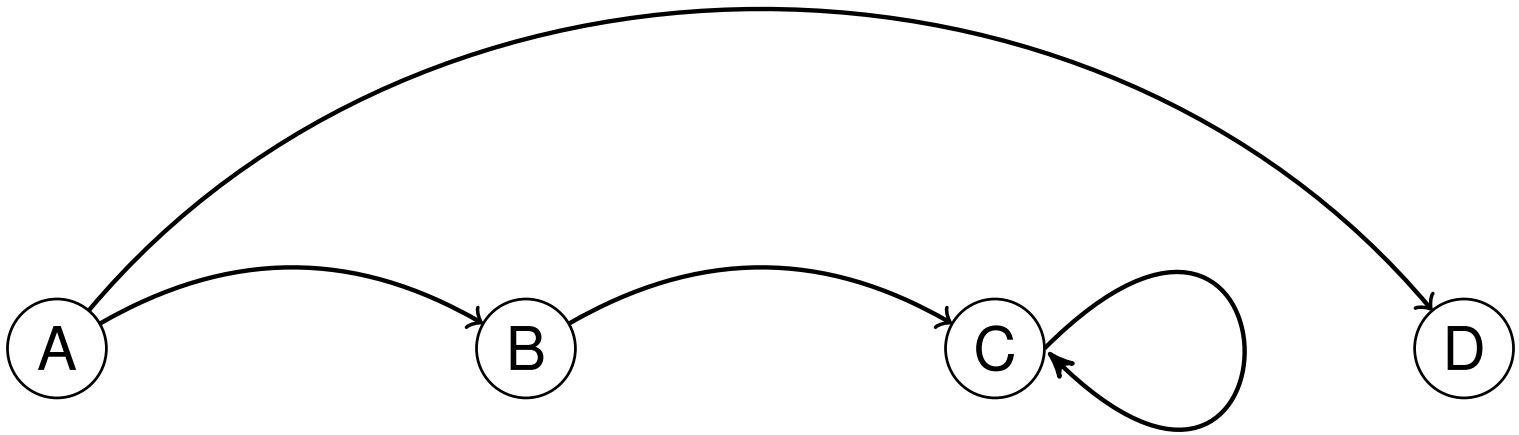
Directed Graphs

- $G = (V,\ A)$ 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**.

Drawing directed graphs

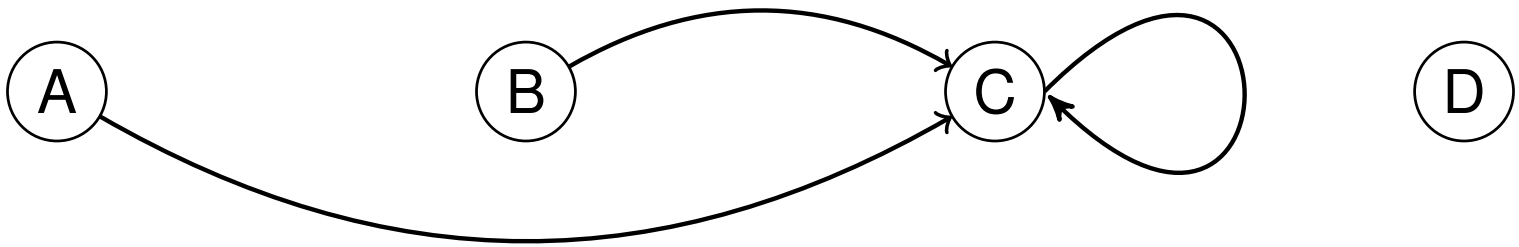
A directed graph

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



Composition example

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



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

- $(A,\ C) \in A \circ A$ by the path $A \to B \to C$
- $(B,\ C) \in A \circ A$ 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 $s_0 \in \pi_1(R)$ (that is there is a pair $(s_0,\ s_1) \in R$).

- The distance from s_0 to s_0 is 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?

Let R be the co-actor relation

```
drop view  if exists 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**.

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. Lets do this all the way up to bacon_number_8 ...

SQL : Bacon number 8

```
drop view if exists bacon_number_8;

create view bacon_number_8 as
 select distinct ca.pid2 as pid,
                 8 as bacon_number
 from bacon_number_7 as bn7
 join coactors as ca on ca.pid1 = bn7.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));
```

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 ;
```

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	102
3	553
4	849
5	325
6	95
7	17

bacon_number_8 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**:

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

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).

CONTEST!!!

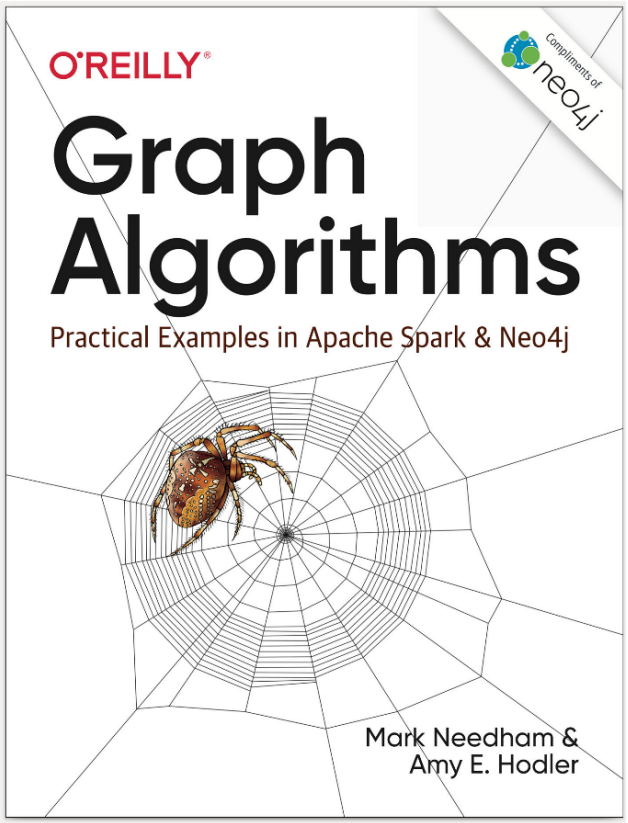
The challenge

HyperSQL implements SQL's **notoriously complicated** (and non-examinable) recursive query constructs. Write a recursive query that takes *n* as a parameter and computes all actors with Bacon number *n*.

Send solutions to `tgg22@cam.ac.uk`.

PRIZES!!!

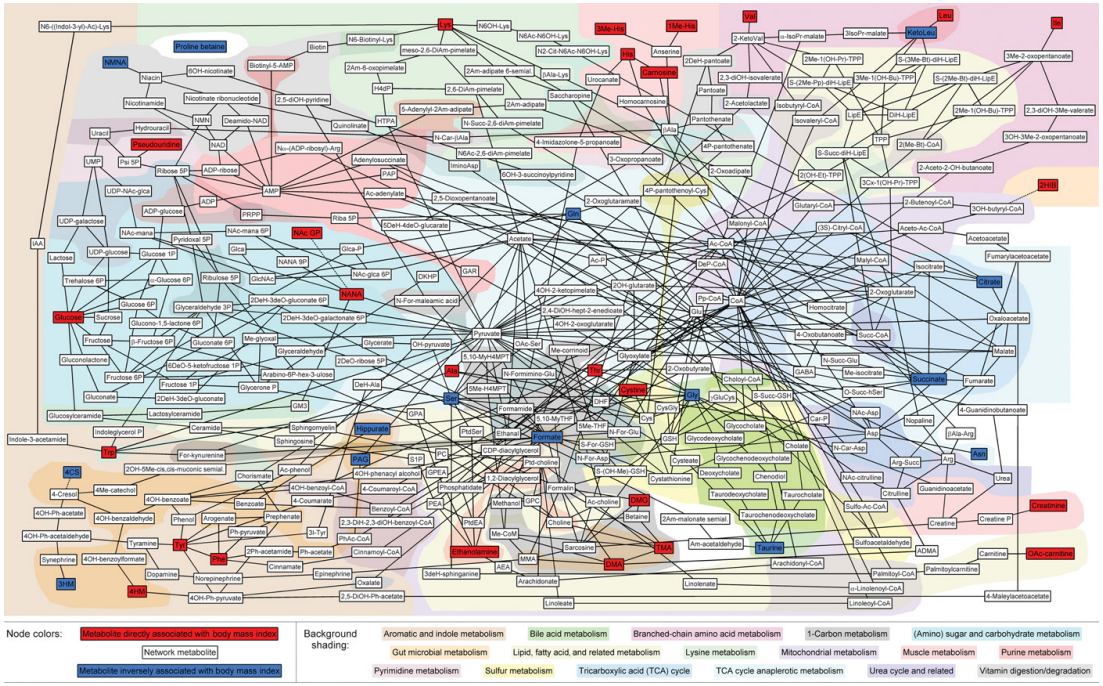
Lecture 8: Graph-oriented Databases



- Model data as a graph (nodes and arcs between nodes).
- Provide a large number of graph algorithms.
- Implement graphs in main memory in such a way that the graph algorithms can be computed efficiently.

The Graph Algorithms book is available on the course web site. Only the contents of the following slides are examinable.

Graph databases are optimised for “Data Science” queries on graphs



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

The global air transport network

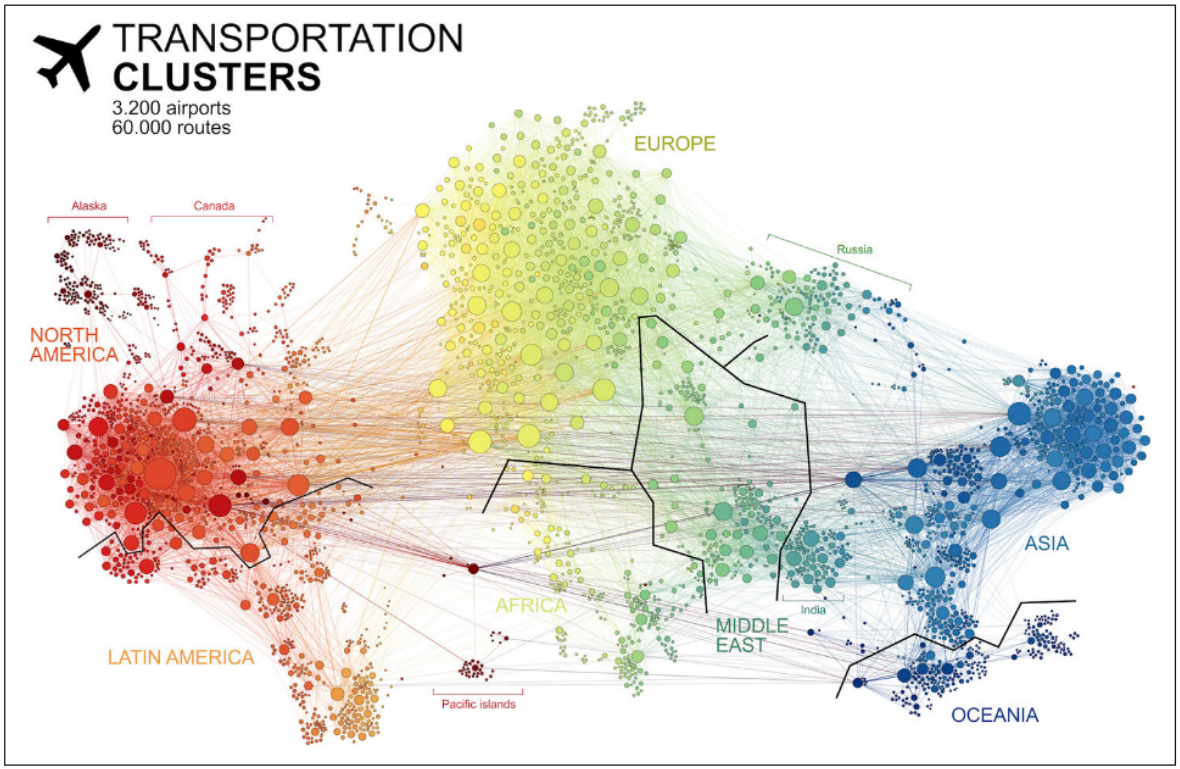
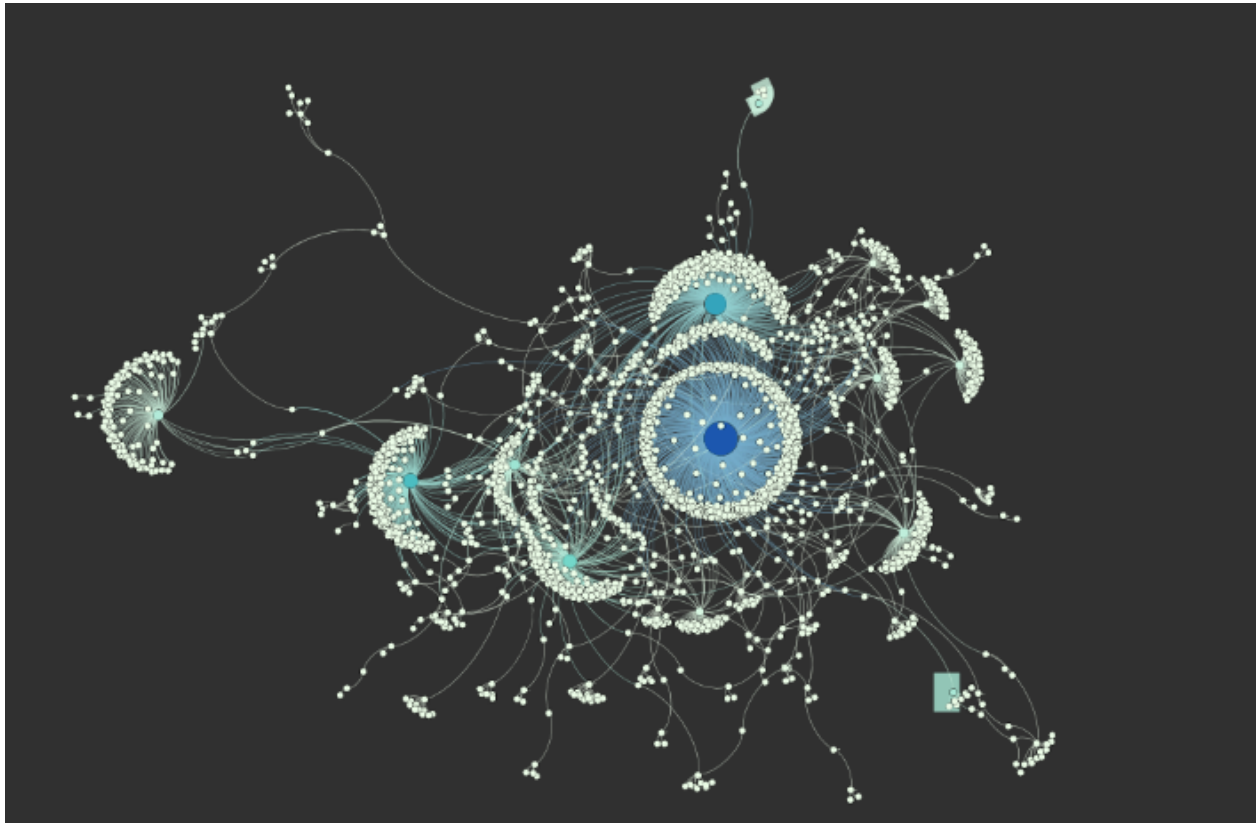


Figure 1-3. Air transportation networks illustrate hub-and-spoke structures that evolve over multiple scales. These structures contribute to how travel flows.

- This air transport graph is from page 5 of Graph Algorithms.
- Analysis have used graph algorithms to better understand how flight delays propagate around the globe.

Social networks



- From **Building Social Network Visualizations** (https://gwu-libraries.github.io/sfm-ui/posts/2017-09-08-sna).
- Graph algorithms are used to recommend new friend links.

Types of graph algorithms for Data Science

- **Community:** How are nodes clustered?
- **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?

Can’t we simply represent graphs in relational tables?

Of course!

But graph-oriented systems optimise implementation

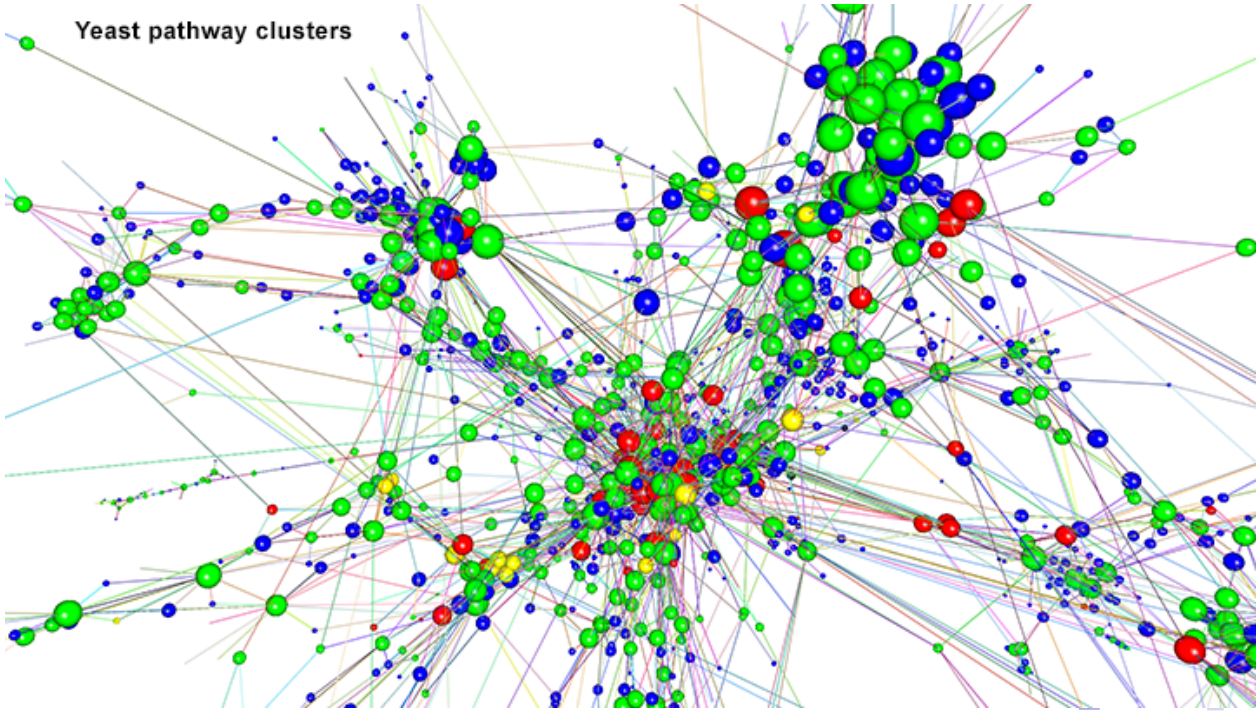
For example, in memory representations can use pointers to implement referential links.

Many SQL-bases systems are fighting back...

Some SQL-based systems are adding features for in-memory tables optimised in similar ways.

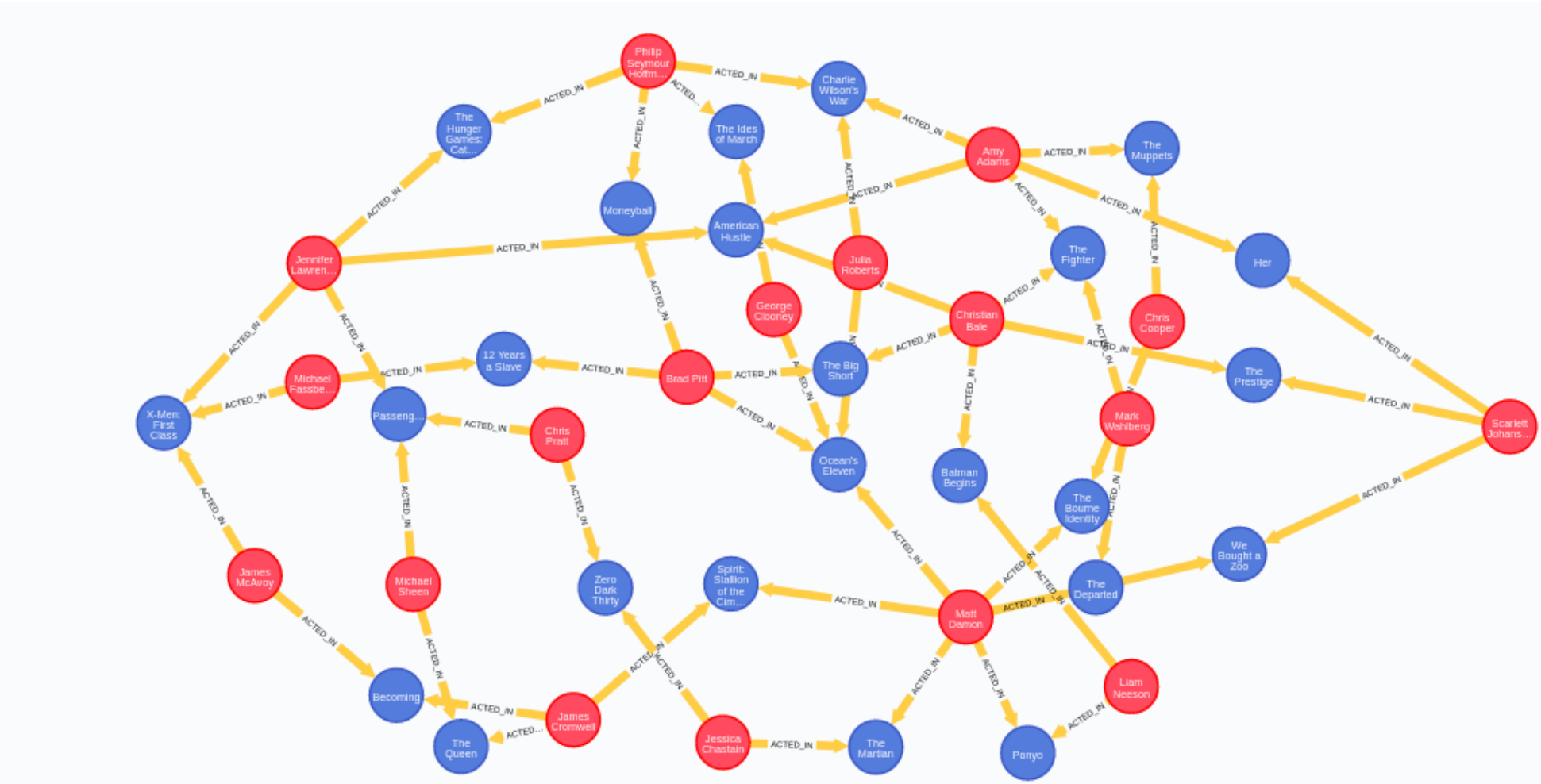
We are using Neo4j

- A Neo4j database contains **nodes** and binary **relationships** between nodes.
- Nodes and relationships can have attributes (called **properties**).
- Neo4j has a query language called **Cypher** that contains path-oriented constructs.
- Complex graph algorithms are implemented in Java and can be called from Cypher queries.



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 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;
```

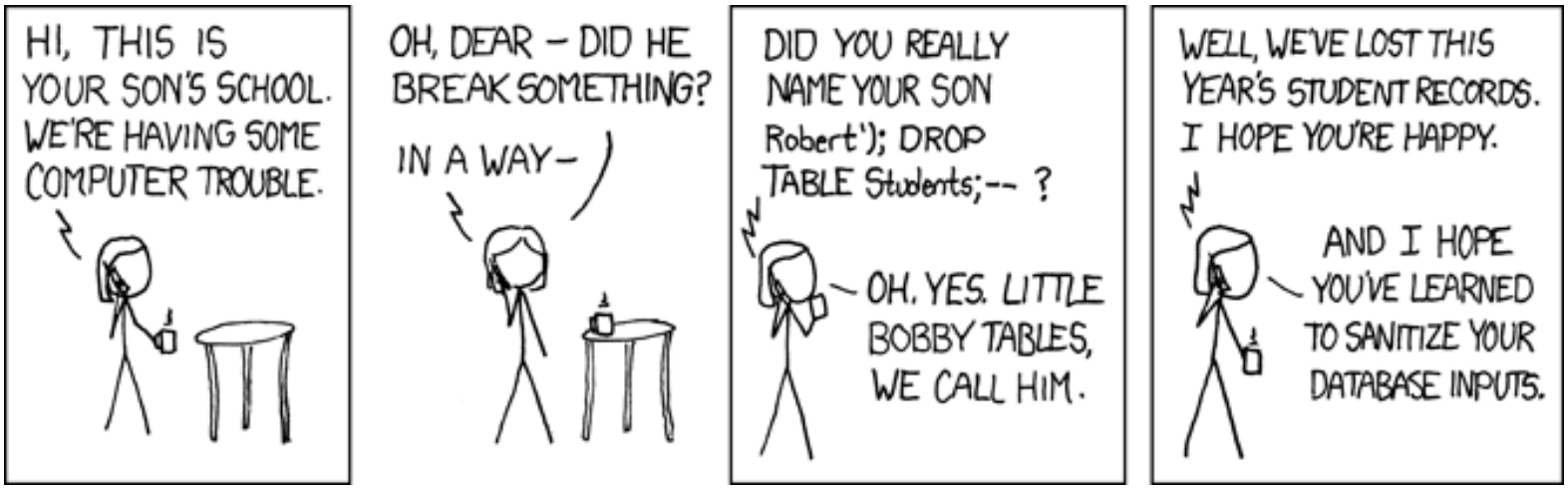
bacon_number	total
1	12
2	102
3	553
4	849
5	325
6	95
7	17

Last Slide!

What have we learned?

- Having a conceptual model of data is very useful, no matter which implementation technology is employed.
- There is a trade-off between fast reads and fast writes.
- There is no databases system that satisfies all possible requirements!
- 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!

The End



(http://xkcd.com/327)