Databases

Timothy G. Griffin

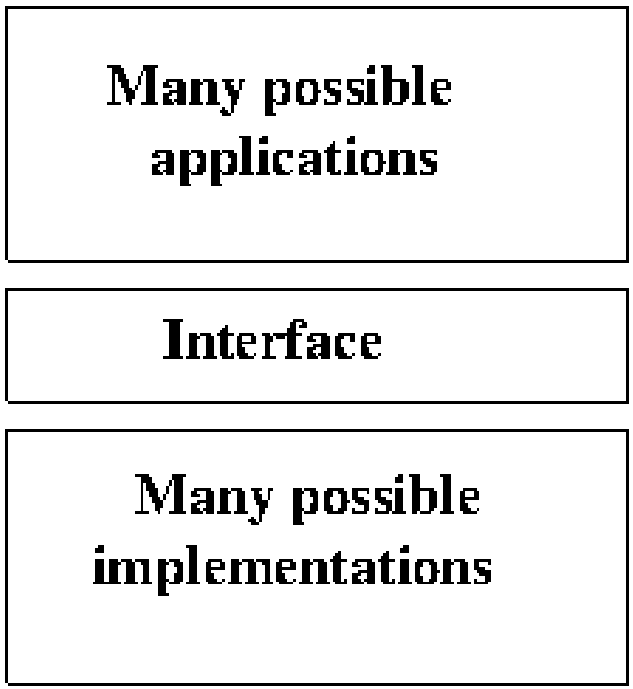
Computer Laboratory
University of Cambridge, UK

Michaelmas 2018

Lecture 1

- What is a Database Management System (DBMS)?
- CRUD and ACID
- Three data models covered
  - Relational
  - Graph-oriented
  - Document-oriented
- Trade-offs in application design
- Trade-offs in DBMS design

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!

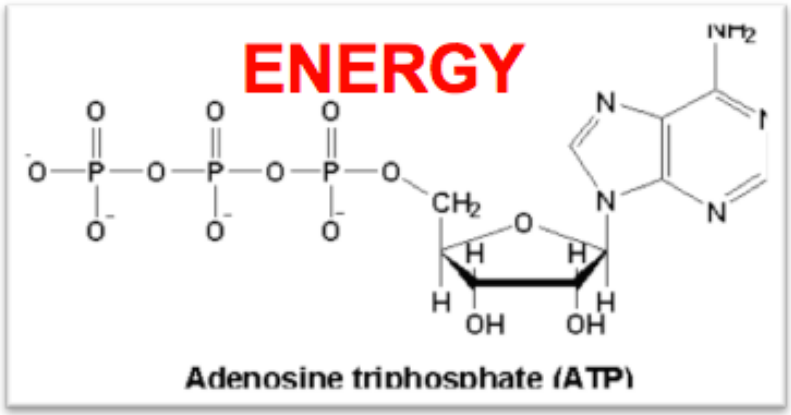
This is found everywhere, not just in computing ...





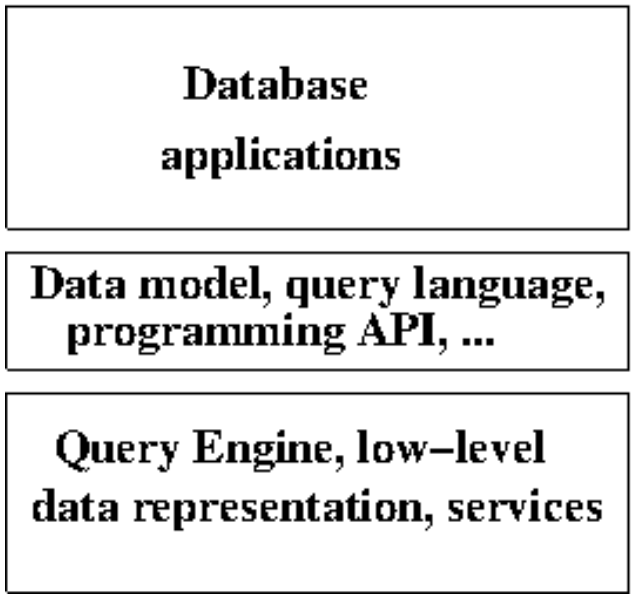


Evolution worked it out long long ago!



Photosynthesis ■ ■ ■ ■ ■ Cellular oxidation

What is a Database Management System (DBMS)?



- This course will stress **data models** and **query languages**. We will not cover programming APIs or network APIs.
- A query engine knows about low-level details hidden by the interface(s). It uses this knowledge to optimize query evaluation.
- Primary service: persistent storage.
- Other services typically implemented:
  - CRUD operations,
  - ACID 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

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).

This course looks at 3 data models

3 models

Relational Model: Data is stored in tables. SQL is the main query language.

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

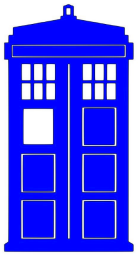
Aggregate-oriented Model: Also called document-oriented database. Optimised for read-oriented databases.

The relational model has been the industry mainstay for the last 35 years. The other two models are representatives of an ongoing revolution in database systems often described under the (misleading) "NoSQL" banner.

This course uses 3 database systems

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

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

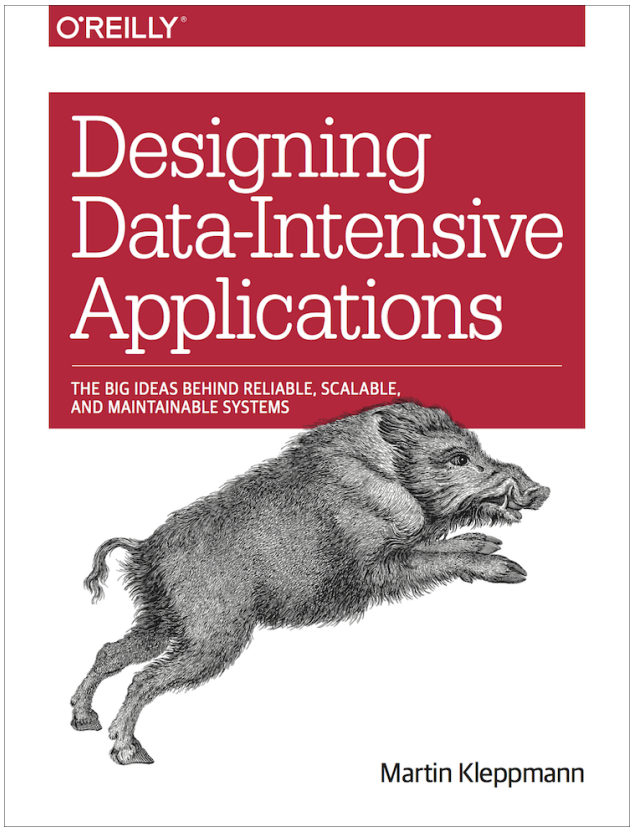


DoctorWho A bespoke **doc**ument-oriented DBMS. Stores JSON objects. Query language is Java. (Under the hood and hidden from you: DoctorWho is implemented with Berkeley DB, a Key-value store.)

IMDb : Our data source

- We have processed raw data available from IMDb (plain text data files at `http://www.imdb.com/interfaces`)
- We started with a snapshot of 11 August, 2017 (4,488,872 movies and 5,431,330 people)
- We then extracted 120 movies and 9283 associated people (actors, directors, etc)
  - These are the top 10 movies for each year from 2006 through 2017, according to the Rotten Tomatoes website.
- This data set was used to generate three database instances, `relational-movie-db`, `graph-movie-db`, and `document-movie-db`.

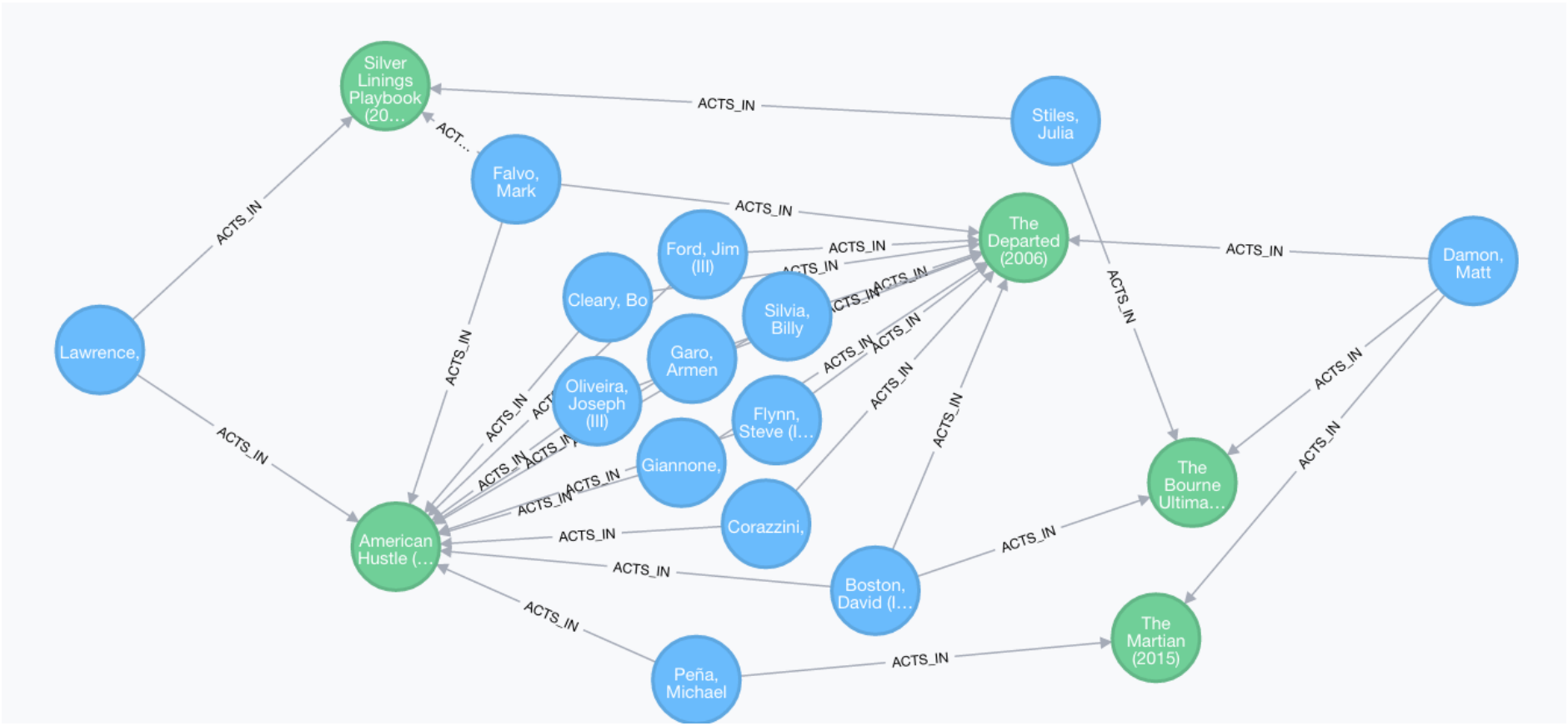
Big thanks to Martin Kleppmann



- Martin is software engineer, entrepreneur, author and speaker.
- He co-founded Rapportive (acquired by LinkedIn in 2012) and Go Test It (acquired by Red Gate Software in 2009)
- Martin provided invaluable assistance with the preparation of this course and with the design and implementation of the infrastructure for the practicals
- `https://martin.kleppmann.com`

Neo4j: Example of path-oriented query in Cypher

```
match path=allshortestpaths(
      (m:Person {name : 'Lawrence, Jennifer (III)'} )
                  [:ACTS_IN*]-
      (n:Person {name : 'Damon, Matt'}))
return path
```



Why are there so many DBMS options?

- We will see that there is no one system that nicely solves all data problems.
- There are several **fundamental trade-offs** faced by application designers and system designers.
- A database engine might be optimised for a particular class of queries.
- The query language might be tailored to the same class.

One important trade-off involves redundant data.

Redundant data

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

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.

Trade-offs often change as technology changes

Expect more dramatic changes in the coming decades ...

1956: A 5 megabyte hard drive

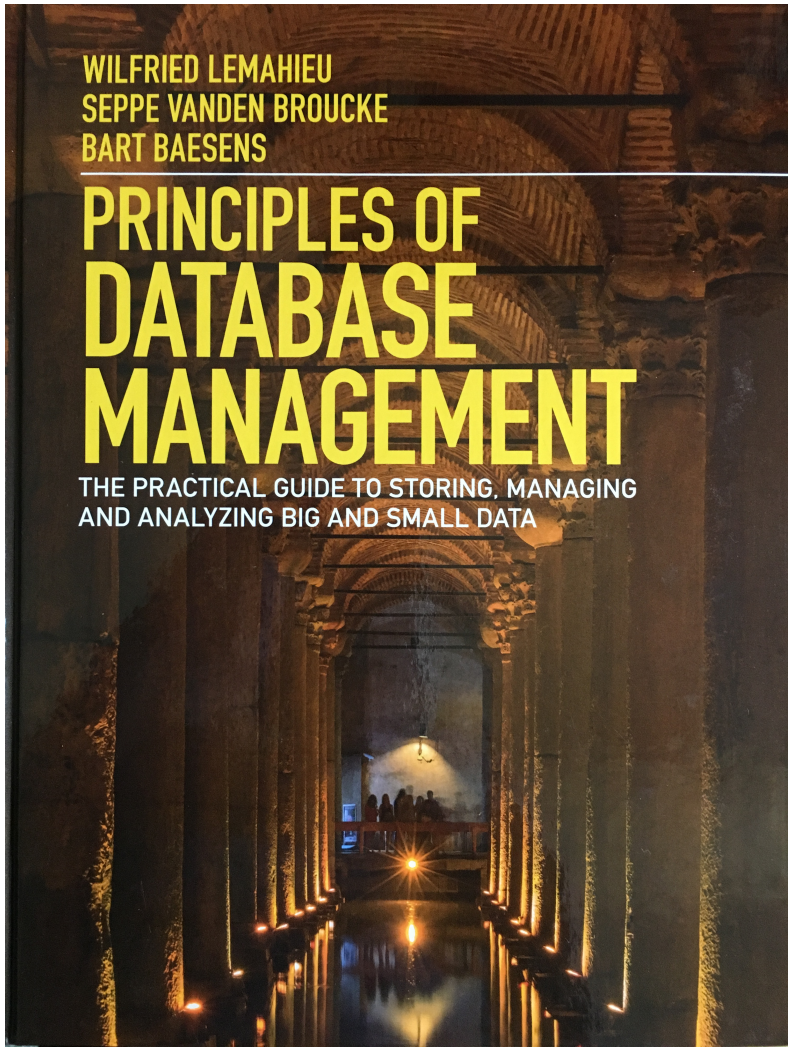
A modern server

Outline

	date	topics
1	4/10	What is a Database Management System (DBMS)?
2	9/10	Entity-Relationship (ER) diagrams
3	11/10	Relational Databases ...
4	16/10	... and SQL
	17/10	**Relational DB practical due**
5	18/10	Some limitations of SQL ...
6	23/10	... that can be solved with Graph Database
	24/10	**Graph DB practical due**
7	25/10	Document-oriented Database
8	30/10	Data warehouse and star schema
	31/11	**Document DB practical due**

Get started **NOW** on the practicals!

New 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
8. Data warehouse and star schema
   - Chapter 17

Lecture 2

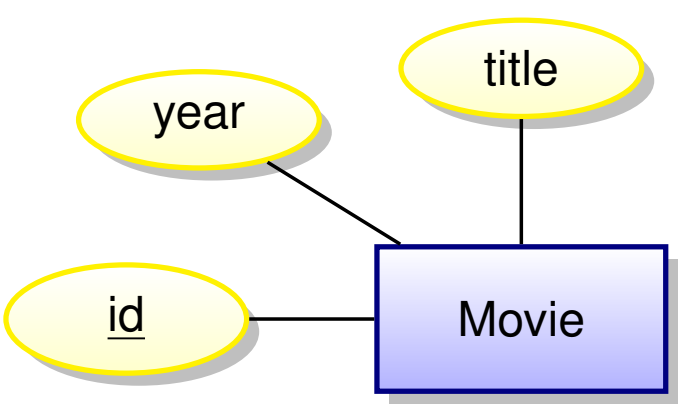
- Conceptual modeling with Entity-Relationship (ER) diagrams
- Entities, attributes, and relationships
- Weak entities
- Cardinality of a relationship

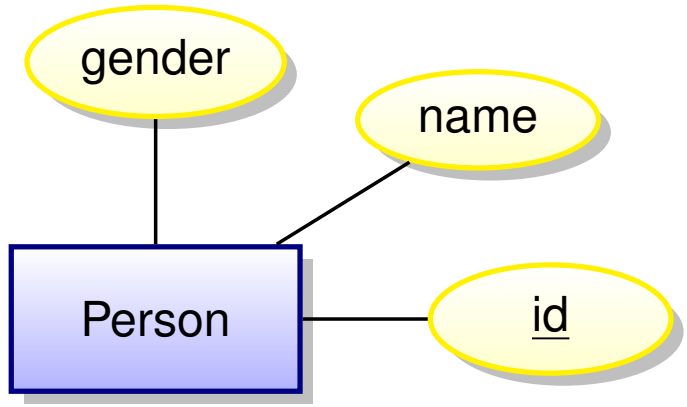
Conceptual modeling with Entity-Relationship (ER) diagrams

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

- title: Gravity, year : 2013, id : 2945287
- title : The Bourne Ultimatum, year : 2007, id : 3527294

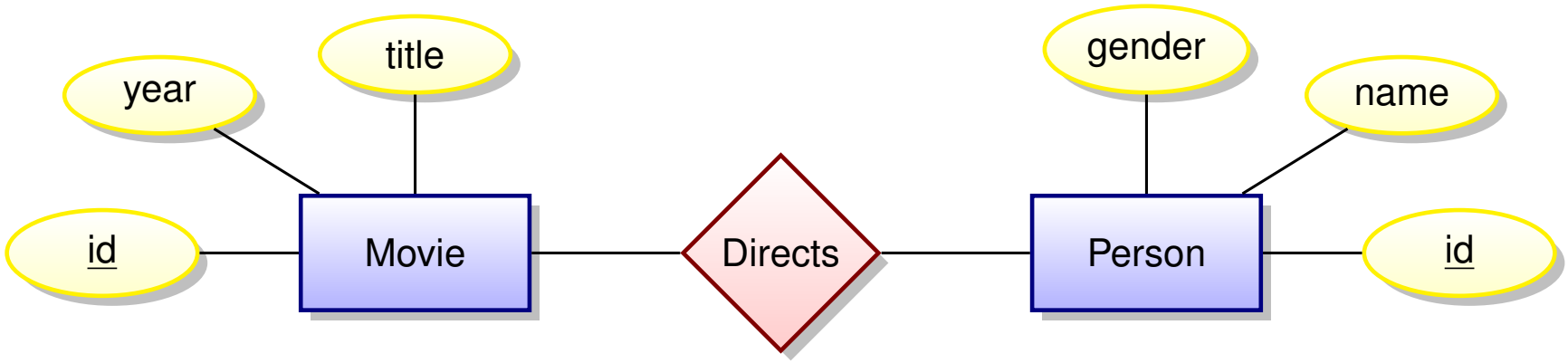
Instances of the Person entity

- name: Jennifer Lawrence, gender: female, id : 3018535
- name: Matt Damon, gender: male, id : 473946

Where do keys come from?

They 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



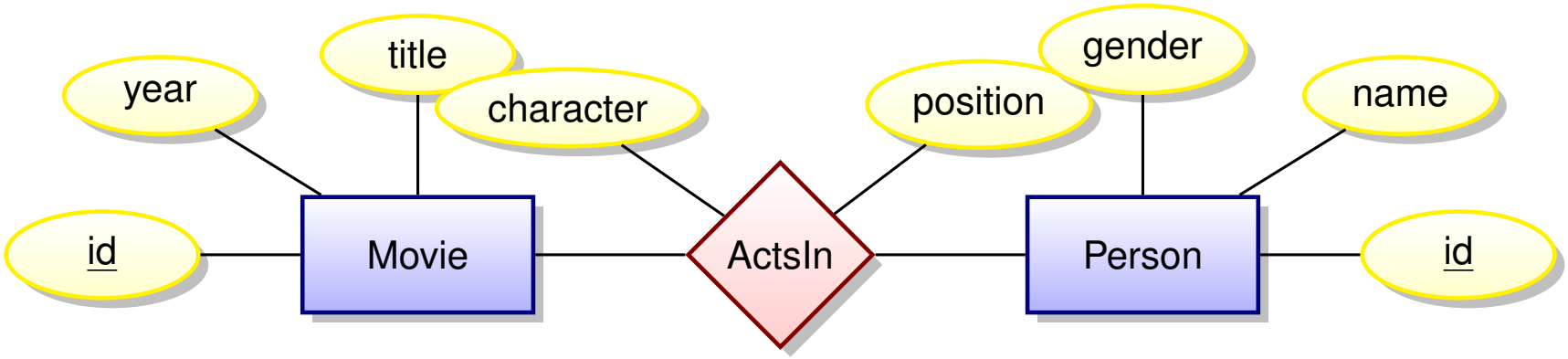
- Relationships (diamonds) represent the verbs of our domain.
- Relationships are between entities.

Relationship instances

Instances of the Directs relationship (ignoring entity attributes)

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

Relationships can have attributes



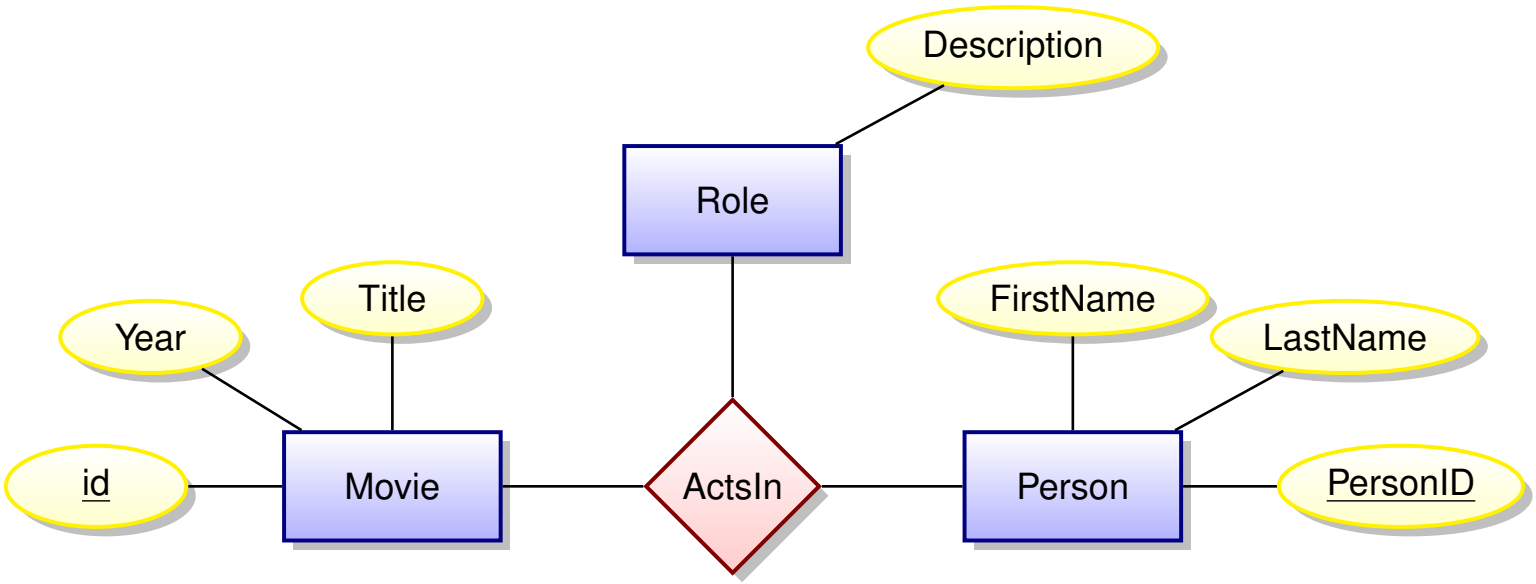
- Attribute **character** indicates the role played by a person
- Attribute **position** indicates the order listed in movie credits

Relationship instances

Instances of the ActsIn relationship (ignoring entity attributes)

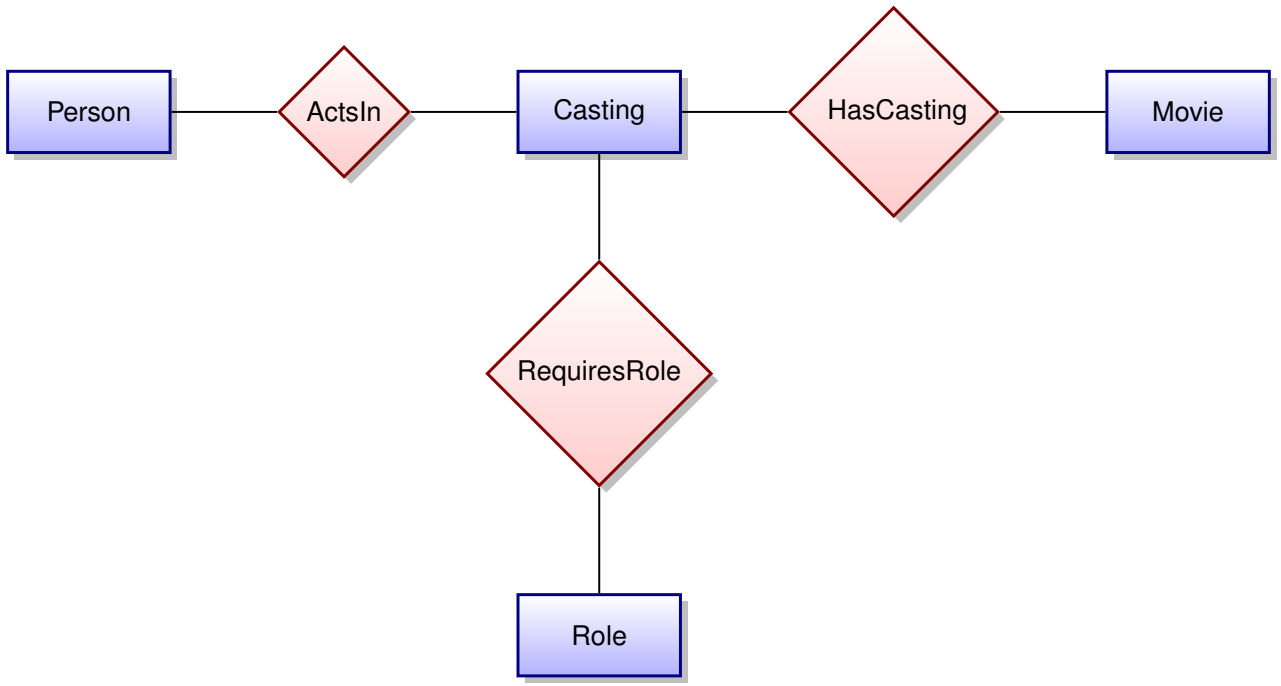
- Ben Affleck plays Tony Mendez in Argo, billing position 1
- Julie Deply plays Celine in Before Midnight, billing position 2
- Bradley Cooper plays Pat in Silver Linings Playbook, billing position 1
- Jennifer Lawrence plays Tiffany in Silver Linings Playbook, billing position 2
- Tim Allan plays Buzz Lightyear in Toy Story 3, billing position 2

Could **ActsIn** be modeled as a Ternary Relationship?



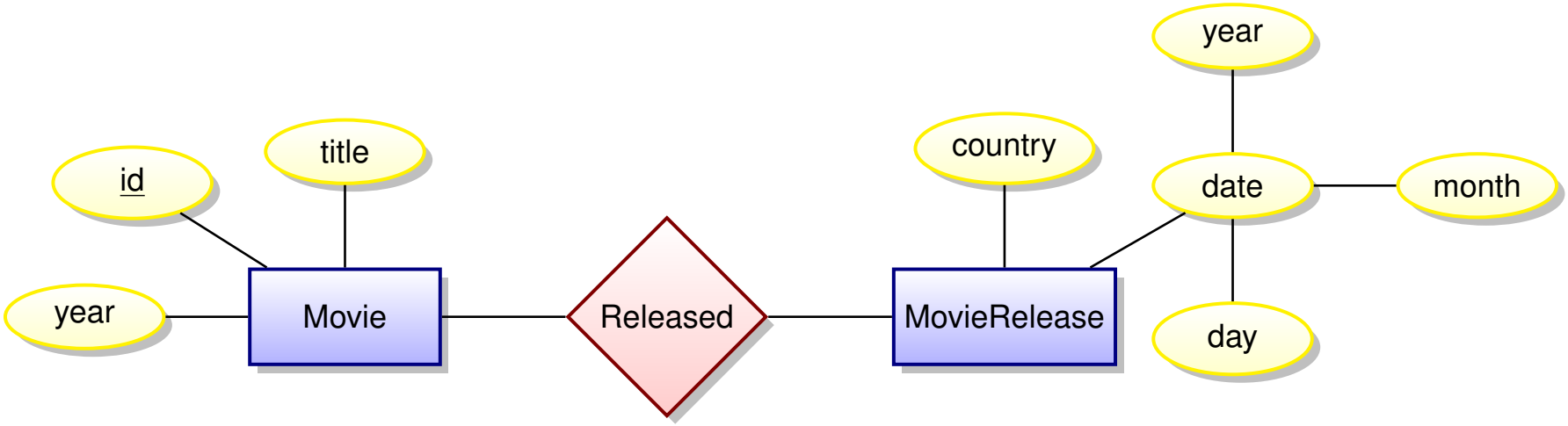
This might be a good model if our movies where in a restricted domain where roles have an independent existence. For example, suppose we are building a database of Shakespearean movies.

Can a ternary relationship be modeled with multiple binary relationships?



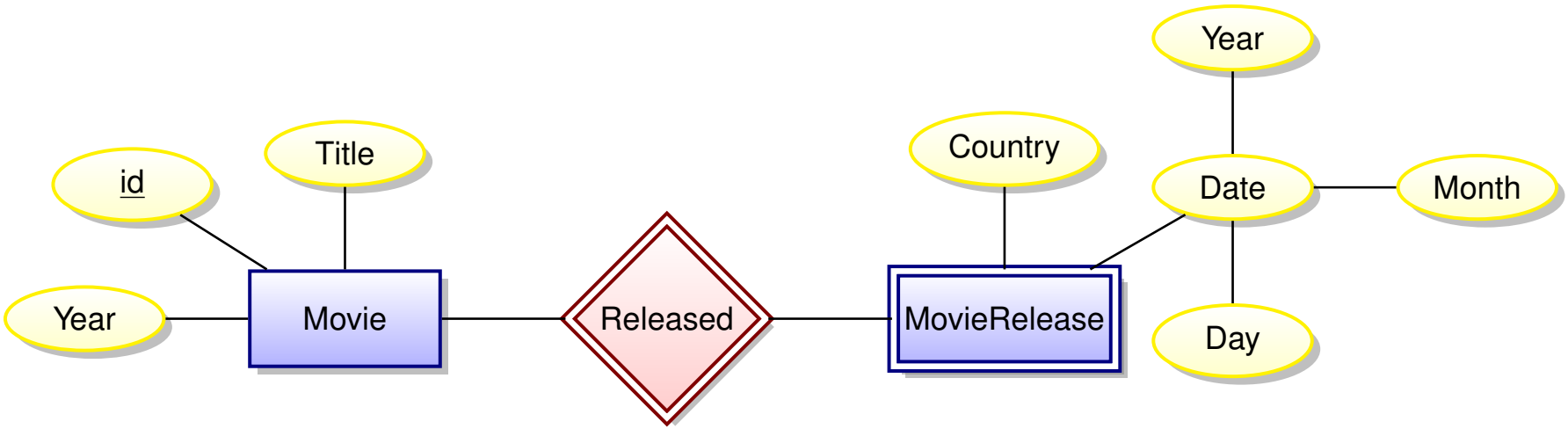
Is the **Casting** entity too artificial?

Attribute or entity with new relationship?



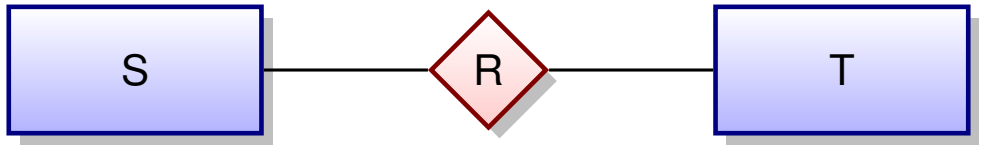
- Should release date be an attribute or an entity?
- The answer may depend on the **scope** of your data model.
- If all movies within your scope have at most one release date, then an attribute will work well.
- However, if you scope is global, then a movie can have different release dates in different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

Weak entities



- MovieRelease is an example of a **weak entity**
- The existence of a weak entity depends on the existence of another entity. In this case, a release date exists only for a given movie.
- Released is called an **identifying relationship**

Cardinality of a relationship



The relation R is

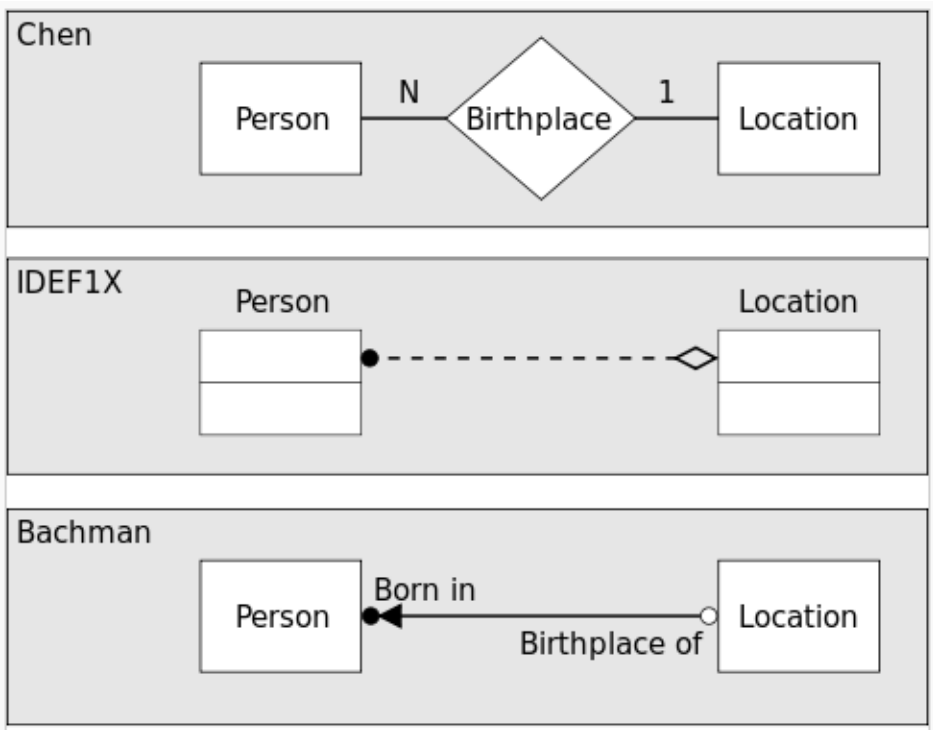
one-to-many: Every member of T is related to at most one member of S.

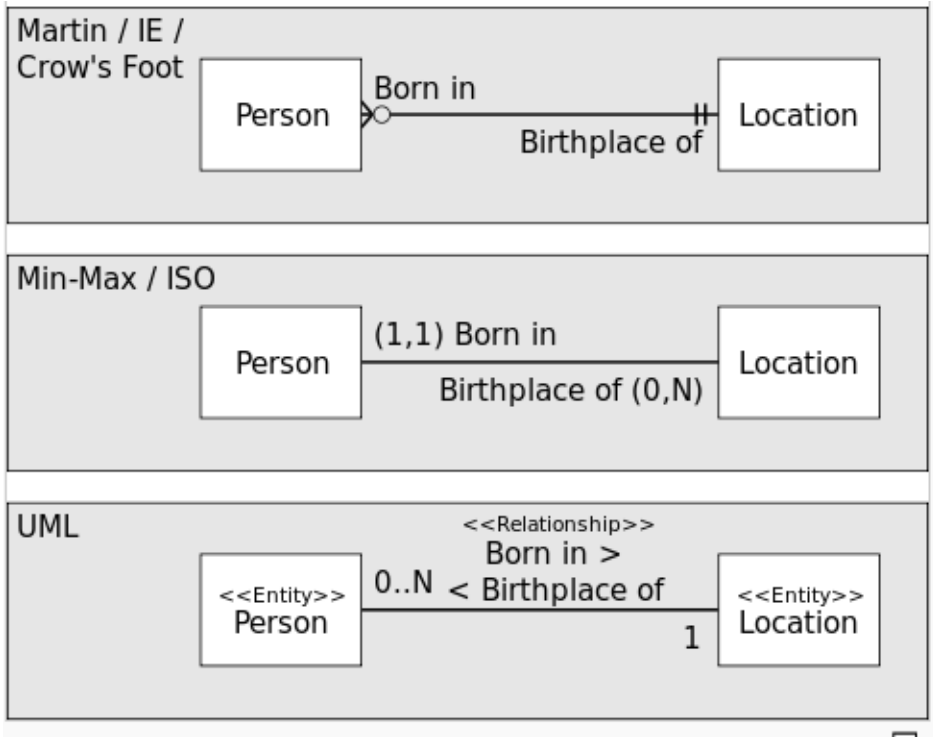
many-to-one: Every member of S is related to at most one member of T.

one-to-one: R is both many-to-one and one-to-many.

many-to-many: No constraint.

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





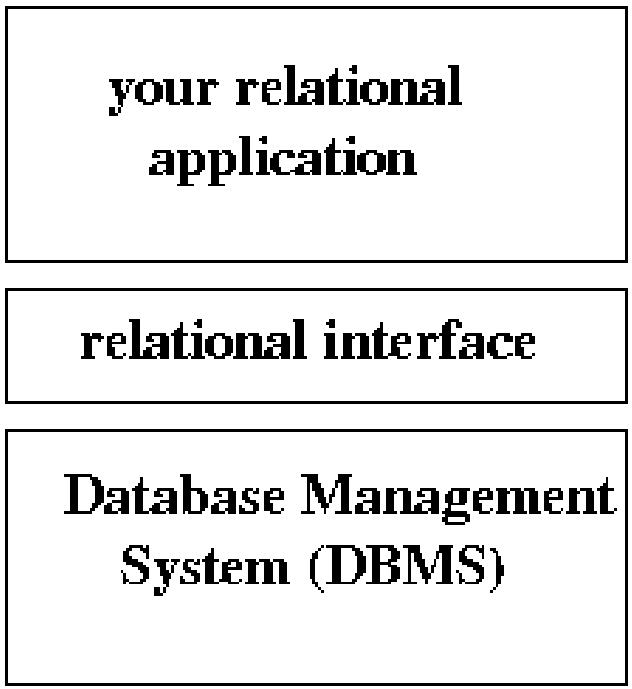
Various diagrammatic notations used to indicate a one-to-many relationship (`https://en.wikipedia.org/wiki/Entity-relationship_model`).

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

Lectures 3 and 4

- The relational Model
- The Relational Algebra (RA)
- SQL
- Implementing an ER model in the relational model
- Update anomalies
- Avoiding 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'

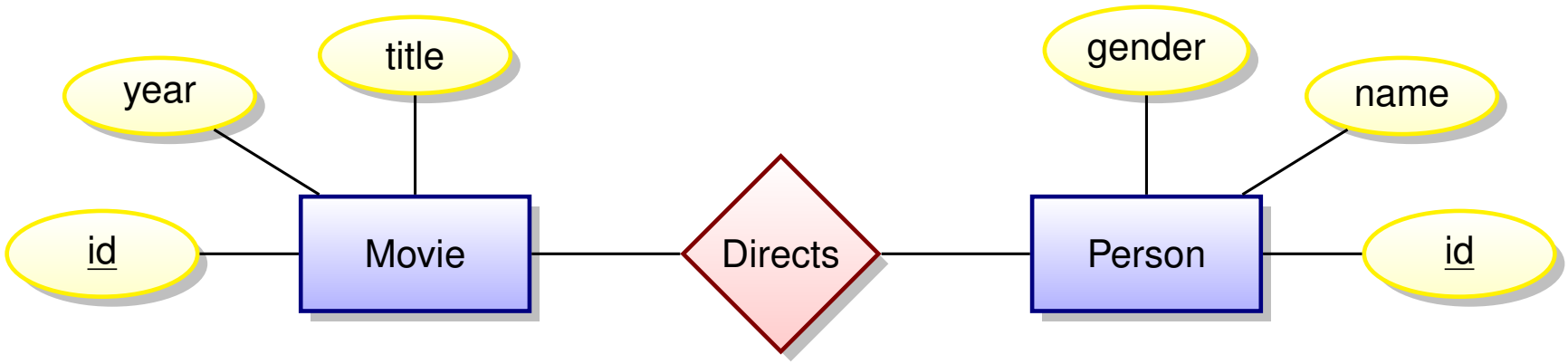
$\Longrightarrow$

Students $\bowtie$ Colleges

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

(We will look at joins in SQL very soon ...)

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

How about one table?

DirectsComplete

mid	title	year	pid	name	gender
2544956	12 Years a Slave (2013)	2013	1390487	McQueen, Steve (III)	male
2552840	4 luni, 3 saptamâni si 2 zile (2007)	2007	1486556	Mungiu, Cristian	male
2589088	Afghan Star (2009)	2009	3097222	Marking, Havana	female
2607939	American Hustle (2013)	2013	1835629	Russell, David O.	male
2611256	An Education (2009)	2009	3404232	Scherfig, Lone	female
2622261	Anvil: The Story of Anvil (2008)	2008	751015	Gervasi, Sacha	male
2626541	Argo (2012)	2012	16507	Affleck, Ben	male
2629853	Aruitemo aruitemo (2008)	2008	1133907	Koreeda, Hirokazu	male
. . .	. . .	. . .	. . .	. . .	. . .

This works, but ....

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

id	title	year
2544956	12 Years a Slave (2013)	2013
2552840	4 luni, 3 saptamâni si 2 zile (2007)	2007
2589088	Afghan Star (2009)	2009
2607939	American Hustle (2013)	2013
2611256	An Education (2009)	2009
2622261	Anvil: The Story of Anvil (2008)	2008
2626541	Argo (2012)	2012
2629853	Aruitemo aruitemo (2008)	2008
. . .	. . .	. . .

People

id	name	gender
1390487	McQueen, Steve (III)	male
1486556	Mungiu, Cristian	male
3097222	Marking, Havana	female
1835629	Russell, David O.	male
3404232	Scherfig, Lone	female
751015	Gervasi, Sacha	male
16507	Affleck, Ben	male
1133907	Koreeda, Hirokazu	male
. . .	. . .	. . .

What about the relationship?

Directs

movie_id	person_id
2544956	1390487
2552840	1486556
2589088	3097222
2607939	1835629
2611256	3404232
2622261	751015
2626541	16507
2629853	1133907
...	...

(No, this relation does not actually exist in our IMDb databases — more on that later ...)

We can recover DirectsComplete using a relational query

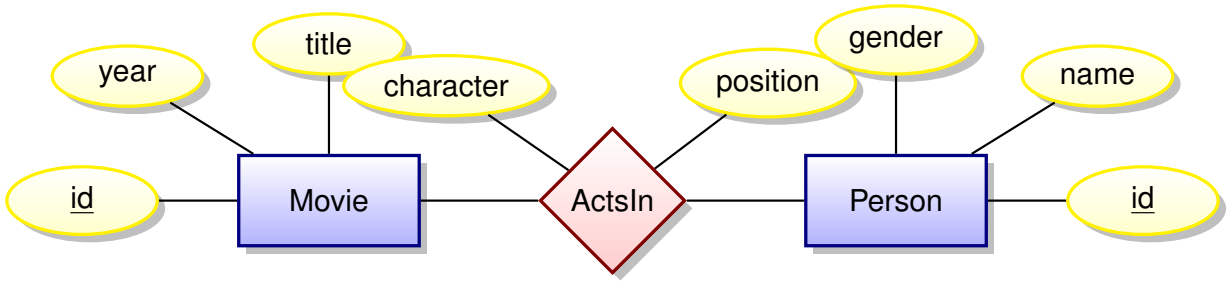
The SQL query

```
select movies.id as mid, title, year,
       people.id as pid, name, gender
from movies
join directs on movie_id = movies.id
join people on people.id = person_id
```

might return something like

MID	TITLE	YEAR	PID	NAME	GENDER
2544956	12 Years a Slave (2013)	2013	1390487	McQueen, Steve (III)	male
2552840	4 luni, 3 saptamâni si 2 zile (2007)	2007	1486556	Mungiu, Cristian	male
2589088	Afghan Star (2009)	2009	3097222	Marking, Havana	female
2607939	American Hustle (2013)	2013	1835629	Russell, David O.	male
2611256	An Education (2009)	2009	3404232	Scherfig, Lone	female
2622261	Anvil: The Story of Anvil (2008)	2008	751015	Gervasi, Sacha	male
2626541	Argo (2012)	2012	16507	Affleck, Ben	male
2629853	Aruitemo aruitemo (2008)	2008	1133907	Koreeda, Hirokazu	male
...	...	...	...	...	...

In a similar way with the ActsIn relationship



The ActsIn Relationship requires attributes

movie_id	person_id	character	position
2544956	146271	Judge Turner	4
2544956	2460265	Mistress Ford	32
2544956	173652	Mr. Moon	9
2544956	477824	Tibeats	35
2544956	256114	Edward	42
2544956	2826281	Tea Seller	NULL
...	...	...	...

More on NULL soon ...

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 relational representation

A relational schema

$$ActsIn(\underline{movie\_id}, \underline{person\_id})$$

With referential integrity constraints

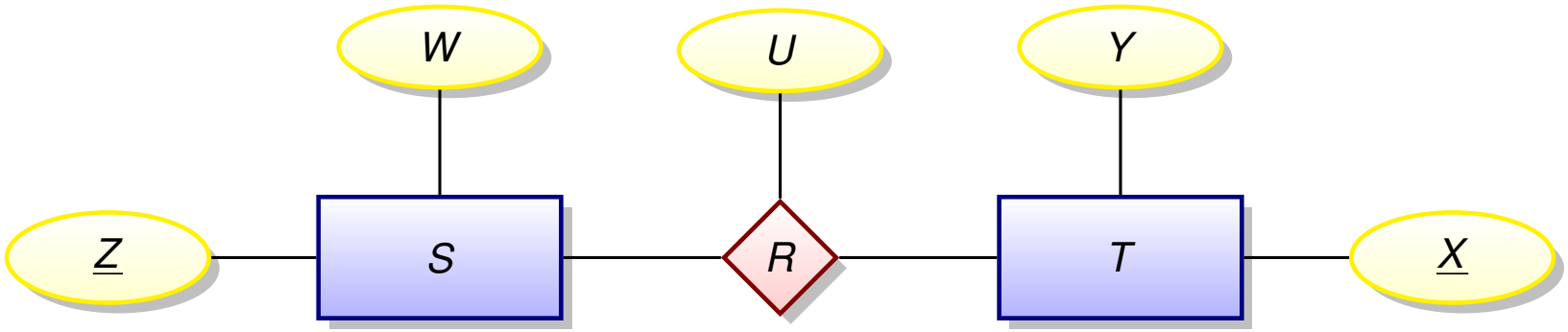
$$\pi_{movie\_id}(ActsIn) \subseteq \pi_{id}(Movies)$$

$$\pi_{person\_id}(ActsIn) \subseteq \pi_{id}(People)$$

Foreign Keys in SQL

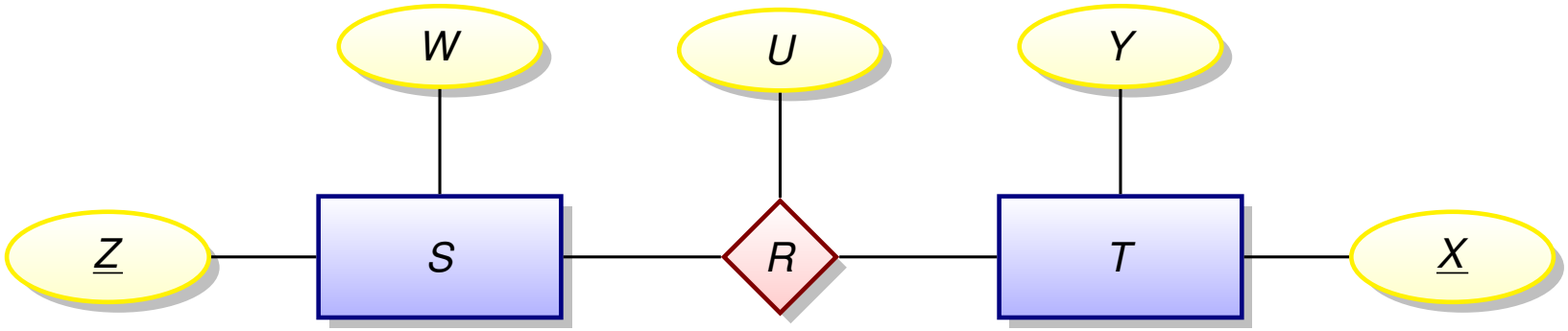
```
create table ActsIn (
   movie_id int not NULL,
   person_id int not NULL,
   character varchar(255),
   position  integer,

   primary key (movie_id, person_id),
   constraint actsin_movie
         foreign key (movie_id)
         references Movies(id),
   constraint actsin_person
         foreign key (person_id)
         references People(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 ($M : 1$)	$R(X, \underline{Z}, U)$
one to one ($1 : 1$)	$R(\underline{X}, Z, U)$ and/or $R(X, \underline{Z}, U)$

Implementation can differ from the “clean” approach

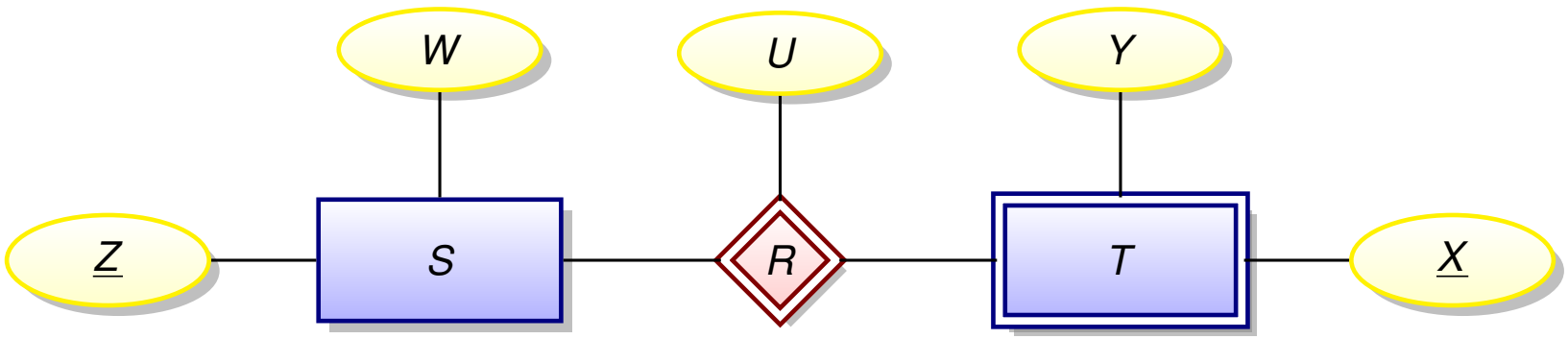


Suppose R is one to many

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

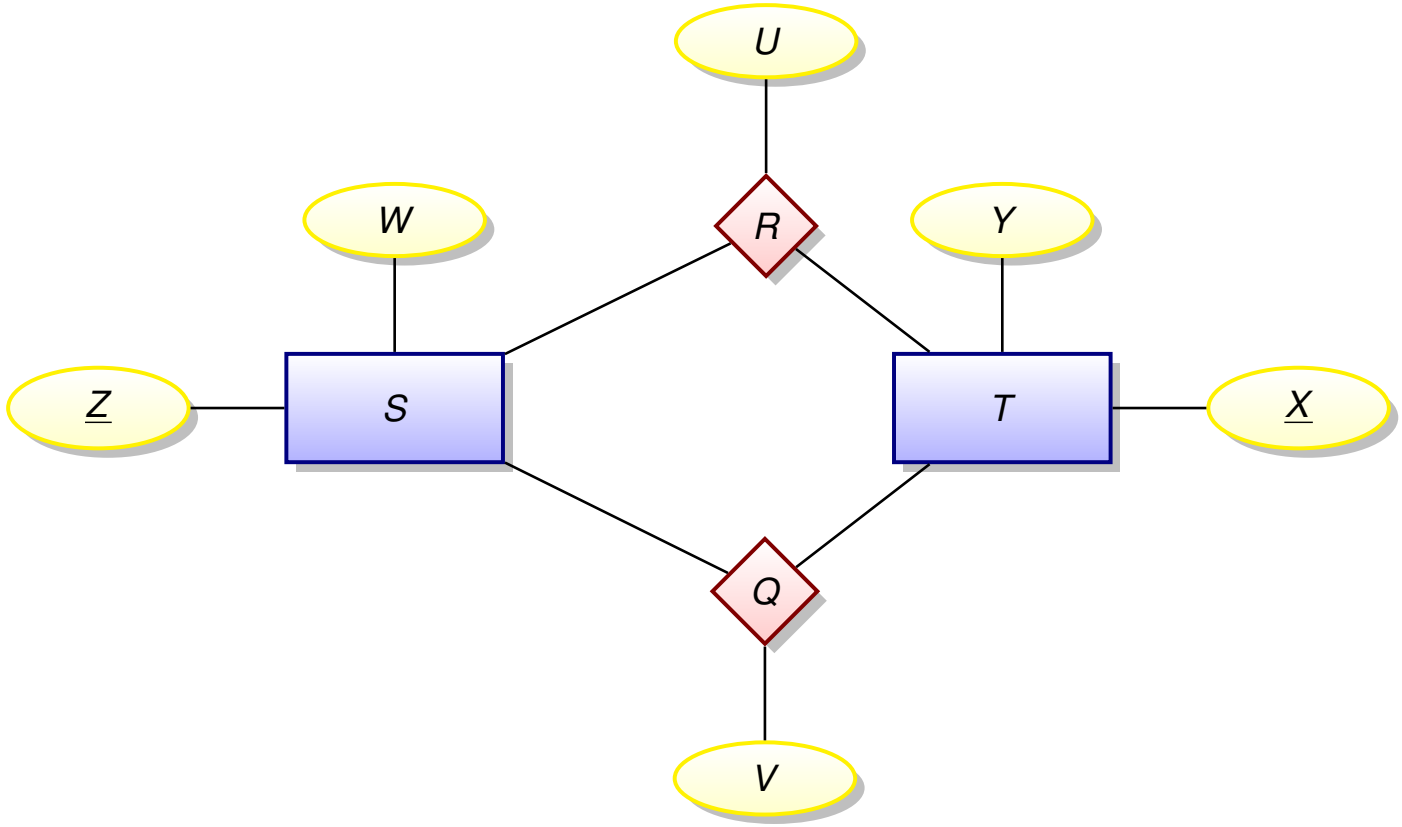


This is always a one to many relationship!

- 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 $T(\underline{Y},\ \underline{Z},\ \underline{U})$.
- This is exactly what we have done in our movies database with these tables: release_dates, certificates, color_info, genres, keywords, languages, locations, running_times.

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 8 relationships into the `credits` table!

```
SELECT type, count(*) AS total
FROM credits
GROUP BY type
ORDER BY total DESC;
```

Using our database this query produces the output

```
TYPE                 TOTAL
-------------------  -----
actor                 8467
producer              1108
writer                 289
editor                 161
director               139
cinematographer        131
composer               125
costume_designer        90
```

Was this a good idea?

Discuss!

Lectures 5 and 6

- 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
- Graph-oriented databases: optimised for computing transitive closure
- Neo4j
- Bacon number with Neo4j

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!

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**!

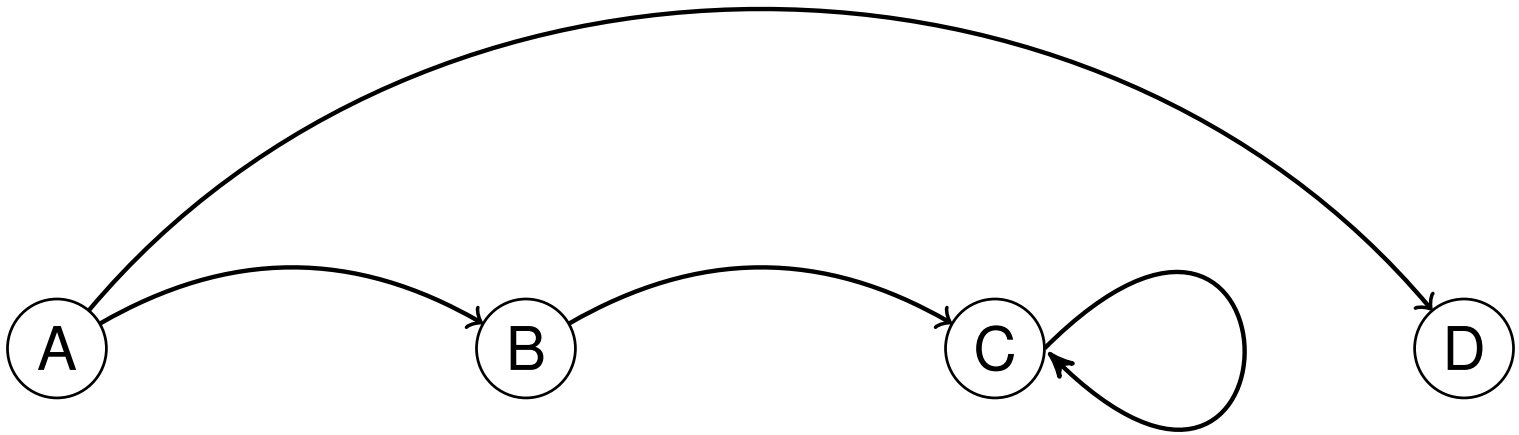
Directed Graphs

- $G = (V,\ A)$ 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

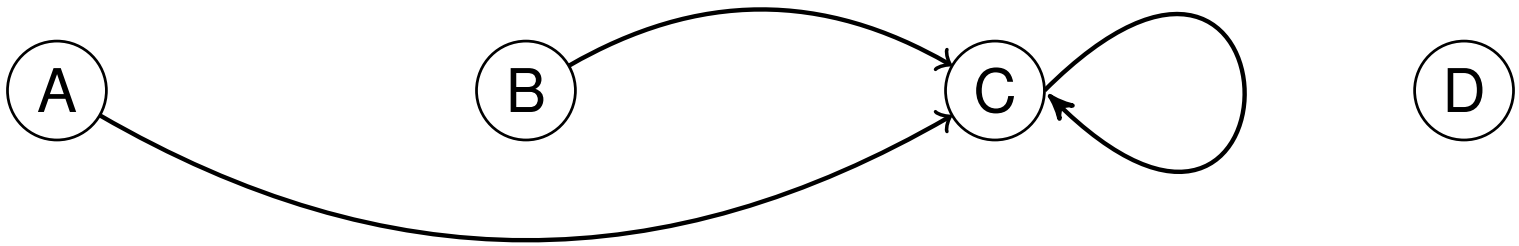
A directed graph

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



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 compostion** 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 coactor relation

```
select DISTINCT c1.person_id as pid1,
                c2.person_id as pid2
from credits AS c1
join credits AS c2 on  c2.movie_id = c1.movie_id
where c1.type = 'actor'
  and c2.type = 'actor';
```

On our database this relation contains 933,360 rows. Note that this relation is **reflexive** and **symmetric**.

We will often use the coactor relation, so let's make a **view** of it ...

```
create view coactors as
  select DISTINCT c1.person_id as pid1,
                  c2.person_id as pid2
  from credits AS c1
  join credits AS c2 on  c2.movie_id = c1.movie_id
  where c1.type = 'actor'
    and c2.type = 'actor';
```

Views

This view `coactors(pid1, pid2)` can now be used in SQL queries as if it were a table.

SQL : Bacon numbers 1 and 2

```
create view bacon_number_1 as
 select distinct pid2 as pid, 1 as bn
 from coactors
 where pid1 = 121299 and not pid2 = 121299;

create view bacon_number_2 as
 select distinct pid2 as pid, 2 as bn
 from coactors
 join bacon_number_1 on pid1 = pid
 where pid2 not in (select pid from bacon_number_1)
   and not pid2 = 121299;
```

Kevin Bacon's id is 121299.

SQL : Bacon numbers 3 and 4

```
create view bacon_number_3 as
 select distinct pid2 as pid, 3 as bn
 from coactors
 join bacon_number_2 on pid1 = pid
 where pid2 not in (select pid from bacon_number_1)
   and pid2 not in (select pid from bacon_number_2)
   and not pid2 = 121299;

create view bacon_number_4 as
 select distinct pid2 as pid, 4 as bn
 from coactors
 join bacon_number_3 on pid1 = pid
 where pid2 not in (select pid from bacon_number_1)
   and pid2 not in (select pid from bacon_number_2)
   and pid2 not in (select pid from bacon_number_3)
   and not pid2 = 121299;
```

Bacon Numbers

```
select bn, count(*) as total from bacon_number_1 group by bn
union
select bn, count(*) as total from bacon_number_2 group by bn
union
select bn, count(*) as total from bacon_number_2 group by bn
union
select bn, count(*) as total from bacon_number_4 group by bn
union
select bn, count(*) as total from bacon_number_5 group by bn
union
select bn, count(*) as total from bacon_number_6 group by bn;
```

Results

BN	TOTAL
1	105
2	1428
3	4308
4	980
5	203

Bacon numbers 2, with no views

```
select distinct c4.person_id as pid, 2 as bn
from credits AS c1
join credits AS c2 on  c2.movie_id  = c1.movie_id
join credits AS c3 on  c3.person_id = c2.person_id
join credits AS c4 on  c4.movie_id  = c3.movie_id
where c4.person_id not in (
   select c2.person_id
   from credits AS c1
   join credits AS c2 on  c2.movie_id = c1.movie_id
   where c1.type = 'actor'     and c2.type = 'actor'
     and c1.person_id = 121299 and not c2.person_id = 121299
)
and c1.type = 'actor'     and c2.type = 'actor'
and c3.type = 'actor'     and c4.type = 'actor'
and c1.person_id = 121299 and not c2.person_id = 121299
and not c4.person_id = 121299;
```

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).

Observations

- We could continue, but the queries to compute higher and higher bacon numbers will grow in size and complexity.
- Performance will degrade rapidly.

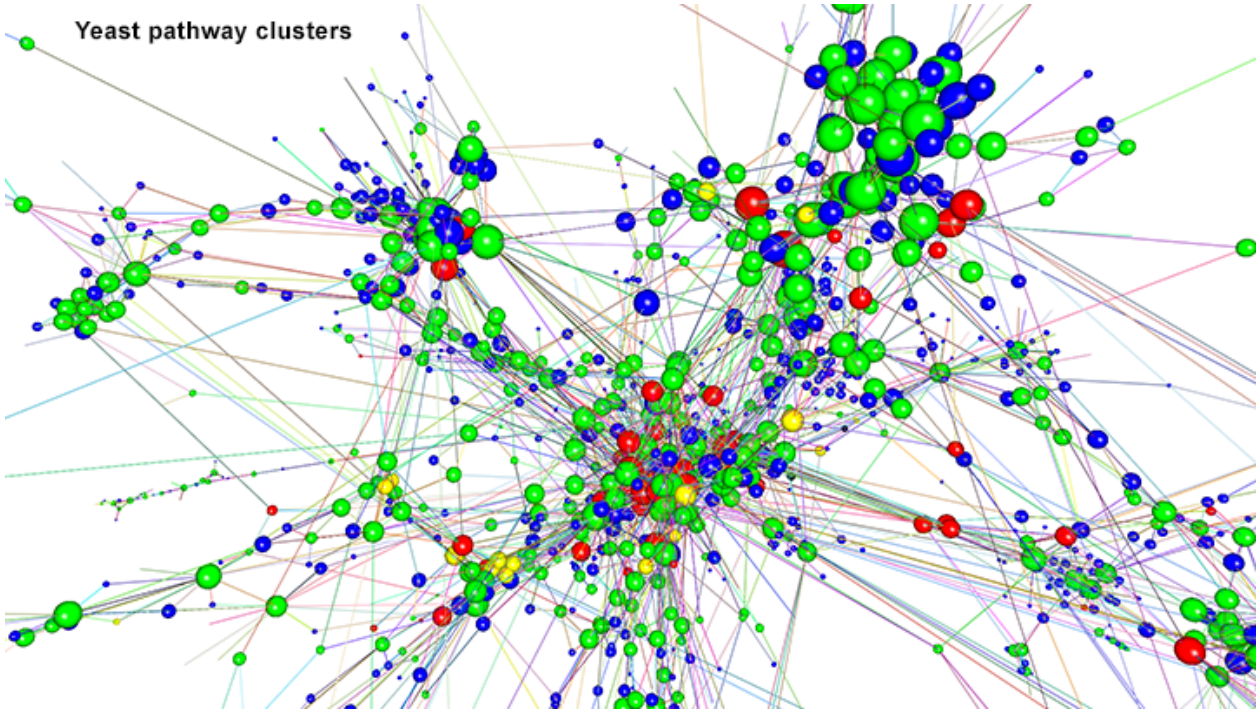
BIG IDEA

Is it possible to design and implement a database system that is **optimised for transitive closure and related path-oriented queries**?

Yes: This is one of the motivations behind graph-oriented databases.

We will be 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. It is designed to explore very large graphs.



Let's compute **all** Bacon numbers

```
MATCH (p:Person)
where p.name <>  "Bacon, Kevin (I)"
with p
match paths=allshortestpaths(
      (m:Person {name : "Bacon, Kevin (I)"} )
         -[:ACTS_IN*]- (n:Person {name : p.name}))
return distinct p.name,
       length(paths)/2 as bacon_number
order by bacon_number desc;
```

```
+-------------------------------------------------+
| p.name                          | bacon_number  |
+-------------------------------------------------+
| "Alberson, Sarah"               | 5             |
| "Allee, Virginia (I)"           | 5             |
| "Allen, Brian Keith"            | 5             |
| "Allen, Tess"                   | 5             |
| "Arbogast, Jeremiah"            | 5             |
    ....                            ...
```

Let's compute count Bacon numbers

```
MATCH (p:Person)
where p.name <>  "Bacon, Kevin (I)"
with p
match paths=allshortestpaths(
      (m:Person {name : "Bacon, Kevin (I)"} )
         -[:ACTS_IN*]- (n:Person {name : p.name}))
return count(distinct p.name) as total,
       length(paths)/2 as bacon_number
order by bacon_number desc;
```

```
+----------------------+
| total | bacon_number |
+----------------------+
| 203   | 5            |
| 980   | 4            |
| 4308  | 3            |
| 1428  | 2            |
| 105   | 1            |
+----------------------+
```

Lecture 7

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

Optimise for reading

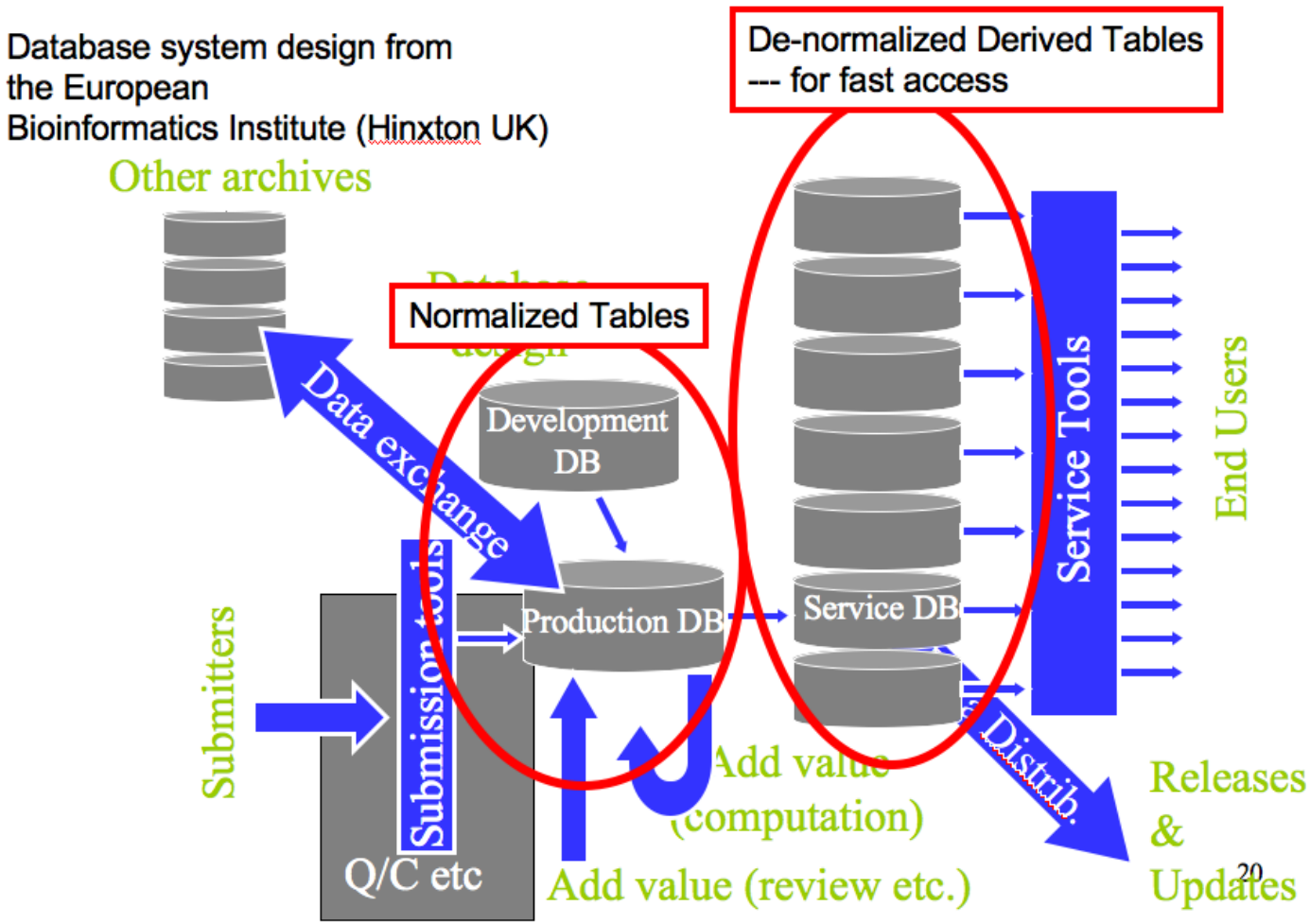
A fundamental tradeoff

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

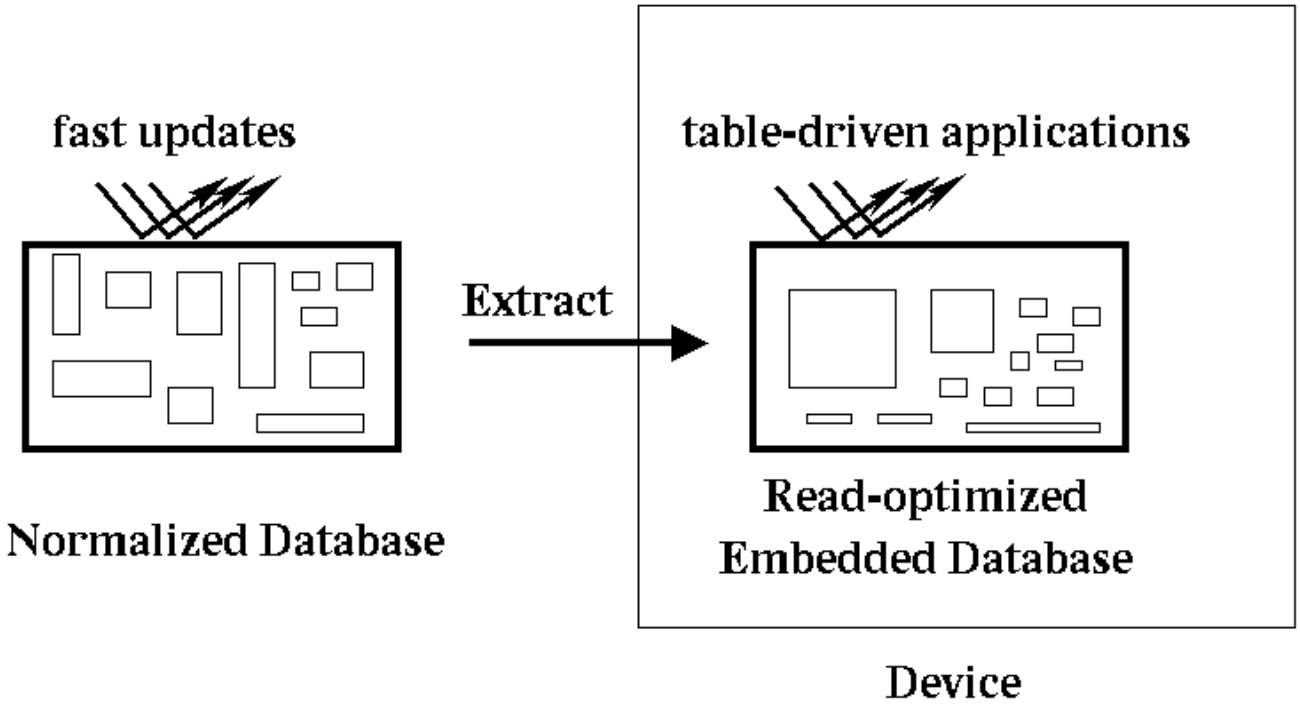
Situations where we might want a read-oriented database

- 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.

Example : Hinxton Bio-informatics



Example : Embedded databases



FIDO = Fetch Intensive Data Organization

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

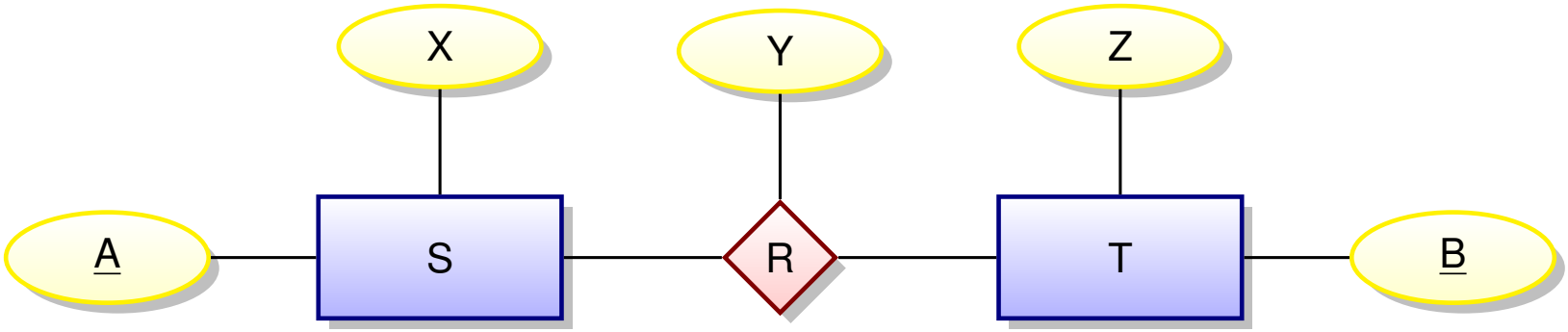
Our working definition

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!

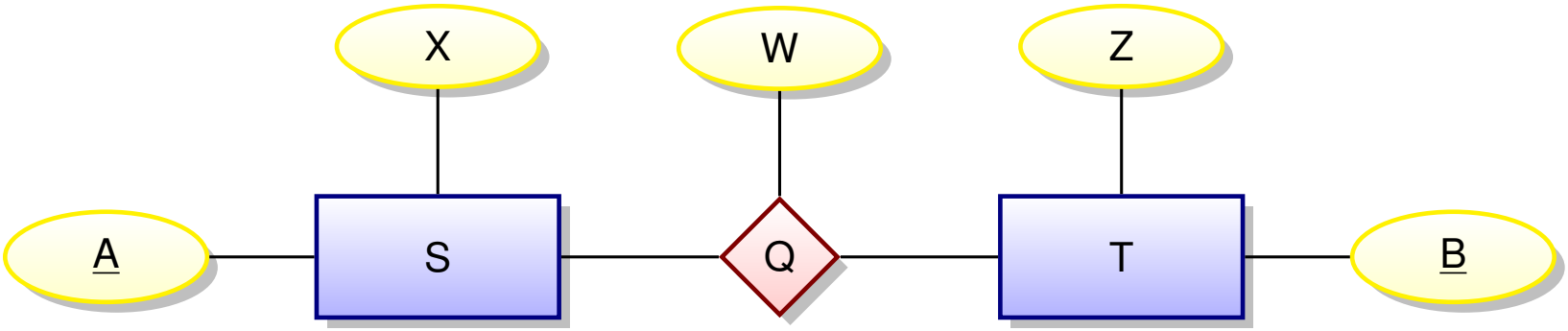
Implement the relationship as one big table?

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}] }
```

IMDb Person example


```
{ "id": 402542, "name": "Coen, Joel", "gender": "male",
  "director_in": [
    { "movie_id": 3256273, "title": "No Country for Old Men (2007)"},
    { "movie_id": 3667358, "title": "True Grit (2010)" },
    { "movie_id": 3026002, "title": "Inside Llewyn Davis (2013)"}],
  "editor_in": [
    { "movie_id": 3256273, "title": "No Country for Old Men (2007)"},
    { "movie_id": 3667358, "title": "True Grit (2010)" },
    { "movie_id": 3026002, "title": "Inside Llewyn Davis (2013)"}],
  "producer_in": [
    { "movie_id": 3256273, "title": "No Country for Old Men (2007)"},
    { "movie_id": 3667358, "title": "True Grit (2010)"},
    { "movie_id": 3026002, "title": "Inside Llewyn Davis (2013)"}],

  "writer_in": [
    { "line_order": 1, "group_order": 1, "subgroup_order": 1,
      "movie_id": 3256273, "title": "No Country for Old Men (2007)",
      "note": "(screenplay)" },
    { "line_order": 1, "group_order": 1, "subgroup_order": 1,
      "movie_id": 3667358, "title": "True Grit (2010)",
      "note": "(screenplay)"
    },
    { "line_order": 1, "group_order": 1, "subgroup_order": 1,
      "movie_id": 3026002, "title": "Inside Llewyn Davis (2013)",
      "note": "(written by)"
    }]
}
```

IMDb Movie example (greatly simplified)

```
{
  "title": "Mad Max: Fury Road (2015)",
  "year": 2015,
  "actors": [
    {
      "character": "Max Rockatansky",
      "name": "Hardy, Tom (I)"
    },
    {
      "character": "Imperator Furiosa",
      "name": "Theron, Charlize"
    }
  ],
  "directors": [
    { "name": "Miller, George (II)" }
  ]
}
```

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.

This describes what might be called a **pure** key-value store. Some key-value stores extend this architecture with some limited capabilities to inspect blocks of data and extract meta-data such as indices. This is the case with **Berkeley DB** used to implement our bespoke data store **DoctorWho**.

How do we query DoctorWho?

We write code!

```
import uk.ac.cam.cl.databases.moviedb.MovieDB;
import uk.ac.cam.cl.databases.moviedb.model.*;

public class GetMovieById {
    public static void main(String[] args) {
        try (MovieDB database = MovieDB.open(args[0])) {
            int id = Integer.parseInt(args[1]);
            Movie movie = database.getMovieById(id);
            System.out.println(movie);
        }
    }
}
```

This code takes two command-line arguments: the directory containing the database and a movie id. It prints the associated JSON object.

Lecture 8

- **On**Line **A**nalytical **P**rocessing (**OLAP**)
- **On**Line **T**ransaction **P**rocessing (**OLTP**)
- Data Warehouse
- Multidimensional data — The Data Cube
- Star Schema

Yet another class of read-oriented databases

OLAP vs. OLTP

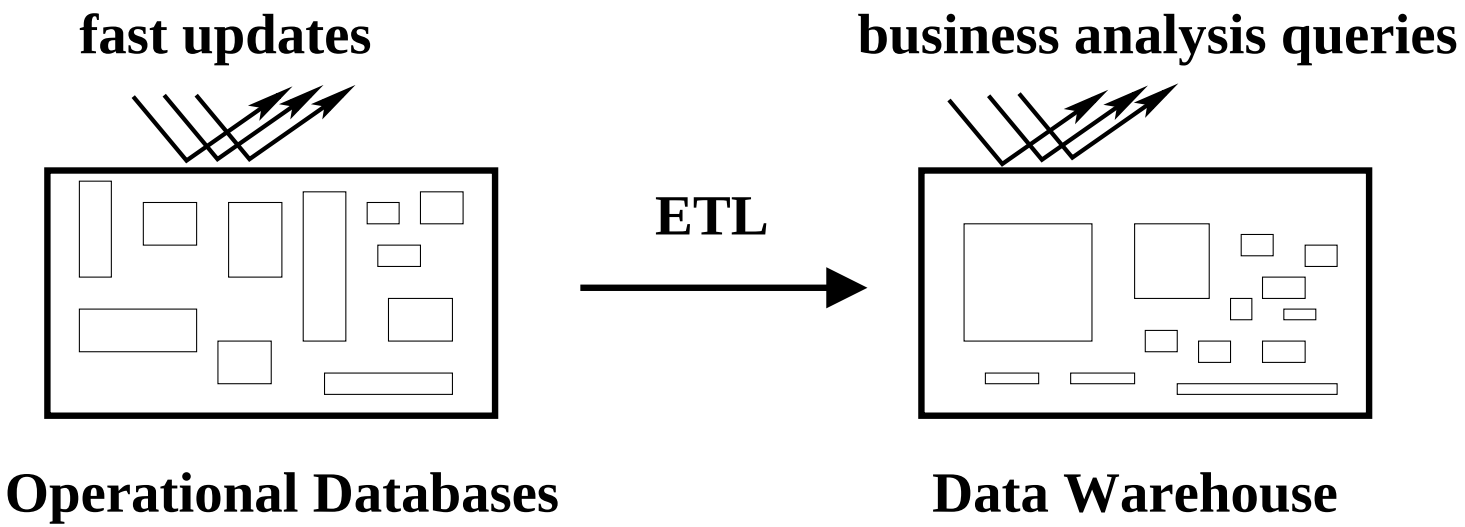
OLTP Online Transaction Processing

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 107!
- But there must be differences that are not illustrated.
- What are these differences?
- This will be the basis of one tripos question.

Limits of SQL aggregation

sale	prodId	storeId	amt
	p1	c1	12
	p2	c1	11
	p1	c3	50
	p2	c2	8

	c1	c2	c3
p1	12		50
p2	11	8	

- Flat tables are great for processing, but hard for people to read and understand.
- Pivot tables and cross tabulations (spreadsheet terminology) are very useful for presenting data in ways that people can understand.
- Note that some table **values** become column or row **names**!
- Standard SQL does not handle pivot tables and cross tabulations well.

A very influential paper [G+1997]

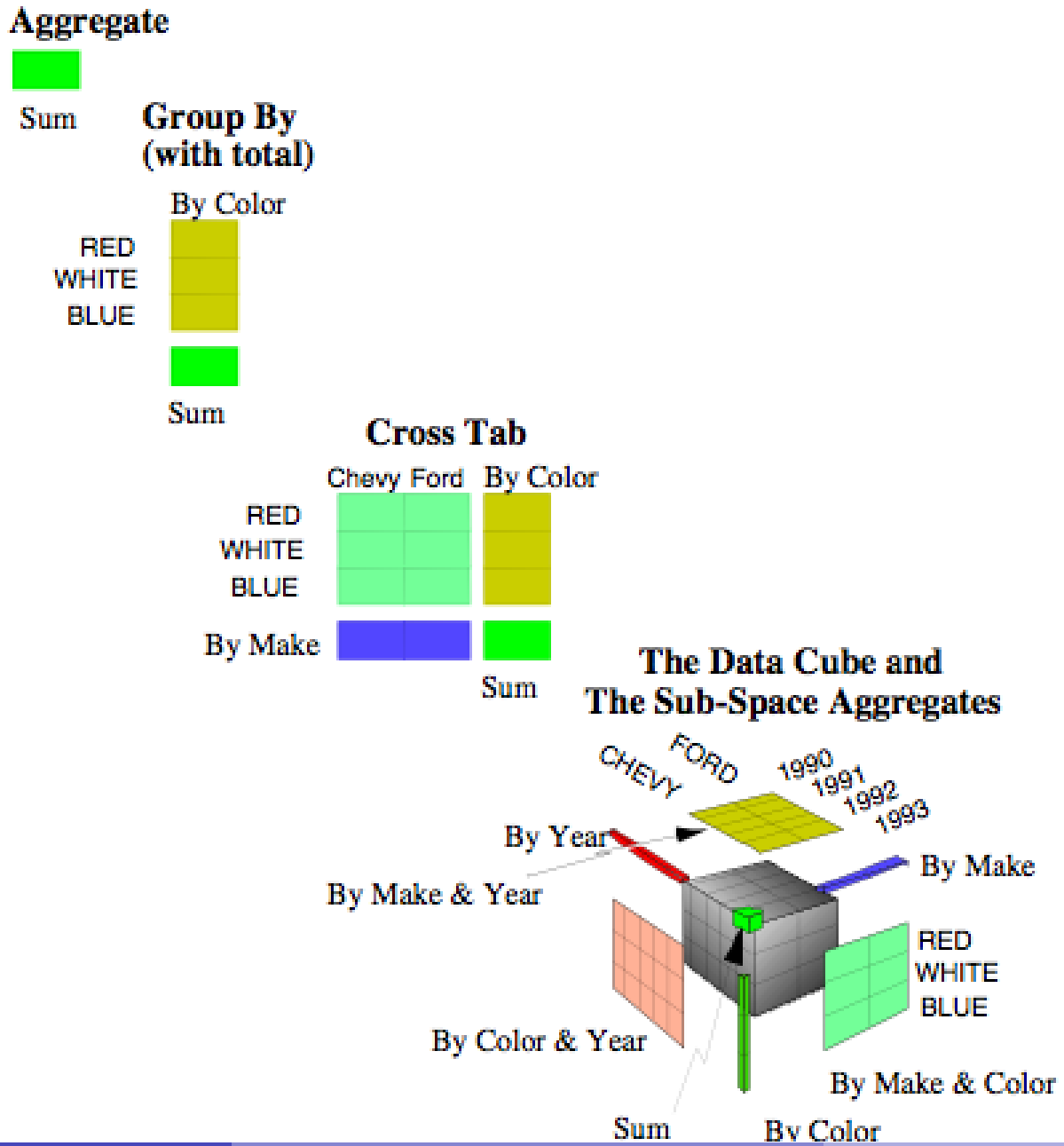
Data Mining and Knowledge Discovery 1, 29–53 (1997)

Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals*

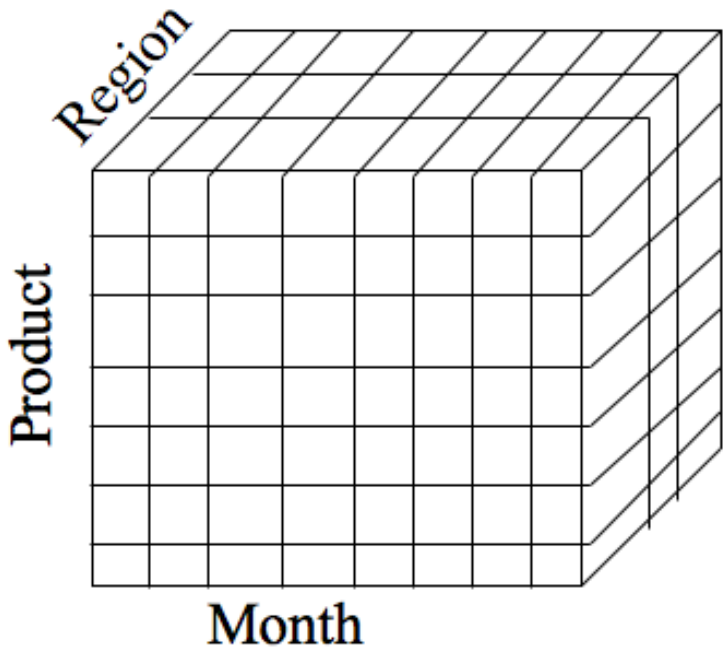
JIM GRAY Gray@Microsoft.com
SURAJIT CHAUDHURI SurajitC@Microsoft.com
ADAM BOSWORTH AdamB@Microsoft.com
ANDREW LAYMAN AndrewL@Microsoft.com
DON REICHART DonRei@Microsoft.com
MURALI VENKATRAO MuraliV@Microsoft.com
Microsoft Research, Advanced Technology Division, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052

FRANK PELLOW Pellow@vnet.IBM.com
HAMID PIRAHESH Pirahesh@Almaden.IBM.com
IBM Research, 500 Harry Road, San Jose, CA 95120

From aggregates to data cubes



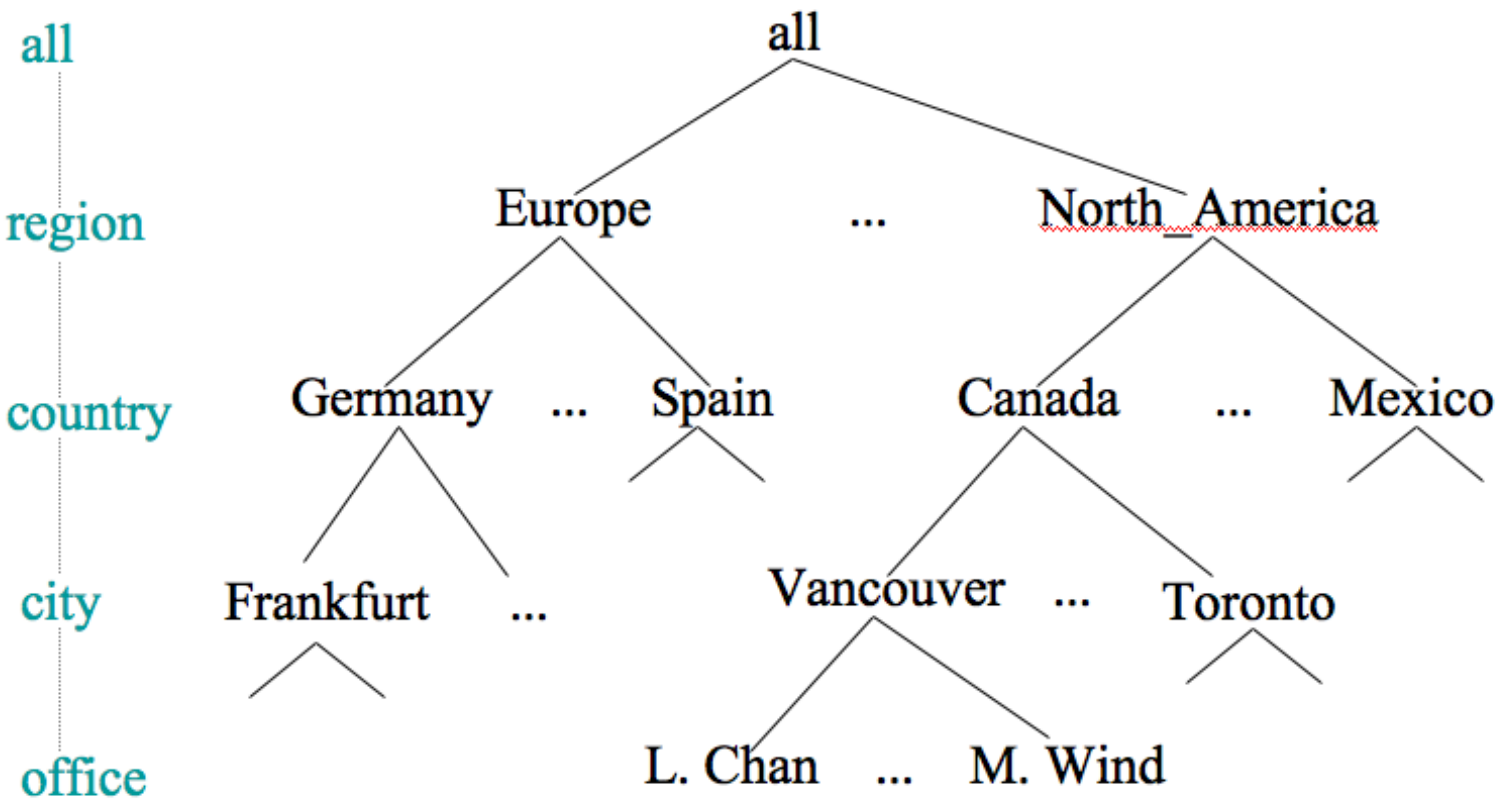
The Data Cube



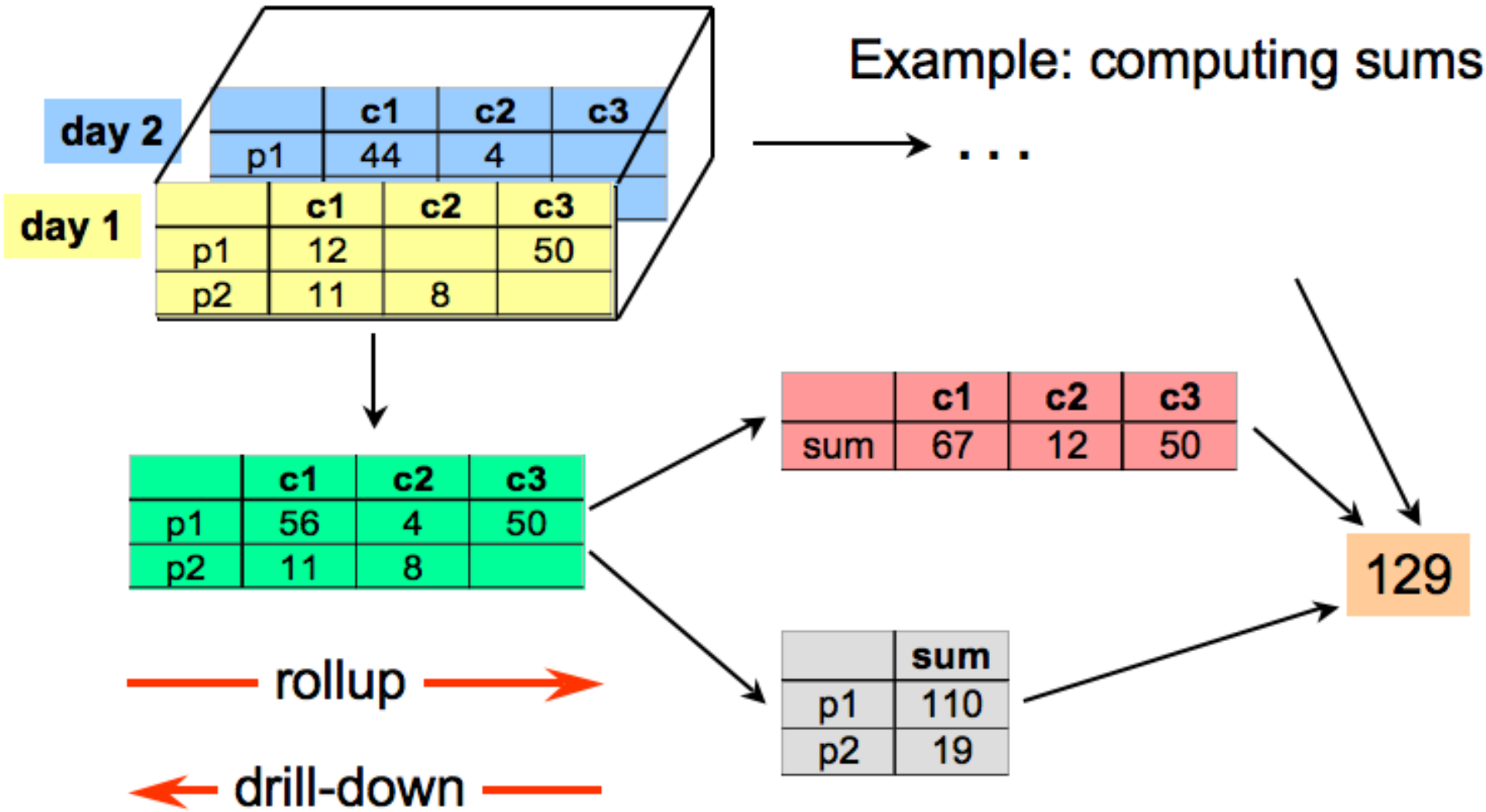
Dimensions:
Product,
Location,
Time

- Data modeled as an *n*-dimensional (hyper-) cube
- Each dimension is associated with a hierarchy
- Each “point” records facts
- Aggregation and cross-tabulation possible along all dimensions

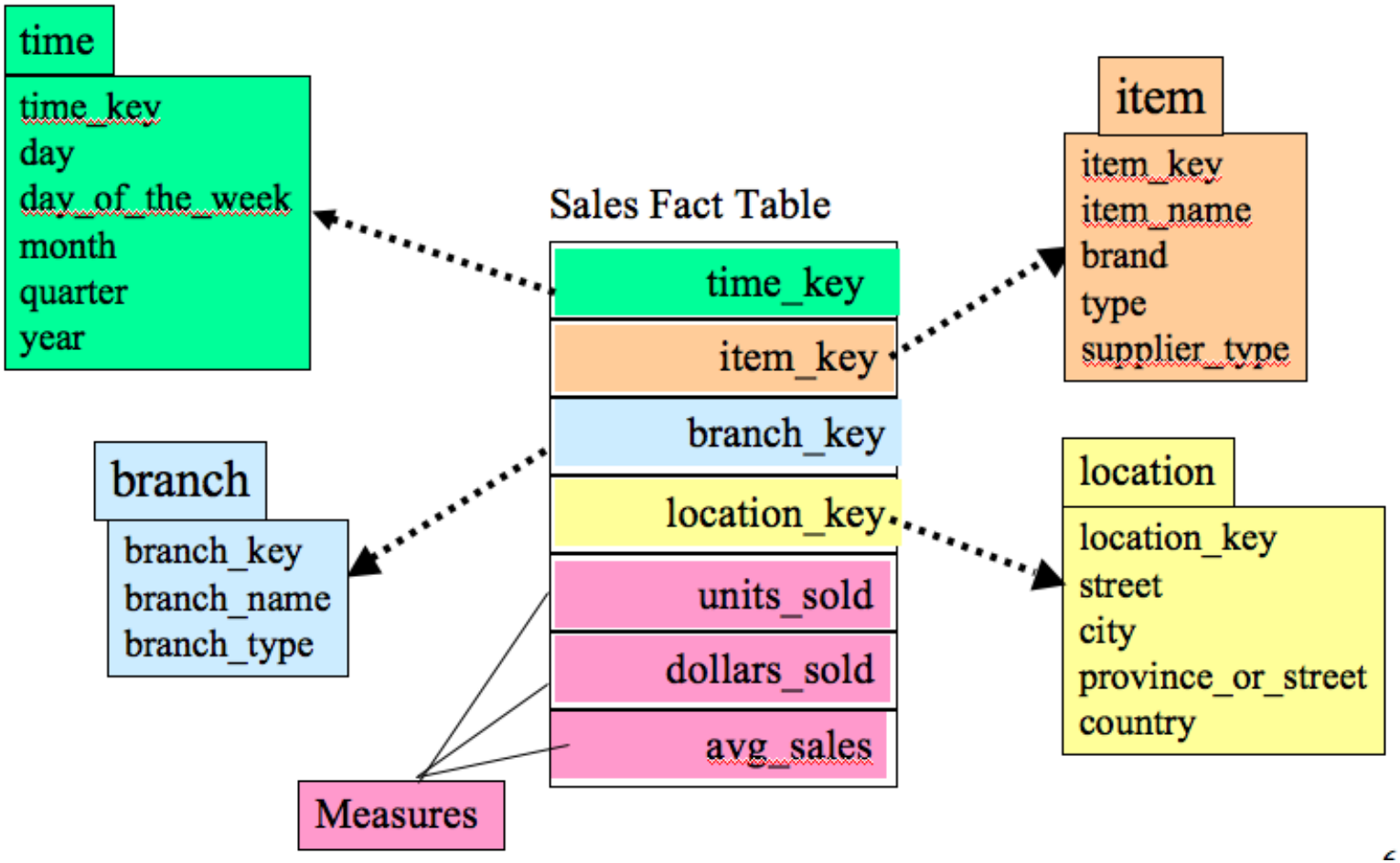
Hierarchy for **Location** Dimension



Cube Operations

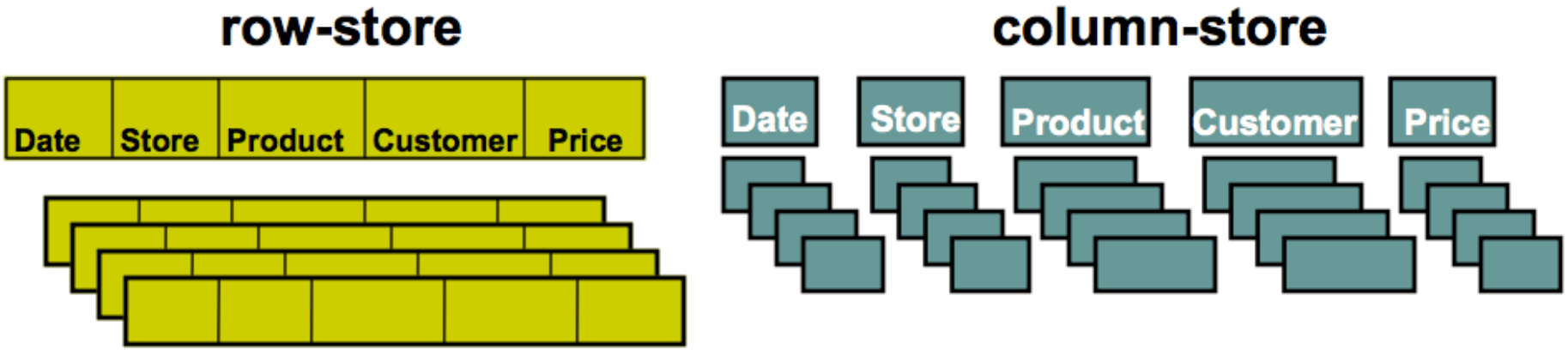


The Star Schema as a design tool



- In practice fact tables can be very large with hundreds of columns.
- Row-oriented table stores can be very inefficient since a typical query is concerned with only a few columns.

Column-oriented implementations



+ easy to add/modify a record

- might read in unnecessary data

+ only need to read in relevant data

- tuple writes require multiple accesses

=> *suitable for read-mostly, read-intensive, large data repositories*

From VLDB 2009 Tutorial: Column-Oriented Database Systems, by Stavros Harizopoulos, Daniel Abadi, Peter Boncz.

Demo Time

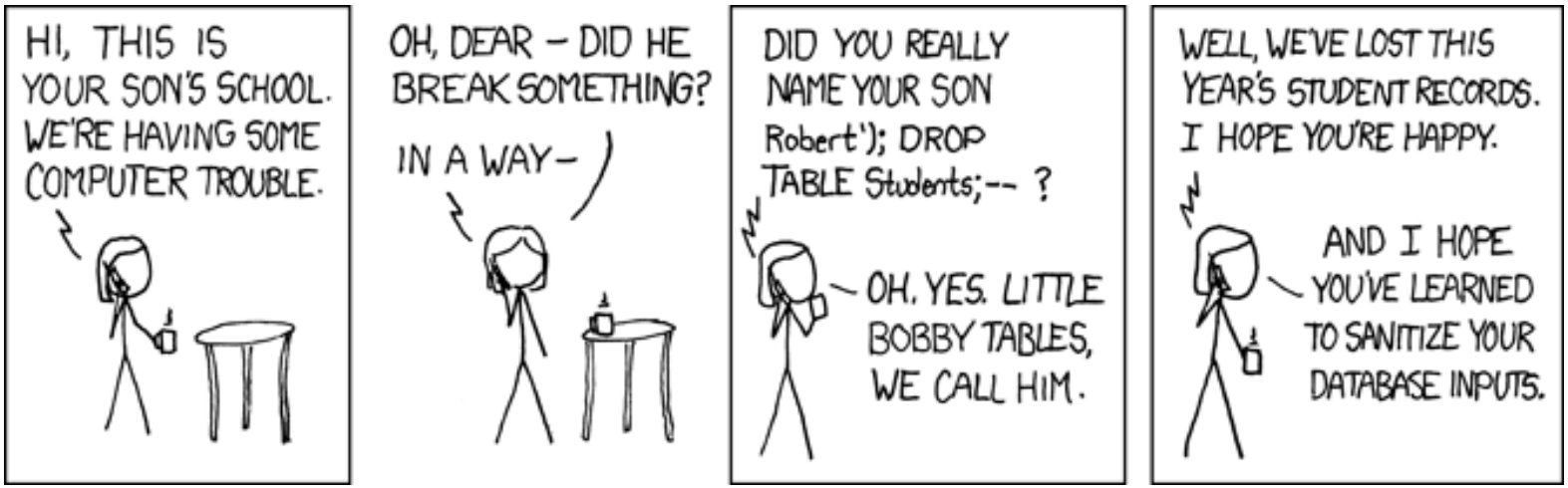
We will look at a small 2-dimensional data cube derived from the movie release dates of our relational database. One dimension will be the date (day $\in$ month $\in$ year) and the other location (country $\in$ region $\in$ area). The fact table will contain a single fact: the number of movie releases on a given date in a given country. The SQL for the example will be provided on the course website.

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)