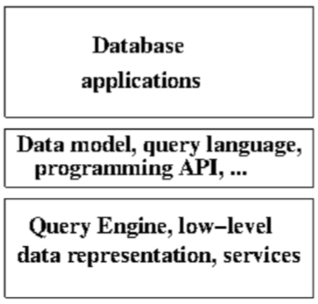
Databases

Introduction

Abstraction of data management: In an ideal world, you should be able to alter the implementation of how the data which is stored is to be used through the idea of abstraction. How the data stored is a form of abstraction that we should be able to ignore when writing the use of the data.

Database Management System (DBMS): A query engine knows about the low-level details hidden by the interface, using this knowledge to optimise the query evaluation. A DBMS first and foremost offers **persistent storage** while offering (1) CRUD operations and (2) ACID transactions.


Database applications
Data model, query language, programming API, ...
Query Engine, low–level data representation, services

N.B. For the purposes of the course, the application APIs and the networking APIs will be ignored.

CRUD:
1. Create
2. Read
3. Update
4. Delete

ACID
1. Atomicity
    a. Either all actions of a transaction are carried out, or none are (even if the system crashes in the middle of a transaction).
2. Consistency
    a. Every transaction applied to a consistent database leaves it in a consistent state.
3. Isolated
    a. Transactions are isolated from the effects of other concurrently executed transactions.
4. Durability
    a. If a transaction completes successfully, then its effects persist.

Different Data Models
1. Relational Model: data is stored in tables (SQL is the main query language).
2. Graph-oriented Model: Data is stored as a graph (nodes and edges), **The Query languages tend to have path-oriented features.**

3. Aggregate-oriented Model: Optimised for reads – they are also called document oriented database.

Why multiple options for DBMS:

1. No one system nicely solves all data problems
2. Several fundamental trade-offs faced by application designers and system designers.
    a. One important trade-off involves redundant data
    b. **Data in a database is redundant if it can be deleted and then reconstructed from the data remaining in the database.**
3. Database engine might be optimised for a particular class of queries
4. Query language might be tailored to the same class.

Data Redundancy (Query Response vs Update Throughout):
Data redundancy is problematic for some applications – if a database supports many concurrent updates, then data redundancy leads to many problems. If a database has little redundancy, then update throughout is typically better since transactions need to only lock a few data items.
However, in a low redundancy database, evaluation of complex queries can be very slow and require large amounts of computing. By precomputing, answering common queries (either fully or partially) can greatly speed up the queries. This introduces redundancy, but it may be appropriate for databases that are read-intensive, with few data modifications.

Source of data for the course (IMDB):

- Data taken is a subset of the information available from IMDb – taken from the top 10 movies from 2006 through 2017.

Relational Databases

ER Diagrams

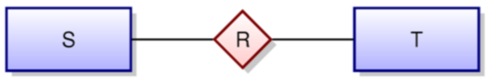
Implementation Independent technique to describe the data that we store in a database. ER was devised by Peter Chen. The technique grew up around relational databases systems but it can help document and clarify design issues for any data model.

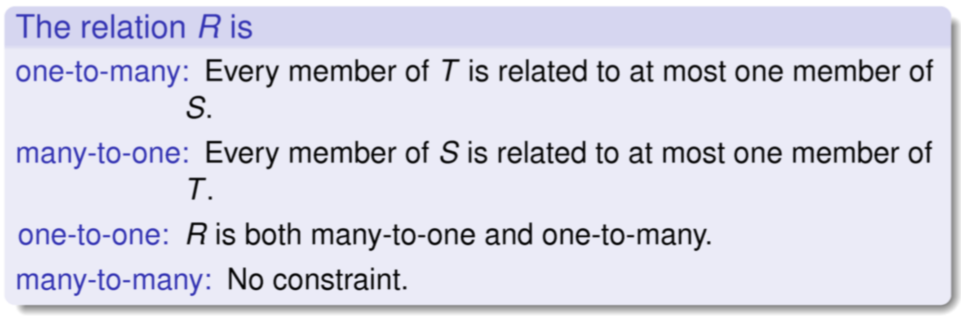
- **Entities (Represented using a Square):** represents the items of our model
- **Attributes (Ovals):** represents properties
- **Key (Underlined):** Key is an attribute whose value uniquely identifies an entity instance.
    - They are often automatically generated to be unique. They might be formed from some algorithm – it is generally a better idea to generate key rather than using things which are out of 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.
- The **scope** of the model is limited – among the vast number of possible attributes that could be associated with an **attribute, we are declaring that our model is concerned with a specific amount of them.**
- **Relationships (Diamonds):** represents the connections between entities – 'the verbs'
    - Relationships can have attributes indicating information about the relationship.

ER Diagrams are very abstract and are independent of implementation.

Weak Entities
Weak entities are entities which depend on the existence of another entity. The relationship which connects the two entity is known as the **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.

Relational DBMSs:

- In the 1970s, in writing a database application, you needed to know a lot about the data's low-level representation.
- Codd instated the system of giving the users a model of data and a language for manipulating that data which is independent of the details of the representation / implementation. The model is based on mathematical relations.

Mathematical Relations

Cartesian Product:
Suppose S and T are sets. The Cartesian product, S x T is:
$S \times T = \{(s,t) : s \in S, t \in T\}$

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

- S and T are referred to as domains.
- We are interested in finite relations R that can be stored.

n-ary relation R is a set:
$R \subseteq S_1 \times S_2 \times \ldots \times S_n = \{(s_1, s_2, \ldots, s_n) : s_i \in S_i\}$

Instead of a set of tuples, we use a table, with column names and record which is effectively a set of items consisting of a value for each column (attribute)

Mathematical vs Database Relation
Named Columns: A name (attribute name) is associated with each domain. Instead of tuples, records are used (set of pairs consisting an attribute name with a value in each domain)

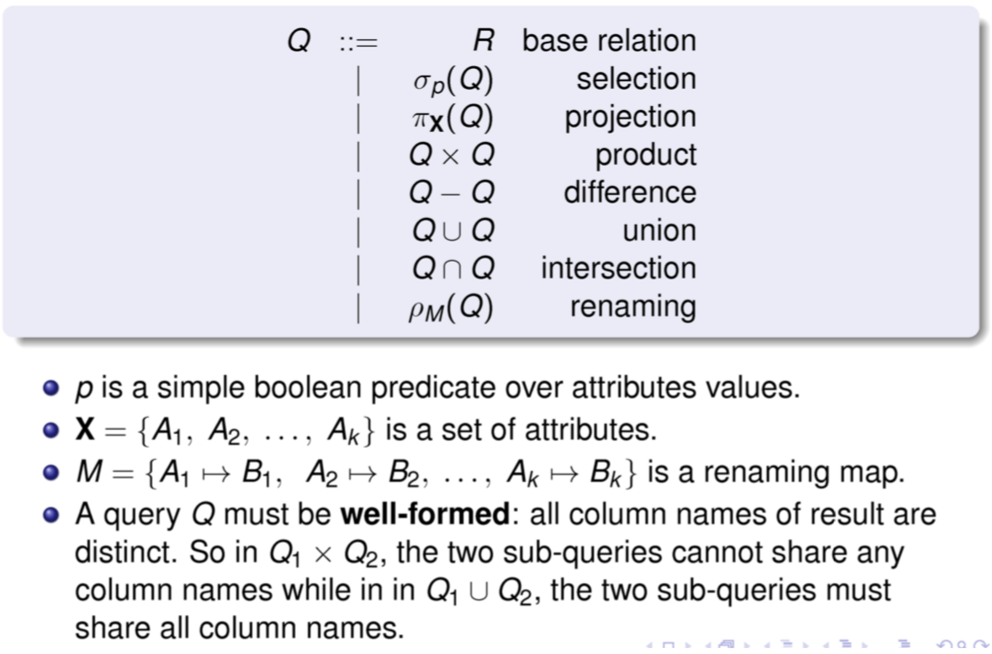
Column Order: Does not matter for database relations. The **schema** of a database relation is defined as R(A1 : S1, A2 : S2, … An : Sn)

Relational Database Query Language

It effectively acts as a function which takes a collection of relation instances and returns a single relation instance. In order to meet Codd's goals, we want a query language that is high level and is independent of physical data representation. A couple of possibilities are relational algebra and SQL.

Relational Algebra and SQL

Relational Algebra Reference



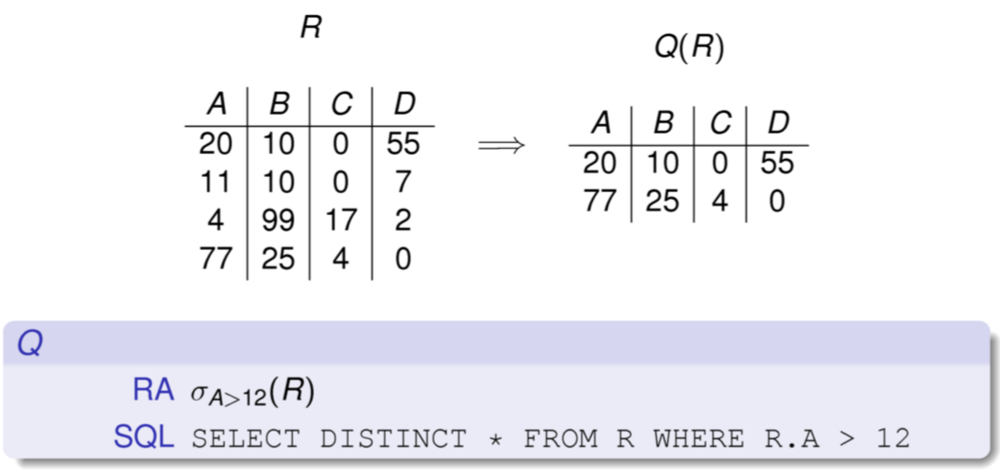
$$\begin{array}{rcrr} Q & ::= & R & \text{base relation} \\ & \mid & \sigma_p(Q) & \text{selection} \\ & \mid & \pi_{\mathbf{X}}(Q) & \text{projection} \\ & \mid & Q \times Q & \text{product} \\ & \mid & Q - Q & \text{difference} \\ & \mid & Q \cup Q & \text{union} \\ & \mid & Q \cap Q & \text{intersection}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

- 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 intro:

- Made up of multiple sub-languages, including Query Language, Data Definition Language and System Administration Language
- It is an evolving language and has changed consistently over multiple stages over standardization.
- (Has its origins at IBM in the early 1970s)

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`

The use of the 'DISTINCT' keyword is important and the reason for it will be discussed later.

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

$\Longrightarrow$

$Q(R,\ S)$

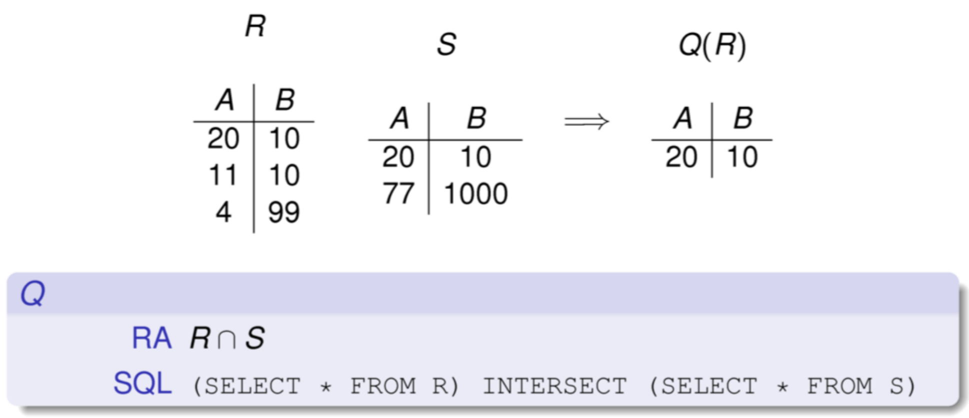
A	B
20	10
11	10
4	99
77	1000

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

$\Longrightarrow$

$Q(R)$

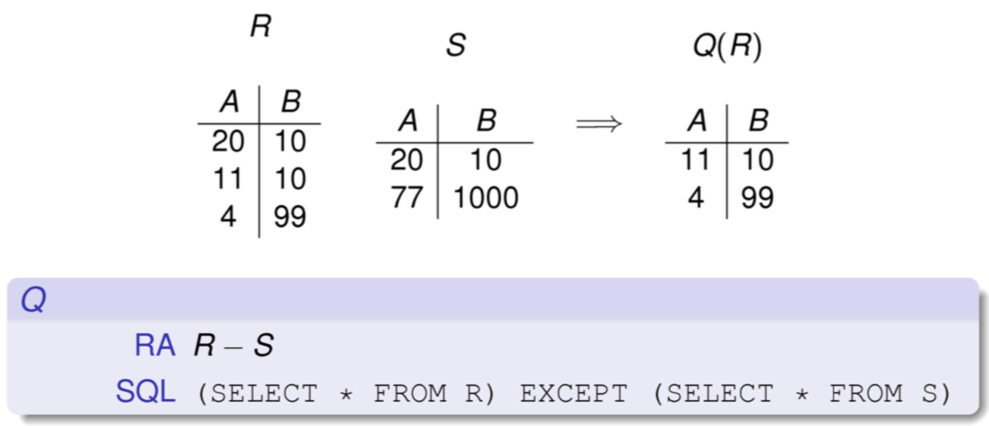
A	B
20	10

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

$\Longrightarrow$

$Q(R)$

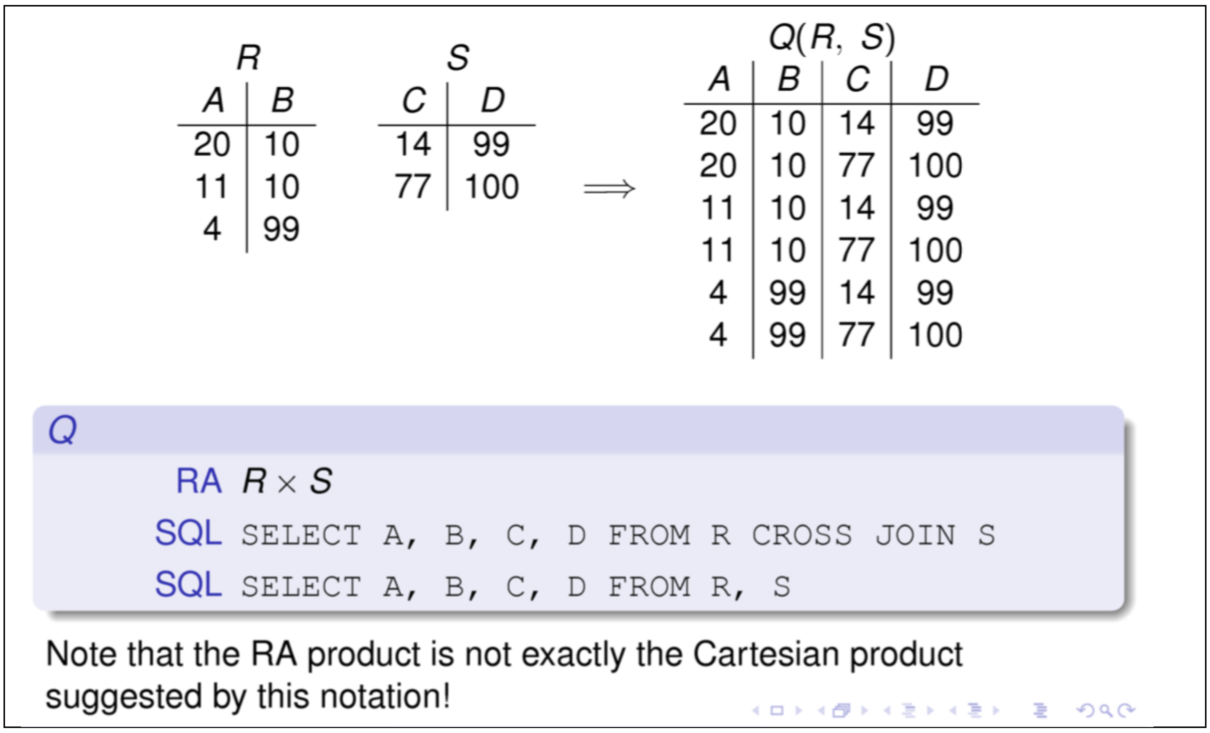
A	B
11	10
4	99

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!

Joins

Natural Join

Notation

- Often, the domain types are ignored and the relational schema is written as R(A) where A is a set of attribute names.
- When R(A, B) is written **you implicitly mean R(A U B).**
- You can do equality on an attribute, eg:
  - u.[A] = v.[A] is equivalent (abbreviates) u.A1 = v.A1 ^ u.A2 = v.A2 … u.An = v.An

Natural Join for R(a, b) and R(b, c) is defined as:

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

In relational algebra, it is written as:

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

Implementation

Since the Entity Relationship does not dictate the implementation, there are a normally a number of options in implementing the model. Normally, each comes with its own drawbacks, so we must weigh them up.

For example, we could simply put all information into one table, however, this has a lot of problems with data redundancy since there is a lot of data which must appear in multiple places.

Anomalies caused by data redundancy:

1. Insertion Anomalies
   a. How can we tell if a newly inserted record is consistent with all other records for that thing? Also, we might want to insert a record without needing to give all the

information, In the case of the movie table, one may want to define a table without defining the director of the movie.

2. Deletion Anomalies
    a. We will wipe out information about sub-entities when the last record which has this contained inside which we might not want to remove.
3. Update Anomalies
    a. All records for the same thing may differ due to things like typos, thus creating a problem.
4. **Concurrency Issues**
    a. **A transaction implementing a conceptually simple update but containing checks may end up locking the entire table.**

Therefore, it is much better to split the database up into a number of different tables each of which you work on.

Join in SQL

- Use a join … on … structure
- Eg:

```
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
```

- **COMPLEXITY**
    - In the worst case, using a method like this to do the join, the function is O(mn) where m and n are the number of elements in each previous table.

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

Observations

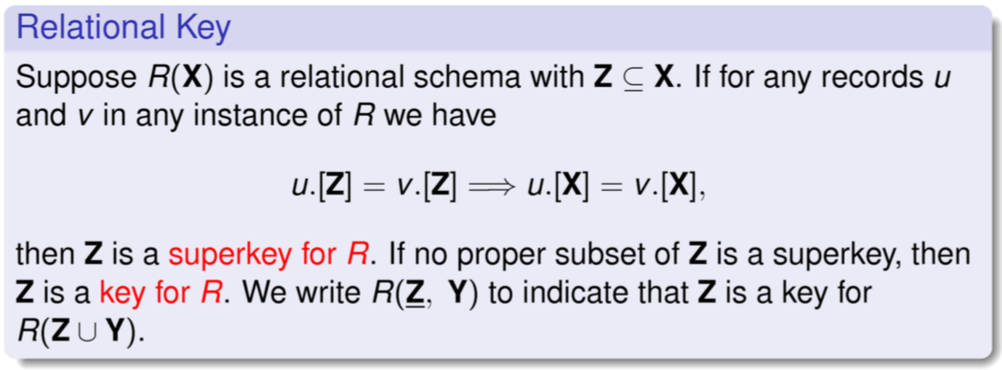
- Both ER entities and ER relationships are implemented as tables
- Good: We avoid many update anomalies
- Bad: We have to work hard to combine information in tables (joins) to produce effective queries.

Ensuring consistency of a Relational Database

In order to ensure that a table representing a relation actually implements a relation using keys and **foreign keys**

Keys

Superkeys



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

Foreign Keys and Referential Integrity

If we have a superkey or key from a different table in another table (normally representing a relationship), it is a foreign counter in that table. A database is said to have **referential integrity** when all foreign key constraints are satisfied.

Operation of Primary Keys and Foreign Keys

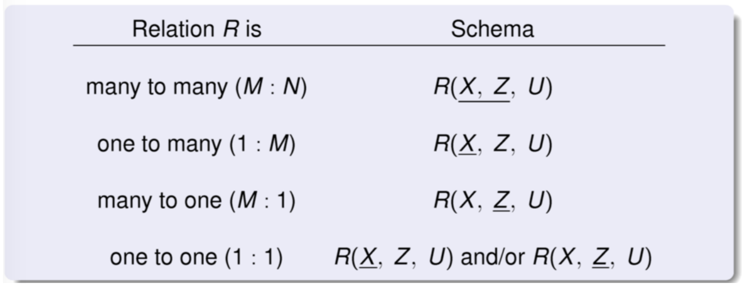
```
create table ActsIn (
   movie_id int not NULL,
   person_id int not NULL,
   character varchar(255),
   position  integer,

   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))
```

primary key automatically contains the constraint of ensuring that it is unique.
The foreign key constraint effectively checks that It actually references an item in the defined table.

Further Implementation

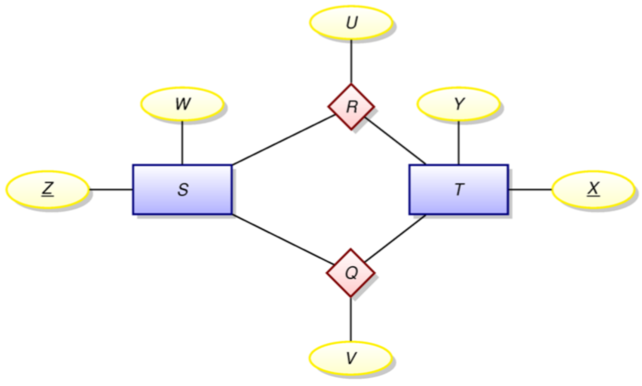
Design of Relationships ('clean' approach)



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

However, items can differ from the clean approach, for example the ability of NULLs means that you could entirely forego a relationship instead simply filling items which don't exist (didn't have a relationship) to have a NULL. This is generally how we would implement **weak entities (since that means it is definitely a 1-1 relationship).**

Implementing multiple relationships in 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 $(x, z, \mathbf{q}, NULL, v)$

While this would work, it definitely introduces new redundancy as given the value of the type column, you can get the value of either the U or V column.

Database Index: An index is a data structure – created and maintained within a database system that can greatly reduce the time required to locate records. However, there are many types of database index (it is not defined in the SQL standards). It will also take more time to complete updates (even if it is faster to read).

A database index can be created and deleted thus:

```
CREATE INDEX index_name on S(B)

DROP INDEX index_name
```

Multisets
SQL in fact creates a multiset not a set (that is multiple rows can have the same values as this is very important as data can repeat). This means that if you want to only select the set equivalent (therefore removing the no distinct elements, you need to use the work 'distinct'.

Using Duplicates
Duplicates are important for aggregate functions (min, max, average, count) and can easily be used by utilising the 'GROUP BY' command. The Group By splits the thing into separate tables for each group before being reassembled when using an aggregate function.

3 Valued Logic
SQL has NULLs – it is a placeholder. It is not a member of any domain (therefore x <> 10 will return a 'false' for NULL). This also means we need three-values logic:

$\wedge$	T	F	⊥
T	T	F	⊥
F	F	F	F
⊥	⊥	F	⊥

$\vee$	T	F	⊥
T	T	T	T
F	T	F	⊥
⊥	T	⊥	⊥

v	$\neg v$
T	F
F	T
⊥	⊥

it is important to note that NULL is ambiguous:

1. Value exists but we don't know it
2. No value is applicable
3. The value is known, but you aren't allowed to see it.
4. **IS NULL can be used to imply NULL = NULL (GROUPING BY NULL) but NULL = NULL is NULL.**

Partial Functions as Relations

- A partial function $f \in S \rightarrow T$ can be thought of as a binary relation where $(s,t) \in f$ iff t = f(s)
- Suppose R is a relation where if (s,t1) is a member of R and (s, t2) is also, then it follows that t1 = t2. Here, R represents a (partial) function.
- Given partial functions $f \in S \rightarrow T \text{ and } g \in T \rightarrow U \text{ then } (f \circ g)(s) = g(f(s)) = g \circ f \in S \rightarrow U$
- Since $Q \circ R \equiv R \bowtie_{2=1} Q$ we can see that **joins are a generalisation of function composition.**

Views: By defining the table as a result of a query, this can now be used in later queries as if it was a table. Eg:

```
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';
```

Now cofactors(pid1 pid2) can be used as if it is a table.

Graph Databases

Definitions

G = (V, A) is a directed graph where:

V is a finite set of vertices (nodes)
A is a binary relation over V **that is $\boldsymbol{A} \subseteq \boldsymbol{V} \times \boldsymbol{V}$**

If $(u, v) \in A$ then we also have an arc from u to v.
The arc (u, v) is also called a directed edge, or a relationship of u to v.

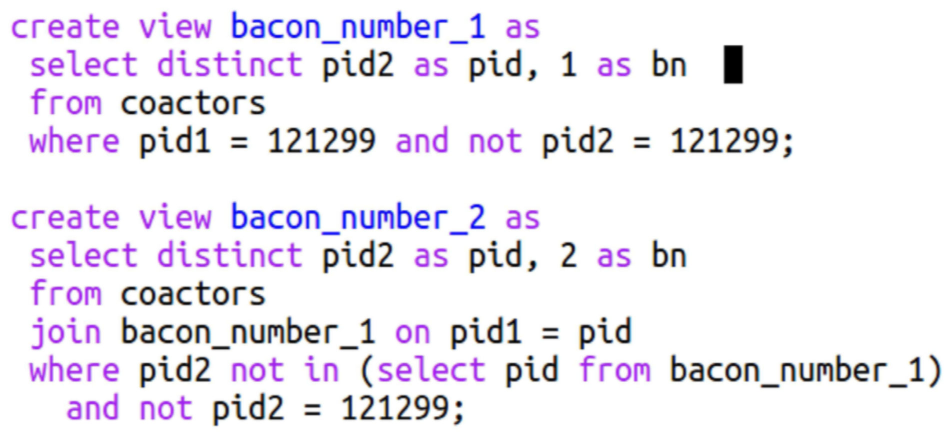
Composition: $A \circ A$ consists of the nodes which can be reached in a path of length 2

Iterated Composition (paths)
Iterated Composition is defined as R^1 = R and $R^{n+1} = R \circ R^n$
A path in a graph where v1, v2, … , vk is the sequence of vertices, we can write the path as v1 -> v2 -> v3 etc
N.B. If G = (V, A) is a directed graph, and (u, v) are 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
The R-distance (hop count): Suppose s0 is a member of the vertices of R, such that there is a pair (s0, s1) which is a member of R:

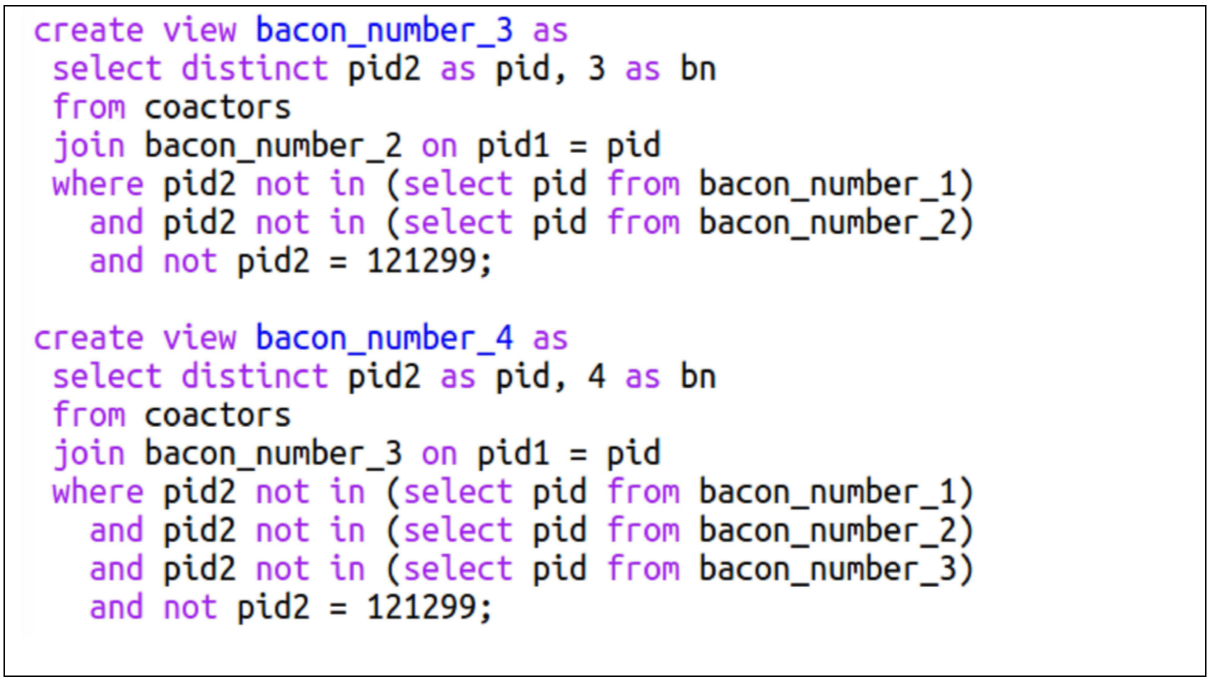
- The distance from s0 to s0 is 0
- If (s0, s1) is a member of R then the distance from s0 to s1 is 1.
- For any other s' which is a member of the vertices of R, the distance from s0 to s' is the least n, such that (s0, s') is a member of R^n
- **THE BACON NUMBER IS THE R-distance where s0 is Kevin Bacon**

BACON NUMBER in SQL



```
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;
```



```
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;
```

Transitive Closure

The transitive closure is the smallest binary such that the relation is a subset of the original relation and is transitive:

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

Therefore, has all paths, hence can be expressed as:

$$R^+ = \bigcup_{n \in \{1,2,3,\dots\}} R^n$$

Since our relations are finite, there must be some k with:

$$R^+ = R \cup R^2 \cup \dots \cup R^k$$

However, k will depend on the contents of R! and hence we **cannot** compute transitive closure using relational algebra.

Hence, the idea comes to design and implement a database system that is **optimised for transitive close and related path-oriented queries.**

Neo4j

- Contains nodes and binary relationships between nodes.
    - The 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.
- The general idea is you use pattern matching to match paths and return their values. An example of how to compute all Bacon Numbers is listed below but you **MUST** look at the introduction to Cypher and ticks to remember how to use Cypher.

Bacon Numbers

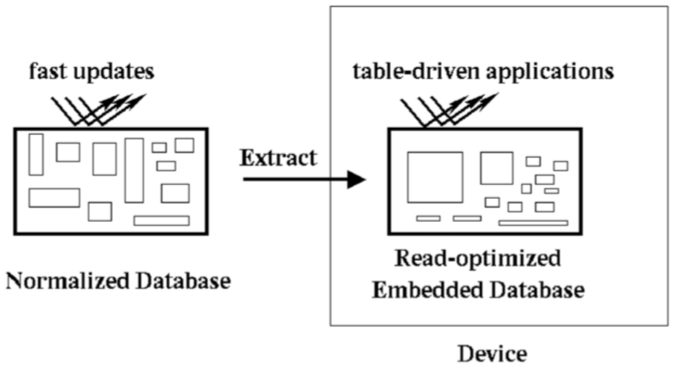
```
MATCH (p:Person)
where p.name <>  "Bacon, Kevin (I)"
with p
match paths=allshortestpaths(
      (m:Person {name : "Bacon, Kevin (I)"} )
         -[:ACTS_IN*]- (n:Person {name : p.name}))
return distinct p.name,
       length(paths)/2 as bacon_number
order by bacon_number desc;
```

Read-Optimised Databases

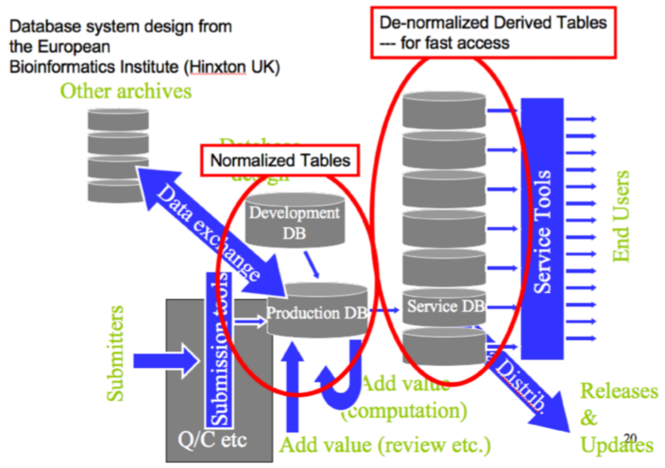
Fundamental Tradeoff: Introducing data redundancy can speed up read-oriented transactions at the expense of slowing down write-oriented transactions. Sometimes, however, the tradeoff is worth it, in order to have a fast reading capability:

1. If data is seldom updated, but read from very often.
2. Reads can afford to be out of sync with the write-oriented database. Then periodically extract read-oriented snapshots and storing them in a database optimised for reading.

(2)



This should be used in a fetch intensive data organisation, where information is required rapidly even if it is okay if the data is out of date. An example of where this is used is for Hinxton Bio-informatics, where there are normalized tables and de-normalised derived tables (which service the end users).



Semi-Structured Data – JSON

You know how it looks like, but it is separated by different tags with items (and the name of the columns stored) in quotation marks.

JSON Example

```
{"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

The data is stored and separated using HTML like start and end tags.

XML Example

```
<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

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

Items are semi-structured, in order to speed up the speed of retrieval of data from the database as well as reducing the amount of redundant data that needs to be stored. The use of semi-structured databases means that we are then able to find items using a key-value retrieval process.

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. The interpretation of such data is left to applications.

This describes a **pure** key-value store. Some key-value stores extend this architecture with some capabilities to inspect blocks of data and extract meta-data such as indices – this is true with BerkleyDB (using the DoctorWho bespoke data store).

DoctorWho
DoctorWho is written in Java – having a lot of functions which are written for the purpose. An example of a piece of code is provided below but again, you should read the introduction which is provided. Though, this is very unlikely to be tested on, since you would need a reference sheet and the code is specific to this storage of data.

DoctorWho Code Example

```
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);
        }
    }
}
```

Application
Sarah Mei
Previously, a relational database was the only type of database that was every used. Today, however, there is a rising sentiment of a NoSQL policy, which suggests that SQL (and the relational model) is not always the most effective type of database. Sarah Mei talked in particular about how it is ineffective in particular for a Social Networking platform.

It is apparently clear that the relational model is bad for social data, so the site chose to use a document database. This is attractive as all the data is located where you need it. However, this is also dangerous, since updating a user's data means going though all the streams where the data appears and updating it there. This is **error prone** and often leads to inconsistent data and mysterious errors (especially with deletion).

She says that if there is no value in the links between documents – the data is not necessarily tied together, then a document database is effective and would work perfectly. However, definitely not good if there is value in the link – especially not in a social network where the links are the most vital part of the entire system.

OLAP vs OLTP

OLTP: **Online Transaction Processing**
OLAP: **Online Analytical Processing**

	OLAP	OLTP
Supports	**analysis**	**Day-to-day operations**
Data is	**Historical**	**Current**
Transactions mostly	**Reads**	**Updates**
Optimised for	**reads**	**Updates**
Data redundancy	**High**	**Low**
Database size	**humongous**	**Large**

OLAP
OLAP is commonly associated with decision support and data warehousing. It generally involves looking at historical data (which won't get updated anymore) and getting information from it which could be used in the future.

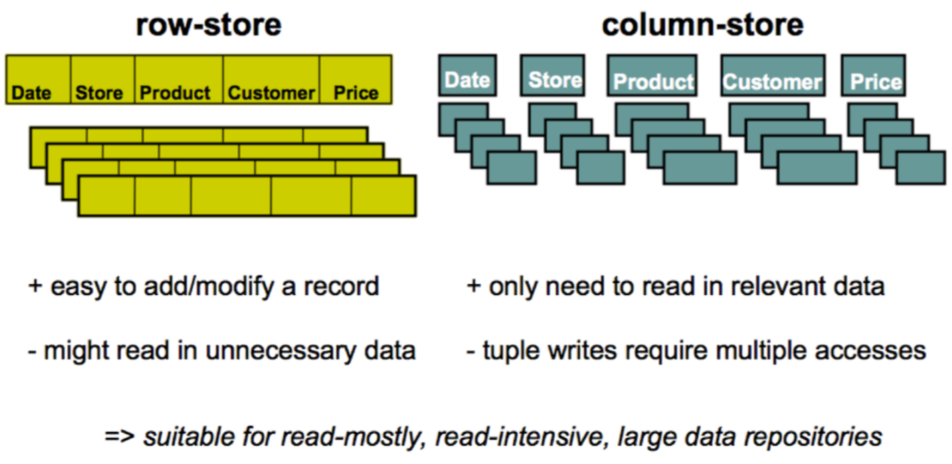
SQL is generally bad for people to understand – you want things like pivot tables and cross tabulation. Standard SQL does not handle these well. Therefore you want to consider other query languages or indeed a different database model – such as graph based databases.

Data Cube
In the data cube, data is modelled as a n-dimensional hypercube. Each dimension is associated with a column (a hierarchy). Each point on the cube records information and aggregation and cross-tabulation is possible along all dimensions.

You can roll up or drill down (going back) into any one dimension, in order to get more information about something in particular

Star Schema
In practise, 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 and needs to retrieve data from every record.

Therefore, completing a column-oriented implementation is often much faster, though it is much harder to modify an item (therefore utilised for OLAP).


row-store
Date
Store
Product
Customer
Price
column-store
Date
Store
Product
Customer
Price
+ 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

Distributed Databases

Why Distribute Data?

1. Scalability
   a. The dataset is too large for a single machine
2. Fault Tolerance
   a. In case one machine (or some) fail, the system can continue to function
3. Lower Latency
   a. Data can be located closer to all users, therefore meaning it would be much faster for them. Also, more servers to simply deal with requests will just mean that it would be faster for everyone.

How distribute data?

1. **Replication**
   a. Information is simply copied between every server (this would not work where the information is too large to be stored)
2. **Partition**
   a. Information is split between systems – therefore would not be faster for users in specific geographic locations – but would be more scalable.
   b. Partitions themselves are often replicated.

In reality, you would generally attempt to do the second (with replication of the partitions as well, meaning all the advantages of data distribution are received.

CAP Concepts

These are the things you attempt to gain in a distributed system, but are often impossible.

1. **CONSISTENCY**
   a. All reads return data that is up-to date.
2. **AVAILABILITY**
   a. All clients can find some replica of the data.
3. **PARTITION TOLERANCE**
   a. The system continues to operate despite arbitrary message loss or failure of part of the system

CAP Principle:

- Assume that network partitions and other connectivity problems will occur.

- Implementing transactional semantics is very difficult and slow
- It is always **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.