Introduction to Databases
Lectures 5 - 8

David J. Greaves

(with grateful thanks to Timothy G. Griffin)

Computer Laboratory
University of Cambridge, UK

Michaelmas Term, 2022-23

Lecture 5 - Transactions, Reliability, Throughput & Consistency.

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

Transaction Processing

A **transaction** on a database is a set of queries and changes that are logically atomic.

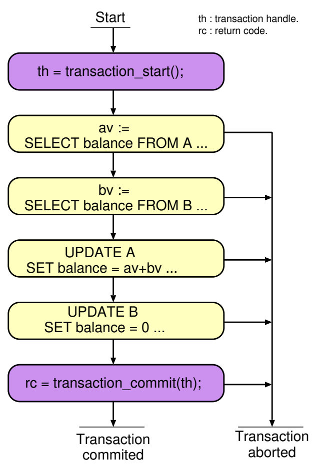
Internal transactions:

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

External 'transactions' (do not really exist):

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

Transaction client flow.



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

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

ACID transaction properties

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

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

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

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

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

ACID vs BASE

Your Ultimate Guide to the Non-Relational Universe!

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

BA: Basically Available,
S: Soft state,
E: Eventual consistency.

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

[Wikipedia: BASE].

Implementing ACID transactions requires locking data

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

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

What is redundant data? Is it bad?

Our definition:

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

Why is redundant data problematic?

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

Redundant data goody:

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

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

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

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

Least F/P iteration example: division.

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

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

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

Normal form representation.

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

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

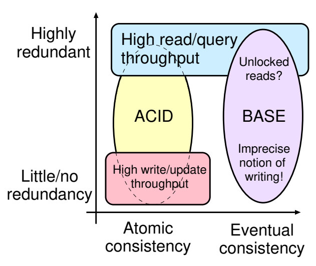
Normal form database schemas.

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

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

* This course does not cover 'textbook' database normal forms anymore.

Redundancy/Consistency/Throughput trade off.



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

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

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

eg. Suppose only one updater?]

Throughput: Why read-oriented databases?

A fundamental tradeoff

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

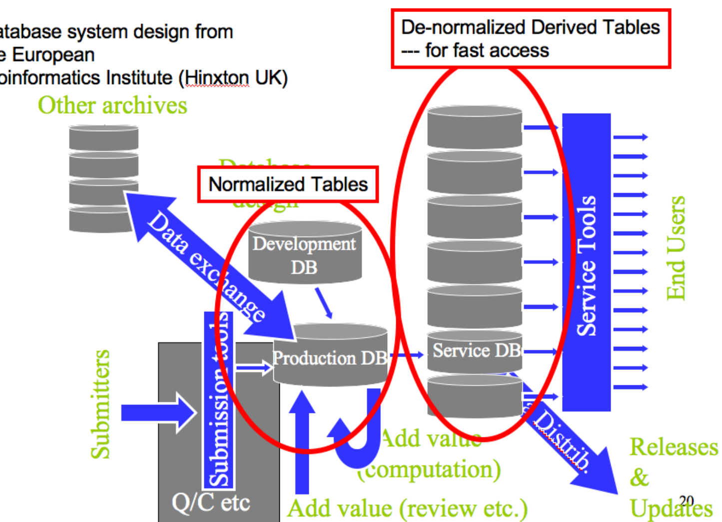
Something to ponder

How do database indexes demonstrate this point?

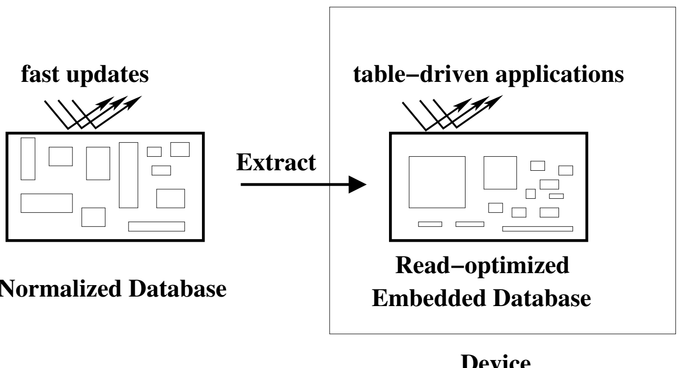
Situations where we might want a read-oriented database

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

Example : Hinxton Bio-informatics



Example: Embedded databases



An embedded database system is a database management system which is tightly integrated with an application software; it is embedded in the application — web: Wikipedia.
For instance: a different SELECT from the main staff table might be held in each electronic door lock.
FIDO = Fetch Intensive Data Organisation

OLAP vs. OLTP.

OLAP — Online Analytical Processing

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

OLTP — Online Transaction Processing

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

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

OLAP vs. OLTP (continued).

Processing power:

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

Update history:

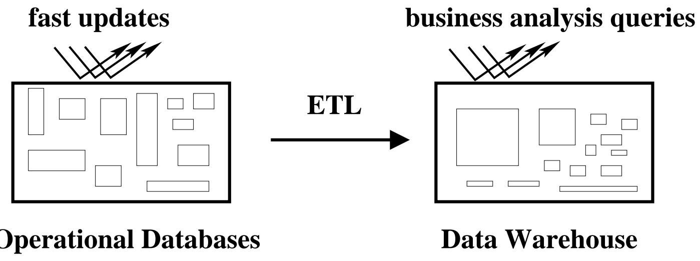
- An update to a relational database occludes the previous value of a field.
- A revision control system (*eg.* git) stores the update history — an additional **dimension** to the stored data/documents.
- Even for OLTP, an update history within a limited time horizon is always stored for ACID durability.

Further dimensions*:

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

* = no longer on the syllabus or examinable.

Example: Data Warehouse (Decision support)



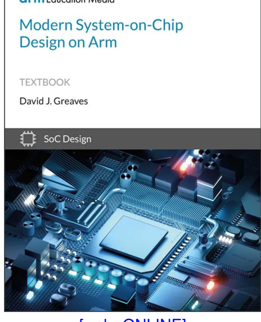
ETL = Extract, Transform, and Load

- [This looks very similar to slide 15!]
- Slide 15 stored date optimised for known *a priori* queries. Size would be an issue for embedded use.
- Here data is pre-processed in many/every conceivable way for visualisation and exploration by (typically) human agents.

Lecture 6 - Semi-structured Document Databases

- Semi-structured data.
- NoSQL movement.
- Document-oriented databases.
- Denormal and BASE possible advantages.
- An example database: DoctorWho .
- Path query languages and *ad hoc* HLL access.

Semi-structured data.



[web: ONLINE]

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

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

What can sensibly or usefully be stored in a database?

Two approaches

Either

Store in two parts:

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

or

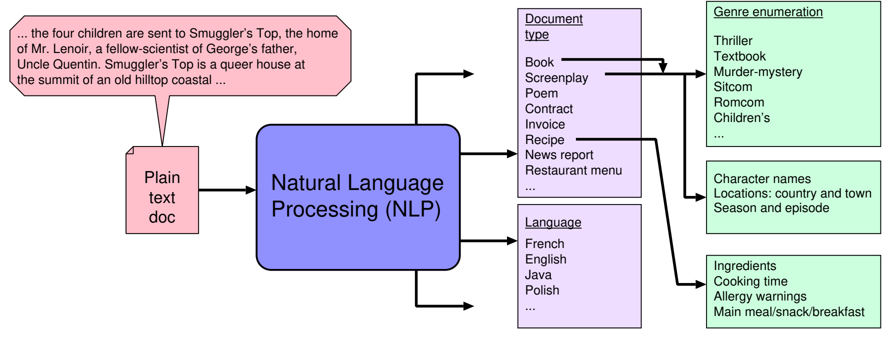
Your Ultimate Guide to the Non-Relational Universe!

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

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

Adding Structure to Unstructured Documents

Real-world data is often analogue and/or noisy



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

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

BASE - Soft state & Eventual consistency

Orthogonal aspects:

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

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

Key/Value Store

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

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

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

Serialising (marshalling or pickling) an object.

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

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

Abstract Syntax (formal spec) of XML and JSON

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

Examples

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

Slightly simplified abstract syntaxes (grammars):

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

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

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

XML – Structured or Unstructured?

Structure spectrum:

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

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

Schema rigorousness spectrum:

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

Document-oriented database systems

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

Un-structured data:

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

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

Document query languages

All sorts of queries are possible:

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

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

Typical document query languages: *eg*. XPath

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

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

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

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

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

[*NB*: pathexp_t details not examinable.]

NoSQL Movement (1)

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

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

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

Different key nestings of (semi-)structured data.

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

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

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

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

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

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

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

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

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

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

DOCtor Who “database” IMDB snapshot

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

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

The provided **DOCtor Who** “database” just has one aggregate containing films and people. Here database is in quotes since its just a Python dictionary mapping from a key to a JSON object.

DOCtor Who: Example person record.

person_id nm0031976 maps to

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

DOCtor Who: Example movie record.

movie_id tt1045658 maps to


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

But how do we query **DOCtor Who** ?

... write python code:

```
import sys      # operating system interface
import os.path # manipulate paths to files, directories
import pickle  # read/write pickled python dictionaries
import pprint  # pretty print JSON
#  The directory holding pickled data
data_dir = sys.argv[1]   # first command-line argument
# use os.path.join so that path works on both Windows and Unix.
doctorwho_path = os.path.join
                 (data_dir, 'imdb_doctorwho_database_v3.pickled')
# Open data dictionary file and un-pickle it.
doctorwhoFile = open(doctorwho_path, mode= "rb")
doctorwho     = pickle.load(doctorwhoFile)

#####################################
# write your query code here ...
movie_key = "tt1109624" # Paddington - 2014
pprint.pprint (doctorwho[movie_key])
```

Quick, in-lecture DEMO

Things to think about (for Tick 2):

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

Branded types – an opposite to semi-structured.

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

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

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

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

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

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

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

The NoSQL schema-free ideal (grail) ?

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

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

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

Semi-structured, Aggregate and NoSQL Summary

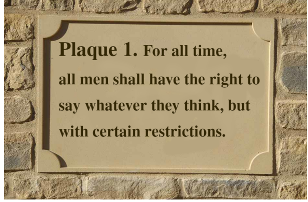
There has been a lot of churn in this area:

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

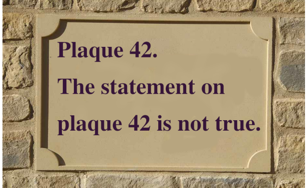
NB: For document database Tripos questions, a well-argued answer can garner full credit, even if completely disagreeing with the expected answer.

Lecture 7 - Further SQL

Declarations always hold:



Recursive declarations sometimes make sense:



(Hmm, no fixed point.)

Another look at SQL

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

Complexity of a Join?

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

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

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

We have already spoken of a table having an index.

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

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

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

Remarks

Typical SQL commands for creating and deleting an index:

```
CREATE INDEX index_name on S(B)

DROP INDEX index_name
```

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

Why the `distinct` in the SQL?

The SQL query

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

will produce a bag (multiset)!

R

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

$\Longrightarrow$

$Q(R)$

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

SQL is actually based on multisets, not sets.

Why Multisets?

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

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

`group by` $\Longrightarrow$

course	mark
spelling	99
spelling	3
spelling	100
spelling	92

course	mark
databases	92
databases	100
databases	92
databases	88

Visualizing the aggregate function **min**

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

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

course	min(**mark**)
spelling	3
databases	88

Looking at this in SQL

```
select course,
       min(mark),
       max(mark),
       avg(mark)
from marks
group by course;
+-----------+-----------+-----------+-----------+
| course    | min(mark) | max(mark) | avg(mark) |
+-----------+-----------+-----------+-----------+
| databases |        88 |       100 |   93.0000 |
| spelling  |         3 |       100 |   73.5000 |
+-----------+-----------+-----------+-----------+
```

What is NULL?

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

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

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

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

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

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

NULL can lead to unexpected results

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

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

The ambiguity of NULL

Possible interpretations of NULL

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

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

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

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

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

SQL's `NULL` has generated endless controversy

C. J. Date [D2004], Chapter 19

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

In defense of Nulls, by Fesperman

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

How can we select on null then?

With our small database, the query

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

returns 4892 records of `NULL`.

The SQL 'IS NULL' predicate:

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

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

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

Flaws? One example of SQL's inconsistency.

Furthermore, the query

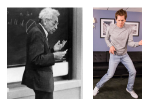
```
SELECT note, count(*) AS total
FROM credits
WHERE note IS NULL GROUP BY note;
```

returns a single record

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

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

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

[Erdős or] Bacon Number 

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

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

Let's try to calculate Bacon numbers using SQL.

First, what is Kevin Bacon's person_id?

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

Result is "nm0000102".

Function composition and relation composition

Function composition operator:

Given two functions, f and g,

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

Relation composition operator:

Given two binary relations

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

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

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

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

Partial functions as relations

Functions of one argument are special cases of relations:

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

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

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

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

Directed Graphs

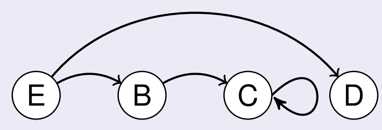
- $G = (V, A)$ is a **directed graph**, where
- V a finite set of **vertices** (also called **nodes**).
- A is a binary relation over V. That is $A \subseteq V \times V$.
- If $(u, v) \in A$, then we have an **arc** from u to v.
- The arc $(u, v) \in A$ is also called a directed edge, or a **relationship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

$$
\begin{aligned}
V &= \{E, B, C, D\} \\
A &= \{(E, B), (E, D), (B, C), (C, C)\}
\end{aligned}
$$

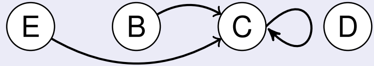
E B C D

Composition example

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



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



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

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

Iterated composition and paths.

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

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

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

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

Observation

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

Shortest path

Definition of R-distance (hop count)

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

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

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

[*NB:* By π_1 we mean extracting the first field, since π_k is the kth projection function.]
[*NB:* This is the 'single-source' shortest path problem. *Algorithms Ia* also considers all-sources shortest path problem.]

Let R be the co-actor relation

```
DROP VIEW  IFEXISTS coactors;

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

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

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

SQL: Bacon number 1

```
DROP VIEW IF EXISTS bacon_number_1;

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

Remember Kevin Bacon's person_id is nm0000102.

SQL: Bacon number 2

```
DROP VIEW IF EXISTS bacon_number_2;

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

SQL: Bacon number 3

```
DROP VIEW IF EXISTS bacon_number_3;

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

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

SQL: Bacon number 9

```
DROP VIEW IF EXISTS bacon_number_9;

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

SQL: Bacon numbers

```
DROP VIEW IF EXISTS bacon_numbers;

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

Bacon Numbers, counted

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

Results

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

bacon_number_9 is empty!

Transitive closure

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

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

Then

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

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

A ‘let rec’ for SQL enables recursion.

The WITH keyword in SQL allows a recursive declaration:

Does this have a least-fixed-point?

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

How about this one?

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

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

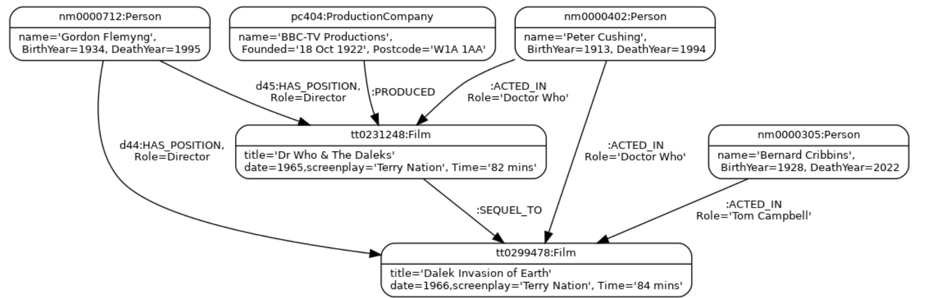
Recursive Bacon SQL query

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

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

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

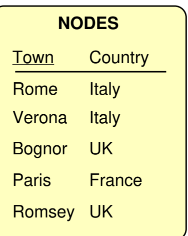
Lecture 8: Graph-oriented Databases



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

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

We could simply store graphs in relational tables?



NODES

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

EDGES

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

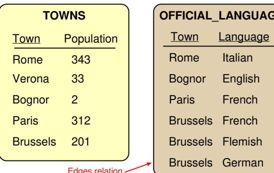
This is a unary relation: the schema range and domain type are both towns.

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

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

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

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

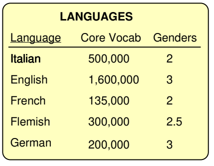


TOWNS

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

OFFICIAL_LANGUAGE

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



LANGUAGES

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

Modelling ternary relations?

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

Neo4j: Cypher immediate data entry.

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

Immediate Node Data:

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

Immediate Edge Data:

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

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

Graph data normalisation.

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

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

Hmm!

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

Better:

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

Databases and Graph Databases

General points:

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

Neo4J specific:

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

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

Neo4j — example pattern-matching queries:

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

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

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

Pattern matching. Example 1:

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

Resulting in:

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

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

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

Pattern matching. Example 2: co-actors

Get all co-actors with

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

OR

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

OR

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

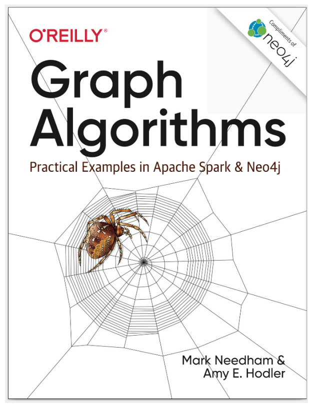
Resulting in:

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

Graph Algorithms (are important)

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

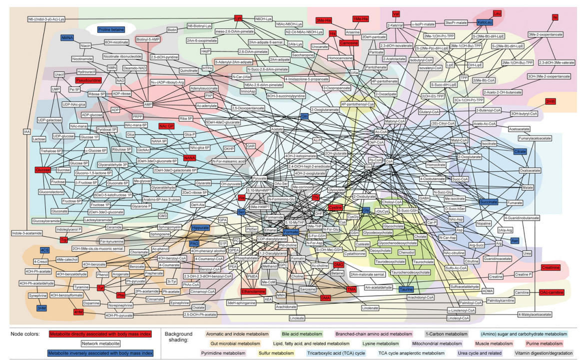
- Breadth-first search, depth-first, shortest path, Page Rank, spanning trees, articulation point, strongly-connected components, cliques, max flow ...



Metrics:

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

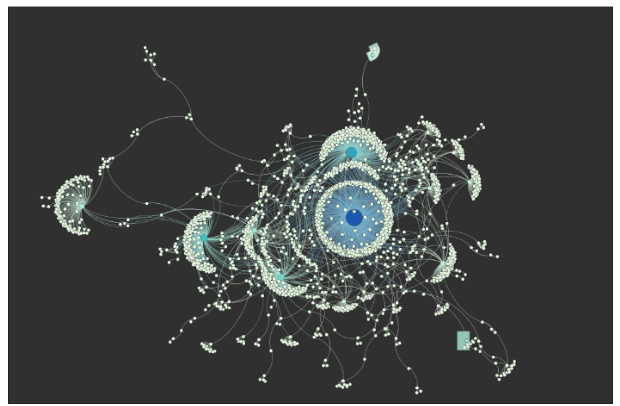
Graph DBMS optimised for Big Data (will not fit in core*)“Data Science” queries.



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

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

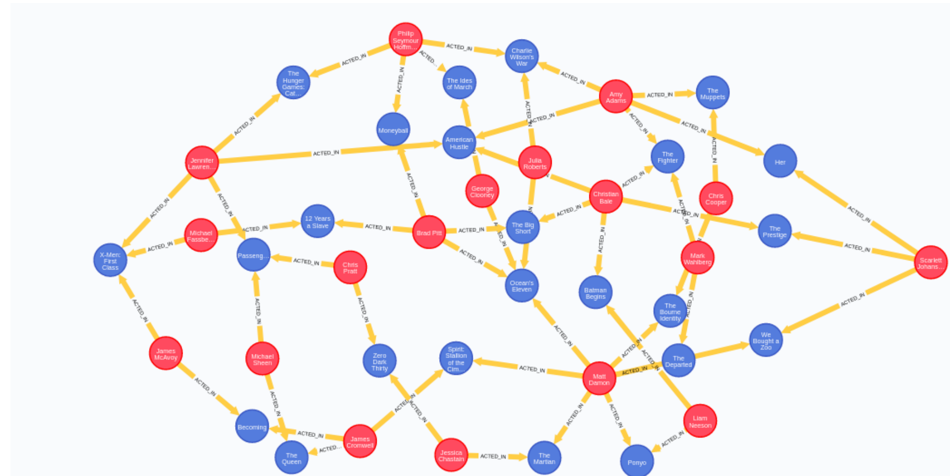
Social networks



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

Neo4j: Example of path-oriented query in Cypher

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



Let's count Bacon numbers with Neo4j/Cypher

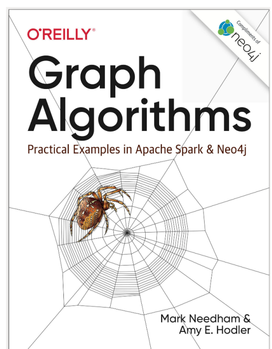
```
MATCH paths=allshortestpaths(
     (m:Person {name : "Kevin Bacon"} )
        -[:ACTED_IN*]-
     (n:Person))
WHERE n.person_id <> m.person_id
RETURN length(paths)/2 AS bacon_number,
       COUNT(distinct n.person_id) AS total
ORDER BY bacon_number;
```

```
 bacon_number   total
 ------------   -----
            1     12
            2    110
            3    614
            4    922
            5    381
            6    123
            7     86
            8     16
```

Graph-oriented DBMS optimisations:

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

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



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

Last Slide!

What have we learned?

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

End of the course.

Some declarations do not quite hold for all time: