1A Databases
Lecture 7 (of 8)

Timothy G. Griffin

Computer Laboratory
University of Cambridge, UK

Michaelmas 2016

```
{ "id": 107303,
  "name": "Bacon, Kevin (I)", "gender": "male",
  "actor_in": [
    { "character": "Jack Brennan", "position": 4,
      "movie_id": 2914879,
      "title": "Frost/Nixon (2008)"}]}
```

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

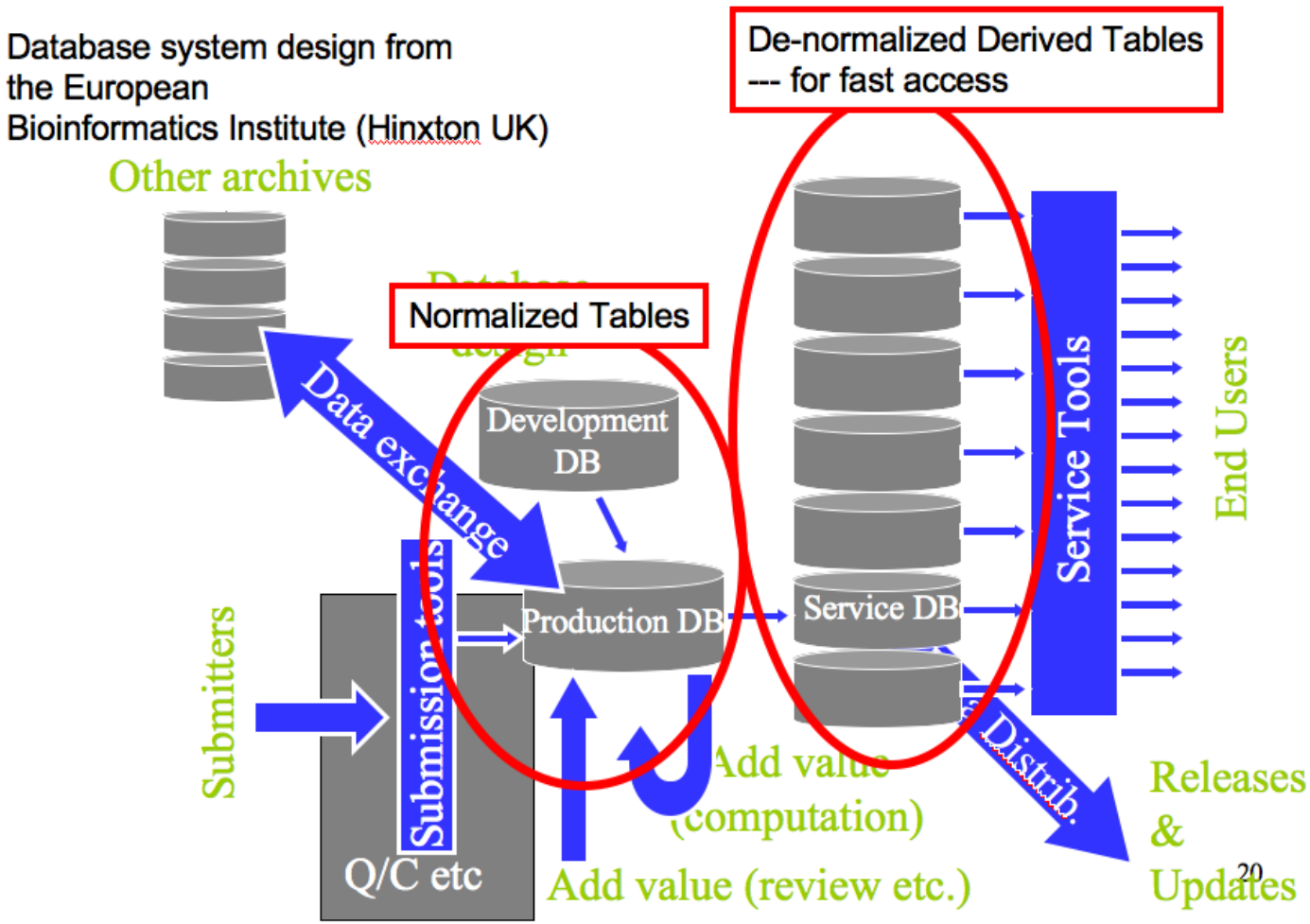
A fundamental tradeoff

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

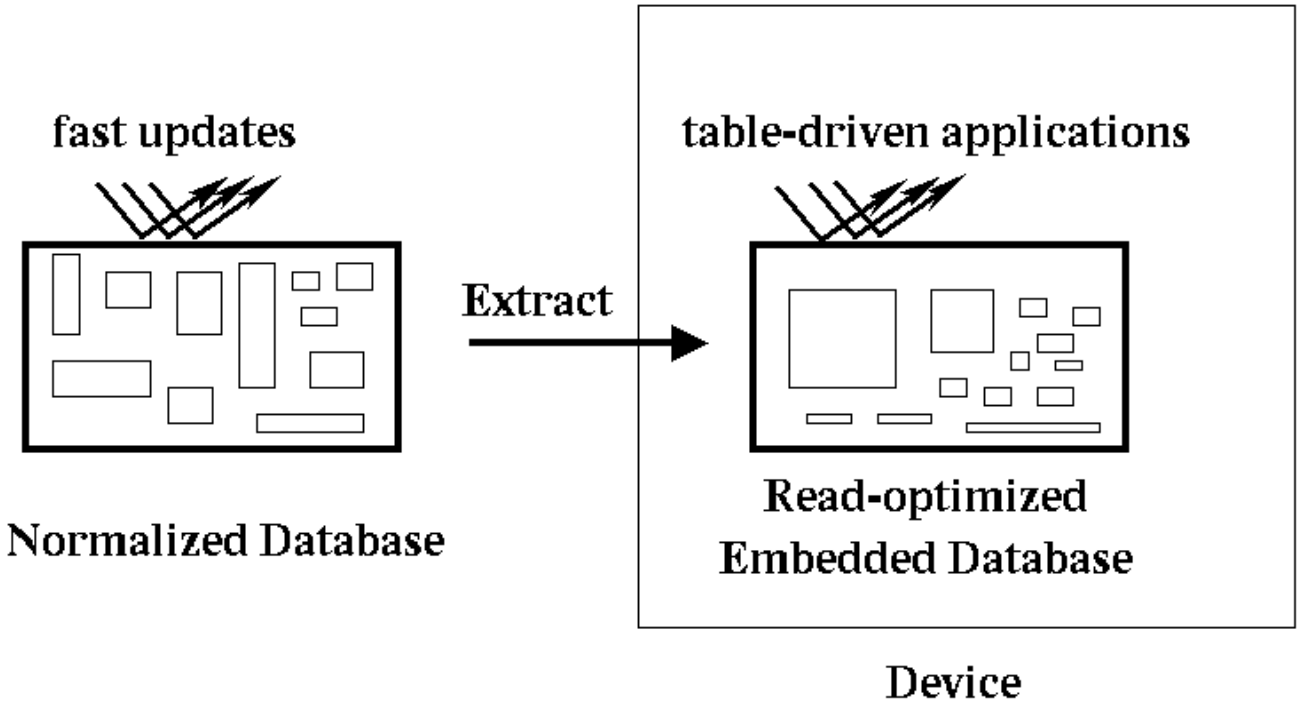
Situations where we might want a read-oriented database

- 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

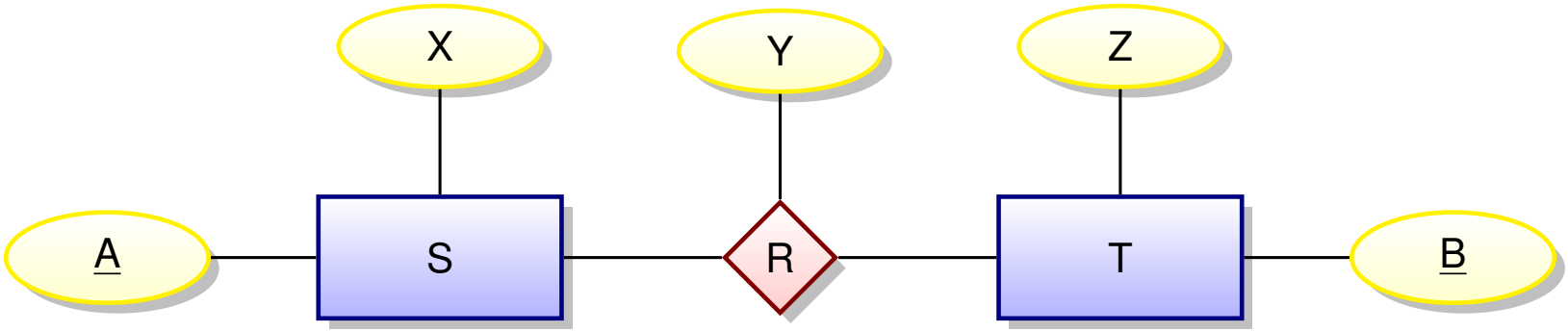
Our working definition

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!

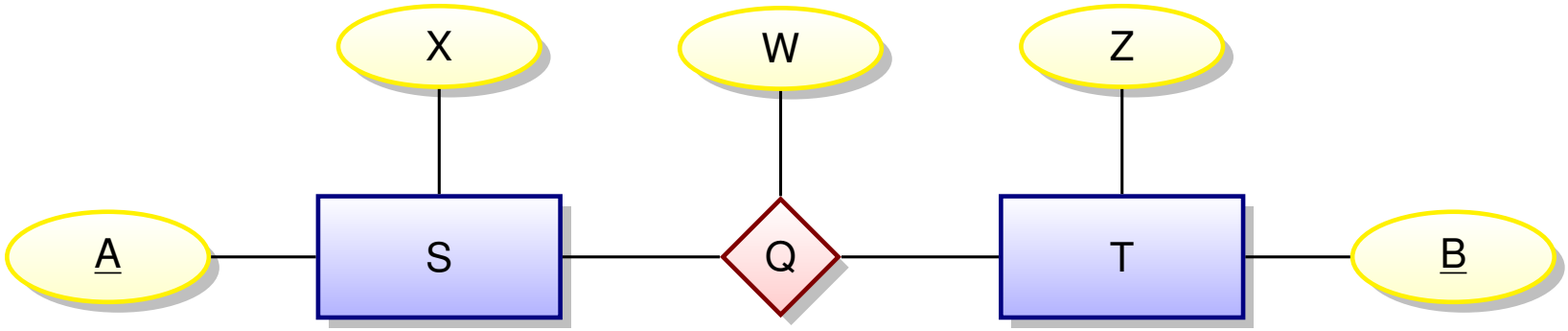
Implement the relationship as one big table?

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	w2
a1	x1	b3	z3	y3	
a1	x1	b4	z4		w1
a2	x2	b1	z1	y4	
a2	x2	b3	z3	y5	
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.