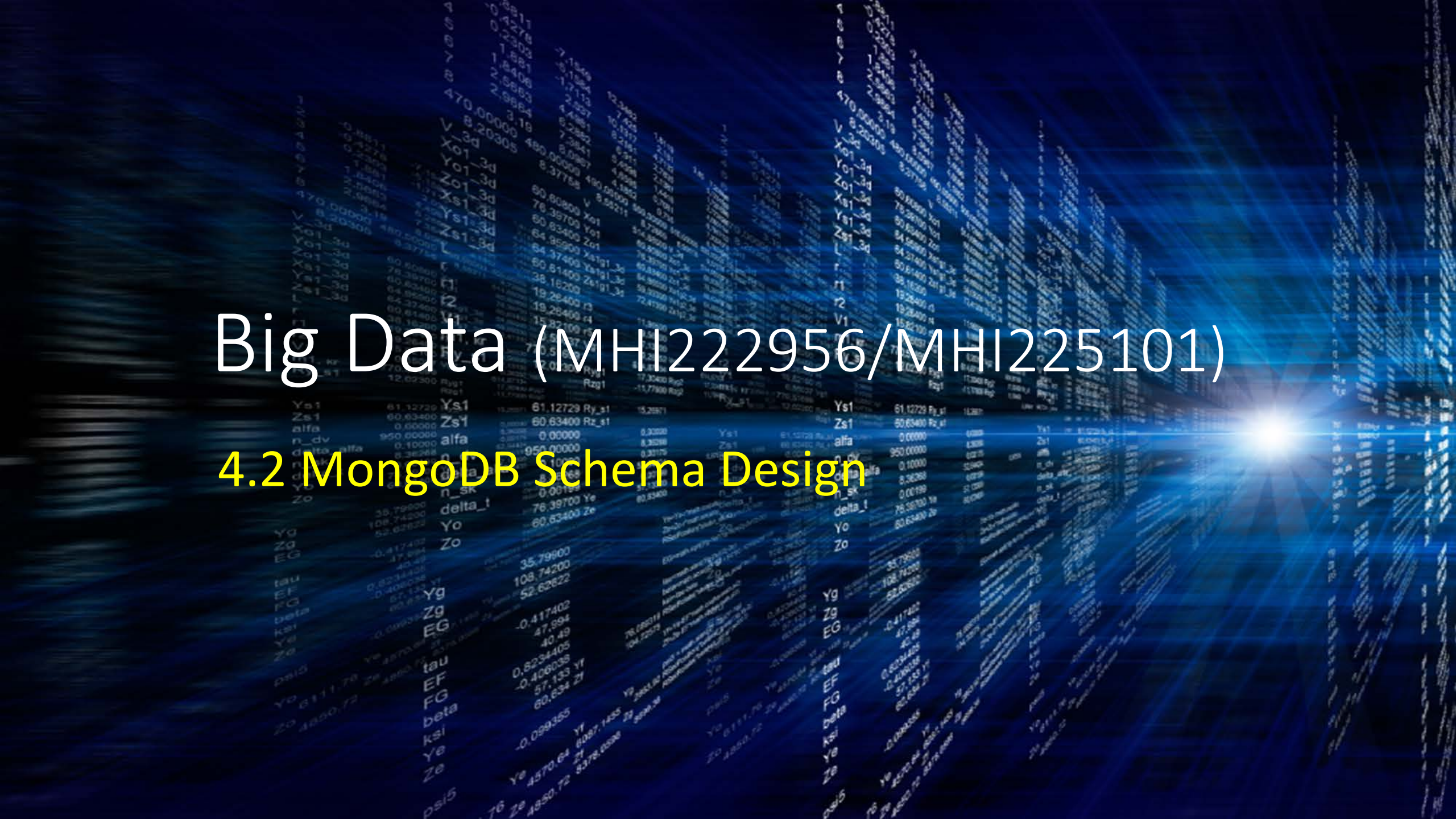


Big Data (MHI222956/MHI225101)

4.2 MongoDB Schema Design

- MongoDB does not enforce a schema
- It's not recommended to proceed completely at random
- Schema design is critical for improving the performance and scalability of non-relational database

Relational vs. MongoDB Schema Design

- With a relational database, a schema need to be defined before you can load data into the database.
- With a NoSQL system, data schema is strictly optional, However, the process of making sense of that information and producing something useful from it actually yields a schema as a byproduct even if you do not realize it.

- Relational database modelling is typically driven by the structure of available data.
- NoSQL data modelling is typically driven by application-specific access patterns.

	Relational database	MongoDB
Design theme	What data do I have?	What questions do I want to ask of my database?
Steps to create the model	1. Define schema 2. Develop app and queries	1. Identifying the queries 2. Define schema
Initial schema	• 3rd normal form • One possible solution	• Many possible solutions
Schema evolution	• Difficult and not optimal • Likely downtime	• Easy • No downtime
Query performance	mediocre	optimized

- Entity-relationship Diagrams (ERD)
  - Widely used in relational database modelling
  - An entity is a thing that exists either physically or logically
  - A relationship captures how entities are related to one another
  - Normalized ERD is converted into relational model (tables)
    - 3rd normal form
- ERD can be used in MongoDB to help the understand of the data and the relations.
- No join in MongoDB

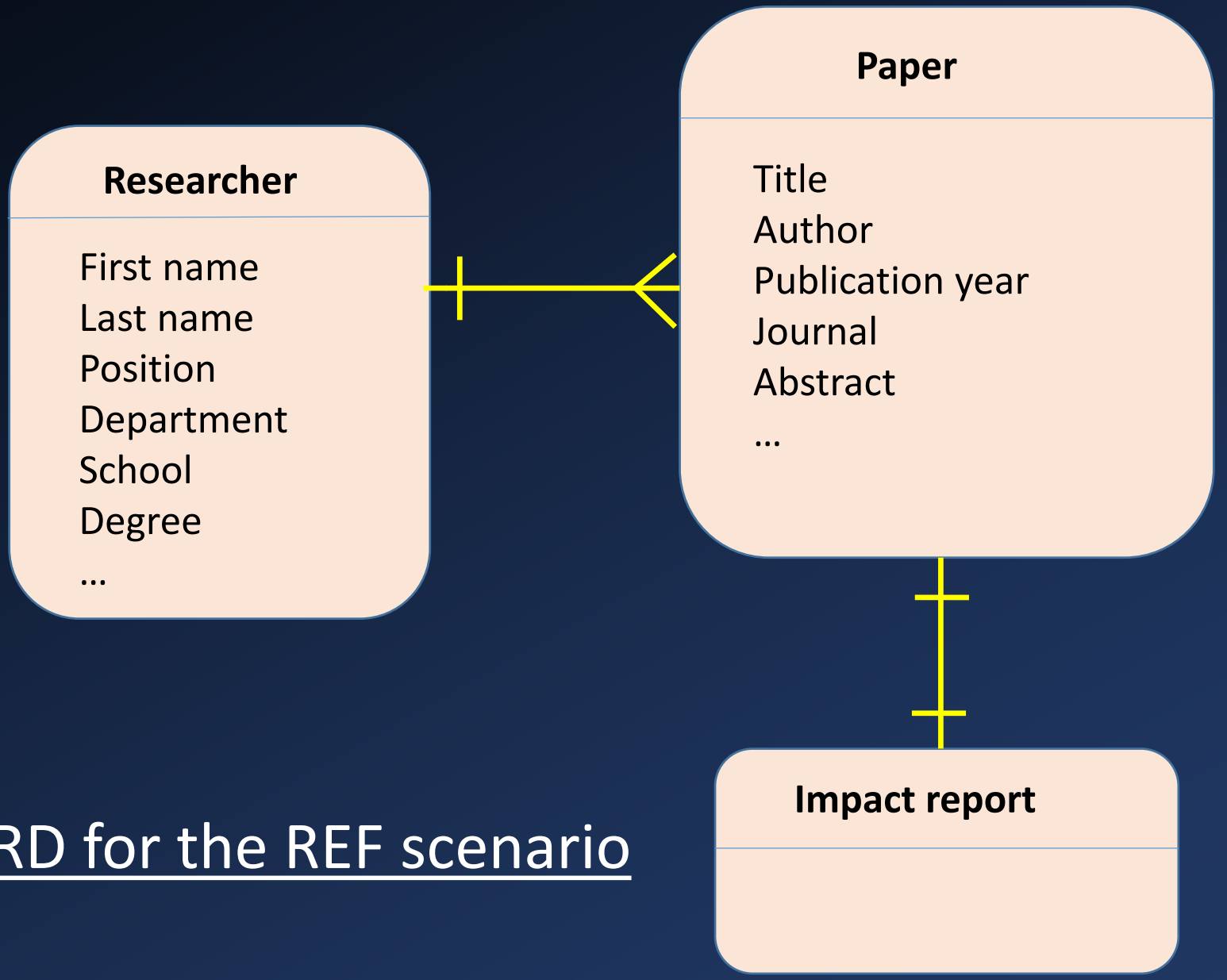
A simplified “REF” scenario:

- REF -- the UK’s system for assessing the quality of research in UK higher education institutions

- Use cases:
  - Find all researchers in a school
  - Find all papers published by a specified researcher
  - Write a impact report on a paper

Entities in this scenario

ERD for the REF scenario

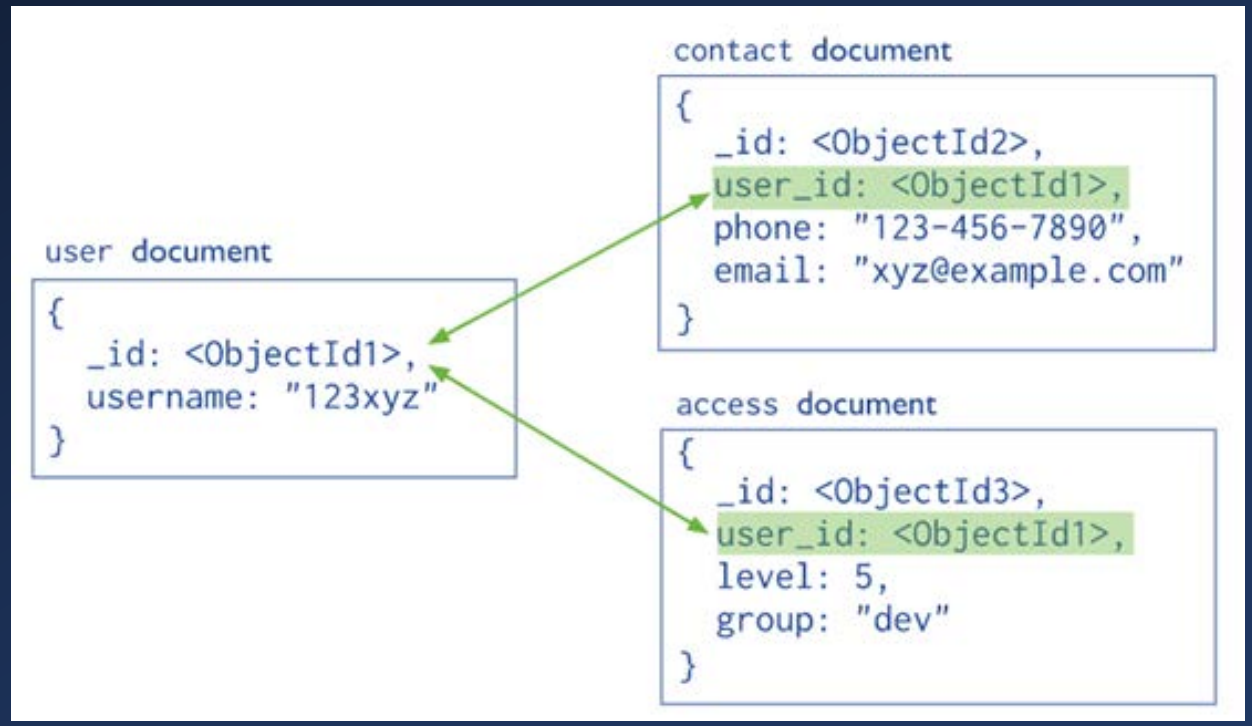


Embedded Data models

- "Denormalized" models: embed related data in a single structure or document
- Request and retrieve related data in a single database operation

References - Normalized Data Models

- Normalized data models describe relationships among documents using references
- References provides more flexibility than embedding
- Client-side applications must issue follow-up queries to resolve the references



	Embedding	Referencing
Pros	• Retrieve all data with a single query • Avoids expense JOINs or $lookup • Update all data with a single atomic operation	• Smaller documents • Less likely to reach 16MB limit • No duplication of data • Infrequently accessed data not accessed on every query
Cons	• Large docs • 16MB document size limit	• Two queries or $lookup required to retrieve all data

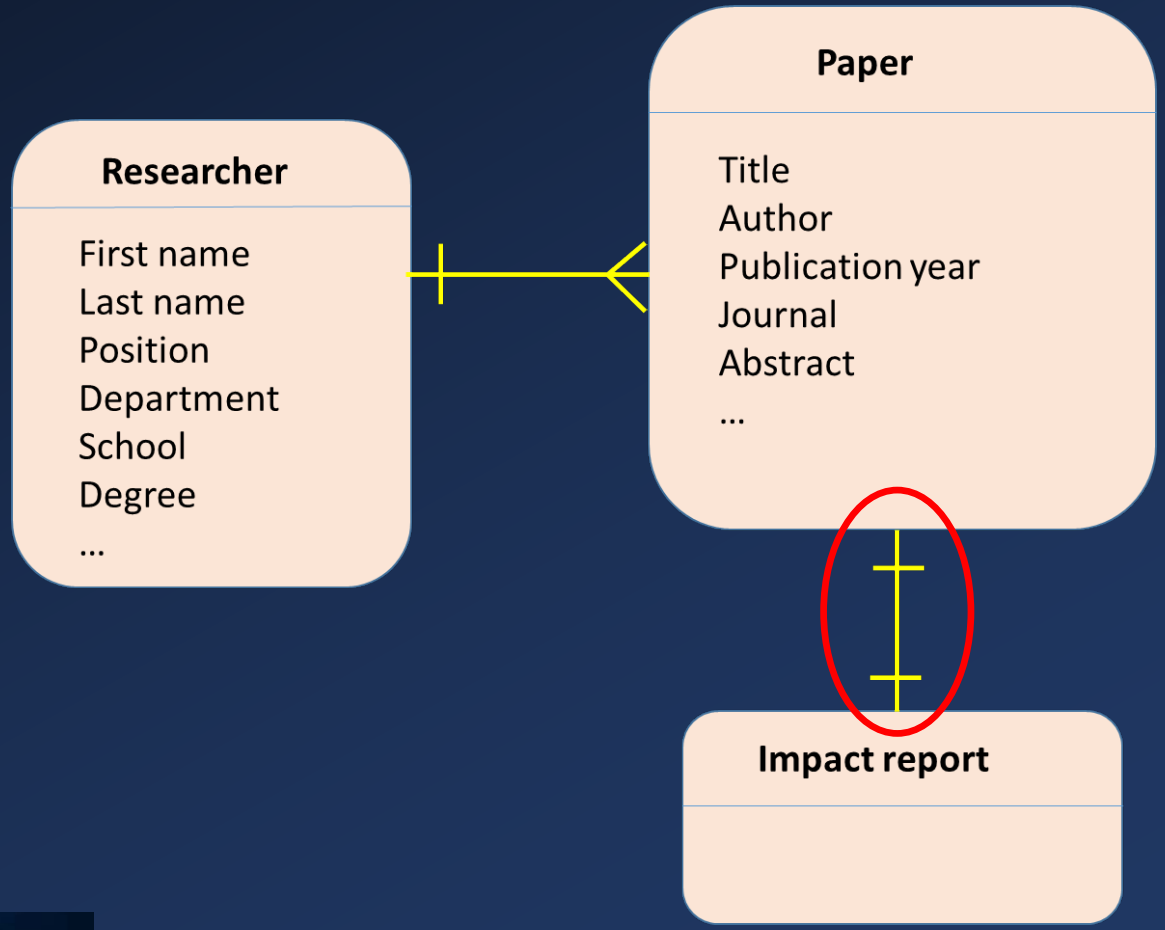
Modelling the relationships

- Relationships in MongoDB
  - represent how various documents are logically related to each other
- Types of relatioships:

  1:1; 1:N; N:1; N:N
- Relationships can be modelled via:
  - Embedded model
  - Referenced model

One-to-One relationships

One-to-one relationships can easily be handled with embedding a document inside another document



- Paper:

```
{
 "_id" : ObjectID("paper01"),
 "Title" : "API based data integration",
 "Author" :  "George Moore",
 "Publication year": 2020,
 "Journal" : "IEEE transaction of Big Data",
 "Abstract" : "Data integration is an important..."
 }
```

- Impact report

```
{
 "_id" : ObjectID("IP01"),
 "paper_id" : ObjectID("paper01"),
 "impact report": "This paper leads to ..."
 }
```

- Embed “impact report” into “Paper”

```
{
 "_id" : ObjectID("paper01"),
 "Title" : "API based data integration",
 "Author" :  "George Moore",
 "Publication year": 2020,
 "Journal" : "IEEE transaction of Big Data",
 "Abstract" : "Data integration is an important..."
 "impact report": "This paper leads to ..."
 }
```

- Paper:

```
{
" _id" : ObjectID("paper01"),
"Title" : "API based data integration",
"Author" :  "George Moore",
"Publication year": 2020,
"Journal" : "IEEE transaction of Big Data",
"Abstract" : "Data integration is an important..."
}
```

- Impact report

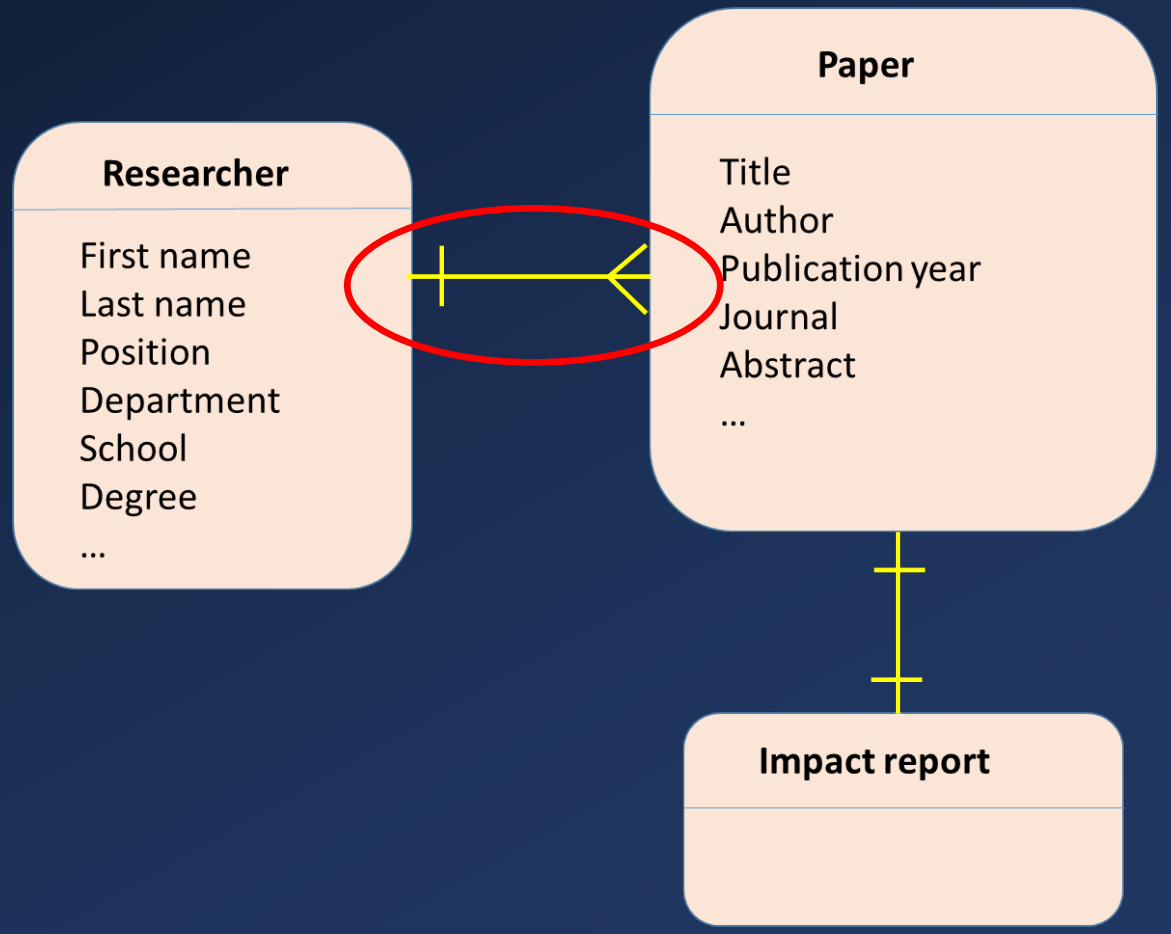
```
{
" _id" : ObjectID("IP01"),
"paper_id" : ObjectID("paper01"),
"impact report": "This paper leads to ..."
}
```

One-to-N relationships

- In MongoDB, there are variety of ways to model “One-to-N” relationships
- Need to consider the size of “N”:
  - One-to-Few
  - One-to-Many
  - One-to-Tons

One-to-Few relationships

- **One-to-few** relationship : a single record is linked with a relatively small number of other data points
- Can embed an array of those few information inside the other document



```
{
 "_id" : ObjectID("researcher01"),
 "First name" : "George",
 "Last name" : "Moore",
 "Position" : "Professor",
 "Department" : "Computing",
 "School" : "SCEBE",
 "Degree" : ["BSc", "MSc", "PhD"],
 "Paper" : [
            {
             "_id" : ObjectID("paper01"),
             "Title" : "API based data integration",
             "Publication year": 2020,
             "Journal" : "IEEE transaction of Big Data",
             "Abstract" : "Data integration is an important..."
             "impact report": "This paper leads to ..."
             }
                        {
             "_id" : ObjectID("paper03"),
             "Title" : "An survey on the creation of smart city",
             "Publication year": 2019,
             "Journal" : "IEEE transaction of IoT",
             "Abstract" : "The concept of smart city has ..."
             "impact report": "This paper has been cited ..."
             }
             ]
}
```

One-to-Many relationships

- **One-to-many** where “many” covers up to a few thousand or so in number
- Use an array of references

```
{
  "_id" : ObjectId("57d7a121fa937f710a7d486e"),
  "manufacturer" : "Elegoo",
  "catalog_number" : 123789,
  "parts" : [
     ObjectID("AAAA"),
     ObjectID("AAAB"),
     ObjectID("G9D6"),
     ...
  ]
}
```

```
{
  "_id" : ObjectId("AAAA"),
  "part_no" : "150ohm-0.5W"
  "name" : "150ohm 1/2 Watt Resistor"
  "qty" : 1
  "cost" : { NumberDecimal("0.13"), currency: "USD" }
}
```

One-to-Tons relationships

- Using Parent Refereeing

Hosts:

```
{
  _id: ObjectId("AAA"),
  name: "goofy.example.com",
  ipaddr: "127.66.66.66",
}
```

Log Message:

```
{
  _id: ObjectId("123"),
  time: ISODate("2014-03-28T09:42:41.382Z"),
  message: "The CPU is on fire!!!",
  host: ObjectId("AAA"),
},
{
  _id: ObjectId("456"),
  time: ISODate("2014-03-28T09:42:41.382Z"),
  message: "Drive is hosed",
  host: ObjectId("AAA"),
}
```

Many-to-Many relationship

- For **Many-to-Many** relationships, we can do a two-way reference

Golden rules -- General considerations for designing Schema in MongoDB:

- Design your schema according to user requirements.
- Optimize your schema for most frequent use cases.
- Favour embedding unless there is a compelling reason not to.
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Avoid Joins and $lookups if they can be avoided
- Do complex aggregation in the schema.

Aggregate oriented database

- An aggregate defines a transaction boundary and is the atomic unit for updates
- Defines a group of data that “belongs together”
- Provides some transaction control, though not to same degree as relational databases
- KV, document and wide column stores are essentially aggregate oriented
- Each key identifies an aggregate, not a simple row

Aggregate oriented database

- In contrast, relational, object and graph databases are aggregate ignorant
- Aggregate-oriented databases work best when most data interaction is done with the same aggregate
- Aggregate-ignorant databases are better when interactions use data stored in many different structures

Aggregate oriented database

- Aggregates are useful in highly distributed sharded databases
- Store all data within an aggregate on same node, most data access involves single nodes, can be parallelised
- Design so that all data needed for main use cases is within an aggregate, stored within a shard
- Data access which does not follow aggregate pattern can be costly
- Some aggregate oriented databases allow you to define indexes for searching within aggregates

Aggregate oriented database

- Some aggregate oriented databases allow you to define indexes for searching within aggregates
- but also to make “join” queries efficient
- Queries which summarise data across many aggregates can be done with MapReduce
- MapReduce commonly done with analytic engine (Hadoop, Spark) but this analytic capability is also supported within some NoSQL databases, e.g MongoDB

Summary

- Relational vs. MongoDB Schema Design
- Embedded Data models
- Referencing - Normalized Data Models
- One-to-One relationships
- One-to-N relationships
- Aggregate oriented database