Large-Scale and Multi-Structured Databases

Document Databases

Data Modelling and Partitioning

Prof. Pietro Ducange

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
UNIVERSITÀ DI PISA
CROSSLAB Innovation for industry 4.0

Some considerations

A *document database* could ***theoretically*** implement a ***third normal form schema***.

Tables, as in relational databases, may be "***simulated***" considering collections with ***JSON*** documents with an identical ***pre-defined structure***.

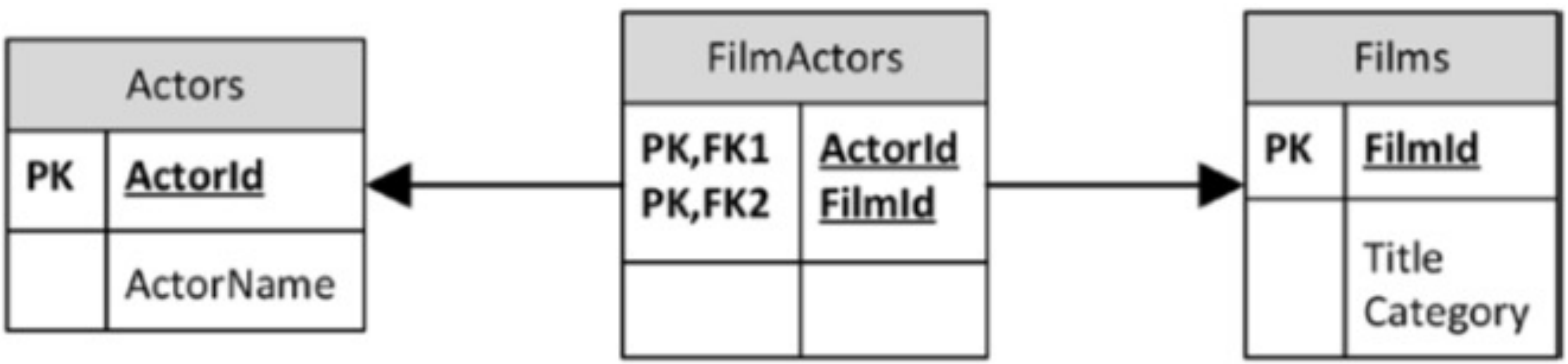


Image extracted from "Guy Harrison, Next Generation Databases, Apress, 2015"

JSON Databases: An example

Document databases usually adopts a ***reduced number*** of collections for modeling data.

Nested documents are used for representing ***relationships*** among the different entities.

Document databases ***do not*** generally provide ***join operations***.

Programmers like to have the JSON structure map ***closely to the object*** structure of their code!!!

Image extracted from “Guy Harrison, Next Generation Databases, Apress, 2015”

Data Modeling: Document Embedding

Film Document

Actor document

Array of actors

```
{ "_id": 115,"Title": "CAMPUS REMEMBER",
              "Category": "Action",
    "Actors" :
        [
          { "actorId": 8,"Name": "MATTHEW JOHANSSON" },
          { "actorId": 45,"Name": "REESE KILMER" },
          { "actorId": 168,"Name": "WILL WILSON" }
        ]
}
```

The solution above allows the user to ***retrieve*** a film and all its actors in a ***single operation***.

However, “actors” result to be ***duplicated*** across multiple documents.

In a complex design this could lead to ***issues*** and possibly ***inconsistencies*** if any of the “actor” attributes need to be changed.

Moreover, some JSON databases have some ***limitations*** of the maximum ***dimension*** of a single document.

Data Modeling: Document Linking

In the solution above, an array of actor ***IDs*** has been embedded into the film document.

The IDs can be used to ***retrieve*** the documents of the actors (in on other collection) who appear in a film.

We are ***rolling back*** to a relational model!!! Now, at least two collections of document must be defined.

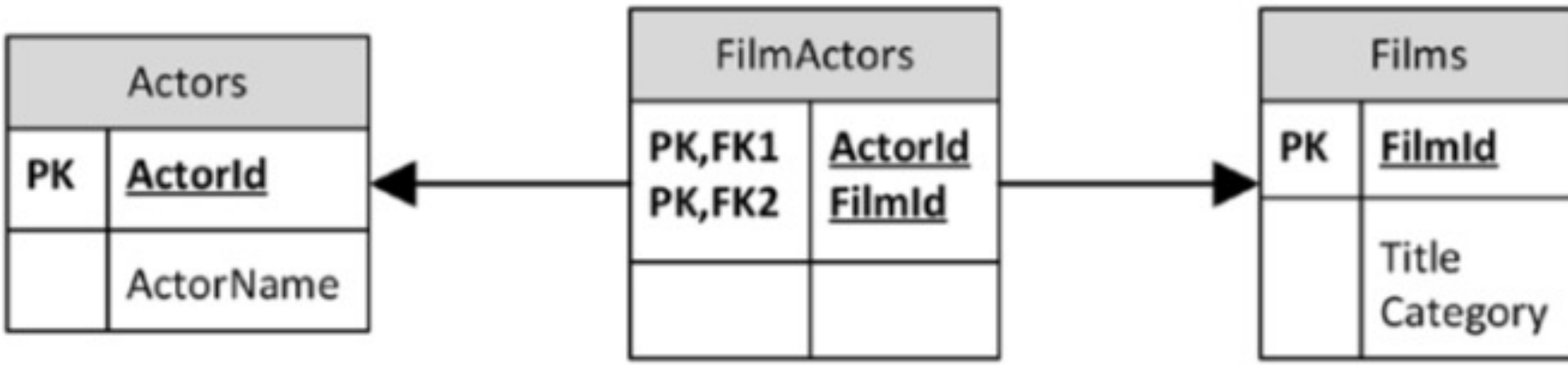
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

UNIVERSITÀ DI PISA

CROSSLAB Innovation for industry 4.0

Data Modeling: Document Linking

We are ***rolling back*** to a the third normal form!!!!!



Data Modeling: Some Discussions

Document linking approaches are usually somewhat an ***unnatural*** style for a document database.

However, for ***some workloads*** it may provide the best ***balance*** between performance and maintainability.

When modeling data for document databases, there is no equivalent of third normal form that defines a "***correct***" model.

In this context, the ***nature of the queries*** to be executed ***drives*** the approach to ***model*** data.

Data Modeling: One to Many Relationship

Let consider the ***customer*** entity which may have associated a list of ***address*** entities.

```
{
   customer_id: 76123,
   name: 'Acme Data Modeling Services',
   person_or_business: 'business',
   address : [
                  { street: '276 North Amber St',
                    city: 'Vancouver',
                    state: 'WA',
                    zip: 99076} ,
                  { street: '89 Morton St',
                    city: 'Salem',
                    state: 'NH',
                    zip: 01097}
                 ]
   }
```

The basic pattern is that the ***one*** entity in a one-to-many relation is the ***primary document***, and the ***many*** entities are represented as an array of ***embedded documents***.

Data Modelling: Many to Many Relationships

Let consider an example of application in which:

- A ***student*** can be enrolled in many courses
- A ***course*** can have many students enrolled to it

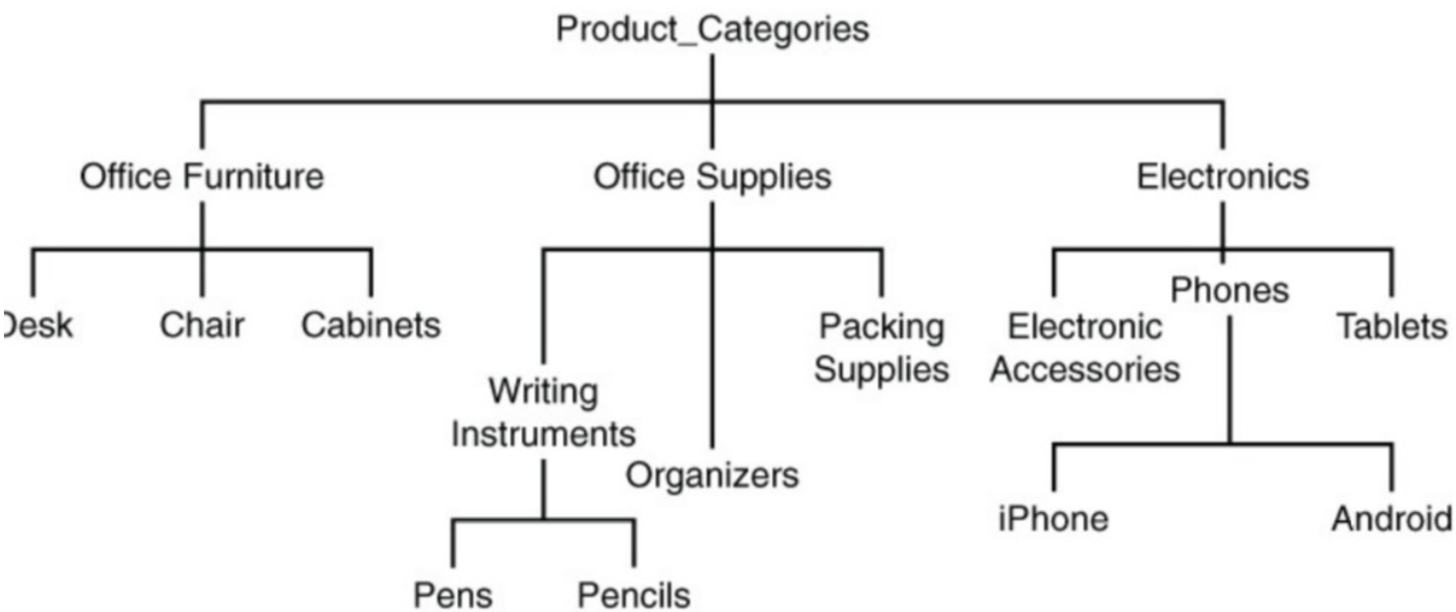
We can model this situation considering the following two collections:

```
{
  { courseID: 'C1667',
    title: 'Introduction to Anthropology',
    instructor: 'Dr. Margret Austin',
    credits: 3,
    enrolledStudents: ['S1837', 'S3737', 'S9825' …
      'S1847'] },
  { courseID: 'C2873',
    title: 'Algorithms and Data Structures',
    instructor: 'Dr. Susan Johnson',
    credits: 3,
    enrolledStudents: ['S1837','S3737', 'S4321', 'S9825'
      … 'S1847'] },
  { courseID: C3876,
    title: 'Macroeconomics',
    instructor: 'Dr. James Schulen',
    credits: 3,
    enrolledStudents: ['S1837', 'S4321', 'S1470', 'S9825'
      … 'S1847'] },
```

```
{
 {studentID:'S1837',
   name: 'Brian Nelson',
   gradYear: 2018,
   courses: ['C1667', C2873,'C3876']},
 {studentID: 'S3737',
   name: 'Yolanda Deltor',
        gradYear: 2017,
        courses: [ 'C1667','C2873']},
     …
}
```

We have to take care when updating data in this kind of relationship. Indeed, the DBMS will ***not*** control the ***referential integrity*** as in relational DBMSs.

Modeling Hierarchies (I)

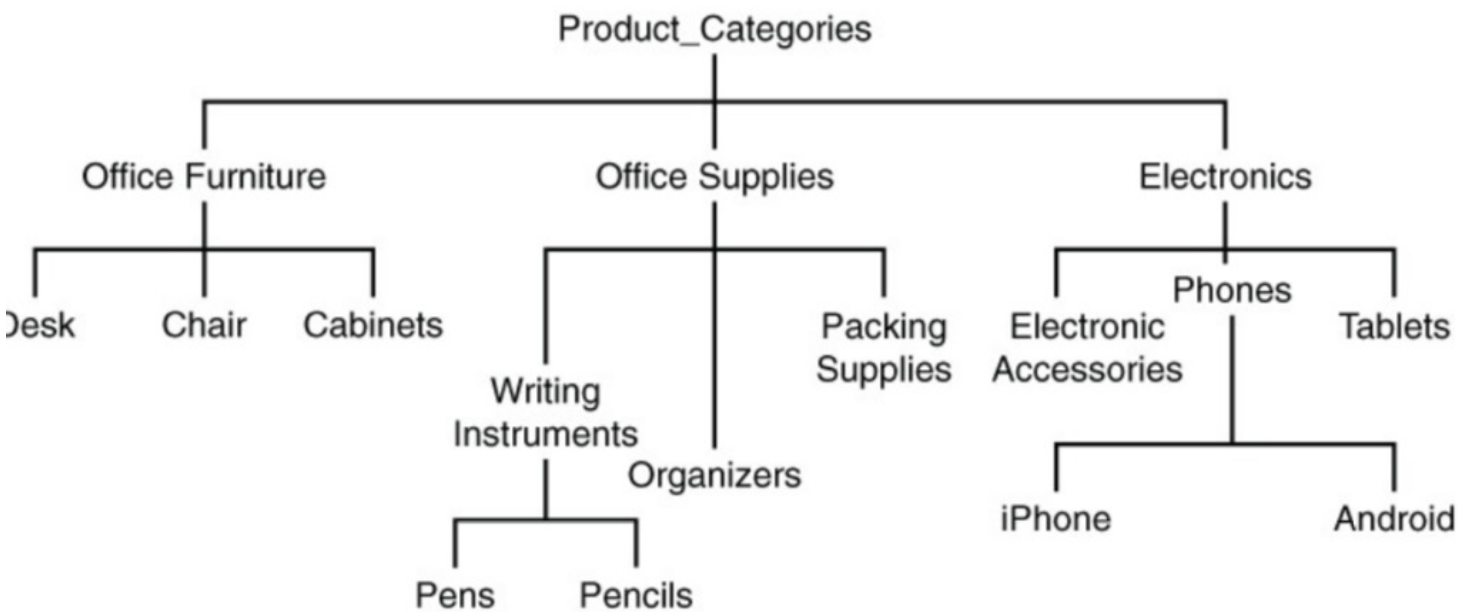


Parent reference solution:

```
{
   {productCategoryID: 'PC233', name:'Pencils',
     parentID:'PC72'},
   {productCategoryID: 'PC72', name:'Writing Instruments',
     parentID: 'PC37"},
   {productCategoryID: 'PC37', name:'Office Supplies',
     parentID: 'P01'},
   {productCategoryID: 'P01', name:'Product Categories' }
}
```

This solution is useful if we have frequently to show a specific instance of an object and then show the more general type of that category.

Modeling Hierarchies (II)



Child reference solution:

```
{
  {productCategoryID: 'P01', name:'Product Categories',
        childrenIDs: ['P37','P39','P41']},
       {productCategoryID: 'PC37', name:'Office Supplies',
        childrenIDs: ['PC72','PC73','PC74"]},
        {productCategoryID: 'PC72', name:'Writing
          Instruments', childrenIDs: ['PC233','PC234']'},
         {productCategoryID: 'PC233', name:'Pencils'}
   }
```

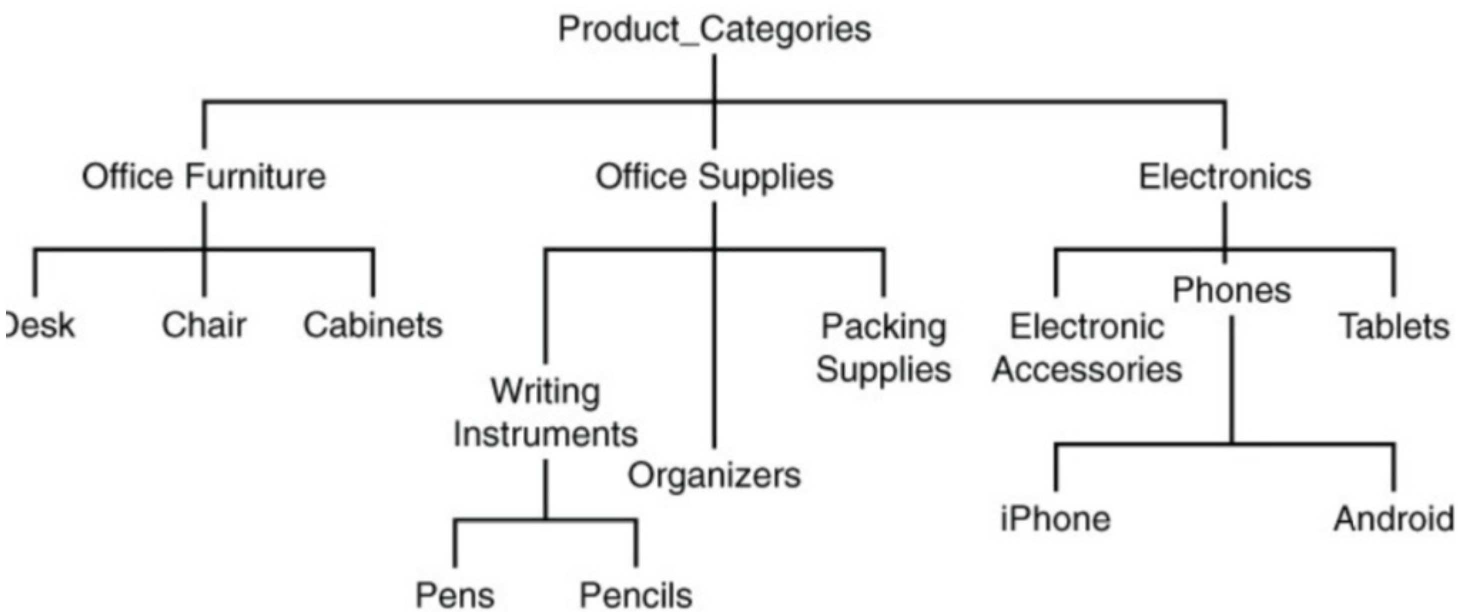
This solution is useful if we have frequently to retrieve the children (or sub parts) of a specific instance of an object.

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

UNIVERSITÀ DI PISA

CROSSLAB Innovation for industry 4.0

Modeling Hierarchies (III)



List of ancestor solutions:

```
{productCategoryID: 'PC233', name:'Pencils',
  ancestors:['PC72', 'PC37', 'P01']}
```

This solution allows to retrieve the full path of the ancestors with one read operation.

A change to the hierarchy may require many write operations, depending on the level at which the change occurred.

Data Modeling: An Example (I)

Scenario: Trucks in a company fleet have to transmit location, fuel consumption and other metrics every three minutes to a fleet management data base (one-to-many relationship between a truck and the transmitted details)

We may consider to generate a new document to add to the DB for each data transmission.

Let consider a document as follows:

```
{
    truck_id: 'T87V12',
    time: '08:10:00',
    date :  '27-May-2015',
    driver_name: 'Jane Washington',
    fuel_consumption_rate: '14.8 mpg',
    …
}
```

Repeated information in each new document

At the end of the day, the DB will include 200 new documents for each truck (we consider 20 transmissions per hour, 10 working hours)

Data Modeling: An Example (II)

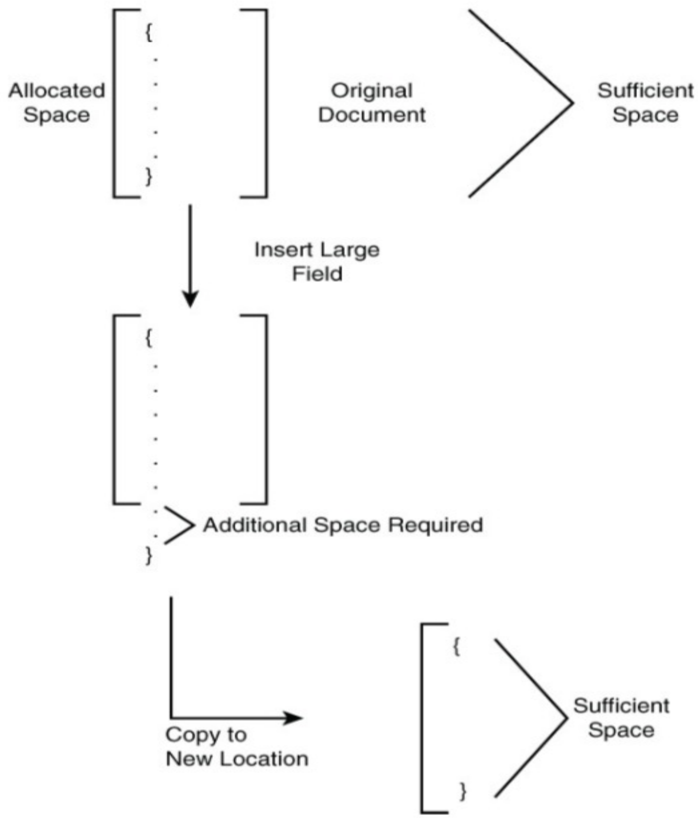
An alternative solution may be to use embedded documents as follows:

```
{
    truck_id: 'T87V12',
    date :  '27-May-2015',
    driver_name: 'Jane Washington',
    operational_data:
                   [
                       {time : '00:01',
                        fuel_consumption_rate: '14.8 mpg',
                        …},
                         {time : '00:04',
                        fuel_consumption_rate: '12.2 mpg',
                        …},
                         {time : '00:07',
                        fuel_consumption_rate: '15.1 mpg',
                        …},
                    …]
}
```

Pay attention: we can have a potential ***performance problem***!

Planning for Mutable Documents (I)

When a document is created, the DBMS allocates a certain amount of space for the document.



If the document ***grows more*** than the allocated space, the DBMS has to ***relocate*** it to another location.

Moreover, the DBMS ***must free*** the previously allocated space.

The previous steps can adversely affect the system performance.

Planning for Mutable Documents (II)

A solution for avoiding to move oversized document is to ***allocate sufficient space*** at the moment in which the document is ***created***.

Regarding the previous problem, the following solution may be adopted:

```
{truck_id: 'T8V12'
 date: '27-May-2015'
 operational_data:
    [{time: '00 : 00',
      fuel_consumption_rate: 0.0}
     {time: '00 : 00',
      fuel_consumption_rate: 0.0}
      .
      .
      .
      .
     {time: '00 : 00',
      fuel_consumption_rate: 0.0}
      ]
```

200 Embedded Documents with Default Values

In conclusion, we have to consider the ***life cycle*** of a document and planning, if possible, the strategies for handling its growing.

Indexing Document Database

In order to avoid the entire scan of the overall database, DBMSs for document databases (for example MongoDB) allow the definition of ***indexes***.

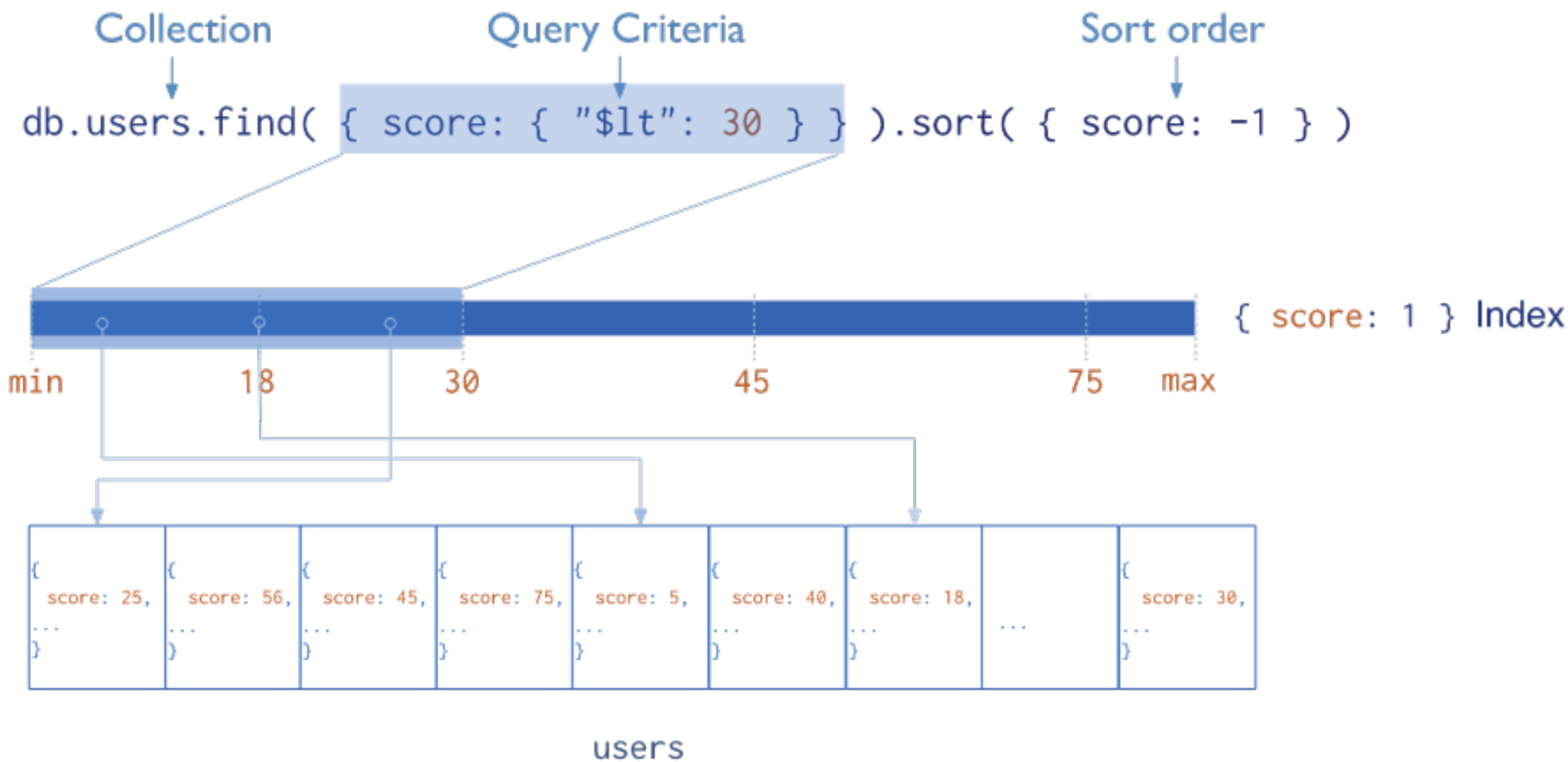
Indexes, like in book indexes, are a ***structured set of information*** that maps from one attribute to related information.

In general, indexes are ***special data structures*** that store a small portion of the collection's data set in an ***easy to traverse*** form.

The index stores the value of a specific field or set of fields***, ordered by the value of the field***.

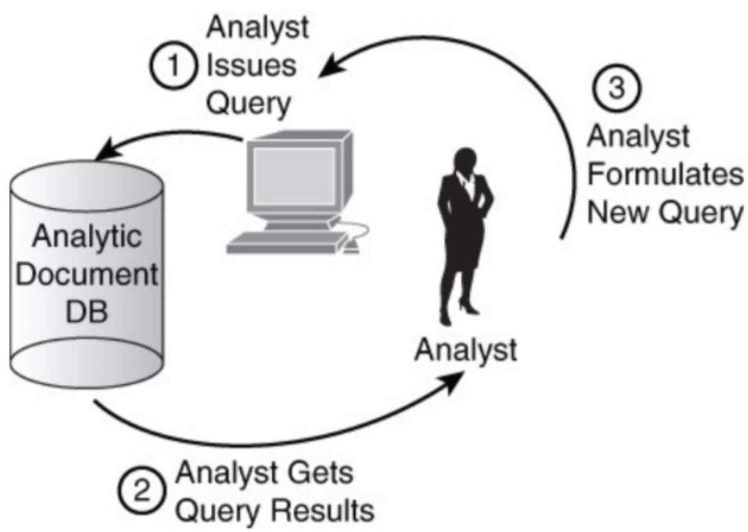
The ordering of the index entries supports ***efficient equality matches*** and ***range-based query operations***.

An Example of Index



Read-Heavy Applications

In the figure we show the classical scheme of a read-heavy application (***business intelligence*** and ***analytics applications***):



In this kind of applications, the use ***of several indexes*** allows the user to quickly access to the database. For example, indexes can be defined for easily retrieve documents describing objects related to a specific ***geographic region*** or to a specific ***type***.

Write-Heavy Applications

The example of the truck information transmission (each three minutes) is a typical write-heavy application.

The ***higher*** the number of ***indexes*** adopted the ***higher*** the amount of ***time*** required for closing a write operation.

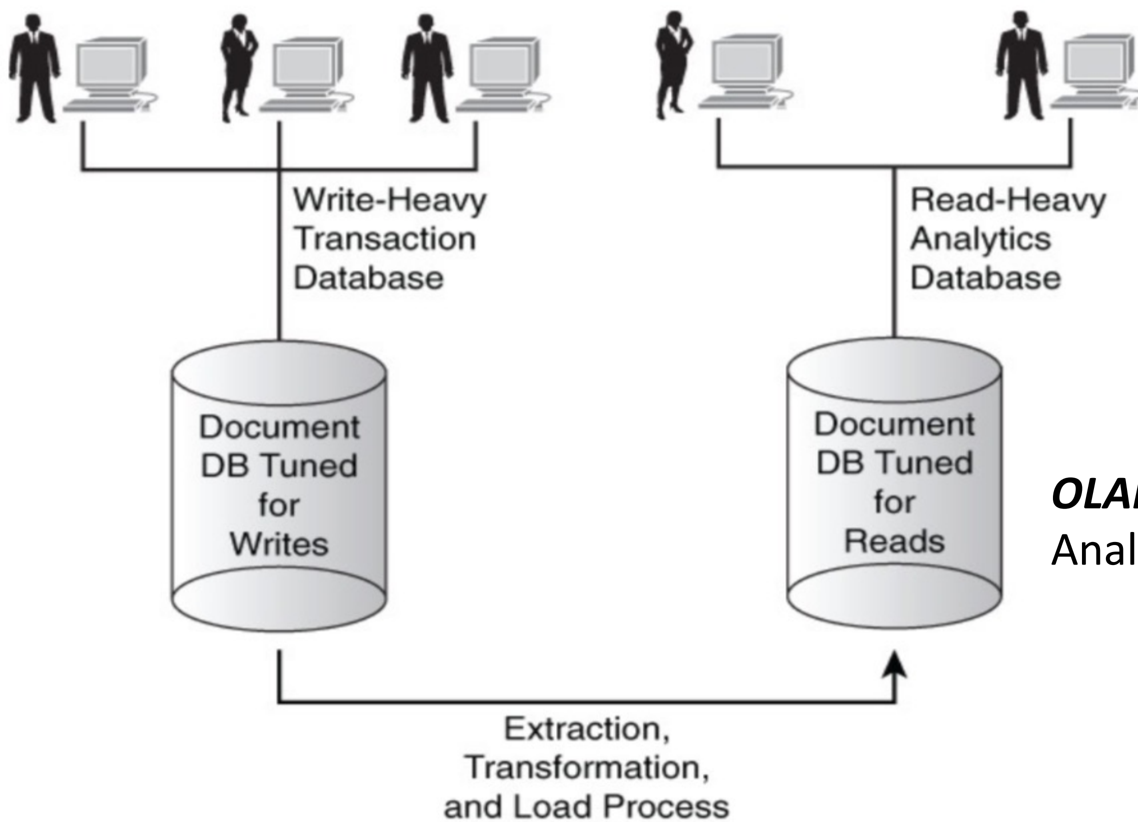
Indeed, all the ***indexes*** must be ***updated*** (and created at the beginning).

Reducing the number of indexes, allow us to obtain systems with ***fast write*** operation responses. On the other hand, we have to accept to deal with ***slow read operations***.

In conclusion, the number and the type of indexes to adopt must be identified as a ***trade-off*** solution.

Transactions Processing Systems

These systems are designed for fast write operation and targeted reads, as shown in the figure below:

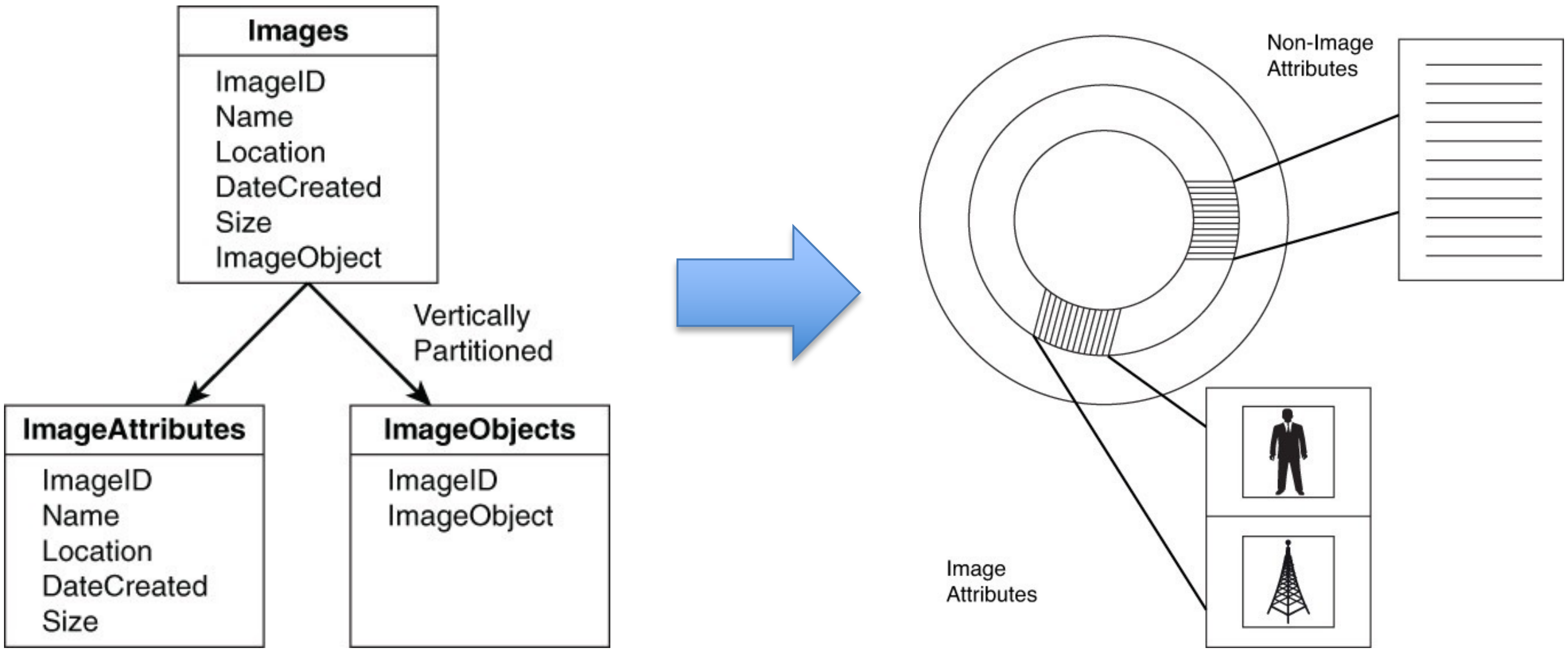


OLTP: online transaction processing

OLAP: On-Line Analytical Processing,

Vertical Partitioning

Vertical partitioning is a technique for improving (relational) database performance by ***separating columns*** of a relational table into multiple separate table.



Sharding

- ***Sharding*** or ***horizontal partitioning*** is the process of dividing data into blocks or ***chunks***.
- Each block, labeled as shard, is deployed on a ***specific node*** (server) of a ***cluster***.
- Each node can contain ***only one*** shard.
- In case of ***data replication***, a shard can be hosted by more than one node.

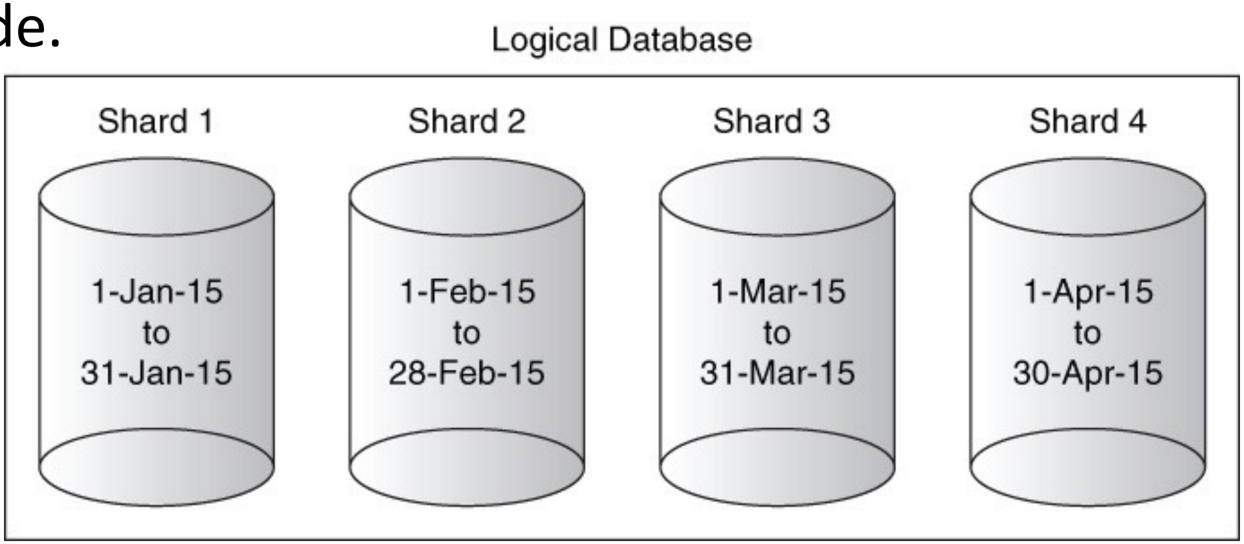


Image extracted from: "Dan Sullivan, NoSQL For Mere Mortals, Addison-Wesley, 2015"

Advantages of Sharding

- Allows handling ***heavy loads*** and the ***increase*** of system users.
- Data may be ***easily distributed*** on a variable number of servers that may be ***added*** or ***removed*** by request.
- Cheaper than ***vertical scaling*** (adding ram and disks, upgrading CPUs to a single server).
- Combined with ***replications***, ensures a ***high availability*** of the system and ***fast*** responses.

Shard Keys

To implement sharding, document database designers have to select a ***shard key*** and a ***partitioning method***.

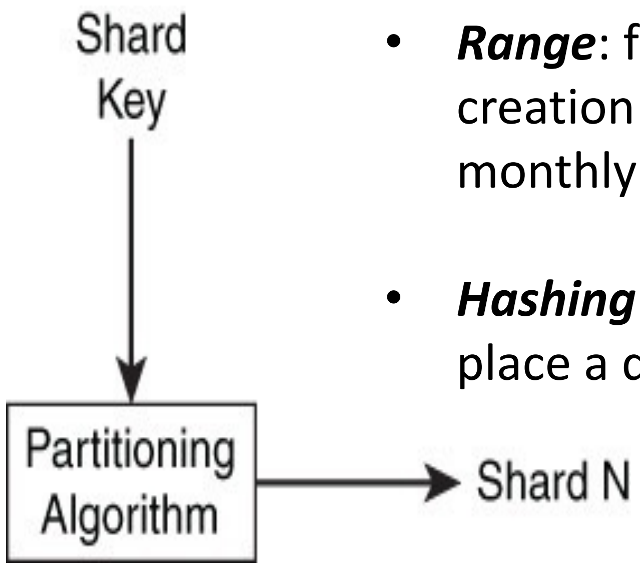
A shard key is ***one or more*** fields, that exist ***in all documents*** in a collection, that is used to separate documents.

Examples of shard keys may be: Unique ***document ID***, ***Name***, ***Date***, such as creation date, ***Category*** or type, Geographical region.

Actually, ***any atomic field*** in a document may be chosen as a shard key.

Partition Algorithms

There are three main categories of partition algorithms, based on:



- ***Range***: for example, if all documents in a collection had a creation date field, it could be used to partition documents into monthly shards.

- ***Hashing***: a hash function can be used to determine where to place a document. ***Consistent hashing*** may be also used.

- ***List***: for example, let imagine a product database with several types (electronics, appliances, household goods, books, and clothes). These product types could be used as a shard key to allocate documents across five different servers.

Suggested Readings

Chapter 4 of the book "*Guy Harrison, Next Generation Databases, Apress, 2015*".

Chapters 7,8 of the book "*Dan Sullivan, NoSQL For Mere Mortals, Addison-Wesley, 2015*"