

# 7

## Document Databases

### MongoDB

Reading: [Me] chapter 7

# NoSQL databases

NoSQL CATEGORY	EXAMPLE DATABASES	DEVELOPER
Key-value database	Dynamo Riak Redis Voldemort	Amazon Basho Redis Labs LinkedIn
Document databases	MongoDB CouchDB OrientDB RavenDB	MongoDB, Inc. Apache OrientDB Ltd. Hibernate Rhinos
Column-oriented databases	HBase Cassandra Hypertable	Apache Apache (originally Facebook) Hypertable, Inc.
Graph databases	Neo4J ArangoDB GraphBase	Neo4j ArangoDB, LLC FactNexus

# Mongo DB - Local Installation

1. Download MongoDB Community Server

<https://www.mongodb.com/try/download/community>

Installs the server and the GUI client.

2. Shell download

<https://www.mongodb.com/try/download/tools>

MongoDB Shell

3. Add path to the **bin** directory of the shell as environment variable.

4. Test: enter on command line.

mongosh -help

5. If you want to download additional mongodb tools:

<https://www.mongodb.com/try/download/database-tools>

# Document Databases

NoSQL document databases have a similar concept as key-value databases: document databases also store data as key-value pairs. The difference is that the value part consists of a "tagged document", i.e. a document in the format of a markup language, like

- JSON
- BSON
- XML

The value part is always a document. The document can contain different data types.

Related key-document pairs are stored in a collection. [collection basically refers to table in relational database](#)

In regard to data accessing possibilities, what important advantage does a tagged document have in comparison to a key-value database?

[in the document we have tags and we can search for the tags without knowing keys. But in key-value database we must know the key to get value.](#)

# Document Databases

RDBMS	Key-Value	Document
database	database / cluster	database
table	bucket	collection
tuple	key-value	document
PK / rowID	key	objectID

## Example of a MongoDB document

```
{
  "_id" : ObjectId("53e3663ccb3bd259f9252f67"),
  "type" : ["accelerator", "maker-space", "co-lab"],
  "name" : "Munich Urban Colab",
  "desc" : "In the Munich Urban Colab starters and founders find the help
            they need. Equipped with .....",
  "address" : { "str" : "Freddy-Mercury-Street 5",
                "pCode" : 80932,
                "city" : "Munich" },
  "location" : { "type" : "Point",
                 "coordinates" : [ 7.0075, 51.45902 ] }
}
```

What structural elements of the document would not be possible in a relational (normalized) table?

nested structure and embedded document

tagged document

document always starts and ends with curly brackets.

## Example of a MongoDB document

```
{
  "_id" : ObjectId("53e3663ccb3bd259f9252f67"),
  "type" : ["accelerator", "maker-space", "co-lab"],
  "name" : "Munich Urban Colab",
  "tags" : "munich colab start-up founder space",
  "desc" : "In the Munich Urban Colab starters and founders find the help they need.
           Equipped with .....",
  "address" : [
    { "street": "123 Fake Street", "floor": "15th", "pCode": "12345", "city": "Faketon"},
    { "street": "1 Some Other Street", "pCode": "12345", "city": "Boston"},
    { "str" : "Freddy-Mercury-Street 5", "postalCode" : 80932, "city" : "Munich"}
  ],
  "location" : { "type" : "Point",
                 "coordinates" : [ 7.0075, 51.45902 ]
  }
}
```

What structural elements of the document would not be possible in a relational (normalizd) table?

array of embedded documents

problem with embedded documents is that they don't have unique identifiers which makes searching more difficult.

## Example of a MongoDB document

```
{
  "_id" : ObjectId("53e3663ccb3bd259f9252f67"),
  "type" : ["accelerator", "maker-space", "co-lab"],
  "name" : "Munich Urban Colab",
  "tags" : "munich colab start-up founder space",
  "desc" : "In the Munich Urban Colab starters and founders find the help they need.
           Equipped with .....",
  "address" : [
    { "street": "123 Fake Street", "floor": "15th", "pCode": "12345", "city": "Faketon"},
    { "street": "1 Some Other Street", "pCode": "12345", "city": "Boston"},
    { "str" : "Freddy-Mercury-Street 5", "postalCode" : 80932, "city" : "Munich"}
  ],
  "location" : { "type" : "Point",
                 "coordinates" : [ 7.0075, 51.45902 ]
                }
}
```

### Relationship Cardinalities:

1:1

1:N

N:1

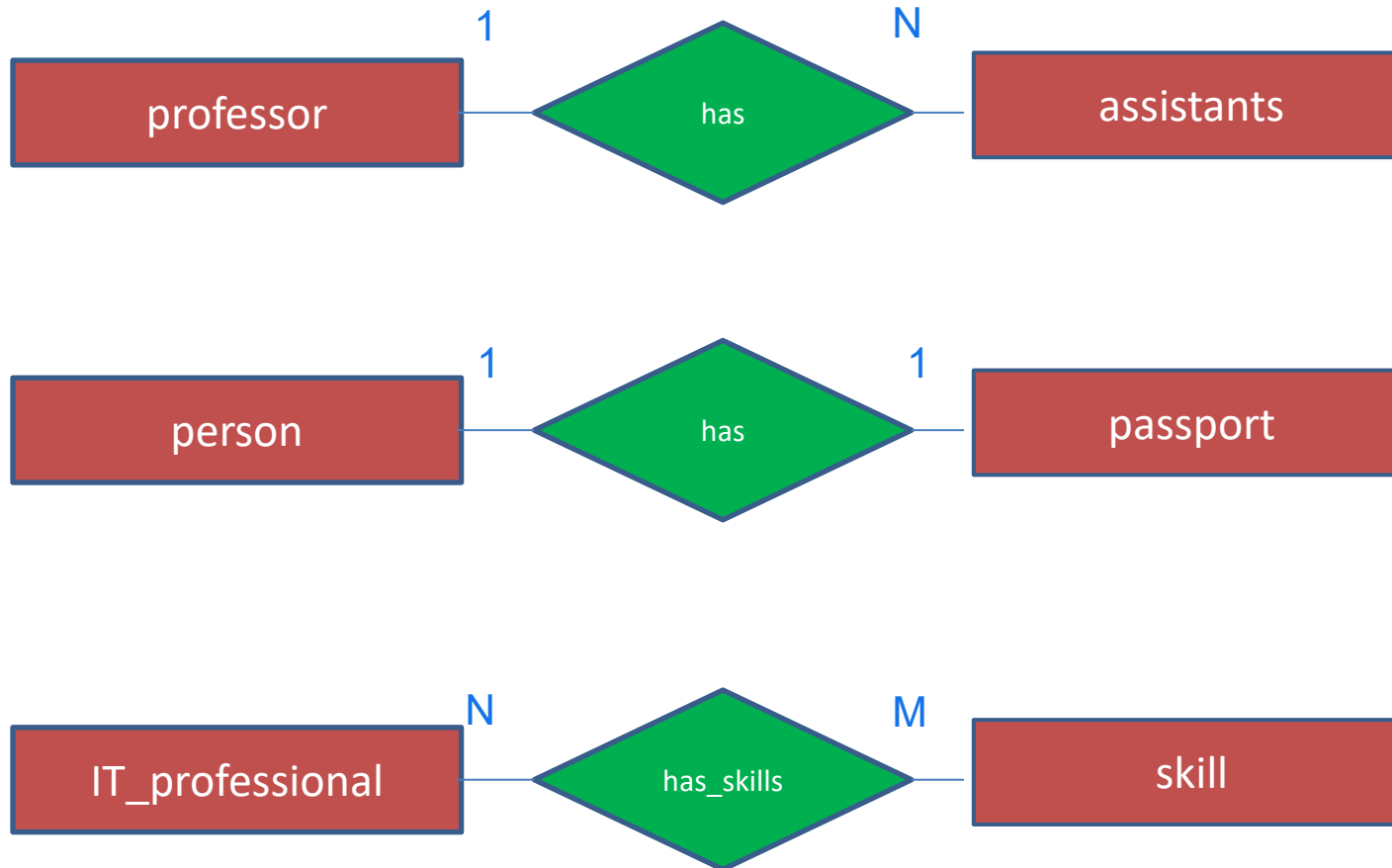
N:M

Which of these are well supported by document databases? **1:1 and 1:N**

1:1 is like one key and one value  
1:N is basically an array



## Example of a MongoDB document



Relationship Cardinalities:

1:1

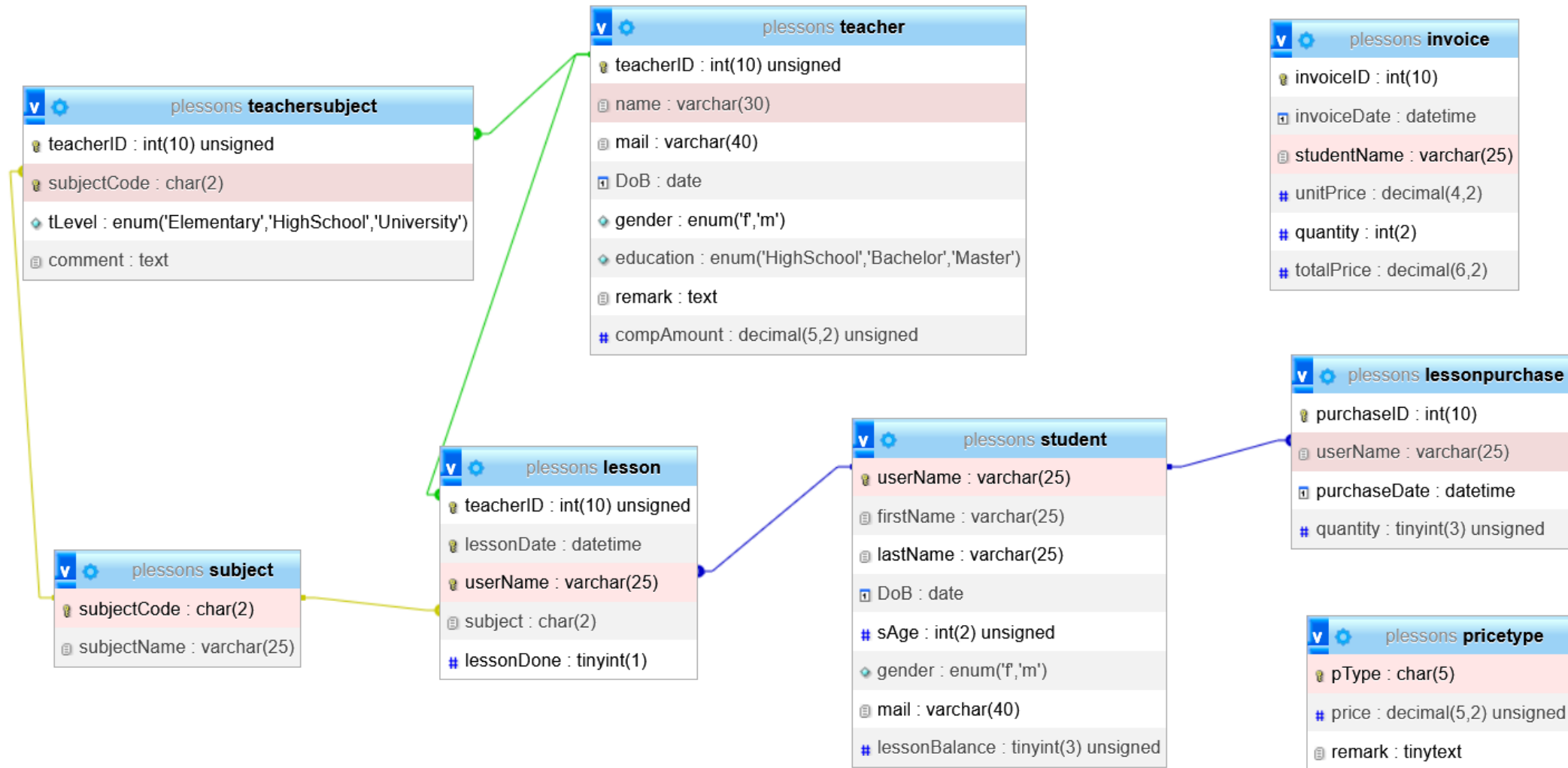
1:N

N:1

N:M

Which of these are well supported by document databases?

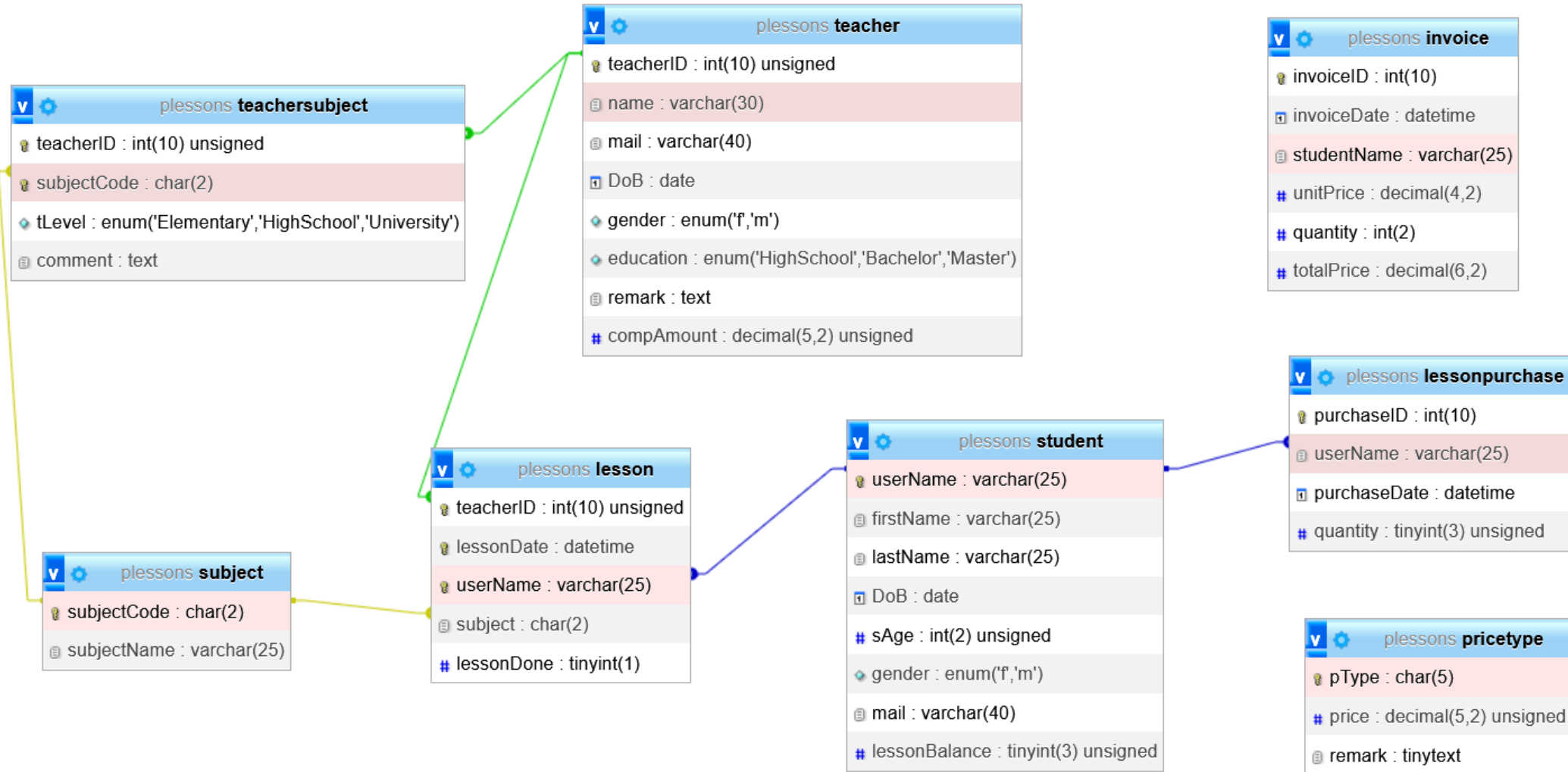
we can implement this last relationship as 1:N, we have documents and let's say each document describes IT professional, then we can have their skills as embedded document. we can query what skill does a person have but we can't query other way around.



RDBMS: Data is spread out across the tables.

Document DB: The documents are self-contained

so in documents we need to have possibly all the information because we don't have joining and referencing 10



we can't include lesson because in that table rows are constanly growing and embedded documents have a limit.

Assume that we work with teacher documents in a document db. What information could we include in the teacher table to make it self-contained?

we can integrate subject or teacherssubject, students in teacher.

# Document databases

- Via tags the contents of the documents can be queried.  
→ sophisticated and more complex queries are possible
- The "tags" provide the documents with a structure. However, there is no predefined schema. One could call it a flexible or dynamic schema, since each document can have different tags and each document can introduce new tags.
- The responsibility for modeling a data structure does not come with the database design but with the application.
- Unlike RDBMS, where data is distributed across different relations (to avoid anomalies and to map relationships), documents of a document database are self-contained and independent in terms of information content.
- The documents are not connected among each other, that is there is no relationship between documents.
- 1:1 and 1:N relationships are usually implemented within the document itself.

# Data Type Validation

- MongoDB accepts fields and data types into a document per default without validation.  
“MongoDB uses a flexible schema model, which means that documents in a collection do not need to have the same fields or data types by default. “
- If the application needs validation on data types, you need to create schema validation rules.  
“Schema validation lets you create validation rules for your fields, such as allowed data types and value ranges.”
- Schema validation rules do NOT have to be predefined. Indeed, MongoDB discourages this approach:  
“When your application is in the early stages of development, schema validation may impose unhelpful restrictions because you don't know how you want to organize your data. Specifically, the fields in your collections may change over time.”
- Schema validation rules can be defined at any later stage.
- Examples: <https://www.mongodb.com/docs/manual/core/schema-validation/specify-json-schema/>

## 12-Byte MongoDB Object ID

`_id: ObjectId("640d683998ac446f58fe75d8"),`

For each document, MongoDB creates a 12-byte object ID:

- A 4-byte timestamp, representing the objectId's creation, measured in seconds since the Unix epoch.
- A 5-byte random value generated once per process. This random value is unique to the machine (3 bytes) and process (2 bytes).
- A 3-byte incrementing counter, initialized to a random value.


Each process on every machine does its own ID generation without colliding with other MongoDB server instances. This relates well to the MongoDB distributed nature.

It is possible to override the system-generated object\_id by setting the id value manually: *we should do this when we have a really good reason to.*  
`_id: "value"`

The timestamp part can be returned with: `ObjectId.getTimestamp()`

```
lesson> ObjectId('640d683998ac446f58fe75d9').getTimestamp()
ISODate("2023-03-12T05:50:49.000Z")
```

```
lesson> db.teacher.find().sort({ _id: -1 }).limit(1)
```

 to find last one created

## Mongo DB basic commands

Command	Explanation
use <database name> Use lesson	Switches to specified database, makes the specified database the current database. Creates the database if it does not exist.
db	Displays the name of the current database
show dbs	Shows all databases that have at least one collection
db.help()	displays a list of mongo commands
db.createCollection("Name")	Creates a collection with the name "Name".
db.<collectionName>.isCapped()	shows if a collection is capped
show collections	Displays the collections of the current database
db.dropDatabase()	Deletes the current database (=deletes all collections from the database)
db.<collectionName>.drop()	Deletes the collection

# Syntax

1. Commands start with: **db.**
2. Followed by the **name of the collection** (to which the operation refers).
3. Then followed by the **method()**

Query commands refer to one collection

Examples:

`db.<collectionname>.countDocuments()`

Count documents of a collection

`db.<collectionname>.find({})`

Display all documents of a collection

`db.collection.find(query, projection, options)`

`find()` returns a cursor to the documents that match the specified query criteria. Per default, it returns the whole document. If you only want to get certain fields back, you need to add a corresponding projection.



## Adding / Removing a field to /out of collection

Removing a field out of collection:

```
plessons> db.teacher.updateMany({}, {$unset:
{payment: 1}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 24,
  modifiedCount: 24,
  upsertedCount: 0
}
```

upsertedCount means if the key exists, if it does it will be updated, if not inserted

Adding a field to collection:

```
plessons> db.teacher.updateMany({}, { $set:
{payment: "0.00"} })
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 24,
  modifiedCount: 24,
  upsertedCount: 0
}
```

## Updating Arrays with Positional Operator \$

```
db.students.insertMany( [  
  { "_id" : 1, "points" : [ 85, 80, 80 ] },  
  { "_id" : 2, "points" : [ 88, 90, 92 ] },  
  { "_id" : 3, "points" : [ 85, 100, 90 ] }  
] )
```

```
db.students.updateOne(  
  { _id: 1, points: 80 },  
  { $set: { "points.$" : 82 } }  
)
```

Collection after update?

```
{ "_id" : 1, "grades" : [ 85,82,80 ] }  
{ "_id" : 2, "grades" : [ 88, 90, 92 ] }  
{ "_id" : 3, "grades" : [ 85, 100, 90 ] }
```

The positional \$ operator identifies the element in an array to update without explicitly specifying the position of the element in the array: It updates the 1<sup>st</sup> element that matches the condition.

→ the positional \$ operator acts as a placeholder for the first element that matches the query condition

## Updating Arrays that contain embedded documents

```
{ _id: ObjectId("640d683998ac446f58fe75d8"),
  teacherID: '1',
  name: 'Krawinkel',
  students: [ {
    fName: 'Rose',
    lName: 'Singer',
    DoB: '2014-03-28',
    startLessons: '2021-01-15'
  }, {
    fName: 'Donald',
    lName: 'Black',
    DoB: '2011-05-050',
    startLessons: '2020-01-15'
  }, {
    fName: 'Donald',
    lName: 'Black',
    DoB: '2013-06-050',
    startLessons: '2020-01-15' } ] }
```

Embedded documents do not have a key.

Updating an embedded document which is an element in an array:

- use dot.notation
- use \$ positional operator

```
db.teacher.updateOne( {"students.fName":"Donald"},
  {$set: {"students.$.startLessons": "2020-03-01"}} )
```

The positional \$ operator identifies the element in an array to update: It updates the 1<sup>st</sup> element that matches the condition.

# Embedded Documents

## Syntax for inserting embedded documents:

```
db.teacher.updateOne(  
  { condition which teacher },  
  { $set : {fieldname:  
    [  
      {first student document},  
      {second student document},  
      {.....},{....}  
    ]}  
  }  
)
```

```
lesson>db.teacher.updateOne({ name:"Angel"},  
{ $set :  
{students: [  
  {fName: "Rose",lName: "Singer",DoB:"2014-03-  
28",startLessons:"2021-01-15"},  
  {fName: "Donald",lName: "Black",DoB:"2011-05-  
050",startLessons:"2020-01-15"}  
]}  
}))
```

## dot.notation for retrieving embedded documents:

```
db.<collectionName>.find( { "fieldMainDocument.fieldEmbeddedDocument": "....." } )  
db.teacher.find({"students.fName": "Rose"})  
db.teacher.find({$and: [{name: "Krawinkel"}, {"students.fName": "Rose"}]}))
```

# Limitations of the Embedding Concept

Embedding is the most natural choice for MongoDB.

MongoDB typically needs to load the entire document, even if only a small portion of it is accessed. On updates to a document, the entire document usually needs to be rewritten.

→ In regard to performance, it is better to keep documents small.

If Documents become large because

- There is a large number of embedded documents
- Embedded documents are large themselves (have many fields)

→ It may be better to separate parent and embedded documents into two collections.

If embedded documents are update-and-write heavy

→ It may be better to separate parent and embedded documents into two collections.

If there are regular queries on the embedded documents only

→ It may be better to separate parent and embedded documents into two collections because of more flexibility in performing queries.

# Embedded or Referenced Documents

Embedded documents **if we separate them we need referencing**

are stored inside a parent document. Whenever a parent document is retrieved, all its embedded documents are also retrieved.

## Referenced documents

- are stored in a separate collection. One can retrieve them separately.
- They reference each other with the help of the \$ref notation:  
`{"$ref": "collectionName1", "$id": ObjectId("89aba98c00a") }`
- This reference is only notational!

MongoDB does not support the join concept of FK-PK links. So, **if we want to retrieve a referenced document, we must effectively run 2 queries:**

1. A first query reads the referenced ID value from one document and with this retrieved ID
2. A second query is needed to retrieve the data from the referenced document.

# Reference Concept

Let us assume that we need a student focused approach, as well. We could split our data into teacher collection and student collection and reference teachers from the student collection.

student collection:

```
{
  _id: ObjectId("6415523d0ed201caee1a9e5e"),
  userName: 'Wolf',
  firstName: 'Tom',
  lastName: 'Salcher',
  DoB: '2013-02-20',
  gender: 'm',
  mail: 'tom.salcher@postbox.xx',
  lessonBalance: 0
}
```

```
db.student.updateOne({userName: "Wolf"},
{$set: {teachers: [
{"$ref": "teacher",
"$id": ObjectId("5126bc054aed4daf9e2ab772")}
]
}
})
```

# Reference Concept

Let us assume that we need a student focused approach, as well. We could split our data into teacher collection and student collection and reference teachers from the student collection.

student collection:

```
{
  _id: ObjectId("6415523d0ed201caee1a9e5e"),
  userName: 'Wolf',
  firstName: 'Tom',
  lastName: 'Salcher',
  DoB: '2013-02-20',
  gender: 'm',
  mail: 'tom.salcher@postbox.xx',
  lessonBalance: 0
}
```

student collection with reference to teacher collection:

```
{
  _id: ObjectId("6415523d0ed201caee1a9e5e"),
  userName: 'Wolf',
  firstName: 'Tom',
  lastName: 'Salcher',
  DoB: '2013-02-20',
  gender: 'm',
  mail: 'tom.salcher@postbox.xx',
  lessonBalance: 0,
  teachers : {
    "$ref" : "teacher",
    "$id" : ObjectId("5126bc054aed4daf9e2ab772")
  }
}
```



# Reference Concept

Student collection with references to teacher collection

student document "Darthvader"

```
db.student.updateOne({userName: "Darthvader"},  
{$set: {teachers: [{ "$ref": "teacher", "$id": ObjectId("640d683998ac446f58fe75ea") } ]}})
```

Querying:

1. Reading the referenced objectId out of student document:

```
db.student.find({userName: "Darthvader"}) #returns objectId of references teacher,  
ObjectId("640d683998ac446f58fe75ea")
```

2. Querying teacher collection using returned objectId

```
db.teacher.find(ObjectId("640d683998ac446f58fe75ea"))
```

Inserting multiple references: Student document "Rose" (takes lessons with two teachers):

```
db.student.updateOne({userName: "Rose"},  
{$set: {teachers: [{ "$ref": "teacher", "$id": ObjectId("640d683998ac446f58fe75ea") },  
{ "$ref": "teacher", "$id": ObjectId("640d683998ac446f58fe75de") } ]}})
```

# SQL versus Document Queries

Given: Relational PostgreSQL course database lessons and MongoDB collection teacher with embedded student documents

Return all teachers:

SQL: `select * from teacher`

MongoDB: `db.teacher.find()`

Return all students of teacher 1:

SQL: `select distinct username from lesson where t_id=1`

MongoDB: `db.teacher.find( {t_id: 1}, {student.username: 1})`

Return all teachers of student Donald:

SQL: `select distinct t_id from lesson where username="Donald"`

MongoDB not well supported because we might have multiple Donalds and it just matches the first one so NOT SUPPORTED

# SQL versus Document Queries

Given: Relational PostgreSQL course database lessons and MongoDB collection teacher with embedded student documents.

Return all students:

SQL: `select * from student`

MongoDB: doable but not well supported because we would get all the teacher documents and if we limit it to see only embedded students than we would have duplicates.

Return all teachers that teach English

SQL: `select * from teacher_subject where subject="eng"`

MongoDB: it is possible but problem here is that in SQL we have defined how subjects are spelled, meaning EN, eng, English... and we know what we are looking for but in MongoDB this could be all. So firstly we would need to check what values we have here with OR operator or we can do validation to only allow specific spelling.

## Capped Collection = Collection with fixed size limit

Capped collections are collections with a fixed, predetermined size limit. The maximum size is defined in bytes and / or the maximum number of documents allowed. If the collection has reached the predefined size / prefixed number of documents, the oldest documents are simply overwritten.

Advantages:

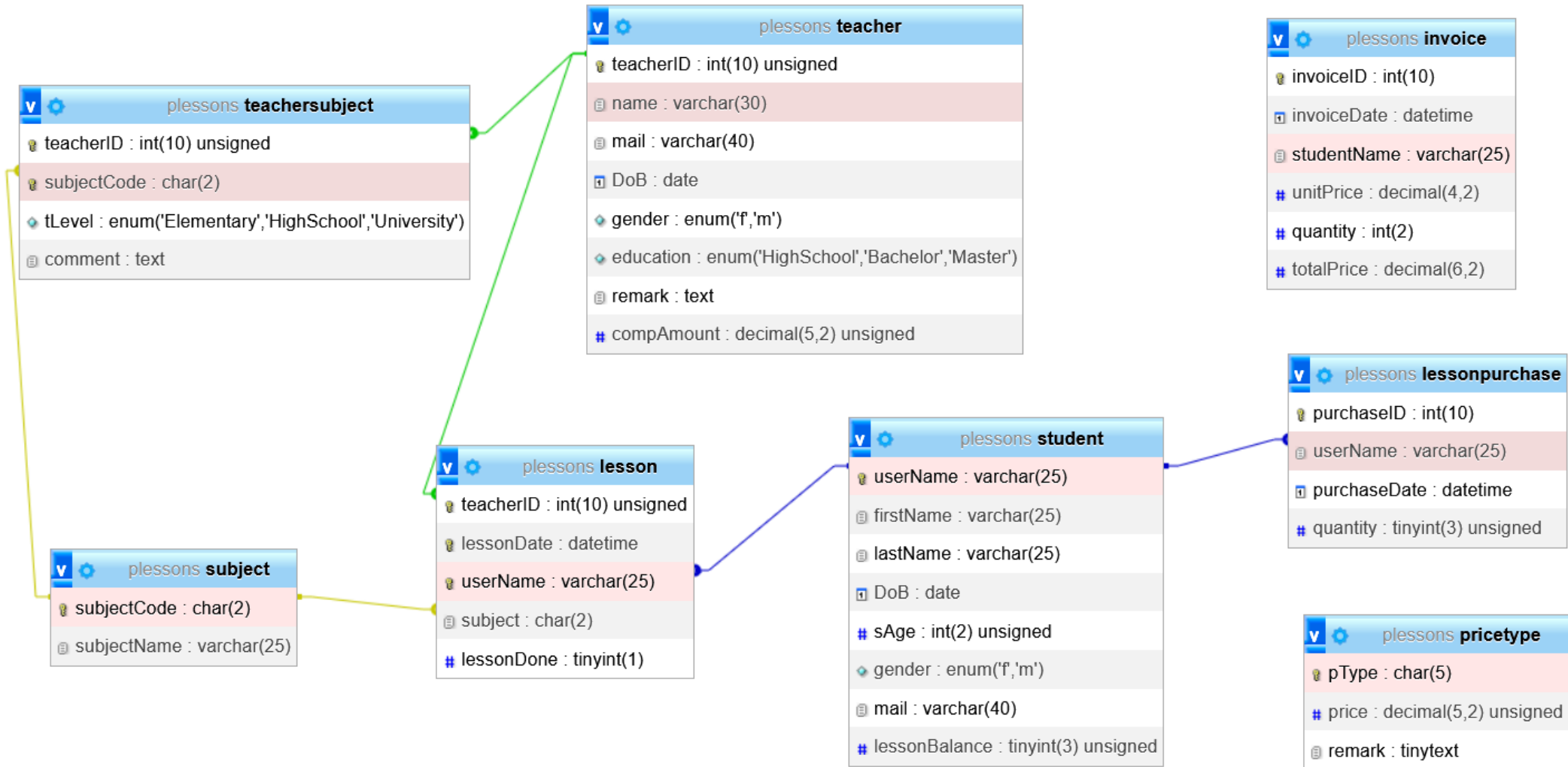
- Capped collections are sorted according to the insertion time.
- Automatic removal / overwriting of old documents (can be an advantage depending on the use case)
- Very suitable if documents (contents) are stored only for a certain period of time and then should be deleted automatically.

Capped collections cannot be distributed to shards.

Application scenario: log files

Commands:

<code>db.createCollection("log", { capped : true, size : 5242880, max : 5000 } )</code>	<code>#creates a capped collection</code>
<code>db.&lt;collection name&gt;.isCapped()</code>	<code>#checks if a collection is capped.</code>



How could we remodel the database and business case (!) to work well with a document database? [platform for teachers where they can advertise themselves, students can search this platform, teacher pay monthly fee](#)

# Use Cases for Document Databases

if the business idea / application

- can be well represented with document-like, self-contained objects.
- It relies mainly on 1:1 or 1:N relationships.
- It does not need many joins. (Support for joins is weak.)
- It profits from better locality (that is: all relevant information is ideally in one document).
- It needs schema flexibility.

Use Cases: . Catalog applications, content management systems, web applications

# MongoDB data types

string	strings are utF-8
Integer	Can be 32-bit or 64-bit
Double	to store floating-point values
arrays	to store a list of values into one key
Timestamps	For MongoDB internal use; values are 64-bit record when a document has been modified or added
Date	a 64-bit integer that represents the number of milliseconds since the unix epoch (January 1, 1970)
objectId	unique and ordered Consist of 12 bytes, where the first four bytes are a timestamp that reflects the objectId's creation
Binary	data to store binary data (images, binaries, etc.)
zero	to store NULL value

<https://docs.mongodb.com/manual/reference/bson-types/>

Command	Description
\$regex	Match by any PCRE-compliant regular expression string (or just use the // delimiters as shown earlier)
\$ne	Not equal to
\$lt	Less than
\$lte	Less than or equal to
\$gt	Greater than
\$gte	Greater than or equal to
\$exists	Check for the existence of a field
\$all	Match all elements in an array
\$in	Match any elements in an array
\$nin	Does not match any elements in an array
\$elemMatch	Match all fields in an array of nested documents
\$or	or
\$nor	Not or
\$size	Match array of given size
\$mod	Modulus
\$type	Match if field is a given datatype
\$not	Negate the given operator check



Command	Description
\$set	Sets the given field with the given value
\$unset	Removes the field
\$inc	Adds the given field by the given number
\$pop	Removes the last (or first) element from an array
\$push	Adds the value to an array
\$pushAll	Adds all values to an array
\$addToSet	Similar to push, but won't duplicate values
\$pull	Removes matching values from an array
\$pullAll	Removes all matching values from an array