POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Systems and Methods for Big and Unstructured Data Project

Author(s): **Fabio Lusha - 10882532**

Enrico Simionato - 10698193

Irfan Cela - 10694934

Bianca C. Savoiu Marinas - 10684465

Alberto Sandri - 10698469

Group Number: **31**

Academic Year: 2022-2023

Contents

Part I

Introduction

1 | Introduction to the problem

1.1. Problem description

Scientific research is the engine that leads us through the future providing the technology to improve every single detail of our life. Nowadays the speed at which humanity and technology advance is outstanding thanks to scientists' progress in their fields of study. The reaching of new results in the research materializes through scientific articles. Since the advent of the World Wide Web, publishers started to post their papers on the net reaching an audience far larger than before. This has allowed a more convenient and faster way to share knowledge, therefore boosting academic research and consequently the publishing of new papers. Researchers need to navigate and access this massive quantity of knowledge, which is growing every day, thus, if we want to bolster human progress, new methods of organizing and managing these data are mandatory.

1.2. Purpose

The purpose of the project is to create a bibliographic database, namely a system able to store and manage data regarding scientific articles and all their meaningful characteristics and relations with authors and the place where they are published.

1.3. Assumptions and important concepts

The problem we are addressing is very wide, and given the heterogeneity of the data different assumptions can be made. We shaped our database by taking into consideration the following aspects:

- A scientific paper can be written by one or more authors;
- All the authors of a paper are considered to be at the same level, so the position of coauthor is not considered;
- The papers the system considers were published in journals, books or were presented

at conferences;

- A paper can appear in just one publication among journals, books, and conferences;
- A single affiliation of an author can be registered when a paper is written;
- Papers can have multiple fields of study (fos) and keywords;
- Not every paper has to be related to a field of study;
- Not every paper has to have related keywords;
- The name of the author may be complete or abbreviated;
- A paper may or may not reference other papers or be referenced by others.

2 | ER model

As a bibliographic database, the main focus is on scientific articles, also called scientific papers, that can be considered the central entity. Every article is written by one or more authors and each writer is associated with an organization. Articles have several features like title, year of publication, language, abstract, DOI, and URL and can have many keywords and can be related to many fields of study. Furthermore, a paper can reference other papers, and each article is published in a book, in a journal, or presented during a conference.

The place where an article is published is called in a general way *publication* and each one of them has different properties: conferences take place in a specific location, books have an ISBN and a publisher, and journals have an ISSN, volume, issue and a publisher. So the `Publication` entity is extended with a hierarchy that is total and exclusive: a node has to be instantiated as one of the specific types of publication and cannot be more than one type at a time.

The group opted for a straightforward model with only the core concepts and relationships to make it easily understandable and represent it concisely. Of course, the problem domain is very wide and complex but in this modeling phase, it was taken into account also the available data to build the model, trying to leave it quite general so that it would be easier to modify it and make it more complex in further versions.

In the following ER model, we chose to represent the cardinalities of the relationships always starting from one, so we don't consider entities that are not related to others. When adding data to the database we can have, in an initial state, some isolated nodes, but in the ER we show a stable and correct state of the system.

As we can see from the diagram, we use in general the attribute `id` as an identifier of the entities, but in some cases in the real world, there are better options, for instance, the `Book` might be identified by the attribute `ISBN`. We made this assumption because our dataset was not complete, and a lot of `Book` entities didn't have this property. The same reasoning can be applied to `Paper` with `DOI`, `Journal` with `ISSN`, and `Author` with `ORCiD` that we disregarded because not present in the used data file.

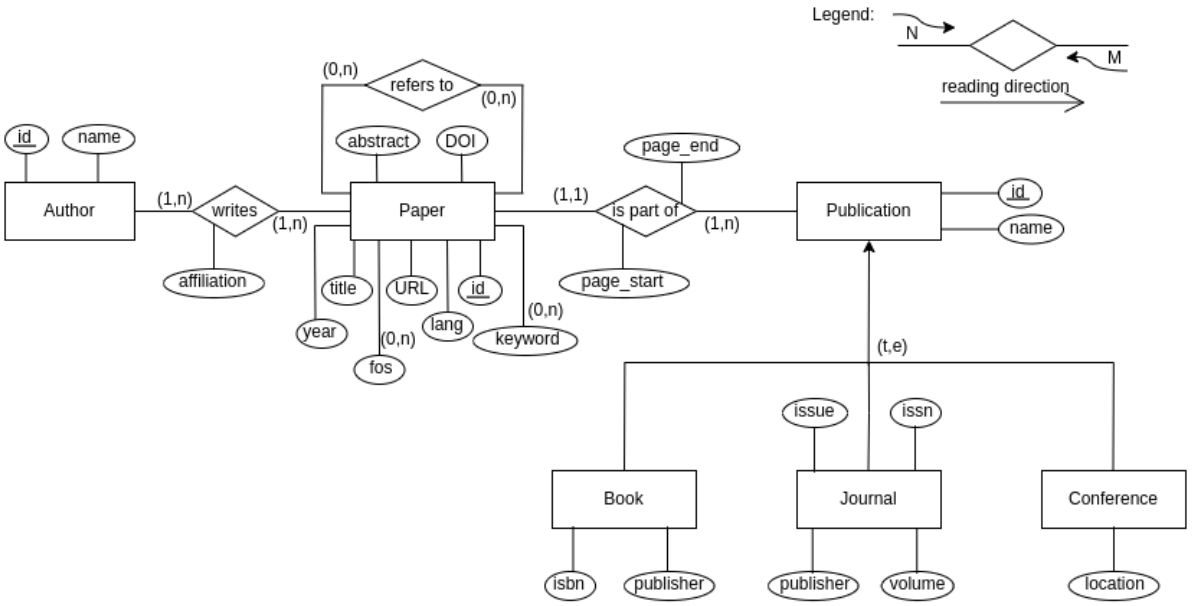


Figure 2.1: ER model of the bibliographic database.

More detailed description of entities

- **Paper**, as said before, is the most important entity, indeed it is in the middle of the model. Each paper is characterized by an ID, a title, a publication year, an abstract, a list of keywords, a list of fields of study, a DOI which is a unique global identifier, and a URL to reach the site where it is available for consultation.
- **Author** of a paper is identified by an ID and has a name, that contains both the name and the surname of the author.
- **Publication** is a total and exclusive hierarchy that represents the physical place where a paper is published, it has an ID and a name called the venue, and it is specialized in:
  - **Conference** that has a location where it took place;
  - **Book** that has a global identifier called ISBN and a publisher;
  - **Journal** that has a global identifier called ISSN, a publisher, a volume and issue.

More detailed description of relationships

- **WRITES** links Author and Paper, each author can write one or more papers, and each paper is written by one or more authors. This relationship has an attribute affiliation that is the organization the author was part of when the article was written.
- **REFERS TO** links two Paper entities, each paper can reference many papers and can be referenced by many papers.
- **IS PART OF** links Paper and Publication, each paper is published on a single publication and each publication can contain several papers. This relationship has as attributes the page_start and page_end of the article within the publication.

More detailed observations

- DOI could have been a good identifier for papers since it is globally unique, but in the used dataset not all articles had it.
- ORCiD is a unique identifier for authors that would have been a valid key for the entity Author but this data was not present in the used dataset.
- Since the focus is on the entity Paper and a simple model for the domain was built, it seemed not meaningful to have a separate entity to model the organization whose author is part of. In addition, that would have had just one attribute. For this reason, the concept of affiliation was modeled as an attribute of the relationship WRITES, storing multiple affiliations for the single author.

Part II

Neo4j

3 | Dataset description

The main dataset used can be downloaded from the site www.aminer.org, in particular, the latest version of data was employed, namely DBLP-Citation network V13. The dataset was not entirely imported in Neo4j because it was huge and our machines with limited computing power were not able to handle it, so just a part of it was taken, containing information about 2315 papers.

The following table shows the dataset's fields that were used, associated with their respective meaning, name, and role in the created database. The fields publisher and location were not present in the original dataset, that's why we created this information from scratch, but this aspect is further discussed in the following chapter.

Field name	Type	Description	Name in Neo4j	Usage in Neo4j
_id	String	paper ID	id	node Paper
title	String	paper title	title	node Paper
year	Integer	paper year	year	node Paper
lang	String	paper language	lang	node Paper
doi	String	paper DOI	doi	node Paper
url	String	paper URL	url	node Paper
abstract	String	paper abstract	abstract	node Paper
keywords[i]	String	paper keyword	keyword	node Keyword
fos[i]	String	paper field of study	fos	node Fos
references[i]	String	paper reference	REFERENCES	relation REFERENCES
page_start	Integer	start page	page_start	attribute IS_PART_OF
page_end	Integer	end page	page_end	attribute IS_PART_OF
authors._id	String	author ID	id	node Author
authors.name	String	author name	name	node Author
authors.org	String	author affiliation	affiliation	attribute WRITES
venue.raw	String	publication name	name	node Publication
isbn	String	book ISBN	isbn	node Book
publisher	String	publisher name	publisher	node Book/Journal
issn	String	journal ISSN	issn	node Journal
volume	String	journal volume	volume	node Journal
issue	String	journal issue	issue	node Journal
location	String	conference location	location	node Conference

Table 3.1: Description of the data used in the project and relationship between original dataset and imported data.

4 | Data upload

4.1. Pre-processing

Before uploading the data in Neo4j, the dataset was downloaded from the website mentioned above, and just a part of it has been extracted. Then some ad-hoc Python scripts have been used to do a first-level cleaning of the data since we found some inconsistency in it. Specifically, the cleaning involved the following aspects:

- Some numerical attributes were wrapped in NumberInt(...) string. This is not a standard way to store numeric values in JSON so it couldn't be parsed by the Neo4j JSON parser. We unwrapped all numeric values stored this way, leaving only the actual value;
- Some of the ISSN attributes did not conform to the standard and correct pattern, so the wrong ones have been eliminated;

After that, to distinguish the type of publication, a Python script was made to infer it based on the attributes of the paper: if it has an ISBN is a Book, if it has an ISSN, volume or issue is a Journal otherwise it is a Conference.

To have a more complete domain it has been decided to add the fields publisher to Book and Journal, and location to Conference. These values were taken from the web and added randomly to the papers using a Python script.

Lastly, since just a subset of the dataset has been used, there were very few connections among the nodes, so, thanks to other Python scripts, some values were added randomly to the fields keyword, fos, and references of each paper such that the graph is more connected and more meaningful queries can be performed.

4.2. Upload process

The JSON file called `dataset.json` was used to upload the data, it was put in the `import` folder of Neo4j, and then with the following commands, the data were imported

using cypher-shell.

To be sure that at the beginning the database is empty, all possible parameters, nodes, and relationships are deleted with this command.

```
:params {};
MATCH(n) DETACH DELETE n;
```

To use the importing commands it is first necessary to install the `apoc` library and emulate the following steps to be able to use the `apoc.load.json` with which we read the dataset file. For security reasons, procedures that use internal APIs are disabled by default. They can be enabled by specifying config in `NEO4J_HOME/conf/neo4j.conf` e.g. `dbms.security.procedures.unrestricted=apoc.*`.

You can also whitelist procedures and functions, in general, to be loaded using

`dbms.security.procedures.whitelist=apoc.coll.*,apoc.load.*`

We are particularly interested in the last one.

1. **Create Paper nodes with their attributes, create Fos and Keyword nodes**
   The following command creates all nodes with label `Paper`, setting the attributes `id`, `title`, `year`, `lang`, `doi`, `url`, `abstract`; then it creates the nodes with label `Keyword` requiring that the value is not null, moreover `Paper` and `Keyword` are linked with the relationship `HIGHLIGHTS`. The same thing is done for `Fos` nodes, that are linked with `Paper` nodes through `BELONGS_TO` relationship.

```
CAll apoc.load.json('dataset.json') YIELD value
CREATE (p:Paper {id:value._id, title:value.title, year:value.year,
    lang:value.lang, doi:value.doi, url:value.url, abstract:value.
    abstract})
WITH p, value
UNWIND value.keywords AS kw
WITH p, value, kw WHERE kw IS NOT NULL
MERGE (k:Keyword {keyword:kw})
MERGE (p)-[h:HIGHLIGHTS]->(k)
WITH p, value
UNWIND value.fos AS fos
WITH p, value, fos WHERE fos IS NOT NULL
MATCH (p:Paper {id:value._id})
MERGE (f:Fos {fos:fos})
MERGE (p)-[b:BELONGS_TO]->(f)
```

2. **Create Author nodes and the relationship WRITES**
   The following command creates all nodes with label `Author` requiring that `id` and

`name` attributes are not null. Next, all `Author` nodes are linked with their `Paper` nodes using the `WRITES` relationship, assigning the property `affiliation` when this information is present.

```
CALL apoc.load.json('dataset.json') YIELD value
UNWIND value.authors AS aut
WITH aut, value
WHERE aut._id IS NOT NULL AND aut.name IS NOT NULL
MERGE (a:Author {id:aut._id, name:aut.name})
WITH a, aut, value
MATCH (p:Paper {id:value._id})
CREATE (a)-[w:WRITES {affiliation:aut.org}]->(p)
```

3. **Create Book nodes**

   The nodes with label `Book` are created, setting also the label `Publication`, and these entities have the attributes `name` and `publisher`. The command also links `Paper` nodes to `Book` nodes using the `IS_PART_OF` relationship, adding to it the properties `page_start` and `page_end`. We also allow these attributes to be null, so we don't check their presence when adding them, because in graph databases we can have heterogeneous nodes' structures without any problem and this permits us to explore the peculiarity and strength of this technology.

```
CALL apoc.load.json('dataset.json') YIELD value
UNWIND value.venue AS ven
WITH value, ven
WHERE value.publication_type = "Book"
CREATE (b:Publication:Book {isbn:value.isbn, publisher:value.
   publisher, name:ven.raw})
WITH b, value
MATCH (p:Paper {id:value._id})
CREATE (p)-[i:IS_PART_OF {page_start:toInteger(value.page_start),
   page_end:toInteger(value.page_end)}]->(b)
```

4. **Create Conference nodes**

   The nodes with label `Conference` are created, setting also the label `Publication`, and `name` and `location` attributes of the conference. Then `Paper` nodes are linked with `Conference` nodes using the `IS_PART_OF` relationship setting the properties `page_start` and `page_end`.

```
CALL apoc.load.json('dataset.json') YIELD value
UNWIND value.venue AS ven
WITH value, ven
WHERE value.publication_type = "Conference"
```

```
CREATE (c:Publication:Conference {name:ven.raw, location:value.
    location})
WITH c, value
MATCH (p:Paper {id:value._id})
CREATE (p)-[i:IS_PART_OF {page_start:toInteger(value.page_start),
    page_end:toInteger(value.page_end)}]->(c)
```

5. **Create Journal nodes**

   The nodes with label `Journal` are created, setting also the label `Publication` and its properties `name`, `publisher`, `issn`, `volume` and `issue`. Then `Paper` nodes are linked with `Journal` nodes using the `IS_PART_OF` relationship setting the properties `page_start` and `page_end`.

```
CALL apoc.load.json('dataset.json') YIELD value
UNWIND value.venue AS ven
WITH value, ven
WHERE value.publication_type = "Journal"
CREATE (j:Publication:Journal {issn:value.issn, publisher:value.
    publisher, name:ven.raw, volume:value.volume, issue:value.issue
    })
WITH j, value
MATCH (p:Paper {id:value._id})
CREATE (p)-[i:IS_PART_OF {page_start:toInteger(value.page_start),
    page_end:toInteger(value.page_end)}]->(j)
```

6. **Create REFERENCES relationship**

   With the following command, the `REFERENCES` relationship between `Paper` nodes are created.

```
CALL apoc.load.json('dataset.json') YIELD value
UNWIND value.references AS ref
WITH value,ref
WHERE ref IS NOT NULL
MATCH (a:Paper {id:value._id})
MATCH (b:Paper {id:ref})
MERGE (a)-[r:REFERENCES]->(b)
```

Label	Quantity
Paper	2315
Author	3546
Fos	155
Keyword	4343
Book	372
Conference	612
Journal	1315
Total	12658

Table 4.1: Summary with node quantity for each label.

5 | Graph diagram

In the realization process from the ER model to the graph diagram, some changes have been made to better exploit the features of Neo4j, prioritizing connections between data. For this reason nodes with the following labels are present in the graph:

- **Paper** as it is the central entity, has a lot of connections to the other nodes, and its possible attributes are `id`, `title`, `year`, `lang`, `doi`, `url`, and `abstract`;
- **Author** has `id` and `name` as attributes;
- **Keyword** contains a single attribute `keyword`;
- **Fos** contains a single attribute `fos` that represent the field of study;
- **Publication** has the attribute `name` and it can also be associated with one of the following labels:
  - **Book** contains `isbn` and `publisher`;
  - **Conference** contains a `location`;
  - **Journal** contains `publisher`, `issn`, `volume`, `issue`.

Concerning the relationships:

- **REFERENCES** links a `Paper` that references another `Paper`, the connection is made by using the papers' `id`;
- **WRITES** links an `Author` to a `Paper`, it could also contains the `affiliation` attribute;
- **HIGHLIGHTS** links a `Paper` to a `Keyword`;
- **BELONGS_TO** links a `Paper` to a `Fos` ;
- **IS_PART_OF** links a `Paper` to a `Publication`, it could also contain the attributes `page_start` and `page_end` .

5.1. Differences concerning the ER model

`Keyword` and `Fos` were multi-valued attributes in the ER diagram and have been mapped as nodes in the graph. This choice was made since these values are shared among multiple `Paper` nodes creating many connections. In addition, the efficiency increases because when a query is performed just the record files about nodes and relationships need to be uploaded in main memory instead of the file with the properties, that would have been required if these fields were treated as attributes of the `Paper` node; so the file with the properties is not needed when querying these elements and this allows to use the main feature of Neo4j, which is the faster retrieval of data using relationships between nodes, that are a natural embedded structure of the graph databases.

To also take advantage of the fact that multiple labels can be assigned to a single node, the nodes with labels `Book`, `Journal` or `Conference` all have the label `Publication` too. This allows us to easily perform queries involving a generic publication and also to further extend the database with new types of publications, such as a thesis.

When importing the data in the graph database we encounter some differences from the ER diagram, because we don't always have the attributes that were assumed in the modeling part of the database. While importing the data, for some elements we checked if their properties were null before adding them, thus, in Neo4j we don't have those attributes at all. Otherwise, even if an element can be considered null because, for instance, it has an empty string, it will be added anyway to the respective node. We decided to avoid further checks on these attributes to have a faster importing process.

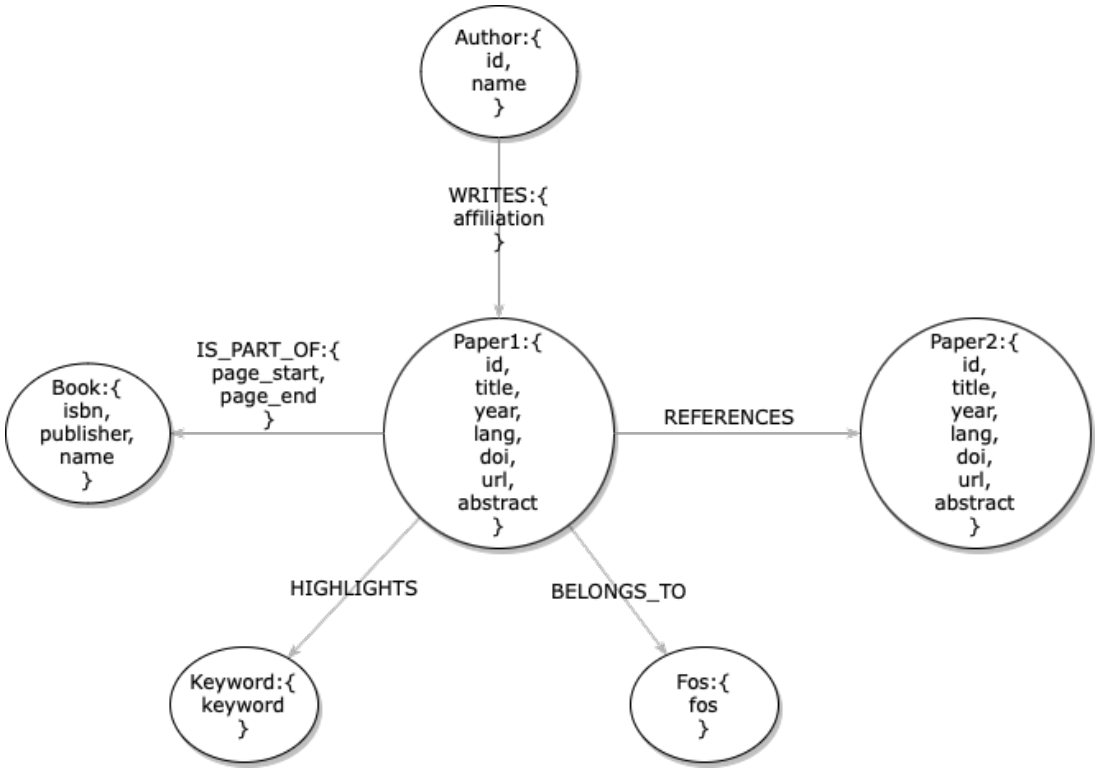


Figure 5.1: Graph diagram of a Paper that references another one, it has a single Author, a Keyword, a Fos and it is part of a Book.

6 | Commands and queries

The following commands and queries are written thinking principally on the way the database will be used. We think a user with the necessity to upload or search for something in the system, will be able to do that with what we provide. Longer and more difficult commands and queries can be written but we want to present only the ones that we find more meaningful.

We choose to use parameters instead of writing fixed values inside the queries to make the code as flexible as possible. For this reason, the commands and queries can be also used changing the values of the parameters and obtaining the results of interest. To execute the following code is required to run first the parameters regarding the command or query we want to perform and then the command or query itself.

6.1. Commands

The following commands are provided to make possible the update of the data. Each command can be easily modified to update parts of the database with specific features, for example, it is possible to use the queries of the next section to update the obtained nodes and their attributes.

1. **Create a node, its attributes, and some relationships**
   The command creates a set of new nodes. In particular, the query aims to add a new paper to the database, whose author is already present within the system, and also set its properties. Here we add a new paper, set also its attributes, and create the nodes of the keywords and the fields of study of the paper if needed. The `MERGE` command is what is used to create nodes and relationships checking also whether the new instances are already present within the database. This command allows us to avoid the creation of many copies of the same entity. The following command after adding the specified nodes creates also the relationships between them: the ones that relate the paper to its author, to its keywords, and to the specific study.

   Parameters

```
:param auth_id => "53f463b3dabfaee4dc8430d5";
:param auth_name => "Annabel Sebag";
:param affiliation => "53f463b3dabfaee4dc8430d5";

:param doi => "10.1145/1596685.1596829";
:param paper_id => "53e997cbb7602d9701fbcee3";
:param paper_title => "Yankee gal";
:param year => 2009;
:param lang => "en";
:param page_start => 155;
:param page_end => 155;
:param url => ["http://dx.doi.org/10.1145/1596685.1596829","http://
   doi.acm.org/10.1145/1596685.1596829","db/conf/siggraph/
   siggraph2009festival.html#Sebag09c","https://doi.org
   /10.1145/1596685.1596829"];
:param abstract => "This is a graduate film from the students of
   Supinfocom Valenciennes.";

:param k1 => "yankee gal";
:param k2 => "supinfocom valenciennes";
:param k3 => "graduatethe query aims"animation";
:param fos2 => "siggraph";
```

Command

```
MATCH (a:Author {id:$auth_id, name: $auth_name})
MERGE (new_p:Paper
        {doi:$doi,
        id:$paper_id,
        title:$paper_title,
        year:$year,
        lang:$lang,
        page_start:$page_start,
        page_end:$page_end,
        url:$url,
        abstract:$abstract})
MERGE aff = (a)-[:WRITES {affiliation:$affiliation}]->(new_p)
MERGE f1 = (new_p)-[:BELONGS_TO]->(:Fos {fos: $fos1})
MERGE f2 = (new_p)-[:BELONGS_TO]->(:Fos {fos: $fos2})
MERGE k1 = (new_p)-[:HIGHLIGHTS]->(kw1:Keyword {keyword:$k1})
MERGE k2 = (new_p)-[:HIGHLIGHTS]->(kw2:Keyword {keyword:$k2})
MERGE k3 = (new_p)-[:HIGHLIGHTS]->(kw3:Keyword {keyword:$k3})
RETURN aff, f1, f2, k1, k2, k3
```

2. **Create the relationship between existing nodes and set its properties**

The command creates a new relationship between already existing nodes. In particular here is created the relationship which states that the author `"James Ostell"` has written the paper titled `"Grow"` when he was affiliated to organization `"Federal Institute of Technology, Switzerland"`. For better identification of the book and for avoiding linking all the books having the same title we could have used the identifier of the book instead of the title. We used `MERGE` so if the relationship already exists with the specified affiliation it won't be added.

Parameters

```
:param name => "James Ostell";
:param title => "Grow";
:param affiliation => "Federal Institute of Technology, Switzerland
    ";
```

Command

```
MATCH (a:Author {name:$name}), (np:Paper {title:$title})
MERGE res = (a)-[:WRITES {affiliation:$affiliation}]->(np)
RETURN res;
```

3. **Set a new label for a node**
   We want to modify the properties of an already existing node. In particular, we want to add one label to a specific node. At first, we have new entries that are sparse data and we don't know the exact type and source so we want to infer it, in particular, we check if they have the `isbn` and in that case, we update the data, setting the nodes as books. We do this by controlling the papers and using the `SET` keyword to update the properties, and if there are some misaligned data already in the graph this corrects that too. In the command, we can return the nodes in order to be sure that the information was updated correctly. We could do the same for the relationships, and we could also add more than one label.

```
MATCH (p:Publication)
WHERE p.isbn IS NOT NULL
SET p:Book
RETURN p;
```

4. **Update attributes of a relationship (analog for nodes)**
   The next command updates the values of the attributes of a specific relationship. In particular, the number of the starting and ending pages of the paper with id `53e99785b7602d9701f40556` are set through the keyword `SET`. If the user makes a mistake while inserting the properties of the relationship `IS_PART_OF`, let's say the

pages, the user can always modify afterwards the entries with the `SET` operation. With the same syntax, if new properties come up or something was missed when adding nodes or relationships it is possible to create new attributes and set them to a specific value.

Parameters

```
:param id => "53e99785b7602d9701f40556";
```

Command

```
MATCH (paper:Paper)-[i:IS_PART_OF]->(:Book)
WHERE paper.id = $id
SET i.page_start = 15
SET i.page_end = 33
```

5. **Remove attributes of a relationship (analogous for nodes)**
   The following commands remove some attributes from a relationship. In particular, we remove the attributes concerning the starting and ending numbers of pages within a book when these numbers are negative. We also remove these numbers when the index of the starting page is greater than the index of the ending page because it doesn't make sense to have such data. This can be seen as some sort of data cleaning, useful to remove incorrect data.

```
MATCH (:Paper)-[i:IS_PART_OF]->(:Book)
WHERE i.page_start < 0
REMOVE i.page_start
```

```
MATCH (:Paper)-[i:IS_PART_OF]->(:Book)
WHERE i.page_end < 0
REMOVE i.page_end
```

```
MATCH (:Paper)-[i:IS_PART_OF]->(:Book)
WHERE NOT i.page_start IS NULL AND NOT i.page_end IS NULL AND i.
   page_end - i.page_start < 0
REMOVE i.page_start, i.page_end
```

6. **Delete a node**
   With the following command we delete the node `Book` titled `"Mathematics and Computers in Simulation"`, using the `DELETE` keyword. Before performing the delete action, we use the `DETACH` keyword that assures that all the relationships in which the node was involved are detached from the node itself in order to be able to remove it.

Parameters

```
:param name => "Mathematics and Computers in Simulation";
```

Command

```
MATCH (b:Book {name:$name})
DETACH DELETE b;
```

6.2. Queries

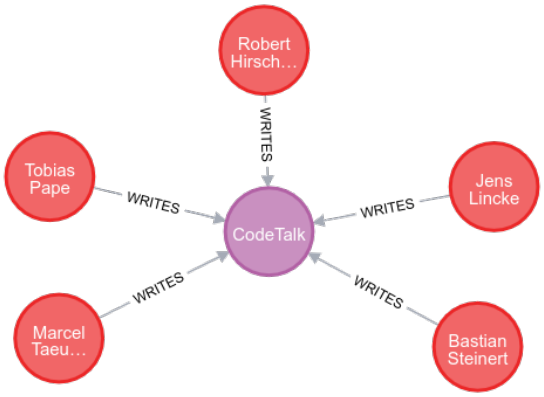
1. **Authors who wrote a certain paper**
   The query returns a graph with all the authors who wrote the papers titled `"CodeTalk"`. This is a very simple query that can be very useful in everyday searches.

   ### Parameters

   ```
   :param title => "CodeTalk";
   ```

   ### Query

   ```
   MATCH g = (:Author)-[:WRITES]->(:Paper {title:$title})
   RETURN g
   ```



2. **Previous affiliations of an author**
   The query returns all the organizations to whom the author `"Annabel Sebag"` has been affiliated sorted by the year in which she wrote the paper with that affiliation.

   ### Parameters

   ```
   :param name => "Annabel Sebag";
   ```

   ### Query

```
MATCH (:Author {name:$name})-[r:WRITES]-(p:Paper)
WITH r.affiliation AS affiliation, p.year AS year
ORDER BY year DESC
RETURN DISTINCT affiliation, year
```

"affiliation"	"year"
"Autour De Minuit"	2013
"Premium Films, France"	2009
"Paris, France"	2007
"Supinfocom Arles, Paris, France"	2007
"Supinfocom Valenciennes, Paris, France"	2007
"Supinfocom Valenciennes, Paris, France"	1950

3. **Authors who wrote for a journal in a specific year**

   The query returns all the authors who wrote at least a paper for the journal named `"Briefings in Bioinformatics"` in the year 2005. In the return statement we use the keyword `DISTINCT` in order to count only once each author because we only want to know if the author wrote on that paper in the specified year, and not how many times.

   Parameters

```
:param name => "Briefings in Bioinformatics";
:param year => 2005;
```

   Query

```
MATCH (a:Author)-[:WRITES]->(p:Paper)-[:IS_PART_OF]->(j:Journal)
WHERE p.year = $year AND j.name = $name
RETURN DISTINCT a.name AS authorName
```

"authorName"
"L. V. Sudoplatov"
"V. E. Davidson"
"Yu. S. Postol'nik"
"I. S. Molchadskii"
"N. Yu. Taits"
"L. A. Gavur"
"N. P. Fedorin"
"A. P. Tolstopyat"
"N. I. Yalovoi"
"Yu. L. Rastorguev"

4. **Papers that belong to a Journal, written by authors while they were affiliated to a certain organization**
   The query returns the names of all the journals written by authors while they were affiliated with the organization named `"Wien"`.

   In some of the queries we have seen before, we retrieved from the database the affiliations related to the authors. It is also interesting to see how easily we can retrieve the authors that are related to a certain affiliation, and then explore the relationships of the author in order to obtain meaningful data, useful for everyday search in the bibliography database. The next query is very fast because centered mostly on relationships, a natural characteristic of graphs.

   Parameters

```
:param affiliation => "Wien";
```

   Query

```
MATCH (:Author)-[:WRITES {affiliation:$affiliation}]->(:Paper)-[:
    IS_PART_OF]->(b:Journal)
RETURN DISTINCT b.name as journalName;
```

"journalName"
"Datenschutz und Datensicherheit"
"Math. Meth. of OR"
"Computing"
"Wirtschaftsinformatik"
"Biological Cybernetics"
"HMD - Praxis Wirtschaftsinform."
"Unternehmensforschung"
"Zeitschr. für OR"

5. **Authors which published more papers in journals in a specific year**
   The query returns the five authors that published more papers in the year 2000 in journals sorted by the number of papers they published. We use the function `count()` to obtain for each author the relative number of papers and limit the output to the first five authors discovered during the search.

   Parameters

```
:param year => 2000;
```

   Query

```
MATCH (a:Author)-[:WRITES]->(p:Paper)-[:IS_PART_OF]->(:Journal)
WHERE p.year = $year
WITH a, count(*) AS paperNum
RETURN a.name AS authorName, paperNum
ORDER BY paperNum DESC LIMIT 5;
```

"authorName"	"paperNum"
"David Redmiles"	4
"Douglas Blank"	2
"Steve Vestal"	2
"François Hennecart"	1
"Dietmar Dietrich"	1

6. **Number of papers written in conferences by authors whose name starts with specific letters per venue**

The query returns for each venue the number of papers that have been presented in a conference considering only papers that are written by authors whose name starts for "A" or "S".

Parameters

```
:param letter1 => "A";
:param letter2 => "S";
```

Query

```
MATCH (a:Author)-[:WRITES]->(p:Paper)-[:IS_PART_OF]->(c:Conference)
WHERE a.name STARTS WITH $letter1 OR a.name STARTS WITH $letter2
WITH c.name AS conference, c.location AS location,
     count(DISTINCT p) AS number_of_papers
RETURN DISTINCT conference, location, number_of_papers
ORDER BY number_of_papers DESC LIMIT 5;
```

"Conference"	"Location"	"number_of_papers"
"SIGGRAPH Computer Animation Festival"	"Bangkok, Thailand"	2
"SIGGRAPH Computer Animation Festival"	"Barcelona, Spain"	2
"NEMS"	"Vancouver, Canada"	1
"AINA Workshops"	"Moscow, Russia"	1
"SIGGRAPH Computer Animation Festival"	"Copenhagen, Denmark"	1

7. **Authors who wrote the most referenced papers which belong to publications of type book, while they were affiliated to a specific organization**
   The query returns the authors whose papers have been referenced the most in papers that are part of books while they were affiliated to "University of Mannheim". We also want to retrieve only authors that published at least 2 papers, so we filter furthermore the result.

   Parameters

```
:param affiliation => "University of Mannheim";
```

Query

```
MATCH (a:Author)-[w:WRITES]->(:Paper)<-[r:REFERENCES]-(:Paper)-[:
    IS_PART_OF]->(:Book)
WHERE w.affiliation = $affiliation
WITH a, count(DISTINCT r) AS totalReferences
WHERE totalReferences > 2
RETURN a.name AS authorName, totalReferences;
```

"author"	"totalReferences"
"Heiner Stuckenschmidt"	3
"Anne Schlicht"	3

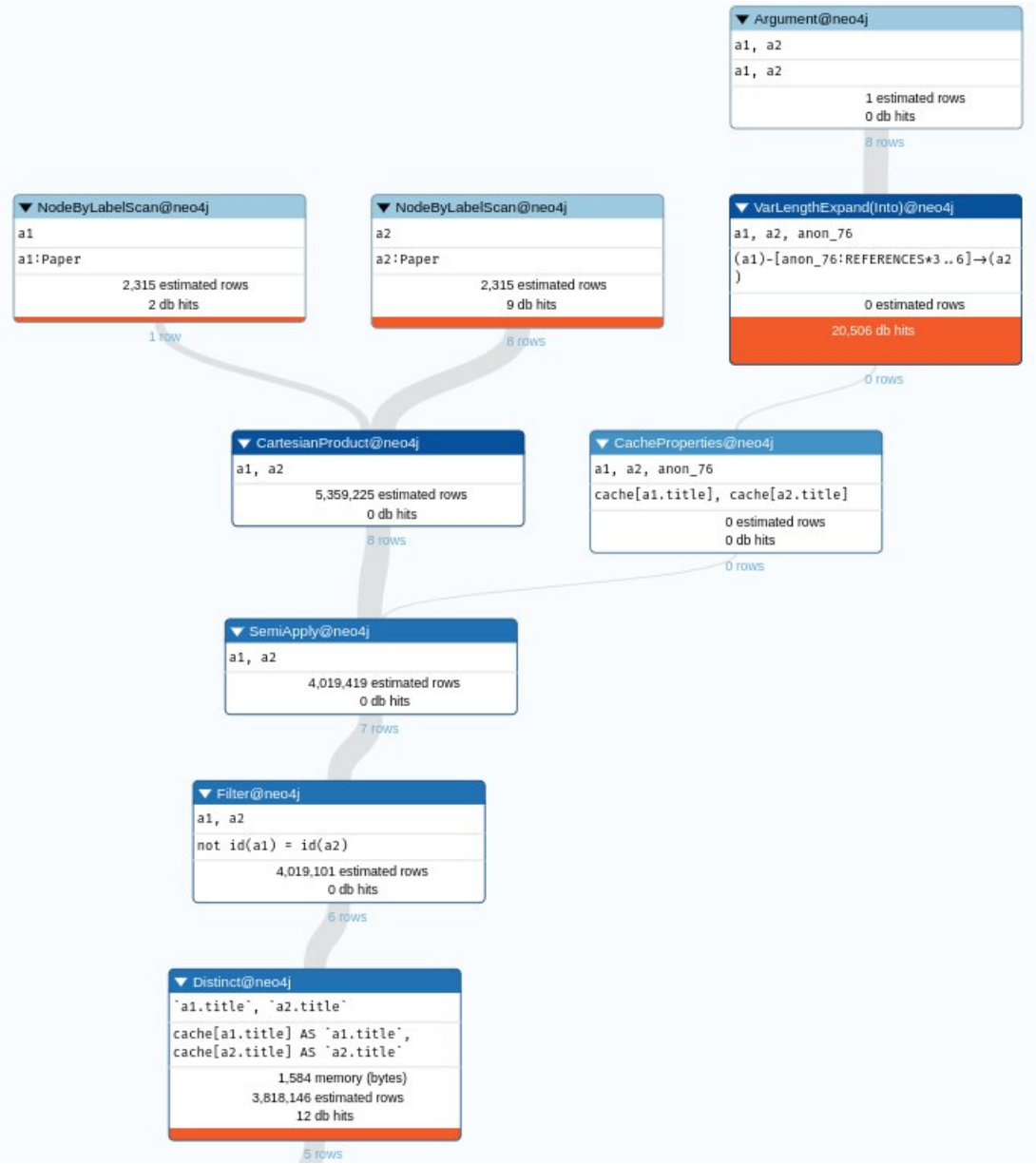
8. **Papers indirectly referenced by other papers at a certain distance of steps**
   The query returns different papers linked to each other by the relation `REFERENCES` at a distance of steps between 3 and 6. We limit the results in order to get the output as soon as the process finds 5 rows that respect the filter of the search.

```
MATCH (a1:Paper), (a2:Paper)
WHERE id(a1) <> id(a2) AND (EXISTS {MATCH (a1)-[:REFERENCES
    *3..6]->(a2)})
RETURN DISTINCT a1.title AS firstPaper, a2.title AS secondPaper
LIMIT 5;
```

"firstPaper"	"secondPaper"
"3GIO."	"The relationship between canopy parameters and spectrum of winter whe at under different irrigations in Hebei Province."
"3GIO."	"A solution to the problem of touching and broken characters."
"3GIO."	"Timing yield estimation using statistical static timing analysis"
"3GIO."	"360°"
"3GIO."	"300"

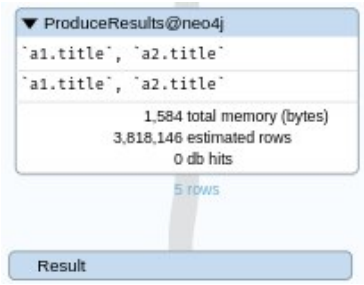
The performance is not optimal and using the `PROFILE` command in the same query we can see how the search is actually made in the graph using this specific interrogation on the database.

```
PROFILE
MATCH (a1:Paper), (a2:Paper)
WHERE id(a1) <> id(a2) AND (EXISTS {MATCH (a1)-[:REFERENCES
    *3..6]->(a2)})
RETURN DISTINCT a1.title AS firstPaper, a2.title AS secondPaper
    LIMIT 5;
```



We can notice that the execution of the query is done in parallel trying on one side to perform the `MATCH` of the two papers separately and on the other side to find the relationships between the two papers given by `REFERENCES` from 3 to 6 steps away. The two results of the search are then merged together.

In this process, we estimated a lot of rows and wasted time, by doing the controls individually.



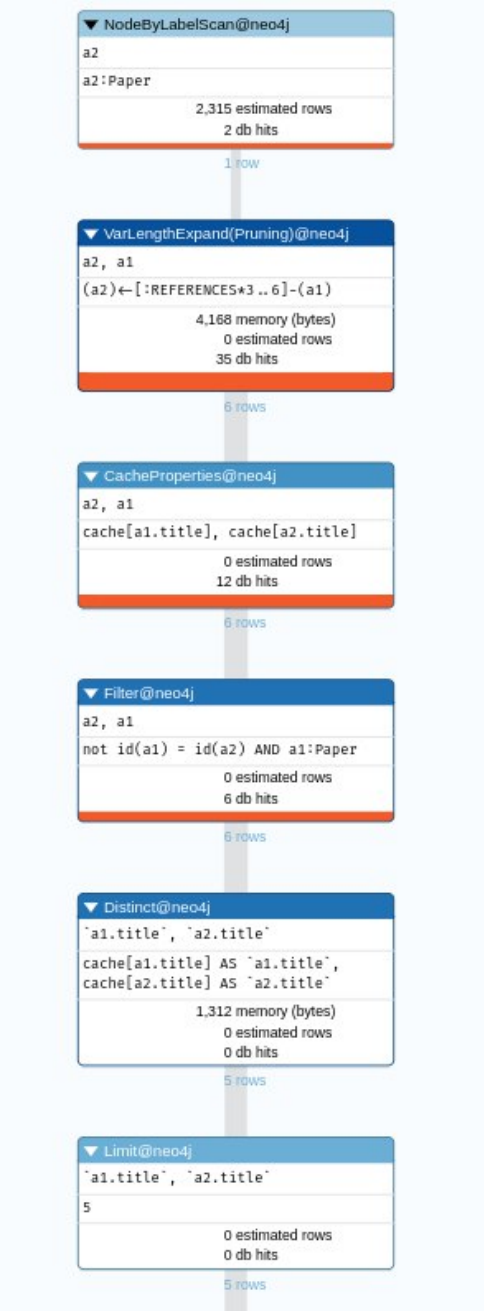
To make the query more efficient we can bring the filtering of the data a step ahead, so we can check directly in the `MATCH` command for the relationships between the papers.

```
MATCH (a1:Paper)-[:REFERENCES*3..6]->(a2:Paper)
WHERE id(a1) <> id(a2)
RETURN DISTINCT a1.title AS firstPaper, a2.title AS secondPaper
LIMIT 5;
```

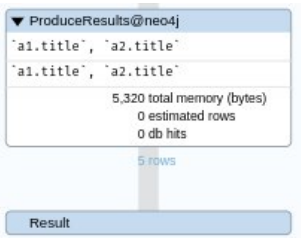
We notice that the results can be different from the precedent approach, due to the use of `LIMIT` that stops the research as soon as it gets the 5 results of interest and in this case, the search is done differently so the outputs are not the same.

"firstPaper"	"secondPaper"
"Editorial."	"3GIO."
"Discussion"	"3GIO."
"Acknowledgements"	"3GIO."
"Colophon"	"3GIO."
"Capstone"	"3GIO."

Using this version of the query, we have that the search is more linear because, in `MATCH`, we don't look for the papers separately but once matched a `Paper` we search for its recursive `REFERENCES` relationships and match at once the two papers and their connections. This approach is fast and improves efficiency.



We can see that reaching the result the command required more memory, but we didn't estimate useless rows of data, extracting only what we are interested in without an extended search on the graph.



Alternatively, we can tackle this search from another point of view. We can use a query based precisely on the steps we are interested in, making the query more explicit and getting other results, because in the recursive form the process could stop before reaching this exact amount of steps, especially using the `LIMIT` keyword.

```
MATCH (a1:Paper)-[:REFERENCES]->(a2:Paper)-[:REFERENCES]->(a3:Paper
    )-[:REFERENCES]->(a4:Paper)-[:REFERENCES]->(a5:Paper)-[:
    REFERENCES]->(a6:Paper)
WHERE id(a1) <> id(a2) AND id(a2) <> id(a3) AND id(a3) <> id(a4)
    AND id(a4) <> id(a5) AND id(a5) <> id(a6)
RETURN DISTINCT a1.title AS firstPaper, a6.title AS secondPaper
LIMIT 5;
```

"firstPaper"	"secondPaper"
"Funding"	"Foreword"
"Funding"	"FMOL"
"Funding"	"GCM."
"Funding"	"Digitrama"
"Funding"	"Editorial"

9. **Best collaborator of all times for each author in terms of papers written in journals**

   The following query allows the researcher to understand each author who is the best collaborator of all time considering their collaborations in journals. In the result, we can sometimes see symmetry in the data, if an author has another as the best collaborator and vice versa.

   Firstly we do a `MATCH` imposing our constraints. Secondly, we group by authors and co-authors counting for each couple how many papers were published in journals they wrote together, and we order the table by this number in descending order. On the next `WITH` clause for each author we extract only the best collaborator, namely the one with whom he wrote more papers in journals.

   Parameters Query

```
MATCH (a:Author)-[:WRITES]->(p:Paper)<-[:WRITES]-(ca:Author), (p)
    -[:IS_PART_OF]->(:Journal)
WITH DISTINCT a, ca, count(DISTINCT p) AS cnt ORDER BY cnt DESC
WITH DISTINCT a, collect(ca.name) AS colabs, collect(cnt) AS cnts
```

```
RETURN a.name AS author, colabs[0] AS bestCollaborator, cnts[0] AS
    numberOfPapers LIMIT 5
```

"author"	"bestCollaborator"	"numberOfPapers"
"David J. Lipman"	"Dennis A. Benson"	10
"Dennis A. Benson"	"David J. Lipman"	10
"James Ostell"	"Dennis A. Benson"	9
"Ilene Karsch-Mizrachi"	"David J. Lipman"	6
"K. Prachar"	"W. Nöbauer"	6

10. **Papers which are at the base of a given field of study**

    The query returns the papers which are at the base of a given field of study. To be the base for a field of study means to not reference directly or indirectly other papers of that field of study. With this query, we retrieve the papers that are probably at the base of some field of study because an upper bound of 7 is chosen to check the indirect references.

    A variant of this query can be to retrieve the authors who wrote the base papers of a field of study, so the authors are the "founders" of that field. The query is more meaningful if run on a very large set of data.

    Parameters

```
:param fos => "Statistics";
```

    Query

```
MATCH (founder:Paper)-[:BELONGS_TO]->(fos:Fos{fos:$fos})<-[:
    BELONGS_TO]-(p:Paper)
WHERE id(founder) <> id(p)
WITH collect(p) AS test, founder, fos
WHERE all(p IN test WHERE NOT EXISTS((founder)-[:REFERENCES*7]->(p)
    ))
RETURN fos AS field, collect(DISTINCT founder.title) AS
    founderPapers
```

```
╒═══════════════════╤═══════════════════════════════════════════════════════════════════════════════╕
│"field"            │"founderPapers"                                                                │
╞═══════════════════╪═══════════════════════════════════════════════════════════════════════════════╡
│{"fos":"Statistics"}│["Timing yield estimation using statistical static timing analysis","E       │
│                   │xperiments","BUCHBESPRECHUNGEN","Anecdotes","Education.","BPEL4WS.","A        │
│                   │lphabet.","Forward","OBITUARY.","Deadline","Dice","Buchbesprechungen",        │
│                   │"AirEOD"]                                                                      │
└───────────────────┴───────────────────────────────────────────────────────────────────────────────┘
```

11. **Total pages written by the authors of an organization within a journal**
    The query returns the total number of pages written by authors affiliated with the organization `"Federal Institute of Technology, Switzerland"` within journals. The function `avg()` can be used to compute the average of the pages written by an author in the papers.

    Parameters

    ```
    :param affiliation=>"Federal Institute of Technology, Switzerland";
    ```

    Query

    ```
    MATCH (a:Author)-[w:WRITES {affiliation:$affiliation}]->(p:Paper)-[
       i:IS_PART_OF]->(j:Journal)
    WHERE i.page_start IS NOT NULL AND i.page_end IS NOT NULL
    WITH a, sum(i.page_end - i.page_start) AS num_pag
    ORDER BY num_pag DESC
    RETURN a AS Author, num_pag;
    ```

    ```
    ╒═══════════════════════════════════════════════════════════╤═══════╕
    │"Author"                                                   │"num_pag"│
    ╞═══════════════════════════════════════════════════════════╪═══════╡
    │{"name":"C. Genillard","id":"53f458e5dabfaee4dc81bc6e"}    │19     │
    ├───────────────────────────────────────────────────────────┼───────┤
    │{"name":"A. Strohmeier","id":"53f45b8bdabfaee1c0b4094c"}   │19     │
    └───────────────────────────────────────────────────────────┴───────┘
    ```

12. **Organizations which have published their own more papers within a specific journal with associated fields of study**
    The query returns the organizations which have exclusively written more papers, which means that the authors of the papers have all the same affiliation on `"Commun . ACM"`. The query provides the fields of study on which the considered papers were focused on.

    Parameters

    ```
    :param journalTitle => "Commun. ACM";
    ```

Query

```
MATCH (a1:Author)-[w1:WRITES]->(p1:Paper)-[:IS_PART_OF]->(j:Journal
    {name:$journalTitle}), (p1:Paper)-[:BELONGS_TO]->(fos:Fos), (a2
   :Author)-[w2:WRITES]-(p1:Paper)
WHERE a1 <> a2 AND w1.affiliation IS NOT NULL
WITH w1.affiliation AS affiliation, collect(DISTINCT fos) AS
   fieldsOfStudy, collect(DISTINCT w2.affiliation) AS others, count
   (DISTINCT p1) AS numberOfPapers
WHERE ALL(organization IN others WHERE affiliation=organization)
WITH affiliation, fieldsOfStudy, numberOfPapers
ORDER BY numberOfPapers DESC
RETURN affiliation, fieldsOfStudy, numberOfPapers
LIMIT 2
```

"affiliation"	"fieldsOfStudy"	"numberOfPapers"
"National Center for Supercomputing Applications, University of Illinois, Urbana-Champaign"	[{"fos":"World Wide Web"},{"fos":"Information infrastructure"},{"fos":"Pure mathematics"},{"fos":"Water content"},{"fos":"Moisture"},{"fos":"Real-time computing"},{"fos":"Revenue"},{"fos":"Mobile device"},{"fos":"Changeover"}]	1
"Univ. of Virginia, Charlottesville"	[{"fos":"Algorithm"},{"fos":"Computer programming"}]	1

13. **Recommended books given one book**

    The query returns the book considered the nearest to the one given as a parameter. The similarity between books is evaluated in terms of the number of keywords associated with the papers linked to the books. In addition, fields of study which are common to the two books are shown.

```
:param title => "RoboCup 2009";
```

Query

```
MATCH (p1:Paper)-[:IS_PART_OF]->(b1:Book),
(p1:Paper)-[:BELONGS_TO]->(fos1:Fos)<-[:BELONGS_TO]-(p2:Paper),
(p1)-[:HIGHLIGHTS]->(k1:Keyword)<-[:HIGHLIGHTS]-(p2),
(p2:Paper)-[:IS_PART_OF]->(b2:Book)
WHERE b1 <> b2 AND p1 <> p2 AND b1.name = $title
WITH b1, b2, collect(DISTINCT fos1) AS fos, count(DISTINCT k1) AS
   score
ORDER BY score DESC
RETURN b2.name AS relatedBook, fos, score
LIMIT 3;
```

"relatedBook"	"fos"	"score"
"SIGGRAPH Sketches"	[{"fos":"Mobile device"},{"fos":"Mobile computing"}]	3
"SIGGRAPH Electronic Art and Animation Catalog"	[{"fos":"Artificial intelligence"}]	3
"SIGGRAPH Abstracts and Applications"	[{"fos":"Suurballe's algorithm"}]	1

14. **Possible new interesting collaborations between authors in the same fields of study**

    The query returns the new possible interesting collaborations, by which we mean the couple of authors who haven't written a paper together but have written many times with common authors and wrote about similar topics, so they are involved in the same fields of study.

    Parameters

```
:param minimum_collabs =>1;
```

    Query

```
MATCH
(a1:Author)-[w1:WRITES]->(p1:Paper)<-[w2:WRITES]-(a2:Author),
(a2:Author)-[w3:WRITES]->(p2:Paper)<-[w4:WRITES]-(a3:Author),
(a1)-[:WRITES]->(p3:Paper)-[:BELONGS_TO]->(f1:Fos)<-[:BELONGS_TO]-(
    p4:Paper)<-[:WRITES]-(a3)
WHERE NOT EXISTS ((a1)-[:WRITES]->(:Paper)<-[:WRITES]-(a3)) AND a1
    <> a3 AND a1 <> a2 AND a2 <> a3
WITH a1, a3, collect(DISTINCT f1.fos) AS fields, count(DISTINCT f1)
     AS common_fields, count(DISTINCT p1) AS common_collabs1, count(
    DISTINCT p2) AS common_collabs2
WHERE common_collabs1 > $minimum_collabs AND common_collabs2 >
    $minimum_collabs
WITH a1, a3, fields, common_fields, common_collabs1 +
    common_collabs2 AS common_collabs
ORDER BY common_fields DESC, common_collabs DESC
RETURN DISTINCT a1.name as Author1, a3.name as Author2, fields,
    common_collabs, common_fields
LIMIT 5
```

"Author1"	"Author2"	"fields"	"common_collabs"	"common_fields"
"Ilene Karsch-Mizrachi"	"Mark S. Boguski"	["Router","Engineering"]	8	2
"Mark S. Boguski"	"Ilene Karsch-Mizrachi"	["Router","Engineering"]	8	2
"K. Mayrhofer"	"H. Muthsam"	["Statistical static timing analysis","Statistics"]	7	2
"K. Mayrhofer"	"J. Tichý"	["Statistical static timing analysis","Statistics"]	7	2
"K. Mayrhofer"	"H. Rindler"	["Statistical static timing analysis","Statistics"]	7	2

15. **Shortest connection between two papers with no common field of study**
With this query, we want to find what is the shortest path, by navigating the references, between two papers that do not have any field of study in common. Firstly we bound the papers node and the associated field of studies to variables `p1/p2` and `fos1/fos2` respectively and we impose that `p1` and `p2` must be different. Secondly, we collect the fields of study associated with each paper and put a condition stating that the intersection of these two collection must be zero. Finally, we call the `shortestPath` function and impose all the conditions on the path to be found; we put an upper bound of 7 to the length of the path to avoid searching too deep in the graph and limited the search to the first valid path found using the clause `WITH sp LIMIT 1`, where `sp` refers to the result of the call to the `shortestPath` function.

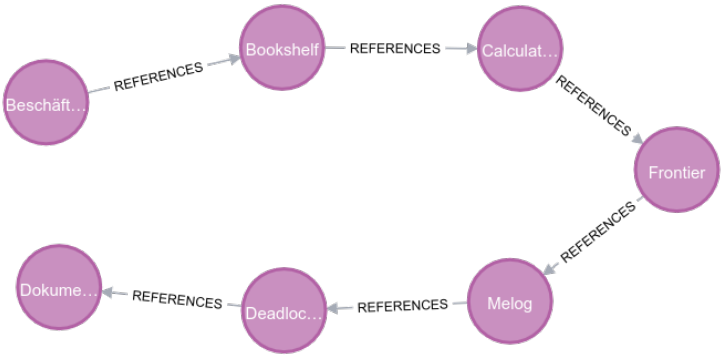
Parameters

```
:param fos1 => "Irrigation";
:param fos2 => "Artificial intelligence";
```

Query

```
MATCH (fos1:Fos) <-[:BELONGS_TO]- (p1:Paper) -[:BELONGS_TO]-> (:Fos
    {fos:$fos1}),
      (fos2:Fos) <-[:BELONGS_TO]- (p2:Paper) -[:BELONGS_TO]-> (:Fos
    {fos:$fos2})
WHERE p1 <> p2
WITH p1, collect(DISTINCT fos1) AS foss1, p2, collect(DISTINCT fos2
   ) AS foss2
WHERE all(f IN foss1 WHERE NOT f IN foss2)
MATCH sp = shortestPath((p1)-[:REFERENCES*..7]->(p2))
WHERE length(sp) > 1
RETURN sp LIMIT 1;
```

Below we can see the result of the query run with the aforementioned parameters.



Part III

MongoDB

7 | Introduction to the problem

In this part, we implement a bibliographic database through MongoDB, which is a document-oriented database. In the previous part, using Neo4j, the main focus was on the relationships between nodes, instead, MongoDB's paradigm is different. It shifts the attention to the document itself, which is the atomic and fundamental element of the database. In this case, each document represents a single paper and contains all the related information. All the articles are stored in the same collection called `bibliography`. A bibliographic database is a use case where the documental approach may be very successful. The main focus of this kind of database is to store the information regarding every single paper without the need to relate many articles together and without doing very complex queries.

8 | Document structure

Since we are using a documental database, papers are the main entity and, because of this, it is reasonable to store the text and the structure of the articles. These fields are added to the attributes listed in the previous part regarding graph databases. In particular, each paper is composed of many sections and each section has a title, several paragraphs, and a certain number of image URLs with associated captions. Moreover, each section can contain subsections that have the same structure as a section.

In addition, also the email and bio of the authors have been added as well as the publication date of the article.

The following table shows the document's fields associated with their respective MongoDB type and meaning.

Field name	Type	Description
_id	String	paper ID
title	String	paper title
authors	Array of Object	list of authors
authors._id	String	author ID
authors.name	String	author name
authors.org	String	author affiliation
authors.email	String	author email
authors.bio	String	author short bio
keywords	Array of String	paper keywords
fos	Array of String	paper fields of study
publication_type	String	paper type
venue	String	publication name
volume	Int32	journal volume
issue	Int32	journal issue
page_start	Int32	start page
page_end	Int32	end page
date	Date	publication date
lang	String	paper language
isbn	String	book ISBN
issn	String	journal ISSN
doi	String	paper DOI
publisher	String	publisher name
location	String	conference location
abstract	String	paper abstract
url	Array of String	paper URLs
references	Array of String	paper references
sections	Array of Object	paper sections
sections.title	String	section title
sections.paragraphs	Array of String	section paragraphs
sections.subsections	Array of Object	section subsections
sections.figures	Array of Object	section figures
sections.figures.URL	String	figure URL
sections.figures.caption	String	figure caption

Table 8.1: Description of the fields with related type and meaning.

Notes:

- `publication_type` can only assume "Journal", "Book" or "Conference" as value;
- a paper of type "Journal" can have the fields `issn`, `volume`, `issue` and `publisher`;
- a paper of type "Book" can have the fields `isbn` and `publisher`;
- a paper of type "Conference" can have the field `location`;
- `subsections` have the same structure as `sections`, so they could contain multiple `subsections` and so on in a recursively way. We decided to consider only one level of subsections, so `subsections` don't have the `subsections` field.

The following code snippet shows our document structure represented in JSON format.

```
{
    "_id": <String>,
    "title": <String>,
    "authors": [
        {
            "_id": <String>,
            "name": <String>,
            "email": <String>,
            "bio": <String>,
            "org": <String>
        }
    ],
    "abstract": <String>,
    "keywords": [<String>],
    "fos": [<String>],
    "publication_type": <String>,
    "venue": <String>,
    "volume": <Int32>,
    "issue": <Int32>,
    "issn": <String>,
    "publisher": <String>,
    "isbn": <String>,
    "location": <String>,
    "date": <ISODate>,
    "page_start": <Int32>,
    "page_end": <Int32>,
    "lang": <String>,
    "doi": <String>,
    "url": [<String>],
    "references": [<String>],
    "sections": [
```

```
        {
            "title": <String>,
            "paragraphs": [<String>],
            "figures": [
                {
                    "URL": <String>,
                    "caption": <String>
                }
            ],
            "subsections": [
                {
                  "title": <String>,
                  "paragraphs": [<String>],
                  "figures": [
                      {
                          "URL": <String>,
                          "caption": <String>
                      }
                  ]
                }
            ]
        }
    ]
}
```

Below we provide some examples of the three main paper types that appear in our document with their associated attributes. The difference in the attributes is minimal and, as mentioned beforehand, it concerns the fields related to the publishing type of the paper.

Model of the journal: This is the document structure for the papers published in journals.


```
_id: "53e99905b7602d9702145cf7"
title: "An O(n) bin-packing algorithm for uniformly distributed data."
authors: Array
  0: Object
      _id: "53f45291dabfaeecd69dcbf8"
      name: "J Csirik"
      email: "j.csirikf8@gmail.com"
      bio: "My name is J Csirik"
  1: Object
keywords: Array
    0: "bin packing"
    1: "bin packing problem"
    2: "heuristic algorithm"
    3: "probabilistic analysis"
fos: Array
    0: "computer communication networks"
    1: "algorithm"
    2: "probabilistic analysis of algorithms"
    3: "probabilistic logic"
    4: "bin packing problem"
    5: "mathematics"
page_start: 313
page_end: 319
lang: "en"
volume: 36
issue: 4
doi: "10.1007/BF02240206"
url: Array
    0: "http://dx.doi.org/10.1007/BF02240206"
    1: "http://link.springer.com/article/10.1007/BF02240206"
abstract: "We give a first-fit type algorithm, with running timeO(n), for the cla…"
references: Array
    0: "53e9be4ab7602d9704b09bda"
    1: "53e9a81fb7602d970316858e"
    2: "53e9ac28b7602d97035eb4f5"
    3: "53e9ac82b7602d9703658b34"
    4: "53e9a162b7602d9702a5483c"
publication_type: "Journal"
publisher: "IEEE Systems, Man, and Cybernetics Society"
venue: "Computing"
sections: Array
  0: Object
      title: "Excellent"
    paragraphs: Array
        0: "These gummy strawberries are YUMMY and very flavorful. We are big fans…"
        1: "Funny thing about this order: I thought these were a different kind of…"
        2: "These are really great tasting and oh so cute.  They were perfect for …"
    figures: Array
      0: Object
          URL: "https://java.com/nulla/sed.html"
          caption: "Great, healthy dog food for dogs with food allergies"
    subsections: Array
      0: Object
          title: "Huge pack of chewy gummies"
        paragraphs: Array
        figures: Array
      1: Object
  1: Object
  2: Object
  3: Object
date: 1983-10-02T23:49:56.000+00:00
```

Model of the book: This is the document structure for the papers published in books.


```
_id: "53e997ecb7602d9701fe6bb9"
title: "Applicative Information Systems"
authors: Array
  0: Object
     _id: "53f462afdabfaee2a1d9fe5d"
     name: "Mario Coppo"
     email: "mario.coppo5d@gmail.com"
     bio: "My name is Mario Coppo"
  1: Object
  2: Object
keywords: Array
   0: "applicative information systems"
   1: "information system"
fos: Array
   0: "information system"
   1: "computer science"
   2: "theoretical computer science"
   3: "recursive functions"
page_start: 35
page_end: 64
lang: "en"
isbn: "3-540-12727-5"
doi: "10.1007/3-540-12727-5_2"
url: Array
   0: "http://dx.doi.org/10.1007/3-540-12727-5_2"
abstract: "Without Abstract"
references: Array
   0: "53e9b90bb7602d97044f09eb"
   1: "53e99a2bb7602d97022845a9"
   2: "53e99c1ab7602d97024cae1e"
   3: "53e9a500b7602d9702e1ecf6"
publication_type: "Book"
publisher: "Association for Computing Machinery (ACM)"
venue: "CAAP"
sections: Array
  0: Object
     title: "These are delicious!"
    paragraphs: Array
       0: "we love Pop Chips, but they ceased being available in our area.  So I …"
       1: "My parents were visiting us in Las Vegas from the east coast. I was at…"
       2: "Lower in calories and great taste.  I like them just as much as the re…"
       3: "Best chips I've ever eaten but you should sell them in 11 to 15 oz bag…"
    figures: Array
      0: Object
         URL: "http://usatoday.com/eget/eros/elementum.jsp"
         caption: "For the price, these are great for all. Even a crowd. Much better than…"
      1: Object
    subsections: Array
      0: Object
         title: "Unbelievable - healthy and incredibly tasty"
        paragraphs: Array
           0: "These chips are really good. Crisp and delicious. My favorite flavor i…"
           1: "After years of being addicted to Smart Puffs, as soon as I tried these…"
         figures: Array
  1: Object
     title: "Not Your Grandma's Cookies, But They Get The Job Done"
    paragraphs: Array
    figures: Array
    subsections: Array
date: 1954-07-31T08:06:59.000+00:00
```

Model of the conference: This is the document structure for the papers published at conferences.


```
_id: "53e997e9b7602d9701fe36df"
title: "Coalitional affinity games"
authors: Array
  0: Object
      _id: "53f43d50dabfaee1c0ad4b85"
      name: "Simina Brânzei"
      email: "simina.brânzei85@gmail.com"
      bio: "My name is Simina Brânzei"
  1: Object
      _id: "5440ede1dabfae805a709079"
      name: "Kate Larson"
      email: "kate.larson79@gmail.com"
      bio: "My name is Kate Larson"
keywords: Array
    0: "hedonic game"
    1: "coalitional affinity game"
    2: "stable coalition structure"
    3: "interesting class"
    4: "stability property"
    5: "coalition structure"
    6: "stability gap"
    7: "core solution concept"
    8: "social welfare"
    9: "affinity game"
fos: Array
    0: "mathematical economics"
    1: "computer science"
    2: "solution concept"
    3: "social welfare"
    4: "bounded function"
page_start: 1319
page_end: 1320
lang: "en"
doi: "10.1145/1558109.1558272"
url: Array
    0: "http://dx.doi.org/10.1145/1558109.1558272"
    1: "http://doi.acm.org/10.1145/1558109.1558272"
abstract: "We present and analyze coalitional affinity games, a family of hedonic…"
references: Array
    0: "53e9a938b7602d970328d1f2"
publication_type: "Conference"
location: "Los Angeles, USA"
venue: "Autonomous Agents & Multiagent Systems/Agent Theories, Architectures, …"
sections: Array
  0: Object
      title: "Disappointed"
    paragraphs: Array
        0: "This toy did not last. I have a boxer and he chewed the rope off the j…"
        1: "My dog destroyed the rubber "cord" in 5 minutes (worse yet, swallowed …"
        2: "I BOUGHT THIS FOR MY DOG AND HE TOOK ONE LOOK AT IT PLAYED WITH IT FOR…"
        3: "When i took this out my puppy couldn't wait to get into it.  I put the…"
    figures: Array
      0: Object
          URL: "https://ox.ac.uk/pede/lobortis.jpg"
          caption: "Of al the Stash iced teas this is the best, and it is still not all th…"
      1: Object
      2: Object
      3: Object
    subsections: Array
      0: Object
          title: "Hours of endless entertainment"
        paragraphs: Array
        figures: Array
      1: Object
  1: Object
  2: Object
  3: Object
date: 2007-10-22T21:29:26.000+00:00
```

8.1. Differences concerning the previous models

The main advantage of this approach is the fact that it is very easy to get all the relevant data of an article because all the information is stored together and usually allows to avoid performing expensive joins that would be necessary using a relational database.

Furthermore, MongoDB is flexible permitting to have heterogeneous documents, we need to maintain the same skeleton for all documents in the same collection to perform meaningful queries and have a clean database, but also differences between documents are admitted. As highlighted in the notes above, different types of articles can have different fields and if an article doesn't have certain data is not necessary to store an empty field, simply the attribute is not created. In addition, this flexibility makes easy the storage of the text structure that could, in principle, have many levels of subsections.

One drawback is the duplication of data, for example, an author may have written more than one paper that is stored in the database, so the author's information is saved multiple times. This situation could lead to some inconsistency and requires attention when data updates are performed. But, in the field of bibliography, it is seldom necessary to do updates since all data refer to a specific article and if a paper is modified it is more probable to create a new document with the new version than update the older one.

9 | Data upload

9.1. Pre-processing

The assignment for this part of the project defined a basic structure of the document with the fields it must have. Some of the fields defined by this structure were not present in our previous dataset so we had to fix this aspect by doing some pre-processing of the data before working with it. The starting point has been the dblp dataset, cleaned with python scripts just like in the Neo4j part, but also enriched with new information and new filters on the data. In particular, the main work revolved around the following fields.

Author email and bio Using a Python script, we added the emails of the authors adopting the following format:

```
author.name + last-two-digits-of-the-author._id + "@gmail.com"
```

Using the same script, we added the bio using the format:

```
"My name is " + author.name + "." + "I'm currently working for " +
    author.org
```

The second sentence, `'"I'm currently working for " + author.org'` is present only if the author has an affiliation.

Then we generated random dates adopting the format `ISODate` of `MongoDB` from www.mockaroo.com and added them to the original dataset with a Python script.

Sections and subsections The sections and subsections fields, as mentioned in the assignment instructions were mandatory for the document structure. However, the documents in our dataset didn't have those fields so we had to manually generate them. We did so by using a Python script. More specifically the titles of the sections and subsections and the text for the paragraphs were extracted from the fields `Summary` and `Text` respectively, present in an external dataset named `Reviews.csv`[1], which consists of food

[1] Amazon Fine Food Reviews, accessed on the 14/11/2022
https://www.kaggle.com/datasets/snap/amazon-fine-food-reviews

reviews from Amazon. The script first generates a set of unique sections with a random number of paragraphs and a random number of subsections, which are themselves sections but without a subsections field (the boundaries of this random number were tuned in the script). We decided to maintain only a level of subsections to have a more concise document that satisfies the requirements. Secondly, a (bounded) random number of sections were added to each paper document of the original dataset.

URLs and captions Images URLs have been generated from www.mockaroo.com, while captions are from the `Summary` field of the same dataset `Reviews.csv` used to generate the sections. We wrote a Python script to add both values to the original dataset.

Venue The `venue` field was generated from the `venue.raw` given by the initial dataset. In the version of the `dblp` that we downloaded the venue was a composed field with a lot of information, such as the `raw`, the `name`, the `_id`, and the `type`. We chose to maintain only the `raw`, which encoded the more interesting data and we assigned it to the field named `venue`.

9.2. Upload process

The upload of the data was quite straightforward since our dataset was already in JSON format. Since the importing feature of MongoDB was much more efficient than Neo4j we were able to load a much bigger dataset containing 56K documents.

After uploading the data we performed the following filters:

- We deleted some fields from all the documents because we were not interested in maintaining all the data, but just the ones that matched more or less the chosen structure for the collection. We used the `$unset` command to remove specific fields, such as the `pdf` and the `year`, that was superfluous having the field `date`. An example of this procedure is reported in the third command of the following section.
- We checked all the fields to remove empty fields, so we cleaned the dataset by removing uninteresting values. We executed this cleaning part firstly by checking if the field equals the empty value `""` and then using the `$unset` command on the respective element. As before we can see an example of this procedure also in the third command of the following section.
- We executed some additional commands to filter the data, from the `FILTERING` option of `MongoDB Compass`, such as the following one:

```
{$and: [
    { title: { $not: { $regex: '^[1-9]' } }},
    { page_start: { $regex: '^[0-9]+$' }},
    { page_end: { $regex: '^[0-9]+$' }},
    { volume : {$not : { $eq : "null"}}},
    { abstract: {$not : {$eq : "null"}}},
]}
```

Controls similar to the one just reported can be executed at any time, for instance after acquiring some new data from an external source, to have only documents with interesting values.

10 | Commands and queries

10.1. Commands

1. **Inserting a document**
   To insert in our database only one document at a time, we use the following function `insertOne()`, which takes in input one document and inserts it in the DB. If the document does not specify an `_id` field, as is our case, then mongod[1] will add the `_id` field and assign a unique `ObjectId()` for the document before inserting it. Since when importing this dataset the `_id` were defined as String we decided to explicit the `_id` creation so we can use the `toString()` method to cast it to a String type and be consistent with the data we loaded from the dataset. As shown below, for this example we have retrieved a real-world paper and shaped its information to our document structure. Firstly, we searched for the author by name and since we didn't find him, we created a new `_id` for him. For text formatting reasons in the preview below, we have omitted the values of the `abstract` and `sections` fields. The `references` field is an empty array because all the referenced papers by this one are not present in our DB.

```
db.bibliography.insertOne(
{
    _id: ObjectId().toString(),
    title: "Reinforcement Learning for Improving Agent Design",
    authors: [
        {
            _id: ObjectId().toString(),
            name: "David Ha",
            email: "hadavid@google.com",
            bio: "AI resercher at Google Brain, based in Tokyo,
   Japan",
            org: "Google Brain"
        }
    ],
```

[1] mongod is the primary daemon process for the MongoDB system

```
    abstract: "...",
    keywords: ["computer science", "ai", "reinforcement learning",
   "cumulative reward", "joint learning"],
    fos: ["computer science", "ai", "reinforcement learning"],
    publication_type: "Journal",
    publisher: "MIT Press Direct",
    venue: "Artificial Life",
    volume: 25,
    issue: 4,
    issn: "352-365",
    date: ISODate('2019-12-02T10:49:36.000'),
    page_start: 23,
    page_end: 46,
    lang: "en",
    doi: "doi.org/10.1162/artl_a_00301",
    url: [
        "https://arxiv.org/abs/1810.03779",
        "https://arxiv.org/abs/1810.03779v3",
        "https://doi.org/10.48550/arXiv.1810.03779"],
    references: [],
    sections: []
})
```

An easy way to know if the papers referenced by the one we are about to insert are present in the database is to query it using the DOIs of the referenced paper (like is shown in the code snippet below) and then get the `_id` values to be added to the references.

```
db.bibliography.aggregate([{
    $match: {
        doi: {$in: [
            "10.1109/CMPSAC.2002.1044548",
            "10.1007/s11704-011-0127-6",
            "10.1016/j.datak.2008.09.003"
            ]}}}, {
    $project: {"_id": 1}}
])
```

Using the database structure presented earlier, we realized that managing the data could not be so straightforward, especially when a new document insertion is required. These drawbacks are due to the data sources we had because only a small percentage of the papers had the values for DOI and ORCID but we would have shrunk too much our dataset keeping only those articles. In a further improved version, some precautions could be taken to have a more clean and easy-to-use database

if the DOI field is known for all documents and the ORCID for all authors.

With the above assumptions, we could use the DOI for the references because it is a globally unique and independent identifier usable in any database implementation. In this way, it would be easier to insert a new document because the new paper could be added to the collection even if the references are not present in the dataset, and the related papers could be inserted afterward. Likewise, using the ORCID as the author identifier would also make it easier to add new documents because the ORCID has the same global uniqueness property. It would not be necessary to check if the author is already present in the database to set the same id, risking creating inconsistency, but we could insert it just as it is.

Despite the further checks required in our implementation, when adding the document, we decided to avoid imposing too many limitations on the structure of the data. If the papers have the DOI and the authors have the ORCID is easier to search the collection, but we cannot guarantee the presence of these fields, so to have a more flexible database, we accept papers without DOI and ORCID. A less bounded dataset leads to the necessity of adding some checks to maintain data consistency and requires being more careful when inserting data to avoid duplication.

2. **Update a single document**
   The command `updateOne` allows one to search for a document that matches a specific filter and modifies it accordingly to the update rules expressed inside the query. In general, the command modifies only the first found document, but if no document matches the search, then no changes are applied to the database.

   The `$set` operator sets a new value for the specified attribute or creates it if it does not exist. The `$unset` operator deletes a specified document attribute if the document has the field otherwise, nothing changes. The following command might be useful when new data regarding papers become available or when some information needs to be updated due to mistakes. The command allows to find a document with the field `_id` equal to `"53e997bab7602d9701fa09cb"` and changes its field `publication_type` to `"Conference"` because this is its actual type and also changes its field `publisher` to `"Association of Forensic Document Examiners"` with the command `$set`. This operator is also used to create the field `location` to which is assigned the value `"Grenoble, France"`. The update part of the query also allows the removal of the fields `page_start` and `page_end` by using the command `$unset`.

```
1 db.bibliography.updateOne( {
```

```
    _id: "53e997bab7602d9701fa09cb" }, [ {
        $set: { publication_type: "Conference" } }, {
        $set: { publisher: "Association of Forensic Document
   Examiners" } }, {
        $set: { location: "Grenoble, France" } }, {
        $unset: [
            "page_start",
            "page_end" ] } ] )
```

```
> db.bibliography.updateOne(
    {"_id": "53e997bab7602d9701fa09cb"},
    [
        {$set: {"publication_type": "Conference"}},
        {$set: {"publisher": "Association of Forensic Document Examiners"}},
        {$set: {"location": "Grenoble, France"}},
        {$unset: ["page_start", "page_end"]}
    ]
)
< { acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0 }
```

In this case, only one document could have been found because the field `_id` is the unique identifier. It is necessary to be careful in updating the database in order to avoid updating the wrong documents, deleting fields, and changing important values.

Another important use case of `updateOne` is associated with the need to replace a document with a different one. This exchange can be done using the operator `$replaceWith` and specifying the document to remove and the one that has to be put inside the database instead of it.

3. **Remove fields**

   Uploading the dataset, as we previously explained in the chapter about the upload process, we ended up with a lot of data, some of which were not of our interest. Using the following command we consider all the data entries of the collection `bibliography` and without any filtering, we just remove the `year` field from all the documents. We decided to remove this field because we already have the `date` so this is unnecessary.

```
db.bibliography.updateMany( {
    }, {
    $unset: { year: "" } } )
```

In order to maintain only interesting values about the papers, we also remove the empty fields. It doesn't make sense to keep fields that are equal to an empty value, so, for instance, if `doi` is just an empty string, we delete it from the respective document. We keep only useful data that respect the structure of the dataset we want to work with to extract knowledge. We executed the following command for all the fields for the sake of having a clear collection.

```
db.bibliography.updateMany( {
    doi: { $eq: "" } }, {
    $unset: { doi: "" } } )
```

4. **Delete a group of documents**

   The command `deleteMany()` deletes a group of documents that match the condition given inside the specified filter. In this case, the documents that have as date one that occurs before the `1950` are considered obsolete, so we want to delete them. For this reason, it's enough to execute a check on the field `date`, that is of date type, so we have to specify the `ISODate` format and delete all the documents previous to `'1950-01-01T00:00:00.000Z'`.

```
db.bibliography.deleteMany( {
    date: { $lt: ISODate("1950-01-01T00:00:00.000Z") } } )
```

After executing the command successfully, `MongoDB` acknowledges the write operation and returns the number of documents that were eliminated from the collection.

```
> db.bibliography.deleteMany(
        {date: {$lt: ISODate('1950-01-01T00:00:00.000Z')}}
  )
< { acknowledged: true, deletedCount: 3151 }
```

5. **Update a group of documents**

   The command `updateMany` allows the modification of multiple documents at the same time. Since in the database some articles have `"Elsevier"` as `publisher`, and some others have `"Elsevier Ltd."`, and since both values represent the same publisher, we want to merge them. With the following command, all documents that have value `"Elsevier Ltd."` in the field `publisher` are retrieved, then the value is changed in `"Elsevier"`.

```
db.bibliography.updateMany( {
    publisher: "Elsevier Ltd." }, {
    $set: { publisher: "Elsevier" } } )
```

```
> db.bibliography.updateMany(
    {publisher: "Elsevier Ltd."},
    {$set: {publisher: "Elsevier"}}
  )
< { acknowledged: true,
    insertedId: null,
    matchedCount: 1774,
    modifiedCount: 1774,
    upsertedCount: 0 }
```

10.2. Queries

1. **Papers published in a certain period on a specific type of publication**
   The query's goal is to find all the papers published in a particular time frame delimited by two dates (included) of a type of publication, such as a journal. More specifically, the search is conducted on papers published between January 2005 and December 2010 in journals. To be compliant with the data type of the `date` field, the date must be specified in a `ISODate` format. In the projection, we only show `title` and `venue` of a paper, that is the name of the Journal, and we suppress the field `_id` because not informative.

```
db.bibliography.find( {
    $and: [ {
        date: { $gte: ISODate('2005-01-01T00:00:00.000Z') } }, {
        date: { $lte: ISODate('2010-12-31T00:00:00.000Z') } }, {
        publication_type: "Journal" } ] }, {
    _id: 0,
    title: 1,
    venue: 1 } ).limit(5)
```

```
{ title: 'MESHMdl', venue: 'Pervasive and Mobile Computing' }
{ title: 'Detection and Recovery Techniques for Database Corruption',
  venue: 'IEEE Trans. Knowl. Data Eng.' }
{ title: 'General and Interval Type-2 Fuzzy Face-Space Approach to Emotion Recognition.',
  venue: 'IEEE T. Systems, Man, and Cybernetics: Systems' }
{ title: 'Existence of resolvable H-designs with group sizes 2, 3, 4 and 6',
  venue: 'Des. Codes Cryptography' }
{ title: 'Particle shape analysis of volcanic clast samples with the Matlab tool MORPHEO',
  venue: 'Computers & Geosciences' }
```

Performance: It's obvious to see that this query iterates through the values of a field to filter some documents. If the time range inserted to do the query is small,

we can say that the query is highly selective, that is, a query that returns only a few documents from the whole set of scanned documents. Selective queries are the ones that make the best usage of indexes, so we thought to do a comparison test of the performance of the query with and without an index on the date.
To have a more meaningful result for the test we removed the `db.collection.limit()` function, otherwise, the automatic rearrangement of the queries would limit the query to stop as soon as it matches 5 correct instances. With this limit on the query, the load on the system would be so light that we would see a negligible difference between the query run with and without the indexes.
To get information about the query performance we use `db.collection.explain()`, which returns information about the query plan. Passing the `executionStats` argument to this method makes it return an additional section where a set of useful performance markers are shown.

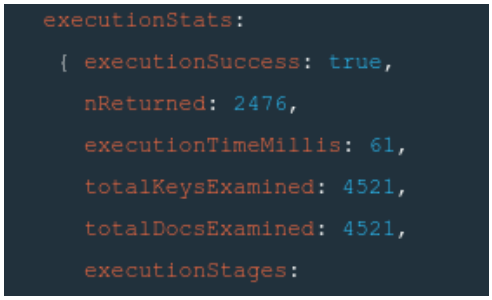
Results We first try running the query without an index on the field `date`. Using the method `explain` with the argument `executionStats` we get the following result (we show only a selection of the markers present in the `executionStats` section, the ones meant for this discussion):

```
executionStats:
 { executionSuccess: true,
   nReturned: 2476,
   executionTimeMillis: 360,
   totalKeysExamined: 0,
   totalDocsExamined: 56453,
```

MongoDB runs the query optimizer to choose the winning plan, executes the winning plan to completion, and returns statistics describing the execution of the winning plan. In the reported results we can see different parameters that explain the query, and among these, we focus our comparison, especially on the following:

- `nReturned`: number of documents that match the query condition
- `executionTimeMillis`: total time in milliseconds required for query plan selection and query execution
- `totalKeysExamined`: number of index entries scanned
- `totalDocsExamined`: number of documents examined during query execution

We can clearly see that the documents passing the filter are about 4% of the total, so we can say that this query is selective enough and can proceed to test it with the addition of an index. We'd like also to point out that the `totalKeysExamined` parameter has a value of 0, meaning that this query has not performed a search on indexes. Meanwhile, the `totalDocsExamined` shows that all the documents in the DB were examined. We proceed to create an index on the field `date` and we run the query again. This time the execution plan and stats return different values:



We can immediately notice that the execution time has gone down to 61ms from the 360ms of before, a performance gain of more than 5x. Since the indexes are stored in an ordered way the query executioner can iterate through the dates with a much faster binary search algorithm. This means we have to do a much smaller number of accesses to the keys of the database, as shown by the value of `totalKeysExamined` parameter in the figure. In this particular case, we also have to access the DB (as shown by the `totalDcosExamined` parameter) because the `publication_type` field is not stored in an index, so we have to retrieve the value of this field from the document itself to complete the filtering condition[2]. Nonetheless, the much lower number of accesses to DB provided by the index on `date` is sufficient to experience a significant gain in performance.

2. **Search a given word inside the title of papers written on books and published by a specific publisher**
   MongoDB supports query operations that perform a text search, but to enable this functionality is necessary to define a textual index that makes the search possible.

   To achieve more efficient execution of queries, we can define indexes, which are data structures that store useful information to speed up the search. When MongoDB executes a query without indexes, it has to evaluate the conditions on all the documents of the considered collection. If an index is defined and used, MongoDB

[2]This type of query, where not all the field needed to compute the query are stored as indexes are called "not covered queries"

can limit the number of analyzed documents and requires less time to compute the results. The following command is necessary for creating indexes, and it is essential before executing the following query.

```
db.bibliography.createIndex( {
    title: "text" } )
```

```
> db.bibliography.createIndex( {
      title: "text" } )
< 'title_text'
```

Once the index is created, the query can be run. It returns the titles of the documents whose title presents the word `"software"` and that were published in a book by the publisher `"Elsevier"`.

```
db.bibliography.find( {
    $text: { $search: "software" },
    publication_type: "Book",
    publisher: "Elsevier" }, {
    title: 1 } ).limit(5)
```

```
< { _id: '53e9980eb7602d970202579c',
    title: 'Energy Aware Software' }
  { _id: '53e9997eb7602d97021c25b7',
    title: 'The Art and Science of Software Architecture' }
  { _id: '53e99976b7602d97021b5bdf',
    title: 'A roadmap for software maintainability measurement' }
  { _id: '53e99860b7602d970209c51f',
    title: 'On the analysis of evolution of software artefacts and programs' }
  { _id: '53e99998b7602d97021da4d6',
    title: 'S-RaP: a concurrent, evolutionary software prototyping process' }
```

3. **Visualize the common values between the field of study and keywords in each document**
   The following query extracts all those documents with at least one keyword as their field of study.

   In particular, we project `title`, `fos`, `keywords`, and their sizes, and then we find all those documents that contain more than thirty keywords, and more than fifteen fields of study. We select such a huge number of fields of study and keywords because we are interested in papers that are involved in research that can be important in many areas.

   Finally, we project again `title` and the intersection set between keywords and fields of study to extract and show the ones that match.

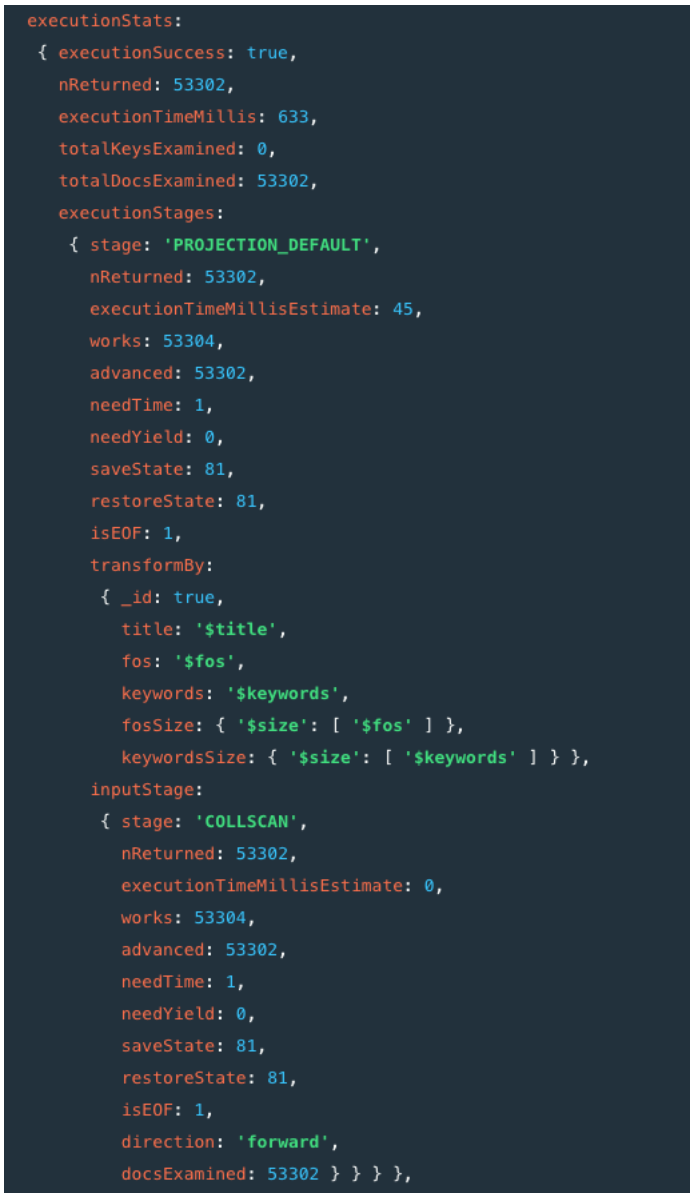
```
db.bibliography.aggregate( [ {
    $project: {
        title: "$title",
        fos: "$fos",
        keywords:"$keywords",
        fosSize: { $size: "$fos" },
        keywordsSize: { $size: "$keywords" } } }, {
    $match: { $and: [ {
        keywordsSize: { $gt: 30 },
        fosSize: { $gt: 15 } } ] } }, {
    $project: {
        title: "$title",
        intersection: { $setIntersection: [
            "$fos",
            "$keywords" ] } } } ] )
```

Performance: Let's first consider the query just explained with the relative outcome of the execution, using `explain("executionStats")` in order to analyze its performance:

```
db.bibliography.aggregate( [ {
    $project: {
        title: "$title",
```

```
4           fos: "$fos",
5           keywords:"$keywords",
6           fosSize: { $size: "$fos" },
7           keywordsSize: { $size: "$keywords" } } }, {
8       $match: { $and: [ {
9           keywordsSize: { $gt: 30 },
10          fosSize: { $gt: 15 } } ] } }, {
11      $project: {
12          title: "$title",
13          intersection: { $setIntersection: [
14              "$fos",
15              "$keywords" ] } } } ] ).explain("executionStats")
```

The query firstly scrolls through all the documents, just selecting some fields that we want to project and calculating the size of the `fos` and of the `keywords`. In the execution, this is shown by the fact that all the documents that are examined at this first step are also returned. This procedure has as `explain.executionStats.executionTimeMillis` a value of `633 ms` and doesn't reduce the papers to work on. We can also see from the `explain.executionStats. totalKeysExamined` that the number of index entries is zero because we don't use any indexing on the fields involved.



```
executionStats:
 { executionSuccess: true,
   nReturned: 53302,
   executionTimeMillis: 633,
   totalKeysExamined: 0,
   totalDocsExamined: 53302,
   executionStages:
    { stage: 'PROJECTION_DEFAULT',
      nReturned: 53302,
      executionTimeMillisEstimate: 45,
      works: 53304,
      advanced: 53302,
      needTime: 1,
      needYield: 0,
      saveState: 81,
      restoreState: 81,
      isEOF: 1,
      transformBy:
       { _id: true,
         title: '$title',
         fos: '$fos',
         keywords: '$keywords',
         fosSize: { '$size': [ '$fos' ] },
         keywordsSize: { '$size': [ '$keywords' ] } },
      inputStage:
       { stage: 'COLLSCAN',
         nReturned: 53302,
         executionTimeMillisEstimate: 0,
         works: 53304,
         advanced: 53302,
         needTime: 1,
         needYield: 0,
         saveState: 81,
         restoreState: 81,
         isEOF: 1,
         direction: 'forward',
         docsExamined: 53302 } } } },
```

After the scrolling, we execute a `match` that narrows down the papers and reduces the computation. As follows, we can see that this part of the query requires also `610 ms`, given the fact that has to analyze again all the documents, but after the execution, only `9` matching papers are returned, and shown with the projection.

```
{ '$match': { '$and': [ { keywordsSize: { '$gt': 30 } }, { fosSize: { '$gt': 15 } } ] },
  nReturned: 9,
  executionTimeMillisEstimate: 610 },
{ '$project':
   { _id: true,
     title: '$title',
     intersection: { '$setIntersection': [ '$fos', '$keywords' ] } },
  nReturned: 9,
  executionTimeMillisEstimate: 610 } ],
```

Given the fact that the query examined so far has to scroll over all the documents twice, we thought of the following alternative, that first executes the `match` part and then shows the intersection between the `fos` and the `keywords`, without using two projections.

```
db.bibliography.aggregate( [
    {$match: {
        $expr: {
            $and: [ {
                $gt: [{$size: "$keywords"}, 30 ] }, {
                $gt: [{$size: "$fos"}, 15]
                } ] } } } , {
    $project: {
        title: "$title",
        intersection: { $setIntersection: [
            "$fos",
            "$keywords" ] } } } ] ).explain("executionStats")
```

As before the query starts examining all the documents of the collection and we have no index entries, not having any index on the involved fields. We can see some important differences from the previous procedure looking at mainly two fields of the `explain.executionStats` reported below. We can understand from the `explain.executionStats.executionTimeMillis` that the execution time is lower, requiring only `246 ms`, doing the filtering of the papers as the first action of the query. Afterward, the projection of the result required only `24 ms`, because the documents were already narrowed down. In fact, we can notice that the `match` examined all the collections, but returned only `9` matches, exactly like before, only now this operation was done before any other, so it improved the overall execution.

```
executionStats:
 { executionSuccess: true,
   nReturned: 9,
   executionTimeMillis: 246,
   totalKeysExamined: 0,
   totalDocsExamined: 53302,
   executionStages:
    { stage: 'PROJECTION_DEFAULT',
      nReturned: 9,
      executionTimeMillisEstimate: 24,
      works: 53304,
      advanced: 9,
      needTime: 53294,
      needYield: 0,
      saveState: 53,
      restoreState: 53,
      isEOF: 1,
      transformBy:
       { _id: true,
         title: '$title',
         intersection: { '$setIntersection': [ '$fos', '$keywords' ] } },
      inputStage:
       { stage: 'COLLSCAN',
         filter:
          { '$expr':
             { '$and':
                [ { '$gt': [ { '$size': [ '$keywords' ] }, { '$const': 30 } ] },
                  { '$gt': [ { '$size': [ '$fos' ] }, { '$const': 15 } ] } ] } },
         nReturned: 9,
         executionTimeMillisEstimate: 24,
         works: 53304,
         advanced: 9,
         needTime: 53294,
         needYield: 0,
         saveState: 53,
         restoreState: 53,
         isEOF: 1,
         direction: 'forward',
         docsExamined: 53302 } } },
```


In conclusion, this alternative query is more efficient than the previous one, requiring less time to compute. From this analysis we noticed that is better to execute the filtering part, the `match`, as soon as possible in the query, in order to reduce the number of documents we have to work on, speeding up the process since the operations are executed like a pipeline.

4. **Organizations that held more conferences in the same location**
   The following query matches all the papers that are of type `Conference`. We select only the events held by the `"IEEE Global Telecommunications Conference (Globecom)"` then for each author involved in the conference, we check the affil-

iation. After every step of the query, we obtain a new temporary collection of documents. For instance, with the `unwind` command, we deconstruct the array of `authors` from the input papers to output a new document for each element and, in this way, we create a much bigger dataset.

With this query, we are interested in knowing how many times an organization held a conference in a specific location through one of its members. Regrouping the data by conference location and affiliation of the author involved, we can count the number of times the organization took part in an event held there. Finally, the result is sorted by the number of engagements and limited only to the first more active organizations.

```
db.bibliography.aggregate( [ {
    $match: { $and: [ {
        publication_type: { $eq: "Conference" } }, {
        venue: { $eq: "IEEE Global Telecommunications Conference (
   Globecom)" } } ] } }, {
    $unwind: "$authors" }, {
    $group: {
        _id : {
            location: "$location",
            organization: "$authors.org" },
        locationPerOrganization: { $sum: 1 } } }, {
    $project: {
        location: "$location",
        author: "$organization",
        locPerOrgConference: "$locationPerOrganization" } }, {
    $sort: { locPerOrgConference : -1 } }, {
    $limit: 5 } ] )
```

```
{ _id:
   { location: 'Melbourne, Australia',
     organization: 'Univ Calif San Diego, Dept Elect & Comp Engn, La Jolla, CA 92093 USA' },
  locPerOrgConference: 6 }
{ _id: { location: 'San Francisco, USA' },
  locPerOrgConference: 5 }
{ _id:
   { location: 'Copenhagen, Denmark',
     organization: 'Univ Padua, Dept Informat Engn, Padua, Italy' },
  locPerOrgConference: 4 }
{ _id:
   { location: 'Kuala Lumpur, Malaysia',
     organization: 'Ericsson Res Canada, Montreal H4P 2N2, PQ, Canada' },
  locPerOrgConference: 4 }
{ _id:
   { location: 'Sao Paulo, Brazil',
     organization: 'Univ Waterloo, Dept Elect & Comp Engn, Waterloo, ON N2L 3G1, Canada' },
  locPerOrgConference: 4 }
```

5. **Papers with a pattern structure**

   With this query, we count the number of documents that satisfy a given pattern specified in the `$match` part of the query. In this example, we are searching for papers with four sections, and at least one must have four paragraphs, four figures, and one subsection with just one paragraph. Using `$size`, it's possible to check the dimension of an array, while using `$elemMatch` is possible to define the criteria that at least one document in an array must satisfy. Finally, through the dot notation, we can access a field of an embedded document.

```
db.bibliography.aggregate( [ {
    $match: { $and: [ {
        sections: { $size: 4 } }, {
        sections: { $elemMatch: {
            paragraphs: { $size: 4 },
            figures: { $size:4 },
            subsections: { $size: 1 },
            "subsections.paragraphs": { $size: 1 } } } } ] } }, {
    $group: {
        _id: true,
        count: { $sum: 1 } } } ] )
```

```
> db.bibliography.aggregate( [ {
    $match: { $and: [ {
        sections: { $size: 4 } }, {
        sections: { $elemMatch: {
            paragraphs: { $size: 4 },
            figures: { $size:4 },
            subsections: { $size: 1 },
            "subsections.paragraphs": { $size: 1 } } } } ] } }, {
    $group: {
        _id: true,
        count: { $sum: 1 } } } ] );
< { _id: true, count: 284 }
```

The following query is a slight change of the previous one just to retrieve a paper with the wanted structure to show that indeed the query works.

```
> db.bibliography.findOne( {
   $and: [ {
        sections: { $size: 4 } }, {
        sections: { $elemMatch: {
            paragraphs: { $size: 4 },
            figures: { $size:4 },
            subsections: { $size: 1 },
            "subsections.paragraphs": { $size: 1 } } } } ] } , {
   title: 1, sections: 1 } )
```

6. **Number of papers published by the current staff of an organization**
   What we want to achieve with this query is to know how many papers have been published by the researchers working at a certain organization. The output list is ordered in descending order, showing the organizations whose researchers have published the most. Notice that a paper published by authors from different organizations is counted for all the organizations involved. In the first part of the query, we unwind the documents by `authors` and do a filtering of inconsistent data, since the focus of this query is to know which organizations have published the most. Secondly, we proceed by grouping documents by the author affiliation (`authors.org`) and the paper itself. This operation is needed to filter out the documents referring to the same paper and written by authors that are affiliated with the same organization. If we hadn't done this, the same paper would have been counted multiple times, for all the authors who participated in writing it and that are part of the same organization, inflating that organization's contribution. Notice that in the result an organization with name `"Corresponding author."` appears, probably this value has been used by the maintainer of the DB to denote independent authors or authors for whom the affiliated organization was not known.

```
db.bibliography.aggregate( [ {
    $unwind: "$authors" }, {
    $match: { $and: [ {
        "authors.org": { $ne: null } }, {
        "authors.org": { $ne: '' } } ] } }, {
    $group: {
        _id: {
            org: "$authors.org",
            p_id: "$_id"} } }, {
    $group: {
        _id: "$_id.org",
```

```
        papers: {$sum: 1} } }, {
    $sort: { papers: -1 } }, {
    $limit: 5 } ] )
```

```
{ organization: 'Corresponding author.', numberOfPapers: 318 }
{ organization: 'IEEE', numberOfPapers: 73 }
{ organization: 'Microsoft Research', numberOfPapers: 61 }
{ organization: 'Carnegie Mellon University, Pittsburgh, PA',
  numberOfPapers: 55 }
{ organization: 'Carnegie Mellon University',
  numberOfPapers: 53 }
```

7. **Number of pages published in Journals by authors affiliated with a certain organization**

   This query is similar to the one above, but here we want to retrieve which organizations have contributed the most in writing papers published in journals. More specifically, we sum the number of pages written in a specific journal by authors who are currently affiliated with that specific organization. This objective is obtained by first unwinding the authors and filtering them to remove some inconsistent data. Secondly, we proceed by grouping documents by the author affiliation (`authors.org`) and the paper itself (the other fields of `venue` and `pages` do not contribute to the grouping because they are a single field of the paper document). This operation achieves the same goal as the one described in the query before. In the second `$group` stage, we group the documents by organization and journal, and we have an accumulator to sum all the pages written by the staff of that organization in that particular journal. Finally, the last stages are only for the projection and pretty printing of the output.

```
db.bibliography.aggregate( [ {
    $unwind: "$authors" }, {
    $match: { $and: [ {
        "authors.org": { $ne: null } }, {
        "authors.org": { $ne: '' } }, {
        publication_type: { $eq: "Journal" } }, {
        $expr: { $lte: [
            "$page_start",
            "$page_end" ] } } ] } }, {
        $addFields: { pages: { $subtract: [
            "$page_end",
            "$page_start" ] } } }, {
        $group: { _id: {
            org: "$authors.org",
```

```
            paperId: "$_id",
            journal: "$venue",
            pages: "$pages" }, } }, {
        $group: {
        _id: {
            org: "$_id.org",
            journal: "$_id.journal" },
        pages: { $sum: "$_id.pages" } } }, {
    $sort: { pages: -1 } }, {
    $project: {
        organization: "$_id.org",
        journal: "$_id.journal",
        numberOfPages: "$pages" } }, {
    $project: { _id: 0 } }, {
    $limit: 5 } ] )
```

```
{ organization: 'Dept. of Commun. Eng., Nat. Central Univ., Taoyuan, Taiwan',
  journal: 'Selected Areas in Communications, IEEE Journal  ',
  numberOfPages: 1519 }
{ organization: 'Dept. of Electr. & Comput. Eng., Univ. of Maryland, College Park, MD, USA#TAB#',
  journal: 'Selected Areas in Communications, IEEE Journal  ',
  numberOfPages: 1519 }
{ organization: 'Corresponding author.',
  journal: 'Applied Mathematics and Computation',
  numberOfPages: 596 }
{ organization: 'university of edinburgh',
  journal: 'Electronic Notes in Theoretical Computer Science',
  numberOfPages: 418 }
{ organization: 'DI École Normale Supérieure, France',
  journal: 'Foundations and Trends® in Computer Graphics and Vision',
  numberOfPages: 200 }
```

8. **Couples of the field of study and keyword which appear more frequently within the papers**

   The query returns the association between fields of study and keywords which are more present within the database and how many times they appear together. Initial filtering is done in order to eliminate the papers which don't contain any keywords or any field of study. The two clauses for doing the filtering can be modified by increasing the required number of keywords or the required number of fields of study. This kind of change can be interesting in order to understand which fields of study and keywords become more relevant when the subset of the considered paper is different. The results are ordered by the number of occurrences of the couples `field of study` - `keyword` by decreasing order. This can be done by the operator `"$sort"`. We filter out the couples of fields of study and keywords with less than

100 in order to get rid of some misleading information due to not coupled elements. The threshold can be adjusted.

```
db.bibliography.aggregate( [ {
    $match: {
        $expr: { $gte: [ { $size: "$fos" }, 0 ] } } }, {
    $match: {
        $expr: { $gte: [ { $size: "$keywords" }, 0 ] } } }, {
    $unwind: { path: "$fos" } }, {
    $unwind: { path: "$keywords" } }, {
    $group: {
        _id: {
            fieldOfStudy: "$fos",
            keyword: "$keywords" },
        keywordsCount: { "$sum": 1 } } }, {
    $match: { "keywordsCount": { $gte: 100 } } }, {
    $sort: { keywordsCount: -1 } }, {
    $limit: 5 }, {
    $project: {
        fieldOfStudy: "$fos",
        keyword: "$keywords",
        score: "$keywordsCount" } } ] )
```

```
{ _id: { fieldOfStudy: 'computer science', keyword: 'data mining' },
  score: 1066 }
{ _id: { fieldOfStudy: 'computer science', keyword: 'internet' },
  score: 960 }
{ _id:
   { fieldOfStudy: 'computer science',
     keyword: 'computer science' },
  score: 844 }
{ _id: { fieldOfStudy: 'computer science', keyword: 'real time' },
  score: 798 }
{ _id:
   { fieldOfStudy: 'artificial intelligence',
     keyword: 'feature extraction' },
  score: 643 }
```

9. **Papers that reference old papers with the same publisher**
   In the beginning, we filter the documents retrieving only those that have a `publisher` then, we perform a join between papers and their references using `$lookup` with the condition that the `_id` of the paper must be in the `references` field of the other one. In the `$let` command we specify the variables to use in the pipeline that are related to the outer document and that are accessed using `$$`. Then we require both

papers to have the same publisher and that the referenced article was published at least 10 years before using `$dateDiff`. The `_id`, `title`, `publisher`, and `date` of the referenced papers that satisfy these conditions are added to the field called `referencedPapers` in the outer document. Lastly, we filter documents keeping only those with at least one joined document, we project only a few fields to make the answer more readable and to check that it is correct, and we limit the number of returned papers.

```
db.bibliography.aggregate( [ {
    $match: { publisher: { $exists: true } } }, {
    $lookup: {
        from: "bibliography",
        let: {
            pub: "$publisher",
            ref: "$references",
            d: "$date" },
        pipeline: [ {
            $match: { $expr: { $and: [ {
                $in: [
                    "$_id",
                    "$$ref" ] }, {
                $eq: [
                    "$publisher",
                    "$$pub" ] }, {
                $gte: [ {
                    $dateDiff: {
                        startDate: "$date",
                        endDate: "$$d",
                        unit: "year" } },
                    10 ] } ] } } }, {
            $project: {
                _id: 1,
                title: 1,
                publisher: 1,
                date: 1 } } ],
        as: "referencedPapers" } }, {
    $match: { "referencedPapers.0": { $exists: true } } }, {
    $project: {
        _id: 1,
        title: 1,
        publisher: 1,
        date: 1,
        referencedPapers: 1 } }, {
    $limit: 3 } ] )
```

```
{ _id: '53e997ddb7602d9701fd5406',
  title: 'An Asymmetric Edge Adaptive Filter for Depth Generation and Hole Filling in 3DTV',
  publisher: 'Springer London',
  date: 1987-05-11T05:40:26.000Z,
  referencedPapers:
   [ { _id: '53e99866b7602d97020a0cb7',
       title: 'Depth-image-based rendering for 3DTV service over T-DMB',
       publisher: 'Springer London',
       date: 1959-09-30T21:11:50.000Z } ] }
{ _id: '53e997e3b7602d9701fd8218',
  title: 'A Programming Language for Deriving Hypergraphs',
  publisher: 'Springer New York',
  date: 2016-04-24T22:35:04.000Z,
  referencedPapers:
   [ { _id: '53e9982cb7602d970204e40f',
       title: 'Programmed Graph Grammars',
       publisher: 'Springer New York',
       date: 1975-08-21T00:47:13.000Z } ] }
{ _id: '53e997e9b7602d9701fe48ad',
  title: 'Bagging Ranking Trees',
  publisher: 'Open Library of Humanities',
  date: 2017-02-14T07:18:45.000Z,
  referencedPapers:
   [ { _id: '53e99809b7602d970201e553',
       title: 'Comparing Partial Rankings',
       publisher: 'Open Library of Humanities',
       date: 2006-03-05T20:31:19.000Z } ] }
```


The `$lookup` can be very expensive from a computational point of view and require much time, so to speed the process up it is convenient, when possible, to reduce the input data to the join phase by filtering the documents. This can be done by searching for a specific author, title, publisher, or something meaningful in a `$match` condition before the `$lookup`. This is possible since all the operations in a query are executed on the database as a pipeline.

10. **Recommended books given one book read**

    The query returns the book considered the nearest to the given one, in this case, `"Pattern Recognition Letters"`. One book is suggested if it has a field of study in common with the book `"Pattern Recognition Letters"`. The similarity between books is then evaluated in terms of the number of equal keywords associated with the papers belonging to both considered books.

```
db.bibliography.aggregate( [ {
    $match: { publication_type: "Book" } }, {
    $unwind: { path: "$fos" } }, {
    $unwind: { path: "$keywords" } }, {
    $lookup: {
        from: "bibliography",
```

```
        localField: "fos",
        foreignField: "fos",
        as: "otherPaper" } }, {
    $match: { venue: "Pattern Recognition Letters" } }, {
    $unwind: { path: "$otherPaper" } }, {
    $match: { $expr: { $ne: [
        "$_id",
        "$otherPaper._id" ] } } }, {
    $match: { $expr: { $ne: [
        "$venue",
        "$otherPaper.venue" ] } } }, {
    $unwind: { path: "$otherPaper.keywords" } }, {
    $match: { $expr: { $eq: [
        "$keywords",
        "$otherPaper.keywords" ] } } }, {
    $group: {
        _id: { suggestedBook: "$otherPaper.venue" },
        keywordsCount: { $sum: 1 } } }, {
    $sort: { keywordsCount: -1 } }, {
    $limit: 10 }, {
    $project: {
        title: "$suggestedBook",
        similarity: "$keywordsCount" } } ] )
```

```
{ _id: { suggestedBook: 'Pattern Recognition' },
  similarity: 58 }
{ _id: { suggestedBook: 'ICIP' }, similarity: 28 }
{ _id: { suggestedBook: 'Neurocomputing' }, similarity: 20 }
{ _id: { suggestedBook: 'Expert Syst. Appl.' }, similarity: 17 }
{ _id: { suggestedBook: 'Inf. Sci.' }, similarity: 13 }
{ _id: { suggestedBook: 'ISNN (2)' }, similarity: 13 }
{ _id: { suggestedBook: 'IEEE J. Biomedical and Health Informatics' },
  similarity: 11 }
{ _id: { suggestedBook: 'ICASSP' }, similarity: 10 }
{ _id: { suggestedBook: 'IEEE Trans. Pattern Anal. Mach. Intell.' },
  similarity: 10 }
{ _id: { suggestedBook: 'Neural Computing and Applications' },
  similarity: 10 }
```

11. **Authors who have referenced themselves the most within their papers**
    The query returns the authors who have referenced themselves the most. An author does a self-reference when the considered paper presents a reference to another paper the author has written. The results are the name of the authors and the number of times they have made a self-reference, the output is ordered using this number.

```
db.bibliography.aggregate( [ {
    $unwind: { path: "$references" } }, {
    $lookup: {
        from: "bibliography",
        localField: "references",
        foreignField: "_id",
        as: "referencedPaper" } }, {
    $match: { $expr: { $gt: [ {
        $size: "$referencedPaper" }, 0 ] } } }, {
    $unwind: { path: "$referencedPaper" } }, {
    $match: { $expr: { $ne: [
        "$_id",
        "$otherPaper._id" ] } } }, {
    $match: { "authors": { $exists: true } } }, {
    $match: { "referencedPaper.authors": { $exists: true } } }, {
    $unwind: { path: "$authors" } }, {
    $unwind: { path: "$referencedPaper.authors" } }, {
    $match: { $expr: { $eq: [
        "$authors", "$referencedPaper.authors" ] } } }, {
    $group: {
        _id: {
            authorId: "$authors._id",
            authorName: "$authors.name"},
        aut: { $sum: 1 } } }, {
    $sort: {"aut": -1} }, {
    $limit: 10 }, {
    $project: {
        "_id": 0,
        "name": "$_id.authorName",
        "autoreference": "$aut"} } ] )
```

```
{ name: 'K. V. S. Prasad', autoreference: 4 }
{ name: 'Cheng-Lin Liu', autoreference: 4 }
{ name: 'Sam Toueg', autoreference: 4 }
{ name: 'Qiu-Feng Wang', autoreference: 4 }
{ name: 'Yingxu Wang', autoreference: 4 }
{ name: 'Kathleen Romanik', autoreference: 4 }
{ name: 'Martín Abadi', autoreference: 3 }
{ name: 'Michal Parnas', autoreference: 3 }
{ name: 'Kazuhisa Makino', autoreference: 3 }
{ name: 'Karel Hrbacek', autoreference: 3 }
```

Part IV

Spark

11 | Introduction to the problem

In this part, we implement a bibliographic database relying on the infrastructure provided by Apache Spark. Apache Spark is a scalable platform for performing complex analytics. It manages data within a distributed system and this guarantees fast processing, fault tolerance, and high scalability. The framework allows using different technologies for implementing the database. We use a logic paradigm that looks like the relational one. We rely on the `Dataframe` object, which can be imagined as a table containing data but with a lot of enhancing features. We use `pyspark`, which is the Python library for creating and managing the database in Spark, in order to translate the conceptual model defined in chapter 2. Articles, authors, and various types of publications become the tables containing the instances of the data. Apache Spark is a good approach when dealing with huge amounts of data and in the real world, a bibliography database contains a lot of papers. On the other hand, in our application, we use a small subset of the published papers and we are running Spark only on one machine so we are not fully exploiting the real power of the platform. In treating this problem from a new point of view, though, we try to take advantage of the functions available in `pyspark` and of its features as much as we can.

12 | Dataset structure

In this part, we implemented a database with a very similar structure to the ER model described in chapter 2. We decided to create six different tables:

- **df_papers** is the table containing all the attributes concerning a single paper. With respect to the ER model here we have `publication_type` and `publication_id` that are necessary in order to do the join with a certain type of publication; `page_start`; `page_end`; `date` instead of `year`; and `references` which is a list that substitutes the corresponding relationship; we also have the lists of `fos` and `keywords` related to the paper and the other attributes identified in the modeling part of the bibliography;
- **df_aut** is the table related to authors and contains `email` and `bio` in addition to the ER model;
- **df_aff** is the table with the author's affiliation when the paper was written, this table maps the `writes` relationship;
- **df_journals** is the table with the journals;
- **df_books** is the table with the books;
- **df_conferences** is the table with the conferences.

  We decided to represent the entity Publication with the sub-entities, for this reason, all contain `publication_id` and `publication_type` that are used in the join with `df_papers`. The attribute `name` in Publication in the ER model, as well as in other parts of the document, is called `venue`.

The following table shows all the fields associated with their respective type, meaning, and table.

Field name	Type	Description	Table
publication_id	long	publication id	df_papers, df_books, df_journals, df_conferences
paper_id	string	paper ID	df_papers, df_aff
title	string	paper title	df_papers
keywords	array of string	paper keywords	df_papers
fos	array of string	paper fields of study	df_papers
references	array of string	paper references	df_papers
page_start	integer	start page	df_papers
page_end	integer	end page	df_papers
lang	string	paper language	df_papers
doi	string	paper DOI	df_papers
url	array of string	paper URLs	df_papers
abstract	string	paper abstract	df_papers
publication_type	string	paper type	df_papers
date	timestamp	publication date	df_papers
author_id	string	author ID	df_aut, df_aff
name	string	author name	df_aut
email	string	author email	df_aut
bio	string	author short bio	df_aut
organization	string	author affiliation	df_aff
venue	string	publication name	df_books, df_journals, df_conferences
isbn	string	book ISBN	df_books
publisher	string	publisher name	df_books, df_journals
volume	integer	journal volume	df_journals
issue	integer	journal issue	df_journals
issn	string	journal ISSN	df_journals
location	string	conference location	df_conferences

Table 12.1: Description of the fields with related type, meaning, and table.

Below we provide the actual schema of each table obtained by invoking the function `printSchema()`.

```
root
 |-- publication_id: long (nullable = false)
 |-- paper_id: string (nullable = true)
 |-- title: string (nullable = true)
 |-- keywords: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- fos: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- references: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- page_start: integer (nullable = true)
 |-- page_end: integer (nullable = true)
 |-- lang: string (nullable = true)
 |-- doi: string (nullable = true)
 |-- url: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- abstract: string (nullable = true)
 |-- publication_type: string (nullable = true)
 |-- date: timestamp (nullable = true)
```

Figure 12.1: Paper table - df_papers.

```
root
 |-- author_id: string (nullable = true)
 |-- name: string (nullable = true)
 |-- email: string (nullable = true)
 |-- bio: string (nullable = true)
```

Figure 12.2: Author table - df_aut.

```
root
 |-- paper_id: string (nullable = true)
 |-- author_id: string (nullable = true)
 |-- organization: string (nullable = true)
```

Figure 12.3: Affiliation table - df_aff.

```
root
 |-- venue: string (nullable = true)
 |-- volume: integer (nullable = true)
 |-- issue: integer (nullable = true)
 |-- publisher: string (nullable = true)
 |-- issn: string (nullable = true)
 |-- publication_id: long (nullable = false)
```

Figure 12.4: Journal table - df_journals.

```
root
 |-- venue: string (nullable = true)
 |-- isbn: string (nullable = true)
 |-- publisher: string (nullable = true)
 |-- publication_id: long (nullable = false)
```

Figure 12.5: Book table - df_books.

```
root
 |-- venue: string (nullable = true)
 |-- location: string (nullable = true)
 |-- publication_id: long (nullable = false)
```

Figure 12.6: Conference table - df_conferences.

13 | Data upload

The library `pyspark` provides many functions to upload data stored in different formats. In our case, the dataset was stored in the JSON format, specifically, it is the one used in MongoDB except for some minor changes, so we used the API for this format. To upload the data we used the object `DataFrameReader`, which is accessible as an attribute of our `SparkSession` instance. More specifically using the `options` method we specified some instructions on how to read the data, and with the `schema` method we supplied a specific schema for the data to be read in order to keep only the fields specified in it. We created a DataFrame for each entity of our ER diagram of chapter 2 and we also translated the many-to-may relationship `Writes` into a table to keep track of the `affiliation` field. For each table, we did some additional filtering on its attributes in order to clean the data and we also did some minor changes that we will now explain. The dataset we inherited from the MongoDB part had a simple structure: a single document, the paper, where all other entities (authors, publication, affiliation) were stored as sub-documents, a collection of sub-documents or simple fields that are present only when needed. Therefore, we have to extrapolate the information regarding the authors and the publications from the papers. In particular, a more complex process was applied to build the dataframes containing information concerning books, conferences, and journals. The following sections briefly explain the processing done within the upload step for each entity contained in the spark database and the reasoning behind it.

13.1. Author table

This table was created by extracting from each Paper document the array containing the authors' documents, with their fields. To do this, we first had to load the data using the load process described in the section before. Secondly, we unwrapped each array using the `explode` function to have a table where each row represents the author document. Finally, we did some filtering on null values and some minor operations, such as renaming, deleting duplicates, or dropping columns.

13.2. Paper, Book, Conference, Journal tables

To load the data relevant to the entities Book, Conference, and Journal we had to address two issues: how to extract and integrate the different types of publications and how to link each paper to the publication medium in which it was published. The first issue is solved by uploading in the DataFrame of each publication type only the fields regarding that specific entity, namely the attributes present in the ER model of section 2, and filtering each DataFrame removing the duplicate rows. We also imposed that the fields have to be not null. To overcome the second issue we opted to introduce a "foreign key" column in the paper DataFrame which stores the unique identifier of the publication medium where it was published. To make the linking with the "foreign key" approach work, we added a unique identifier to each single publication instance cleaned through the previous step. This identifier is generated using a hashing function, specifically the function `xxhash64` present in the pyspark library, which uses the 64-bit variant of the xxHash algorithm. We fed this function with the columns we identified as the primary key, to generate the unique identifier.

The other possible solution we thought of was the creation of an array inside each publication containing all the papers present inside it. In the end, we adopted the first solution because our focus is on the paper, which is the main entity of the database and we assume that the vast majority of the queries will probably have the paper as a starting point, and then retrieve all the information correlated to the paper. In this way, opting for the first solution and considering the assumption mentioned here, the queries done on the papers should be more efficient.

The process we used to upload Books, Journals and Conferences is the following:

1. Upload the paper documents from the JSON input file in a DataFrame with the structure we defined in chapter 12, except for the attribute `publication_id` that will be added later;
2. For each type of publication:
   (a) Upload the data from the JSON file keeping only the fields relative to a single publication type and the field `_id`, which is the identifier of a paper;
   (b) Filter out the rows of the dataframe which have the publication's identifying fields set to null, namely `('venue', 'volume', 'issue', 'issn')` for the journals, `('isbn', 'venue')` for the books and `('venue')` for the conferences;
   (c) Group each publication by the uniquely identifying fields and collect the the

other fields, especially `_id`, in lists;

(d) Select the first element of the list of non-identifying attributes, namely `'publisher'` for journals and books, and `'location'` for the conferences. This choice is motivated by the fact that these fields were set using random strings extracted from some datasets we found on the web. In real scenarios, where we assume to have meaningful data, the best thing to do would be to implement a strategy that keeps the last inserted value for the field or the one that appears the most;

(e) Create the publication identifier (field `publication_id`) using the previous explained hash function;

(f) Create a new DataFrame containing papers, constructed by joining the previous imported Paper DataFrame (Point 1) with the DataFrame of the publication on the field `_id`, which is the paper identifier. In this way we attach to the newly generated Paper DataFrame the `publication_id` field, linking the paper to it's publication. Notice that in order to perform the join we first have to *explode* the field `_id` of the publication DataFrame;

3. At this step of the process, in addition to the dataframes for the journals, for the books and for the conferences we also have other the three newly created paper dataframes obtained by the joining operations. These lastly mentioned dataframes have the same structure of the originial one with the addition of the fields that identify the paper publication. By merging this the three dataframes we obtain the final dataframe of the papers.

In addition to this major operation some minor ones were performed, such as renaming or dropping columns, but these are straightforward to understand, so we won't discuss them any further in this report.

13.3. Affiliation table

This table was created by uploading, from the JSON file, only the identifiers of the papers and their array of authors keeping the id of the author and the organization to which they were affiliated. After filtering the null values, the authors' array is expanded, using the `explode()` function. As before, some other minor operations were also done, such as renaming or dropping of columns.

14 | Commands and queries

14.1. Commands

1. **Add a new paper to the DataFrame**
   With this command, we want to show how to add a new row to our Paper DataFrame. In our example, we suppose that the paper has been saved in a JSON file and has a structure consistent with the schema defined in chapter 12 for the input dataset. This way, we can reuse the loading procedure used before for the entire dataset loading. We first create a new DataFrame for the single paper. Once the paper is loaded, we can proceed to extract other information relevant to the paper, like the authors, and the publication medium but we don't show this part here to have a more concise description (the actual implementation is present on the notebook). After this operation, we proceed to insert the foreign_key associated with the publication medium in the DataFrame containing our paper to insert. This operation is done using the `select` method so we can insert the column as the first one to stay uniform with the structure because in spark even the order of the columns is important. To create the column from a literal value we use the function `lit`. Now the new paper is ready to be added, we have only to first check if the paper is already present in the DataFrame, and we do this by filtering by `paper_id` and verifying that the collection is empty. Then to add the paper we use the method `union` which is applied to DataFrame and returns a new DataFrame containing the union of rows of both DataFrames.

```
new_paper_file = 'single_paper.json'

new_paper = spark.read.options(**OPTIONS)
                      .json(new_paper_file, schemaPaper)\
                      .withColumnRenamed("_id", "paper_id")

# Other code for loading publication, authors, etc
# ...

# Inserting foreign_key at position 0
```

```
new_paper = new_paper.select(lit(foreign_key).alias('publication_id'), '*')

paper_id = new_paper.head(1)[0].asDict()['paper_id']

if df_papers.filter(col('paper_id') == paper_id).collect() == []:
    df_papers = df_papers.union(new_paper)
```

2. **Update one single row of a dataframe or multiple rows**
   The word update has to be intended as the creation of a new dataframe instance which is generated from the previous data changed in the way we want. Indeed, the Dataframe data structure, available in Spark, is immutable so it cannot be modified. Since the modifications are not allowed, to update a single entry the process is a little longer than the one used on other databases.

   The update process can be useful for uploading papers when new data enter the database or when parts of the data were wrong and need to be corrected. There may be many handmade ways to update a dataframe since there are no standard methods. All of them are similar in their underlying idea.

   The command modifies the DOI and URL of a paper with an identifier equal to `'53e997e4b7602d9701fdb48a'`. For doing this operation we first filter the dataframe keeping only the row to be modified. We add a column for each value we want to insert under the name `new_fieldName`. Then we drop the old columns containing the previous values and we rename the new columns with the name of the old ones. Finally, we make the union between the entire dataframe, without the row we want to modify, and the new entry. The `union` function allows to merge two DataFrame with the same schema; if the two structures are different, then the function gives an error.

   The function `lit` is used for creating a new column containing a literal, namely a constant value. The function `array` is used for creating a new array column.

```
from pyspark.sql.functions import lit, array

updated_df_papers = df_papers\
    .filter(col('paper_id') == '53e997e4b7602d9701fdb48a')
updated_df_papers = updated_df_papers\
    .withColumn('new_doi', lit('10.1007/11944577_37'))\
    .withColumn('new_url', array([lit('https://link.springer.com/chapter/10.1007/11944577_37')]))
updated_df_papers = updated_df_papers\
    .drop(col('doi')).drop(col('url'))\
```

```
    .withColumnRenamed('new_doi', 'doi')\
    .withColumnRenamed('new_url', 'url')

updated_df_papers = df_papers.filter(col('paper_id') != '53
    e997e4b7602d9701fdb48a').union(updated_df_papers)
updated_df_papers.filter(col('paper_id') == '53
    e997e4b7602d9701fdb48a').select('paper_id', 'title', 'doi', 'url
    ').show(truncate = False)
```

An analogous method can be applied when we want to update multiple rows following some rule. In this case, the filter will keep many rows and not just one.

```
+------------------------+-----------------------------------------------------+-----------------+-----------------------------------------------------+
|              paper_id|                                                title|              doi|                                                  url|
+------------------------+-----------------------------------------------------+-----------------+-----------------------------------------------------+
|53e997e4b7602d9701fdb48a|CitizenTalk: application of chatbot infotainment to ...|10.1007/11944577_37|[https://link.springer.com/chapter/10.1007/11944577_37]|
+------------------------+-----------------------------------------------------+-----------------+-----------------------------------------------------+
```

3. **Remove an entire column**

   In Spark, we can remove an entire column from a data frame due to deleting information that is no more informative. In these cases, we delete them to free memory for new data.

   To realize our objective, we use the function `drop`, which removes entirely values associated with a field of the data frame.

   In our database, we can use the explained function over the column `lang` because we have realized that all the papers have the same value, `en`. Then, we print the schema of the data frame to see if the function ran properly.

```
df_papers_without_lang = df_papers \
    .drop('lang')

df_papers_without_lang.printSchema()
df_papers_without_lang.show(1)
```

```
root
 |-- publication_id: long (nullable = false)
 |-- paper_id: string (nullable = true)
 |-- title: string (nullable = true)
 |-- keywords: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- fos: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- page_start: integer (nullable = true)
 |-- page_end: integer (nullable = true)
 |-- doi: string (nullable = true)
 |-- url: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- abstract: string (nullable = true)
 |-- publication_type: string (nullable = true)
 |-- date: timestamp (nullable = true)
```

4. **Delete all the papers published before a certain year**
   In order to delete from the DataFrame `df_papers` the rows that represent papers published before a certain year is necessary to create a new DataFrame extracting from the original one only the papers that satisfy the desired condition. We decided to remove all the papers that were published before the `1950` because we consider them obsolete. To check the year in which a paper was published is necessary to use the function `year` in order to extract the information from the `date`, that is of type `TimeStamp`. Based on the condition `year('date') > '1950'` we drop with the `where` clause all the rows that do not satisfy it, saving all the others in the new DataFrame. Finally, with the following command, we show the result of these actions and we present the new `df_papers` ordering the visualization by date, so it is immediately visible that there are no papers realized before `1951`.

```
from pyspark.sql.functions import year

df_papers = df_papers\
    .filter(year('date') > '1950')

df_papers.select('title', 'publication_type', 'date')\
    .orderBy('date')\
    .show(truncate = False)
```

```
+-----------------------------------------------------------------------------------------------------------+----------------+-------------------+
|title                                                                                                      |publication_type|date               |
+-----------------------------------------------------------------------------------------------------------+----------------+-------------------+
|Generalized one-unambiguity                                                                                |Conference      |1951-01-03 03:43:07|
|Predicting PDZ domain mediated protein interactions from structure.                                        |Journal         |1951-01-03 03:43:07|
|A New EDI-based Deinterlacing Algorithm                                                                    |Journal         |1951-01-03 03:43:07|
|Search-based Execution-Time Verification in Object-Oriented and Component-Based Real-Time System Development|Conference      |1951-01-03 03:43:07|
|Multi-structural signal recovery for biomedical compressive sensing.                                       |Journal         |1951-01-03 03:43:07|
|Corpus-based linguistic indicators for aspectual classification                                            |Book            |1951-01-03 03:43:07|
|A comparative study of ad hoc techniques and evolutionary methods for multi-armed bandit problems         |Journal         |1951-01-03 03:43:07|
|Fast Solution of the Radial Basis Function Interpolation Equations: Domain Decomposition Methods          |Journal         |1951-01-03 03:43:07|
|Local Hausdorff Dimension                                                                                  |Journal         |1951-01-03 03:43:07|
|Grammatical Evolution Rules: The Mod and the Bucket Rule                                                   |Book            |1951-01-03 03:43:07|
|Global optimization by canonical dual function                                                             |Journal         |1951-01-03 03:43:07|
|Processing UML Models with Visual Scripts                                                                  |Book            |1951-01-03 03:43:07|
|Balancing buffer utilization in meshes using a "restricted area" concept                                   |Journal         |1951-01-03 03:43:07|
|The Animation of Autonomous Actors Based on Production Rules                                               |Book            |1951-01-03 03:43:07|
|Composing graphical languages                                                                              |Conference      |1951-01-03 03:43:07|
```

5. **Create a new column with the paper's total number of pages**
   With this command, we create a new DataFrame `df_papers_total_pages` starting from `df_papers` but with an additional column called `total_pages` using the function `withColumn()`. This new column contains the total number of pages of the paper computed as the difference between `pages_end` and `page_start` by doing also some consistency checks on these values. Then only some fields are displayed just for better reading.

```
df_papers_total_pages = df_papers \
    .filter((col('page_start') >= 0) & (col('page_end') >= 0) & (
   col('page_start') <= col('page_end'))) \
    .withColumn('total_pages', col('page_end') - col('page_start'))

df_papers_total_pages \
    .select(col('title'), col('page_start'), col('page_end'), col('
   total_pages')) \
    .show(5, truncate=False)
```

```
+-----------------------------------------------------------------+----------+--------+-----------+
|title                                                            |page_start|page_end|total_pages|
+-----------------------------------------------------------------+----------+--------+-----------+
|Problem Decomposition and the Learning of Skills                 |17        |31      |14         |
|X-tract: Structure Extraction from Botanical Textual Descriptions|2         |2       |0          |
|Cognitive agent programming                                      |1385      |1385    |0          |
|Constraint based vectorization                                   |195       |204     |9          |
|Automatic input rectification                                    |80        |90      |10         |
+-----------------------------------------------------------------+----------+--------+-----------+
```

14.2. Queries

1. **Retrieve all papers published on a specific issue and volume of a Journal**
   With this simple query, we want to retrieve all the papers published on a specific issue and on a specific volume of a specific Journal. The specifics of the journal

are saved in variables to be used later. To perform the query we first do a filtering (which is equivalent to a `WHERE`) on the DataFrame containing the journals, to extract only the row concerned with our paper. After that, we perform the default `inner` join with the papers' DataFrame. Notice that, besides specifying the column on which to perform the join, we do an additional check on the `publication_type` to ensure that the join operations are done only for journals. This type of checking is made easy thanks to the possibility offered by Spark to specify more complex join conditions.

```
venue, volume, issue = ('BMC Bioinformatics', '14', '1')

df_papers_q1 = df_journals\
                .filter((col('venue') == venue) &
                        (col('volume') == volume) &
                        (col('issue') == issue))\
                 .join(df_papers,
                       (df_journals['publication_id'] == df_papers['
    publication_id']) &
                          (df_papers['publication_type'] == 'Journal'
    )
                      )

df_papers_q1.select(['paper_id', 'title']).show(truncate=60)
```

```
+------------------------+------------------------------------------------------------+
|                paperID|                                                       title|
+------------------------+------------------------------------------------------------+
|53e997f4b7602d9701ff55f7|BioCause: Annotating and analysing causality in the biome...|
|53e99860b7602d970209d6b9|Predicting PDZ domain mediated protein interactions from ...|
|53e99867b7602d97020a2868|Jimena: efficient computing and system state identificati...|
|53e9989bb7602d97020d129a|Reconciliation-based detection of co-evolving gene families.|
|53e998c7b7602d97021001f0|SeqSIMLA: a sequence and phenotype simulation tool for co...|
|53e998cdb7602d97021066c0|Cheminformatic models based on machine learning for pyruv...|
|53e99938b7602d9702177171|SynTView - an interactive multi-view genome browser for n...|
|53e99953b7602d970219688f|CoMAGC: a corpus with multi-faceted annotations of gene-c...|
|53e9998bb7602d97021d0801|Large-scale extraction of accurate drug-disease treatment...|
+------------------------+------------------------------------------------------------+
```

2. **Papers written in the last 20 years and containing a chosen string within a keyword**

   Find the papers written in the last twenty years whose keywords contains at least one keyword having `artificial` as substring. We require that these papers have the DOI set to a not null value. The results are ordered ascending by the date and only 15 elements are printed.

For filtering a Dataframe variable we can use the function `filter(condition)` of `pyspark` library. This function returns a new DataFrame containing only the rows which make the condition true and eliminate the others. We can use the function `like(string)` as a filtering condition on a column of the DataFrame for keeping only the rows whose considered attribute contains the substring passed as parameter. The `sort` functions can be used for ordering the rows of the DataFrame using the field passed as an argument to the function. The limit function is used for keeping only the first fifteen rows of the dataframe. The function `current_timestamp` returns the timestamp in the current instant. Here the computation of the number of years, needed for passing from is done without considering leap years for simplicity.

```
df = df_papers.withColumn('current time', current_timestamp())
df \
    .filter((((unix_timestamp('current time') - unix_timestamp('
    date')) / 3600 / 24 / 365) < 20) &
             (col('doi').isNotNull())) \
    .select('paper_id',
             'title',
             'date',
             explode('keywords').alias('keyword')) \
    .filter(col('keyword').like('%artificial%')) \
    .distinct() \
    .select('title',
             'date',
             'keyword') \
    .sort(col('date').asc()) \
    .limit(15) \
    .show(truncate=55)
```

```
+-------------------------------------------------+-------------------+--------------------------------+
|                                            title|               date|                         keyword|
+-------------------------------------------------+-------------------+--------------------------------+
|  Neural Model of a Grid-Based Map for Robot Sonar|2003-02-11 07:22:08|            artificial neural net|
|Extreme learning machine for predicting HLA-Pep...|2003-02-26 15:03:43|        artificial neural network|
|Extreme learning machine for predicting HLA-Pep...|2003-02-26 15:03:43|       artificial neural networks|
|Acquiring Empirical Knowledge to Support Intell...|2003-03-05 06:17:08|ontologies artificial intelligence|
|IIBR-a system for managing/refining structural ...|2003-03-05 17:12:35|           artificial intelligence|
|IIBR-a system for managing/refining structural ...|2003-03-05 17:12:35|  learning artificial intelligence|
|Building Complex Systems Using Developmental Pr...|2003-04-28 08:14:14|                  artificial life|
|Nonlinear model structure design and constructi...|2003-05-06 06:48:19|  learning artificial intelligence|
|Graph Kernel-Based Learning for Gene Function P...|2003-08-27 02:04:03|  learning artificial intelligence|
|                    Natural evolution strategies|2003-09-04 09:41:35|  learning artificial intelligence|
|                    Human information processing|2003-09-04 09:41:35|           artificial intelligence|
|Optimization with neural networks: a recipe for...|2003-09-10 21:35:07|        artificial neural network|
|An Incremental Approach for Niching and Buildin...|2003-10-27 12:55:11|  learning artificial intelligence|
|                    Structural Logical Relations|2003-10-28 11:35:16|       artificial neural networks|
|                    Intelligent Process Platform|2003-10-31 11:24:26|ontologies artificial intelligence|
+-------------------------------------------------+-------------------+--------------------------------+
```

3. **Retrieve papers published in conferences that have topics about multiagent systems**

   This query retrieves the papers with `multiagent system` within its keywords, so we are interested only in a subset of our database. In extracting them, we select at first some of the columns of the dataframe associated with papers, and in particular, we call the explode function over the `keywords` column to duplicate the rows and associate to them an element of the exploded column. Then, we extract a subset of all the generated rows by selecting only those that have as keyword `multiagent system`. We extract only the columns `title`, `publication_type`, `paper_id`, `publication_id`, and `date`, and we save the result in a variable called `nested_query`.

   Thus, we perform another query over the created dataframe by first calling a join operation to join it with the one associated with the conferences, matching the variable `publication_id`, and then we select only those papers of interest. We reorder the rows by date in a descending way, and then we extract only the title of the paper and the venue of the conference. Finally, we show the dataframe obtained.

```
nested_query = df_papers \
    .select('title',
            'publication_type',
            'paper_id',
            'publication_id',
            'date',
            explode('keywords').alias('keyword')) \
    .filter(col('keyword').isin('multiagent system')) \
    .drop('keyword')
df_conferences \
    .join(nested_query, 'publication_id') \
    .filter(col('publication_type') == 'Conference') \
    .orderBy('date', ascending=False) \
    .select(col('title').alias('paper title'),
            col('venue').alias('conference venue')) \
    .show(5, truncate=50)
```

```
+--------------------------------------------------+--------------------------------------------------+
|                                       paper title|                                    conference venue|
+--------------------------------------------------+--------------------------------------------------+
|                           Normative system games|Autonomous Agents & Multiagent Systems/Agent Th...|
|                           Social Default Theories|     Logic Programming and Non-monotonic Reasoning|
|Verification & validation of an agent-based for...|               Spring Simulation Multiconference|
|A Self-Organizing System for Online Maintenance...|                          Intelligent Environments|
|Profile recommendation in communities of practi...|                                              SBIA|
+--------------------------------------------------+--------------------------------------------------+
```

4. **Retrieve the most prolific organizations regarding the conferences**
   This query tries to retrieve the organizations that published more than `10` conferences, so the most prolific ones in these kinds of publications. We first execute a join between the DataFrame that contains the papers and the one that contains the affiliations, because to obtain the result we need information from them both. We need to filter on the `publication_type` that is present in `df_papers` in order to consider only the conferences and, then, we group by `organization`, present in the `df_aff`, and with `collect_set()` we collect for each organization the list of the related papers, avoiding duplicates. Finally, we filter the result, retrieving only those organizations whose associated list has more than `10 paper_id`. In showing the most prolific organizations we truncate the columns' text using `truncate = 50`, to have a compact and clear visualization.

```
from pyspark.sql.functions import collect_set, size

df_aff \
    .join(df_papers, df_papers.paper_id == df_aff.paper_id, 'inner'
    ) \
    .drop(df_papers.paper_id) \
    .select('paper_id',
            'organization',
            'publication_type') \
    .filter((col('organization').isNotNull()) &
            (col('organization') != "") &
            (col('publication_type') == "Conference")) \
    .groupBy('organization') \
    .agg(collect_set('paper_id').alias('papers')) \
    .filter(size(col('papers')) > 10) \
    .show(truncate=50)
```

```
+--------------------------------------------------+--------------------------------------------------+
|                                      organization|                                            papers|
+--------------------------------------------------+--------------------------------------------------+
|       Carnegie Mellon University, Pittsburgh, PA|[53e998dbb7602d9702118f29, 53e9987db7602d97020b...|
|   Carnegie Mellon University, Pittsburgh, PA, USA|[53e998c7b7602d97020fdf46, 53e99905b7602d970214...|
| Georgia Institute of Technology, Atlanta, GA, USA|[53e99827b7602d9702048d5f, 53e998f6b7602d970213...|
|          Microsoft Research Asia, Beijing, China|[53e99976b7602d97021b91db, 53e9980eb7602d970202...|
|             Microsoft Research, Redmond, WA, USA|[53e99885b7602d97020bfe89, 53e99930b7602d970216...|
|National University of Singapore, Singapore, Si...|[53e99813b7602d970202da43, 53e99876b7602d97020b...|
|              Tsinghua University, Beijing, China|[53e998c7b7602d97020fdf46, 53e99998b7602d97021d...|
|       University of Washington, Seattle, WA, USA|[53e998aab7602d97020e600e, 53e99967b7602d97021a...|
+--------------------------------------------------+--------------------------------------------------+
```

5. **Retrieve some statistics about papers**
   In this query, we retrieve some statistics about papers published from the year 2015

on. We group the articles by publication year using the `year()` function to extract the year from the date. Then some aggregate functions are performed to obtain insights like the total number of papers published using `count()`, the total number of pages written, the minimum and the maximum number of pages in an article using `sum()`, `min()`, `max()` and lastly the mean of pages written per article and the variance using `avg()` and `variance()`. To round the fractional numbers we use `format_number()`. Years are shown in decreasing order. In this query we use the DataFrame obtained by running the command 5.

```
from pyspark.sql.functions import sum, min, max, avg, format_number
    , variance

df_papers_total_pages \
    .filter(year(col('date')) >= 2015) \
    .groupBy(year(col('date')).alias('year')) \
    .agg(count('paper_id').alias('total_papers'),
         sum('total_pages').alias('total_pages'),
         min('total_pages').alias('min_pages'),
         max('total_pages').alias('max_pages'),
         format_number(avg('total_pages'), 2).alias('avg_pages'),
         format_number(variance('total_pages'), 2).alias('var_pages
    ')) \
    .sort(col('year').desc()) \
    .show()
```

```
+----+-----------+-----------+---------+---------+---------+---------+
|year|total_papers|total_pages|min_pages|max_pages|avg_pages|var_pages|
+----+-----------+-----------+---------+---------+---------+---------+
|2022|        289|       2918|        0|       41|    10.10|    47.79|
|2021|        455|       5881|        0|     1585|    12.93| 5,509.97|
|2020|        556|       5345|        0|      145|     9.61|    94.19|
|2019|        459|       4334|        0|      135|     9.44|    80.38|
|2018|        504|       4756|        0|      135|     9.44|    74.82|
|2017|        423|       3818|        0|       49|     9.03|    50.92|
|2016|        728|       6687|        0|       55|     9.19|    45.74|
|2015|        530|       5364|        0|      158|    10.12|    92.43|
+----+-----------+-----------+---------+---------+---------+---------+
```

6. **Papers that are referenced the most and have at least 30 references**
   With this query, we want to know which papers present in our DB are referenced the most absolutely. We achieve this goal by first unwrapping the references for each paper using the `explode` function. Then, we proceed grouping by the reference

we previously *exploded* and use the `count` aggregation function to actually count the papers who have that particular reference, saving this value in the column `references_count`. Lastly, we do a filtering on `references_count` to extract only the papers with at least 30 references, and a join to get some significant data about the papers we extracted with the previous operations.

```
df_papers \
    .select('paper_id',
            'title',
            explode(col('references')).alias('reference')) \
    .groupBy('reference') \
    .agg(count('paper_id').alias('references_count')) \
    .filter(col('references_count') > 30) \
    .join(df_papers, col('reference') == df_papers.paper_id) \
    .sort(col('references_count').desc()) \
    .select(['title', 'references_count']) \
    .show(truncate=50)
```

```
+--------------------------------------------------------------+----------------+
|title                                                         |references_count|
+--------------------------------------------------------------+----------------+
|Distinctive Image Features from Scale-Invariant Keypoints     |195             |
|A simple transmit diversity technique for wireless communications|62              |
|Light field rendering                                         |56              |
|Network information flow                                      |55              |
|Mining Sequential Patterns                                    |44              |
|Symbolic Model Checking                                       |44              |
|Geodesic Active Contour.                                      |40              |
|Differential Power Analysis                                   |38              |
+--------------------------------------------------------------+----------------+
```

7. **Couples of the field of study and keywords which appear more frequently within the papers**

   The query returns the association between fields of study and keywords which are more present inside the database and how many times they appear together. Initial filtering is done in order to eliminate the papers which don't contain any keyword or field of study. The two clauses for doing the filtering can be modified by increasing the required number of keywords or the required number of fields of study. This kind of change can be interesting in order to understand which fields of study and keywords become more relevant when the subset of the considered paper is different. The results are ordered by the number of occurrences of the couples `field of study - keyword` by decreasing order. We filter out the couples of fields of study and

keywords with less than 100 occurrences in order to get rid of some misleading information due to not coupled elements. The threshold can be adjusted.

```
from pyspark.sql.functions import year, col, size

df = df_papers \
    .filter(
    (col('doi').isNotNull()) &
    (year(col('date')) >= 2000) &
    (size(col('fos')) > 0) &
    (size(col('keywords')) > 0)) \
    .select('fos', explode('keywords').alias('keyword')) \
    .select('keyword', explode('fos').alias('fos')) \
    .groupby('fos', 'keyword') \
    .count() \
    .withColumnRenamed('count', 'couple count') \
    .filter(col('couple count') > 100) \
    .sort(col('couple count').desc()) \
    .limit(15) \
    .show(truncate=False)
```

```
+------------------------------+------------------------------+------------+
|fos                           |keyword                       |couple count|
+------------------------------+------------------------------+------------+
|computer science              |internet                      |247         |
|computer science              |data mining                   |242         |
|computer science              |computer science              |226         |
|computer science              |real time                     |189         |
|computer science              |protocols                     |178         |
|artificial intelligence       |feature extraction            |159         |
|feature extraction            |feature extraction            |159         |
|computer science              |quality of service            |156         |
|computer science              |satisfiability                |155         |
|computer science              |feature extraction            |154         |
|computer science              |real time systems             |146         |
|the internet                  |internet                      |137         |
|computer science              |computational modeling        |136         |
|computer science              |learning artificial intelligence|132         |
|computational complexity theory|computational complexity      |131         |
+------------------------------+------------------------------+------------+
```

8. **Retrieve the organizations associated with an author name for each field of study**

   For each field of study associated with papers written by any author named `Hao Wang`,

this query retrieves all the organizations that have contributed to that field of study through the searched authors.

For the complexity of the problem, we split the query into two nested queries. At first, we query the data frame associated with authors by filtering it to extract the author `Hao Wang`. Then we select only the `author_id` column, and we collect the values in a list called `sub_nested_query` through the functions `flatMap` and `collect`.

Next, we are interested in filtering also the data frame associated with affiliation by extracting rows that have `author_id` present inside the `sub_nested_query` list. We also filter the same data frame by extracting only the rows with a not-null value of the organization column, and then we assign the result to the `nested_query` variable.

Finally, we join over the column `paper_id` the last result with the data frame associated with papers. From the joined data frame, we explode the column `fos`, and we associate to each of those elements called `field_of_study` the value of the columns `paper_id` and `organization`. We group the rows by the `field_of_study` values, and then we aggregate through the function `agg` the organizations that are collected in a set by the last operation. We are performing these operations because we are interested in all the organizations in which the author has worked in his carrier. In the end, we rearrange the rows by ordering the column `field_of_study` alphabetically.

```
author_name = 'Hao Wang'

sub_nested_query = df_aut \
    .filter(col('name') == author_name) \
    .select('author_id') \
    .rdd.flatMap(lambda x: x) \
    .collect()

nested_query = df_aff \
    .filter(col('author_id').isin(sub_nested_query)) \
    .filter(col('organization') != 'null')

df_papers \
    .join(nested_query, 'paper_id') \
    .select('paper_id',
            'organization',
            explode('fos').alias('field_of_study')) \
    .groupBy('field_of_study') \
```

```
    .agg(collect_set('organization').alias('organization')) \
    .orderBy('field_of_study') \
    .show(14, truncate=80)
```

```
+--------------------------+--------------------------------------------------------------------------------+
|            field_of_study|                                                                    organization|
+--------------------------+--------------------------------------------------------------------------------+
|    active queue management|[Department of Information Engineering, The Chinese University of Hong Kong, ...|
|                 algorithm|                                      [Fuyang Teachers College, Fuyang, Anhui]|
|          algorithm design|[Department of Information Engineering, The Chinese University of Hong Kong, ...|
|    artificial intelligence|                    [Visual Commun. Lab., Nokia Res. Center, Beijing, China|c|]|
|bandwidth signal processing|                                                [University of Utah, Salt Lake City]|
|               binary image|                    [Visual Commun. Lab., Nokia Res. Center, Beijing, China|c|]|
|          business process|       [Centre for Logic and Information, St. Francis Xavier University, Canada]|
|          cluster analysis|                                      [Fuyang Teachers College, Fuyang, Anhui]|
|           colors of noise|                    [Visual Commun. Lab., Nokia Res. Center, Beijing, China|c|]|
|     computational geometry|                                                [University of Utah, Salt Lake City]|
|          computer network|[Department of Information Engineering, The Chinese University of Hong Kong, ...|
|          computer science|[Department of Information Engineering, The Chinese University of Hong Kong, ...|
|           computer vision|                    [Visual Commun. Lab., Nokia Res. Center, Beijing, China|c|]|
|       connected component|                    [Visual Commun. Lab., Nokia Res. Center, Beijing, China|c|]|
+--------------------------+--------------------------------------------------------------------------------+
```

9. **Retrieve the most prolific publishers**

   In this query, we join the `df_books` and the `df_journals` on the `publisher` column. Due to this join and to the fact that the two DataFrames have both the column `venue` we have to rename this column in at least one of the tables, to have distinguishable data. We decided to rename both columns to have more clear data in the new DataFrame, created by the join operation. In this case, combining the related tuples from books and journals, we are interested only in journals' publications that arrived at least at volume `10`, while we consider all the books without filtering them. After the execution of the join, we drop the duplicates and group them by `publisher`. Then, we create two sets related to the publisher, one for the books and one for the journals, and we concatenate them to obtain for each publisher a single list with all its publications. We define sets and not lists because they don't allow to have duplicates of values. Now, we can retrieve the most prolific publishers, just by filtering on the size of the list of publications associated with the publishers. In our dataset, there are not a lot of publishers that have more than `500` publications, so these are the most prolific ones, but the filtering can be modified at any time, depending on the data, to obtain meaningful results.

```
from pyspark.sql.functions import collect_set, concat, size

df_journals \
    .withColumnRenamed('venue', 'venueJournals') \
    .filter((col('volume')) > 10) \
```

```
    .join(df_books, df_books.publisher == df_journals.publisher, '
   inner') \
    .drop(df_journals.publisher) \
    .withColumnRenamed('venue', 'venueBooks') \
    .select('venueBooks',
            'venueJournals',
            'publisher') \
    .dropDuplicates(['venueBooks',
                     'venueJournals',
                     'publisher']) \
    .groupBy('publisher') \
    .agg(collect_set('venueBooks').alias('books'),
         collect_set('venueJournals').alias('journals')) \
    .withColumn('total_publications_per_publisher', concat('books',
    'journals')) \
    .filter(size('total_publications_per_publisher') > '500') \
    .select('publisher',
            'total_publications_per_publisher') \
    .show(truncate=50)
```

```
+----------------------------------------+--------------------------------------------------+
|                               publisher|                 total_publications_per_publisher|
+----------------------------------------+--------------------------------------------------+
|                 Taylor and Francis Ltd.|[Special Interest Group on Software Engineering...|
|                                Elsevier|[SAS, VAST, Focus on Scientific Visualization, ...|
|Association for Computing Machinery (ACM)|[Special Interest Group on Software Engineering...|
|                          Springer Verlag|[IEEE METRICS, ISQED, Special Interest Group on...|
+----------------------------------------+--------------------------------------------------+
```

10. **Prolific authors with different organizations**

    The query retrieves authors who worked for at least three different organizations and have published at least three papers with at least five fields of study and five references each. To obtain this it is necessary to join the paper table with the affiliation DataFrame on `paper_id` and then join it with the author DataFrame on `author_id`. We group on `author_id` to get all the aggregate information for each author. To count the different organizations an author has worked for we use `approx_count_distinct()`, which gives a faster response counting approximately the different values, if interested in the exact value it is possible to use `countDistinct()`. In the `HAVING` part, for consistency, we check that only one name is associated with the grouped `author_id` and in the `SELECT` part we explode that field to obtain a single string instead of an array. The result is decreasingly ordered by the number of papers and then the number of organizations.

```
from pyspark.sql.functions import approx_count_distinct

df_papers \
    .filter((size(col('fos')) >= 5) &
            (size(col('references')) >= 5)) \
    .join(df_aff, df_papers.paper_id == df_aff.paper_id, 'inner') \
    .drop(df_aff.paper_id) \
    .join(df_aut, df_aff.author_id == df_aut.author_id, 'inner') \
    .drop(df_aff.author_id) \
    .groupBy('author_id') \
    .agg(count('paper_id').alias('papers_count'),
         approx_count_distinct('organization').alias('
   organizations_count'),
         collect_set('name').alias('name')) \
    .filter((size('name') == 1) &
            (col('papers_count') >= 3) &
            (col('organizations_count') >= 3)) \
    .orderBy(col('papers_count').desc(),
             col('organizations_count').desc()) \
    .select(explode('name').alias('name'),
            'papers_count',
            'organizations_count') \
    .show(5)
```

```
+------------------+-----------+------------------+
|              name|papers_count|organizations_count|
+------------------+-----------+------------------+
|   Rachid Guerraoui|          22|                11|
|Thomas A. Henzinger|          21|                14|
|        Dacheng Tao|          20|                13|
|     Moshe Y. Vardi|          20|                 7|
|    Thomas S. Huang|          19|                10|
+------------------+-----------+------------------+
```

Bibliography

[1] **GitHub repository that contains all the scripts mentioned in the document** www.github.com/IrfEazy/smbud-project