CHAPTER 6:

JOIN THE NOSQL MOVEMENT

from Introducing Data Science

by Davy Cielen, Arno D. B. Meysman, and Mohamed Ali

This chapter cover

- Understanding NoSQL databases and why they're used today.
- Identifying the different NoSQL and relational databases.
- Defining the ACID principle and how it relates to the NoSQL BASE principle.
- Learning why the CAP theorem is important for multi-node database setup.
- Applying the data science process to a project with the NoSQL database Elasticsearch.

SECTION 6.1:

INTRODUCTION TO NOSQL

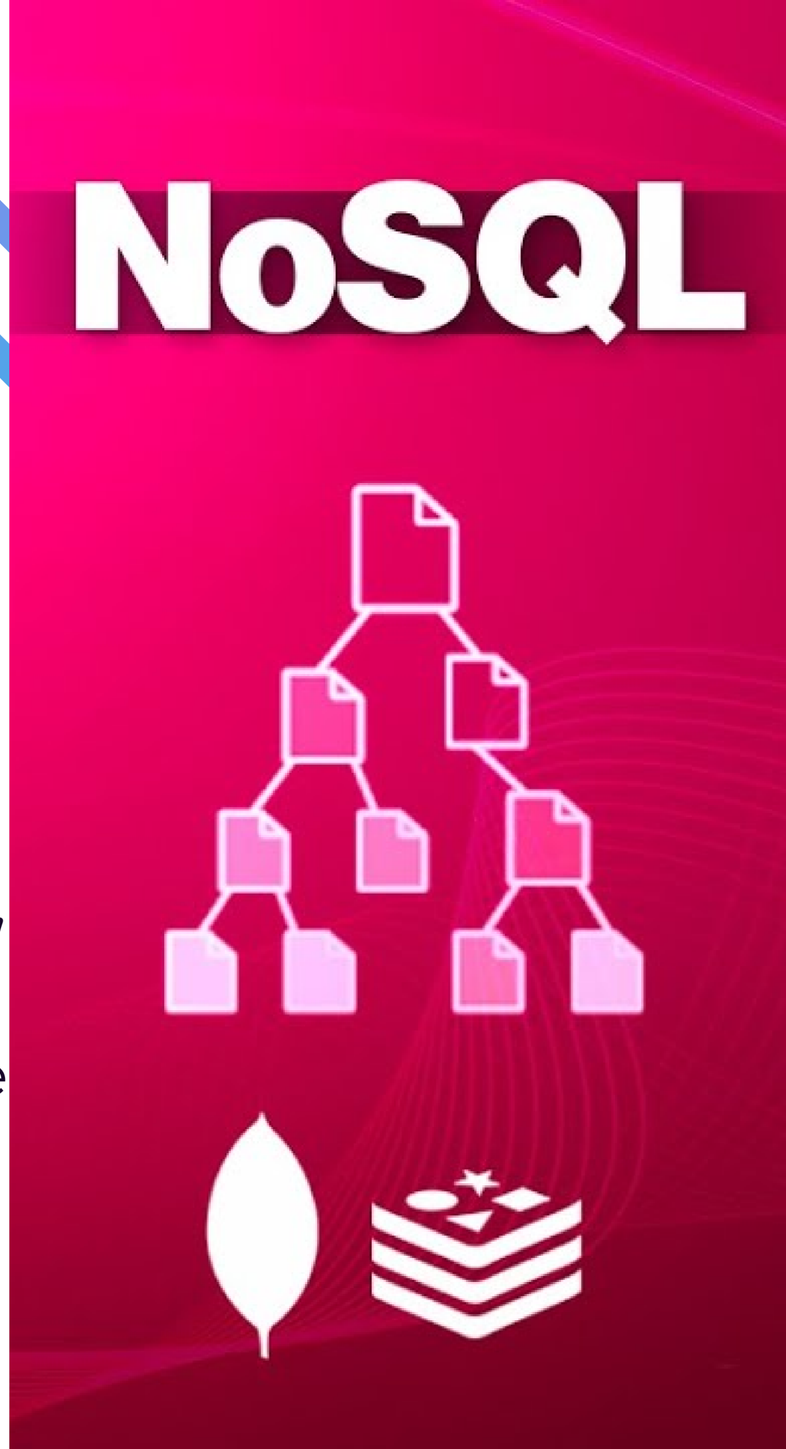
We'll look into NoSQL databases in general and answer these questions: Why do they exist? Why not until recently? What types are there and why should you care?

What is NoSQL?

NoSQL is a type of database management system that is designed to handle large volumes of unstructured or semi-structured data. Unlike traditional relational databases, NoSQL databases do not rely on a fixed schema and are highly scalable and flexible. They are often used in big data applications, web applications, and other scenarios where data needs to be processed quickly and efficiently.

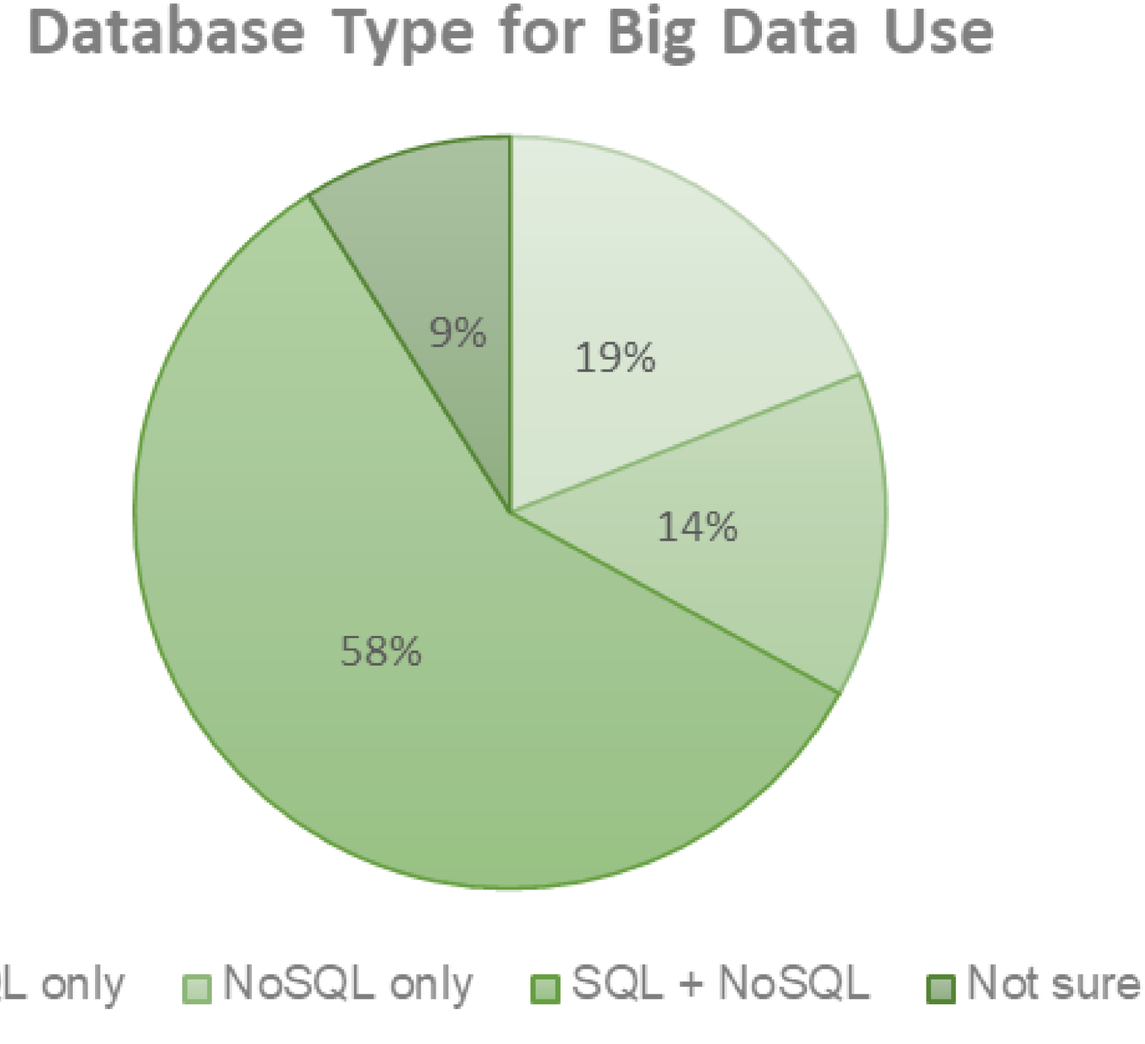
Why do they exist?

NoSQL databases are designed to address the limitations of traditional relational databases, which were designed to handle structured data and run on a single server. As data volumes grew and more complex data structures emerged, it became clear that a new type of database was needed that could handle unstructured or semi-structured data, scale horizontally across multiple servers, and provide high availability and fault tolerance. NoSQL databases were developed to meet these needs and have become increasingly popular in recent years due to their flexibility, scalability, and performance advantages over traditional relational databases.



Why not until recently?

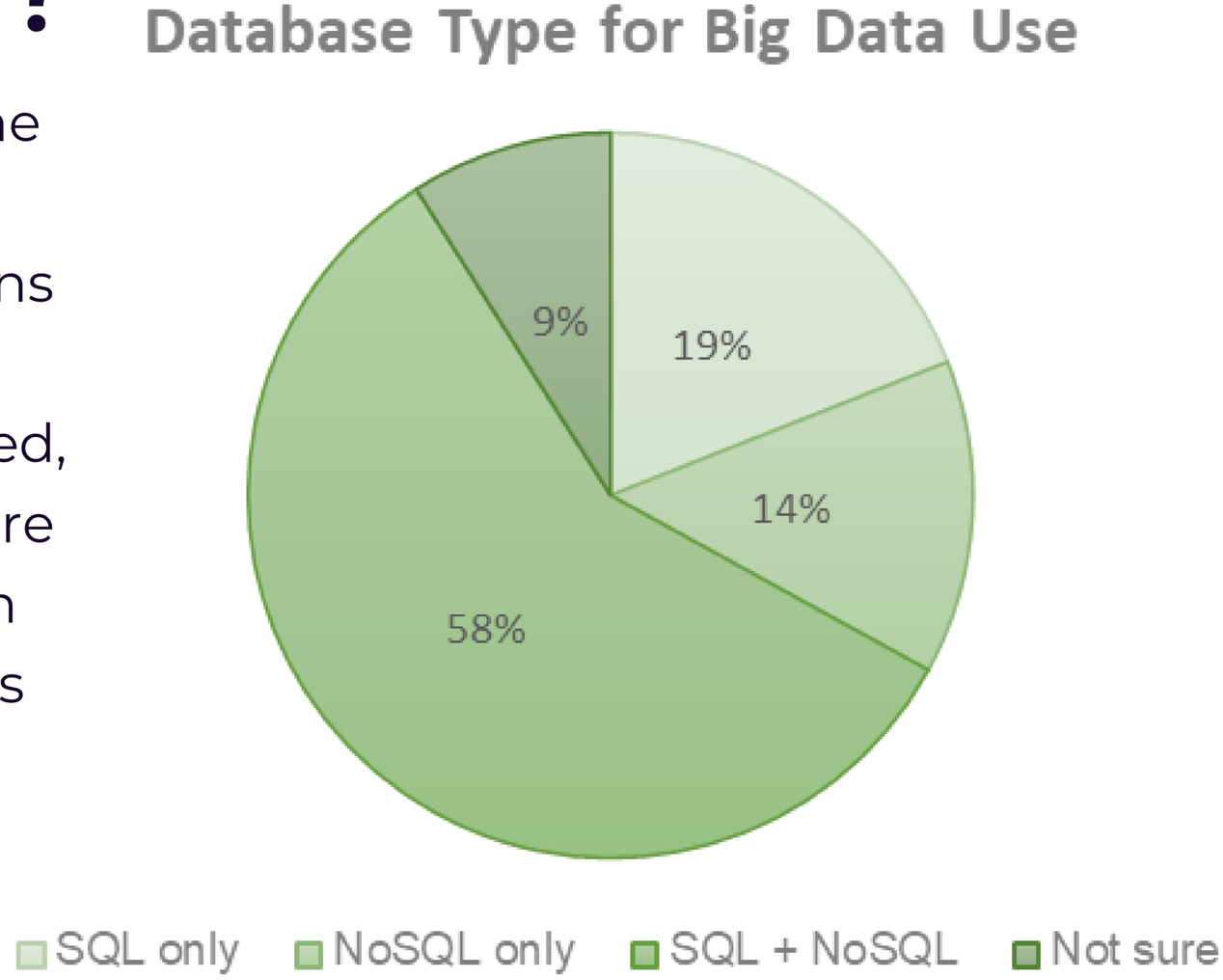
NoSQL databases have been around for several decades, but they were not widely adopted until recently due to several factors. They were originally designed to handle large volumes of unstructured data, which was not a common problem in the past. However, with the rise of big data and the Internet of Things (IoT), more and more companies are generating massive amounts of unstructured data that needs to be processed quickly and efficiently.



*Stats are taken from here

Why not until recently?

Early NoSQL databases lacked some of the features and functionality of relational databases, such as support for transactions and complex queries. However, as data volumes grew and new use cases emerged, NoSQL databases evolved to become more robust and feature-rich. Advancements in cloud computing and distributed systems also played a significant role in the increased adoption of NoSQL databases. These advancements made it easier to deploy and manage NoSQL databases at scale, further fueling their popularity.



*Stats are taken from here

What are differences between NoSQL and SQL?

Data model

Relational databases (RD) use a tabular data model with fixed schemas, while NoSQL databases use flexible data models that can handle unstructured or semi-structured data.

Scalability

RD are typically designed to run on a single server and scale vertically, while NoSQL databases are designed to scale horizontally across multiple servers.

Querying

Relational databases use SQL (Structured Query Language) for querying data, while NoSQL databases use various query languages or APIs depending on the type of database.

What are differences between NoSQL and SQL?

ACID compliance

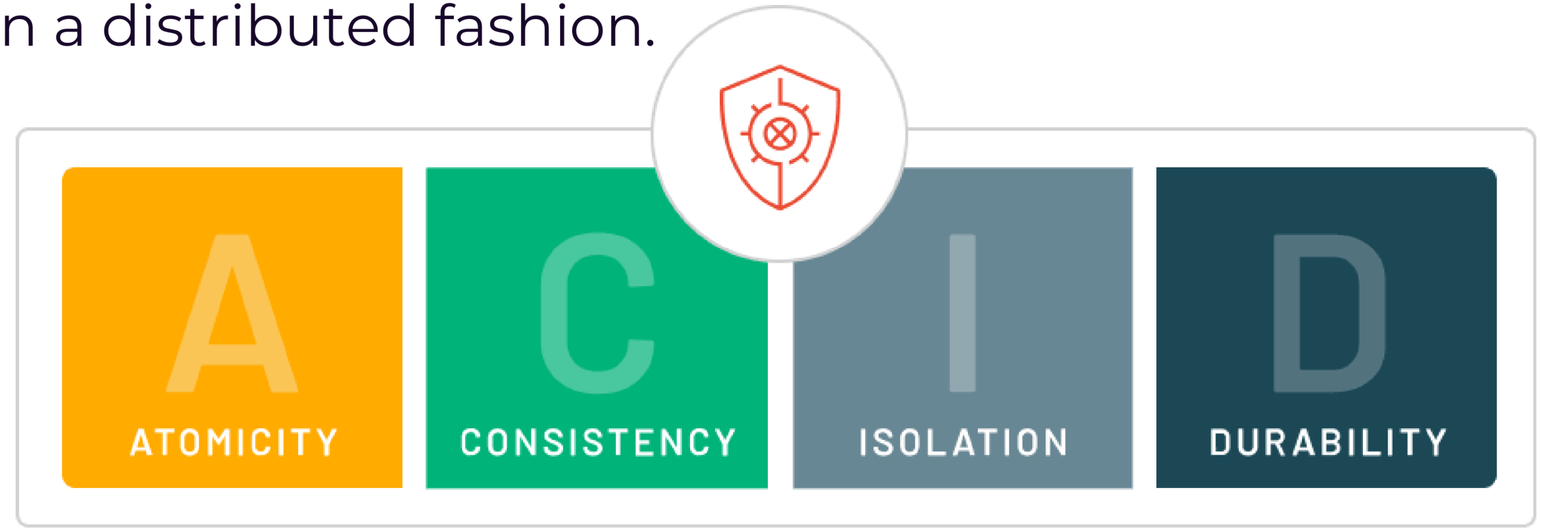
RD are typically ACID-compliant, meaning they guarantee consistency, isolation, durability, and atomicity of transactions. NoSQL databases may sacrifice some of these guarantees in favor of scalability and performance.

Cost

RD can be expensive to license and maintain, especially for large-scale deployments. NoSQL databases are often open source or have lower licensing costs, making them more cost-effective for certain use cases.

How NoSQL databases rewrite ACID into BASE principles

NoSQL databases rewrite ACID (Atomicity, Consistency, Isolation, Durability) into BASE (Basically Available, Soft state, Eventually consistent) principles to work better in a distributed fashion.



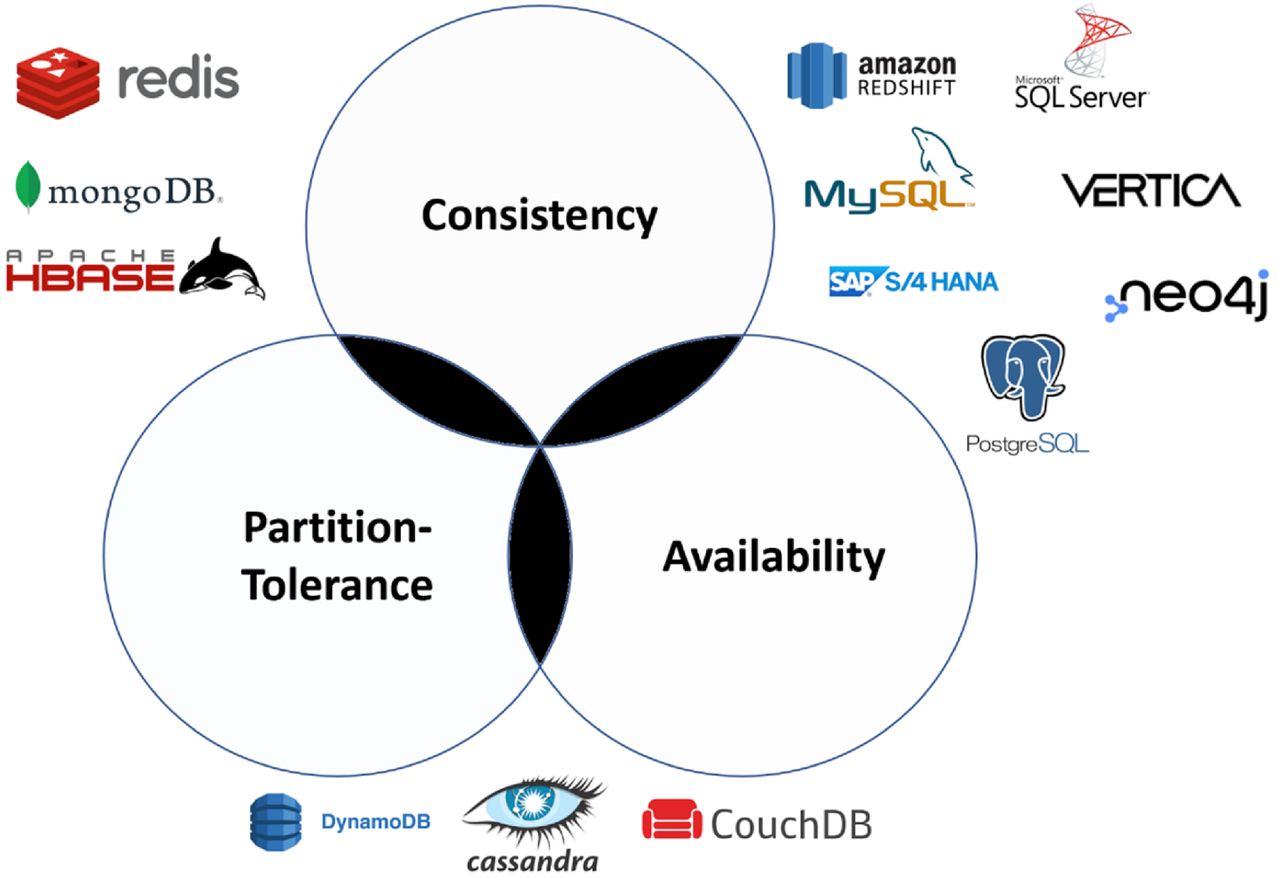
ACID is a set of properties that guarantee the reliability and consistency of transactions in a relational database. However, these properties can be difficult to maintain in a distributed environment where data is spread across multiple nodes.

How NoSQL databases rewrite ACID into BASE principles

BASE principles provide an alternative approach that prioritizes availability and partition tolerance over strict consistency. Basically Available means that the system should always be available for read and write operations even if some nodes fail. Soft state means that the system can tolerate temporary inconsistencies or conflicts between nodes. Eventually consistent means that the system will eventually converge to a consistent state over time.

By adopting BASE principles, NoSQL databases can achieve high scalability and fault tolerance while still providing acceptable levels of consistency for many use cases. However, it's important to note that not all NoSQL databases follow BASE principles - some may still prioritize strict consistency over availability or partition tolerance depending on their specific use case.

How NoSQL databases rewrite ACID into BASE principles



The CAP theorem is important for multi-node database setups because it highlights the trade-offs that must be made when designing a distributed system. The CAP theorem states that in a distributed system, it is impossible to simultaneously achieve all three of the following guarantees:

Why the CAP theorem is important for multi-node database setup?

Consistency

Every read operation receives the most recent write or an error.

Availability

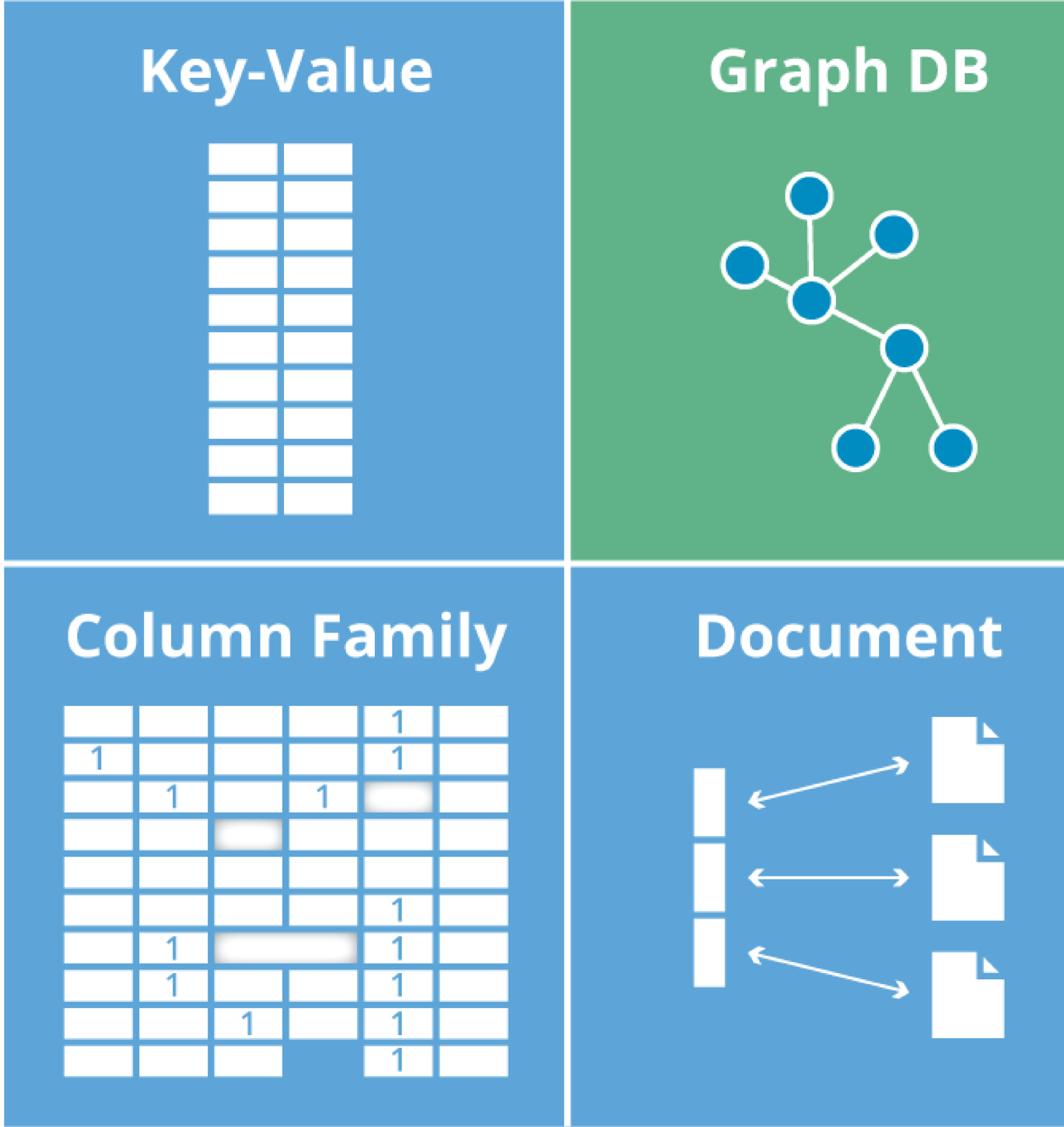
Every non-failing node returns a response for all read and write requests in a reasonable amount of time.

Partition tolerance

The system continues to function even when network partitions occur, meaning that nodes are unable to communicate with each other.

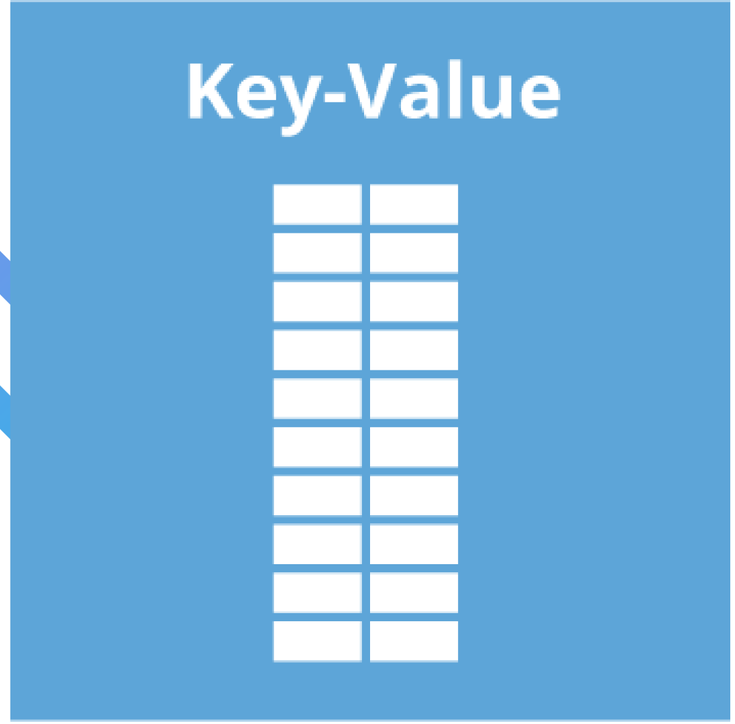
What types should you care?

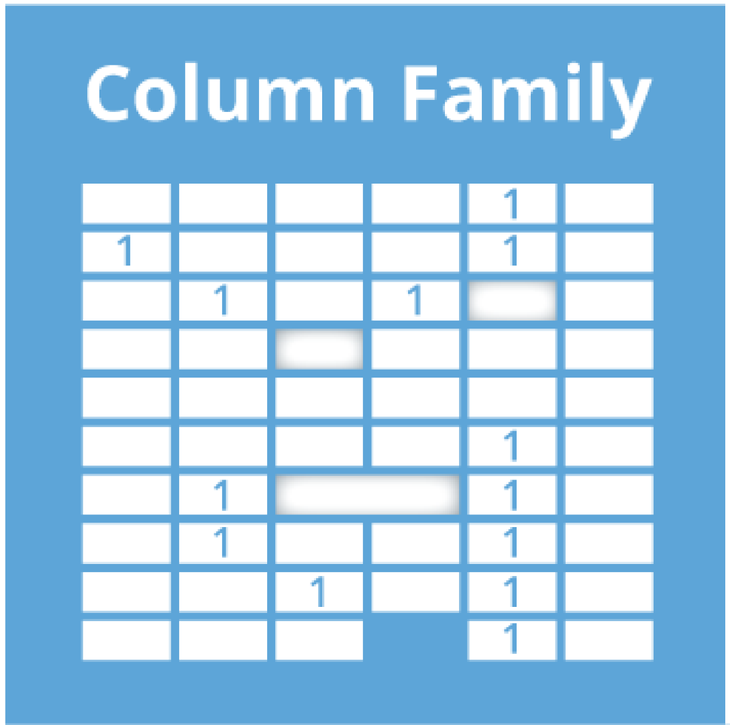
There are several types of NoSQL databases, including document stores, key-value stores, column-family stores, and graph databases. Each type is designed to handle different types of data and use cases.



What types should you care?

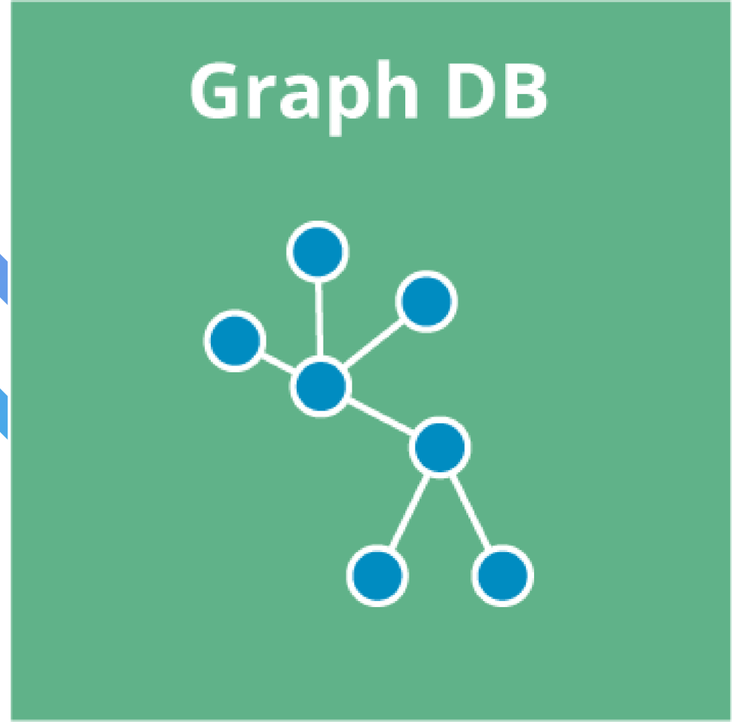
Document stores are ideal for storing and querying semi-structured data such as JSON or XML documents. Key-value stores are optimized for high-speed read and write operations on simple data structures such as strings or integers. Column-family stores are designed to handle large volumes of structured data with high write throughput. Graph databases are used to store and query complex relationships between entities.

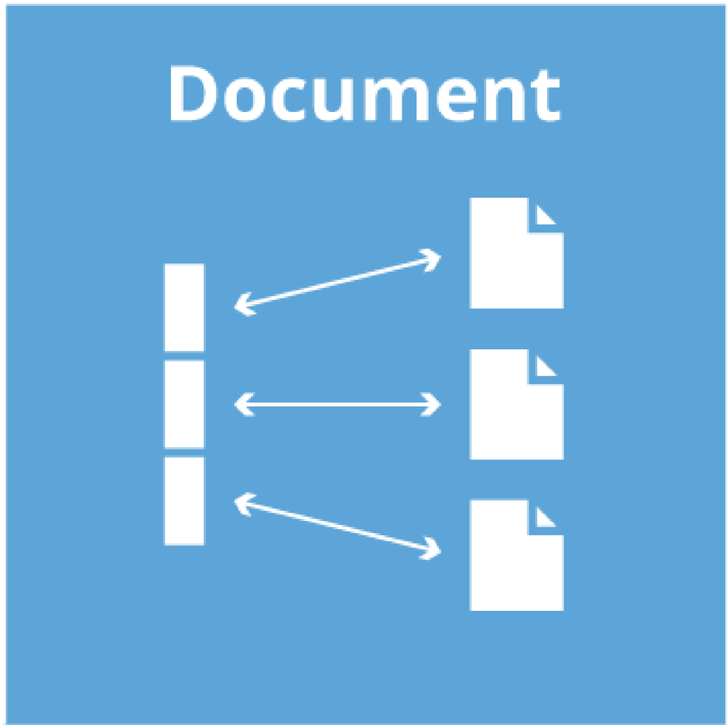




What types should you care?

You should care about the different types of NoSQL databases because each type has its own strengths and weaknesses, and choosing the right type for your application can have a significant impact on performance, scalability, and cost. By understanding the different types of NoSQL databases and their use cases, you can make an informed decision about which one is best suited for your needs.





SECTION 6.2:

NOSQL IN PRACTICE

We'll tackle a real-life problem—disease diagnostics and profiling—using freely available data, Python, and a NoSQL database. Please check out the README file to to see the contents of this section

Case study: What disease is that?

In this section, we explore the common scenario of people searching for medical symptoms on search engines like Google and how a dedicated database for disease diagnostics would be more helpful. We discuss that such databases exist but are not accessible to the public due to well-protected data. The section introduces the case study of disease diagnostics and explains that the primary goal is to explore data by querying for disease symptoms. This case study will help us understand how NoSQL databases can be used to solve real-life problems in the field of healthcare.

The process we use

01

Setting the research goal

The primary goal is to set up a disease search engine that would help general practitioners in diagnosing diseases.

02

Data collection

Data will be collected from Wikipedia, although other sources could be used.

03

Data preparation

Techniques will be applied to clean and transform the data from Wikipedia into a format suitable for analysis.

04

Data exploration

The use case is special in that step 4 of the data science process is also the desired end result - making the data easy to explore.

The process we use

05

Data modeling

Document-term matrices that are used for search are often the starting point for advanced topic modeling. But we won't apply in this.

06

Presentation

The disease diagnostics tool can be turned into a self-service diagnostics tool by allowing a physician to query it via a web application or other user interface. Disease profiling can also be taken to the level of a user interface, such as producing a word cloud that visually summarizes search results.

ITEM REQUEST

PYTHON3

ELASTICSEARC

KIBANA

Elasticsearch is a distributed, open-source search and analytics engine that is built on top of Apache Lucene. It provides a scalable search solution that can be used to index and search large volumes of data in real-time. Elasticsearch is designed to be highly available, fault-tolerant, and easy to use. It supports a wide range of use cases, including full-text search, structured search, analytics, and machine learning. Elasticsearch is commonly used in applications such as e-commerce websites, log analysis tools, and enterprise search solutions.

STEP 1: SETTING THE RESEARCH GOAL

Which is to create a disease search engine that would help general practitioners in diagnosing diseases. The primary objective is to explore data by querying for disease symptoms. The case study will use NoSQL databases to solve real-life problems in the field of healthcare. Data will be collected from Wikipedia and prepared for analysis using techniques such as cleaning and transformation. Elasticsearch and Python will be used to implement the database, which can be turned into a self-service diagnostics tool via a web application or other user interface.

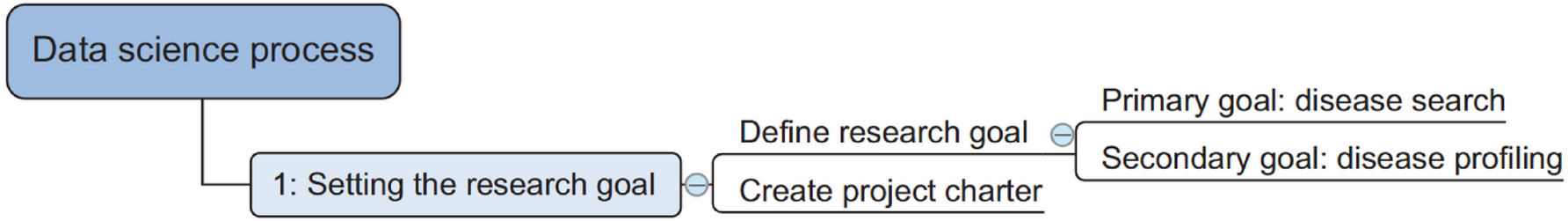


Figure 6.17 Step 1 in the data science process: setting the research goal

STEP 2 & 3: DATA RETRIEVAL & PREPARATION

We have two possible sources: internal data and external data.

— Internal data: You have no disease information lying around. If you currently work for a pharmaceutical company or a hospital, you might be luckier

— External data: All you can use for this case is external data. You have several possibilities, but you'll go with Wikipedia

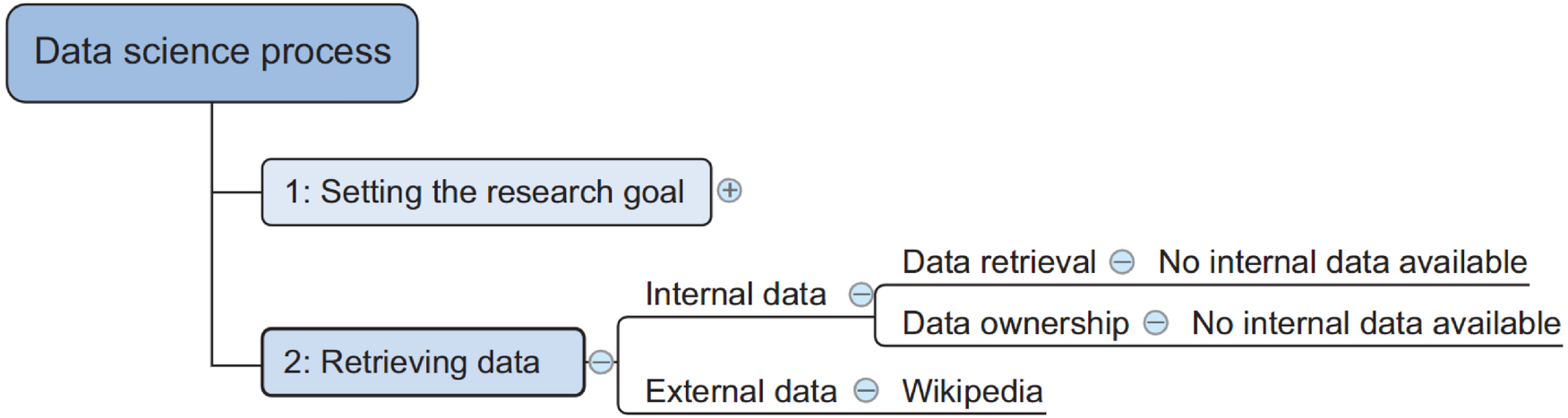


Figure 6.18 Data science process step 2: data retrieval. In this case there's no internal data; all data will be fetched from Wikipedia.

STEP 2 & 3: DATA RETRIEVAL & PREPARATION

Once data has entered the Elasticsearch index, it can't be altered; all you can do then is query it. So we'll need to prepare the data first.
As shown in figure 6.19 there are three distinct categories of data preparation toconsider:

- Data cleansing
- Data transformation
- Combining data

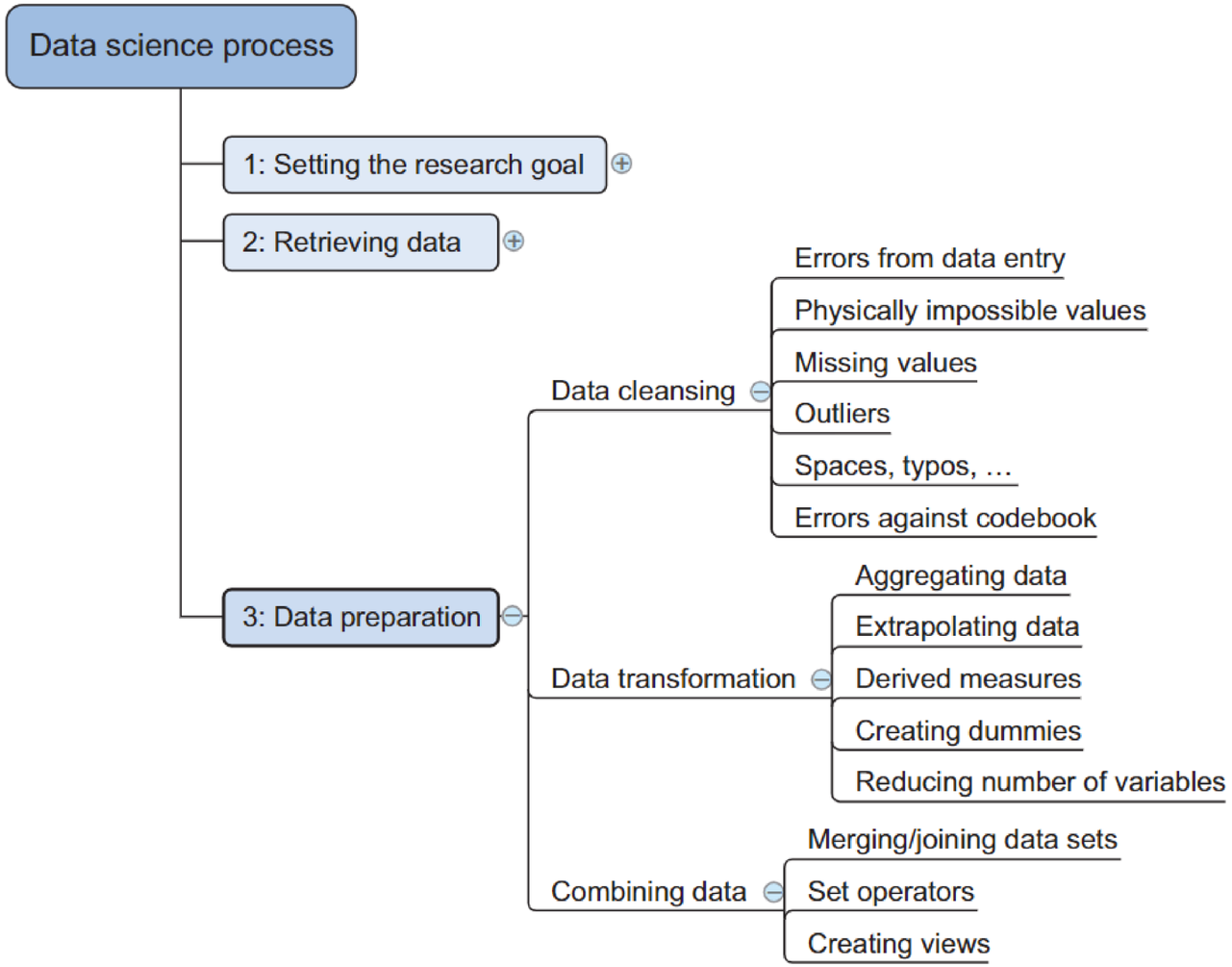


Figure 6.19 Data science process step 3: data preparation

LET'S START WITH PYTHON

Please read the Official Docs of Elasticsearch to know how to install locally with .zip file on Windows or any platform.

Active our Python virtual environment and install libraries we need for the book:

- elasticsearch: for communicate with our database
- wikipedia: to tap into Wikipedia

```
pip install elasticsearch

pip install wikipedia
```

THE FIRST THING YOU NEED IS A CLIENT!

Don't forget:

— Download and install elasticsearch and kibana .zip file
— Then unzip it and run the executables from bin folder
— Elasticsearch will run on port 9200 and Kibana on port 5601

```
# Elasticsearch client used to communicate with database
client = Elasticsearch('http://localhost:9200')
indexName = "medical" #index name
client.indices.create(index=indexName) # create index
```

[20] ✓ 0.2s Python

```
{'acknowledged': True, 'shards_acknowledged': True, 'index': 'medical'}
```

CREATE A MAPPING TO DOCUMENT TYPE

Although this is true for simple cases, you can't avoid having aschema in the long run, so let's create one, as shown in the following listing.

Note: '***string***' data type is not supported from Elasticsearch 7.x version. Instead, we use the '***text***' for full text search or keyword data type for exact matches.

```
#create a mapping and attribute it to the disease doc type
diseaseMapping = {
        'properties': {
            'name': {'type': 'text'},
            'title': {'type': 'text'},
            'fulltext': {'type': 'text'}
        }
    }
# The "diseases" doc type is updated with a mapping. Now we define the data it should expect.
client.indices.put_mapping(index=indexName,doc_type='diseases',body=diseaseMapping,
include_type_name=True)
```

[22] ✓ 0.1s Python

LET’S MOVE ON TO WIKIPEDIA

The first thing you want to do is fetch the List ofdiseases page, because this is your entry point for further exploration:

```
dl = wikipedia.page("Lists_of_diseases")
```

But we’re more interested in the listing pages because they contain links to the diseases. Check out the links:

```
dl.links
```

LET’S MOVE ON TO WIKIPEDIA

And then we have the List of diseases page comes with more links than we'll use.

But only the alphabetic lists interest you, so keep only those

```
['Airborne disease',
 'Contagious disease',
 'Cryptogenic disease',
 'Disease',
 'Disseminated disease',
 'Endocrine disease',
 'Environmental disease',
 'Eye disease',
 'Health On the Net Foundation',
 'Lifestyle disease',
 'List of abbreviations for diseases and disorders',
 'List of autoimmune diseases',
 'List of cancer types',
 'List of childhood diseases and disorders',
```

You've probably noticed that the subset is hardcoded, because you know they're the 16th to 43rd entries in the array. If Wikipedia were to add even a single link before the ones you're interested in, it would throw off the results.

```
diseaseListArray = []

for link in dl.links:
    if link.startswith("List of diseases"):
        try:
            page_title = link
            page = wiki_wiki.page(page_title)

            # Check if the page exists
            if page.exists():
                diseaseListArray.append(page)
            else:
                # Try alternative titles or variations
                variations = [
                    page_title,
                    f"List_of_diseases_{page_title[-1]}",
                    f"List_of_diseases_{page_title[-1].upper()}"
                ]
                for variation in variations:
                    alt_page = wiki_wiki.page(variation)
                    if alt_page.exists():
                        diseaseListArray.append(alt_page)
                        break
                else:
```

STEP 4: DATA EXPLORATION

Data exploration is what marks this case study, because the primary goal of the project (disease diagnostics) is a specific way of exploring the data by querying for disease symptoms. Figure 6.26 shows several data exploration techniques, but in this case it's non-graphical: interpreting text search query results

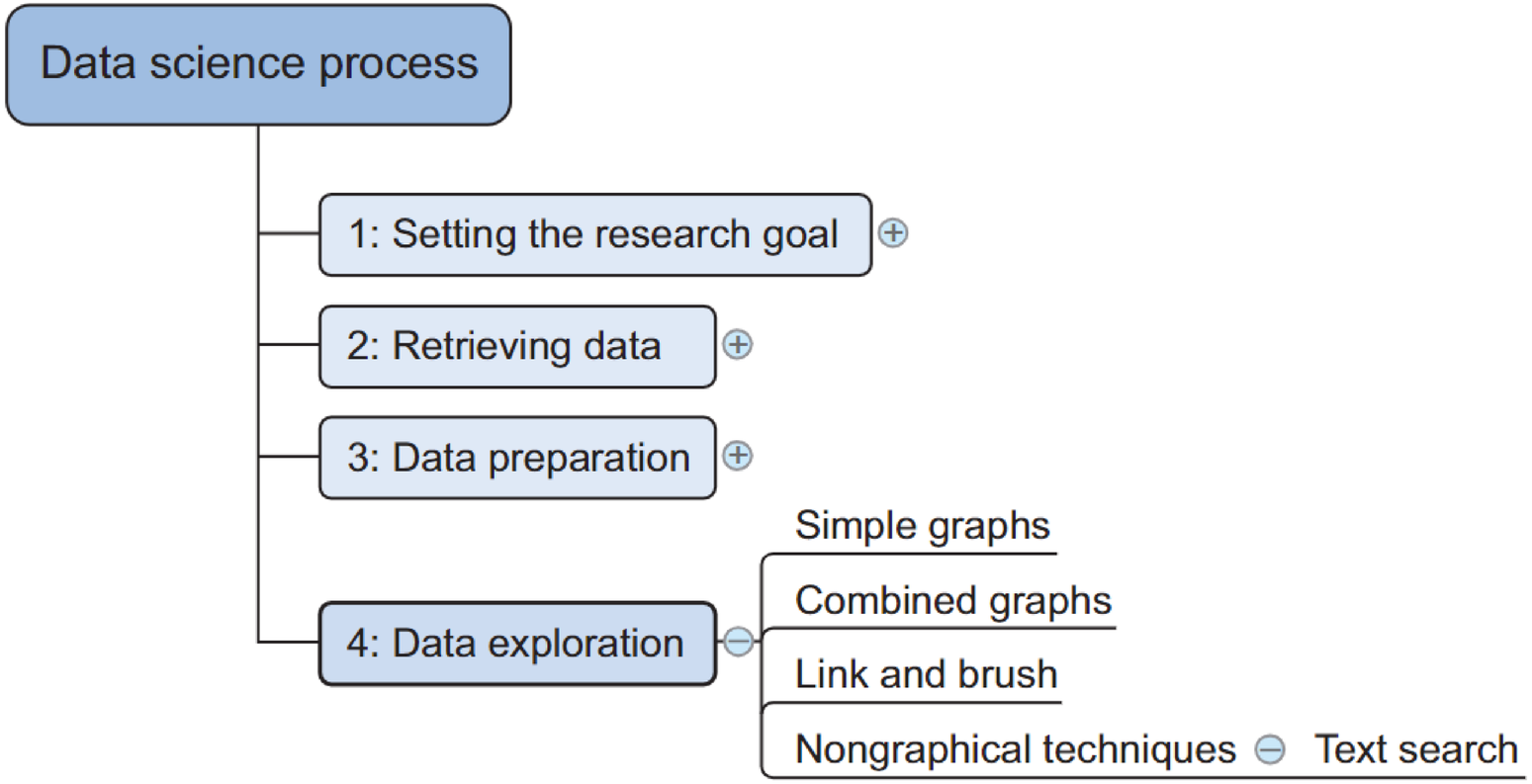


Figure 6.26 Data science process step 4: data exploration

STEP 4: DATA EXPLORATION

The moment of truth is here: can you find certain diseases by feeding your search engine their symptoms? Let’s first make sure you have the basics up and running. Import the Elasticsearch library and define global search settings:

```
# Elasticsearch client used to communicate with database
client = Elasticsearch('http://localhost:9200')
# indexName = "medical" #index name
# client.indices.create(index=indexName) # create index
docType="diseases"
searchFrom = 0
searchSize= 3
```

✓ 0.0s Python

STEP 4: DATA EXPLORATION

Lupus is a type of autoimmune disease, where the body's immune system attacks healthy parts of the body. Let's see what symptoms your search engine would need to determine that you're looking for lupus.

Start off with three symptoms: fatigue, fever, and joint pain. Your imaginary patient has all three of them (and more), so make them all mandatory by adding a plus sign before each one

FINDING LUPUS FIRST TRIAL

```
searchBody={
    "fields":["name"],
    "query":{
        "simple_query_string" : {
            "query": '+fatigue+fever+"joint pain"',
            "fields": ["fulltext","title^5","name^10"]
        }
    }
}
client.search(index=indexName, doc_type=docType, body=searchBody, from_=searchFrom,
size=searchSize)
```

✓ 0.0s Python

STEP 4: DATA EXPLORATION

The dictionary named searchBody contains the search request information we'll send.

We want the name field in our results.

The query part. Other things are possible here, like aggregations. More on that later.

A simple query string is a type of query that takes input in much the same way the Google homepage would.

These fields are the fields in which it needs to search. They are not to be confused with the fields it has to return in the search results (specified in the second code line above).

```
searchBody={
"fields":["name"],
"query":{
    "simple_query_string" : {
        "query": '+fatigue+fever+"joint pain"',
        "fields": ["fulltext","title^5","name^10"]
        }
    }
}
client.search(index=indexName,doc_type=docType, body=searchBody, from_ =
     searchFrom, size=searchSize)
```

Like a query on Google the + sign indicates the term is mandatory. Encapsulating two or more words in quotes signals you want to find them exactly like this.

The search is executed. Variables indexName, docType, searchFrom, and searchSize were declared earlier: indexName = "medical" , docType="diseases" , searchFrom = 0 , searchSize = 3.

RESULT OF SEARCH

In searchBody, which has a JSON structure, you specify the fields you'd like to see returned, in this case the name of the disease should suffice. You use the query string syntax to search in all the indexed fields: fulltext, title, and name. By adding ^ you can give each field a weight. If a symptom occurs in the title, it's five times more important than in the open text; if it occurs in the name itself, it's considered ten times as important. Notice how "joint pain" is wrapped in a pair of quotation marks. If you didn't have the " " signs, joint and pain would have been considered as two separate keywords rather than a single phrase.

```
'hits': [{'_index': 'medical',
  '_type': 'diseases',
  '_id': 'Acute gouty arthritis',
  '_score': 6.2221518,
  '_source': {'name': 'Acute gouty arthritis',
   'title': 'Gout',
   'fulltext': 'Gout ( GOWT) is a form of inflammatory arthritis cha
  'fields': {'name': ['Acute gouty arthritis']}},
 {'_index': 'medical',
  '_type': 'diseases',
  '_id': 'Acute lymphoblastic leukemia',
  '_score': 5.11135,
  '_source': {'name': 'Acute lymphoblastic leukemia',
   'title': 'Acute lymphoblastic leukemia',
   'fulltext': 'Acute lymphoblastic leukemia (ALL) is a cancer of th
  'fields': {'name': ['Acute lymphoblastic leukemia']}},
 {'_index': 'medical',
  '_type': 'diseases',
  '_id': 'Acute lymphocytic leukemia',
  '_score': 5.11135,
  '_source': {'name': 'Acute lymphocytic leukemia',
   'title': 'Acute lymphoblastic leukemia',
   'fulltext': 'Acute lymphoblastic leukemia (ALL) is a cancer of th
  'fields': {'name': ['Acute lymphocytic leukemia']}}]}
```


RESULT OF SEARCH

The results are sorted by their matching score, the variable _score.
The matching score is no simple thing to explain; it takes into consideration how well the disease matches your query and how many times a keyword was found, the weights you gave, and so on.

LATER TRIAL

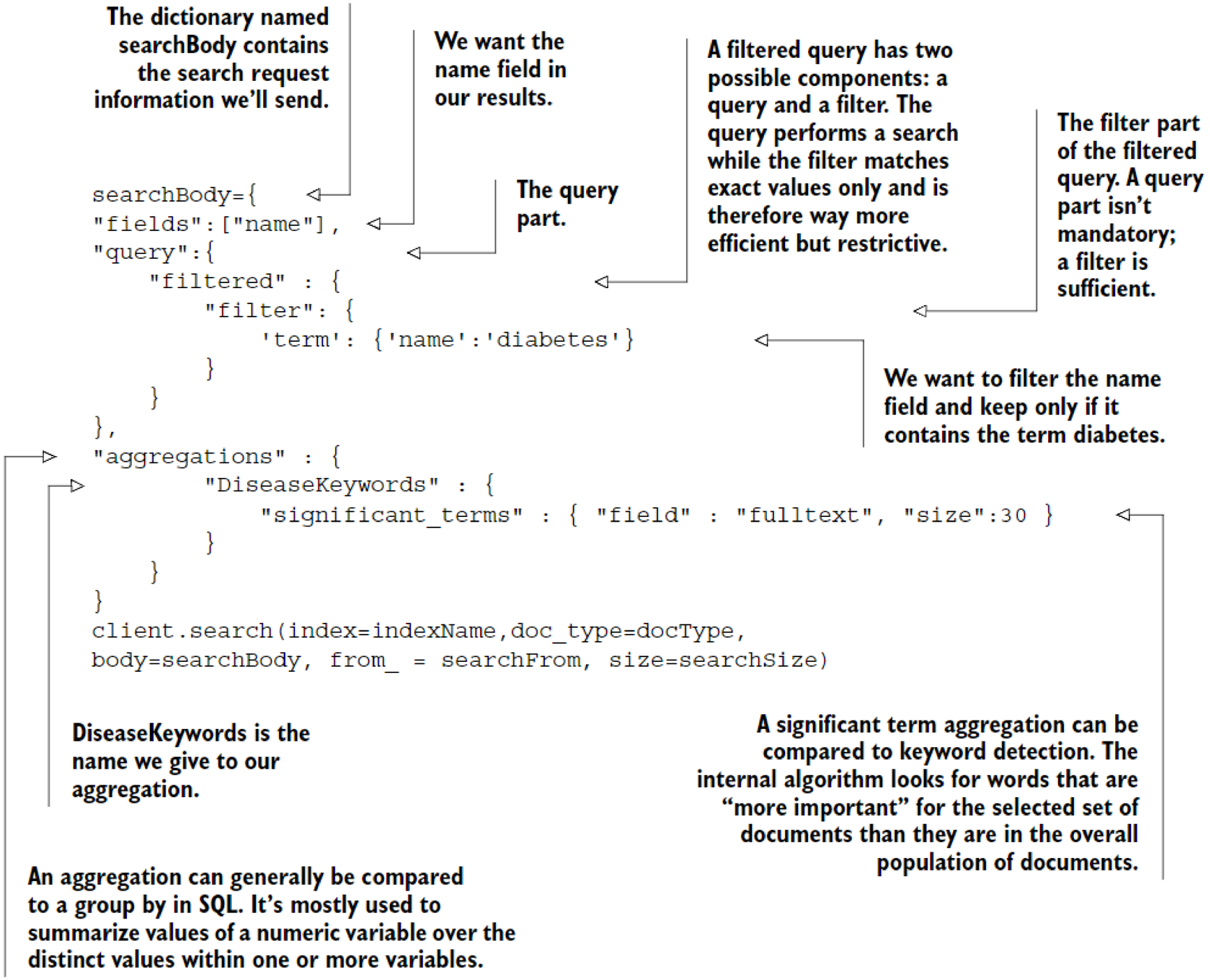
So on and add other symptoms and finally get the desired result. In here we add: "query": '+fatigue+fever+"joint pain"+rash+"chest pain"',

```
{u'_shards': {u'failed': 0, u'successful': 5, u'total': 5},
 u'hits': {u'hits': [{u'_id': u'Lupus erythematosus',
    u'_index': u'medical',
    u'_score': 0.010452312,
    u'_type': u'diseases',
    u'fields': {u'name': [u'Lupus erythematosus']}}],
  u'max_score': 0.010452312,
  u'total': 1},
 u'timed_out': False,
 u'took': 11}
```

Figure 6.29 Lupus third search: with enough symptoms to determine it must be lupus

PROJECT SECONDARY OBJECTIVE

We’ve once again arrived at data exploration. We can adapt the aggregations query and use your new field to give you bigram key concepts related to diabetes:


The dictionary named searchBody contains the search request information we'll send.
We want the name field in our results.
The query part.
A filtered query has two possible components: a query and a filter. The query performs a search while the filter matches exact values only and is therefore way more efficient but restrictive.
The filter part of the filtered query. A query part isn't mandatory; a filter is sufficient.
We want to filter the name field and keep only if it contains the term diabetes.
searchBody={
"fields":["name"],
"query":{
"filtered" : {
"filter": {
'term': {'name':'diabetes'}
}
}
},
"aggregations" : {
"DiseaseKeywords" : {
"significant_terms" : { "field" : "fulltext", "size":30 }
}
}
}
client.search(index=indexName,doc_type=docType,
body=searchBody, from_ = searchFrom, size=searchSize)
DiseaseKeywords is the name we give to our aggregation.
A significant term aggregation can be compared to keyword detection. The internal algorithm looks for words that are "more important" for the selected set of documents than they are in the overall population of documents.
An aggregation can generally be compared to a group by in SQL. It's mostly used to summarize values of a numeric variable over the distinct values within one or more variables.

PROJECT SECONDARY OBJECTIVE

Although it won’t show on the small amount of data at your disposal, the filter is a way faster than the search. A search query will calculate a search score for each of the dis-eases and rank them accordingly, whereas a filter simply filters out all those that don’t comply.

A filter is thus far less complex than an actual search: it’s either “yes” or “no” and this is evident in the output. The score is 1 for everything; no distinction is made within the result set. The output consists of two parts now because of the significant terms aggregation. Before you only had hits; now you have hits and aggregations. First, have a look at the hits in figure 6.31

```
u'hits': {u'hits': [{u'_id': u'Diabetes mellitus',
   u'_index': u'medical',
   u'_score': 1.0,
   u'_type': u'diseases',
   u'fields': {u'name': [u'Diabetes mellitus']}},
  {u'_id': u'Diabetes insipidus, nephrogenic type 3',
   u'_index': u'medical',
   u'_score': 1.0,
   u'_type': u'diseases',
   u'fields': {u'name': [u'Diabetes insipidus, nephrogenic type 3']}},
  {u'_id': u'Ectodermal dysplasia arthrogryposis diabetes mellitus',
   u'_index': u'medical',
   u'_score': 1.0,
   u'_type': u'diseases',
   u'fields': {u'name': [u'Ectodermal dysplasia arthrogryposis diabetes mellitus']}}],
 u'max_score': 1.0,
 u'total': 27},
u'timed_out': False,
u'took': 44}
```

Figure 6.31 Hits output of filtered query with the filter “diabetes” on disease name

PROJECT SECONDARY OBJECTIVE

The keywords are returned in lowercase because they are stored in the index using the standard analyzer. We didn't specify any custom settings during indexing, so the default standard analyzer was used. Some interesting keywords in the top 30 include "avp" (a gene related to diabetes), "thirst" (a symptom of diabetes), and "Amiloride" (a medication for diabetes). However, we are missing multi-term keywords because we only stored individual terms in the index by default.

PROJECT SECONDARY OBJECTIVE

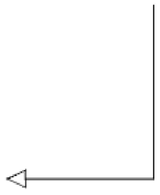
Certain terms only make sense when combined with others, but we currently miss out on those relationships. Storing n-grams (combinations of words) would capture such relationships, but it comes with storage and performance trade-offs. Determining the optimal level of n-grams depends on your specific data and use case. Generally, bigrams (combination of two terms) can be useful for disease profiling. To create meaningful bigram aggregations, you would need to store them as bigrams in the index. This often requires revisiting the data preparation phase and making necessary changes.

STEP 3 REVISITED: DATA PREPARATION FOR DISEASE PROFILING

We will prepare the data for disease profiling. This involves applying techniques to clean and transform the data that was collected in Step 2. It is important to note that data preparation is an iterative process in the data science process, and when the data was initially indexed, virtually no data cleansing or transformation was done. Therefore, we need to apply techniques such as stop word filtering to clean and transform the data. Stop words are common words that are often discarded because they can pollute the results. It is worth mentioning that stop word filtering is just one example of a technique that can be used for data cleansing, and we can try other techniques as well.

UPDATING ELASTICSEARCH INDEX SETTINGS

```
settings={
    "analysis": {
            "filter": {
                "my_shingle_filter": {
                    "type":             "shingle",
                    "min_shingle_size": 2,
                    "max_shingle_size": 2,
                    "output_unigrams":  False
                }
            },
            "analyzer": {
                "my_shingle_analyzer": {
                    "type":             "custom",
                    "tokenizer":        "standard",
                    "filter": [
                        "lowercase",
                        "my_shingle_filter"
                    ]
                }
            }
        }
    }
client.indices.close(index=indexName)
client.indices.put_settings(index=indexName , body = settings)
client.indices.open(index=indexName)
```



Before you can change certain settings, the index needs to be closed. After changing the settings, you can reopen the index.

UPDATING ELASTICSEARCH INDEX SETTINGS

We create two new elements: the token filter called “my shingle filter” and a newanalyzer called “my_shingle_analyzer.” Because n-grams are so common, Elastic-search comes with a built-in shingle token filter type. All you need to tell it is that you want the bigrams “min_shingle_size” : 2, “max_shingle_size” : 2, as shown in figure 6.34. You could go for trigrams and higher, but for demonstration purposes this will suffice.

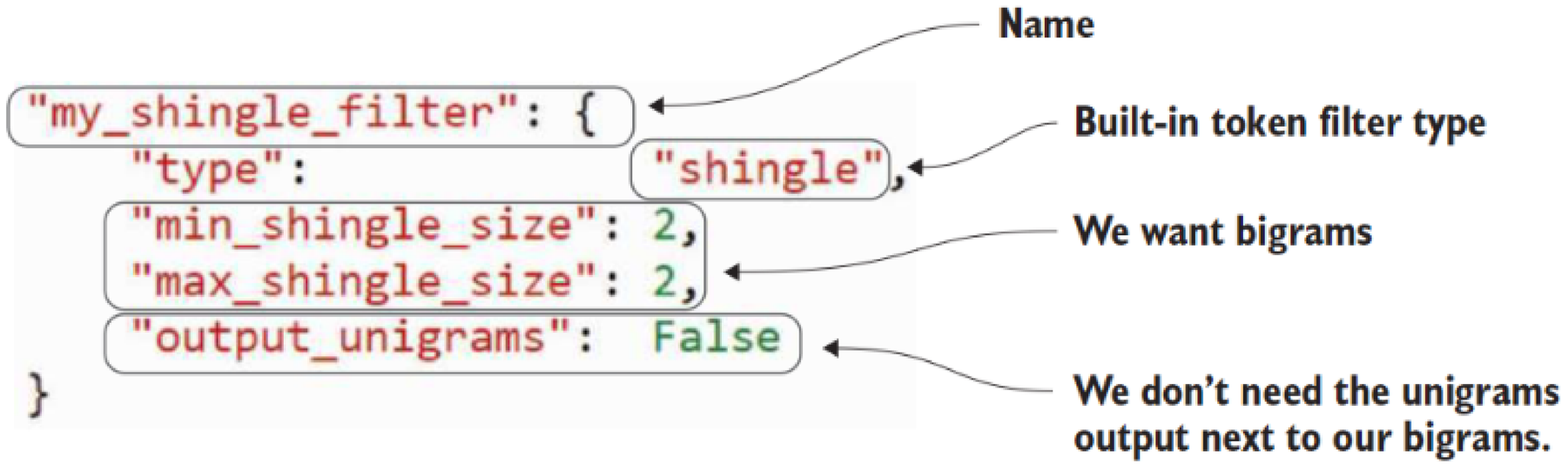


Figure 6.34 A shingle token filter to produce bigrams

UPDATING ELASTICSEARCH INDEX SETTINGS

The analyzer shown in figure 6.35 is the combination of all the operations required to go from input text to index. It incorporates the shingle filter, but it’s much more than this. The tokenizer splits the text into tokens or terms; you can then use a lowercase filter so there’s no difference when searching for “Diabetes” versus “diabetes”. Finally, you apply your shingle filter, creating your bigrams

UPDATING ELASTICSEARCH INDEX SETTINGS

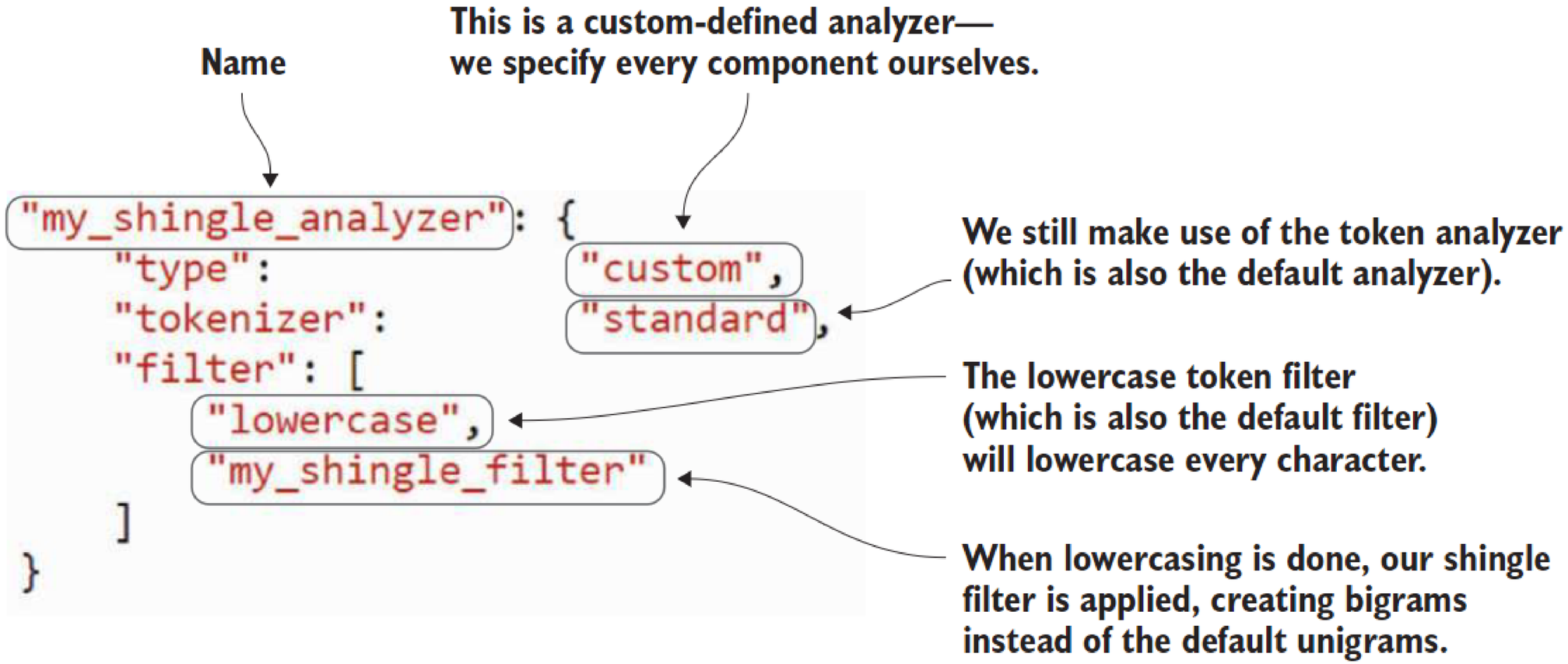


Figure 6.35 A custom analyzer with standard tokenization and a shingle token filter to produce bigrams

CREATE MORE ADVANCED ELASTICSEARCH DOCTYPE MAPPING

```
docType = 'diseases2' #document type we will index

diseaseMapping = {
        'properties': {
            'name': {'type': 'string'},
            'title': {'type': 'string'},
            'fulltext': {
                "type": "string",
                "fields": {
                    "shingles": {
                        "type":     "string",
                        "analyzer": "my_shingle_analyzer"
                    }
                }
            }
        }
    }
```

The new disease mapping differs from the old one by the addition of the fulltext.shingles field that contains your bigrams.

UPDATING ELASTICSEARCH INDEX SETTINGS

Within fulltext you now have an extra parameter, fields. Here you can specify all the different isotopes of fulltext. You have only one; it goes by the name shingles and will analyze the fulltext with your new my_shingle_analyzer. You still have access to your original fulltext, and you didn’t specify an analyzer for this, so the standard one will be used as before. You can access the new one by giving the property name followed by its field name: fulltext.shingles. All you need to do now is go through the previous steps and index the data using the Wikipedia API, as shown inthe following listing

STEP 4 REVISITED: DATA EXPLORATION FOR DISEASE PROFILING

We've once again arrived at data exploration. You can adapt the aggregations query and use your new field to give you bigram key concepts related to diabetes:

IGNIFICANT TERMS AGGREGATION ON “DIABETES” WITH BIGRAMS

```
searchBody={
"fields":["name"],
"query":{
    "bool": {
        "filter": {
            'term': {'name':'diabetes'}
            }
        }
},
"aggregations" : {
        "DiseaseKeywords" : {
            "significant_terms" : { "field" : "fulltext", "size" : 30 }
        },
        "DiseaseBigrams": {
            "significant_terms" : { "field" : "fulltext.shingles", "size" : 30 }
        }
    }
}
client.search(index=indexName,doc_type=docType, body=searchBody, from_ = searchFrom,
size=searchSize)
```

✓ 0.0s Python

```
{u'_shards': {u'failed': 0, u'successful': 5, u'total': 5},
 u'aggregations': {u'DiseaseBigrams': {u'buckets': [{u'bg_count': 22,
     u'doc_count': 10,
     u'key': u'the diabetes',
     u'score': 63.434343434343425},
    {u'bg_count': 18,
     u'doc_count': 9,
     u'key': u'a passer',
     u'score': 62.833333333333314},
    {u'bg_count': 18,
     u'doc_count': 9,
     u'key': u'excessive discharge',
     u'score': 62.833333333333314},
    {u'bg_count': 18,
     u'doc_count': 9,
     u'key': u'passer through',
     u'score': 62.833333333333314},
    {u'bg_count': 20,
     u'doc_count': 9,
     u'key': u'from diabetes',
```


IGNIFICANT TERMS AGGREGATION ON “DIABETES” WITH BIGRAMS

STEP 4 REVISITED: DATA EXPLORATION FOR DISEASE PROFILING

There are other interesting bigrams, unigrams, and probably also trigrams. Taken asa whole, they can be used to analyze a text or a collection of texts before reading them. Notice that you achieved the desired results without getting to the modeling stage. Sometimes there’s at least an equal amount of valuable information to be found in data exploration as in data modeling. Now that you’ve fully achieved your secondary objective, you can move on to step 6 of the data science process: presentation and automation.

STEP 6: PRESENTATION AND AUTOMATION

We focus on presentation and automation. Our primary objective is to create a self-service diagnostics tool that physicians can query via a web application. Although we do not provide instructions on how to build a website, we suggest reading the sidebar "Elasticsearch for web applications" if you plan on doing so. Additionally, we can take our secondary objective of disease profiling to the next level by producing a word cloud that visually summarizes search results. To set up something like this, we recommend using the word_cloud library in Python. By automating these processes and creating user-friendly interfaces, we can make our data more accessible and useful for stakeholders.

Thank You

Summary by:

A41316 - Nguyễn Hữu Khoa
huukhoa203@gmail.com

A42718 - Lê Thảo Quyên
irisle2003@gmail.com