Piotr Fulmański

NoSQL
Theory and examples

Early Access Version
Edition 1.0, October 2021

Simple Introduction Series

NoSQL. Theory and examples

SIMPLE INTRODUCTION SERIES

www: https://fulmanski.pl
email: book@fulmanski.pl
GitHub: https://github.com/fulmanp/NoSQL-Theory-and-examples

Edition: 1
First published: 1.0, January 12, 2022 (planned)
This edition: 1.0, October 2021 (early access)
Build number: 202110302359

eBook (pdf, epub)
ISBN-13: 978-83-957405-0-3

While the author has used good faith efforts to ensure that the information and instructions contained in this work are accurate, the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. The author makes no warranty, express or implied, with respect to the material contained herein.

If any code samples, software or any other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

All trademarks and registered trademarks appearing in this book are the property of their own respective owners.

Release Notes

(**A** – add, **N** – new, **U** – update)

Edition 1.0
release: July 2021

- Chapter: 9: Time series databases (**A**, **U**)

Edition 1.0
release: February 2021

- Chapter: 9: Time series databases (**N**)

Edition 1.0
release: January 2021

- Chapter: 10: Apache Pig (**N**)
- Change text highlighting (**U**)

Edition 1.0
release: October 2020, November 2020

- Fix layout

Edition 1.0
release: September 2020

- Migrate from iBooks Author to Pages as iBooks Author is no longer updated since July 1, 2020.
- Preface (**U**)
- Chapter 1: SQL, NoSQL, NewSQL (**U**)
- Chapter 7: Graph stores (**N**)
- Appendix D: SQL recursive queries (**N**)

Edition 1.0
release: May 2020

- Chapter 1: SQL, NoSQL, NewSQL (**U**)
- Chapter 3: NoSQL (**N**)
- Chapter 4: Column family stores (**N**)
- Chapter 6: Document store (**N**)

- Appendix A: JSON (**N**)
- Appendix B: XML (**N**)
- Appendix C: HTTP (**N**)

Edition 1.0
release: April 2020

- Preface (**N**)
- Chapter 1: SQL, NoSQL, NewSQL (**N**)
- Chapter 2: SQL. Relational model (**N**)
- Chapter 3: Key-value stores (**N**)

Table of contents

Preface

Who this book is for

This book is addressed to **all the people who want to understand what a NoSQL is** and what were the reasons behind its arisen. It may be good both for someone with no computer scientist background and for those who have some IT experience but want to understand why the tools they work every day looks this and no other way. I don't want to dive deep into details of specific technologies or solutions. Instead of that, I want to **explain why things are as they are**. Whenever it is possible a general way of thinking is presented. Just as you can't eat a cookie only by reading about it, it will be difficult to understand NoSQL without practical examples. Therefore, **examples of the selected databases basic usage are an integral part of presented contents**. I will skip all the details related to database features or installation process that can be found on dedicated web pages. All examples are intended to give basic understanding how to **C**reate, **R**ead, **U**pdate and **D**elete data (so called CRUD set of operations). After reading this book you should be able to:

- basic use of selected databases from the NoSQL family;
- choose the right base depending on the task to be solved;
- know the strengths and weaknesses of the proposed solutions, both those based on SQL and NoSQL databases.

Early access

This book is a work in progress, presented in early access version. Early access allows to publish and share some ideas before the final version appears. This way, participating in early access, you may contribute how the final version will look like. English is not my native language and I know that I make a lot of mistakes but I hope that text is more than readable and at least a little bit understandable. I believe that everything can be better and there is always a space for improvements. I can say that the Toyota Way is the way I live and work focusing on continuous improvement, and respect for people. That is why I would be very grateful if you somehow contribute improving this book. Any comments, corrections and suggestions are more than welcome. I write this book not for myself but to share what I know with others, so help me make it's contents better.

If this book is buggy or not complete yet why it's not free? Simply because I need money to finish it and to make it better. Everything costs. The most precious is time. I, as all other people, have to work to live and to support my family. And this consumes most of my days. Then I have a choice: play with kids or write a book. I choose playing with kids. I don't want to reduce a time spent with my family because I don't want to be a virtual parent. So I have just a little time for book. If I could reduce my job engagement (from full-time to half-time) I could spend more time on book.

I believe that there is no book like this on the market and I want to make it better and better. I can do this. I don't have to ask publisher if they agree to prepare another version. If something deserves for improvement I simply do this and publish right after that. Paying for a book you allow me to spending more time on book without sacrificing my family life. Having money I can pay for professional translation, text correction or simply buy better images.

What will you learn in this book?

I don't want to write this book and forget. My idea is to keep it as accurate and up to date as it is only possible so you can expect updates in a future even if I reach stable "final" version. As I wrote above, always there is something to improve. As for now book covers the following topics (chapter titles highlighted in red indicate finished or, if stated explicitly, in progress content).

Preface This is what you are reading right now. Here I explain what you can expect in this book. I also try to convince you to actively participate in shaping it's contents.

Chapter 1 SQL, NoSQL, NewSQL In this chapter a general overview of a huge *SQL databases is given. I will try to answer for the question why there are so many different families of databases. Are they really needed or it is just a mindless rush? If we need them, what factors plays crucial role? What was the main reasons behind the new databases types invention? What profits do we have replacing one database by other? Is it probable that one type will replace other?
This chapter introduces concepts that will be discussed in more details in the following chapters (chapter 2 and 3).

Chapter 2 SQL. Relational model Although this book is about NoSQL databases, it is good to have some reference point. Relational (SQL) seems to be perfect choice because it is quite natural to consider every novelty in context of a well known model used for many years. In this chapter we will point out key features of SQL model, its strengths and weakness.

Chapter 3 NoSQL In this chapter we will discuss when and why SQL is not enough for business and IT enterprises. What factors plays crucial role in nowadays system development. We will try to justify that NoSQL

rising was a natural consequence of the changes taking place in the world.

Chapter 4 Column family stores In this section a foundations of column family stores will be given.

Chapter 5 Key-value stores In this section a foundations of key-value stores will be given.

Chapter 6 Document stores In this section a foundations of document stores will be given.

Chapter 7 Graph stores In this section a foundations of graph stores will be given.

Chapter 8 Column databases In this section a foundations of column databases will be given. (IN PROGRESS)

Chapter 9 Time series databases In this section a foundations of time series databases will be given. (IN PROGRESS)

Chapter 10 Apache Pig In all previous chapters I talk about "classic" NoSQL, being more precisely: about NoSQL Stores/Databases. But what does it mean: *database* or *store* in our everyday practice? Do you care about its internals, about its logical and/or physical data storage model? Or rather you take care about what you can do and how you can do? Saying the truth, if only we have an SQL-like language we can use to operate on "something" then we will call this "something" a (relational) database, even if, technically speaking, it would be something very different than (relational) database. Some system don't have to be a database to be used as a database. And this is the main thought in this and further chapter: I will show you some systems which behaves like database even though they are not databases. In most cases, no one calls them NoSQL systems but it's seems to be a future of NoSQL systems:

SQL-like interface created over underlying sophisticated and versatile storage system to work with any kind of data.

Appendix A: JSON

Appendix B: XML

Appendix C: HTTP

Appendix D: SQL recursive queries

What you will NOT learn in this book?

This book is not intended to be a user guide for a given NoSQL database. Even if I use some of them I do this to show key features of a whole class of databases, their pros and cons. Today everything changes so fast that it may happen that database used in examples is no more supported. Writing a book is a time consuming task and, especially in IT area, some fact described at the beginning of the book ma be outdated when author complete final chapters. It does not change the general nature of the considerations contained in this book as I made an effort to present base, universal and essential functionality which is quite common among databases of a given type. That is why **you will not find a chapters devoted to installation or configuration** (except some simple cases when necessary).

The number of topics covered in this book is quit big, and it shouldn't surprise you that some of them are only announced. There is no way to fully describe all of them. I don't think it's possible to know everything. There's a tremendous amount of information we get every day because the technology is evolving so fast. **Let me know if you think that some parts should be described in more details or in a completely different way.**

Give this book a try, and please let me know what you think. Any feedback is very much encouraged and welcomed! If you think that my time is worth this effort, you can support what I'm doing now and help me finalize this project. Please use email (book@fulmanski.pl) or GitHub (https://github.com/fulmanp/NoSQL-Theory-and-examples) to give your positive or negative, but in all cases constructive, feedback.

Thank you for your engagement.

Piotr Fulmański

Conventions used in this book

For your convenience I will use the following typographical convention:

Italic
Indicates new terms.

Italic
Indicates old terms but for some reason I want to distinguish them from normal text flow, definitions, citations.

`Constant width`
Indicates source code, filenames, file extensions, variables, parameters, etc.

`Constant width`
Indicates commands or any other text that you should type literally (as it is given).

`Constant width`
Indicates parts of scripts or commands which you need to pay special attention to.

Bold
Indicates statements which you need to pay special attention to. Sometimes it is used in combination with previous styles, for example:

`Constant width with` **`bolded`** `part`

This way I will mark for example crucial parameter in some important command.

```
This is how source
code is displayed
```

```
This is how a terminal
text is displayed with
bolded command prompt.
```

Something worth to remember or just one-sentence summary of some part of a section or chapter.

NOTE

Note block

I use this block to give you some additional explanation or information, possibly loosely related to a main text.

CHAPTER 1

SQL, NoSQL, NewSQL

General overview of *SQL databases

- Why there are so many different families of databases?
- What factors were the main reason for NoSQL to appear?
- Is NoSQL going to replace SQL?

Data and database

We can define **database** as *an organized collection of data stored in accordance with specific rules*. In this sense postage stamps collection, books stored in a shelf (in some systematic or chaotic way) or even kid's cars collection are examples of databases. In practice we used to think about databases in much more narrower and thus more useful sense.

First, we think about pure immaterial data – we store numbers, texts, images and songs but not real objects. Probably because real objects are much more harder to manipulate in an automatic way than sequences of characters.

We define **data** as *a set of values of qualitative or quantitative variables (properties) describing some object or phenomenon.*

Although the terms *data*, *information* and *knowledge* are often used interchangeably, each of these terms has a distinct meaning. Data is a dumb set of values. Nothing more. When the data is processed and transformed in such a way that it becomes useful to the users, it is known as information. So when data starts to "speak", when something valueless is turned into priceless, we have an information. Going further with other data "transformations" we reach to DIKW (data, information, knowledge, wisdom) pyramid. The DIKW pyramid shows that *data*, produced by events, can be enriched with context to create **information**, information can be supplied with meaning to create **knowledge** and knowledge can be integrated to form **wisdom**, which is at the top. There is a nice saying (by Miles Kington):

Knowledge is knowing a tomato is a fruit.
Wisdom is not putting it in a fruit salad.

And this is a true essence of the problem we are discuss.

As an example you can consider number 38. This is a pure number and the only fact you can say about it is that it consist of three tens and eight ones, but even that is not certain, as the number may not be in decimal system. At this moment you can call 38 a ***data***. If I will tell you, that this number represents temperature, you will have a deeper understanding of this pure number. I may add, that temperature in kelvins and then you will know that it is rather cold or I may say, that temperature of may body expressed in centigrade scale (Celsius degrees, typically used in my country; 38 degree Celsius = 100.4 degree Fahrenheit = 311.15 kelvin) which means I am sick and have a fever. Now data becomes an ***information*** because you know what this number means. Going further, if you combine this information with observation how I behave, look and fill you will obtain a ***knowledge*** of a certain disease. ***Wisdom*** is at the top of this data transformation process and in this case explains all the steps we take when we notice 38 centigrade on a thermometer.

Different data forms required different storage methods. For data it is enough to use a simple text file. Information is quite well saved in relational databases or some simple NoSQL stores. Knowledge demands more sophisticated system like graph databases or a search engine like Elasticsearch.

NOTE

One data, many data...

The Latin word *data* is the plural of *datum* (en. *(thing) given*) neuter past participle of *dare* (en. *to give*). In consequence, *datum* should be

used in the singular and *data* for plural, though, in non-specialist, everyday writing, *data* is most commonly used in the singular, as a mass noun (like *information*, *sand* or *rain*) and this is becoming more and more popular. The first English use of the word *data* is from the 1640s. Using the word *data* to mean *transmittable and storable computer information* was first done in 1946. The expression *data processing* was first used in 1954. [D]

Second, we pay a great attention to automatic way of processing. The best known tool allowing us to do so nowadays are computers. That is why immaterial data is so useful for us – we can turn them into digital data and feed them a computer systems to make them do for us things we won't do ourself.

This should explains why nowadays we define database as a *digital data collected in accordance with the rules adopted for a given computer program specialized for collecting, storing and processing this data*. Such a program (often a package of various programs) is called a *database management system* (DBMS).

The database management system (DBMS) is the software that interacts with end users, applications, and the database itself to capture and analyze the data. It serves as an intermediate layer isolating end user from all "unnecessary" technical details. In common language we use the term *database* to loosely refer to any of the DBMS, the database system or an application associated with the database.

> *System don't have to be a database to be used as a database.*

Making one step forward, you can say that system don't have to be a database to be used as a database. And this is the main thought in last chapters of this book. I show there some systems which behaves like

database even though they are not databases. In most cases, no one calls them database systems but from user's perspective, thanks to SQL-like interface created over underlying sophisticated and versatile storage system allowing to work with any kind of data we perceive them as databases.

Database or store?

A *datastore* (*store*) is, as the name indicates, a place where data is stored. The simplest example of a store is a flat file saved on your disk. You can also save data in a *database*, in which the data are stored physically in files, but those files are managed by some, very often sophisticated, management system. Viewed from this perspective, database are a special type of datastore. Not all NoSQL databases have a builtin "manager", or their functionality is very limited, so the management is done in the application level. That is why you may see them just as an another one storage system. Simply speaking, simple NoSQL databases (for example key-value) are very often referred as a *store* while those more complicated (graph for example) as a *database*, but this is not rule of the thumb.

SQL

We can classify database-management systems according to the database models that they support. Not going far into the past we can say that first large-scale used model, dominant in the market for more than 20 years, were relational databases arise in the 1970s. We refer them as **SQL** databases because **S**tructured **Q**uery **L**anguage (pronounce it as *S-Q-L* or *sequel*) was used by the vast majority of them for writing and querying data. SQL (in a sense: SQL databases) utilizes Edgar F. Codd's **relational model** based on tabular data representation:

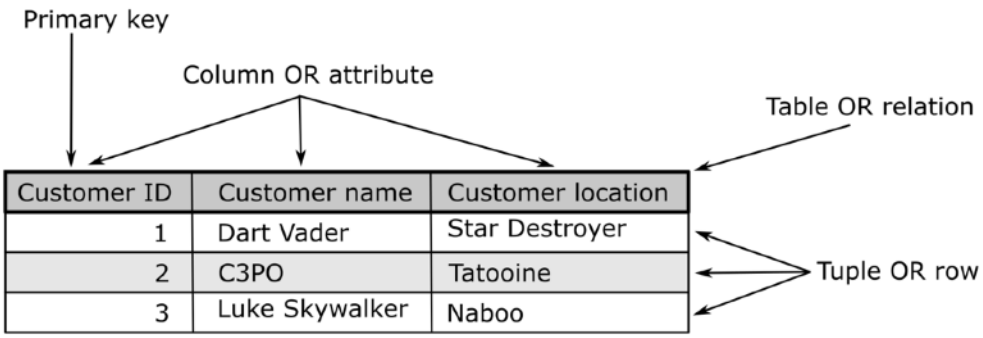

FIGURE 1.1 Database model used by SQL

Database model used by SQL (see figure 1.1) assume that data is represented in terms of *tuples* grouped into *relations*. We can think about relations as spreadsheets table while tuples as a rows of this table.

Every tuple in turn consist of one or more *attributes* which resembles spreadsheet's columns. Main properties of this model are:

1. You may have any number of tables in our database.
2. In every table you can have any but precisely defined number of columns.
3. Using keys (which are unique identifier for every row within a given table) you can define relationships between tables.

Having some data organized this way, you can do some basic operation on them. Taking for example the data from the `customer` and `order` table:

`customer` table

id	name
1	Alice
2	Betty
3	Carolina
4	Diana
5	Emma
6	Fiona

`order` table

id	customerID	total
1	1	100.00
2	1	33.00
3	4	5.00
4	2	250.00
5	3	64.00

6	6	172.00

(notice that the `customerID` column in the `order` table refers to the `id` column in the `customer` table) you can:

- Retrieve data stored in database:

Query

```
SELECT name
FROM customer;
```

Result

```
Alice
Betty
Carolina
Diana
Emma
Fiona
```

- Retrieve data stored in database imposing some conditions:

Query

```
SELECT customerID
FROM order
WHERE total > 100;
```

Result

```
2
6
```

- Retrieve **joined** data stored in database imposing some conditions:

Query

```
SELECT name
FROM customer, order
WHERE total > 100
      AND customer.id = order.customerID
```

or

```
SELECT name
FROM customer
INNER JOIN order ON customer.id = order.customerID
WHERE total > 100;
```

Please note `JOIN` existence in the above query. In this case, to get data you want, you *have to* combine (join) data coming from different tables. **Joins are typical and indispensable nature of relational model.** You can say it is almost impossible to get useful data making queries with no joins.

Result

```
Betty
Fiona
```

- Insert data into database:

Query

```
INSERT INTO customer(id, name)
VALUES (7, 'Helen');
```

Result

customer table

id	name
1	Alice
2	Betty
3	Carolina
4	Diana
5	Emma
6	Fiona
7	Helen

- Update existing data:

Query

```
UPDATE customer
SET name = 'Grace'
WHERE id = 7;
```

Result

customer table

id	name
1	Alice
2	Betty
3	Carolina
4	Diana
5	Emma
6	Fiona
7	Grace

- Delete existing data:

Query

```
DELETE FROM customer
WHERE name = 'Grace';
```

Result

customer table

id	name
1	Alice
2	Betty
3	Carolina
4	Diana
5	Emma
6	Fiona

An inseparable part of this system is a set of rules known as the **normal forms**. What is interesting, relational model defines few levels of conformance specifying how data should be organized into tables. The main goal of all normal forms is to force user to keep data in a form limiting data redundancy and helping to avoid troubles while data is inserted, updated or deleted. Normalization guidelines are cumulative. For a database to be in 2NF (second normal form), it must first fulfill all the criteria of a 1NF (first normal form) database; to be in 3NF, it must be in 2NF, etc.

The main goal of all normal forms is to force user to keep data in a form limiting data redundancy and helping to avoid troubles while data is inserted, updated or deleted.

The way of data organization imposed by normal forms, influence the way we think about real objects. A relational database is like a garage that forces you to take your car apart and store the pieces in little drawers, every time you drive into it.

A relational database is like a garage that forces you to take your car apart and store the pieces in little drawers, every time you drive into it.

NOTE

Transaction model and ACID

The relational model does not itself define the way in which the database handles concurrent data change requests named **transactions**. To ensure consistency and integrity of data an **ACID transaction model** is used and became de facto the standard for all serious relational database implementations. An ACID transaction should be

- ***A**tomic*. The transaction can not be divided – either all the statements in the transaction are applied to the database or none are.
- ***C**onsistent*. The database remains in a consistent state before and after transaction execution.

- ***I**solated*. While multiple transactions can be executed by one or more users simultaneously, one transaction should not see the effects of other in-progress transactions.

- ***D**urable*. Once a transaction is saved (committed) to the database, its changes are expected to persist even if there is a failure of operating system or hardware.

From one side ACID along with relations is the source of the power of relational databases. On the other hand this is a source of serious and very difficult to overcome problems.

Mismatch between the relational model (the way you store data) and real object propagated to object-oriented programming languages (the way you use data) has serious consequences. Whenever an object is stored into or retrieved from a relational database, multiple SQL operations are required to convert from the object oriented representation to the relational representation. This is cumbersome for the programmer and can lead to performance or reliability issues.

This led to first attempt to replace relational databases with something else. This is why in the late 1980s and early 1990s object-oriented DBMSs were developed. The idea was brilliant: instead of the endless process of joining and disjoining objects, it is much better to avoid this unproductive job and save whole objects as they are used in application. These object-oriented DBMSs (OODBMS), however, never saw wide-spread market adoption. The main reasons for this state of affairs was that they lacked a standard, universal, interface like SQL. People are so used to using SQL, that other interfaces seemed to be awkward or useless. It is true even now – every modern database technology offers SQL-like interface even if internally it is not relational system. OODBMS offered the advantages to the application developer, but forgot about those who wished to consume information for business

purposes. This could be the reason that OODBMS systems had completely failed to gain market share. You have to remember that databases don't exist to make programmers life simpler. They represent significant assets that must be accessible to those who want to mine the information for decision making and business intelligence. By implementing a data model that was only understandable and could be used by the programmer, and ignoring the business user of a usable SQL interface, the OODBMS failed to gain support outside programmers world.

> *Databases don't exist to make programmers life simpler. They represent significant assets that must be accessible to those who want to mine the information for decision making and business intelligence.*

This way SQL databases have defended their dominant position in the market. Relational model is undoubtedly characterized by the following set of positive features.

- ACID transactions at the database level makes development and usage easier.
- Most SQL code is portable to other SQL databases.
- Typed columns and constraints helps validate data before it's added to the database which increase consistency of the data stored in database.
- Build in mechanism like views or roles prevents data to be changed or viewed by unauthorized users.

To be honest one cannot forget about negative side of relational model. Most of the following features weren't at that time a problem but they exist.

- ACID transactions may block system for a short time which may be unacceptable. Remember about joins: if you want get some data you have to join multiple tables which means you can't do any inserts or updates (required these tables) at the same time to prevent data inconsistency.
- The object-relational mapping is possible but can be complex and add one more intermediate layer.
- RDBMSs don’t scale out. See next chapter – in short, it is very difficult to increase "infinitely" processing power of relational system. Sharding over many servers can be done but requires new or tuned application code and will be operationally inefficient.
- It is difficult to store high-variability data in tables.
- It is difficult to store data in real time and make real time processing.

Despite its drawbacks relational databases, were (and still are) very well established in IT and seems to be perfectly crafted to all needs and nothing announced that a new era was coming. Unexpectedly in the 2000s non-relational databases became popular. We refer them as **NoSQL** because at the very beginning they used query languages (or methods) different than well known SQL. Today you can observe a tendency to introduce SQL-like querying languages in case of many NoSQL based systems.

Summary

Relational model is undoubtedly characterized by the following set of positive features:

- ACID transactions at the database level makes development and usage easier.

- Most SQL code is portable to other SQL databases.
- Typed columns and constraints helps validate data before it's added to the database which increase consistency of the data stored in database.
- Build in mechanism like views or roles prevents data to be changed or viewed by unauthorized users.

To be honest one cannot forget about negative side of relational model:

- ACID transactions may block system for a short time which may be unacceptable.
- The object-relational mapping is possible but can be complex and add one more intermediate layer.
- RDBMSs don't scale out. Sharding over many servers can be done but requires new or tuned application code and will be operationally inefficient.
- It is difficult to store high-variability data in tables.
- It is difficult to store data in real time and make real time processing.
- Too much overhead in using a full- featured DBMS as a "dumb" data store for may web-based application.
- SQL is an overkill for simple look-up queries.
- Consistency and correctness in the price of availability and performance.

SECTION 3

Big Data – big problem with data

Why was NoSQL created? You could say, SQL was getting old. Nothing could be more wrong. Information technology is one of the few areas in which the system components do not change because of their age. The only impulse for change is usefulness. Never age. An unknown factor had to appear, forcing us to abandon previously known technologies.

When data becomes a problem?

To find an answer for this question, let's make a very simple mental experiment performed on N sheets of paper.

Stamping task *Having a stamp, mark every sheet of paper with it*. It is boring but feasible task. What is important, to complete it with less time (increase performance), we may do one of the following actions:

- Increase the speed of stamping a single sheet employing a *mega man* – a man who can do this much faster than all other known man. Despite of having mega man, this increase in speed has its natural physical limitations – *mega man* can't work faster than himselves.

- You can divide the stack into smaller (sub)stacks and assign each smaller stack to different person. *Increasing stamping speed can be achieved by assigning more people to this task*. What's more, despite

the fact that each of the people, compared to the mega man from the first example, will be much less efficient, this task considered as a whole will be solved much faster if you will employ a right number of helpers.

Numbering task *Numbering all N cards with N natural numbers from 1 to N. Numbered pages should be stored as a pile of pages preserving increasing order from the bottom to the top.*
In this case there is no possibility to divide task into smaller (sub)tasks. The only chance to increase performance is to increase processing power of a one single processing unit (employ mega man). But this, as you know, has some unbreakable physical limitations.

Trying to solve both tasks you face a problem of **task parallelization** and **system scaling**. Generally speaking, **scalability** *is the property of a system to handle a growing amount of work by adding resources to the system*. There are two options to scale system:

- either by increasing the power of single execution unit (mega man) – this is called **vertical scaling**;
- or increase the number of execution unit – this is called **horizontal scaling**. Horizontal scaling seems to be more perspective. There is "only" one small but important detail: *your task should be divisible into independent subtasks*. Unfortunately this is not always the case as you have seen in the numbering task. You will say that stamping task is a *parallelizable task* while numbering task is a *non-parallelizable task*.

> *When there is relatively little data, non-scalable (or at most vertically scalable) systems are sufficient.*

Task parallelization is about possibility of division one task into independent subtasks. Scaling system vertically you can effectively solve non-parallelizable tasks, while horizontal scaling works for

parallellizable tasks. As it is not possible to scale system vertically indefinitely, horizontal scaling is much more desired feature. Whenever in this book I use a term *scalable task* I mean a task you can effectively solve scaling your system horizontally thanks to the ability to divide that task into parallelly processable subtasks. When does a task become a problem for us? Notice that need for scaling occurs only in case of voluminous set of data (we say often: *large data* or *big data*). For small *N* there should be no problems to complete both stamping task and numbering task by only one man in reasonable time. When there is a relatively little data, non-scalable (or at most vertically scalable) systems are sufficient. So data may be a problem when there is *a lot* of it. But what does *a lot* mean? How is it today? Do we have a lot of data today?

> *Data may be a problem when there is a lot of it.*

Volume

It is assumed that today organizations and users world-wide create over 2.5 EiB (2^{60}, more than 10^{18} , 5 billions DVD) of data a day. As a point of comparison, the Library of Congress currently holds more than 300 TiB (2^{40} , more than 10^{12} , 65000 DVD) of data. Almost any aspect of our live is, or can be, a source of data (another question is, if we really need them?) and they can be generated by human, machines as well as environment. The most common are (H is used to denote human, M – machines, E – environment):

- (H) social media, such as Facebook and Twitter,
- (H) scientific and research experiments, such as physical simulation,
- (H) online transactions, such as point-of-sale and banking,

- (M) sensors, such as GPS sensors, RFIDs, smart meters and telematics,
- (M) any working device we monitor for our safety, such as planes or cars,
- (E) weather conditions, cosmic radiation.

NOTE

How much data?

Every minute (in 2018):

- 4.166.667 likes made by Facebook users,
- 347.222 tweets on Tweeter,
- 100.040 cals on Skype,
- 77.166 hours of movies from Netflix,
- 694 Uber users take ride,
- 51.000 application are downloaded from AppStore. [HMD]

Today, we collect all data we can get regardless whether we really need them or not. The term **data lake** was created for this type of data "collection". They acts as a lake with lots of different things inside, not always easily accessible and visible. We do this (collect data) not necessarily because we need it but because we can. *Maybe one day we will need it* – we think.

Over time, large amounts of data hide what you previously stored.

Over time, large amounts of data can hide what you previously stored. This is like my basement: I know that some elements I need are somewhere in it. Unfortunately, because there are so many elements, it's cheaper, easier and faster to buy new one than try to find them in my basement element lake.

What is worse, **volume** is not the only problem with data. Along with that, there are more factors playing also very important role: **velocity** and **variety**. All of them together constitutes something we call nowadays **big data**.

NOTE

More Vs

According to some sources, today big data features may be summed up in more than three different Vs. Today for a dataset to be considered as a big Big Data, it must possess characteristics commonly referred to as the **Five Vs**: *volume*, *velocity*, *variety*, *veracity* and *value*. The more we use Big Data, the greater impact on our live it has. Going further, you might consider up to 10 factors: among *volume*, *velocity*, *variety*, *veracity* and *value* defined so far we should also consider: *validity*, *volatility*, *variability*, *viscosity* and *vulnerability*. There are also some attempts to distinguish more v's: 17 or even 51. [BDC]

This growth of data size is outpacing the growth of storage capacity, leading to the emergence of information management systems where *data is stored in a distributed way, but accessed and analyzed as if it resides on a single machine*. Unfortunately, an increase in processing

power does not directly translate to faster data access, triggering a rethink on existing database systems.

> *SQL solutions were not able to cope with the current needs of working with large data sets in real time.*

This justifies why we've heard about NoSQL so "recently" (relational (SQL) databases are known from the 1970s, nonrelational databases (NoSQL) from the 2000s). Until the data explosion, the existing solutions (SQL) proved to be sufficient. Now they are not able to cope with the current needs of working with large data sets in real time.

CHAPTER 5

Key-value stores

General overview of key-value stores

- Basic ideas and features
- Working with Riak

Basic ideas

Array is one of the first (if not the first) data structures taught to computer science students. After numbers, like integers and floats, characters, and boolean variables the array is the simplest, the most universal and present (at least in some equivalent form) in almost any programming language. In its most basic form, we can say that an array is an ordered collection of *values* (objects) of the same type. Each value in the array is associated with an integer number called *index*. Indexes are taken from given interval with no gaps – each number from this interval corresponds to exactly one index which in turn corresponds to exactly one value.

Although arrays are good, they are no perfect. Mostly because of restriction to using integers as indexes (very often from language specific range – range starting always from 0 (zero) is the best known) and limiting values to the same type. Generalisation of an array is an *associative array* where it is allowed to use arbitrary type for identifiers and array's values. Depending on programming language, associative arrays are recognisable by a number of different names, including *dictionary*, *map*, *hash map*, *hash table*, and *symbol table*. Regardless of the name, the key feature is ability to store any value pointed by almost any other value being the key.

In its simplest form, we can say that key-value store is a dictionary. We will use a term *dictionary,* instead of any other, because in my opinion it best describes all the related concepts. A book named a dictionary has a list of words (keys) and each word (key) has one definitions. Definition

may be simple or compound consisting of many sub-definitions depending on its complexity. The paper based dictionary is a simple (analog, non-computer) key-value store where word entries represent keys and definitions (sometimes very elaborated) represent values. If only dictionary entries (words) are sorted alphabetically, definition retrieval is fast. There is no need to scan the entire dictionary item by item, key by key to find what we are looking for. On the other hand there is no option to find something by scanning its contents (definitions) – we can do this, but it would take too much time.

> *A key-value store is a simple database that when presented with a simple string (the key) returns an arbitrary large BLOB (value).*

Like the dictionary, a key-value store is also indexed by the key. The key points directly to the value, which we can get without need for any search, regardless of the number of items in our store; an access is almost instantaneous. A key-value store is a simple database that when presented with a simple string (the key) returns an arbitrary large BLOB value. Sometimes the key also may be a BLOB. Because database is in itself a very simple, also very simply is its query language. Being more precisely, there is no query language because set of operation (queries) is limited to add and remove key-value pairs into/from a database.

NOTE

BLOB

BLOB (*binary large object*, or sometimes: *basic large object*) is a sequence of bytes stored as a single entity. Most basic examples of BLOBs are images, audio or other multimedia objects, but also binary executable code. In general, BLOB is anything stored in computer system, which is not represented as a basic or „standard" data type

such as number, character, string, timestamp or UUID. In case of BLOBs we don't care what really BLOB is. We simply treat it as a sequence of bytes.

The story of *blob* term is quite interesting. Blobs were originally just big amorphous chunks of data invented by Jim Starkey at DEC, who describes them as "the thing that ate Cincinnati, Cleveland, or whatever" referring to *The Blob* movie (1958). Later, Terry McKiever, a marketing person for Apollo, felt that it needed to be an acronym and invented the backronym *Basic Large Object*. Then Informix invented an alternative backronym, *Binary Large Object*. [BLOB]

Key-value stores vs. relational databases

Simplicity is a key word associated with key-value databases where everything is simple. Unlike in relational databases, there are no tables, so there are no features associated with tables, such as columns and constraints on columns. If there are no tables, there is no need for joins. In consequence foreign keys do not exists and so key-value databases do not support a rich query language such as SQL. Saying the truth, their query language is very primitive.

The only extra feature supported by some key-value databases are *buckets*, or *collections*. We use them for creating separate *namespaces* within a database. Keys from one namespace do not collides with keys from other so we can use the same keys in more than one namespace.

> *Contrary to relational database where meaningless keys are used, the keys in key-value databases are meaningful.*

Contrary to relational database where meaningless keys are used, the keys in key-value databases are meaningful – see Key is the key section for more details.

While in relational database we avoid duplicating data, in key-value (in NoSQL in general) databases it is a common practice.

Limitations of key-value databases

There are a lot of key-value databases. Bellow there are some general limitations which are in line with the general idea of this database type. For a given database some of them may not be true.

- The only way to look up values is by key. If you do not think over the strategy of creating key names well, you may face the problem of inability to fetch the data stored in the database. This is really very important so a separate section Key is the key is devoted to this topic.

- Range queries are not supported out of the box. Because keys are of the form of any arbitrary string or more generally BLOB there is no method to define range. It would be possible with for example regular expressions executed on the database side but as for now no key-value database support this feature. Instead, based on proper key naming strategy, on the application side, you can generate a sequence of keys and use them to retrieve values.

- Queries from one key-value database may not be portable to the other. This is generally true in NoSQL systems as there is no standard query language comparable to SQL existing for relational databases.

Essential features of key-value databases

Despite a huge variety of key-value databases there exists a set of features common for all of them:

- simplicity,
- speed,
- scalability.

Simplicity

As it was stated in a previous section, simplicity is a key word describing key-value databases.

Ask yourself, how many times do you really need relational database? Is it really indispensable when developing simple application with persons (company staff) and skills they have? We spend our time to develop relational database with all of its requirements (do you remember about normal forms?). For what? Finally our application retrieves aggregations from a database to display person by person with their skills on a simple web page.

If you follow one of the agile method, you need a flexible tool to rapidly test your changing ideas. With key-values if you want to track additional

attributes or remove some of them after your program is ready, you can simply add / change code to your program to take care of those attributes. There is no need to change database code to tell the database about the new attributes set.

> *If we follow one of the agile method we need a flexible tool to rapidly test our changing ideas.*

In key-value databases, you work with a very simple *data model* which resembles dictionary. The syntax for manipulating data is simple. Regardless of the type of an operation, you specify a namespace, and a key to indicate you want to perform an action on a key-value pair. Type of it depends on your call. There are three operations performed on a key-value store: `put`, `get`, and `delete`.

- `put` adds a new key-value pair to the table or updates a value if this key is already present. What should be stressed here, an *update* means *replace* existing value with a new one. It's obvious when value is a „real" BLOB like a media file. If our BLOB is a JSON, you might tend to think about each of its attribute-value pairs separately. The problem is that your's BLOB interval structure is not visible to database and even if you want to change one of several hundred values you have to resend almost the same JSON.

- `get` returns the value for a given key.

- `delete` removes a key and its value from the table.

Other feature which simplifies programmers fate is typelessnes which is a consequence of the fact that values are, generally speaking, BLOBs so you can put everything you want. It's up to the application to determine what type of data is being used, such as an integer, string, JSON, XML file, or even binary data like image. This feature is especially useful when the data type changes or you need to support two or more data types for the same attribute. Imagine for example a network of sensors

where some of them returns integer value, other logical state or enumeration or even a string. There is no problem with this in key-value database.

Speed

In this case speed is a consequence of simplicity. There is no need for complicated query resolving logic. Every query directly specify the key and always it is only one key. The only job is to find the value corresponding to it. Supported with internal design features optimising performance, key-value databases delivers high-throughput for applications with data-intensive operations.

Scalability

As I mentioned in NoSQL: motivations section, scalability is another most wanted feature all databases wants to have. Working with key-value databases you have no relational dependencies and all write and read requests are independent and this seems to be a perfect state for scaling.

Key is the key

Key is the key in effective key-value databases usage. It is so important, that it is worth to devote a separate part dedicated only to this topic.

As already stated, keys are used to index, or you can say: uniquely identify, a value within a namespace in a key-value database. This makes keys sound pretty simple, and sometimes they look so. On the other hand, keys are the only method you can get the value you are looking for. In key-value databases, generally speaking, there is no method to scan or search values so the right key naming strategy is crucial. I think that term *strategy* in this context is better than any other because correct keys names allow you to win information war and is factor which makes some application much faster and responsive than others.

Although it's not strict rule, while working with relational databases counters or sequences are very often used to generate keys. Working with numbers is the easiest way to ensure that every new call for a new key returns a value (number in this case) which is unique and unused so far. That's why application designers use these (numbers) somehow routinely to make keys (primary keys) for rows of data stored in a table. In relational databases keys are used to connect (join) data stored in one table with others tables' data. Storing a primary key to a row in another table is known as a *foreign key*. This is the main purpose, and because of the way relational databases work, it makes sense (sometimes it is considered as a good practice) to have such a meaningless keys in this case.

Keys in relational databases are used to join data. Their numeral values are the easiest way to guarantee their uniqueness.

In key-value databases the rules are different. Although you may think about key-value databases as built on the basis of very simple table with many rows and just two columns: first for the key and second for the value, they do not have a built-in table structure. If there are no tables, there are no rows and columns so the question arise: *how to "join", combine or somehow collect all information related to a given object?* The answer is: u*se right aggregation and key names*.

Let's go back to our Star Wars based example from Example subsection in Relational model: normal forms. In third normal form you have three distinct tables. Now imagine that you want to keep customer data shown below in key-value database:

customer table

number	name	location
10	Dart Vader	Star Destroyer
20	Luke Skywalker	Naboo
30	C3PO	Tatooine

First attempt may look like this:

```
CustomerDetails[10] = 'Dart Vader'
```

Drawbacks of this are obvious. First, you have no information what customer's detail you have under index 10: her/his name or maybe age or maybe something totally different. Second: how you can store other informations related to this customer?

Usage another namespace might be a solution:

```
CustomerName[10] = 'Dart Vader'
CustomerLocation[10] = 'Star Destroyer'
```

but this approach leads to potentially huge namespace set which is not easy to maintain and use. Some of us can live with this. But will this still be working if you want to store information about an invoice details?

invoice details table

invoice number	invoice item	item name	item quantity	item price
1	1	lightsaber	1	100
1	2	black cloak	2	50
1	3	air filter	10	2
2	1	battery	1	25
3	1	lightsaber	5	75
3	2	belt	1	5
4	1	wires	1	10

Of course you can use all of the data as values and put them into your key-value database, for example in the following JSON document:

```
InvoiceDetails[1] =
{
  "Invoice details" : [
    {"Item name" : "lightsaber",
    "Item quantity" : 1,
    "Item price" : 100},

    {"Item name" : "black cloak",
    "Item quantity" : 2,
    "Item price" : 50},

    {"Item name" : "air filter",
    "Item quantity" : 10,
```

```
    "Item price" : 2}
  ]
}
```

or even you can write it as

```
InvoiceDetails[1] =
{
  "Invoice number" : 1,
  "Invoice details" : [
    {"Item name" : "lightsaber",
    "Item quantity" : 1,
    "Item price" : 100},

    {"Item name" : "black cloak",
    "Item quantity" : 2,
    "Item price" : 50},

    {"Item name" : "air filter",
    "Item quantity" : 10,
    "Item price" : 2}
  ],

  "Customer details" : {
    "Customer name" : "Dart Vader"
    "Customer location" : "Star Destroyer"
  }
}
```

Hmm... It's not bad but not as good as it may be. Notice that you have to parse this JSON every time to get even very basic piece of information, like *customer name*.

Avoid to use many namespaces. Remember: *key is the key*

I hope that with this examples I was able to convince you that something should be changed in our approach. Because key-values databases are simple, you have not too many possibilities. As mentioned

earlier, you can construct meaningful names that entail needed information. For example:

```
Shop[Customer:10:name] = 'Dart Vader'
Shop[Customer:10:location] = 'Star Destroyer'
```

Do not follow relational pattern

Never ever copy relational data model. With this example you face another important issue related with keys. Let's say that now you want to put information related with invoice details. You can do this in many different ways. Following relational pattern for C3PO you have:

```
Shop[customer:30:invoice:2] = ...
Shop[customer:30:invoice:4] = ...
```

And this is what you should avoid. In this particular case you obtain discontinuous invoice range which makes queries almost impossible. If range is discontinuous, every time you obtain nonexistent key you have no idea if there are no more items or it is only a gap and you should continue increasing counter to get all remaining data. If range is continuous, you can start from 1 and increase counter till you obtain nonexistent key which is interpreted as there are no more items. More adequately would be enumerating invoices independently per customer:

```
Shop[customer:30:invoice:1] = ...
Shop[customer:30:invoice:2] = ...
```

so you could iterate over all invoices related with customer identified by the number 30.

Mind aggregation you expect to use

On the other hand, if you suppose that you will use the data most often for order processing, another key naming convention might by more relevant

```
Shop[invoice:1:customerDetails] = ...
Shop[invoice:1:details] = ...
Shop[invoice:2:customerDetails] = ...
Shop[invoice:2:details] = ...
Shop[invoice:3:customerDetails] = ...
Shop[invoice:3:details] = ...
Shop[invoice:4:customerDetails] = ...
Shop[invoice:4:details] = ...
```

As you can see in this case, following relational pattern in numbering invoices sounds good.

Again this is a sign for us that correct key naming is a strategy and should be chosen very carefully with respect to the aggregation boundaries we have discussed in previous part (NoSQL chapter, Motivations section, Aggregations subsection) and application (developers) future needs.

Mind range queries you expect to use

Dealing with ranges of values is another thing which should be considered. If you expect you will need in the future process our invoices by date or date range, following naming convention:

```
Shop[invoice:20171009:1:customerDetails] = ...
Shop[invoice:20171009:1:details] = ...
Shop[invoice:20171010:1:customerDetails] = ...
Shop[invoice:20171010:1:details] = ...
Shop[invoice:20171010:2:customerDetails] = ...
Shop[invoice:20171010:2:details] = ...
Shop[invoice:20171013:1:customerDetails] = ...
Shop[invoice:20171013:1:details] = ...
```

would be better than:

```
Shop[invoice:1:customerDetails] = ...
Shop[invoice:1:details] = ...
Shop[invoice:1:date] = "20171009"
```

```
Shop[invoice:2:customerDetails] = ...
Shop[invoice:2:details] = ...
Shop[invoice:2:date] = "20171010"
Shop[invoice:3:customerDetails] = ...
Shop[invoice:3:details] = ...
Shop[invoice:3:date] = "20171010"
Shop[invoice:4:customerDetails] = ...
Shop[invoice:4:details] = ...
Shop[invoice:4:date] = "20171013"
```

With a pattern of the following fom:

```
invoice:<DATE>:<INVOICE_NUMBER>:[details | customerDetails]
```

you can generate a sequence of keys for a given date `<DATE>` starting from invoice number `<INVOICE_NUMBER>` of value `1` and continue as long as for some number *n* the key will be invalid.

If you have developed relational data models, you might have noticed parallels between the key-naming convention we have just presented and *tables*, *columns* names and *primary keys*. Concatenating a *table name* with a *primary key*, and a *column name* to get:

```
Customer:10:name
```

key is equivalent to a relational *table* `customer`, with a *column* called `name`, and a `row` identified by the *primary key* `ID` of `10`. It's worth to stress that you should avoid key naming conventions which mimics relational databases schema. If it is the case it's worth to consider database replacement. Using key-value database as relational do not seems to be reasonable but maybe in some specific cases (for example when the total number of tables is low) could be effective.

Values

Working with key-value database you have to carefully select key naming strategy. Similarly **you have to balance aggregation boundaries for values to make writes and reads more efficient as well as reduce latency**.

Bellow there are some strategies. If they are good or bad depends on you – we will try to highlight their pros and cons.

Values which are big aggregates
The following aggregate may be an example of this strategy

```
{
  "Invoice number": 1,
  "Invoice details": [
    {"Item name": "lightsaber",
    "Item quantity": 1,
    "Item price": 100},

    {"Item name": "black cloak",
    "Item quantity": 2,
    "Item price": 50},

    {"Item name": "air filter",
    "Item quantity": 10,
    "Item price": 2}
    ],
```

```
    "Customer details": {
      "Customer name": "Dart Vader",
     "Customer location": "Star Destroyer"
  }
}
```

The advantage of using a structure such as this is that much of the information about invoice is available with a single key lookup. By storing all the informations together, you might reduce the number of disk seeks that must be performed to read all the needed data.

On the other hand when an additional item is added or even existing one is edited (changed), the whole structure has to be written to a disc. As structure grows in size, the time required to read and write the data can increase.

Another drawback is that you have to read such a big structure even if you need only a small piece of information – this way you waste time for reading and memory for storing it.

Keep together values commonly used
Another approach is to store only commonly used values together. For example under the separate keys invoice and customer details. Now you have more seeks and more reads operations but you spend less time reading particular data.

Small values supports cache
Assume that your database keeps data you have read before in memory buffer (cache) so in case you want them again, database could serve them much faster than from disc. Of course the size of cache is limited so you may be able to store, say 2 big structures or 10 smaller. Now if you want to print third customer name, you have to remove one big structure and replace it with a new one or, in the second case, remove one small structure (for example with second customer name) and replace it with a third customer name. In the second case if you need second customer items there is a high chance that all of them are still in

the memory while in the first case the probability that you don't have to reloaded them is much lower.

SECTION 6

Summary

- No tables, so there are no features associated with tables, such as columns types or constraints on columns.
- There is no tables so there is no need for joins. In consequence foreign keys do not exists.
- Do not support a rich query language such as SQL. Saying the truth, query language is very primitive and limited to simple select, insert and delete equivalent commands.
- Contrary to relational databases where meaningless keys are used, the keys in key-value databases are meaningful and play crucial role.
- Although key-value databases don't have any structure we have to very carefully balance aggregation boundaries for values to make writes and reads more efficient as well as reduce latency.

SECTION 7

Working with Riak

Creating object

To facilitate key handling, objects in Riak are collected in buckets. Buckets are essentially a flat namespace and may also be seen as a common prefix for a set of keys or a table name – if you want to have some reference to relational model.The basic form of writes (object creation) is (in this chapter we will use HTTP protocol, so if you need some explanations how it works, please read Appendix C: HTTP):

```
PUT /types/<type>/buckets/<bucket>/keys/<key>
```

There is no need to intentionally create buckets. They pop into existence when keys are added to them, and disappear when all keys have been removed from them. If you don't specify a bucket's type, the type default will be applied. If we're using HTTP, POST can be used instead of PUT. The only difference between POST and PUT is that you should POST in cases where you want Riak to auto-generate a key.

Here is an example of storing an object (short text: `Test string 01`) under the key `key001` in the bucket `bucket001`, which bears the type `type001`:

```
nosql@nosql:~$ curl -X PUT \
>      -H "Content-Type: text/plain" \
>      -d "Test string 01" \
>      http://localhost:8098/types/type001/buckets/bucket001/
keys/key001
```

Riak replays with:

```
Unknown bucket type: type001
```

Notice that although you don't have to create bucket in advance, you have to create and activate a type you want to use – the above command will only work if the `type001` bucket type has been created and activated. The step below allows to create the bucket type:

```
nosql@nosql:~$ sudo riak-admin bucket-type create type001
'{"props":{}}'
type001 created
```

and than activate it:

```
nosql@nosql:~$ sudo riak-admin bucket-type activate type001
type001 has been activated
```

Now you can again try to add an object:

```
nosql@nosql:~$ curl -X PUT \
>      -H "Content-Type: text/plain" \
>      -d "Test string 01" \
>      http://localhost:8098/types/type001/buckets/bucket001/
keys/key001
```

Hmmm... Riak replays with no errors and no any other messages... It's time to get something from your database.

Reading object

You can think of writes in Riak as analogous to HTTP PUT (POST) requests. Similarly, reads in Riak correspond to HTTP GET requests. You specify a bucket type, bucket, and key, and Riak either returns the object that's stored there–including its siblings (more on that later)–or it returns `not found` (the equivalent of an HTTP `404 Object Not Found`).

Here is the basic command form for retrieving a specific value under a given key from a bucket:

```
GET /types/<type>/buckets/<bucket>/keys/<key>
```

and how you can use it:

```
nosql@nosql:~$ curl http://localhost:8098/types/typ
e001/buckets/bucket001/keys/key001
Test string 01
```

If there's no object stored in the location where you attempt a read, you will get the `not found` response:

```
nosql@nosql:~$ curl http://localhost:8098/types/typ
e001/buckets/bucket001/keys/key002
not found
```

Updating objects

If an object already exists under a certain key and you want to write a new object to that key, Riak needs to know what to do, especially if multiple writes are happening at the same time. Which of the objects being written should be deemed correct? These question can arise quite frequently in distributed, eventually consistent systems.

Riak decides which object to choose in case of conflict using *causal context*. These objects track the causal history of objects. They are attached to all Riak objects as metadata, and they are not readable by humans. Using causal context in an update would involve the following steps:

- Fetch the object.
- Modify the object's value (without modifying the fetched context object).
- Write the new object to Riak.

The most important thing to bear in mind when updating objects is this: **you should always read an object prior to updating it unless you are certain that no object is stored there**. If you are storing sensor data in Riak and using timestamps as keys, for example, then you can be sure that keys are not repeated. In that case, making writes to Riak without first reading the object is fine. If you are not certain, however, then it is recommend always reading the object first.

When using curl, the context object is attached to the `X-Riak-Vclock` header:

```
nosql@nosql:~$ curl -i http://localhost:8098/types/
type001/buckets/bucket001/keys/key001
HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgzGDKBVI8ypz/frbujHgFZDNlMCU
y5rEy9LiyXOXLAgA=
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.10.9 (cafe not found)
Link: </buckets/bucket001>; rel="up"
Last-Modified: Thu, 06 Sep 2018 10:26:38 GMT
ETag: "1KTnww1Q52P9d666EtbP41"
Date: Thu, 06 Sep 2018 10:28:56 GMT
Content-Type: text/plain
Content-Length: 14
```

As you can see, for HTTP, you can get the header information shown before the data by using `-i` /`--include` option. curl understands also the `-D` / `--dump-header` option when getting files from both FTP and HTTP, and it will then store the headers in the specified file:

```
nosql@nosql:~$ curl -D headers.txt http://localhost:8098/ty
pes/type001/buckets/bucket001/keys/key001
Test string 01
nosql@nosql:~$ cat headers.txt
HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgzGDKBVI8ypz/frbujHgFZDNlMCUy5rEy9Liy
XOXLAgA=
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.10.9 (cafe not found)
Link: </buckets/bucket001>; rel="up"
Last-Modified: Thu, 06 Sep 2018 10:26:38 GMT
ETag: "1KTnww1Q52P9d666EtbP41"
Date: Thu, 06 Sep 2018 10:29:30 GMT
Content-Type: text/plain
Content-Length: 14
```

When performing a write to the same key, that same header needs to accompany the write for Riak to be able to use the context object. Before

you will do this, let's check what will happen if you will ignore `X-Riak-Vclock:`

```
nosql@nosql:~$ curl -X PUT \
>      -H "Content-Type: text/plain" -d "Test string 01_updat
ed" \
>      http://localhost:8098/types/type001/buckets/bucket001/
keys/key001
```

```
nosql@nosql:~$ curl -D headers.txt http://localhost:8098/ty
pes/type001/buckets/bucket001/keys/key001
Siblings:
1KTnww1Q52P9d666EtbP41
38C20HjUtlR8syP3CMGlVW
```

Something goes wrong – you will be back to this case a little bit later. Let's create another object na update it according to the above steps (using `X-Riak-Vclock` header):

```
nosql@nosql:~$ curl -X PUT -H "Content-Type: text/plain" -d
"Test string 02" http://localhost:8098/types/type001/bucket
s/bucket001/keys/key002
```

```
nosql@nosql:~$ curl -D headers.txt http://localhost:8098/ty
pes/type001/buckets/bucket001/keys/key002
Test string 02
```

```
nosql@nosql:~$ cat headers.txt
HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgzGDKBVI8ypz/frZuctCDCCUy5rE
yzN/McpUvCwA=
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.10.9 (cafe not found)
```

```
Link: </buckets/bucket001>; rel="up"
Last-Modified: Thu, 06 Sep 2018 10:30:30 GMT
ETag: "4j7mSGlt44VnZkh8eaCltP"
Date: Thu, 06 Sep 2018 10:30:55 GMT
Content-Type: text/plain
Content-Length: 14
```

Having a new object (string `Test string 02`) you can try to modify it:

```
nosql@nosql:~$ curl -X PUT \
>      -H "Content-Type: text/plain" \
>      -H "X-Riak-Vclock: a85hYGBgzGDKBVI8ypz/frZuctCDCCUy5rE
yzN/McpUvCwA=" \
>      -d "Test string 02 udated" \
>      http://localhost:8098/types/type001/buckets/bucket001/
keys/key002
```

```
nosql@nosql:~$ curl -D headers.txt http://localhost:8098/ty
pes/type001/buckets/bucket001/keys/key002
Test string 02 updated
```

```
nosql@nosql:~$ cat headers.txt
HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgzGDKBVI8ypz/frZuctCDCCUy5bE
y8G5lucqXBQA=
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.10.9 (cafe not found)
Link: </buckets/bucket001>; rel="up"
Last-Modified: Thu, 06 Sep 2018 10:31:55 GMT
ETag: "5RUinoCMbFaMyGdfofm5bZ"
Date: Thu, 06 Sep 2018 10:32:00 GMT
Content-Type: text/plain
Content-Length: 22
```

Notice that after update our X-Riak-Vclock has been changed from:

```
a85hYGBgzGDKBVI8ypz/frZuctCDCCUy5rEyzN/McpUvCwA=
```

to:

```
a85hYGBgzGDKBVI8ypz/frZuctCDCCUy5bEy8G5lucqXBQA=
```

Siblings and conflict resolution

A sibling is created when Riak is unable to resolve the canonical version of an object being stored, i.e. when Riak is presented with multiple possible values for an object and can't figure out which one is most causally recent. In our case you have such a conflict: there are two objects (string `Test string 01` and `Test string 01_updated`) under the same key `key001` in the bucket `bucket001`, which bears the type `type001`:

```
nosql@nosql:~$ nosql@riak:~$ curl -D headers.txt http://loc
alhost:8098/types/type001/buckets/bucket001/keys/key001
Siblings:
1KTnww1Q52P9d666EtbP41
38C20HjUtlR8syP3CMGlVW
```

As you can see, reading an object with sibling values will result in some form of "multiple choices" response (e.g., `300 Multiple Choices` in HTTP). If you are using the HTTP interface and want to view all sibling values, you can attach an `Accept: multipart/mixed` header to your request to get all siblings in one request:

```
nosql@nosql:~$ curl -X GET \
>      -H "Accept: multipart/mixed" \
>      -D headers.txt \
>      http://localhost:8098/types/type001/buckets/bucket001/
keys/key001

--LJ8Xy5UHU8z3L3OtKvMwM7BX9DM
Content-Type: text/plain
Link: </buckets/bucket001>; rel="up"
Etag: 1KTnww1Q52P9d666EtbP41
Last-Modified: Thu, 06 Sep 2018 11:26:38 GMT

Test string 01
--LJ8Xy5UHU8z3L3OtKvMwM7BX9DM
Content-Type: text/plain
Link: </buckets/bucket001>; rel="up"
Etag: 38C20HjUtlR8syP3CMGlVW
Last-Modified: Thu, 06 Sep 2018 11:27:10 GMT

Test string 01 updated
--LJ8Xy5UHU8z3L3OtKvMwM7BX9DM--
```

```
nosql@nosql:~$ cat headers.txt
HTTP/1.1 300 Multiple Choices
X-Riak-Vclock: a85hYGBgzGDKBVI8ypz/frbujHgFZDNnMCUy5bEy/Kth
vcqXBQA=
Vary: Accept, Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.10.9 (cafe not found)
Last-Modified: Thu, 06 Sep 2018 10:27:10 GMT
ETag: "6nZEerBvuMlMlcQITGUY7C"
Date: Thu, 06 Sep 2018 21:28:56 GMT
Content-Type: multipart/mixed; boundary=LJ8Xy5UHU8
z3L3OtKvMwM7BX9DM
Content-Length: 421
```

You can request individual siblings by adding the `vtag` query parameter specifying which sibling to retrieve:

```
nosql@nosql:~$ curl http://localhost:8098/types/type001/buc
kets/bucket001/keys/key001
Siblings:
1KTnww1Q52P9d666EtbP41
38C20HjUtlR8syP3CMGlVW
```

```
nosql@nosql:~$ curl http://localhost:8098/types/type001/buc
kets/bucket001/keys/key001?vtag=1KTnww1Q52P9d666EtbP41
Test string 01
```

```
nosql@nosql:~$ curl http://localhost:8098/types/type001/buc
kets/bucket001/keys/key001?vtag=38C20HjUtlR8syP3CMGlVW
Test string 01 updated
```

To resolve the conflict, store the resolved version with the `X-Riak-Vclock` given in the response – in this case:

```
a85hYGBgzGDKBVI8ypzfrbujHgFZDNnMCUy5bEy/KthvcqXBQA=
```

```
nosql@nosql:~$ curl -X PUT \
>     -H "Content-Type: text/plain" \
>     -H "X-Riak-Vclock: a85hYGBgzGDKBVI8ypz/frbujHgFZDNnMCU
y5bEy/KthvcqXBQA=" \
>     -d "Test string 01 updated" \
>     http://localhost:8098/types/type001/buckets/bucket001/
keys/key001
```

```
nosql@nosql:~$ curl -X GET http://localhost:8098/types/type
001/buckets/bucket001/keys/key001
Test string 01 updated
```

Deleting objects

The delete command looks like this:

```
DELETE /types/<type>/buckets/<bucket>/keys/<key>/
```

and you can use it as it is shown below:

```
nosql@nosql:~$ curl http://localhost:8098/types/type001/buc
kets/bucket001/keys/key002
Test string 02 updated
```

```
nosql@nosql:~$ curl -X DELETE http://localhost:8098/types/t
ype001/buckets/bucket001/keys/key002
```

```
nosql@nosql:~$ curl http://localhost:8098/types/type001/buc
kets/bucket001/keys/key002
not found
```

Bibliography

[AC] Apache Cassandra

1. *What is Data Modeling?*, retrieved 2020-04-24,
   `https://cassandra.apache.org/doc/latest/data_modeling/intro.html#what-is-data-modeling`

2. *Query-driven modeling*, retrieved 2020-04-24,
   `https://cassandra.apache.org/doc/latest/data_modeling/intro.html#query-driven-modeling`

[ACDB] Apache Couch DB

1. *Introduction to Views*, retrieved 2020-05-29,
   `https://docs.couchdb.org/en/stable/ddocs/views/intro.html`

[ATP] Apache Tinker Pop

1. *Apache TinkerPop Documentation*, available from *TinkerPop Compendium* web page (under the *Reference Documentation* link), retrieved 2020-19-18,
   `https://tinkerpop.apache.org/docs/current/`

[BDC] Big Data Characteristics

1. *Big Data characteristics,* retrieved: 2019-02-16,
   `https://fulmanski.pl/tutorials/computer-science/big-data/big-data-concepts-and-terminology/#big_data_characteristics`

2. Arockia Panimalar S., Varnekha Shree S., Veneshia Kathrine, *The 17 V's Of Big Data*, International Research Journal of Engineering and Technology (IRJET), Volume: 04, Issue: 09, Sep-2017, pp. 329-333 retrieved 2020-10-06,
   `https://www.irjet.net/archives/V4/i9/IRJET-V4I957.pdf`

3. Nawsher Khan, Arshi Naim, Mohammad Rashid Hussain, Quadri Noorulhasan Naveed, Naim Ahmad, Shamimul Qamar, *The 51 V's Of Big Data: Survey, Technologies, Characteristics, Opportunities, Issues and Challenges*, COINS '19: Proceedings of the International Conference on Omni-Layer Intelligent SystemsMay 2019, pp. 19-24, retrieved 2020-10-06, https://doi.org/10.1145/3312614.3312623

[BLOB]

1. *The true story of BLOBs*, retrieved: 2020-02-12, https://web.archive.org/web/20110723065224/http://www.cvalde.net/misc/blob_true_history.htm

[CAP] CAP Theorem

1. *CAP theorem*, retrieved 2021-01-26, https://en.wikipedia.org/wiki/CAP_theorem

[CPU]

1. Xeon, https://en.wikipedia.org/wiki/Xeon#3000-series_"Conroe"

2. CPU Mega List, https://www.cpubenchmark.net/CPU_mega_page.html

3. Year on Year Performance, Updated 28th of January 2021, https://www.cpubenchmark.net/year-on-year.html

[CURL] cURL

1. *cURL Command Tutorial with Examples*, retrieved 2020-05-29, https://www.booleanworld.com/curl-command-tutorial-examples/

2. *curl FAQ*, retrieved 2020-05-24, https://curl.haxx.se/docs/faq.html

[CQL] Cypher Query Language

1. *Cypher Query Language*, retrieved 2020-09-11, https://neo4j.com/developer/cypher/

2. *Neo4j Cypher Refcard*, retrieved 2020-09-11 https://neo4j.com/docs/cypher-refcard/current/

3. *Cypher, The Graph Query Language. The Standard Query Language for Graph Database Technology*, retrieved 2020-09-11, `https://neo4j.com/cypher-graph-query-language/`

4. *Neo4j Console*, retrieved 2020-10-08, `http://console.neo4j.org`

5. *Neo4j Sandbox*, retrieved 2020-10-08, `https://neo4j.com/sandbox`

6. *Neo4j Cypher Manual: Syntax: Values and types*, retrieved 2020-10-08, `https://neo4j.com/docs/cypher-manual/current/syntax/values/`

[CRR] Codd's Relational Rules

1. E. F. Codd, *Is Your DBMS Really Relational?*, Computerworld, Oct. 14, 1985.

2. *Codd's Twelve Rules*, retrieved 2020-04-06, `https://computing.derby.ac.uk/c/codds-twelve-rules/`

3. Joe Celko, *Joe Celko's Data and Databases: Concepts in Practice,* The Morgan Kaufmann Series in Data Management Systems, Morgan Kaufmann; 1 edition (August 10, 1999), p. 244-246.

[D] Data

1. *Data,* retrieved: 2020-02-16,
https://en.wikipedia.org/wiki/Data

[DB] Data Base

1. *DBMS Keys: Primary, Candidate, Super, Alternate and Foreign (Example)*, retrieved 2020-04-30,
https://www.guru99.com/dbms-keys.html

2. Markus Winand, *Row Pattern Matching in SQL:2016*, Published on Mar 8, 2017, retrieved 2020-09-03,
https://www.slideshare.net/MarkusWinand/row-pattern-matching-in-sql2016

3. *Learn PostgreSQL Recursive Query By Example*, retrieved 2020-09-04,
https://www.postgresqltutorial.com/postgresql-recursive-query/

4. *Recursive CTE's*, retrieved 2020-09-04,
https://www.essentialsql.com/recursive-ctes-explained/

5. *Recursive SQL Queries with PostgreSQL*, retrieved 2020-09-05,
https://towardsdatascience.com/recursive-sql-queries-with-postgresql-87e2a453f1b

6. *WITH (Common Table Expressions)*, MySQL 8.0 Reference Manual, retrieved 2020-10-19,
https://dev.mysql.com/doc/refman/8.0/en/with.html

7. *To find infinite recursive loop in CTE*, retrieved 2020-10-20,
https://stackoverflow.com/a/31745768

8. Ben Richardson, *Working with XML Data in SQL Server*, October 11, 2019, retrieved 2020-09-03,
https://www.sqlshack.com/working-with-xml-data-in-sql-server/

9. Andrew Eisenberg, Jim Melton, *SQL/XML is Making Good Progress*, ACM SIGMOD Record, Volume 31, Number 2, June 2002, retrieved 2020-09-03, https://sigmodrecord.org/publications/sigmodRecord/0206/standard.pdf

10. J. E. Funderburk, S. Malaika, B. Reinwald, *XML programming with SQL/XML and XQuery*, IBM SYSTEMS JOURNAL, VOL 41, NO 4, 2002, pp. 642-665, retrieved 2020-09-03, http://ict.udlap.mx/people/carlos/is346/files/reinwald.pdf

11. *Temporal database. Example*, retrieved 2020-09-03, https://en.wikipedia.org/wiki/Temporal_database#Example

12. Peter Vanroose, *Temporal Data & Time Travel in PostgreSQL*, FOSDEM 2015 - PGDay 30 January 2015 Marriott Hotel, Brussels, retrieved 2020-09-03, https://wiki.postgresql.org/images/6/64/Fosdem20150130PostgresqlTemporal.pdf

[GatDai] Alan Gates, Daniel Dai, *Programming Pig*, Second edition, O'Reilly Media, 2016.

[GOO] GOogle research

1. *https://research.google.com/archive/gfs.html*, retrieved 2020-05-05, https://research.google.com/archive/gfs.html

2. *https://research.google.com/archive/mapreduce.html*, retrieved 2020-05-05, https://research.google.com/archive/mapreduce.html

3. *https://research.google.com/archive/bigtable.html*, retrieved 2020-05-05, https://research.google.com/archive/bigtable.html

[GRE] Gremlin

1. Kelvin R. Lawrence, *PRACTICAL GREMLIN: An Apache TinkerPop Tutorial*, Version 283-Preview, September 3rd 2020, retrieved 2020-09-18, http://kelvinlawrence.net/book/Gremlin-Graph-Guide.html

2. *Gremlin steps (instruction set)* in *Gremlin (query language)*, retrieved 2020-09-18, https://en.wikipedia.org/wiki/Gremlin_(query_language)

3. *Graph Traversal Steps* in *TinkerPop Documentation* retrieved 2020-10-06, https://tinkerpop.apache.org/docs/current/reference/#graph-traversal-steps

[HA] High Availability

1. *Amazon S3*, retrieved 2019-01-03, https://aws.amazon.com/s3/

2. Marcin Szeliga, *Wysoka dostępność serwerów SQL. Klastry niezawodności infrastruktury*, IT professional, 11/2018, p. 28.

[HMD] How Much Data

1. *Data Never Sleeps 5.0*, retrieved: 2019-02-16, https://www.domo.com/learn/data-never-sleeps-5,

2. *How Much Data Is Generated Every Minute?,* retrieved: 2019-02-16, https://www.socialmediatoday.com/news/how-much-data-is-generated-every-minute-infographic-1/525692/

3. *How Much Data is Created on the Internet Each Day?,* retrieved: 2019-02-16,

https://blog.microfocus.com/how-much-data-is-created-on-the-internet-each-day/

4. *How much data do we create every day?,* retrieved: 2019-02-16, https://techstartups.com/2018/05/21/how-much-data-do-we-create-every-day-infographic/

5. *How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read*, retrieved: 2019-02-16, https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#430a855960ba

6. Dhaval Shah, *Millions of data points flying in tight formation. Part 2: How Big Data could improve commercial aviation safety*, December 19, 2014, retrieved 2021-02-04, https://www.aerospacemanufacturinganddesign.com/article/millions-of-data-points-flying-part2-121914/

7. Stacey Higginbotham, *Sensor Networks Top Social Networks for Big Data*, September 13, 2010, retrieved 2021-02-04, https://gigaom.com/2010/09/13/sensor-networks-top-social-networks-for-big-data-2/

8. Sarah Charley, *10 years of LHC physics, in numbers*, 03/30/20, retrieved 2021-02-04, https://www.symmetrymagazine.org/article/10-years-of-lhc-physics-in-numbers

9. Michelle Starr, *Less Than 1% of Large Hadron Collider Data Ever Gets Looked At*, 6 January 2019, retrieved 2021-02-04, https://www.sciencealert.com/over-99-percent-of-large-hadron-collider-particle-collision-data-is-lost

10. *Processing: What to record?*, retrieved 2021-02-04, https://home.cern/science/computing/processing-what-record

11. *Processing LHC data*, retrieved 2021-02-04, https://videos.cern.ch/record/1541893 available at: *Processing: What to record?*, https://home.cern/science/computing/processing-what-record

12. *7 Key Facts and Figures – CERN Data Centre, As of 1 June 2018_V1*, retrieved 2021-02-04, https://information-technology.web.cern.ch/sites/information-technology.web.cern.ch/files/CERNDataCentre_KeyInformation_01June2018V1.pdf

13. Mélissa Gaillard, *CERN Data Centre passes the 200-petabyte milestone*, 6 July, 2017, retrieved 2021-02-04, https://home.cern/news/news/computing/cern-data-centre-passes-200-petabyte-milestone

14. 9 A. M. M. Scaife, *Big telescope, big data: towards exascale with the Square Kilometre Array*, 20 January 2020, https://doi.org/10.1098/rsta.2019.0060, retrieved 2021-02-04, https://royalsocietypublishing.org/doi/10.1098/rsta.2019.0060

15. Sarah Binns, *The World's Largest Telescope Will Generate More Data Than the Entire Internet 5 min read*, retrieved 2021-02-04, https://academicstories.com/story/inspiring-ideas/the-world-s-largest-telescope-will-generate-more-data-than-the-entire-internet

16. By Brian Kent and Joseph Masters, *Day and night, we're mapping the sky (and producing terabytes of data),* August 20, 2018, retrieved 2021-02-04, https://public.nrao.edu/blogs/how-much-data/

17. *Storage. What data to record*, retrieved 2021-04-02, https://home.cern/science/computing/storage

[HTTP] HTTP

1. *RFC 7230: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing 3. Message Format*, retrieved 2020-08-29, https://tools.ietf.org/html/rfc7230#section-3

[JS] Java Script

1. *A brief history of JSON*, retrieved 2020-05-11, https://blog.sqlizer.io/posts/json-history/

2. *Standard ECMA-404. The JSON Data Interchange Syntax*, retrieved 2020-05-10, https://www.ecma-international.org/publications/standards/Ecma-404.htm

[McCreKel] Daniel G. McCreary and Ann M. Kelly, *Making Sense of NoSQL. A guide for managers and the rest of us*, Manning Publications, 2013.

[MIS] Miscellaneous

1. *25 Most Famous Libraries Of The World*, retrieved: 2020-03-07, http://www.mastersinlibraryscience.net/25-most-famous-libraries-of-the-world/

2. *THE BEST LIBRARIES IN THE WORLD*, retrieved: 2019-03-07, https://www.thebestcolleges.org/amazing-libraries/

3. *Why 7 Seconds Could Make or Break Your Mobile App*, retrieved 2020-04-20, https://asostack.com/why-7-seconds-could-make-or-break-your-mobile-app-f41000fb2a17

4. *UX best practices: How to design scannable app screenshots*, retrieved 2020-04-20, https://www.freecodecamp.org/news/ux-best-practices-how-to-design-scannable-app-screenshots-89e370bf433e/

[MRvsDaC] *MapReduce vs classic divide-and-conquer approach*, retrieved 2020-05-05,
https://fulmanski.pl/tutorials/computer-science/big-data/processing-concepts-for-big-data/#types_batch_map_reduce_vs_divide_and_conquer

[NewSQL]

1. *The beginning of the end of NoSQL*, retrieved 2020-01-29, https://blogs.the451group.com/information_management/2010/11/12/the-beginning-of-the-end-of-nosql/

2. *What we talk about when we talk about NewSQL,* retrieved 2020-01-29, https://blogs.the451group.com/information_management/2011/04/06/what-we-talk-about-when-we-talk-about-newsql/

3. Andrew Pavlo, Matthew Aslett, *What's Really New with NewSQL?,* CM SIGMOD Record, 45, . (2016), pp. 45-55, retrieved 2020-01-29, https://db.cs.cmu.edu/papers/2016/pavlo-newsql-sigmodrec2016.pdf

4. Michael Stonebraker *New SQL: An Alternative to NoSQL and Old SQL For New OLTP Apps,* retrieved 2020-01-29, https://cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/fulltext

[RBE] Rule Based Engine

1. *Drools*, retrieved 2020-09-08, https://www.drools.org

2. *Jess*, retrieved 2020-09-08, https://jess.sandia.go

[TSDB] Time Series Data Base

1. *DBMS popularity broken down by database model*, retrieved 2021-02-01,
https://db-engines.com/en/ranking_categories

2. *Understanding Metrics and Time Series*, in: OpenTSDB 2.4 documentation, retrieved 2021-02-06,
http://opentsdb.net/docs/build/html/user_guide/query/timeseries.html

3. Ted Dunning, Ellen Friedman, *Time Series Databases. New Ways to Store and Access Data*, 2015, O'Reilly.

4. *Why NOT to Build a Time-Series Database*, Outlyer, Nov 7, 2018, retrieved 2021-02-05
https://medium.com/dataseries/why-not-to-build-a-time-series-database-e1e63a535357

5. *OpenTSDB FAQ*, retrieved 2021-02-10,
http://opentsdb.net/faq.html

[X] XML and related things

1. *Web Toolkit Online. XPath Tester*, retrieved 2020-05-13,
https://www.webtoolkitonline.com/xml-xpath-tester.html

2. *Code Beautify. XPath Tester*, retrieved 2020-05-13,
https://codebeautify.org/Xpath-Tester

3. *Blooming FLWOR - An Introduction to the XQuery FLWOR Expression*, retrieved 2020-05-13,
www.stylusstudio.com/xquery-flwor.html

4. *Wikipedia:XML_Schema_(W3C):Criticism*, retrieved 2020-08-29,
https://en.wikipedia.org/wiki/XML_Schema_(W3C)#Criticism

[???BOOK???]