

NoSQL

Introduction to NoSQL: Lecture 3

Piotr Fulmański



FACULTY OF MATHEMATICS
AND COMPUTER SCIENCE
University of Lodz

NoSQL Theory and examples

by Piotr Fulmański

Piotr Fulmański, 2021

PIOTR FULMAŃSKI

NoSQL Theory and examples



SIMPLE INTRODUCTION SERIES

- The most crucial motivation and its consequences
- A radical change in mindse
- BASE, CAP and consistency

What is NoSQL?

- It's much **more than rows** in tables. NoSQL systems store and retrieve data from many formats: key-value, graph, column-family, document, and of course rows in tables.
- It's **free of joins**. NoSQL systems allow us to extract our data using simple interfaces **without joins**.
- It's **schema-free**. In most cases you don't have to create an entity-relational model.
- NoSQL systems are **easily scalable**.
 - In consequence they work on multiple processors and can run well on low-cost separate computer systems - no need for expensive nodes to get high-speed performance.
 - Scalability supported by NoSQL systems is close to linear. In consequence increasing the number of processing units by factor of N you get increase in performance by factor close to N.
- NoSQL is a response to nowadays business data related factors:
 - volume and velocity, referring to the ability to handle large datasets that arrive fast;
 - variability, referring to how diverse data types don't fit into structured tables;
 - and agility, referring to how fast an organization responds to business changes.

What is not NoSQL?

- It's **not against the SQL** language. SQL as well as other query languages are used with NoSQL databases.
- NoSQL is **not Big Data**.
- It's **not about cloud computing**.
- It's **not close group of companies**, software or product. Anybody can be a big player in this market if only offer innovative solutions to business problems.

Motivations

Availability and speed

Today people don't read. They scan.

Application	2016	2018
airbnb	words: 6, font size: 43	words: 4, font size: 91
tinder	words: 6, font size: 50	words: 3, font size: 112
Spotify	words: 4, font size: 45	words: 3, font size: 100
Pinterest	words: 10, font size: 40	words: 3, font size: 90

Motivations

Scalability

If we agree that availability and speed is a key feature we must have, this raises a short question: *How?*

The answer: *use scaling.*

Scalability is the property of a system to handle a growing amount of work by adding resources to the system.

When availability is considered, easy scalability is one of the most required feature.

Motivations

Scalability - how it is possible: no joins

If you agree that scalability is a key feature you must have, this raises again the same short question: *How?*

What should characterize data to make their parallel processing possible?

Data should not be dependent each other. If you add, delete or update one part of it, it should not affect any other part. **There should be no connections between data parts.**

One of well known connection of that type, linking data and allowing traversal through them are **relational foreign keys**. Keys are the fundamental mechanism used to combine data distributed among tables.

Unfortunately what is necessary for the relational databases to function, at the same time is its limitation. **You can't operate on data in parallel as you have to obey keys constraint.** To use any piece of data almost always you have to join them by keys. **Without keys and joins almost nothing can't be done in relational systems.**

Motivations

No joins, no schema... Schema-free trap

At the first sight you may feel happy as in NoSQL no schemas are required. You don't have to think ahead how you split your data into tables and columns. You are finally free from having to think about normal forms.

If you think that this way your life would be simpler and easier from now, you are wrong. Completely wrong!

If you think that schema-free freedom allows you to do what you want and when you want, you are on the straight way to total disaster.

In SQL you never ask *who* and *how* is going to use data. In NoSQL both questions are the most fundamental and future actions depend on the answers to these questions.

Motivations

No joins, no schema... Schema-free trap

A radical change in mindset

No schema doesn't mean no rules. **You have to change your point of view from being universal of *one size fits all* type to being precise and focused.**

While in SQL databases whatever task you do, your database has the same form, in NoSQL data is modeled **around** specific queries.

Query-driven data modeling means that the data access patterns and application queries determine the structure and organization of data.

Wow! Déjà vu? Pre-relational era is going back?

Motivations

Query-driven data modeling... Aggregations

Data modeling in Apache Cassandra column store

Data modeling is the process of identifying entities and their relationships. In relational databases, data is placed in normalized tables with foreign keys used to reference related data in other tables. **Queries that the application will make are driven by the structure of the tables and related data are queried as table joins.**

In Cassandra, data modeling is query-driven. **The data access patterns and application queries determine the structure and organization of data** which then used to design the database tables.

Data is modeled around specific queries. Queries are best designed to access a single table, which implies that all entities involved in a query must be in the same table to make data access (reads) very fast. Data is modeled to best suit a query or a set of queries. A table could have one or more entities as best suits a query. As entities do typically have relationships among them and queries could involve entities with relationships among them, **a single entity may be included in multiple tables.**

Motivations

Query-driven data modeling... Aggregations

Query-driven modeling in Apache Cassandra column store

Unlike a relational database model in which queries make use of table joins to get data from multiple tables, **joins are not supported in Cassandra so all required fields (columns) must be grouped together in a single table.** Since each query is backed by a table, **data is duplicated across multiple tables in a process known as denormalization.** Data duplication and a high write throughput are used to achieve a high read performance.

Motivations

Query-driven data modeling... Aggregations

Because in NoSQL there are no keys and joins we can't combine data coming from multiple sources (multiple tables) into one group provided to the user.

Instead **you organize your data into one self-contained group**. Whatever you want to get with one query should be enclosed within it.

You shouldn't think in terms of data organized into tables according to normal forms rules but rather in terms of **atomic group holding all data accessed together with just one call**.

Such a base unit of data organization, an atomic and self-contained group of data, is named ***aggregate***. Under the name *aggregation* you should understand the whole process of transforming your data into such a units.

Aggregate should contain all data needed to complete single request. And that should be only that data which are really needed.

Motivations

Query-driven data modeling... Aggregations

Consider a group of people taking part in different projects:

```
person A: project 1
person B: project 1
person B: project 2
person C: project 2
person A: project 3
person C: project 3
```

Motivations

Query-driven data modeling... Aggregations

If you expect or know that information about persons involved in the project will be needed, you should operate with the following aggregates:

```
{
  {project: project1,
   person: [personA: {...},
            personB: {...}]},

  {project: project2,
   person: [personB: {...},
            personC: {...}]},

  {project: project3,
   person: [personA: {...},
            personC: {...}]}
}
```

If additionally you need an information about the total working time of each person in all their projects, you should use another aggregates:

```
{
  {person: personA,
   totalWorkingTime: 123},

  {person: personB,
   totalWorkingTime: 27},

  {person: personC,
   totalWorkingTime: 62}
}
```

BASE

Query-driven data modeling... Aggregations

Basic availability means that the **database appears to work most of the time**. It allows systems to be temporarily inconsistent so that transactions are manageable. In BASE systems, the information and service capability are basically available. This means that there can be a partial failure in some parts of the distributed system but the rest of the system continues to function.

Soft-state means that stores don't have to be write-consistent, nor do different replicas have to be mutually consistent all the time. **Some inaccuracy is temporarily allowed and data may change while being used. State of the system may change over time, even without input.** This is because of eventual consistency.

Eventual consistency means that there may be times when the database is in an inconsistent state. Eventually, when all service logic is executed, the system is left in a consistent state.

Contrary to ACID systems, which are **pessimistic** and because of this are ready to survive any disaster you can ever imagine, BASE are **optimistic** as they assume that eventually, in not so distance future, all systems will catch up and become consistent.

CAP theorem

The CAP theorem is about how distributed database systems behave in the face of network instability.

According to the CAP theorem introduced by Eric Brewer in 2000, any distributed database system can have at most two of the following three desirable properties.

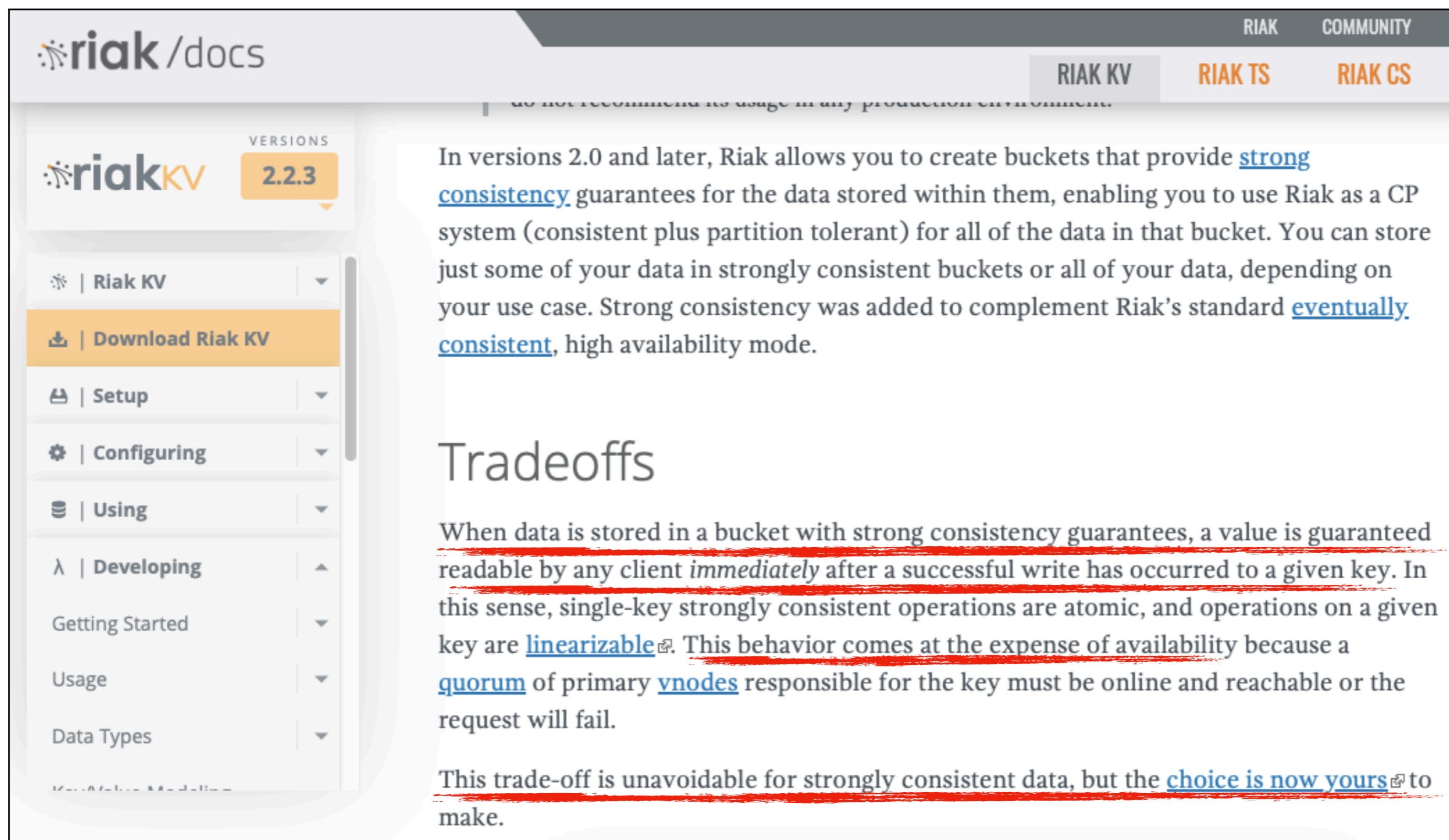
Consistency. Consistency is about having a single, up-to-date, readable version of our data available to all clients. Your data should be consistent – no matter how many clients read the same items from replicated and distributed partitions, you should get consistent results. All writes are atomic and all subsequent requests retrieve the new value.

High availability. This property states that the database will always allow clients to make operations like select or update on items without delay. Internal communication failures between replicated data shouldn't prevent operations on it. The database will always return a value as long as a single server is running.

Partition tolerance. This is the ability of the system to keep responding to client requests even if there's a communication failure between database partitions. The system will still function even if network communication between partitions is temporarily lost.

The CAP theorem helps us understand that **once you partition your data, you must determine which of two exclusive options best match your business requirements: consistency or availability.**

(in)consistency



The screenshot shows the Riak KV documentation page. The sidebar on the left contains a 'VERSIONS' dropdown set to '2.2.3' and a list of navigation links: 'Riak KV', 'Download Riak KV', 'Setup', 'Configuring', 'Using', 'Developing', 'Getting Started', 'Usage', and 'Data Types'. The main content area has a header with 'RIAK KV', 'RIAK TS', and 'RIAK CS' tabs. The text in the main area discusses strong consistency guarantees in Riak versions 2.0 and later, mentioning that data is guaranteed readable by any client immediately after a successful write. It also discusses tradeoffs, noting that this behavior comes at the expense of availability because a quorum of primary vnodes must be online and reachable.

riak /docs

RIAK COMMUNITY

RIAK KV RIAK TS RIAK CS

riakKV 2.2.3

VERSIONS

⌵ | Riak KV

⬇ | Download Riak KV

🏠 | Setup

⚙️ | Configuring

☰ | Using

λ | Developing

Getting Started

Usage

Data Types

do not recommend its usage in any production environment.

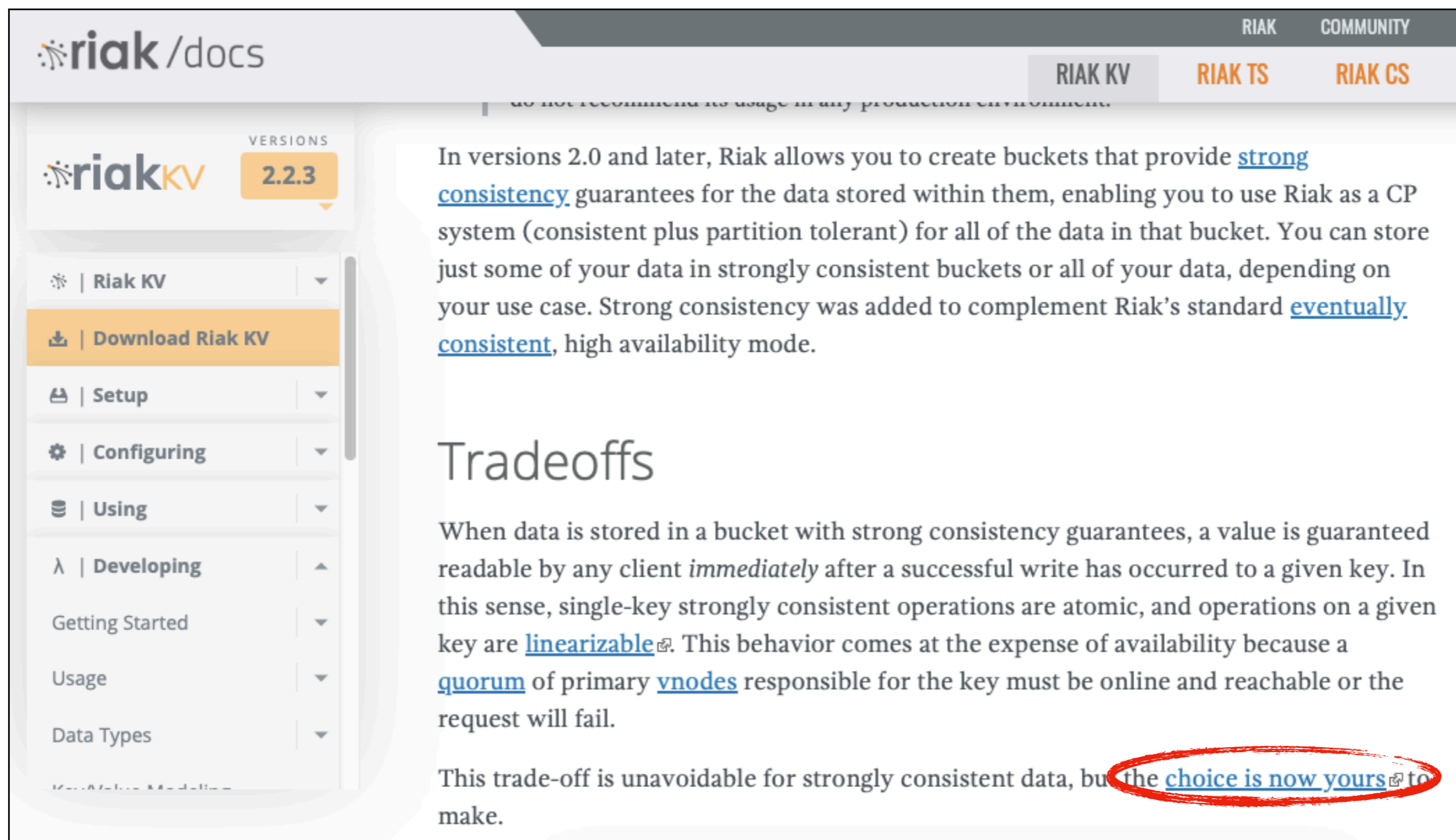
In versions 2.0 and later, Riak allows you to create buckets that provide [strong consistency](#) guarantees for the data stored within them, enabling you to use Riak as a CP system (consistent plus partition tolerant) for all of the data in that bucket. You can store just some of your data in strongly consistent buckets or all of your data, depending on your use case. Strong consistency was added to complement Riak's standard [eventually consistent](#), high availability mode.

Tradeoffs

When data is stored in a bucket with strong consistency guarantees, a value is guaranteed readable by any client *immediately* after a successful write has occurred to a given key. In this sense, single-key strongly consistent operations are atomic, and operations on a given key are [linearizable](#). This behavior comes at the expense of availability because a [quorum](#) of primary [vnodes](#) responsible for the key must be online and reachable or the request will fail.

This trade-off is unavoidable for strongly consistent data, but the [choice is now yours](#) to make.

(in)consistency



The screenshot shows the Riak KV documentation page. The sidebar on the left contains a 'VERSIONS' dropdown set to '2.2.3' and a list of navigation links: 'Riak KV', 'Download Riak KV', 'Setup', 'Configuring', 'Using', 'Developing', 'Getting Started', 'Usage', and 'Data Types'. The main content area has a header with 'RIAK KV', 'RIAK TS', and 'RIAK CS' tabs. The text discusses strong consistency guarantees and tradeoffs.

do not recommend its usage in any production environment.

In versions 2.0 and later, Riak allows you to create buckets that provide [strong consistency](#) guarantees for the data stored within them, enabling you to use Riak as a CP system (consistent plus partition tolerant) for all of the data in that bucket. You can store just some of your data in strongly consistent buckets or all of your data, depending on your use case. Strong consistency was added to complement Riak's standard [eventually consistent](#), high availability mode.

Tradeoffs

When data is stored in a bucket with strong consistency guarantees, a value is guaranteed readable by any client *immediately* after a successful write has occurred to a given key. In this sense, single-key strongly consistent operations are atomic, and operations on a given key are [linearizable](#). This behavior comes at the expense of availability because a [quorum](#) of primary [vnodes](#) responsible for the key must be online and reachable or the request will fail.

This trade-off is unavoidable for strongly consistent data, but [the choice is now yours](#) to make.

(in)consistency

Choice is now yours

WIKIPEDIA The Free Encyclopedia

Search Wikipedia Search

Create account Log in

Participate in the 2023 international science photo competition!

CAP theorem

17 languages

Contents [hide]

(Top)

Explanation

History

See also

References

External links

Article Talk

Read Edit View history Tools

From Wikipedia, the free encyclopedia

In [theoretical computer science](#), the **CAP theorem**, also named **Brewer's theorem** after computer scientist [Eric Brewer](#), states that any [distributed data store](#) can provide only **two of the following three** guarantees:^{[1][2][3]}

Consistency

Every read receives the most recent write or an error.

Availability

Every request receives a (non-error) response, without the guarantee that it contains the most recent write.

Partition tolerance

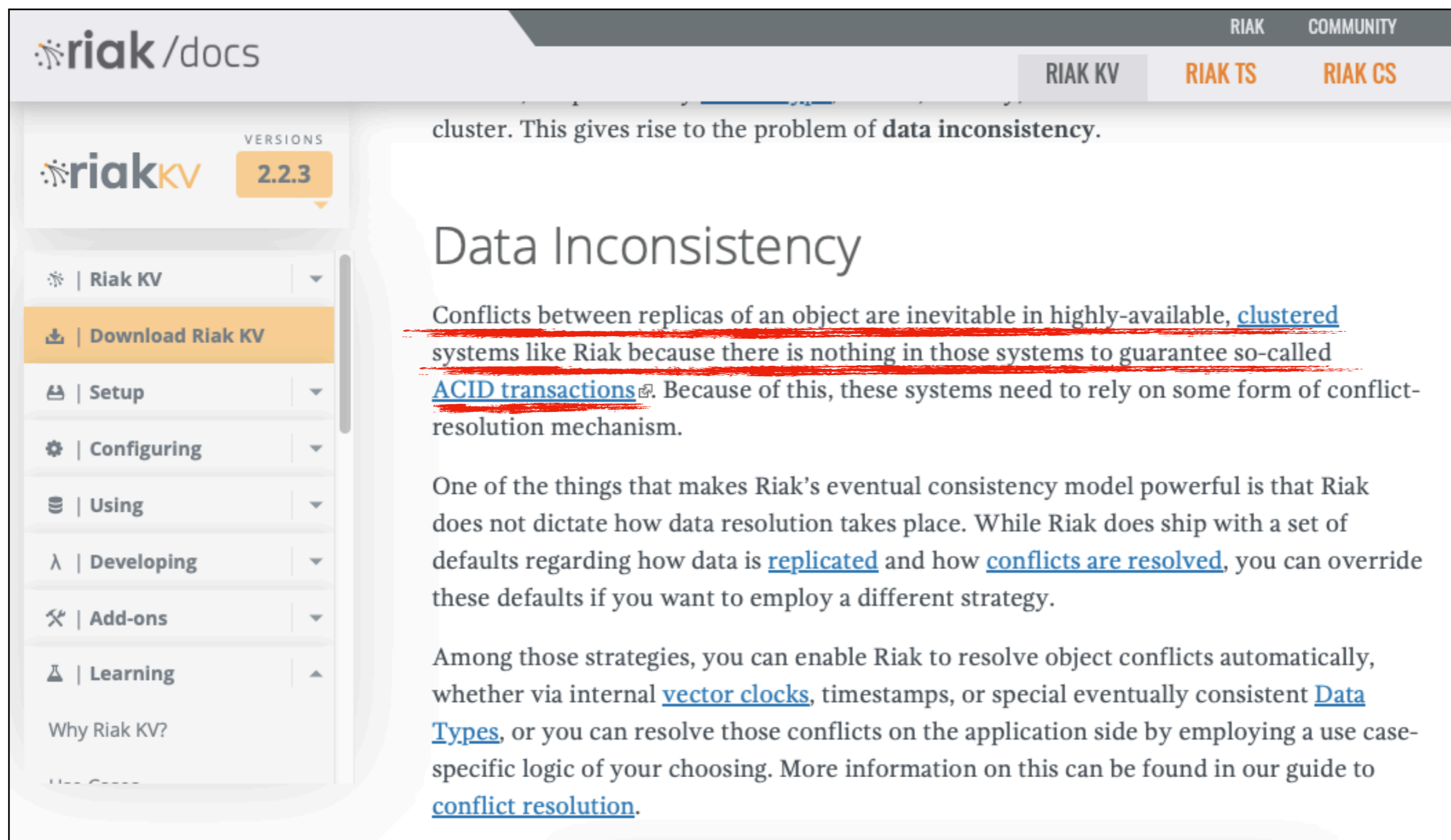
The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

When a [network partition](#) failure happens, it must be decided whether to do one of the following:

- cancel the operation and thus decrease the availability but ensure consistency
- proceed with the operation and thus provide availability but risk inconsistency.

Thus, if there is a network partition, one has to choose between consistency or availability. Note that consistency as defined in the CAP theorem is quite different from the consistency guaranteed in [ACID database transactions](#).^[4]

(in)consistency



riak/docs

RIAK COMMUNITY

RIAK KV RIAK TS RIAK CS

riakKV 2.2.3

VERSIONS

- ⌵ | Riak KV
- ⬇ | Download Riak KV
- ⌵ | Setup
- ⌵ | Configuring
- ⌵ | Using
- ⌵ | Developing
- ⌵ | Add-ons
- ⬆ | Learning

Why Riak KV?

cluster. This gives rise to the problem of **data inconsistency**.

Data Inconsistency

Conflicts between replicas of an object are inevitable in highly-available, clustered systems like Riak because there is nothing in those systems to guarantee so-called ACID transactions. Because of this, these systems need to rely on some form of conflict-resolution mechanism.

One of the things that makes Riak's eventual consistency model powerful is that Riak does not dictate how data resolution takes place. While Riak does ship with a set of defaults regarding how data is replicated and how conflicts are resolved, you can override these defaults if you want to employ a different strategy.

Among those strategies, you can enable Riak to resolve object conflicts automatically, whether via internal vector clocks, timestamps, or special eventually consistent Data Types, or you can resolve those conflicts on the application side by employing a use case-specific logic of your choosing. More information on this can be found in our guide to conflict resolution.

Consistency

Working with NoSQL system you have to remember that **your data is only guaranteed to be almost accurate**. This makes a big difference compared to relational system where data is **always accurate**.

- **Casual consistency.** Casual consistency means that the database reflects the order in which operations were performed.
- **Read-your-writes consistency.** Read-your-writes consistency means that once you have updated a record, all of our subsequent reads of that record will return the updated value.
- **Session consistency.** Session consistency means read-your-writes consistency but at session level. Session can be identified with a conversation between a client and a server. As long as the conversation continues, you will read everything you have wrote during this conversation. If the session ends and you start another session with the same server, there is no this guarantee that you can read values you have wrote during previous conversation.
- **Monotonic read consistency.** Monotonic read consistency means that whenever you make a query and see a result, you will never see an earlier version of the value.
- **Monotonic write consistency.** Monotonic write consistency means that every time you make several update commands, they would be executed in the order you issued them.

Bibliography

- [Ful] Piotr Fulmański, *NoSQL. Theory and examples*, Piotr Fulmański, 2021