UNIVERSITÀ DEGLI STUDI DI PADOVA

J. Pazzini
PADOVA UNIVERSITY, INFN

6 - NOSQL DATABASES

Management and Analysis of Physics Datasets - Module B

Physics of Data

A.A. 2023/2024

DB PARTITIONING

How DBs can be distributed across multiple nodes→ partitioning and sharding

Partitioning → subdividing the DB into multiple smaller parts (not necessarily on multiple nodes of a cluster)

Vertical Partitioning

id	feat1	feat2	feat3
…	…	…	…
100	"foo"	12.3	[1.9, -2.0, 0.3]
101	"something"	1.3e-5	[1.3, 2.3, 5.4]
102	"bar"	-0.2	[5.3, 1.2, -3.4]
…	…	…	…

id	feat3
…	…
100	[1.9, -2.0, 0.3]
101	[1.3, 2.3, 5.4]
102	[5.3, 1.2, -3.4]
…	…

id	feat1	feat2
…	…	…
100	"foo"	12.3
101	"something"	1.3e-5
102	"bar"	-0.2
…	…	…

DB PARTITIONING

How DBs can be distributed across multiple nodes⟶ partitioning and sharding

Partitioning → subdividing the DB into multiple smaller parts (not necessarily on multiple nodes of a cluster)

Horizontal Partitioning (*sharding*)

id	feat1	feat2	feat3
…	…	…	…
100	"foo"	12.3	[1.9, -2.0, 0.3]
101	"something"	1.3e-5	[1.3, 2.3, 5.4]
102	"bar"	-0.2	[5.3, 1.2, -3.4]
…	…	…	…

id	feat1	feat2	feat3
…	…	…	…
101	"something"	1.3e-5	[1.3, 2.3, 5.4]
102	"bar"	-0.2	[5.3, 1.2, -3.4]
…	…	…	…

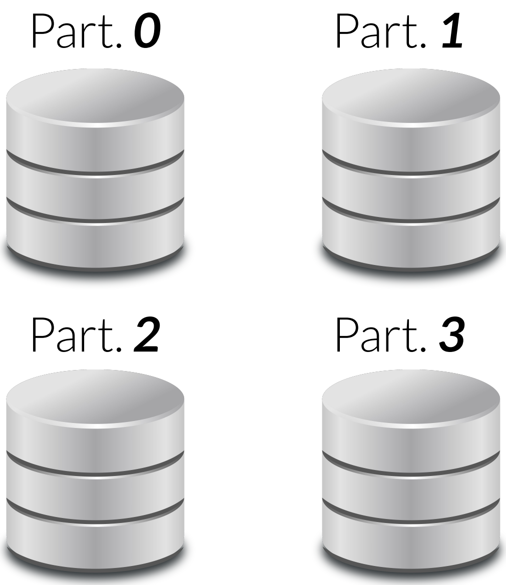
id	feat1	feat2	feat3
…	…	…	…
100	"foo"	12.3	[1.9, -2.0, 0.3]
…	…	…	…

SHARDING AND HASHING

Records are sharded to a number P of partitions typically based on Hashing

- A simple method is to define a Key ***k***, perform Hashing ***hash(k)***, and choose as partition ***hash(k)%P***

id	feat1	feat2	feat3
…	…	…	…
100	"foo"	12.3	[1.9, -2.0, 0.3]
101	"something"	1.3e-5	[1.3, 2.3, 5.4]
102	"bar"	-0.2	[5.3, 1.2, -3.4]
…	…	…	…



SHARDING AND HASHING

Records are sharded to a number P of partitions typically based on Hashing

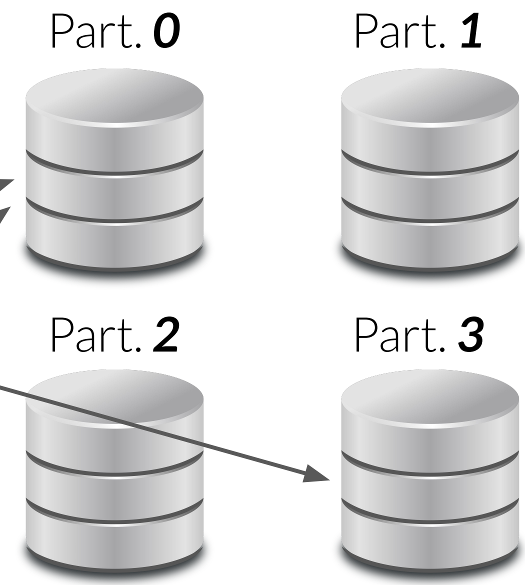
- A simple method is to define a Key ***k***, perform Hashing ***hash(k)***, and choose as partition ***hash(k)%P***

Key ***k*** → hash(***k***)

100 → 6893038736648420296

id	feat1	feat2	feat3
…	…	…	…
100	"foo"	12.3	[1.9, -2.0, 0.3]
101	"something"	1.3e-5	[1.3, 2.3, 5.4]
102	"bar"	-0.2	[5.3, 1.2, -3.4]
…	…	…	…

hash(k)%P
…
0
3
0
…



CONSISTENT HASHING

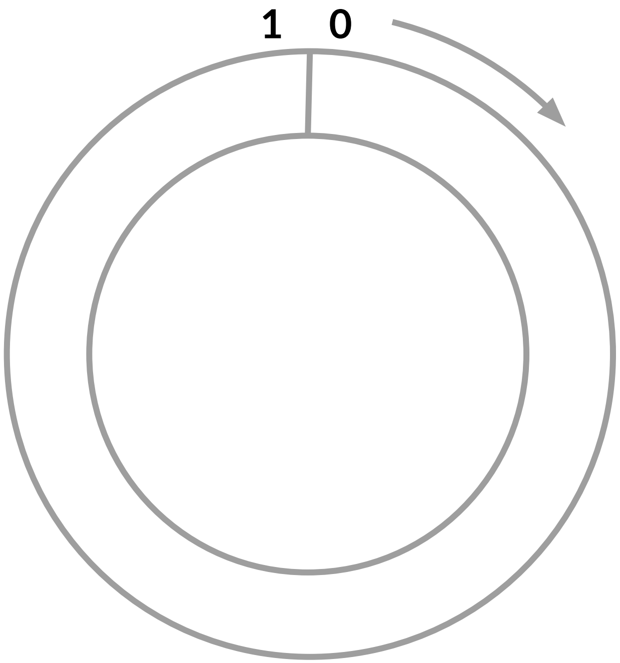
How can we handle sharding over a number ***M ≠ P*** of partitions?

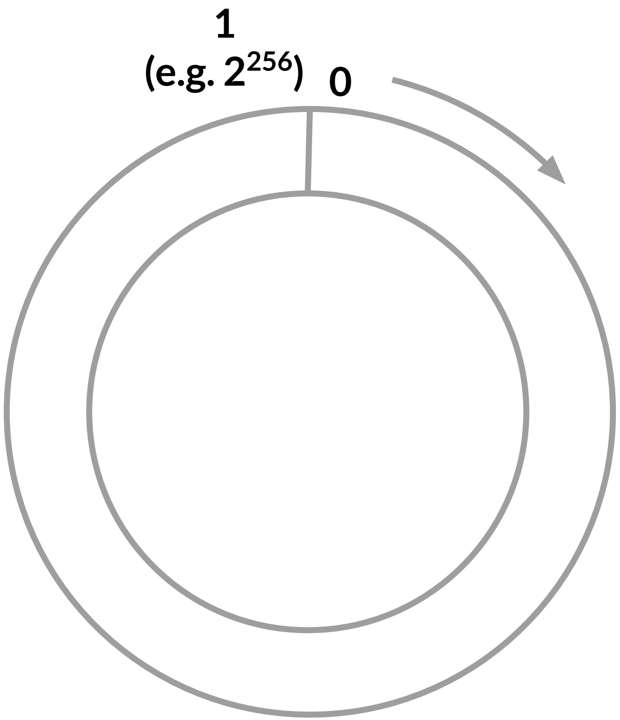
→ adding more partitions

→ removing 1+ partition

Instead of mapping the ***hask(K)*** with the module of partitions, we map ***hash(K)*** into a *ring of values*

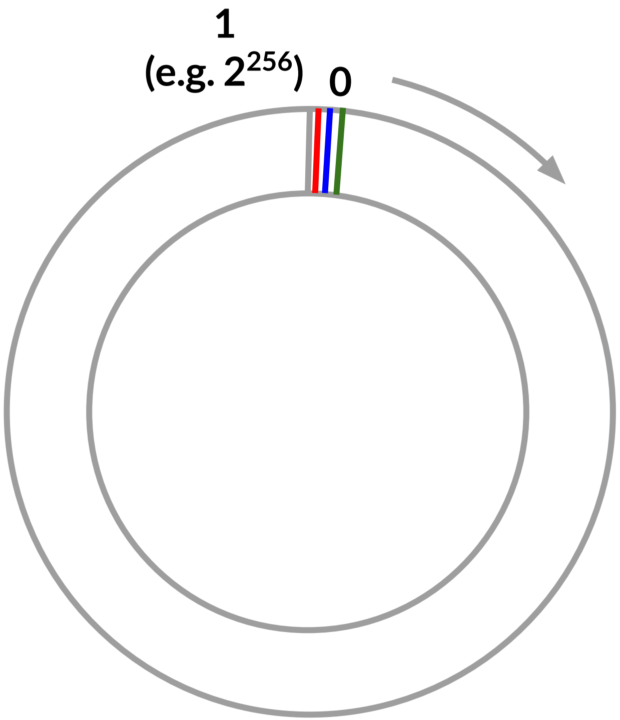
e.g. in SHA256
→ 2^{256} possible hashes (~10^{77})





Under the simple sharding approach shown before every hash digest is mapped to a partition according to its ***mod(P)***

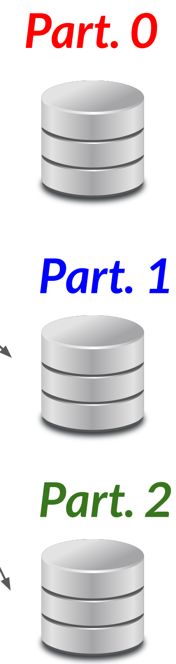
Assuming we have ***P=3***



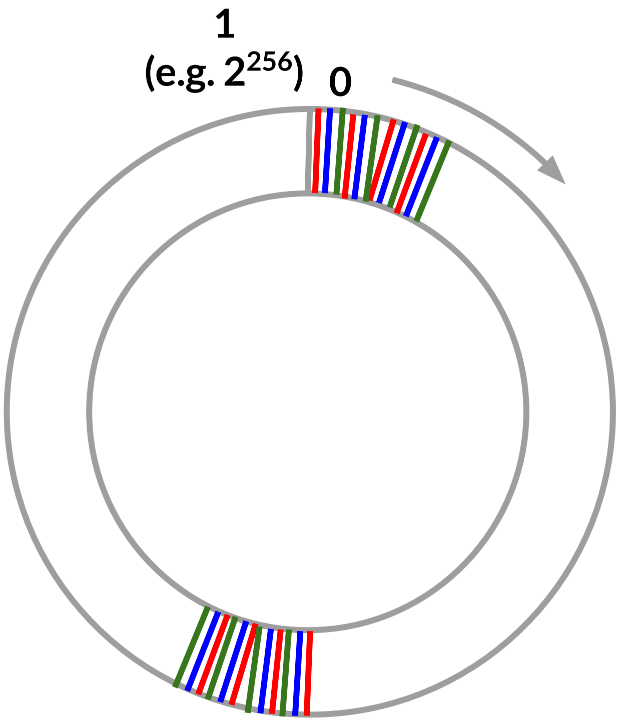
Under the simple sharding approach shown before every hash digest is mapped to a partition according to its ***mod(P)***

Assuming we have ***P=3***

```
hash(K_a) == 0   ⇒ Part. 0
hash(K_b) == 1   ⇒ Part. 1
hash(K_c) == 2   ⇒ Part. 2
```



CONSISTENT HASHING



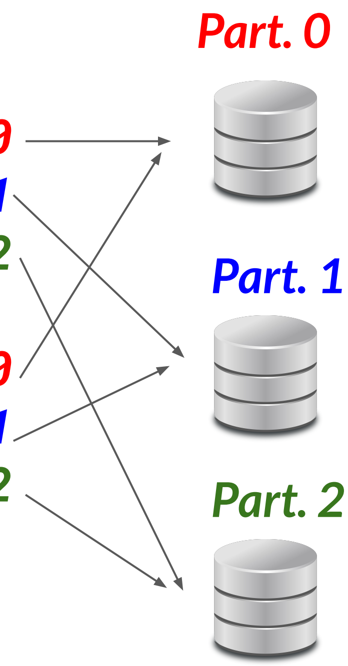
Under the simple sharding approach shown before every hash digest is mapped to a partition according to its ***mod(P)***

Assuming we have ***P=3***

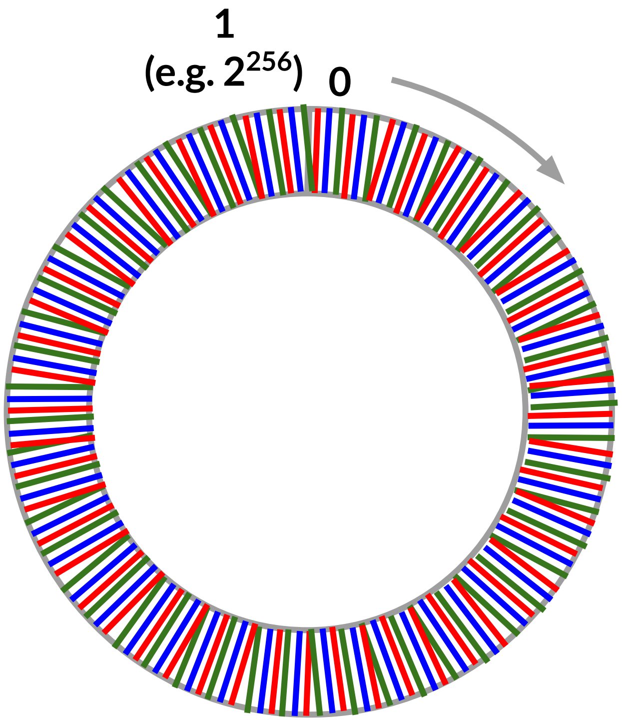
```
hash(K_a) == 0  ⇒ Part. 0
hash(K_b) == 1  ⇒ Part. 1
hash(K_c) == 2  ⇒ Part. 2

hash(K_d) == 3  ⇒ Part. 0
hash(K_e) == 4  ⇒ Part. 1
hash(K_f) == 5  ⇒ Part. 2
```

...



CONSISTENT HASHING

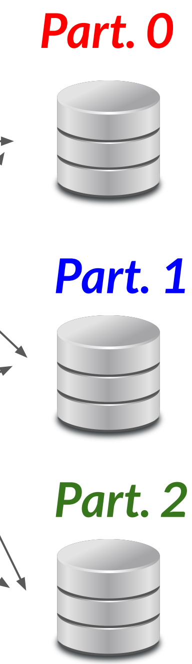


Under the simple sharding approach shown before every hash digest is mapped to a partition according to its ***mod(P)***

Assuming we have ***P=3***

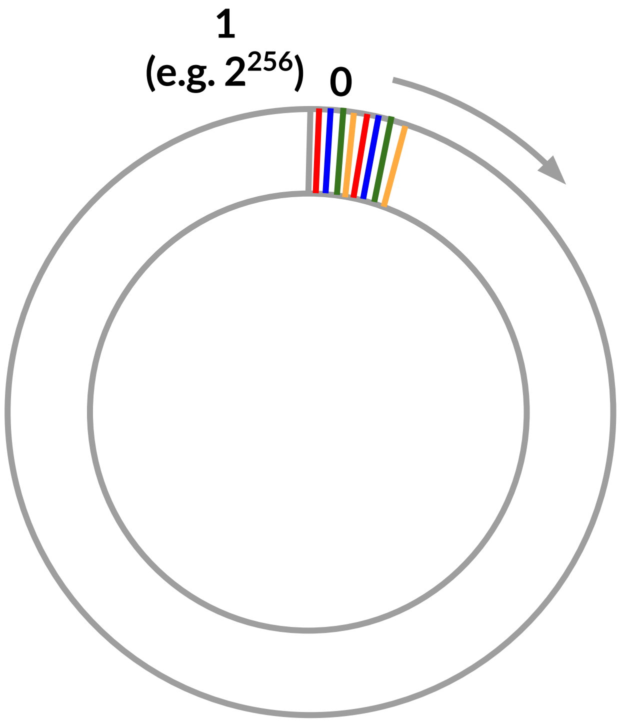
```
hash(K_a) == 0   ⇒ Part. 0
hash(K_b) == 1   ⇒ Part. 1
hash(K_c) == 2   ⇒ Part. 2

hash(K_d) == 3   ⇒ Part. 0
hash(K_e) == 4   ⇒ Part. 1
hash(K_f) == 5   ⇒ Part. 2
```



...

CONSISTENT HASHING



Adding a new partition (***Part. 3***) would imply re-evaluating every single record's address

All data will have to be moved across partitions

hash(K_a) == 0 ⇒ **Part. 0**
hash(K_b) == 1 ⇒ **Part. 1**
hash(K_c) == 2 ⇒ **Part. 2**

hash(K_d) == 3 ⇒ ~~**Part. 0**~~ **Part. 3**
hash(K_e) == 4 ⇒ ~~**Part. 1**~~ **Part. 0**
hash(K_f) == 5 ⇒ ~~**Part. 2**~~ **Part. 1**

...

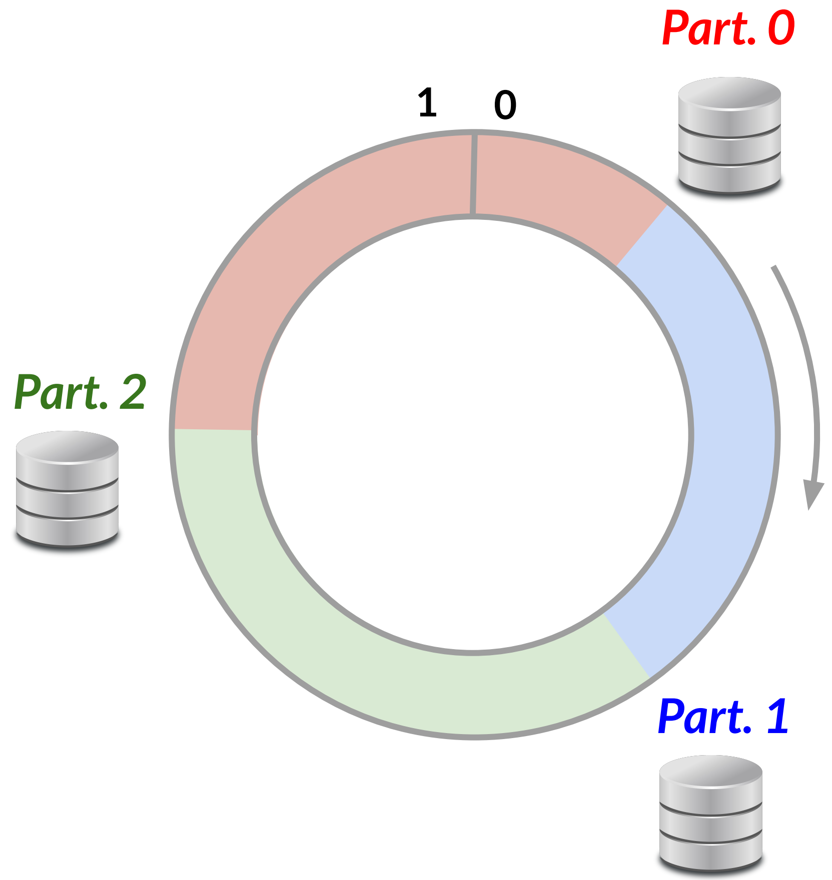
CONSISTENT HASHING

In ***consistent hashing*** partitions are associated with a range of hashes (a sector of the ring)

Each key is first hashed ***hash(K)***

Its digest is "rounded up" to the max value of the sector

The record value is then associated with that partition



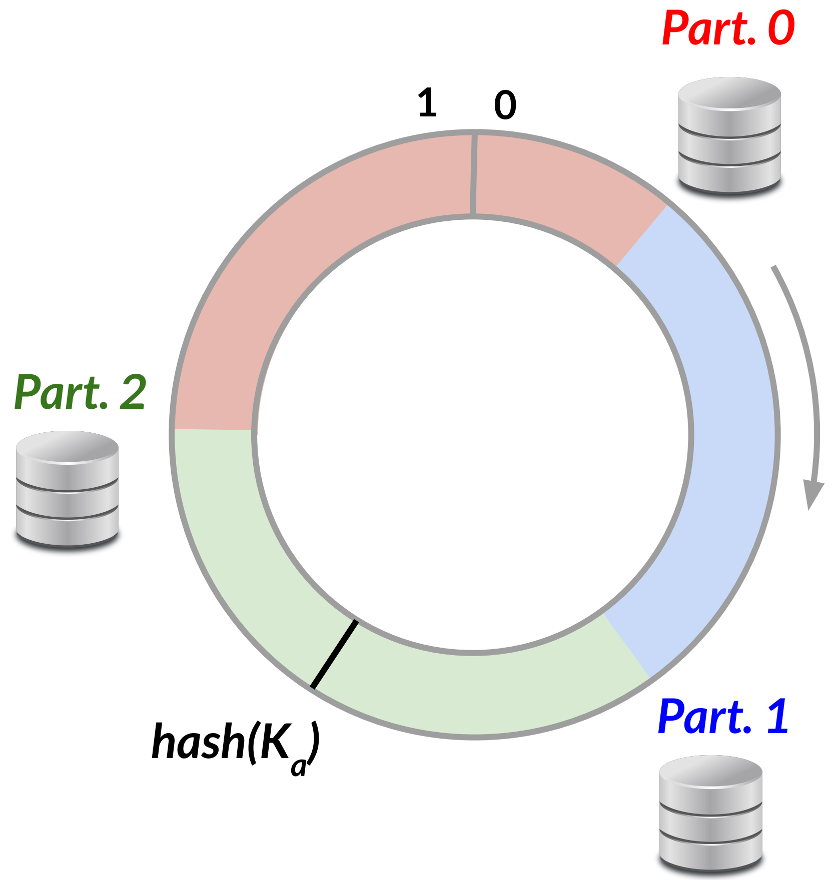
CONSISTENT HASHING

In ***consistent hashing*** partitions are associated with a range of hashes (a sector of the ring)

Each key is first hashed ***hash(K)***

Its digest is "rounded up" to the max value of the sector

The record value is then associated with that partition



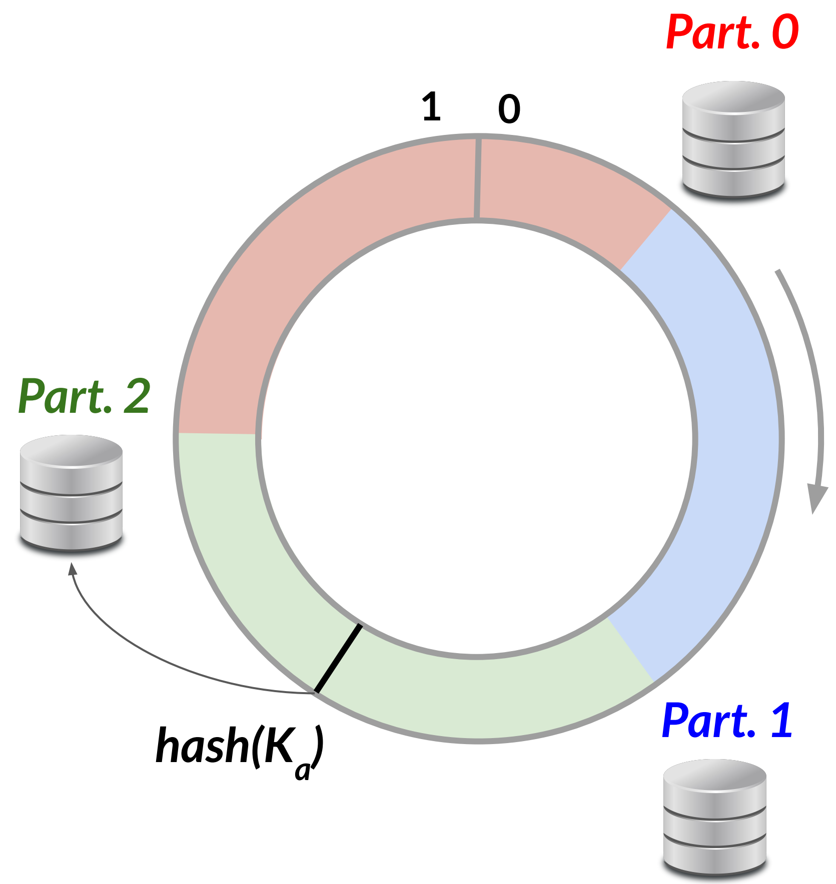
CONSISTENT HASHING

In ***consistent hashing*** partitions are associated with a range of hashes (a sector of the ring)

Each key is first hashed ***hash(K)***

Its digest is "rounded up" to the max value of the sector

The record value is then associated with that partition



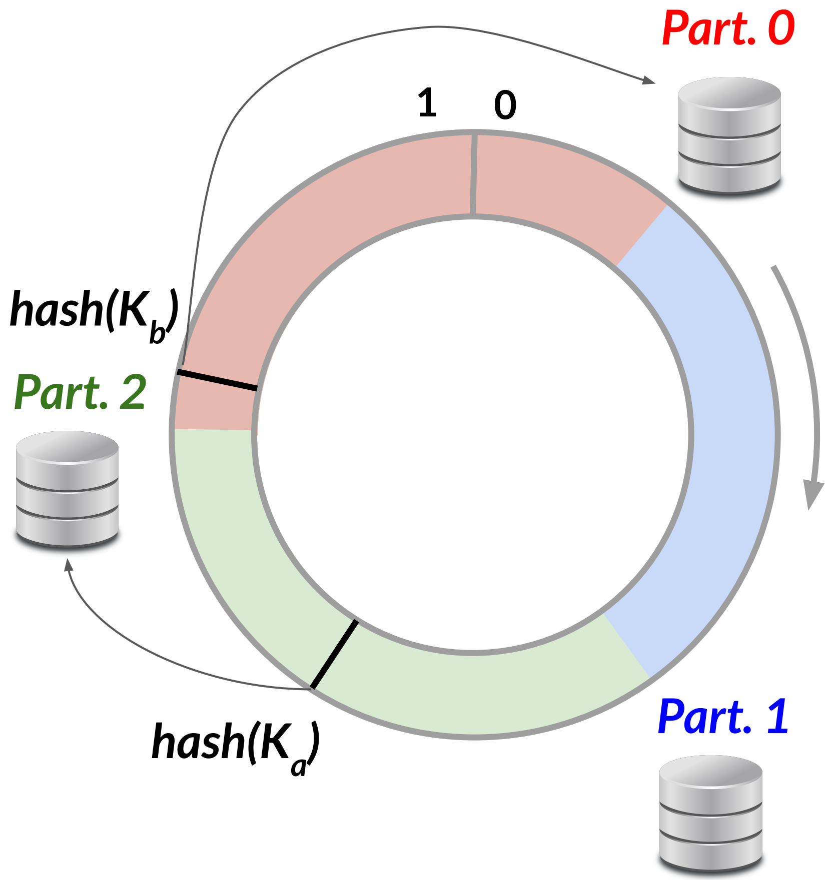
CONSISTENT HASHING

In ***consistent hashing*** partitions are associated with a range of hashes (a sector of the ring)

Each key is first hashed ***hash(K)***

Its digest is "rounded up" to the max value of the sector

The record value is then associated with that partition

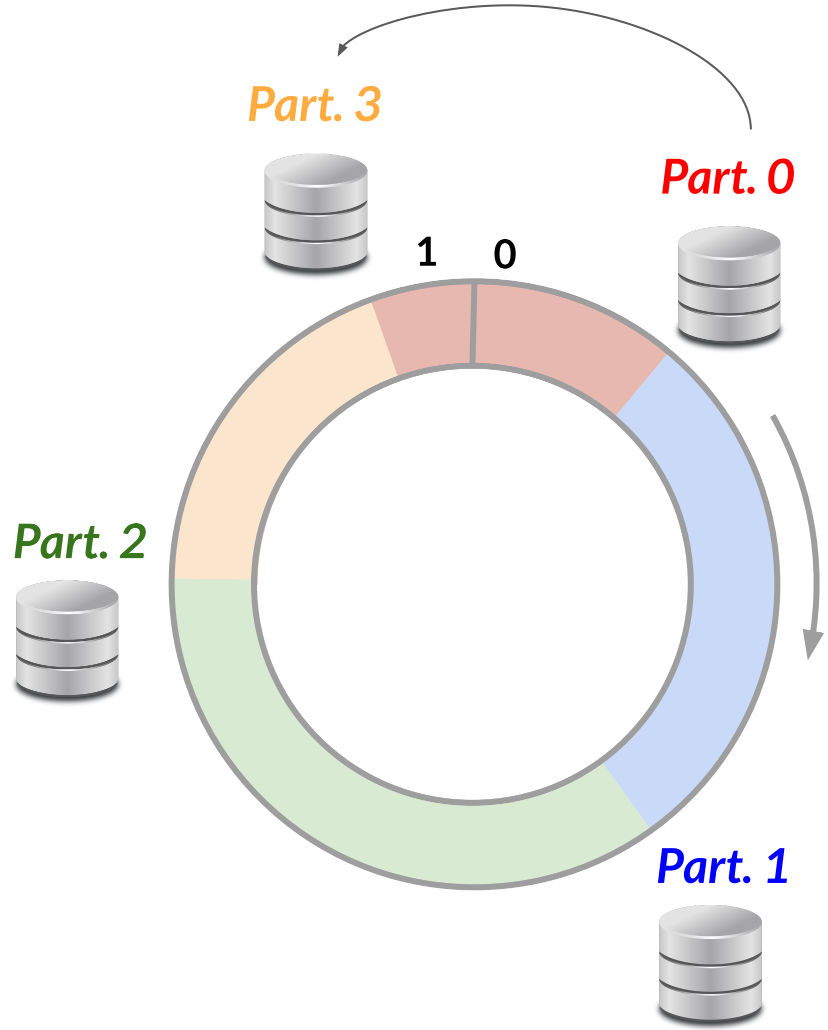


CONSISTENT HASHING

If a new partition (e.g. ***Part. 3***) is added, one does NOT have to move ALL data from ALL partitions

Only the data from the neighbor partitions is moved

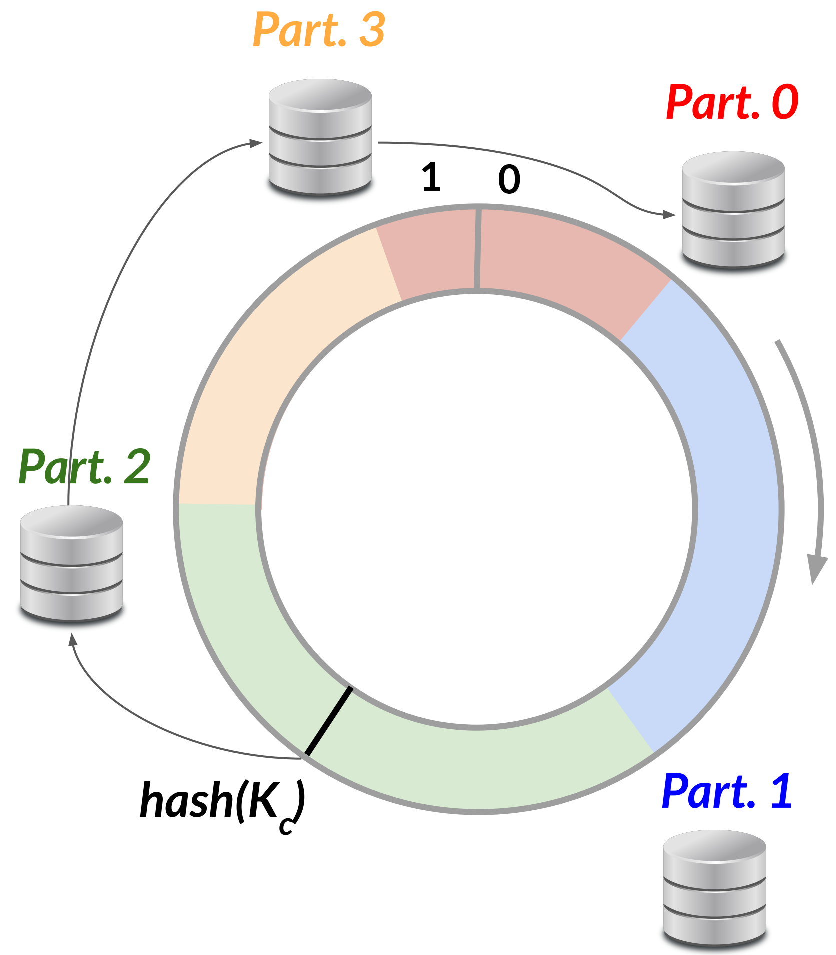
E.g. only FRACTION of the data from Part. 0 is moved to Part. 3



Consistent hashing is also used for replication of data across many partitions

Given a replication factor of ***n:***

When a new record is stored in its appropriate partition, a copy is relayed on the ***n-1*** closest partitions on the ring



E.g. replication factor = 3

DATABASE MODELS

Relational

Key-Value

Wide-Column

Document

Graph

others…

Relational →

Easily the most widely used
Designed for **structured data** and their relationship
Also (improperly) known as SQL DB

Key-Value, Wide-Column, Document, Graph →

NoSQL (Not Only SQL) DB
Different implementations depending on the data-model
Mostly designed for **non structured data** and horizontal scalability

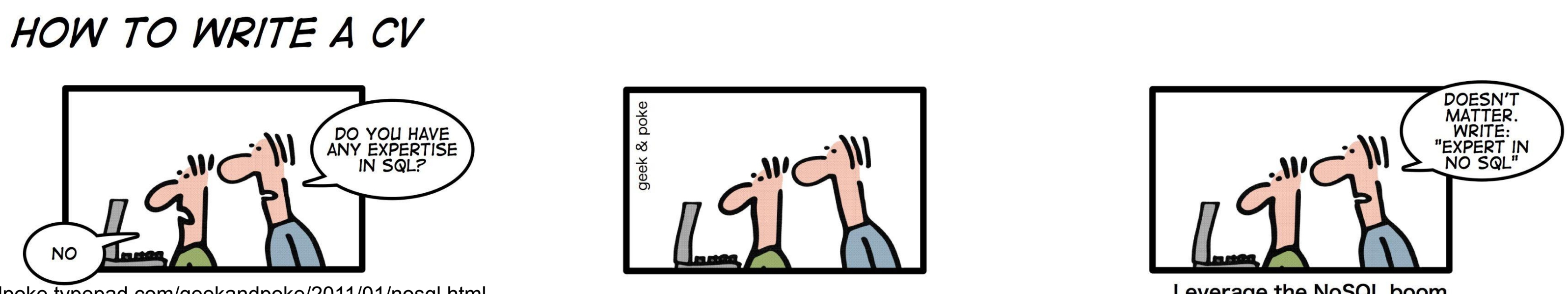
NoSQL DATABASES

NoSQL DB is a term introduced in ~1998 to describe a DB (although relational) not relying on the SQL language

With the rapid increase of large-data management requirements in the late 2000s, the NoSQL term was later (~2009) generalized to define DBMSs with combinations of the following:

- ***massively (horizontally) distributed systems***
- ***data models other than relational***
- ***database language not relying on SQL***

The term NoSQL was more of a catchphrase originally, and is nowadays intended as ***Not-Only SQL DB*** (although Not-Only *Relational* DB would be better…)



https://geekandpoke.typepad.com/geekandpoke/2011/01/nosql.html

A list of common "themes" across most NoSQL DBs can be found:

1. **Flexible data models**
   typically have more flexible schemas compared to the relational model (also allowing for semi- and non-structured data)

2. **Distributed horizontal scalability**
   allow to scale-out horizontally with commodity hardware

3. **Simpler queries**
   as a rule of thumb, simpler queries (e.g. no JOINs) but usually faster than SQL-based

4. **Looser consistency models**
   most often (almost never) not compliant with ACID

BASE

In contrast to the ACID transaction model of RDBs, NoSQL DBs tend to follow other (looser) transaction models, such as (but not only!) ***BASE***

Basically Available → *Individual nodes in the computer network are usually accessible most of the time*

Soft state → *The DB state is not strictly persistent and can change without input (no strict consistency)*

Eventually consistent → *Consistency is ensured only eventually, namely after some long-enough time after the transaction*

Does it mean all NoSQL DBs are AP?

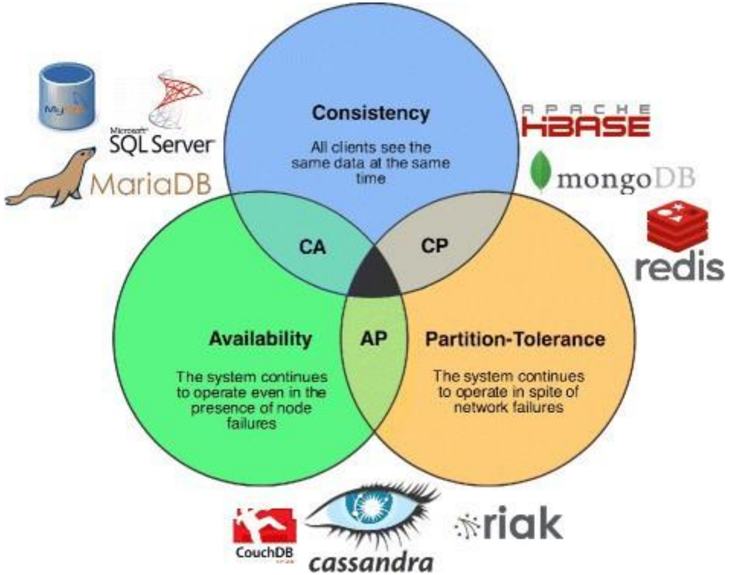
ACID-vs-BASE

While ACID is very hard to implement in NoSQL DBs, it's important to realize that there is *no strictly-defined transaction model* that applies to every NoSQL DBs

- Not all NoSQL DBs favour availability (***AP***) over consistency (***CP***)

- Both ***A*** and ***C*** can be provided in a "gradient" of degrees
  e.g. *highly*-available / *eventually*-available

⇒ It's not a black-or-white situation



KEY-VALUE STORE

Simplest way of storing data → each record is a Value assigned to a Key
(somewhat similar to the idea of a python dictionary, or a C++ map)

key	0xd458c00eb1c3ba9f7e2f

Key-value stores are ***schema-less***

→ data can be stored in arbitrary formats

→ no need for metadata to illustrate the data schema

To read/write/update the value, data is looked-up by key

Going without a schema makes key-value stores very fast and easy to shard across multiple nodes

Often used to store data in memory (not on disk) for very-fast record caching

⇒ *in-memory database!*

KEY-VALUE STORE

exp:cms:01	"Some text"
exp:cms:02	0x1F3A85CA
exp:atlas:99	-1.99e-03

Strings
Hashes
Lists
Sets
Sorted Sets
...

```
redis> GET exp:cms:02
"0x1F3A85CA"
redis> GET exp:atlas:99
"-1.99e-03"
```

```
redis> SET exp:cms:01 "Some text"
OK
redis> SET exp:cms:02 0x1F3A85CA
OK
redis> SET exp:atlas:99 -1.99e-03
OK
```

```
redis> DEL exp:atlas:99
(integer) 1
redis> GET exp:atlas:99
(nil)
```

KEY-VALUE STORE

exp:cms:01	"Some text"
exp:cms:02	0x1F3A85CA
exp:atlas:99	-1.99e-03

- Strings
- Hashes
- Lists
- Sets
- Sorted Sets
- …

Redis

Amazon Dynamo DB

Memcached

…

https://hub.docker.com/_/redis

DOCUMENT DB

Instead of storing key-value pairs as individual records, document-oriented DBs do store entire "documents" of pairs of fields and values

```
{
     "_id" : "books:fantasy:0001"
     "Title" : "Harry Potter and the Philosopher's Stone",
     "Authorinfo" : [
          "Firstname" : "Joanne",
          "Midname" : "Kathleen",
          "Surname" : "Rowling"
          ]
     "ISBN": 0-7475-3269-9,
     "Pages": 233,
     "Description" : "Harry Potter and the Philosopher's Stone is a fantasy novel
     written by British author J. K. Rowling.",
     "Url" : "https://en.wikipedia.org/wiki/Harry_Potter_and_the_Philosopher%27s_Stone",
}
```

Each document is essentially a dictionary (usually a JSON):

- Flexible schema & ideal for semi-structured data
- Multiple documents can be fit into "collections" based on `_id`

DOCUMENT DB

```
{
        "_id" : "books:fantasy:0001"
        "Title" : "Harry Potter and the Philosopher's Stone",
```

```
{
        "_id" : "books:fantasy:0002"
        "Title" : "The Fellowship of the Ring",
        "Authorinfo" : [
```

```
{
        "_id" : "books:tech:0099"
        "Title" : SQL & NoSQL Databases",
        "Authorinfo" : {[
                "Firstname" : "Andreas",
                "Surname" : "Meier"
                ],
                [
                "Firstname" : "Michael",
                "Surname" : "Kaufmann"
                ],}
        "ISBN": 3658245484,
        "Pages": 245,
        "Description" : "Explores relational (SQL) and non-relational (NoSQL) databases.",
        "Url" :
        "https://www.amazon.com/SQL-NoSQL-Databases-Consistency-Architectures/dp/3658245484"
}
```

Library books collection

No JOIN-like relations, but can be queried by the document fields

Overall, a good choice for most applications when no schema is known a-priori

DOCUMENT DB

```
{
        "_id" : "books:fantasy:0001"
        "Title" : "Harry Potter and the Philosopher's Stone",
        "Authorinfo" : [
                "Firstname" : "Joanne",
                "Midname" : "Kathleen",
                "Surname" : "Rowling"
                ]
        "ISBN": 0-7475-3269-9,
        "Pages": 233,
        "Description" : "Harry Potter and the Philosopher's Stone is a fantasy novel
        written by British author J. K. Rowling.",
        "Url" : "https://en.wikipedia.org/wiki/Harry_Potter_and_the_Philosopher%27s_Stone",
}
```

MongoDB

ChouchDB

…

https://hub.docker.com/_/mongo

OTHER NOTABLE DBs

Time Series DB →

Records are data indexed by time-stamp.
Great for data produced by sensors and aggregated/queried based on time windows.
Usually provide functionalities for computing metrics

influxdb

Prometheus

https://hub.docker.com/r/prom/prometheus/

FullText Search Engine →

Records are bodies of text as documents (unstructured data)
Google-like queries based on text return a ranking of "best-fitting" documents

ElasticSearch

Solr

https://hub.docker.com/_/elasticsearch