Large-Scale and Multi-Structured Databases

Key-value Databases Insights

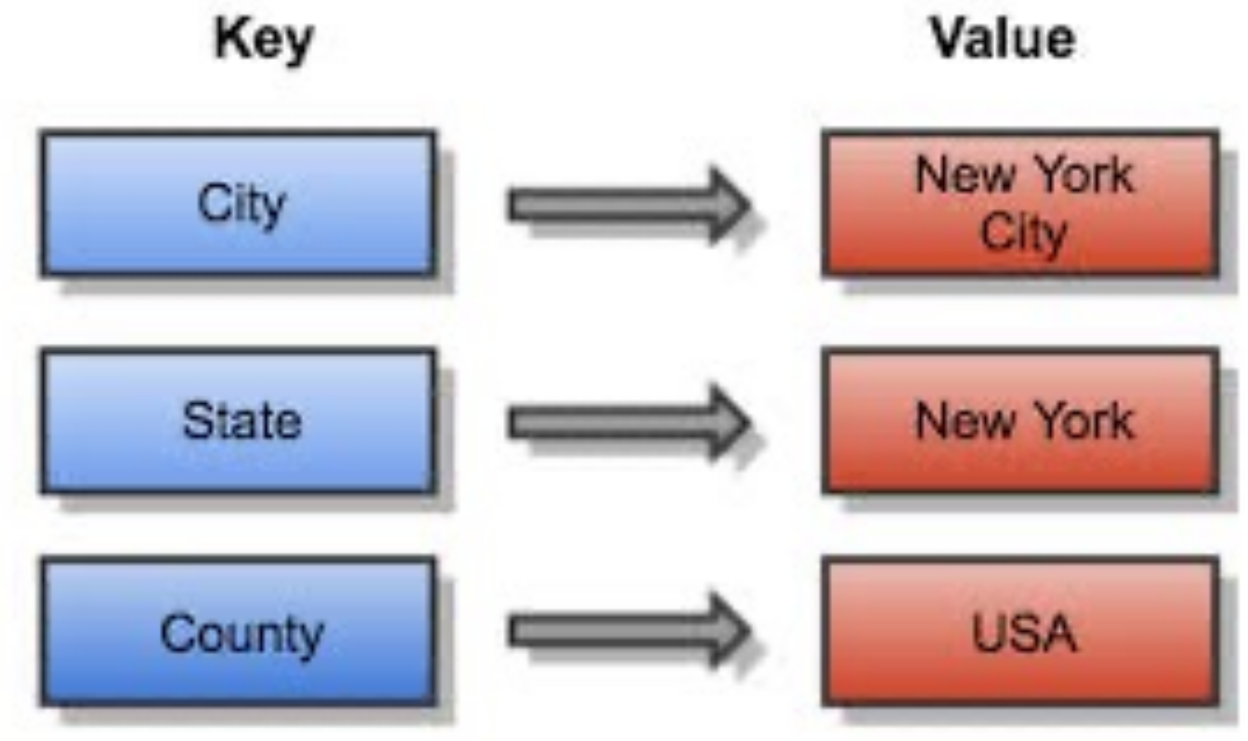
Prof. Pietro Ducange

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Università di Pisa

CROSSLAB Innovation for industry 4.0

Key-Value Databases



From Key to Values

In general, ***values*** may be ***strings***, ***numbers***, ***list*** or other ***complex structures***.

In order to ***identify a value*** in the database, we actually need the "***address***" of the specific location in which this value is stored.

Hash functions as usually used to obtain the address, namely a number, from an ***arbitrary key***.

Usually, hash functions returns values that ***seem*** to be ***random*** values.

Values returned by hash functions may be not unique (***collision problems***!!)

About Keys

At a minimum, a key is specified as a ***string*** of characters

```
Patient:A384J:Allergies
```

Strings representing keys should not be ***too long***.

Long keys will use more memory and key-value databases tend to be ***memory-intensiv***e systems already.

At the same time, avoid keys that are ***too short***. Short keys are more likely to lead to ***conflicts*** in key names

About Values

A value is an ***object***, typically a set of bytes, that has been associated with a key.

Values can be integers, floating-point numbers, strings of characters and even ***complex objects*** such as picture, video, and JSON files.

Key-value implementations will vary in the ***types*** of ***operations*** supported on values.

Limits on the dimension of a single value may be fixed by the different ***frameworks*** for Key-Value databases.

Namespace

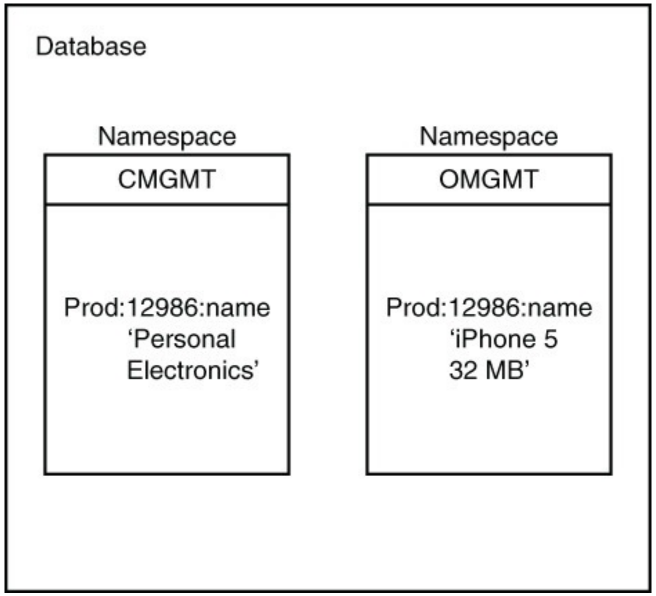
A name space is a ***collection*** of key-value pairs that has ***no duplicate keys***.

It is allowed to have ***duplicate values*** in a namespace.

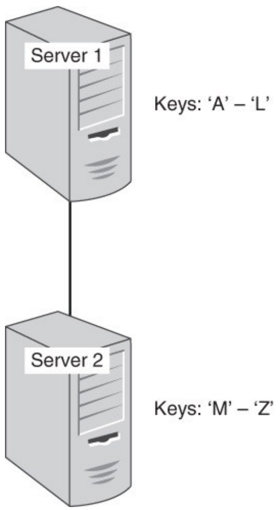
Namespaces enable duplicate keys to exist without causing conflicts by maintaining separate collections of keys.

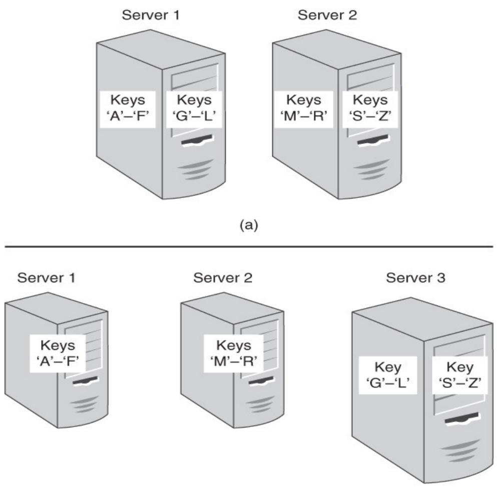
Namespaces are helpful when ***multiple applications*** use the same key-value database.

Namespaces allows to organize data into ***subunits***.



Data Partitioning





(b)

- Subsets of data (partitions or ***shards***) may be handled by the different nodes of a cluster.
- A cluster may contain ***more than one*** partition.
- ***Several strategies*** exists for data partitioning.
- The main objective of partitioning data would be to ***evenly balance***, both write and read loads, among servers.
- If needed, ***additional*** nodes would be ***easily*** added to the cluster and data appropriately ***relocated***.

Partition Keys

- A ***partition key*** identifies the specific partition in which the value has been stored.
- ***Any key*** in a key-value database is used as a ***partition key*** .
- In the previous example, the first letter of a key (string) acts as the value of partition key.
- Usually, hashing functions are adopted for actually ***identifying*** the specific cluster or partition.

Schema-less

We are ***not*** required to ***define*** all the ***keys*** and ***types of values*** we will use prior to adding them to the database.

We may decide to ***change*** how storing the attributes of a specific entity.

Regarding the example in the table, we might decide that storing a customer’s full name in a single value is a bad idea, thus we will separate first and last names.

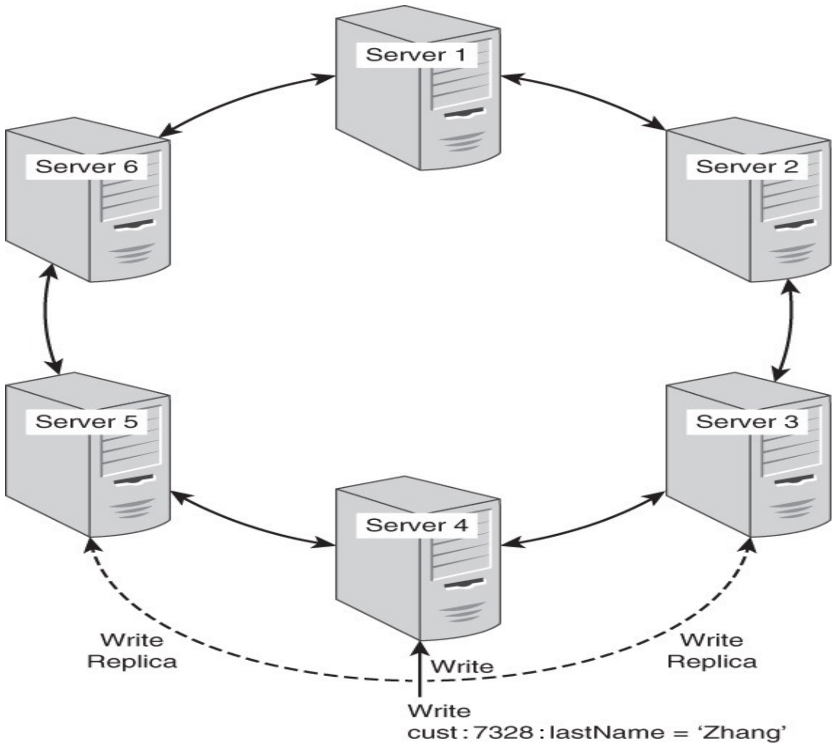
We need to ***update*** the application ***code*** to handle both ways of representing customer names or convert all instances of one form into the other.

Key-Value Database	
Keys	Values
cust:8983:firstName	‘Jane’
cust:8983:lastName	‘Anderson’
cust:8983:fullName	‘Jane Anderson’

About Clusters

- In Key-Value databases, clusters tend to be ***loosely coupled***.
- This means that the server ***are fairly independent*** and complete many functions on their own with ***minimal coordination*** with other servers in the cluster.
- Each server is ***responsible*** for the operations on ***its own partitions*** and routinely ***send messages*** to each other to indicate they are still functioning.
- When a node ***fails***, the other nodes in the cluster can respond by ***taking over the work*** of that node.

Rings: logical structures for organizing partitions



- Let consider a ***hashing function*** that generates a number from a key and calculates the modulo.
- Whenever a piece of data is written to a server, it is also written to the two servers linked to the original server (***high availability***).
- If Server 4 fails, both Server 3 and Server 5 could respond to read/write requests for the data on Server 4.
- When Server 4 is ***back online***, Servers 3 and 5 ***can update it***

Replication

High availability is ensured by using ***replication***, namely saving ***multiple copies*** of the data in the nodes of a cluster.

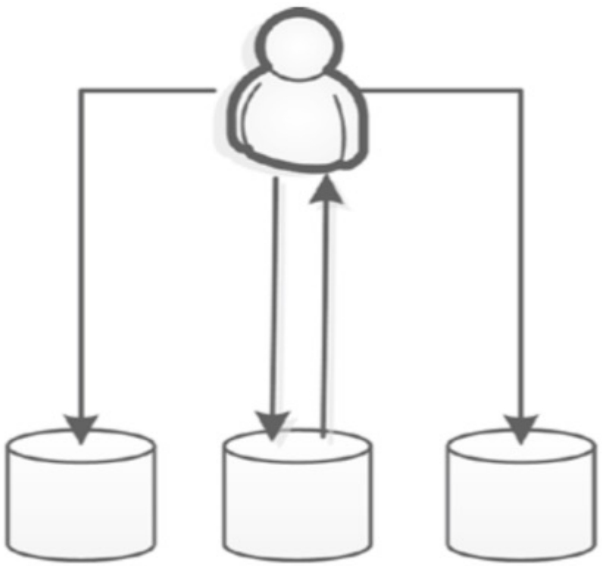
The ***number*** of data ***replicas*** is often a ***parameter*** to set.

The ***higher*** number of ***replicas***, the ***less*** likely we will ***loose*** data, the ***lower*** the ***performance*** of the systems (in terms of response time).

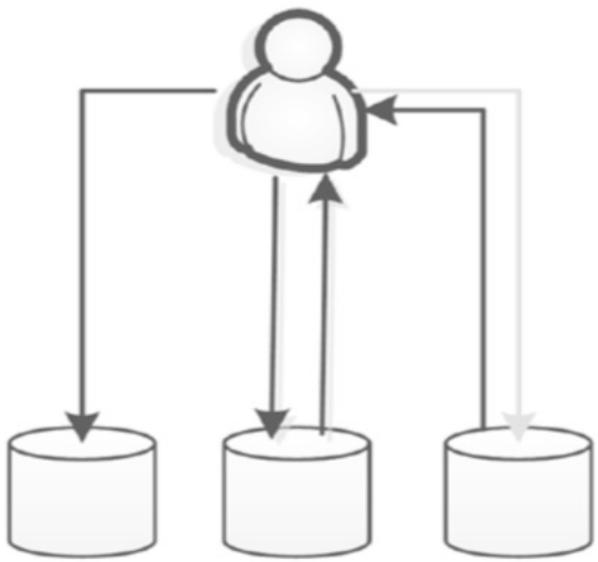
The ***lower*** the number of ***replicas***, the ***better*** the ***performance*** of the systems, the ***higher*** the probability of ***loosing data***.

A ***low*** number of replicas may be used whenever ***data*** is easily ***regenerated*** and ***reloaded***.

Write/Read operations with Replicas



N=3 W=3 R=1
Slow writes, fast reads, consistent
There will be 3 copies of the data.
A write request only returns when all 3 have written to disk.
A read request only needs to read one version.



N=3 W=2 R=2
Faster writes, still consistent (quorum assembly)
There will be 3 copies of the data.
A write request returns when 2 copies are written – the other can happen later.
A read request reads 2 copies make sure it has the latest version.

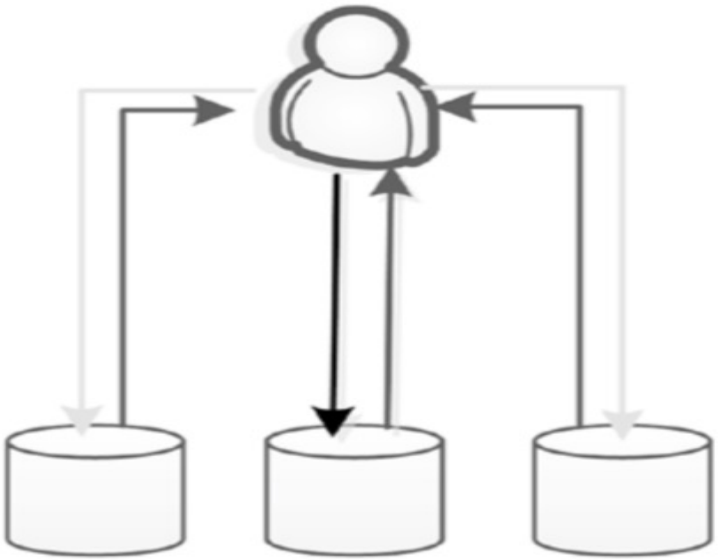
N = # of replicas

W = # of copies to be written before the write can complete

R = # of copies to be read for reading a data record

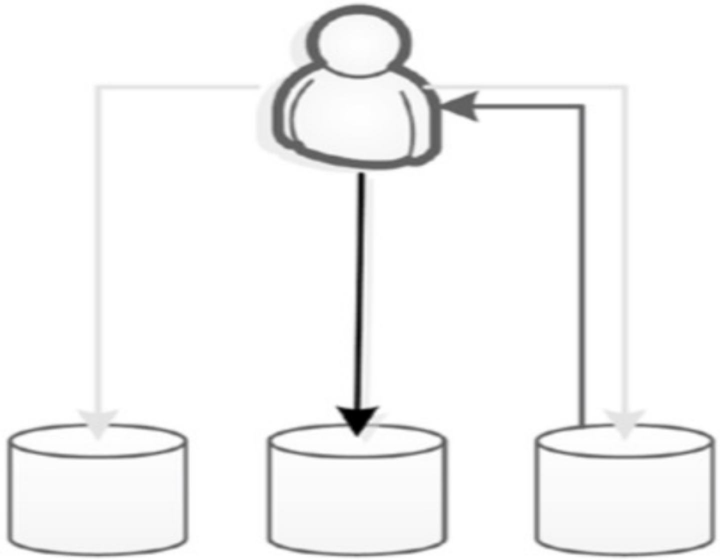
Image extracted from "Guy Harrison, Next Generation Databases, Apress, 2015"

Write/Read operations with Replicas



N=3 W=1 R=N
Fastest write, slow but consistent reads

There will be 3 copies of the data.
A write request returns once the first copy is written – the other 2 can happen later.
A read request reads all copies to make sure it gets the latest version.
Data might be lost if a node fails before the second write.



N=3 W=1 R=1
Fast, but not consistent

There will be 3 copies of the data.
A write request returns once the first copy is written – the other 2 can happen later.
A read request reads a single version only: it might not get the latest copy.
Data might be lost if a node fails before the second write.

Image extracted from "Guy Harrison, Next Generation Databases, Apress, 2015"

Hash Mapping

Key	Hash Value
customer:1982737: firstName	e135e850b892348a4e516cfcb385eba3bfb6d209
customer:1982737: lastName	f584667c5938571996379f256b8c82d2f5e0f62f
customer:1982737: shippingAddress	d891f26dcdb3136ea76092b1a70bc324c424ae1e
customer:1982737: shippingCity	33522192da50ea66bfc05b74d1315778b6369ec5
customer:1982737: shippingState	239ba0b4c437368ef2b16ecf58c62b5e6409722f
customer:1982737: shippingZip	814f3b2281e49941e1e7a03b223da28a8e0762ff

In the example above, the Hash Value is a number in ***hexadecimal format.***

Hash Function Properties

One of the important characteristics of hash algorithms is that ***even small changes*** in the input can lead to ***large changes*** in the output.

Hash functions are generally designed to ***distribute*** inputs ***evenly*** over the set of all possible outputs. The output space can be quite large

This is especially useful when ***hashing keys***.

No matter how similar your keys are, they are evenly distributed across the range of possible output values.

Hash Function: An Example of Load Distribution

Assume we have a cluster of ***16 nodes*** and each node is responsible for one partition.

The key 'cust:8983:firstName' has a hash value of

`4b2cf78c7ed41fe19625d5f4e5e3eab20b064c24`

and would be assigned to partition 4.

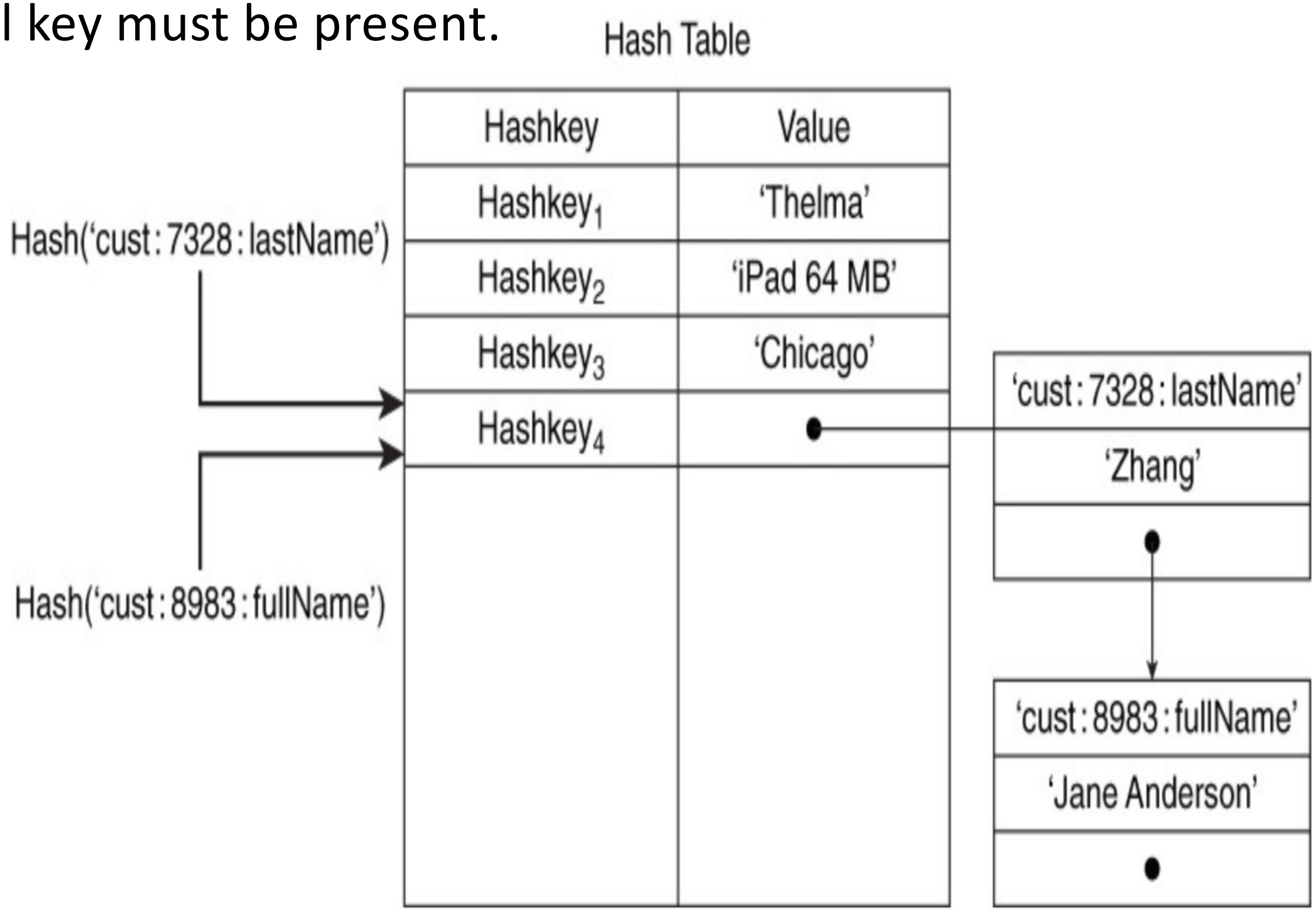
The key 'cust:8983:lastName' has a hash value of

`c0017bec2624f736b774efdc61c97f79446fc74f`

would be assigned to node 12 (c is the hexadecimal digit for the base-10 number 12).

Collision Resolution Strategy

From a logical point of view, the table that projects a hashed key to the corresponding value may include a list of values. In each block of the list, also the original key must be present.



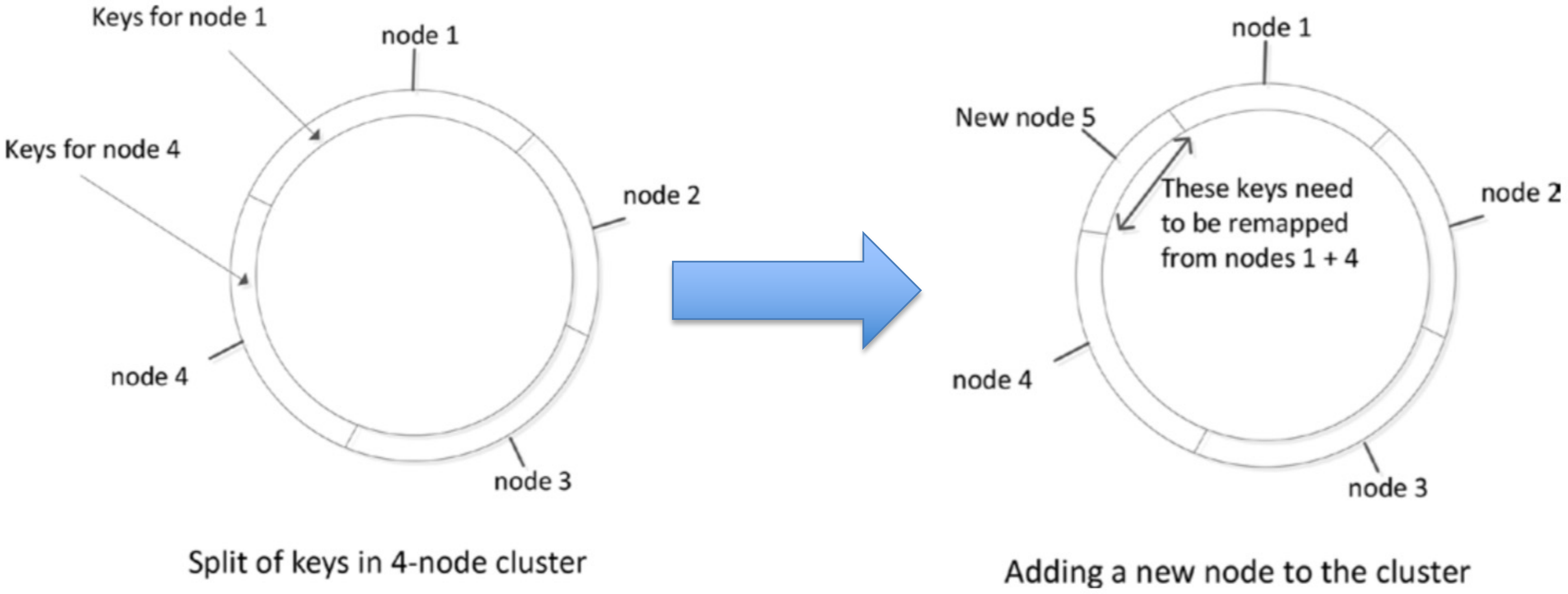
Consistent Hashing (I)

Use Case: adding or removing a node, even for a short time period.

Problem: we need to change the ***hashing function*** and to re-locate all the data among the servers of the cluster.

Solution: to exploit the ring structure and a consistent ***hashing function*** that allows us to remap only the keys mapped to the neighbors of the new node.

Consistent Hashing (II)

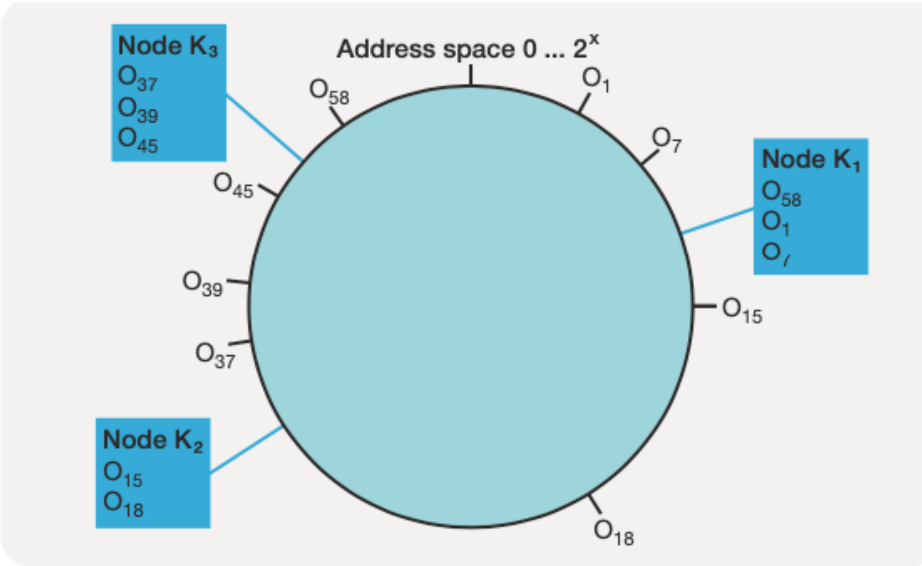


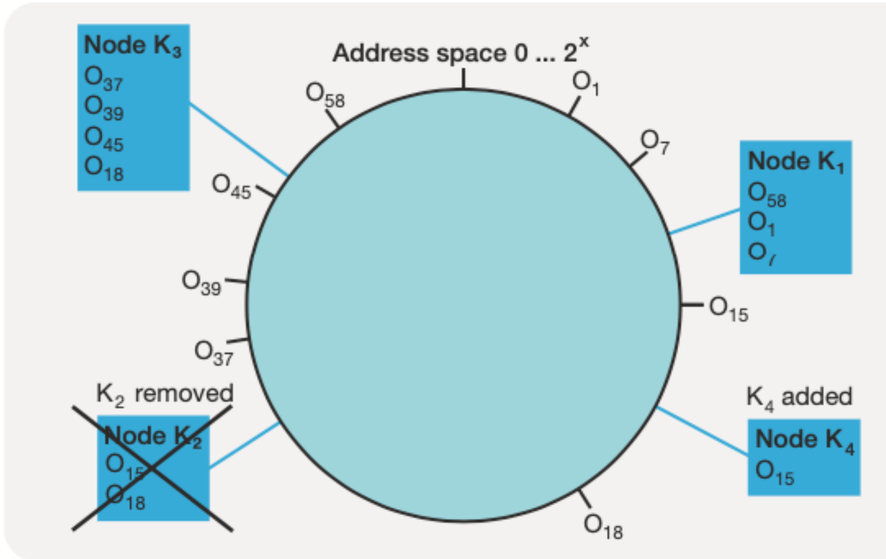
Consistent hashing ensures a good load balance among servers

Consistent Hashing (III)

The same ***hashing*** function is applied to both the ***keys*** and the ***server/partition ID/Address/Name.***

The hash values must be in the ***same range***, for example hexadecimal on 32 bit representation





The actual server (Node) k_j associated to a specific key (object) o_i is its ***successor*** in the hashing/address space.

Images extracted from "Andreas Meier, Michael Kaufmann , SQL & NoSQL databases : models, languages, consistency options and architectures for big data management, 2019"

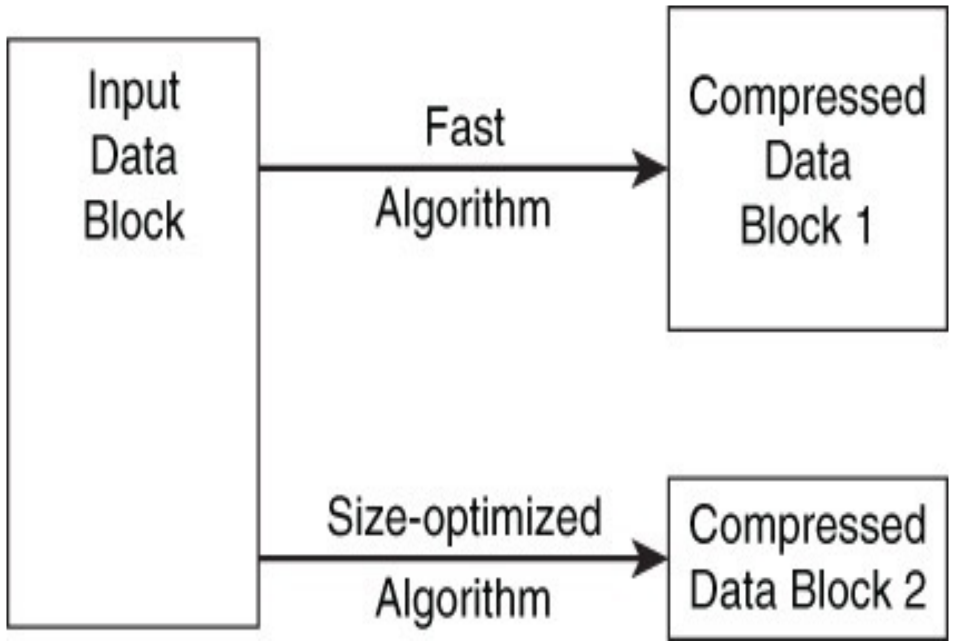
Data Compression for KV Databases

Key-value databases are ***memory intensive.***

Operating systems can exploit ***virtual memory*** management, but that entails writing data to disk or flash storage.

Reading from and writing to ***disk*** is significantly ***slower*** than reading from RAM memory.

One way to optimize memory and persistent storage is to use data ***compression techniques***.



Look for compression algorithms that ensure a ***trade-off*** between the ***speed*** of compression/decompression and the ***size*** of the compressed data.

Using Key-Value Databases

If ***data organization*** and ***management*** is more important than the performances, classical relational databases are more suitable rather than key-value databases.

However, if we are more interested to the ***performances*** (high availability, short response time, etc.) and/or the data model is not ***too much complicated*** (no hierarchical organization, limited number of relationships) we may use key-values databases.

Indeed, key-value stores are really ***simple*** and ***easy*** to handle, data can be modeled in a less complicated manner than in RDBMS

From RBDMS to Key-Value Store (I)

Let consider the following data structures:

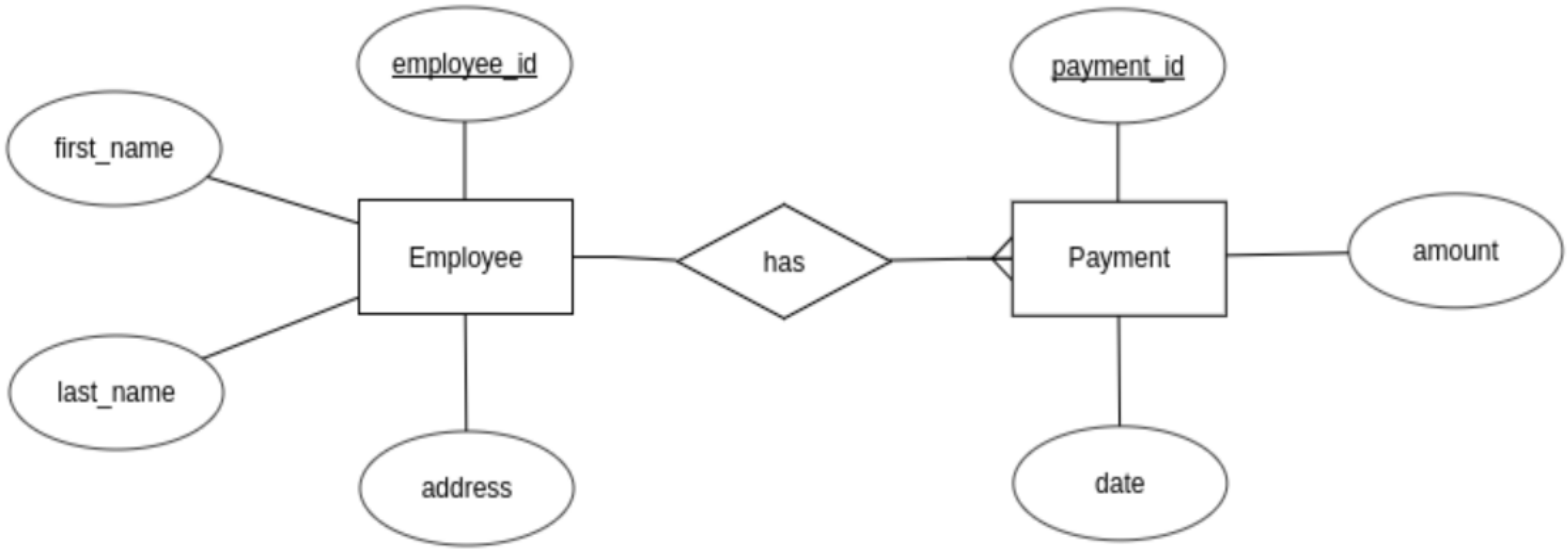


Image extracted from: https://medium.com/@wishmithasmendis/from-rdbms-to-key-value-store-data-modeling-techniques-a2874906bc46

From RBDMS to Key-Value Store (II)

In a relational model we can define the following tables. The one-to-many relationship is handled by using a ***foreign key***.

employee_id	first_name	last_name	address
1	John	Doe	New York
2	Benjamin	Button	Chicago
3	Mycroft	Holmes	London

FOREIGN KEY

payment_id	employee_id	amount	date
1	1	50,000	01/12/2017
2	1	20,000	01/13/2017
3	2	75,000	01/14/2017
4	3	40,000	01/15/2017
5	3	20,000	01/17/2017
6	3	25,000	01/18/2017

Image extracted from: https://medium.com/@wishmithasmendis/from-rdbms-to-key-value-store-data-modeling-techniques-a2874906bc46

From RBDMS to Key-Value Store (II)

Now, we want to ***translate*** the data modelling from a relational model to a key-value model.

Take in mind that keys ***embed*** information regarding ***Entity Name***, ***Entity Identifie***r and ***Entity Attributes.***

Thus, we can translate the ***Employees table*** as follows:

employee_id	first_name	last_name	address
1	John	Doe	New York
2	Benjamin	Button	Chicago
3	Mycroft	Holmes	London

```
employee:$employee_id:$attribute_name = $value

employee:1:first_name = "John"
employee:1:last_name = "Doe"
employee:1:address = "New York"

employee:2:first_name = "Benjamin"
employee:2:last_name = "Button"
employee:2:address = "Chicago"

employee:3:first_name = "Mycroft"
employee:3:last_name = "Holmes"
employee:3:address = "London"
```

From RBDMS to Key-Value Store (III)

As further step, we have to translate the ***Payment table*** and to manage the one-to-many relationship.

In this case, we can define the following key-value configuration:

payment:$payment_id:$employee_id:$attribute_name = $value

The Payment table con be translated as follows:

payment_id	employee_id	amount	date
1	1	50,000	01/12/2017
2	1	20,000	01/13/2017
3	2	75,000	01/14/2017
4	3	40,000	01/15/2017
5	3	20,000	01/17/2017
6	3	25,000	01/18/2017

```
payment:1:1:amount = "50000"
payment:1:1:date = "01/12/2017"

payment:2:1:amount = "20000"
payment:2:1:date = "01/13/2017"

payment:3:2:amount = "75000"
payment:3:2:date = "01/14/2017"

payment:4:3:amount = "40000"
payment:4:3:date = "01/15/2017"

payment:5:3:amount = "20000"
payment:5:3:date = "01/17/2017"

payment:6:3:amount = "25000"
payment:6:3:date = "01/18/2017"
```

From RBDMS to Key-Value Store (IV)

At the end of the translation process, data will be organized in a ***unique bucket*** as follows:

```
employee:1:first_name = "John"
employee:1:last_name = "Doe"
employee:1:address = "New York"

employee:2:first_name = "Benjami
employee:2:last_name = "Button"
employee:2:address = "Chicago"

employee:3:first_name = "Mycroft'
employee:3:last_name = "Holmes"
employee:3:address = "London"

payment:1:1:amount = "50000"
payment:1:1:date = "01/12/2017"

payment:2:1:amount = "20000"
payment:2:1:date = "01/13/2017"

payment:3:2:amount = "75000"
payment:3:2:date = "01/14/2017"

payment:4:3:amount = "40000"
payment:4:3:date = "01/15/2017"

payment:5:3:amount = "20000"
payment:5:3:date = "01/17/2017"

payment:6:3:amount = "25000"
payment:6:3:date = "01/18/2017"
```

Image extracted from: https://medium.com/@wishmithasmendis/from-rdbms-to-key-value-store-data-modeling-techniques-a2874906bc46

Suggested Readings

Chapter 4 of the book *“Dan Sullivan, NoSQL For Mere Mortals, Addison-Wesley, 2015”*

Chapter 3 of the book *“Guy Harrison, Next Generation Databases, Apress, 2015”*

Chapter 5.2.2 of the book *“Andreas Meier, Michael Kaufmann, SQL & NoSQL databases : models, languages, consistency options and architectures for big data management, 2019”*

Web pages accessible with the links spread along the slides.

Images

If not specified, the images shown in this lecture have been extracted from:

“Dan Sullivan, NoSQL For Mere Mortals, Addison-Wesley, 2015”

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

UNIVERSITÀ DI PISA

CROSSLAB Innovation for industry 4.0