Large-Scale and Multi-Structured Databases

Key-value Databases Introduction

Prof Pietro Ducange

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

UNIVERSITÀ DI PISA

CROSSLAB Innovation for industry 4.0

From Arrays…

1	True
2	True
3	False
4	True
5	False
6	False
7	False
8	True
9	False
10	True

- An array is an ***ordered*** list of ***values***.
- Each ***value*** in the array is associated with an integer ***index***.
- The ***values*** are all the ***same type***

Limitations:

- The ***index*** can only be an ***integer***.
- The ***values*** must all have the ***same type***.

…to Associative Arrays…

‘Pi’	3.14
‘CapitalFrance’	‘Paris’
17234	34468
‘Foo’	‘Bar’
‘Start_Value’	1

- Associative arrays ***generalize*** the idea of an ***ordered list***.
- The ***index*** can be a ***generic identifier*** such as a string, an integer an so on.
- The ***values*** can be of ***different types***.

…to Key-Value Databases

- Data are ***persistently*** stored.
- There is ***no*** need for ***tables*** if we do not need groups of related attributes.
- The unique requirement is that ***each value*** has a ***unique identifier*** in the form of the key.
- Keys must be ***unique*** within the ***namespace*** defined in each specific ***bucket***

Database

Bucket 1	
'Foo1'	'Bar'
'Foo2'	'Bar2'
'Foo3'	'Bar7'

Bucket 2	
'Foo1'	'Baz'
'Foo4'	'Baz3'
'Foo6'	'Baz2'

Bucket 3	
'Foo1'	'Bar7'
'Foo4'	'Baz3'
'Foo7'	'Baz9'

Essential Features of Key-Value Databases

- Simplicity
- Speed
- Scalability

*Developers tend to use key-value databases when **ease** of **storage** and* **retrieval** *are **more important** than organizing data into more complex data structures, such as **tables** or **networks**.*

Simplicity

- If we do not need all the ***features of the relational models*** (joining tables or running queries about multiple entities in the database), we may exploit key-value databases

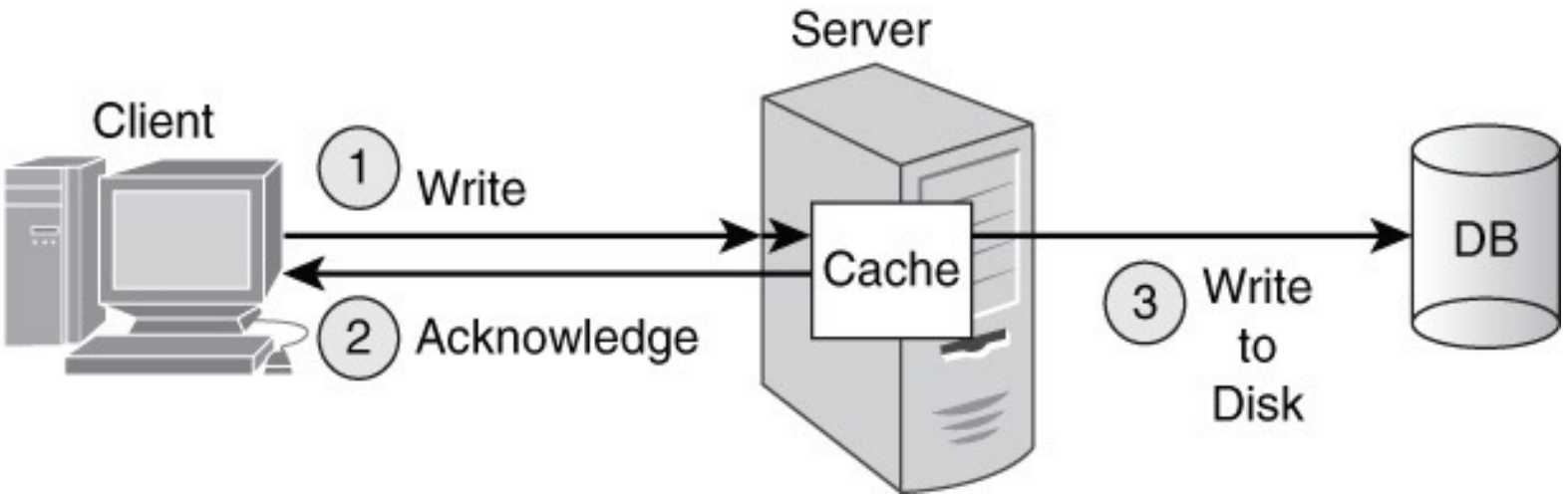
- We do ***not*** have to ***define*** a database ***schema*** nor to define data ***types*** for each ***attribute***.

- We can simply modify the code of our program if we need to handle ***new attributes***, ***without*** the necessity of ***modifying*** the database.

- Key-value databases are ***flexible*** and ***forgiving***: we can make mistakes assigning a wrong type of data to an attribute or use different type of data for a specific attribute.

Speed

Using the scheme below, the key-value database can ***write*** the recently updated value to disk, while the user program is doing something else.



Read operations can be ***faster*** if data is stored ***in memory***. This is the motivation for using a ***cache***: the first time the program fetches the data, it will need to read from the disk but after that the results can be saved in memory.

Scalability

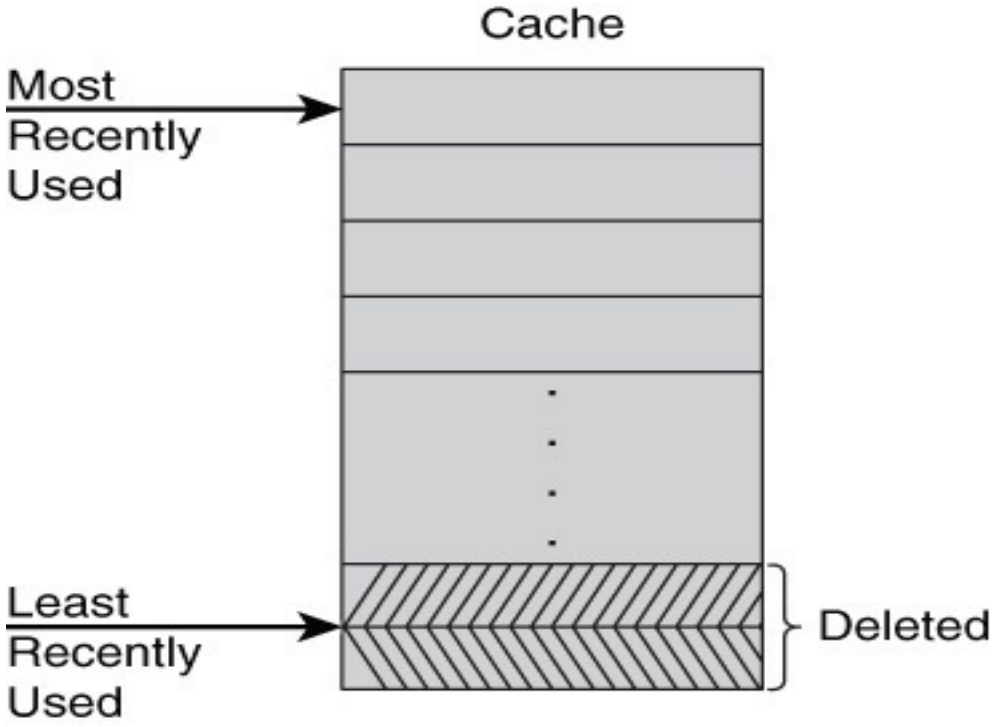
Definition: the scalability is the capability to ***add*** or ***remove*** servers from a cluster of servers as needed ***to accommodate*** the ***load*** on the system.

When dealing with ***NoSQL databases***, employed in web and other large-scale applications, it particularly important to correctly handle both ***reads*** and ***writes*** when the system has to be scaled.

In particular, in the framework of ***key-value database***, two main approaches are adopted, namely ***Master-Slave Replication*** and ***Masterless Replication***.

The RAM is not infinite!

When the key-value database ***uses all the RAM memory*** allocated to it, the database will need to ***free*** some records.



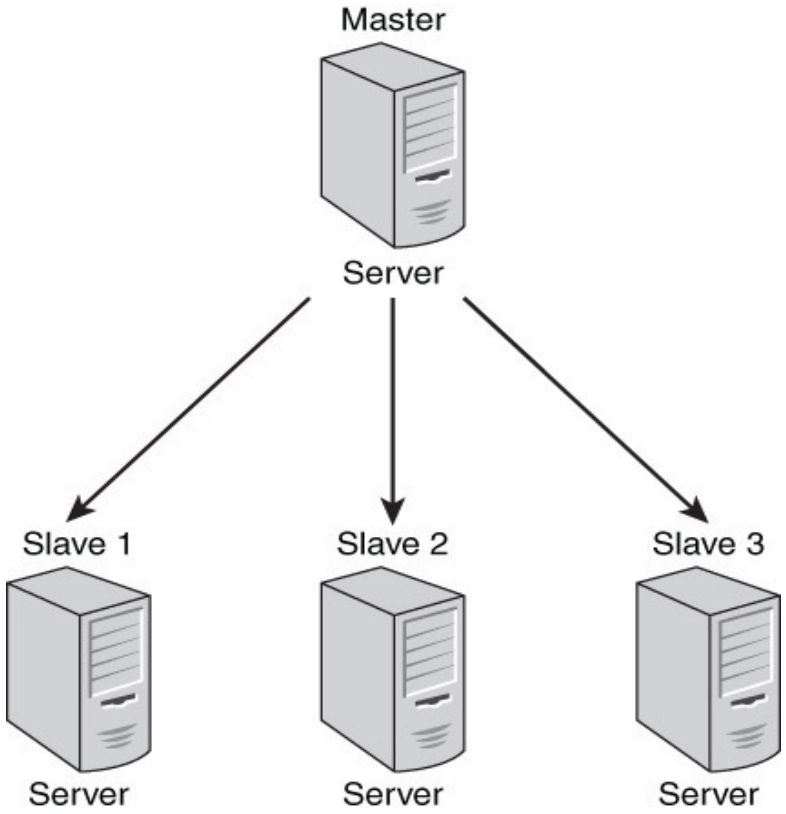
Several algorithms exist for selecting the records to remove from the RAM.

The ***least recently used*** (LRU) is one of the most used algorithm.

Data that has not been read or written as recently as other data can be removed from the cache.

Use case: key-value database used to store items in customers' online ***carts***

Master-Slave Replication



- The ***master*** is a server that accepts ***write*** and ***read*** requests
- The ***master*** is in charge of ***collecting*** all the ***modification*** of the data and ***updating*** data to all the slaves
- The ***slaves*** may only respond to ***read requests***
- This architecture is particularly ***suitable*** for applications with a ***growing demand for read operations***

Master-Slave Replication: Pros and Cons

Pros:

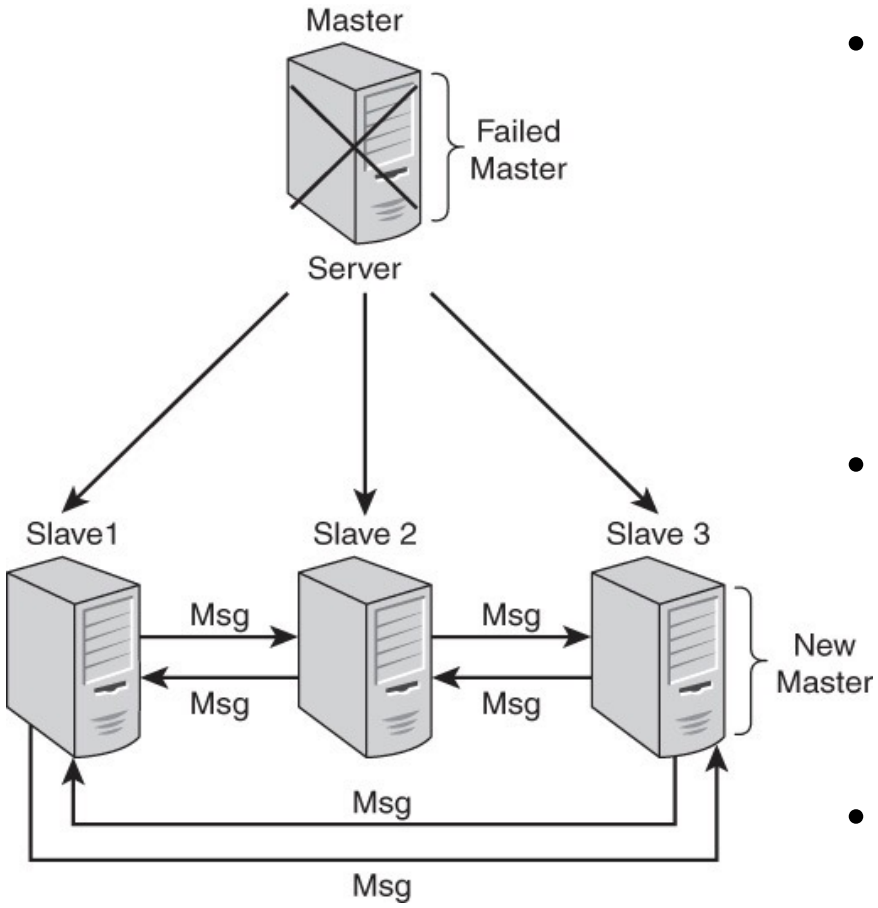
- Very simple architecture: only the master communicates with other servers.

- There is ***no need to coordinate*** write operations or resolve conflicts between multiple servers accepting writes.

Cons:

- if the master fails, the cluster cannot accept writes.
- In this case, the entire system fails or at least loses a critical capacity, such as accepting writes.

The Higher the Complexity, the Better the Availability



- ***Each*** single slave ***server*** may send a simple ***message*** to ask a random server in the cluster if it is still active and if it received recently messages from the master.
- If a number of slave servers ***do not receive a message*** from the master within some period of time, the slaves may determine the master has failed.
- The slaves initiate a protocol to ***promote*** one of the slaves as a ***new master.***

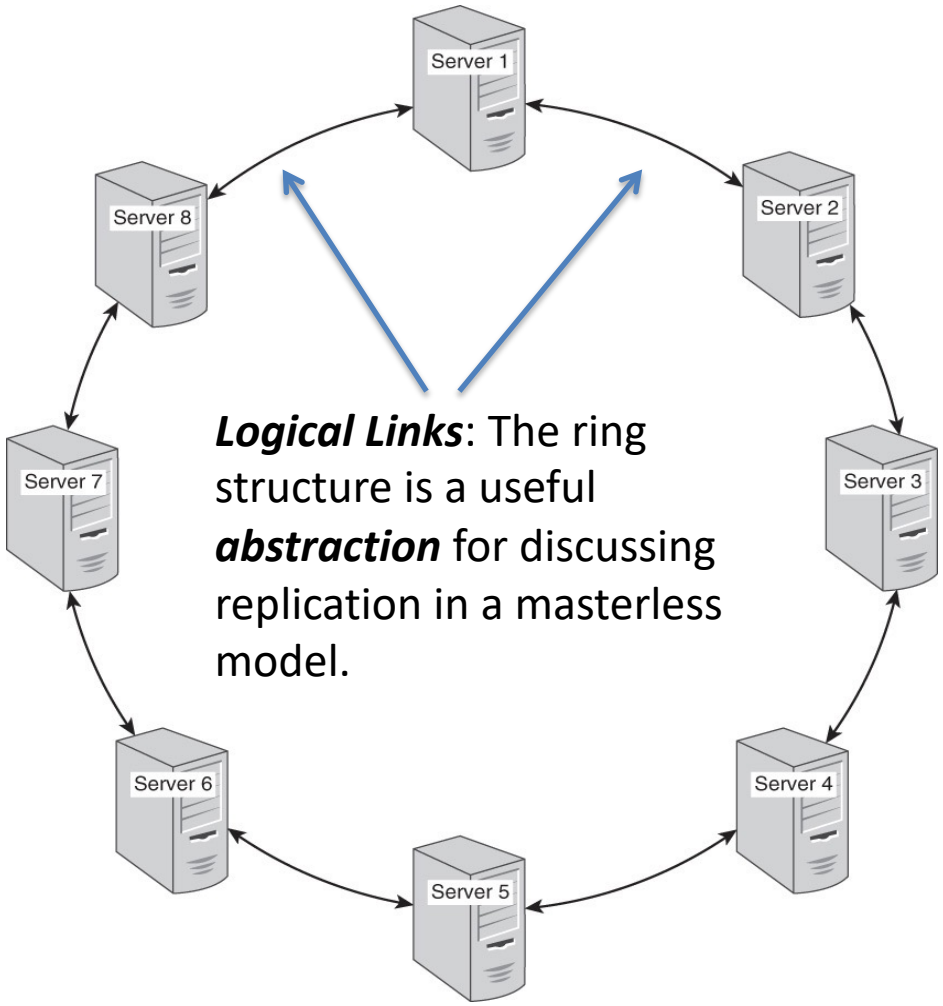
Masterless Replication

Use case: applications in which the number of writes may increase and achieve peaks! Example: the website which sells ticket of concerts.

The ***master-slave replication*** architecture ***will suffer*** an increasing number of write requests (only the master accept writes and all the updates must be sent to the slaves).

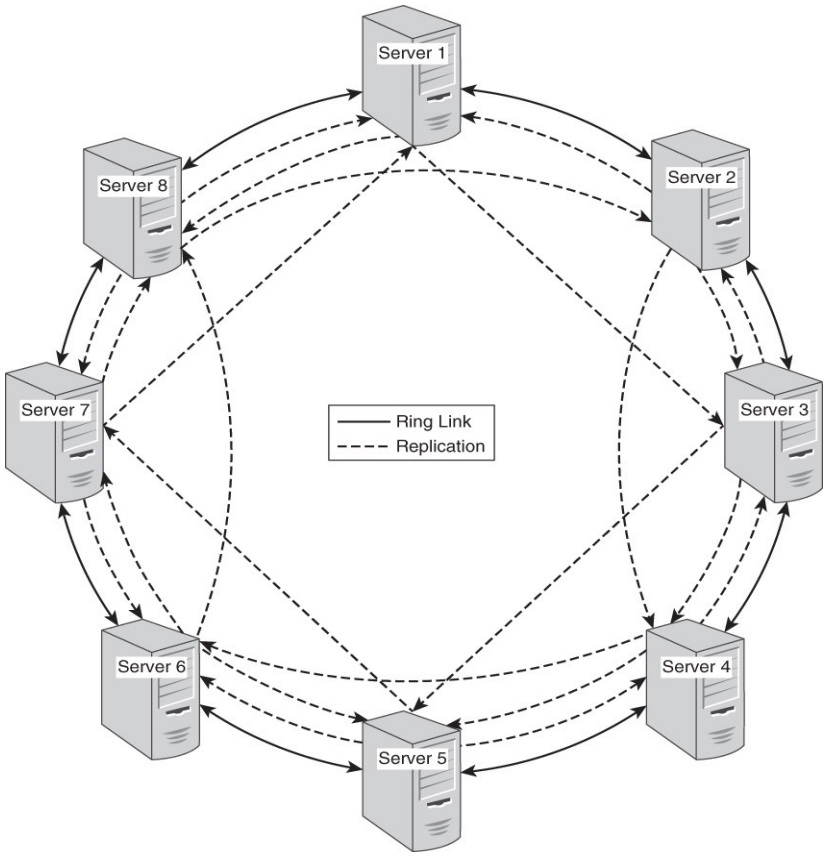
Solution: a replication model in a cluster of servers where all nodes may accept write requests.

Masterless Replication: Architecture



- There is ***not*** a single server that has the ***master copy*** of updated data.
- There is not a single server that can copy its data to all other servers.
- Each ***node*** is in ***charge*** of helping its ***neighbors***.

Handling Replication: An Example

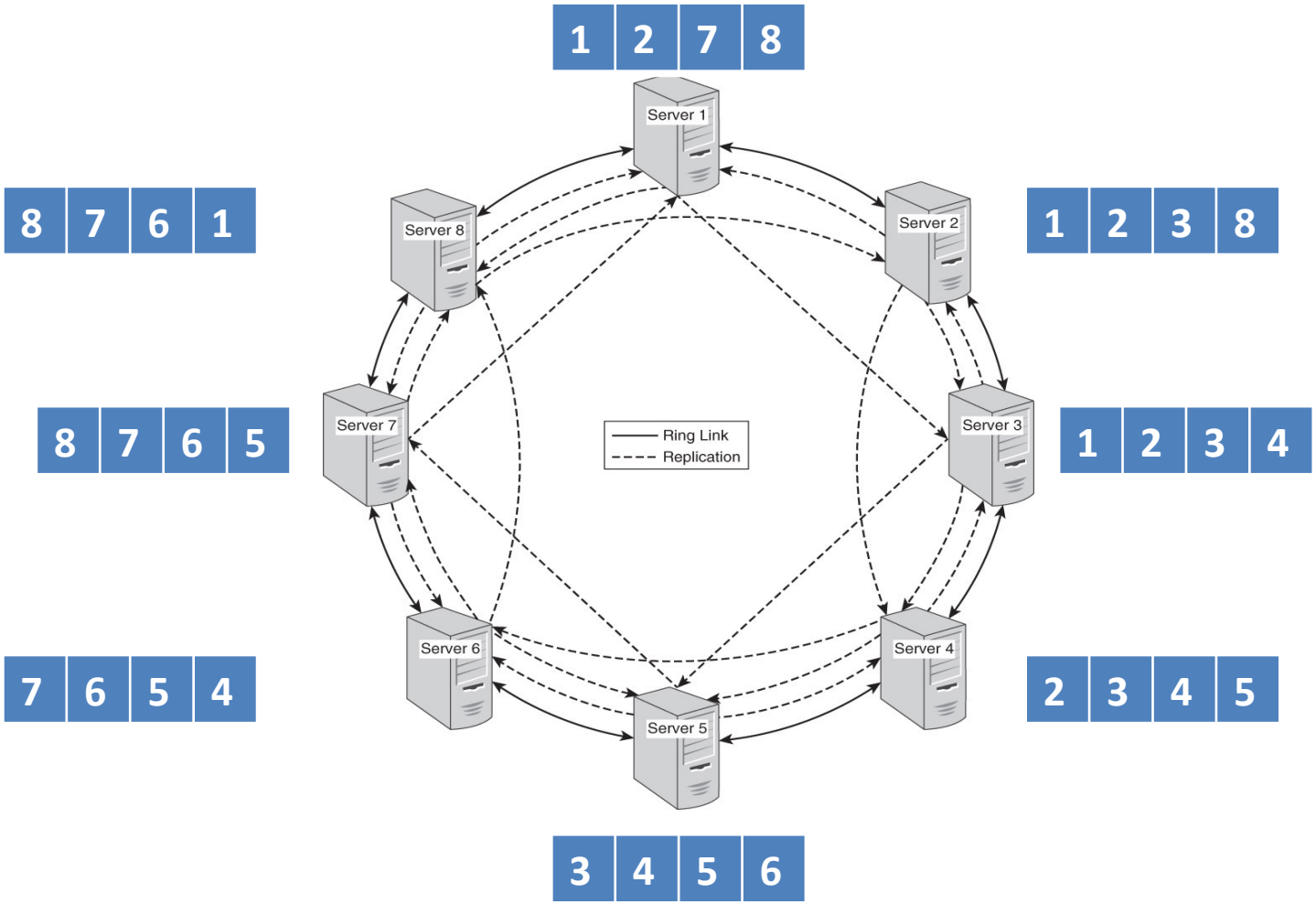


Each time there is a write operation to one of the servers, it replicates that change to the three other servers holding its replica (we consider 4 data replicas).

Each server replicates to its two neighbors and to the server two links ahead

The ring structure is an ***abstraction***. Usually, in a data center, all the servers are connected to a single network hub and are ***able to communicate with each other***.

Handling Replication: An Example



How to Construct a Key: Some Guidelines

In relational model, the ***primary key*** is used to retrieve a row in a table and it must be ***unique***.

Similarly, a ***key***, in a key-value database is used to retrieve a value and must be ***unique*** in a specific namespace.

When designing ***relational databases***, using ***meaningless*** primary keys is a good practice.

In the framework of ***key-value databases,*** adopting meaningful keys is ***preferable***.

The key is the unique mean to retrieve a value!

Meaningless PK in Relational DB: An Example

Suppose that the surname attribute, followed by _stateCode, is a PK in a table.

If we specify the entry of Katherine Smith who lives in Texas, we should use SMITH_TX as value for the PK attribute in the table.

If Katherine moves to NY, now the PK entry should be changed, but it would violate the principle that ***primary keys are immutable.***

Moreover, but we would have to *update all references* to that key in other tables.

Thus, it is important to select PK without specific meanings.

Meaningful keys in KV DB (I)

In key-value there are no tables, nor columns, nor pre-defined attributes.

A specific value is associated only to its key.

Let's suppose to manage a ***cart application*** using a key-value database.

If we use ***meaningless keys***, we access the cart as follows (like in a relational model, without any meaning):

Cart[12387]= "SK AK231312"

We refer just to a product of userID 12387.

Meaningful keys in KV DB (II)

We may define add ***more interpretable*** namespace such as in the following:

CustName[12387] = “Pietro Ducange”.

Now we have just the information on the customer name.

We need to define additional namespaces for each attribute, thus moving back to the relational models (defining a scheme).

We can build keys that ***embed*** information regarding ***Entity Name***, ***Entity Identifie***r and ***Entity Attributes***

Meaningful keys: an Example

Key
customer:1982737: firstName
customer:1982737: lastName
customer:1982737: shippingAddress
customer:1982737: shippingCity
customer:1982737: shippingState
customer:1982737: shippingZip

For each key, we may store in the database a specific value.

For handling customer information, we can use just one namespace (table).

We may also exploit easily replica and sharding.

The general rule for defining keys is the following (single Entity, no relationships):

EntityName : Entity id: EntityAttribute

From Key to Values

In general, ***keys*** 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 are usually used to obtain the address.
The hash functions returns 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***!!)

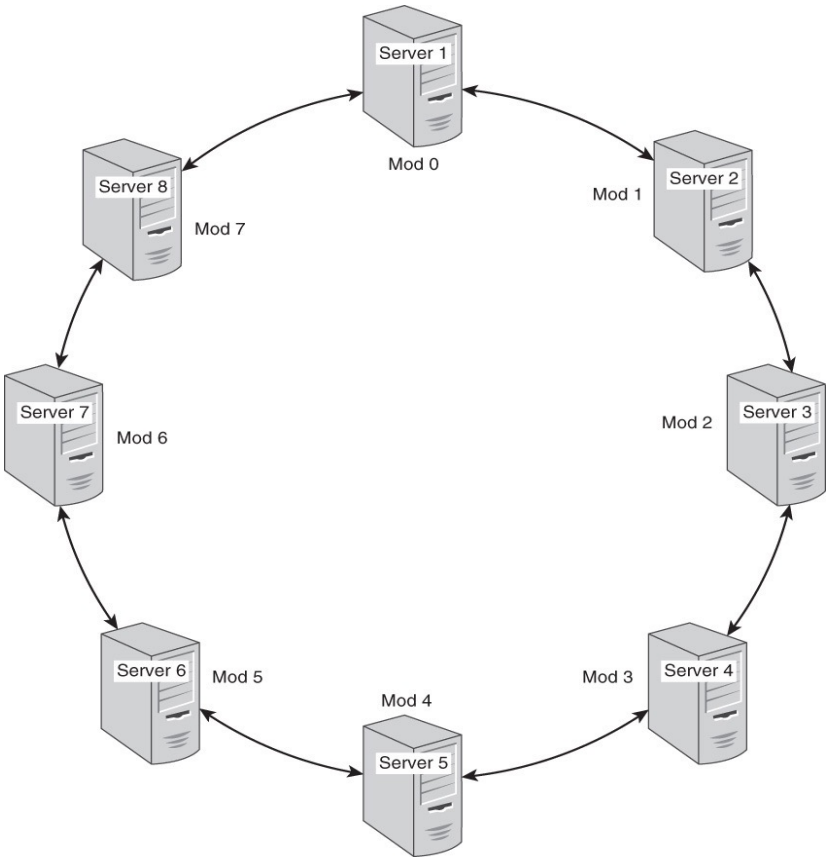
Hash Mapping

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.***

The ***SHA-1 hash*** function has been used in the example above.

Hashing for Selecting Servers



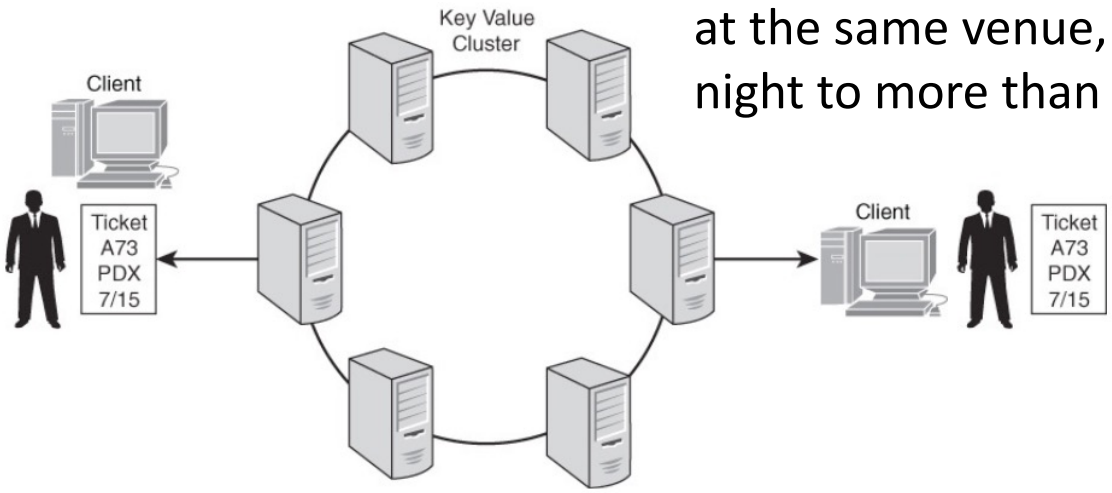
We would like to ***balance*** the write loads among the servers in a masterless replication architecture.

For the sake of simplicity, let consider the ID of a specific server as the address to identify.

If we ***divide*** the Hash Value by the number of servers, we can collect the ***reminder***, namely the ***modulus***.

The modulo span from 0 to 7.
Each of the 8 servers can be ***assigned*** a number (the address) between 1 and 8.

Hashing for Selecting Servers: An Example



Problem: to avoid to sell tickets to the same seat, at the same venue, in the same city, on the same night to more than one person.

In the picture, two fans want to purchase set A73 at the Civic Center in Portland, Oregon, on July 15.

We can use the following key for the specific ticket: `TICK:A73:CivCen:PDX:0715`

Anyone trying to purchase that same seat on the same day would generate the same key, that will be transformed by the hashing+modulo function in the same Server Address->There is no chance for another server to sell that seat!

No Typing for the Values

Key-value databases do not expect us to specify types for the values we want to store.

Examples for storing the address of a customer:

- String:`'1232 NE River Ave, St. Louis, MO'`
- List of strings: `('1232 NE River Ave', 'St. Louis',’MO')`
- JSON format: `{'Street:' : '1232 NE River Ave', 'City' : 'St. Louis',: 'State' : 'MO' }`

Some Considerations

- Consider the ***design characteristics*** of the key-value database we choose to use.
- Consult the ***documentation*** for limitations on keys and values.
- When choosing a key-value database, consider the ***trade-off of various features.***
- One key-value database might offer ***ACID transactions*** but set ***limit*** on the dimension of keys and values.
- Another key-value data store might allow for large values but limit keys to numbers or strings.
- Our ***application requirements*** should be considered when weighting the ***advantages*** and ***disadvantages*** of different database systems.

Operations Allowed in Key-Value Databases

In the following, we list the operations allowed in a Key-Value database:

- To ***retrieve*** a value by key
- To ***set*** a value by key
- To ***delete*** values by key

Pay attention: If we want to do more, such as search for all addresses in which the city is "St. Louis," we will have to do that with an application program.

Complex Queries with KV Database

```
appData[cust:9877:address] = '1232 NE River Ave, St.
  Louis, MO'
```

```
define findCustomerWithCity(p_startID, p_endID, p_City):
    begin
    # first, create an empty list variable to hold all
    # addresses that match the city name
    returnList = ();
    # loop through a range of identifiers and build keys
    # to look up address values then test each address
    # using the inString function to see if the city name
    # passed in the p_City parameter is in the address
    # string. If it is, add it to the list of  addresses
    # to return
    for id in p_startID to p_endID:
         address = appData['cust:' + id + ':address'];
         if inString(p_City, Address):
            addToList(Address,returnList );
    # after checking all addresses in the ranges specified
    # by the start and end ID return the list of addresses
    # with the specified city name.
    return(returnList);
end;
```

Indexes

Word	Keys
'IL'	'cust:2149:state' , 'cust:4111:state'
'OR'	'cust:9134:state'
'MA'	'cust:7714:state' , 'cust:3412:state'
'Boston'	'cust:1839:address'
'St. Louis'	'cust:9877:address' , 'cust:1171:address'
	. . . .
'Portland'	'cust:9134:city'
'Chicago'	'cust:2149:city' , 'cust:4111:city'

Some key-value databases incorporate ***search functionality*** directly into the database

A built-in search system would index the string values stored in the database and create an ***index*** for rapid retrieval

The search system keeps a ***list of words*** with the keys of each key-value pair in which that word appears

Suggested Readings

Chapter 3 of the book "*Dan Sullivan, NoSQL For Mere Mortals, Addison-Wesley, 2015*"

Images

Al the images shown in this lecture have been extracted from:

"Dan Sullivan, NoSQL For Mere Mortals, Addison-Wesley, 2015"