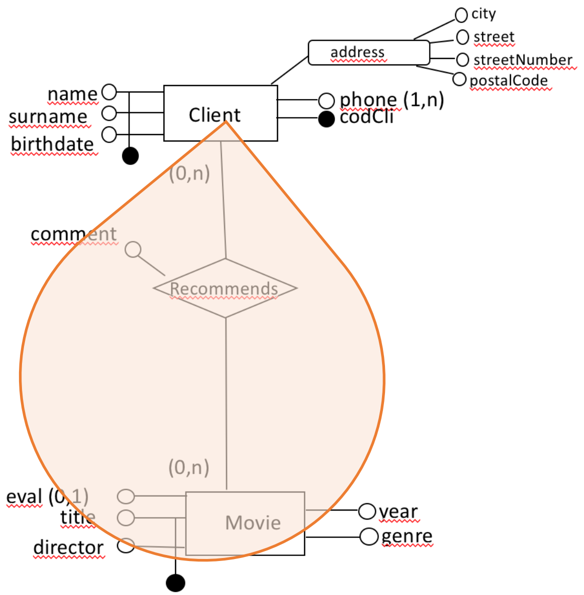
Architecture of aggregate-oriented NoSQL systems

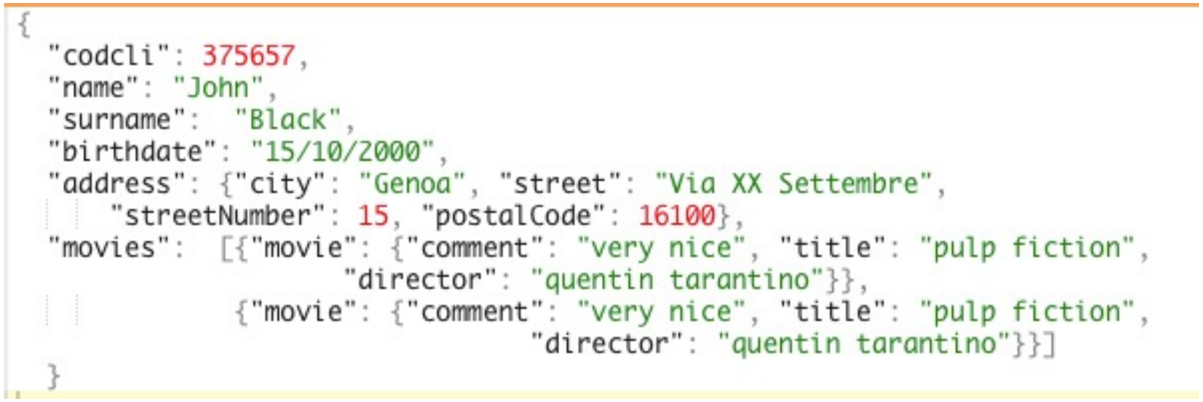
Back to aggregates (logical view)



Aggregate-oriented schema (using the meta-notation, for schema representation in aggregate-oriented NoSQL systems – simplified JSON schema information)

client: { codCli, name, surname, birthdate,
address: {city, street, streetNumber, postalCode},
movies: [movie: {comment, title, director,…}]
}

JSON: instance level (for data representation and exchange)



```
{
  "codcli": 375657,
  "name":  "John",
  "surname":   "Black",
  "birthdate":  "15/10/2000",
  "address": {"city": "Genoa", "street": "Via XX Settembre",
      "streetNumber": 15, "postalCode": 16100},
  "movies":   [{"movie": {"comment": "very nice", "title": "pulp fiction",
                 "director": "quentin tarantino"}},
              {"movie": {"comment": "very nice", "title": "pulp fiction",
                             "director": "quentin tarantino"}}]
  }
```

Back to aggregates (logical view)

An aggregate is a data unit with a complex structure

Each aggregate = (at least) one (key, value) pair

key = partitioning key

Aggregate-oriented schema

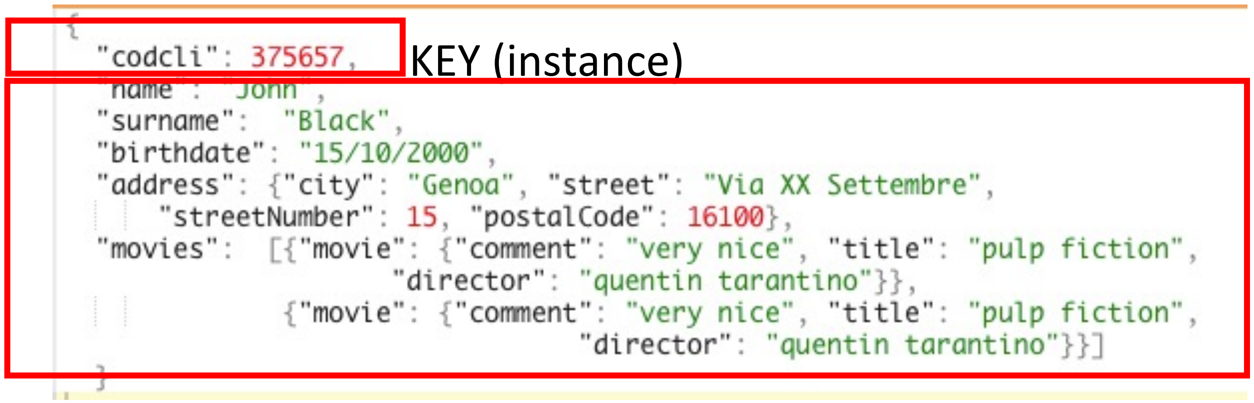
The key might be an identifier but it is not mandatory (often it is not)

```
client: {  codCli,
             name, surname, birthdate,
             address: {city, street, streetNumber, postalCode},
             movies: [{movie: {comment, title, director,…}}]
        }
```

KEY (schema)

VALUE (schema)

JSON: instance level



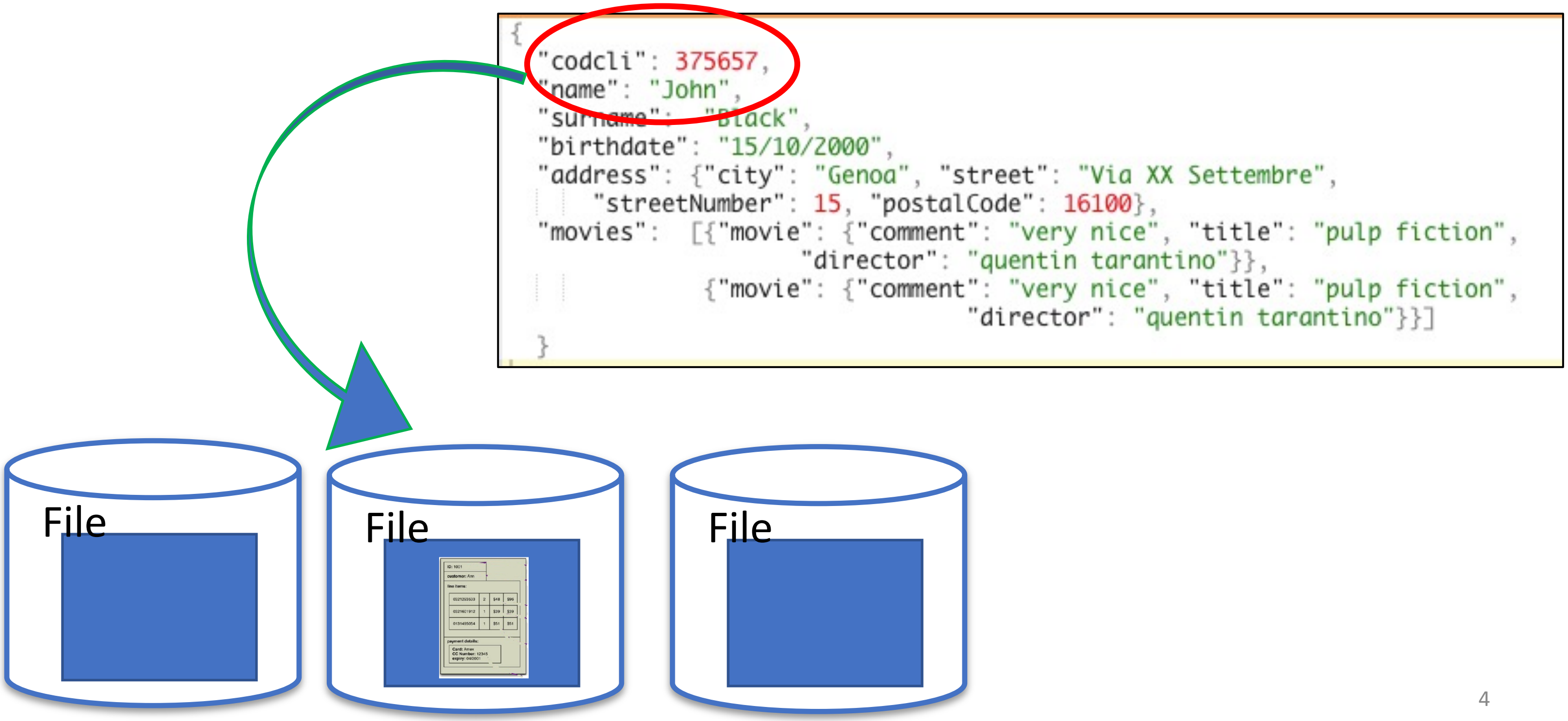
```
{
  "codcli": 375657,
  "name": "John",
  "surname":  "Black",
  "birthdate": "15/10/2000",
  "address": {"city": "Genoa", "street": "Via XX Settembre",
       "streetNumber": 15, "postalCode": 16100},
  "movies":  [{"movie": {"comment": "very nice", "title": "pulp fiction",
                  "director": "quentin tarantino"}},
             {"movie": {"comment": "very nice", "title": "pulp fiction",
                             "director": "quentin tarantino"}}]
}
```

KEY (instance)

VALUE (instance)

Aggregates (physical view)

- All the data about a unit of interest (an aggregate) are kept together on the same node
- Partitioning separates different units (different aggregates) on different nodes
- The aggregate key used as partitioning key



```
{
  "codcli": 375657,
  "name": "John",
  "surname": "Black",
  "birthdate": "15/10/2000",
  "address": {"city": "Genoa", "street": "Via XX Settembre",
      "streetNumber": 15, "postalCode": 16100},
  "movies":  [{"movie": {"comment": "very nice", "title": "pulp fiction",
                    "director": "quentin tarantino"}},
              {"movie": {"comment": "very nice", "title": "pulp fiction",
                          "director": "quentin tarantino"}}]
  }
```

Summary

- Later:
  - presention of specific aggregate-oriented NoSQL systems
  - customization of the aggregate-oriented design methodology to represent aggregates inside each system

- Now:
  - reference architectural properties of aggregate-oriented NoSQL systems

Aggregate-oriented data stores: architectural features in short

Feature	**In aggregate-oriented data stores**
Reference architecture	**Often** P2P
Reference scenarios	Transactional (read/write intensive)
Partitioning	**Often** hash-based or range-based, efficient usage of RAM and indexes
Replication	Often leader-less
Consistency	**Often** eventual consistency
Availability	**Often** high, in general tunable
Fault tolerance	High (**often** no master, P2P ring)
Transactions	**Usually**, no ACID transactions
CAP theorem	AP or CP

Distributed architecture

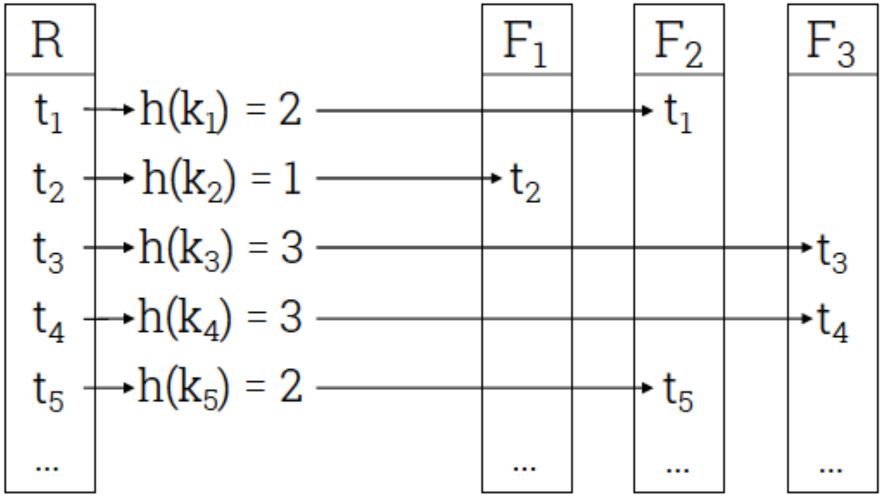
- **Often** (but not always) there is no master and every node is a peer
- All the nodes in a cluster play the same role
- *Each node in a cluster can accept read and write requests, regardless of where the data is actually located in the cluster*
- When a node goes down, read/write requests can be served from other nodes in the network
- You can add nodes to the cluster to improve the capacity of the cluster (scalability)
- Gossip protocols for intra-ring communication

(Back to) Hash-based partitioning

- Many (but not all) aggregate-oriented systems implement a distributed storage based on hash-based partitioning

(Back to) Hash-based partitioning

- Applies a hash function to some attribute that yields the partition number in *{1,…,p}*



(just one node has to be accessed and the node contains all data of interest)

Distributes data evenly if hash function is good
Good for point queries on key and joins
Not good for range queries and point queries not on key

(all nodes have to be accessed but only few nodes contain data of interest)

Hash-based partitioning: rebalancing

- Suppose a hash-based partitioning is applied:
  - *H(v) = v mod p*, where *p* is the number of nodes
- If the number of nodes changes, most of the data must be moved

- *Before p = 10*
  H(10) = 0
  H(11) = 1
  H(12) = 2

- *After p = 11*
  H(10) = 10
  H(11) = 0
  H(12) = 1

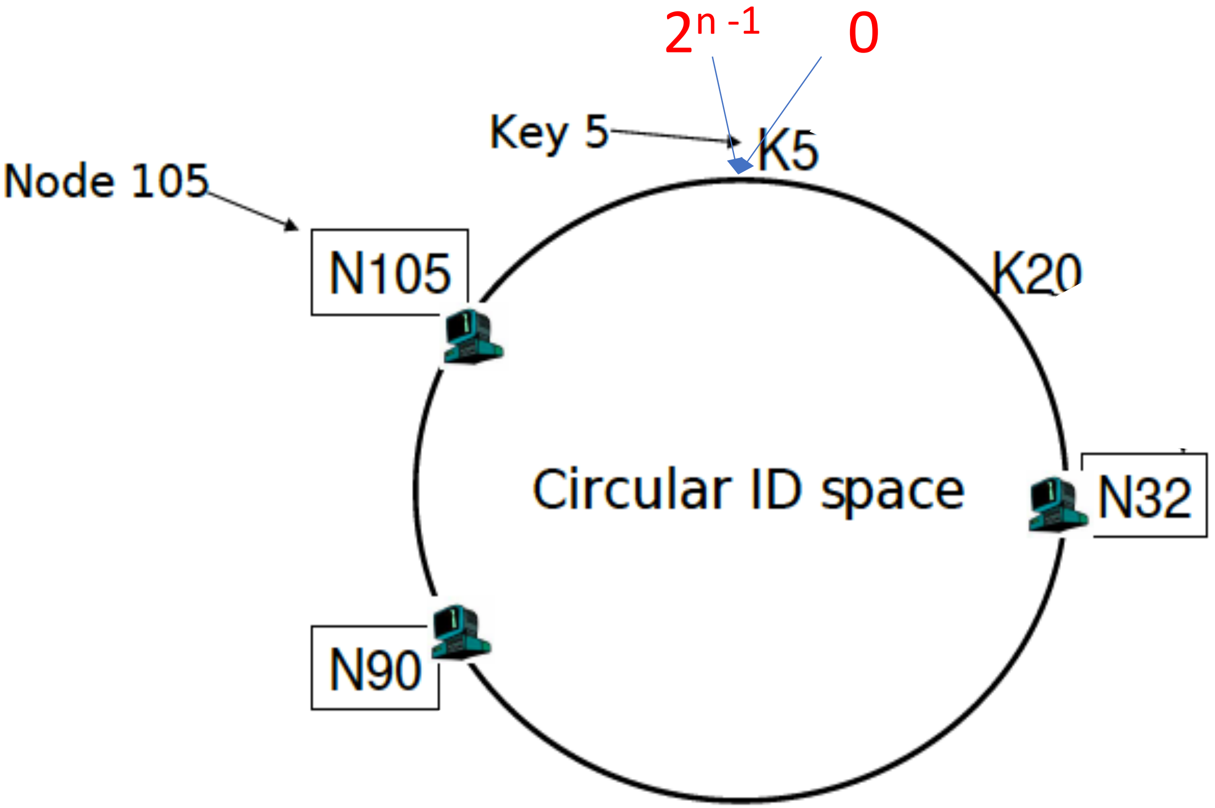
- Typical situation in a large scale distributed data system
- New approaches have to be adopted for better scaling

Partitioning with consistent hashing

- Initially proposed in the context of distributed caching systems
- Use a simple, non-mutable hash function h that maps both the server address and the aggregate keys to the same large address space A
- Assuming to use n bits for representing hash values, hash values correspond to the interval $[0, 2^n-1]$
- The hash function can be implemented in different ways
- Example: $H(k) = k \bmod 2^n$

Consistent hashing: ring

- Organize A as a ring, scanned in clockwise order
- Each element has a successor, the successor of $2^n - 1$ being 0



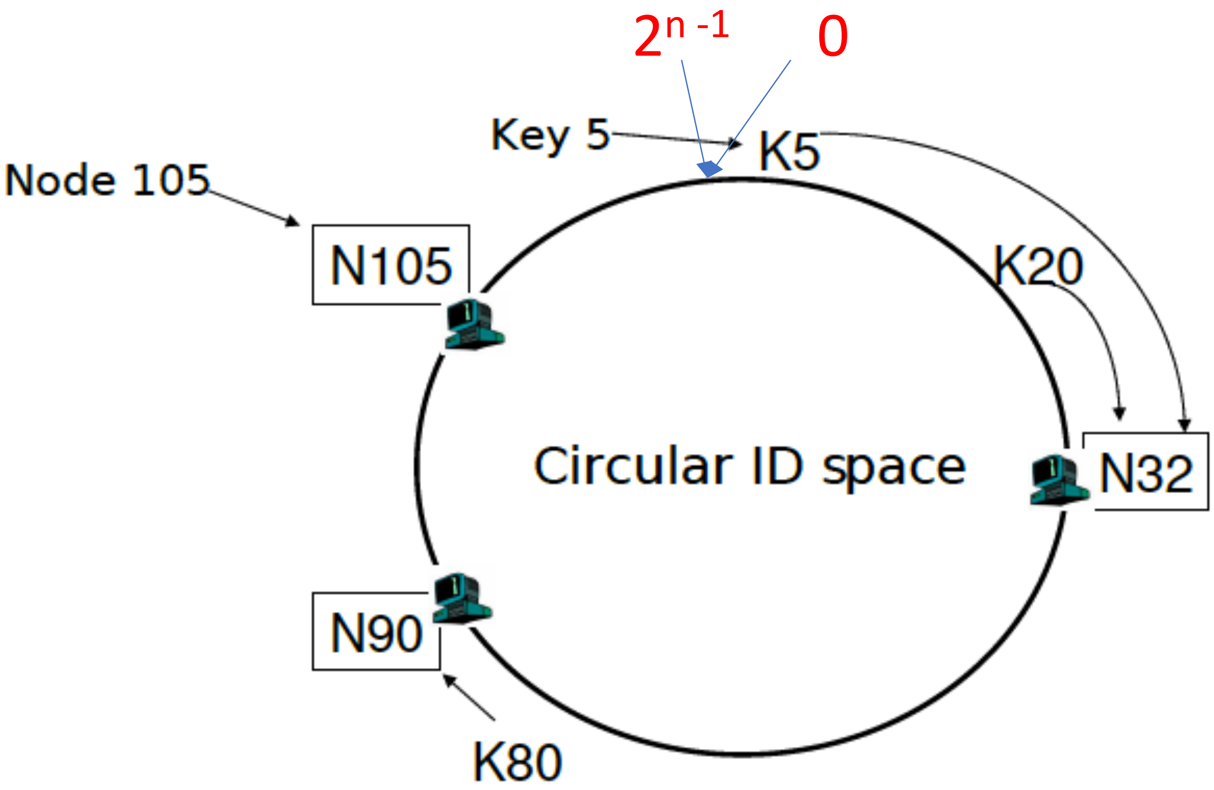
Consistent hashing: storage rule

- Each node is associated with ad id, used for hashing (no details on that)

- Aggregates with key value hashed into k are assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space (successor node of k)

- If S and S’ are two adjacent nodes (clock-wise) in the ring, all the keys in range (h(S), h(S’)] are mapped to S’

Consistent hashing: example

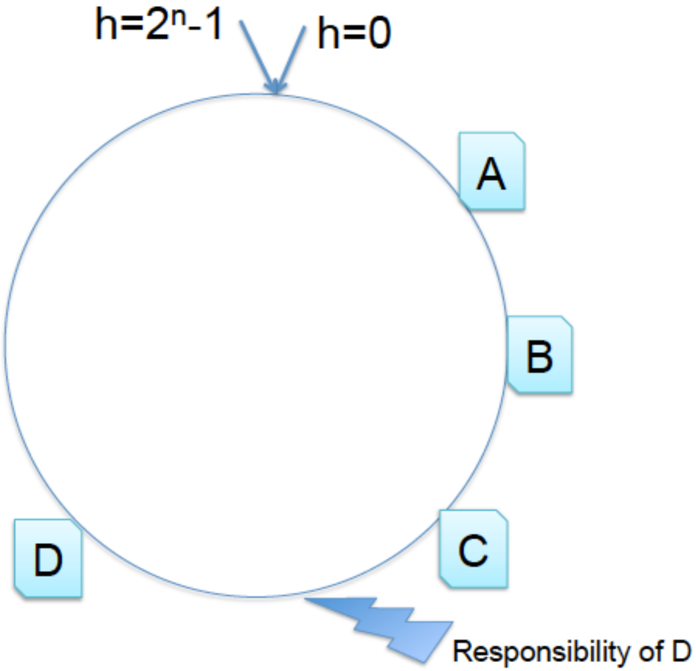


A key is stored at its successor: node with next higher ID

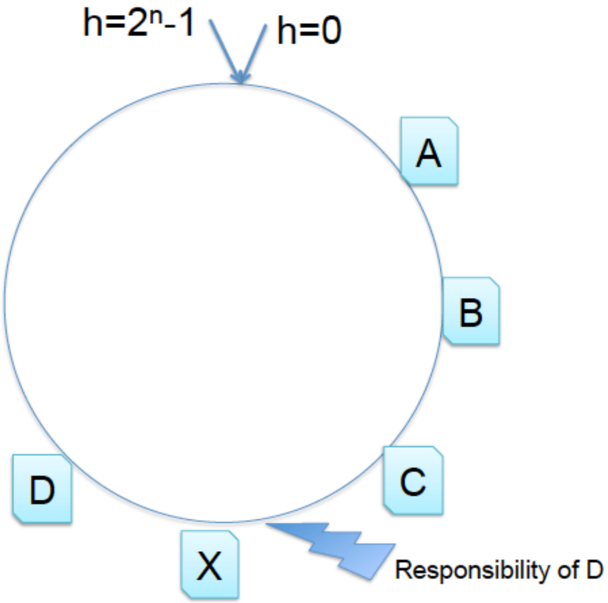
Consistent hashing: scalability

- When a new server is added, we do not need to re-hash the whole data set
- Instead, the new server takes place at a position determined by the hash value on the ring, and part of the objects stored on its predecessor must be moved
- The reorganization is local, as all the other nodes remain unaffected

Consistent hashing: scalability



When X joins:
compute the hash value of X on the ring h(X)



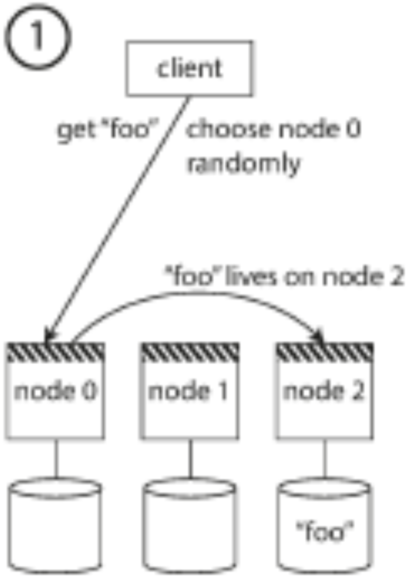
Redistribute the load at D

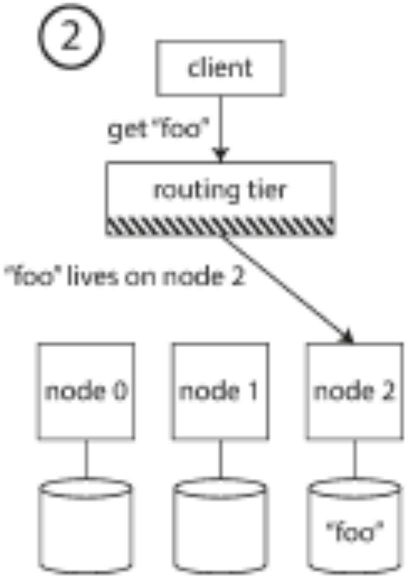
Consistent hashing: routing

- Main question: where is the hash directory (servers locations)?

1. Each node locally records (some) information about the hash directory
   - Typical approach in P2P architectures

2. On a specific ("Master") node, acting as a load balancer
   - raises scalability issues
   - typical approach in Master-slave architectures (do you remember HDFS?)





Consistent hashing: routing (under option 1)

- Suppose a client wants to read items with partition key equal to k
- Assume a client knows only one node S but it does not know how to access h(k)
- Remember that in a P2P network, requests can be sent to any node

- How is it possible to locate the node storing k, starting from S?

Consistent hashing: routing (under option 1)

a) Each node records its successor on the ring

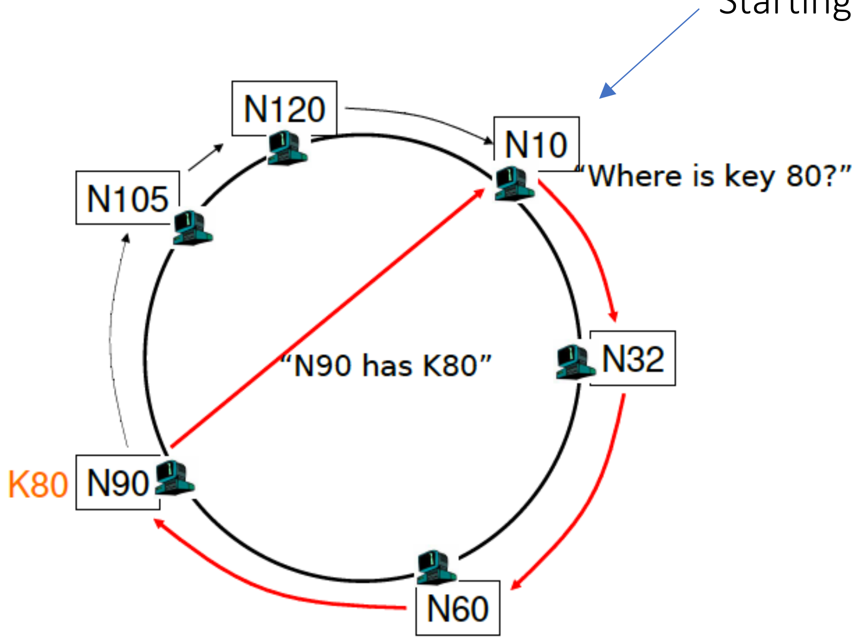
b) Each node records log(n) carefully chosen other nodes

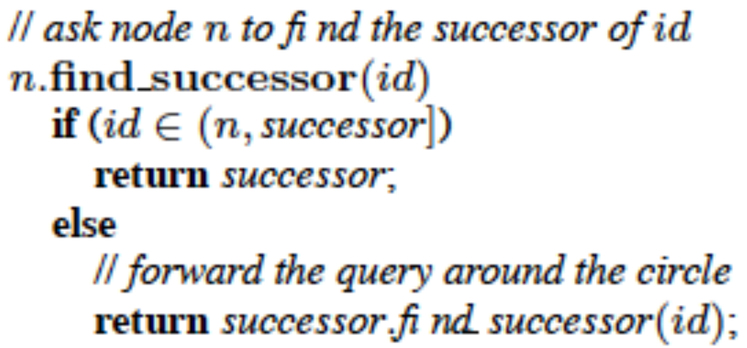
c) Full duplication of the hash directory at each node

Consistent hashing: routing (under option 1.a)

Each node records its successor on the ring

- The starting node S sends a message to its successor
- Hop-by-hop from there
- This is O(N), where N is the number of nodes, no good
- Low maintainance protocol (smallest amount of metadata at each node – address of the success)


Starting node S

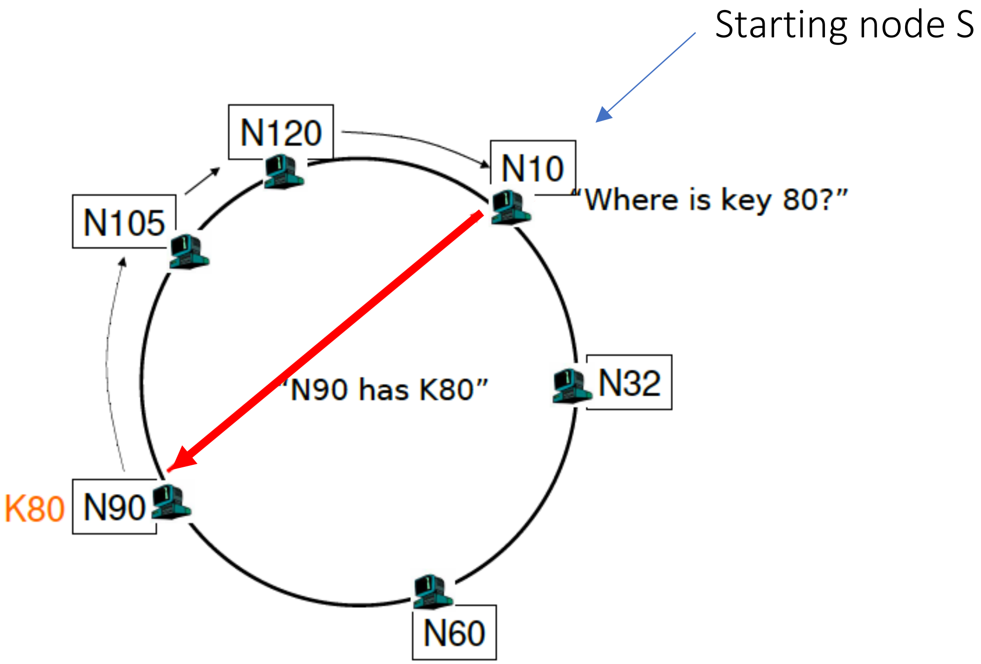


```
// ask node $n$ to find the successor of $id$
$n$.find_successor($id$)
    if ($id \in (n, successor]$)
        return $successor$;
    else
        // forward the query around the circle
        return $successor$.find_successor($id$);
```

Consistent hashing: routing (under option 1.c)

Full duplication of the hash directory at each node

- Ensures 1 message for routing
- Heavy maintenance protocol which can be achieved through gossiping (broadcast of any event affecting the network topology)



Consistent hashing: routing (under option 1.b)

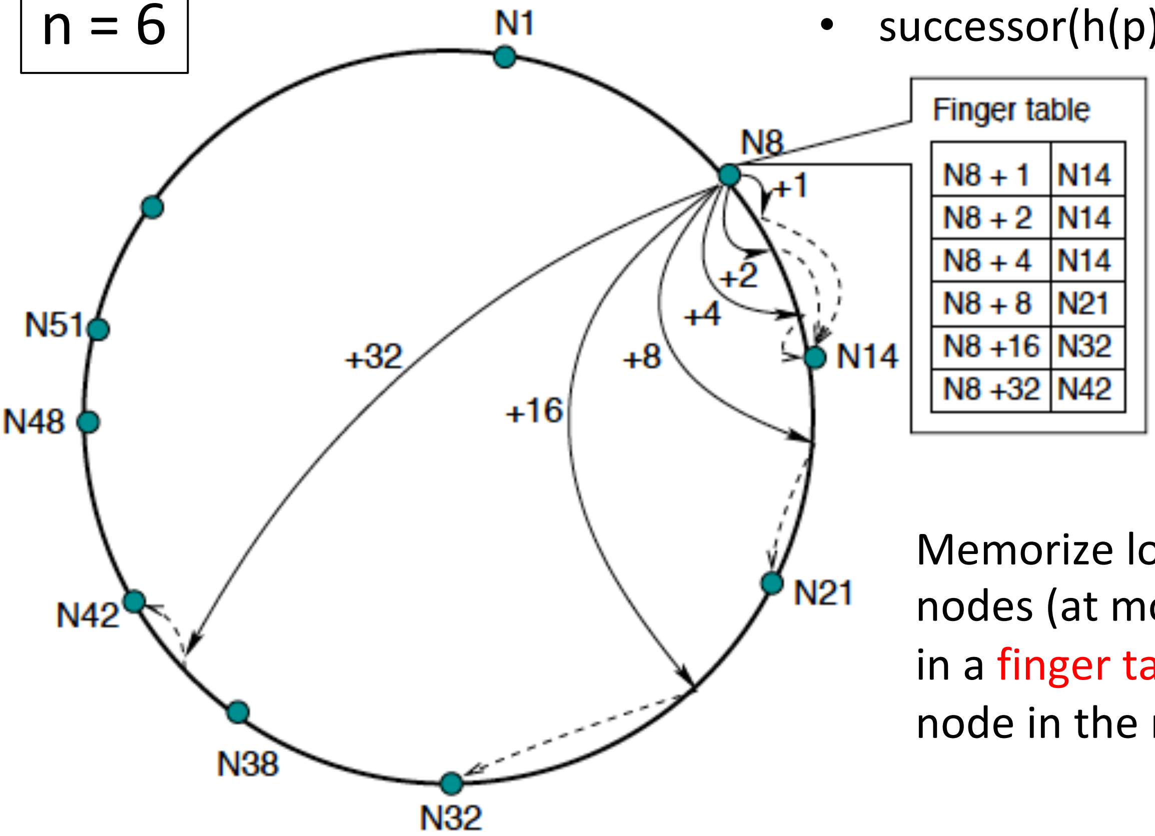
- Each node records log(n) carefully chosen other nodes
- Ensures O(log(N)) messages for routing queries, where N is the number of nodes
- Convenient trade-off for highly dynamic networks
- Based on the CHORD algorithm

CHORD algorithm

i-th friend of a node p:

- node that follows p by at least 2^{i-1}, i =1,…,n
- the first node with hash value equal or following value $h(p) + 2^{i-1}$, clockwise responsible for key $h(p) + 2^{i-1}$
- successor($h(p) + 2^{i-1}$) → p.finger(i)

n = 6



Finger table

N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 +16	N32
N8 +32	N42

$N8 + 2^0$
$N8 + 2^1$
$N8 + 2^2$
$N8 + 2^3$
$N8 + 2^4$
$N8 + 2^5$

Memorize locations of other (friend) nodes (at most n in a ring with 2^n nodes) in a finger table associated with each node in the ring

- Low number of friends, low space for storing the finger table
- Each node knows more about nodes closely following it on the circle than about nodes farther away

CHORD algorithm: routing

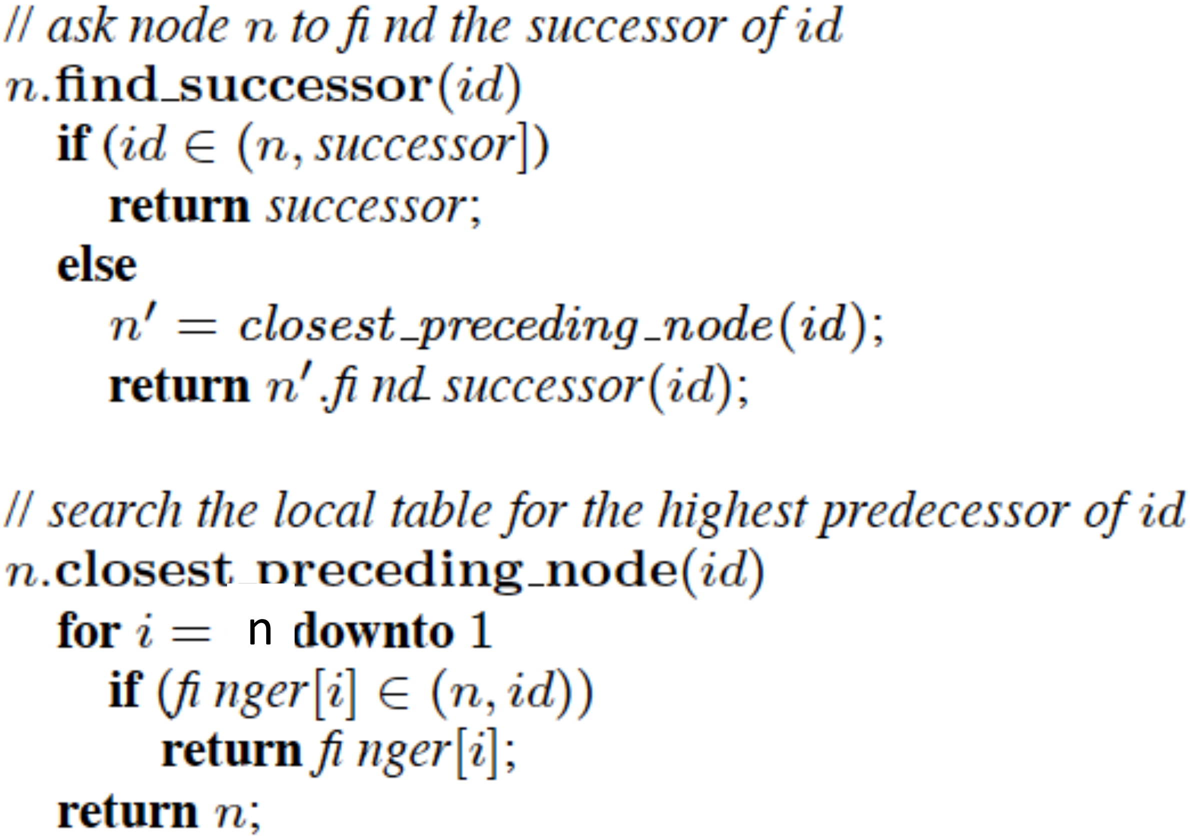
- A node p cannot (in general) find directly the node p’ responsible for a key k (successor(k), following k on the ring) but p can find a friend which holds a more accurate information about k

CHORD algorithm: routing

- If the starting node S is responsible for the searched key k, it replies to the client
- If k falls between S and its successor, node S returns its successor
- Otherwise, S searches its finger table for the node S' whose ID most immediately precedes k , and then apply again the procedure at node S
- The reason behind this choice of S' is that the closer S' is to k, the more it will know about the identifier circle in the region of k

- It can be proved that with high probablity the search converges in O(log(N)) hops

CHORD algorithm: routing



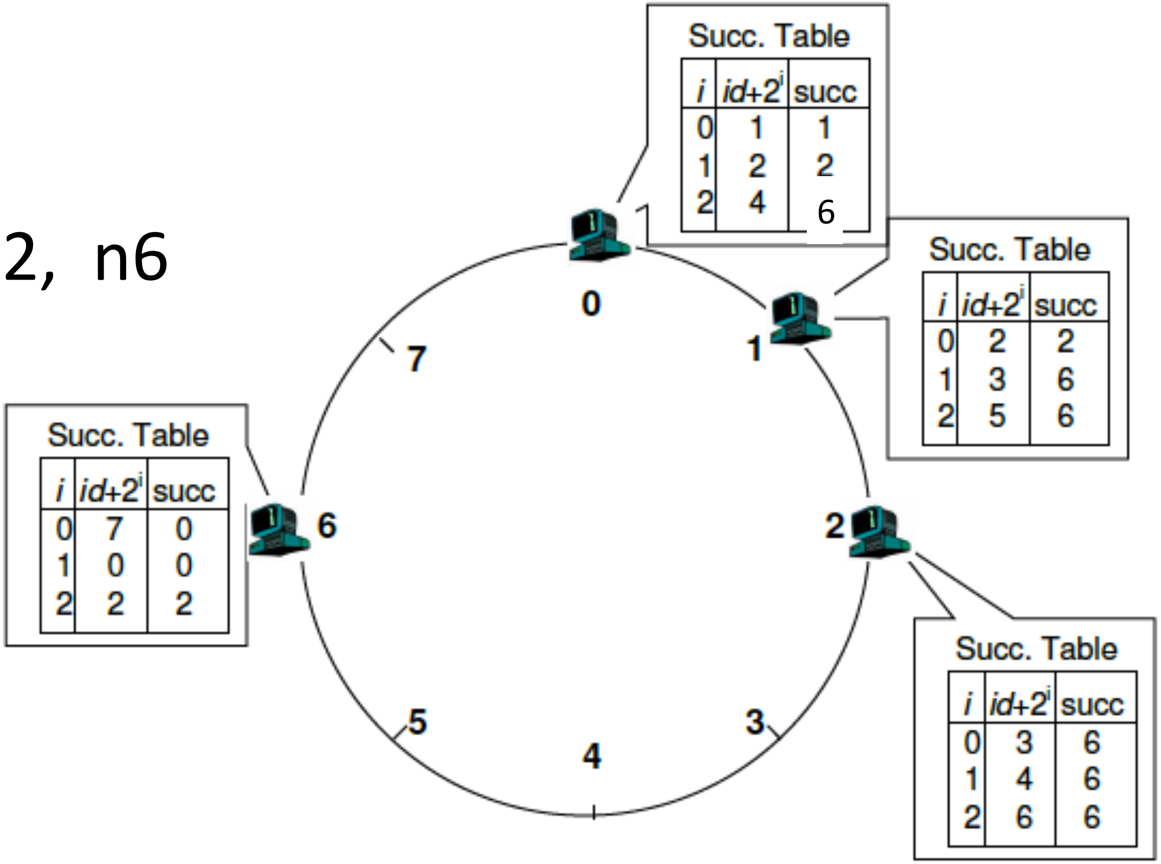
```
// ask node $n$ to find the successor of id
$n$.find_successor($id$)
   if ($id \in (n, successor]$)
      return successor;
   else
      $n' = closest\_preceding\_node(id)$;
      return $n'.find\_successor(id)$;

// search the local table for the highest predecessor of id
$n$.closest_preceding_node($id$)
   for $i =$ n downto 1
      if ($finger[i] \in (n, id)$)
         return $finger[i]$;
   return $n$;
```

Example

Nodes:

n0, n1, n2, n6



Succ. Table

i	$id+2^i$	succ
0	1	1
1	2	2
2	4	6

Succ. Table

i	$id+2^i$	succ
0	2	2
1	3	6
2	5	6

Succ. Table

i	$id+2^i$	succ
0	3	6
1	4	6
2	6	6

Succ. Table

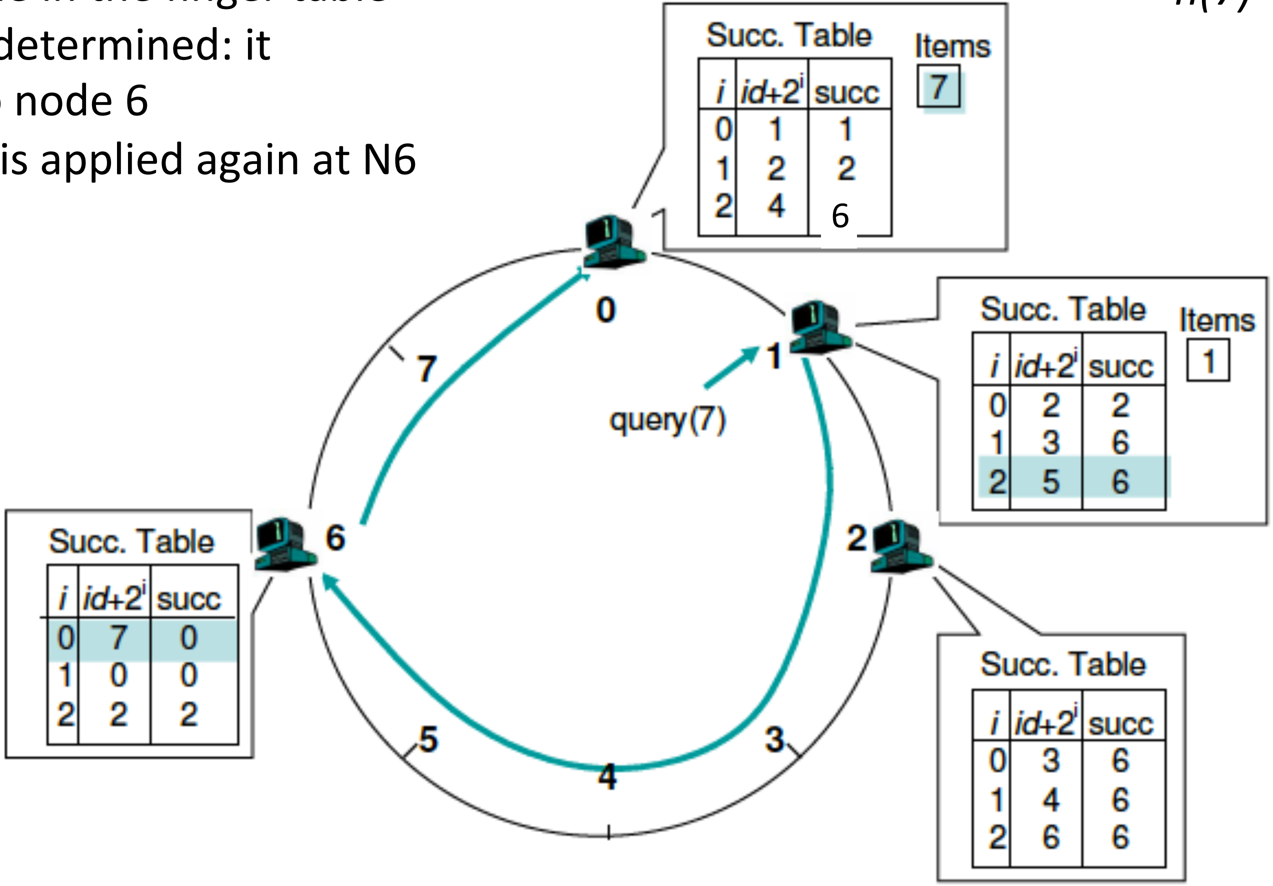
i	$id+2^i$	succ
0	7	0
1	0	0
2	2	2

H(ni) = i

Example

1. N1 does not store 7: h(7) = 0 and N1 can only stores values hash values in (0,1] (from the figure, we notice that N1 only stores 1)
2. the closest node in the finger table preceding 0 is determined: it corresponds to node 6
3. the procedure is applied again at N6

h(7) = 0



Succ. Table

i	id+2^i	succ
0	1	1
1	2	2
2	4	6

Items

7

Succ. Table

i	id+2^i	succ
0	2	2
1	3	6
2	5	6

Items

1

Succ. Table

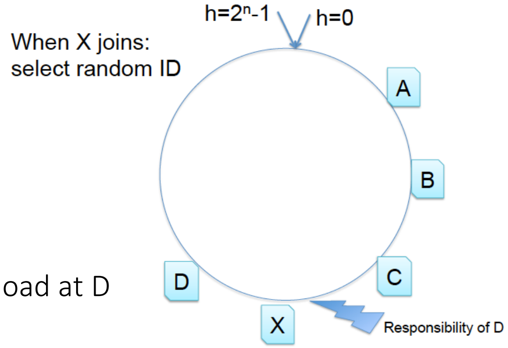
i	id+2^i	succ
0	7	0
1	0	0
2	2	2

Succ. Table

i	id+2^i	succ
0	3	6
1	4	6
2	6	6

Request: a client send to node 1 the requests for reading/writing aggregates with key value equal to 7

CHORD algorithm: scalability



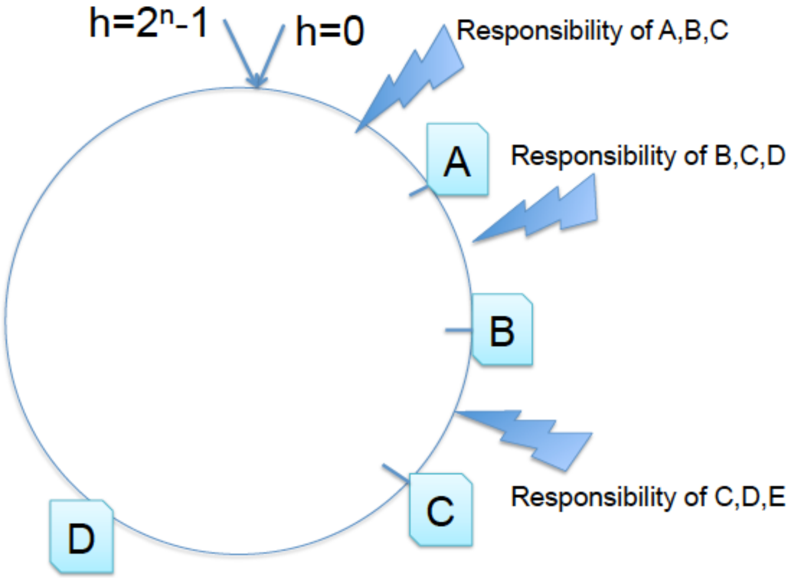
Redistribute the load at D

- When a node p wants to join, it uses a contact node p’ which carries out three tasks
    1. p must initialize its own routing table
       ⇒p’ uses its routing table to locate p friends
    2. the routing table of the existing nodes must be updated to reflect the addition of p
       ⇒more tricky
    3. finally p takes from its successor all the items k such that h(k)≤h(p)
- Step 2 is performed by using a “stabilization” protocol that each node runs periodically in the background and which updates Chord’s finger tables and successor pointers

- No details (see additional references on AulaWeb)

Consistent hashing: replication

- Often leader-less, quorums
- Let m=degree of replication
- Assign key k to m nodes:
  successor(h(k)),
  successor(successor(h(k)))
  successor(…successor(h(k)) …)
  m -1 successors forward



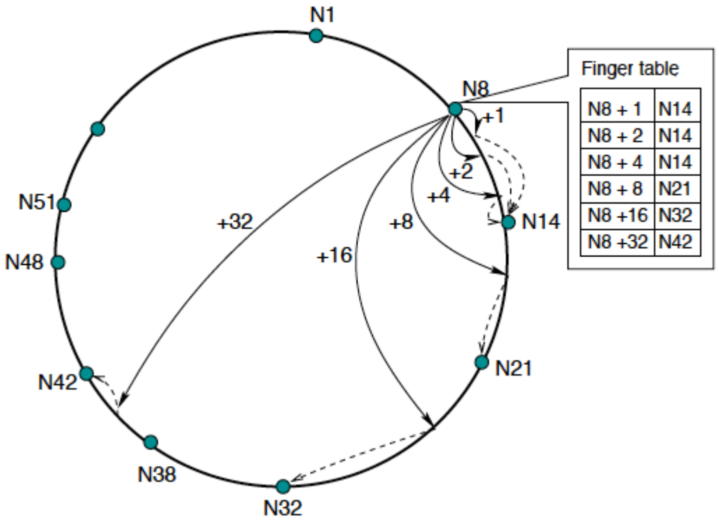
m = 3

Consistent hashing: fault tolerance

- Through replication for being able to access data after a fault
- Specific approaches for taking care of the (possibily shared) knowledge about the hash directory (i.e., the partitioning)
    - If the hash directory is stored in one Master node, the node becomes a single point of failure and such data must be replicated as well
    - If the hash directory is shared and locally stored on each node (as in Chord), other approaches are possible

Consistent hashing: fault tolerance (in CHORD)

- The correctness of the Chord protocol relies on the fact that each node knows its successor
- When considering fault tolerance, the successor of a node p can be considered as the first live node following p on the ring
- If nodes 14, 21, and 32 fail simultaneously, the successor of node 8 is node 38 but, based on the finger table, the successor is set to node 42 (no finger points to 38)

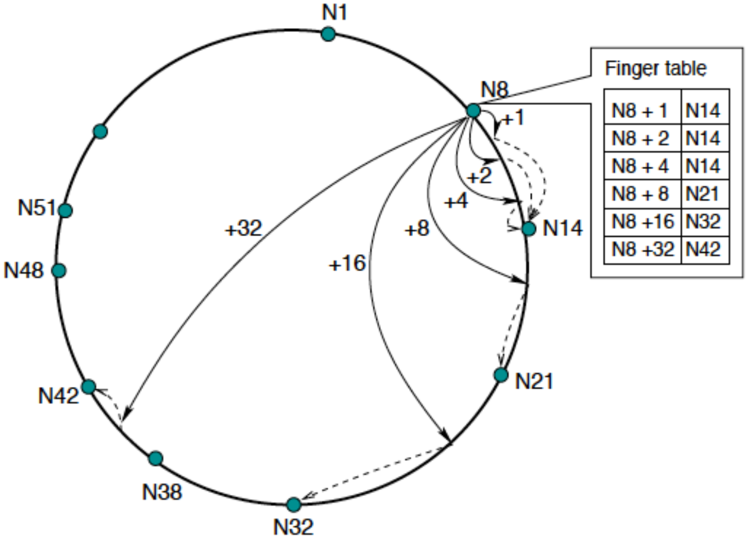


Finger table

N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 +16	N32
N8 +32	N42

Consistent hashing: fault tolerance (in CHORD)

- Incorrect successor will lead to incorrect lookups
  - Consider a query for key 30 initiated by node 8
  - Node 8 will return node 42 (the first node it knows about from its finger table), instead of the correct successor, node 38
- To increase robustness, each node maintains a successor list of size r, containing the node's first r successors
- If a node's immediate successor does not respond, the node can substitute the second entry in its successor list
- All r successors would have to simultaneously fail in order to disrupt the Chord ring, an event that can be made very improbable with modest values of r
- In the previous example, setting r = 4, solve the problem



Finger table

N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 +16	N32
N8 +32	N42

Consistency and transactions

- Traditional database systems satisfy ACID properties
- ACID properties favour Consistency wrt Availability and performance
- In distributed data systems: consistency means replica consistency
- But scalability is a problem
- Give up Consistency and Isolation in exchange for high availability and high performance
- No strong consistency but eventual consistency
- Relaxed properties: from ACID to BASE

Consistency and transactions: from ACID to BASE

- Atomicity
  - The transaction corresponds to a single indivisible action
- Consistency
  - The transaction cannot leave the database in an inconsistent state
- Isolation
  - Transactions cannot interfere with each other and see intermediate results
- Durability
  - Once a transaction commits, the results are made persistent

Consistency and transactions: from ACID to BASE

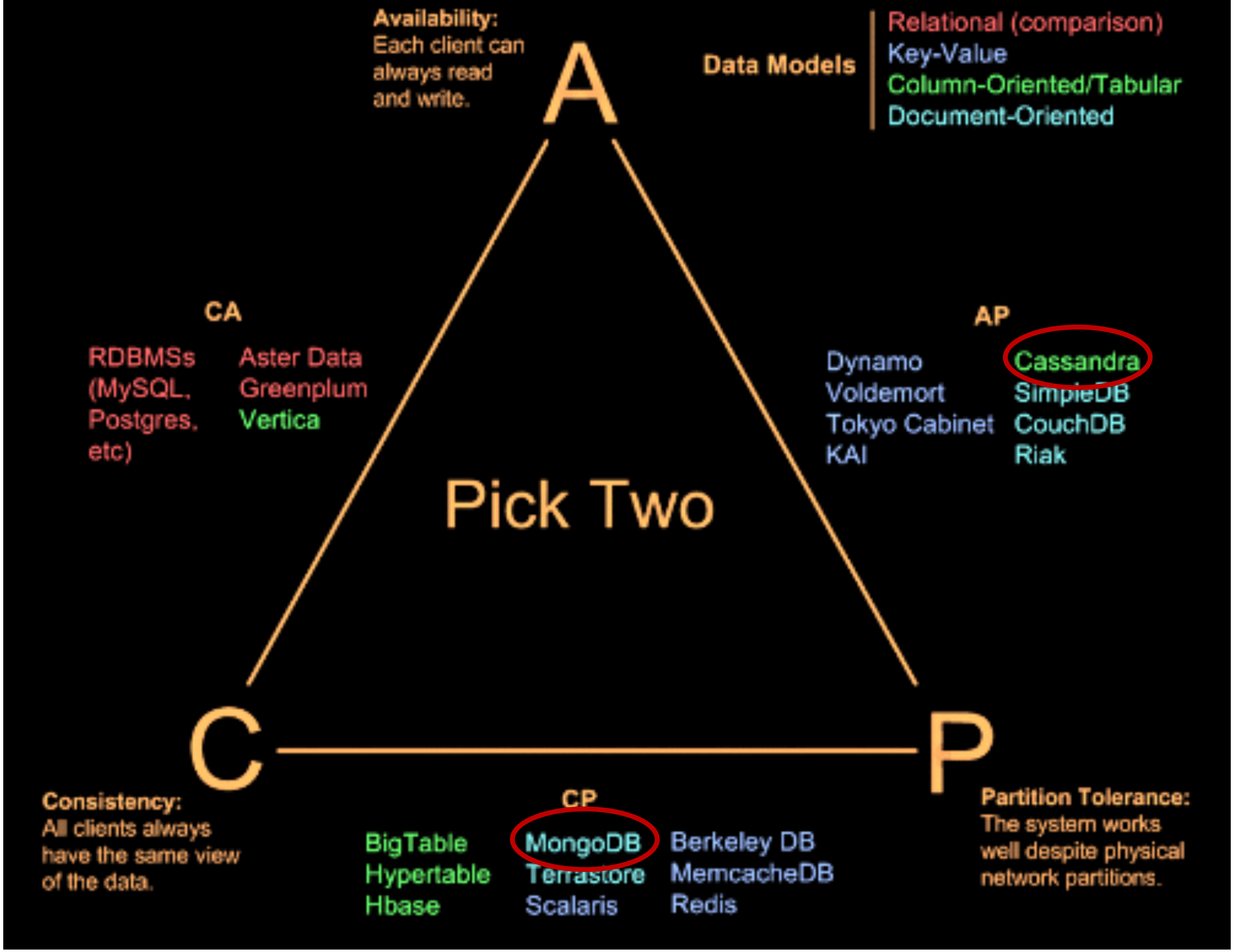
- Basically Available
  - The database appears to work most of the time
- Soft-state
  - Stores do not have to be write-consistent, nor do different replicas have to be mutually consistent all the time
- Eventual Consistency
  - Stores exhibit consistency at some point in the figure

Soft state and durability

- Durability in ACID transactions:
  - When Write is committed, the change is permanent
  - Strict durability is costly
  - In some cases, strict durability is not essential and it can be traded for scalability (write performance)

- A simple way to relax durability:
  - Store data in memory and flush to disk regularly
  - delay write on disks
  - if the system shuts down, we loose updates in memory (unless we use logging mechanisms)

Visual Guide to NoSQL Systems



Availability: Each client can always read and write.

A

Data Models

- Relational (comparison)
- Key-Value
- Column-Oriented/Tabular
- Document-Oriented

CA

RDBMSs (MySQL, Postgres, etc)

Aster Data
Greenplum
Vertica

AP

Dynamo
Voldemort
Tokyo Cabinet
KAI

Cassandra
SimpleDB
CouchDB
Riak

Pick Two

C

P

Consistency: All clients always have the same view of the data.

CP

BigTable
Hypertable
Hbase

MongoDB
Terrastore
Scalaris

Berkeley DB
MemcacheDB
Redis

Partition Tolerance: The system works well despite physical network partitions.