Big Data Architectures

Ioana Manolescu
INRIA Saclay & Ecole Polytechnique
ioana.manolescu@inria.fr
http://pages.saclay.inria.fr/ioana.manolescu/

M2 Data and Knowledge
Université de Paris Saclay

Dimensions of distributed data management systems

- **Data model**:
  - Relations, trees (XML, JSON), graphs (RDF, others…), nested relations
  - Query language
- **Heterogeneity** (DM, QL): none, some, a lot
- **Scale**: small (~10-20 sites) or large (~10.000 sites)
- **ACID** properties
- **Control**:
  - Single master w/complete control over N slaves (Hadoop/HDFS)
  - Sites publish independently and process queries as directed by single master/*mediator*
  - Many-mediator systems, or peer-to-peer (P2P) with *super-peers*
  - Sites completely independent (P2P)

Today's lecture

- P2P architectures:
  - highest degree of peer autonomy
  - high degree of distribution
- Cloud Big Data management architectures
  - Cloud computing
  - Structured data management on top of cloud services

PEER-TO-PEER DATA MANAGEMENT ARCHITECTURES

Peer-to-peer architectures

- Idea: easy, **large-scale** sharing of data with **no central point of control**
- All the peers play identical roles
- Peers may join the peer network or leave it at any time

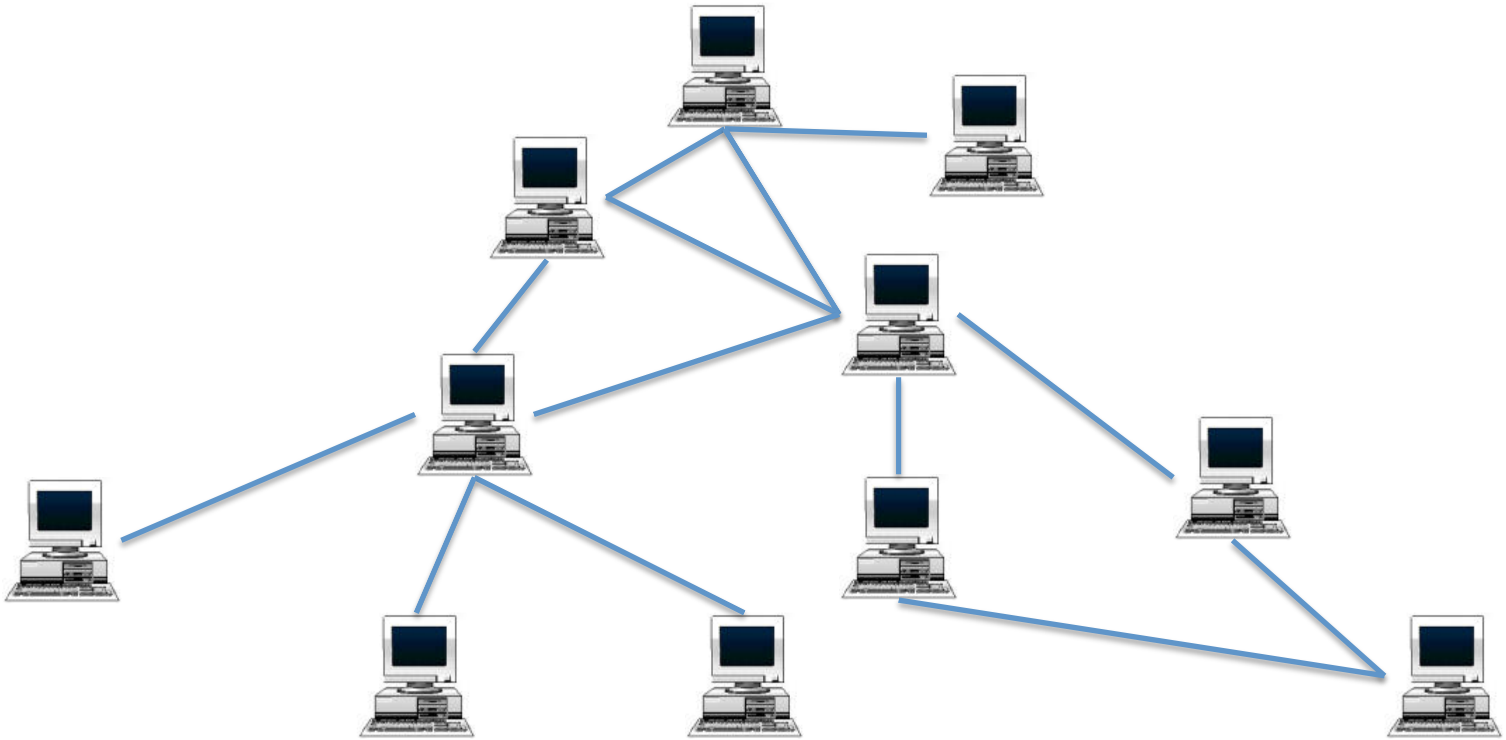
- **Advantages**:
  - Distribute work; preserve peer independence
- **Disadvantages**:
  - Lack of control over peers which may leave or fail → need for mechanisms to cope with peers joining or leaving (*churn*)
  - Schema unknown in advance; need for data discovery

Peer-to-peer architectures

- **Large-scale** sharing of data with **no central point of control**
- Two main families of P2P architectures:
  - Unstructured P2P networks
    - Each peer is free to connect to other peers;
    - Variant: super-peer networks
  - Structured P2P networks
    - Each peer is connected to a set of other peers determined by the system
- Also: hybrid P2P architectures
  - A "central" subset of the network is structured, the rest is unstructured

Unstructured P2P networks

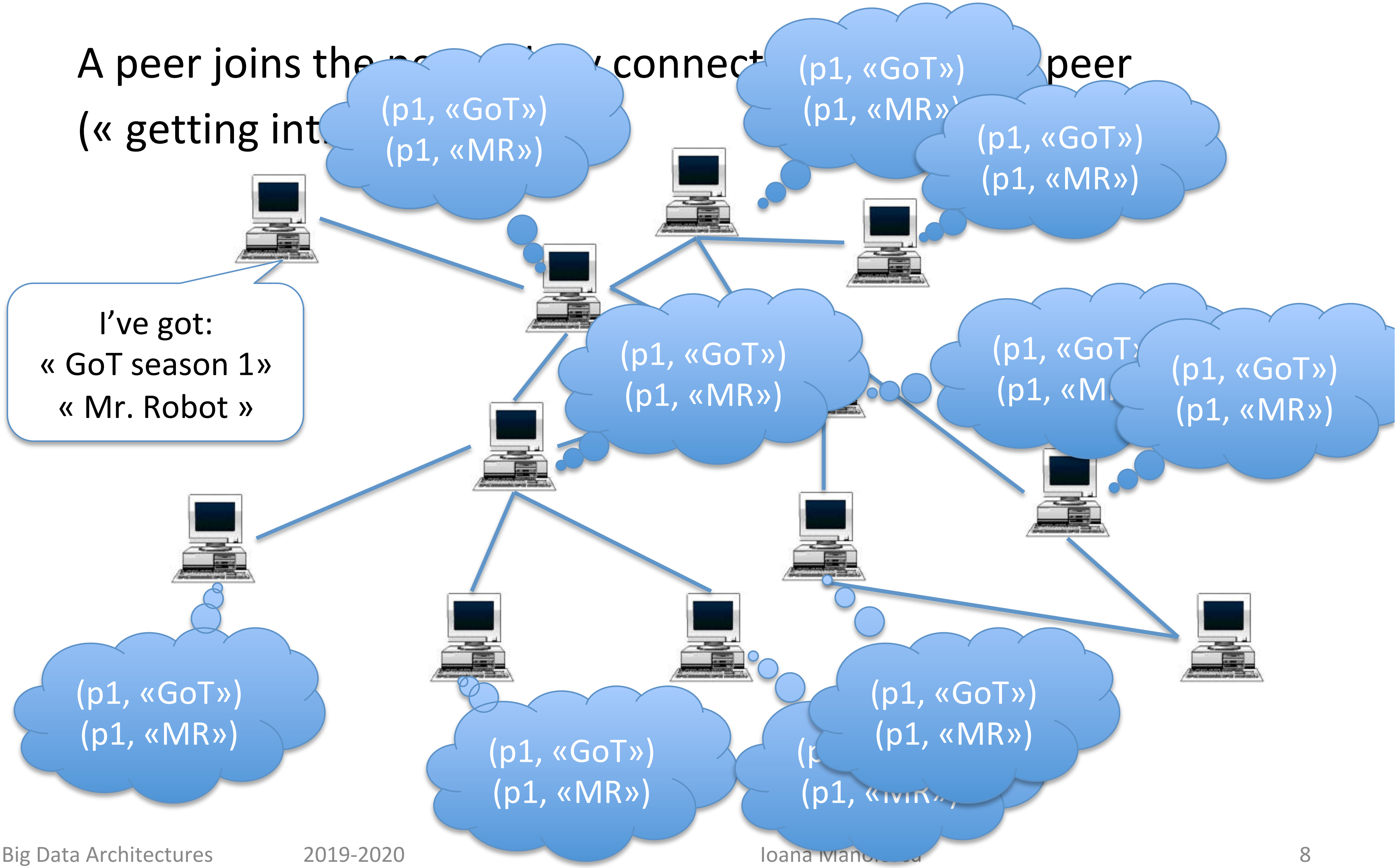
A peer joins the network by connecting to another peer (« getting introduced »)



Each peer may advertise data that it publishes→peers « know their neighbors » up to some level

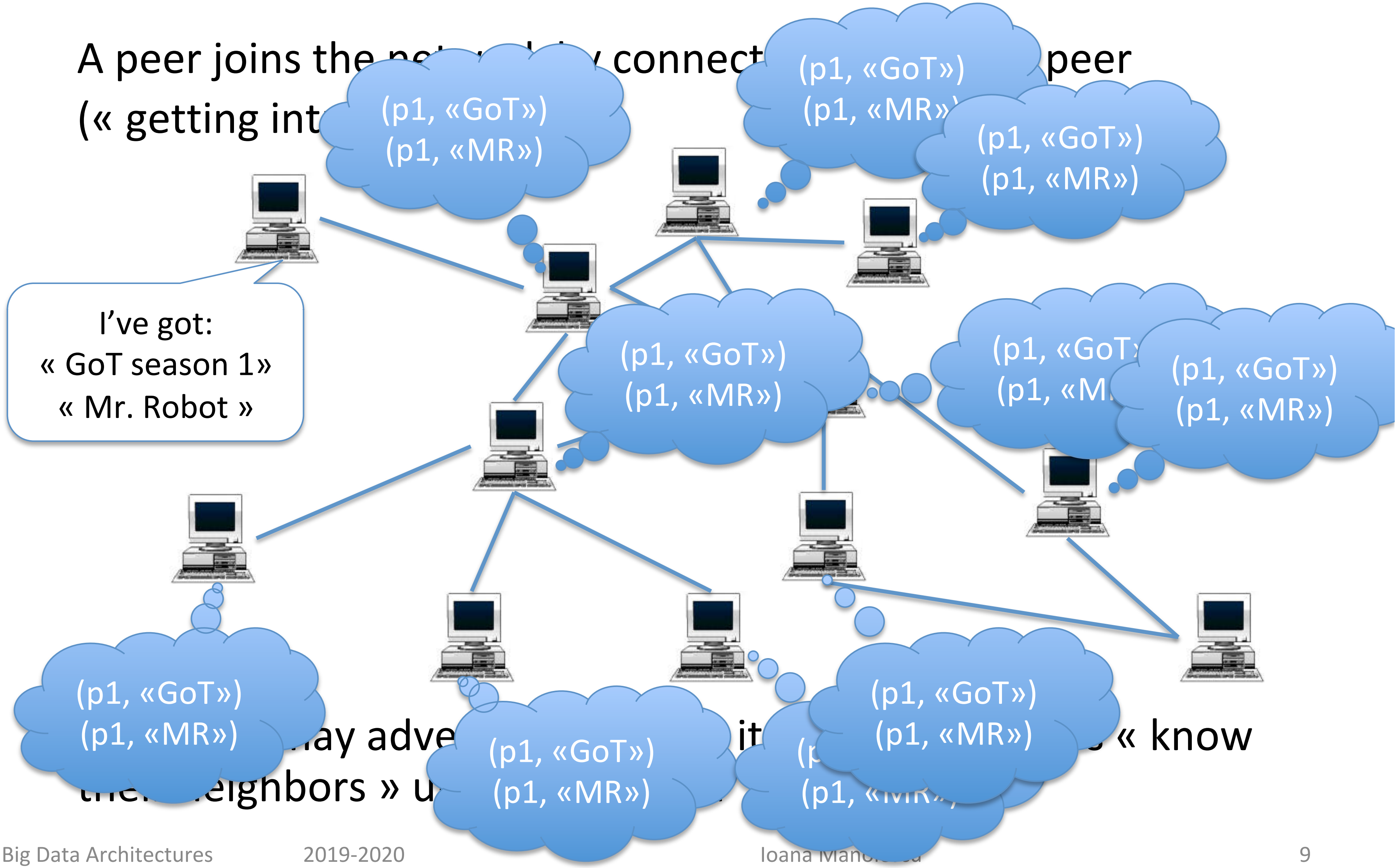
Unstructured P2P networks

A peer joins the ne[...] connect[...] peer
(« getting int[...]



Unstructured P2P networks

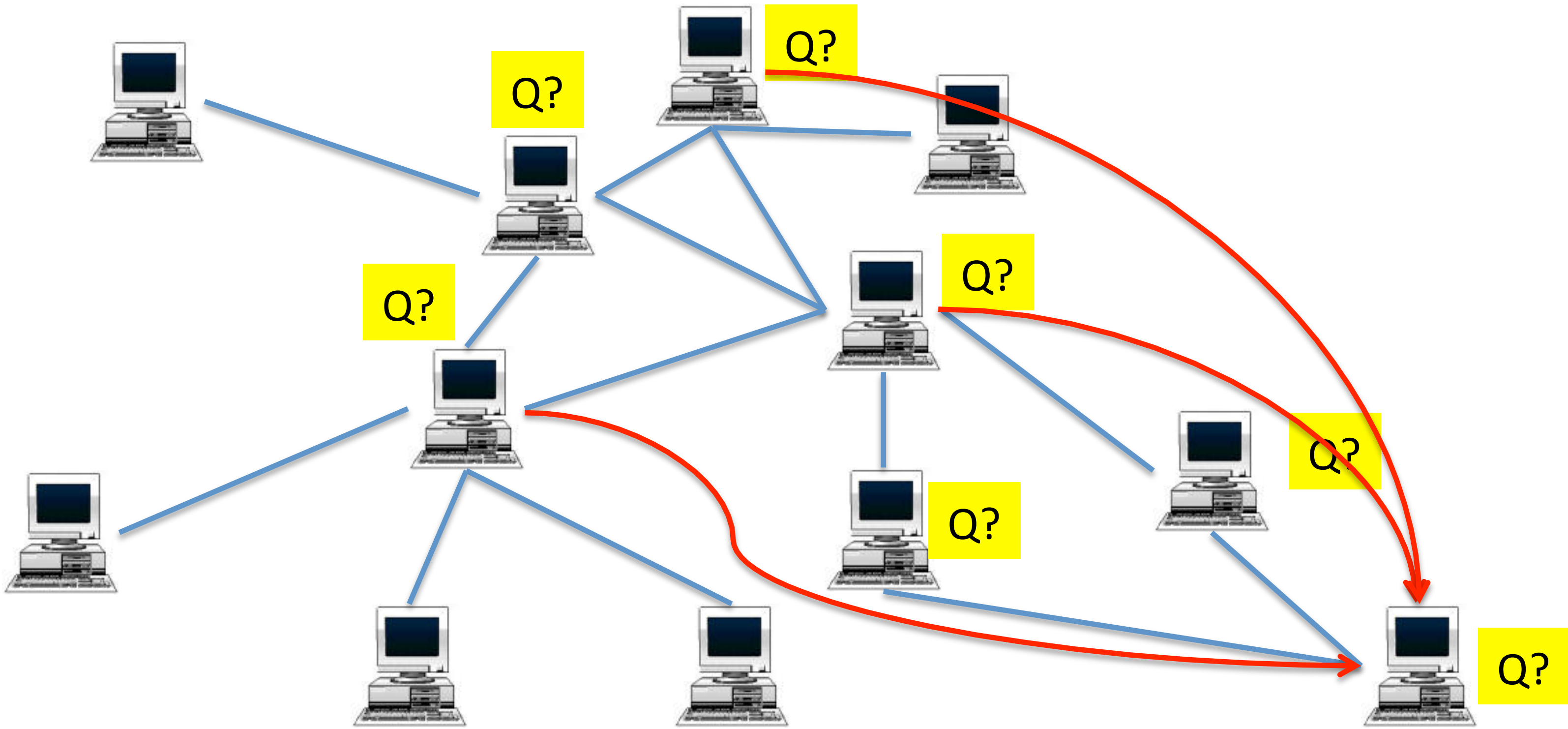
A peer joins the ne… connect… peer
(« getting int…



…ay adve… it… « know
the… eighbors » u…

Unstructured P2P networks

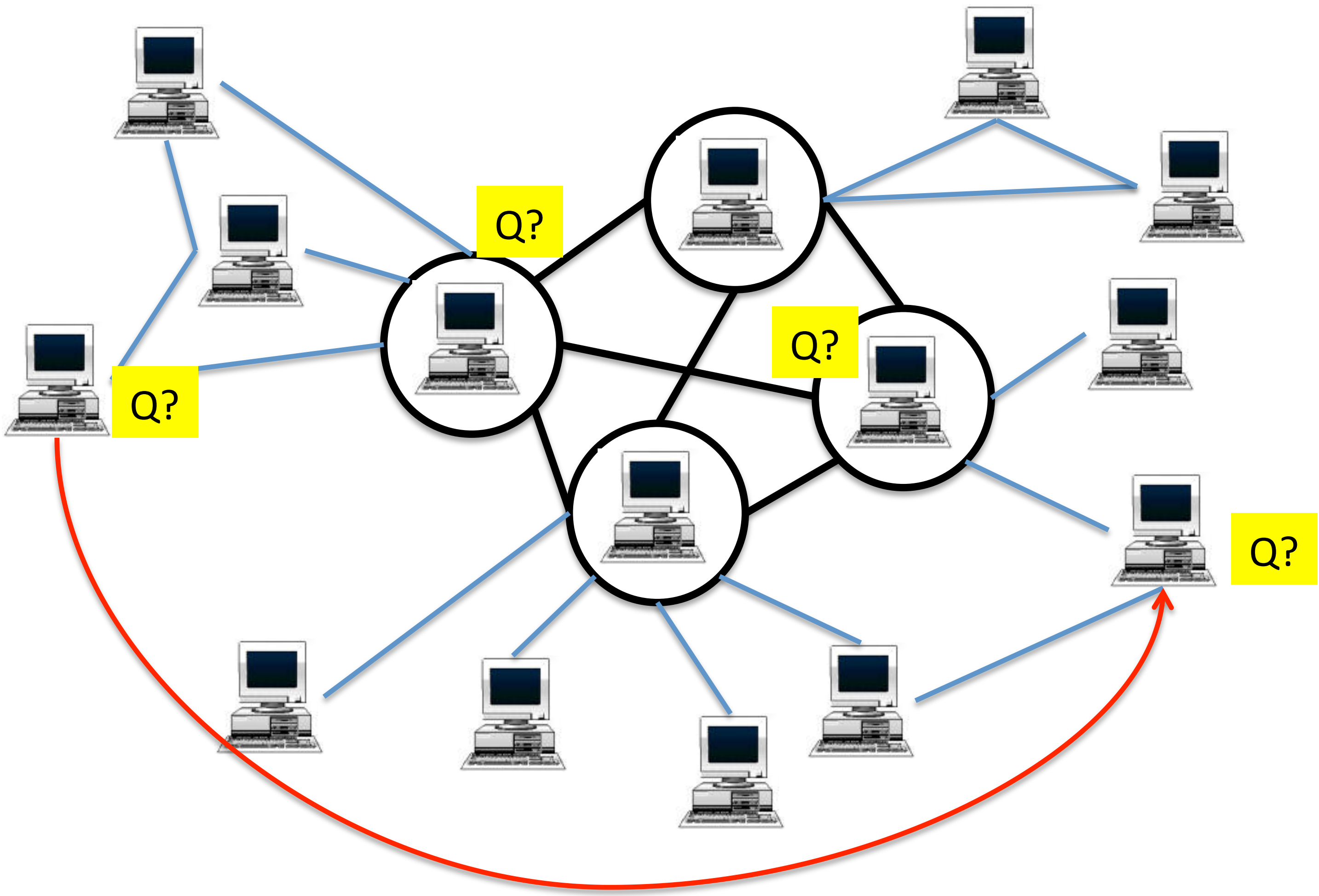
Queries are evaluated by propagation from the query peer to its neighbors and so on recursively (flooding)



To avoid saturating the network, queries have TTL (time-to-live)

This may lead to missing answers → a. replication; b. superpeers

Hybrid P2P network



- Small subset of superpeers all connected to each other
- Specialized by data domain, e.g. [Aa—Bw], [Ca—Dw], … or by address space
- Each peer is connected at least to a superpeer, which routes the peer's queries

Structured P2P network

- Peers form a **<u>logical address space</u>** 0… 2^k-1
  - Some positions may be empty
  - The peer position is obtained with the help of a **hash function**
  - e.g., H(peer IP address)=n, 0 <= n <= 2^k-1
  - This also leads to the name: **distributed hash table, DHT**

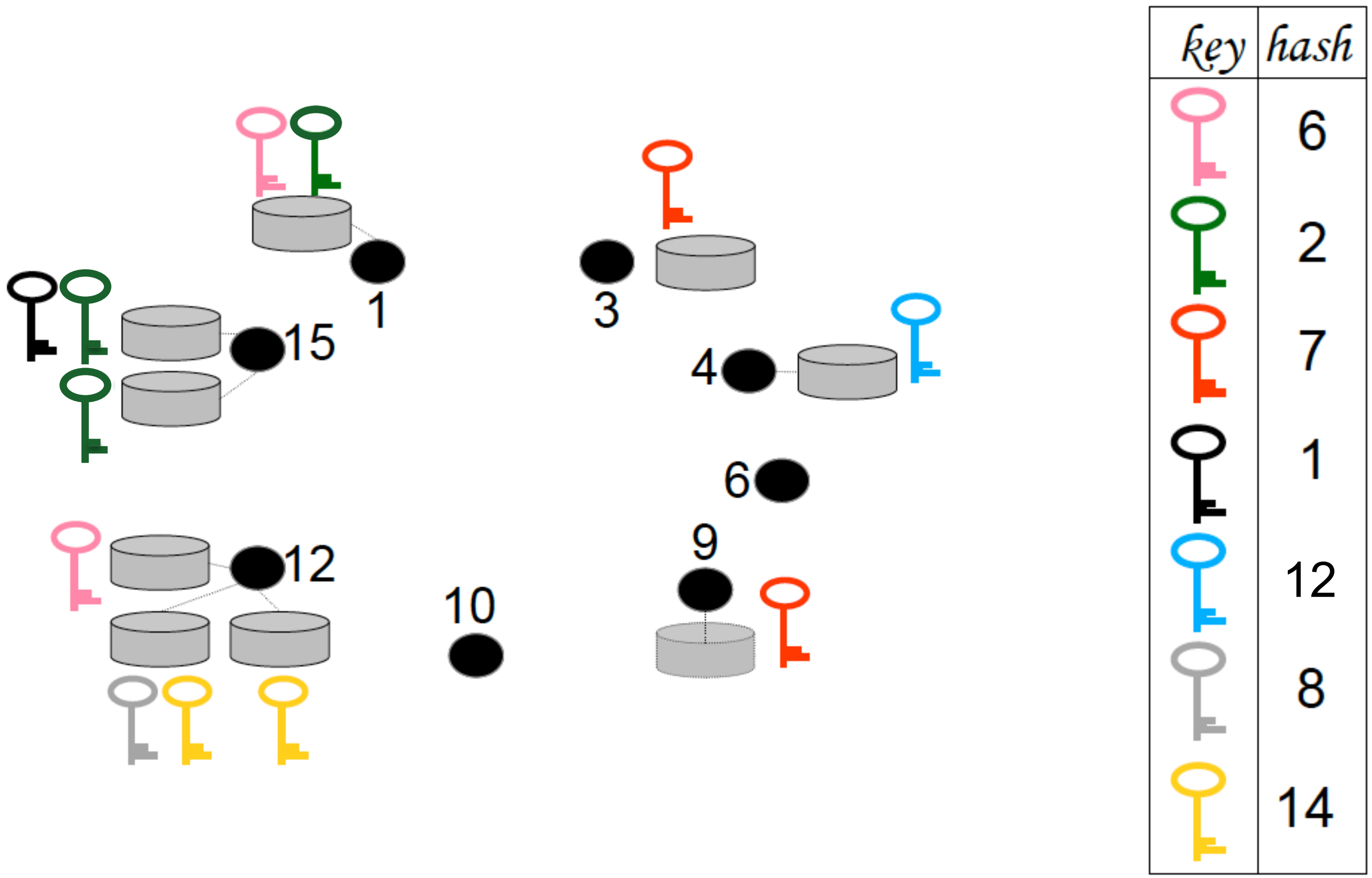
- The global data catalog is created and distributed across the peers, using the same hash function (see next)

Catalog construction (indexing) in structured P2P networks

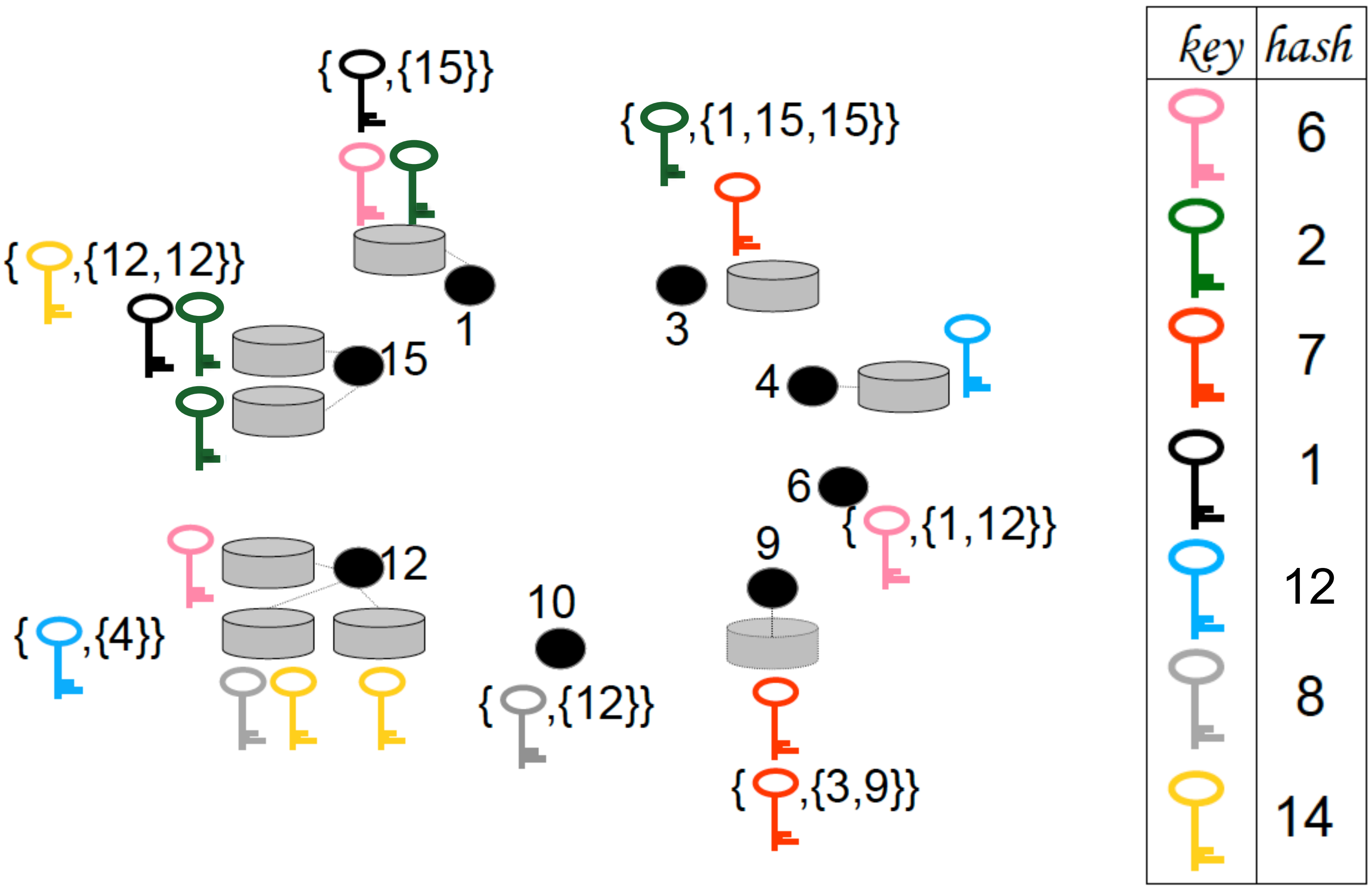
- The **catalog** is built as a set of key-value pairs
  - *Key*: expected to occur in search queries, e.g. «GoT », « Mr Robot »
  - **Value**: the address of content in the network matching the key, e.g. « peer5/Users/a/movies/GoT »

- A **hash function** is used to map every *key* into the address space; this distributes (*key*, value) pairs
  - H(*key*)=n → the (*key*, value) pair is sent to peer n
  - If n is empty, the next peer in logical order is chosen

Catalog construction (indexing) in structured P2P networks

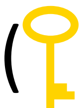


Catalog construction (indexing) in structured P2P networks



Searching in structured P2P networks

Locate all items characterized by  ?
Hash()=6
Peer 6 knows all the locations

Locate all items characterized by ?
Hash()=14
Peer 15 knows all the locations

How do we find peers 6 and 15?

Connections between peers in structured P2P networks

A peer's connections are dictated by the network organization and the logical address of each peer in the space 0… 2^k-1

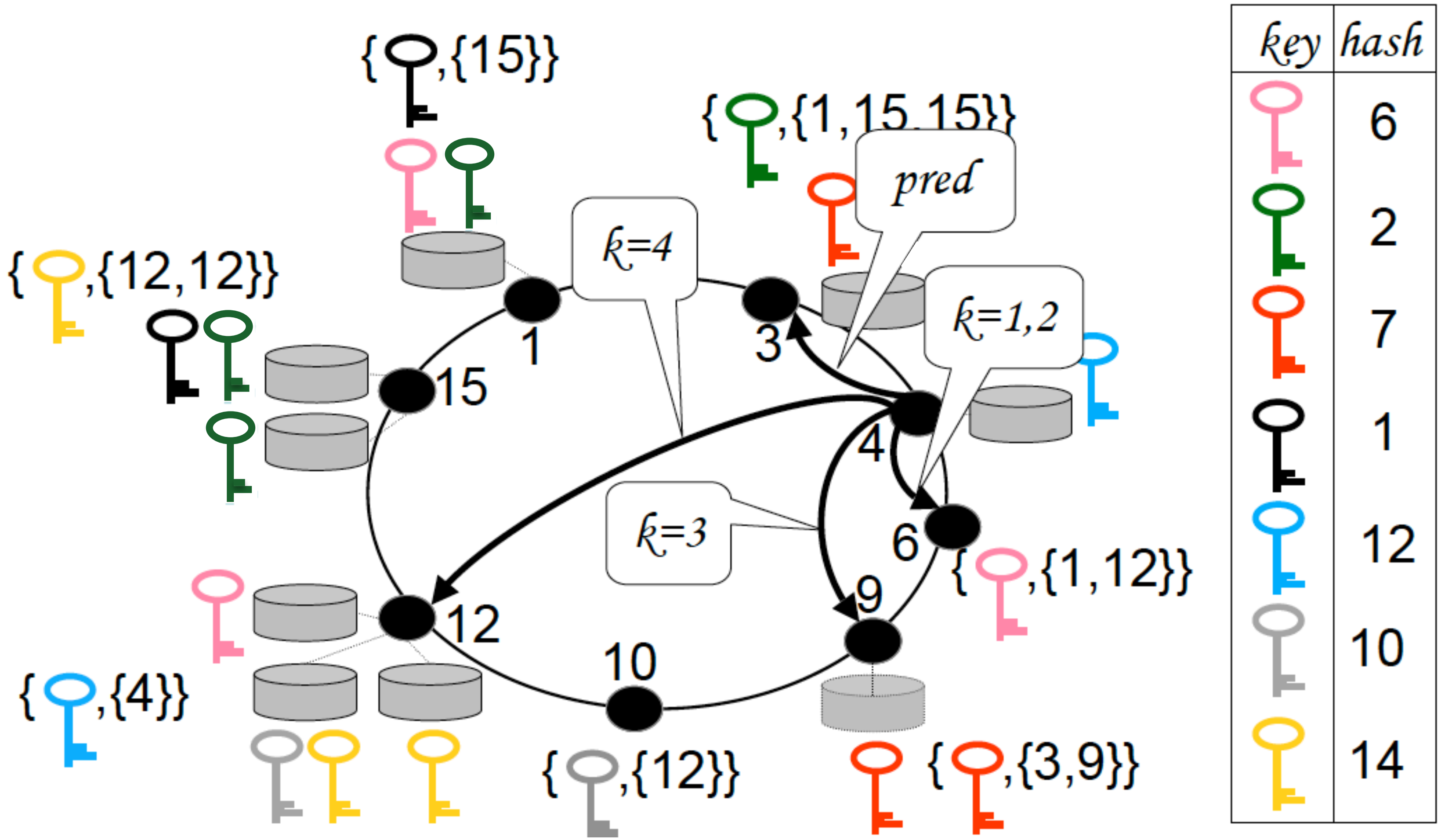
Example: Chord (MIT, most popular)

Each peer n is connected to

- *n+1, n+2, …, n+2^{k-1}*, or to the first peer *following* that position in the address space;
- The *predecessor of n*

The connections are called *fingers*

Connections between peers in Chord



Searching in structured P2P networks

Locate all items characterized by  ?

- Hash()=6
- Peer 6 knows all the locations

How does peer 1 find peer 6?

- 6-1=5; 2 <= $\log_2(5)$ <= 3, thus 6 is after 1 + 2^2
- Redirect the question to the 2nd finger of 1.

How does peer 10 find peer 3?

- (3 $-_{\text{modulo } 16}$ 10)=9; 3 <= $\log_2(9)$ <= 4, thus 3 is after 10 + 2^3
- Redirect the question to the 3rd finger of 10.

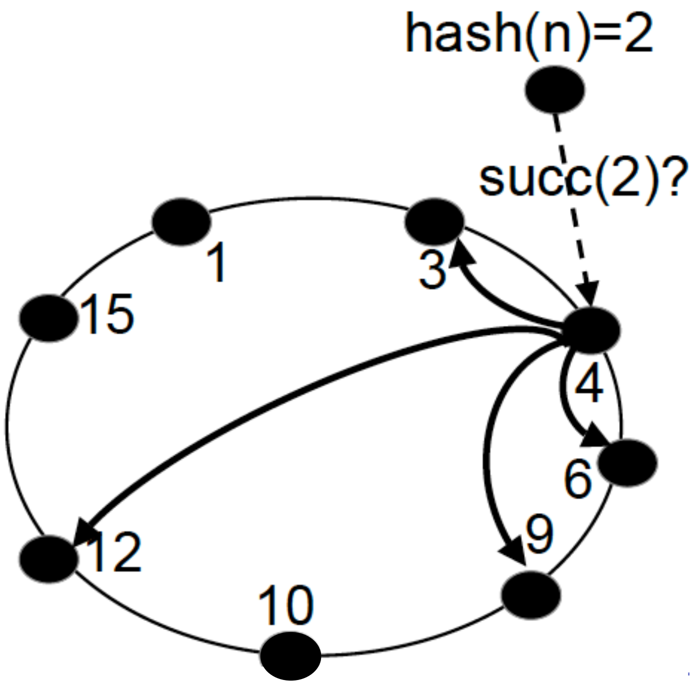
Quick traversals of the ring ($\log_2(N)$ hops)

Peers joining in Chord

To join, a peer n must know (any) peer n' already in the network

Procedure **n.join(n')**:

s = n'.findSuccessor(n);
buildFingers(s);
successor=s;



Peers joining in Chord

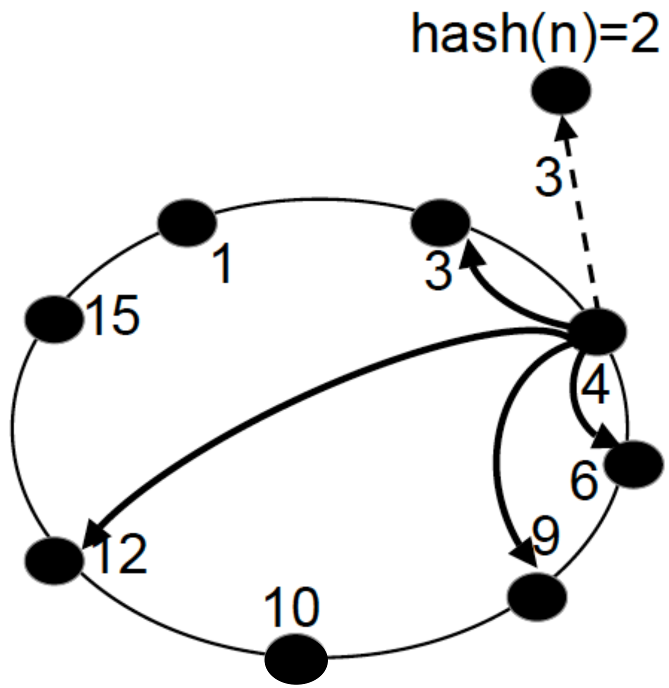
To join, a peer n must know (any) peer n' already in the network

Procedure **n.join(n')**:

s = n'.findSuccessor(n);

buildFingers(s);

successor=s;



Peers joining in Chord

To join, a peer n must know (any) peer n' already in the network

Procedure **n.join(n')**:

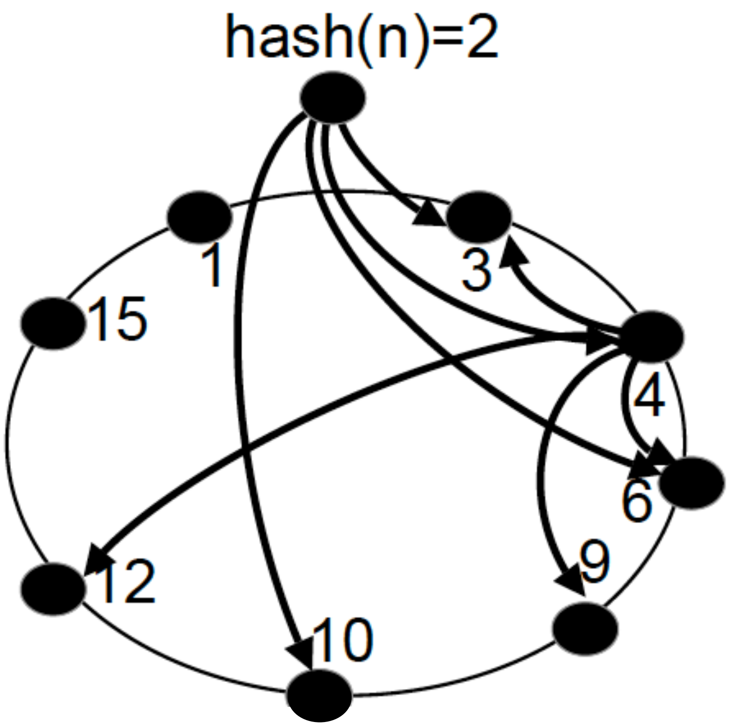
s = n'.findSuccessor(n);

buildFingers(s);

successor=s;

If 3 had some key-value pairs for the key 2, 3 gives them over to 2

The network is not *stabilized* yet…



Network stabilization in Chord

Each peer periodically runs stabilize()

n.stabilize():

x = n.succ().pred()

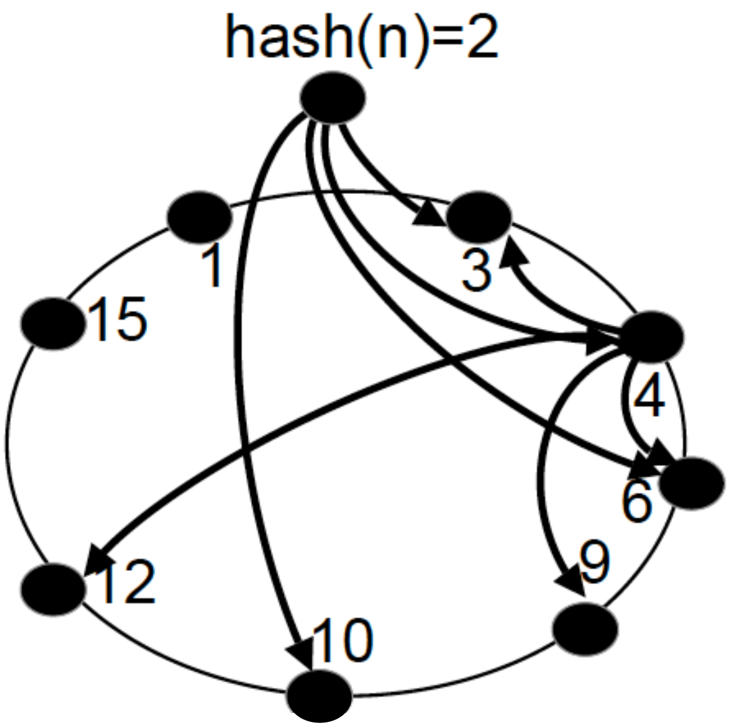
if (n < x < succ) then succ = x;

succ.notify(n)

n.notify(p):

if (pred < p < n)

then pred = p



Network stabilization in Chord

First stabilize() of 2: 3 learns its new predecessor

n.stabilize():

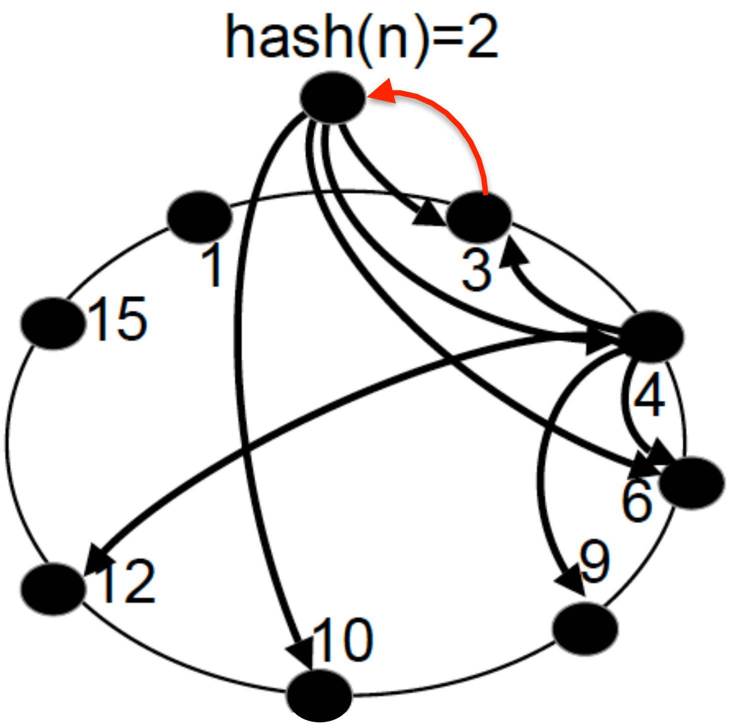
x = n.succ().pred()

if (n < x < succ) then succ = x;

succ.notify(n)

n.notify(p):

if (pred < p < n)
then pred = p



Network stabilization in Chord

First stabilize() of 1: 1 and 2 connect

n.stabilize():

x = n.succ().pred()

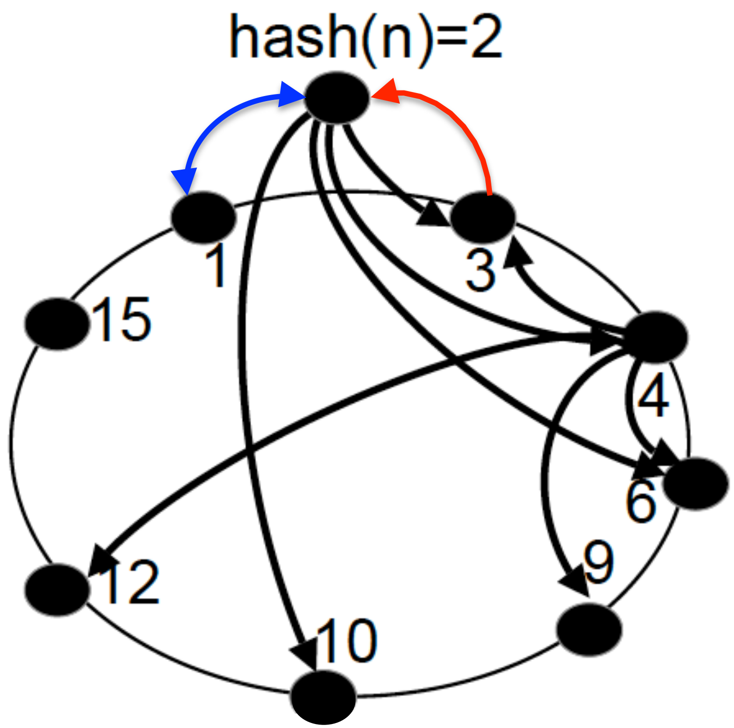
if (n < x < succ) then succ = x;

succ.notify(n)

n.notify(p):

if (pred < p < n)

then pred = p



Peer leaving the network

- The peer leaves (with some advance notice, « in good order »)

- Network adaptation to peer leave:
  - (key, value) pairs: those of the leaving peer are moved to its successor
  - Routing: P notifies successor and predecessor, which reconnect "over P"

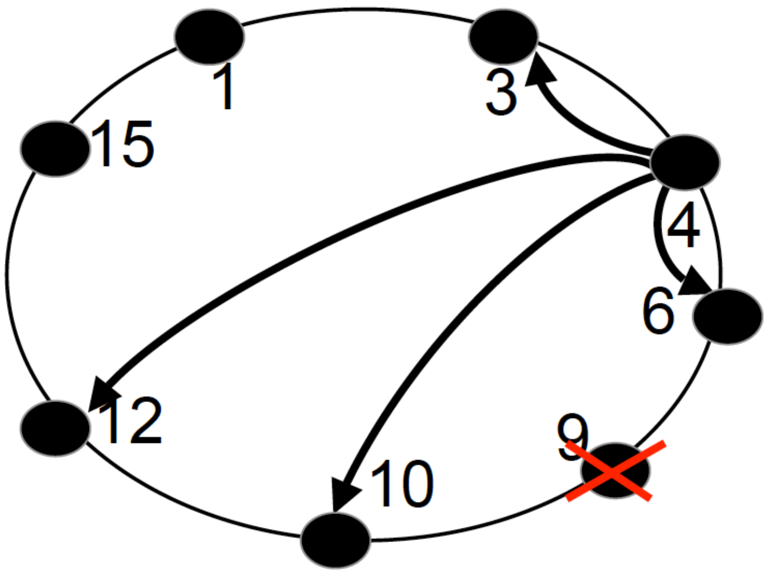
Peer failure

- Without warning
- In the absence of replication, the (key, value) pairs held on P are lost
  - Peers may also re-publish periodically

Example Running stab(), 6 notices 9 is down

6 replaces 9 with its next finger 10 →

all nodes have correct successors, but fingers are wrong



Routing still works, even if a little slowed down

Fingers must be recomputed

Peer failure

Chord uses successors to adjust to any change

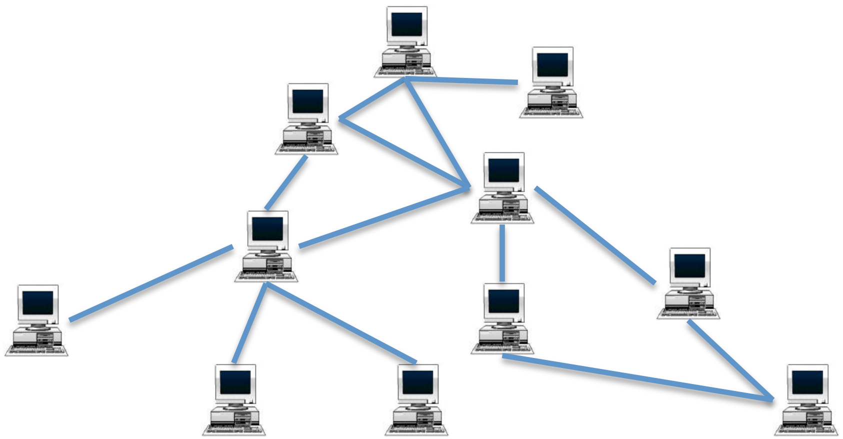
- Adjustment may « slowly propagate » along the ring, since it is relatively rare

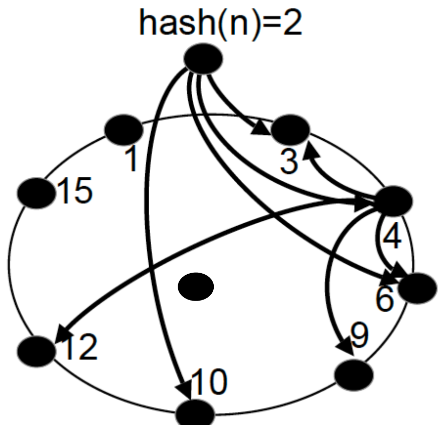
To prevent erroneous routing due to successor failure, each peer maintains a list of its *r* direct successors ($2\ log_2N$)

- When the first one fails, the next one is used...
- All *r* successors must fail simultaneously in order to disrupt search

Gossip in P2P architectures

- Constant, « background » communication between peers
- Structured or unstructured networks
- Disseminates information about peer network, peer data



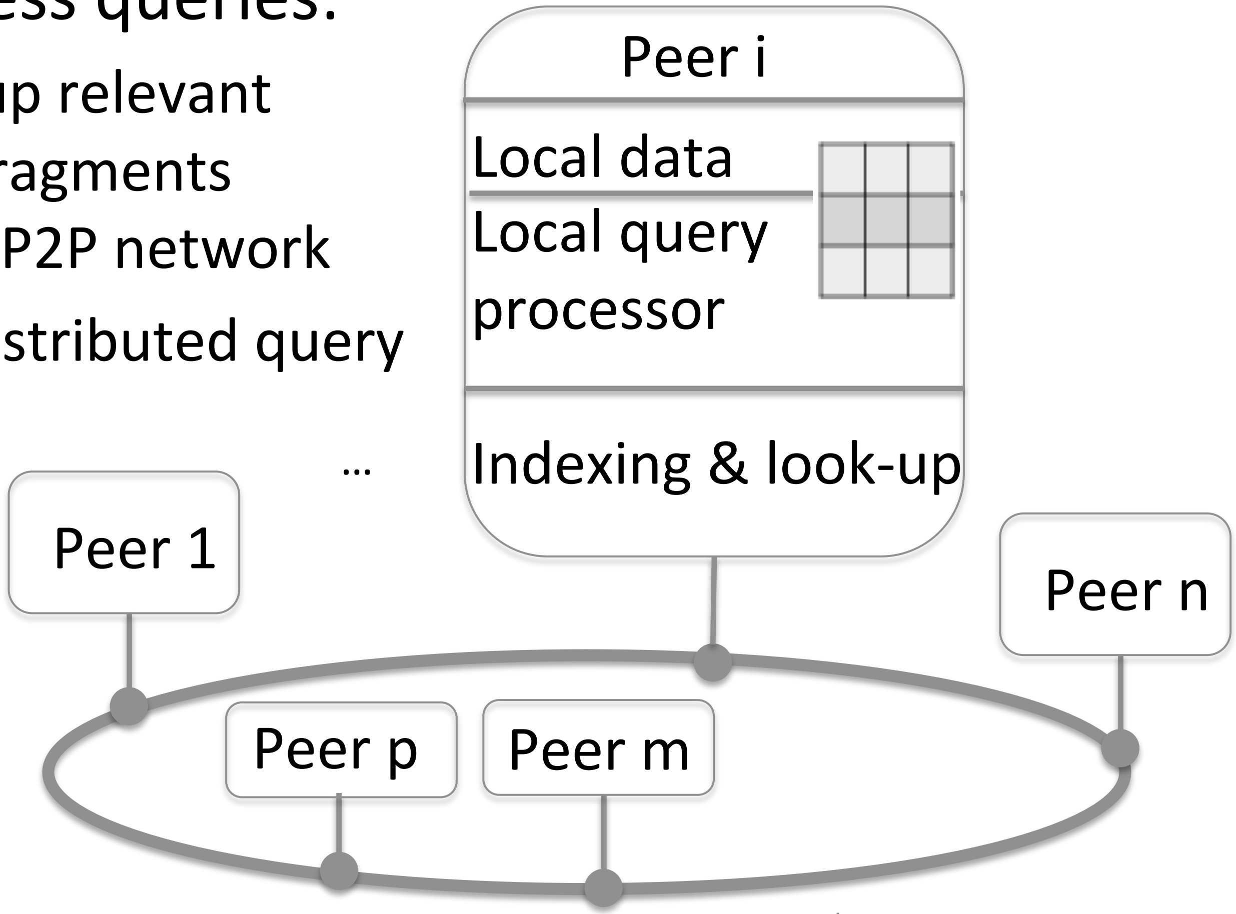


Peer-to-peer networks: wrap-up

- **Data model**:
  - Catalog and search at a simple key level
- **Query language**: keys
- **Heterogeneity**: not the main issue
- **Control**:
  - peers are autonomous in storing and publishing
  - query processing through symetric algorithm (except for superpeers)

Peer-to-peer data management

- Extract key-value pairs from the data & index them
- To process queries:
  - Look up relevant data fragments in the P2P network
  - Run distributed query plan



Example: storing relational data in P2P data management platform

- Each peer stores a horizontal slice of a table
- Catalog **at the granularity of the table**:
  - Keys: table names, e.g. Singer, Song
  - Value: fragment description, e.g., peer1:postgres:sch1/Singer&u=u1&p=p1,
  - Query: select Singer.birthday
    from Singer, Song
    where Song.title= « Come Away » and
    Singer.sID=Song.singer
  - What can happen?
- Try other granularities

Modern P2P data management system: Cassandra

Cassandra

– Partitioned row store, fully symetric structured P2P architecture
– Based on the Dynamo K-V system [CHG+07]
– Some nesting; indexes. Queries: select, project.

Table **songs**:

id	song_order	album	artist	song_id	title
62c36092...	4	No One Rides for Free	Fu Manchu	7db1a490...	Ojo Rojo
62c36092...	3	Roll Away	Back Door Slam	2b09185b...	Outside Woman Blues
62c36092...	2	We Must Obey	Fu Manchu	8a172618...	Moving in Stereo
62c36092...	1	Tres Hombres	ZZ Top	a3e64f8f...	La Grange

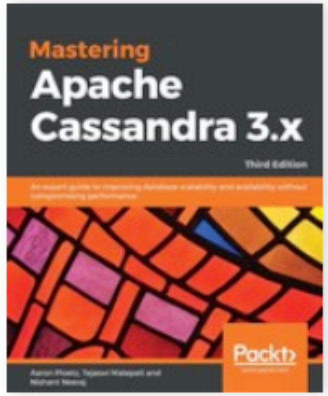
ALTER TABLE songs *ADD tags set<text>*;

UPDATE songs SET tags = tags + {'2007'} WHERE id = 8a172618...;
UPDATE songs SET tags = tags + {'covers'} WHERE id = 8a172618...;
UPDATE songs SET tags = tags + {'1973'} WHERE id = a3e64f8f-...;

SELECT id, tags from songs;

id	tags
7db1a490-5878-11e2-bcfd-0800200c9a66	{rock}
a3e64f8f-bd44-4f28-b8d9-6938726e34d4	{blues, 1973}
8a172618-b121-4136-bb10-f665cfc469eb	{2007, covers}

Modern P2P data management system: Cassandra



Large Cassandra deployments:

- Apple: over 75,000 nodes storing over 10 PB of data
- Netflix: 2,500 nodes, 420 TB, over 1 trillion requests per day

CAP trade-off: timeout for deciding when a node is dead

« *During gossip exchanges, every node maintains a **sliding window of inter-arrival times of gossip messages from other nodes in the cluster**. Configuring the phi_convict_threshold property adjusts the sensitivity of the failure detector. Lower values increase the likelihood that an unresponsive node will be marked as down, while higher values decrease the likelihood that transient failures causing node failure.*
*Use the default value for most situations, but **increase it to 10 or 12 for Amazon EC2** (due to frequently encountered network congestion) to help prevent false failures.*
*Values **higher than 12 and lower than 5** are not recommended.* »

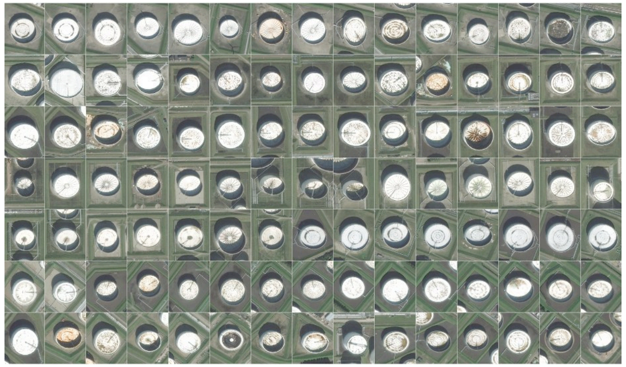
STRUCTURED DATA MANAGEMENT IN CLOUD ENVIRONMENTS

Cloud computing

- Idea: delegate **large-scale storage and large-scale computing** to remote centers
  - Run by the (only) enterprise using them: "private clouds"
    - Large companies can afford the cost to own and operate a cloud service: La Poste, Orange, ...
  - Run by a company who rents out storage and computing services: "commercial clouds"
    - Main players: Amazon (has basically created the industry), Google, Microsoft

Advantages of cloud computing

- Allow companies to **focus on their main business** not on IT
- Allow **scaling the resource usage** up and down according to the needs
- Comes at a **cost**



https://www.wired.com/2015/03/orbital-insight/

Examples:

- Satellite image data processing company which needs significant computing resources (only) when it has an order from a client
- Shops with more clients as Christmas approaches

How cloud services work (1/3)

- **Storage**
  - Users host files on trusted servers
  - The service is paid by the GB and day
    - Total cost = sum(file size x file storage time)
- **Computing**
  - Users buy virtual computers ("virtual machines")
  - Service paid by the durage of use of the VM
  - Each virtual computer is hosted by some physical computer in the cloud provider's cluster
  - If a physical machine fails, the virtual machine will be recreated elsewhere and the work will restart

How cloud services work (2/3)

- **Computing** (continued)
  - There are typically different sizes (capacities) of virtual machines
    - Small (**S**), Medium (**M**), Large (**L**), Extra-Large (**XL**)
    - The difference is in the *computing speed*
- **Fast storage of small-granularity data**, typically in memory in the cloud
  - For: metadata (catalog, user management, ...)
  - Key-value stores, document stores
  - Pay per operation (put, get)
- Other services
  - E.g. messaging **queues** to synchronize different applications

How cloud services work (3/3): cloud computing models

- **Infrastructure-as-a-service**
  - The vendor provides access to computing resources such as servers, storage and networking.
  - Clients use their own platforms and applications within a service provider's infrastructure.
    They do not host but they develop, deploy and administer in the cloud.
- **Platform-as-a-service**
  - The vendor provides: storage and other computing resources, prebuilt tools to develop, customize and test their own applications.
  - Clients do not host and mostly do not administer either. They still develop and deploy in the cloud.

How cloud services work (3/3): cloud computing models

- **Software-as-a-service**
  - The vendor provides: storage and other computing resources; software and applications via a subscription model (or pay-per-use...)
  - Clients access the applications remotely.
    They do not store, host, develop nor administer.

Cloud services

	amazon web services™	Google Cloud Platform	Windows Azure
File storage service	Amazon Scalable Storage Service (S3)	Google Cloud Storage	Windows Azure BLOB Storage
Virtual machines	Amazon Elastic Compute Cloud (EC2)	Google Compute Engine	Windows Azure Virtual Machines
Fine-granularity data store	Amazon DynamoDB	Google High Replication Datastore	Windows Azure Tables
Queue service	Amazon Simple Queue Service (SQS)	Google Task Queues	Windows Azure Queues

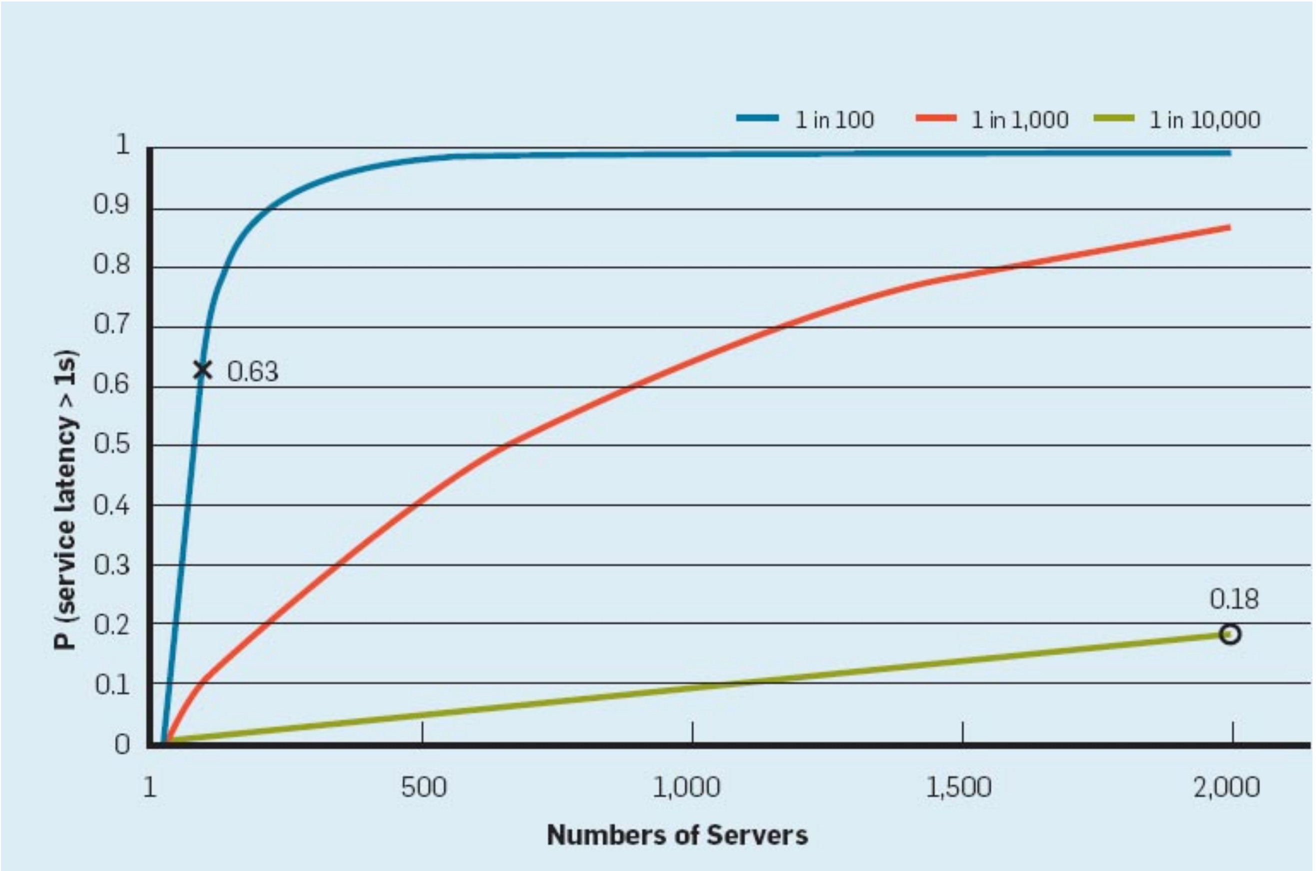
Performance in large-scale clusters

- On-site (within a company or organization) or off-site (cloud)
- There may be **variable latency** across the cluster, i.e. some machine(s) may temporarily be slow, due to
  - Shared resources (CPU, cache, memory, network)
  - Maintenance, software upgrade
  - Global resource sharing, e.g., network switches, distributed file systems
  - Garbage collection
  - Energy management
- The variability gets **amplified by scale** (see next)

Latency variations in large-scale clusters

Consider a setting where each server responds
- in 10ms, 99% of the time
- in 1s, 1% of the time (1 in 100, blue curve)

If a client needs to talk to 100 servers, the probability of >1s latency is 63% !



Source: Dean and Barroso, "The Tail at Scale", Communications of ACM, 2013

Structured data management in cloud platforms

- The cloud provides:
  - Distributed file system; Virtual machines; Fine-granularity (e.g., key-value or document) store; Distributed message queues
- Based on this, need to propose architectures for:
  - Storing very large volumes of fine-granularity data (relational, XML, JSON, graphs...) and querying it
  - Transactions
  - Concurrency control

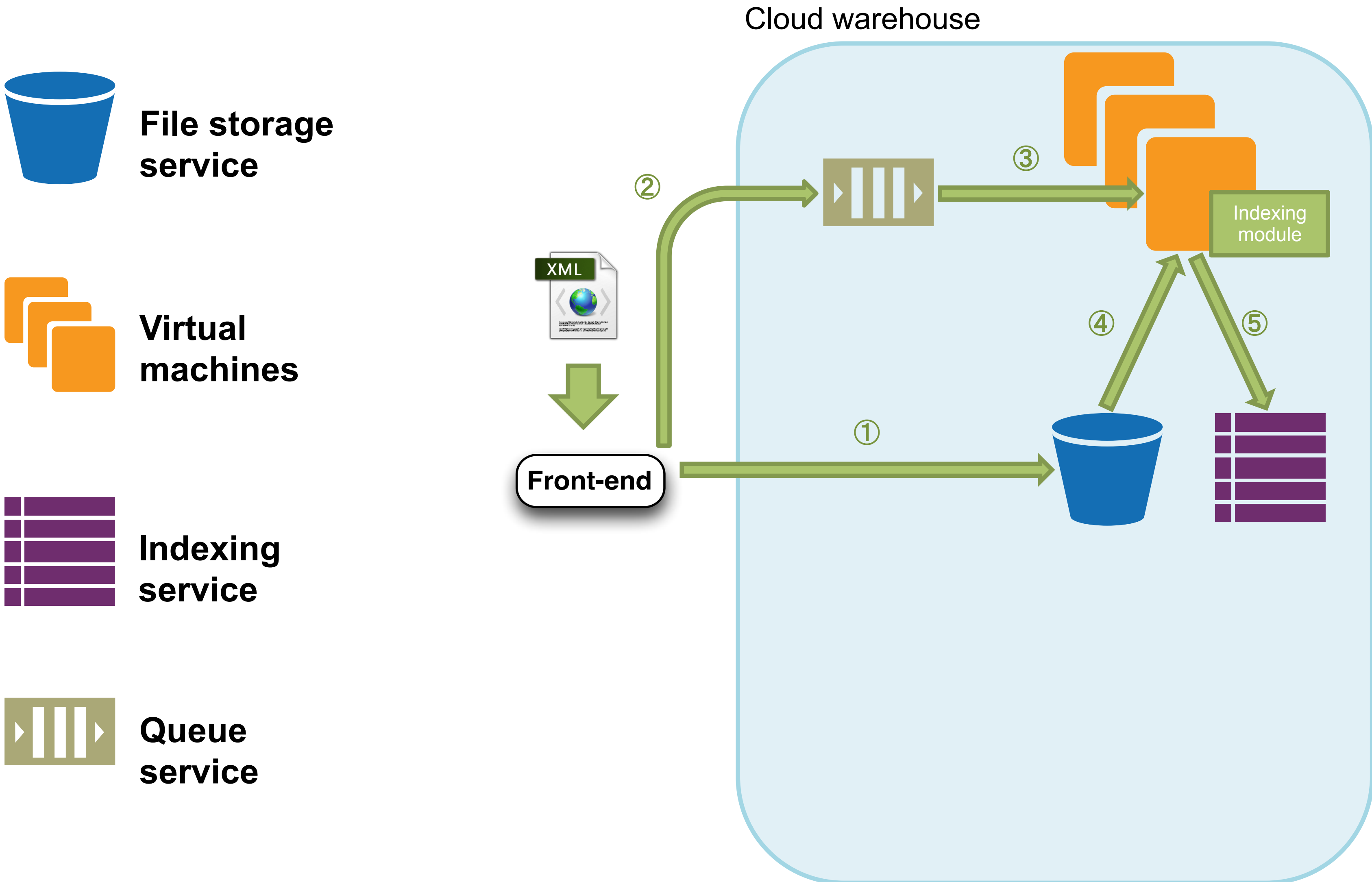
AMADA: XML data management within the Amazon cloud [CCM13]

	amazon web services™	Google Cloud Platform	Windows Azure
File storage service	Amazon Scalable Storage Service (S3)	Google Cloud Storage	Windows Azure BLOB Storage
Virtual machines	Amazon Elastic Compute Cloud (EC2)	Google Compute Engine	Windows Azure Virtual Machines
Fine-granularity data store	Amazon DynamoDB	Google High Replication Datastore	Windows Azure Tables
Queue service	Amazon Simple Queue Service (SQS)	Google Task Queues	Windows Azure Queues

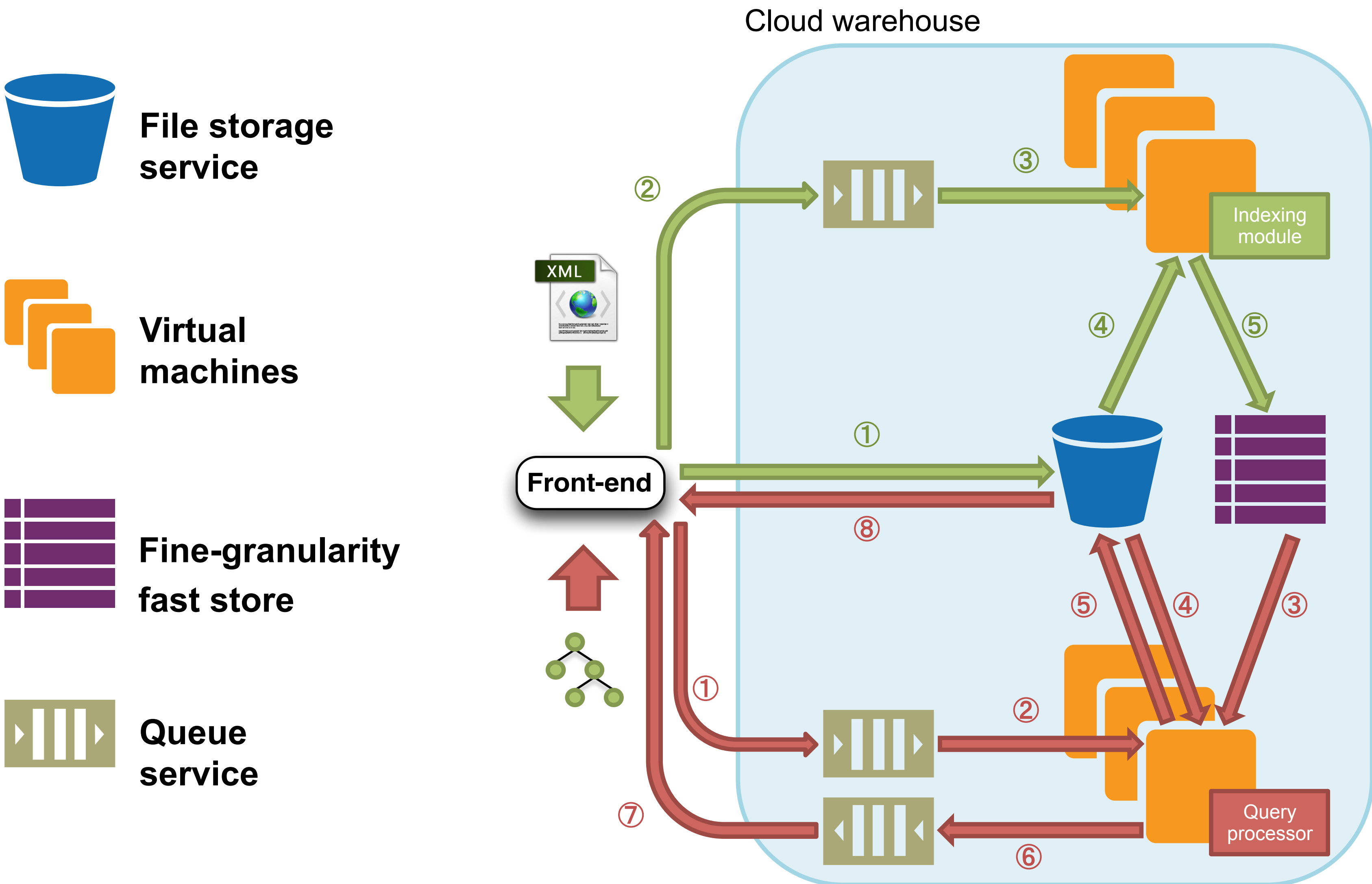
AMADA: XML data management within the Amazon cloud [CCM13]

- Functionality:
- XML and RDF storage and fine-grain indexing in the cloud
- Data storage:
  1. Blob storage in cloud file system (i.e., S3)
  2. Fine-granularity indexing in k-v store
- Data querying:
  1. Consult the index to delimit the documents (graphs) which must be accessed
  2. Evaluate query over relevant documents (graphs)

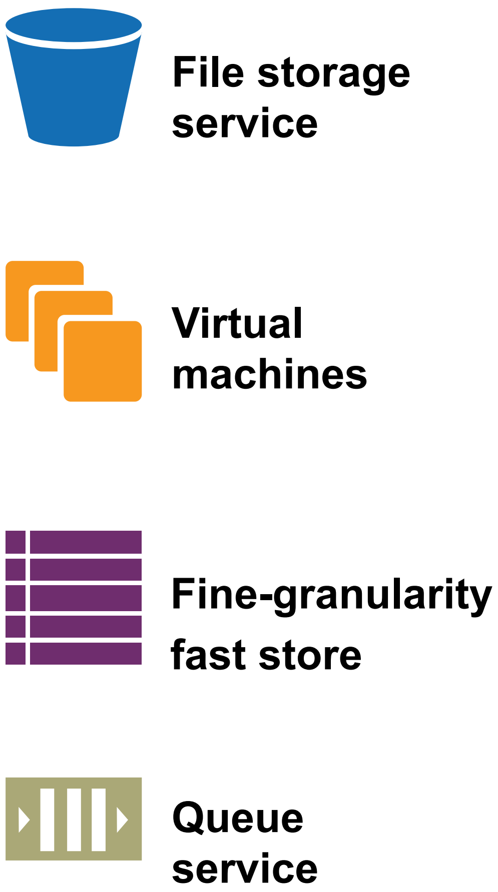
AMADA: XML data management within the Amazon cloud [CCM13]

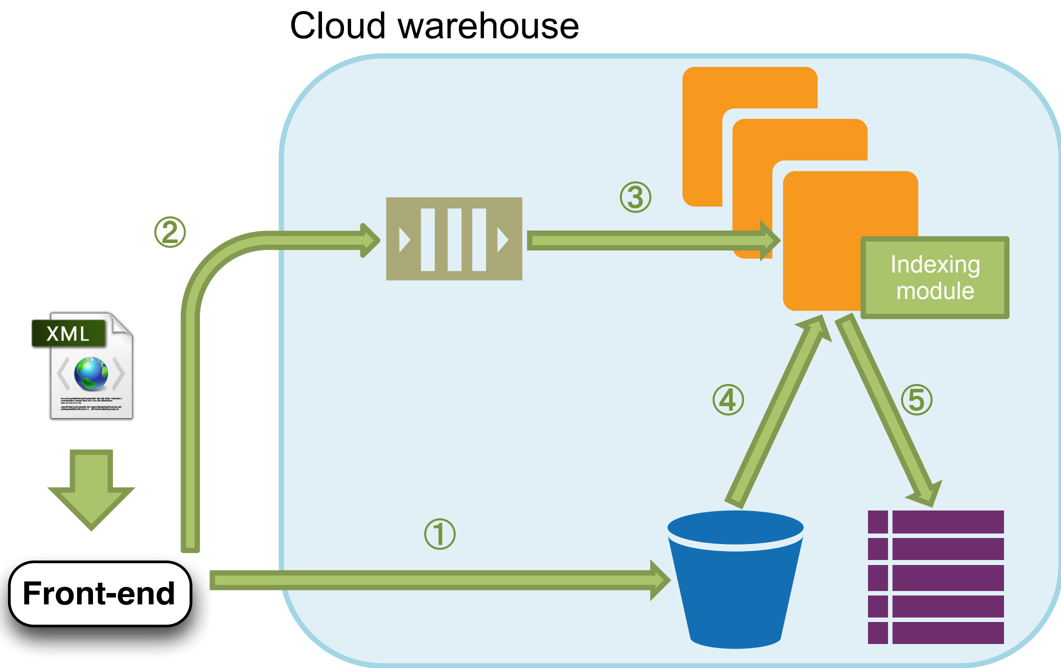


AMADA architecture



Indexing and storage costs



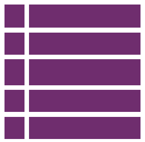


Indexing cost depends on:
- Documents set (*D*)
- Indexing strategy (*I*)

Indexing and storage costs

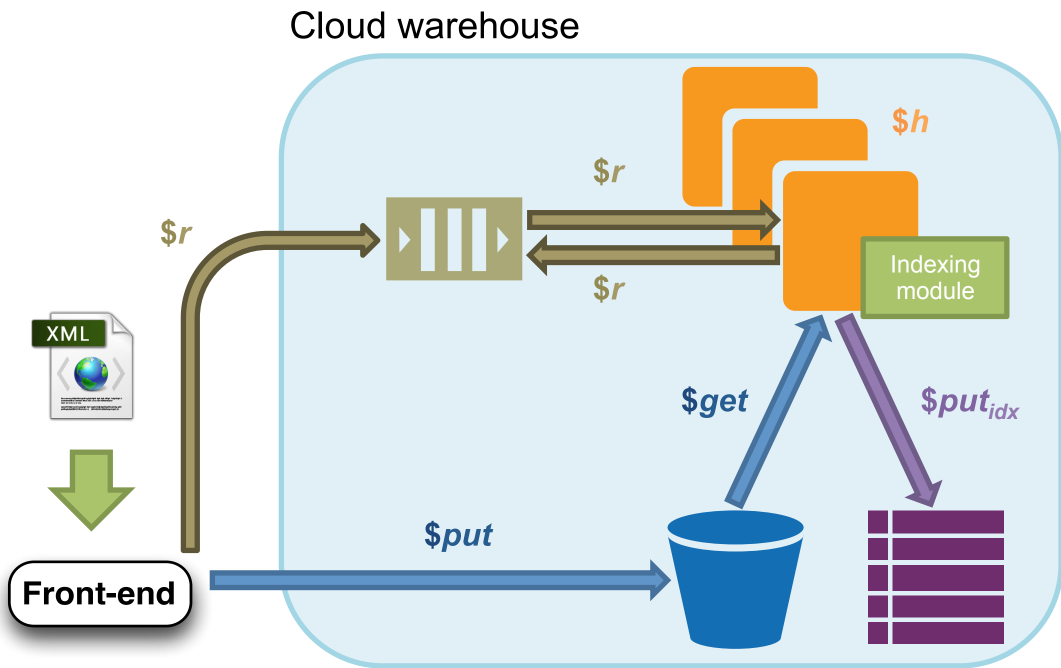
File storage service

Virtual machines



Fine-granularity fast store

Queue service



Storage cost

Querying cost

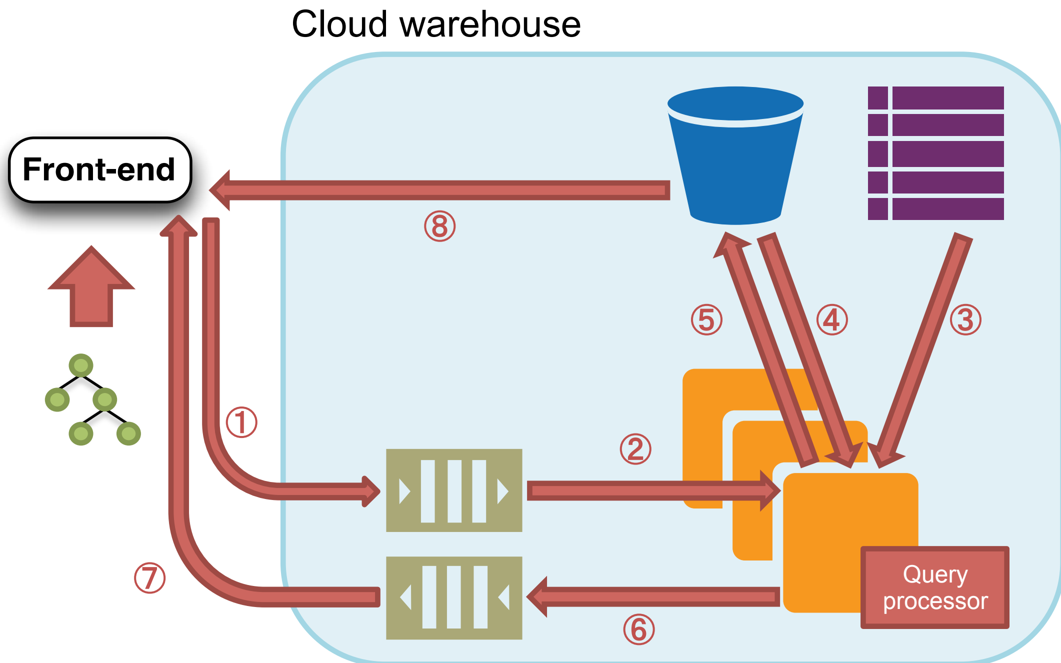
File storage service

Virtual machines

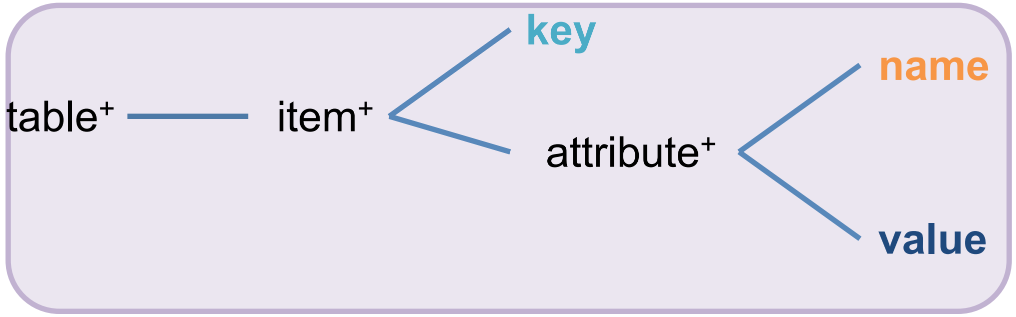
Fine-granularity fast store

Queue service

Querying cost depends on:
- Query (*q*)
- Documents set (*D*)
- Indexing strategy (*I*)

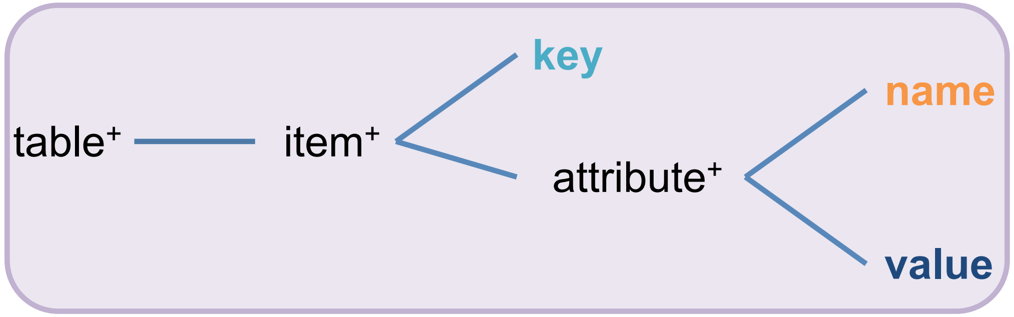


Fine-granularity fast store in Amazon cloud: DynamoDB



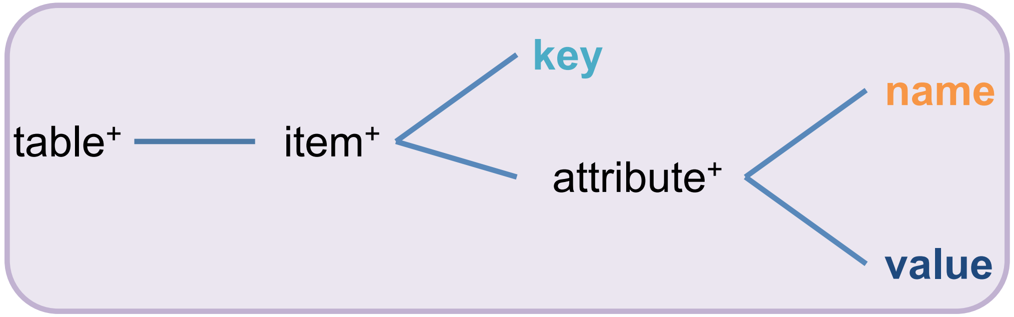
Indexing fine-granularity data in the Amazon cloud

DynamoDB data model



Indexing strategy *I*: Function associating (**key**,(**name**,**value**)$^+$)$^+$ to a document

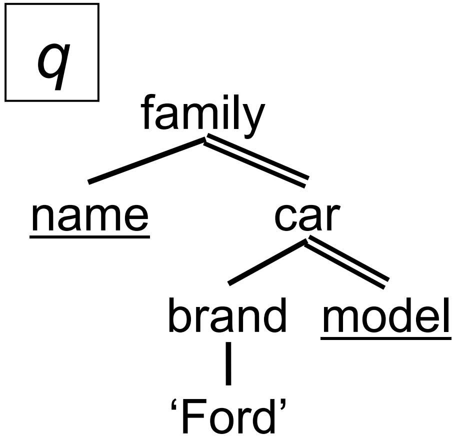
Indexing fine-granularity data in the Amazon cloud

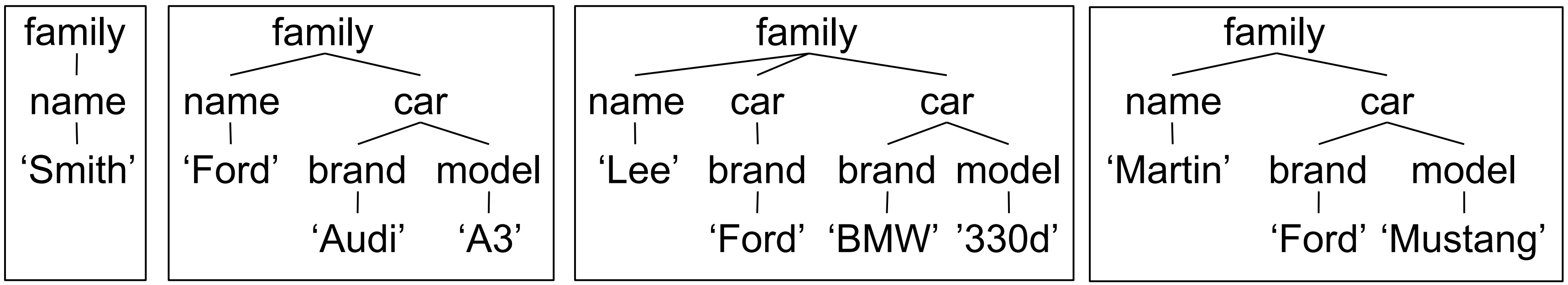


Indexing strategy *I*: Function associating (**key**,(**name**,**value**)$^+$)$^+$ to a document

- Four indexing strategies
  - *Label-URI (LU)*
  - *Label-URI-Path (LUP)*
  - *Label-URI-ID (LUI)*
  - *Label-URI-Path/Label-URI-ID (2LUPI)*

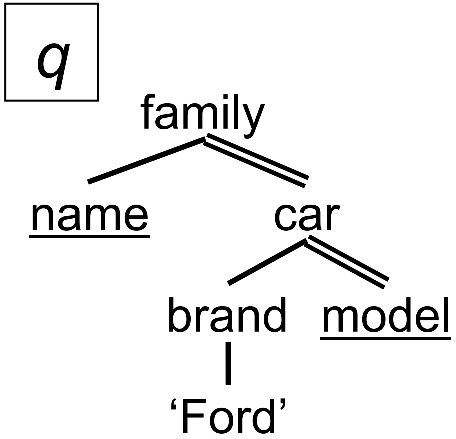
XML indexing: example





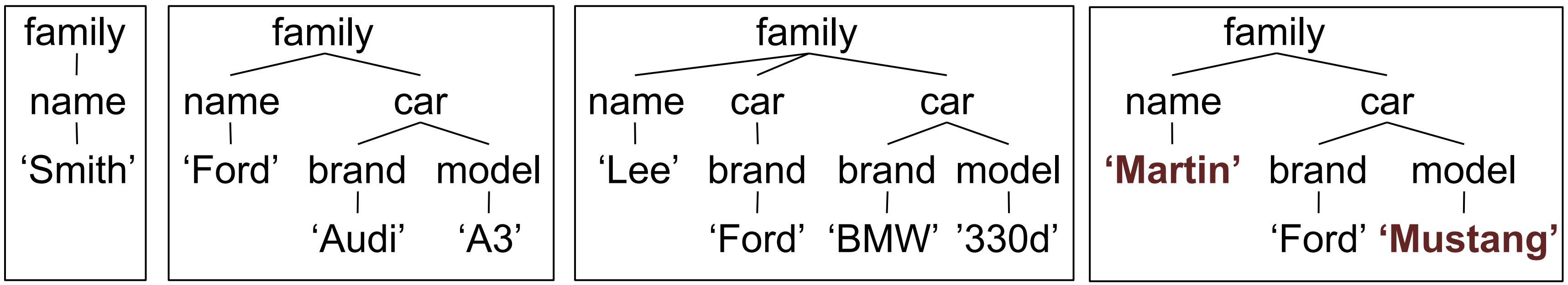
doc1.xml *doc2.xml* *doc3.xml* *doc4.xml*

XML indexing: example



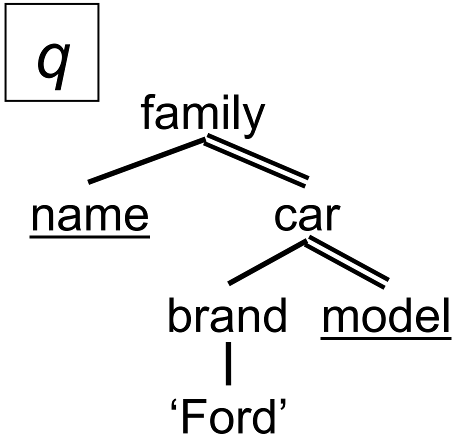
Result:

doc4.xml	**Martin**	**Mustang**



doc1.xml *doc2.xml* *doc3.xml* ***doc4.xml***

Label-URI (LU) strategy



Index: (key, (name, value))

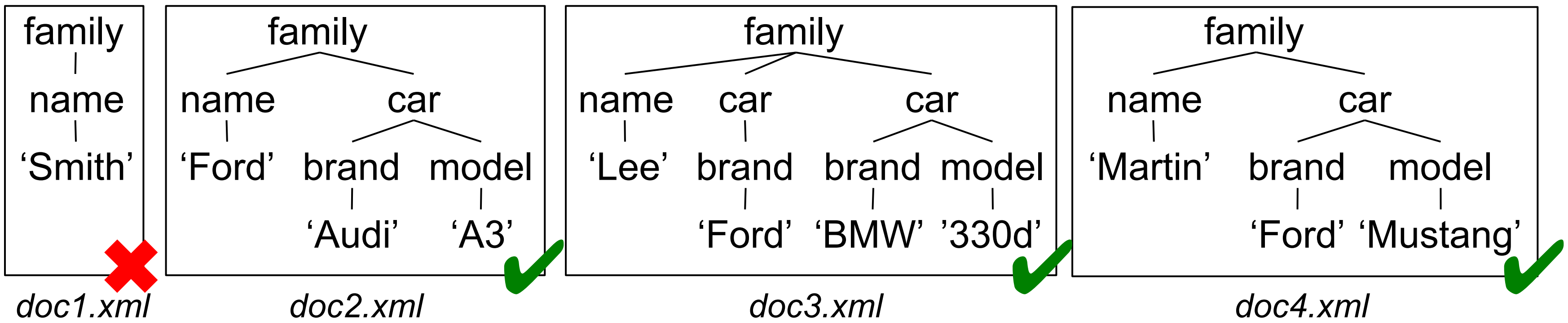
efamily	doc1.xml	doc2.xml	doc3.xml	doc4.xml
	Ø	Ø	Ø	Ø

ename	doc1.xml	doc2.xml	doc3.xml	doc4.xml
	Ø	Ø	Ø	Ø

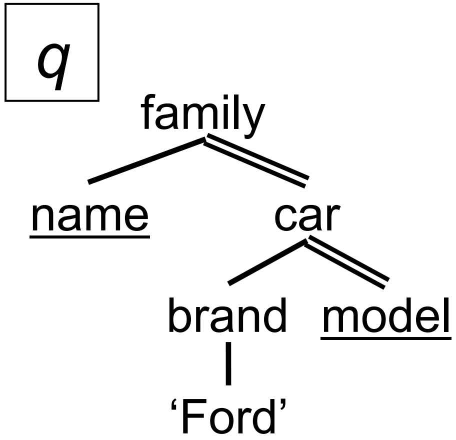
wFord	doc2.xml	doc3.xml	doc4.xml
	Ø	Ø	Ø

...

Look-up: Intersection of URI sets associated to each query node



Label-URI-ID (LUI) strategy



Index: (key, (name, value))

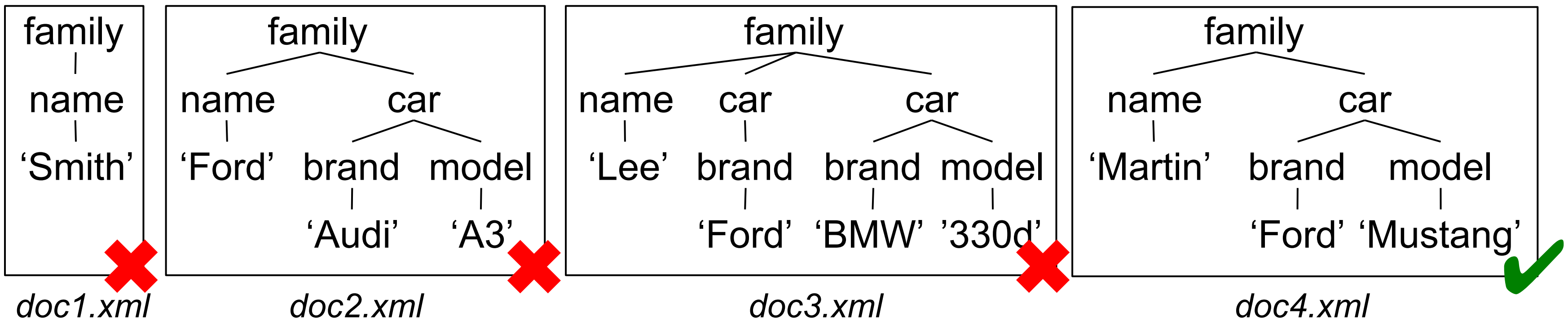
efamily	doc1.xml	doc2.xml	doc3.xml	doc4.xml
	[1 3 0]	[1 8 0]	[1 11 0]	[1 8 0]

ename	doc1.xml	doc2.xml	doc3.xml	doc4.xml
	[2 2 1]	[2 2 1]	[2 2 1]	[2 2 1]

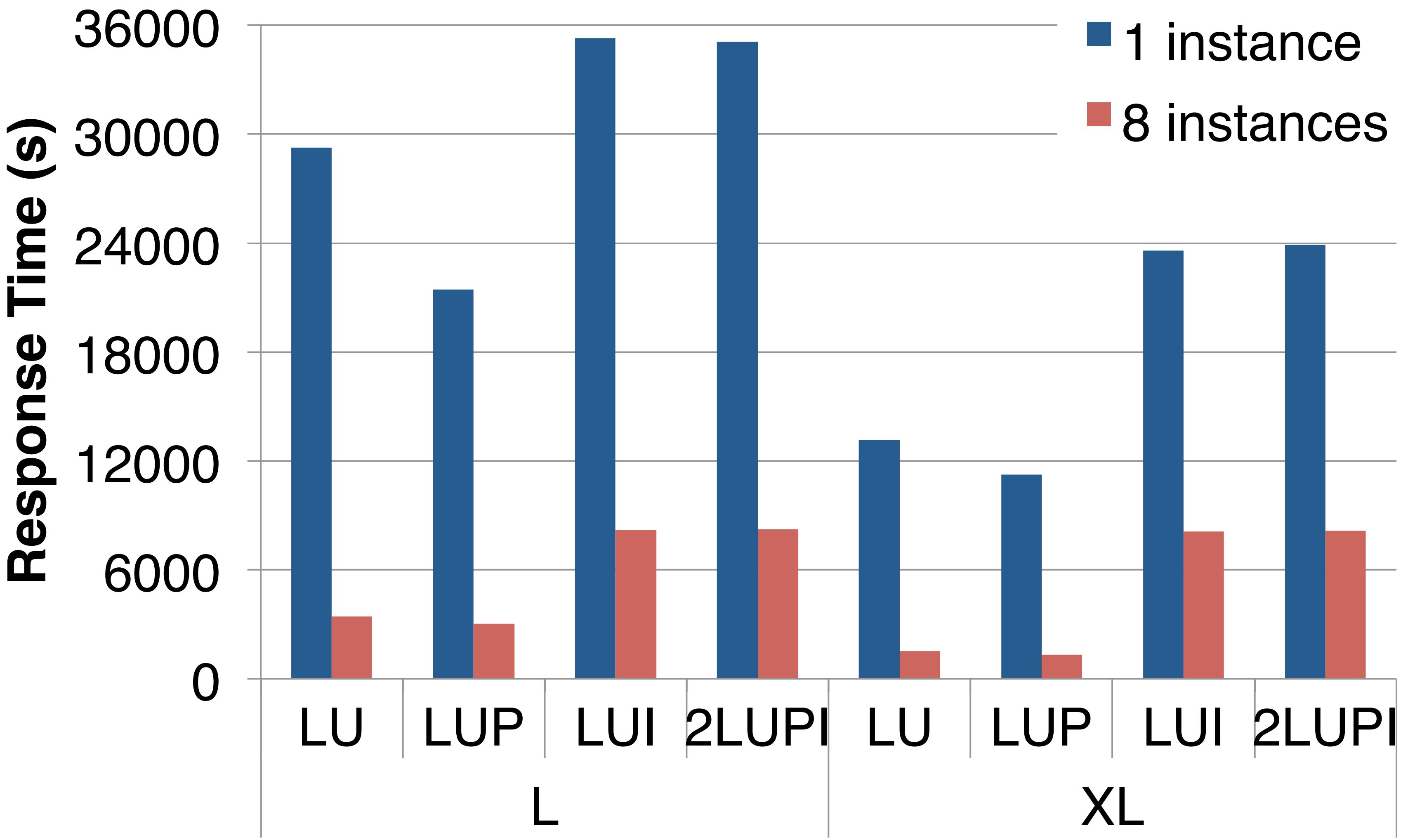
wFord	doc2.xml	doc3.xml	doc4.xml
	[3 1 2]	[6 3 3]	[6 3 2]

…

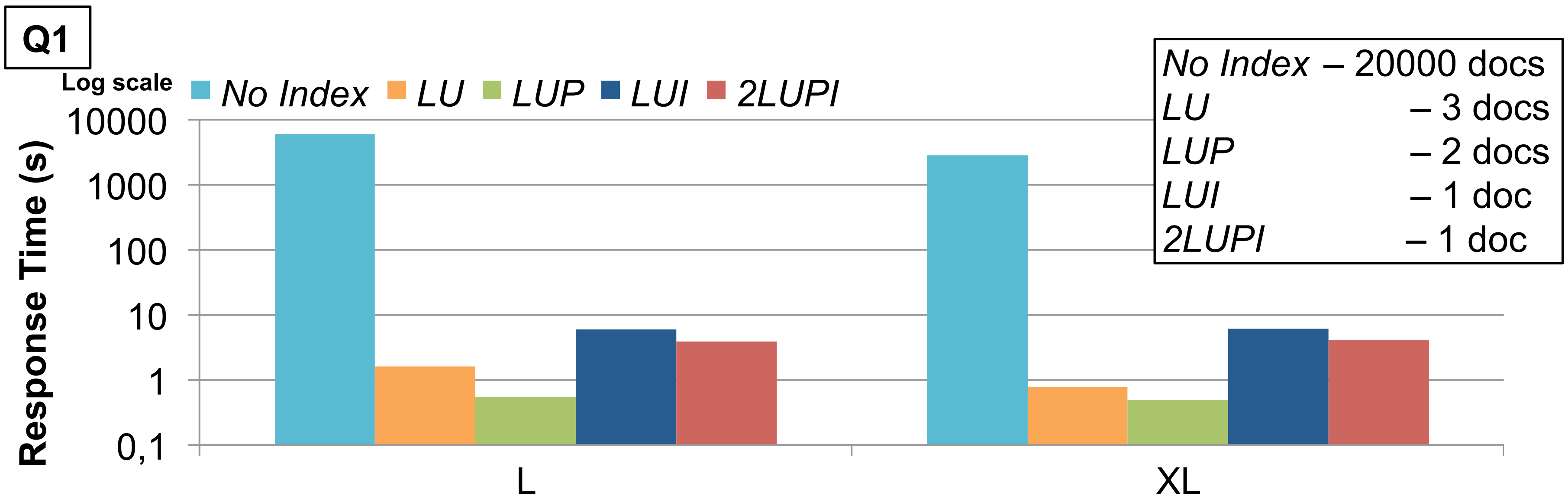
Look-up: Structural join over IDs associated to each query node



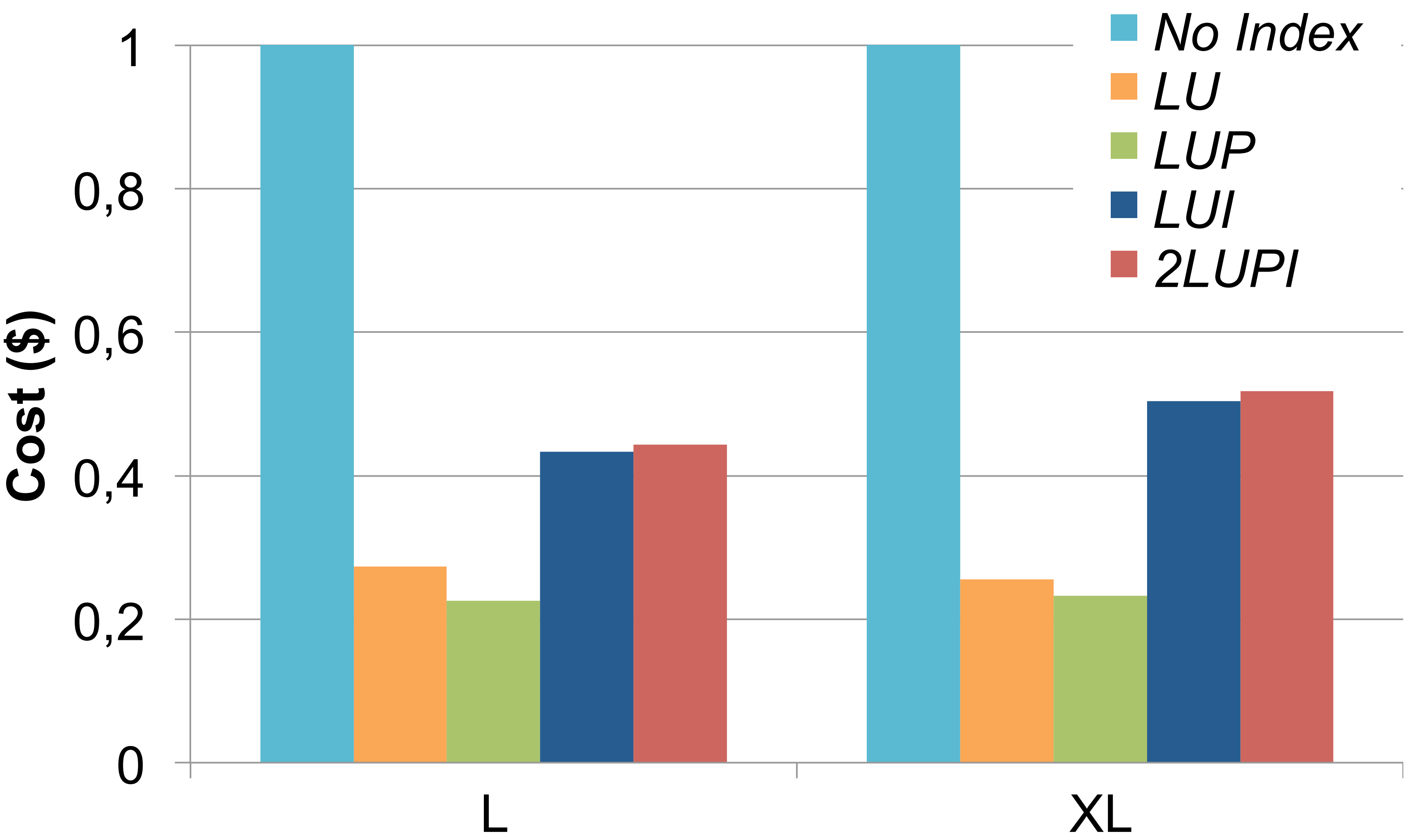
Query answering (eight runs)



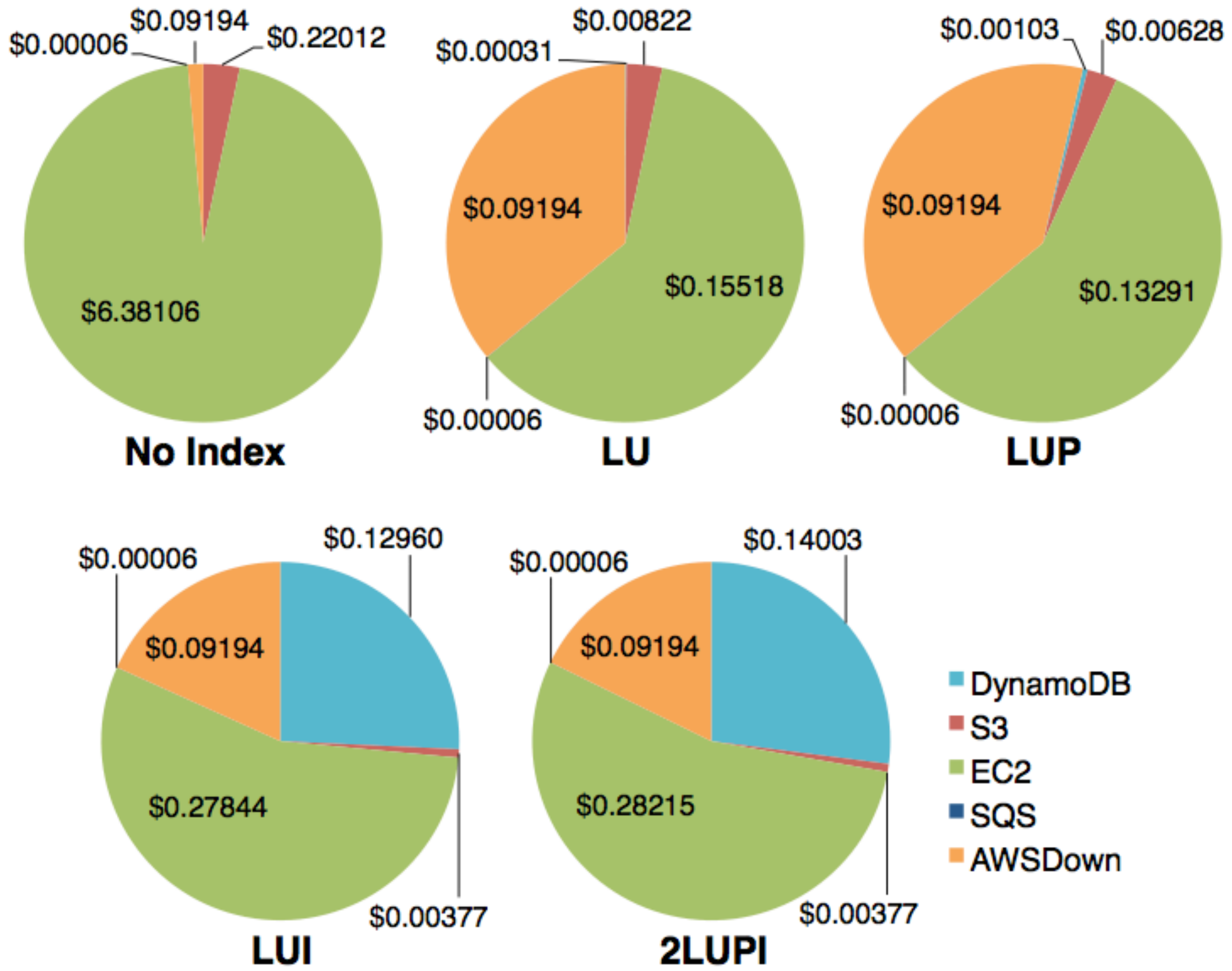
Query answering time



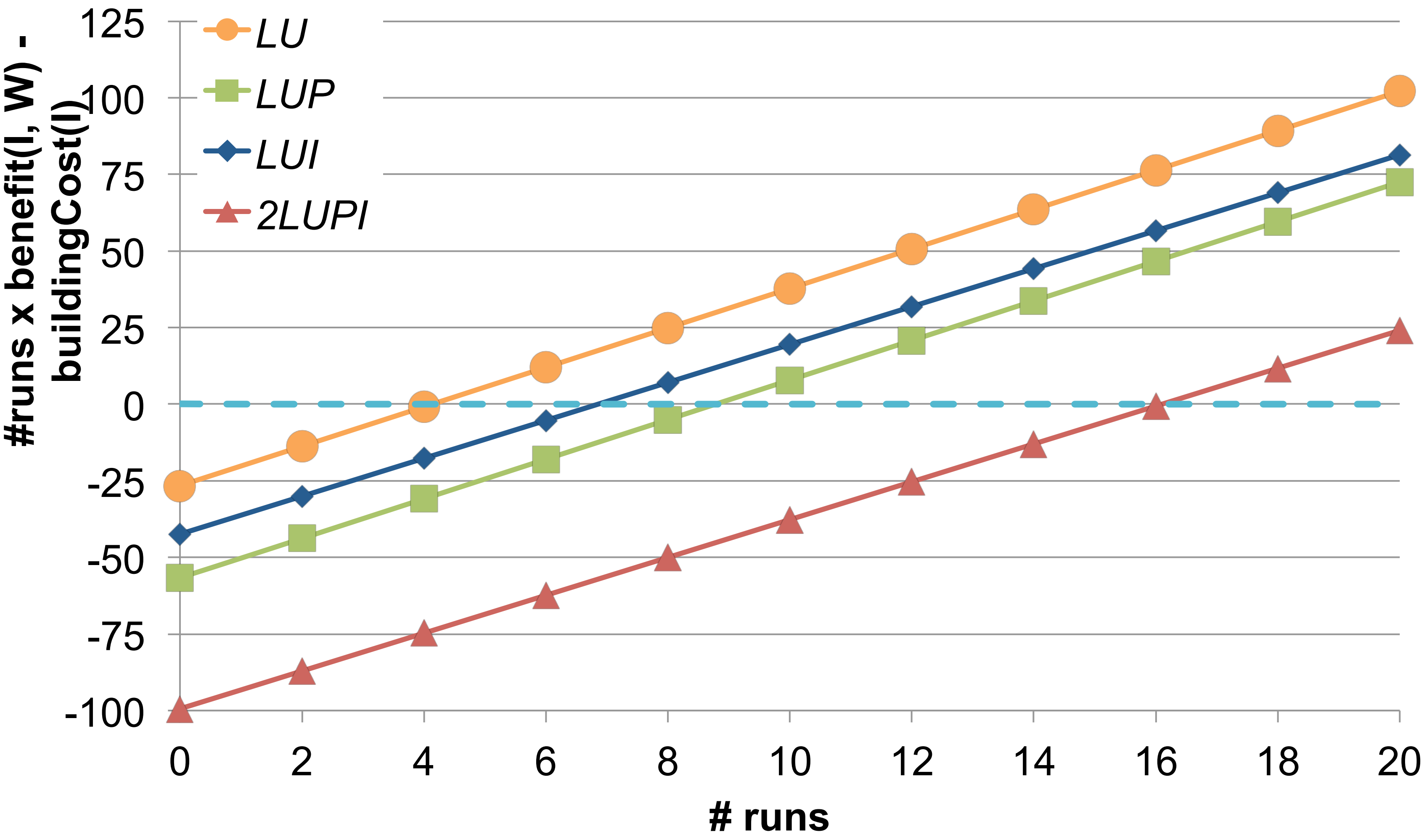
Query answering <u>cost</u>



Query answering cost detail (XL)



Index cost amortization



Cloud Data Warehouse Services

- Software-as-a-Service (SaaS) solutions
- Snowflake in the Amazon Cloud
- Google BigQuery (https://cloud.google.com/bigquery/)
- Amazon Redshift (https://aws.amazon.com/redshift/)
- Microsoft Azure SQL Data Warehouse (https://azure.microsoft.com/)

Cloud Data Warehouse Services

The need: efficient data processing at very large scale → distributed system

Previous solution: share-nothing architectures (MapReduce or Spark clusters)

- Each node stores some data and computes on it
- *Storage and computations are distributed at the same time*

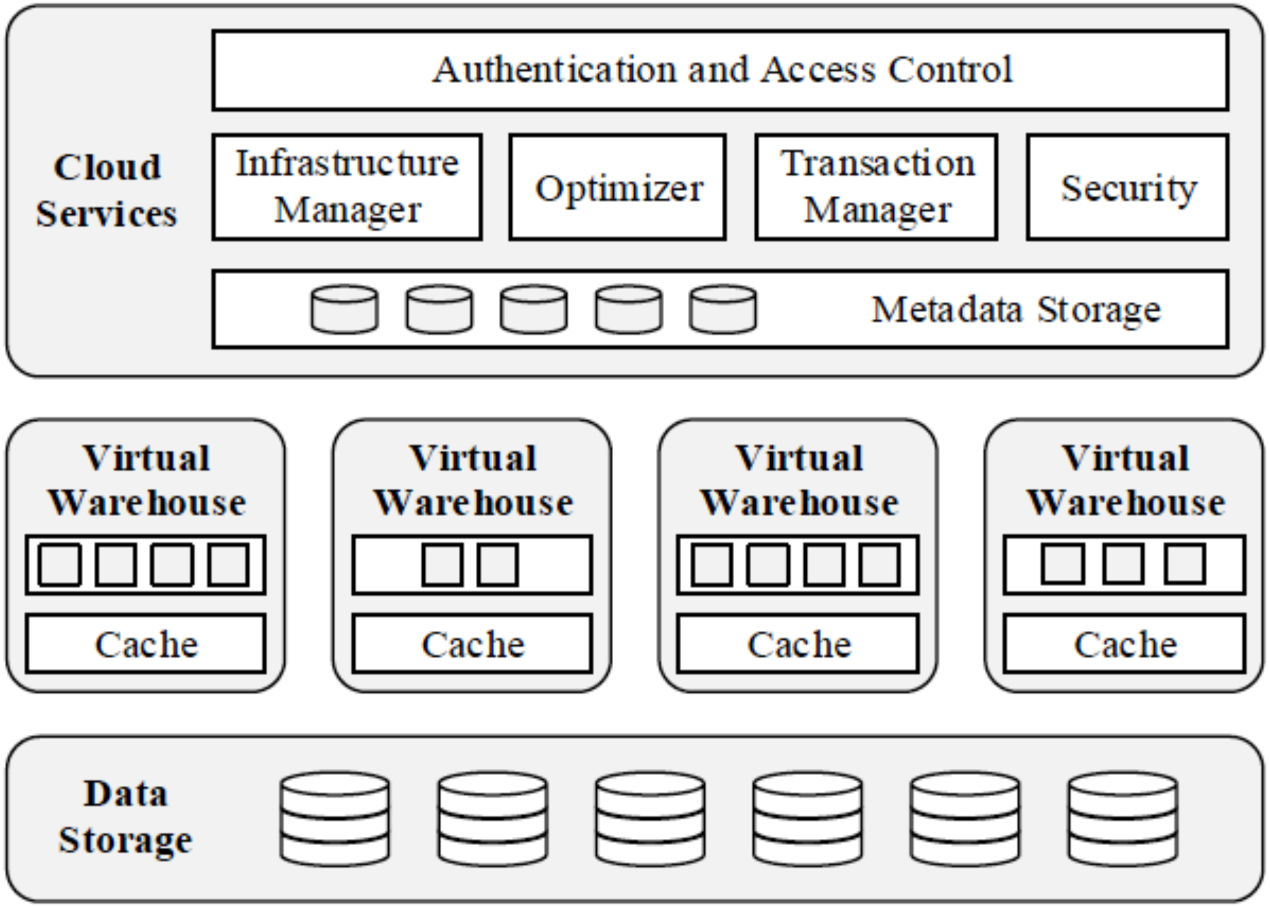
Limitations:

1. Heterogeneous workload, e.g. bulk loading (high I/O, little to no CPU)... intensive computing (little to no I/O, high CPU) → hard to chose machines!
2. Membership changes are frequent in the cloud, lead to data shuffles and negatively impact performance
3. Hard to do online upgrade in symetric, homogeneous architecture

Snowflake: separating storage from computations in the cloud [DGZ+16]

- **Store data** in the Amazon S3 service.
- **Store metadata** (catalog, user management, ...) in high-performance, transactional key-value store
- **Perform computations** in virtual warehouses (Snowflake proprietary software)
  - A virtual warehouse (VW) has a set of workers. It is created on demand by a Snowflake client.
  - A client can have many VWs.
  - Each query is handled within one VW.

Snowflake architecture



Proprietary SQL engine (with a few more goodies) in each Virtual Warehouse

Snowflake Virtual Warehouses

- Every VW has access to the same data (from S3)
- A worker belongs to exactly one VW
- Virtual Warehouses come in "T-shirt sizes" (XS to XXL)
  - Hides cloud provider nodes (even their number) from Snowflake client
  - Allows independent Snowflake pricing policy
  - Allows "smart" investment of cloud services cost: for a load task, book a VW of (4 nodes for a 15h) vs. (32 nodes for 2h).

Snowflake Virtual Warehouses

- S3 much slower access than the local disk of a node → **a worker's local disk acts as cache**
  – LRU caching of the data last needed on this node
  – Cache granularity: **table columns**
- To avoid redundant caching of the same data fragments across the nodes of a single VW, the Optimizer assigns files (table) cache data to nodes using **hashing on the the (table, column) name**
  – If this data is cached, it is only cached on a specific VW node

Query evaluation in Snowflake

1. Selective data access
   - Each table is stored as as set of **shards**
   - Inside each shard, data is stored **as a set of (compressed) columns**
   - **Headers** built for each column within the shard
     - Minimum and maximum values
     - No need to read a shard if the query predicate is incompatible with the header information

2. Query optimizer
   - Cost- and statistic-based
   - Headers computed even on intermediary results
   - Some decisions taken at runtime

3. Intermediary query results written in node local disks, then (if needed) to S3

Concurrency control in Snowflake

- Handled globally using fine-granularity data store
- An update creates a new version of a table (multi version concurrency control): no finer-granularity update
- Each version has a timestamp
- Possible to explicitly query the version at or after a certain timestamp
- Each version stays available 90 days after deletion

GLOBAL COURSE WRAP-UP

Big Data Architectures

- Modern development of distributed computing platforms leads to unprecedented explosion in Big Data architectures and systems
- Dimensions of analysis:
  - Scale of distribution
  - Data model, query language they support
  - CAP compromise (which level of consistency)
  - Performance influenced by:
    - Data storage, indexing, query evaluation algorithms...
    - Query optimization
    - Synchronization operations
  - Success influenced by performance, ease of use, killer applications

References

- [CCM13] Web data indexing in the cloud: efficiency and cost reductions. Jesus Camacho-Rodriguez, Dario Colazzo, Ioana Manolescu. EDBT, 2013
- [CHC+07] G. De Candia, D. Hastorun, M. Jampani et al. Dynamo: Amazon's highly available key-value store, ACM SOSP 2007
- [DGZ+16] The Snowflake Elastic Data Warehouse. Benoît Dageville, Thierry Cruanes, Marcin Zukowski et al., SIGMOD, 2016
- [SMK+01] Chord: A scalable peer-to-peer lookup service for internet applications, I. Stoica, R. Morris, F. Kaashoek et al. ACM SIGCOMM Computer Communication Review, 2001