

Architectures for massive data management (part 2)

Ioana Manolescu

INRIA Saclay

ioana.manolescu@inria.fr

<http://pages.saclay.inria.fr/ioana.manolescu/>

M2 Data and Knowledge
Université de Paris Saclay

Architectures for Massive DM
D&K / UPSay 2015-2016

Ioana Manolescu

1

Dimensions of distributed 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)

Architectures for Massive DM
D&K / UPSay 2015-2016

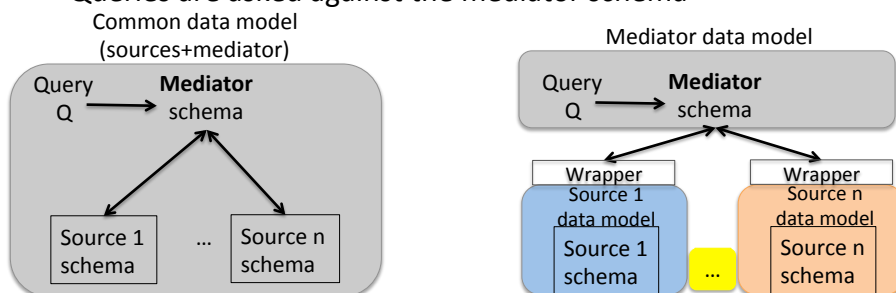
Ioana Manolescu

2

MEDIATOR SYSTEMS

Mediator systems

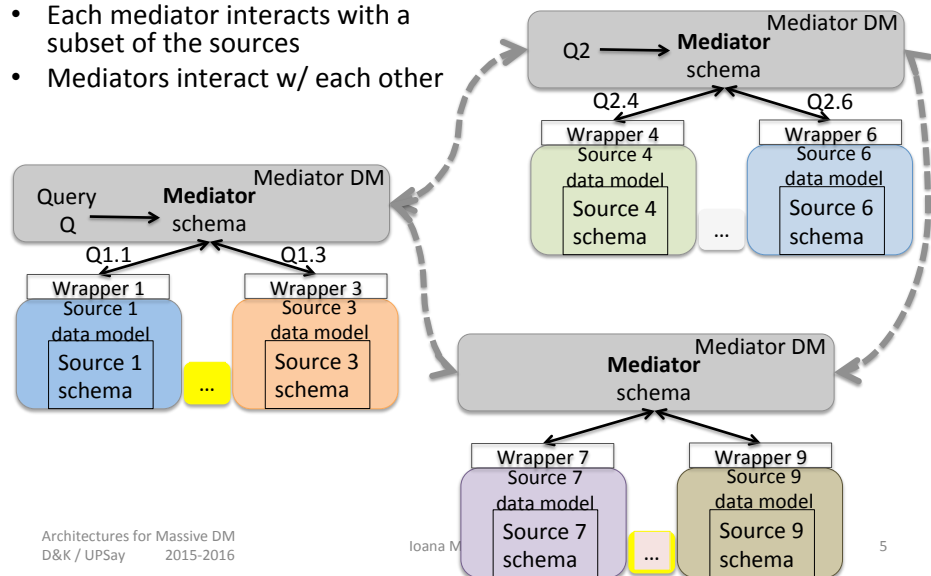
- A set of **data sources**, of the same or different data model, query language; source schemas
- A **mediator** data model and mediator schema
- Queries are asked against the mediator schema



- **ACID:** mostly read-only; **size:** small
- **Control:** Independent publishing; mediator-driven integration

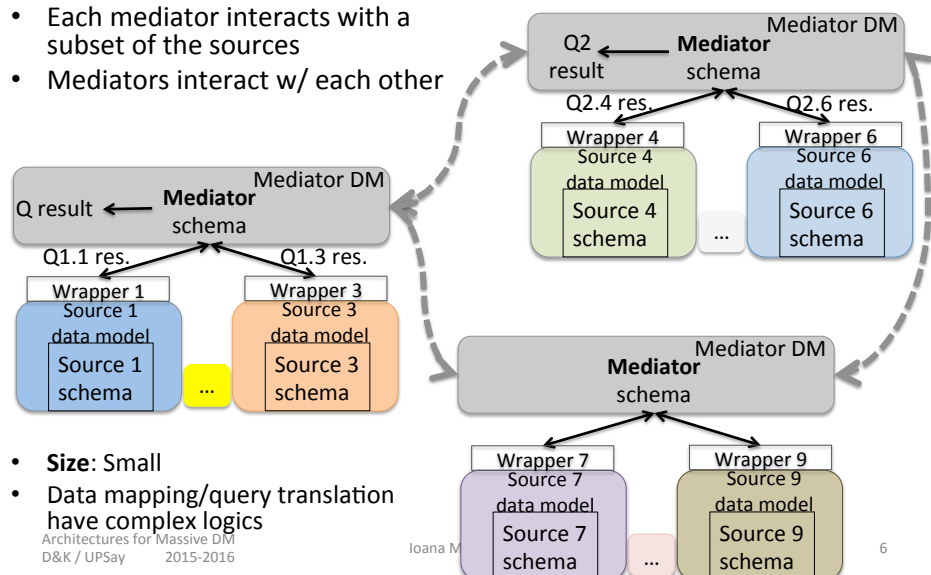
Many-mediator systems

- Each mediator interacts with a subset of the sources
- Mediators interact w/ each other



Many-mediator systems

- Each mediator interacts with a subset of the sources
- Mediators interact w/ each other



Integration approach in mediator systems

- Global-as-view:
 - Mediator (global) schema defined as view based on the source schemas
 - Query over the global schema requires view unfolding
- Local-as-view:
 - Source (global) schema defined as views over the mediator schema
 - Query over the global schema requires query rewriting using views

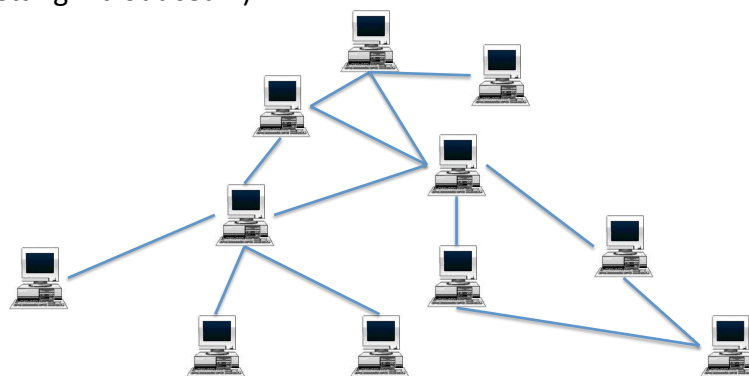
PEER-TO-PEER NETWORKS

Peer-to-peer architectures

- Idea: easy, **large-scale** sharing of data with **no central point of control**
- **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
- Two variants:
 - **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

Unstructured P2P networks

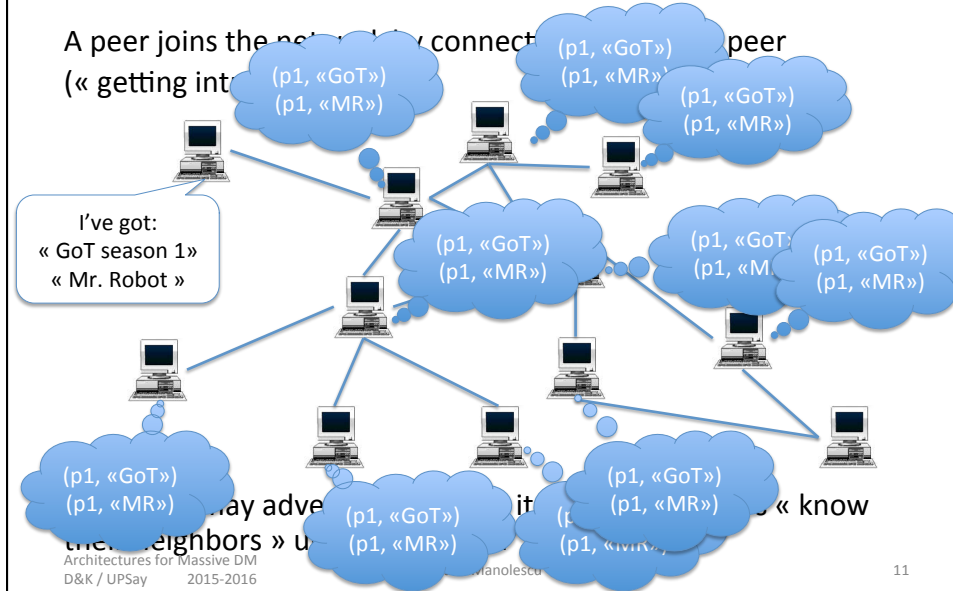
A peer joins the network by connecting to another peer
(« getting introduced »)



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

Unstructured P2P networks

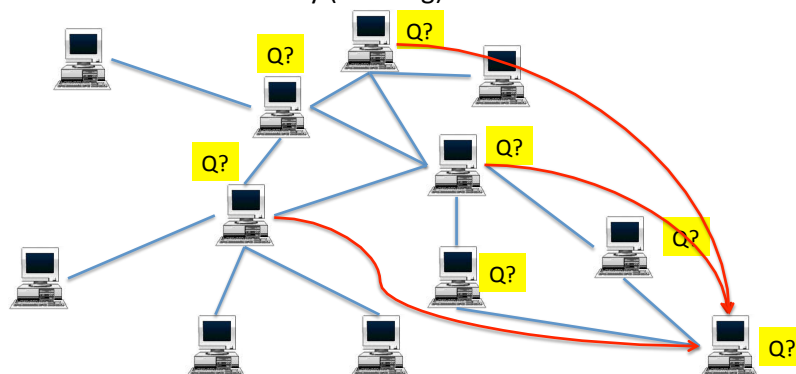
A peer joins the network and connects to a peer (« getting into the network »)



11

Unstructured P2P networks

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



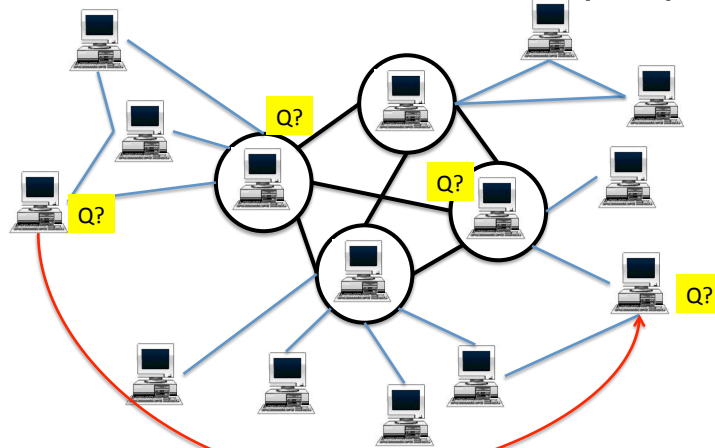
To avoid saturating the network, queries have TTL (time-to-live)
This may lead to missing answers → a. replication; b. superpeers

Architectures for Massive DM
D&K / UPSay 2015-2016

Ioana Manolescu

12

Unstructured P2P with superpeers



- 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 to at least to a superpeer, which routes the peer's queries

Architectures for Massive DM
D&K / UPSay 2015-2016

Ioana Manolescu

13

Indexing (catalog construction) in structured P2P networks

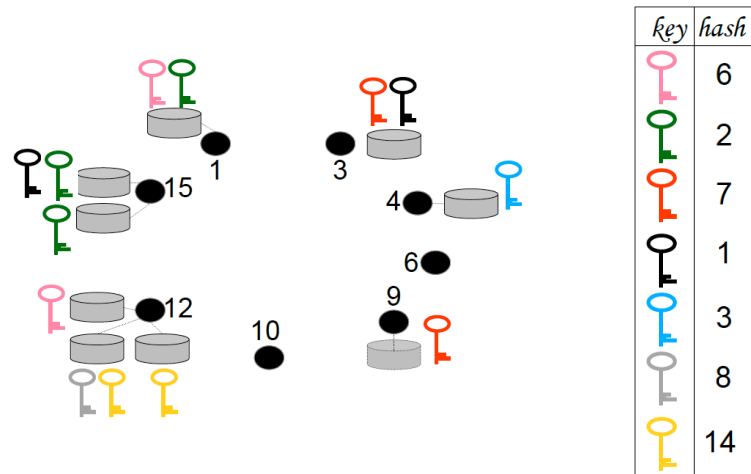
- Peers form a **logical address space** $0 \dots 2^k - 1$
 - Some positions may be vacant
- The **catalog** is built as a set of key-value pairs
 - **Key**: expected to occur in search queries, e.g. «GoT», «Mr Robot»
 - **Value**: the address of content in the network matching the key, e.g. «peer5/Users/a/movies/GoT»
- A **hash function** is used to map every key into the address space; this distributes $(key, value)$ pairs
 - $H(key)=n \rightarrow$ the $(key, value)$ is sent to peer n
 - If n is unoccupied, the next peer in logical order is chosen
- The catalog is **distributed across the peers**
(also the name: **distributed hash table, DHT**)

Architectures for Massive DM
D&K / UPSay 2015-2016

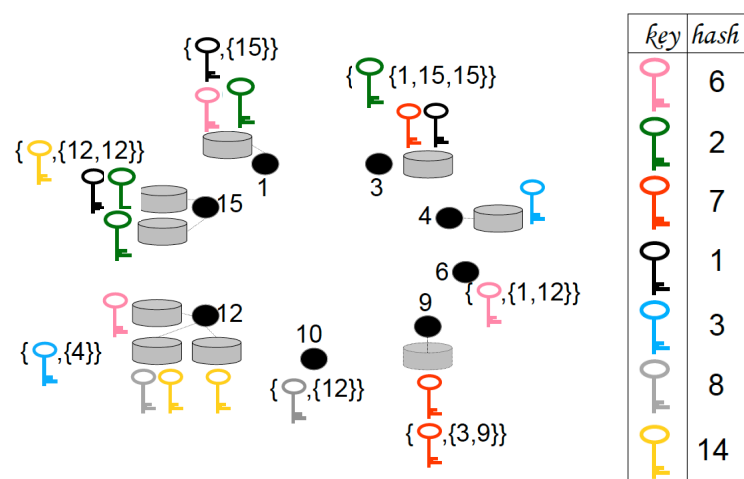
Ioana Manolescu

14

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 $0 \dots 2^k - 1$

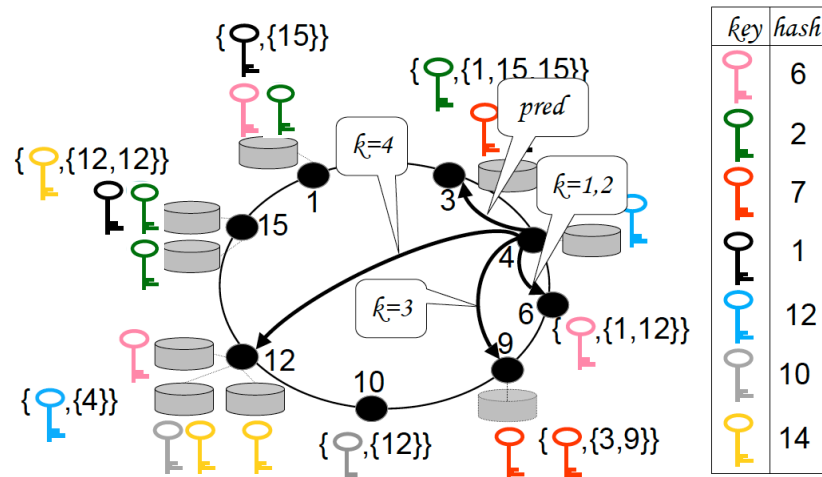
Example: Chord (widely popular)

Each peer n is connected to

- $n+1, n+2, \dots, n+2^{k-1}$, 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

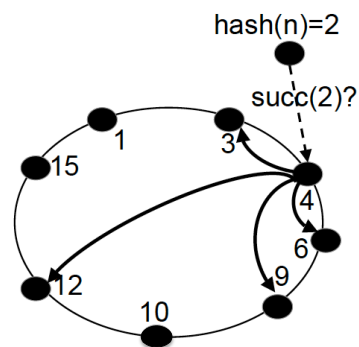


Peers joining in Chord

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;
```

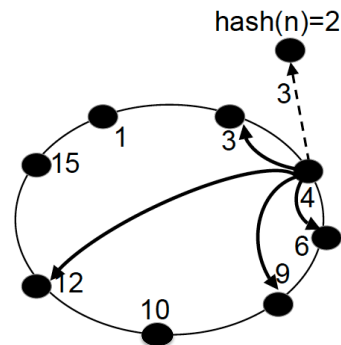


Peers joining in Chord

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

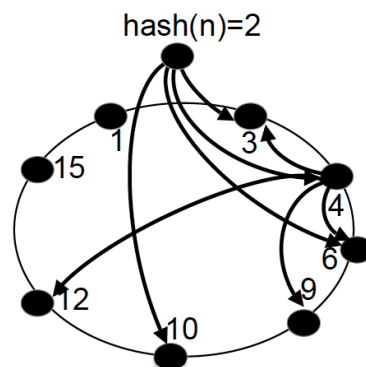
To join, a peer **n** must know (any) peer **n'** already in the network

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

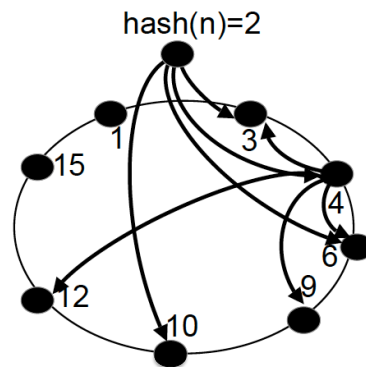
Each peer periodically runs stabilize()

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

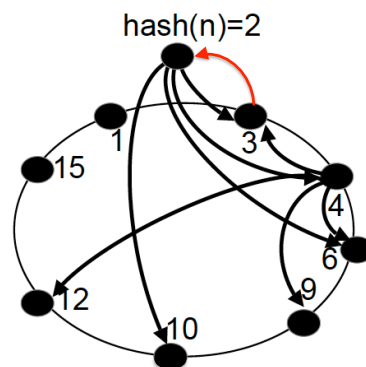
First stabilize() of 2: 3 learns its new predecessor

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

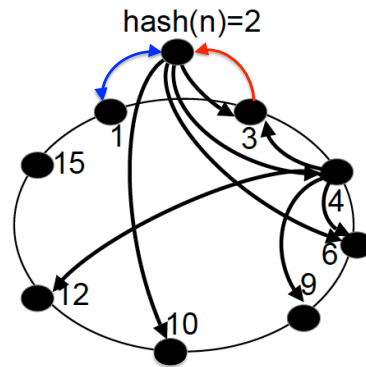
First stabilize() of 1: 1 and 2 connect

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: the leaving peer P sends are sent to the 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

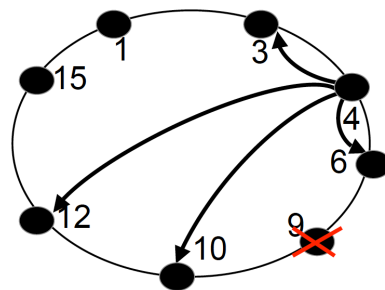
Example Running `stab()`, 6 notices 9 is down

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 » go along the ring, since it is relatively rare

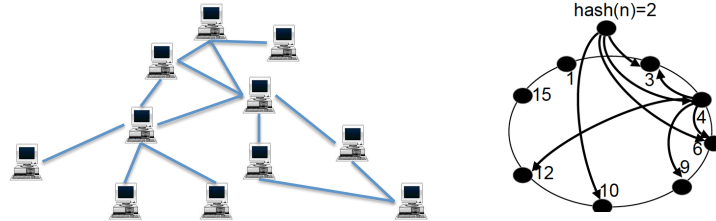
To prevent erroneous routing due to successor failure, each peer maintains a list of its r direct successors ($2 \log_2 N$)

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



E.g. Cassandra (« Big Table » system):

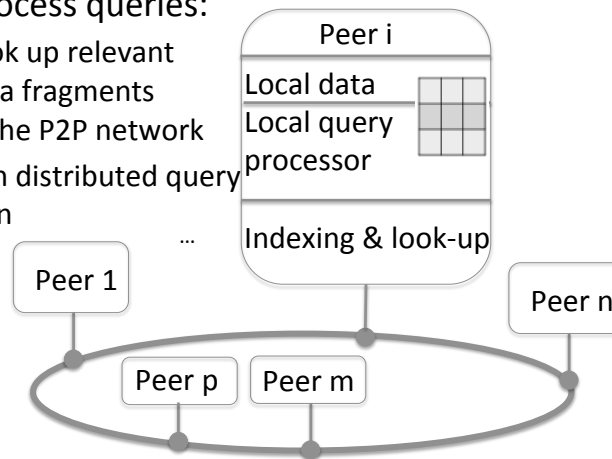
« During gossip exchanges, every node maintains a **sliding window of inter-arrival times of gossip messages from other nodes in the cluster**. Configuring the [phi_convict_threshold](#) property adjusts the sensitivity of the failure detector. Lower values increase the likelihood that an unresponsive 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. »

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 symmetric algorithm (except for superpeers)
- **Concurrency:** conflicting data operations are always at the same peer → local concurrency control!

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: relational 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: [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

NOSQL SYSTEMS

NoSQL systems

- NoSQL = Not Only SQL
- Goal 1: more flexible **data models**
 - One attribute could be a set...
 - Tuples could have heterogeneous attributes...
 - Trees, graphs, no types etc.
- Goal 2: **lighter architectures and systems**
 - Fewer constraints, faster development
 - Among the heaviest aspects of data in relational databases: concurrency control
- Goal 3: large-scale **distribution**
- NoSQL systems may have weaker concurrency control (recall: CAP theorem from course 1)

Some NoSQL systems

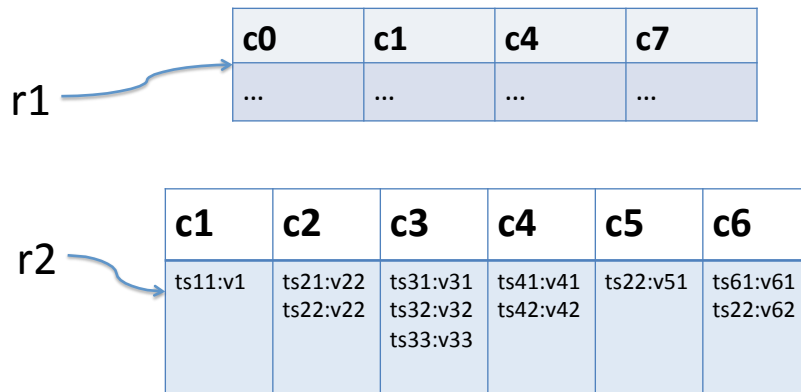


Google Bigtable [CDG+06]

- One of the earliest NoSQL systems
- **Goal:** store data of varied form to be used by Google applications:
 - Web indexing, Google Analytics, Finance etc.
- **Approach:**
 - very large, heterogeneous-structure table
- Data model:
 - Row key → column key → timestamp → value**

Different rows can have different columns, each with their own timestamps etc.

Google Bigtable



Google Bigtable

- **Row key → column key → timestamp → value**
- Rows stored **sorted** in lexicographic order by the key
- Row range dynamically partitioned into **tablets**
 - Tablet = distribution / partitioning unit
- Writes to a row key are atomic
 - concurrency control unit = row
- Access control unit = **column families**
 - Family = typically same-type, co-occurring columns
 - « At most hundreds for each table »
 - E.g. **anchor** column family in Webtable

Apache projects around Hadoop



Hive: relational-like interface on top of Hadoop

HiveQL language:

```
CREATE table pokes (foo INT, bar STRING);
```

```
SELECT a.foo FROM invites a WHERE a.ds='2008-08-15';
```

```
FROM pokes t1 JOIN invites t2 ON (t1.bar = t2.bar)
INSERT OVERWRITE TABLE events SELECT t1.bar, t1.foo,
t2.foo;
```

+ possibility to plug own Map or Reduce function when needed...

39

Apache projects around Hadoop



- **HBASE:** very large tables on top of HDFS («*goal: billions of rows x millions of columns* »), based on « *sharding* »
- Apache version of Google's BigTable [CDG+06] (used for Google Earth, Web indexing etc.)
- Main strong points:
 - Fast access to individual rows
 - read/write consistency
 - Selection push-down (~ Hadoop++)
- Does not have: column types, query language, ...

40

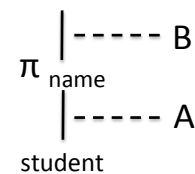
Apache projects around Hadoop



PIG: rich dataflow (« SQL + PL/SQL » style) language on top of Hadoop

Suited for many-step data transformations (« extract-transform-load »)

```
A = LOAD 'student' USING PigStorage()
  AS (name:chararray, age:int, gpa:float);
B = FOREACH A GENERATE name;
DUMP B;
```



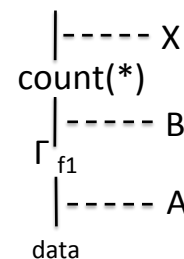
- Flexible data model (~ nested relations)
- Some nesting in the language (< 2 FOREACH ☺)

Apache projects around Hadoop



PIG: rich dataflow (« SQL + PL/SQL » style) language on top of Hadoop

```
A = LOAD 'data' AS (f1:int,f2:int,f3:int);
DUMP A;
(1,2,3) (4,2,1) (8,3,4) (4,3,3) (7,2,5) (8,4,3)
B = GROUP A BY f1;
DUMP B;
(1,{{(1,2,3)}}) (4,{{(4,2,1),(4,3,3)}}) (7,{{(7,2,5)}})
(8,{{(8,3,4),(8,4,3)}})
X = FOREACH B GENERATE COUNT(A);
DUMP X;
(1L) (2L) (1L) (2L)
```



42

Apache projects around Hadoop



Cassandra

- (Large, distributed) relations on top of Hadoop
- Some nesting (a field can be a collection); indexes; SQL-like access rights
- Queries: select, project. No joins.

Table **songs**:

id	song_order	album	artist	song_id	title
62e36092...	4	No One Rides for Free	Fu Manchu	7db1a490...	Ojo Bojo
62e36092...	3	Roll Away	Back Door Slam	2b09185b...	Outside Woman Blues
62e36092...	2	We Must Obey	Fu Manchu	8a172618...	Moving in Stereo
62e36092...	1	Tres Hombres	22 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-bcf4-0800200c9a66	{rock}
a3e64f8f-bd44-4f28-b8d9-6938726e34d4	{blues, 1973}
8a172618-b121-4136-bb10-f665cfc469eb	{2007, covers}

43

Spanner: A More Recent Google Distributed Database [CD+12]

- A few **Universes** (e.g. one for production, one for testing)
- Universe = set of zones
 - **Zone** = unit of administrative deployment
 - One or several zones in a datacenter
 - 1 zone = 1 **zone master** + 100s to 1000s of **span servers**
 - The zone master assigns data to span servers
 - Each span servers answers client requests
 - Each span server handles 100 to 1000 tablets
- **Tablet** = { key → timestamp → string }
- **Table** = set of tablets.

More on the Spanner data model

- Basic: **key** → **timestamp** → **value**
- **Directory** (or **bucket**): set of contiguous keys that share a common prefix
 - Data moves around by the bucket/directory
- On top of the basic model, applications see a **surface relational model**
 - Rows x columns (tables with a **schema**)
 - **Primary keys**: each table must have one or several primary-key columns

Spanner tables

- Tables can be organized in **hierarchies**
 - Tables whose primary key **extends the key of the parent** can be stored **interleaved** with the parent
 - Example: photo album metadata organized first by the user, then by the album

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```

Users(1)	
Albums(1,1)	Directory 3665
Albums(1,2)	
Users(2)	
Albums(2,1)	Directory 453
Albums(2,2)	
Albums(2,3)	

Spanner replication

- Used **for very high-availability** storage
- Store data with a **replication** factor (3 to 5)
- Applications can control:
 - Which datacenters store which data
 - How far data is from users (to control read latency)
 - How far replicas are from each other (to control write latency)
 - How many replicas are maintained
- Concurrency control relies on a **global timestamp mechanism** called « TrueTime » (see next)

Spanner TrueTime service

- `TT.now()` returns a **Ttinterval [earliest; latest]**
 - Uncertainty interval made explicit
 - The interval is guaranteed to contain the absolute time during which `TT.now()` was invoked
 - TrueTime clients **wait** to avoid the uncertainty
- Based on GPS and atomic clocks
 - Implemented by a set of **time master machines** per datacenter and a **timeslave daemon** per machine
 - Every daemon polls a variety of masters to **reduce vulnerability** to
 - Errors from a single master
 - Attacks

Spanner consistency guarantees

- **Linearizability:**
If transaction **T1** commits before **T2** starts
then the commit timestamp of **T1** is guaranteed to be
smaller than the commit timestamp of **T2**

→ globally meaningful commit timestamps
→ globally-consistent reads across the database at a
timestamp

May not read the *last* version, but one from 5-10 seconds
ago! (Last globally committed version.)

Spanner consistency guarantees

- Linearizability:
If transaction **T1** commits before **T2** starts
Then the commit timestamp of **T1** is
guaranteed to be smaller than the commit
timestamp of **T2**

→ globally meaningful commit timestamps
→ globally-consistent reads across the database
at a timestamp

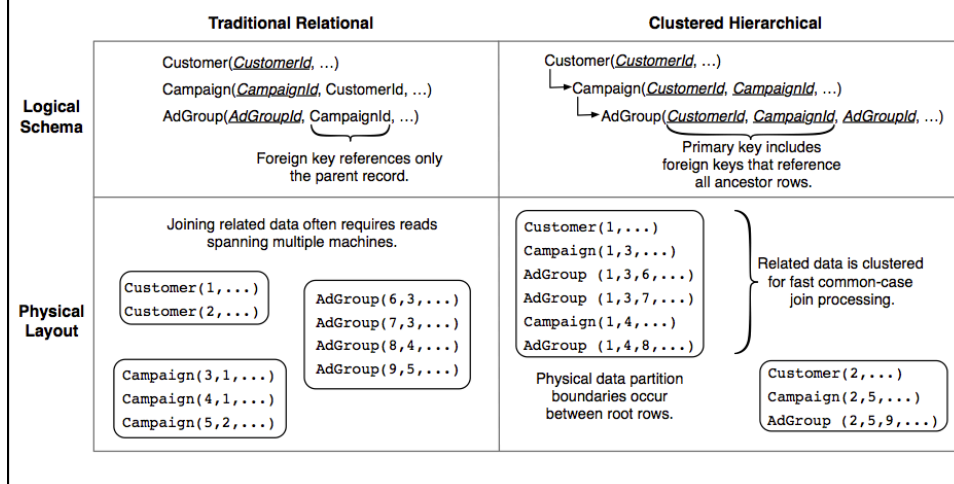
« Some authors have claimed that general two-phase commit is too expensive to support, because of the performance or availability problems it brings. We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions. »

F1: Distributed Database from Google [SVS+13]

- Built on top of Spanner
- Goals:
 - Scalability, availability
 - Consistency (= ACID)
 - Usability (= full SQL + transactional indexes etc.)
- F1 from genetics « Filial 1 Hybrid » (cross mating of very different parental types)
 - F1 is a hybrid between relational DBs and scalable NoSQL systems

F1 data model

- Clustered, inlined table hierarchies as in Spanner

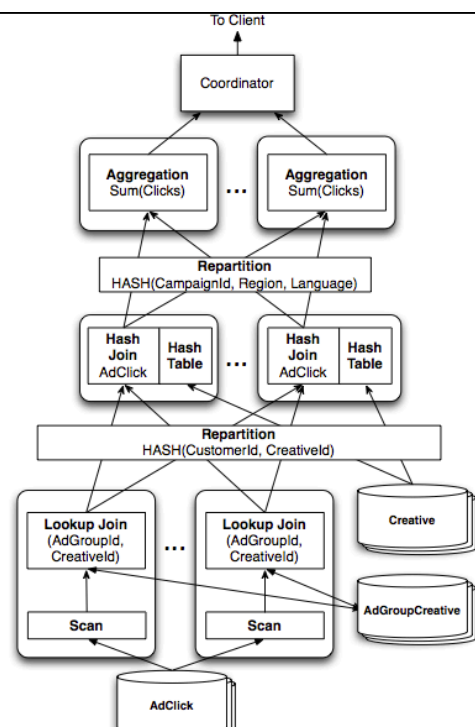


Transactions in F1

- **Snapshot** (read-only) transactions (no locks)
 - Read at Spanner global safe timestamp, typically 5-10 seconds old, from a local replica
 - Default for SQL and MapReduce. All clients see the same data at the same timestamp.
- **Pessimistic** transactions
 - Shared or exclusive locks; may abort
- **Optimistic** transactions
 - Read phase (no lock) then write phase
 - Each row has last modification timestamp
 - To commit T1, F1 creates a short pessimistic T2 which attempts to read all of T1's rows. If T2 has a different version than T1, then T1 is aborted. Otherwise, T1 commits.
 - Only works with previously existent rows → insertion phantoms may still occur

Query optimization in F1

```
SELECT agcr.CampaignId, click.Region,
       cr.Language, SUM(click.Clicks)
FROM AdClick click
JOIN AdGroupCreative agcr
  USING (AdGroupId, CreativeId)
JOIN Creative cr
  USING (CustomerId, CreativeId)
WHERE click.Date = '2013-03-23'
GROUP BY agcr.CampaignId, click.Region,
         cr.Language
```



Controversy around NoSQL (1)


OF THE **ACM** | HOME | CURRENT ISSUE | NEWS | **BLOGS** | OPINION | RESEARCH | PRACTICE | CAREERS | MAGAZINE ARCHIVE

Home / Blogs / BLOG@CACM / The "NoSQL" Discussion has Nothing to Do With SQL / Full Text

BLOG@CACM
The "NoSQL" Discussion has Nothing to Do With SQL

By Michael Stonebraker
 November 4, 2009
 Comments (12)

VIEW AS: SHARE:



Recently, there has been a lot of buzz about NoSQL. In fact there are at least two conferences on each coast. Seemingly this buzz comes from proponents of:

- document-style stores in which a database is a collection of (key, value) pairs plus a payload. A class of system include CouchDB and MongoDB. These are called **document stores** for simplicity.
- key-value stores whose records consist of a key and a value. Usually, these are implemented by distributed systems and we call these **key-value stores** for simplicity.

There are **no valid performance arguments** to adopt NoSQL systems:

1. For many-nodes architecture (horizontal scale-out), use **sharding** (fragmentation) of very large tables.
2. For single-node performance, the **bottleneck is in the interface between the program and the DB** (think JDBC). To solve this, just use embedded database libraries (e.g. BerkeleyDB).


→ There is no problem that NoSQL systems solve best!

Controversy around NoSQL (2)

www.meetup.com/Silicon-Valley-NoSQL/events/153896752/

Silicon Valley NoSQL Meetup Group

Home Members Sponsors Photos Discussions More [Join us!](#)



"The Rights and Wrongs of the NoSQL Phenomenon" with Dr. C. Mohan

6 days ago · 5:30 PM
 The Innovation Center



[...] there is currently a **mad rush** to develop and adopt a plethora of NoSQL systems [...]

In rushing to develop these systems to overcome some of the shortcomings of the relational systems, **many good principles** of the latter, which go beyond the relational model and the SQL language, **have been left by the wayside**.

Now many of the features that were initially discarded as unnecessary in the NoSQL systems are **being brought in, but unfortunately in ad hoc ways**.

Hopefully, the lessons learnt over three decades with relational and other systems would not go to waste and we wouldn't let history repeat itself with respect to simple minded approaches leading to enormous pain later on for developers as well as users of the NoSQL systems!

Controversy around NoSQL (3)

By Michael Stonebraker

September 30, 2010

[Comments \(14\)](#)



According to a [recent ReadWriteWeb blog post](#) by Audrey Watters, 44% of enterprise users questioned had never heard of NoSQL and an additional 17% had no interest. So why are 61% of enterprise users either ignorant about or uninterested in NoSQL? This post contains my two cents worth on the topic.

At a recent trade show I attended, which highlighted NoSQL engines, there were many Web developers, mostly from startups. However, I was struck by the absence of enterprise users. Hence, my (totally unscientific) experience confirms the basic point of the above blog post.

Moreover, in my experience, most information among enterprise users occurs by word of mouth.

1. No ACID equals no interest
2. A low-level query language is death
3. NoSQL means no standards

NoSQL systems in perspective

- What for?
 - Data model flexibility, performance, distribution?
- What kind of workload?
 - Reads, writes? **Concurrency control needs?**
- Durability of the code ? Open source? Who maintains it etc. Compare with expected lifetime of the project.
- Portable / compiles into major frameworks? E.g. many systems on top of Hadoop etc.
- Interesting compromise solutions to some large-scale distributed DB problems

References

- [CDG+06] **Bigtable: A Distributed Storage System for Structured Data.** Fay Chang, Jeffrey Dean, Sanjay Chemawat, Wilson Hsieh, Deborah Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert Gruber. OSDI, 2006
- [CDE+12] **Spanner: Google's Globally-Distributed Database.** James Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes et al. OSDI, 2012
- [SVS+13] **F1: A Distributed SQL Database That Scales.** Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey et al. PVLDB, 2013