Advanced databases and data models

Theme 4.1 NoSQL

Fang Wei-Kleiner

ADIT, IDA
Linköping University

HT 2012

NoSQL: Motivation

30, 40 years history of well-established database technology, all in vain? Not at all! But both setups and demands have drastically changed:

- main memory and CPU speed have exploded, compared to the time when System R (the mother of all RDBMS) was developed
- at the same time, huge amounts of data are now handled in real-time
- both data and use cases are getting more and more dynamic
- social networks (relying on graph data) have gained impressive momentum
- full-texts have always been treated shabbily by relational DBMS

NoSQL: Facebook (Architechture)

- Memcached
  - distributed memory caching system
  - caching layer between web and database servers
  - based on a distributed hash table (DHT)
- HipHop for PHP
  - developed by Facebook to improve scalability
  - compiles PHP to C++ code, which can be better optimized
  - PHP runtime system was re-implemented
- Cassandra
  - developed by Facebook for inbox searching
  - data is automatically replicated to multiple nodes
  - no single point of failure (all nodes are equal)
- Hadoop/Hive
  - implementation of Google's MapReduce framework
  - performs data-intensive calculations
  - (initially) used by Facebook for data analysis

NoSQL: Facebook (Components)

- Varnish
  → HTTP accelerator, speeds up dynamic web sites
- Haystack
  → object store, used for storing and retrieving photos
- BigPipe
  → web page serving system; serves parts of the page (chat, news feed, ...)
- Scribe
  → aggregates log data from different servers

NoSQL: Facebook

hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html

Architecture: Hadoop Cluster

- 21 PB in Data Warehouse cluster, spread across 2000 machines:
- 1200 machines with 8 cores, 800 machines with 16 cores
- 12 TB disk space per machine, 32 GB RAM per machine
- 15 map-reduce tasks per machine

Workload

- daily: 12 TB of compressed data, and 800 TB of scanned data
- 25,000 map-reduce jobs and 65 million files per day

NoSQL: Facebook

HBase

- Hadoop database, used for e-mails, IM and SMS
- has recently replaced MySQL, Cassandra and few others
- built on Google's BigTable model

Conclusion

- classical database solutions have turned out to be completely insufficient
- heterogeneous software architecture is needed to match all requirements

NoSQL: Not only SQL

- RDBMS are still a great solution for centralized, tabular data sets
- NoSQL gets interesting if data is heterogeneous and/or too large
- most NoSQL projects are open source and have open communities
- code bases are up-to-date (no 30 years old, closed legacy code)
- they are subject to rapid development and change
- cannot offer general-purpose solutions yet, as claimed by RDBMS

NoSQL: Not only SQL

www.techrepublic.com/blog/10things/10-things-you-should-know-about-nosql-databases/1772

10 Things: Five Advantages

- Elastic Scaling → scaling out: distributing data instead of buying bigger servers
- Big Data → opens new dimensions that cannot be handled with RDBMS
- Goodbye DBAs (see you later?) → automatic repair, distribution, tuning, ...
- Economics → based on cheap commodity servers and less costs per transaction/second
- Flexible Data Models → non-existing/relaxed data schema, structural changes cause no overhead

NoSQL: Not only SQL

www.techrepublic.com/blog/10things/10-things-you-should-know-about-nosql-databases/1772

10 Things: Five Challengees

- Maturity → still in pre-production phase, key features yet to be implemented
- Support → mostly open source, start-ups, limited resources or credibility
- Administration → require lot of skill to install and effort to maintain
- Analytics and Business Intelligence → focused on web apps scenarios, limited ad-hoc querying
- Expertise → few number of NoSQL experts available in the market

NoSQL: Concepts

nosql-database.org

Definition

Next Generation Databases mostly addressing some of the points: being non-relational, distributed, open source and horizontally scalable. The original intention has been modern web-scale databases. The movement began early 2009 and is growing rapidly. Often more characteristics apply as: schema-free, easy replication support, simple API, eventually consistent/BASE (not ACID), a huge data amount, and more.

–Stefan Edlich

NoSQL: Concepts

Scalability: system can handle growing amounts of data without losing performance.

- Vertical Scalability (scale up)
  - add resources (more CPUs, more memory) to a single node
  - using more threads to handle a local problem
- Horizontal Scalability (scale out)
  - add nodes (more computers, servers) to a distributed system
  - gets more and more popular due to low costs for commodity hardware
  - often surpasses scalability of vertical approach

NoSQL: Concepts

CAP Theorem: Consistency, Availability, Partition Tolerance

`Brewer [ACM PODC'2000]: Towards Robust Distributed Systems`

Consistency

- after an update, all readers in a distributed system see the same data
- all nodes are supposed to contain the same data at all times

Example

- single database instance will always be consistent
- if multiple instances exist, all writes must be duplicated before write operation is completed

NoSQL: Concepts

CAP Theorem: Consistency, Availability, Partition Tolerance

`Brewer [ACM PODC'2000]: Towards Robust Distributed Systems`

Availability

- all requests will be answered, regardless of crashes or downtimes

Example

- a single instance has an availability of 100% or 0%, two servers may be available 100%, 50%, or 0%

Partition Tolerance

- system continues to operate, even if two sets of servers get isolated

Example

- system gets partitioned if connection between server clusters fails
- failed connection won't cause troubles if system is tolerant

NoSQL: Concepts

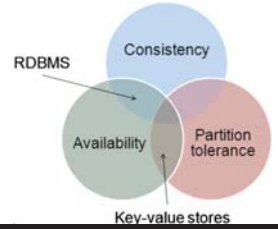
CAP Theorem: Consistency, Availability, Partition Tolerance

`Brewer [ACM PODC'2000]: Towards Robust Distributed Systems`

Theorem

(Gilbert, Lynch SIGACT'2002): only 2 of the 3 guarantees can be given in a shared-data system

- (Positive) consequence: we can concentrate on two challenges
- ACID properties needed to guarantee consistency and availability
- BASE properties come into play if availability and partition tolerance is favored



NoSQL: Concepts

ACID: Atomicity, Consistency, Isolation, Durability

- Atomicity → all operations in the transaction will complete, or none will
- Consistency → before and after transactions, database will be in a consistent state
- Isolation → operations cannot access data that is currently modified
- Durability → data will not be lost upon completion of a transaction

NoSQL: Concepts

BASE: Basically Available, Soft State, Eventual Consistency

`Fox et al. [SOSP'1997]: Cluster-Based Scalable Network Services`

- Basically Available → an application works basically all the time (despite partial failures)
- Soft State → is in flux and non-deterministic (changes all the time)
- Eventual Consistency → will be in some consistent state (at some time in future)

NoSQL: Techniques

Basic techniques (widely applied in NoSQL systems)

- distributed data storage, replication (how to distribute the data which is partition tolerant?) → Consistent hashing
- eventual consistency → Vector clock
- distributed query strategy (horizontal scalibility) → MapReduce

NoSQL: Consistent Hashing

Consistent Hashing

`Karger et al. [ACM STOC'1997], Consistent Hashing and Random Trees`

Task

- find machine that stores data for a specified key k
- trivial hash function to distribute data on n nodes: $h(k, n) = k \bmod n$
- if number of nodes changes ($n \pm 1$), all data will have to be redistributed!

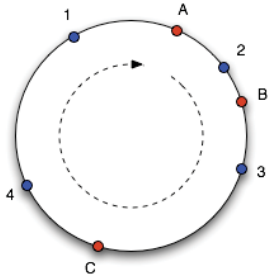
Challenge

- minimize number of nodes to be copied after a configuration change
- incorporate hardware characteristics into hashing model

NoSQL: Consistent Hashing

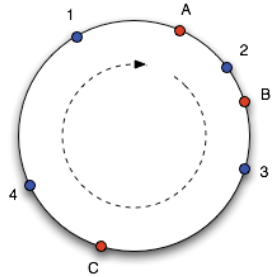
Basic idea

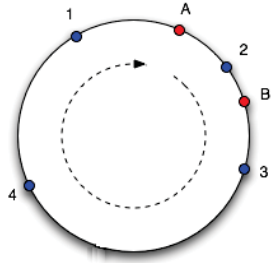
- arrange the nodes in a ring and each node is in charge of the hash values in the range between its neighbor node
- include hash values of all nodes in hash structure
- calculate hash value of the key to be added/retrieved
- choose node which occurs next clockwise in the ring



NoSQL: Consistent Hashing

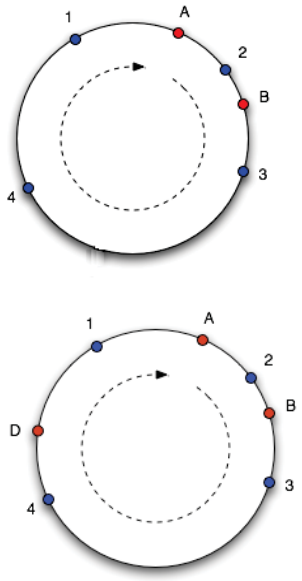
- include hash values of all nodes in hash structure
- calculate hash value of the key to be added/retrieved
- choose node which occurs next clockwise in the ring
- if node is dropped or gets lost, missing data is redistributed to adjacent nodes (replication issue)





NoSQL: Consistent Hashing

- if a new node is added, its hash value is added to the hash table
- the hash realm is repartitioned, and hash data will be transferred to new neighbor
  → no need to update remaining nodes!



NoSQL: Consistent Hashing

Consistent Hashing

`Karger et al. [ACM STOC'1997], Consistent Hashing and Random Trees`

- a replication factor r is introduced: not only the next node but the next r nodes in clockwise direction become responsible for a key
- number of added keys can be made dependent on node characteristics (bandwidth, CPU, ...)
- nifty details are left to the implementation
  (e.g.: `DeCandia et al. [2007], Dynamo: Amazon's Highly Available Key-value Store`

NoSQL: Logical time

Logical Clocks
Challenge

- recognize order of distributed events and potential conflicts
- most obvious approach: attach timestamp (ts) of system clock to each event $e \rightarrow ts(e)$

→ error-prone, as clocks will never be fully synchronized
→ insufficient, as we cannot catch causalities (needed to detect conflicts)
Clock Consistencies
strong consistency (causality): if $ts(e1) < ts(e2)$, then $e1 \rightarrow e2$

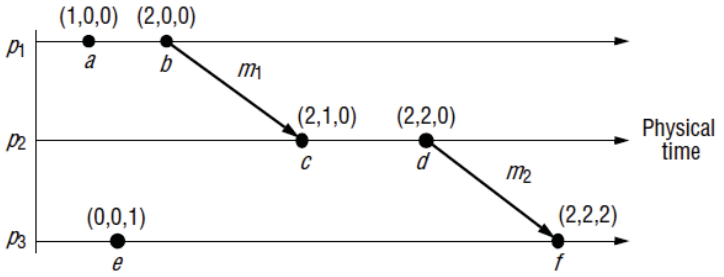
NoSQL: Vector clock

`[book] G. Coulouris et al. Distributed Systems: Concepts and Design, 5th Edition`

- A vector clock for a system of N nodes is an array of N integers.
- Each process keeps its own vector clock, V_i , which it uses to timestamp local events.
- processes piggyback vector timestamps on the messages they send to one another, and there are simple rules for updating the clocks:
  - VC1: Initially, $V_i[j] = 0$, for $i, j = 1, 2, \ldots N$
  - VC2: Just before p_i timestamps an event, it sets $V_i[i] := V_i[i] + 1$
  - VC3: p_i includes the value $t = V_i$ in every message it sends
  - VC4: When p_i receives a timestamp t in a message, it sets $V_i[j] := max(V_i[j], t[j])$, for $j = 1, 2, \ldots, N$

NoSQL: Vector clock

- VC1: Initially, $V_i[j] = 0$, for $i, j = 1, 2, \ldots N$
- VC2: Just before p_i timestamps an event, it sets $V_i[i] := V_i[i] + 1$
- VC3: p_i includes the value $t = V_i$ in every message it sends
- VC4: When p_i receives a timestamp t in a message, it sets $V_i[j] := max(V_i[j], t[j])$, for $j = 1, 2, \ldots, N$

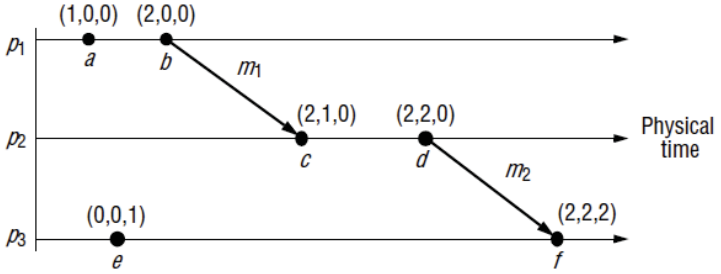


NoSQL: Vector clock

Properties:

- $V = V'$ iff $V[j] = V'[j]$ for $j = 1, 2, \ldots N$
- $V \leq V'$ iff $V[j] \leq V'[j]$ for $j = 1, 2, \ldots N$
- $V < V'$ iff $V \leq V'$ and $V \neq V'$

two events e and e': that $e \rightarrow e' \Leftrightarrow V(e) < V(e')$
$\rightarrow$ Conflict detection!



NoSQL: Techniques

MapReduce Framework `Dean, Ghemawat [OSDI'2004]: MapReduce: Simplified Data Processing on Large Clusters`

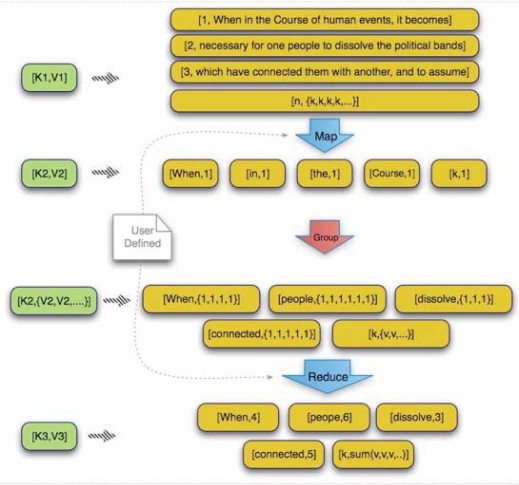
- developed by Google to replace old, centralized index structure
- supports distributed, parallel computing on large data sets
- inspired by (...not equal to...) the map and fold functions in functional
- programming: no side effects, deadlocks, race conditions
- divide-and-conquer paradigm: Map recursively breaks down a problem into sub-problems and distributes it to worker nodes; Reduce receives and combines the sub-answers to solve the problem

NoSQL: Techniques

MapReduce: Functions
Input & Output: each a set of key/value pairs.
Programmer specifies two functions:

- `map(in_key, in_value)` $\rightarrow$ `list(out_key, intermed_value)`: processes input key/value pair, produces set of intermediate pairs
- `reduce(out_key, list(intermed_value))` $\rightarrow$ `list(out_value)`: combines all intermediate values for a particular key, produces a set of merged output values (usually just one)

NoSQL: MapReduce of Word Count




```
map(String input_key,
    String input_value):
  # input_key: document name
  # input_value: document contents
  for each word w in input_value:
     EmitIntermediate(w, "1")
```

```
reduce(String output_key,
       Iterator intermed_values):
  # output_key: a word
  # output_values: a list of counts
  int result = 0
  for each v in intermed_values:
     result += ParseInt(v)
  Emit(AsString(result))
```


NoSQL: MapReduce: Summary

- programmer can focus on map/reduce code
- framework cares about parallelization, fault tolerance, scheduling, ...
- often based on a custom file system, which is optimized for distributed access (Google: GFS, Hadoop: HDFS)
- MapReduce framework can be written for different environments: shared memory, distributed networks, ...
- used in a wide range of scenarios: distributed search (grep), creation of word indexes, sorting distributed data, count page links,...

NoSQL: Systems

Selected Categories

`nosql-databases.org`

- Document Stores
- Key-Value Stores
- Column Stores
- Graph Databases
- Object Databases

→ no taxonomy exists that all parties agree upon
→ might look completely different some years later

NoSQL: Key-value stores

- also called: associative arrays, hash tables/maps
- keys are unique, values may have arbitrary type
- focus: high scalability (more important than consistency)
- traditional solution: BerkeleyDB, started in 1986
- revived by Amazon Dynamo in 2007 (proprietary)
- recent solutions: Redis, Voldemort, Tokyo Cabinet, Memcached

→ (very) limited query facilities; usually `get(key)` and `put(key, value)`

NoSQL: Key-value stores

Amazon Dynamo

DeCandia et al. [ACM SIGOPS'2007], Dynamo: Amazon's Highly Available Key-value Store

- most important requirement: reliability
- runs tens of thousands of servers (mostly commodity hardware)
- one of several databases used at Amazon → components fail continuously
- use of relational databases would lead to inefficiencies and limit scale and availability

Concepts

- replication and partitioning via consistent hashing
- consistency facilitated by object versioning, based on vector clocks

NoSQL: Key-value stores

Amazon Dynamo

DeCandia et al. [ACM SIGOPS'2007], Dynamo: Amazon's Highly Available Key-value Store

Arguments against RDBMS

- complex querying and management functionality is not needed
- systems with ACID properties have poor availability

Consequences

- operations are limited to one key/value pair at a time: no cross-references, no multiple updates, no isolation guarantees
- services are only used internally by Amazon → no security layers needed
- optimistic replication scheme ("always writable")

NoSQL: Key-value stores

Amazon Dynamo

DeCandia et al. [ACM SIGOPS'2007], Dynamo: Amazon's Highly Available Key-value Store

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector Clocks with reconciliation during reads	Version size is decoupled from update rates
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronized divergent replicas in the background
Membership and failure detection	Gossip-based membership protocol and failure detection	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

NoSQL: Key-value stores

Redis

- in-memory database, sponsored by VMWare
- written in ANSI C, bindings in numerous other languages
- replication: master-slave; slaves can write and may have other slaves
- persistence: snapshots (asynchronous transfer from main-memory to disk)
- keys may be of other type than string: lists, (sorted) sets, hashes
- databases supports high level operations: intersection, union, difference
- performance: very fast, no notable difference between read and write
- not suitable for querying data (limited to value retrievals)

NoSQL: Key-value stores

Project Voldemort

`project-voldemort.com/design.php`

- developed for `LinkedIn` network
- database API: `get(key)`, `put(key, value)`, `delete(key)`
- no filters, no joins, no key constraints, no triggers, ...

Reasoning

- only efficient queries are supported, predictable performance
- service-orientation often disallows foreign key constraints
- data can be easily distributed

NoSQL: BigTable

BigTable

`Chang et al. [OSDI'2006], Bigtable: A Distributed Storage System for Structured Data`

Motivation

- scale of Google services is too large for RDBMSs
- key/value model is too limiting to represent complex scenarios

Definition

- "Sparse, distributed, persistent multidimensional sorted map."
- multidimensional: allows for nested data structures (a map of maps)
- sortedness: helpful property in distributed environments

NoSQL: BigTable

BigTable

`Chang et al. [OSDI'2006], Bigtable: A Distributed Storage System for Structured Data`

Building Blocks

- Google File System: raw storage (stores persistent data)
- SSTable: file type, storing immutable, ordered key/value string pairs
- MapReduce: simplifies large-scale data processing (used to read data)
- Lock Service (Chubby): distributed lock manager (master election)
- Scheduler: schedules jobs onto machines

NoSQL: BigTable

BigTable: Google File System

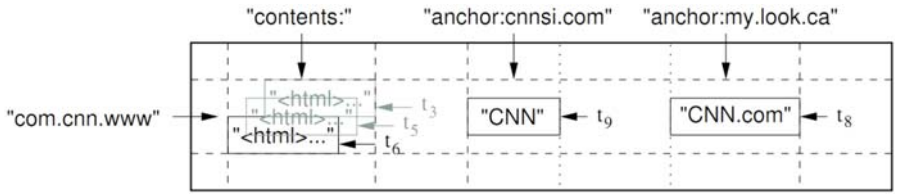
`Ghemawat et al. [SOSP'2003], The Google File System`

- provided as userspace library (not as kernel extension)
- distributed file system, consisting of single master and numerous chunks
- master nodes store all meta data (file name, size, ...) and reference chunks
- shadow masters jump in if master fails (also provide read operations)
- chunk nodes store file contents; files are split into 64 MB large chunks
- main operations: read and append (rare overwritings or compressions)
- chunks are replicated several times (min. 3 times) to avoid fallouts

NoSQL: BigTable

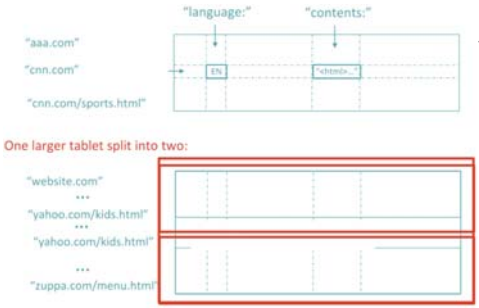
BigTable

`Chang et al. [OSDI'2006], Bigtable: A Distributed Storage System for Structured Data`



- Rows: row name is a reversed URL (why would you store a reversed URL?)
- Columns: contents: page contents; anchor: links to stored web page
- Timestamps: multiple versions per homepage

NoSQL: BigTable



BigTable: Column Families
→ assembles columns of the same type (e.g.: multiple anchors)
Rows

- keys are arbitrary strings (up to 64KB, usually 10-100 bytes)
- data is lexicographically ordered by row keys
- row range is dynamically partitioned to tablets

NoSQL: Hadoop

Hadoop

`hadoop.apache.org`

- software framework for reliable, scalable, distributed computing
- inspired by Google's papers on MapReduce, BigTable, and GFS
- deployed by large companies (e.g. Yahoo!, Facebook, Baidu, IBM)

Components

1. Hadoop Distributed File System (HDFS): counterpart of GFS
2. HBase: counterpart of BigTable
3. Job and Task Trackers: distribute MapReduce jobs received by clients
4. Hadoop Common: core routines, infrastructure

NoSQL: Not only SQL

Conclusion

- NoSQL solutions have become essential in distributed environment
- RDBMS are still prevailing (often known as the only alternative)

Choosing a Database
Before you go for a database system/paradigm, clarify for yourself...

1. Which features are needed? Robust storage vs. realtime results vs. querying
2. What limits are most critical? Memory vs. performance vs. bandwidth
3. How large your data will get? Mega- vs. giga- vs. tera- vs. ...bytes

References

- http://www.informatik.uni-konstanz.de/arbeitsgruppen/dbis/lehre/ss11/advanced-database-technologies/
- Brewer [ACM PODC'2000]: Towards Robust Distributed Systems
- Lynch, Seth [ACM SIGACT'2002]: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services
- Fox et al. [SOSP'1997]: Cluster-Based Scalable Network Services
- Dean, Ghemawat [OSDI'2004]: MapReduce: Simplified Data Processing on Large Clusters
- Karger et al. [ACM STOC'1997], Consistent Hashing and Random Trees
- G. Coulouris et al. Distributed Systems: Concepts and Design, 5th Edition
- DeCandia et al. [ACM SIGOPS'2007], Dynamo: Amazon's Highly Available Key-value Store
- Chang et al. [OSDI'2006], Bigtable: A Distributed Storage System for Structured Data
- Ghemawat et al. [SOSP'2003], The Google File System