Advanced Data Management Technologies

Unit 15 — Introduction to NoSQL

J. Gamper

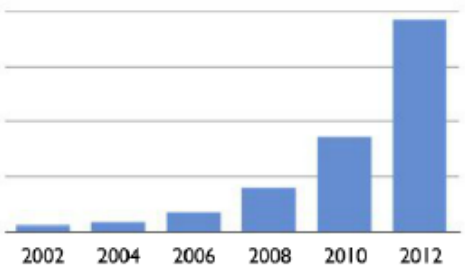
Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Outline

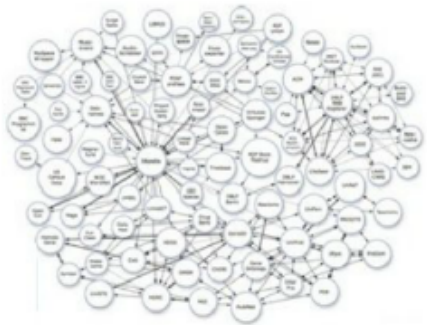
1. **Motivation**
2. **NoSQL**
3. **Categories of NoSQL Datastores**
   - Key-Value Stores
   - Column Stores
   - Document Stores
   - Graph Databases

Outline

New Trends



Big data



Connectivity

P2P Knowledge

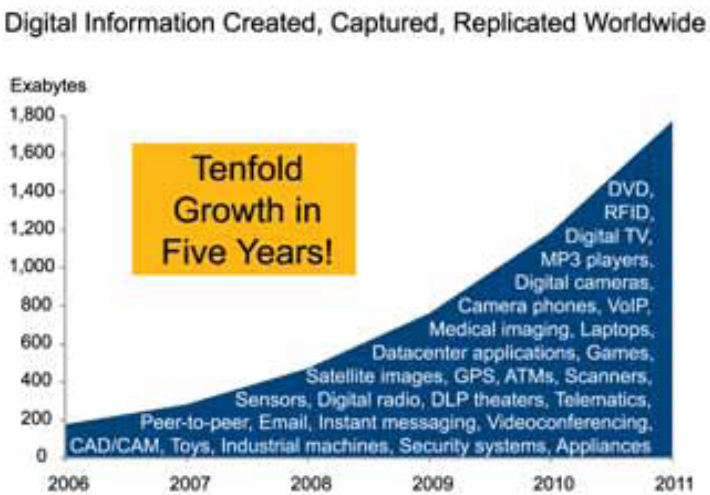
Concurrency

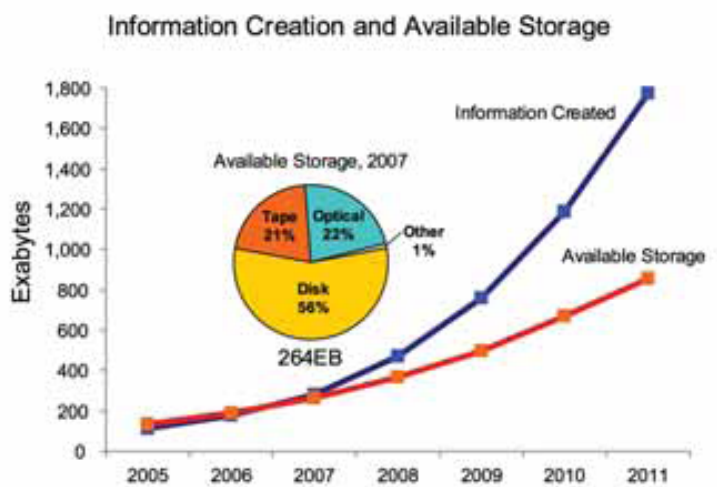
Diversity

Cloud-Grid

Big Data – The Digital Age/1

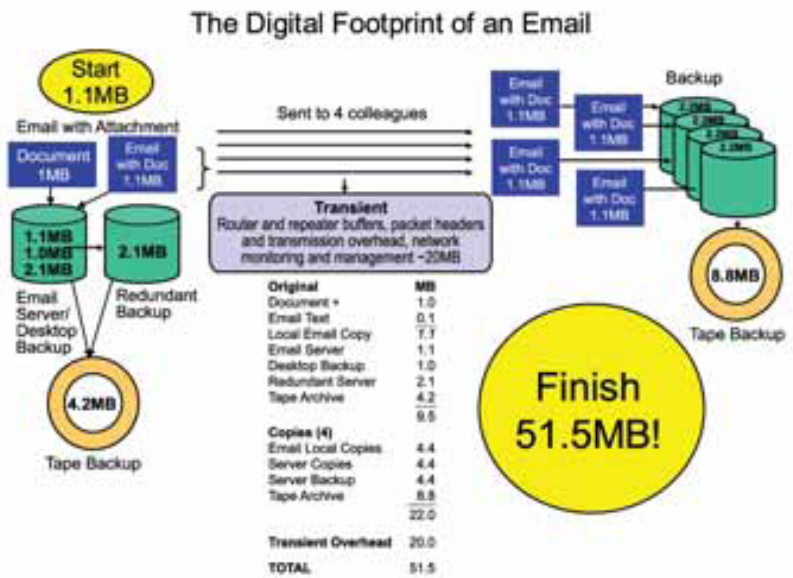
- IDC/EMC annual report "The Diverse and Exploding Digital Universe":
  - The worlds information is doubling every two years. In 2011 the world will create a staggering **1.8 zettabytes**. By 2020 the world will generate 50 times the amount of information ... while IT staff to manage it will grow less than 1.5 times.
  - New "information taming" technologies such as deduplication, compression, and analysis tools are driving down the cost of creating, capturing, managing, and storing information to one-sixth the cost in 2011 in comparison to 2005.
- 1 zettabyte $= 10^{21}$ bytes $=$ 1 bio. terabytes





Big Data – The Digital Age/2

- The New York Stock Exchange generates about 1 terabyte of new trade data per day.
- Facebook hosts approximately 10 billion photos, taking up one petabyte of storage.
- Ancestry.com, the genealogy site, stores around 2.5 petabytes of data.
- The Large Hadron Collider near Geneva will produce about 15 petabytes of data per year.
- But even an email might produce a lot of data.

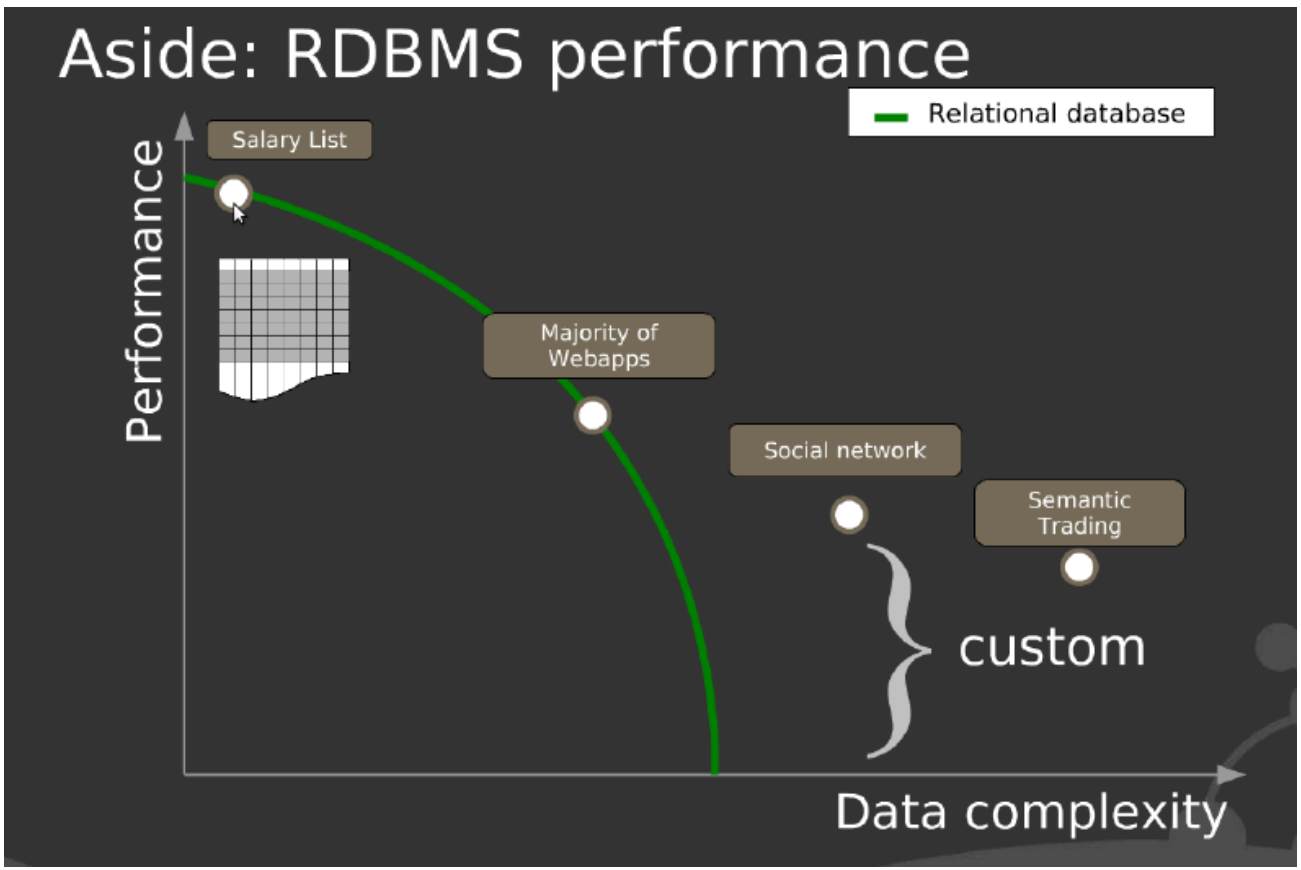


3 V's of Big Data

- More V's are coming up:
  - **V**eracity: accuracy and quality of data is difficult to control
  - **V**alue: it is important to turn big data it into value
  - ...

RDBMSs

- The predominant choice in storing data up until now.
- First formulated in 1969 by Codd
  - We are using RDBMS everywhere!
- BUT, are RDBMSs good in managing todays data?



The Death of RDBMS?

THURSDAY, AUGUST 26, 2010

Death of the Relational Database 2010

Back in 2008 I wrote a piece called Death of the Relational Database. It was one of the most popular pieces I have written, and for many reasons the subject matter is one in which I have an ongoing interest. As a result I wrote current piece to organize my thoughts for to hear me t SXSW Pan

Is the Relational Database Doomed?

By Tony Bain / February 12, 2009 3:00 PM / View Comments

Tweet

The RDBMS is not enough.

Posted by Sebastien Auvray on Nov 26, 2007

Sections Operations & Infrastructure, Architecture & Design, Development Topics CouchDB , Ruby , Couchbase , Dynamic Languages , Distribu Database , Companies , Relational Databases , Languages , Data Access , Database Design , FEATURED NoSQL , Database Management , Scalab Programming , S3 , Database , Performance

Share | dz

While Relational Databases fit a client scalability issues: How to create redu

[Relation Databases] become a s having two database servers that synchronize changes. Having one writing information.

Should you go Beyond Relational Databases?

By Martin Kleppmann

24 June 2009 | Category: Code

What is Wrong with RDBMSs?

- Nothing is wrong. They are great ...
- SQL provides a rich, declarative language
- Database enforce referential integrity
- ACID properties are guaranteed
- Well understood by developers and administrators
- Support by many different languages

ACID Properites

- **A**tomicity – all or nothing
- **C**onsistency – any transaction will take the DB from one consistent state to another with no broken contraints (referential integrity)
- **I**solation – other operations cannot access data that has been modified during a transaction that has not yet completed
- **D**urability – ability to recover the committed transaction updates against any kind of systems failure

But there are some Problems with RDBMSs

- Problem: Complex objects
  - Object/relational impedance mismatch
  - Complicated to map rich domain model
  - Performance issues: many rows in many tables, many joins, ...
- Problem: Schema evolution
  - Adding attributes to an object $\Rightarrow$ have to add columns to a table
  - Expensive for large tables
  - Holding locks on the tables for long time
- Problem: Semi-structered data
  - Relational schema does not easily handle semi-structured data
  - Common solutions
    - Name/Value table: poor performance
    - Serializable as Blob: fewer joins but no query capabilities
- Problem: Relational is hard to scale
  - ACID does not scale well
  - Easy to scale reads, but hard to scale writes

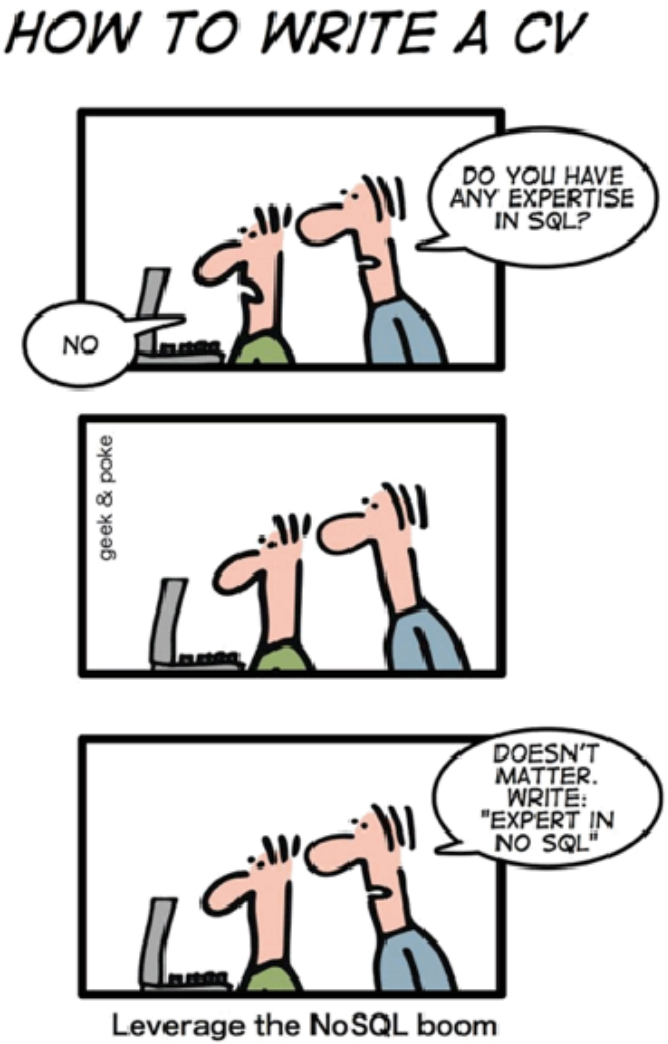
One Size does not Fit All!

- There is nothing wrong with RDBMSs, but one size does not fit all!
- Alternative tools are available, just use the right tool.
- The rise of NoSQL databases marks the end of the era of relational database dominance.
- But NoSQL databases will not become the new dominators.
- Relational will still be popular, and used in the majority of situations.
  - They, however, will no longer be the automatic choice.

Outline

What is NoSQL?

- "~~SQL~~" or "Not Only SQL" or "No to SQL"?
- There is no standard definition!
- The term NoSQL was coined by Carlo Strozzi in 1998
- In 2009 used by Eric Evans to refer to DBs which are non-relational, distributed and not conform to ACID.
- In 2009 first NoSQL conference
- Refers generally to data models that are non-relational, schema-free, non-(quite)-ACID, horizontally scalable, distributed, easy replication support, simple API



Changing Requirements in the Web Age

- ACID properties are always desirable
- But, web applications have different needs from applications that RDBMS were designed for
  - Low and predictable response time (latency)
  - Scalability & elasticity (at low cost!)
  - High availability
  - Flexible schemas and semi-structured data
  - Geographic distribution (multiple data centers)
- Web applications can (usually) do without
  - Transactions, strong consistency, integrity
  - Complex queries

CAP Theorem/1

- Desired properties of web applications:
  - **C**onsistency – the system is in a consistent state after an operation
    - All clients see the same data
    - Strong consistency (ACID) vs. eventual consistency (BASE)
  - **A**vailability – the system is "always on", no downtime
    - Node failure tolerance – all clients can find some available replica
    - Software/hardware upgrade tolerance
  - **P**artition tolerance – the system continues to function even when split into disconnected subsets, e.g., due to network errors or addition/removal of nodes
    - Not only for reads, but writes as well!

- **CAP Theorem (E. Brewer, N. Lynch)**
  - In a "shared-data system", at most 2 out of the 3 properties can be achieved at any given moment in time.

CAP Theorem/2

- CA
  - Single site clusters (easier to ensure all nodes are always in contact)
  - e.g., 2PC
  - When a partition occurs, the system blocks
- CP
  - Some data may be inaccessible (availability sacrificed), but the rest is still consistent/accurate
  - e.g., sharded database
- AP
  - System is still available under partitioning, but some of the data returned my be inaccurate
    - i.e., availability and partition tolerance are more important than strict consistency
  - e.g., DNS, caches, Master/Slave replication
  - Need some conflict resolution strategy

BASE Properties

- Requirements regarding reliability, availability, consistency and durability are changing.
- For a growing number of applications, availability and partition tolerance are more important than strict consistency.
- These properties are difficult to achieve with ACID properties
- The BASE properties forfeit the ACID properties of consistency and isolation in favor of "availability, graceful degradation, and performance"

- BASE properties
  - **B**asically **A**vailable – an application works basically all the time;
  - **S**oft-state – does not have to be consistent all the time;
  - **E**ventual consistency – but will be in some known state eventually.

- i.e., an application works basically all the time (basically available), does not have to be consistent all the time (soft-state) but will be in some known state eventually (eventual consistency

BASE vs. ACID

- Should be considered as a spectrum between the two extremes rather than two altenatives excluding each other

ACID	BASE
Strong consistency	Weak consistency – stale data OK
Isolation	Availability first
Focus on "commit"	Best effort
Nested transactions	Approximate answers OK
Availability?	Aggressive (optimistic)
Conservative (pessimistic)	Simpler!
Difficult evolution (e.g., schema)	Faster
	Easier evolution

NoSQL Pros and Cons

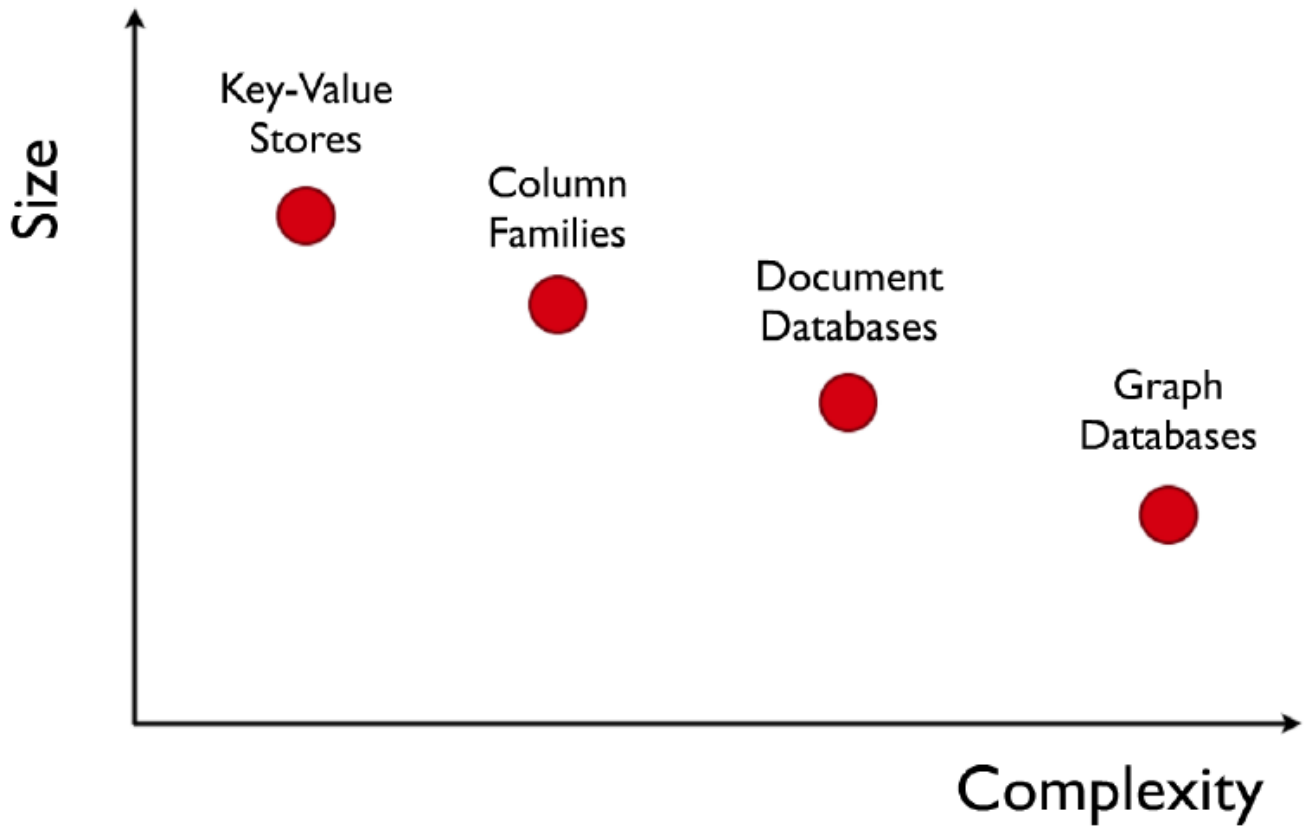
- Advantages
  - Massive scalability (horizontal scalability), i.e., machines can be added/removed
  - High availability
  - Lower cost (than competitive solutions at that scale)
  - (Usually) Predictable elasticity
  - Schema flexibility, sparse & semi-structured data
  - Quicker and cheaper to set up
- Disadvantages
  - Limited query capabilities (so far)
  - Eventual consistency is not intuitive to program
    - Makes client applications more complicated
  - No standardization
    - Portability might be an issue
  - Insufficient access control

Outline

Categories of NoSQL Datastores

- Key-Value stores
  - Simple K/V lookups (DHT)
- Column stores
  - Each key is associated with many attributes (columns)
  - NoSQL column stores are actually hybrid row/column stores
    - Different from "pure" relational column stores!
- Document stores
  - Store semi-structured documents (JSON)
  - Map/Reduce based materialisation, sorting, aggregation, etc.
- Graph databases
  - Not exactly NoSQL ...
  - Cannot satisfy the requirements for high availability and scalability/elasticity very well.

Focus of Different NoSQL Data Models



Comparison of SQL and NoSQL Data Models

Data Model	Performance	Scalability	Flexibility	Complexity	Functionality
Key-value Stores	high	high	high	none	variable (none)
Column Store	high	high	moderate	low	minimal
Document Store	high	variable (high)	high	low	variable (low)
Graph Database	variable	variable	high	high	graph theory
Relational Database	variable	variable	low	moderate	relational algebra

Key-Value Stores

- Simple data model: global collection of key-value pairs.
- Favor high scalability to handle massive data over consistency
  - Rich ad-hoc querying and analytics features are mostly omitted (especially joins and aggregate operations are set aside).
- Simple API with `put` and `get`
- Key-value stores have existed for a long time, e.g., Berkeley DB.
- Recent developments have been inspired by Distributed Hashtables and Amazon's Dynamo
  - DeCandia et al., Dynamo: Amazon's Highly Available Key-value Store, SOSP 07
- Another important free and open-source key-value store is Voldemort.
- Multiple types
  - In memory: Memcache
  - On disk: Redis, SimpleDB
  - Eventually consistent: Dynamo, Voldemort

Dynamo

- P2P key-value store at Amazon, $\approx$ 2007
- Context and requirements at Amazon
  - Infrastructure: tens of thousands of servers and network components located in many data centers around the world
  - Commodity hardware is used, where component failure is the "standard mode of operation"
  - Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services
  - Low latency and high throughput
  - Simple query model: unique keys, blobs, no schema, no multi-access
  - Scale out (elasticity)
- Simple API
  - `get(key)`: returning a list of objects and a context
  - `put(key, context, object)`: no return value
- Key and object values are not interpreted but handled as "an opaque array of bytes"

Voldemort/1

- Key-value store initially developed for and still used at LinkedIn
  - Inspired by Amazon's Dynamo
- Features
  - Written in Java
  - Simple data model and only simple and efficient queries
    - no joins or complex queries
    - no constraints on foreign keys
    - etc.
  - Performance of queries can be predicted well
  - P2P
  - Scale-out / elastic
  - Consistent hashing of keyspace
  - Eventual consistency / high availability
  - Pluggable storage
    - BerkeleyDB, In Memory, MySQL

Voldemort/2

- API consists of three functions:
  - `get(key)`: returning a value object
  - `put(key, value)`: writing an object/value
  - `delete(key)`: deleting an object
- Keys and values can be complex, compound objects as well consisting of lists and maps

Column Stores

- Data model: each key is associated with multiple attributes (i.e., columns)
- Hybrid row/column store
- Inspired by Google BigTable
- Examples: BigTable, HBase, Cassandra

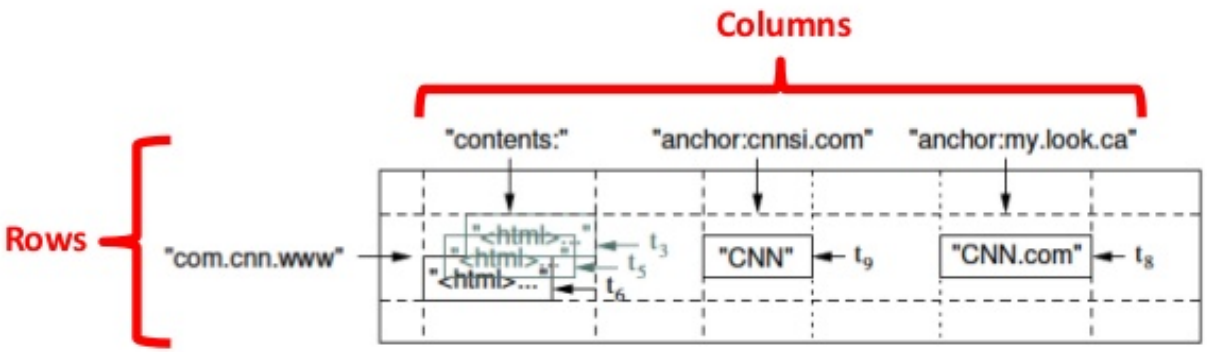
BigTable

- BigTable at Google, $\approx$ 2006
- A distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers
- Observation
  - Key-value pairs are a useful building block, but should not be the only one
- Design goal: data model should be
  - richer than simple key-value pairs, and support sparse semi-structured data,
  - but simple enough that it lends itself to a very efficient flat-file representation

BigTable Data Model/1

- Sparse, distributed, persistent multidimensional sorted map
- Values are stored as arrays of bytes (strings) which are not interpreted
- Values are addressed by (*row*, *column*, *timestamp*) dimensions

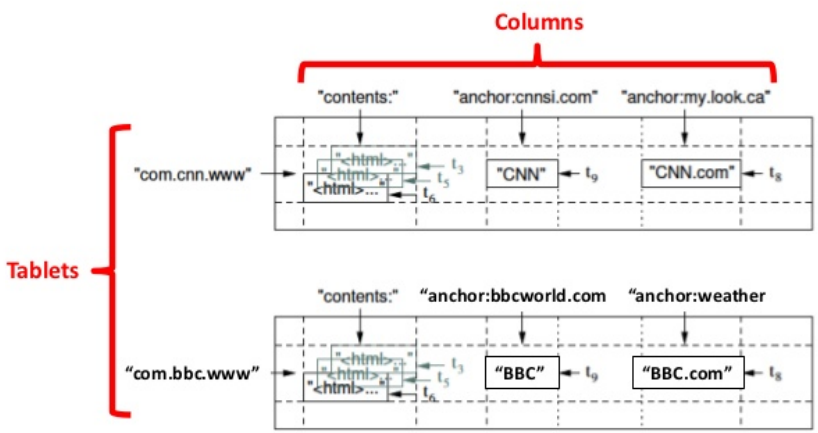
- Example: Multidimensional sorted map with information that a web crawler might emit



  - Flexible number of rows representing domains
  - Flexible number of columns
    - first column contains the content of the web page
    - the others store link text from referring domains
  - Every value has a timestamp

BigTable Data Model/2

- Row
  - Keys are arbitrary strings
  - Data is sorted by row key
- Tablet
  - Row range is dynamically partitioned into tablets (sequence of rows)
  - Range scans are very efficient
  - Row keys should be chosen to improve locality of data access
- Column, Column Family
  - Column keys are arbitrary strings, unlimited number of columns
  - Column keys can be grouped into families
  - Data in a CF is stored and compressed together (Locality Groups)
  - Access control on the CF level

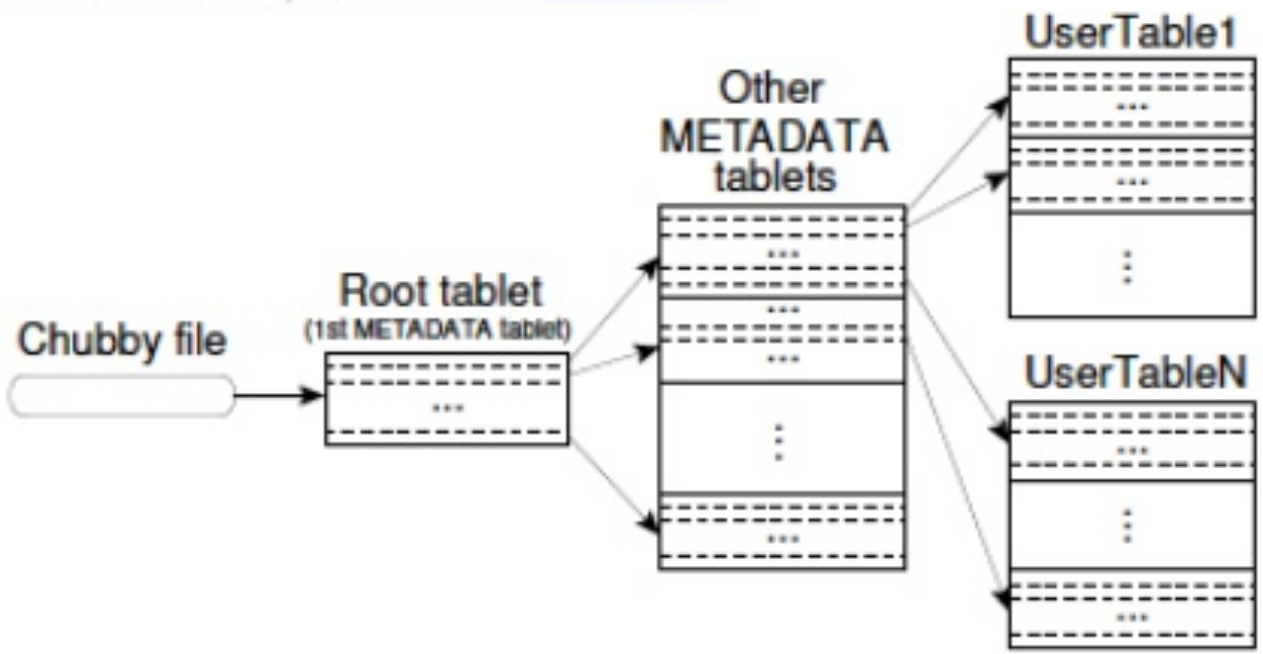


BigTable Data Model/3

- Timestamps
  - Each cell has multiple versions
  - Can be manually assigned
- Versioning
  - Automated garbage collection
  - Retain last N versions or versions newer than TS
- Architecture
  - Data stored on GFS
  - 1 Master server
  - Thousands of Tablet servers

BigTable Architecture

- Data is stored in a 3-level hierarchy similar to B^+-trees
  - Chubby file contains location of root tablet
  - Root tablet contains all tablet locations in Metadata table
  - Metadata table stores locations of actual tablets



Document Stores

- Similar to a key-value database, but with a major difference: value is a document.
- Inspired by Lotus Notes
- Flexible schema
  - Any number of fields can be added
- Document mainly stored in JSON or BSON formats
- Example document:

```
{
   day:  [‘‘2010’’, ‘‘01’’, ‘‘23’’],
   products:  {
     apple:  { price:  ‘‘10’’ quantity:  ‘‘6’’ }
     kiwi:  { price:  ‘‘20’’ quantity:  ‘‘2’’ }
   }
   checkout:  ‘‘100’’
}
```

CouchDB/1

- Schema-free, document store DB
- Documents stored in JSON format (XML in old versions)
- B-tree storage engine
- MVCC model, no locking
- No joins, no PK/FK (UUIDs are auto assigned)
- Implemented in Erlang
  - 1st version in C++, 2nd in Erlang and 500 times more scalable
- Replication (incremental)
- Documents
  - UUID
  - Old versions retained
- Custom persistent views using MapReduce
- RESTful HTTP interface

CouchDB/2

- Main abstraction and data structure is a document
- Consist of named fields that have a key/name and a value
- Field name must be unique in document
- Value may be a string, number, boolean, date, ordered list, map
- References to other documents (URIs, URLs) are possible but not checked by the DB
- Example document

```
‘‘Title’’: ‘‘CouchDB’’,
‘‘Last editor’’ : ‘‘172.5.123.91’’,
‘‘Last modified’’: ‘‘9/23/2010’’,
‘‘Categories’’: [‘‘Database’’, ‘‘NoSQL’’, ‘‘Document Database’’],
‘‘Body’’: ‘‘CouchDB is a ...’’,
‘‘Reviewed’’: false
```

MongoDB

- Document store DB written in C++
- Full index support
- Replication & high availability
- Supports ad-hoc querying
- Fast in-place updates
- Officially supported drivers available for multiple languages
  - C, C++, Java, Javascript, Perla, PHP, Python, Ruby
- Map/Reduce
- GridFS
- Commercial support

MongoDB/2

- A database resides on a MongoDB server
- A MongoDB database consists of one or more collections of documents
- Schema-free, i.e., documents in a collection may be heterogeneous
- Main abstraction and data structure is a document
  - Comparable to an XML document or a JSON document
- Documents are stored in BSON
  - Similar to JSON, but binary representation for efficiency reasons
- Example document:

```
{
title : ‘‘MongoDB’’,
last_editor : ‘‘172.5.123.91’’ ,
last_modified : new Date (‘‘9/23/2010’’) ,
body : ‘‘MongoDB is a ...’’,
categories : [‘‘Database’’, ‘‘NoSQL’’, ‘‘Document Database’’],
reviewed : false
}
```

MongoDB Example

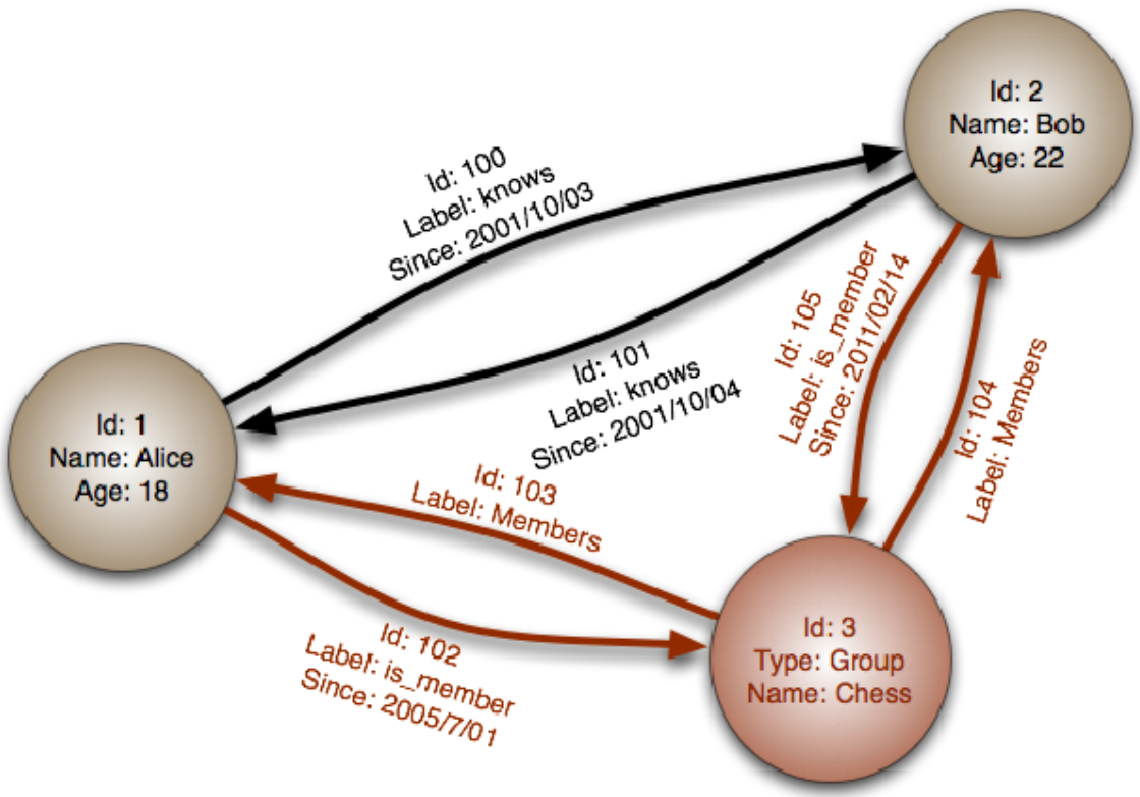
- Create a collection named `mycoll` with 10,000,000 bytes of preallocated disk space and no automatically generated and indexed document-field
  - `db.createCollection(‘‘mycoll’’, size: 10000000, autoIndexId: false)`
- Add a document into `mycoll`
  - `db.mycoll.insert(title: ‘‘MongoDB’’, last_editor: ... )`
- Retrieve a document from `mycoll`
  - `db.mycoll.find(categories: [‘‘NoSQL’’, ‘‘Document Databases’’])`

MongoDB Deployment

And much more...

Graph Databases

- Data Model
  - Nodes
  - Relations
  - Properties
- Inspired by Euler's graph theory
- Examples: Neo4j, InfiniteGraph



Summary

- New trends emerged in the past decade: big data, complexity, connectivity, diversity, etc.
- New requirements: consistency, availability and partitioning tolerance.
- NoSQL provides flexible solution for such requirements.
- NoSQL taxonomy
  - Key-value stores
  - Column stores
  - Document stores
  - Graph databases
- Use the right data model for the right problem.