Introduction to NoSQL

Lecture 19
Outline

• Relational vs. NoSQL databases
  • the value of relational databases
  • new requirements and NoSQL features
  • flexible data models

• Types of NoSQL databases
  • key-value stores
  • document databases
  • column-family databases
  • graph databases

• Concurrency

• Usage patterns
History

- **Relational databases**

1980

- A **standard** data model is basis for standard query language SQL

1990

- **Mature** technologies:
  - Physical organization of data on disk
  - Indexes: $B^+$-Trees, hash indexes
  - Query optimization, operator implementations

2000

- **Concurrency** control (ACID)
  - transactions: atomicity, consistency, isolation, durability

2010

- Many reliable **integration** mechanisms
  - “shared database” integration of multiple applications
Impedance mismatch

- **Mismatch** between tables and data structures in memory
  - For object-oriented languages: invented Object-Relational Mapping (ORM)
  - For other languages (functional, c) – data structures just do not match
Object-oriented databases

1980
Relational databases
1990
Object-oriented databases
2000
2010
Why object-oriented databases disappeared

- They were not useful for integrating applications through databases
- For integration through databases, data should be broken into atomic datum – to be used by different applications
Relational databases predominate
Current Trends: Big Data

Current Trends: Lots of traffic

Current Trends: Cloud Computing

source: http://www.profitbricks.com/what-is-iaas
Scaling up

Two alternatives:
  • Bigger servers
  • Lots of little boxes in massive grids
Partitioning

- **Vertical**: normalization, splitting into smaller tables
- **Horizontal**: splitting single table into multiple sets of rows
  - Horizontal partitioning when rows are distributed across multiple nodes based on some attribute (for example, zip code) is called *sharding*
Parallelism is not natural for relational databases

- SQL designed to run as a single node
- Both vertical partitioning and horizontal partitioning introduce performance bottlenecks:
  - *Increased latency* when querying across more than one shard
  - Indexes are sharded by *one dimension*, so that some searches are optimal, and others are slow or impossible
  - *Cross-shard consistency and durability* is hard to achieve due to the more complex failure modes of a set of servers
New requirements on data management

**Trends**

- **Volume** of data
- **Cloud** comp. (IaaS)
- **Velocity** of data
- **Big** traffic
- **Variety** of data

**Requirements**

- Real **scalability**
  - massive database **distribution**
  - **dynamic** resource management
  - **horizontally** scaling systems
- Frequent **update** operations
- Massive **read** throughput
- **Flexible** database schema
History

- 1980: Relational databases
- 1990
- 2000: NoSQL databases
- 2010
Google BigTable (2006)

- Data model: **three-dimensional indexed sorted map**
  - Input (row, column, timestamp) $\rightarrow$ Output (cell contents)

Column-family

- Columns are grouped in column-families
- Different fields describing html documents are stored in different column-families: for fast search and ranking
Partitioning: tablets

- The entire BigTable is split into **tablets** of contiguous ranges of rows
  - Approximately 100MB to 200MB each
- One machine services 100 tablets
  - Fast recovery in event of tablet failure
  - Fine-grained load balancing
  - 100 tablets are assigned non-deterministically to avoid hot spots of data being located on one machine
- Tablets are split as their size grows
Locating Tablets

• Metadata for tablet locations

• Similar to B-tree index: row ids are sorted: interval is a key, and an IP of a corresponding tablet is a value

• No master node – no bottleneck
Amazon: Dynamo DB (2007)

• Data model:
  simple hash table (map): key-value data store

Dynamo: architecture

- Implemented as **distributed hash table (DHT)** based on **consistent hashing** – hashing into the place on the ring
- Elastic scalability: able to scale out one node at a time, with minimal impact on the system
- Decentralization
General definition of NoSQL databases

- What is “NoSQL”?  
  - term used in late 90s for a different type of technology: Carlo Strozzi: [http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL/](http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/NoSQL/)
  - “Not Only SQL”?  
    - but many RDBMS are also “not just SQL”

- “NoSQL is an accidental term with no precise definition”  
  - first used at an informal meetup in 2009 in San Francisco (presentations from Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB, and MongoDB)
Common characteristics

• Not relational
• Cluster-friendly
• Schema-less
• Open source
Data models

1. Key - value (hash table)
2. Key - document
3. Wide-column
4. Graph
1. Key-value stores

- Value can be anything
- Search only by key – no structure inside the value

- Basic operations:
  - **Get** the value for the key
    \[\text{value} := \text{get}(\text{key})\]
  - **Put** a value for a key
    \[\text{put}(\text{key}, \text{value})\]
  - **Delete** a key-value
    \[\text{delete}(\text{key})\]
Key-value Stores: Representatives

- redis
- riak
- LevelDB
- Memcached
- MapDB
- Oracle Berkeley DB 12c
- Oracle NOSQL Database
- Amazon DynamoDB
- Project Voldemort

Ranked list: http://db-engines.com/en/ranking/key-value+store
2. Document stores

- Also key-value pairs
- But **value** is a semi-structured text data - **document**
- Documents are **self-describing** pieces of data
- Hierarchical **tree** data structures
  - Nested associative arrays (maps), collections, scalars
  - XML, JSON (JavaScript Object Notation), BSON, ...
- **Can query inside document**: building search **indexes** on various keys/fields
Data Formats

• **Structured Text Data**
  - JSON, BSON (Binary JSON)
    - JSON is currently number one data format used on the Web
  - XML: eXtensible Markup Language
  - RDF: Resource Description Framework

• **Binary Data**
  - often, we want to store objects (class instances)
  - objects can be binary serialized (marshalled)
    - and kept in a key-value store
  - there are several popular serialization formats
    - Protocol Buffers, Apache Thrift
JSON: Basic Information

- **Text-based open standard** for data interchange
  - Serializing and transmitting structured data

- **JSON = JavaScript Object Notation**
  - Originally specified by Douglas Crockford in 2001
  - Derived *from JavaScript* scripting language
  - Uses conventions of the C-family of languages

- Filename: *.json

- Internet media (MIME) type: application/json

[http://www.json.org](http://www.json.org)
JSON: Data Types (1)

- **object** – an unordered set of **key+value** pairs
  - these pairs are called **properties** (members) of an object
  - syntax: `{ key: value, key: value, key: value, ... }

- **array** – an ordered collection of **values** (elements)
  - syntax: `[ comma-separated values ]`
JSON: Data Types (2)

- **value** – *string* in double quotes / *number* / true or false (i.e., *Boolean*) / *null* / *object* / *array*
- Can be nested
JSON: Data Types (3)

- **string** – sequence of zero or more Unicode characters, wrapped in **double quotes**
  - Backslash escaping
JSON: Data Types (4)

- **number** – like a C or Java number
  - Integer or float
  - Octal and hexadecimal formats are not used
```json
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": 10021
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ]
}
```
XML basics

• XML: eXtensible Markup Language
  • W3C standard (since 1996)
• both human and machine readable

<element attribute="value">content</element>

rule of thumb: data = element tag, metadata = attribute
XML example: books.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year><price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year><price>29.99</price>
  </book>
  <book category="computers">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year><price>39.95</price>
  </book>
</bookstore>
```
Equivalent representation of books.xml using JSON
{
   "bookstore": [
      {
         "category": "cooking", "year": 2005, "price": 30.00, "title": "Everyday Italian", "author": "Giada De Laurentiis"},
      {
         "category": "computers", "year": 2003, "price": 49.99, "title": "XQuery Kick Start", "author": "James McGovern"},
      {
      {
         "category": "computers", "year": 2003, "price": 39.95, "title": "Learning XML", "author": "Erik T. Ray"}
   ]
}
XML Features

• Document may be valid according to a schema:
  • DTD, XML Schema, etc.
• Technologies for parsing: DOM, SAX
• Advanced search technologies:
  • XPath, XQuery, XSLT (transformation)

• XML is great for configurations, meta-data, etc.
• XML databases are not widely used
• Currently, JSON format rules:
  • compact, easier to write, has all features typically needed
Two main properties of structured documents: both JSON and XML

- **Schema-less** – can add new attributes “on-the-fly”
- **Self-describing** data – data and metadata are stored in the same document
Binary Data

- **Data** objects to be stored often originate from memory structures (objects, class instances)

- Before *storing*, these objects must be **serialized**
  - Key-value stores can store a binary *value*

- **Serialization** (marshalling) can be done
  - By your *own* proprietary (de)serializator
  - Using “*standard*” language-specific *way* (Java serialization)
  - Using a *cross-language* standard: ProtoBuf, Apache Thrift
Protocol Buffers

• Technique for **serializing structured** data
• Developed by **Google** since 2008
  ○ BSD Licence

• Philosophy:
  1. **Define** the structure of the data
     • Using an ProtoBuf *interface description language*
  2. **Automatically** create source code in multiple programming languages for (de)serialization of such data
     • Compilers for Java, C++, Python, JavaScript, PHP, ...
Protocol Buffers: Example

// file: addressbook.proto
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}

enum PhoneType {
  MOBILE = 0; HOME = 1; WORK = 2;
}
message PhoneNumber {
  required string number = 1;
  optional PhoneType type = 2 [default = HOME];
}

repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}

source: https://developers.google.com/protocol-buffers/
Protocol Buffers: Example 2 - Java

• **Compile** this source by:
  
  protoc --java_out=jdir addressbook.proto
  protoc --cpp_out=cppdir addressbook.proto
  protoc --python_out=pdir addressbook.proto

• **Result** looks like this (Java):

  [GitHub Link](https://github.com/jgilfelt/android-protobuf-example/blob/master/src/com/example/tutorial/AddressBookProtos.java)
Most documents have JSON format

key=3 -> { "personID": "3",  
            "firstname": "Martin",  
            "likes": [ "Biking","Photography" ],  
            "lastcity": "Boston",  
            "visited": [ "NYC", "Paris" ] }

key=5 -> { "personID": "5",  
            "firstname": "Pramod",  
            "citiesvisited": [ "Chicago", "London","NYC" ],  
            "addresses": [  
                { "state": "AK",  
                  "city": "DILLINGHAM" },  
                { "state": "MH",  
                  "city": "PUNE" } ],  
            "lastcity": "Chicago" }
Example in MongoDB syntax
  • Query language expressed via JSON
  • clauses: where, sort, count, sum, etc.

SQL: SELECT * FROM users
MongoDB: db.users.find()

SELECT * FROM users WHERE personID = "3"
db.users.find({"personID":"3"})

SELECT firstname,lastcity FROM users WHERE personID=5
db.users.find({"personID":"5"},{firstname:1,lastcity:1})
Schema-less?

```javascript
anOrder["price"]*anOrder["qty"]
```

- Need to know the names of attributes
- **Implicit schema**: figure out the meaning of data
Document Databases: Representatives

Key-value vs document: boundaries are blurry

Indexes metadata about the value

Customer_id: 7231

Document – may have id
3. Column-family Stores

- Also called: wide-column, columnar

- Data model: **rows** that have **many columns** associated with a **row key**. Data is **physically stored by column families**

- **Column families** are groups of related data (columns) that are often **accessed together**
  - e.g., for a **customer** we typically access all **profile** information at the same time, but not customer’s **orders**
Column-family Stores: Representatives

Ranked list: http://db-engines.com/en/ranking/wide+column+store
Common for key-value, key-document, row-col_family: aggregates

- We often operate in the world of clusters of objects

- **Aggregate**: complex structure that you can save as a single unit, retrieve as a single unit and work with it as a single unit

- A value, a document, a column-family is a single unit - aggregate
Aggregate-oriented databases

• There is no general strategy to set aggregate boundaries
• Aggregates give the database information about which bits of data will be manipulated together
• These should be stored on the same cluster node
Relational model: aggregate ignorant

- Relational databases are aggregate-ignorant
  - It is not a bad thing, it is a feature
  - Allows to easily look at the data in different ways
  - Best choice when there is no primary structure for data manipulation
Aggregate example: order

What if we want to calculate how many units are sold in total?
New classification of NoSQL

Aggregate databases:
- Key-value
- Document
- Wide-column

Graph databases
4. Graph databases

• Not aggregated: Very hard to model relationships between aggregates in aggregate-oriented databases
• Break things apart into smaller units
• Moving across multiple relationships in relational databases: – too many joins cause very bad performance
Graph database example
Graph databases: mission

• To store entities and relationships between them
  • Nodes are instances of objects
  • Nodes have properties, e.g., name
  • Edges have directional significance
  • Edges have types e.g., likes, friend, ...

• Nodes are organized by relationships
  • Allow to find interesting patterns
  • example: Get all nodes that are “employee” of “Big Company” and that “likes” “NoSQL Distilled”
Graphs in RDBMS

- When we store a graph-like structure in RDBMS, it is for a single type of relationship
  - “Who is my manager”

- Adding another relationship usually means a lot of schema changes

- In RDBMS we model the graph beforehand based on the traversal we want
  - If the traversal changes, the data will have to change
  - Graph DBs: the relationship is not calculated but persisted
Graph Databases: Representatives

Ranked list: http://db-engines.com/en/ranking/graph+dbms
Consistency and concurrency
Consistency

• RDBMSs need ACID transactions – because data is in pieces
• We cannot afford that data is updated in chunks and parts of it are overridden
• We use transactions to wrap things together
• Graph databases do ACID updates
Aggregate consistency

• Aggregates themselves are transaction boundaries
• Isolated atomic update of an aggregate, not between 2 aggregates
Multi-client system

• ACID requires additional handling, because we cannot lock the entire table in web app domain
• Holding a transaction open – degrades performance
Offline lock

Overrides last update – last update is lost
Offline lock

[Diagram showing the process of offline lock with version stamps v101 and v102]
Consistency

• Logical consistency: when the same piece of data is broken into multiple chunks
• Multi-client consistency: performance vs. resilience
Example: booking hotel rooms

• If the connection is temporarily lost at time of booking
• 2 alternatives
  • Prohibit
  • Allow double-booking

• Consistency vs availability
• This is a business choice, not a technical choice
CAP theorem

• Tradeoff between:
  • Consistency
  • Availability
  • Partition tolerance

• Can have only 2 out of 3

• Consistency vs response time of your server
• Even if all the nodes are available – want fast response
In partitioned systems

Partition

<table>
<thead>
<tr>
<th>Consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
</tr>
</tbody>
</table>

Choose one
CAP theorem and DBMSs

**Availability:** Each client can always read and write.

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

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

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

**Pick Two**

- **CA**
  - RDBMSs (MySQL, Postgres, etc)
  - Aster Data Greenplum Vertica

- **AP**
  - Dynamo Voldemort Tokyo Cabinet KAI
  - Cassandra SimpleDB CouchDB Riak

- **CP**
  - BigTable Hypertable Hbase
  - MongoDB Terrastore Scalary
  - Berkeley DB MemcacheDB Redis
When to use NoSQL

• Large amounts of data
• Complex evolving schema
• The domain matches graph or document
• Ease of development: rapid time to market
• Projects that give you a strategic advantage

http://www.tim-wellhausen.de/papers/NoSQL-Patterns/NoSQL-Patterns.html
What with the application integration?

• This has changed too
• Integration through database:
  • Not safe
  • Resistance to schema change – multiple apps are affected
  • Business logic split across applications
• Now integrating data is achieved through web services (REST)
Future?

- 1980: Relational databases
- 1990: NoSQL databases
- 2000: Polyglot persistence
- 2010: Polyglot persistence
One Example of NoSQL Usage: Facebook

Facebook statistics (Spring 2014)

- **1.28 billion** users (1.23B active monthly)
- **300 PB** of user data stored
- **10 billion** messages sent daily
- **250 billion** stored photos (350 million uploaded daily)

2009: 10,000 servers
2010: 30,000 servers
2012: 180,000 servers (estimated)

Database Technology Behind Facebook

Apache Hadoop [http://hadoop.apache.org/]
  - Hadoop File System (HDFS)
    - over 100 PB in a single HDFS cluster
  - an open source implementation of MapReduce:
    - Enables efficient calculations on massive amounts of data

Apache Hive [http://hive.apache.org/]
  - SQL-like access to Hadoop-stored data
  - integration of MapReduce query evaluation

Database Technology Behind Facebook II

Apache HBase [http://hbase.apache.org/]
- a Hadoop column-family database
- used for e-mails, instant messaging and SMS
- replacement for MySQL and Cassandra

Memcached [http://memcached.org/]
- distributed key-value store
- used as a cache between web servers and MySQL servers since the beginning of FB

Database Technology Behind Facebook III

Apache Giraph [http://giraph.apache.org/]
- graph database
- facebook users and connections is one very large graph
- used since 2013 for various analytic tasks

RocksDB [http://rocksdb.org/]
- high-performance key-value store
- developed internally in FB, now open-source