Data 604 – Data Management

Big Data and NoSQL Databases

Data 604: Data Management

Topics:

- CAP Theorem
- Other Data formats XML and JSON
- Big Data Databases
  - NoSQL Column Store and Document Databases
  - Examples (Mongo, Hbase, Cassandra, Kudu)
  - Google BigTable
  - Snowflake
- SQL Engines
  - Workloads
  - Engines for each workload
  - SQL Landscape

Jyotsna Potarazu - Adjunct Professor

Learning Objectives

- Identify the other alternatives to relational and NOSQL databases such as XML and JSON
- Understand the evolution of NOSQL and big data
- Understand the concepts behind NOSQL data storage and how it differs from relational databases
- Identify different types of NOSQL databases and use cases for each
- Demonstrate ability to interact with and retrieve data from Mongo, Neo4J, and Hbase

Basic Concepts - XML

- Introduced by the World Wide Web Consortium (W3C) in 1997
- Simplified subset of the Standard Generalized Markup Language (SGML)
- Aimed at storing and exchanging complex, structured documents
- Users can define new tags in XML (↔ HTML)

Basic XML Syntax

- Combination of a start tag, content, and end tag is called an XML element
- XML is case-sensitive
- Example

```
<author>
  <name>
  <first name>Bart</first name>
  <last name>Baesens</last name>
  </name>
  </author>
```

JSON

- JSON and YAML are primarily optimized for data interchange and serialization instead of representing documents as is the case for XML
- JavaScript Object Notation (JSON) provides a simple, lightweight representation whereby objects are described as name–value pairs
    - JSON provides two structured types: objects and arrays
    - Primitive types supported: string, number, Boolean, and null
    - JSON is human- and machine-readable based upon a simple syntax and also models the data in a hierarchical way
    - Structure of a JSON specification can be defined using JSON Schema
    - JSON is not a markup language and is not extensible
    - JSON documents can be simply parsed in JavaScript using the built-in `eval()` function
    - Modern web browsers also include native and fast JSON parsers

JSON - Example


```
{
 "winecellar": {
  "wine": [
   {
    "name": "Jacques Selosse Brut Initial",
    "year": "2012",
    "type": "Champagne",
    "grape": {
     "_percentage": "100",
     "__text": "Chardonnay"
    },
    "price": {
     "_currency": "EURO",
     "__text": "150"
    },
```

JSON – Example cont.

```
"geo": {
      "country": "France",
      "region": "Champagne"
     },
     "quantity": "12"
    },
    {
     "name": "Meneghetti White",
     "year": "2010",
     "type": "white wine",
     "grape": [
      {
       "_percentage": "80",
       "__text": "Chardonnay"
      },
      {
       "_percentage": "20",
       "__text": "Pinot Blanc"
      }
     ],
     "price": {
      "_currency": "EURO",
      "__text": "18"
     },
     "geo": {
      "country": "Croatia",
      "region": "Istria"
     },
     "quantity": "20"
    }
   ]
  }
}
```

Big Data

Table 1: Progression of Big Data

Parameter ↓	1990s ↓	2000s ↓	2010 and beyond ↓
Volume	Terabyte	Petabyte	Exabyte and higher
Variety	Structured	Semistructured	Unstructured/Semistructure
Velocity	Daily	Seconds	Microseconds

Source: Gartner (April 2017)

Table 2: Comparison of Relational and Nonrelational Data Stores

Relational ↓	Nonrelational ↓
Atomicity, consistency, isolation and durability (ACID): Support for transactions is part of the core design. In the CAP theorem paradigm, relational databases are CA (consistent and available).	Many well-known offerings do not support ACID transactions and adhere to "eventual consistency." Some provide limited support for consistency (e.g., at document level or single-row level but not at database level).

Source: Gartner (April 2017)

Online DB Playground

- http://www.pdbmbook.com/playground (companion to textbook Principles of Database Management)
- Need to create account, but then can experiment with different databases

NoSQL

- No SQL = Not Only SQL
- A category of recently introduced data storage and retrieval technologies not based on the relational model
- Scaling out rather than scaling up
- Natural for a cloud environment
- Supports schema on read
- Largely open source
- Not ACID compliant!
- BASE – basically available, soft state, eventually consistent

The NoSQL Movement

- RDBMSs put a lot of emphasis on keeping data consistent.
  - Entire database is consistent at all times (ACID)
- Focus on consistency may hamper flexibility and scalability
- As the data volumes or number of parallel transactions increase, capacity can be increased by
  - Vertical scaling: extending storage capacity and/or CPU power of the database server
  - Horizontal scaling: multiple DBMS servers being arranged in a cluster

The NoSQL Movement

- RDBMSs are not good at extensive horizontal scaling
  - Coordination overhead because of focus on consistency
  - Rigid database schemas
- Other types of DBMSs needed for situations with massive volumes, flexible data structures, and where scalability and availability are more important → NoSQL databases

The NoSQL Movement

- NoSQL databases
  - Describes databases that store and manipulate data in formats other than tabular relations, i.e., non-relational databases (NoREL)
- NoSQL databases aim at near-linear horizontal scalability by distributing data over a cluster of database nodes for the sake of performance as well as availability
- Eventual consistency: the data (and its replicas) will become consistent at some point in time after each transaction

The NoSQL Movement

	Relational Databases	NoSQL Databases
Data paradigm	Relational tables	Key–value (tuple) based Document based Column based Graph based XML, object based Others: time series, probabilistic, etc.
Distribution	Single-node and distributed	Mainly distributed
Scalability	Vertical scaling, harder to scale horizontally	Easy to scale horizontally, easy data replication
Openness	Closed and open source	Mainly open source
Schema role	Schema-driven	Mainly schema-free or flexible schema
Query language	SQL as query language	No or simple querying facilities, or special-purpose languages
Transaction mechanism	ACID: Atomicity, Consistency, Isolation, Durability	BASE: Basically Available, Soft state, Eventual consistency
Feature set	Many features (triggers, views, stored procedures, etc.)	Simple API
Data volume	Capable of handling normal-sized datasets	Capable of handling huge amounts of data and/or very high frequencies of read/write requests

Key–Value Stores

- Key–value-based database stores data as (key, value) pairs
  - Keys are unique
  - Hash map, or hash table or dictionary

Key-value stores

- Key-value stores
  - A simple pair of a key and an associated collection of values. Key is usually a string. Database has no knowledge of the structure or meaning of the values.

Key Name	Value
School	UMBC

Key–Value Stores

```
import java.util.HashMap;
import java.util.Map;
public class KeyValueStoreExample {
	public static void main(String... args) {
		// Keep track of age based on name
		Map<String, Integer> age_by_name = new HashMap<>();

		// Store some entries
		age_by_name.put("wilfried", 34);
		age_by_name.put("seppe", 30);
		age_by_name.put("bart", 46);
		age_by_name.put("jeanne", 19);

		// Get an entry
		int age_of_wilfried = age_by_name.get("wilfried");
		System.out.println("Wilfried's age: " + age_of_wilfried);

		// Keys are unique
		age_by_name.put("seppe", 50); // Overrides previous entry
	}
```

Key–Value Stores

- Keys (e.g., “bart”, “seppe”) are hashed by means of a so-called **hash function**
  - A hash function takes an arbitrary value of arbitrary size and maps it to a key with a fixed size, which is called the hash value
  - Each hash can be mapped to a space in computer memory

Key
wilfried
seppe
bart
jeanne

Hash	Key
01	(wilfried,34)
03	(seppe,30)
07	(bart,46)
08	(jeanne,19)

Key–Value Stores

- NoSQL databases are built with horizontal scalability support in mind
- Distribute hash table over different locations
- Assume we need to spread our hashes over three servers
  - Hash every key (“wilfried”, “seppe”) to a server identifier
  - index(hash) = mod(hash, nrServers) + 1

Key–Value Stores

Key
wilfried
seppe
bart
jeanne

Hash function

Hash	Index (= mod 3 + 1)
01	2
03	1
07	2
08	3

Server #1	Entries
03	(seppe,30)

Server #2	Entries
01	(wilfried,34)
07	(bart,46)

Server #3	Entries
08	(jeanne,19)

Sharding!

Key–Value Stores

- Example: Memcached
  - Implements a distributed memory-driven hash table (i.e., a key–value store), which is put in front of a traditional database to speed up queries by caching recently accessed objects in RAM
  - Caching solution

Key–Value Stores

```
import java.util.ArrayList;
import java.util.List;
import net.spy.memcached.AddrUtil;
import net.spy.memcached.MemcachedClient;

public class MemCachedExample {
 public static void main(String[] args) throws Exception {
 List<String> serverList = new ArrayList<String>() {
 {
 this.add("memcachedserver1.servers:11211");
 this.add("memcachedserver2.servers:11211");
 this.add("memcachedserver3.servers:11211");
 }
 };
```

Key–Value Stores

```java
MemcachedClient memcachedClient = new MemcachedClient(
AddrUtil.getAddresses(serverList));


// ADD adds an entry and does nothing if the key already exists
// Think of it as an INSERT
// The second parameter (0) indicates the expiration - 0 means no expiry
memcachedClient.add("marc", 0, 34);
memcachedClient.add("seppe", 0, 32);
memcachedClient.add("bart", 0, 66);
memcachedClient.add("jeanne", 0, 19);



// SET sets an entry regardless of whether it exists
// Think of it as an UPDATE-OR-INSERT
memcachedClient.add("marc", 0, 1111); // <- ADD will have no effect
memcachedClient.set("jeanne", 0, 12); // <- But SET will
```

Key–Value Stores

```
// REPLACE replaces an entry and does nothing if the key does not exist
// Think of it as an UPDATE
memcachedClient.replace("not_existing_name", 0, 12); // <- Will have no effect
memcachedClient.replace("jeanne", 0, 10);

// DELETE deletes an entry, similar to an SQL DELETE statement
memcachedClient.delete("seppe");

// GET retrieves an entry
Integer age_of_marc = (Integer) memcachedClient.get("marc");
Integer age_of_short_lived = (Integer) memcachedClient.get("short_lived_name");
Integer age_of_not_existing = (Integer) memcachedClient.get("not_existing_name");
Integer age_of_seppe = (Integer) memcachedClient.get("seppe");
System.out.println("Age of Marc: " + age_of_marc);
System.out.println("Age of Seppe (deleted): " + age_of_seppe);
System.out.println("Age of not existing name: " + age_of_not_existing);
System.out.println("Age of short lived name (expired): " + age_of_short_lived);

memcachedClient.shutdown();

 }
}
```

Key–Value Stores

- Request coordination
- Consistent hashing
- Replication and redundancy
- Eventual consistency
- Stabilization
- Integrity constraints and querying

Request Coordination

- In many NoSQL implementations (e.g., Cassandra, Google's BigTable, Amazon's DynamoDB), all nodes implement the same functionality and are all able to perform the role of request coordinator
- Need for membership protocol
  - Dissemination
    - Based on periodic, pairwise communication
  - Failure detection

Consistent Hashing

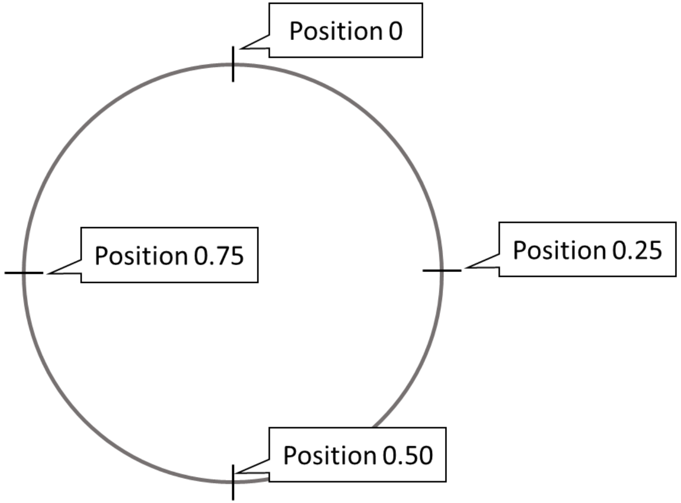
- **Consistent hashing** schemes are often used, which avoid having to remap each key to a new node when nodes are added or removed
- Suppose we have a situation in which ten keys are distributed over three servers (n = 3) with the following hash function:
  - $h(key) = key$ modulo n

Consistent Hashing

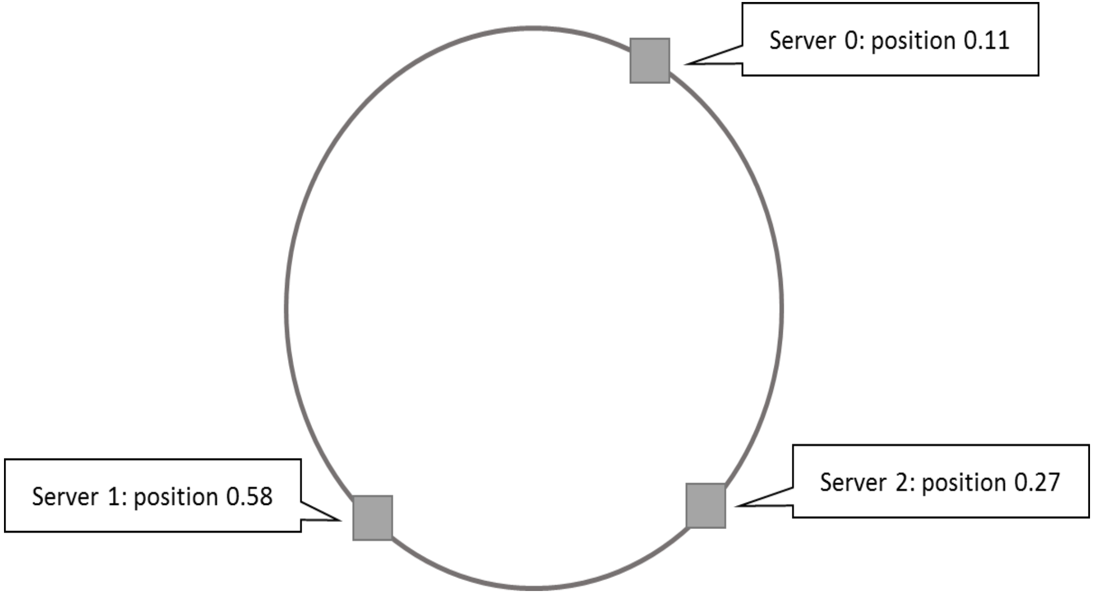
	n		
key	3	2	4
0	0	0	0
1	1	1	1
2	2	0	2
3	0	1	3
4	1	0	0
5	2	1	1
6	0	0	2
7	1	1	3
8	2	0	0
9	0	1	1

Consistent Hashing

- At the core of a consistent hashing setup is a so-called **“ring”-topology**, which is basically a representation of the number range [0,1]:

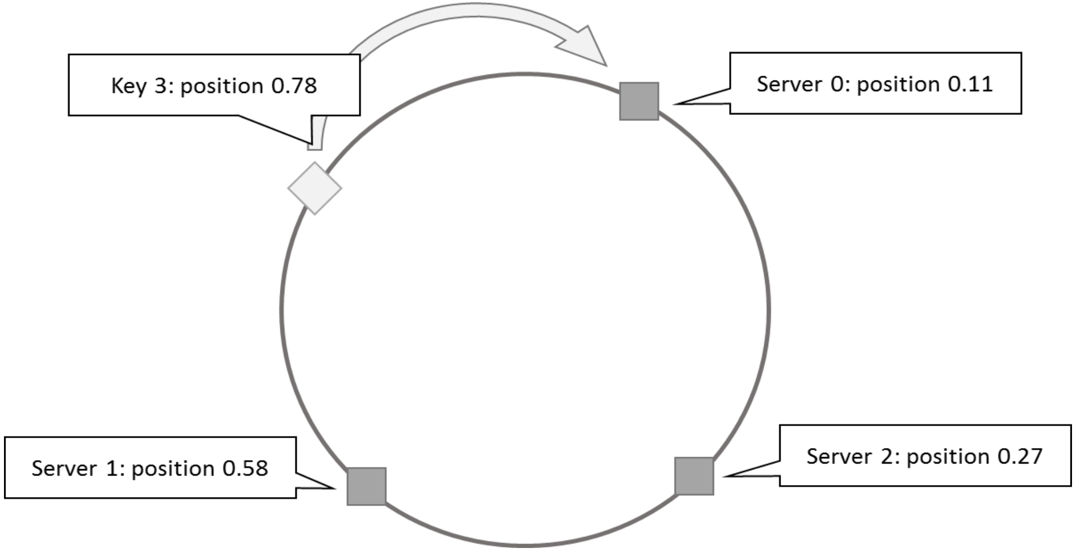


Consistent Hashing



Consistent Hashing

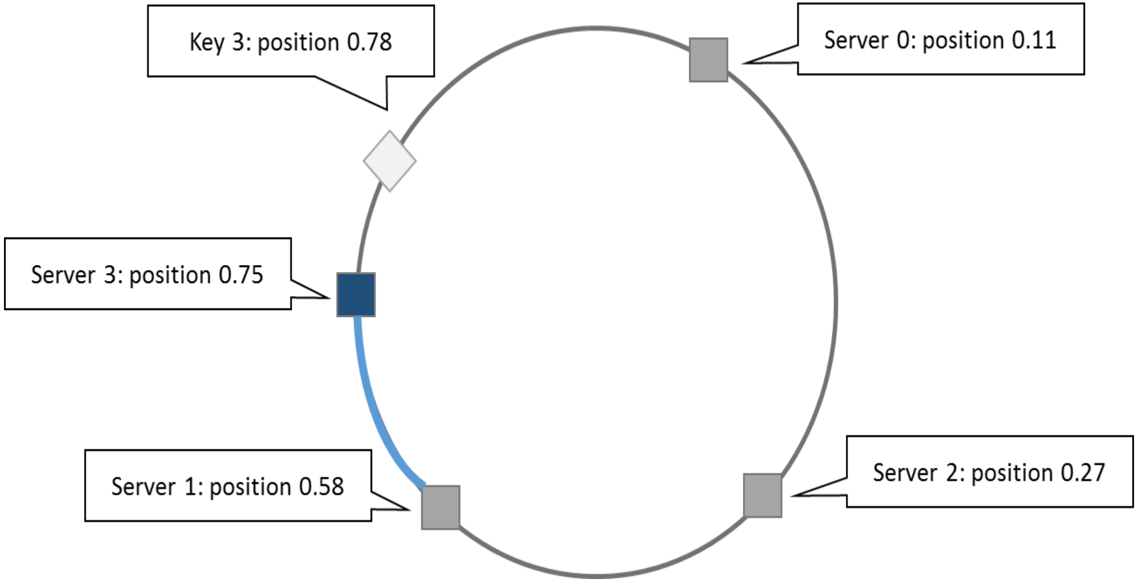
- Hash each key to a position on the ring, and store the actual key–value pair on the first server that appears clockwise of the hashed point on the ring



Consistent Hashing

- Because of the uniformity property of a “good” hash function, roughly $1/n$ of key–value pairs will end up being stored on each server
- Most of the key–value pairs will remain unaffected in the event that a machine is added or removed

Consistent Hashing



Replication and Redundancy

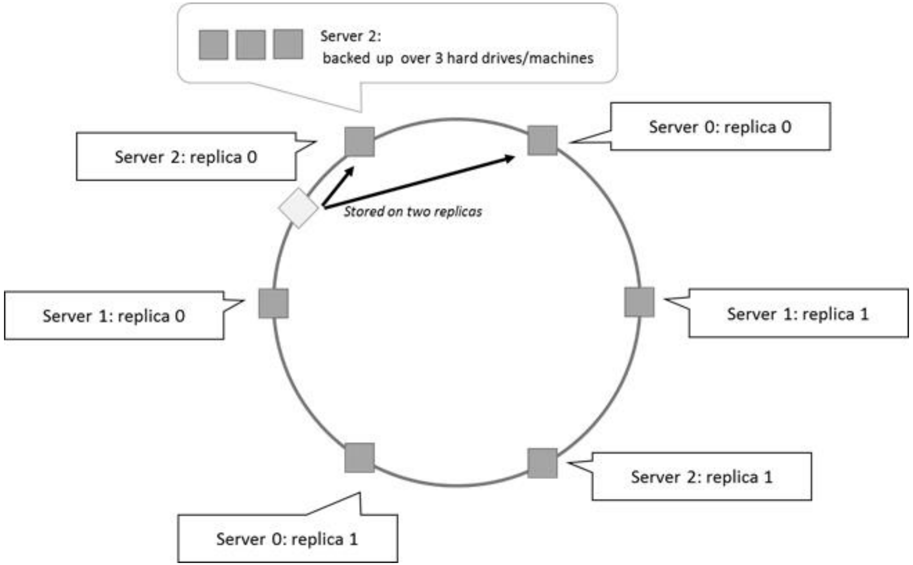
- Problems with consistent hashing:
  - If two servers end up being mapped close to one another, one of these nodes will end up with few keys to store
  - In the case that a server is added, all of the keys moved to this new node originate from just one other server
- Instead of mapping a server *s* to a single point on our ring, we map it to multiple positions, called **replicas**
- For each physical server *s*, we hence end up with *r* (the number of replicas) points on the ring
- Note: each of the replicas still represents the same physical instance (↔ redundancy)
  - Virtual nodes

Replication and Redundancy

- To handle data replication or redundancy, many vendors extend the consistent hashing mechanism so that key–value pairs are duplicated across multiple nodes
  - e.g., by storing the key–value pair on two or more nodes clockwise from the key's position on the ring

Replication and Redundancy

- It is also possible to set up a full redundancy scheme in which each node itself corresponds to multiple physical machines, each storing a fully redundant copy of the data



Eventual Consistency

- Membership protocol does not guarantee that every node is aware of every other node *at all times*
  - it will reach a consistent state over time
- State of the network might not be perfectly consistent at any moment in time, though will become eventually consistent at a future point in time
- Many NoSQL databases guarantee so-called **eventual consistency**

Eventual Consistency

- Most NoSQL databases follow the **BASE** principle
  - Basically Available, Soft state, Eventual consistency
- **CAP theorem** states that a distributed computer system cannot guarantee the following three properties at the same time:
  - Consistency (all nodes see the same data at the same time)
  - Availability (guarantees that every request receives a response indicating a success or failure result)
  - Partition tolerance (the system continues to work even if nodes go down or are added)

Eventual Consistency

- Most NoSQL databases sacrifice the consistency part of CAP in their setup, instead striving for eventual consistency
- The full BASE acronym stands for:
  - Basically Available: NoSQL databases adhere to the availability guarantee of the CAP theorem
  - Soft state: the system can change over time, even without receiving input
  - Eventual consistency: the system will become consistent over time

Stabilization

- The operation which repartitions hashes over nodes in case nodes are added or removed is called **stabilization**
- If a consistent hashing scheme is being applied, the number of fluctuations in the hash–node mappings will be minimized.

Integrity Constraints and Querying

- Key–value stores represent a very diverse gamut of systems
- Full-blown DBMSs versus caches
- Only limited query facilities are offered
  - e.g. put and set
- Limited to no means to enforce structural constraints
  - DBMS remains agnostic to the internal structure
- No relationships, referential integrity constraints, or database schema can be defined

Tuple and Document Stores

- A **tuple store** is similar to a key–value store, with the difference that it does not store pairwise combinations of a key and a value, but instead stores a unique key together with a vector of data
- Example:
  - `marc -> ("Marc", "McLast Name", 25, "Germany")`
- No requirement to have the same length or semantic ordering (schema-less!)

Tuple and Document Stores

- Various NoSQL implementations do, however, permit organizing entries in semantical groups (aka collections or tables)
- Examples:
  - `Person:marc -> ("Marc", "McLast Name", 25, "Germany")`
  - `Person:harry -> ("Harry", "Smith", 29, "Belgium")`

Tuple and Document Stores

- **Document stores** store a collection of attributes that are labeled and unordered, representing items that are semi-structured
- Example:

```
{
 Title = "Harry Potter"
 ISBN = "111-1111111111"
 Authors = [ "J.K. Rowling" ]
 Price = 32
 Dimensions = "8.5 x 11.0 x 0.5"
 PageCount = 234
 Genre = "Fantasy"
}
```

Tuple and Document Stores

- Most modern NoSQL databases choose to represent documents using JSON

```
{
        "title": "Harry Potter",
        "authors": ["J.K. Rowling", "R.J. Kowling"],
        "price": 32.00,
        "genres": ["fantasy"],
        "dimensions": {
                "width": 8.5,
                "height": 11.0,
                "depth": 0.5
        },
        "pages": 234,
        "in_publication": true,
        "subtitle": null
}
```

Tuple and Document Stores

- Items with keys
- Filters and queries
- Complex queries and aggregation with MapReduce
- SQL after all …

Items with Keys

- Most NoSQL document stores will allow you to store items in tables (collections) in a schema-less manner, but will enforce that a primary key be specified
    - e.g. Amazon’s DynamoDB, MongoDB (_id)
- A primary key will be used as a partitioning key to create a hash and determine where the data will be stored

Column-Oriented Databases

- All data for a column is stored together with the data serving as the primary key
- Improves retrieval time to read group of data from specific columns which is optimal for data warehouses. Necessary data for retrieval is stored together
- Takes longer if one has to retrieve all data for one row as is needed for OLTP systems.
- Databases: Hbase, Kudu, Parquet, Greenplum, PostgresSQL, MapD

Wide-column stores

- Wide-column stores
  - Rows and columns with 2 dimensional key-value store. Distribution of data based on both key values (records) and columns, using “column groups/families”
  - Groups of columns for given rows can be grouped and stored together
  - Example: Hbase, Apache Cassandra (based on Google BigTable and Amazon DynamoDB)

Document Stores

- Document stores
  - Like a key-value store, but “document” goes further than “value”. Document is structured so specific elements can be manipulated separately in JSON or XML.
  - Example: MongoDB
  - Book = [{ “title”: “Fire & Blood”, “author”: “George Martin”, “year”, “2003”}, { “title”: “A Dance With Dragons”, “author”: “George Martin”, “year”, “2016”}]

Graph-oriented Databases

- Graph-oriented database
  - Maintain information regarding the relationships between data items. Nodes with properties. Connections between nodes (relationships) can also have properties.
  - Borrows the concept of relationships from RDBMs in that it emphasizes relationships. However it is a simpler node based design that does not use associative tables to make it faster to read and visualize patterns to answer business questions. Useful for social networks and fraud.

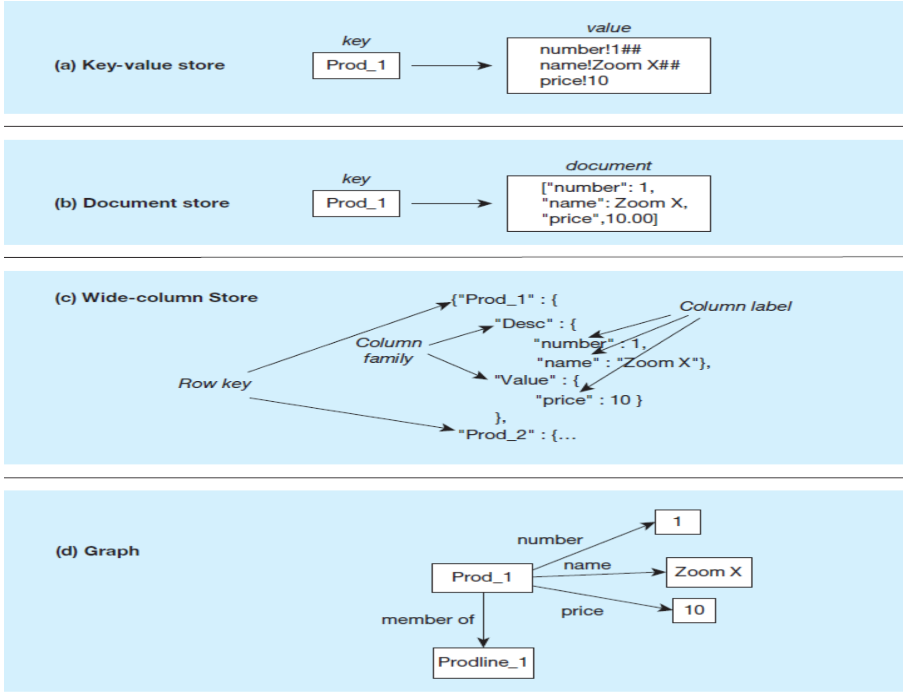
Comparison of NoSQL Types

Source: www.slideshare.net/bscofield/nosql-codemash-2010. Courtesy of Ben Scofield.

	Key-Value Store	Document Store	Column Oriented	Graph
Performance	High	High	High	Variable
Scalability	High	Variable/High	High	Variable
Flexibility	High	High	Moderate	High
Complexity	None	Low	Low	High
Functionality	Variable	Variable (Low)	Minimal	Graph theory

NoSQL Framework

Some of the example structures have been adapted from Kauhanen (2010)



MongoDB

- A document-store database,
- Documents stored as JSON binary object
- Collections
  - Equivalent to tables in a relational database
  - A set of documents intended to be stored together
- Documents
  - Equivalent to rows in a relational database
  - Documents do not need to have the same structure (unlike rows)
  - _id property for uniquely identifying a row
- Relationships
  - _id property serves as “primary key”
  - Another document can have a “foreign” key as another JSON property

MongoDB Sample

a) A document in the Product collection

```
{
   "_id": "1",
   "name": "OLED TV",
   "desc": "75in TV",
   "width": 60,
   "height": 30,
   "depth": 5,
  "reviews": [
      {
         "author": 1,
         "ratingstars": 4,
         "comment": "Amazing TV"
      },
      {
         "author": 2,
         "ratingstars": 2,
         "comment": "Very disappointed with the TV"
      }
   ]
}
```

b) A document in the Author collection

```
{
      "_id": 1
      "First Name": "Jane",
      "Last Name: "Smith"
}
```

Column-Oriented Databases

- A **column-oriented DBMS** is a database management system that stores data tables as sections of columns of data
- Useful if:
  - Aggregates are regularly computed over large numbers of similar data items
  - Data are sparse, i.e., columns with many null values
- Can also be an RDBMS, key–value, or document store

Column-Oriented Databases

- Example

Id	Genre	Title	Price	Audiobook price
1	fantasy	My first book	20	30
2	education	Beginners guide	10	null
3	education	SQL strikes back	40	null
4	fantasy	The rise of SQL	10	null

- Row-based databases are not efficient at performing operations that apply to the entire dataset
  - Need indexes which add overhead

Column-Oriented Databases

- In a column-oriented database, all values of a column are placed together on disk

  Genre: fantasy:1,4 education:2,3

  Title: My first book:1 Beginners guide:2 SQL strikes back:3 The rise of SQL:4

  Price: 20:1 10:2,4 40:3

  Audiobook price: 30:1

- A column matches the structure of a normal index in a row-based system
- Operations such as find all records with price equal to 10 can now be executed directly
- Null values do not take up storage space anymore

Column-Oriented Databases

- Disadvantages
  - Retrieving all attributes pertaining to a single entity becomes less efficient
  - Join operations will be slowed down
- Examples
  - Google BigTable, Cassandra, HBase, and Parquet

HBase

- First Hadoop database inspired by Google’s Bigtable
- Runs on top of HDFS
- NoSQL-like data storage platform
  - No typed columns, triggers, advanced query capabilities, etc.
- Offers a simplified structure and query language in a way that is highly scalable and can tackle large volumes

HBase

- Similar to RDBMSs, HBase organizes data in tables with rows and columns
- HBase table consists of multiple rows
- A row consists of a row key and one or more columns with values associated with them
- Rows in a table are sorted alphabetically by the row key

HBase

- Each column in HBase is denoted by a column family and qualifier (separated by a colon, ":")
- A column family physically co-locates a set of columns and their values
- Every row has the same column families, but not all column families need to have a value per row
- Each cell in a table is hence defined by a combination of the row key, column family and column qualifier, and a timestamp

HBase

- Example: HBase table to store and query users
- The row key will be the user id
- column families:qualifiers
  - name:first
  - name:last
  - email (without a qualifier)

HBase

```
hbase(main):001:0> create 'users', 'name', 'email'
0 row(s) in 2.8350 seconds

=> Hbase::Table - users


hbase(main):002:0> describe 'users'
Table users is ENABLED
users
COLUMN FAMILIES DESCRIPTION
{NAME => 'email', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false', K
EEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', C
OMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '6
5536', REPLICATION_SCOPE => '0'}
{NAME => 'name', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false', KE
EP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', CO
MPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65
536', REPLICATION_SCOPE => '0'}
2 row(s) in 0.3250 seconds
```

HBase

```
hbase(main):003:0> list 'users'
TABLE
users
1 row(s) in 0.0410 seconds

=> ["users"]
```

HBase

```
hbase(main):006:0> put 'users', 'seppe', 'name:first', 'Seppe'
0 row(s) in 0.0200 seconds

hbase(main):007:0> put 'users', 'seppe', 'name:last', 'vanden Broucke'
0 row(s) in 0.0330 seconds

hbase(main):008:0> put
'users', 'seppe', 'email', 'seppe.vandenbroucke@kuleuven'
0 row(s) in 0.0570 seconds

hbase(main):009:0> scan 'users'
ROW                  COLUMN+CELL
 seppe               column=email:, timestamp=1495293082872, value=seppe.vanden
                     broucke@kuleuven.be
 seppe               column=name:first, timestamp=1495293050816, value=Seppe
 seppe               column=name:firstt, timestamp=1495293047100, value=Seppe
 seppe               column=name:last, timestamp=1495293067245, value=vanden Broucke

1 row(s) in 0.1170 seconds
```

HBase

```
hbase(main):011:0> get 'users', 'seppe'
COLUMN                         CELL
 email:                        timestamp=1495293082872, value=seppe.vandenbroucke@kuleuven.be
 name:first                    timestamp=1495293050816, value=Seppe
 name:firstt                   timestamp=1495293047100, value=Seppe
 name:last                     timestamp=1495293067245, value=vanden Broucke
4 row(s) in 0.1250 seconds


hbase(main):018:0> put 'users', 'seppe', 'email', 'seppe@kuleuven.be'
0 row(s) in 0.0240 seconds


hbase(main):019:0> get 'users', 'seppe', 'email'
COLUMN                         CELL
 email:                        timestamp=1495293303079, value=seppe@kuleuven.be
1 row(s) in 0.0330 seconds
```

HBase

- HBase's query facilities are very limited
- Essentially a key–value, distributed data store with simple get/put operations
- Includes facilities to write MapReduce programs
- Hbase (similar to Hadoop) doesn't perform well on less than five HDFS DataNodes with an additional NameNode
  - Only makes the effort worthwhile when you can invest in, set up, and maintain at least 6–10 nodes

Cassandra

- Wide Column store, key-value distributed database
- Developed by one of the creators of “Dynamo’ for Facebook to optimize and workaround issues with ‘Inbox Searches’
- Use by well known vendors such as Apple and Netflix
- Supports Hadoop by API and MapReduce
- Scales across locations

KUDU

- Column store alternative for structured data
- Uses horizontal partitioning
- Fits within the Hadoop ecosystem to resolve integrity issues while maintaining high performance. It fits between the sequential access systems such as HDFS and the low latency, random read systems such as Cassandra and HBASE
- Stores data on Linux but can share HDFS partitions
- It does not have a SQL language but can be paired with Impala to write SQL Queries against KUDU

KUDU - Code

- Cloudera Quick Start VM - https://github.com/cloudera/kudu-examples
- Cloudera Python code example - https://github.com/cloudera/kudu-examples/commit/26b81875168b2ccfd87c965e873bc876f1ad16ef
- Cloudera – Impala with Kudu http://www.cloudera.com/documentation/enterprise/latest/topics/kudu_impala.html

Neo4j

- Neo4j – one of the more well known graph databases https://neo4j.com/developer/graph-db-vs-rdbms/
- Python info for Neo4j: https://neo4j.com/developer/python/

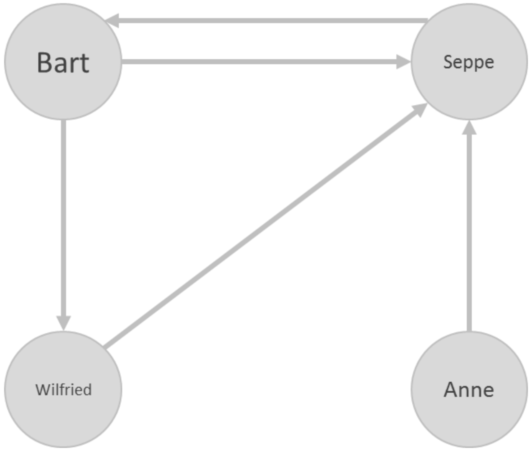
```
@app.route("/graph")
def get_graph():
    db = get_db()
    results = db.run("MATCH (m:Movie)<-[:ACTED_IN]-(a:Person) "
             "RETURN m.title as movie, collect(a.name) as cast "
             "LIMIT {limit}", {"limit": request.args.get("limit", 100)})
```

Graph Databases

- Location-based services
- Recommender systems
- Social media (e.g., Twitter and FlockDB)
- Knowledge-based systems

Graph-Based Databases

- **Graph databases** apply graph theory to the storage of information of records
- Graphs consist of **nodes** and **edges**



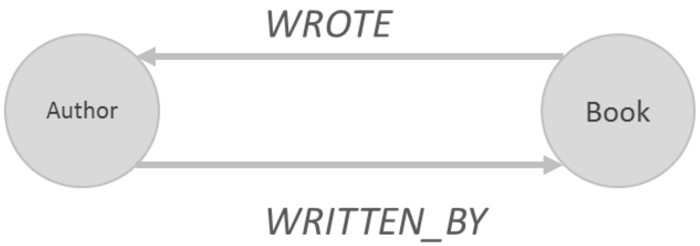
Graph-Based Databases

- One-to-one, one-to-many, and many-to-many structures can easily be modeled in a graph
- Consider the N–M relationship between books and authors
- RDBMS needs three tables: Book, Author and Books_Authors
- SQL query to return all book titles for books written by a particular author would look like this:

```
SELECT title
FROM books, authors, books_authors
WHERE author.id = books_authors.author_id
 AND books.id = books_authors.book_id
 AND author.name = "Bart Baesens"
```

Graph-Based Databases

- In a graph database (using **Cypher query language** from Neo4j)




```
MATCH (b:Book)<-[:WRITTEN_BY]-(a:Author)
WHERE a.name = "Bart Baesens"
RETURN b.title
```

Graph-Based Databases

- A graph database is a hyper-relational database, in which JOIN tables are replaced by more interesting and semantically meaningful relationships that can be navigated and/or queried using graph traversal based on graph pattern matching.

Graph-Based Databases

- Cypher Overview (Neo4j)
- Exploring a social graph

Cypher Overview

- Cypher is a declarative, text-based query language, containing many similar operations as SQL
- Contains a special **MATCH** clause to match those patterns using symbols that look like graph symbols as drawn on a whiteboard
- Nodes are represented by parentheses, representing a circle: `()`
- Nodes can be labeled in case they need to be referred to elsewhere, and be further filtered by their type, using a colon: `(b:Book)`
- Edges are drawn using either `--` or `-->`, representing a unidirectional line or an arrow representing a directional relationship, respectively

Cypher Overview

- Relationships can be filtered by putting square brackets in the middle:
  `(b:Book)<-[:WRITTEN_BY]-(a:Author)`

Cypher Overview

```
MATCH (b:Book)
RETURN b;

MATCH (b:Book)
RETURN b
ORDER BY b.price DESC
LIMIT 20;

MATCH (b:Book)
WHERE b.title = "Beginning Neo4j"
RETURN b;

MATCH (b:Book {title:"Beginning Neo4j"})
RETURN b;
```

Cypher Overview

- JOIN clauses are expressed using direct relational matching

```
MATCH (c:Customer)-[p:PURCHASED]->(b:Book)<-[:WRITTEN_BY]-
(a:Author)
WHERE a.name = "Wilfried Lemahieu"
 AND c.age > 30
 AND p.type = "cash"
RETURN DISTINCT c.name;
```

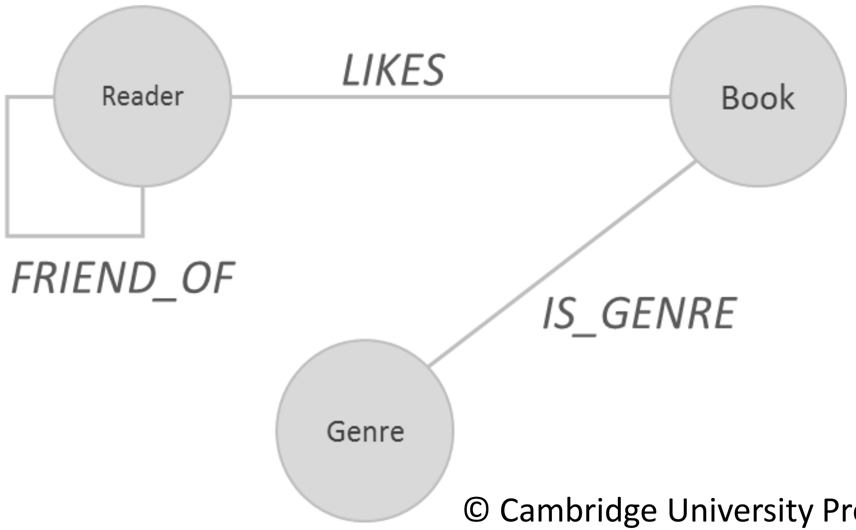
Cypher Overview

- Graph databases are great at managing tree structures
- Example:
  - A tree of book genres and books can be placed under any category level
  - A query to fetch a list of all books in the category “Programming” and all its subcategories
- Cypher can express queries over hierarchies and transitive relationships of any depth simply by appending an asterisk * after the relationship type and providing optional min..max limits

```
MATCH (b:Book)-[:IN_GENRE]->(:Genre)
 -[:PARENT*0..]-(:Genre {name:"Programming"})
RETURN b.title;
```

Exploring a Social Graph

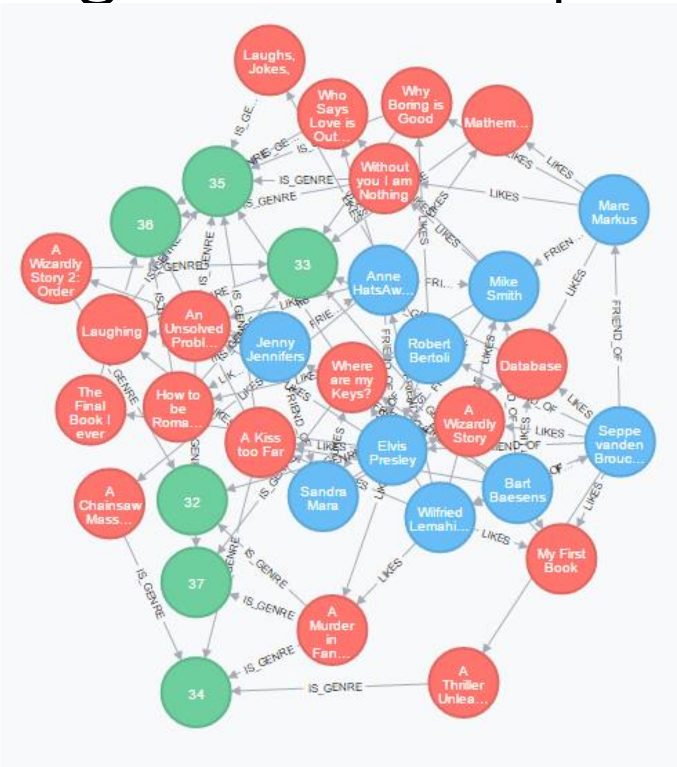
- Example: a social graph for a book-reading club, modeling genres, books, and readers



Exploring a Social Graph

```
CREATE (Bart:Reader {name:'Bart Baesens', age:32})

CREATE (Seppe:Reader {name:'Seppe vanden Broucke', age:30})

…

CREATE (Fantasy:Genre {name:'fantasy'})

CREATE (Education:Genre {name:'education'})

…

CREATE (b01:Book {title:'My First Book'})

CREATE (b02:Book {title:'A Thriller Unleashed'})

…

CREATE

 (b01)-[:IS_GENRE]->(Education),

 (b02)-[:IS_GENRE]->(Thriller),

…

CREATE

 (Bart)-[:FRIEND_OF]->(Seppe),

 (Bart)-[:FRIEND_OF]->(Wilfried),

…

CREATE

 (Bart)-[:LIKES]->(b01), (Bart)-[:LIKES]->(b03),
 (Bart)-[:LIKES]->(b05), (Bart)-[:LIKES]->(b06),

 …
```

Exploring a Social Graph



Exploring a Social Graph

- Who likes romance books?

```
MATCH (r:Reader)--(:Book)--(:Genre {name:'romance'})
RETURN r.name
```

Returns:

Elvis Presley

Mike Smith

Anne HatsAway

Robert Bertoli

…

Exploring a Social Graph

- Who are Bart's friends that liked Humor books?

```
MATCH (me:Reader)--(friend:Reader)--(b:Book)--(g:Genre)
WHERE g.name = 'humor' AND me.name = 'Bart Baesens'
RETURN DISTINCT friend.name
```

- Can you recommend some humor books that Seppe's friends liked and Seppe has not liked yet?

```
MATCH (me:Reader)--(friend:Reader),
  (friend)--(b:Book),
  (b)--(genre:Genre)
 WHERE NOT (me)--(b)
  AND me.name = 'Seppe vanden Broucke' AND genre.name = 'humor'
 RETURN DISTINCT b.title
```

Exploring a Social Graph

- Get a list of people who have liked books Bart liked, sorted by most liked books in common

```
MATCH (me:Reader)--(b:Book),
 (me)--(friend:Reader)--(b)
WHERE me.name = 'Bart Baesens'
RETURN friend.name, count(*) AS common_likes
ORDER BY common_likes DESC
```

friend.name	common_likes
Wilfried Lemahieu	3
Seppe vanden Broucke	2
Mike Smith	1

Evaluating NoSQL DBMSs

- Most NoSQL implementations have yet to prove their true worth in the field
- Some queries or aggregations are particularly difficult; map–reduce interfaces are harder to learn and use
- Some early adopters of NoSQL were confronted with some sour lessons
  - e.g., Twitter and HealthCare.gov

Evaluating NoSQL DBMSs

- NoSQL vendors start focusing again on robustness and durability, whereas RDBMS vendors start implementing features to build schema-free, scalable data stores
- NewSQL: blend the scalable performance and flexibility of NoSQL systems with the robustness guarantees of a traditional RDBMS

Google BigTable

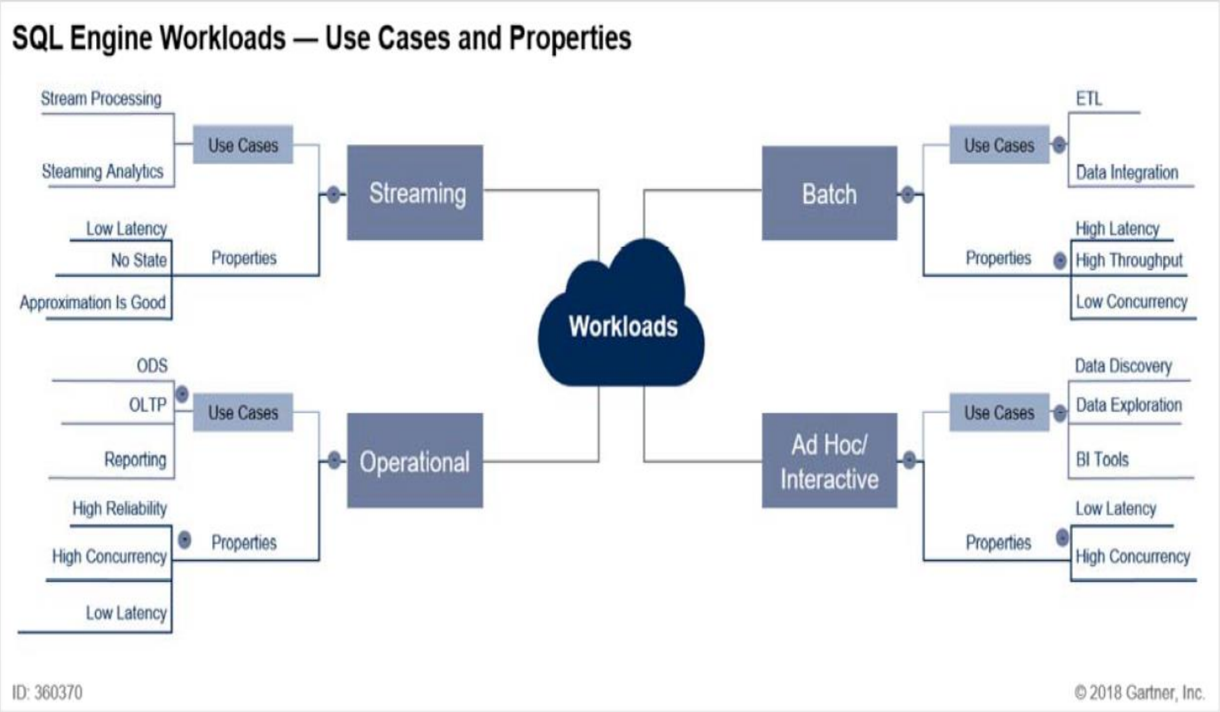
- Distributed data storage system that can scale to petabytes
- Simple data model that treats data an uninterpreted strings
- Geared for machine learning applications
- Google BigTable URL https://cloud.google.com/bigtable
- Documentation: https://cloud.google.com/bigtable/docs

Data Sharding/Horizontal Partitioning

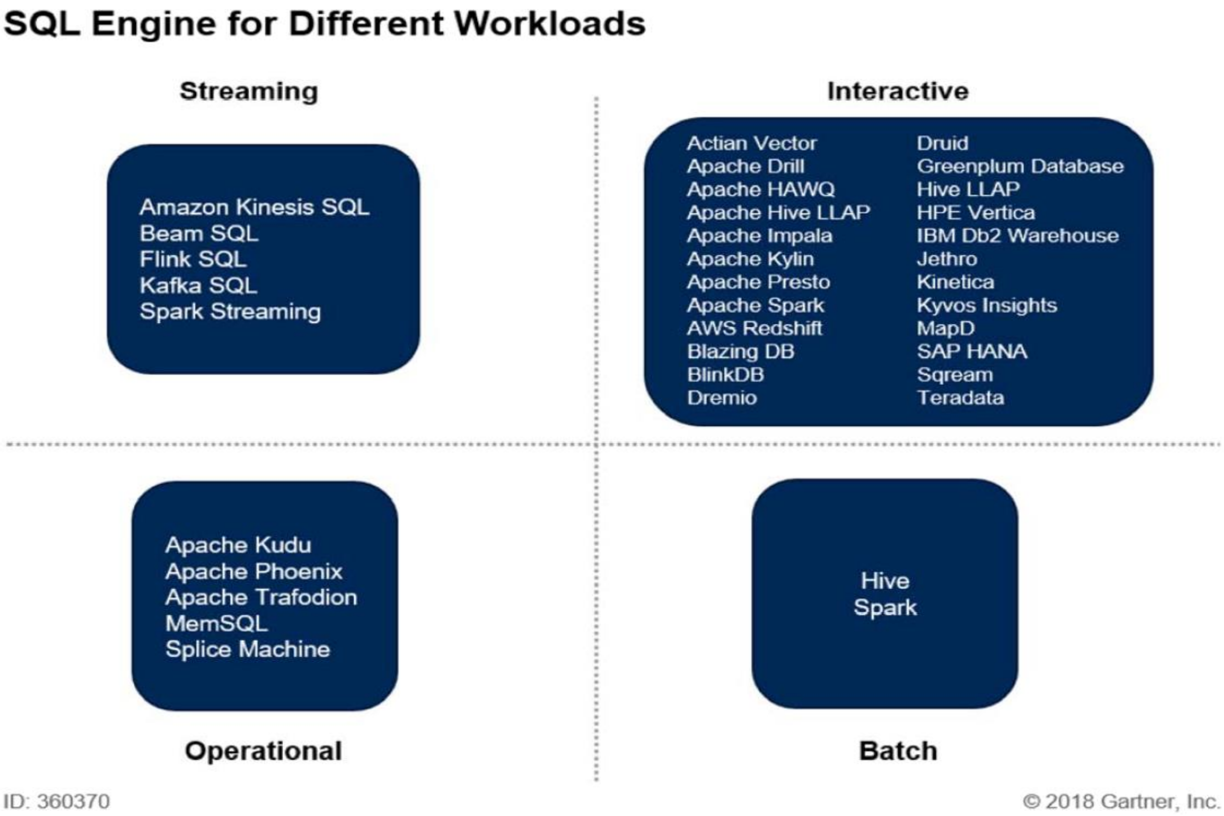
- Sharding
  - Data is chunked and distributed across multiple instances and schemas
  - Either shards data across independence database instances or uses distributed storage engine while keeping logical database instance intact
  - NewSQL databases (voltDB, clustrix)
- Horizontal partitioning - tables are chunked to reduce row size often by constraints or grouping of data
  - Optimal when searches include the constraint as a filter
  - If that constraint is not used, it can significantly slow down performance
  - Use in enterprise editions of relational database systems

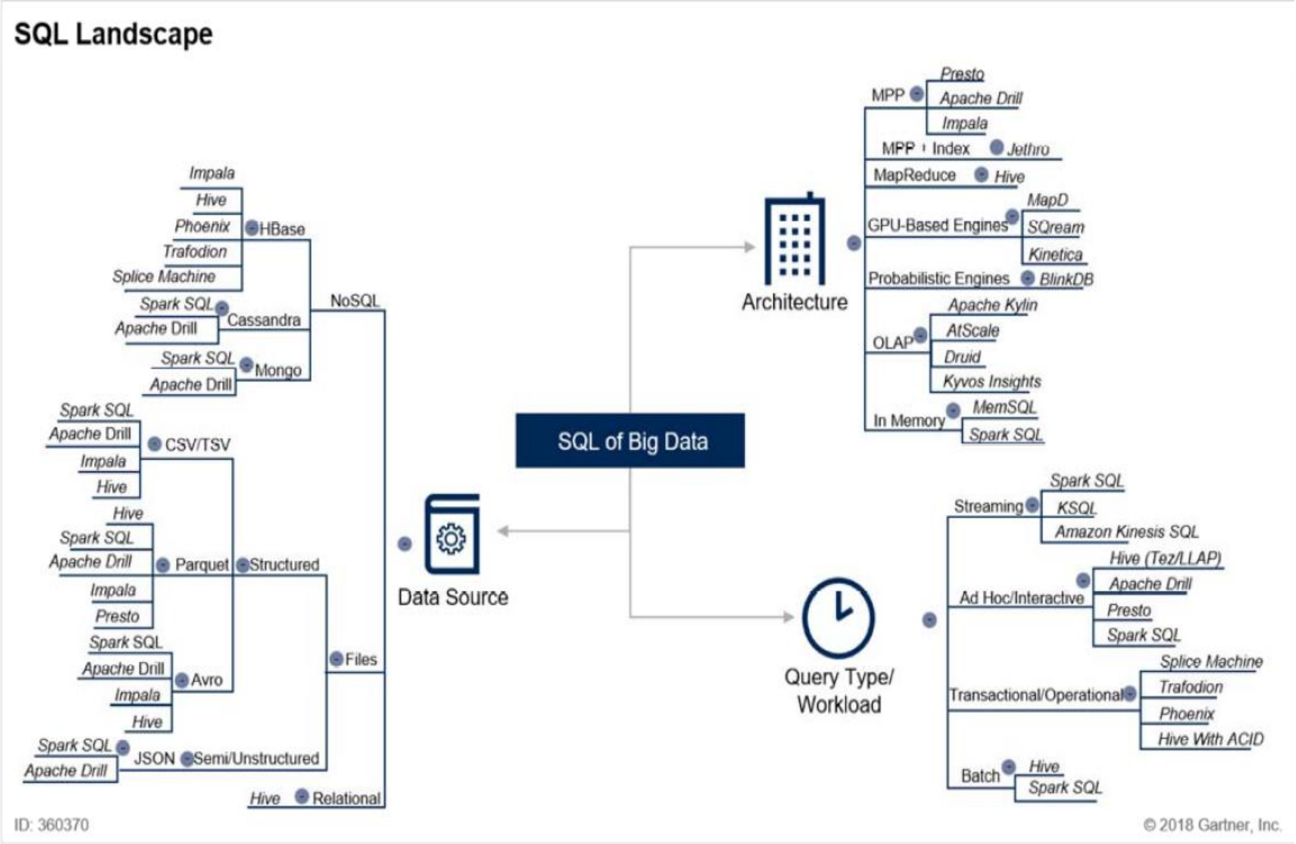
Evaluating NoSQL DBMSs

	RDBMSs	NoSQL	NewSQL
Relational	Yes	No	Yes
SQL	Yes	No	Yes
Column stores	No	Yes	Yes
Scalability	Limited	Yes	Yes
Eventually consistent	Yes	Yes	Yes
BASE	No	Yes	No
Big volumes of data	No	Yes	Yes
Schema-less	No	Yes	No



Source: Gartner (September 2018)


SQL Engine for Different Workloads
Streaming
Amazon Kinesis SQL
Beam SQL
Flink SQL
Kafka SQL
Spark Streaming
Interactive
Actian Vector
Apache Drill
Apache HAWQ
Apache Hive LLAP
Apache Impala
Apache Kylin
Apache Presto
Apache Spark
AWS Redshift
Blazing DB
BlinkDB
Dremio
Druid
Greenplum Database
Hive LLAP
HPE Vertica
IBM Db2 Warehouse
Jethro
Kinetica
Kyvos Insights
MapD
SAP HANA
Sqream
Teradata
Apache Kudu
Apache Phoenix
Apache Trafodion
MemSQL
Splice Machine
Operational
Hive
Spark
Batch
ID: 360370
© 2018 Gartner, Inc.



Source: Gartner (September 2018)