Introduction to NoSQL DB/MongoDB

Advantages of RDBMS

- Relational databases are well-documented and mature technologies, and RDBMSs are sold and maintained by several established corporations.
- SQL standards are well-defined and commonly accepted.
- A large pool of qualified developers have experience with SQL and RDBMS.
- All RDBMS are ACID-compliant, meaning they satisfy the requirements of Atomicity, Consistency, Isolation, and Durability.

Disadvantages of RDBMS

- RDBMSs don't work well with unstructured or semi-structured data due to schema and type constraints ➔ ill-suited for large analytics or IoT event loads.
- The tables in your relational database will not necessarily map one-to-one with an object or class representing the same data.
- When migrating one RDBMS to another, schemas and types must generally be identical between source and destination tables for migration to work (schema constraint).
- For many of the same reasons, extremely complex datasets or those containing variable- length records are generally difficult to handle with an RDBMS schema.

Types of Data

- Data can be broadly classified into four types:
  1. Structured Data:
     - Have a predefined model, which organizes data into a form that is relatively easy to store, process, retrieve and manage
     - E.g., relational data
  2. Unstructured Data:
     - Opposite of structured data
     - E.g., Flat binary files containing text, video or audio
     - Note: data is not completely devoid of a structure (e.g., an audio file may still have an encoding structure and some metadata associated with it)

ACID Properties in DBMS

- **Atomicity**: a transaction to be treated as a single, indivisible, logical unit of work.
- **Consistency**: a database condition in which all data integrity constraints are satisfied.
  - Consistent means that the data accurately reflects any changes up to a certain point in time.
- **Isolation**: Transactions cannot interfere with each other. Transactions are independent.
- **Durability**: Once a transaction is committed, it will remain/persist in the system, even with system failures afterwards.
  - Completed transactions persist even when servers restart or there are power failures.

C and Latency Tradeoff

- Amazon claims that just an extra one tenth of a second on their response times will cost them 1% in sales.
- Google said they noticed that just a half a second increase in latency caused traffic to drop by a fifth.

4 Key Words on NoSQL

- Scale
- Speed
- Cloud
- New Data

What is NoSQL?

- non-relational
- simple API
- schema-free
- open-source
- horizontally scalable (sharding)
- replication support
- eventually consistent /BASE

BASE Properties in NoSQL DBMS

- The CAP theorem ➔ impossible to guarantee strict Consistency & Availability while being able to tolerate network partitions
- NoSQL databases have relaxed ACID properties:

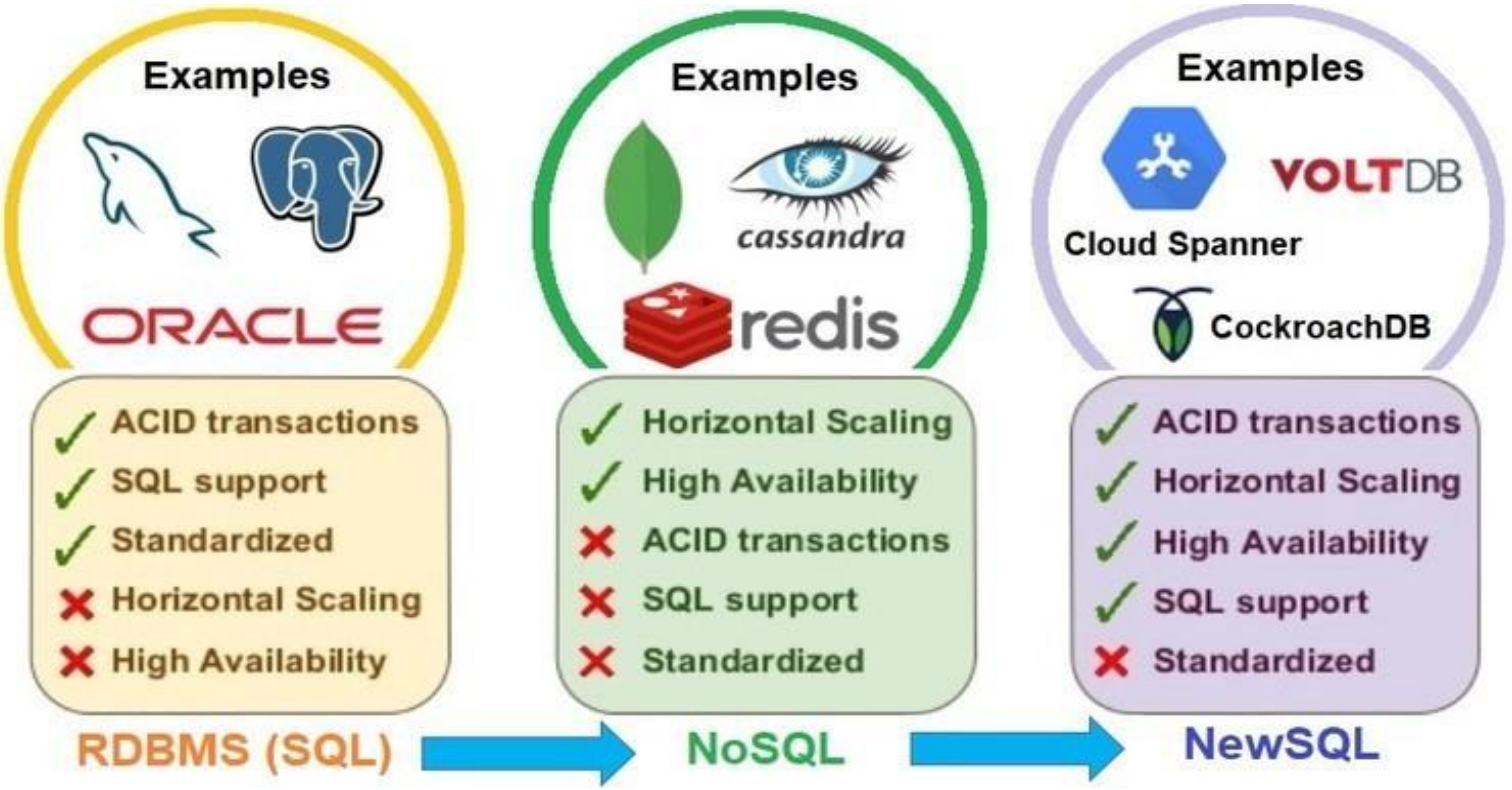
  Basically Available: the system guarantees availability

  Soft-State: state of the system may change over time

  Eventual Consistency: the system will ***eventually*** become consistent

- Eventually Consistent means all replicas will gradually become consistent

Examples



https://medium.com/rabiprasadpadhy/google-spanner-a-newsql-journey-or-beginning-of-the-end-of-the-nosql-era-3785be8e5c38

Advantages of NoSQL DBMS

- Schema-free – allow both semi-structured and unstructured data
- Highly-scalable
- Cloud and Big data support
- Relatively light database administration

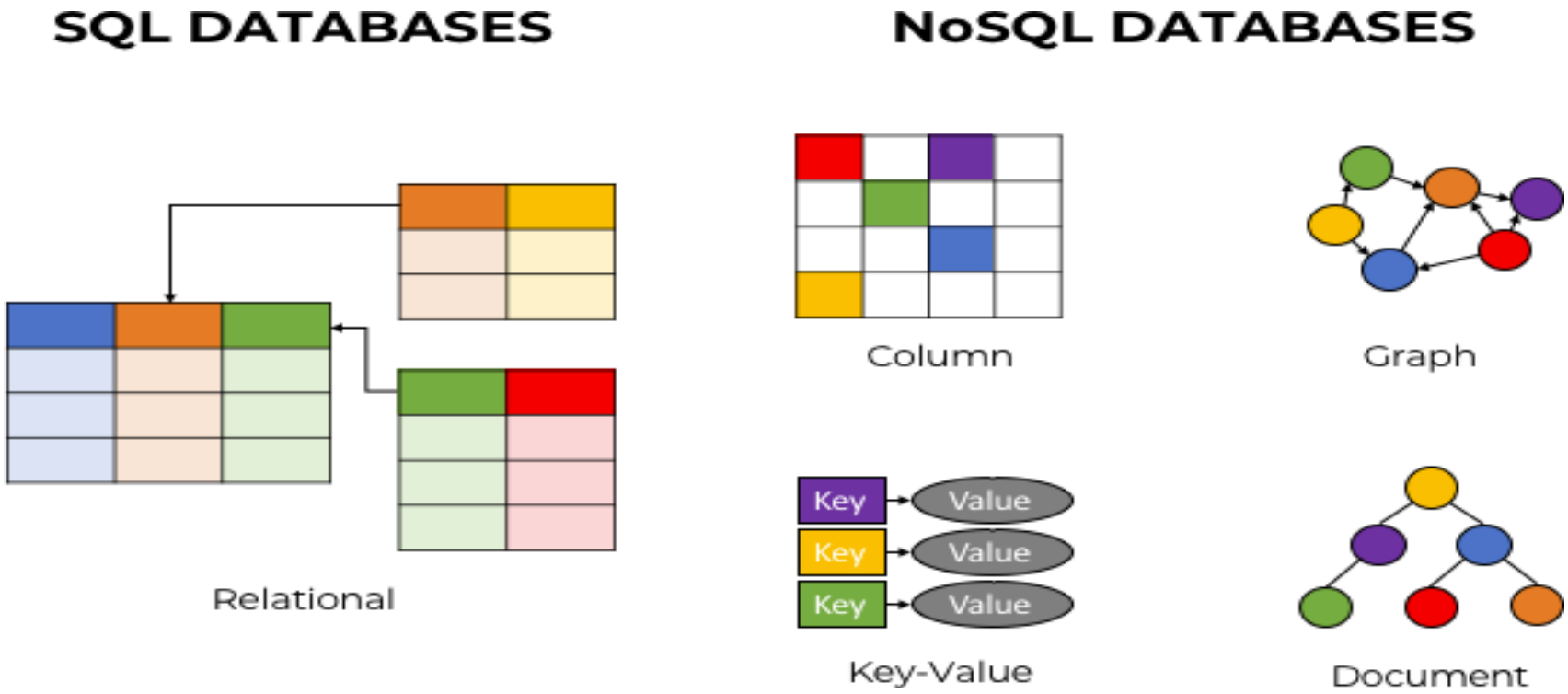
Disadvantages of NoSQL DBMS

- Less mature technology
- Less enterprise level support
- No SQL support

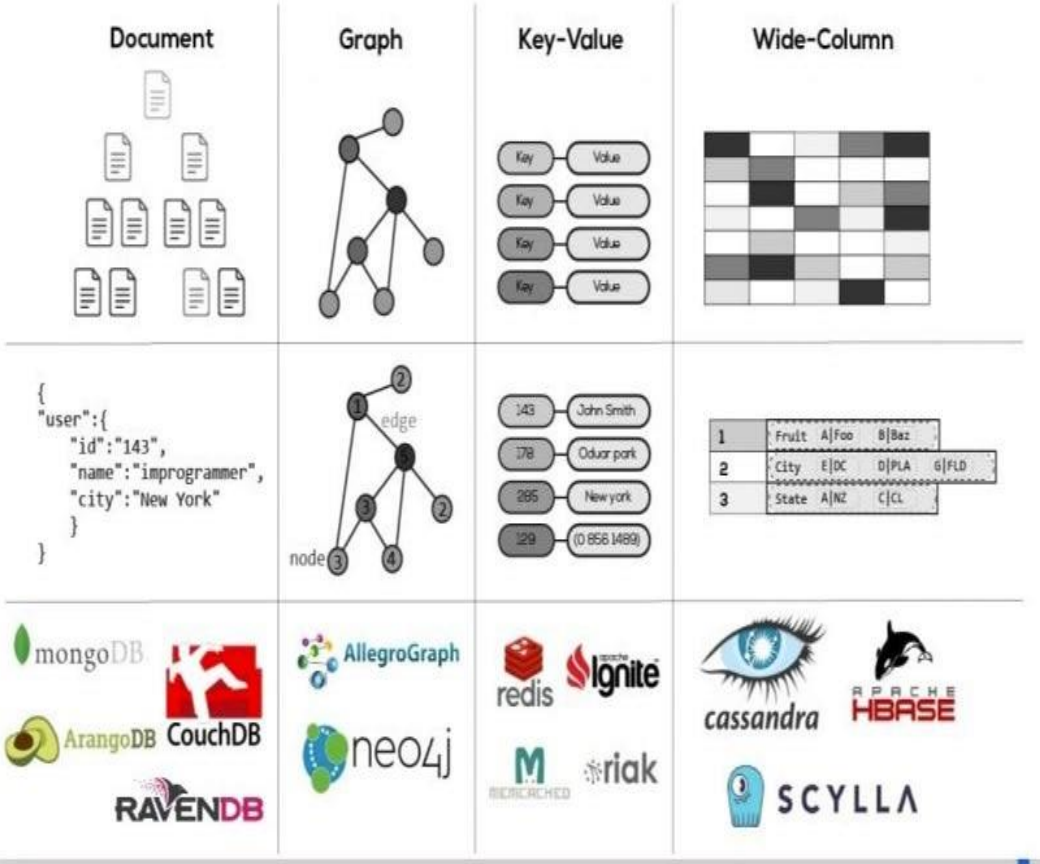
RDBMS vs. NoSQL Databases

	RDBMS	NoSQL
Variety	One type (relational)	Four main types: document, column-oriented, key-value, and graph
Structure	Predefined	Dynamic
Scaling	Primarily vertical	Primarily horizontal
Focus	Data integrity	Data performance and availability

Types of Non-Relational Data Models/Databases



NoSQL DATABASE TYPES



Key-Value Stores

- Simplest type of NoSQL database
- Every database element is stored as a key-value pair
- Support CRUD operations but no join or aggregate functions
- Example: Redis, Amazon DynamoDB, Riak

Wide-Column Stores

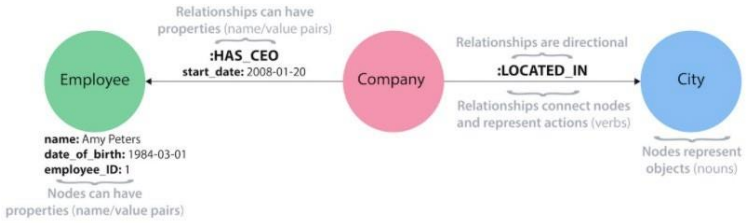
- Wide-column stores also referred to as column family stores
- A hybrid of RDBMS and key-value stores (aka multidimensional key-value stores):
  - Values are stored in groups/families of columns in column order (vs row order)
  - Queried by matching keys
  - Highly scalable to manage petabytes of data across massive, distributed systems
- Example: Hbase, Google BigTable, Cassandra

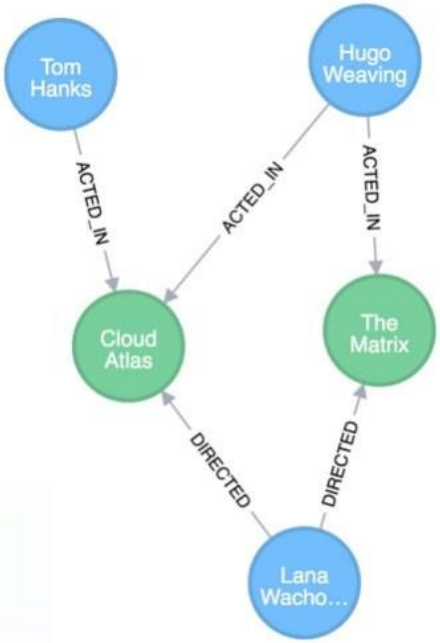
Document Stores

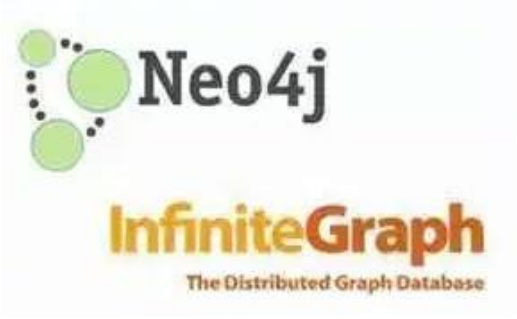
- Data (document) are stored in various standard format/encoding (e.g. json, xml, pdf etc) known as BLOBS (binary large objects) although mostly JSON is used.
- Each document contains different fields including strings, dates, values and arrays.
- Documents provide intuitive and natural way to model data (hierarchical) similar to objected-oriented programming. Documents are like objects.
- Data can be indexed to outperform traditional file systems.
- Example: MongoDB, CouchDB

Graph Stores

- Data are represented as graph structures with nodes and edges.
- Ideal for modelling complex relationships (social networking, network topologies)

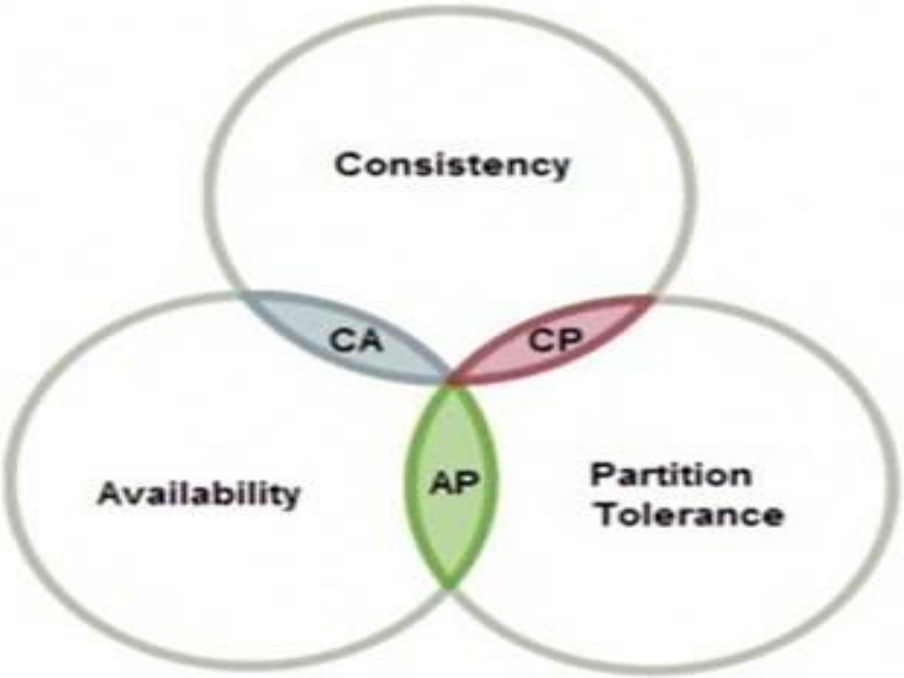






CAP Theorem

- CAP Theorem: Any distributed database with shared data can have at most 2 of the 3 properties: C, A or P



Traditional RDMS

- Consistency
  - Transaction isolation & repeatability
  - RDMS: e.g. banking, financials,…
- Availability
  - Clustered servers
  - 99.999% uptime
- NOT easily partitioned (scalable)

NoSQL

- Partitioning
  - Fast & global scalability
  - NoSQL: big data, mobile gaming, social media

- Availability
  - Clustered servers
  - 99.999% uptime

- Sacrifice consistency

NoSQL Database: Major Players

- Too many document NoSQL databases to name a few distinct ones

29 systems in ranking, July 2014

Rank	Last Month	DBMS	Database Model	Score	Changes
1.	1.	MongoDB	Document store	238.78	+7.33
2.	2.	CouchDB	Document store	23.07	+0.28
3.	3.	Couchbase	Document store	16.58	+0.79
4.	4.	MarkLogic	Multi-model	8.20	-0.02
5.	5.	RavenDB	Document store	5.09	-0.42
6.	6.	GemFire	Document store	2.16	-0.06
7.	7.	OrientDB	Multi-model	1.71	-0.02
8.	8.	Cloudant	Document store	1.70	+0.07
9.	9.	Datameer	Document store	0.88	+0.08
10.	10.	Mnesia	Document store	0.72	+0.01

Key Benefit of NoSQL: O(1) Lookup

- Fast lookup
  - No joining required
  - All data about one domain concept in one document
- Direct programming language representation
  - No mapping or ‘ORM’ layer required
- JSON library
  - Direct result representation and manipulation
  - JavaScript: representation in language data types directly
  - E.g., check out MongoDB node.js driver

MongoDB

https://en.wikipedia.org/wiki/MongoDB

- Name derived from Hu(**MONGO)**us word
- Document Oriented Database
- Built for High – Performance and scalability
- Document based queries for **Easy Readability**
- Replication and failover for **High Availability**
- Auto Sharding for **Easy Scalability**

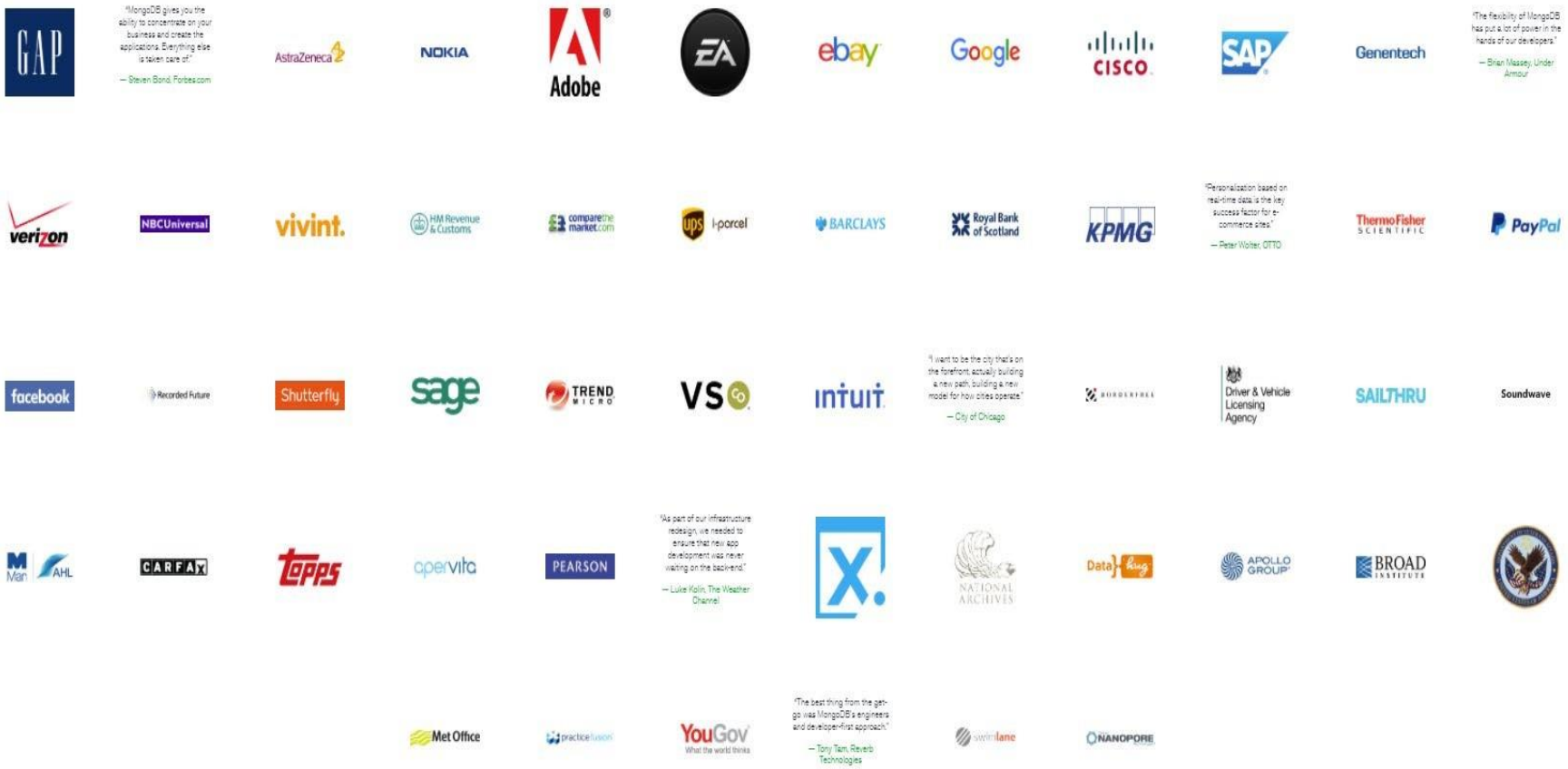
Where to use MongoDB ?

- Ideal for Web Applications
- Applications containing semi-structured data and need flexible schema management
- Caching and High Scalability
- Scenarios where **data availability** and **size of data** are priorities over the **transactions** of data

How does MongoDB Store data?

- Stores data in form of Documents
- JSON like field – value pair
- Documents analogous to structures in programming languages with key – value pair
- Documents stored in **BSON (Binary JSON)** format
- BSON is JSON with additional type information

Companies Using MongoDB



MongoDB: A Document Store

- A record in MongoDB is a document, which is a data structure composed of field and value pairs.
- MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents.

MongoDB Terminology

- Mysql
- Table
- Row
- Column
- Joins
- Group By

- MongoDB

MongoDB Terminology

- Mysql
- Table
- Row
- Column
- Joins
- Group By

- MongoDB
- Collection
- Document
- Field
- Embedded document
- Aggregation

Examples of MongoDB Data


```
{
   "_id": {
     "$oid": "56d61033a378eccde8a8357e"
   },
   "address": {
       "city": "LAWRENCE",
       "number": 1,
       "street": "BAY BLVD",
       "zip": 11559
   },
   "business_name": "SPRAGUE OPERATING RESOURCES LLC.",
   "certificate_number": 3019422,
   "date": "Mar  3 2015",
   "id": "11247-2015-ENFO",
   "result": "Fail",
   "sector": "Fuel Oil Dealer - 814"
}
```

```
{
 "address": {
   "building": "8825",
   "coord": [-73.8803827, 40.7643124],
   "street": "Astoria Boulevard",
   "zipcode": "11369"
 },
 "borough": "Queens",
 "cuisine": "American",
 "grades": [ {
   "date": {"$date": "2014-11-15T00:00:00.000Z"},
   "grade": "Z",
   "score": 38
 },
 {
   "date": {"$date": "2014-05-02T00:00:00.000Z"},
   "grade": "A",
   "score": 10
 },
 {
   "date": {"$date": "2013-03-02T00:00:00.000Z"},
   "grade": "A",
   "score": 7
 },
 {
   "date": {"$date": "2012-02-10T00:00:00.000Z"},
   "grade": "A",
   "score": 13
 }],
   "name": "Brunos On The Boulevard",
   "restaurant_id": "40356151"
}
```

SQL vs MongoDB

PERSON

Pers_ID	Surname	First_Name	City
0	Miller	Paul	London
1	Ortega	Alvaro	Valencia
2	Huber	Urs	Zurich
3	Blanc	Gaston	Paris
4	Bertolini	Fabrizio	Rome

NO RELATION

CAR

Car_ID	Model	Year	Value	Pers_ID
101	Bently	1973	100000	0
102	Rolls Royce	1965	330000	0
103	Peugeot	1993	500	3
104	Ferrari	2005	150000	4
105	Renault	1998	2000	3
106	Renault	2001	7000	3
107	Smart	1999	2000	2

relational database tables in a 1-many relationship

```
{
first_name: "Paul",
surname: "Miller",
city: "London",
location: [45.123,47.232],
cars: [
   { model: "Bentley",
    year: 1973,
    value: 100000, ….},
   { model: "Rolls Royce",
    year: 1965,
    value: 330000, ….},
]
}
```

a single document (collection)

Document in MongoDB

- Stored in Collections
- Has **_id** field – works like Primary keys in Relational databases
- Sample document containing name, age, status and groups

MongoDB Query Language (MQL)

- Field selection
- The find () function in MongoDB Query Language provides the easiest way to retrieve data from multiple documents within one of your collections.
- `mql>db.Employee.find()`
- `Sql>select * from Employee;`

MQL: Condition clause: Where clause

- Here, to return all the A-status employees
- `mql>`db.Employee.find ({ status : "A" })

- sql>`select * from Employee where status='A'`

MQL: Projection: Select specific fields

- The insertion of the tag { NAME : 1 } specifies that only the information from the NAME field should be returned. The results are sorted and presented in ascending order.
- mql>db.Employee.find ({status : "A"}, {NAME: 1});
- sql>select name from Employee where status=’A’;

MQL: Field exclusion

- To query for the opposite, inserting { NAME : 0 } retrieves a list of all fields except for the NAME field.
- mql>db.Employee.find ({status : "A"}, {NAME: 0});
- sql>select age, address, status from Employee;

MQL: Complex operation (Dot notation)

- When you work with more complex document structures such as documents containing arrays or embedded objects, you can use other methods for querying from them.
- mql>db.Employee.find({ "Address.City" : "Irving" }).pretty()
- sql>SELECT E.* from Employee E inner join Dept D on E.EMPID=D.EMPID Where D.City=’Irving’

MQL: Insert multiple documents

- To create an array of documents, define the variable by a name and assign the array of documents.
- mql>document = [{ "Name" : "Brian Lockwood","Age" : "45", status:"A"}, { "Name" : "Charles","Age" : "35", status:"A"}]
  >db.Employee.insert (document)
- sql>Insert into Employee values (‘Brian Lockwood’,45, ‘A’),(‘Charles’,35,’A’)

MQL: Insert nested document

- the address document is embedded in the document. MongoDB Query Language accommodates that, but SQL strictly follows procedural constructs and does not allow insertion of values to non-existing fields.
- mql>document = ({ "Name" : "Robert Jordan","Age" : "37", status:"A", Address: { Street:"Polaris Way", City : "Aliso Viejo",State:"California" }})
- >db.Employee.insert (document)
- sql>Not applicable

MQL: Get distinct status

- To return unique values only
- mql>db.Employee.distinct("status");
- sql>Select distinct status from Employee;

MQL: Sort

- This example sorts the results based on the Age key in ascending order. Sorting is in ascending order unless otherwise specified (-1 flag for descending order).
- mql>db.Employee.find().sort({ Age: 1 })
- >db.Employee.find().sort({ Age: -1 })
- sql>Select * from Employee order by Age ASC;
- Select * from Employee order by Age DESC;

MQL: Limit

- Use the limit () function in MongoDB Query Language to specify the maximum desired number of results
- mql>db.Employee.find().limit(5);
- >db.Employee.skip().limit(5);
- sql>select top(5) * from Employee;
- SELECT * FROM Employee OFFSET 5 ROWS;

MQL: Aggregate

- count () returns the number of documents in the specified collection.
- mql>db.Employee.find().count();
- sql>Select count(*) from Employee;

MQL: Group

- group() takes three parameters: key, initial, and reduce. The purpose of group () and SQL's GROUP BY is to return an array of grouped items.
- mql>db.Employee.aggregate([{"$group" : {_id:"$status", count:{$sum:1}}}])

- sql>Select status,count(*) from Employee Group by status

MQL: Comparison

- Use special parameters $gt, $lt, $gte and $lte to perform greater-than and less-than comparisons in queries.
- mql>db.Employee.find ({ Age: {$gt : "30"} });

- sql>Select * from Employee where Age>30

MQL: Multiple expressions in document

- Use $or to search for multiple expressions in a single query.
- mql>db.Employee.find({ $or : [{ "Name" : "Flynn" }, { "status" :"A"}]})

- sql>select * from Employee where name=’Flynn’ or Status=’A’

MQL: Comparison

- Use special parameters $gt, $lt, $gte and $lte to perform greater-than and less-than comparisons in queries.
- mql>db.Employee.find ({ Age: {$gt : "30"} });

- sql>Select * from Employee where Age>30

MQL

- Combine multiple operators
- Can include limits, skips and sorts

```
db.users.find(
    { age: { $gt: 18 } },
    { name: 1, address: 1 }
).limit(5)
```

← collection
← query criteria
← projection
← cursor modifier

MongoDB Example: Create Table

- No need to create tables (collections) in MongoDB ➔ just start inserting data

SQL

```
CREATE TABLE people (
    id MEDIUMINT NOT NULL
        AUTO_INCREMENT,
    user_id Varchar(30),
    age Number,
    status char(1),
    PRIMARY KEY (id)
)
```

MongoDB

```
db.people.insertOne( {
    user_id: "abc123",
    age: 55,
    status: "A"
 } )
```

However, you can also explicitly create a collection:

```
db.createCollection("people")
```

https://docs.mongodb.com/manual/reference

MongoDB Example: Update Data

- Use updateMany() to update records

SQL

```
UPDATE people
SET status = "C"
WHERE age > 25
```

```
UPDATE people
SET age = age + 3
WHERE status = "A"
```

MongoDB

```
db.people.updateMany(
   { age: { $gt: 25 } },
   { $set: { status: "C" } }
)
```

```
db.people.updateMany(
   { status: "A" } ,
   { $inc: { age: 3 } }
)
```

MongoDB Example: Delete Data

- Use deleteOne() or deleteMany() to delete records

SQL

```
DELETE FROM people
WHERE status = "D"
```

MongoDB

```
db.people.deleteMany( { status: "D" } )
```

```
DELETE FROM people
```

```
db.people.deleteMany({})
```

Remove Operation

- In MongoDB, **db.collection.remove()** method deletes document from the collection

```
db.users.remove(          <-- collection
   { status: "D" }        <-- remove criteria
)
```

MongoDB Example: Alter Table

- Collections do not describe or enforce the structure of its documents

SQL

```
ALTER TABLE people
ADD join_date DATETIME
```

MongoDB

```
db.people.updateMany(
    { },
    { $set: { join_date: new Date() } }
)
```

```
ALTER TABLE people
DROP COLUMN join_date
```

```
db.people.updateMany(
    { },
    { $unset: { "join_date": "" } }
)
```

MongoDB Example: Create Index & Drop Table

- MongoDB provides a createIndex() method to **create one or more indexes on collections**

```
CREATE INDEX idx_user_id_asc
ON people(user_id)
```

```
db.people.createIndex( { user_id: 1 } )
```

```
CREATE INDEX
        idx_user_id_asc_age_desc
ON people(user_id, age DESC)
```

```
db.people.createIndex( { user_id: 1, age: -1 } )
```

```
DROP TABLE people
```

```
db.people.drop()
```

What is Mongoose/**Mongothon**

- Mongoose is an Object Document Mapper (ODM) for Node.js that makes using MongoDB easier by translating documents in a MongoDB database to objects in the program.
- This means that Mongoose allows you to define objects with a strongly-typed schema that is mapped to a MongoDB document.
- **Mongothon** is a MongoDB object-document mapping API for Python, loosely based on the awesome mongoose.
- Doctrine (library that provides a PHP object mapping),
- MongoLink (object/document mapper for java)
- Mandango (The easy, powerful and ultrafast ODM for PHP).

Mongoose

Schema Types that a property is saved as when it is persisted to MongoDB.

- String
- Number
- Date
- Buffer
- Boolean
- Mixed
- ObjectId
- Array

Each data type allows you to specify:

- a default value
- a custom validation function
- indicate a field is required
- a get function that allows you to manipulate the data before it is returned as an object
- a set function that allows you to manipulate the data before it is saved to the database
- create indexes to allow data to be fetched faster