No SQL

Why NoSQL?

- Someone said SQL was bad, so they created NoSQL (Non SQL)
- Now, we understood they can work together (Not Only SQL)

- We started having a lot of data
- RDBMS don’t scale well in a cluster
- Simplicity of design
- Some NoSQL operation are faster
- ACID transactions (Atomicity, Consistency, Isolation, Durability), which are guaranteed in RDBMS, sometimes are not needed

NoSQL Use Cases

1. Massive data volumes
    - Massively distributed architecture required to store the data
    - Google, Amazon, Yahoo, Facebook (10-100K servers)

2. Extreme query workload
    - Impossible to efficiently do joins at that scale with an RDBMS

3. Schema evolution
    - Schema flexibility (migration) is not trivial at large scale
    - Schema changes can be gradually introduced with NoSQL

NoSQL Pros/Cons

- **Pros**
  - Massive scalability
  - High availability
  - Lower cost (than competitive solutions at that scale)
  - (usually) predictable elasticity
  - Schema flexibility, sparse & semi-structured data

- **Cons**
  - Limited query capabilities
  - Eventual consistency is not intuitive to program for
  - No standardization

Benefits of NoSQL (1)

Elastic Scaling

- **RDBMS scale up – bigger load , bigger server**
- NO SQL scale out – distribute data across multiple hosts seamlessly

DBA Specialists

- **RDMS require highly trained expert to monitor DB**
- NoSQL require less management, automatic repair and simpler data models

Big Data

- Huge increase in data

  RDMS: capacity and constraints of data volumes at its limits
- NoSQL designed for big data

Benefits of NoSQL (2)

Flexible data models

- **Change management to schema for RDMS have to be carefully managed**
- NoSQL databases more relaxed in structure of data
  - Database schema changes do not have to be managed as one complicated change unit
  - Application already written to address an amorphous schema

Economics

- **RDMS rely on expensive proprietary servers to manage data**
- No SQL: clusters of cheap commodity servers to manage the data and transaction volumes
- Cost per gigabyte or transaction/second for NoSQL can be lower than the cost for a RDBMS

Drawbacks of NoSQL (1)

Support

- RDBMS vendors provide a high level of support to clients
  - Stellar reputation
- **NoSQLs are open source projects with startups supporting them**
  - **Reputation not yet established**

Maturity

- RDMS mature product: means stable and dependable
  - Also means old no longer cutting edge nor interesting
- **NoSQL are still implementing their basic feature set**

Drawbacks of NoSQL (2)

Administration

- RDMS administrator well defined role
- **No SQL's goal:**
  - **No administrator necessary however**
  - **NO SQL still requires effort to maintain**

Lack of Expertise

- Whole workforce of trained and seasoned RDMS developers
- **Still recruiting developers to the NoSQL camp**

Analytics and Business Intelligence

- RDMS designed to address this niche
- **NoSQL designed to meet the needs of an Web 2.0 application - not designed for ad hoc query of the data**
  - **Tools are being developed to address this need**

No SQL

- Key-value

- Graph database

- Document-oriented

- Column family

CAP Theorem

GIVEN:

- **Many nodes**
- **Nodes contain replicas of partitions of the data**

Consistency

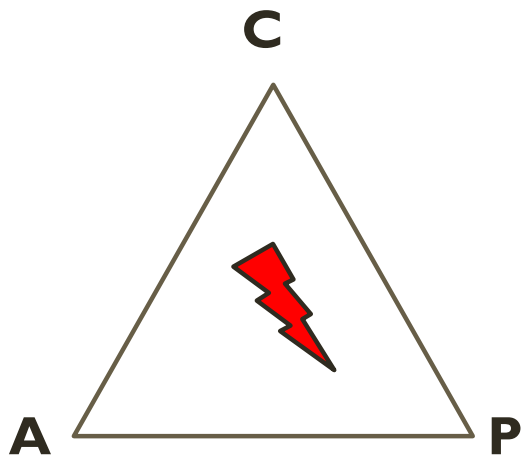
- All replicas contain the same version of data
- Client always has the same view of the data
  (no matter what node)

Availability

- System remains operational on failing nodes
- All clients can always read and write

Partition tolerance

- Multiple entry points
- System remains operational on system split (communication malfunction)
- System works well across physical network partitions



CAP Theorem:
satisfying all three at the same time is impossible

Consistency:

When I ask the same question to any part of the system I should get the same answer.

Consistency:

When I ask the same question to any part of the system I should get the same answer.

Consistency:

When I ask the same question to any part of the system I should get the same answer.

Availability:

When I ask a question I will get an answer.

Availability:

When I ask a question I will get an answer.

Partition Tolerance:

I can ask questions even if the system is having intra-system communication problems

Partition Tolerance:

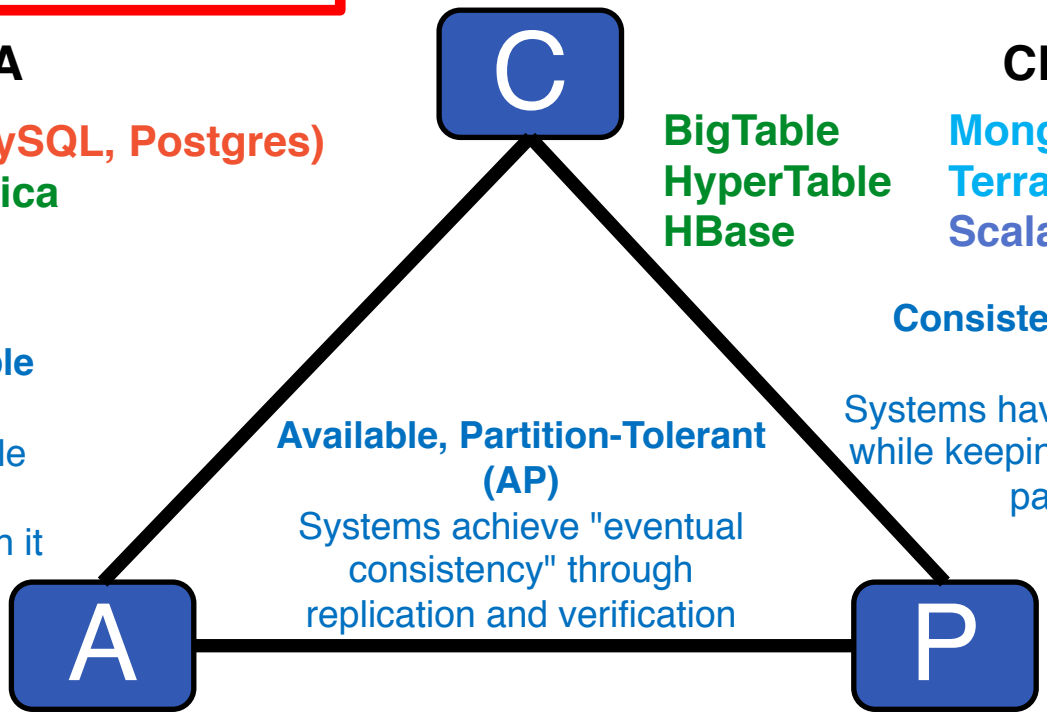
I can ask questions even if the system is having intra-system communication problems

Visualization of CAP theorem in NoSQL environment

All the three characteristics simultaneously are not possible!
Pick two!

Consistency
All clients always have the same view of the data

Relational (comparison)
Key-value
Column-Oriented/Tabular
Document-Oriented



CA

RDBMSs (e.g., MySQL, Postgres)
Vertica

CP

BigTable
HyperTable
HBase

MongoDB
Terrastore
Scalaris

Berkeley DB
MemcacheDB
Redis

Consistent, Available (CA)
Systems have trouble with partitions and typically deal with it with replication

Available, Partition-Tolerant (AP)
Systems achieve "eventual consistency" through replication and verification

Consistent, Partition-Tolerant (CP)
Systems have trouble with availability while keeping data consistent across partitioned nodes

Availability
Each client can always read and write

AP

Dynamo
Voldemort
Tokyo

Cabinet
KAI
Cassandra

SimpleDB
CouchDB
Riak

Partition Tolerance
The system works well despite physical network partitions

dbTrento

CAP theorem for NoSQL

What the CAP theorem really says:

- **If you cannot limit the number of faults, and if requests can be directed to any server, and if you insist on serving every request you receive, then you cannot possibly be consistent**

What we can get:

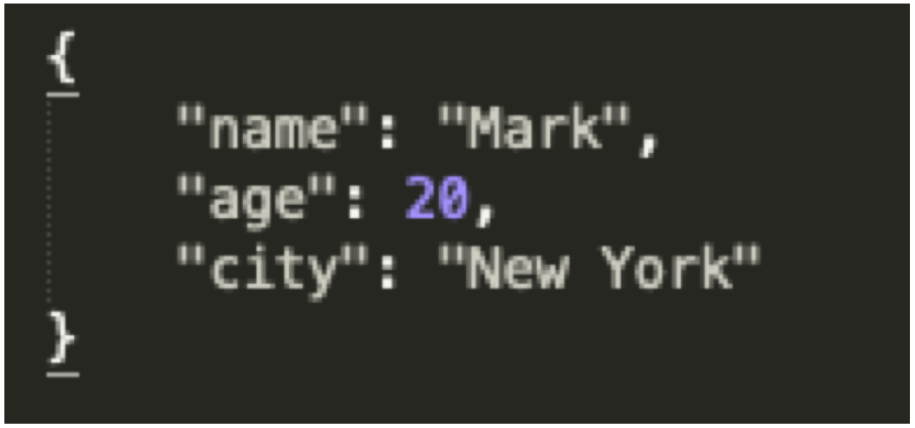
- **You must always give something up: consistency, availability or tolerance to failure and reconfiguration**

NoSQL vs. RDBMS

- **Looser schema definition**
- **Applications written to deal with specific documents/data**
  - Applications aware of the schema definition as opposed to the data
- **Designed to handle distributed, large databases**
- **Trade offs:**
  - No strong support for ad hoc queries but designed for speed and growth of database
    - Query language through the API
  - Relaxation of the ACID properties

Document-Oriented Databases

- Documents are the main concept
- Such a db stores and retrives docs in some standard format:
  - JSON
  - XML
  - ...
- A document is similar to a row (or a record) in relational DB, but more flexible (documents may have different attributes)
- Document are indexed
- Sequence of *key: value*



Who is using Document-Oriented DB?

Big Data	Product & Asset Catalogs	Security & Fraud	Internet of Things	Database-as-a-Service
BROAD INSTITUTE github salesforce ENERNOC	CARFAX UNDER ARMOUR orange edmunds.com	stripe McAfee Top Investment and Retail Banks Intelligence Agencies	Top Global Shipping Company BOSCH Telefónica Top Industrial Equipment Manufacturer	OPENSHIFT CLOUD FOUNDRY Top Media Company Top Investment and Retail Banks

Mobile Apps	Customer Data Management	Single View	Social & Collaboration	Content Management
O2 ADP foursquare	theguardian GILT intuit eHarmony	MetLife CERN stripe Telefónica	cisco Eventbrite foursquare eHarmony	The New York Times MTV Forbes shutterfly ebay

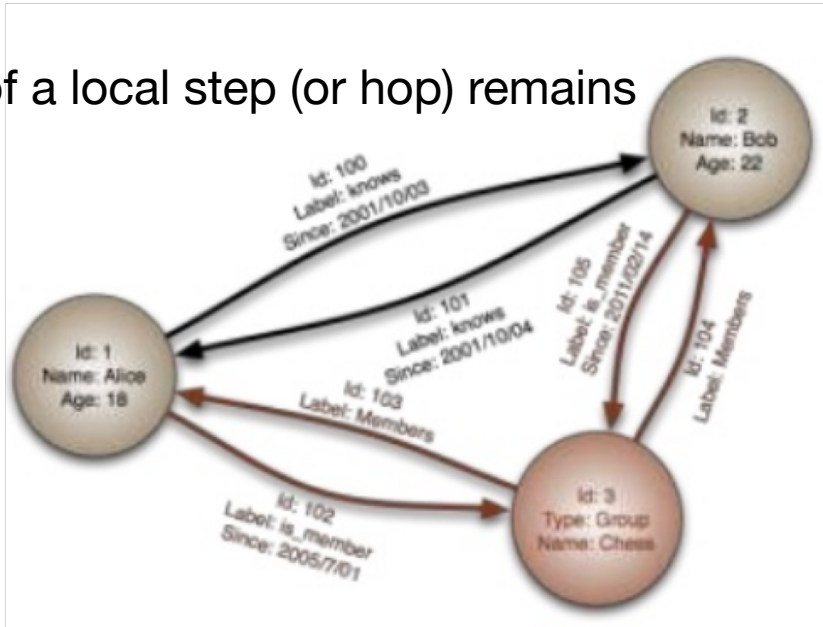
RDBMS vs MongoDB

RDBMS	MongoDB
Table	Collection
Row	JSON Document
Index	Index
Join	Embedding & Linking
Partition	Shard

Keys	Document
Id 1	Document1 with key-value collection
Id 2	Document2 with key-value collection
Id 3	Document3 with key-value collection

Graph Databases

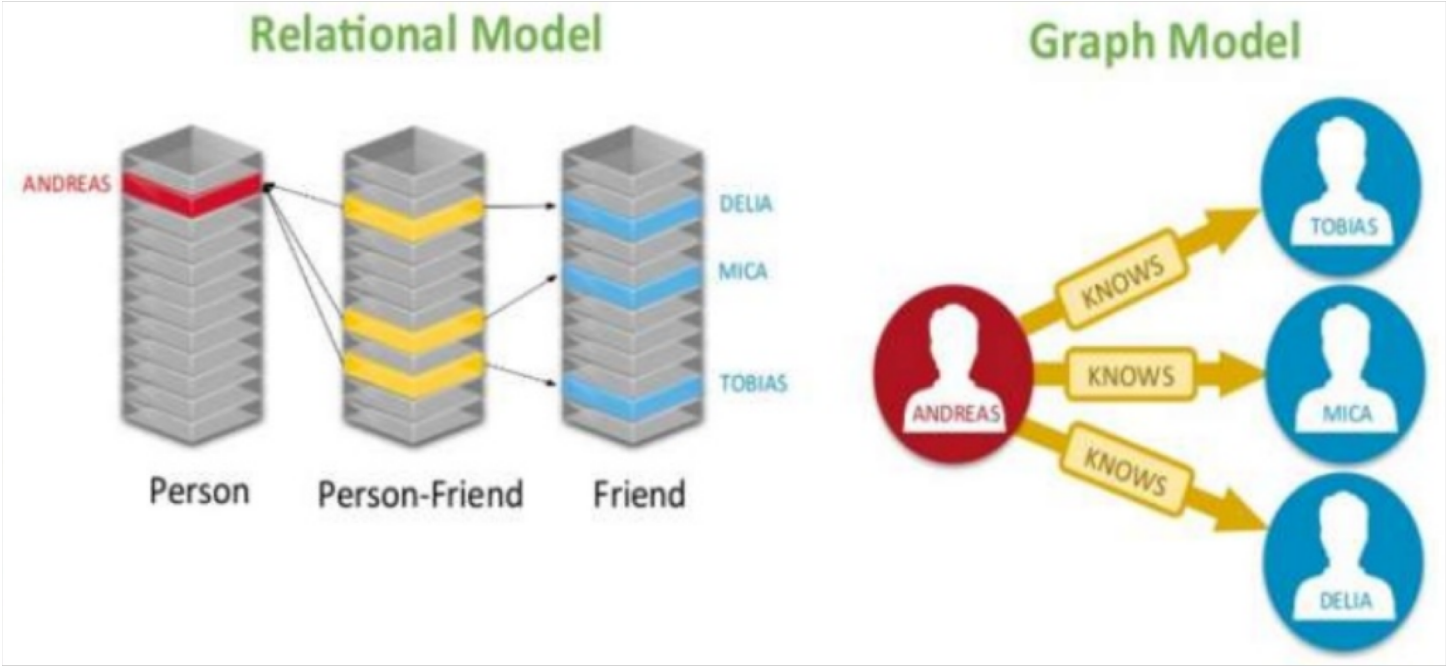
- A graph is a set of nodes and the relationships that connect those nodes
- Nodes and Relationships (may) contain properties to represent data
- Properties are key-value pairs to represent data
- A graph DB stores data in a graph
- Each node knows its adjacent nodes
- As the number of nodes increases, the cost of a local step (or hop) remains the same



Who is using Graph DB?

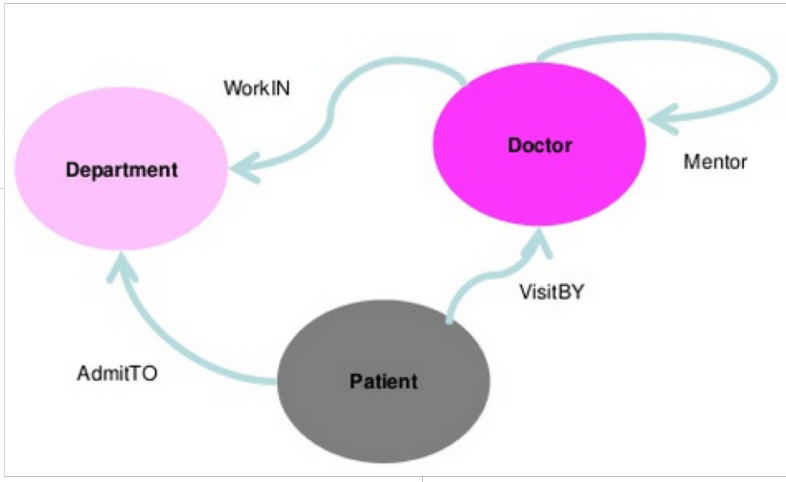
neo4j

RDBMS vs Graph Database



RDBMS	Graph DB
Table	Graph
Row	Node
Columns and Data	Properties and its values
Constraints	Relationships
Joins	Traversal

Converting relational model into Graph model



Mentor

VisitBY

WorkIN

AdmitTO

Mentor

Visit_Doctor
Doctor_ID
Patient_ID
DATE

Patient
ID
SSN
Name
Address
Phone number
Date of birth
Marital status
Gender
Race
Smoker
Drug abuse
Driver's license number
Driver's license State
Zip_code

Doctor
ID
SSN
Name
Yearly salary
Job description
Department Name
Date of birth
Phone number
Address
Marital status
Date of hiring
Specialist
Zip_code

Department
Name
Location
Head
Phone Number
Main Sectary's phone
Status
Number of Patient rooms
Size
Yearly Budget
URL

Admit
Patient_ID
Doctor_ID
DATE

Work
DOCTOR_ID
DEP_ID

#1 Drop Foreign Keys → #2 Join Tables become relationships

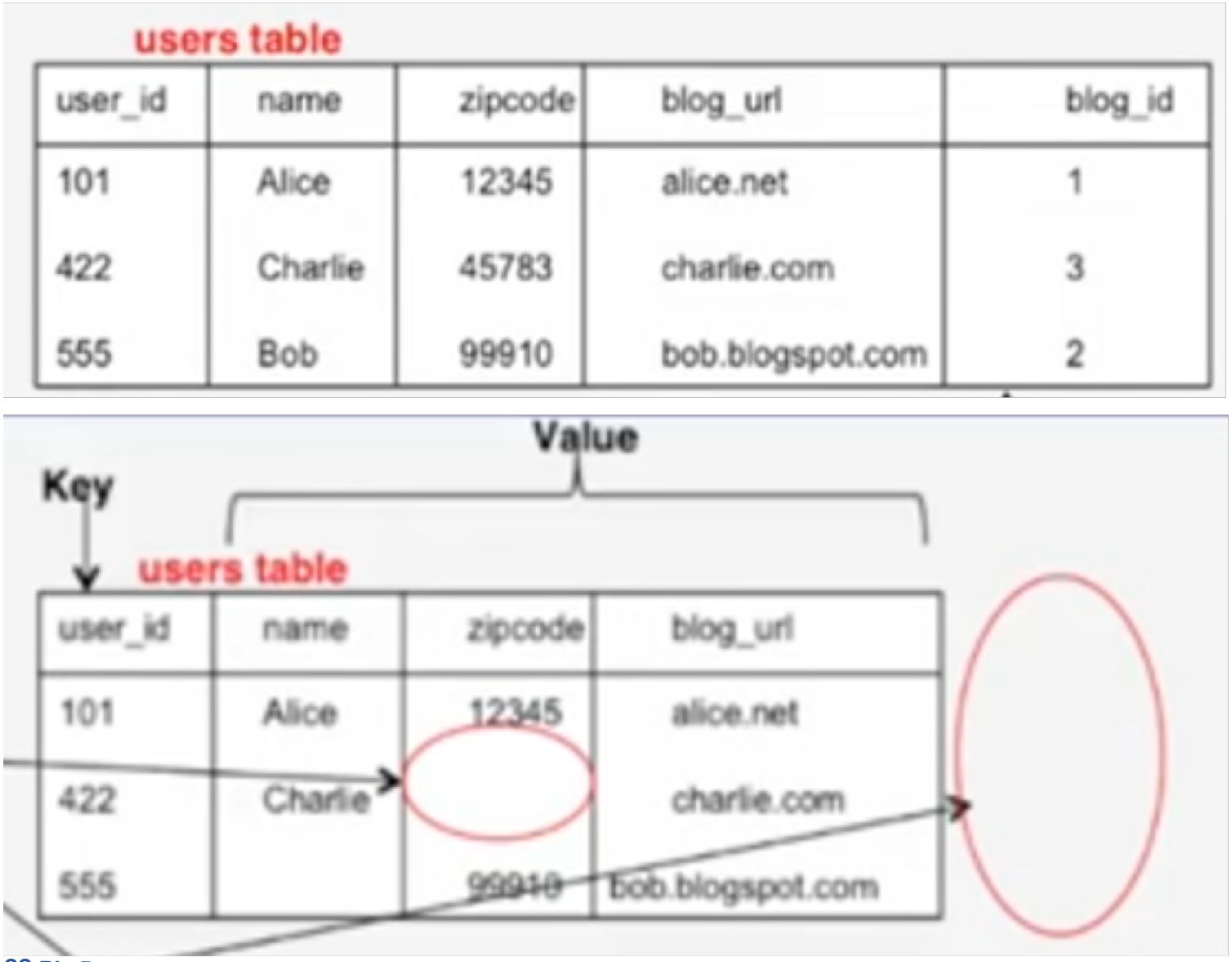
#3 Attributes → Properties

dbTrento

Key-Value Store

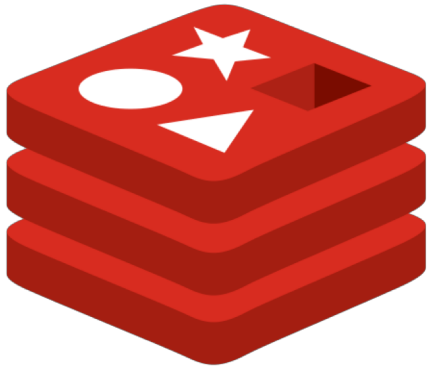
- Data model: (key, value) pairs
- Basic Operations: insert(key, value), fetch(key), update(key, new_value), delete(key)
- Values are stored as "blob", without caring or knowing what is inside
- The application layer has to understand the data
- Stored with an hashing function
- However, many structures cannot be easily modeled as key-value pairs
- Example: in Amazon we may have a key-value store where the key is the item identifier and the value is the information about it (only the text or a multiple data)

Key-Value Store



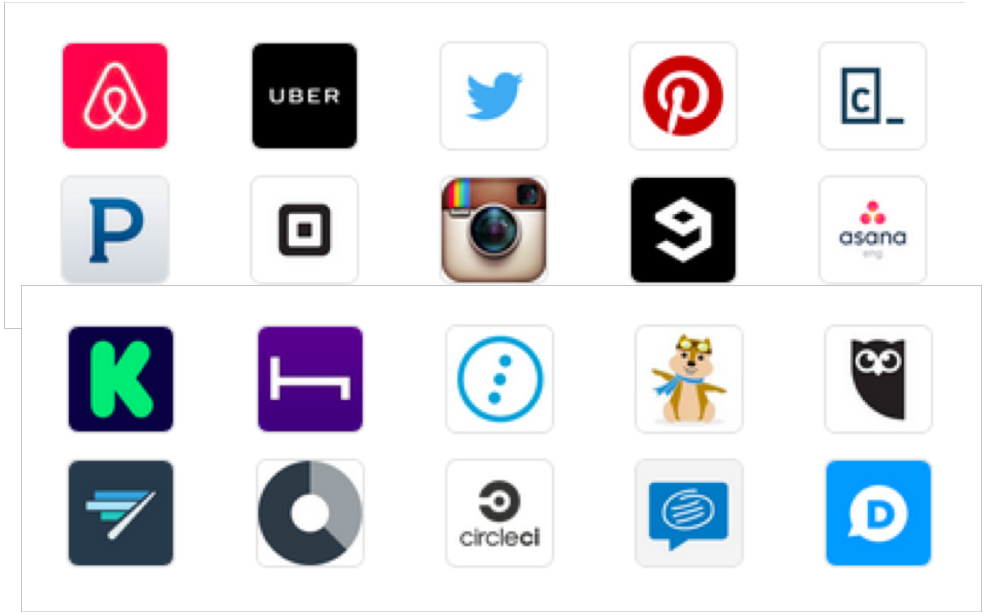
dbTrento

Who is using Key-Value Store?



redis

Made in Sicily!



dbTrento

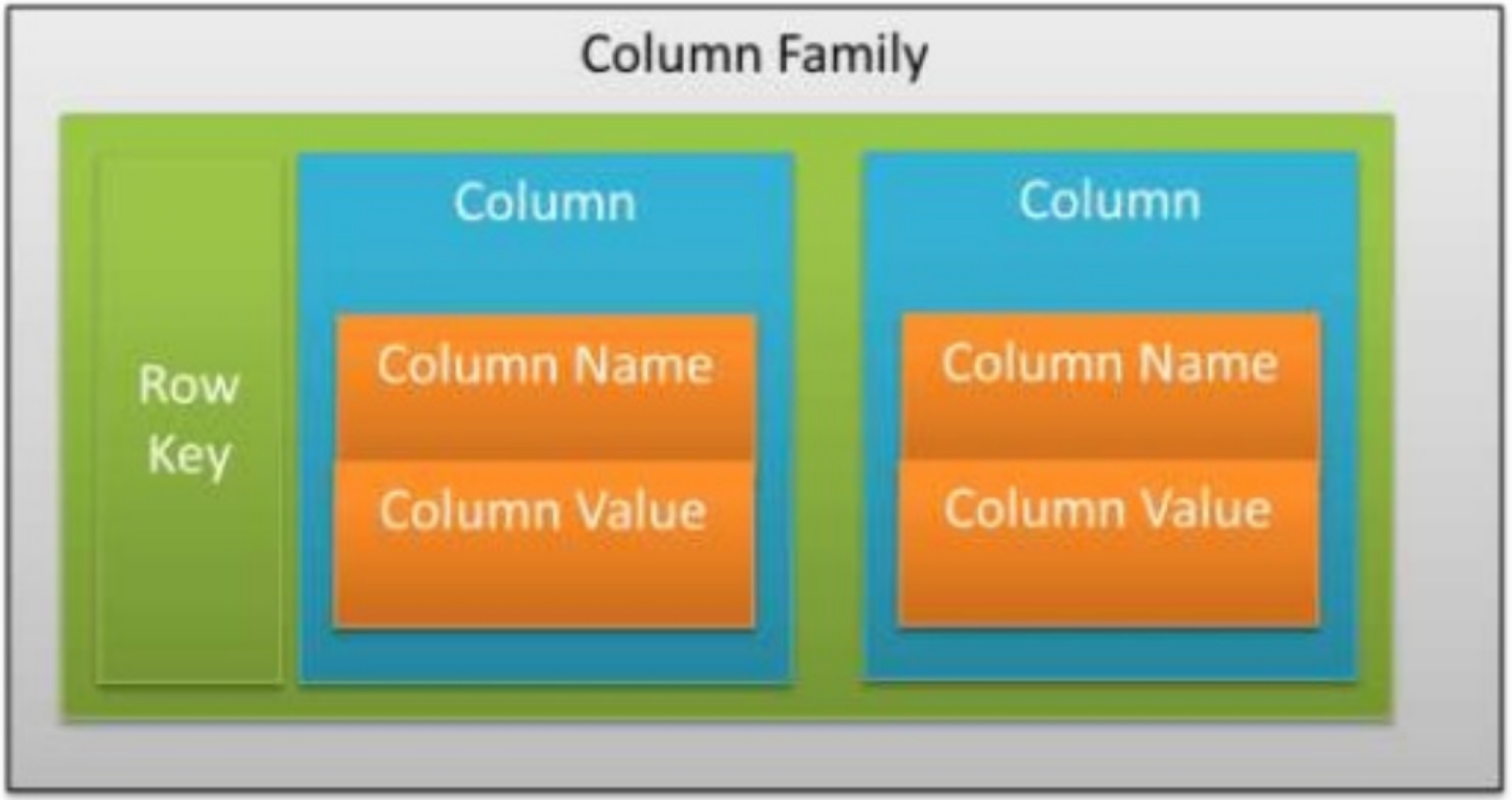
Column Family Store

- Stores data in columnar format
- Each storage block contains data from only one column
- Allow key-value pairs to be stored in a column
- Column family: collection of columns with key-value pairs
- Super column: collection of column families

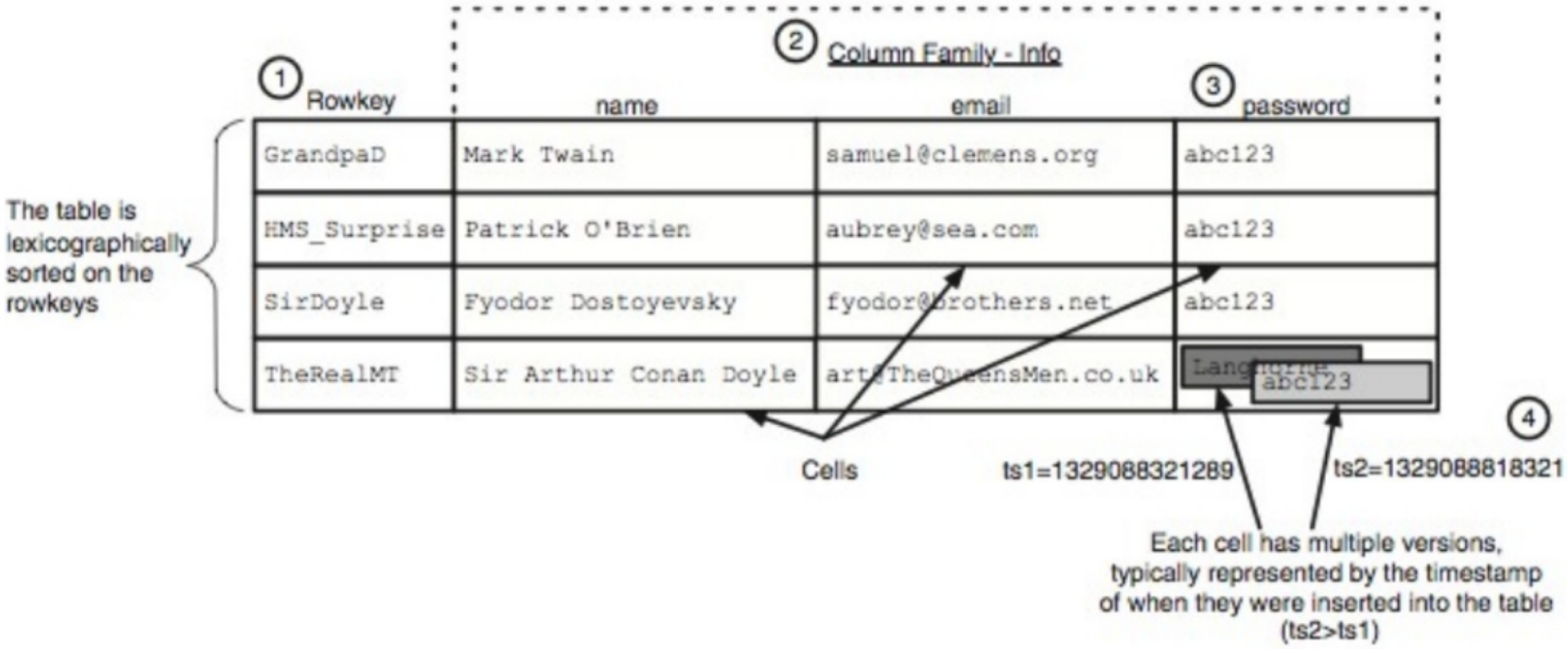
More efficient than row (or document) store (i.e., RDBMS) if:

- Multiple row/record/docs are inserted at the same time, so updates of column blocks can be aggregated
- Retrievals access only some of the columns in a row/record/doc

Column Family Store

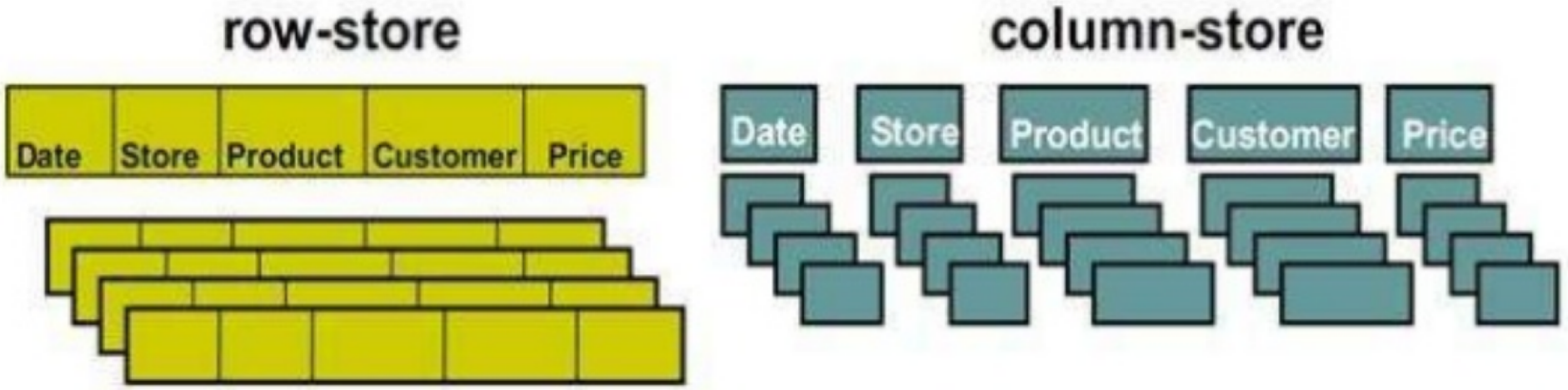


Column Family Store

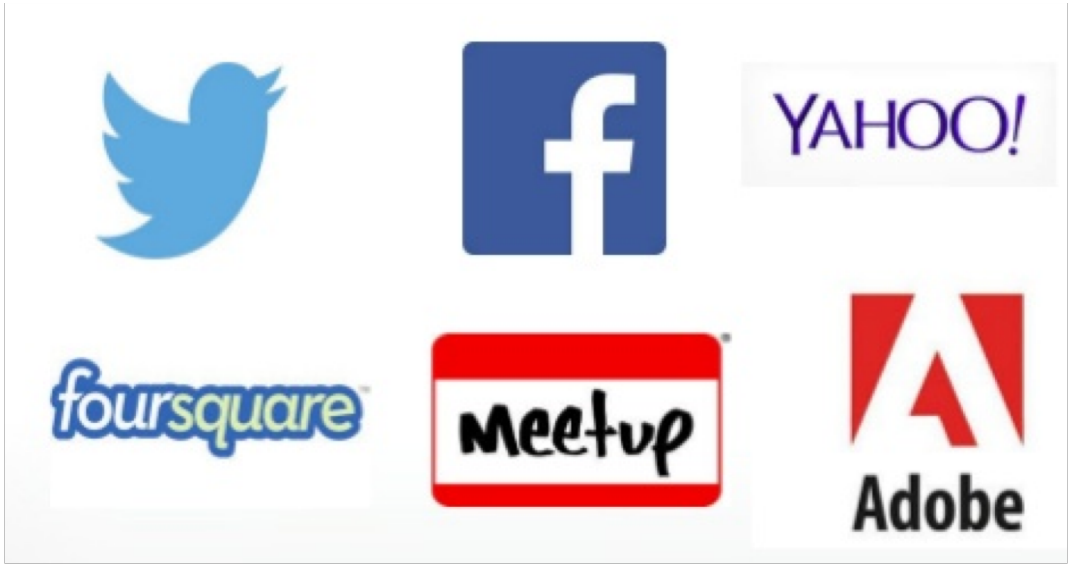


dbTrento

Column Store



Column Store



What is MongoDB?

- **Developed by 10gen**
  - Founded in 2007
- **A document-oriented, NoSQL database**
  - Hash-based, schema-less database
  - No Data Definition Language
  - In practice, this means you can store hashes with any keys and values that you choose
    - Keys are a basic data type but in reality stored as strings
    - Document Identifiers (_id) will be created for each document, field name reserved by system
  - Application tracks the schema and mapping
  - Uses BSON format
    - Based on JSON – B stands for Binary
- **Written in C++**
- **Supports APIs (drivers) in many computer languages**
  - JavaScript, Python, Ruby, Perl, Java, Scala, C#, C++,Haskell, Erlang

Functionality of MongoDB

- **Dynamic schema**
- **Document-based database**
- **Secondary indexes**
- **Query language via an API**
- **Atomic writes and fully-consistent reads**
  - If system configured in that way
- **Master-slave replication with automated failover (replica sets)**
- **Built-in horizontal scaling via automated range-based**
- **Partitioning of data (sharding)**
- **No joins nor transactions**

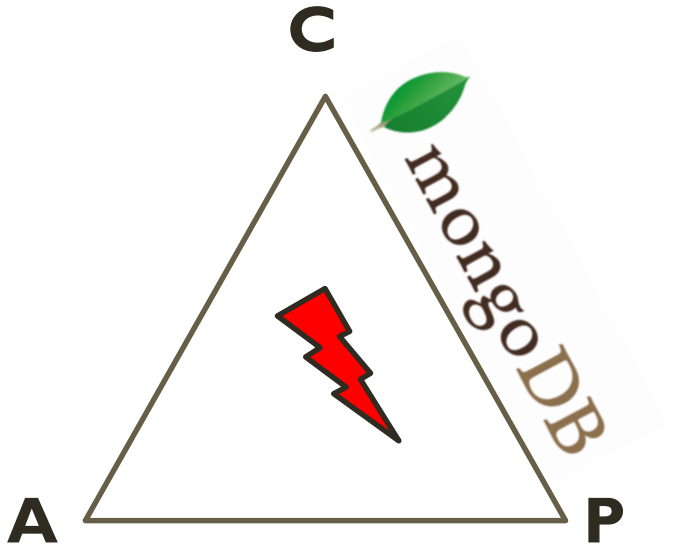
Why use MongoDB ?

- Simple queries
- Functionality provided applicable to most web applications
- Easy and fast integration of data
- Not well suited for heavy and complex transactions system

MongoDB vs. the CAP approach

Focus on Consistency and Partition tolerance

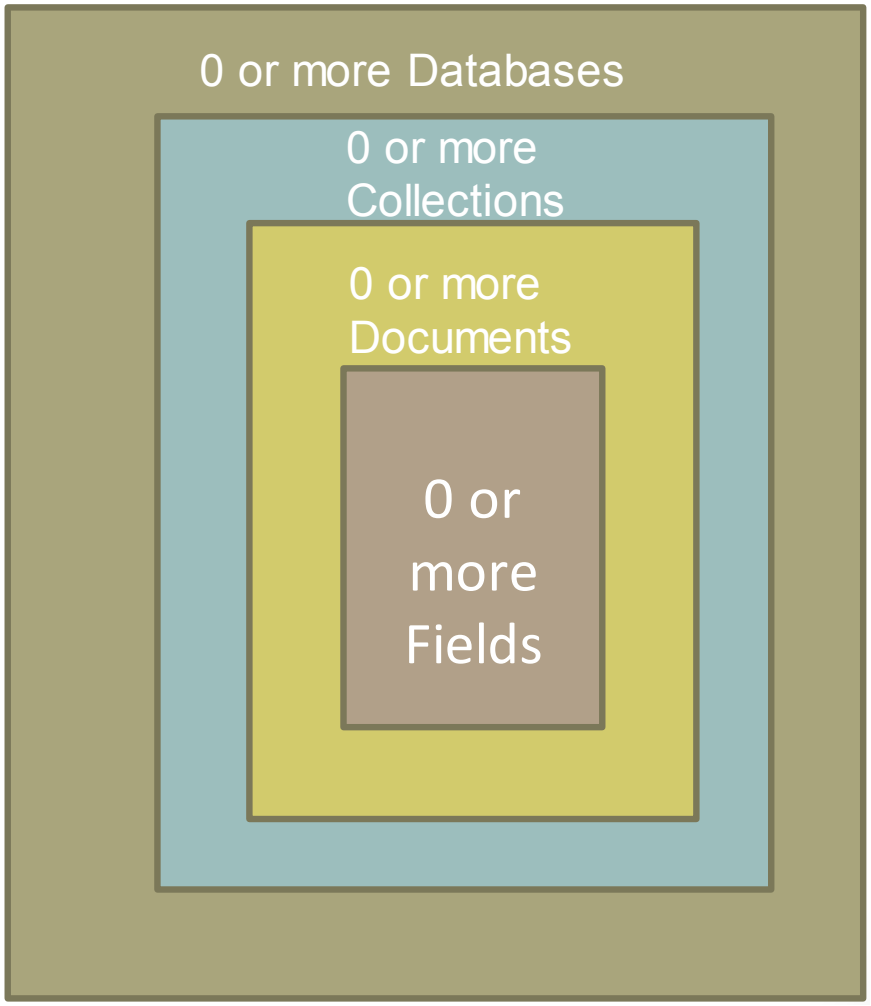
- **Consistency**
  - all replicas contain the same version of the data
- **Availability**
  - system remains operational on failing nodes
- **Partition tolerance**
  - multiple entry points
  - system remains operational on system split



CAP Theorem:
satisfying all three at the same time is impossible

MongoDB: Hierarchical Objects

- A MongoDB instance may have zero or more ‘databases’
- A database may have zero or more ‘collections’.
- A collection may have zero or more ‘documents’.
- A document may have zero or more ‘fields’.
- MongoDB ‘Indexes’ function much like their RDBMS counterparts.



RDBMS vs MongoDB

Parallelisms

RDBMS	MongoDB
Database	Database
Table, View	Collection
Row	Document (JSON, BSON)
Column	Field
Index	Index
Join	Embedded Document
Foreign Key	Reference
Partition	Shard

Collection is not strict about what it STORES

Schema less

Hierarchy is very clear in the design

Embedded documents

MongoDB Processes and configuration

- **mongod – Database instance**
- **mongos - Sharding processes**
  - Analogous to a database router.
  - Processes all requests
  - Decides how many and which mongods should receive the query
  - Mongos collates the results, and sends it back to the client.
- **mongo – an interactive shell (a client)**
  - Fully functional JavaScript environment for use with a MongoDB
- **You can have one mongos for the whole system no matter how many mongods you have**
- **OR**
- **you can have one local mongos for every client if you wanted to minimize network latency.**

Choices made for Design of MongoDB

- **Scale horizontally over commodity hardware**
  - Lots of relatively inexpensive servers

- **Keep the functionality that works well in RDBMSs**
  - Ad hoc queries
  - Fully featured indexes
  - Secondary indexes

- **What doesn't distribute well in RDB?**
  - Long running multi-row transactions
  - Joins
  - Both artifacts of the relational data model (row x column)

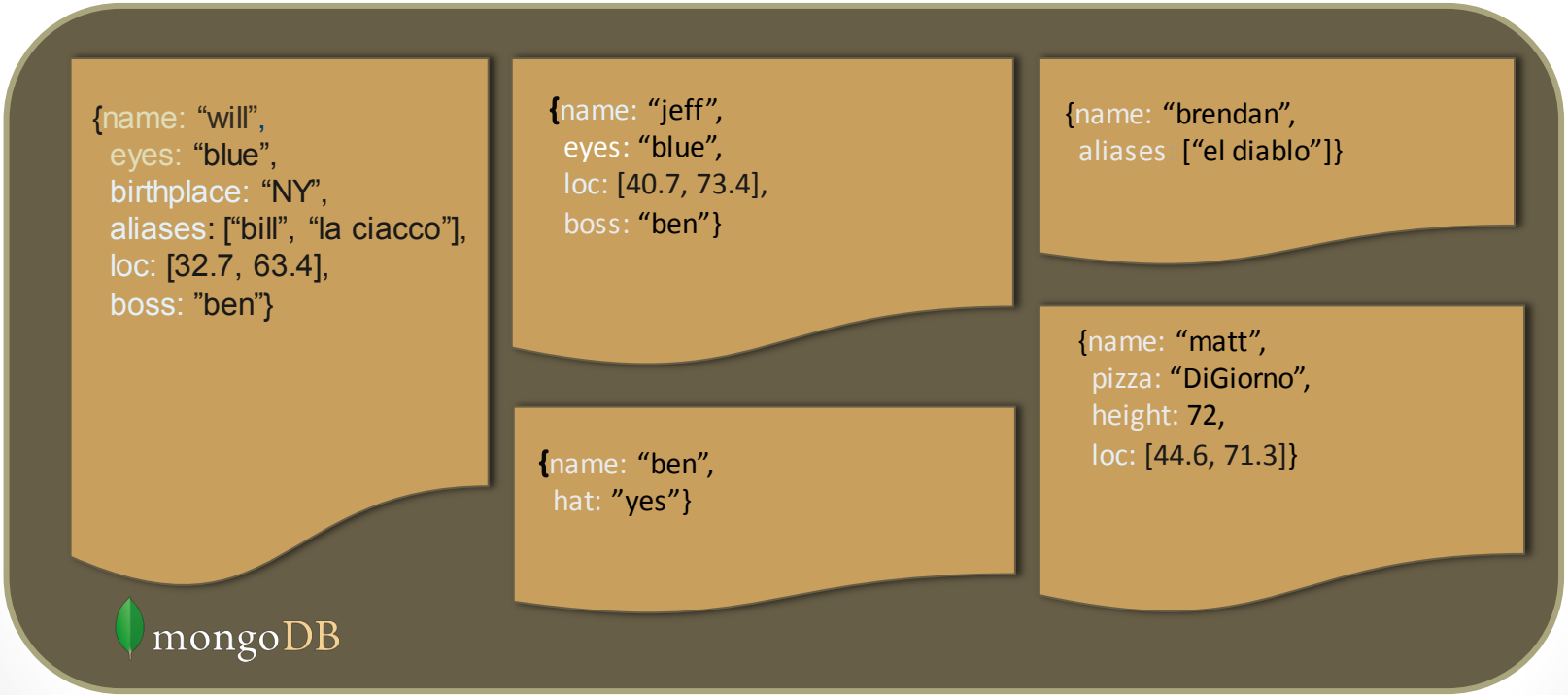
BSON format

- Binary-enc documents
- Zero or m s a single entity
- Each entry type, and a value
- Large eler refixed with a leng

```
{
"_id" : ObjectId("51…"),
"first" : "John",
"last" : "Doe",
"age" : 39
}
```

Schema Free

- MongoDB does not need any pre-defined data schema
- Every document in a collection could have different data
  - Addresses NULL data fields



MongoDB Features

- Document-Oriented storage
- Full Index Support
- Replication & High Availability
- Auto-Sharding
- Querying
- Fast In-Place Updates
- Map/Reduce functionality

AGILE

SCALABLE

Index Functionality

- **B+ tree indexes**
- **An index is automatically created on the _id field (the primary key)**
- **Users can create other indexes to improve query performance or to enforce Unique values for a particular field**
- **Supports single field index as well as Compound index**
  - Like SQL order of the fields in a compound index matters
  - If you index a field that holds an array value, MongoDB creates separate index entries for every element of the array
- **Sparse property of an index ensures that the index only contain entries for documents that have the indexed field. (so ignore records that do not have the field defined)**
- **If an index is both unique and sparse – then the system will reject records that have a duplicate key value but allow records that do not have the indexed field defined**

CRUD operations

- **Create**
  - db.collection.insert(<document>)
  - db.collection.save(<document>)
  - db.collection.update(<query>, <update>, { upsert: true })
- **Read**
  - db.collection.find(<query>, <projection>)
  - db.collection.findOne(<query>, <projection>)
- **Update**
  - db.collection.update(<query>, <update>, <options>)
- **Delete**
  - db.collection.remove(<query>, <justOne>)

collection specifies the actual collection or the ‘table’ to store the document

Create Operations

- **db.collection_name.insert(<document>)**
  - Omit the _id field to have MongoDB generate a unique key
  - Example db.parts.insert({{type: “screwdriver”, quantity: 15 })
  - db.parts.insert({_id: 10, type: “hammer”, quantity: 1 })
- **db.collection_name.update(<query>, <update>, { upsert: true })**
  - Will update 1 or more records in a collection satisfying query
- **db.collection_name.save(<document>)**
  - Updates an existing record or creates a new record

Read Operations

- **db.collection.find(<query>, <projection>).cursor modified**

  - Provides functionality similar to the SELECT command
    - <query> where condition , <projection> fields in result set
  - Example:varPartsCursor= db.parts.find({parts: “hammer”}).limit(5)
  - Has cursors to handle a result set
  - Can modify the query to impose limits, skips, and sort orders.
  - Can specify to return the ‘top’ number of records from the result set

- **db.collection.findOne(<query>, <projection>)**

Query Operators

Name	Description
$eq	Matches value that are equal to a specified value
$gt, $gte	Matches values that are greater than (or equal to a specified value
$lt, $lte	Matches values less than or (equal to) a specified value
$ne	Matches values that are not equal to a specified value
$in	Matches any of the values specified in an array
$nin	Matches none of the values specified in an array
$or	Joins query clauses with a logical OR returns all
$and	Join query clauses with a loginal AND
$not	Inverts the effect of a query expression
$nor	Join query clauses with a logical NOR
$exists	Matches documents that have a specified field

Update Operations

- **db.collection_name.insert(<document>)**
  - Omit the _id field to have MongoDB generate a unique key
  - Example db.parts.insert({{type: “screwdriver”, quantity: 15 })
  - db.parts.insert({_id: 10, type: “hammer”, quantity: 1 })
- **db.collection_name.save(<document>)**
  - Updates an existing record or creates a new record
- **db.collection_name.update(<query>, <update>, { upsert: true }**
  - Will update 1 or more records in a collection satisfying query
- **db.collection_name.findAndModify(<query>, <sort>, <update>,<new>, <fields>,<upsert>)**
  - Modify existing record(s) – retrieve old or new version of the record

Delete Operations

- **db.collection_name.remove(<query>, <justone>)**
  - Delete all records from a collection or matching a criterion
  - <justone> - specifies to delete only 1 record matching the criterion
  - Example: db.parts.remove(type: /^h/ }) - remove all parts starting with h
  - Db.parts.remove() – delete all documents in the parts collections

CRUD examples

```
> db.user.insert({
      first: "John",
      last : "Doe",
      age: 39
})
```

```
> db.user.find ()
{ "_id" : ObjectId("51"),
      "first" : "John",
      "last" : "Doe",
      "age" : 39
}
```

```
> db.user.update(
      {"_id" : ObjectId("51")},
      {
          $set: {
              age: 40,
              salary: 7000}
      }
)
```

```
> db.user.remove({
      "first": /^J/
})
```

SQL vs. MongoDB entities

MySQL

```
START TRANSACTION;
INSERT INTO contacts VALUES
    (NULL, 'joeblow');
INSERT INTO contact_emails
VALUES
   ( NULL, "joe@blow.com",
      LAST_INSERT_ID() ),
   ( NULL,
"joseph@blow.com",
      LAST_INSERT_ID() );
COMMIT;
```

MongoDB

```
db.contacts.save( {
    userName: "joeblow",
    emailAddresses: [
      "joe@blow.com",
      "joseph@blow.com" ] }
);
```

Aggregated funcionalities

Aggregation framework provides SQL-like aggregation functionality

- Pipeline documents from a collection pass through an aggregation pipeline, which transforms these objects as they pass through
- Expressions produce output documents based on calculations performed on input documents
- Example db.**parts**.aggregate ({$group : {_id: type, totalquantity : {$sum:quanity}} })

Map Reduce functionalities

- **Performs complex aggregator functions given a collection of keys, value pairs**
- **Must provide at least a map function, reduction function and a name of the result set**
- db.collection.mapReduce(<mapfunction>, <reducefunction>, { out: <collection>, query: <document>, sort: <document>, limit: <number>, finalize: <function>, scope: <document>, jsMode: <boolean>, verbose: <boolean> })

Indexes: High performance read

- Typically used for frequently used queries
- Necessary when the total size of the documents exceeds the
- amount of available RAM.
- Defined on the collection level
  - Can be defined on 1 or more fields
  - Composite index (SQL) → Compound index (MongoDB)
- B-tree index
- Only 1 index can be used by the query optimizer when
- retrieving data
- Index covers a query - match the query conditions and return the results using only the index;
- Use index to provide the results.

Replication of data

- **Ensures redundancy, backup, and automatic failover**
  - Recovery manager in the RDMS
- **Replication occurs through groups of servers known as replica sets**
  - Primary set –> set of servers that client tasks direct updates to
  - Secondary set –> set of servers used for duplication of data
  - At the most can have12 replica sets
    - Many different properties can be associated with a secondary set i.e. secondary-only, hidden delayed, arbiters, non-voting
  - If the primary set fails the secondary sets ‘vote’ to elect the new primary set

Consistency of data

- **All read operations issued to the primary of a replica set are consistent with the last write operation**
  - Reads to a primary have strict consistency
    - Reads reflect the latest changes to the data
  - Reads to a secondary have eventual consistency
    - Updates propagate gradually
  - If clients permit reads from secondary sets – then client may read a previous state of the database
  - Failure occurs before the secondary nodes are updated
    - System identifies when a rollback needs to occur
    - Users are responsible for manually applying rollback changes

Questions?