NoSQL Data Management: Concepts and Systems

November 15, 2018

BUAN 6320 Database Foundations

NoSQL Overview

- Introduction to NoSQL
- Basic Concepts for NoSQL
  - CAP-Theorem
  - Eventual Consistency
  - Consistent Hashing
  - MVCC-Protocol
  - Query Mechanisms for NoSQL
- Overview of NoSQL Systems

We want in a data system

We need a distributed database system having such features:

- Fault tolerance
- Speed
- High availability
- Consistency
- Scalability
- Sustainability

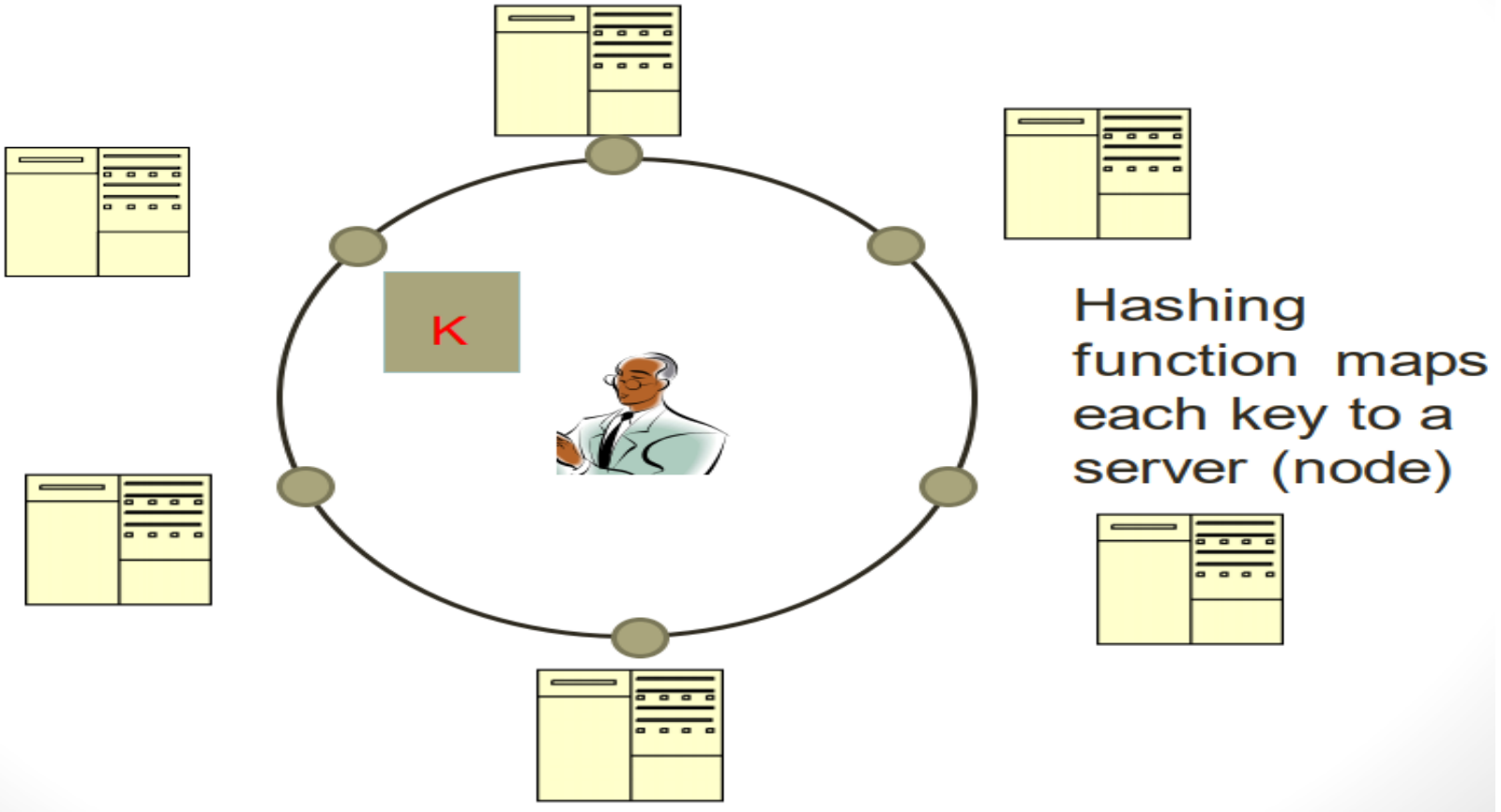
Speed But Deferred-Write

Some statistics about Facebook Search (using Cassandra)

MySQL > 50 GB Data
Writes Average : ~300 ms
Reads Average : ~350 ms

Rewritten with Cassandra > 50 GB Data
Writes Average : 0.12 ms
Reads Average : 15 ms

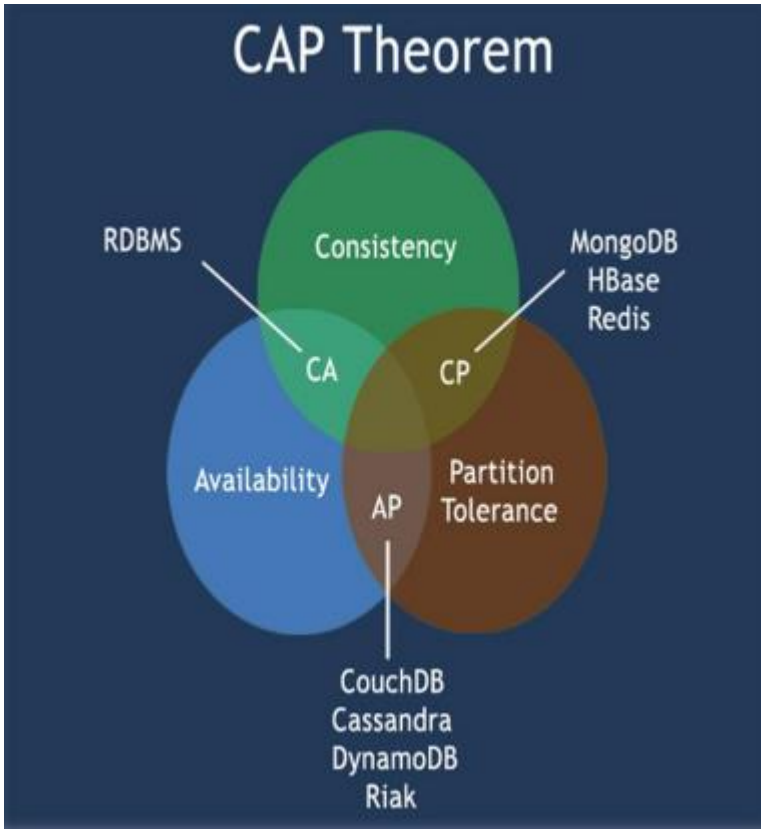
Typical NoSQL Architecture



CAP theorem for NoSQL

Impossible for any shared data-system to guarantee all of the three properties, simultaneously.

- Consistency-once data is written, all future read requests will contain that data
- Availability-the database is always available and responsive
- Partition Tolerance-if part of the database is unavailable, other parts are unaffected



CAP theorem for NoSQL

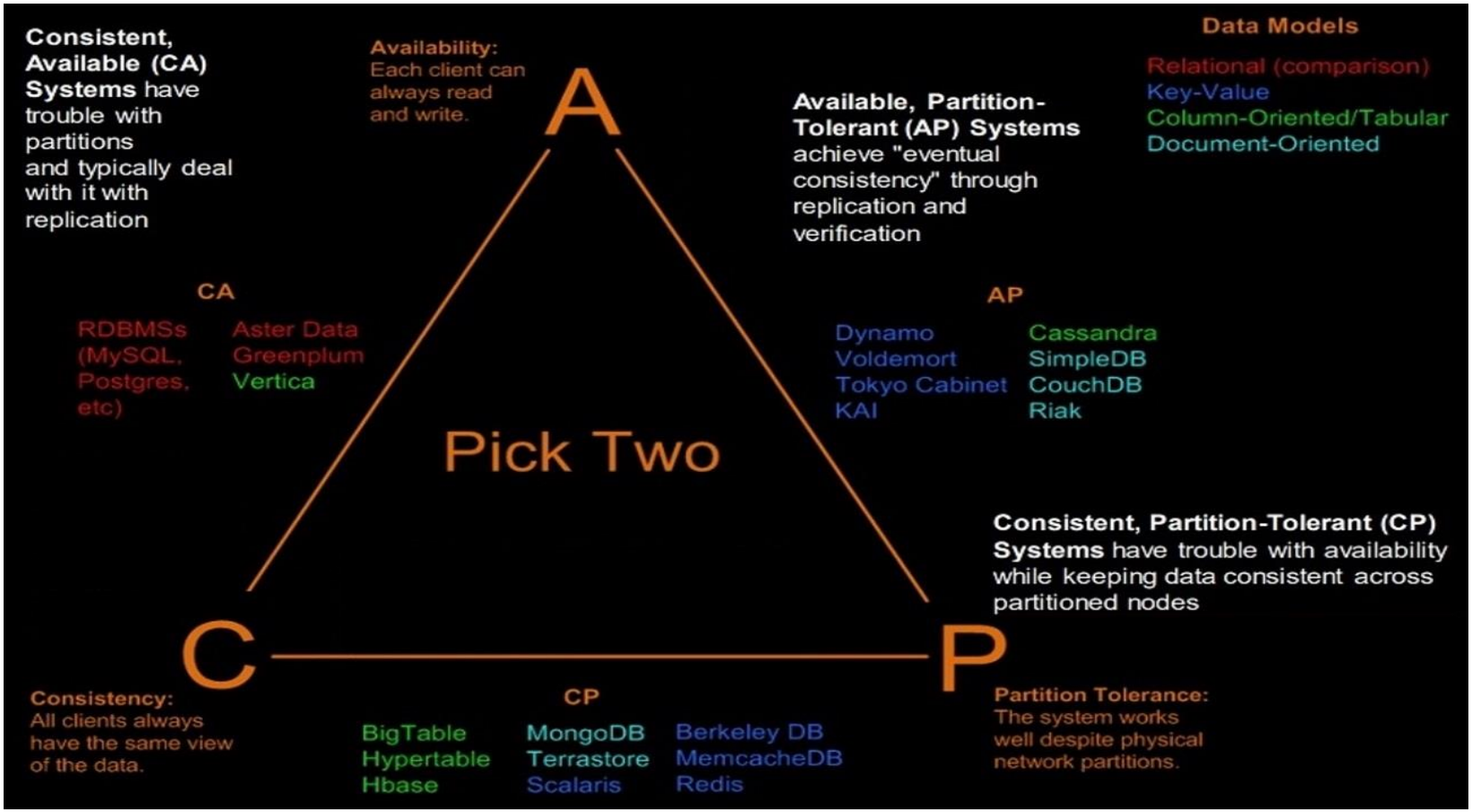
What the CAP theorem really says:

- If you cannot limit the number of faults and requests directed to any server and you insist on serving every request then you cannot possibly be provide consistency.

How it is interpreted:

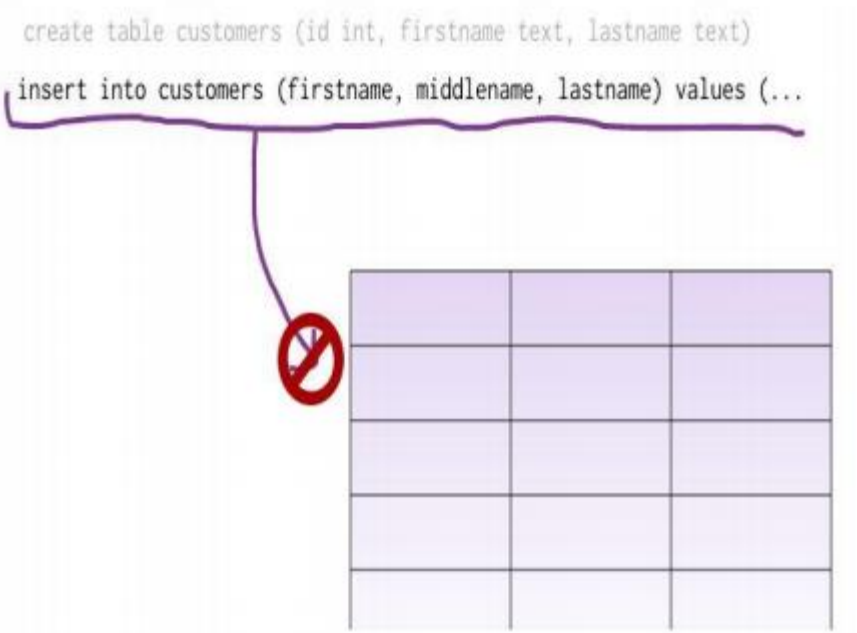
- You must always give something up: consistency, availability or tolerance to failure.

Visual Guide to NoSQL



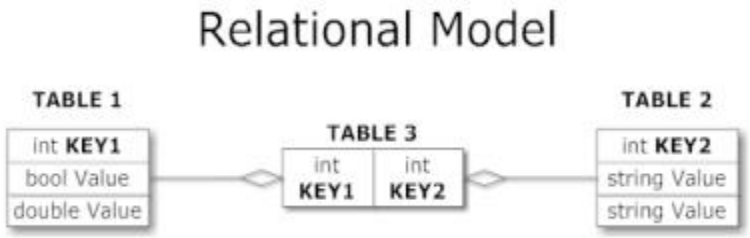
What is a schema data model?

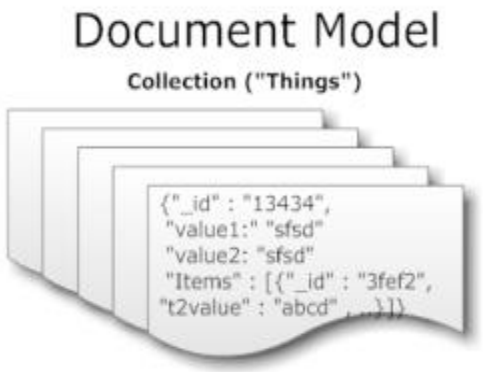
- You can't add a record which does not fit the schema
- You need to add NULLs to unused items in a row
- We should consider the datatypes. i.e.: you can't add a string to an integer field
- You can't add multiple items in a field (You should create another table: primary-key, foreign key, joins, normalization, ... !!!)



What is a schema-less data model?

- There is no schema to consider
- There is no unused cell
- There is no datatype (implicit)
- Most of considerations are done in application layer
- We gather all items in an aggregate (document)





Benefits of NoSQL

- **Elastic Scaling**
  - RDBMS scale up – bigger load , bigger server
  - NoSQL scale out – distribute data across multiple hosts seamlessly
- **DBA Specialists**
  - RDBMS require highly trained expert to monitor DB
  - NoSQL require less management, automatic repair and simpler data models
- **Big Data**
  - Huge increase in data RDBMS: capacity and constraints of data volumes at its limits
  - NoSQL designed for big data

Benefits of NoSQL

Flexible data models

- Change management to schema for RDBMS 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

- RDBMS 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

Benefits of NoSQL

Support

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

Maturity

- RDBMS mature product: means stable and dependable
  - Also means old no longer cutting edge nor interesting

- NoSQL are still implementing their basic feature set

Benefits of NoSQL

Administration

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

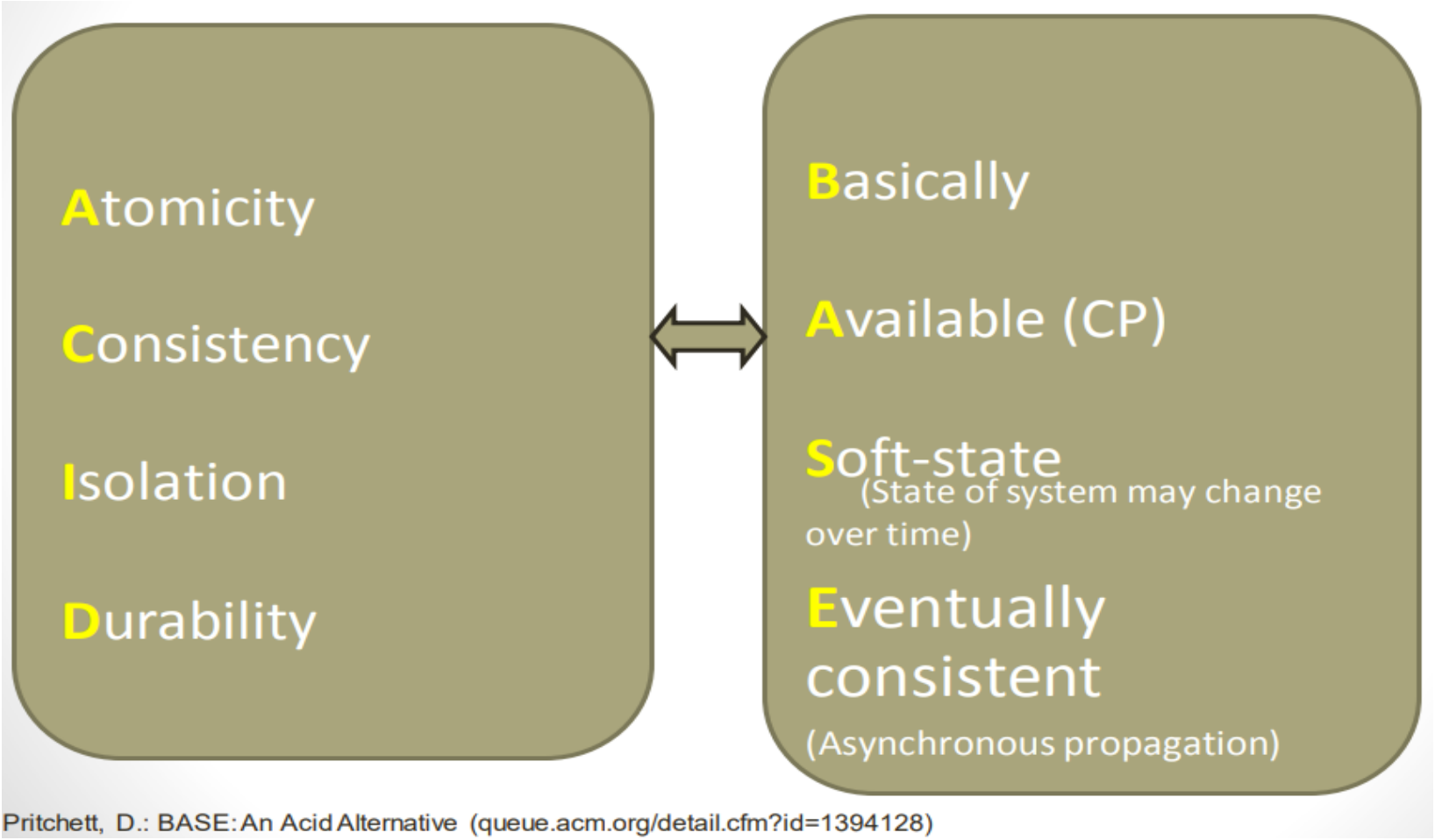
Lack of Expertise

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

Analytics and Business Intelligence

- RDBMS 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

RDBMS ACID to NoSQL BASE

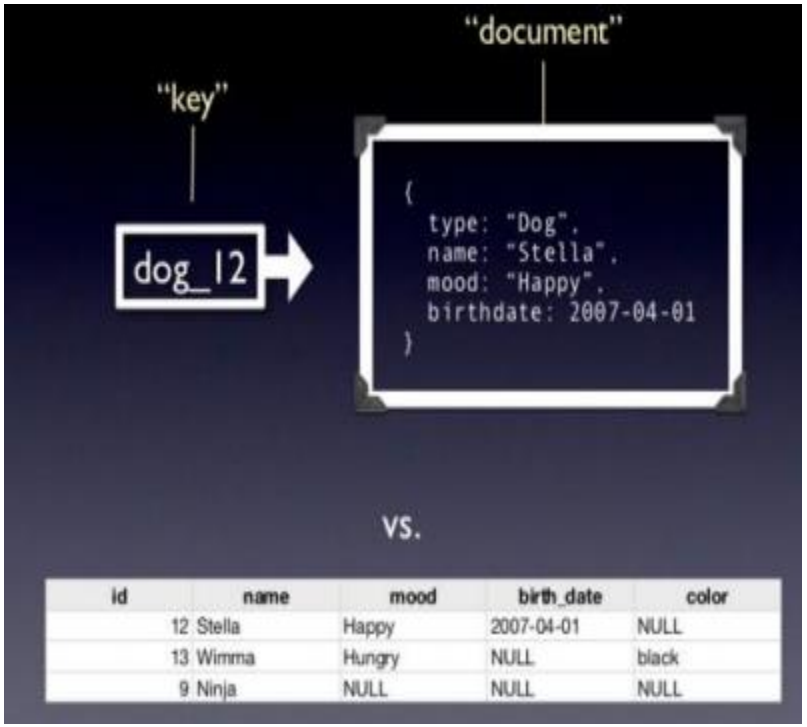


Pritchett, D.: BASE: An Acid Alternative (queue.acm.org/detail.cfm?id=1394128)

Document based data model

Pair each key with complex data structure known as data structure.

- Indexes are done via B-Trees.
- Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.



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
- 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, Java Scala, C#, C++, Haskell, Erlang

BSON format

- Binary-encoded serialization of JSON-like documents
- Zero or more key/value pairs are stored as a single entity
- Each entry consists of a field name, a data type, and a value
- Large elements in a BSON document are prefixed with a length field to facilitate scanning

Functionality of MongoDB

- Dynamic schema - No DDL
- Document-based database
- Secondary indexes
- Query language via an API
- Atomic writes and fully-consistent reads
- 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
- No ERD diagram
- Not well suited for heavy and complex transactions systems

Choices made for Design of MongoDB

• Scale horizontally over commodity hardware

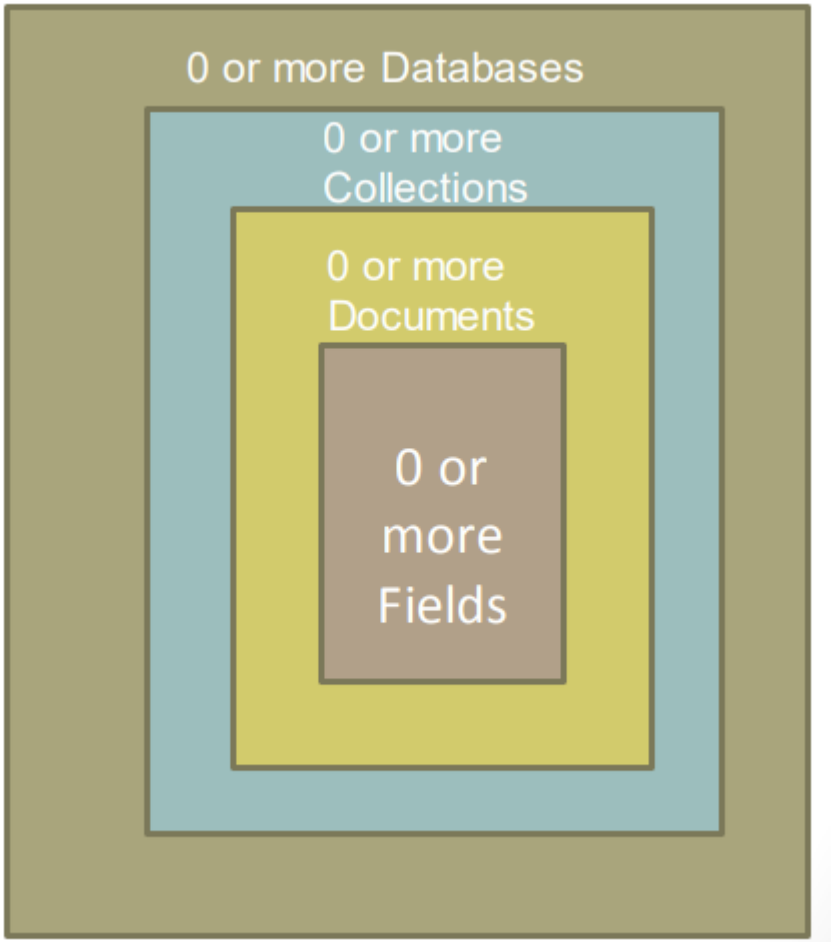
• Lots of relatively inexpensive servers

• Keep the functionality that works well in RDBMSs
– Ad hoc queries
– Fully featured indexes
– Secondary indexes

• What doesn't distribute well in RDBMS?
– Long running multi-row transactions
– Joins (computationally expensive)
– Both artifacts of the relational data model (row x column) 2

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 one or more 'fields'.
- MongoDB 'Indexes' function much like their RDBMS counterparts.



RDBMS Concepts to NoSQL

RDBMS		MongoDB
Database	⇨	Database
Table, View	⇨	Collection
Row	⇨	Document (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 evident in the design
- Embedded Document

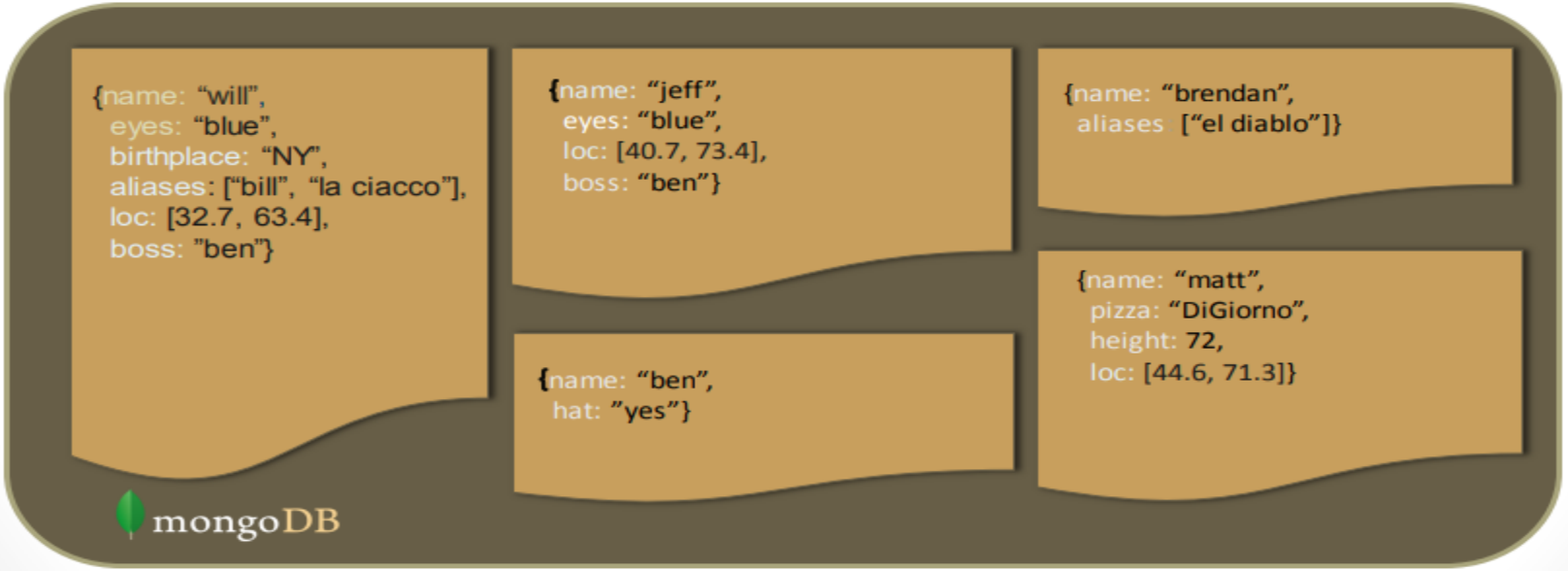
MongoDB Processes and Configuration

- 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.

Schema Free

MongoDB does not need any pre-defined data schema

- Every document in a collection could have different data
  - Addresses NULL data fields



JSON Format

Data is in name / value pairs or label / value pairs

- A name/value pair consists of a field name followed by a colon, followed by a value
  - Example: “name”: “R2-D2”
- Data is separated by commas
  - Example: “name”: “R2-D2”, race : “Droid”
- Curly braces hold objects
  - Example: {“name”: “R2-D2”, race : “Droid”, affiliation: “rebels”}
- An array is stored in brackets []
  - Example [{“name”: “R2-D2”, race : “Droid”, affiliation: “rebels”}, {“name”: “Yoda”, affiliation: “rebels”}]

MongoDB Features

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

Agile

Scalable

Sharding of Data

- Distributes a single logical database system across a cluster of machines
- Uses range-based partitioning to distribute documents based on a specific shard key
- Automatically balances the data associated with each shard
- Can be turned on and off per collection (table)

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>, <option>)
- **Delete**
  - db.collection.remove(<query>, <justOne>)

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

Create Operations

Db.collection specifies the collection or the 'table' to store the document

- 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 , fields in result set
  - Example: var PartsCursor = 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>, <field>, <insert>)
  - 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/
})
```

Aggregated Functionality

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: quantity} } })

Map Reduce Functionality

- 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>})
- More description of map reduce next lecture

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 have 12 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

When and When Not to use NoSQL

When / Why?

- When traditional RDBMS model is too restrictive (flexible schema)
- When ACID support is not “really” needed
- Object-to-Relational (O/R) impedance
- Because RDBMS is neither distributed nor scalable by nature
- Logging data from distributed sources
- Storing Events / temporal data
- Temporary Data (Shopping Carts / Wish Lists / Session Data)
- Data which requires flexible schema
- Polyglot Persistence i.e. best data store depending on nature of data

When Not?

- Financial Data
- Data requiring strict ACID compliance
- Business Critical Data