Big Data

Beyond Relational Data

noSQL databases

- ✓ Document DBs
  - MongoDB
- ✓ Graph databases
  - Neo4j

What is MongoDB?

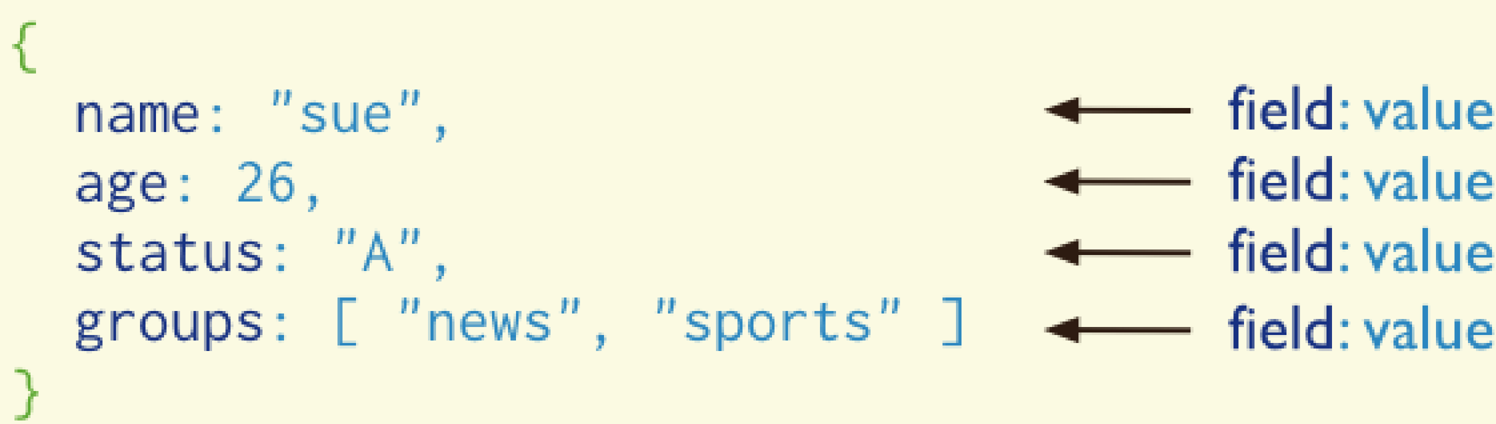
- Developed by 10gen: “**humongous**” DB
  - Founded in 2007
- A document-oriented NoSQL database
  - Hash-based, *schema-less database*
    - No Data Definition Language
    - can store hashes with any keys and values that you choose
      - Keys are a basic data type but in reality stored as strings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

Document DB

- ✓ Documents are the main concept.
- ✓ A Document-oriented database stores and retrieves documents (XML, JSON, BSON and so on).
- ✓ Documents are:
  - Self-describing
  - Hierarchical tree data structures (maps, collection and scalar values)

What is a Document DB?

- Document databases store documents in the value part of the key-value store where:
  - Documents are indexed using a BTree
  - queried using e.g, JavaScript query engine



What is a Document DB?

```
{
  "name": "Phil",
  "age": 26,
  "status": "A"
}
```

```
{
  "name": "Phil",
  "age": 26,
  "status": "A",
  "citiesVisited" : ["Chicago", "LA", "San
Francisco"]
}
```

- ✓ Documents may have different attributes
- ✓ But belongs to the same collection
- ✓ Different from relational databases where columns:
  - Stores the same type of values
  - Or null

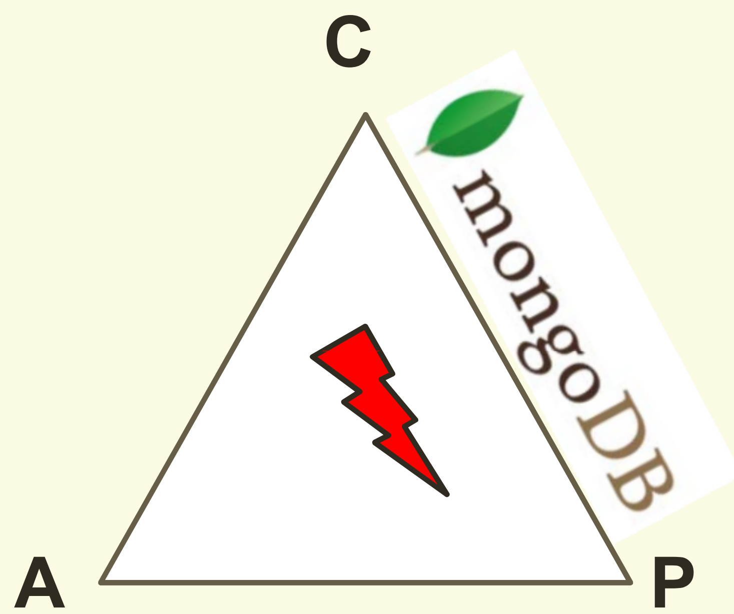
MongoDB

- MongoDB Features
  - Document-Oriented storage
  - Index Support
  - Replication & Availability
  - Auto-Sharding
  - Ad-hoc Querying
  - Fast In-Place Updates
  - Map/Reduce functionality
- Application need
  - Simple queries
  - Functionality provided applicable to most web applications
  - ~~Easy and fast integration of data~~
  - Not well suited for heavy and complex transactions systems

MongoDB: CAP approach

Focus on Consistency and Partition tolerance

- **C**onsistency
  - all replicas contain the same version of the data
- **A**vailability
  - system remains operational on failing nodes
- **P**artition tolarence
  - multiple entry points
  - system remains operational on system split

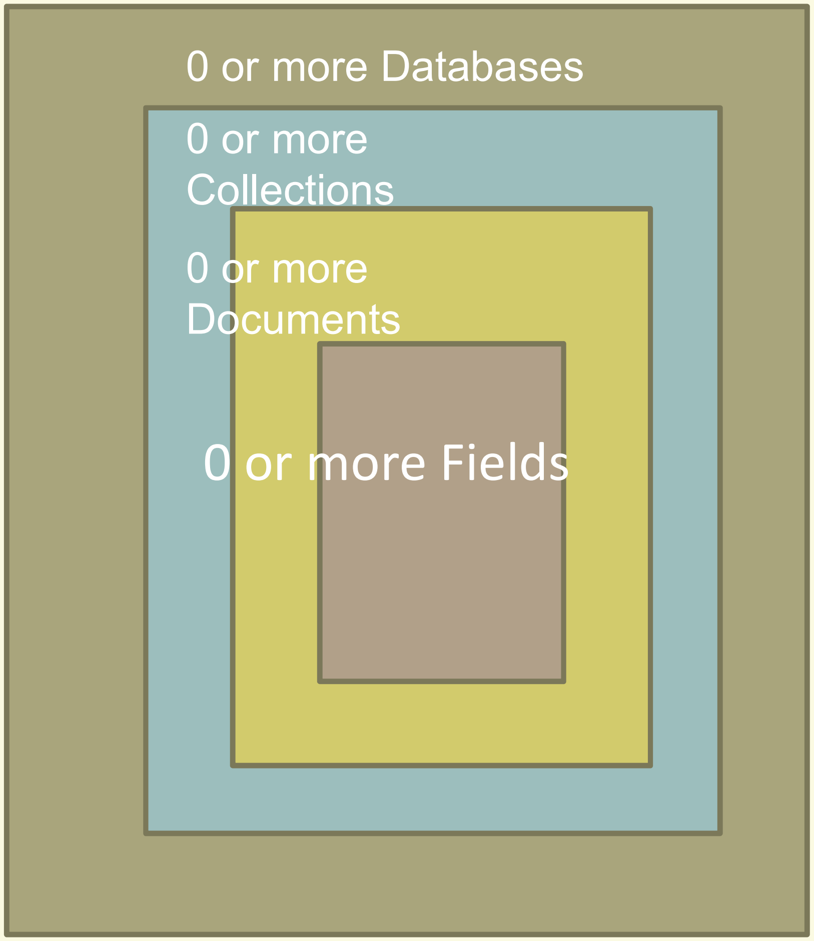


CAP Theorem:
satisfying all three at the same time is impossible

Document DB: Data model

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.



Documents: Data Model

- ✓ Data has a flexible schema
- ✓ This helps in matching document to objects
  - Each document can match the fields of a document also if with different structure
- ✓ Data is represented as a map
- ✓ Relations can be represented as: *references* and *embedded documents*

BSON

- “Binary JSON”
- Binary-encoded serialization of JSON-like docs
- Also allows “referencing”
- Embedded structure reduces need for joins
- Goals
  - Lightweight
  - Traversable
  - Efficient (decoding and encoding)

```
{"hello": "world"}    →    \x16\x00\x00\x00              // total document size
                           \x02                          // 0x02 = type String
                           hello\x00                     // field name
                           \x06\x00\x00\x00world\x00     // field value
                           \x00                          // 0x00 = type EOO ('end of object')
```

http://bsonspec.org/

The _id Field

- By default, each document contains an _id field. This field has a number of special characteristics:
  - Value serves as primary key for collection.
  - Value is unique, immutable, and may be any non-array type.
  - Default data type is ObjectId, which is "small, likely unique, fast to generate, and ordered."

```
{ "_id" :   "37010"
"city" :    "ADAMS",
"pop" :     2660,
"state" :   "TN",}
```

http://docs.mongodb.org/manual/reference/bson-types/

BSON Example

```
{ "_id" :  "37010"
      "city" :          "ADAMS",
      "pop" :           2660,
      "state" :         "TN",
      "councilman" :  { name: "John Smith"
                        address: "13 Scenic Way"
                      }
}
```

```
{                                   {"_id" : "1"
{"_id" : "1"                          "first name": "Bill"
  "first name": "Hassan"              "last name" : "Gates"
  "last name" : "Mir"               }
  "department":  20                 }
}
```

Documents: Structure References

contact document

```
{
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
  phone: "123-456-7890",
  email: "xyz@example.com"
}
```

user document

```
{
  _id: <ObjectId1>,
  username: "123xyz"
}
```

access document

```
{
  _id: <ObjectId3>,
  user_id: <ObjectId1>,
  level: 5,
  group: "dev"
}
```

Documents: Structure Embedded

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
            },
  access: {
             level: 5,
             group: "dev"
           }
}
```

Embedded sub-document

Embedded sub-document

RDB Concepts to Document DB

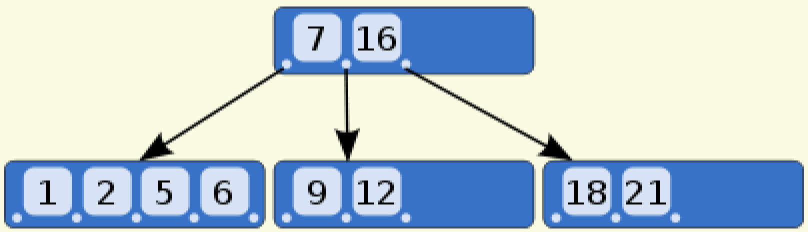
RDBMS		MongoDB
Database	⇨	Database
Table, View	⇨	Collection
Row	⇨	Document (BSON)
Column	⇨	Field
Index	⇨	Index
Join	⇨	Embedded Document
Foreign Key	⇨	Reference
Partition	⇨	Shard

Document DB: Indexing

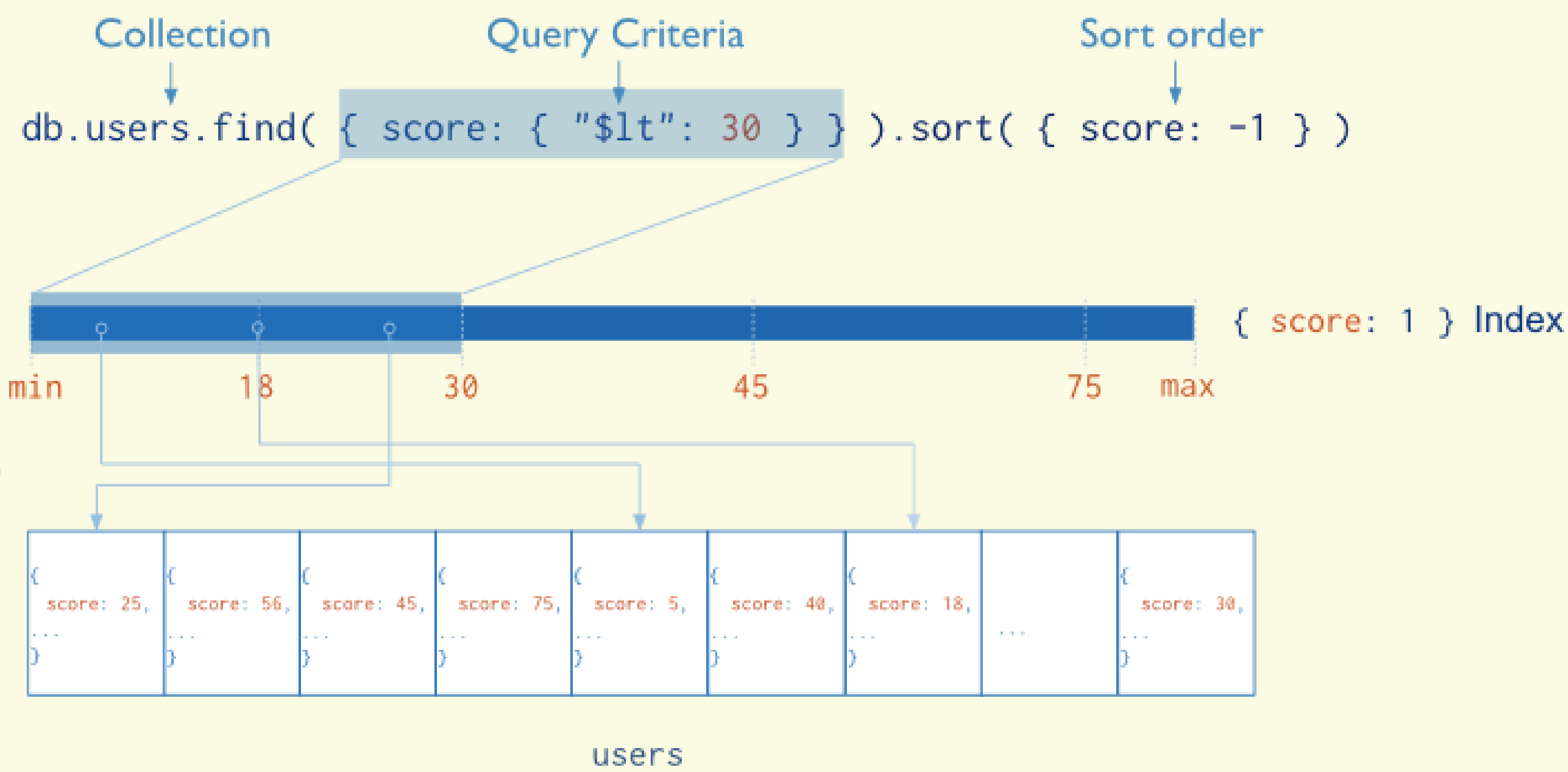
Documents: Indexing

- Indexes allows efficient queries on MongoDB.
- They are used to limit the number of documents to inspect (Otherwise, scan every document in a collection)
- By default MongoDB create indexes only on the _id field

- Indexes are created using B-tree and stores data of fields ordered by values.
- In addition MongoDB returns sorted results by using the index.

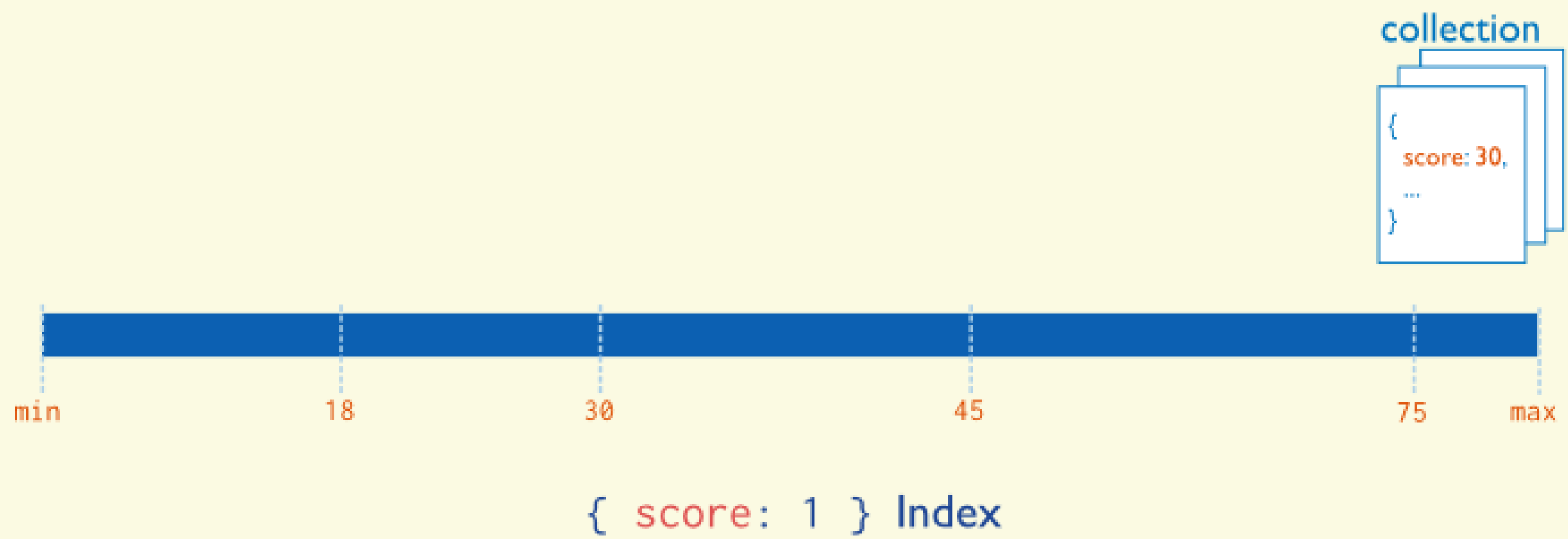


Documents: Indexing



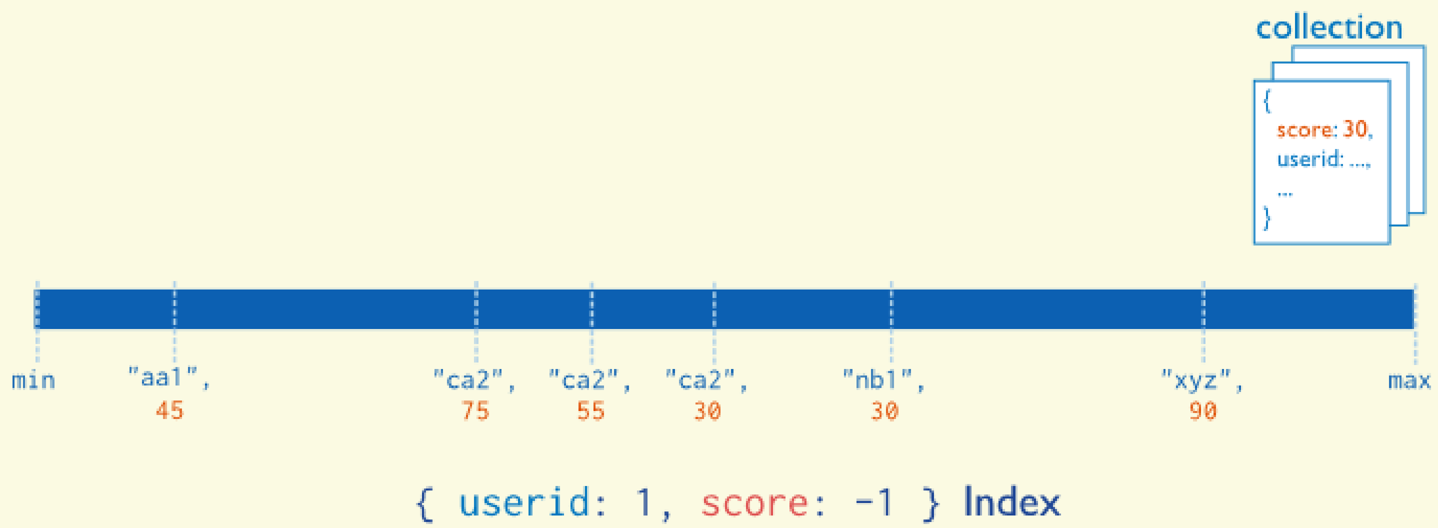
Documents: Index Types

- ✓ Single Field



Documents: Index Types

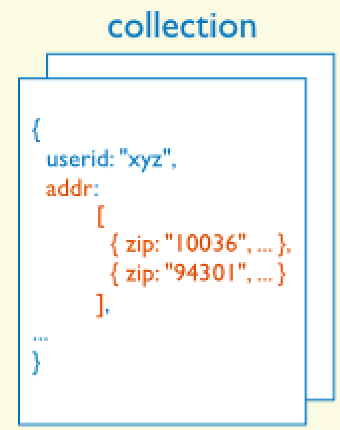
- ✓ Compound Index: indexed by attributes (left to right)

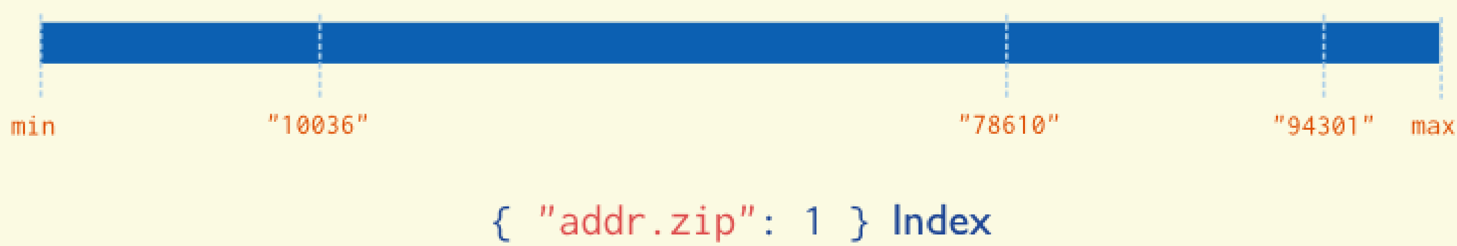


Documents: Index Types

Multikey Index:

- to index content in arrays





Document DB: CRUD

CRUD

- 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>)

CRUD example

```
> 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/
})
```

Query Interface

`db.users.find(`	collection
`{ age: { $gt: 18 } },`	query criteria
`{ name: 1, address: 1 }`	projection
`).limit(5)`	cursor modifier

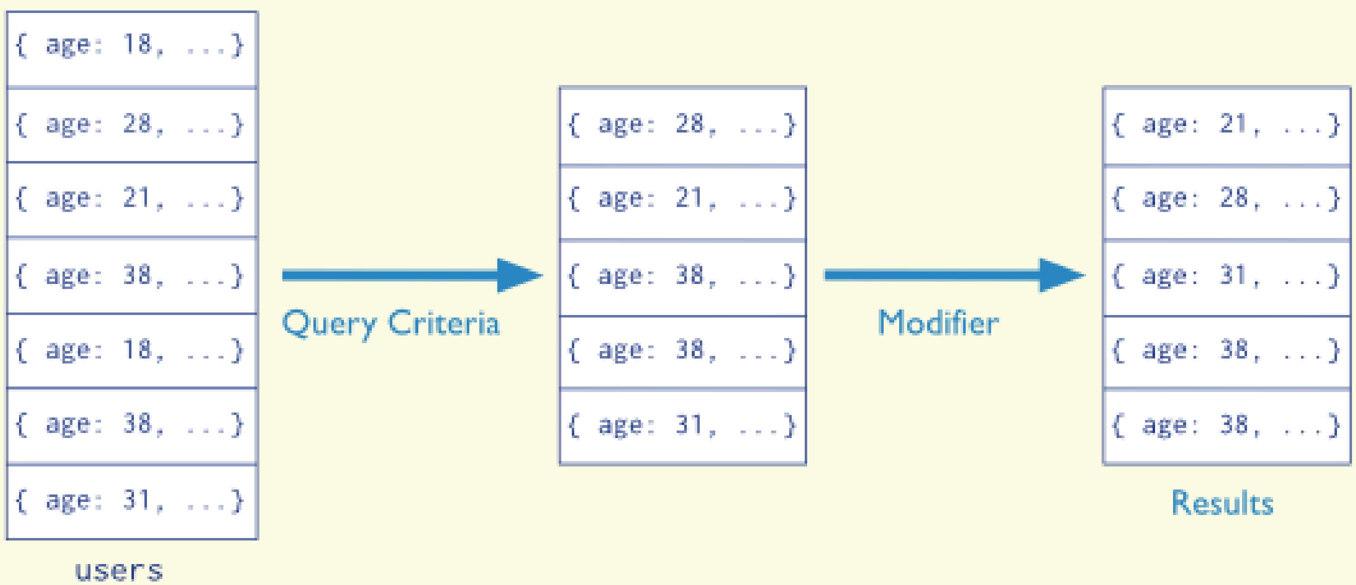
The same query in SQL

`SELECT _id, name, address`	projection
`FROM users`	table
`WHERE age > 18`	select criteria
`LIMIT 5`	cursor modifier

Query Statements

Collection　　　　Query Criteria　　　　Modifier

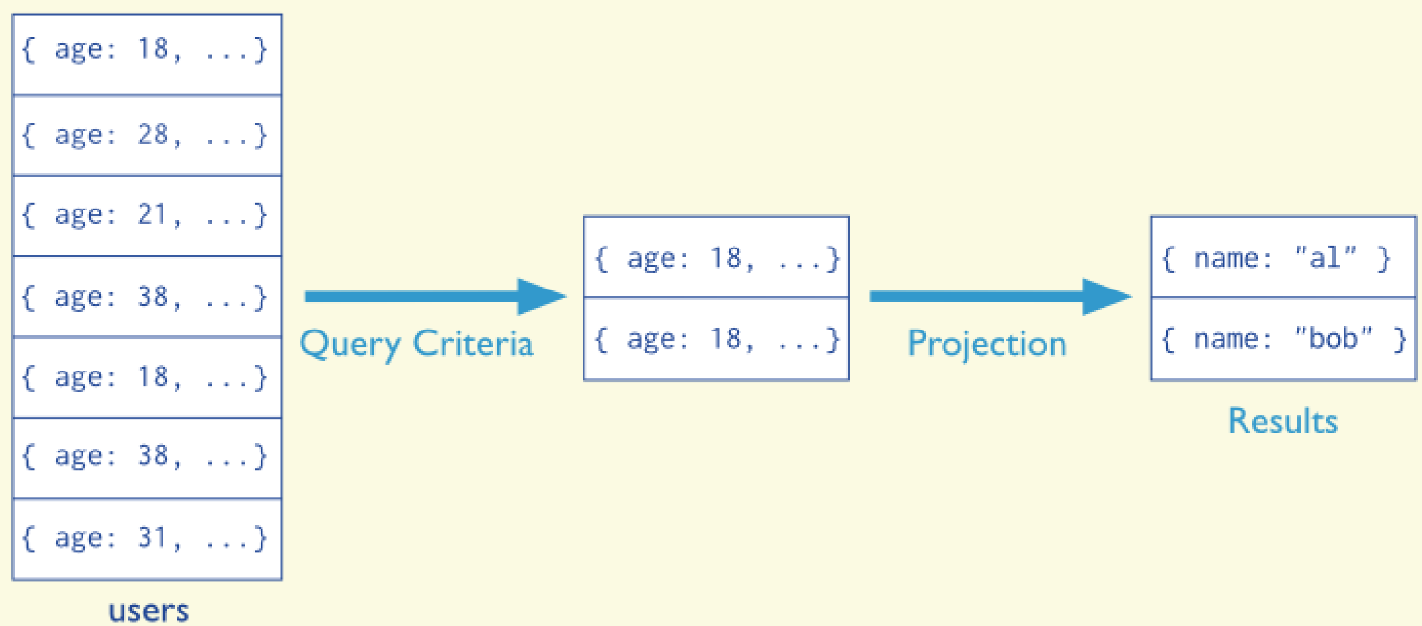
```
db.users.find( { age: { $gt: 18 } } ).sort( {age: 1 } )
```



Projections

Collection　　Query Criteria　　Projection

```
db.users.find( { age: 18 }, { name: 1, _id: 0 } )
```

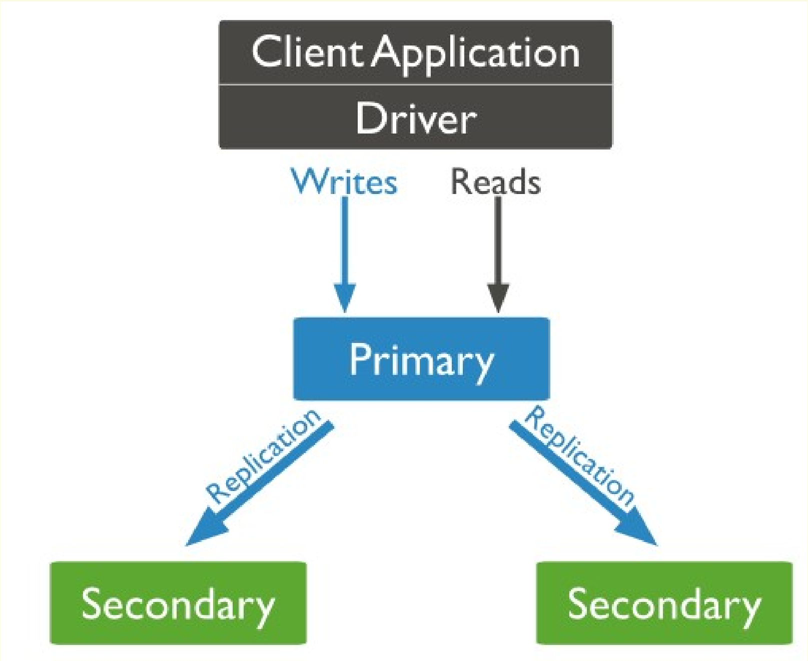


Document DB: Scaling

Replication of data

- Ensures redundancy, backup, and automatic failover
  - Recovery manager in the RDMS
- Replication 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

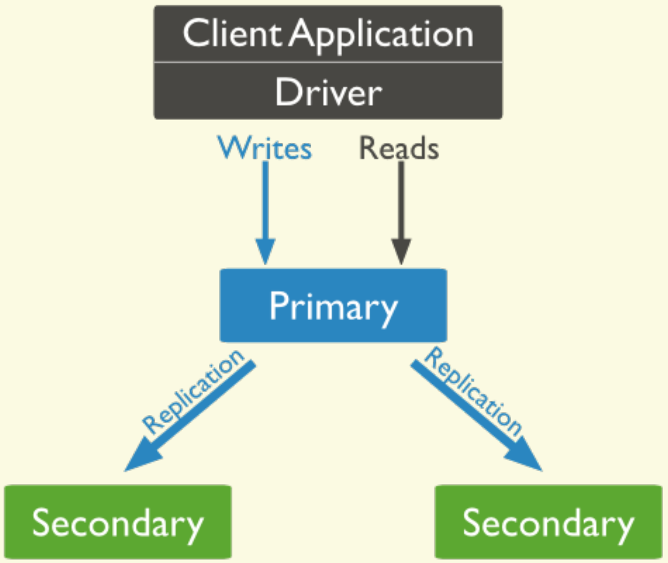
  - If the primary set fails the secondary sets 'vote' to elect the new primary set



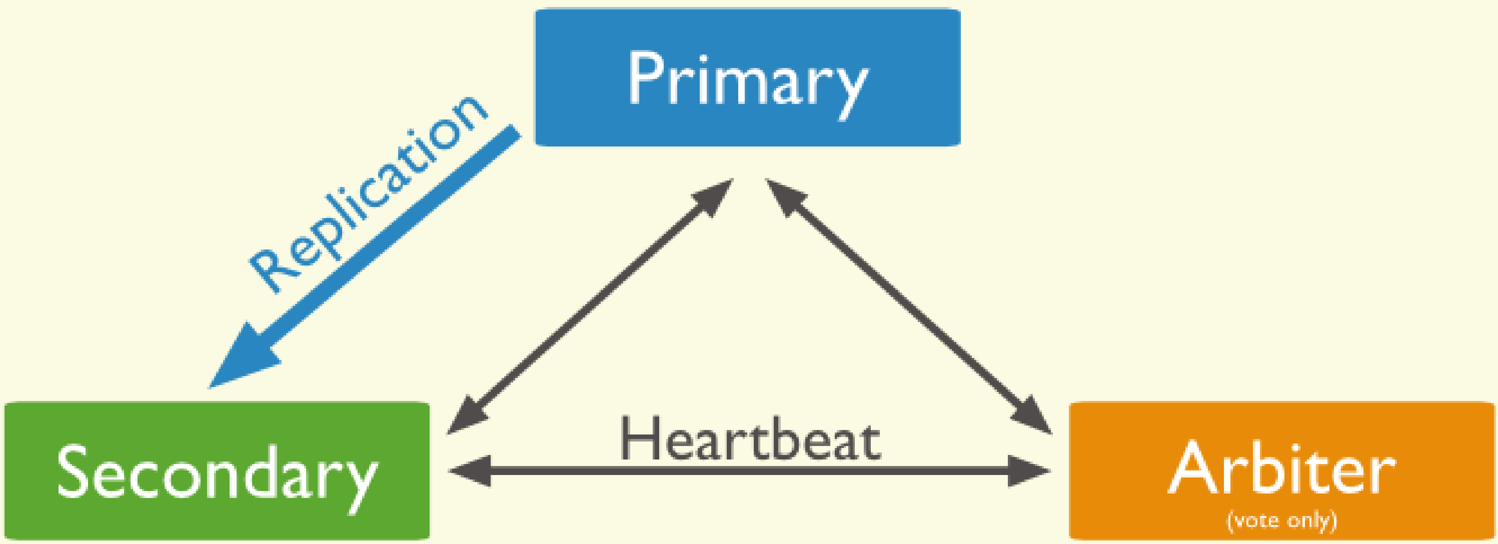
Scaling: Heavy Reads

- ✓ Scaling is achieved by adding more read slaves
- ✓ All the reads can be directed to the slaves.
- ✓ When a node is added it will sync with the other nodes.
- ✓ The advantage of this setting is that we do not need to stop the cluster.

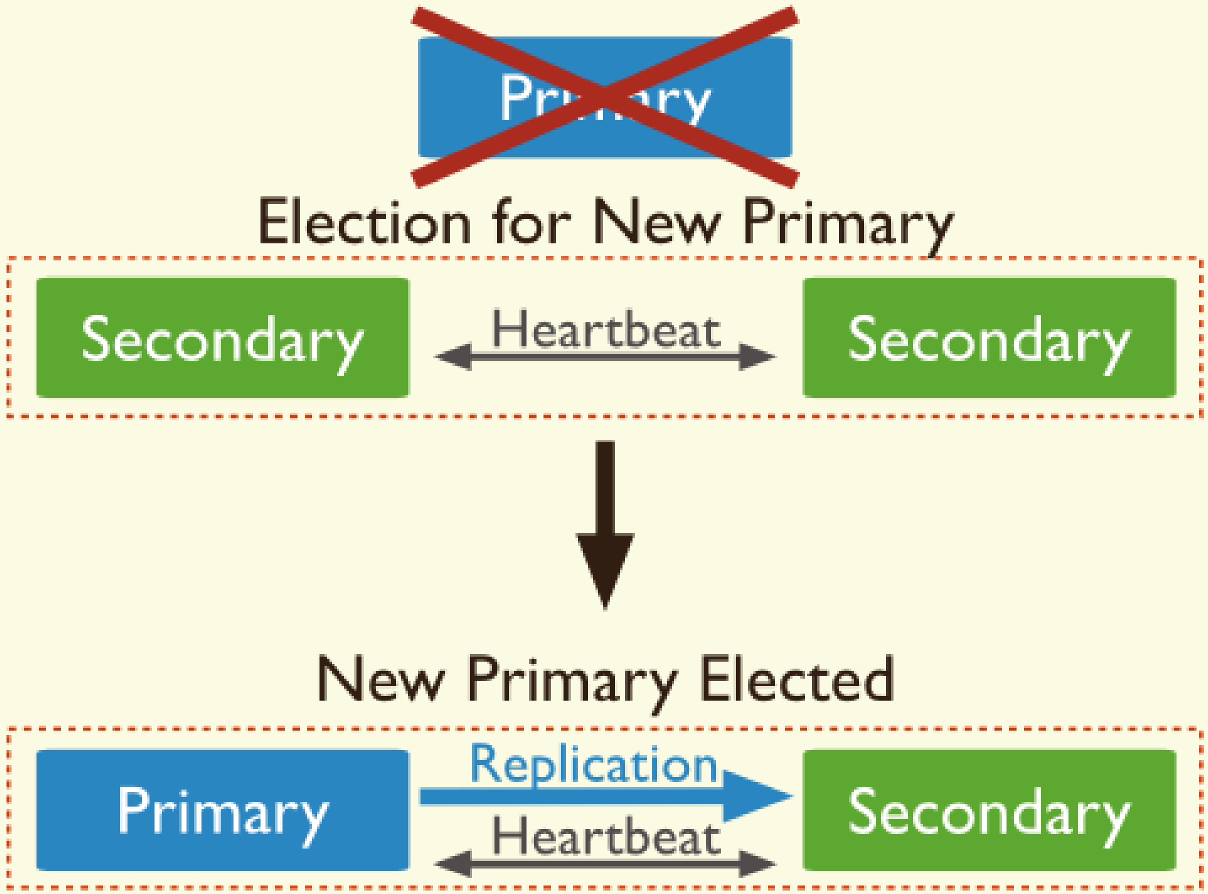
rs.add("mongo_address:27017")



Data Replication

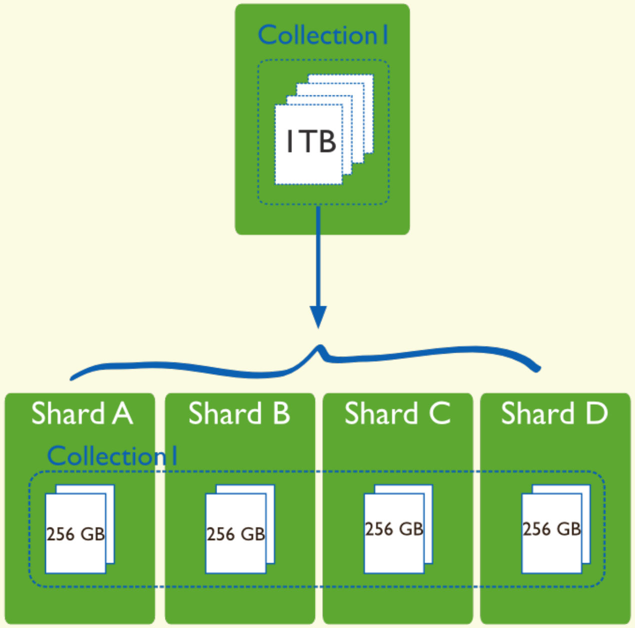


Automatic Failover



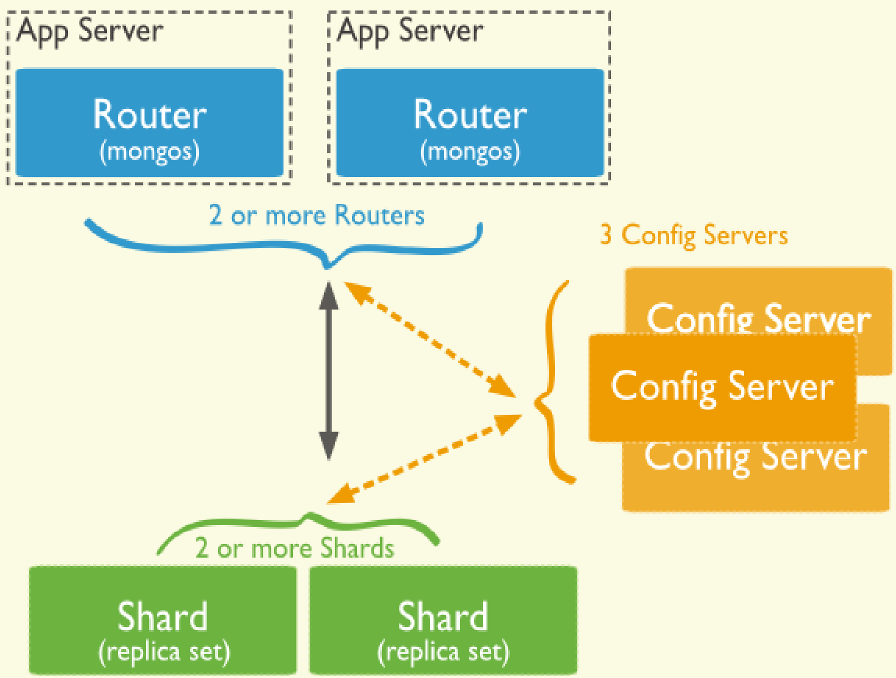
Scaling: Heavy Writes

- ✓ Sharding, or horizontal scaling divides the data set and distributes the data over multiple servers.
- ✓ Each shard is an independent database, and collectively, the shards make up a single logical database.



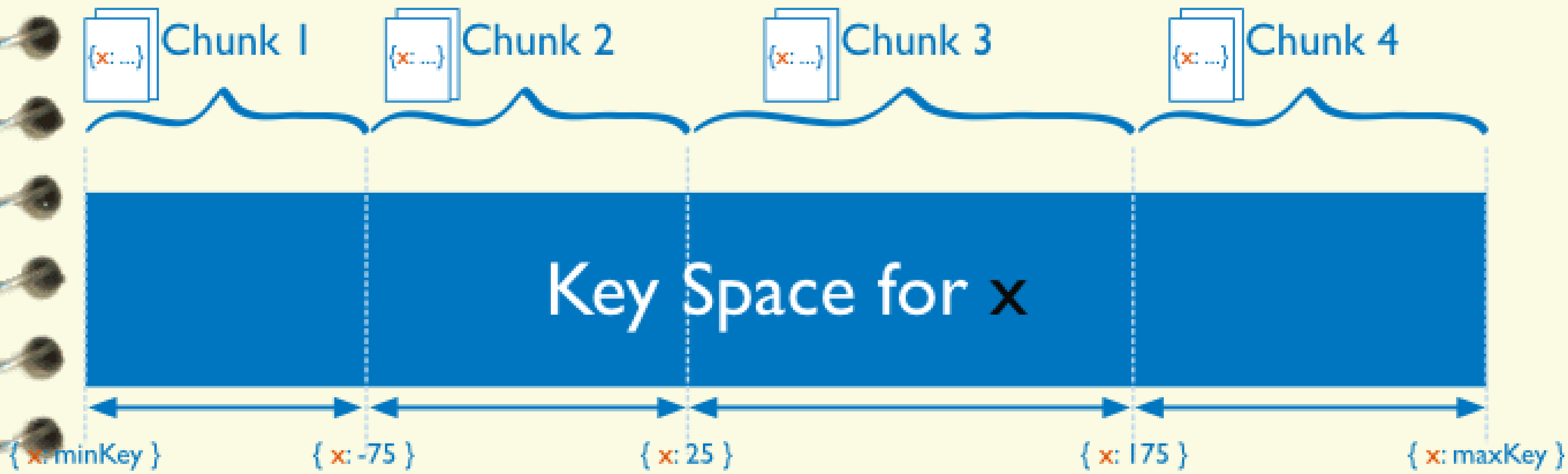
Sharding in MongoDB

- ✓ Shard: store the data
- ✓ Query Routers: interface to client and direct queries
- ✓ Config Server: store cluster’s metadata.



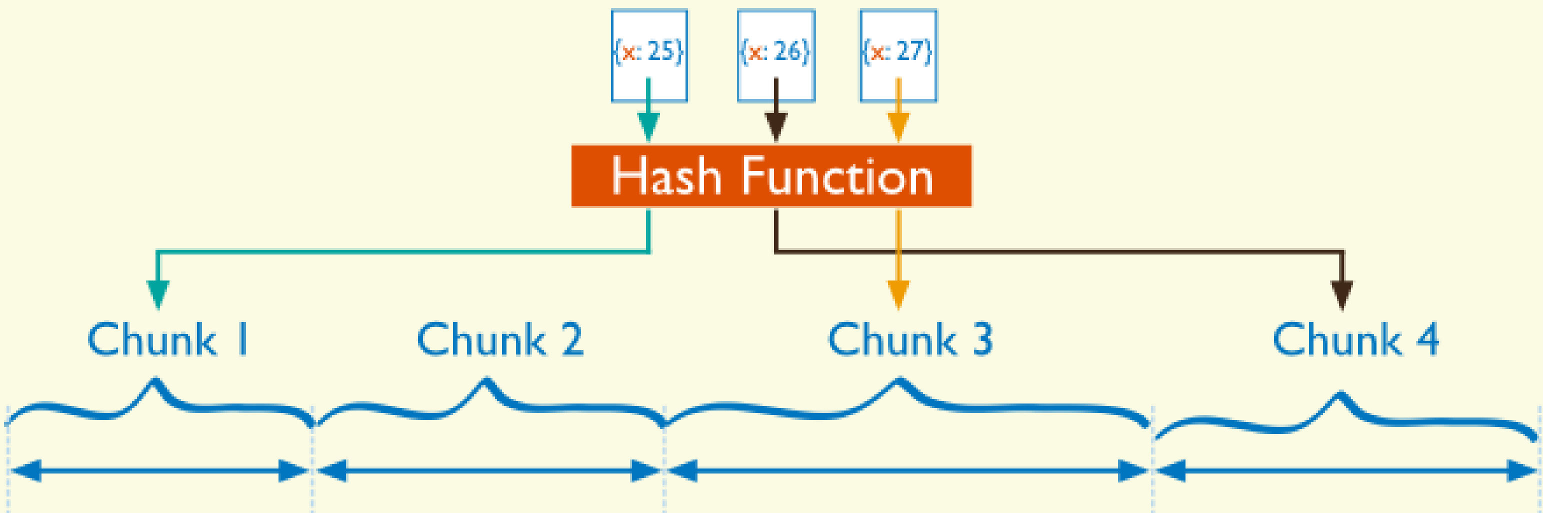
Range Based Sharding

- ✓ divides the data set into ranges determined by the shard key values to provide range based partitioning.



Hash Based Sharding

- computes a hash of a field's value, and then uses these hashes to create chunks



Performance Comparison

- ✓ Range based partitioning supports more efficient range queries.
- ✓ However, range based partitioning can result in an uneven distribution of data.
- ✓ Hash based partitioning, by contrast, ensures an even distribution of data at the expense of efficient range queries.

Document Store: Advantages

- ✓ Documents are independent units
- ✓ Application logic is easier to write. (JSON).
- ✓ Schema Free:
  - Unstructured data can be stored easily, since a document contains whatever keys and values the application logic requires.
  - In addition, costly migrations are avoided since the database does not need to know its information schema in advance.

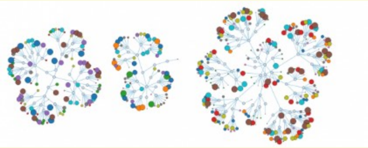
Suitable Use Cases

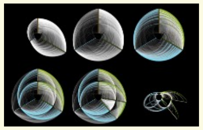
- ✓ **Event Logging:** where we need to store different types of event (order_processed, customer_logged).
- ✓ **Content Management System:** because the schema-free approach is well suited
- ✓ **Web analytics or Real-Time Analytics:** useful to update counters, page views and metrics in general.

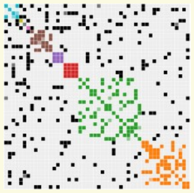
When Not to Use

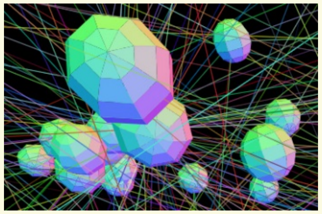
- ✓ **Complex Transactions:** when we need atomic cross-document operations (other options could be RavenDB or RethinkDB)
- ✓ **Queries against Varying Aggregate Structure:** i.e., when the structure of the aggregates vary because of continuous data evolutions

Graph databases









Motivations

- ✓ The necessity to represent, store and manipulate complex data make RDBMS somewhat obsolete

- ✓ Problem 1: Violations of the 1NF
  - Multi-valued attributes
  - Complex attributes

- ✓ Problem 2 : Accommodate Changes
  - acquiring data from autonomous dynamic sources or Web
  - RDBMS require schema renormalization

- ✓ Problem 3: Unified representation for:
  - Data
  - Knowledge (Schemas are a subset of this)
  - Queries [results + def]
  - Models (Concepts)

Existing Approaches

- ✓ RDBMS –need schema renormalization
- ✓ Approaches that try to fix the above mentioned problems:
  - OO Databases [P1], [P2] - graphs [but procedural]
  - XML Databases [P1] (somewhat [P3]) – trees
  - OORDBMS [P1] – graphs with foreign keys
  - RDF triple stores [P1, P2], somewhat [P3]
- ✓ Others
  - Datalog – more efficient fragment of Prolog
  - Network Models - graphs
  - Hierarchical Models – trees

What is a Graph Database?

- ✓ A database with an explicit graph structure: Each node knows its adjacent nodes
- ✓ As the number of nodes increases, the cost of a local step (or hop) remains the same; Plus an Index for lookups
- ✓ Express Queries as Traversals. Fast deep traversal instead of slow SQL queries that span many table joins.
- ✓ Very natural to express graph related problem with traversals (recommendation engine, find shortest path etc..)
- ✓ Seamless integration with various existing programming languages.
  - Two design principle: Declarativity & Change
- ✓ Distinguish between "Database for graph as object"!

Database Representation

- Sailors(<u>sid:integer</u>, sname:char(10), rating: integer, age:real)
- Boats(<u>bid:integer</u>, bname:char(10), color:char(10))
- Reserve(<u>sid:integer, bid:integer, day:date</u>)

Sailors

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

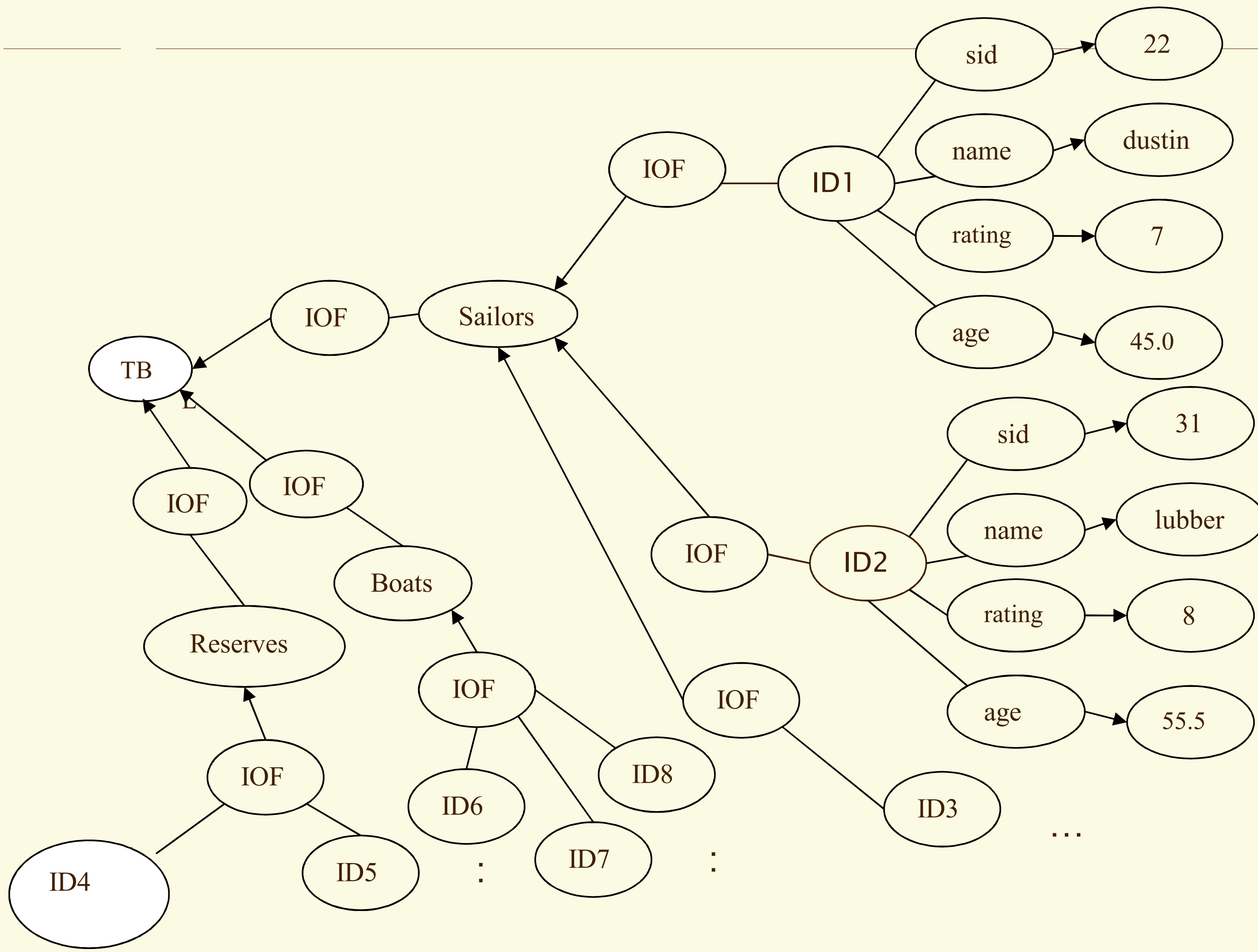
Reserves

<u>sid</u>	<u>bid</u>	<u>day</u>
22	101	10/10/96
58	103	11/12/96

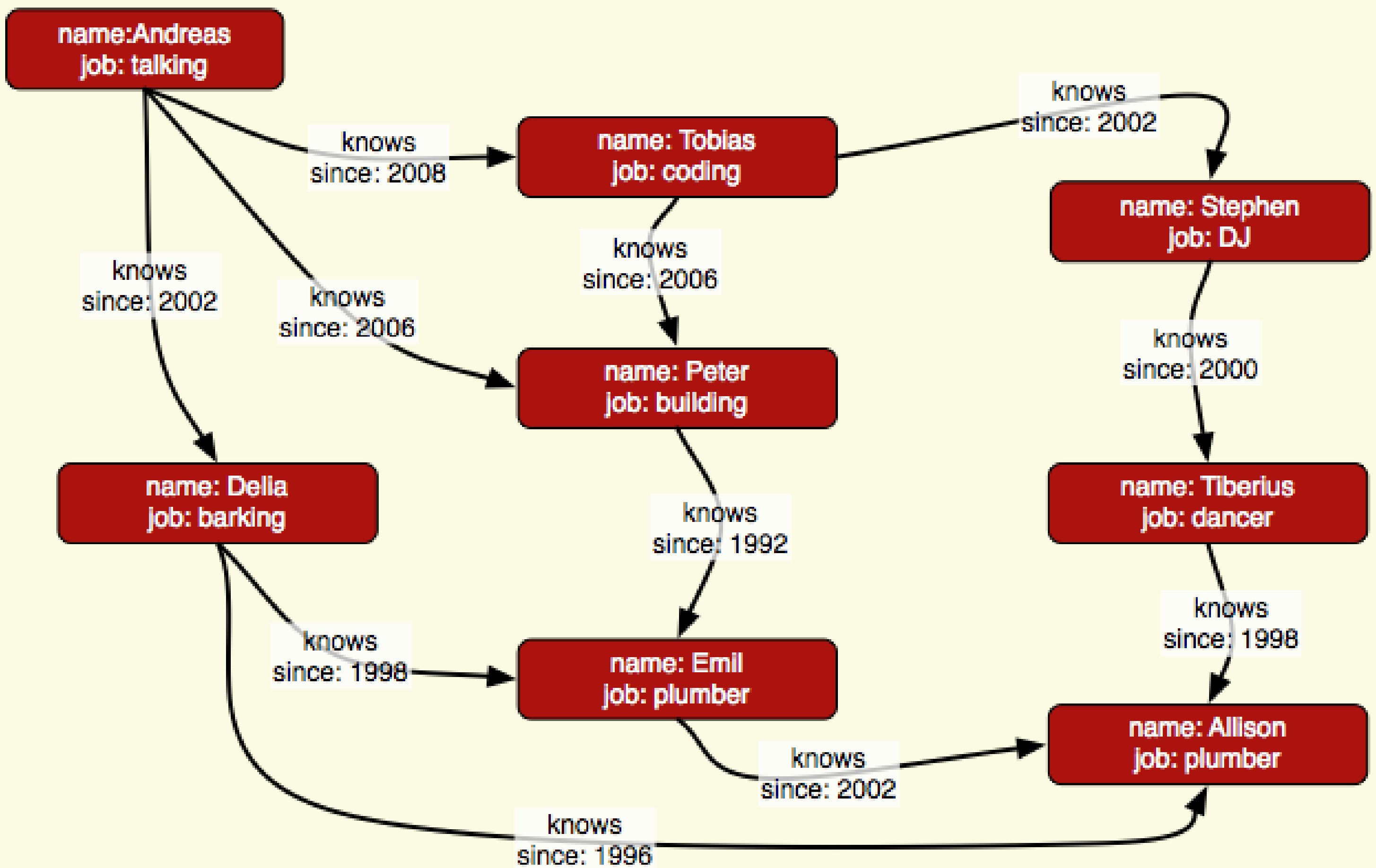
Boats

<u>bid</u>	bname	color
101	Interlake	red
102	Clipper	green
103	Marine	red

Graph Representation

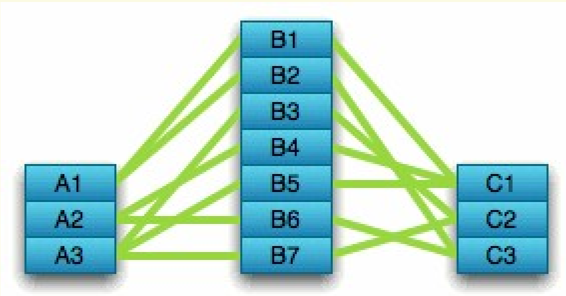


Property Graph

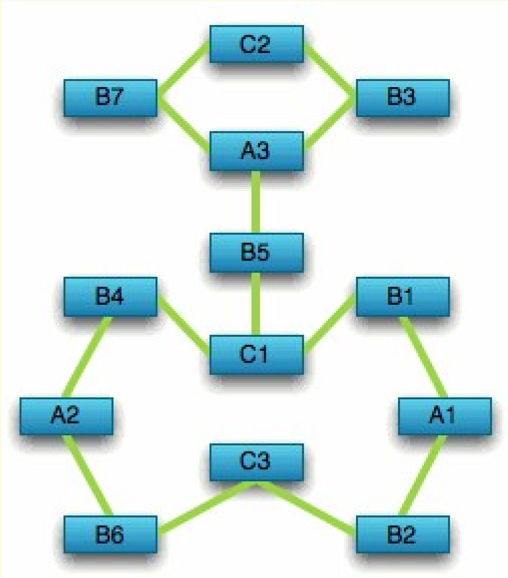


Compared to Relational Databases

Optimized for aggregation



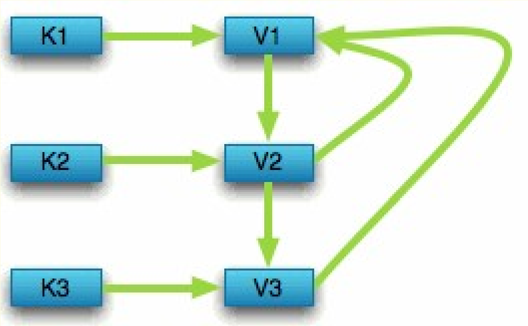
Optimized for connections



Compared to Key Value Stores

Optimized for
simple look-ups

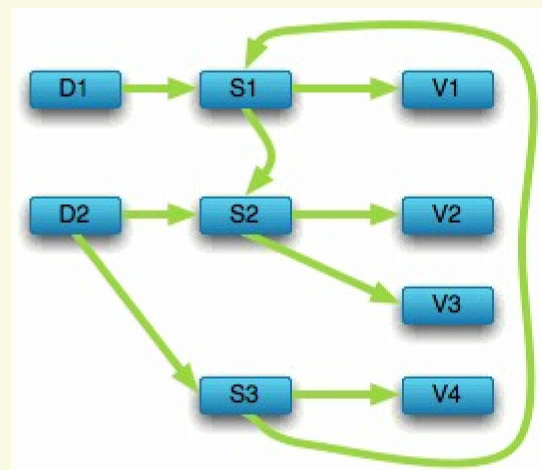
Optimized for
traversing connected data



Compared to Document Stores

Optimized for "trees" of data

Optimized for seeing the forest and the trees, the branches, and the trunks



Social Network “path exists” Performance

- Experiment:
  - ~1k persons
  - Average 50 friends per person
  - `pathExists(a,b)` limited to depth 4

	# persons	query time
Relational database	1000	2000ms
Neo4j	1000	2ms
Neo4j	1000000	2ms

Neo4j?

- ✓ A Graph Database + Lucene Index
- ✓ Property Graph
- ✓ Full ACID (atomicity, consistency, isolation, durability) (?)
- ✓ High Availability (with Enterprise Edition)
- ✓ 32 Billion Nodes, 32 Billion Relationships, 64 Billion Properties
- ✓ Embedded Server
- ✓ REST API

Good For

- ✓ Highly connected data (social networks)
- ✓ Recommendations (e-commerce)
- ✓ Path Finding (how do I know you?)
- ✓ A* (Least Cost path)
- ✓ Data First Schema (bottom-up, but you still need to design)

Summary: noSQL systems

Summary

- ✓ SQL Databases
  - Predefined Schema
  - Standard definition and interface language
  - Tight consistency (ACID)
  - Well defined semantics

- ✓ NoSQL Database
  - No predefined Schema
  - Per-product definition and interface language
  - Getting an answer quickly is more important than getting a correct answer (BASE)

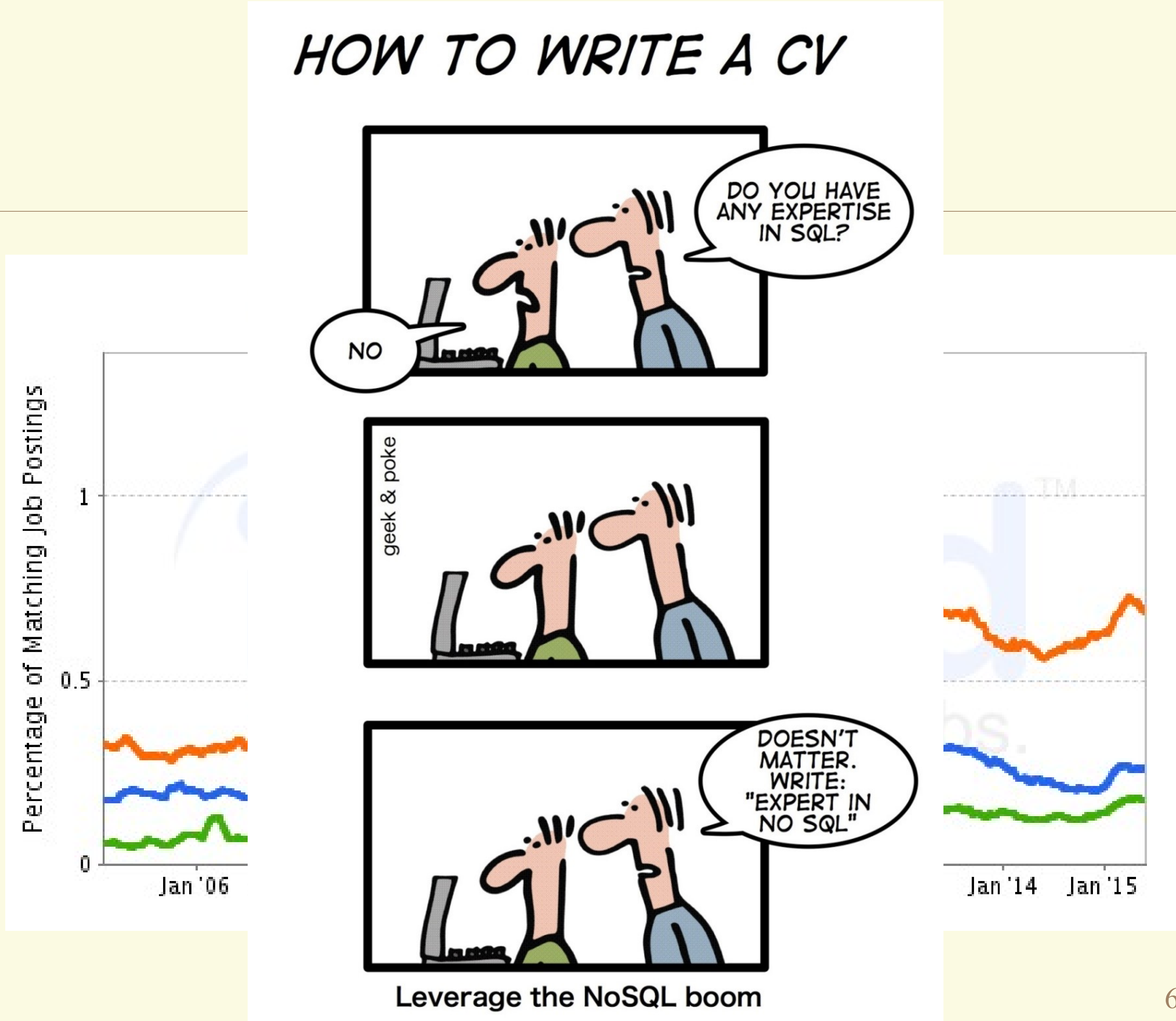
Summary: noSQL Common Advantages

- ✓ Cheap, easy to implement (open source)
- ✓ Data are replicated to multiple nodes (therefore identical and fault-tolerant) and can be partitioned
  - Down nodes easily replaced
  - No single point of failure
- ✓ Easy to distribute
- ✓ Don't require a schema
- ✓ Can scale up and down
- ✓ Relax the data consistency requirement (CAP)

Summary: What are we giving up?

- ✓ joins
- ✓ group by
- ✓ order by
- ✓ ACID transactions (none are strict ACID!)
- ✓ SQL as a sometimes frustrating but still powerful query language
- ✓ easy integration with other applications that support SQL

Job Trends from Indeed.com
noSQL
MongoDB
couchDB
neo4j
Percentage of Matching Job Postings
0.15
0.1
0.05
0
Jan '06
Jan '07
Jan '08
Jan '09
Jan '10
Jan '11
Jan '12
Jan '13
Jan '14
Jan '15


Percentage of Matching Job Postings
1
0.5
0
Jan '06
Jan '14
Jan '15
HOW TO WRITE A CV
DO YOU HAVE ANY EXPERTISE IN SQL?
NO
geek & poke
DOESN'T MATTER. WRITE: "EXPERT IN NO SQL"
Leverage the NoSQL boom