Document Stores

Alberto Abelló & Oscar Romero

Knowledge Objectives

1. Explain the main difference between key-value and document stores
2. Justify why indexing is a first-class citizen for document-stores and it is not for key-value stores

Application Objectives

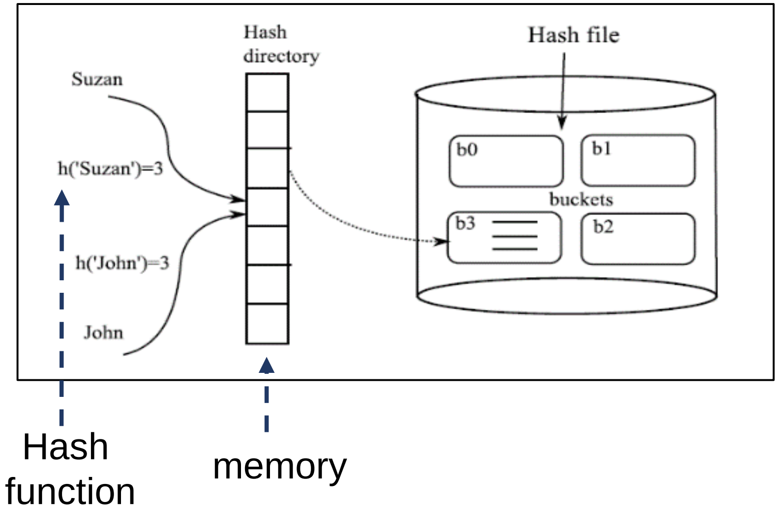
1. Given an application layout and a small query workload, design a document-store providing optimal support according to a given set of criteria

Distributed Architectures

Consistent Hashing

Indexes

- Index – associates a key with its (physical) address
  - Trees – logarithmic search complexity (-> distributed trees seen in the key-value lecture)
  - **Hash tables** – constant search complexity
    - Good for point queries
    - Do not support range search



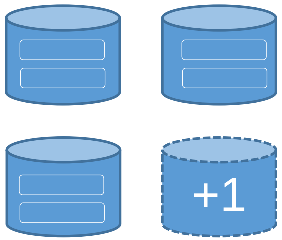
  - And what if the data grows too much?

A Design Alternative: Distributed Hashing

- Distributed Hashing challenges:
  - Dynamicity: grow and shrink rapidly
    - Distribution: Assign buckets to participating nodes

    (all the nodes should share the hash function for it to work)

      E.g., h(x) = x % #servers



    - Adding a new server implies modifying h…
      - Communicating the new h' to all servers
      - Re-hashing all the objects!
  - Location of the hash directory: any access must go through the hash directory
      - Potential bottleneck

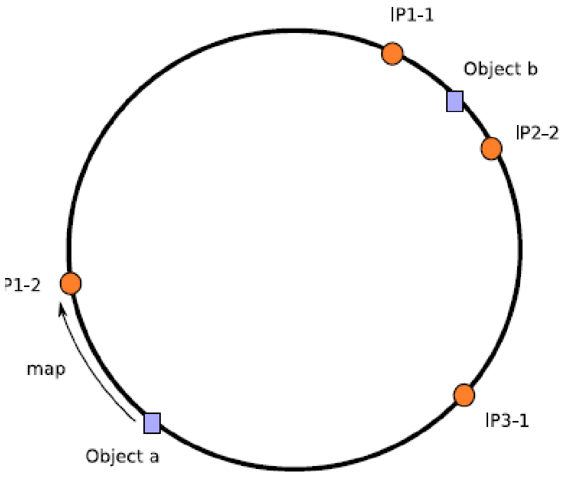
Consistent Hashing: Motivation

- Method initially proposed in the context of distributed caching
  - Results of the most common queries are in *caches* (i.e., in-memory) of several servers
  - A dedicated proxy machine records which server stores which query results
  - Queries are assigned to servers according to a hash function over the query

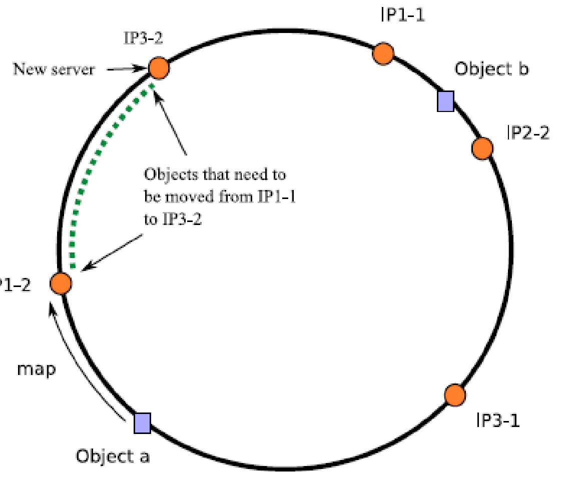
- Currently applied to distribution of data in distributed data stores
  - Supports high dynamicity of the infrastructure (servers may come and go at a rapid pace)
  - Most current key-value (and document-stores) use distributed hashing
    - Memcached
    - Voldemort
    - Cassandra
    - Dynamo / SimpleDB
    - CouchDB
    - MongoDB (current release)

Consistent Hashing

- Coping with dynamicity:
  - The hash function <u>never</u> changes
    - Choose a large domain *D* (address space) and map server IPs and object keys to such domain
    - Organize *D* as a ring so each node has a successor (clockwise)
    - Objects are assigned as follows:
      - For an object O, $f(O) = D_o$
      - $D_{o'}$ and $D_{o''}$ are two nodes in the ring such that
        - $D_{o'} < D_o <= D_{o''}$
      - O is assigned to $D_{o''}$
  - Adding a new server is straightforward
    - It is placed in the ring and part of its successors objects transferred
- Further refinements:
  - Assign to the same server several hash values (virtual servers) to balance load (and deal with heterogeneity)



Mapping of objects to servers

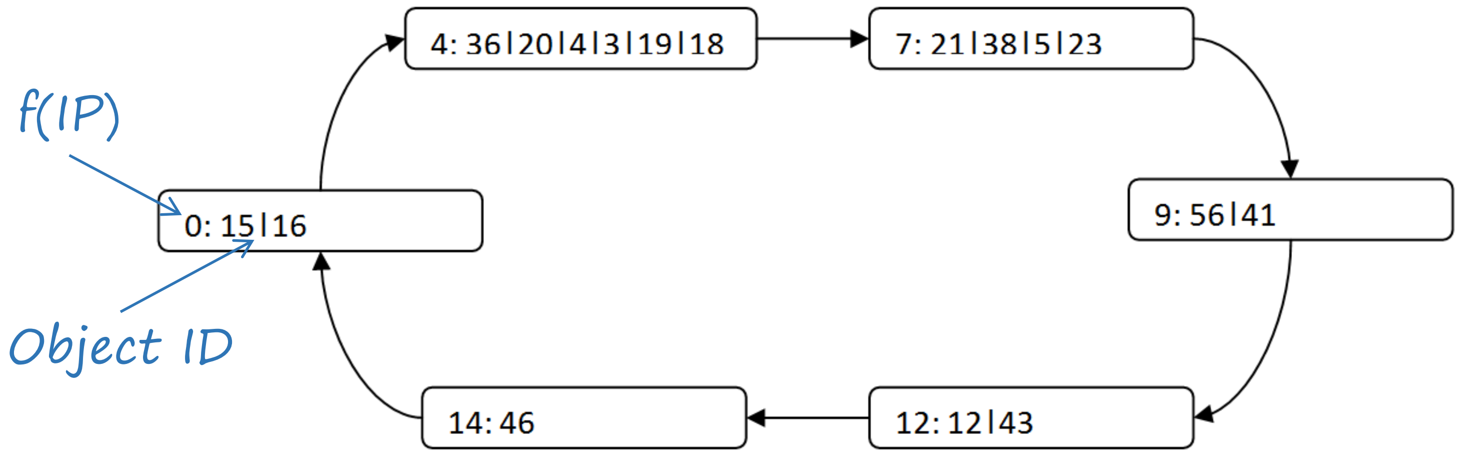


Server IP3-2 is added, with local re-hashing

Activity: Consistent Hashing

- Objective: Understand how consistent hashing works
- Tasks:
  1. (5’) By pairs, solve the following exercise
     - What happens in the structure when we register a new server with IP address “37”? Draw the result
  2. (5’) Discussion

f(IP) = IP % D



Current state of the consistent hash (D=16)

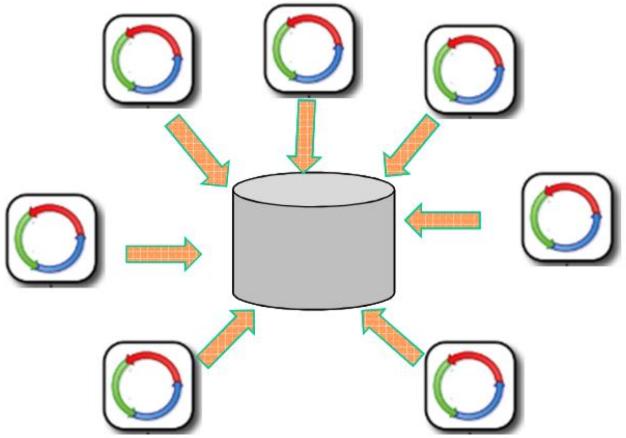
Document-Oriented DBs

Key-value enhancements

Integrated Databases vs Application Databases

- SQL and relational databases played a key role as **integration mechanism between applications**
  - Multiple applications using a common integrated database

- All applications operate on a consistent set of persistent data
- More complex structure
- Changes by different apps need to be coordinated
- Different apps have different performance needs, thus call for different index structures
- Complex access policies



A different approach, treat your database as an **application database**

Application Databases

- An application database is only directly accessed by a single application, which makes it much **easier to maintain and evolve**
- Interoperability concerns can now shift to the interfaces of the application
  - During the 2000s we saw a shift to web services, where applications would communicate over HTTP
- You are able to use **richer data structures** (compared to SQL)
  - Usually represented as documents in XML or, more recently JSON

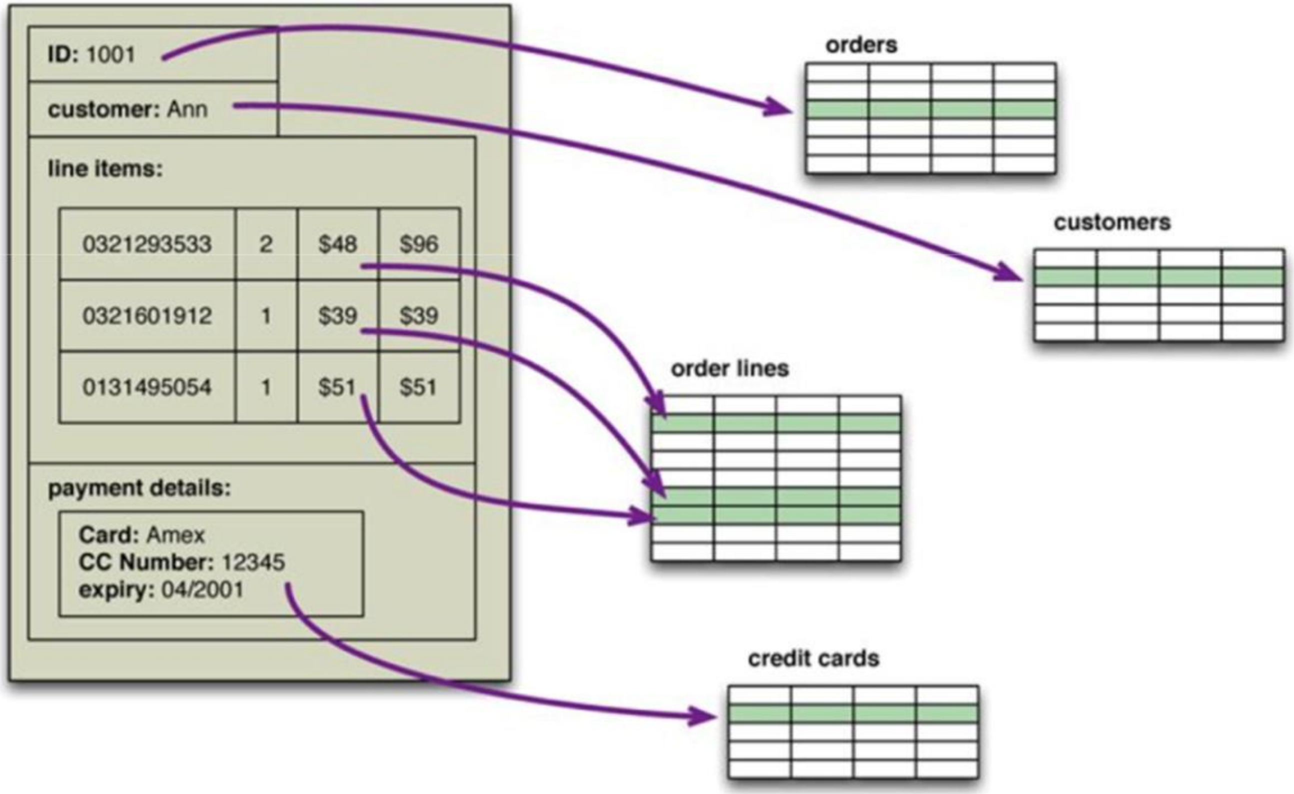
Aggregate data models

- The relational model divides the information that we want to store into **tuples** (rows): this is a very simple structure for data
- **Aggregate orientation** takes a different approach. It recognizes that often you want to operate on data in units that have a **more complex structure**
  - Think of it as a complex record that allows lists and other record structure to be nested inside
- Key-value, document and column-family DBs can all be seen as aggregate-oriented databases
  - They differ in how they structure the aggregate and consequently how they allow for it to be accessed

Aggregate data models

Aggregate: collection of related objects that we wish to treat as a unit

- Example: A purchase order



Aggregate data models: benefits

- Dealing with aggregates makes it much easier for the databases to handle **operating on a cluster**, since the aggregate makes a natural unit for replication.
  - Also a natural unit to use for distribution (all the data for an aggregate stored together in one node)
  - And the atomic unit for updates (transactional control)

- They reduce **the impedance mismatch problem**, i.e., the difference between the stored data and the in-memory data structures

From K-V to Documents: Structuring the value

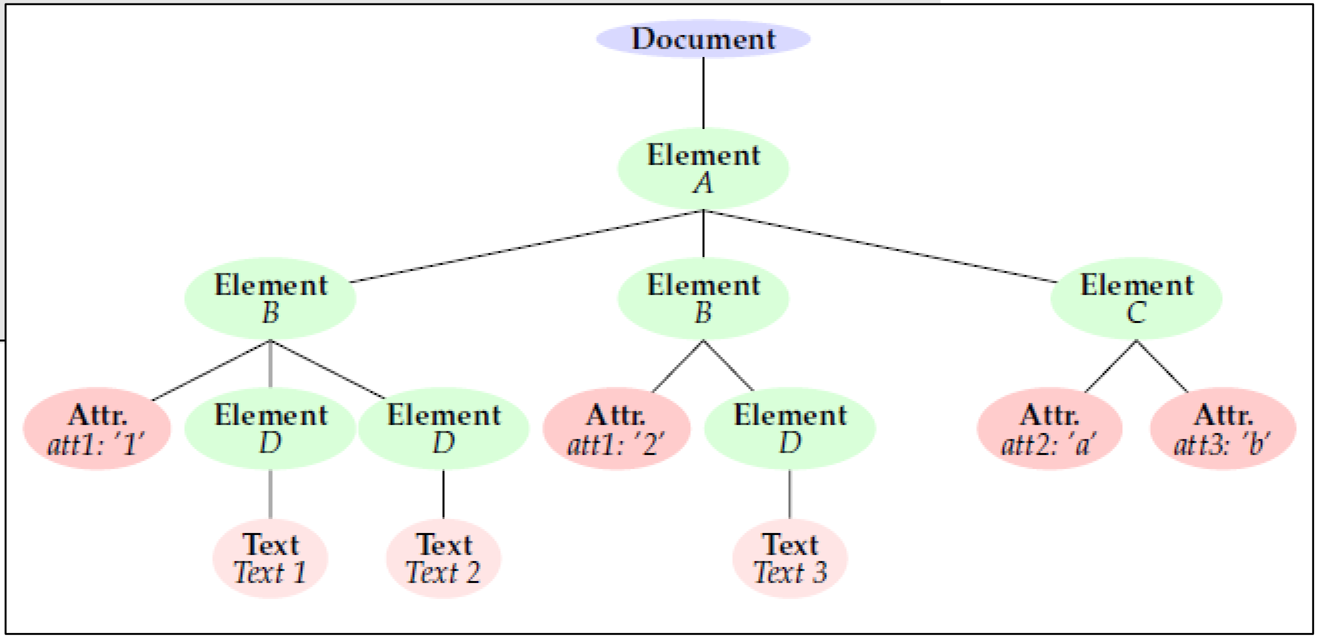
- Essentially, Document stores are Key-Value stores
  - Same design and architectural features
- The value is a document
  - XML (e.g., eXist)
  - JSON (e.g., MongoDB and CouchDB)
- Tightly related to the Web
  - Normally, they provide RESTful HTTP APIs
- So... what is the benefit of having documents?
  - New data model (collections and documents)
    - <u>New atom: from rows to documents</u>
  - Indexing

Types of Document Stores: XML

- XML is a semistructured data model proposed as the standard for data exchange on the Web
  - Can be represented as a tree
    - Document: the root node of the XML document, denoted by "/"
    - Element: element nodes that correspond to the tagged nodes in the document
    - Attribute: attribute nodes attached to Element nodes
    - Text: text nodes, i.e., untagged leaves of the XML tree
- Support Xpath, Xquery and XSLT
  - Xpath is a language for addressing portions of an XML document
    - Subset of XQuery
  - XQuery is a query language for extracting information from collections of XML documents
  - XSLT is a language for specifying transformations (from XML to XML)
- XML document stores
  - eXist, MarkLogic
    - Natively supported
  - XML extensions for Oracle, PostgreSQL, etc.
    - Mapped to relational (impedance mismatch!)

XML Example

```
<?xml version="1.0"
      encoding="utf-8"?>
<A>
  <B att1='1'>
    <D>Text 1</D>
    <D>Text 2</D>
  </B>
  <B att1='2'>
    <D>Text 3</D>
  </B>
  <C att2="a"
     att3="b"/>
</A>
```



An XML document is a *labeled*, *unranked*, *ordered* tree

Types of Document Stores: JSON

- JSON is a lightweight data interchange format
  - Brackets ([]) represent ordered lists
  - Curly braces ({}) represent key-value dictionaries
    - Keys must be strings, delimited by quotes (")
    - Values can be strings, numbers, booleans, lists, or key-value dictionaries
- Natively compatible with JavaScript
  - JSON stands for JavaScript Object Notation
  - Web browsers are natural clients for MongoDB / CouchDB
- JSON document stores
  - MongoDB, CouchDB
    - Natively supported
  - JSON extensions for Oracle, PostgreSQL, etc.
    - Mapped to relational (impedance mismatch!)

http://www.json.org

JSON Example

- Definition:
  - A document is an object represented with an unbounded nesting of array and object constructs

```
{
    "title": "The Social network",
    "year": "2010",
    "genre": "drama",
    "summary": "On a fall night in 2003, Harvard undergrad and computer
    programming genius Mark Zuckerberg sits down at his computer
    and heatedly begins working on a new idea. In a fury of blogging
    and programming, what begins in his dorm room soon becomes a global
    social network and a revolution in communication. A mere six years
    and 500 million friends later, Mark Zuckerberg is the youngest
    billionaire in history... but for this entrepreneur, success leads
    to both personal and legal complications.",
    "country": "USA",
  "director": {
      "last_name": "Fincher",
      "first_name": "David",
      "birth_date": "1962"
  },
  "actors": [
    {
        "first_name": "Jesse",
        "last_name": "Eisenberg",
        "birth_date": "1983",
        "role": "Mark Zuckerberg"
    },
    {
        "first_name": "Rooney",
        "last_name": "Mara",
        "birth_date": "1985",
        "role": "Erica Albright"
    },
    {
        "first_name": "Andrew",
        "last_name": "Garfield",
        "birth_date": "1983",
        "role": "  Eduardo Saverin "
    },
    {
        "first_name": "Justin",
        "last_name": "Timberlake",
        "birth_date": "1981",
        "role": "Sean Parker"
    }
  ]
}
```

MongoDB

An example of Document Store

MongoDB Data Model

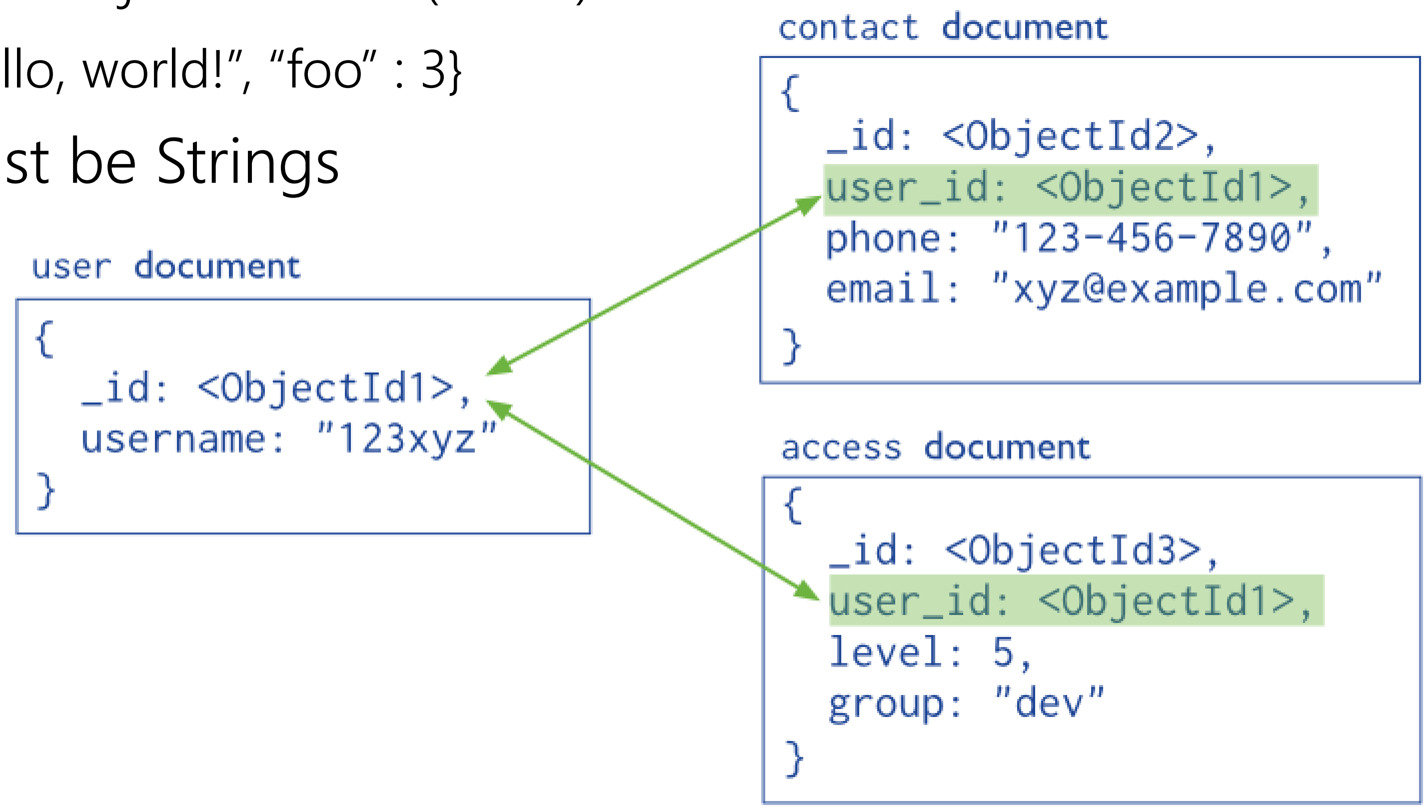
- Collections
  - Definition: A grouping of MongoDB documents
    - A collection exists within a single database
    - Collections **do not enforce a schema**
  - MongoDB Namespace: *database.collection*

- Documents
  - Definition: JSON documents (serialized as BSON)
    - Basic atom
    - Aggregated view of data
    - Identified by _*id* (user or system generated)
    - May contain
      - References (<u>NOT FKs!</u>)
      - Embedded documents

MongoDB Document Example

- Ordered set of keys with associated values
- Data structure:
  - Map, Hash, Dictionary or Object → JSON (BSON)

    e.g., {“greeting” : “Hello, world!”, “foo” : 3}
- Keys in a document must be Strings
  - No duplicate keys



MongoDB Document Example

- Plain document

```
{
  _id: <ObjectId1>,
  username: “123xyz”,
  contact_phone: “123-456-7890”,
  contact_email: “xyz@example.com”,
  access_level: 5
  access_group: “dev”
}
```

- Embedded sub-documents

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

Embedded sub-document

Embedded sub-document

- Array of sub-documents

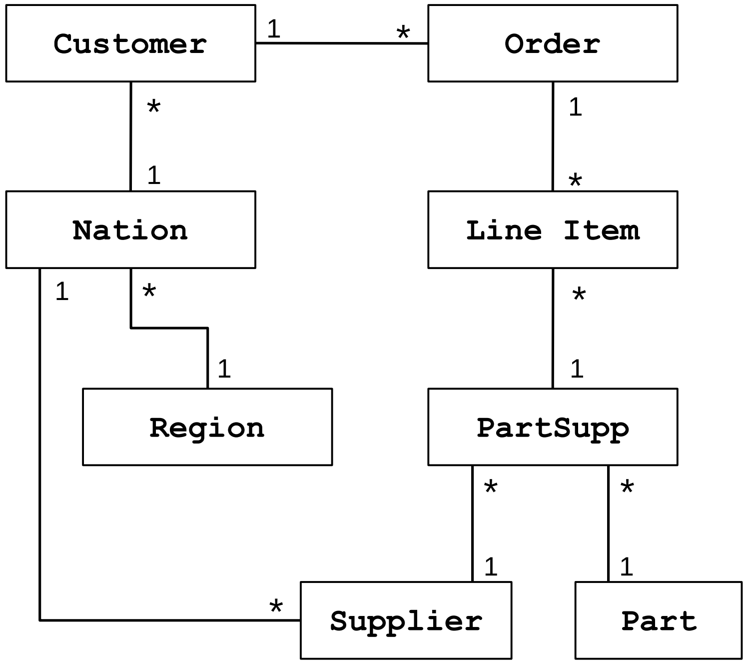
```
{
  _id: <ObjectId1>,
  username: “123xyz”,
  contacts: [
              { type: “work”, phone: “123-456-7890”, email: “xyz@example.com” },
              { type: “home”, phone: “098-765-4321”, email: “xyz@home.com” },
            ]
  access_level: 5
  access_group: “dev”
}
```

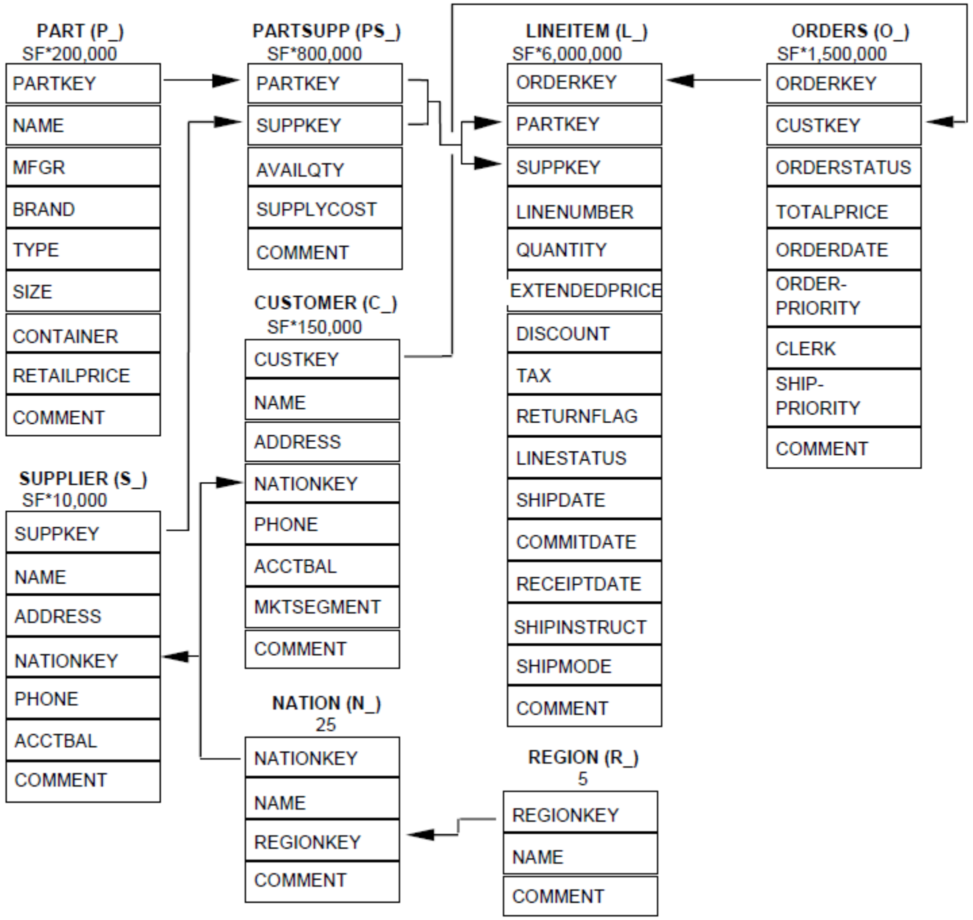
Designing Documents

- Follow one basic rule: 1 fetch for the whole data set at hand
  - Aggregate data model: check the data **needed by your application simultaneously** (queries)
    - <u>Do not think relational-wise!</u>
  - Use indexes to identify finer data granularities
- Consequences:
  - Independent documents
    - Avoid pointing at other docs
  - Massive denormalization
  - A change in the application layout might be dramatic
    - It may entail a massive rearrangement of the database documents

Activity: Modeling in MongoDB

- Objective: Learn how to model documents
- Tasks:
  1. (15’) Model the TPC-H database
  2. (5’) Discussion

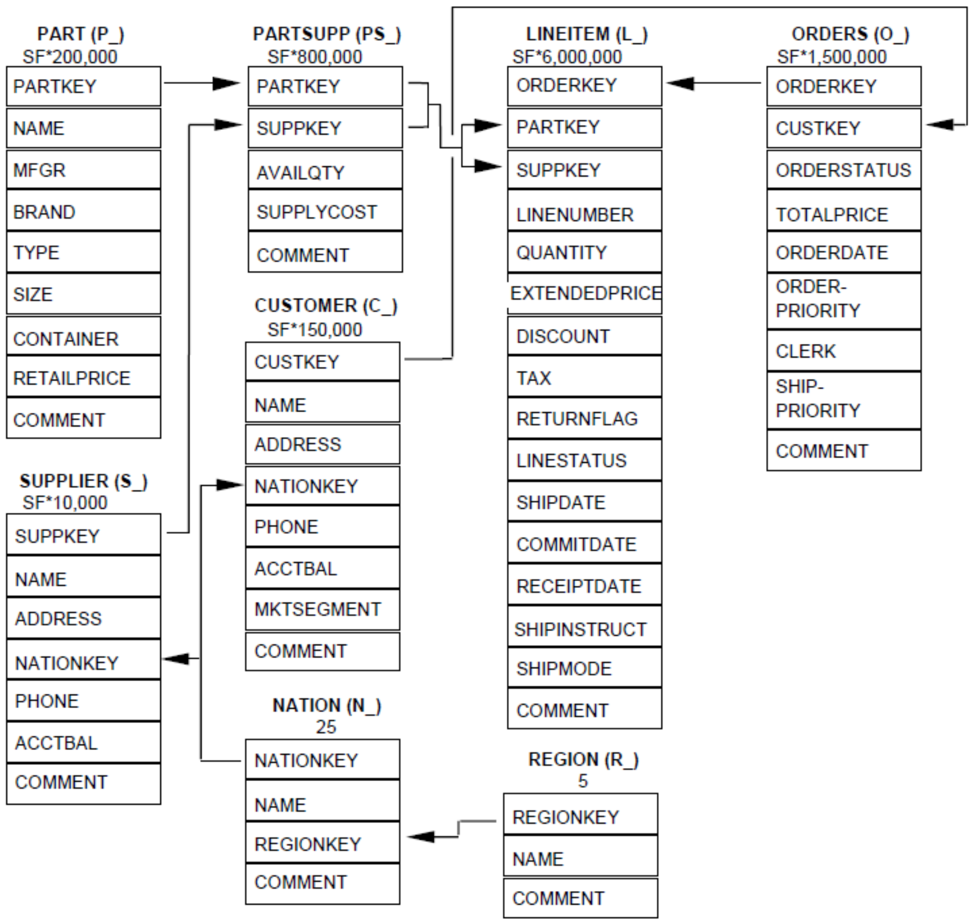




Activity: Modeling in MongoDB

- Objective: Learn how to model documents
- Tasks:
  1. (15’) Model the TPC-H database
     - According to the query below
  2. (5’) Discussion

```
SELECT l_orderkey,
sum(l_extendedprice*(1-l_discount)) as
revenue, o_orderdate, o_shippriority
FROM customer, orders, lineitem
WHERE c_mktsegment = '[SEGMENT]' AND
c_custkey = o_custkey AND l_orderkey =
o_orderkey AND o_orderdate < '[DATE]'
AND l_shipdate > '[DATE]'
GROUP BY l_orderkey, o_orderdate,
o_shippriority
ORDER BY revenue desc, o_orderdate;
```



MongoDB API

MongoDB Shell

- show dbs
- show collections
- show users
- use <*database*>
- coll = db.<*collection*>
- insert(*document*)
- save(*document*) -- updates an existing document or inserts a new document
- find(*query, projection*); -- `coll.find( {name:"Joe" }, { name: true } )`
- update(*query, update*);
- deleteOne(*query*) or deleteMany(*query*); -- removes one or many docs from a collection
- drop(); -- removes a collection from the database
- createIndex(*keys, options*); -- creates an index on the specified fields

- Notes:
  - *db* refers to the current database
  - *query* is a document (query-by-example)

https://www.mongodb.com/docs/mongodb-shell/

MongoDB: Syntax

Global variable

Query-by-example (Depending on the method: document, array of documents, etc.)

```
db.[collection-name].[method]([query],[options])
```

- Collection methods: insert, update, remove, find, …

```
db.restaurants.find({"name": "x"})
```

- Cursor methods: forEach, hasNext, count, sort, skip, size, ...

```
db.restaurants.find({"name": "x"}).count()
```

- Database methods: createCollection, copyDatabase, ...

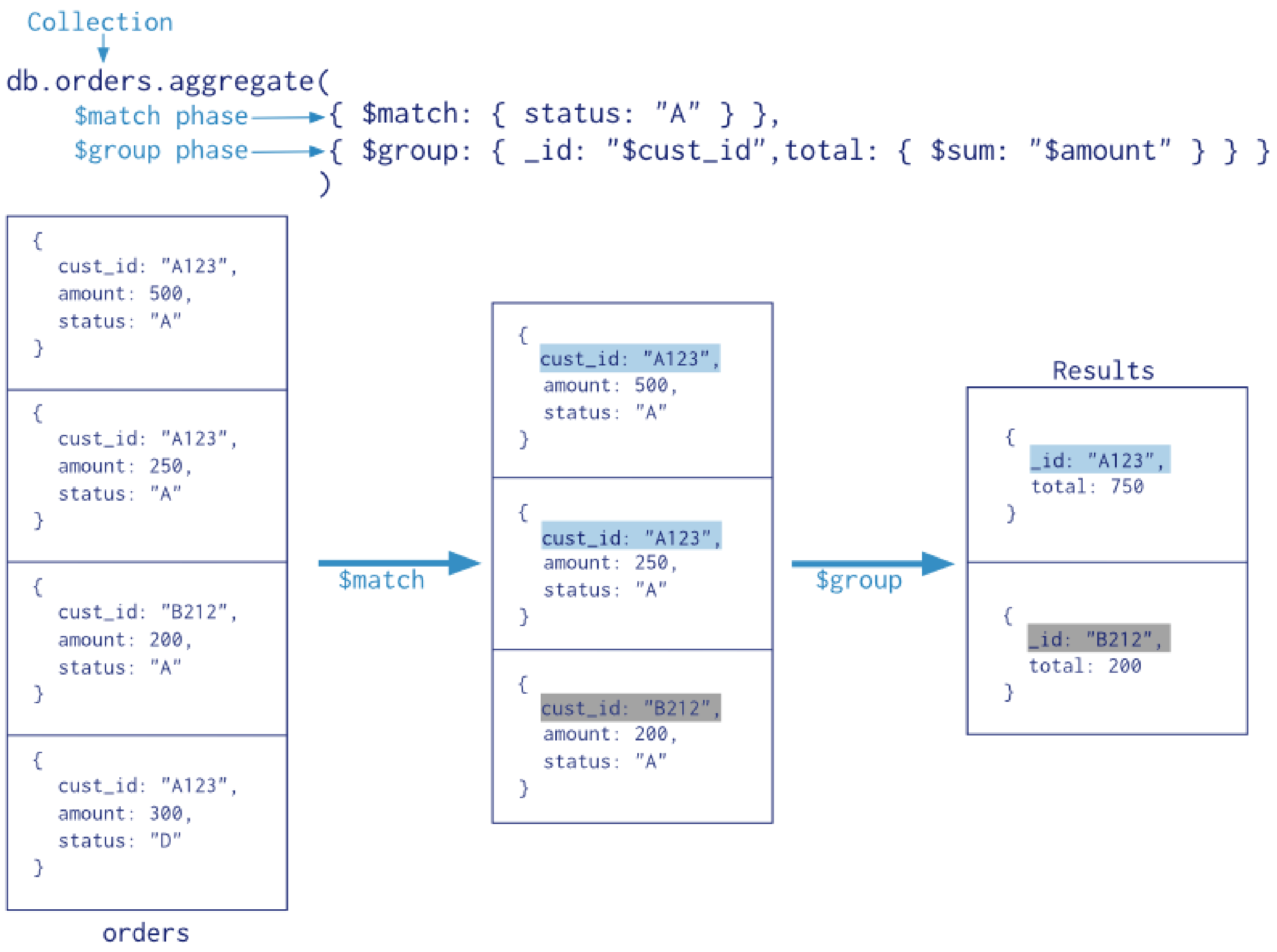
```
db.createCollection("collection-name")
```

- …

MongoDB: Querying

- Find and findOne methods
  - database.collection.find()
  - database.collection.find({ qty: { $gt: 25 } })
  - database.collection.find({ field: { $gt: value1, $lt: value2 } });
- The Aggregation Framework
  - An aggregation pipeline
  - Documents enter a multi-stage pipeline that transforms the documents into an aggregated result
    - Filters that operate like queries
    - Document transformations that modify the form of the output
    - Grouping
    - Sorting
    - Other operations
- MapReduce (deprecated)

MongoDB: Aggregation Framework



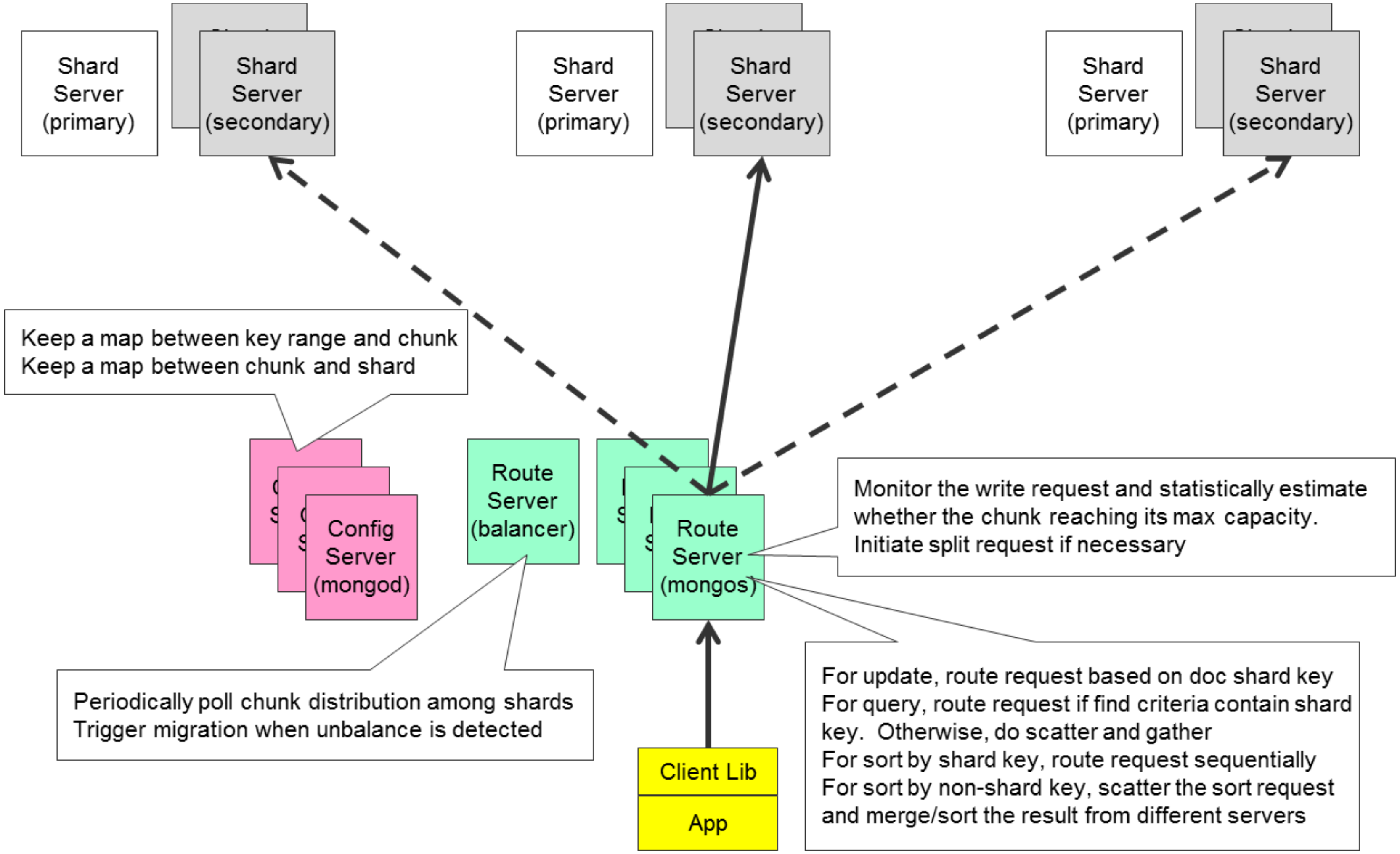
https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/

MongoDB Architecture

MongoDB Notation: Shard Clusters

- Shards
  - Nodes containing data
  - A shard may contain several chunks
- Config Servers
  - Nodes containing the global catalog (e.g., hash directory)
- Query routers or Route servers
  - Nodes containing a copy of the hash directory to redirect queries

MongoDB Architecture



http://horicky.blogspot.com.es/2012/04/mongodb-architecture.html

Shard Clusters Management

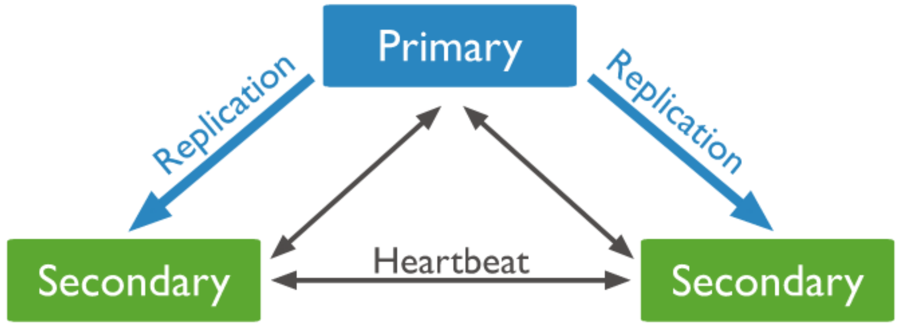
- Query routers are replicas of the config servers
  - Secondary versioning (config servers)
  - Eager replication (to both config servers and query routers)
    - 2PCP (potential distributed deadlocks!)
- Config Servers
  - The hash directory is mandatorily replicated to avoid single-point failures
    - MongoDB asks for 3 config servers
  - Writes happen if:
    - A shard splits
    - A chunk migrates between servers (e.g., adding servers)
- Query routers
  - Read from config servers
    - When they start (o restart)
    - Every time a split / migration happens

Splitting/Migrating Chunks

- Default chunk size: 64MB
- The query router (mongos) asks a shard to split
  - Inserts and updates trigger splits
- Shards rearrange the data (data migration)
  - During the migration, requests to that chunk address the origin shard
  - Changes made during the migration are afterwards applied in the destination shard
  - Finally, changes in the hash directory are made in the config servers
    - Query routers eagerly synchronized
- A *balancer* avoids uneven distributions

Replication

- Each shard (in a shard cluster) is a replica set
  - Maps to a mongod instance (with its config servers)
- Replica Set: Master versioning with lazy replication
  - One master
    - Write / Update / Delete
  - Several replicas
    - Reads
- Replica Set management
  - The master has a recovery system (journaling): WAL
  - Members interconnected by heartbeats
  - If the master fails, voting phase to decide a new master
  - If a replica fails, it catches up with the master once back



Pluggable Storage Engine Architecture

- MongoDB 3.0 introduced the concept of "pluggable storage engine"
  - MMAP V1 – based on consistent hashing (see previous slides)
  - WiredTiger – based on LSM (distributed B+)
  - In-Memory
- WiredTiger
  - Moves to LSM (welcome range queries!)
  - We can choose if store data row-oriented or column-oriented
    - First boosts writes, hurts reads. Just the opposite for the second one
  - First-class citizen compression
- Each node on a MongoDB cluster can use a different storage engine

Query Optimization

- The aggregation framework creates a left-deep tree access plan and applies pipelining
  - Note that the first operation is executed in parallel in all nodes contaning data (exploiting data locality)
  - From there on, a node takes care of the query and data is shipped to it to execute the rest of the pipeline
- MongoDB barely applies any optimization technique in its querying flow
  - First versions: Nothing!
  - From version 2.6: Primitive rule-based optimization approach https://www.mongodb.com/docs/manual/core/aggregation-pipeline-optimization/
- Be careful when creating your pipes (you are the most important optimizer!)
  - Push selections and projections to the beginning of the pipeline
  - A cost-based approach badly needed…

https://www.mongodb.com/docs/manual/core/query-optimization/

Indexing

- Indexes are (physically) the same as in a relational database. Same rules apply:
  - Selective queries
  - Must fit in memory
- However, indexing management is way poorer
  - No cost-based models
  - For a new query, all indexes are run in parallel and the best plan is chosen from there on (sigh)
  - The plan is recalculated when massive inserting happens or when the database restarts
- Better you do the job
  - Monitor your queries:
    - https://www.mongodb.com/docs/manual/tutorial/analyze-query-plan/
  - Use $hint to force MongoDB choose an index
    - https://www.mongodb.com/docs/manual/reference/method/cursor.hint/

Limitations (I)

- Architectural Issues
  - Thumb rule: 70% of the database must fit in memory
  - Be careful with updates! (padding)
    - Holes caused by reallocation
    - Compact the database from time to time
    - In WiredTiger this is left for the compactation (the delta memstore smooths it)
  - Limited number of collections per database
  - Theoretically, sharding is automatic and transparent. But in practice it is not. Most typical ones:
    - Max. number of elements to migrate (when balancing the workload)
    - LSM + sequential keys will hit only one node (be careful with the key!!)
- Document Issues
  - The resulting document of an aggregation pipeline cannot exceed the maximum document size (16Mb)
    - GridFS for larger documents
  - No more than 100 nesting levels (i.e., embedded documents nesting)
  - Attribute names are kept as they are (no catalog)

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

Limitations (II)

- Querying Issues
  - Limited transaction support
    - ACID guarantees only at document level
    - Strong / loose consistency parametrizable (write-concern) https://www.mongodb.com/docs/manual/reference/write-concern/
  - Thumb rule: A query must attack a single collection
  - Arrays are a mess! → $unwind
  - No optimizer!
  - Be careful with distributed queries (parallelism not really exploited): https://www.mongodb.com/docs/manual/core/distributed-queries/
- The aggregation framework
  - Pipe stages are limited to 100MB of RAM: https://www.mongodb.com/docs/manual/core/aggregation-pipeline-limits/
    - Disk usage for bigger pipeline operators must be specified
      - Performance deteriorates hugely!
      - Does not solve the problem in many cases
- Indexing
  - MongoDB runs all the queries with all indexing possibilities in parallel: https://www.mongodb.com/docs/manual/core/query-plans/
  - From there on, that index will be always chosen (!!!)
  - Check its performance with:
    - Add .explain() to see the access plan
    - Nice explanation here: https://www.compose.io/articles/explain-explain-understanding-mongo-query-behavior/

MongoDB: Conclusions

- MongoDB has its limitations, but it is one of the most supported and mature NOSQL tools
  - Still, its robustness is far away from a relational database
- Many tools and functionalities available
  - Managing and Monitoring
    - OpsManager: Be careful! The terms of use say your data is periodically sent to MongoDB (the Company)
    - db.collection.explain()
  - GUI: MongoDB Compass
  - Supporting GeoSpatial data and queries
  - Support from 3rd parties
    - Tableaux
    - Pentaho BI Suite
    - Cubes: OLAP-lightweight Engine (http://cubes.databrewery.org/)
    - Good pluggability with almost any language (Python, Ruby, Perl, Java, Scala, PHP, etc.)
  - Most Cloud providers offer MongoDB as a service
    - Amazon, DigitalOcean, Rackspace, Openshift, Azure, etc.
    - Compose (MongoDB as a Service) + Heroku (great combo!)

Summary

- Document stores
  - Semi-structured value
  - Indexing
- Designing document stores

Bibliography

- S. Abiteboul et al. Web Data Management, 2012
- J. Dittrich et al. Hadoop++: Making a yellow elefant run like a cheetah (without it even noticing)
- D. Jiang et al. The performance of MapReduce: An In-depth Study. VLDB'10
- E. Brewer, "Towards Robust Distributed Systems," Proc. 19th Ann. ACM Symp.Principles of Distributed Computing (PODC 00), ACM, 2000, pp. 7-10.
- F. Chang et all. Bigtable: A Distributed Storage System for Structured Data. OSDI'06
- Sanjay Ghemawat et al. The Google File System. OSDI'03
- Jeffrey Dean et al. MapReduce: Simplified Data Processing on Large Clusters. OSDI'04
- D. Battre et al. Nephele/PACTs: A Programming Model andExecution Framework forWebScale Analytical Processing. SoCC'10
- L. Liu and M.T. Özsu (Eds.). Encyclopedia of Database Systems. Springer, 2009
- P. Sadalge, M. Fowler. NoSQL Distilled. Addison Wesley, 2012.