Document Stores

bbilalli@essi.upc.edu

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

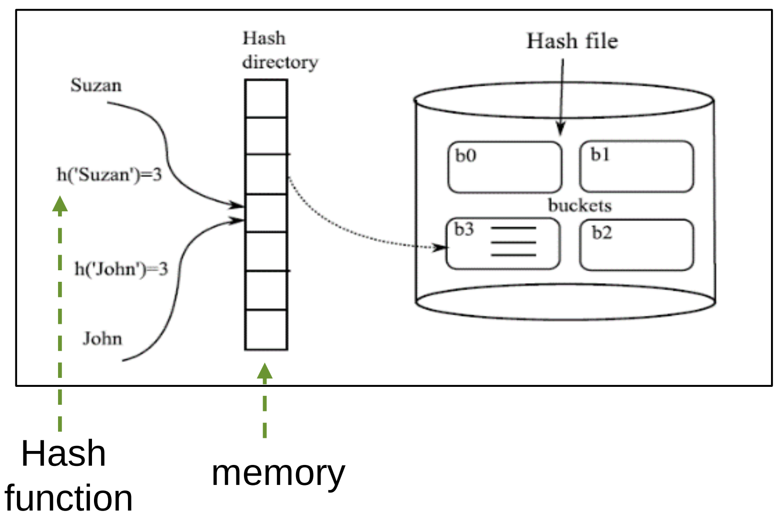
CONSISTENT HASHING

DISTRIBUTED ARCHITECTURES

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 queries & nearest neighbor
  - Does not adapt easily to **dynamic collections** (grow and shrink rapidly)



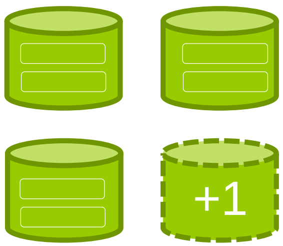
- What if the data grows too much?
  - Distribute the hash structure

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*

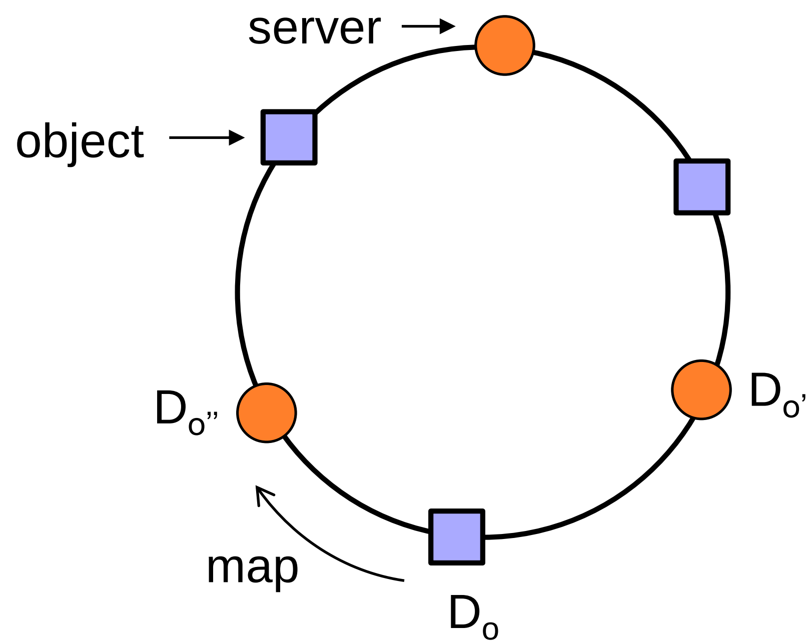
    E.g., h(x) = x % #servers



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

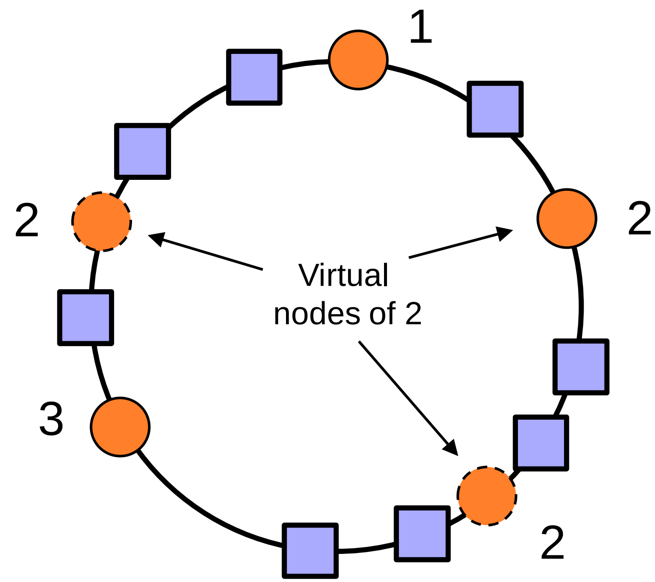
Consistent Hashing

- 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
- Coping with dynamicity:
  - The hash function **never** changes
    - Choose a very large domain *D* (address space) and map server IP addresses and object keys to such domain
    - Organize *D* as a ring in clockwise order so each node has a successor
    - Objects are assigned as follows:
      - For an object O, $f(O) = D_o$
      - Let $D_{o'}$ and $D_{o''}$ be the 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



- Coping with dynamicity:
  - The hash function **never** changes
    - Choose a very large domain *D* (address space) and map server IP addresses and object keys to such domain
    - Organize *D* as a ring in clockwise order so each node has a successor
    - Objects are assigned as follows:
      - For an object O, $f(O) = D_o$
      - Let $D_{o'}$ and $D_{o''}$ be the 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

Consistent Hashing: Example



- Further refinements:
  - Assign to the same server several hash values (virtual servers) to balance load
  - Lazy update of the hash directory

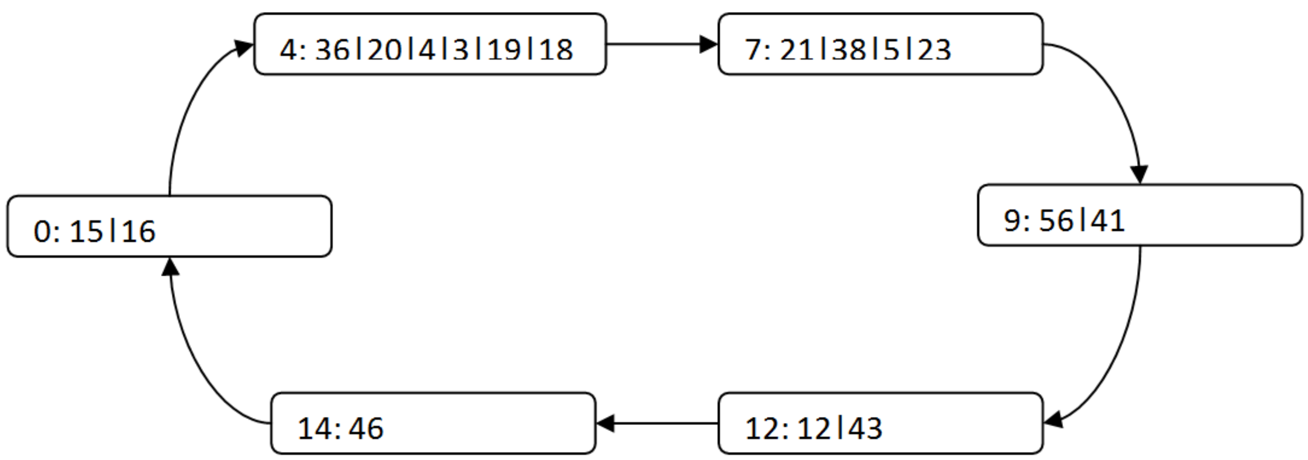
Distributed Hashing in Practice

- Most current key-value (and document-stores) use distributed hashing
- Consistent Hashing
  - Memcached / CouchDB
  - MongoDB
  - Cassandra
  - Dynamo / SimpleDB
  - Voldemort

Activity: Consistent Hashing

- *Objective: Understand how the consistent hash 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

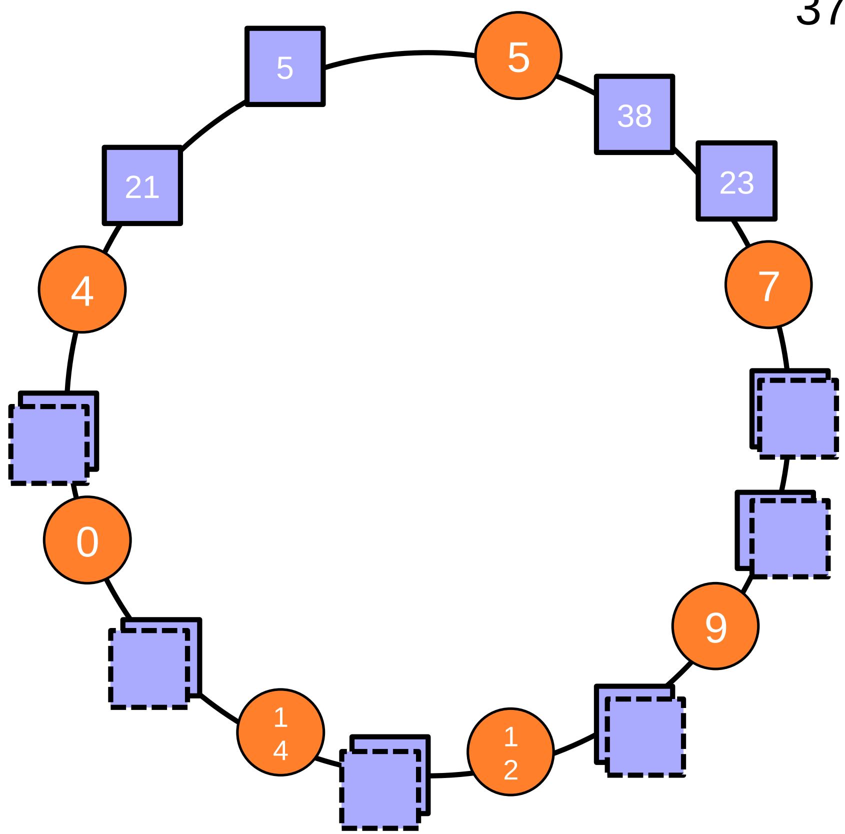


Current state of the consistent hash (D=16)

Activity: Consistent Hashing

- *Objective: Understand how the consistent hash 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*

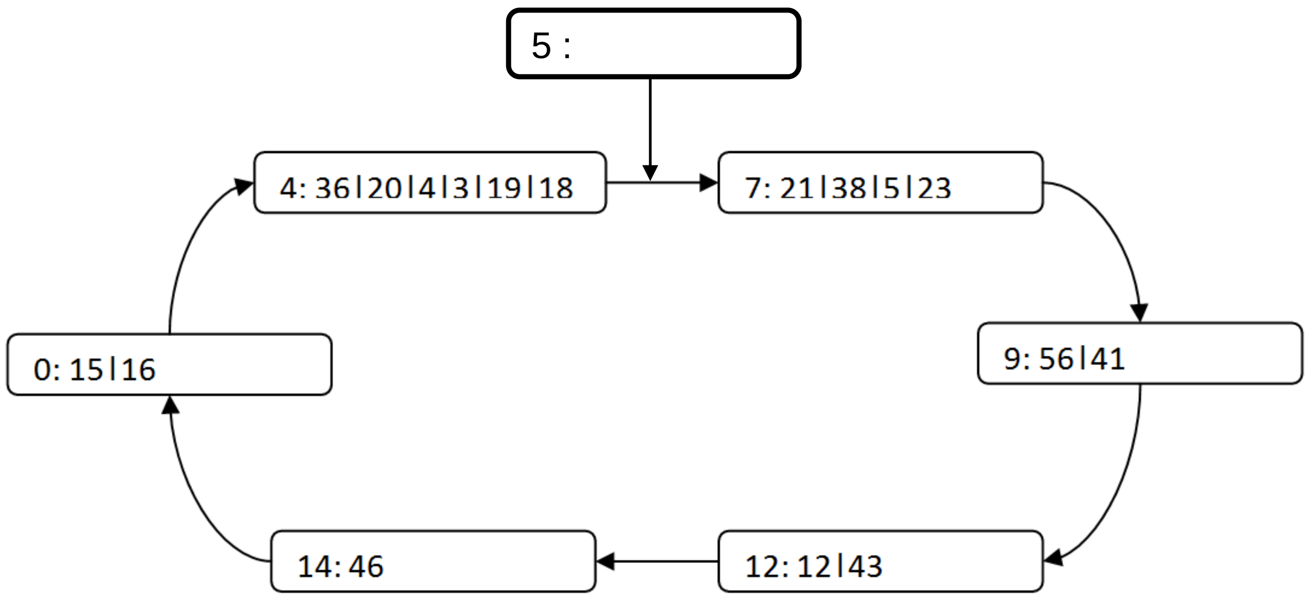
37 % 16 = 5



Activity: Consistent Hashing

- *Objective: Understand how the consistent hash 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*

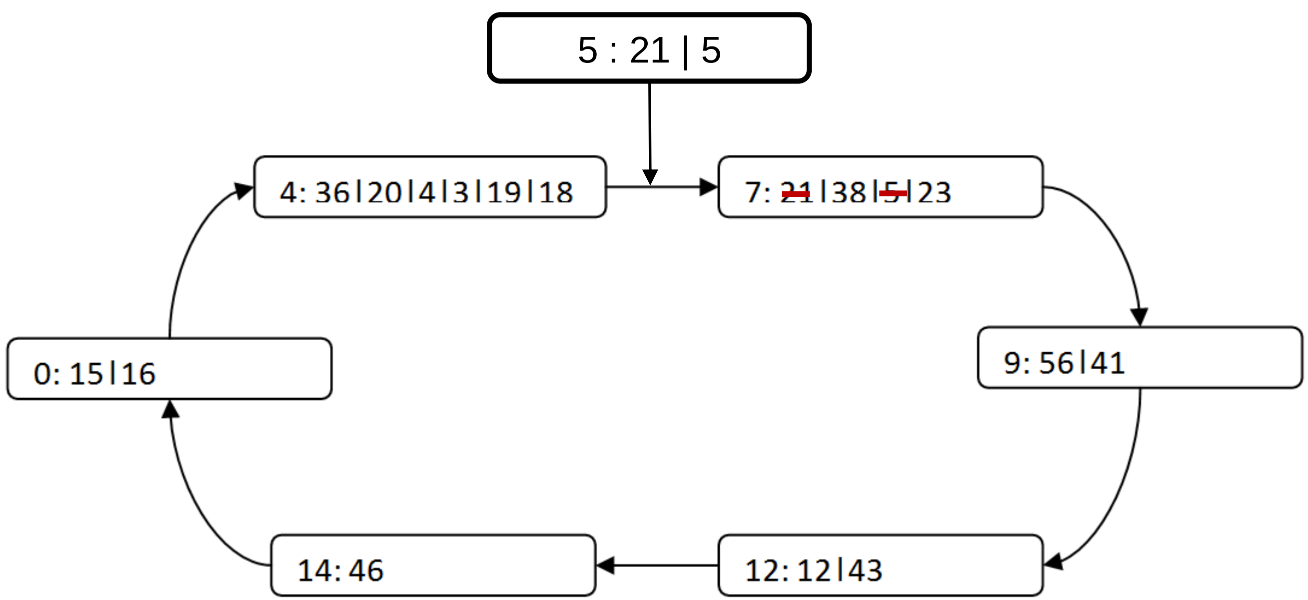
f(IP) = IP % D



Activity: Consistent Hashing

- *Objective: Understand how the consistent hash 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

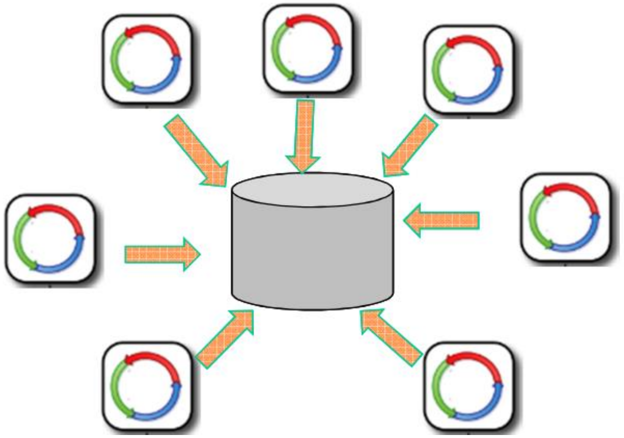


KEY-VALUE ENHANCEMENTS

DOCUMENT-ORIENTED DBS

Application databases (1)

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

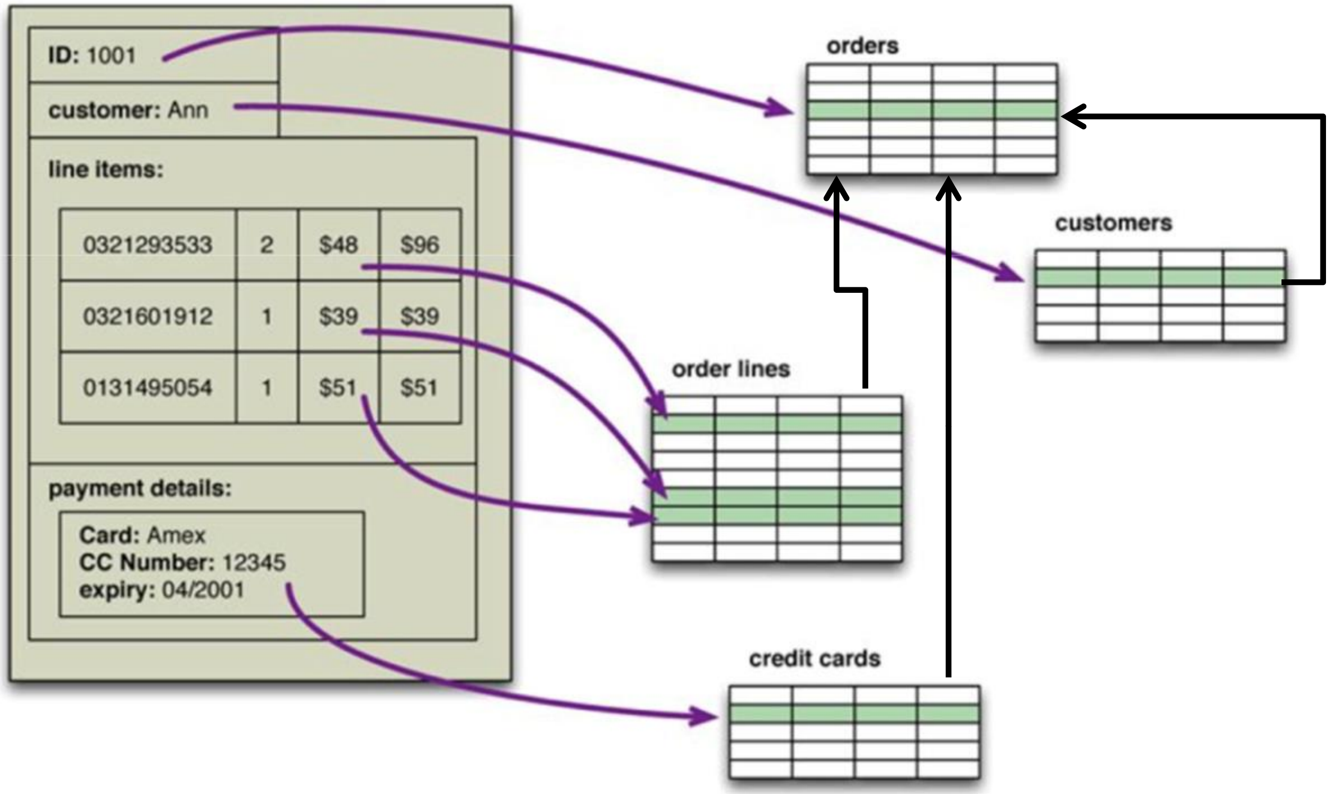
Application databases (2)

- 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
- With a service you are able to use **richer data structures** (compared to SQL)
  - Usually represented as documents in XML or, more recently JSON

Aggregate data models (1)

- 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 (3)



An order, which looks like a single *aggregate*

Aggregate data models (2)

- What is good about these models?
  - 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 and sharding.
    - Also a natural unit to use for distribution (all the data for an aggregate stored together in one node)
  - Also, it may help **solving the impedance mismatch problem**, i.e., the difference between the relational model and the in-memory data structures
    - The impedance mismatch is naturally solved in document-stores

Structuring the Value

- Essentially, they 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)
    - **New atom: from rows to documents**
  - Indexing

Types of Document-Stores

- JSON-like documents
  - MongoDB
  - CouchDB
- 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
  - Web browsers are natural clients for MongoDB / CouchDB

  http://www.json.org/index.html

JSON Example

- Definition:

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

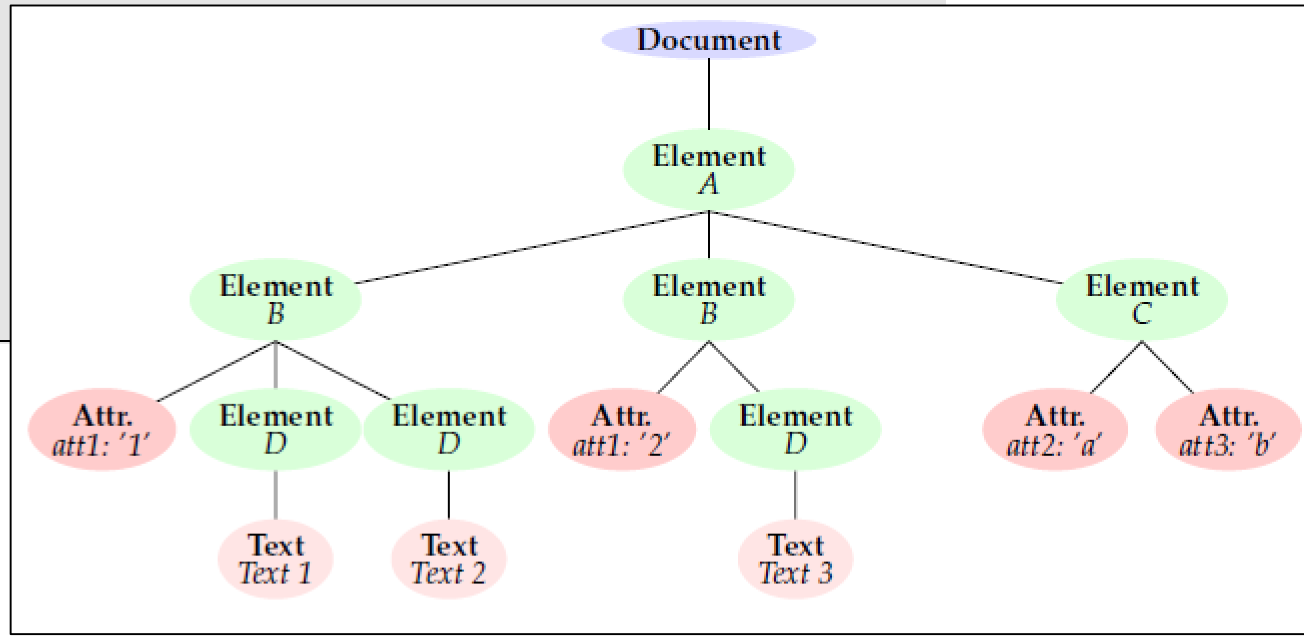

```
{
    "title": "The Social network",
    "year": "2010",
    "genre": "drama",
    "summary": "On a fall night in 2003, Harvard undergrad and computer
    programming genius Mark Zuckerberg sits down at his computer
    and heatedly begins working on a new idea. In a fury of blogging
    and programming, 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"
    }
  ]
}
```

Type of Document-Stores

- XML-like documents
  - eXist, MarkLogic
    - Natively supported
  - Relational extensions for Oracle, PostgreSQL, etc.
    - Mapped to relational (impedance mismatch!)
- XML is a semistructured data model proposed as the standard for data exchange on the Web
  - Can be elegantly 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 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

An Example of Document-Store

MONGODB

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
    - Identified by *_id* (user or system generated)
    - Aggregated view of data
    - May contain
      - References (**NOT FKs!**) and
      - Embedded documents

MongoDB: Document Example (1)

- 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

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"
}
```

ObjectIds (_id)

- Unique value representing the key of the document
- 12 bytes

0	1	2	3	4	5	6	7	8	9	10	11
Timestamp				Machine			PID		Increment		

MongoDB: Document Example (2)

- Embedded documents
  - Insert document as sub-doc.

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

Embedded sub-document

Embedded sub-document

- Move attrs to the root document
  - All data directly in one doc.
  - Simpler document structure

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

MongoDB: Document Example (3)

- Array of nested documents
  - Many sub documents related to root
  - JSON array of documents

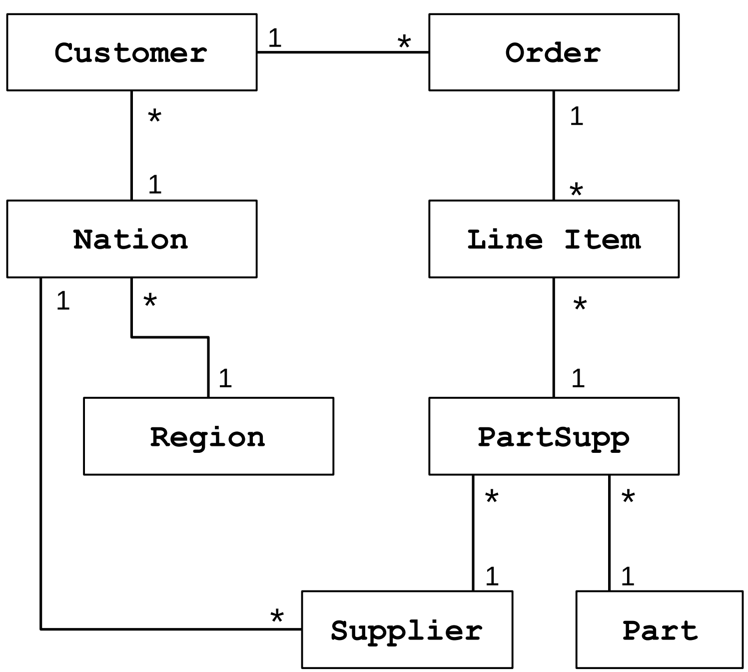
```
{
  _id: <ObjectId1>,
  username: “123xyz”,
  contacts: [
              { type: “work”, phone: “123-456-7890”, email: “xyz@example.com” },
              { type: “home”, phone: “098-765-4321”, email: “xyz@home.com” },
            ]
  access_level: 5
  access_group: “dev”
}
```

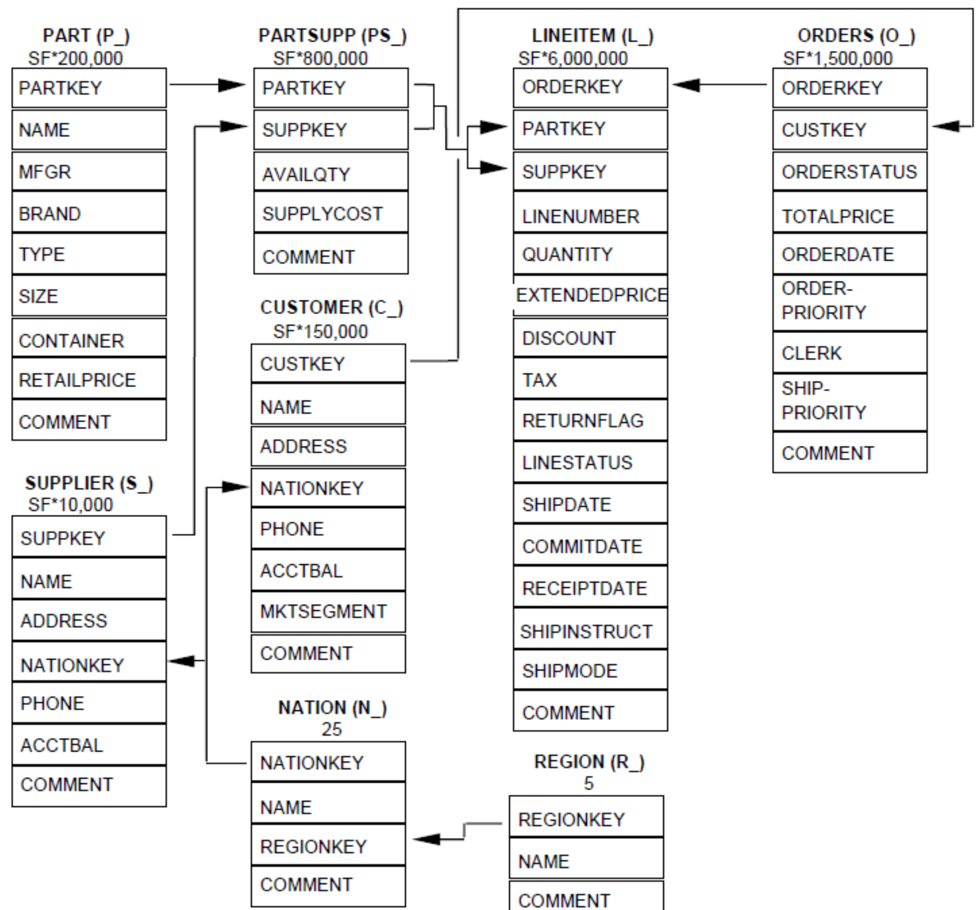
Designing Document Stores

- Follow one basic rule: 1 fetch for the whole data set at hand
  - Aggregate data model: check the data needed by your application simultaneously
    - **Do not think relational-wise!**
  - Use indexes to identify finer data granularities
- Consequences:
  - Independent documents
    - Avoid pointing FKs (i.e., 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 (I)

- *Objective: Learn how to model documents*
- *Tasks:*
  1. *(15') Model the TPC-H database in a normalized way*
     1. *A table is a collection*
     2. *An instance is a document*
  2. *(5') Discussion*





Activity: Modeling in MongoDB (I) - discussion

- Using the RDBMS notation

Customer

<u>CustKey</u>	NatKey	Name	Phone
1	4	Fisnik	234

Orders

<u>OrderKey</u>	CustKey	Status	Price
15	1	Active	50

PartSupp

<u>PartKey</u>	<u>SuppKey</u>	Qty	Cost
8	10	100	45

Part

<u>PartKey</u>	Name	Brand	RetPrice
8	Shoe	Nike	40

Supplier

<u>SuppKey</u>	NatKey	Name	Phone
10	4	AVA	12-345

Nation

<u>NatKey</u>	RegKey	Name
4	1	Spain

Region

<u>RegKey</u>	Name
1	EU

LineItem

<u>OrderKey</u>	<u>LineNum</u>	PartKey	SuppKey	Qty
15	1	8	10	1

Acticity: Modeling in MongoDB (II) - discussion

- Potential models

```
//in customers
{
"id":1,
"nation":[{"name":Spain,"region":"EU"}]
"name":"Fisnik",
"phone":"234"
}
```

```
//in orders
{
"id":15,
"custkey":1,
"lineItems": [
{
  "lineNumber": 1,
  //product
  "name":"shoe",
  "brand":"nike"
  "qty": 1,
  //supplier
   …
     //nation
      …
}
],
"status":"active",
"totalPrice": 50,
}
```

Acticity: Modeling in MongoDB (II) - discussion

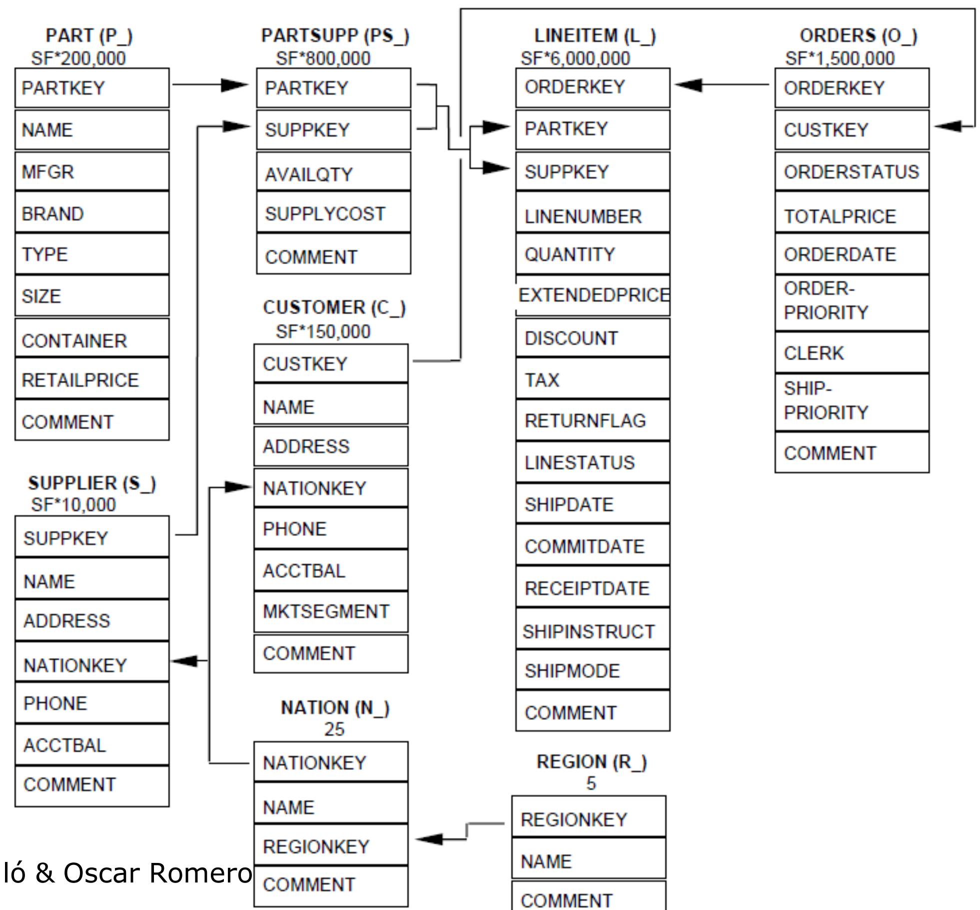
```
//in customers
{
"id":1,
"nation":[{"name":Spain,"region":"EU"}]
"name":"Fisnik",
"phone":"234"
}
"orders": [
{
 "id":15,
 "custkey":1,
 "lineItems": [
 {
   "lineNumber": 1,
   //product
   "name":"shoe",
   "brand":"nike"
   "qty": 1,
   //supplier
    …
      //nation
      …
  }
 ],
 "status":"active",
 "totalPrice": 50,
 ]
 }
}
```

```
//in supplier
{
"id":1,
"nation":[{"name":Spain,"region":"EU"}]
"name":"AVA",
"phone":"12-345"
}
 "products":[
  {
   //product
   "name":"shoe",
   "brand":"nike"
   "qty": 1,
   "supplyCost": 45,
   …
  }
 ]
}
```

Activity: Modeling in MongoDB (II)

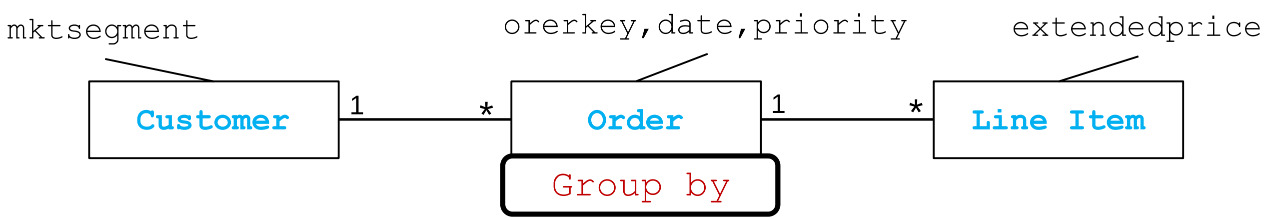
- *Objective: Learn how to model documents*
- *Tasks:*
  1. *(15') Model the TPC-H database according to the query below*
  2. *(5') Discussion*

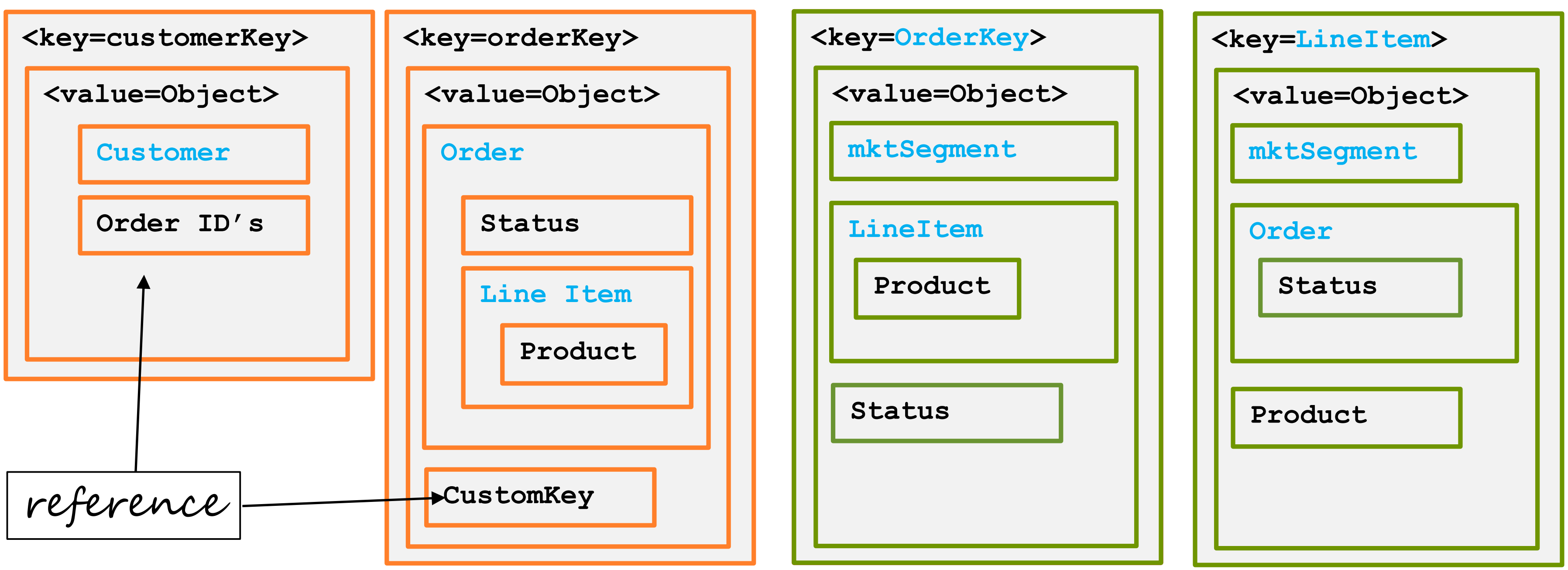
```
SELECT l_orderkey,
sum(l_extendedprice*(1-
l_discount)) as revenue,
o_orderdate, o_shippriority
FROM customer, orders, lineitem
WHERE c_mktsegment = '[SEGMENT]'
AND c_custkey = o_custkey AND
l_orderkey = o_orderkey AND
o_orderdate < '[DATE]' AND
l_shipdate > '[DATE]'
GROUP BY l_orderkey,
o_orderdate,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



Activity: Modeling in MongoDB (II) - discussion

```
SELECT l_orderkey, sum(l_extendedprice*(1-l_discount)) as revenue, o_orderdate, o.priority
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;
```





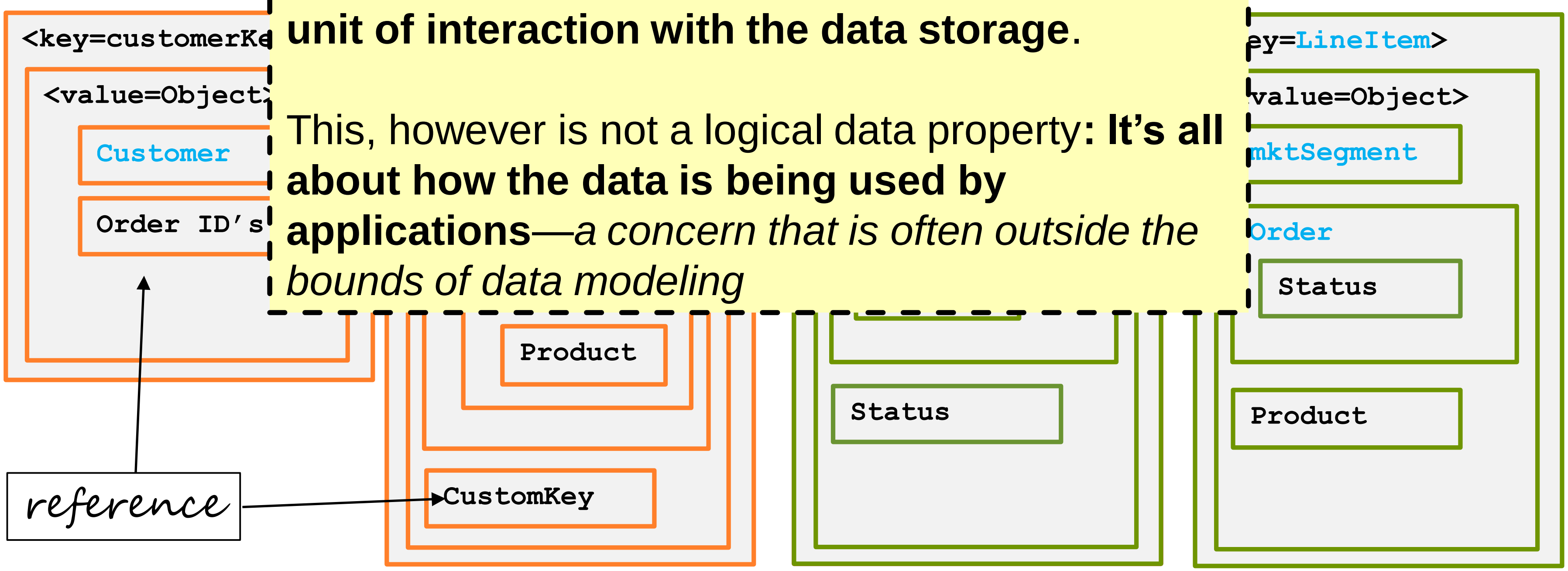
Activity: Modeling in MongoDB (II) - discussion

```
SELECT l_orderkey, sum(l_extendedprice*(1-l_discount)) as revenue, o_orderdate, o.priority
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;
```

Like most things in modeling, there’s no universal answer for how to draw your aggregate boundaries.

Yet, we have a semantics to consider: focus on **the unit of interaction with the data storage**.

This, however is not a logical data property**: It’s all about how the data is being used by applications**—*a concern that is often outside the bounds of data modeling*



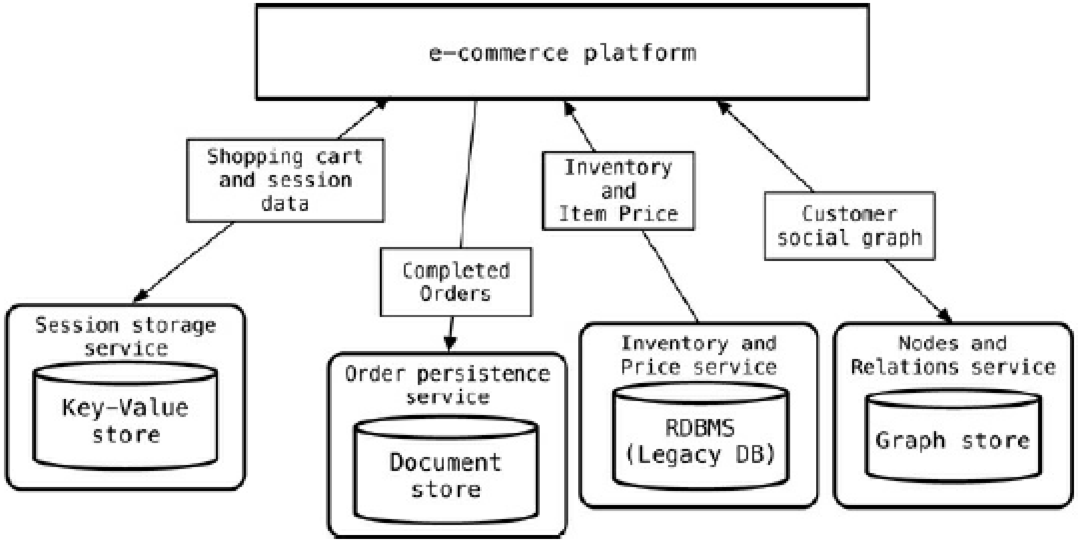
Consequences of aggregate modeling

- Aggregates help greatly with running on a cluster
  - **put together data that is commonly accessed together**.

  -> minimize the # of nodes to access when gathering the data.
  - But there are still lots of cases where data that's related *is accessed differently*.
    - What if we want to process orders individually?
    - What if a retailer wants to analyze its product sales over the last few months?
- The database is ignorant of the relationships (embedded IDs) in data
  - Atomicity is only supported within the contents of a **single aggregate**
  - No ACID guarantees while altering many aggregates
    - If you update multiple aggregates at once, you have to deal yourself with a failure
  - Thus, if you have data based on lots of relationships, you should prefer a relational database, or even better a graph database.

Alternative modeling

- Frequent (analytics) queries are used 60% of the time [1]
  - E.g., the GUI can be used to limit the interaction
- Polyglot persistence
  - Different databases are designed to solve different problems.
  - Using a single database engine for all of the requirements usually leads to non- performant solutions;
  - Increases complexity in programming and operations



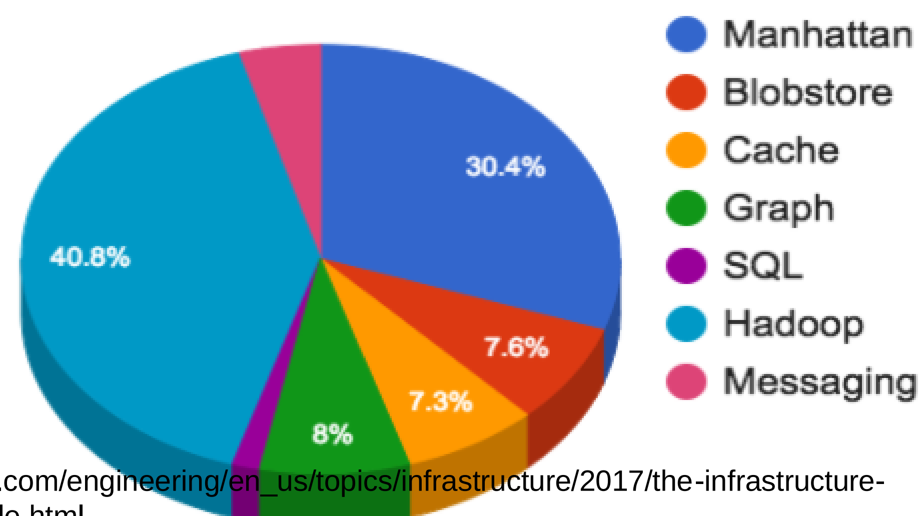
[1] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin,K. Karanasos, and S. Rao. “Computation Reuse in Analytics Job Service at Microsoft”. In: SIGMOD. 2018

Altern

- Frequ ... me [1]
  - E.g
- Polygl
  - Dif
  - Us ... ually leads to
  - Inc

The storage and messaging teams provide the following services:

1. Hadoop clusters running both compute and HDFS
2. Manhattan clusters for all our low latency key value stores
3. Graph stores sharded MySQL clusters
4. Blobstore clusters for all our large objects (videos, pictures, binary files…)
5. Cache clusters
6. Messaging clusters
7. Relational stores (MySQL, PostgreSQL and Vertica)



https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html

[1] A. Jinda ... and S. Rao. "Computation Reuse in Analytics Job Service at Microsoft". In: SIGMOD. 2018

MongoDB Shell (1)

- `mongo` is a Javascript shell
  - Allows to connect to a MongoDB instance (interactive or .js scripts)

```
$ mongo some-host:30000/myDB
MongoDB shell version: 2.4.0
connecting to: some-host:30000/myDB
>
```

  - You can rely on Javascript documentation
  - + MongoDB specific functionality

```
> help
    db.help()                    help on db methods
    db.mycoll.help()             help on collection methods
    sh.help()                    sharding helpers
    ...

    show dbs                     show database names
    show collections             show collections in current database
    show users                   show users in current database
    ...
```

MongoDB Shell (2)

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

http://docs.mongodb.org/manual/reference/mongo-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: Insert

- Insert will automatically add an `id` key to the document (if one does not already exist)

document or array of documents

```
db.users.insert({"name":"Sergi"})
```

```
insertMany(
[{"name":"Sergi"},
{"name":"Victor"},
{"name":"Clara"}])
```

- Recommended: create your own ID before inserting documents
  - Should not rely on MongDB IDs
  - Just add a field "_id" to inserted documents
- MongoDB checks that the document does not exceed 16MB

Documentation: https://docs.mongodb.org/manual/tutorial/insert-documents/

MongoDB: Delete

<query>
document

- Removing by example

```
db.users.remove({"name":"Sergi"})
```

- Remove all documents (collection and indexes remain intact)

```
db.users.remove()
```

- Dropping a collection

```
db.users.remove("users")
```

Documentation: https://docs.mongodb.org/manual/tutorial/remove-documents/

MongoDB: Modify schema example

```
{
  "_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
  "name" : "joe",
  "friends" : 32,
  "enemies" : 2
}
```

↓

```
{
  "_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
  "username" : "joe",
  "relationships" :
     {
        "friends" : 32,
        "enemies" : 2
     }
}
```

Get document to update

```
> var joe = db.users.findOne({"name":"joe"});
```

Create “username” / Delete “name”

```
> joe.username = joe.name;
> delete joe.name;
```

Create relationships / Delete “friends” and “enemies”

```
>joe.relationships={"friends":joe.friends,"enemies":joe.enemies};
> delete joe.friends;
> delete joe.enemies;
```

Store updated document

```
> db.users.update({"name":"joe"},joe)
```

Documentation: https://docs.mongodb.org/manual/tutorial/modify-documents/

MongoDB: Updating

- Looks for docs that match the query and updates spec. fields

```
db.users.update({"name":"Sergi"},
{$set:{"age":"29"}})
```

update operator (unset, inc, min, …) → $set

document or pipeline ← {"age":"29"}

- Update only the first document that matches the query

  db.collection.updateOne(*filter*, *update*, *options*)

- "Upsert option"
  - If no document matches the query, it creates a new document
  - Should be avoided

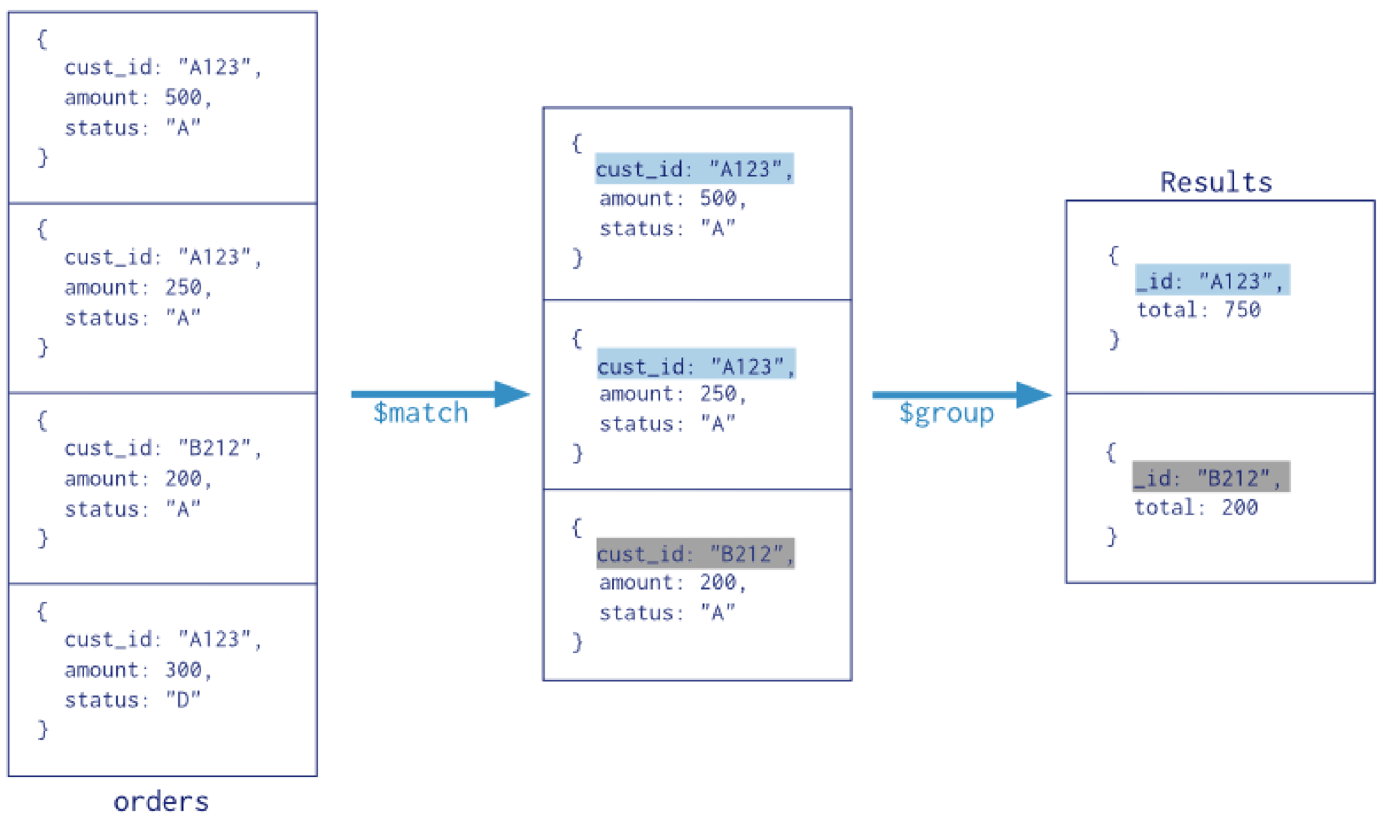
Documentation: https://docs.mongodb.org/manual/tutorial/modify-documents/

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 results
    - *Filters* that operate like queries
    - *Document transformations* that modify the form of the output
    - Grouping
    - Sorting
    - Other operations
- MapReduce

MongoDB: The Aggregation Framework

```
    Collection
        ↓
db.orders.aggregate(
    $match phase ──► { $match: { status: "A" } },
    $group phase ──► { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
                   )
```



MongoDB: The Aggregation Framework

Pipeline stages: ($match, $group, $addfields, $sort, $unwind …)

The name of the computed field

```
db.orders.aggregate(
                    {$match: {status:"A"}},
                    {$group: {_id: "$cust_id", total:{$sum: "$amount"}}}
                   )
```

Required field: to identify the field for the group by

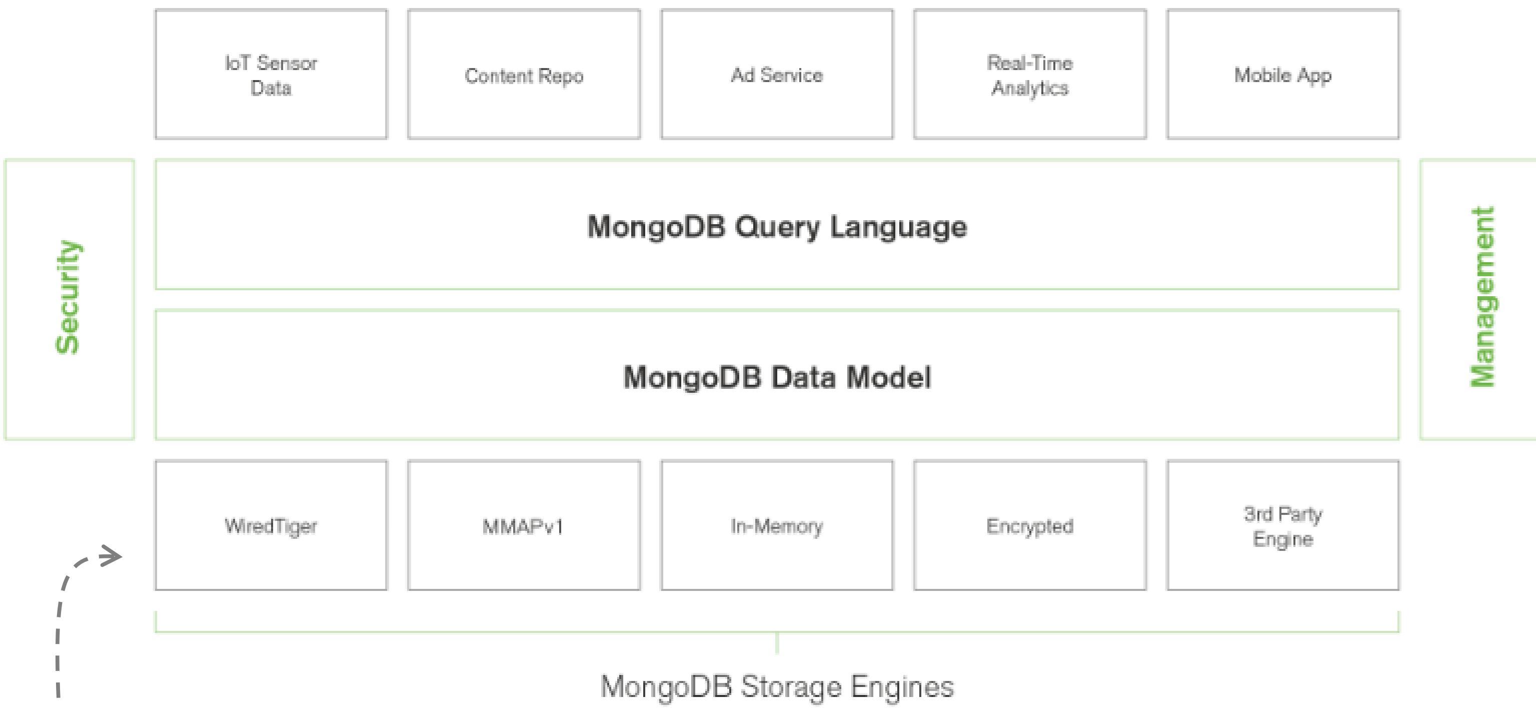
References the field

Pipeline operators: $sum, $max, $min …

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

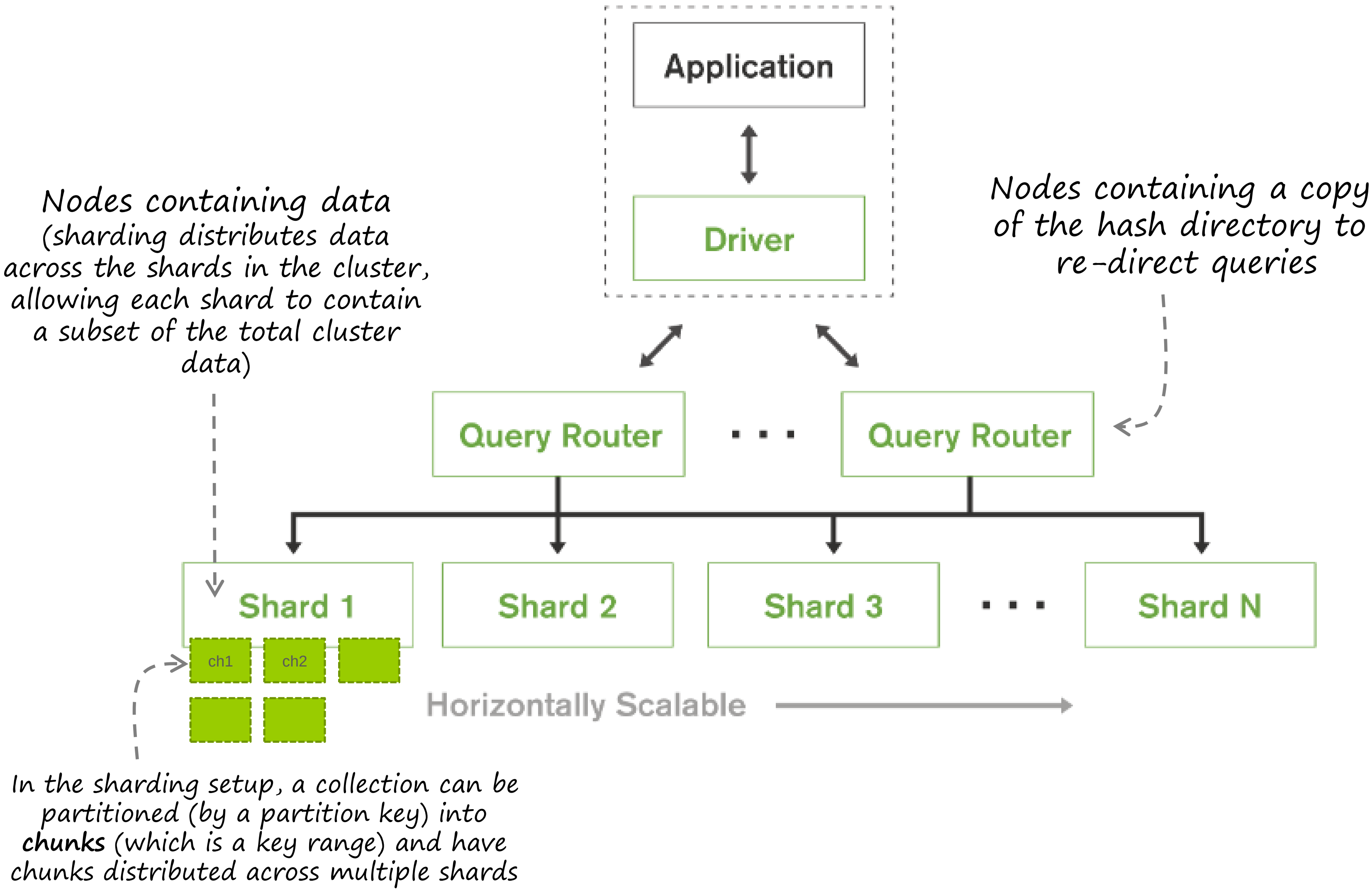
MongoDB Architecture

MongoDB Architecture



different engines perform
better for specific workloads

MongoDB Storage: Sharding (1)



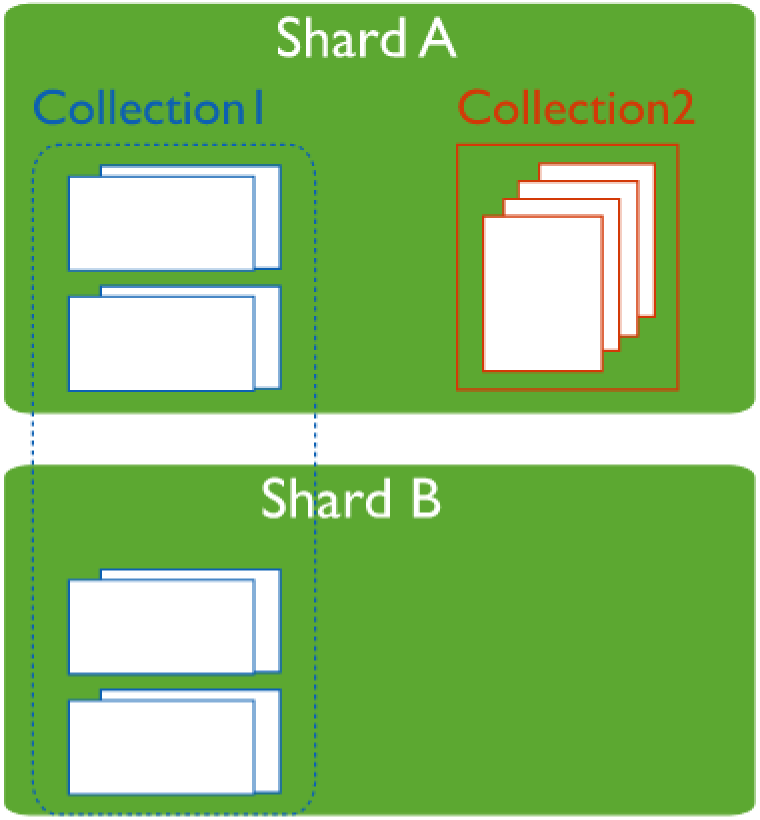
Sharding (2)

Shard key

- The shard key is used to distribute the collection's documents.
- The shard key is mandatory and consists of a field or multiple fields in the documents.

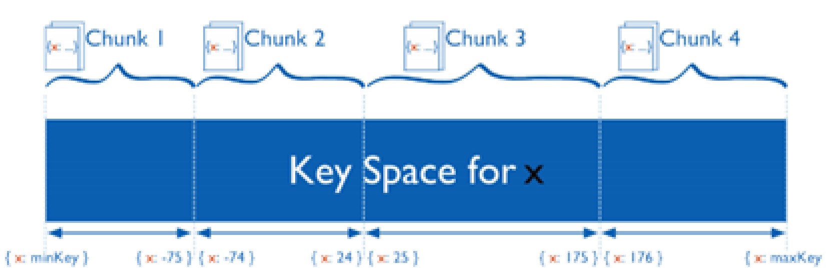
Chunk

- Sharded data is partitioned into chunks.

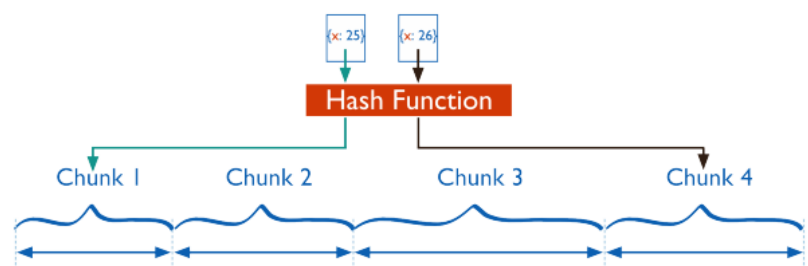


MongoDB: Fragmentation 2.X

- Two sharding strategies
  - **Range-based:** MongoDB determines the chunks by range
    - Adequate for range queries
  - **Hash-based:** Consistent hashing (a hash function determines the chunks)
    - A Hash-indexed field is required



vs.

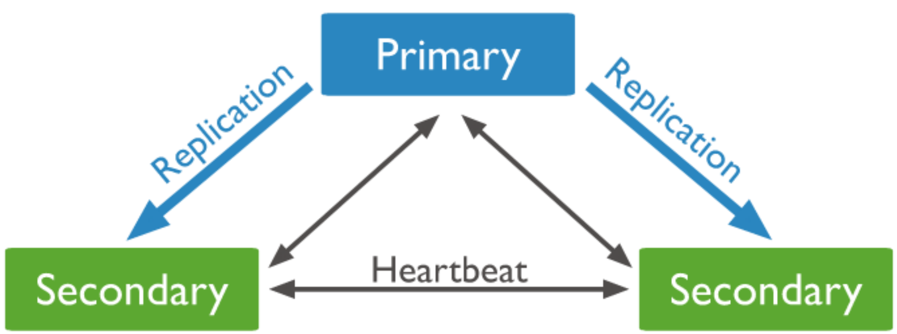


MongoDB Replication

- 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

- 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



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 http://docs.mongodb.org/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…

http://docs.mongodb.org/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: http://docs.mongodb.org/manual/tutorial/analyze-query-plan/
  - Use $hint to force MongoDB choose an index

  http://docs.mongodb.org/manual/reference/method/cursor.hint/#cursor.hint

MongoDB: Well-Known Limitations

- Architectural Issues http://docs.mongodb.org/manual/reference/limits/
  - 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
  - A database cannot be bigger than 32TBs
  - Theoretically, sharding is automatic and transparent. But in practice it is not. Most typical ones:
    - Size of the sharding key is limited (512 bytes)
    - 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)

MongoDB: Conclusions

- MongoDB has its flaws, but it is one of the most supported and mature NOSQL tools
  - Still, it is robustness is far away from a relational database
- Managing and Monitoring
  - OpsManager: Be careful! The terms of use say your data is periodically sent to MongoDB (the Company)
    - db.collection.explain()
- Visualization: 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!)

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.