Database Management Systems
INFO 210

NoSQL – Recent developments in database research and application
Lecture 14

Franz Wotawa
TU Graz, Institut for Software Technologie
Inffeldgasse 16b/2
wotawa@ist.tugraz.at

Notes

This slides are provided from Prof. Dr. Reinhard Pichler from the Institute of Logic and Computation, Databases & Artificial Intelligence Group, TU Vienna.

The content is not completely representing all aspects of today's database research and applications but at least the main streams

References

- Textbooks:
  - Pramod J. Sadalage and Martin Fowler:
    NoSQL Distilled. Addison-Wesley 2013
  - Lena Wiese:
    Advanced Data Management for SQL, NoSQL, Cloud and Distributed Databases. de Gruyter, 2015
- Scientific Articles:
  - Felix Gessert, Wolfram Wingerath, Steffen Friedrich, Norbert Ritter: NoSQL database systems: a survey and decision guidance. Computer Science - R&D 32(3-4): 353-365 (2017).

3

References (cont.)

- Further Scientific Articles:
  - Eric A. Brewer: CAP twelve years later: How the "rules" have changed. IEEE Computer 45(2): 23-29 (2012)
  - Daniel Abadi: Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. IEEE Computer 45(2): 37-42 (2012)
  - Dan Pritchett: BASE: An ACID Alternative. ACM Queue 6(3): 48-55 (2008)
- Copyright of all figures is with one of these resources.

4

Contents

5

Every minute on the Internet

Data comes along several dimensions:
- "Volume" or the amount of data
- "Velocity" or the speed of data
- "Variety" or the range of data types and sources

Special data sources: Time dependent data (sensor values (e.g. from a temperature sensor))

Unstructured or **semi-structured** data

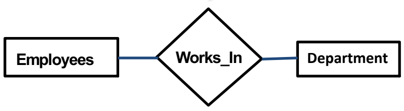
Structured data (e.g., relational DBs coming with entities & relationships and functional dependencies,…)

Text

.... Dynamic testing is another widely used alternative as it is sometimes the only way to reason about functional and quantitative properties of product of an SPL. However, it is most of the time impractical to enumerate, measure, and test all in- dividual products of an SPL. As a result, machine learning (ML) tech- niques are more and more considered to predict the behavior of an SPL out of a (small) sample of configurations .....

XML


```
- <Parts>
  - <Part>
      <Id>4478</Id>
      <Part_Name>1000 Ohm Resistor</Part_Name>
      <Total_Available>25000</Total_Available>
      <Price>0.01</Price>
    </Part>
  - <Part>
      <Id>3328</Id>
      <Part_Name>15000 Ohm Resistor</Part_Name>
      <Total_Available>75000</Total_Available>
      <Price>0.02</Price>
    </Part>
  - <Part>
      <Id>4725</Id>
      <Part_Name>555 Timer IC</Part_Name>
      <Total_Available>1500</Total_Available>
      <Price>0.25</Price>
    </Part>
  </Parts>
```

Relational DBMSs

- Strengths:
  - Strong formal foundation (logic, relational algebra)
  - Standard data model and query language (SQL)
  - Concurrency: support multi-user access; transactions
  - Data integration: store data of many applications in one database
- Previous alternative proposals:
  - XML Stores, Object-Oriented DBMSs
  - Have been integrated into RDBMSs and complement/extend now their functionality

7

ACID property of Relational DBMS

- Transactions have the following four properties:
  - **Atomicity** – Either a transaction completes or nothing happens at all.
  - **Consistency** – A transaction must start in a consistent state and leave the system in a consistent state.
  - **Isolation** – Intermediate results of a transaction are not visible outside the current transaction.
  - **Durability** – Once a transaction was committed, the effects are persistent, even after a system failure.

Limitations of Relational DBMSs

- Schema: must be known before any data can be added
- Data structure:
  - RDBMS are designed for structured data; limited support for unstructured data (text, pictures/videos)
  - Requires horizontal and vertical homogeneity of data (same structure per row, same type per column).
  - Data belonging to an entity is split into several tables
  - Impedance Mismatch: relational model vs. in-memory data structures (e.g.: OO, nested structures)
- Cost:
  - major vendors: closed source with strict licensing fees
  - usual licensing model: pay per user / computing node

9

Time Consumption in RDBMSs

- M. Stonebraker et al. have found that in a typical RDBMS, only 6.8% of the execution time is spent on "useful work".
- The rest is spent on:
  - buffer management (34.6%): disk I/O
  - locking (16.3%): to control concurrency
  - latching (14.2%): to protect shared data structures from race conditions caused by multi-threading
  - logging (11.9%): to guarantee durability
  - other code parts (16.2%)
- Conclusion: the end of the "one-size-fits-all" era

Reference: Michael Stonebraker, et al.: The End of an Architectural Era (It's Time for a Complete Rewrite). VLDB 2007: 1150-1160

10

New Data Management Challenges

- Volume: processing "big data"
- Complex data structures:
  - Nested structures
  - Complex relationships (graph structure)
- Schema Independence / Schema Evolution:
  - to facilitate integration of data from various sources
  - coping with changing structure of the data
  - allow self-descriptiveness
- Sparseness:
  - Many data items are unknown or non-existent
  - Would result in many NULLs in relational tables

11

Reasons for Emergence of New DB Systems

- Application vs. integration database:
  - Shift from idea of integration DB to application DB, i.e.: each application maintains its own database
  - Data exchange via web services
  - Use rich structure for communication (to reduce number of round trips in the interaction)
  - advantages of application DB: more freedom in data modelling, shift features to application (e.g. security)
- Use of Clusters
  - Scale out to cope with huge amounts of data;
  - RDBMSs are not well-suited for running on clusters.

12

The Term "NoSQL"

- History:
  - June 2009: meetup in San Francisco for "open-source, distributed, non-relational databases".
  - organized at the time of the Hadoop Summit 2009;
  - search for a Twitter hashtag without many Google hits;
  - NoSQL was chosen
- Some of the Systems presented at this 2009 meetup: https://blog.oskarsson.nu/post/22996140866/nosql-debrief
  - Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB, **MongoDB**

13

The Term "NoSQL" (cont.)

- NoSQL = "Not only SQL"
- NoSQL as an umbrella term for all those DB systems that share most of the following properties:
  - open-source, distributed, non-relational
  - schemaless
  - restricted querying capabilities (e.g., no joins)
  - horizontally scalable; running well on clusters
  - no (or weaker form of) transactions, not guaranteeing ACID properties, use BASE instead

Basically **A**vailable, **S**oft state, **E**ventual consistency

14

Pioneering Contributions by Companies

- Early systems:
  - Google: Bigtable (column family store)
  - Amazon: DynamoDB (key-value store)
  - Facebook: Cassandra (column family store), later used by Twitter
  - LinkedIn: Project Voldemort (open-source implementation of Amazon's DynamoDB)
- Characteristics of early systems:
  - Huge number of users
  - Huge number of reads/writes
  - Non-critical transactional systems (ACID not needed)

15

Benefits of NoSQL Database Systems

- High performance:
  - Handling huge amounts of data
  - Specific optimizations (e.g.: compression in column stores; path queries in graph databases)
- Horizontal scalability:
  - Add nodes to the cluster as load or DB grows
- Schemaless:
  - Simpler storage mechanisms: performance
  - No schema needed upfront, flexible when data changes
- No (or weaker forms of) transactions:
  - Simpler transaction handling; not blocking
  - Reduced logging requirements

16

Disadvantages of NoSQL Systems

- No standardized Query Language:
  - Users need to learn a separate query language for each system (even within the same category)
- Weaker form (or no support of) transactions
  - no ACID support (in particular, consistency)
- Limited support of ensuring integrity:
  - Responsibility for uniqueness constraints, referential integrity, etc. is shifted to the application.

17

Contents

- Motivation
- **Four Categories of NoSQL Systems**
- Further Aspects of Data Modelling
- Document Stores
- MongoDB Primer

18

Four Categories

- Key-value stores:
  - Stores values indexed by keys
- Document stores:
  - Similar to key-value stores but DBMS can understand the format of the value; typically JSON, XML
- Columnar databases, column family stores:
  - Stores data organized in columns
  - Can be thought of as two-dimensional indexing of values: first by id (like key) then by column name
- Graph Databases:
  - Uses graphs to represent data; vertices represent entities; edges represent (directed) relationships

19

Examples of Systems in the Four Categories

See http://nosql-database.org/ (over 200 systems);
(see also https://db-engines.com/en/ranking)

- Key-value stores:
  - Riak, Redis, BerkeleyDB, DynamoDB (Amazon); open source versions: Dynomite, Project Voldemort
- Document stores:
  - **MongoDB**, CouchDB, Terrastore
- Columnar databases, column family stores:
  - MonetDB, HBase, Cassandra, Hypertable, Accumulo
- Graph Databases:
  - Neo4J, HyperGraphDB, Infinite Graph

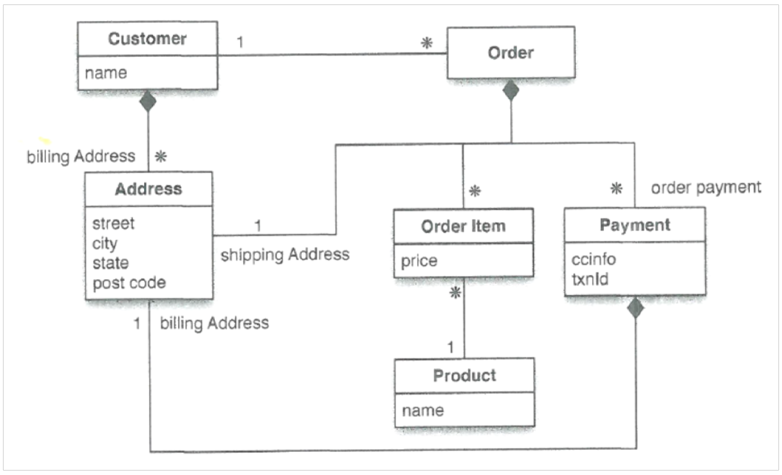
20

Complex Structures

- "Aggregate Orientation":
  - Data organization in RDBMS: by tuples
  - many applications: access to more complex structures
  - aggregate: = collection of objects that we want to treat as a unit, e.g. unit of access to storage, unit of updates, unit of communication
  - in data modelling: denormalization
  - main goal of NoSQL data modelling: read optimization
- Consequences of aggregate orientation:
  - Well-suited for running on a cluster
  - Consequence for transactions: easy to support atomic manipulation of single aggregate; difficult to support atomic manipulation of several aggregates

21

Complex Structures: Example



22

Complex Structures: Example (cont.)

```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}

// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
  }
 ],
"shippingAddress":[{"city":"Chicago"}]
```

```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
```

23

Aggregates in NoSQL Systems

- Key Value Stores:
  - arbitrary structure of values: opaque to DB system; only known to the application.
- Document Stores:
  - nested structure (XML, JSON) visible to the DB system
- Column-Oriented DBs:
  - Organize columns that are usually accessed together in column families (pioneer: Bigtable)
- Graph Databases:
  - aggregate-ignorant (like RDMBSs)
  - little information stored per entity; most important information lies in the relationships.

24

Contents

- Motivation
- Four Categories of NoSQL Systems
- **Further Aspects of Data Modelling**
- Document Stores
- MongoDB Primer

Relationships

- Data accesses:
  - aggregates help to put together data that is commonly accessed together.
  - Relationships between aggregates: usually, not all related data is combined in one aggregate; in particular, applications may have different aggregate boundaries.
- Example:
  - Customers vs. orders: one application may access customer data together with order information; other application may access only the orders.
  - Shall we store customers and orders as one aggregate or as separate aggregates?

Relationships (cont.)

- Modelling inter-aggregate relationships:
  - IDs in one aggregate to reference other aggregates
  - In general, this reference-semantics is not visible to the DB system; however, e.g. Riak allows link information in metadata and supports link-walking
  - Graph DBs are optimized for traversing relationships.
- Challenge with inter-aggregate relationships:
  - how to ensure atomicity of changes to related aggregates (ACID property of transactions in RDMBS)?

27

Materialized Views

- Materialized Views in RDBMSs:
  - Lack of aggregate structure in RDBMS: easier to support different ways of accessing the data
  - Views: virtual tables; convenient for encapsulation
  - Materialized views: views cached on disk
- NoSQL Systems:
  - Typically don’t have views but some allow precomputed and cached queries (referred to as materialized views)
- Update strategies:
  - Eagerly: whenever base data is updated
  - Lazily: update materialized views at regular intervals

28

Schemaless Databases

- Freedom and flexibility in NoSQL Systems:
  - key-value store: arbitrary contents in value
  - document store: arbitrary structure of documents
  - Column-oriented: arbitrary combination of columns
  - flexibility mainly applies to data within an aggregate, not when aggregate boundaries have to be modified.
- "Schemaless":
  - schema on write (RDBMS) vs. schema on read
  - implicit schema: DB system is ignorant of schema but application programs assume a schema
  - may be difficult to maintain: requires understanding of the schema in the application code.

Contents

- Motivation
- Four Categories of NoSQL Systems
- Further Aspects of Data Modelling
- **Document Stores**
- MongoDB Primer

Document Stores

❖ Document stores:
- ➢ Similar to key-value stores but DBMS can understand the format of the value
- ➢ Typical data formats: XML, JSON (JavaScript Object Notation), BSON (binary JSON)
- ➢ Allows advanced querying (e.g.: filters on field values) and relationships (references) between data objects
- ➢ Joins usually not supported for performance reasons; consequence: denormalizing data when modelling or combining documents on application level.
- ➢ Atomicity usually (only) of documents guaranteed

➢ In this lecture: look at MongoDB

31

Introduction to MongoDB

❖ Open Source

❖ Data format: stores data in BSON

❖ Name: from "humongous ": = enormous, gigantic.

❖ Brief History:

2007: started as part of developing cloud computing stack

2009: version 1 of MongoDB released

Constantly extended feature set, e.g.:
- Sharding (v1.6, 2010), Journaling (v1.8, 2011)
- Aggregation framework (v2.2, 2012)
- Document validation (v3.2, 2015)
- Multi-document ACID transactions (v.4.0, 2018)

32

Typical Use Cases

- Log data: rather than storing pure text, extract the various fields from a log file (e.g., host, user, time, path, etc.)
- e-commerce, e.g.:
  - product catalog
  - shopping cart
- Inventory management
- Content management:
  - storing content and metadata of web sites
  - storing users' comments

see also https://docs.mongodb.com/ecosystem/use-cases/

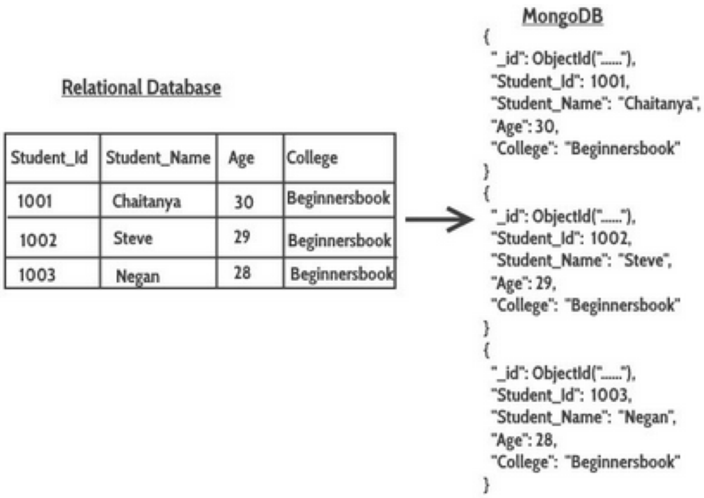
33

JSON (JavaScript Object Notation),

- Sytnax (cf. www.json.org):
  - JSON object = collection of name/value pairs in { }
  - the name is a string
  - the value can be
    - of a simple type: a number, string, true/false
    - an array: = sequence of values in [ ]
    - a nested JSON object: in { }
- Remark:
  - Two fields with the same name in principle allowed (in particular, also in BSON used by MongoDB)
  - most MongoDB interfaces disallow duplicate names because of the used representation (e.g., hash table)

34

Structure of the Database: SQL vs. MongoDB

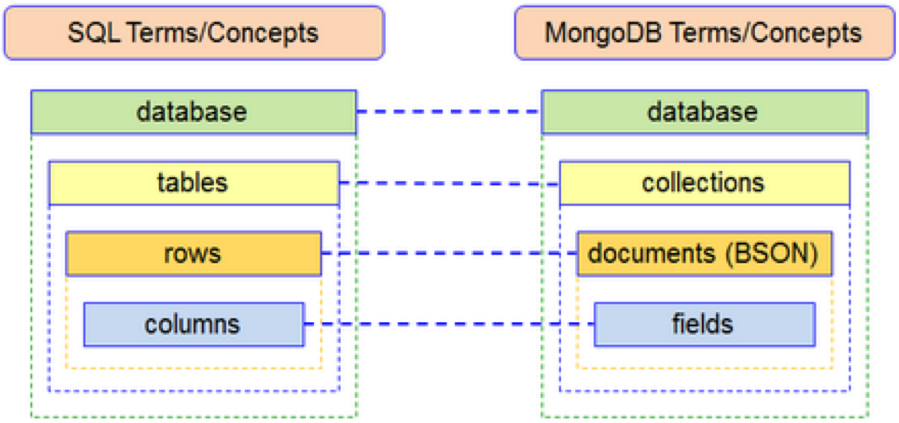


Student_Id	Student_Name	Age	College
1001	Chaitanya	30	Beginnersbook
1002	Steve	29	Beginnersbook
1003	Negan	28	Beginnersbook

```
{
"_id": ObjectId("......"),
"Student_Id": 1001,
"Student_Name": "Chaitanya",
"Age": 30,
"College": "Beginnersbook"
}
{
"_id": ObjectId("......"),
"Student_Id": 1002,
"Student_Name": "Steve",
"Age": 29,
"College": "Beginnersbook"
}
{
"_id": ObjectId("......"),
"Student_Id": 1003,
"Student_Name": "Negan",
"Age": 28,
"College": "Beginnersbook"
}
```

35

Structure of the Database: SQL vs. MongoDB (cont.)



36

Data Modelling

- Basic principles:
  - Keep intended data access in mind
  - aim for atomic reads/writes if possible
  - various ways to model 1:n and n:m relationships (choose the most suitable one for expected workload)
  - denormalization is an option
  - respect the max. document size of 16 MB
  - use document validation (e.g.: warnings if a document gets an unexpected field: check if this is a bug or a feature of the schema design)
  - decide which indexes shall be created

37

Modelling 1:n Relationships

- Relational model:
  - typically 2 tables (one for each of the entities)
  - model 1:n relationship with a foreign key
- MongoDB: several possibilities:
  - two collections with reference (analogous to a foreign key); reference can be _id or some key attribute: recommended for "one-to-millions"
  - array of references in the "one-side" document, e.g: array with _ids or order-numbers in customer document: recommended for "one-to-thousands"
  - embed an array of documents in the other document, e.g.: array of order-documents in a customer document; for "one-to-few"

38

Modelling 1:n Relationships (cont.)

- Combination of two referencing methods:
  - useful if fast access from "both sides" is required

```
db.person.findOne()
{
   _id: ObjectID("AAF1"),
   name: "Kate Monster",
   tasks [
          ObjectID("ADF9"),
          ObjectID("AE02"),
          ObjectID("AE73")
          // etc
   ]
}
```

```
db.tasks.findOne()
{
   _id: ObjectID("ADF9"),
   description: "Write lesson plan",
   due_date:  ISODate("2014-04-01"),
   owner: ObjectID("AAF1")
   // Reference to Person document
}
```

39

Modelling n:m Relationships

- Relational model:
  - typically three tables: auxiliary table for the relation-ship, one table for each entity,
  - foreign keys from auxiliary table into each of the others
- MongoDB: several possibilities, e.g.:
  - three collections with references as in relational case
  - references as for 1:n relationship, but from both sides, e.g.: store references to courses in student document (i.e., array of course IDs) and references to students in course document (i.e., array of student IDs).

40

Denormalization

- Idea:
  - allow redundancy to avoid joins when querying
  - disadvantage: updates are not atomic anymore
  - makes most sense for high read-to-write ratio
- Example 1: denormalizing into "1-side"

```
db.products.findOne()
{    name : 'left-handed smoke shifter',
     catalog_number: 1234,
     parts : [
       { id : ObjectID('AAAA'), name : '#4 grommet' },
       { id: ObjectID('F17C'), name : 'fan blade assembly' },
       { id: ObjectID('D2AA'), name : 'power switch' },
       // etc
     ] }
```

41

Denormalization (cont.)

- Example 2: denormalizing into "many-side"
  - increases the cost of updates yet further
  - high read-to-write ratio even more important

```
db.parts.findOne()
{  _id : ObjectID('AAAA'),
   partno : '123-aff-456',
   name : '#4 grommet',
   product_name : 'left-handed smoke shifter',
   product_catalog_number: 1234,
   qty: 94,
   cost: 0.94,
   price: 3.99
}
```

42

Document Validation

- MongoDB allows to specify rules for validating documents:
  - defined for a collection
  - in the form of a JSON Schema or a query
  - allows specification of action: error/warning

```
db.createCollection( "contacts", {
    validator: { $jsonSchema: {
      bsonType: "object",
      required: [ "phone", "name" ],
      properties: {
       phone: {
         bsonType: "string",
         description: “string; required"
       },
       name: {
         bsonType: "string",
         description: "string; required"
       }
      }
    } },
    validationAction: "warn"
} )
```

43

Replication

MongoDB uses Master-Slave distribution:

- MongoDB allows the definition of replica sets, i.e.: one primary (master) and several secondaries (slaves)
- Writes are only executed at the master and then propagated to the slaves.
- The secondary (slave) servers are in hot standby, i.e. they keep an up-to-date copy of the data and are ready to take over when the primary server goes down.
- When the secondary servers do not hear from the primary for more than 10 seconds (configurable), they hold an election to vote for a new primary.
- A replica set may also contain an arbiter, i.e.: a node that has a vote in an election but carries no data.

44

Read Preferences

- MongoDB allows definition of Read preferences to define the read behaviour of a replica set:
  - primary resp. secondary: reads are *exclusively* executed on primary resp. secondary server;
  - primaryPreferred resp. secondaryPreferred: reads are executed on primary resp. secondary *if available*;
  - nearest: reads are executed on server which is nearest in terms of network latency.
  - For a transaction with reads & writes: read preference primary required
- Example: db.getMongo().setReadPref('secondaryPreferred‘);

45

Write Concerns

- Write concerns:
  - A write operation is acknowledged in a replica set once the write is acknowledged by the primary server;
  - This behavior can be changed by write concerns on different levels (single operation, transaction, session)
  - Values, in particular "majority" or number of servers ("majority" = $\lfloor N/2 \rfloor + 1$ with N = #servers in replica set)
- Example:

```
db.books.insert(
    { name: "Mastering MongoDB 4.x", isbn: "1001" },
    { writeConcern: {w: 2, wtimeout: 5000} } )
```

write is acknowledged when it is confirmed by 2 servers
(i.e., primary plus one secondary). If the timeout of 5000 ms
is exceeded, an error is returned.

46

Read Concerns

- Read concerns:
  - Can be defined for read operations (queries), e.g.: find(), aggregate(), count(), etc.;
  - Values, in particular: "local": current value on the read target, "majority" (most recent value on a majority of the servers).
  - default: "local".
- Example: db.persons.find({ age: 22}).
  readConcern(“majority").maxTimeMS(5000)

read the most recent values from a majority of servers;
returns an error, if the timeout of 5000 ms is exceeded.

47

Transactions

- Multi-document transactions with ACID properties have been introduced in version 4.0 (2018).
- Atomicity of single document access is guaranteed also without transactions (and recommended for performance).
- Example:

```
session = db.getMongo().startSession( );
session.startTransaction( { writeConcern: { w: "majority" } } );
db.persons.insertMany( [{ name: "alice" , age: 22, location: “vienna"},
                        { name: "bob" , age: 23 location: "graz"}]);
session.commitTransaction();  // respectively: session.abortTransaction();
session.endSession();
```

48

Sharding

- Sharding is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations.
- Database systems with large data sets or high throughput applications can challenge the capacity of a single server. For example, high query rates can exhaust the CPU capacity of the server. Working set sizes larger than the system's RAM stress the I/O capacity of disk drives.

Two methods for addressing system growth

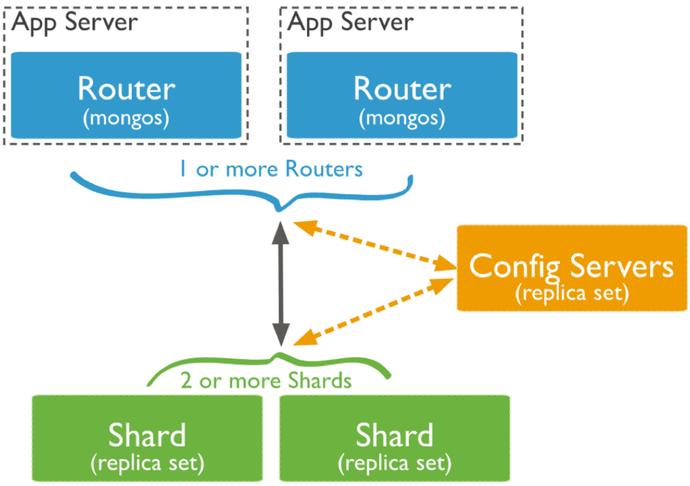
- ***Vertical Scaling*** involves increasing the capacity of a single server, such as using a more powerful CPU, adding more RAM, or increasing the amount of storage space.
- ***Horizontal Scaling*** involves dividing the system dataset and load over multiple servers, adding additional servers to increase capacity as required.

Sharding

- MongoDB supports sharding on collection level (for horizontal scaling)
- A sharded cluster consists of the following elements:
  - Two or more shards, which can be replica sets
  - One or more query routers ("mongos")
  - A replica set of config servers (for metadata and configuration data of the entire cluster).
- Sharding is defined via a shard key (= field that exists in every document in the collection to be sharded)
- Sharding Strategies:
  - Hashed sharding: chunks defined by hash values
  - Ranged sharding: chunks as ranges of shard key values

51

Components in a Sharded Cluster



52

Advantages of Sharding

- **Reads / Writes**
  MongoDB **distributes the read and write workload** across the shards in the sharded cluster. For **queries that include the shard key** or the prefix of a compound shard key, mongos can **target the query at a specific shard** or set of shards.

- **Storage Capacity**
  Sharding **distributes data across the shards** in the cluster, allowing each shard to contain a subset of the total cluster data.

- **High Availability**
  A sharded cluster can **continue to perform partial read / write** operations even if one or more shards are unavailable. You can deploy config servers as **replica sets**

Contents

54

Basic MongoDB Commands (1)

To do this	Run this command	Example
Start server	mongod	mongod
Start shell	mongo	mongo
Show current database	db	db
Select or switch database [1]	use <database name>	use mydb
Execute a JavaScript file	load(<filename>)	load (myscript.js)
Display help	help	help
show all databases	show dbs	show dbs
Show all collections in current database	show collections	show collections

55

Basic MongoDB Commands (2)

To do this	Run this command	Example
Insert new document in a collection [	db.collection.insert(*<document>*)	db.books.insert({"isbn": 9780, "title": "Alice in Wonderland“, ...})
Insert multiple documents into a collection	db.collection.insertMany([*<document1>, <document2>, ...*]) -or- db.collection.insert([*<document1>, <document2>, ...*])	db.books.insertMany([{"isbn": 9780, "title": "Alice in Wonderland“, ...}, {"isbn": 9781, "title": ...}])
Show all documents in the collection	db.collection.find()	db.books.find()
Filter documents by field value condition	db.collection.find(*<query>*)	db.books.find({"title":"Alice in Wonderland"})

56

Basic MongoDB Commands (3)

To do this	Run this command	Example
Show only some fields of matching documents	db.collection.find(*<query>, <projection>)*	db.books.find({"title":"Alice in Wonderland"}), {title:true, _id:false})
Show first document that matches the query condition	db.collection.findOne(*<query>, <projection>*)	db.books.findOne({}, {_id:false})
Update specific fields of a single document	db.collection.update(*<query>, <update>*)	db.books.update({title : Alice in Wonderland"}, {$set : {category :"Fiction"}})
Remove certain fields of a single document the query condition	db.collection.update(*<query>, <update>*)	db.books.update({title : "Alice in Wonderland"}, {$unset : {author:""}})

57

Basic MongoDB Commands (4)

To do this	Run this command	Example
Remove certain fields of all documents that match the query condition	db.collection.update(*<query>, <update>*, {multi:true})	db.books.update({title : "Alice in Wonderland"}, {$unset : {author:""}}, {multi:true})
Delete all documents matching a query condition	db.collection.remove(*<query>*)	db.books.remove({"category" :"Fiction"})
Delete all documents in a collection	db.collection.remove({})	db.books.remove({})
Create an index	db.collection.createIndex({indexField:type}) Type 1 means ascending; -1 means descending	db.books.createIndex({title:1})

58

Basic MongoDB Commands (5)

To do this	Run this command	Example
Create an index on multiple fields (compound index)	db.collection.createIndex({index Field1:type1, indexField2:type2, ...})	db.books.createIndex({title:1, author:-1})
Show all indexes in a collection	db.collection.getIndexes()	db.books.getIndexes()
Drop an index	db.collection.dropIndex({indexField:type})	db.books.dropIndex({author:-1})
number of documents in the collection	cursor.count()	db.books.find().count()
Display formatted result	cursor.pretty()	db.books.find({}). pretty()

59

Next Class

- Summarizing lecture content