NoSQL: MongoDB and DocumentDB

Dominic Duggan
Stevens Institute of Technology

1

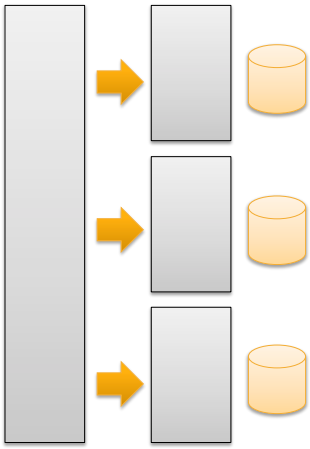
Recall: Relational Database Summary

- Database Schema
  - Normalized for efficiency
- SQL for ad-hoc queries
- Transactional updates
  - Atomic
  - Consistent
  - Isolated
  - Durable

2

Challenge: Big Data™

- Historical approach:
  vertical scaling
  - Limited
- Modern approach:
  horizontal scaling
  - *Sharding*
  - Azure: Federated SQL databases
  - Applications see data partitioning
  - No joins across partitions



3

SQL vs NoSQL

Relational

- Database Schema
  - Business data model
- SQL for ad-hoc queries
- ACID properties
  - Atomic
  - Consistent
  - Isolated
  - Durable

NoSQL

- Unstructured
- Map-Reduce
- BASE properties
  - Basically Available
  - Soft state
  - Eventually consistent

4

NoSQL taxonomy

- Key-Value stores
- Column stores
- Document stores

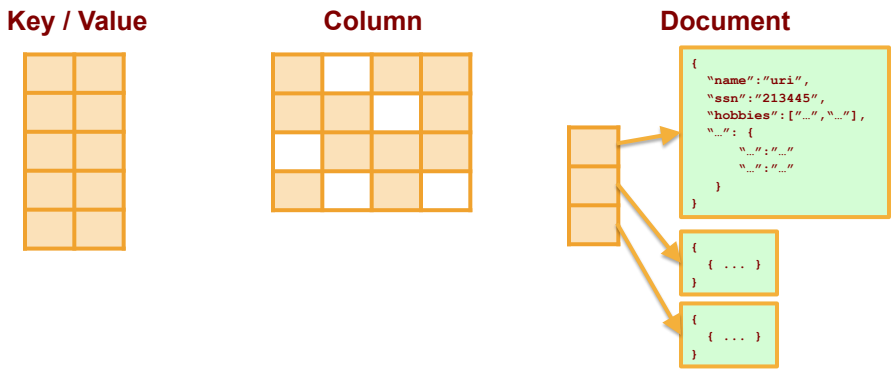
5

NoSQL taxonomy

• Key-Value stores	Amazon Dynamo
•	**Azure Tables**
• Column stores	Google Bigtable, Cassandra
• Document stores	CouchDB, MongoDB, **Azure CouchDB**

6

A Universe of Data Models



7

Key/Value

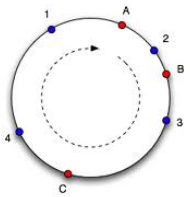
- Have the key? Get the value
  - Map/Reduce (sometimes)
  - Good for
    - cache aside (e.g. Hibernate 2nd level cache)
    - Simple, id based interactions (e.g. user profiles)
- In most cases, values are opaque

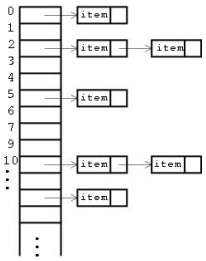
K1	V1
K2	V2
K3	V3
K4	V1

8

Key/Value

- Scaling out is relatively easy
  - just hash the keys
- Some will do that automatically for you
- Fixed vs. consistent hashing

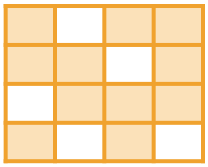




9

Column Based

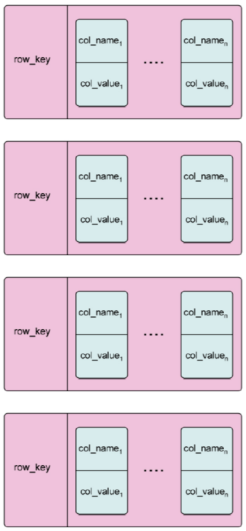
- Mostly derived from Google's BigTable papers
- One giant table of rows and columns
  - Column == pair (name and a value, sometimes timestamp)
  - Each row can have a different number of columns
  - Table is sparse:
    (#rows) × (#columns) ≥ (#values)



10

Column Based

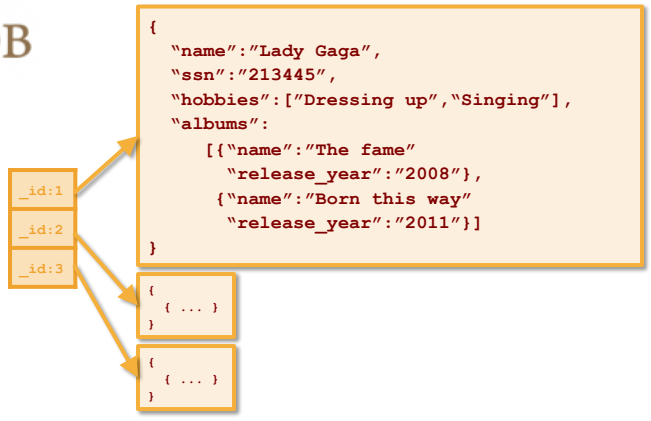
- Query on row key
  - Or column value (aka secondary index)
- Good for a constantly changing, (albeit flat) domain model



11

Document

- Think JSON (or BSON, or XML)



12

Document

- Semi-structured data
  - Arrays, nested documents
- Ad hoc queries
  - MongoDB: Stored predicates
  - DocumentDB: DocumentDB SQL
- Very intuitive model
- Flexible schema

```
> db.people.find({age: {$gt: 27}})
{ "_id" : ObjectId("4bed80b20b4acd070c593bac"), "name" : "John", "age" : 28 }
{ "_id" : ObjectId("4bed80bb0b4acd070c593bad"), "name" : "Steve", "age" : 29 }
```

13

Comparison

	RDBMS	Azure Tables	MongoDB	Document DB
Organization	Flat tables	Key-value	BSON	JSON
Schema	Yes	No	No	No
Query	SQL	Map-Reduce	Stored Predicates	DDB SQL
Foreign Keys	Yes	No	No	No
Joins	Yes	No	No	Intra-document
Transactions	Yes	No	No	Yes
Consistency	Strong	Eventual	Eventual	Session

14

MONGODB

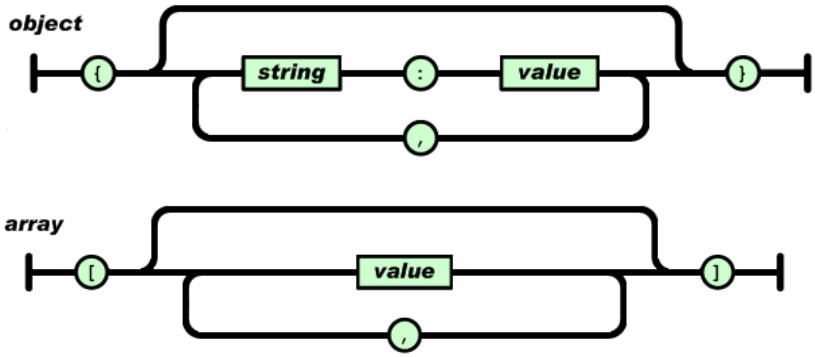
15

MongoDB vs Relational

- Scale out vs scale up
  - Automatic rebalancing
- Document (JSON) vs row
- Schema-free
- Stored Javascript vs stored procedures
- MapReduce vs SQL
- Files vs tables
- Secondary indexing

16

- JavaScript Object Notation (JSON) specify: literal values for types in JavaScript.



value can be any string, number, object, array, or the literal values true, false, or null.

Examples

```
var cities = [”Boston",
              ”New York",
              ”Washington D.C."];

var employee =
    {
        "Name": ”Joe",
        "Salary": 50000,
        "Skills": [”Azure",
                   ”Cassandra",
                   ”Hadoop”]
     };
```

Javascript & JSON

```
<script type="text/javascript">
function getCatalog() {
  val url = "http://www.example.org/json/catalog";
  $.ajax ({ url: url, type: “GET”,
            success: callback });
}





</script>
```

19

Javascript & JSON

```
<script type="text/javascript">
function getCatalog() {
  val url = "http://www.example.org/json/catalog";
  $.ajax ({ url: url, type: “GET”,
            success: callback });
}
function callback(response) {
  jsonContent = eval("(" + response+ ")");
  for (i=0; i<jsonContent.catalog.item.length; i++)
  {
      ... jsonContent.catalog.item[i].title ...
  }
}
</script>
```

20

MONGODB ESSENTIALS

21

MongoDB Essentials

- **Document:** JSON syntax
  - But more types (BSON)!
- Keys are ordered
- No duplicates, names case-sensitive
- Fields have types
- Reserved chars: . And $
- Special key: "_id"

22

MongoDB Essentials

- **Collection:** group of documents
- Schema-free

```
{ “ greeting” : “Hello, world!” }
{ “foo” : 5 }
```

- Why collections?
  - Manageability
  - Faster (type-free) queries
  - Data locality
  - Per-collection indexes
- Subcollections: `blog.posts`, `blog.authors`, etc

23

MongoDB Essentials

- **Database:** One per application
- Reserved DB names
  - admin: global
  - local: part of replicated database
  - config: sharding
- Namespace: e.g. `cms.blog.posts`
  - Database: `cms`
  - Collection: `blog.posts`

24

Running MongoDB

- Server: `mongod`
- Default directory
  - Unix: `/data/db`
  - Windows: `C:\data\db`
- Default port: 27017
  - http: MongoDB port + 1000

25

Running MongoDB

- Shell: `mongo`
- MongoDB client
- Javascript interpreter
- Default database: `test`
  - Switch: `use foobar`
- Bound variable: db

26

Creating a document

```
post =
{
  “title” : “My Blog Post”,
  “content” : “Here’s my post.”,
  “date” : new Date()
}

db.blog.insert(post)

db.blog.find()
{
  “_id” : ObjectId(“4b23c3ca7525f35f94b60a2d”),
  “title” : “My Blog Post”,
  “content” : “Here’s my post.”,
  “date” : “Sat Dec 12 2009 11:23:21 GMT-0500 (EST)”
}
```

27

Updating a document

```
post.comments = [ ]

db.blog.update({“title” : “My Blog Post”}, post)

db.blog.find()
{
  “_id” : ObjectId(“4b23c3ca7525f35f94b60a2d”),
  “title” : “My Blog Post”,
  “content” : “Here’s my post.”,
  “date” : “Sat Dec 12 2009 11:23:21 GMT-0500 (EST)”
  “comments” : [ ]
}

db.blog.delete({“title” : “My Blog Post”})
```

28

Shell Commands

```
show dbs
show collections
show users
show profile
use db-name
db.help()
db.foo.help()
db.foo.find()
db.foo.find( {title : “My Blog Post”} )
it
```

29

Iterating over Subcollections

```
var collections =
    [“posts”, “comments”, “authors”];

doStuff(db.blog.posts);
doStuff(db.blog.comments);
doStuff(db.blog.authors);

for (i in collections)
{
   doStuff(db.blog[collections[i]]);
}
```

30

Data Types

- Null
- Boolean
- 32-bit integer
- 64-bit integer
- 64-bit floating point
- String
- Symbol
- Object id
- Date
- Regular expression
- Code
- Binary data
- Maximum value
- Minimum value
- Undefined
- Array
- Embedded document

31

Data Types: Remarks

- Numbers
  - MongoDB: 4-byte int, 8-byte int, 8-byte float
  - Javascript: float
  - 8-byte int: approximate value in shell
- Dates
  `new Date(…)`
- `_id` and ObjectId
  - `ObjectId = (Timestamp || Machine || PID || Increment)`
  - Auto-generation of _id: client-side

32

Data Types: Remarks

- Embedded Documents = denormalized data

```
{
    "name" : "John Doe",
    "address" : {
        "street" : "123 Park Street",
        "city" : "Anytown",
        "state" : "NY"
    }
}
```

33

CRUD: CREATE, READ, UPDATE, DELETE

34

Insertion

- Single insertion

```
db.foo.insert({“bar” : “baz”})
```

- Batch insertion

35

Deletion

- Remove all data

```
db.mailinglist.remove()
```

- Remove specific data

```
db.mailinglist.remove({“opt-out” : true})
```

- Remove collection (fast remove)

```
db.drop_collection(mailinglist)
```

36

Document Replacement

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

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


Document Replacement

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

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

Document Replacement

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

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

39

Document Replacement

```
joe.username = “joe”;
delete joe.name;
```

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

40

Document Replacement

```
delete joe.friends;
delete joe.enemies;
db.users.update({"name" : "joe"}, joe);
```

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

41

Modifiers for Partial Updates

```
db.analytics.find()

{
  "_id" : ObjectId("4b253b067525f35f94b60a31"),
  "url" : "www.example.com",
  "pageviews” : 52
}

db.analytics.update(
  {“url” : “www.example.com”},
  ...
  {“$inc” : { “pageviews” : 1 } }
)
```

42

Modifiers for Partial Updates

```
db.users.findOne()

{
  “_id” : ObjectId(“4b253b067525f35f94b60a31”),
  ”name” : “joe”,
  “age” : 30,
  “sex” : “male”,
  “location” : “Wisconsin”
}

db.users.update(
  {“_id” : ObjectId(“4b253b067525f35f94b60a31”)},
  ...
  {“$set” : {“favorite book” : “war and peace”}}
)
```

43

Modifiers for Partial Updates

```
db.users.findOne()

{
  "_id" : ObjectId("4b253b067525f35f94b60a31"),
  "title" : "A Blog Post”,
  "content" : "...",
  "author" : {
    "name" : "joe”,
    "email" : "joe@example.com"
  }
}
db.blog.posts.update(
  {“author.name” : “joe”},
  ...
  {“$set” : {“author.name” : “joe schmoe”}}
)
```

44

Pushing onto a List

```
db.blog.posts.findOne()
{
  "_id" : ObjectId("4b2d75476cc613d5ee930164"),
  "title" : "A blog post",
  "content" : "...”
}
db.blog.posts.update({"title" : "A blog post"},
  {$push : {"comments" : ... {"name" : "joe",
                              "email" : "joe@example.com",
                              "content" : "nice post."}}})
db.blog.posts.findOne()
{
  "_id" : ObjectId("4b2d75476cc613d5ee930164"),
  "title" : "A blog post",
  "content" : "...",
  "comments" : [ {"name" : "joe”,
                  "email” : "joe@example.com",
                  "content" : "nice post.” } ]
}
```

45

Pushing onto a List

```
db.blog.posts.update({"title" : "A blog post"},
  {$push : {"comments" : ... {"name" : "bob",
                              "email" : "bob@example.com",
                              "content" : "good post."}}})

db.blog.posts.findOne()
{
  "_id" : ObjectId("4b2d75476cc613d5ee930164"),
  "title" : "A blog post",
  "content" : "...",
  "comments" : [ {"name" : "joe”,
                  "email” : "joe@example.com",
                  "content" : "nice post.” },
                 {"name" : ”bob”,
                  "email” : ”bob@example.com",
                  "content" : ”good post.” } ]
}
```

46

Pushing onto a List

```
db.papers.update(
  { },
  ... {$push : {"authors cited" : "Richie"}}
)

db.papers.update(
  {"authors cited" : "Richie”},
  ... {$push : {"authors cited" : "Richie"}}
)

db.papers.update(
  {"authors cited" : {"$ne" : "Richie"}},
  ... {$push : {"authors cited" : "Richie"}}
)
```

47

Adding to a Set

```
db.users.findOne(
  {"_id" : ObjectId("4b2d75476cc613d5ee930164")}
)

{
  "_id" : ObjectId("4b2d75476cc613d5ee930164"),
  "username" : "joe",
  "emails" : ["joe@example.com",
              "joe@gmail.com",
              "joe@yahoo.com”]
}
```

48

Adding to a Set

```
db.users.update(
  {"_id" : ObjectId("4b2d75476cc613d5ee930164")},
  ... {"$addToSet" : {"emails" : "joe@gmail.com"}})

db.users.update(
  {"_id" : ObjectId("4b2d75476cc613d5ee930164")},
  ... {"$addToSet" : {"emails" : "joe@hotmail.com"}})

db.users.findOne(
  {"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
  "_id" : ObjectId("4b2d75476cc613d5ee930164"),
  "username" : "joe",
  "emails" : ["joe@example.com", "joe@gmail.com",
              "joe@yahoo.com", "joe@hotmail.com”]
}
```

49

Adding Multiple Unique Elements

```
db.users.update(
  {"_id" : ObjectId("4b2d75476cc613d5ee930164")},
  {"$addToSet" : ... {"emails" : {"$each" : ["joe@php.net",
                                             "joe@example.com",
                                             "joe@python.org"]}}})


db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
  "_id" : ObjectId("4b2d75476cc613d5ee930164"),
  "username" : "joe",
  "emails" : ["joe@example.com",
              "joe@gmail.com",
              "joe@yahoo.com",
              "joe@hotmail.com"
              "joe@php.net"
              "joe@python.org" ]
}
```

50

Removing List Elements

```
{$pop : {key : 1}}
{$pop : {key : -1}}

db.lists.insert(
  {"todo" : ["dishes", "laundry", "dry cleaning"]})

db.lists.update({}, {"$pull" : {"todo" : "laundry"}})

db.lists.find()
{
  "_id" : ObjectId("4b2d75476cc613d5ee930164"),
  "todo" : ["dishes", "dry cleaning"]
}
```

51

Positional Modifications

```
db.blog.posts.findOne()
{
  "_id" : ObjectId("4b329a216cc613d5ee930192"),
  "content" : "...",
  "comments" : [ ...
                 { "comment" : "good post",
                   "author" : "John",
                   "votes" : 0 },
                 ... ]
}

db.blog.update({"post" : post_id},
   ... {"$inc" : {"comments.0.votes" : 1}})

db.blog.update({"comments.author" : "John"},
   ... {"$set" : {"comments.$.author" : "Jim"}})
```

52

Non-Atomic Update

```
// check if we have an entry for this page
blog = db.analytics.findOne({url : "/blog"})

// if we do, add one to the number of views and save
if (blog) {
  blog.pageviews++;
  db.analytics.save(blog);
}
// otherwise, create a new document for this page
else {
  db.analytics.save({url : "/blog", pageviews : 1})
}
```

53

Atomic Update via Upsert Option

```
db.analytics.update({"url" : "/blog"},
                    {"$inc" : {"visits" : 1}},
                    true)

db.math.remove()
db.math.update({"count" : 25},
               {"$inc" : {"count" : 3}},
               true)
db.math.findOne()
{
  "_id" : ObjectId("4b3295f26cc613d5ee93018f"),
  "count" : 28
}
```

54

Multi-Updates

```
db.users.update(
  {birthday : "10/13/1978"},
  ... {$set : {gift : "Happy Birthday!"}},
  false,
  true)

db.runCommand({getLastError : 1})
{
  "err" : null,
  "updatedExisting" : true,
  "n" : 5,
  "ok" : true
}
```

55

Non-atomic Query & Update

```
{
  "_id" : ObjectId(),
  "status" : state,
  "priority" : N
}

ps = db.processes.find({"status" : "READY")
                  .sort({"priority" : -1})
                  .limit(1)
                  .next()
db.processes.update(
  {"_id" : ps._id},
  {"$set" : {"status" : "RUNNING"}})
do_something(ps);
db.processes.update(
  {"_id" : ps._id},
  {"$set" : {"status" : "DONE"}})
```

56

Atomic Query & Update

```
ps = db.runCommand(
  { "findAndModify" : "processes",
    ... "query" : {"status" : "READY"},
    ... "sort" : {"priority" : -1},
    ... "update" : {"$set" : {"status" : "RUNNING”}}
  }).value
{
  "ok" : 1,
  "value" : { "_id" : ObjectId("4b3e7a18005cab32be6291f7"),
              "priority" : 1,
              "status" : "READY"
            }
}
do_something(ps);
db.processes.update(
  {"_id" : ps._id},
  {"$set" : {"status" : "DONE"}})
```

57

FindAndModify Parameters

Key	Parameter
findAndModify	Collection name
query	Query document
sort	Critera for sorting results
update	Modifier document
remove	Boolean specifying whether to remove document
new	Boolean: return updated document or preupdated document. Default: preupdate.

58

Executing Update Commands

- Fire-and-forget
- Safe updates
  - Run `getLastError` immediately after
  - Problem: latency

59

QUERYING

60

Query

- Return all

```
db.users.find()
```

- Filter results

```
db.users
 .find({“dept” : “sales”, “location” : “nyc”})
```

- Project keys (_id always returned)

```
db.users
 .find({“dept” : “sales”, “location” : “nyc”},
       {“name” : 1, “email” : 1})
```

61

Comparison

- Operators:
  - “$lt”
  - “$lte”
  - “$gt”
  - “$gte”
  - “$ne”
- Example

```
db.users.find({“age” : {“$gte” : 18, “$lte” : 30}})
start = new Date(“01/01/2007”)
db.users.find({“registered” : {“$lt” : start}})
```

62

Comparison

- Set membership:

```
db.raffle.find(
   {“ticket_no” :
        {“$in” : [725, 542, 390]}})
```

- Disjunction:

```
db.raffle.find(
   {“$or” :
      [ {“ticket_no” :
           {“$in” : [725, 542, 390]}},
        {“winner” : true}
      ]
   })
```

63

Type-Specific Queries

- Regular Expressions

```
db.users.find({“name” : /joey?/i})
```

- Null
  - Matches itself

```
db.coll.find({“x” : null})
```

  - Also matches “does not exist”

```
db.coll.find({“x” : {“$in” : [null],
                     “$exists” : true}})
```

64

Querying Arrays

- Array: any element can match search key
- Insertion:

```
db.food.insert(
   {“fruit” : [“apple”, ”banana”, ”peach”]})
```

- Query:

```
db.food.insert({“fruit” : ”banana”})

db.food.insert({“fruit” :
                {“$all” : [“apple”, ”banana”]} })

db.food.insert({“fruit” : {“$size” : 3}})
```

65

Array Slicing

- First 10 comments:

```
db.blog.posts.findOne(criteria,
      {"comments" : {"$slice" : 10}})
```

- Last 10 comments:

```
db.blog.posts.findOne(criteria,
      {"comments" : {"$slice" : 10}})
```

- Range of comments:

```
db.blog.posts.findOne(criteria,
      {"comments" : {"$slice" : [23, 10]}})
```

66

Querying for Embedded Keys

- Example database:

```
{
  "name" : {
       "first" : "Joe",
        "last" : "Schmoe"
  },
  "age" : 45
}
```

- Query:

~~db.people.find({"name" :~~
~~{"first" : "Joe", "last" : "Schmoe"}})~~

```
db.people.find(
         {"name.first" : "Joe", "name.last" : "Schmoe"})
```

67

Embedded Document Matches

- Issue: embedded doc match must match the whole doc
- Example database:

```
{
  "content" : "...",
  "comments" :
      [ ... {“author” : “joe”, “score” : 3, ...} ...]
}
```

- Query:

~~db.blog.find({"comments" :~~
~~{"author" : "joe", "score" : {"$gte" : 5}}})~~
~~db.blog.find({"comments.author" : "joe",~~
~~"comments.score" : {"$gte" : 5}}),~~

```
db.people.find({“comments” :
             {“$elemMatch” :
             {”author" : "Joe", ”score" : {”$gte” : 5}}}})
```

$where Queries

- Example database:

```
db.foo.insert({"apple" : 1, "banana" : 6, "peach" : 3})
db.foo.insert({"apple" : 8, "spinach" : 4, ”banana" : 4})
```

- Query:

```
db.foo.find({“$where” : function () {
  for (var current in this) {
    for (var other in this) {
      if (current != other &&
          this[current] == this[other]) {
        return true;
      }
    }
  }
  return false;
}});
```

Cursors

- Assign result of database query:

```
var cursor = db.foo.find()
While (cursor.hasNext()) {
  obj = cursor.next();
  // do something
}
```

- Iterator interface:

```
Var cursor = db.people.find();
cursor.forEach(function(x) {
  print(x.name);
});
```

Cursor Options

- Options: `limit(), skip(), sort()`
- Add options using builder pattern

```
var cursor =
  db.people.find().sort({"x" : 1}).limit(1).skip(10);

var cursor =
  db.people.find().limit(1).sort({"x" : 1}).skip(10);

var cursor =
  db.people.find().skip(10).limit(1).sort({"x" : 1});
```

- Execute query:

```
cursor.forEach(function(x) {
  print(x.name);
});
```

Paginating without skip

- Avoid long skips - expensive

```
var page1 = db.foo.find(criteria).limit(100)
var page2 = db.foo.find(criteria).skip(100).limit(100)
var page3 = db.foo.find(criteria).skip(200).limit(100)
```

- Alternative: Keep track of current position via key

```
var page1 = db.foo.find().sort({"date" : -1}).limit(100)
var latest = null; // display first page
while (page1.hasNext()) {
  latest = page1.next();
  display(latest);
}
// get next page
var page2 =
  db.foo.find({"date" : {"$gt" : latest.date}});
page2.sort({"date" : -1}).limit(100);
```

Wrapped Queries

- Plain query

```
var cursor = db.foo.find({"foo" : "bar"})
```

- Wrapping

```
var cursor = db.foo.find({"foo" : "bar"}).sort({“x” : 1})
```

- Other options

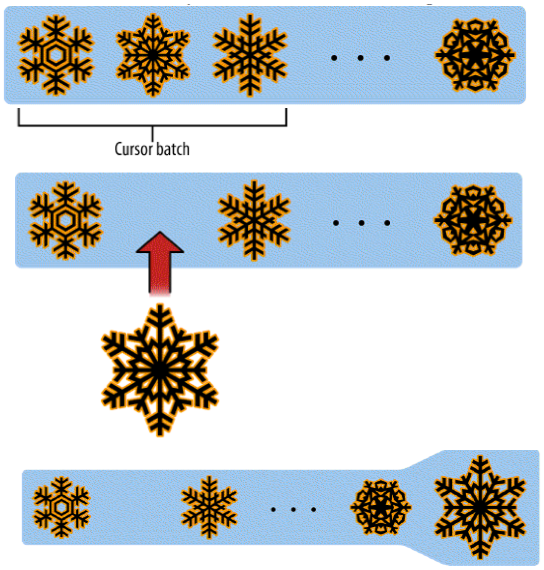
```
$maxscan : integer
$min : document
$max : document
$hint : document
$explain : boolean
$snapshot : boolean
```

`$snapshot` for Consistent Result

- Typical scenario:

```
cursor = db.foo.find();
while (cursor.hasNext()) {
  var doc = cursor.next();
  doc = process(doc);
  db.foo.save(doc);
}
```

$snapshot for Consistent Result




```
cursor =
  db.foo
    .find()
    .snapshot();
```

75

Indexes

- Rule of thumb: create index with all keys in query
- Example query:
  `db.people.find({"username" : "mark"})`
- Create index
  `db.people.ensureIndex({"username" : 1})`
- Example table scan
  ```
  db.people.find({"date" : date1})
            .sort({"date" : 1, "username" : 1})
  ```
- Create index
  `db.ensureIndex({"date" : 1, "username" : 1})`

76

AGGREGATION

77

Aggregation

- `count()`:
  - Number of documents in a collection
  - Number of results for a query
- `distinct()`:
  - All distinct values for a key
- `group()`:
  - Group for each key value, with result document

78

Group

- Example: Stock Prices database

```
{"day" : "2010/10/03",
 "time" : "10/3/2010 03:57:01 GMT-400",
 "price" : 4.23}
{"day" : "2010/10/04",
 "time" : "10/4/2010 11:28:39 GMT-400",
 "price" : 4.27}
{"day" : "2010/10/03",
 "time" : "10/3/2010 05:00:23 GMT-400",
 "price" : 4.10}
{"day" : "2010/10/06",
 "time" : "10/6/2010 05:27:58 GMT-400",
 "price" : 4.30}
{"day" : "2010/10/04",
 "time" : "10/4/2010 08:34:50 GMT-400",
 "price" : 4.01}
```

79

Group

- Latest stock prices

```
[
  {"time" : "10/3/2010 05:00:23 GMT-400",
   "price" : 4.10},
  {"time" : "10/4/2010 11:28:39 GMT-400",
   "price" : 4.27},
  {"time" : "10/6/2010 05:27:58 GMT-400",
   "price" : 4.30}
]
```

80

Group

- Query

```
db.runCommand({"group" : {
  "ns" : "stocks",
  "key" : "day",
  "initial" : {"time" : 0},
  "$reduce" : function(doc, prev) {
    if (doc.time > prev.time) {
      prev.price = doc.price;
      prev.time = doc.time;
    }
  },
  "condition" : {"day" : {"$gt" : "2010/09/30"}}
}})
```

81

Group

- Query

```
db.runCommand({"group" : {
  "ns" : "stocks",
  "key" : "day",
  "initial" : {"time" : 0},
  "$reduce" : function(doc, prev) {
    if (doc.time > prev.time) {
      prev.price = doc.price;
      prev.time = doc.time;
    }
  },
  "condition" : {"day" : {"$gt" : "2010/09/30”}
                         {“$exists” : true}}
}})
```

82

Group

- Result

```
{
  "retval" :
    [ {
        "day" : "2010/10/04",
        "time" : "Mon Oct 04 2010 11:28:39 ...”
        "price" : 4.27
      },
      ...
    ],
  "count" : 734,
  "keys" : 30,
  "ok" : 1
}
```

83

Group

- Example: Blog posts

```
{
  “day” : “2010/10/12”
  “tags" : [ “sledding”, “nosql” ]
}
{
  “day” : “2010/10/12”
  “tags" : [ “winter”, “nosql” ]
}
{
  “day” : “2010/10/13”
  “tags" : [ “php” ]
}
...
```

84

Group

- Query: For blog site, find most popular tag for day

```
db.posts.group({
  "key" : {”day" : true},
  "initial" : {"tags" : {}},
  "$reduce" : function(doc, prev) {
      for (i in doc.tags) {
        if (doc.tags[i] in prev.tags) {
          prev.tags[doc.tags[i]]++;
        } else {
          prev.tags[doc.tags[i]] = 1;
        }
      } ...
  }})
```

85

Group

- Result

```
[
  {"day" : "2010/01/12",
   "tags" :
      {"nosql" : 4, "winter" : 10, "sledding" : 2}},
  {"day" : "2010/01/13",
   "tags" : {"soda" : 5, "php" : 2}},
  {"day" : "2010/01/14",
   "tags" :
      {"python" : 6, "winter" : 4, "nosql": 15}}
]
```

86

Group

- Query: For blog site, find most popular tag for day

```
db.posts.group({
  "key" : {”day" : true},
  "initial" : {"tags" : {}},
  "$reduce" : function(doc, prev) { ... }
  "finalize" : function(prev) {
     var mostPopular = 0;
     for (i in prev.tags) {
       if (prev.tags[i] > mostPopular) {
         prev.tag = i;
         mostPopular = prev.tags[i];
       }
     }
     delete prev.tags
}})
```

87

Group

- Result

```
[
  {"day" : "2010/01/12", "tag" : "winter"},
  {"day" : "2010/01/13", "tag" : "soda"},
  {"day" : "2010/01/14", "tag" : "nosql”}
]
```

88

MAP-REDUCE

89

Map-Reduce

- Map
  - Parallel processing
  - `this`: reference to arg doc
  - `emit`: key and value result
- Reduce
  - Combine results under keys
  - Arguments:
    - Key
    - List of results

90

Example: Find all keys

```
map = function() {
        for (var key in this) {
          emit (key, {count : 1});
        }
      }

reduce = function(key, emits) {
           total = 0;
           for (var i in emits) {
             total += emits[i].count;
           }
           return {"count" : total};
         }
```

91

Result

```
mr = db.runCommand({"mapreduce" : "foo",
                    "map" : map, "reduce" : reduce})
{
  "result" : "tmp.mr.mapreduce_1266787811_1",
  "timeMillis" : 12,
  "counts" : {"input" : 6, "emit" : 14, "output" : 5
   },
   "ok" : true
}
db[mr.result].find()
{ "_id" : "_id", "value" : { "count" : 6 } }
{ "_id" : "a", "value" : { "count" : 4 } }
{ "_id" : "b", "value" : { "count" : 2 } }
{ "_id" : "x", "value" : { "count" : 1 } }
{ "_id" : "y", "value" : { "count" : 1 } }
```

92

Example: Categorize Web Pages

```
map = function() {
  for (var i in this.tags) {
    var recency = 1/(new Date() - this.date);
    var score = recency * this.score;
    emit(this.tags[i], {"urls" : [this.url],
                        "score" : score});
  }
};
reduce = function(url, emits) {
  total = 0;
  for (var i in emits) {
    total += emits[i].count;
  }
  return {"count" : total};
}
```

93

Optional keys to Map-Reduce

- "`finalize`" : function
- "`keeptemp`" : boolean
- "`output`" : string
- "`query`" : string
  - Filter before map
- "`sort`" : document
  - Sort before map
- "`limit`" : integer
  - # of inputs to map
- "`scope`" : document
- "`verbose`" : boolean

94

Optional keys to Map-Reduce

```
db.runCommand({"mapreduce" : "analytics",
  "map" : map, "reduce" : reduce,
  "query" : {"date" : {"$gt" : week_ago}}})

db.runCommand({"mapreduce" : "analytics",
  "map" : map, "reduce" : reduce,
  "limit" : 10000, "sort" : {"date" : -1}})

db.runCommand({"mapreduce" : "webpages",
  "map" : map, "reduce" : reduce,
  "scope" : {now : new Date()}})
```

95

Aggregation vs Map-Reduce

- Aggregation
  - Performed in-memory
  - Limits on size of results
- MapReduce
  - Intended for batch processing
  - No limits
  - Slow

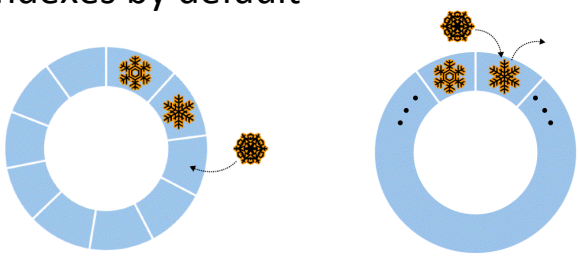
96

CAPPED COLLECTIONS

97

Capped Collections

- Organized as circular queue
- Items cannot increase in size
- No explicit delete
  - Age-out
- No indexes by default



98

Capped Collections

- Very fast insertion
  - Memcpy
- Fast query results
  - When sorted by insertion order (*natural sort*)
- Automatic aging-out of old data

- Application: *oplog* for replication (below)
- Application: caching small documents

99

Capped Collections

- Create a collection:

```
db.createCollection("my_collection",
     {capped: true, size: 100000});
```

- Limit # of docs:

```
db.createCollection("my_collection",
     {capped: true, size: 100000, max: 100});
```

- Convert existing collection:

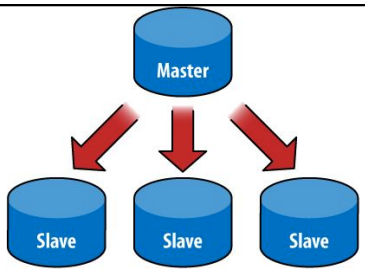
```
db.runCommand(
      {convertToCapped: "test",
       size: 10000});
```

100

REPLICATION

101

Master-Slave Replication



- Start the master

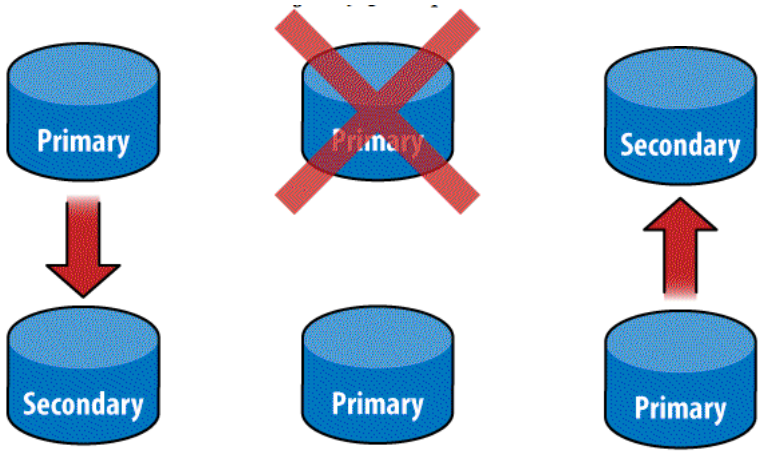
```
$ mkdir -p ~/dbs/master
$ ./mongod --dbpath ~/dbs/master --port 10000
–master
```

- Start a slave

```
$ mkdir -p ~/dbs/slave
$ ./mongod --dbpath ~/dbs/slave --port 10001
--slave --source localhost:10000
```

102

Replica Sets



103

Replica Sets

- Start a server in the replica set "blort"

```
$ mkdir -p ~/dbs/node1 ~/dbs/node2
$ ./mongod --dbpath ~/dbs/node1 --port 10001
           --replSet blort/morton:10002
```

- Start a second server

```
$ ./mongod --dbpath ~/dbs/node2 --port 10002
           --replSet blort/morton:10001
```

- Start a third server

```
$ ./mongod --dbpath ~/dbs/node3 --port 10003
           --replSet blort/morton:10001
$ ./mongod --dbpath ~/dbs/node3 --port 10003
        --replSet blort/morton:10001,morton:10002
```

104

Initializing a Replica Set

```
$ ./mongo morton:10001/admin
MongoDB shell version: 1.5.3
connecting to localhost:10001/admin
> db.runCommand({"replSetInitiate" : {
 "_id" : "blort",
 "members" : [
    {
      "_id" : 1,
      "host" : "morton:10001”
    },
    {
      "_id" : 2,
      "host" : "morton:10002”
    }
  ]
}})
```

105

Types of Nodes

- Standard
  - Replicate data
  - May become primary
  - Participates in voting for primary
- Passive
  - Replicate data
  - Participates in voting
- Arbiter
  - Voting only

106

Node Priority

- Priority = 0 ⇒ passive node
- Priority > 0 ⇒ primary based on priority
  - Freshness of data breaks ties

```
members.push({ "_id" : 3,
               "host" : "morton:10003",
               "priority" : 40  });

members.push({ "_id" : 4,
               "host" : "morton:10004",
               "arbiterOnly" : true ... });
```

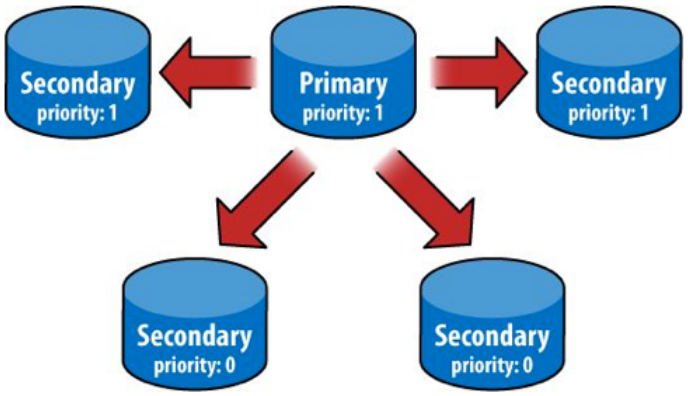
107

Failover and Primary Election

- Primary: track nodes using heartbeat
  - No quorum ⇒ fall back to secondary
  - Prevent *split brain* (network partition)
- Primary assumed most up-to-date
  - Recovery: nodes *resync* with new primary
  - Later ops rolled back, up-to-date copy from primary

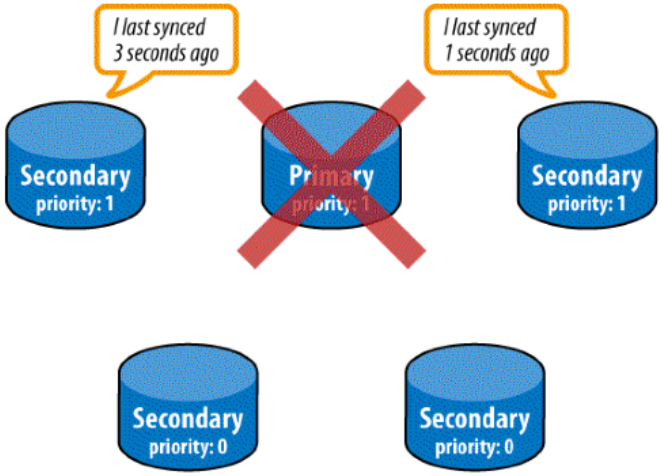
108

Failover and Primary Election



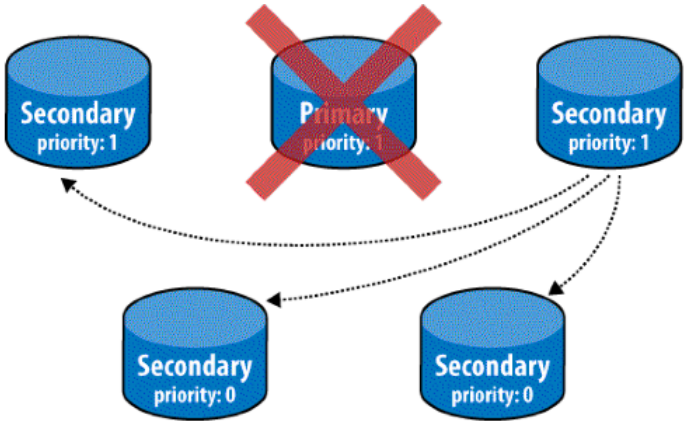
109

Failover and Primary Election



110

Failover and Primary Election



111

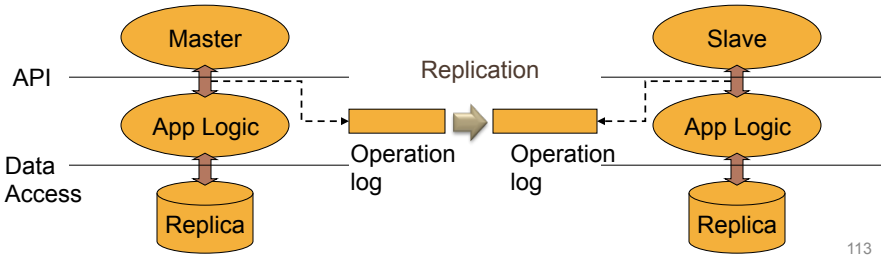
Slave Use Cases

- Backing up data
- Read scaling
  - Send queries directly to slave
  - `slaveOkay` query option
- Off-loading data processing
  - Run slave with both `--slave` and `--master`
  - Distinguish locally updated & mirrored data

112

How Replication is done

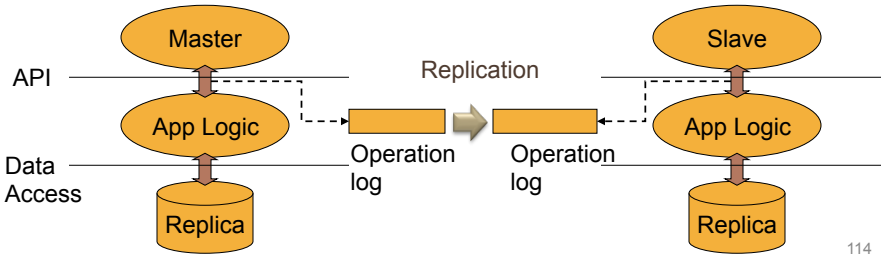
- Oplog: operation log
  - Logs include timestamps
  - Ops transformed to be idempotent
    e.g. x++ ⇒ x=3
  - Stored in capped collection



113

How Replication is done

- Syncing:
  - Initially slave copies all data
  - Thereafter queries oplog
  - Out of sync: slave too far behind
    - Avoid: ensure oplog is large enough



114

Blocking for Replication

- Provide guarantee about replication

```
db.runCommand({getLastError: 1, w: N});
```

  - Wait for N replicas to ack (incl master)
  - N < 2 ⇒ don't block
  - N = 2 ⇒ block until one slave acks

- Tradeoff: reliability vs performance
  - Pick N =2 or N = 3

115

SHARDING

116

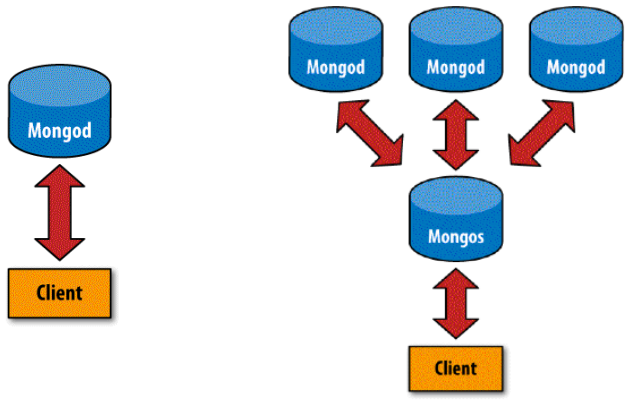
Sharding

- Manual sharding
  - Connections to several databases
  - Adding/removing nodes
  - Redistributing data
- Autosharding
  - Automatic data splitting & distribution
  - Handled by cluster

117

Autosharding in MongoDB

- Break up data into chunks
- Router (mongos) routes requests



118

Shard Keys

- Define shard key e.g. timestamp
- Partition data based on ranges
- Incrementing shard keys

  $[t_0,t_1), [t_1,t_2), \ldots, [t_k,\infty)$

  $\Rightarrow [t_0,t_1), [t_1,t_2), \ldots, [t_k,t_{k+1}), [t_{k+1},\infty)$
- Random shard keys
  - Uniformly distribute high write load
  - E.g. hash of timestamp
  - Similar to choosing keys

119

Shard Keys

- Suppose collection sharded on name key:
  - A-F, G-P, Q-Z
- Example operations

  `db.people.find({"name" : "Susan"})`
  - Send to Q-Z shard

  `db.people.find({"name" : {"$lt" : "L"}})`
  - Send to A-F, then G-P, responses to client

  `db.people.find().sort({"email" : 1})`
  - Query all shards, merge sort of results
  - Cursors for each shard, batched results to client

  `db.people.find({"email" : "joe@example.com"})`
  - Send to all shard

120

Setting up Sharding

- Run config server
```
$ mkdir -p ~/dbs/config
$ ./mongod --dbpath ~/dbs/config --port 20000
```
- Run router
```
$ ./mongos --port 30000 --configdb localhost:20000
```
- Run shard
```
$ mkdir -p ~/dbs/shard1
$ ./mongod --dbpath ~/dbs/shard1 --port 10000
```
- Connect shard to cluster
```
$ ./mongo localhost:30000/admin
> db.runCommand({addshard : "localhost:10000",
                 allowLocal : true})
```

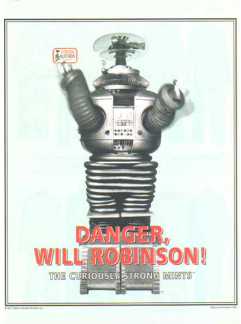
121

Sharding Data

- Ex: Shard `bar` collection in `foo` database on `_id` key
- Enable sharding
```
db.runCommand({"enablesharding" : "foo"})
```
- Shard the collection
```
db.runCommand({"shardcollection" : "foo.bar",
               "key" : {"_id" : 1}})
```
- Run shard

122

Robust Config



- Multiple config servers
  - Connected to router (mongos)

```
./mongos --configdb localhost:20001,
                    localhost:20002,
                    localhost:20003
```

  - Synchronize using 2PC
- Multiple routers
  - Ex: one router per app server
- Replicated shards

```
db.runCommand({"addshard" :
               "foo/prod.example.com:27017"})
```

123

MONGODB C# DRIVER

124

Namespaces

- Minimum
```
using MongoDB.Bson;
using MongoDB.Driver;
```
- Additional:
```
using MongoDB.Driver.Builders;
using MongoDB.Driver.GridFS;
using MongoDB.Driver.Linq;
```
- Optional:
```
using MongoDB.Bson.IO;
using MongoDB.Bson.Serialization;
using MongoDB.Bson.Serialization.Attributes;
using MongoDB.Bson.Serialization.Conventions;
using MongoDB.Bson.Serialization.IdGenerators;
using MongoDB.Bson.Serialization.Options;
using MongoDB.Bson.Serialization.Serializers;
using MongoDB.Driver.Wrappers;
```

125

Connection String

- General format:
```
mongodb://[username:password@]hostname[:port]
          [/[database][?options]]
mongodb://hostname
mongodb://username:password@hostname
mongodb://hostname/database
```
- Connect to several servers (replica set or shards):
```
mongodb://server1,server2:27017,server2:27018
mongodb://server1,server2:27017,server2:27018?
          connect=replicaset
```
- Connect directly to an instance:
```
mongodb://server2?
          connect=direct;readpreference=nearest
```

126

Authentication

- Specify credential store (`MongoCredentialStore`)

```
var url = MongoUrl.Create(
   "mongodb://test:user@localhost:27017/?safe=true");
var settings = url.ToServerSettings();
var adminCredentials =
  new MongoCredentials("admin", "user", true);
settings.CredentialsStore
        .Add("admin", adminCredentials);

var fooCredentials =
  new MongoCredentials("foo", "user", false);
settings.CredentialsStore
        .Add("foo", fooCredentials);

var server = MongoServer.Create(settings);
```

127

Model

- Example model:

```
public class CredentialSet
{
    [BsonId]
    public ObjectId _id { get; set; }
    public string Title { get; set; }
    public string Username { get; set; }
    public string Password { get; set; }
    public string WebSite { get; set; }
    public string Notes { get; set; }
    public int Owner { get; set; }
    public DateTime LastUpdate { get; set; }
}
```

128

Accessing Collection

- Connect to database

```
MongoServer server =
    MongoServer.Create(“mongodb://myserver”);
MongoDatabase db =
    server.GetDatabase("TheDatabase");
```

- Get reference to collection

```
MongoCollection passwords =
    db.GetCollection(“passwords”);
```

129

Accessing Collection

- Connect to database

```
MongoServer server =
    MongoServer.Create(“mongodb://myserver”);
MongoDatabase db =
    server.GetDatabase("TheDatabase");
```

- Get reference to collection

```
MongoCollection<CredentialSet> passwords =
    db.GetCollection<CredentialSet>(“passwords”);
```

- Save (upsert) document to collection

```
var password = new CredentialSet();
// set property values
passwords.Save(password);
```

130

Insertion

- Using a BSON class

```
MongoCollection<BsonDocument> books =
    database.GetCollection<BsonDocument>("books");
BsonDocument book = new BsonDocument {
    { "author", "Ernest Hemingway" },
    { "title", "For Whom the Bell Tolls" }
};
books.Insert(book);
```

- Using a model class

```
MongoCollection<Book> books =
database.GetCollection<Book>("books");
Book book = new Book {
    Author = "Ernest Hemingway",
    Title = "For Whom the Bell Tolls"
};
books.Insert(book);
```

131

Retrieval

- Find one document

```
MongoCollection<Book> books;
Book book = books.FindOne();
```

- Find doc & override returned type

```
MongoCollection<Book> books;
BsonDocument document =
     books.FindOneAs<BsonDocument>();
```

132

Queries

- Query must implement IMongoQuery
  - Query builder class
  - Instance of QueryDocument
  - QueryWrapper class

133

Query Document

- Instance of QueryDocument

```
MongoCollection<BsonDocument> books;
var query =
  new QueryDocument("author", "Kurt Vonnegut");
foreach (BsonDocument book in books.Find(query)) {
    // do something with book
}
```

- Query Builder

```
MongoCollection<BsonDocument> books;
var query = Query.EQ("author", "Kurt Vonnegut");
foreach (BsonDocument book in books.Find(query)) {
    // do something with book
}
```

134

Query Document

- Query wrapper

```
MongoCollection<BsonDocument> books;
var query =
  Query.Wrap(new { author = "Kurt Vonnegut" });
foreach (BsonDocument book in books.Find(query)) {
    // do something with book
}
```

- Query wrapper with type override

```
MongoCollection<BsonDocument> books;
var query = Query.EQ("author", "Kurt Vonnegut");
foreach (Book book in books.FindAs<Book>(query)) {
    // do something with book
}
```

135

FindAndModify

- Example

```
{ findAndModify: "people",
  query:
    { name: "Tom", state: "active", rating: { $gt: 10 } },
  sort: { rating: 1 },
  update: { $inc: { score: 1 } }
}
```

- C# version

```
var jobs = database.GetCollection("jobs");
var query = Query.And(Query.EQ("inprogress", false),
                      Query.EQ("name", "Biz report”));
var sortBy = SortBy.Descending("priority");
var update = Update.Set("inprogress", true)
                   .Set("started", DateTime.UtcNow);
var result = jobs.FindAndModify(query, sortBy, update,
    true /* return new document */ );
var chosenJob = result.ModifiedDocument;
```

136

MapReduce

```
var map =
    "function() {" +
    "    for (var key in this) {" +
    "        emit(key, { count : 1 });" +
    "    }" +
    "}”;
var reduce =
    "function(key, emits) {" +
    "    total = 0;" +
    "    for (var i in emits) {" +
    "        total += emits[i].count;" +
    "    }" +
    "    return { count : total };" +
    "}”;
var mr = collection.MapReduce(map, reduce);
foreach (var document in mr.GetResults()) {
    Console.WriteLine(document.ToJson());
}
```

137

Cursor

- Create and customize

```
var query =
   Query.EQ("author", "Ernest Hemingway");
var cursor = books.Find(query);
cursor.Skip = 100;
cursor.Limit = 10;
```

- Enumerate using foreach

```
foreach (var book in cursor) {
    // do something with book
}
```

- Query using LINQ extension methods

```
var firstBook = cursor.FirstOrDefault();
var lastBook = cursor.LastOrDefault();
```

138

DOCUMENTDB

139

DocumentDB Resources

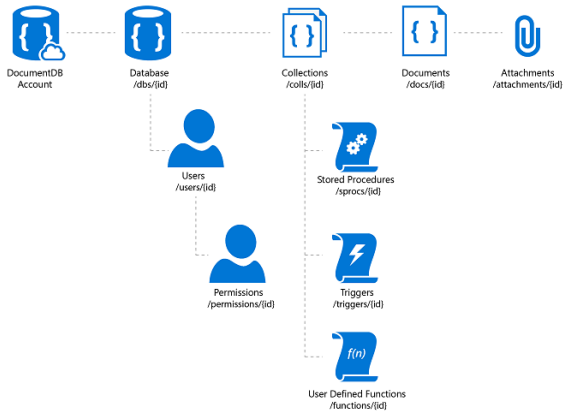


Image © Microsoft

140

DocumentDB Database

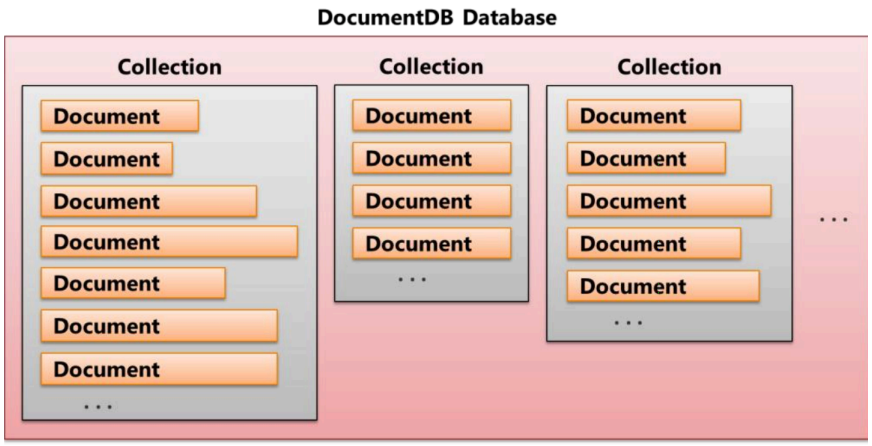


Image © Chappell Associates

141

DocumentDB API

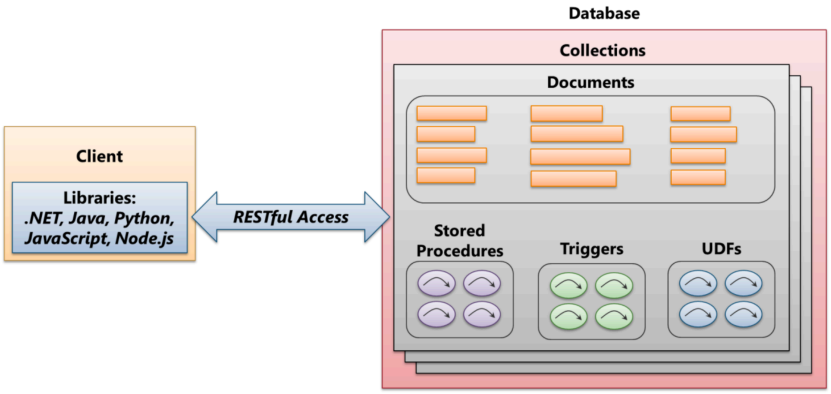


Image © Chappell Associates

142

DocumentDB API

- RESTful API
  - GET: Retrieve document
  - PUT: Replace document
  - POST: Create document
    - Also perform SQL request
  - DELETE: Remove document

143

DocumentDB API

- Stored Procedures
- Triggers
- User-Defined Functions (UDFs)

144

DocumentDB API

- Stored Procedures
  - Written in Javascript
  - Materialize JSON document into variables
  - Transactional execution
- Triggers
- User-Defined Functions (UDFs)

145

Stored Procedures (sprocs)

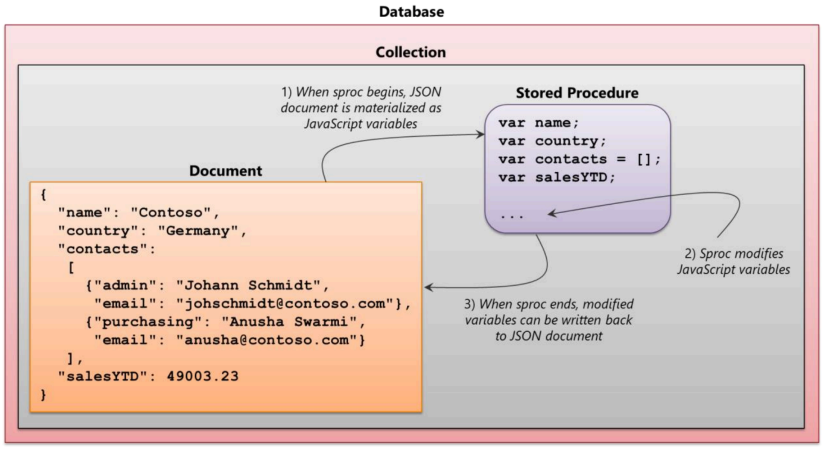


Image © Chappell Associates

146

DocumentDB API

- Stored Procedures
- Triggers
  - Like sprocs (JS, transactional)
  - React to events
  - Pre-trigger e.g. validate update request
  - Post-trigger e.g. log update
- User-Defined Functions (UDFs)

147

DocumentDB API

- Stored Procedures
- Triggers
- User-Defined Functions (UDFs)
  - Read-only
  - Extend DDB SQL

```
SELECT *
FROM customers c
WHERE udf.calculateTax(c.salesYTD) > 1000
```

148

DOCUMENTDB SQL

149

Example Database

```
{
    "id": “JonesFamily",
    "lastName": “Jones”,
    "parents": [
       { "firstName": “Joe” },
       { "firstName": “Jane”}
    ],
    "children": [
       {
           "firstName": “Margaret”, "gender": “female”, "grade”:5
       }
    ],
    "address": {
       "state": “NJ”,
       ”county": “Hudson”
       "city": “Hoboken”
    },
    "creationDate": 1431620472
}
```

150

Example Database

```
{
    "id": “SmithFamily",
    "lastName": “Smith”,
    "parents": [
       { "firstName": “Sam” },
       { "firstName": “Sara”}
    ],
    "children": [
       { "firstName": “Peter”, "gender": “male”, "grade" : 8 }
       { "firstName": “Paul”, "gender": “male”, "grade" : 3 }
    ],
    "address": {
       "state": “OH”,
       ”county": “Cuyoga”
       "city": “Cleveland”
    },
    "creationDate": 1431620472
}
```

151

Example Queries

- Select with filters

```
SELECT *
FROM Families f
WHERE f.lastName = “Jones”
```

- Nested projection

```
SELECT *
FROM Families f
WHERE f.address.state = “NJ”
```

152

Example Queries

- Select with filters

```
SELECT f.address
FROM Families f
WHERE f.lastName = “Jones”
```

- Nested projection

```
SELECT f.address.state
FROM Families f
WHERE f.lastName = “Jones”
```

153

Example Queries

- Select with column filtering

```
SELECT f.address.state, f.address.city
FROM Families f
WHERE f.lastName = “Jones”
```

- Result:

```
[
  {
    "state": “NJ”,
    "city": “Hoboken”
  }
]
```

154

Example Queries

- Select with synthetic result object

```
SELECT { "state" : f.address.state,
         "city" : f.address.city }
FROM Families f
WHERE f.lastName = “Jones”
```

- Result:

```
[
  { "$1" : {
      "state": “NJ”,
      "city": “Hoboken”
    }
  }
]
```

155

Example Queries

- Select with column aliasing

```
SELECT { "state" : f.address.state,
         "city" : f.address.city } AS AddressInfo
FROM Families f
WHERE f.lastName = “Jones”
```

- Result:

```
[
  { "AddressInfo" : {
      "state": “NJ”,
      "city": “Hoboken”
    }
  }
]
```

156

Iteration

- Select with list results

```
SELECT *
FROM Families.children
```

- Result:

```
[
 [
  { "firstName”:“Margaret”, "gender”:“female”, "grade”:5 }
 ],
 [
  { "firstName": “Peter”, "gender": “male”, "grade": 8 }
  { "firstName": “Paul”, "gender": “male”, "grade": 3 }
 ]
]
```

157

Iteration

- Select with iteration over list results

```
SELECT *
FROM c IN Families.children
```

- Result:

```
[
  { "firstName”:“Margaret”, "gender”:“female”, "grade”:5 }
  { "firstName": “Peter”, "gender": “male”, "grade": 8 }
  { "firstName": “Paul”, "gender": “male”, "grade": 3 }
]
```

158

Iteration

- Select with iteration over list results

```
SELECT c.firstName
FROM c IN Families.children
WHERE c.grade = 8
```

- Result:

```
[
 { "firstName": “Peter” },
]
```

159

JOIN

- Cross-product (intra-document only)

```
SELECT f.id as familyName,
       c.firstName as childName
FROM Families f
JOIN c IN Families.children
```

- Pseudo-code:

```
foreach f in Families
  foreach c in f.children
    return new Tuple(...)
```

160

JOIN

- Cross-product (intra-document only)

```
SELECT f.id as familyName,
       c.firstName as childName
FROM Families f
JOIN c IN Families.children
```

- Result:

```
[
  { "firstName": “Margaret”, "familyName": “Jones” }
  { "firstName": “Peter”, "familyName": “Smith” }
  { "firstName": “Peter”, "familyName": “Smith” }
]
```

161

Parameterized Queries

- Motivation: Prevent SQL Injection attacks
- Example:

```
{
  "query": "SELECT * FROM Families f WHERE
f.lastName = @lastName AND f.address.state =
@addressState",
  "parameters": [
     {"name": "@lastName", "value": "Jones"},
     {"name": "@addressState", "value": "NJ"},
    ]
}
```

162

.NET API

163

Creating Documents

```
using (client = new DocumentClient(new Uri(endpoint),
                                   authKey))
{
  var database = new Database { Id = ”FamilyDB" };
  database = await client
    .CreateDatabaseAsync(database);

  var collection =
    new DocumentCollection { Id = "Families" };
  collection = await client
    .CreateDocumentCollectionAsync(database.SelfLink,
                                   collection);

  await client.CreateDocumentAsync(collection.SelfLink,
                                   smithFamily);
  await client.CreateDocumentAsync(collection.SelfLink,
                                   jonesFamily);
```

164

Querying Documents

```
using (client = new DocumentClient(new Uri(endpoint),
                                   authKey))
{
  ...
  query = client.
    CreateDocumentQuery(collection.DocumentsLink,
      "SELECT * FROM c IN Families.children");
  var children = query.AsEnumerable();

  foreach (var child in children) {
    Console.WriteLine(”Child named {0}", child.firstName);
  }

  //cleanup test database
  await client.DeleteDatabaseAsync(database.SelfLink);
}
```

165

Self-Link

- System Properties for every Resource:

```
{
  "id": "52cdef7c4bab8bd675297d8b",

  "_rid": "tyhbAN053QABAAAAAAAAAA==",
  "_ts": 1436843806,
  "_self": "dbs/tyhbAA==/colls/tyhbAN053QA=/docs/tyhbAN053QABAAAAAAAAAA==/",
  "_etag": "01006199-0000-0000-0000-55a47f1e0000",
  "_attachments": "attachments/"
}
```

- Resource id
  - System-defined address
- Self link: URI of the form
  - dbs/*db_rid*/colls/*coll_rid*/docs/*doc_rid*/

166

Example: Deleting a document

```
Database db = client.CreateDatabaseQuery()
                    .Where(d => d.Id == "FamilyDb")
                    .AsEnumerable()
                    .Single();

DocumentCollection coll =
   client.CreateDocumentCollectionQuery(db.SelfLink)
                     .Where(c => c.Id == ”Families")
                     .AsEnumerable()
                     .Single();

Document doc = client.CreateDocumentQuery(coll.SelfLink)
                     .Where(f => f.Id == "Jones")
                     .AsEnumerable()
                     .Single();

await client.DeleteDocumentAsync(doc.SelfLink);
```

167

Avoiding Self Link

NB No trailing “/”!

- Build up the self link manually

```
var docLink =
  string.Format("dbs/{0}/colls/{1}/docs/{2}",
                ”FamilyDb", ”Families", ”Jones");
await client.DeleteDocumentAsync(docLink);
```

- Use a library factory method

```
Uri docUri = UriFactory.CreateDocumentUri(",
                ”FamilyDb", ”Families", ”Jones");
await client.DeleteDocumentAsync(docUri);
```

168

User-Defined Function

```
UserDefinedFunction regexMatchUdf =
  new UserDefinedFunction
     {
       Id = "REGEX_MATCH",
       Body = @"function (input, pattern) {
                   return input.match(pattern) !== null;
                };",
     };

UserDefinedFunction createdUdf =
  client.CreateUserDefinedFunctionAsync(
     collectionSelfLink /* link of the parent collection*/,
     regexMatchUdf)
  .Result;
```

169

User-Defined Function

```
UserDefinedFunction regexMatchUdf =
  new UserDefinedFunction
     {
       Id = "REGEX_MATCH",
       Body = @"function (input, pattern) {
                   return input.match(pattern) !== null;
                };",
     };


SELECT Families.id, Families.address.city
FROM Families
WHERE udf.REGEX_MATCH(Families.address.city, "Hob.*")
```

170

LINQ Queries

- DDB SQL:

```
SELECT { "state" : f.address.state,
         "city" : f.address.city }
FROM Families f
```

- LINQ:

```
Input
  .Select(family => new {
    state = family.address.state,
    city = family.address.city
   });
```

171

LINQ Queries

- DDB SQL

```
SELECT { "state" : f.address.state,
         "city" : f.address.city }
FROM Families f
WHERE f.lastName = “Jones”
```

- LINQ:

```
Input
  .Select(family => new {
     state = family.address.state,
     city = family.address.city
   })
  .Where(lastName == “Jones”);
```

172

Nested Queries

- DDB SQL

```
SELECT f.id as familyName,
       c.firstName as childName
FROM Families f
JOIN c IN Families.children
```

- LINQ:

```
Input.SelectMany(family =>
  family
    .children
    .Select(child => new {
      familyName = family.id,
      childName = child.firstName
    })
```

173

Conclusions

- Document-oriented databases
  - Designed for replication and scale
  - Semi-structured (no schema)
- MongoDB
  - Not transactional
  - Ad-hoc queries
- DocumentDB
  - Transactional stored procedures
  - Extended SQL (intra-document joins)

174