NoSQL and MongoDB

Objectives :

- **Introducing Big Data**
- **Introducing NoSQL**
- **MongoDB**

What is Big Data? 1

What's in a name.... 1.1

is pre-existing data small?

is size the only challenge when dealing with Big Data?

Big Data [just google for more definitions]**:**

Information that can not be processed or analysed using *traditional* processes or tools.

or

A *new* generation of technologies and architectures, designed to economically extract value from very large volumes of a wide variety of data, by enabling high velocity capture, discovery, and/or analysis.

or

...

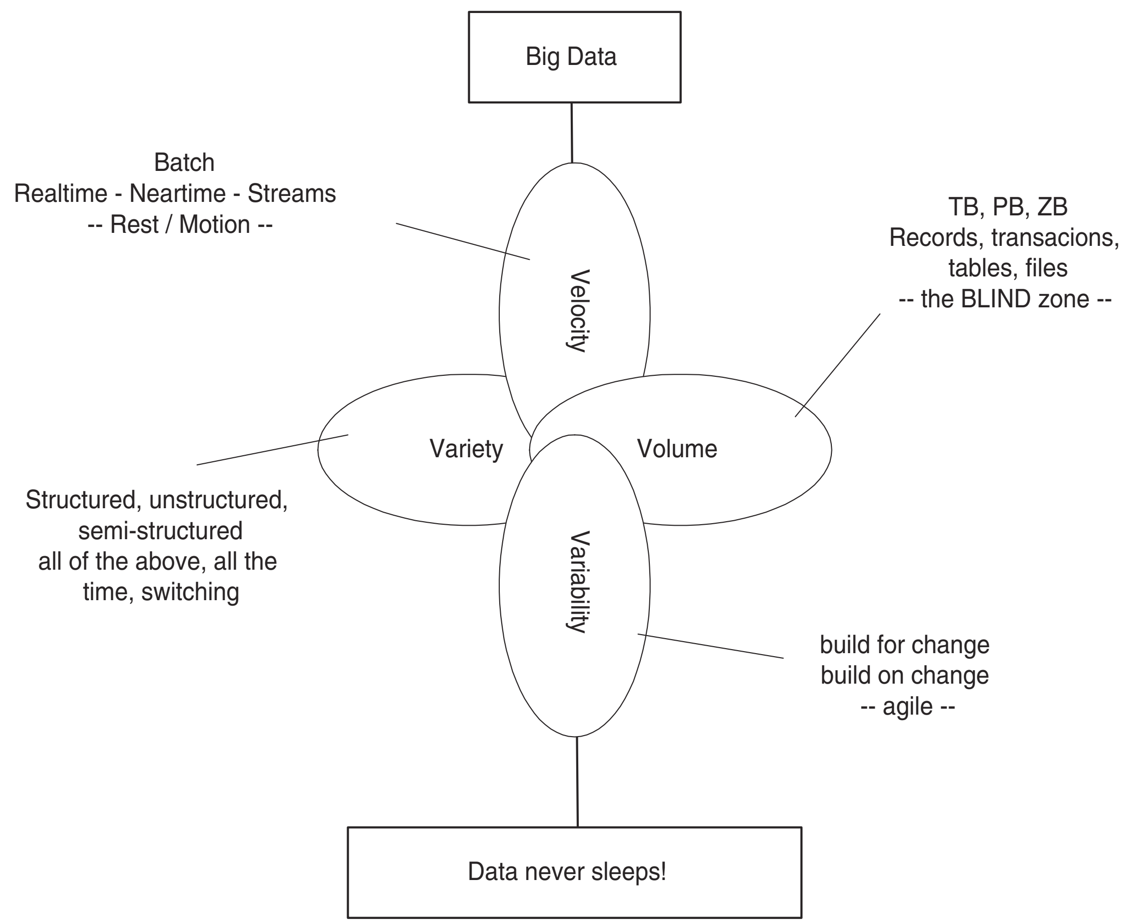
The consequences of *change* 1.2

We, you, the world, everything is changing!

- **instrumentation**
  [sensors]
- **inter-connectivity**
  - humans - social media, micro blogging, and the like
  - machines - M2M
    [smart metering]
- **intelligence**
  [ever so small microships are added *everywhere*!]

RFID

The consequences of *change (II)*



The consequences of *change* (III)

Observation: more data becomes available; less data gets analysed and turned into information!

- because we do not know it IS available - or understand it is ‘relevant’
- because we do not have the TOOLS to analyse that data

DW/BI - ETT/ETL:

- **processes can not keep up with the streams (volumes)/generation rate/nature/structure/... of the data to be processed for storage in a DW/DM/ODS**
- **processes ‘massage’ data; analysis tools are biased as result of that**

[Data Vault]

- **Data is stored in an aggregated format - the ‘*grain*’ should be specified at a more detailed level**
- **Database engines can NOT cope with the amount of data to be stored - massive computing strength**

Technological impact: scale up - scale out

- **Scale up**

  remove I/O constraints to improve CPU consistency
  [perhaps using RAM storage caches]

  typically a 'shared something' architecture
  [shared disk?]

  most frequently used today

  As volumes increases, volatility increases, ...
  - tiered storage - distinct storage models
  - remove redundancy
  - selective retention
  - sampling
  - compression
  - parallel processing
  - re-evaluate RI, constraints, normalisation
  - etc

Technological impact: scale up - scale out (II)

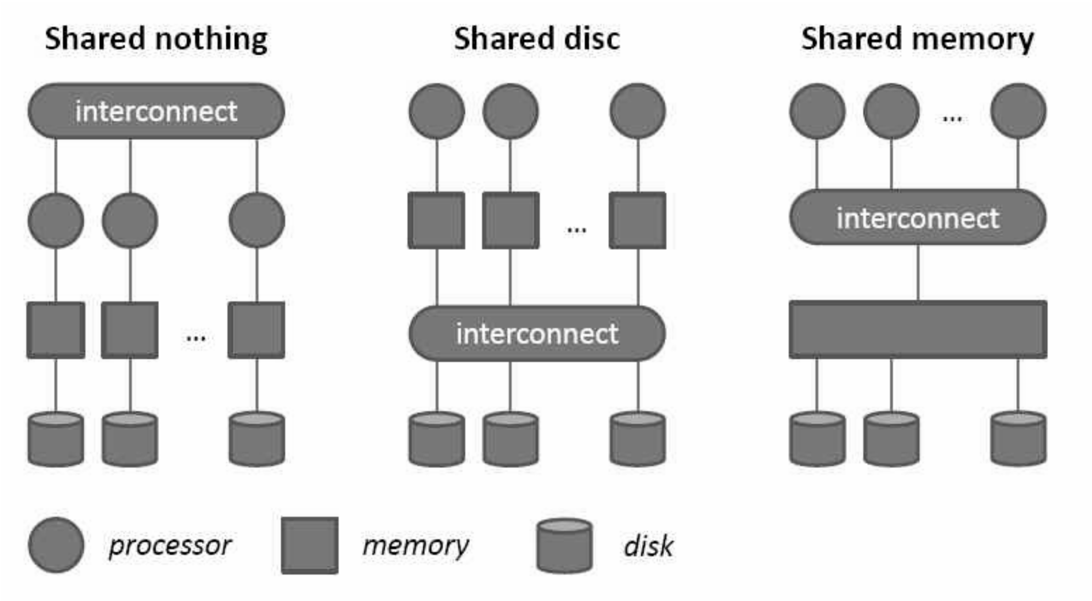
- **Scale out**

  combine 'commodity hardware' servers/clusters/racks

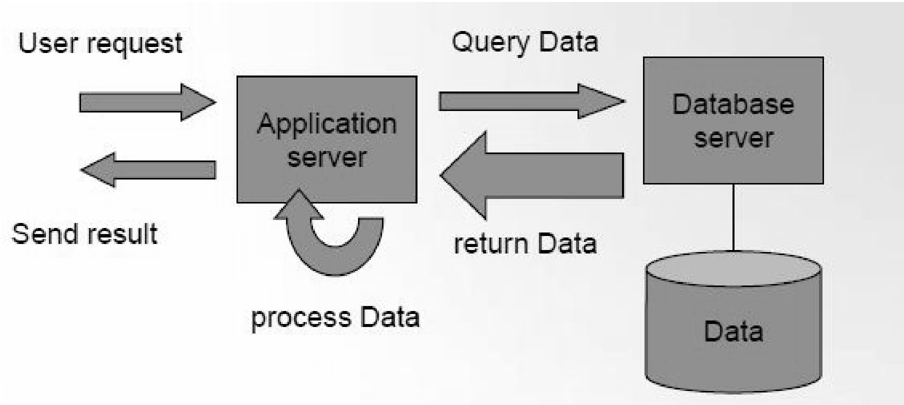
  typically a 'shared nothing architecture'

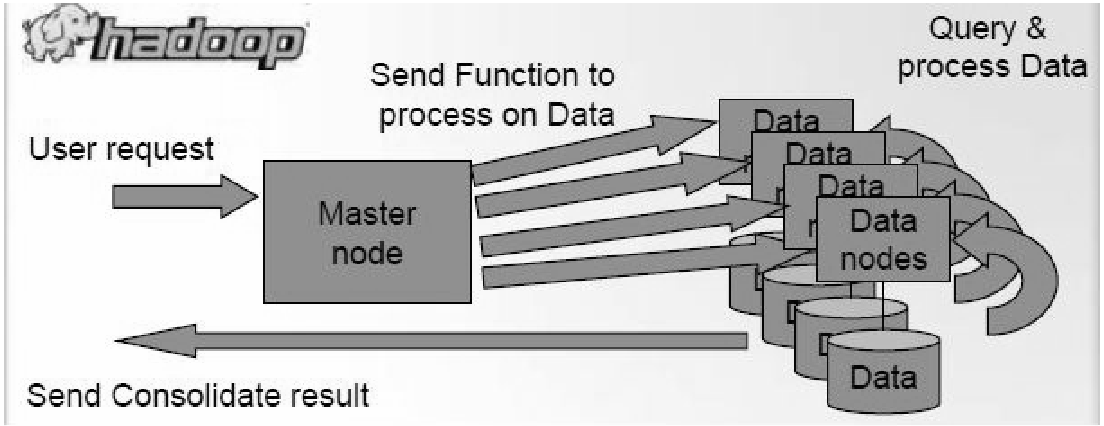
  - functional scaling
    [one server per function idea]
  - sharding
    [multiple server 'serve' a function
  - partitioning is 'key' - use eg. replication for scaling and availability

Technological impact: scale up - scale out (III)



Technological impact: scale up - scale out (IV)





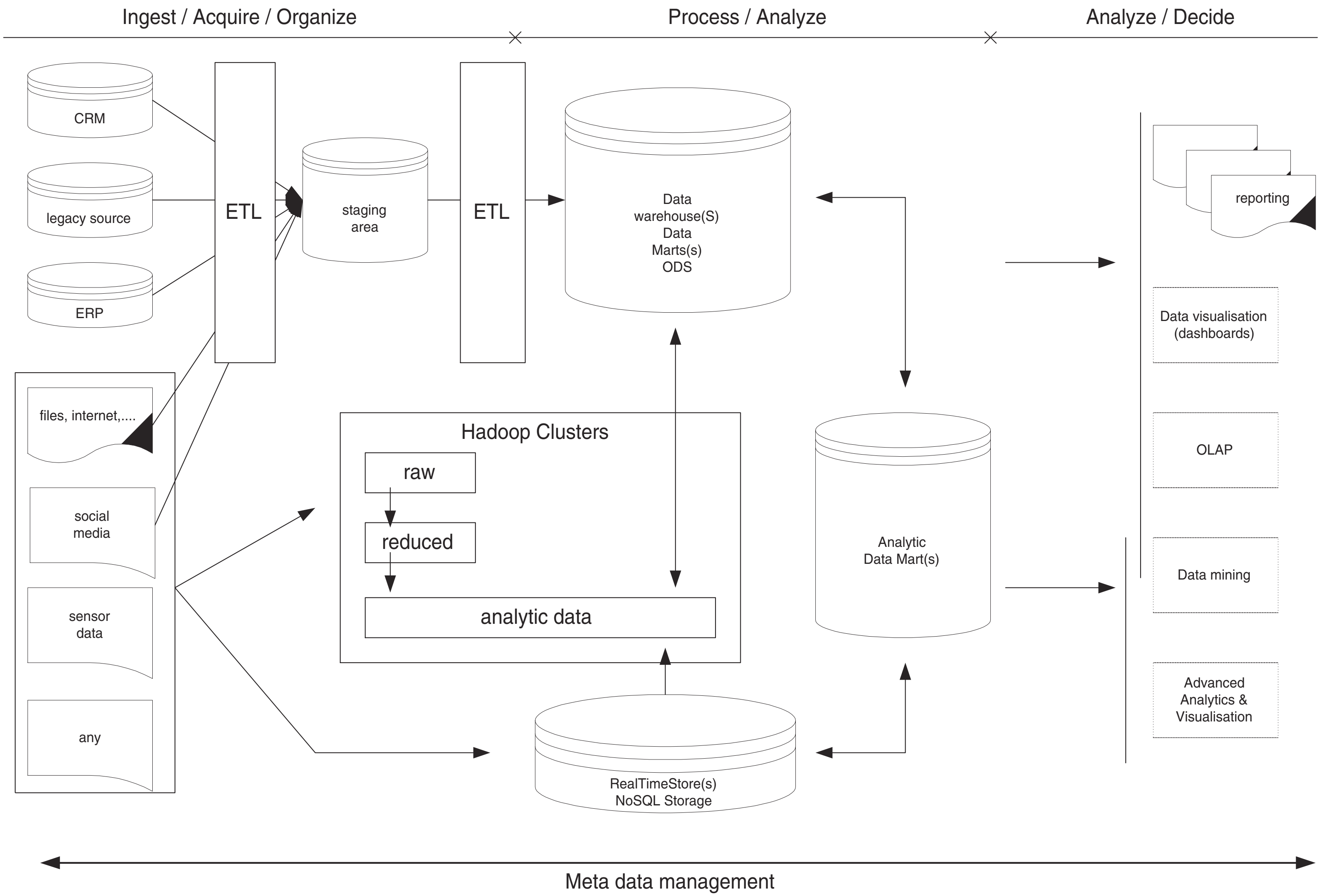
Enterprise Data Architecture

- **Data Warehouse**
  - **analyses** structured data **from** structured sources
  - **insight into know, stable structures and measurements**
    [built with questions in mind]
  - **extensive quality control - ETL**
  - **data is ‘public’**

  high known value per byte!

- **Big Data Warehouse / Big Data add-on**
  [for lack of a better phrase]
  - **semi-structured/unstructured data -** analysis and discovery
    [built with discovery in mind]
  - **less/no quality control**
  - **data is ‘not’ public**

  low know value per byte!



NoSQL and MongoDB

Business Intelligence 2.0

The processes, techniques, and tools that support business decision making based on information technology - offering users what they need to make informed decisions!

A combination of technologies:

- **Data Warehousing (DW)** ***(make available)***
- **Big Data extensions** ***(make available)***
- **BI 'Tools' & 'Technologies'** ***(enable)***
    - On-Line Analytical Processing
    - Data Mining
    - Data Visualization - Decision analysis (what-if)
    - CRM
    - Scorecards, Dashboards
    - Advanced Analytics
    - ...

A shift in approach - or a continuous evolution?

Traditional approach: structured and repeatable analysis

- Business decides what questions to ask
- IT builds the appropriate environment
  [eg. structures data, ...]

BI approach: iterative and exploratory analysis

- IT provides an infrastructure suitable for user explorations
- Business explores and investigates - what is there to asks and discover?

DW - reporting, structure, ‘end user’ oriented?

Big Data - analysis, discovery, ‘power user’ oriented?
[business knowledge, statistician, ...]

New 'storage' systems have emerged to address requirements of 'Big Data' data management

NoSQL data stores - ie.

- **Not Only SQL data stores**
- **NoSQL data stores**

In short:

- **scalable SQL databases, horizontal scaling (shared nothing architectures)**
- **replicating and partitioning data over thousands of nodes**
- **distribute "simple operation" workload over thousands of nodes** (key lookups, read and writes a small number of records, no complex queries/joins)

Multiple types
[not all are introduced below - see http://nosql-database.org/]

What is the problem with relational databases?

2.1

P#1: You have to convert all your information from their natural representations into tables

P#2: You have to reconstruct your information from tabular data

P#3: You have to model your data into tables before you can store it

P#4: Columns of tables can only store similar data

P#5: Relational systems may not scale as well other systems

P#6: Joins between foreign systems with different record identifiers tend to be difficult

P#7: SQL dialects vary making it difficult to port applications between databases

P#8: Complex business rules are not easily expressible in SQL

P#9: SQL systems frequently do not perform well using approximate terms and fuzzy searches

P#10: SQL systems don't store and validate complex documents efficiently

Key features

2.2

1. ability to horizontally scale simple operations across nodes

2. ability to replicate and distribute (partition) data across nodes

3. simple call level interface (in contrast to SQL considered *too complex*)

4. weak concurrency model: forget ACID - go for BASE

5. efficient use of distributed indexes and RAM for data storage

6. ability to dynamically add new attributes to data records

SQL vs NoSQL + comments (I)

	SQL	NoSQL
types	one ‘logical’ database, with somewhat distinct ‘physical’ implement	many different types [columnar, key/value, document, graph, array, other]
history	1970	2000
storage	table/row/column aka. file/record/field storage	depends - records, documents ++unstructured++
schema	‘static’ schema’s - structure pre-determined	‘dynamic’ schema - is there a schema? ++unstructured++ ++schema free++
scaling	vertical	horizontal ++easier, cheaper++

SQL vs NoSQL + comments (II)

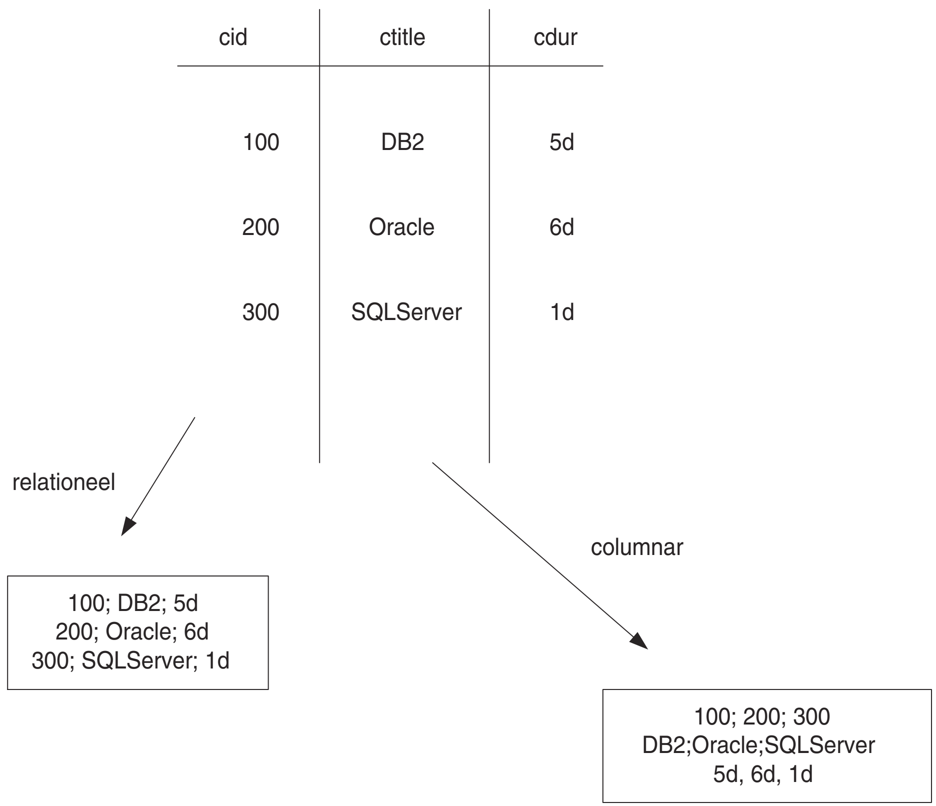
	SQL	NoSQL
development model	initially: propraiatary; later: open source	open source ++agile++
transaction support	yes ++	depends - not always
DML	SQL ++SQL++	OO APIs (perhaps also SQL) --infancy--
security & access control	fully implemented ++	
constraints	implemented, depending on... ++	often not enforced --
consistency	typically strong ACID-like	typically weaker BASE-like

NoSQL database types

- **Columnar Databases**
  [wide column store - 'big table' clones]
  - **stores data tables as sections of columns of data**
    [rather than as rows of data]
    [hybrid row/column structure]
  - **data stored together with meta-data ('a map')**
    [typically including row identification, attribute name, attribute value, and timestamp]
  - **sparse - or not**

*for example**:** Bigtable, HBase, Hypertable, Cassandra*

NoSQL database types (II)

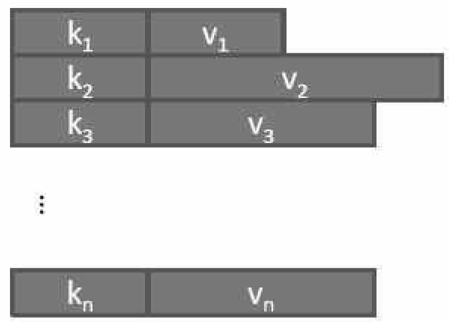


[easier aggregation, compression, self indexing]

NoSQL database types (III)

- **Key/Value Databases**
  - **values (data) stored based on programmer-defined** keys
  - **system is agnostic as to the semantics of the value**
  - **requests are expressed in terms of keys**

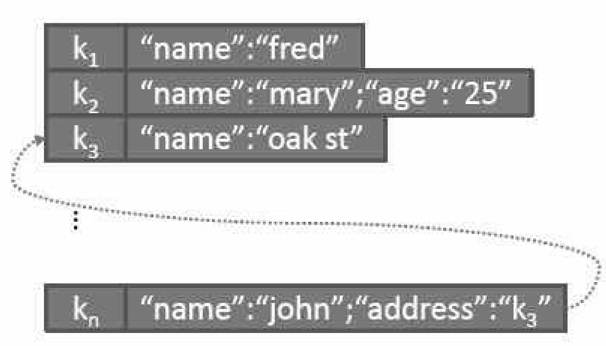
    put(key, value)
    get(key): value
  - **indexes can be/are defined over keys**
    [some systems support secondary indexes over (part of) the value]



for example: Berkley DB, Oracle NoSQL, LevelDB, AmazonDynamo, Mem-cached, ...

NoSQL database types (IV)

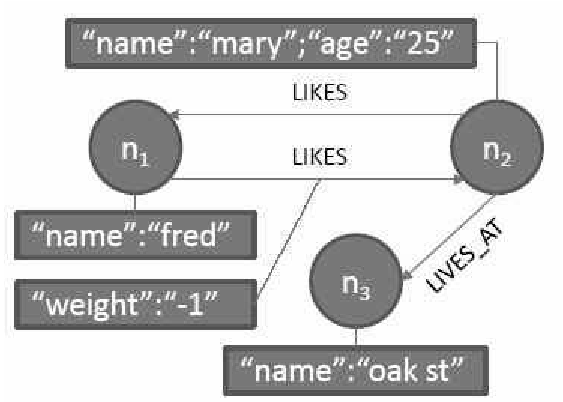
- **Document Data Model**
  - **documents are stored based on programmer-defined key** [a key-value store]
  - **system is aware of the arbitrary document structure**
  - **support for lists, pointers and nested documents**
  - **requests are expressed in terms of key (or attribute, if index exists)**
  - **support for key-based indexes and secondary indexes**



for example: MongoDB, CouchDB, RaptorDB, IBM Lotus Notes

NoSQL database types (V)

- **Graph Data Model**
  - **data is stored in terms of nodes and links**
    both can have (arbitrary) attributes
  - **requests are expressed based on system ids (if no indexes exist)**
    secondary indexes for nodes and links are supported
  - **SPARQL query language:** retrieve nodes by attributes and links by type, start and/or end node, and/or attributes



for example: Neo4j, InfoGrid, IMS

NoSQL database types (VI)

- **Array Data Model**
  - **nested multi-dimensional arrays**
    - cells can be tuples or other arrays
    - can have non-integer dimensions
  - **‘ragged’ arrays allow each row or column to have a different length**
  - **supports multiple flavours of “null”**

for example: SciDB

- **Other types/well-know DBs**
  - **object databases:** db4o
  - **XML databases:** EMC Documentum XDB, Tamino

Schema-less data storage

Most NoSQL databases at least offer the possibility to work:

- schema-less
- with dynamically changing schema’s

The CAP theorem / Brewer’s Conjecture

Real world data storage systems require three properties:

- [data]**<u>C</u>onsistency**
- **<u>A</u>vailability**
- **<u>P</u>artition tolerance**
  [partition: a server/node/rac in a collection of servers/nodes/racs]

Conjecture: in a multi server/node/rac shared nothing environment, it is not possible to satisfy all three requirements effectively with acceptable throughput rates!

In a ‘shared nothing’ environment, there is always P, so only C and A need to be considered - choose between two and loose one!

Transactions, Consistency, Availability

- **In ‘Shared something’ environments, C means ACID:**

  Pessimistic behaviour - force consistency at the end of every trans-action!

  - **Atomicity:** all or nothing
  - **Consistency:** transactions never observe or result in inconsistent data
  - **Isolation:** transactions are not aware of concurrent transactions
  - **Durability:** once committed, the state of a transaction is permanent

  Standard request in typical *core business processes***!**

  [A=> Availability beyond scope of this text]

Transactions, Consistency, Availability

- **In a ‘Shared nothing’ environment, BASE is implemented:**

  Optimistic behaviour - accepts database inconsistencies for a short period of time

  - **A/P => Basically Available/Soft state**
    [amongst other implemented using replication]
  - **C => Eventually consistent**
    [weak consistency: in the absence of failures, everything will be consistent in the end]

Most NoSQL databases implement BASE; depending on the actual NoSQL database in use, different flavours of BASE might be implemented, and some might even implement ACID.

MongoDB 3

Introduction 3.1

- **JSON-style documents (BSON)**
  [document-based queries]
- **schema-free**
  - written in C++ for high performance
  - full index support
  - memory mapped files
  - no transactions (but supports atomic operations)
  - not relational
- **scalability**
  replication - sharding
- **MongoDB = CP, optionally AP** [on top of CP]

Introduction

- **‘utilities’ available:**
  - mongoexport
  - mongoimport
  - *others*
- **language drivers available:** C, C++, Java, Javascript, perl, PHP, Python, Ruby, C#, Erlang, Delphi, ... [*community supported*]
- **OS:** OS X, Linux, Windows, Solaris
- **Opens source, free -** commercial edition available

- **A Mongo deployment (server or instance) holds a set of databases**
  - a database holds a set of collections
  - a collection holds a set of documents
  - a document is a set of fields: key-value pairs (JSON - BSON)
  - key-value-pairs:

    a *key* is a name (string)

    a *value* is a basic type like string, integer, float, timestamp, binary, etc., an embedded document, or an array of values
  - a ‘*special pair*’: _objectid - default artificial key

‘Lazy’ - [most] collections and databases are created when the first document in inserted into them...

Concepts and Structures (II)

- **collections can be ‘capped’**

  need to be created before **they can be used!**

  [no deletes, limited updates tolerated]

  have a ‘fixed’ size

db.createcollection(‘courseColCapped’, ..., ...)

Documents (I)

Document - oriented : collections **store** documents **in BSON format** [collection=?= table]

- **JSON-style documents: BSON (Binary JSON)**
- **support for ‘non-traditional’ data types:** Date **type and a** BinData **type**
  - can reference other documents
  - lightweight (*minimal spatial overhead*), traversable (*find data quickly*), efficient (*linked to C/C++ data types*) - VERY FAST
- **all documents belonging to one and the same collection** can have **heterogeneous data structures!** [remember: no schema’s]
- **typically** [check version]**: 4MB document limit**

Documents (II)

Let’s first introduce JSON...

JavaScript Object Notation

°) a collection of (nested) key-value pairs

°) supporting ordered lists

°) record oriented

... and then talk about BSON [Binary JSON] **- an ‘efficient’ implementation of JSON.**

- **efficient use of storage space**
- **increased scan-speed**
  [large elements in a BSON document are prefixed with a length field]
- **array indices explicitely stored**

Documents (III) - JSON


```
{
  "glossary": {
    "title": "example glossary",
      "GlossDiv": {
       "title": "S",
         "GlossList": {
         "GlossEntry": {
           "ID": "SGML",
               "SortAs": "SGML",
               "GlossTerm": "Standard Generalized Markup Language",
               "Acronym": "SGML",
               "Abbrev": "ISO 8879:1986",
               "GlossDef": {
             "para": "A meta-markup language, used to create DocBook.",
                  "GlossSeeAlso": ["GML", "XML"]
           },
               "GlossSee": "markup"
         }
       }
     }
  }
}
```

Installation - getting started

- **Installation**

  download, unzip, create data directory, create default config file, and get started!

- **Start the MongoDB 'server'**

./bin/mongod

[***bin\mongod.exe***]

- **Start MongoDB 'client' -** interactive JavaScript shell

./bin/mongo

[***bin\mongo.exe***]

[root@everest bin]# ./mongod --dbpath /data/db **--port** 27017 **--config** /etc/mongod.conf

Installation - getting started (II)

Basic commands - examples

use [db name]

show dbs
show collections

Basic operations - an introduction into ... 3.5

- **Insert operations**
  [sample]

```
> use coursedb
switched to db coursedb
> db.courseCol.insert({"Coursename":"DB2","Coursedur":3})
> db.courseCol.insert({"Coursename":"Oracle","Coursedur":5})
> db.courseCol.insert({"Coursename":"SQLServer","Coursedur":2})
> show collections
courseCol
system.indexes
```

Basic operations - an introduction into ... (II)

- **Select operations**
  [sample]

```
> db.courseCol.find({"Coursename":"Oracle"})

{ "_id" : ObjectId("51a089ad17338b27674af7a2"), "Coursename" : "Oracle", "Coursedur" : "5" }

> db.courseCol.find({"Coursename":"Oracle"},{"Coursedur":1});

{ "_id" : ObjectId("51a089ad17338b27674af7a2"), "Coursedur" : "5" }

> db.courseCol.find({Coursedur:{"$gt":2}});

{ "_id" : ObjectId("51a08fc295ce664a0e633cfb"), "Coursename" : "Oracle", "Coursedur" : 5 }
{ "_id" : ObjectId("51a08fd795ce664a0e633cfd"), "Coursename" : "DB2", "Coursedur" : 3 }
```

conditional ops: $gt, $gte, ..., $and, $in, $or, $nor, ...
$limit, $offset, ..., $sort, ...

Basic operations - an introduction into ... (III)

- **...**
  [sample]

```
> db.courseCol.insert({"Coursename":"DB2","Coursedur":3, "Instructor" : "Kris"})

> db.courseCol.find({"Coursename":"DB2"});
{ "_id" : ObjectId("51a08fd795ce664a0e633cfd"), "Coursename" : "DB2", "Coursedur" : 3 }
{ "_id" : ObjectId("51a090dd95ce664a0e633cfe"), "Coursename" : "DB2", "Coursedur" : 3,
"Instructor" : "Kris" }

> db.courseCol.find({"Coursename":"DB2"},{"Instructor":1});
{ "_id" : ObjectId("51a08fd795ce664a0e633cfd") }
{ "_id" : ObjectId("51a090dd95ce664a0e633cfe"), "Instructor" : "Kris" }

> db.courseCol.find({"Instructor":"Kris"});
{ "_id" : ObjectId("51a090dd95ce664a0e633cfe"), "Coursename" : "DB2", "Coursedur" : 3,
"Instructor" : "Kris" }
>
```

Basic operations - an introduction into ... (IV)

- **Update**
  [sample] - !! default !! - only the first doc is updated

```
> db.courseCol.insert({"Coursename":"DB2","Coursedur":3, "Instructor" : "Kris"})

> db.courseCol.find({"Coursename":"DB2"});
{ "_id" : ObjectId("51a09e6595ce664a0e633cff"), "Coursename" : "DB2", "Coursedur" : 3,
"Instructor" : "Kris" }

> db.courseCol.update({"Coursename":"DB2"},{$set : {"Coursedur":6}})
> db.courseCol.find({"Coursename":"DB2"});
{ "_id" : ObjectId("51a09e6595ce664a0e633cff"), "Coursename" : "DB2", "Coursedur" : 6,
"Instructor" : "Kris" }

> db.courseCol.update({"Coursename":"DB2"},{$set : {"CoursedurUSA":8}})
> db.courseCol.find({"Coursename":"DB2"});
{ "Coursedur" : 6, "CoursedurUSA" : 8, "Coursename" : "DB2", "Instructor" : "Kris", "_id" :
ObjectId("51a09e6595ce664a0e633cff") }
```

alternatives: $inc, $set, $push, $pushall, ...

Basic operations - an introduction into ... (V)

- **Remove**
  [sample]

```
> db.courseCol.remove()

db.courseCol.remove({"Coursedur" : {$lt : 7}})
> db.courseCol.find({"Coursename":"DB2"});
```

Indexes

- **full index support**
  [index on any attribute (including multiple, list/arrays, nested)]
  [blocking by default]
- **increase query performance**
- **indexes are implemented as “B-Tree” indexes**
  [unique or not][asc, desc]
  [missing keys: null by default - sparse index]
- **as always: data overhead for inserts and deletes**
- **document TTL in index can be specified**
- **implementation:**
  - **db.<col>.ensureIndex()**
  - **db.<col>.getIndexes(), getIndexKeys(), dropIndex(), reIndex()**
  - **db.system.indexes.find**

Indexes (II)

```
> db.courseCol.ensureIndex( {"Coursename" : 1 })
> db.courseCol.getIndexes()
[
    {},
    {
        "v" : 1,
        "key" : {
            "Coursename" : 1
        },
        "ns" : "test.courseCol",
        "name" : "Coursename_1"
    }
]
```

Indexes (III)

Limitations:

- **collections : max 64 indexes**
- **index key length max 1024 bytes**
- **queries can only use 1 index**
  [careful with concatenated indexes, careful with negations, careful with regexp]
- **indexes have storage requirements, and impact the performance of writes**
- **in memory sort (no-index) limited to 32 MB**

Indexes (IV) - explain, caching

```
> db.courseCol.find({"Coursename":"Oracle"}).explain()
{
    "cursor" : "BtreeCursor Coursename_1",
    "isMultiKey" : false,
    "n" : 1,
    "nscannedObjects" : 1,              "nscanned" : 1,
    "nscannedObjectsAllPlans" : 1,   "nscannedAllPlans" : 1,
    "scanAndOrder" : false,             "indexOnly" : false,
    "nYields" : 0,                          "nChunkSkips" : 0,
    "millis" : 0,                           "indexBounds" : {
        "Coursename" : [
            [
                "Oracle",
                "Oracle"
            ]
        ]
    },
    "server" : "everest.abis.be:27017"
}
```

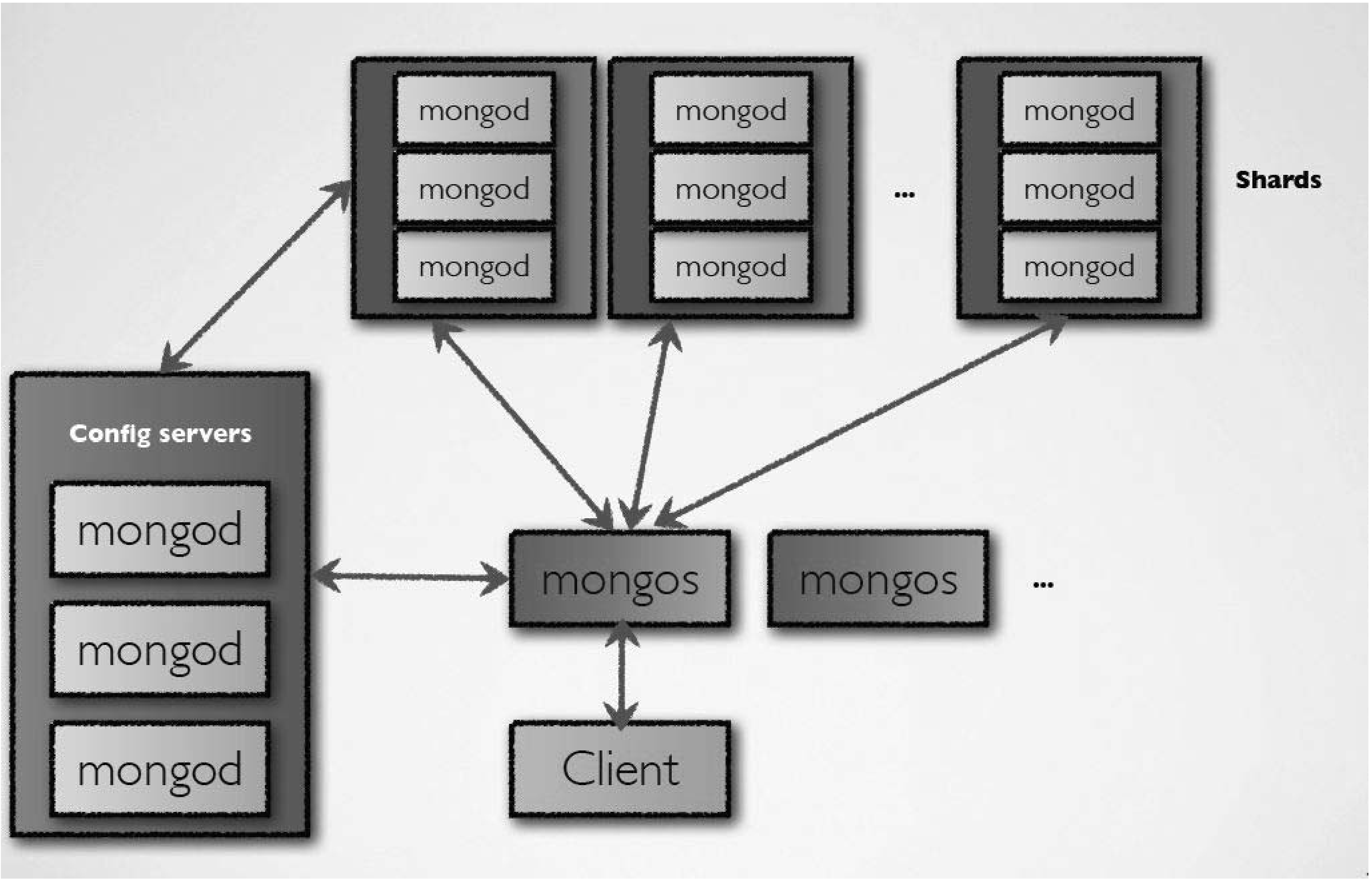
Indexes (V) - explain, caching

The Query Optimizer:

- for each "type" of query, MongoDB periodically tries all useful indexes
- aborts the rest as soon as one plan wins
- the ‘winning plan’ is temporarily cached foreach “type” of query

Hints are supported.

Architecture revisited



Shards

- **a shard is a node on a cluster**
- **a shard can be**
  - **a single mongod**
  - **a replica set**
    [multiple mongod]
- **data is stored on a** shard **in** chunks **of a specific** size
  [by default 64M]
- **MongoDB automatically** splits **and** migrates **chunks as needed**
- **Why use shards?**
  - **scale read/write performance**
  - **increase total RAM - keep ‘working set’ (index + data) in memory**

Config servers

- **stored meta data:**
  store cluster chunk ranges and locations
- **can have only 1 or 3**
  [production: use 3 if not ...]
- **2PC commit (not a replica set)**

```
[root@everest bin]# ./mongod --configsvr --port 27019
[root@zion bin]# ./mongod --configsvr --port 27019
[root@bryce bin]# ./mongod --configsvr --port 27019
```

MongoS

- **acts as a router / balancer**
  installed next to the application server
  routes application requests to the data
  balances chunks
- **no local data (persists to config database)**
- **can have 1 or many**

[root@thegrand bin]# ./mongos --configdb everest:27019, zion:27019, bryce:27019

Start, add, enable shard(ing)

- **start the shard database**
  [can be an already running, non-sharded db]

[root@xenophon bin]# ./mongod --shardsvr --dbpath /data/db **--port** 27018 **--config** /etc/mongod.conf

[root@socrates bin]# ./mongod --shardsvr --dbpath /data/db **--port** 27018 **--config** /etc/mongod.conf

- **add the shard definition on MongoS**

> sh.addShard('xenophon:27018')

> sh.addShard('socrates:27018')

- **enable sharding**

> sh.enableSharding("coursedb");

> sh.shardCollection("coursedb.courseCol", {"coursedur":1})

Sharding - chunks

- **based on** range-partitioning**!**
- **a chunk is a** section **of a range**
  - **a chunk is** split **once it exceeds the maximum size**
    [configuration, default 64M]

    There is no split point if all documents have the same shard key
  - **chunk split is a logical operation**
    [no data is moved]
  - **if split creates too large of a discrepancy of #chunks across shards: rebalancing starts**
    [configuration parameter]

Sharding - chunks (II)

- **rebalancing:**
  - **balancer part of mongos**
  - **migration - balancer lock:**
    - mongos sends *moveChunk* to source shard
    - source shard notifies destination shard
    - destination shard claims the chunk shard-key range
    - destination shard pulls documents from source shard
    - destination shard updates config server - new location of copied chunks
  - **cleanup:**
    - **source shard deletes moved data**
      [waits for open cursors to either close or time out]
    - **mongos releases the balancer lock after old chunks are delet-ed**

Sharding - chunks (III)

Shard key:

- **use a field commonly used in queries**
- **shard key is immutable; shard key values are immutable**
- **shard key requires index on fields contained in key**
- **shard key limited to 512 bytes in size**
- **things to think about:**
  [use your RDBMS skills]
  - cardinality
  - write distribution
  - query isolation
  - data distribution

Sharding - a recap...

- automatic partitioning
- automatic load-balancing
- range-based
- covert to sharded system with no downtime
- application code unaware of data location
- zero code changes

About Replication

- **Why?**
  - **high availability**
    - if a node fails, another node can step in
    - extra copies of data for recovery
  - **Scaling reads = applications with high read requirements can read from replicas**
- **a** *replica set* **- a set of mongod servers**
  - **minimum of 3**
  - **election of a primary (consensus)**
  - **writes go to primary; secondaries replicate from primary**
- **define and start the replica set -**'named' **set**

mongod --replSet <name>

<name> uses a configuration file, listing the other servers in the set

About Replication (II) - oplog

- **change operations are written to the** oplog **of the primary** [+ each secondary]
  - **a capped collection**
  - **contains increasing ordinal to keep track of modification ordering**
  - **must have enough space to allow new secondaries to catch up after copying from a primary**
  - **must have enough space to cope with any applicable** slaveDelay
  - **secondaries query the primary's oplog and apply what they find**

About Replication (III) - failover

Failover:

- **replica set members monitor other set members**
  [heartbeats - bi-directional]
- **if primary not reacheable, a new one is elected**
- **the secondary with the** most up-to-date oplog **is chosen**
  [priority can be set to influence election; secondaries can be banned from becoming primary]
- **if, after election, a secondary has changes not on the new primary, those are** undone**, and moved aside - resync**
- **if you require a guarantee, ensure data is written to a majority of the replica set**

Final remarks

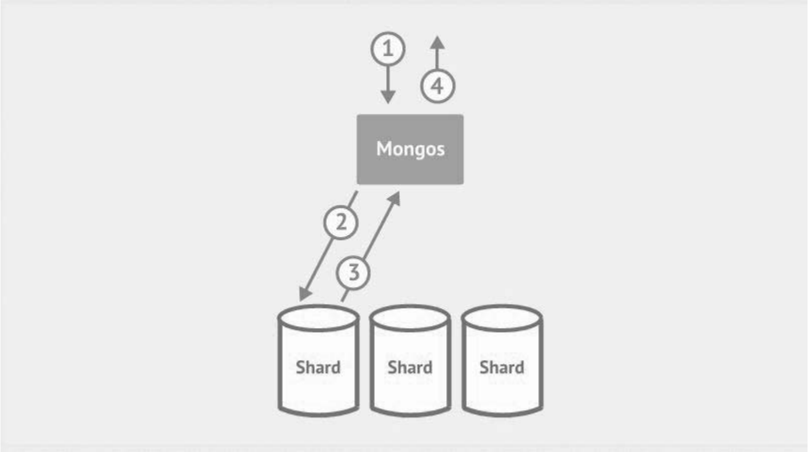
- **Read scaling - have reads access** primary **and/or** secondary **replicas** [slaveOkay]
- **Blocking for replication:** [access data when it is ‘guaranteed’]

Replication - recap...

- automatic failover
- automatic recovery
- all writes to primary node
- rolling outages, zero downtime

Request Routing

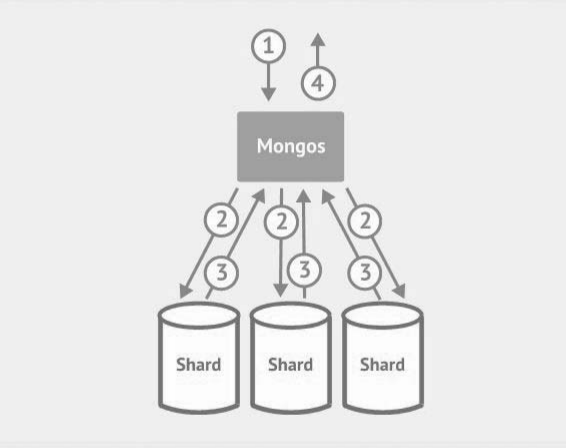
Targeted Queries



(1) request received; (2) routed to shard; (3) result returned; (4)result returned to client

Request Routing (II)

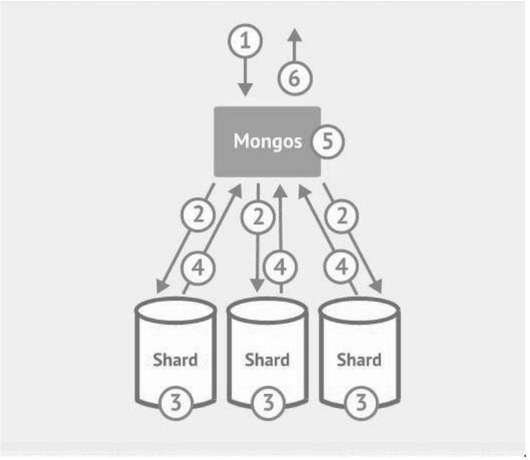
Scatter Gather Queries



(2) request sent to all shards

Request Routing (III)

Scatter Gather Queries with Sort

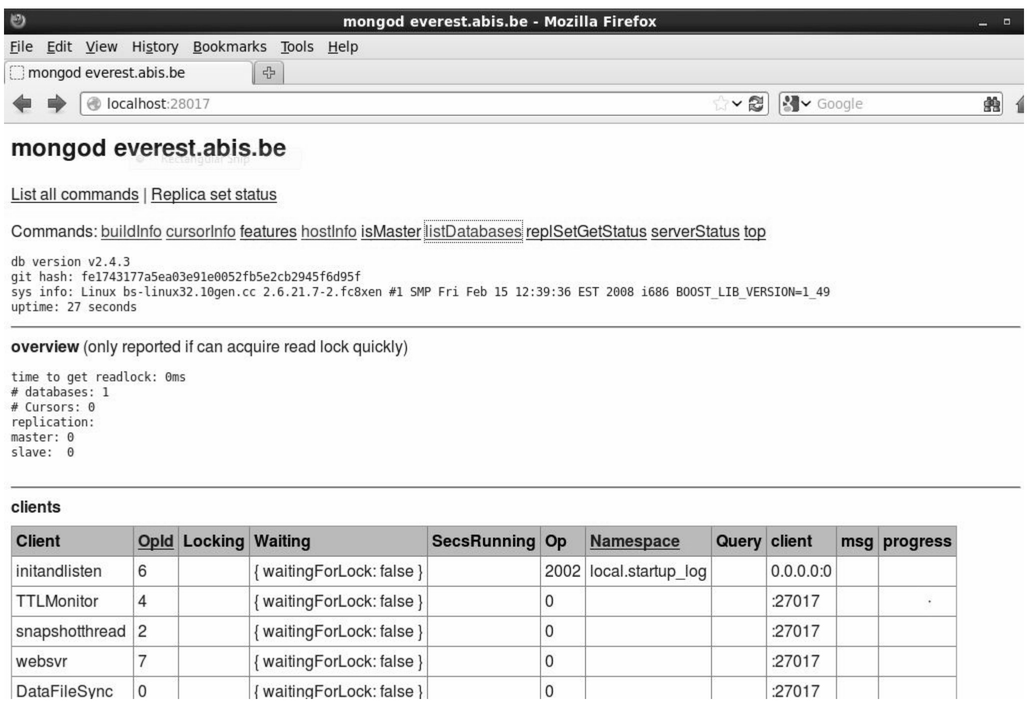


(3) request and sort performed locally; (5) mongos merges the sorted results

REST interface

- **mongod provides a basic REST interface**
  [-- rest, default port 28017]

[root@everest bin]# ./mongod --dbpath /data/db --port 27017 --config /etc/mongod.conf --rest



Other features...

- **GridFS**
  - **store files of any size** (exceeding binary storage data max size)
  - **GridFS leverages existing** replication **or** autosharding **that has been set up**
- **Map Reduce**
  - **queries** [jscript function] **run in all shards parallel** [one thread per node]
  - **flexible aggregation and data processing**
  - **often used**
- **Geospatial Indexing**

  two-dimensional indexing for location-based queries
  [find objects based on location? Find closest n items to x]

```
db.map.insert({location : {longitude : -40, latitude : 78}})
db.map.find({location : {$near : [ -30, 70]}})
```

Thank you!

TRAINING & CONSULTING

ABIS Training & Consulting
Kris Van Thillo
kvanthillo@abis.be