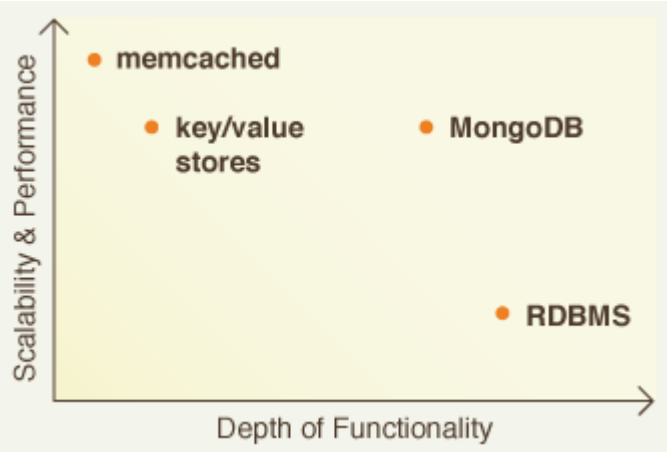
MongoDB

- "hu**mongo**us“ – huge, monstrous (data)
- Jack of all trades
  - Gateway to NoSQL from SQL
  - Mixing RDBMS and KV features
- In the C camp of CAP
  - P is fixed for scaled systems
  - Or in A camp…



Features

Why MongoDB?

Document-oriented

- Documents (objects) map nicely to programming language data types
- Embedded documents and arrays reduce need for joins
- Dynamically-typed (schemaless) for easy schema evolution
- No joins and **no multi-document transactions** for high performance and easy scalability

High performance

- No joins and embedding makes reads and writes fast
- Indexes including indexing of keys from embedded documents and arrays
- Optional streaming writes (no acknowledgements)

High availability

- Replicated servers with automatic master failover

Easy scalability

- Automatic sharding (auto-partitioning of data across servers)
  - Reads and writes are distributed over shards
  - No joins or multi-document transactions make distributed queries easy and fast
- Eventually-consistent reads can be distributed over replicated servers

Rich query language

Platforms

Installation

RedHat Enterprise Linux, CentOS, or Fedora Linux

Debian, Ubuntu or other Linux Systems

other Unix/Linux Systems

OS X

Windows

Packages

MongoDB is included in several different package managers. Generally speaking, it is easier to simply install the prebuilt binaries from above.

For MacPorts, see the **mongodb** package.

For Homebrew, see the **mongodb** formula.

For FreeBSD, see the **mongodb** and **mongodb-devel** ports.

For ArchLinux, see the **mongodb** package in the AUR.

For Debian and Ubuntu, see Ubuntu and Debian packages.

For Fedora and CentOS, see CentOS and Fedora packages.

For Gentoo, MongoDB can be installed by running **emerge mongodb**. See Gentoo Packages.

Drivers

MongoDB currently has client support for the following programming languages:
mongodb.org Supported

- C
- C++
- Erlang
- Haskell
- Java
- Javascript
- .NET (C# F#, PowerShell, etc)
- Node.js
- Perl
- PHP
- Python
- Ruby
- Scala

Shell:
Javascript

Community Supported

ActionScript3

http://github.com/argoncloud

C

libmongo-client

C# and .NET

Clojure

See the Java Language Center

ColdFusion

cfmongodb

Blog post: Part 1 | Part 2 | Part 3

http://github.com/virtix/cfmongodb/tree/0.9

http://mongocfc.riaforge.org/

D

Port of the MongoDB C Driver for D

Dart

https://bitbucket.org/vadimtsushko/mongo-dart

Delphi

…SOK…

Hadoop Adapter

The MongoDB Hadoop Adapter is a plugin for Hadoop that provides Hadoop the ability to use MongoDB as an input source and/or an output source.

What it does, how it works

- MongoDB is a **server process** that runs on Linux, Windows and OS X.
  - It can be run both as a 32 or 64-bit application. We recommend running in 64-bit mode, since Mongo is limited to a total data size of about 2GB for all databases in 32-bit mode.
- Clients **connect** to the MongoDB process, optionally authenticate themselves if security is turned on, and perform a sequence of actions, such as **inserts, queries and updates**.
- MongoDB stores its data in files (default location is /data/db/), and uses **memory mapped files** for data management for efficiency.

- MongoDB is a **collection-oriented, schema-free** document database.
- By *collection-oriented*, we mean that data is grouped into sets that are called 'collections'. Each collection has a unique name in the database, and can contain an unlimited number of documents. Collections are analogous to tables in a RDBMS, except that they don't have any defined schema.
- By *schema-free*, we mean that the database doesn't need to know anything about the structure of the documents that you store in a collection. In fact, you can store documents with different structure in the same collection if you so choose.
- By *document*, we mean that we store data that is a structured collection of key-value pairs, where **keys are strings**, and **values are** any of a rich set of data types, including arrays and **documents**. We call this data format "BSON" for "Binary Serialized dOcument Notation.„

- A **database** holds a set of collections
- A **collection** holds a set of documents

BSON

- BSON is a bin-ary-en-coded seri-al-iz-a-tion of JSON-like doc-u-ments. BSON is designed to be lightweight, traversable, and efficient. BSON, like JSON, supports the embedding of objects and arrays within other objects and arrays
- **BSON and MongoDB**
- MongoDB uses BSON as the data storage and network transfer format for "documents".
- BSON at first seems BLOB-like, but there exists an important difference: the **Mongo database understands BSON internals**. This means that MongoDB can "reach inside " BSON objects, even nested ones. Among other things, this allows MongoDB to build indexes and match objects against query expressions on both top-level and nested BSON keys.
- To be efficient, MongoDB uses a format called BSON which is a **binary representation** of this data. BSON **is faster to scan for specific fields** than JSON. Also BSON adds some additional types such as a date data type and a byte-array (bindata) datatype. BSON maps readily to and from JSON and also to various data structures in many programming languages.
- **Client** drivers **serialize data to BSON**, then transmit the data over the wire to the db. **Data is stored on disk in BSON format**. Thus, on a retrieval, the database does very little translation to send an object out, allowing high efficiency. The client driver unserialized a received BSON object to its native language format.

- So what are the goals of BSON? They are:

1. **Fast scan-ability.** For very large JSON documents, scanning can be slow. To skip a nested document or array we have to scan through the intervening field completely. In addition as we go we must count nestings of braces, brackets, and quotation marks. In BSON, the size of these elements is at the beginning of the field's value, which makes skipping an element easy.
2. **Easy manipulation.** We want to be able to modify information within a document efficiently. For example, incrementing a field's value from 9 to 10 in a JSON text document might require shifting all the bytes for the remainder of the document, which if the document is large could be quite costly. (Albeit this benefit is not comprehensive: adding an element to an array mid-document would result in shifting.) It's also helpful to not need to translate the string "9" to a numeric type, increment, and then translate back.
3. **Additional data types.** JSON is potentially a great interchange format, but it would be nice to have a few more data types. Most importantly is the addition of a "byte array" data type. This avoids any need to do base64 encoding, which is awkward.

   In addition to the basic JSON types of string, integer, boolean, double, null, array, and object, these types include date, object id, binary data, regular expression, and code.

A BSON document

```
{ author: 'joe',
  created : new Date('03/28/2009'),
  title : 'Yet another blog post',
  text : 'Here is the text...',
  tags : [ 'example', 'joe' ],
  comments : [
                { author: 'jim',
                  comment: 'I disagree'
                },
                { author: 'nancy',
                  comment: 'Good post'
                }
  ]
}
```

This document is a blog post, so we can store in a "posts" collection using the shell:

```
 > doc = { author : 'joe', created : new Date('03/28/2009'), ... }
> db.posts.insert(doc);
```

Data types

```
> // v1.8+ shell
> x
{
        "_id" :
ObjectId("4dcd3ebc9278000000005158"),
        "d" : ISODate("2011-05-13T14:22:46.777Z"),
        "b" : BinData(0,""),
        "c" : "aa",
        "n" : 3,
        "e" : [ ],
        "n2" : NumberLong(33)
}
```

```
> x.d instanceof Date
true
> x.b instanceof BinData
true
> typeof x
object
> typeof x.b
object
> typeof x.n
number
> typeof x.c
string
```

Queries are expressed as BSON documents which indicate a query pattern.

```
db.users.find({'last_name': 'Smith'})
```

Field Selection (BSON**)**

```
db.users.find({last_name: 'Smith'}, {'ssn': 1});
db.users.find({}, {thumbnail:0});
```

Sorting (BSON**)**

```
db.users.find({}).sort({last_name: 1});
```

Skip and Limit

```
db.users.find().skip(20).limit(10);
```

```
> db.users.find( { x : 3, y : "abc" } ).sort({x:1});
// select * from users where x=3 and y='abc' order by x asc;
```

Just Syntactic sugar…

```
 > db.users.find( { $query : { x : 3, y : "abc" }, $orderby : { x : 1 } } );
```

Syntax is not user friendly…
With drivers (ie. java), even worse:

```
BasicDBObject doc = new BasicDBObject();
doc.put("name", "MongoDB");
doc.put("type", "database");
doc.put("count", 1);
BasicDBObject info = new BasicDBObject();
info.put("x", 203);
info.put("y", 102);
doc.put("info", info);
coll.insert(doc);

BasicDBObject query = new BasicDBObject();
query.put("j", new BasicDBObject("$ne", 3));
query.put("k", new BasicDBObject("$gt", 10));
cursor = coll.find(query);
try {
   while(cursor.hasNext()) {
      System.out.println(cursor.next());
   }
} finally {
   cursor.close();
}
```

POJO mappers exist…
Jungo – use shell syntax in java

SELECT * FROM users WHERE age>33
db.users.find({age:{$gt:33}})

SELECT * FROM users WHERE age!=33
db.users.find({age:{$ne:33}})

SELECT * FROM users WHERE name LIKE "%Joe%"
db.users.find({name:/Joe/})

SELECT * FROM users WHERE a=1 and b='q'
db.users.find({a:1,b:'q'})

SELECT * FROM users WHERE a=1 or b=2
db.users.find({ $or : [{ a : 1 } , { b : 2 }] })

SELECT * FROM foo WHERE name='bob' and (a=1 or b=2)
db.foo.find({ name : "bob" , $or : [{ a : 1 } , { b : 2 }] })

SELECT * FROM users WHERE age>33 AND age<=40
db.users.find({'age':{$gt:33,$lte:40}})

- db.mycollection.find({ $where : function() { return this.a == 3 || this.b == 4; } });

Cursor, snapshot

```
> var cursor = db.things.find();
> while (cursor.hasNext()) printjson(cursor.next());
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
```

snapshot() mode assures that objects which update during the lifetime of a query are returned once and only once. This is most important when doing a find-and-update loop that changes the size of documents that are returned ($inc does not change size).

```
 > // mongo shell example
> var cursor = db.myCollection.find({country:'uk'}).snapshot();
```

Even with snapshot mode, items inserted or deleted during the query may or may not be returned; that is, this mode is not a true point-in-time snapshot.

- Snapshot mode assures no duplicates are returned, or objects missed, which were present at both the start and end of the query's execution (even if the object were updated). If an object is new during the query, or deleted during the query, it may or may not be returned, even with snapshot mode.
- Note that short query responses (less than 1MB) are effectively snapshotted.

Null

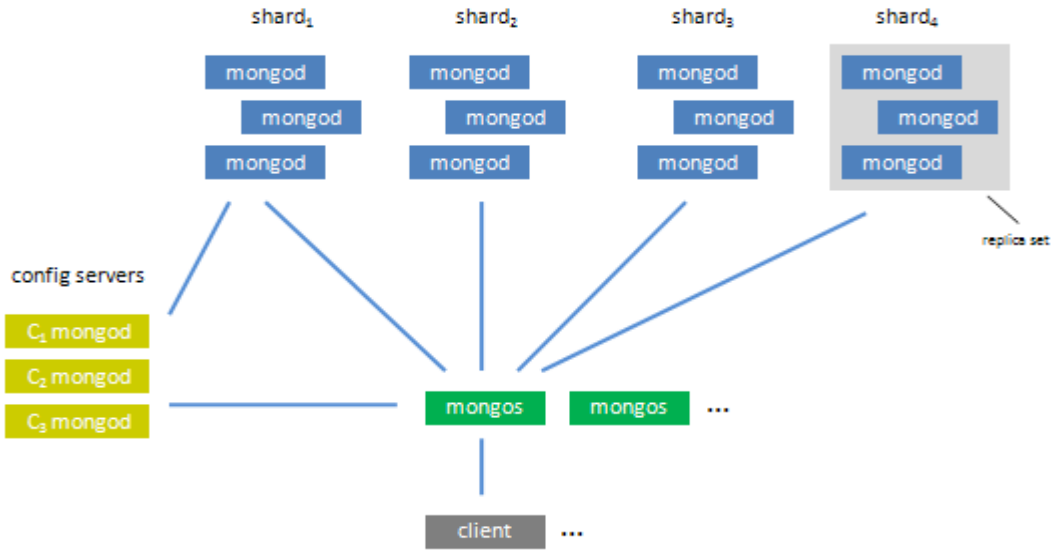
```
> db.foo.insert( { x : 1, y : 1 } )
> db.foo.insert( { x : 2, y : "string" } )
> db.foo.insert( { x : 3, y : null } )
> db.foo.insert( { x : 4 } )

// Query #1
> db.foo.find( { "y" : null } )
{ "_id" : ObjectId("4dc1975312c677fc83b5629f"), "x" : 3, "y" : null }
{ "_id" : ObjectId("4dc1975a12c677fc83b562a0"), "x" : 4 }

// Query #2
> db.foo.find( { "y" : { $type : 10 } } )
{ "_id" : ObjectId("4dc1975312c677fc83b5629f"), "x" : 3, "y" : null }

// Query #3
> db.foo.find( { "y" : { $exists : false } } )
{ "_id" : ObjectId("4dc1975a12c677fc83b562a0"), "x" : 4 }
```

To summarize the three queries:

1. documents where y has the value null or where y does not exist
2. documents where y has the value null
3. documents where y does not exist

Sharding

- Automatic balancing for changes in load and data distribution
- Easy addition of new machines without down time
- Scaling to one thousand nodes
- No single points of failure
- Automatic failover



- One or more shards, **each shard holds a portion of the total data** (managed automatically). Reads and writes are automatically routed to the appropriate shard(s).
- **Each shard is backed by a replica set** – which just holds the data for that shard.
  - A replica set is one or more servers, each holding copies of the same data. At any given time one is primary and the rest are secondaries.
  - If the primary goes down one of the secondaries takes over automatically as primary.
  - All writes and consistent reads go to the primary, and all eventually consistent reads are distributed amongst all the secondaries. (slaveOk option)
  - A client can block until a write operation has been replicated. (getLastError command, to N servers or majority, timeout)
- Multiple **config servers**, each one holds a copy of the meta data indicating **which data lives on which shard**.
- One or more **routers**, each one acts as a server for one or more clients. Clients issue queries/updates to a router and the router routes them to the appropriate shard while consulting the config servers.
- One or more clients, each one is (part of) the user's application and issues commands to a router via the mongo client library (driver) for its language.

mongod is the server program (data or config). **mongos** is the router program.

- Sharding is the partitioning of data among multiple machines in an order-preserving manner.
- Sharding is performed on a per-collection basis.
- Each shard stores multiple "chunks" of data, based on the shard key. MongoDB distributes these chunks evenly.
- In MongoDB, *sharding is the tool for scaling a system, and replication is the tool for data safety, high availability, and disaster recovery*.
  - The two work in tandem yet are orthogonal concepts in the design.
- MongoDB's auto-sharding scaling model shares many similarities with Yahoo's PNUTS and Google's BigTable.

Chunks on shard key **location:**

Machine 1	Machine 2	Machine 3
Alabama → Arizona	Colorado → Florida	Arkansas → California
Indiana → Kansas	Idaho → Illinois	Georgia → Hawaii
Maryland → Michigan	Kentucky → Maine	Minnesota → Missouri
Montana → Montana	Nebraska → New Jersey	Ohio → Pennsylvania
New Mexico → North Dakota	Rhode Island → South Dakota	Tennessee → Utah
	Vermont → West Virgina	Wisconsin → Wyoming

- A **chunk is a contiguous range of data** from a particular collection.
  - Chunks are described as a triple of collection, minKey, and maxKey.
  - Thus, the shard key K of a given document assigns that document to the chunk where **minKey <= K < maxKey**.
- Chunks grow to a maximum size, usually 64MB.
  - Once a chunk has reached that approximate size, the chunk ***splits*** into two new chunks.
  - When a particular shard has excess data, chunks will then ***migrate*** to other shards in the system. The addition of a new shard will also influence the migration of chunks.
  - Balancing is necessary when the load on any one shard node grows out of proportion with the remaining nodes.
    - In this situation, the **data must be redistributed to equalize load** across shards.
- If it is possible that a single value within the shard key range might grow exceptionally large, it is best to **use a compound shard key** instead so that further discrimination of the values will be possible.

Config DB Processes

- Each config server has a complete copy of all chunk information.
- A two-phase commit is used to ensure the consistency of the configuration data among the config servers.
- Note that config server use their own replication model; they are not run in as a replica set.
- If any of the config servers is down, the cluster's meta-data goes read only.
  - However, even in such a failure state, the MongoDB cluster can still be read from and written to.

Array matching

{colors : ["blue", "black"]}
{colors : ["yellow", "orange", "red"]}
db.things.find({ colors : "red" });
db.things.find({colors : {$ne : "red"}}) !!! Univerzalisan kvantalt, vagy -> $nin

{ author: 'joe',
  created : new Date('03/28/2009'),
  title : 'Yet another blog post',
  text : 'Here is the text...',
  tags : ['example', 'joe'],
}
db.things.find({ tags: { $all: [`db`, `store`] } });

$all, $in, $nin, $size, … csak skalar

Embedded objects

```
{
name: "Joe",
address: { city: "San Francisco", state: "CA" } ,
likes: [ 'scuba', 'math', 'literature' ]
}

> db.persons.find( { "address.state" : "CA" } )    //minosites
> db.persons.find( { likes : "math" } )

> db.blogposts.findOne()
{ title : "My First Post", author: "Jane",
  comments : [
      { by: "Abe", text: "First" },
      { by : "Ada", text : "Good post" } ]
}
> db.blogposts.find( { "comments.by" : "Ada" } )  //minosites tombben
> db.blogposts.find( { "comments.0.by" : "Abe" } )  //tomb index
```

Subobject matching (exact!!!)

```
{
        name: "Joe",
        address: {
                city: "San Francisco",
                state: "CA" } ,
        likes: [ 'scuba', 'math', 'literature' ]
}
```

```
db.persons.find( { "address" : { state: "CA" }} )  //don't match

db.persons.find( { "address" : {city: "San Francisco", state: "CA" }} )

db.persons.find( { "address" : {state: "CA" , city: "San Francisco"}} ) //don't match
```

$elemMatch

```
// Document 1
{ "foo" : [
    {
      "shape" : "square",
      "color" : "purple",
      "thick" : false
    },
    {
      "shape" : "circle",
      "color" : "red",
      "thick" : true
    }
] }
```

```
// Document 2
{ "foo" : [
    {
      "shape" : "square",
      "color" : "red",
      "thick" : true
    },
    {
      "shape" : "circle",
      "color" : "purple",
      "thick" : false
    }
] }
```

Who contains purple square?

db.foo.find({"foo.shape": "square", "foo.color": "purple"})

Returns both

db.foo.find({foo: {"shape": "square", "color": "purple"} })

Returns none

db.foo.find({foo: {"$elemMatch": {shape: "square", color: "purple"}}})

Returns Document 1

OK

Remove

db.things.remove({n:1}); // removes all where n == 1

Isolation

If a simultaneous update (on the same collection) grows an object which matched the remove criteria, the updated object **may not be removed** (as the operations are happening at approximately the same time, this may not even be surprising). In situations where this is undesireable, pass **{$atomic : true}** in your filter expression:

```
db.videos.remove( { rating : { $lt : 3.0 }, $atomic : true } )
```

The remove operation is then isolated – however, it will also block other operations while executing. The collection must be **unsharded** to use this option.

Update, save

db.collection.update(criteria, objNew, upsert, multi)

- criteria - query which selects the record to update;
- objNew - updated object or $ operators (e.g., $inc) which manipulate the object
- upsert - update the document if present; insert (a single document) if missing. **Upsert only inserts a single document.**
- multi - indicates if all documents matching criteria should be updated rather than just one. Can be useful with the $ operators below.

If you are coming from SQL, be aware that by default, update() only **modifies the first matched object**. If you want to modify all matched objects, you need to use the multi flag.

```
> // x is some JSON style object
> db.mycollection.save(x); // updates if exists; inserts if new
>
> // equivalent to:
> db.mycollection.update( { _id: x._id }, x, /*upsert*/ true );
```

Find the first document where 'name' is 'Joe' and then increment 'n' by one.

```
db.people.update( { name:"Joe" }, { $inc: { n : 1 } } );
```

```
{ $set : { x : 1 , y : 2 } }
{ $set : { x : 1 }, $inc : { y : 1 } }
```

$inc, $set, $unset, $rename, $bit and/or

!No $set : { x : “$y” }

Arrays

- $push - append
- $pushAll – append array
- $addToSet and $each – add if not contained, add list
- $pop – remove last
- $pull – remove all occurrences/criteria

  { $pull : { field : {$gt: 3} } }
- $pullAll - removes all occurrences of each value

The $ positional operator

```
{ "_id" : ObjectId("4b97e62bf1d8c7152c9ccb74"),
"title" : "ABC",
"comments" : [
        { "by" : "joe", "votes" : 3 },
        { "by" : "jane", "votes" : 7 } ]
}
```

```
> t.update( {'comments.by':'joe'},
{$inc:{'comments.$.votes':1}}, false, true )
```

Padding factor

- When you update an object in MongoDB, the update occurs in-place if the object has not grown in size. This is good for insert performance if the collection has many indexes.
  - If the document did grow in size, however, then it might need to be relocated on disk to find a new disk location with enough contiguous space to fit the new larger document.
- Mongo adaptively learns if objects in a collection tend to grow, and if they do, it adds some padding to prevent excessive movements. This statistic is tracked separately for each collection.

If a document gets smaller (e.g., because of an $unset or $pop), the document does not move but stays at its current location. It thus effectively has more padding. Thus space is never reclaimed if documents shrink by large amounts and never grow again. To reclaim that space run a compact operation (or repair).

Isolation

db.students.update({score: {$gt: 60}, $**atomic**: true}, {$set: {pass: true}}, false, true)

$atomic is not supported with Sharding

atomic

$atomic only means that the update will not yield for other operations while running. It does not mean that the operation is "all or nothing". See here.

Egyebkent pl. ez atomi (azaz match/modify atomos)

```
// decrement y if y > 0 (atomically)
db.stats.update(
  { _id : ‘myid’, y : { $gt : 0 } },
  { $inc : { y : -1 } }
)
```

Query optimizer

The database uses an interesting approach to query optimization. Traditional approaches (which tend to be cost-based and statistical) are not used, as these approaches have a couple of potential issues.

The MongoDB query optimizer is different. It is not cost based -- it does not model the cost of various queries. Instead, the **optimizer simply tries different query plans and learn which ones work well. Of course, when the system tries a really bad plan, it may take an extremely long time to run. To solve this, when testing new plans, MongoDB executes multiple query plans in parallel. As soon as one finishes, it terminates the other executions, and the system has learned which plan is good.** This works particularly well given the system is non-relational, which makes the space of possible query plans much smaller (as there are no joins).

Sometimes a plan which was working well can work poorly -- for example if the data in the database has changed, or if the parameter values to the query are different. In this case, if the query seems to be taking longer than usual, the database will once again run the query in parallel to try different plans.

This approach adds a little overhead, but has the advantage of being much better at worst-case performance.

Database references (linking)

Manual References

Manual references refers to the practice of including one document's _id field in another document. The application can then issue a second query to resolve the referenced fields as needed.

```
original_id = ObjectId()
db.places.insert({
   "_id": original_id
   "name": "Broadway Center"
   "url": "bc.example.net"
})

db.people.insert({
   "name": "Erin"
   "places_id": original_id
   "url":  "bc.exmaple.net/Erin"
})
```

Then, when a query returns the document from the people collection you can, if needed, **make a second query** for the document referenced by the places_id field in the places collection.

```
var e = db. people.findOne({"name": "Erin"});
var eplace = db.places.findOne({_id:e. places_id})
```

For nearly every case where you want to store a relationship between two documents, **use manual references**. The references are simple to create and your application can resolve references as needed.

The only limitation of manual linking is that these references do not convey the database and collection name. If you have documents in a **single collection that relate to documents in more than one collection**, you may need to consider using **DBRefs**.

Many relationship: people.places_id is array

```
original_id = ObjectId()
db.places.insert({
   "_id": original_id
   "name": "Broadway Center"
   "url": "bc.example.net"
})

db.update(
            {"name": "Erin“},
            {"places_id": {$push: original_id}}
)
```

DBRefs

DBRefs are a **convention for representing a document**, rather than a specific reference "type." They include the name of the collection, and in some cases the database, in addition to the ObjectId. Some drivers also have support for automatic resolution of DBRef references.

DBRefs have the following fields:

$ref - The $ref field holds the name of the collection where the referenced document resides.

$id - The $id field contains the value of the _id field in the referenced document.

$db - Optional. Contains the name of the database where the referenced document resides.

Only some drivers support $db references.

```
db.people.insert({
   "name": "Erin"
   "places_id": { $ref : "places", $id : original_id, $db : "db" },
   "url":  "bc.exmaple.net/Erin"
})
```

Support

C++The C++ driver contains **no support** for DBRefs. You can transverse references manually.

C#The C# driver provides access to DBRef objects with theMongoDBRef Classand supplies the FetchDBRef Methodfor accessing these objects.

JavaThe DBRef class provides supports for DBRefs from Java.

JavaScrptThe mongo shell's *JavaScript*interface provides automatic resolution of DBRef objects.

PerlThe Perl driver contains **no support** for DBRefs. You can transverse references manually or use the MongoDBx::AutoDeref CPAN module.

PythonThe Python driver supports automatic resolution of DBRef relationships, and provides the DBref class, and the dereference methodfor interacting with DBRefs.

RubyThe Ruby Driver supports DBRefs using the DBRef classand the deference method.

```
> var other = { s : "other thing", n : 1};
> db.otherthings.save(other);

> var mongo = db.things.findOne();
> print(tojson(mongo));
{"_id" : "497dab624ee47b3a675d2d9c" , "name" : "mongo" , "type" :
"database" , "data" : {"a" : 1 , "b" : 2}}

> mongo.otherthings = new DBRef( 'otherthings' , other._id );
> db.things.save(mongo);

> db.things.findOne().otherthings.fetch();
{"_id" : "497dab624ee47b3a675d2d9c" , "name" : "mongo" , "type" :
"database" , "data" : {"a" : 1 , "b" : 2} , "otherthings" : {"_id" :
"497dbcb36b27d59a708e89a4" , "s" : "other thing" , "n" : 1}}
```

Schema design

Schema design in MongoDB is very different from schema design in a relational DBMS.

- However it is still very important and the first step towards building an application.

In relational data models, conceptually there is a "correct" design for a given entity relationship model independent of the use case.

- This is typically a third normal form normalization. One typically only diverges from this for performance reasons.

In MongoDB, the schema design is not only a function of the data to be modeled but also of the use case.

- The schema design is optimized for our most common use case.
- This has pros and cons – that use case is then typically highly performant; however there is a bias in the schema which may make certain ad hoc queries a little less elegant than in the relational schema.

As we design the schema, the questions we must answer are:

1. When do we embed data versus linking (see below)? Our decisions here imply the answer to question #2:
2. How many collections do we have, and what are they?
3. When do we need atomic operations? These operations can be performed within the scope of a BSON document, but not across documents.
4. What indexes will we create to make query and updates fast?
5. How will we shard? What is the shard key?

Joins

Embedding and Linking

A key question when designing a MongoDB schema is when to embed and when to link.

- Embedding is the nesting of objects and arrays inside a BSON document.
- Links are references between documents.

There are no joins in MongoDB – distributed joins would be difficult on a 1,000 server cluster.

Embedding is a bit like "prejoined" data.

- Operations within a document are easy for the server to handle; these operations can be fairly rich.

Links in contrast must be processed client-side by the application; the application does this by issuing a follow-up query.

Generally, for "contains" relationships between entities, embedding should be chosen.

Use linking when not using linking would result in duplication of data.

Example

Our content management system will have posts. Posts have authors.
We'd like to support commenting and voting on posts. We'd also like posts to be taggable for searching.

two MongoDB collections, one called posts and one called users

```
 > db.users.insert( { _id : "alex", name: { first:"Alex", last:"Benisson" }, karma : 1.0 } )
```

```
> db.posts.findOne()
{
  _id : ObjectId("4e77bb3b8a3e000000004f7a"),
  when : Date("2011-09-19T02:10:11.3Z",
  author : "alex",
  title : "No Free Lunch",
  text : "This is the text of the post.  It could be very long.",
  tags : [ "business", "ramblings" ],
  votes : 5,
  voters : [ "jane", "joe", "spencer", "phyllis", "li" ],
  comments : [
    { who : "jane", when : Date("2011-09-19T04:00:10.112Z"),
      comment : "I agree." },
    { who : "meghan", when : Date("2011-09-20T14:36:06.958Z"),
      comment : "You must be joking.  etc etc ..." }
  ]
}
```

In a relation database: Users, posts ,tags ,voters and a comments collection.
Grabbing all the information on a single post would then be a little involved. To grab the whole post:

```
> db.posts.findOne( { _id : ObjectId("4e77bb3b8a3e000000004f7a") } );
```

To get all posts written by alex:

```
> db.posts.find( { author : "alex" } )
```

If the above is a common query we would create an index on the author field:

```
> db.posts.ensureIndex( { author : 1 } )
```

The post documents can be fairly large. To get just the titles of the posts by alex :

```
> db.posts.find( { author : "alex" }, { title : 1 } )
```

We may want to search for posts by tag:

```
> // make and index of all tags so that the query is fast:
> db.posts.ensureIndex( { tags : 1 } )
> db.posts.find( { tags : "business" } )
```

What if we want to find all posts commented on by meghan?

```
> db.posts.find( { comments.who : "meghan" } )
```

Let's index that to make it fast:

```
> db.posts.ensureIndex( { "comments.who" : 1 } )
```

No one can vote more than once. Suppose calvin wants to vote for the example post above.

The following update operation will record calvin's vote. Because of the $nin sub-expression, if calvin has already voted, the update will have no effect.

```
> db.posts.update( { _id : ObjectId("4e77bb3b8a3e000000004f7a"),
                     voters : { $nin : "calvin" } },
                   { votes : { $inc : 1 }, voters : { $push : "calvin" } );
```

Note the above operation is atomic : if multiple users vote simultaneously, no votes would be lost.

Suppose we want to display the title of the latest post in the system as well as the full user name for the author. This is a case where we must use client-side linking:

```
> var post = db.posts.find().sort( { when : -1 } ).limit(1);
> var user = db.users.find( { _id : post.author } );
> print( post.title + " " + user.name.first + " " + user.name.last );
```

A final question we might ask about our example is how we would shard.

If the users collection is small, we would not need to shard it at all.

If posts is huge, we would shard it. The key should be chosen based on the queries that will be common. We want those queries to involve the shard key.

Sharding by _id is one option here.

If finding the most recent posts is a very frequent query, we would then shard on the **when** field.

BSON ObjectId's begin with a timestamp. Thus sorting by _id, when using the ObjectID type, results in sorting by time. Note: granularity of the timestamp portion of the ObjectID is to one second only.

```
> // get 10 newest items
> db.mycollection.find().sort({id:-1}).limit(10);
```

Summary of Best Practices

- "First class" objects, that are at top level, typically have their own collection.
- Line item detail objects typically are embedded.
- Objects which follow an object modelling "contains" relationship should generally be embedded.
- Many to many relationships are generally done by linking.
- Collections with only a few objects may safely exist as separate collections, as the whole collection is quickly cached in application server memory.
- Embedded objects are a bit harder to link to than "top level" objects in collections.
- It is more difficult to get a system-level view for embedded objects. When needed an operation of this sort is performed by using MongoDB's map/reduce facility.
- If the amount of data to embed is huge (many megabytes), you may reach the limit on size of a single object. See also GridFS.
- If performance is an issue, embed.

Choosing Indexes

A second aspect of schema design is index selection. As a general rule, where you want an index in a relational database, you want an index in Mongo.

- The _id field is automatically indexed.
- Fields upon which keys are looked up should be indexed.
- Sort fields generally should be indexed.

The MongoDB profiling facility provides useful information for where an index should be added that is missing.

Note that adding an index slows writes to a collection, but not reads. Use lots of indexes for collections with a high read : write ratio (assuming one does not mind the storage overage). For collections with more writes than reads, indexes are expensive as keys must be added to each index for each insert.

Geospatial indexes… GIS

How Many Collections?

As Mongo collections are polymorphic, one could have a collection objects and put everything in it! This approach is taken by some object databases. This is not recommended in MongoDB for several reasons, mainly performance. Data within a single collection is roughly contiguous on disk. Thus, "table scans" of a collection are possible, and efficient. Just like in relational dbs, independent collections are very important for high throughput batch processing.

Aggregation

SELECT COUNT(*) FROM users
db.users.count()

SELECT COUNT(*) FROM users where AGE > 30
db.users.find({age: {'$gt': 30}}).count()

SELECT COUNT(AGE) from users
db.users.find({age: {'$exists': true}}).count()

SELECT a,b,sum(c) csum from coll where active=1 group by a,b

```
db.coll.group(
        {key: { a:true, b:true },
         cond: { active:1 },
         reduce: function(obj,prev) { prev.csum += obj.c; },
         initial: { csum: 0 }
         });
```

Note: currently one must use map/reduce instead of group() in sharded MongoDB configurations.
group returns an array of grouped items.

Group on condition: … keyf : function(doc) { return {"x" : doc.x}; }

```
{ domain: "www.mongodb.org"
, invoked_at: {d:"2009-11-03", t:"17:14:05"}
, response_time: 0.05
, http_action: "GET /display/DOCS/Aggregation"
}
```

Show me stats for each http_action in November 2009:

```
db.test.group(
  { cond: {"invoked_at.d": {$gte: "2009-11", $lt: "2009-12"}}
  , key: {http_action: true}
  , initial: {count: 0, total_time:0}
  , reduce: function(doc, out){ out.count++; out.total_time+=doc.response_time }
  , finalize: function(out){ out.avg_time = out.total_time / out.count }
  } );
```

Result:

```
 {
   "http_action" : "GET /display/DOCS/Aggregation",
   "count" : 1,
   "total_time" : 0.05,
   "avg_time" : 0.05
 }
]
```

Aggregation framework

Simplified map/reduce

Similar to GROUP BY

Pipelined aggregation

Expressed by JSON (not a surprise)

Aggregate command

```
db.people.aggregate( { [pipeline] } )
db.runCommand( { aggregate: "people", { [pipeline] } } )
```

- $project - SELECT
- $match - WHERE
- $limit
- $skip
- $unwind
- $group – GROUP BY
- $sort – ORDER BY

Conceptually, documents from a collection pass through an aggregation pipeline, which transforms these objects they pass through. For those familiar with UNIX-like shells (e.g. bash,) the concept is analogous to the pipe (i.e. |) used to string text filters together.

```
SELECT cust_id,SUM(price) FROM orders WHERE active=true GROUP BY cust_id
db.orders.aggregate([
 { $match:{active:true} },
 { $group:{_id:"$cust_id",total:{$sum:"$price"}} }
])

SELECT COUNT(*) FROM users
db.users.aggregate([
 { $group: {_id:null, count:{$sum:1}} }
])
```

Result

The aggregation operation in the previous section returns a document with two fields:

- result which holds an array of documents returned by the pipeline
- ok which holds the value 1, indicating success, or another value if there was an error

HAVING

Pipeline!

SELECT cust_id,SUM(price) as total FROM orders WHERE active=true GROUP BY cust_id HAVING total>50

```
db.orders.aggregate([
 { $match:{active:true} },
 { $group:{_id:"$cust_id",total:{$sum:"$price"}} },
 { $match: {total: {$gt:50}} }
])
```

```
{
 title : “this is my title” ,
 author : “bob” ,
 posted : new Date () ,
 pageViews : 5 ,
 tags : [ “fun” , “good” , “fun” ] ,
 comments : [
          { author :“joe” , text : “this is cool” } ,
          { author :“sam” , text : “this is bad” }
 ],
 other : { foo : 5 }
}
```

author names grouped by tags:

```
db.article.aggregate(
  { $project : {
     author : 1,
     tags : 1,
  } },
  { $unwind : “$tags” },
  { $group : {
     _id : { tags : 1 },
     authors : { $addToSet : “$author” }
  } }
);
```

Unwind: on next slide

Unwind

```
{
 title : “this is my title” ,
 author : “bob” ,
 posted : new Date () ,
 pageViews : 5 ,
 tags : [ “fun” , “good” , “fun” ] ,
 comments : [
          { author :“joe” , text : “this is cool” } ,
          { author :“sam” , text : “this is bad” }
 ],
 other : { foo : 5 }
}

db.article.aggregate(
   { $project : {
      author : 1 ,
      title : 1 ,
      tags : 1
   }},
   { $unwind : "$tags" }
);
```

```
[
        {
                "_id" :
ObjectId("4e6e4ef557b77501a49233f6"),
                "title" : "this is my title",
                "author" : "bob",
                "tags" : "fun"
        },
        {
                "_id" :
ObjectId("4e6e4ef557b77501a49233f6"),
                "title" : "this is my title",
                "author" : "bob",
                "tags" : "good"
        },
        {
                "_id" :
ObjectId("4e6e4ef557b77501a49233f6"),
                "title" : "this is my title",
                "author" : "bob",
                "tags" : "fun"
        }
]
```

- $unwind is most useful in combination with $group.
- The effects of an unwind can be undone with the$group pipeline operators.
- If you specify a target field for $unwind that does **not exist** in an input document, the document **passes through** $unwind unchanged.
- If you specify a target field for $unwind that is **not an array**, aggregate() generates an **error**.
- If you specify a target field for $unwind that holds an **empty array** ([]), then that field is **removed** from the result while all other fields are passed through unchanged.

$project

- Include fields from the original document.
- Exclude fields from the original document.
- Insert computed fields.
- Rename fields.
- Create and populate fields that hold sub-documents.

```
db.article.aggregate(
   { $project : {
      title : 1,
      doctoredPageViews : { $add:["$pageViews", 10] },
      bar : "$other.foo",
      stats : {
         pv : "$pageViews",
         foo : "$other.foo",
         dpv : { $add:["$pageViews", 10] }
      }
    }}
);
```

Keyf – computing a field for $group

$group

Begin by specifying an identifier (i.e. a _id field) for the group you're creating with this pipeline.
You can specify

a single field from the documents in the pipeline,
a previously computed value,
or an aggregate key made up from several incoming fields.

```
db.article.aggregate(
   { $group : {
      _id : "$author",
      docsPerAuthor : { $sum : 1 },
      viewsPerAuthor : { $sum : "$pageViews" }
   }}
);
```

Aggregate Ops: $addToSet, $first, $last , $max , $min , $avg , $push , $sum
Expression ops: basic logical, comparison, arithmetic ($add,$divide ,$mod ,$multiply ,$subtract), string, date, nullcheck

Warning

The aggregation system currently stores $group operations in **memory**, which may cause problems when processing a larger number of groups.

Unless the $sort operator can use an index, in the current release, **the sort must fit within memory**. This may cause problems when sorting large numbers of documents.

- **In future versions there may be pipleine optimization phase** in the pipleine that reorders the operations to increase performance without affecting the result. However, at this time **place$match operators at the beginning of the pipeline when possible**.
- If your aggregation operation requires only a subset of the data in a collection, use the $match to restrict which items go in to the top of the pipeline, as in a query. When placed early in a pipeline, these $match operations use suitable indexes to scan only the matching documents in a collection.

The aggregation framework is compatible with sharded collections.

- When operating on a sharded collection, the aggregation pipeline splits the pipeline into two parts. The aggregation framework pushes all of the operators up to and including the first $group or $sort to each shard.
  - If an early $match can exclude shards through the use of the shard key in the predicate, then these operators are only pushed to the relevant shards.
- Then, a second pipeline on the mongos runs. This pipeline consists of the first $group or $sort and any remaining pipeline operators: this pipeline runs on the results received from the shards.
  - The mongos pipeline merges $sort operations from the shards. The $group, brings any “sub-totals” from the shards and combines them: in some cases these may be structures. For example, the $avg expression maintains a total and count for each shard; the mongos combines these values and then divides.

Map/Reduce

```
db.runCommand(
 { mapreduce : <collection>,
   map : <mapfunction>,
   reduce : <reducefunction>,
   out : <see output options below>
   [, query : <query filter object>]
   [, sort : <sorts the input objects using this key. Useful for optimization, like sorting by the emit key for fewer reduces>]
   [, limit : <number of objects to return from collection, not supported with sharding>]
   [, finalize : <finalizefunction>]
}
);
```

The out directives are:

"collectionName" - By default the output will be of type "replace".

- { replace : "collectionName" } - the output will be inserted into a collection which will atomically replace any existing collection with the same name.
- { merge : "collectionName" } - This option will merge new data into the old output collection. In other words, if the same key exists in both the result set and the old collection, the new key will overwrite the old one.
- { **reduce** : "collectionName" } - If documents exists for a given key in the result set and in the old collection, then a reduce operation (using the specified reduce function) will be performed on the two values and the result will be written to the output collection. If a finalize function was provided, this will be run after the reduce as well.
- { inline : 1} - With this option, no collection will be created, and the whole map-reduce operation will happen in RAM. Also, the results of the map-reduce will be returned within the result object. Note that this option is possible only when the result set fits within the 16MB limit of a single document. In v2.0, this is your only available option on a replica set secondary.
- { **sharded** : 1} - **MongoDB 1.9+** If true and combined with an output mode that writes to a collection, the output collection will be sharded using the _id field.

Result object

```
{
 [results : <document_array>,]   //in case of inline
 [result : <collection_name> | {db: <db>, collection: <collection_name>},]
 timeMillis : <job_time>,
 counts : {
     input :  <number of objects scanned>,
     emit  : <number of times emit was called>,
     output : <number of items in output collection>
 } ,
 ok : <1_if_ok>
 [, err : <errmsg_if_error>]
}
```

Example

Comment docs:

```
{
   text: "Hello world!“
   username: "jones",
   likes: 20,
}
```

Map:

```
 function() {
   emit( this.username, {count: 1, likes: this.likes} );
  }
 //This essentially says that we'll be grouping by username and aggregating using an object with fields for count and likes.
```

Reduce:

```
  function(key, values) {
   var result = {count: 0, likes: 0};
   values.forEach(function(value) {
     result.count += value.count;
     result.likes += value.likes;
   });
   return result;
  }
```

Result collection:

//group id -> aggregate value

```
{"_id" :”jones” , "value" : {count: 11, likes: 51} }
{"_id" :”joe” , "value" : {count: 1, likes: 5} }
```

Iterative Reduce

Notice that **the result document has the same structure as the documents emitted by the map function**. This is important because, when the **reduce** function is run against a given key, it's **not guaranteed to process every single value for that key** (or username). In fact, the reduce function **may have to run more than once**.

For example, while processing the comments collection, the map function might encounter ten comments from the user "jones." It then sends those comments' data to be reduced, and this results in the following aggregate object:

```
{ count: 10, likes: 247 }
```

Later, the map function encounters one more comment document by "jones." When this happens, the values in the extra comment must be reduced against the already existing aggregate value. If the new emitted document looks like this:

```
{ count: 1, likes: 5 }
```

Then the reduce function will be invoked in this way:

```
reduce("jones", [ {count: 10, likes: 247}, { count: 1, likes: 5} ] )
```

And the resulting document will be a simple combination (or reduction) of those values:

```
{ count: 11, likes: 252 }
```

So long as you understand that the reduce function might be invoked more than once for the same key, it's easy to see why the this function must return a value whose structure matches the map function's emitted value.

function reduce(key_obj, [value_obj, value_obj, ...]) -> value_obj

The map/reduce engine may invoke reduce functions iteratively; thus, these functions must be idempotent. That is, the following must hold for your reduce function:

for all k,vals : reduce(k, [reduce(k,vals)]) == reduce(k,vals)

This also means the following is true:

reduce(k, [A, B]) == reduce(k, [B, A])

> The output of the map function's emit (the second argument) and the value returned by reduce should be the same format to make iterative reduce possible. If not, there will be weird bugs that are hard to debug.
>
> Currently, the return value from a reduce function cannot be an array (it's typically an object or a number).

Example

The following example assumes we have an events collection with objects of the form:
{ time : <time>, user_id : <userid>, type : <type>, ... }

We then use MapReduce to extract **all users who have had at least one event of type "sale":**

```
> m = function() { emit(this.user_id, 1); }
> r = function(k,vals) { return 1; }
> res = db.events.mapReduce(m, r, { query : {type:'sale'}, out : 'example1' });
> db[res.result].find().limit(2)
{ "_id" : 8321073716060 , "value" : 1 }
{ "_id" : 7921232311289 , "value" : 1 }
```

If we also wanted to output the **number of times the user had experienced the event** in question, we could modify the reduce function like so:

```
> r = function(k,vals) {
...     var sum=0;
...     for(var i in vals) sum += vals[i];
...     return sum;
... }
```

Note, here, that **we cannot simply return vals.length**, as the reduce may be called multiple times.
…reduce(34,1,1,1,1,1,1,1,) …return vals[0]+vals.length-1

Incremental Map-Reduce

Out {reduce} + query combination

If the data set over which you'd like to perform map-reduce aggregations is **constantly growing**, then you may want to take advantage of incremental map-reduce. The **prevents you from having to aggregate over the entire data set each time** you want to see your results.

1. First, run a map-reduce job over an existing collection, and output the data to it's own output collection.
2. When you have more data to process, run a second map-reduce job, but use the query option to the filter the documents to include only new documents.
3. Use the reduce output option. This will use your reduce function to merge the new data into the existing output collection.

Session docs:

```
{userid:"a", ts: ISODate('2011-11-03 14:17:00'), length: 95}
…
```

```
db.runCommand( {
                    mapreduce:"session",
                    map:mapf,
                    reduce:reducef,
                    out: "session_stat",
                    finalize:finalizef
});
```

```
function mapf(){
  emit(this.userid,   {userid:this.userid, total_time:this.length,
count:1, avg_time:0}); }

function reducef(key, values) {
   var r = {userid:key, total_time:0, count:0, avg_time:0};
   values.forEach(function(v)   {
                                        r.total_time += v.total_time;
                                        r.count += v.count;
                                    });
   return r;
}

function finalizef(key, value) {
   if (value.count > 0) value.avg_time = value.total_time /
value.count;
   return value;
}

db.runCommand( {
                    mapreduce:"session",
                    map:mapf,
                    reduce:reducef,
                    query: {ts: {$gt:ISODate('2011-11-03 00:00:00')}},
                    out: { reduce: "session_stat" },
                    finalize:finalizef
});
```

Sharded Environments

Sharded input
If the input collection is sharded, MongoS will automatically dispatch the map/reduce job to each of the shard, to be executed in parallel. There is no special option required. MongoS will then wait for jobs on all shards to finish.

Sharded output
By default the output collection will **not be sharded**. The process is:
- MongoS dispatches a map/reduce finish job to the shard that will store the target collection.
- that shard will pull results from all other shards, run a final reduce/finalize, and write to the output.

If using the sharded option in the out object, the output will be sharded using "_id" as the shard key.

In version prior to 2.1, the process is:
- MongoS pulls the results from each shard, doing a merge sort to get them ordered.
- on the fly, it does reduce/finalize as needed. Then writes the result to the output collection in sharded mode.
- though MongoS does some processing, only a small amount of memory is required even for large datasets.
- there is currently a limitation in that shard chunks do not get automatically split and migrated during insertion. Doing it manually is required until the chunks are granular and balanced.

Sharded output for mapreduce has been overhauled for v2.2. Using it for prior version is not recommended due to many limitations.

For v2.2 and above, the process is:
- if the target collection does not exist, it is created as sharded on _id. Even if empty, its initial chunks are created based on the result of the 1st step of MR.
- MongoS dispatches a map/reduce finish job to every shard that owns a chunk, in parallel.
- each shard will pull results it owns from all other shards, run a final reduce/finalize, and write to the target collection.
- During further MR jobs, chunk splitting will be done as needed.
- Balancing of chunks for the target collection is automatically prevented during post-processing to avoid concurrency issues.

Who is using

http://www.mongodb.org/display/DOCS/Production+Deployments

- Archiving
- Content Management
- Ecommerce
- Education
- Finance
- Gaming
- Government
- Metadata Storage
- News & Media
- Online Advertising
- Online Collaboration
- Real-time stats/analytics
- Social Networks
- Telecommunications

SAP, MTV, Sourceforge, Disney, IGN, the guardian, Forbes, The New York Times, Chicago Tribune, GitHub, Springer, UK Government, CNN Turk

Well Suited

- Archiving and event logging
- Document and Content Management Systems - as a document-oriented (JSON) database, MongoDB's flexible schemas are a good fit for this.
- ECommerce. Several sites are using MongoDB as the core of their ecommerce infrastructure (often in combination with an RDBMS for the final order processing and accounting).
- Gaming. High performance small read/writes are a good fit for MongoDB; also for certain games geospatial indexes can be helpful.
- High volume problems. Problems where a traditional DBMS might be too expensive for the data in question. In many cases developers would traditionally write custom code to a filesystem instead using flat files or other methodologies.
- Mobile. Specifically, the server-side infrastructure of mobile systems. Geospatial key here.
- Operational data store of a web site MongoDB is very good at real-time inserts, updates, and queries. Scalability and replication are provided which are necessary functions for large web sites' real-time data stores. Specific web use case examples:
    - content management
    - comment storage, management, voting
    - user registration, profile, session data
- Projects using iterative/agile development methodologies. Mongo's BSON data format makes it very easy to store and retrieve data in a document-style / "schemaless" format. Addition of new properties to existing objects is easy and does not generally require blocking "ALTER TABLE" style operations.
- Real-time stats/analytics

Less Well Suited

- Systems with a heavy emphasis on **complex transactions such as banking systems and accounting**. These systems typically require multi-object transactions, which MongoDB doesn't support. It's worth noting that, unlike many "NoSQL" solutions, MongoDB does support atomic operations on single documents. As documents can be rich entities; for many use cases, this is sufficient.
- Traditional Non-Realtime Data Warehousing. **Traditional relational data warehouses** and variants (columnar relational) are well suited for certain business intelligence problems – especially **if you need SQL (see below) to use client tools** (e.g. MicroStrategy) with the database. For cases where the analytics are realtime, the data very complicated to model in relational, or where the data volume is huge, MongoDB may be a fit.
- Problems requiring SQL.

Performance benchmark

http://blog.michaelckennedy.net/2010/04/29/mongodb-vs-sql-server-2008-performance-showdown/

http://www.cubrid.org/blog/dev-platform/nosql-benchmarking/

http://amesar.wordpress.com/2011/10/19/mongodb-vs-cassandra-benchmarks/