Jared Rosoff
@forjared
Overview

Challenges
• Variably typed data
• Complex objects
• High transaction rates
• Large data size
• High availability

Opportunities
• Agility
• Cost reduction
• Simplification
Changes Impacting the traditional RDBMS

VOLUME AND TYPE OF DATA
- Trillions of records
- 10’s of millions of queries per second
- Volume of data
- Semi-structured and unstructured data

AGILE DEVELOPMENT
- Iterative and continuous
- New and emerging apps

NEW ARCHITECTURES
- Systems scaling horizontally, not vertically
- Commodity servers
- Cloud Computing
Changes are significantly increasing the cost of delivering applications

DEVELOPER PRODUCTIVITY DECREASES

- Needed to add new software layers of ORM, Caching, Sharding and Message Queue
- Polymorphic, semi-structured and unstructured data not well supported

COST OF DATABASE INCREASES

- Increased database licensing cost
- Vertical, not horizontal, scaling
- High cost of SAN
How we got here
Challenges of enterprise data modeling

- Variably typed data
- Complex data objects
- High transaction rates
- Large data size
- High availability
- Agile development
Variably typed data

- Metadata management
- EAV anti-pattern
- Sparse table anti-pattern
<table>
<thead>
<tr>
<th>Entity</th>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Type</td>
<td>Book</td>
</tr>
<tr>
<td>1</td>
<td>Title</td>
<td>Inside Intel</td>
</tr>
<tr>
<td>1</td>
<td>Author</td>
<td>Andy Grove</td>
</tr>
<tr>
<td>2</td>
<td>Type</td>
<td>Laptop</td>
</tr>
<tr>
<td>2</td>
<td>Manufacturer</td>
<td>Dell</td>
</tr>
<tr>
<td>2</td>
<td>Model</td>
<td>Inspiron</td>
</tr>
<tr>
<td>2</td>
<td>RAM</td>
<td>8gb</td>
</tr>
<tr>
<td>3</td>
<td>Type</td>
<td>Cereal</td>
</tr>
</tbody>
</table>
## Sparse Table

<table>
<thead>
<tr>
<th>Type</th>
<th>Title</th>
<th>Author</th>
<th>Manufacturer</th>
<th>Model</th>
<th>RAM</th>
<th>Screen Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book</td>
<td>Inside Intel</td>
<td>Andy Grove</td>
<td>Dell</td>
<td>Inspiron</td>
<td>8gb</td>
<td>12”</td>
</tr>
<tr>
<td>Laptop</td>
<td></td>
<td></td>
<td>Dell</td>
<td>Inspiron</td>
<td>8gb</td>
<td>12”</td>
</tr>
<tr>
<td>TV</td>
<td></td>
<td></td>
<td>Panasonic</td>
<td>Viera</td>
<td>52”</td>
<td></td>
</tr>
<tr>
<td>MP3</td>
<td>Margin Walker</td>
<td>Fugazi</td>
<td>Dischord</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Constant schema changes
- Space inefficient
- Overloaded fields
## Concrete Table Inheritance

### Laptop

<table>
<thead>
<tr>
<th>Type</th>
<th>Manufacturer</th>
<th>Model</th>
<th>RAM</th>
<th>Screen Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laptop</td>
<td>Dell</td>
<td>Inspiron</td>
<td>8gb</td>
<td>12”</td>
</tr>
</tbody>
</table>

### TV

<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Model</th>
<th>Screen Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panasonic</td>
<td>Viera</td>
<td>52”</td>
</tr>
</tbody>
</table>

### Book

<table>
<thead>
<tr>
<th>Title</th>
<th>Author</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Margin Walker</td>
<td>Fugazi</td>
<td>Dischord</td>
</tr>
</tbody>
</table>
Complex objects
High transaction rates

Challenges

• Constant load from client
• Far in excess of single server capacity
• Can never take the system down
Challenges

- Adding more storage over time
- Aging out data that’s no longer needed
- Minimizing resource overhead of “cold” data
High availability

Twitter is currently down for Unplanned maintenance.
We expect to be back in about an hour. Thanks for your patience.
Agile development
### The data model is the problem

<table>
<thead>
<tr>
<th>Category</th>
<th>Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variably typed data</td>
<td>- Rigid schemas</td>
</tr>
<tr>
<td>Complex Objects</td>
<td>- Normalization can be hard</td>
</tr>
<tr>
<td></td>
<td>- Dependent on joins</td>
</tr>
<tr>
<td>High transaction rate</td>
<td>- Vertical scaling</td>
</tr>
<tr>
<td></td>
<td>- Poor data locality</td>
</tr>
<tr>
<td>High Availability</td>
<td>- Difficult to maintain consistency &amp; HA</td>
</tr>
<tr>
<td></td>
<td>- HA a bolt-on to many RDBMS</td>
</tr>
<tr>
<td>Agile Development</td>
<td>- Schema changes</td>
</tr>
<tr>
<td></td>
<td>- Monolithic data model</td>
</tr>
</tbody>
</table>
A new data model
var post = { author: “Jared” ,
            date: new Date(),
            text: “NoSQL Now 2012”,
            tags: [“NoSQL”, “MongoDB”]}

> db.posts.save(post)
Queries

```
>db.posts.find()

{  
  _id: ObjectId("4c4ba5c0672c685e5e8aabf3"),
  author: "Jared",
  date: "Sat Jul 24 2010 19:47:11 GMT-0700 (PDT)",
  text: "NoSQL Now 2012",
  tags: ["NoSQL", "MongoDB"]
}
```

Notes:
- `_id` is unique, but can be anything you’d like
Indexes

Create index on any Field in Document

// 1 means ascending, -1 means descending

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

>db.posts.find({author: 'Jared'})

{ _id : ObjectId("4c4ba5c0672c685e5e8aabf3"),
  author : "Jared",
  ... }
Query Operators

- Conditional Operators
  - $all, $exists, $mod, $ne, $in, $nin, $nor, $or, $size, $type
  - $lt, $lte, $gt, $gte

// find posts with any tags
> db.posts.find( {tags: {$exists: true}} )

// find posts matching a regular expression
> db.posts.find( {author: /^Jar*/i} )

// count posts by author
> db.posts.find( {author: ‘Jared’} ).count()
Atomic Operators

• $set, $unset, $inc, $push, $pushAll, $pull, $pullAll, $bit

> comment = { author: "Brendan",
             date: new Date(),
             text: "I want a freakin pony"}

> db.posts.update( { _id: "..." },
                   $push: {comments: comment} );
Nested Documents

```json
{
  _id: ObjectId("4c4ba5c0672c685e5e8aabf3").
  author: "Jared",
  date: "Sat Jul 24 2010 19:47:11 GMT-0700 (PDT)",
  text: "NoSQL Now 2012",
  tags: ["NoSQL", "MongoDB"],
  comments: [
    {
      author: "Brendan",
      date: "Sat Jul 24 2010 20:51:03 GMT-0700 (PDT)",
      text: "I want a freakin pony"
    }
  ]
}
```
Index on Nested Fields

// Index nested documents
> db.posts.ensureIndex( "comments.author":1 )
  > db.posts.find({’comments.author’:’Brendan’})

// Index on tags
> db.posts.ensureIndex( tags: 1)
> db.posts.find( { tags: ’MongoDB’ } )

// geospatial index
> db.posts.ensureIndex( “author.location”: “2d” )
> db.posts.find( “author.location” : { $near : [22,42] } )
db.posts.aggregate(
  { $project : {
    author : 1,
    tags : 1,
  } },
  { $unwind : "tags" },
  { $group : {
    _id : { tags : 1 },
    authors : {
      $addToSet : "author"
    }
  } }
);
It’s highly available
Replica Sets

- **Primary**
  - Write
  - Read

- **Secondary**
  - Read

Asynchronous Replication

Driver
Replica Sets

Driver

Write
Read

Primary

Automatic Leader Election

Secondary

Read
Replica Sets

- **Driver**
  - Read
  - Write
  - Read
  - Read

- **Primary**
  - Read

- **Secondary**
  - Read
  - Read
With tunable consistency
Strong Consistency
Eventual Consistency
Durability
Durability

- Fire and forget
- Wait for error
- Wait for fsync
- Wait for journal sync
- Wait for replication
Fire and forget

Driver → Primary

write

apply in memory
Get last error

Driver

write

getLastError

Primary

apply in memory

write

Wait for Journal Sync

Driver

write

getLastError

j:true

Primary

apply in memory

Write to journal
Wait for replication

Driver

Primary

Secondary

write

getLastError

w:2

apply in memory

replicate
## Write Concern Options

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;n:integer&gt;</code></td>
<td>Replicate to N members of replica set</td>
</tr>
<tr>
<td>“majority”</td>
<td>Replicate to a majority of replica set members</td>
</tr>
<tr>
<td><code>&lt;m:modeName&gt;</code></td>
<td>Use custom error mode name</td>
</tr>
</tbody>
</table>
Tagging

{  
  _id: "someSet",
  members: [
    {  
      _id: 0, host:"A", tags: { dc: "ny"}
    },
    {  
      _id: 1, host:"B", tags: { dc: "ny"}
    },
    {  
      _id: 2, host:"C", tags: { dc: "sf"}
    },
    {  
      _id: 3, host:"D", tags: { dc: "sf"}
    },
    {  
      _id: 4, host:"E", tags: { dc: "cloud"}
    }
  ],
  settings: {
    getLastErrorModes: {
      veryImportant: { dc: 3 },
      sortOfImportant: { dc: 2 }
    }
  }
}
• Between 0..1000
• Highest member that is up to date wins
  – Up to date == within 10 seconds of primary
• If a higher priority member catches up, it will force election and win
Slave Delay

- Lags behind master by configurable time delay
- Automatically hidden from clients
- Protects against operator errors
  - Accidentally delete database
  - Application corruptions data
Arbiters

• Vote in elections
• Don’t store a copy of data
• Use as tie breaker
Single Server

Data Center

Primary
Replica set

Data Center

Zone 1: Primary
Zone 2: Secondary
Zone 3: Secondary
Backup Node

Data Center

Zone 1: Primary
Zone 2: Secondary
Zone 3: Secondary (hidden = true)

backups
Disaster Recovery

Active Data Center

Zone 1: Primary priority = 1
Zone 2: Secondary priority = 1

Standby Data Center

Secondary priority = 0
Multi Data Center

West Coast DC

Central DC

Zone 1

Primary
priority = 2

Zone 2

Secondary
priority = 2

East Coast DC

Zone 1

Secondary
priority = 1

Zone 2

Secondary
priority = 1

Abiter
Sharding
Architecture

client -> mongos
client -> mongos
client -> mongos
client -> mongos

mongos

config
config
config

Config Servers

mongod
mongod
mongod

Shard

mongod
mongod
mongod

Shard

mongod
mongod
mongod

Shard
> db.runCommand( { shardcollection: "test.users",
key: { email: 1 } } )

{  
    name: "Jared",
    email: "jsr@10gen.com",
}
{  
    name: "Scott",
    email: "scott@10gen.com",
}
{  
    name: "Dan",
    email: "dan@10gen.com",
}
Chunks
Chunks

dan@10gen.com
jsr@10gen.com
scott@10gen.com
Chunks

Split!

dan@10gen.com

scott@10gen.com

jsr@10gen.com
Chunks

-∞  Split!  +∞

This is a chunk  This is a chunk

dan@10gen.com  jsr@10gen.com  scott@10gen.com
Chunks

dan@10gen.com
jsr@10gen.com
scott@10gen.com
Chunks

-∞ → dan@10gen.com → jsr@10gen.com → scott@10gen.com → +∞
Chunks

Split!

-∞

+∞

dan@10gen.com

jsr@10gen.com

scott@10gen.com
Chunks

<table>
<thead>
<tr>
<th>Min Key</th>
<th>Max Key</th>
<th>Shard</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td><a href="mailto:dan@10gen.com">dan@10gen.com</a></td>
<td>1</td>
</tr>
<tr>
<td><a href="mailto:dan@10gen.com">dan@10gen.com</a></td>
<td><a href="mailto:jsr@10gen.com">jsr@10gen.com</a></td>
<td>1</td>
</tr>
<tr>
<td><a href="mailto:jsr@10gen.com">jsr@10gen.com</a></td>
<td><a href="mailto:scott@10gen.com">scott@10gen.com</a></td>
<td>1</td>
</tr>
<tr>
<td><a href="mailto:scott@10gen.com">scott@10gen.com</a></td>
<td>+∞</td>
<td>1</td>
</tr>
</tbody>
</table>

- Stored in the config servers
- Cached in MongoS
- Used to route requests and keep cluster balanced
Balancing

Chunks!

Shard 1

Shard 2

Shard 3

Shard 4

mongos
balancer

config

config

config
Balancing

Shard 1

Shard 2

Shard 3

Shard 4

mongos balancer

config

Imbalance
Balancing

Shard 1: 1 2 3 4 5 6 7 8 9 10 11 12
Shard 2: 21 22 23 24
Shard 3: 33 34 35 36
Shard 4: 45 46 47 48

mongos balancer

Move chunk 1 to Shard 2

config
config
config
Balancing

mongos balancer

Shard 1

Shard 2

Shard 3

Shard 4

config

config

config
Balancing

mongos
balancer

Chunk 1 now lives on Shard 2

Shard 1
Shard 2
Shard 3
Shard 4
<table>
<thead>
<tr>
<th>Queries</th>
<th>Method</th>
<th>MongoDB Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>By Shard Key</td>
<td>Routed</td>
<td><code>db.users.find({email: &quot;jsr@10gen.com&quot;})</code></td>
</tr>
<tr>
<td>Sorted by shard key</td>
<td>Routed in order</td>
<td><code>db.users.find().sort({email:-1})</code></td>
</tr>
<tr>
<td>Find by non shard key</td>
<td>Scatter Gather</td>
<td><code>db.users.find({state: &quot;CA&quot;})</code></td>
</tr>
<tr>
<td>Sorted by non shard key</td>
<td>Distributed merge sort</td>
<td><code>db.users.find().sort({state:1})</code></td>
</tr>
</tbody>
</table>
| Inserts | Requires shard key | db.users.insert({
|         |                  |   name: “Jared”,
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>email: “<a href="mailto:jsr@10gen.com">jsr@10gen.com</a>”})</th>
</tr>
</thead>
</table>
| Removes | Routed            | db.users.delete({
|         |                  |   email: “jsr@10gen.com”}) |
|         | Scattered         | db.users.delete({name: “Jared”}) |
| Updates | Routed            | db.users.update(
|         |                  |   {email: “jsr@10gen.com”},
|         |                  |   {$set: {state: “CA”}}}) |
|         | Scattered         | db.users.update(
|         |                  |   {state: “FZ”},
|         |                  |   {$set:{state: “CA”}}, false, true ) |
1. Query arrives at Mongoose
2. Mongoose routes query to a single shard
3. Shard returns results of query
4. Results returned to client
1. Query arrives at MongoS
2. MongoS broadcasts query to all shards
3. Each shard returns results for query
4. Results combined and returned to client
1. Query arrives at MongoDB
2. MongoDB broadcasts query to all shards
3. Each shard locally sorts results
4. Results returned to MongoDB
5. MongoDB merge sorts individual results
6. Combined sorted result returned to client
Use cases
Sample Use Cases

Content Management
- SAP
- Viacom
- Forbes
- The White House
- Wordnik
- Craigslist

Operational Intelligence
- Buddy Media
- GitHub
- Intuit
- Traackr
- Custom Ink
- Open Sky
- Gilt

Meta Data Management
- Forbes.com
- Social Coding
- Intuit
- Traackr
- Shutterfly

User Data Management
- Disney
- Viber
- Ebay
- Foursquare
- Cisco

High Volume Data Feeds
- Athena Capital Research
- The New York Times
- Stripe
- Chartbeat
# High Volume Data Feeds

<table>
<thead>
<tr>
<th>Machine Generated Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>• More machines, more sensors, more data</td>
</tr>
<tr>
<td>• Variably structured</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stock Market Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>• High frequency trading</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Social Media Firehose</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Multiple sources of data</td>
</tr>
<tr>
<td>• Each changes their format constantly</td>
</tr>
</tbody>
</table>
High Volume Data Feed

Asynchronous writes

Flexible document model can adapt to changes in sensor format

Write to memory with periodic disk flush

Scale writes over multiple shards
Operational Intelligence

Ad Targeting
- Large volume of state about users
- Very strict latency requirements

Real time dashboards
- Expose report data to millions of customers
- Report on large volumes of data
- Reports that update in real time

Social Media Monitoring
- What are people talking about?
Operational Intelligence

- Low latency reads
- Parallelize queries across replicas and shards
- In database aggregation
- Flexible schema adapts to changing input data
- Can use same cluster to collect, store, and report on data

API

Dashboards
Intuit relies on a MongoDB-powered real-time analytics tool for small businesses to derive interesting and actionable patterns from their customers’ website traffic

### Problem
- Intuit hosts more than 500,000 websites
- wanted to collect and analyze data to recommend conversion and lead generation improvements to customers.
- With 10 years worth of user data, it took several days to process the information using a relational database.

### Why MongoDB
- Intuit hosts more than 500,000 websites
- wanted to collect and analyze data to recommend conversion and lead generation improvements to customers.
- With 10 years worth of user data, it took several days to process the information using a relational database.

### Impact
- In one week Intuit was able to become proficient in MongoDB development
- Developed application features more quickly for MongoDB than for relational databases
- **MongoDB was 2.5 times faster than MySQL**

*We did a prototype for one week, and within one week we had made big progress. Very big progress. It was so amazing that we decided, “Let’s go with this.”* -Nirmala Ranganathan, Intuit
Behavioral Profiles

1. See Ad

2. See Ad

3. Click

4. Convert

Rich profiles collecting multiple complex actions

Dynamic schemas make it easy to track vendor specific attributes

Scale out to support high throughput of activities tracked

Indexing and querying to support matching, frequency capping

```json
```
Meta Data Management

Data Archiving
- Meta data about artifacts
- Content in the library

Information discovery
- Have data sources that you don’t have access to
- Stores meta-data on those stores and figure out which ones have the content

Biometrics
- Retina scans
- Finger prints
db.archives.
  find({ "country": "Egypt" });

{ type: "Artefact",
  medium: "Ceramic",
  country: "Egypt",
  year: "3000 BC"
}

{ ISBN: "00e8da9b",
  type: "Book",
  country: "Egypt",
  title: "Ancient Egypt"
}

Indexing and rich query API for easy searching and sorting

Flexible data model for similar, but different objects
Shutterfly uses MongoDB to safeguard more than six billion images for millions of customers in the form of photos and videos, and turn everyday pictures into keepsakes.

**Problem**

- Managing 20TB of data (six billion images for millions of customers) partitioning by function.
- Home-grown key value store on top of their Oracle database offered sub-par performance.
- Codebase for this hybrid store became hard to manage.
- High licensing, HW costs.

**Why MongoDB**

- JSON-based data structure.
- Provided Shutterfly with an agile, high performance, scalable solution at a low cost.
- Works seamlessly with Shutterfly’s services-based architecture.

**Impact**

- 500% cost reduction and 900% performance improvement compared to previous Oracle implementation.
- Accelerated time-to-market for nearly a dozen projects on MongoDB.
- Improved Performance by reducing average latency for inserts from 400ms to 2ms.

The “really killer reason” for using MongoDB is its rich JSON-based data structure, which offers Shutterfly an agile approach to develop software. With MongoDB, the Shutterfly team can quickly develop and deploy new applications, especially Web 2.0 and social features. -Kenny Gorman, Director of Data Services.
Content Management

News Site
- Comments and user generated content
- Personalization of content, layout

Multi-Device rendering
- Generate layout on the fly for each device that connects
- No need to cache static pages

Sharing
- Store large objects
- Simple modeling of metadata
Flexible data model for similar, but different objects

GridFS for large object storage

Geo spatial indexing for location based searches

Horizontal scalability for large data sets

```json
{  
  camera: "Nikon d4",
  location: [ -122.418333, 37.775 ]
}

{  
  camera: "Canon 5d mkII",
  people: [ "Jim", "Carol" ],
  taken_on: ISODate("2012-03-07T18:32:35.002Z")
}

{  
  origin: "facebook.com/photos/xwdf23fsdf",
  license: "Creative Commons CC0",
  size: {  
    dimensions: [ 124, 52 ],
    units: "pixels"
  }
}
```
**Problem**

- Analyze a staggering amount of data for a system build on continuous stream of high-quality text pulled from online sources
- Adding too much data too quickly resulted in outages; tables locked for tens of seconds during inserts
- Initially launched entirely on MySQL but quickly hit performance road blocks

**Why MongoDB**

- Migrated 5 billion records in a single day with zero downtime
- MongoDB powers every website requests: 20m API calls per day
- Ability to eliminated memcached layer, creating a simplified system that required fewer resources and was less prone to error.

**Impact**

- **Reduced code by 75% compared to MySQL**
- Fetch time cut from 400ms to 60ms
- Sustained insert speed of 8k words per second, with frequent bursts of up to 50k per second
- Significant cost savings and 15% reduction in servers

---

*Life with MongoDB has been good for Wordnik. Our code is faster, more flexible and dramatically smaller. Since we don’t spend time worrying about the database, we can spend more time writing code for our application.* - Tony Tam, Vice President of Engineering and Technical Co-founder
Identity Management

Social Graphs
- Scale out to large graphs
- Easy to search and process

Identity Management
- Authentication, Authorization, and Accounting
Social Graphs

Native support for Arrays makes it easy to store connections inside user profile

Sharding partitions user profiles across available servers

Documents enable disk locality of all profile data for a user
Review
Data modeling is hard

• Variety, Velocity and Volume make it difficult
• Documents are easier for many use cases
Modern deployments are hard

- Distributed by default
- High availability
- Cloud deployment
MongoDB is better

• Document oriented data model
• Highly available deployments
• Strong consistency model
• Horizontally scalable architecture