High-performance NoSQL techniques

Michael Kennedy
@mkennedy
http://blog.michaelckennedy.net
Objectives

• Discuss the spectrum of NoSQL DBs and related perf.
  – Focusing on Document Databases
• Understand four key techniques for Doc DB perf.
  1. Document design
  2. Indexes
  3. Server-side operations
  4. Scale out
NoSQL: What is NoSQL?

Our NoSQL definition:

“Database systems which are cluster-friendly and which trade inter-entity relationships for simplicity and performance.”
NoSQL: The 4 types of NoSQL databases

- **Key-value stores:**
  - Amazon DynamoDB
  - Riak
  - Memcached

- **Column-oriented databases:**
  - Hbase
  - Cassandra
  - Amazon SimpleDB

- **Graph databases:**
  - FlockDB
  - Neo4J

- **Document databases:**
  - MongoDB
  - CouchDB
  - RavenDB
NoSQL: Key-value stores - how do they store data?

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>data-blob</td>
</tr>
<tr>
<td>2</td>
<td>data-blob</td>
</tr>
<tr>
<td>3</td>
<td>data-blob</td>
</tr>
<tr>
<td>4</td>
<td>data-blob</td>
</tr>
<tr>
<td>5</td>
<td>data-blob</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
NoSQL: Column-oriented DBs - how do they store data?

- **account**
  - username: mkennedy
  - password: F839B0C232A3051334
  - email: michaelk@develop.com

- **orders**
  - OID-27489: details...
  - OID-15723: details...
  - OID-83461: details...
  - OID-90242: details...
  - OID-01424: details...
  - OID-57238: details...
NoSQL: Graph DBs - how do they store data?
NoSQL: Document DBs - how do they store data?

```json
{
    "_id" : ObjectId("524ca37bd588bf0e4c1ff713"),
    "Name" : "Intensive C++ Training",
    "ActiveCourse" : true,
    "NewCourse" : false,
    "CourseHighlights" : "...",
    "Prerequisites" : "...",
    "Engagements" : [
        {
            "_id" : ObjectId("524ca37bd588bf0e4c1ff714"),
            "CourseId" : ObjectId("524ca37bd588bf0e4c1ff713"),
            "StartDate" : ISODate("2010-03-15T07:00:00Z"),
            "..." : "..."
        },
        {
            "_id" : ObjectId("524ca37bd588bf0e4c1ff715"),
            "CourseId" : ObjectId("524ca37bd588bf0e4c1ff713"),
            "StartDate" : ISODate("2011-04-11T07:00:00Z"),
            "..." : "..."
        }
    ],
    "CourseAliases" : [],
    "UrlPath" : "intensive-c++-training"
}
```
NoSQL: Not as different as you think

- Document databases
- Column-oriented databases
- Key-value stores
- RDBMSs
- Graph databases

Aggregates

Small and related
Modeling a complex data set: RDBMS version

LearningLine has a complex schema. Small subset:

- Users
  - start and complete tasks
  - start and complete activities
  - favorite and vote on both

- Tasks
  - contain activities
  - contain tags
  - can be voted and favorited

- Activities
  - contain type (video, exercise)
  - can be voted and favorited

etc.
Modeling a complex data set with documents: the database entities
Modeling a complex data set with documents: the classes
Choosing a document database: MongoDB

http://www.mongodb.org
MongoDB: Why MongoDB

http://www.google.com/trends/explore#q=mongodb%2C%20couchdb%2C%20%2Fm%2F04f32m3%2C%20ravendb%2C%20raik&cmpt=q
Technique 1: Model your documents correctly

• Your document model can either
  – Allow you to access documents efficiently with few queries
  – Require you to access many documents to answer simple questions
Your model should match your app's access patterns

• Consider a book with user reviews (votes and comments).
• Do you usually want the reviews anytime you access the book?
  – yes: Embed the votes within the book
  – no: Normalize them with soft foreign keys.
To embed or not to embed?

• Book reviews are a 1-to-many relationship.
• Should reviews be embedded within book or 'normalized'?

```json
{ // book
  _id: 1,
  Title: "...",
  Reviews: [
    { userId: 832, comments: "...", stars: 4, /* ... */ },
    { /*...*/ }
  ],
}
```
To embed or not to embed?

Beyond the basic access patterns (are reviews and books always used together?), you need to consider the size and number of subdocuments:

• Is your embedded set:
  – Small and bounded
    • embed if used together most of the time
  – Huge or unbounded
    • do not embed (ever)
Don't be afraid of rich documents

At first, you may be concerned about creating documents that are too big or too hard to query. The database can probably handle it.

```json
{ // Book
  "_id": ObjectId("525867313a93bb2198103c16"),
  "ISBN": "0393045218",
  "ImageUrls": [ { "Size": 1, "Url": "..." }, {"Size": 2,"Url": "..."}, ... ],
  "Ratings": [ { "UserId": 3, "Value": 7 }, { "UserId": 23, "Value": 5 }],
  //...
}
```

```javascript
// 0.001 sec for 250,000 books, 1.2 million ratings
db.Book.find({"Ratings.Value": {$gt: 5 }})
  .sort({"Ratings.Value": -1})
  .limit(10)
```
Integration database

- Brittle, hard to scale, hard to change
Application databases

- Less brittle, easier to scale, easier to change
Access patterns: more diverse queries

Application DBs (rather than integration DBs) tend to have more focused access patterns. But there is still a spectrum of use-cases.

• Guidance:

The more diverse the set of queries and use-cases for a given dataset, the more you may lean towards normalization. *

* Note: See map-reduce / aggregation guidance later in this presentation.
Single biggest performance problem (and fix)?

Incorrect indexes
(too few or too many)
Technique 2: Add indexes (empirically)

• Wrong or missing indexes are the #1 reason for bad performance
  – a database without indexes is probably wrong
• More important than in RDBMS
• Choosing the right indexes is key to a good application
Basic cursor

- If no indexes, Mongo uses a **basic cursor**
  - linear search, inefficient: $O(n)$
  - traversal of collection
    - retrieves documents
      - may use disk (slow)
      - may wreck cached data

- `_id` is the only field always indexed
Basic cursor example

- Find $x == 7$
- Linear search

```javascript
{x:7, _id:}
```
Indexes use B-tree data structures

• Indexes = B-tree on collection
  – same as RDBMS
  – generalization of a binary search tree: nodes can have more than 2 children
  – "B" does not mean "binary"

• Much better performance
  – $O(\log(n))$
  – try to keep index in memory
  – no need for full scan => less potential disk access
B-tree cursor example

- Find x == 7
- Tree traversal

```
{ x: 7, _id: }
```
Creating simple indexes

- **EnsureIndex** creates index only when not already there
- Optional name – generated otherwise
- Order (ascending or descending) not meaningful

```csharp
// Typed
books.EnsureIndex(IndexKeys<Book>.Ascending(b => b.ISBN));

// Untyped
books.EnsureIndex(IndexKeys.Ascending("ISBN"));
// or
books.EnsureIndex("ISBN");

// Named index
books.EnsureIndex(IndexKeys<Book>.Ascending(
    b => b.ISBN), IndexOptions.SetName("MyIndex")
);

// Native
db.books.ensureIndex( { ISBN: 1 } )
```
Indexing sub document

- It is common to index a field on subdocuments
- Same type of key

```csharp
books.EnsureIndex(IndexKeys<Book>.Ascending(b => b.Author.FirstName));
books.EnsureIndex(IndexKeys<Book>.Ascending(b => b.Author.LastName));

// Same as
books.EnsureIndex("Author.FirstName");
books.EnsureIndex("Author.LastName");
```
Multi key indexes

- When the value is an array, the index contains one key per value
- Speeds up "Contains" query

```csharp
class Book {
    public List<string> Categories { get; set; }
}

// Basic cursor
books.Where(b => b.Categories.Contains("Novel"));
books.EnsureIndex(IndexKeys<Book>.Ascending(b => b.Categories));

// B-tree cursor
books.Where(b => b.Categories.Contains("Novel"));

// Basic cursor
db.books.find({Categories: "Novel"})

db.books.EnsureIndex({Categories: 1})

// B-tree cursor
db.books.find({Categories: "Novel"})
Choosing the right index

- There is a balance between too few indexes and too many
  - too few => slow queries
  - too many => slow writes, uses memory
- There is a limit of 64 indexes per collection
- Focus on usual queries
  - rare queries can use basic cursor
- Consider leveraging compound indexes for simple queries
  - e.g. (a, b covers a and (a, b))
Creating indexes in the background

- By default, EnsureIndex returns when the index is fully built
  - this can take a while
  - the DB is frozen during index generation in this case
- Background indexes are non-blocking
  - does not wait until the index is fully built
  - DB is not frozen during index generation
  - Once the index in built, it behaves like a foreground index

```javascript
books.EnsureIndex(IndexKeys<Book>.Ascending(b => b.ISBN),
IndexOptions.SetBackground(true));
```

```javascript
db.books.EnsureIndex({ISBN:1}, {background:true})
```
Troubleshooting performance

- First, analyze the **profile** trace
  - find slow queries
  - find repeated queries
  - find unused indexes

- Once you isolate the query, **explain** it
  - get the execution plan: find the index used
  - if necessary, create new indexes
Profiling

• Save slow queries to profiling collection
• Can explore the profiling documents for index creation ideas
Enabling profiling

- You must enable profiling to log slow queries

```csharp
database.SetProfilingLevel(ProfilingLevel.Slow, TimeSpan.FromMilliseconds(100));
if(database.GetProfilingLevel().Level != ProfilingLevel.All)
{
    // do something about it
}
```

- Can check if profiling is enabled:

```csharp
//native
db.setProfilingLevel(0)
db.system.profile.drop()
db.createCollection( "system.profile", { capped: true, size:4000000 } )
db.setProfilingLevel(1, 5)
```
Finding slow queries

- The profiling information is saved to the `system.profile` collection.
- Class `SystemProfileInfo` already models the information in the collection
- Query the profile collection like any collection

```csharp
database.GetProfilingInfo(Query<SystemProfileInfo>.
    Where(x => x.Duration > TimeSpan.FromMilliseconds(100)));

// Or do it the link way
database.GetCollection<SystemProfileInfo>("system.profile").Where(…);
```

- Interesting fields:
  - Query (query)
  - Duration (millis)
  - Collection (namespace)
Profile queries examples

```csharp
var profile =
database.GetCollection<SystemProfileInfo>("system.profile").AsQueryable();

var duration = TimeSpan.FromMilliseconds(10);
var slowQueries = profile.Where(x => x.Duration >= duration);

var updateQueries = profile.Where(x => x.Op == "update");

var inmemorySorting = profile.Where(x => x.ScanAndOrder);
```
Analyze a query

- By default, MongoDB chooses an index:
  - finds all relevant indexes
  - tries them all
  - the first one wins
  - caches the result

- You can specify an index manually

- You can examine the execution plan of a query ("Explain")
Explaining a query

- Any query can be explained

```javascript
var myBooks = books.Where(b => b.ISBN = "a");
BsonDocument queryPlan = books.Explain();
QueryPlan typed = BsonSerializer.Deserialize<QueryPlan>(queryPlan);
```

- Look at
  - `cursor` (want B-Tree, not basic cursor)
  - `millis`: Time in milliseconds - is it really that slow?
  - `n`: Document count matching the query. If `n` is too large: hopeless
  - `nscanned`: scanned index entries and documents
  - `nscannedObjects`: scanned documents
  - `indexOnly`: the index covers the query, no need to access document
Technique 3: Update documents in place

- Retrieving an entire document just to update a value is inefficient and has concurrency issues.
  - MongoDB has server-side operators to handle this
Solution 3: atomic update

- MongoCollection.Update provides atomic update
  - can affect multiple documents
  - atomic (transacted but limited one collection)
  - efficient: minimal data transfer

- Update has two parts: query and update
  - each can be type safe
  - consider using Query<T> and Update<T> helper classes

```csharp
books.Update(query, update);

// Single update
books.Update(
    Query<Book>.EQ(b => b.ISBN, "123-234-345"),
    Update<Book>.Set(b => b.Price, 32));

// Multiple updates
books.Update(
    Query<Book>.LT(b => b.Published, yesterday),
    Update<Book>.Set(b => b.IsOnSale, true));
```
Update: Query<T>

- Query<T>.xxx(lambda, value)
  - e.g. Query<Person>.EQ(x => x.Name, “Alice”);
  - think p => p.Name == “Alice”
- Query<T>.Where(lambda) is even easier

<table>
<thead>
<tr>
<th>Query method</th>
<th>Linq equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ(x=&gt; x.Prop, v)</td>
<td>x =&gt; x.Prop == value</td>
</tr>
<tr>
<td>GT(x =&gt; x.Prop, v)</td>
<td>x =&gt; p.Prop &gt; value</td>
</tr>
<tr>
<td>GTE(x =&gt; x.Prop, v)</td>
<td>x =&gt; x.Prop &gt;= value</td>
</tr>
<tr>
<td>LT(x =&gt; x.Prop, v)</td>
<td>x =&gt; p.Prop &lt; value</td>
</tr>
<tr>
<td>LTE(x =&gt; x.Prop, v)</td>
<td>x =&gt; x.Prop &lt;= value</td>
</tr>
<tr>
<td>In(x =&gt; x.Prop, v)</td>
<td>x =&gt; x.Prop.Contains(v)</td>
</tr>
</tbody>
</table>
Technique 4: Scaling out

• Scale-out is the great promise of NoSQL
• MongoDB has two modes of scale out
  – Replication
  – Sharding

Real-word statistics from one company

• 120,000 DB operations / second
• 2GB of app-to-db I/O / second
Scaling: RDBMS

Typically vertical scaling

Start here

$ / perf

Scale here

$$$$$ / perf
MongoDB: performance (at scale)

- Well suited to scale horizontally
  - documents as aggregate roots
- Two very different (but combinable modes)
  - sharding
  - replica sets

Start here
$ / perf

Scale here
$$ / perf
Replication vs. scalability

- **Replication** is not the primary way to scale
  - even though you may get better read performance, not much better write performance unless your app is very read heavy
- **Sharding** is the primary way to improve single query speed
Replication basics

- A **replica set** is comprised of mongod instances
  - one **primary** centralizes all writes
  - all other instances are **secondary**. Read only access
How does it work? - oplog

- The **oplog** (operation log) contains the list of operations that modify the database
  - stored in **oplog.rs** capped collection in **system.local** database
  - each replica has its own oplog

```json
{
    "ts" : Timestamp(1382640299, 1), // When
    "h" : NumberLong(2253890413614977032),
    "v" : 2,
    "op" : "i", // Insert
    "ns" : "test.Person", // Target collection
    "o" : { // What to insert
        "_id" : ObjectId("52696aab42904c4d7a770295"),
        "Name" : "Alice"
    }
}
```
How does it work? Data replication

- Primary creates one oplog entry for every change
- Secondaries replicate the oplog of the primary
- Secondaries replay oplog entries on databases – could be delayed
How does it work? Heartbeat

• Each replica monitors a heart beat with other replicas
  – knows which replica is up
  – heart beat every 2 seconds
  – replica considered dead after missing 10 seconds of heart beats
How does it work? Failure and election

• When the primary fails, its heart beat stops
• Other replicas **vote** to elect a new primary
  – based on replica **priority** and **status of replication**
• mongod process can participate in election without storing or serving data (**arbiter**)
Connection to server: multiple server

- You should specify multiple servers in case the primary fails
  
  `mongodb://server1,server2,server3/?replicateSet=set0`

- Will select the primary automatically

```bash
mongodb://server1,server2,server3/?replicateSet=set0
```

![Diagram showing connection to servers](image_url)
Accessing secondaries: read preference

- By default, connects to **only** primary
  - even if you specify a secondary server
- `readPreference` specifies the server processing the query

<table>
<thead>
<tr>
<th>Read preference</th>
<th>Connect to</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary</td>
<td>only the primary</td>
</tr>
<tr>
<td>secondary</td>
<td>only a secondary</td>
</tr>
<tr>
<td>preferPrimary</td>
<td>try primary, then secondary</td>
</tr>
<tr>
<td>preferSecondary</td>
<td>try secondary, then primary</td>
</tr>
<tr>
<td>nearest</td>
<td>The closest server</td>
</tr>
</tbody>
</table>
Specifying the read preference

- Specify the read preference in the connection string

```csharp
client = new MongoClient("mongodb://s1,s2?replicaSet=set0&readPreference=primary");
var server = client.GetServer();
```

- Or specify it for the database

```csharp
var db = server.GetDatabase("myDatabase", new MongoDBDatabaseSettings(){ReadPreference = new ReadPreference(ReadPreferenceMode.Secondary)});
```
Reading from secondary issues

- By default, reading from a secondary can pose **consistency issues**
  - **strict consistency** = "Everybody sees the same state"
  - **read my writes** = "I read what I write"
  - **eventually consistency** = "Eventually, my read will be consistent with my write but nothing is guaranteed right now"
Default: **Strict consistency**

- By default, all reads and writes go through primary
- No consistency issue
Read from secondary: "eventual" consistency

- Writes to primary but read from secondary
- Secondary might be "late"
Write concerns for “read my writes” consistency

• Write concern specifies how much to wait
  – by default, wait only for write completion on primary
  – can specify to wait for primary
## Write concerns

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Don't wait for network (one way)</td>
</tr>
<tr>
<td>Unacknowledged (0)</td>
<td>Don't wait for write</td>
</tr>
<tr>
<td>Acknowledged (1)</td>
<td>Wait for primary</td>
</tr>
<tr>
<td>W2 (2), W3 (3), W4 (4)</td>
<td>Wait for 2, 3 or 4 servers (including primary)</td>
</tr>
<tr>
<td>WMajority (&quot;majority&quot;)</td>
<td>Wait for a majority of servers</td>
</tr>
</tbody>
</table>
Setting write concerns

• At the database level

```csharp
var db = server.GetDatabase("myDB", 
    new MongoDBDatabaseSettings() { 
        ReadPreference = new ReadPreference(ReadPreferenceMode.Secondary), 
        WriteConcern = WriteConcern.WMajority 
    });
```

• Override for one operation

```csharp
myCollection.Insert(someDocument, WriteConcern.Unacknowledged);
```
Non-core service replicas

- Consider using a replica for analytics/reporting services
- For backup, consider making it hidden so that it does not get client requests
Summary

• Choose the right kind of DB for the job at hand
• Document databases are the best shot at general DBs
• MongoDB is the most popular document DB
• The key techniques of scaling MongoDB are
  1. Document design
  2. Indexes
  3. In-place updates
  4. Scale out via replication and sharding
Thanks for coming!

STAY IN TOUCH

Blog: blog.michaelckennedy.net
Twitter: @mckennedy
GitHub: github.com/mikeckennedy