CONTENTS INCLUDE:

Getting Started with NoSQL and Data Scalability

By Eugene Ciurana

INTRODUCTION

The DZone Refcard #43 is an introduction to system high availability and scalability terminology and techniques (http://refcardz.dzone.com/refcardz/scalability). The next logical step is the scalable handling of massive data volumes resulting from having these powerful processing capabilities.

This Refcard demystifies NoSQL and data scalability techniques by introducing some core concepts. It also offers an overview of current technologies available in this domain and suggests how to apply them.

What is Data Scalability?

Data scalability is the ability of a system to store, manipulate, analyze, and otherwise process ever increasing amounts of data without reducing overall system availability, performance, or throughput.

Data scalability is achieved by a combination of more powerful processing capabilities and larger but efficient storage mechanisms.

Relational and hierarchical databases scale up by adding more processors, more storage, caching systems, and such. Soon they hit either a cost or a practical scalability limit because they are difficult or impossible to scale out. These database management systems are designed as single units that must maintain data integrity, and enforce schema rules to guarantee it. This rigidity is what promotes upward, but not outward, scalability.

Hot Tip Oracle RAC is a cluster of multiple computers with access to a common database. This is considered only vertically scalable because the processing (usually in the form of stored procedures) may scale out, but the shared storage facilities don't scale with the cluster.

Data integrity and schemas are suited for handling transactional, normalized, uniform data. They handle unstructured or rapidly evolving data structures with difficulty or exponentially larger costs.

Hot Tip Data replication is not the same as data scalability!

SCALABLE DATA ARCHITECTURES

There are two general kinds of architectures used for building scalable data systems: **data grids** and **NoSQL** systems.

Implementations of either often share characteristics from the other.

Data Grids

Data grids process workloads defined as independent jobs that don't require data sharing among processes. Storage or network may be shared across all nodes of the grid, but intermediate results have no bearing on other jobs progress or on other nodes in the grid, such as a MapReduce cluster.

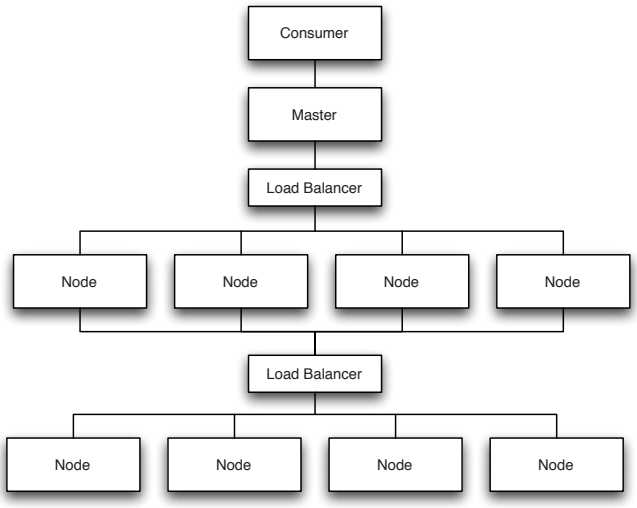


Figure 1 - Data Grid

Data grids expose their functionality through a single API (either a Web service or native to the application programming language) that abstracts its topology and implementation from its data processing consumer.

Areas of Application

- Financial modeling
- Data mining
- Click stream analytics
- Document clustering
- Distributed sorting or grepping
- Simulations
- Inverted index construction
- Protein folding

NoSQL

NoSQL describes a horizontally scalable, non-relational database with built-in replication support. Applications interact with it through a simple API, and the data is stored in a "schema-free", flat addressing repository, usually as large files or data blocks. The repository is often a custom file system designed to support NoSQL operations.

Hot Tip NoSQL is intended as shorthand for "not only SQL." Complete architectures almost always mix traditional and NoSQL databases.

NoSQL setups are best suited for non-OLTP applications that process massive amounts of structured and unstructured data at a lower cost and with higher efficiency than RDBMs and stored procedures.

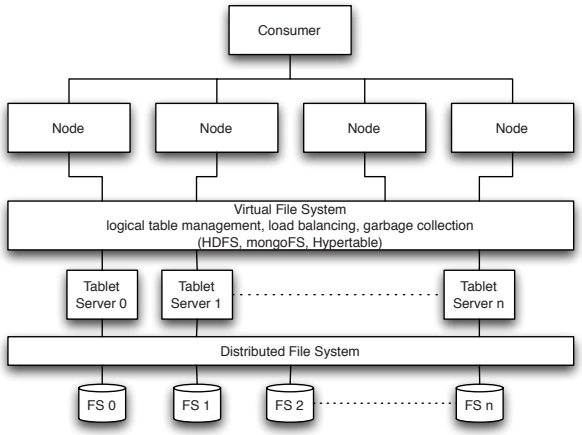


Figure 2 - NoSQL Topology

NoSQL storage is highly replicated (a commit doesn't occur until the data is successfully written to at least two separate storage devices) and the file systems are optimized for write-only commits. The storage devices are formatted to handle large blocks (32 MB or more). Caching and buffering are designed for high I/O throughput. The NoSQL database is implemented as a data grid for processing (mapReduce, queries, CRUD, etc.)

Areas of Application

- Document storage
- Object databases
- Graph databases
- Key/value stores
- Eventually consistent key/value stores

This list is the implementation counterpart to the data grid areas of application; the data store feeds the computational network and, together, they form the NoSQL database.

NoSQL vs RDBMS vs OO Analogies

NoSQL	RDBMS	OO
Kind	Table	Class
Entity	Record	Object
Attribute	Column	Property

IS NoSQL FOR YOU?

Preparation:

- Don't fall prey to the NoSQL fad
- Don't be stubborn; neither NoSQL nor traditional databases apply to all cases
- Apply the CAP Theorem to your use cases to determine feasibility

Brewer's (CAP) Theorem

It's impossible for a distributed computer system to simultaneously provide all three of these guarantees:

- Consistency (all nodes see the same data at the same time)
- Availability (node failures don't prevent survivors from continuing to operate)
- Partition tolerance (no failures less than total network failures cause the system to fail)

Since only two of these characteristics are guaranteed for any given scalable system, use your functional specification and business SLA (service level agreement) to determine what your minimum and target goals for CAP are, pick the two that meet your requirements, and proceed to implement the appropriate technology.

Hot Tip Rule of Thumb: NoSQL's primary goal is to achieve horizontal scalability. It attains this by reducing transactional semantics and referential integrity.

Use Figure 3 to identify the best match between your application's CAP requirements and the suggested SQL and NoSQL systems listed.

Figure 3 - CAP Selection Chart
(source: Nathan Hurst's Blog)

mongoDB

mongoDB is a document-based NoSQL database that bridges the gap between scalable key-value stores like Datastore and Memcache DB, and RDBMS's querying and robustness capabilities. Some of its main features include:

- **Document-oriented storage** - data is manipulated as JSON-like documents
- **Querying** - uses JavaScript and has APIs for submitting queries in every major programming language
- **In-place updates** - atomicity
- **Indexing** - any attribute in a document may be used for indexing and query optimization
- **Auto-sharding** - enables horizontal scalability
- **Map/reduce** - the mongoDB cluster may run smaller MapReduce jobs than a Hadoop cluster with significant cost and efficiency improvements

mongoDB implements its own file system to optimize I/O throughput by dividing larger objects into smaller chunks. Documents are stored in two separate collections: files containing the object meta-data, and chunks that form a larger document when combined with database accounting information. The mongoDB API provides functions for manipulating files, chunks, and indices directly. The administration tools enable GridFS maintenance.

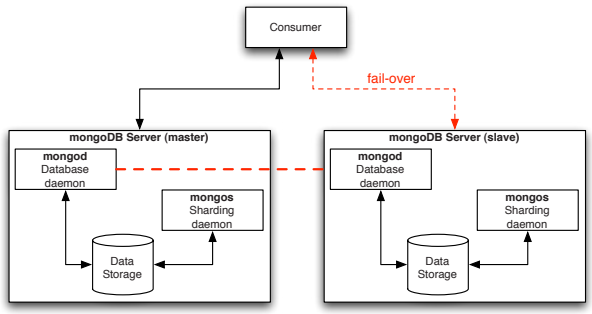


Figure 5 - mongoDB Cluster

A mongoDB cluster consists of a master and a slave. The slave may become the master in a fail-over scenario, if necessary. Having a master/slave configuration (also known as Active/Passive or A/P cluster) helps ensure data integrity since only the master is allowed to commit changes to the store at any given time. A commit is successful only if the data is written to GridFS and replicated in the slave.

Hot Tip mongoDB also supports a limited master/master configuration. It's useful only for inserts, queries, and deletions by specific object ID. It must not be used if updates of a single object may occur concurrently.

Caching

mongoDB has a built-in cache that runs directly in the cluster without external requirements. Any query is transparently cached in RAM to expedite data transfer rates and to reduce disk I/O.

Document Format

mongoDB handles documents in BSON format, a binary-encoded JSON representation. BSON is designed to be traversable, lightweight and efficient. Applications can map BSON/JSON documents using native representations like dictionaries, lists, and arrays, leaving the BSON translation to the native mongoDB driver specific to each programming language.

BSON Example

```
{
  'name' : 'Tom',
  'age' : 42
}
```

Language	Representation
Python	`{ 'name' : 'Tom', 'age' : 42 }`
Ruby	`{ "name" => "Tom", "age" => 42 }`
Java	`BasicDBObject d; d = new BasicObject(); d.put("name", "Tom"); d.put("age", 42);`
PHP	`array( "name" => "Tom", "age" => 42);`

Dynamic languages offer a closer object mapping to BSON/JSON than compiled languages.

The complete BSON specification is available from: http://bsonspec.org/

mongoDB Programming

Programming in mongoDB requires an active server running the mongod and the mongos database daemons (see Figure 5), and a client application that uses one of the language-specific drivers.

Hot Tip All the examples in this Refcard are written in Python for conciseness.

Starting the Server

Log on to the master server and execute:

```
[servername:user] ./mongod
```

The server will display its status messages to the console unless stdout is redirected elsewhere.

Programming Example

This example allocates a database if one doesn't already exist, instantiates a collection on the server, and runs a couple of queries.

The mongoDB Developer Manual is available from: http://www.mongodb.org/display/DOCS/Manual

```
#!/usr/bin/env jython

import pymongo

from pymongo import Connection

connection = Connection('servername', 27017)

db = connection['people_database']

peopleList = db['people_list']

person = {
  'name' : 'Tom',
  'age'  : 42 }
```

```
peopleList.insert(person)

person = {
  'name' : 'Nancy',
  'age'  : 69 }

peopleList.insert(person)

# find first entry:
person = peopleList.find_one()

# find a specific person:

person = peopleList.find_one({ 'name' : 'Joe'})

if person is None:
  print "Joe isn't here!"
else:
  print person['age']

# bulk inserts
persons = [{ 'name' : 'Joe' }, {'name' : 'Sue'}]

peopleList.insert(persons)

# queries with multiple results

for person in peopleList.find():
  print person['name']

for person in peopleList.find({'age' : {'$ge' : 21}}).sort('name'):
  print person['name']

# count:

nDrinkingAge = peopleList.find({'age' : {'$ge' : 21}}).count()

# indexing

from pymongo import ASCENDING, DESCENDING

peopleList.create_index([('age', DESCENDING), ('name', ASCENDING)])
```

The PyMongo documentation is available at: http://api.mongodb.org/python - guides for other languages are also available from this web site.

The code in the previous example performs these operations:

- Connect to the database server started in the previous section
- Attach a database; notice that the database is treated like an associative array
- Get a collection (loosely equivalent to a table in a relational database), treated like an associative array
- Insert one or more entities
- Query for one or more entites

Although mongoDB treats all these data as BSON internally, most of the APIs allow the use of dictionary-style objects to streamline the development process.

Object ID

A successful insertion into the database results in a valid Object ID. This is the unique identifier in the database for a given document. When querying the database, a return value will include this attribute:

```
{
  "name" : "Tom",
  "age" : 42,
  "_id" : ObjectId('999999')
}
```

Users may override this Object ID with any argument as long as it's unique, or allow mongoDB to assign one automatically.

Common Use Cases

- **Caching** - more robust capabilities, plus persistence, when compared against a pure caching system
- **High volume processing** - RDBMS may be too expensive or slow to run in comparison
- **JSON data and program objects storage** - many RESTful web services provide JSON data; they can be stored in mongoDB without language serialization overhead (especially when compared against XML documents)
- **Content management systems** - JSON/BSON objects can represent any kind of document, including those with a binary representation

mongoDB Drawbacks

- **No JOIN operations** - each document is stand-alone
- **Complex queries** - some complex queries and indices are better suited for SQL
- **No row-level locking** - unsuitable for transactional data without error prone application-level assistance

If any of these is part of the functional requirements, a SQL database would be better suited for the application.

GIGASPACES XAP

The GigaSpaces eXtreme Application Platform is a data grid designed to replace traditional application servers. It operates based on an event-processing model where the application dispatches objects to the processing nodes associated with a given data partition. The system may be configured so that data state on the grid may trigger events, or an application may dispatch specific commands imperatively. GigaSpaces XAP also manages all threading and execution aspects of the operation, including thread and connection pools. GigaSpaces XAP implements Spring transactions and auto-recovery. The system detects any failed operations in a computational node and automatically rolls back the transaction; it then places it back in the space where another node picks it up to complete processing.

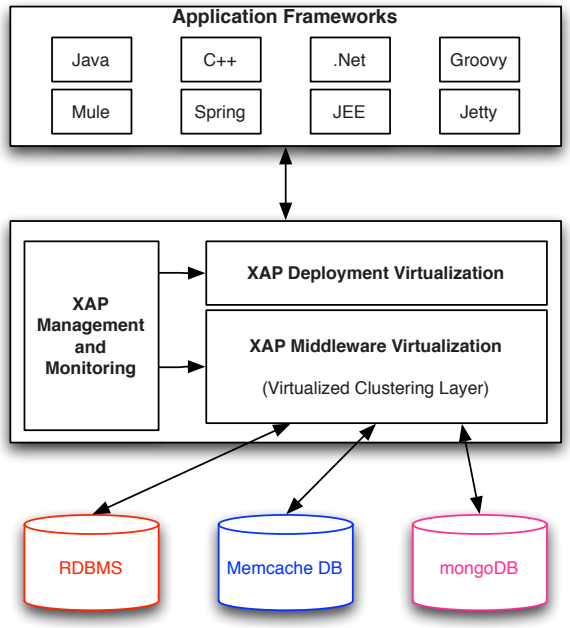


Figure 4 - GigaSpaces XAP Data Grid

GigaSpaces provides data persistence, distributed processing, and caching by interfacing with SQL and NoSQL data stores. The GigaSpaces API abstracts all back-end operations (job dispatching, data persistence, and caching) and makes

it transparent to the application. GigaSpaces XAP may implement distributed computing operations like MapReduce and run them in its nodes, or it may dispatch them for processing to the underlying subsystem if the functionality is available (e.g. mongoDB, Hadoop). The application may implement transactions using the Spring API, the GigaSpaces XPA transactional facilities, or by implementing a workflow where specific NoSQL stores handle entities or groups of entities. This must be implemented explicitly in systems like mongoDB, but may exist in other NoSQL systems like Google App Engine's Datastore.

GigaSpaces XAP Programming

The GigaSpaces XAP API is very rich and covers many aspects beyond NoSQL and data scalability areas like configuration management, deployment, web services, third-party product integration, etc.

The GigaSpaces XAP documentation is at:
http://www.gigaspaces.com/wiki/display/XAP71/Programmer%27s+Guide

NoSQL operations may be implemented over these APIs:

- **SQLQuery** - allows querying a space using a SQL-like syntax and regular expressions; do not confuse it with JDBC support.
- **Persistency** - mostly supports RDBMSs but may implement other persistency mechanisms through the External Data Source Components API.
- **memcached** - support for key/value pair distributed dictionaries available to any client in the grid; entities are automatically made available across all nodes. The memcached API is implemented on top of the data grid, and it's interchangeable with other memcached implementations.
- **Task Execution** - allows synchronous and asynchronous job execution on specific nodes or clusters

The GigaSpaces XAP API is in a minority of stateful NoSQL systems. Most NoSQL systems strive to achieve statelessness to increase scalability and data consistency.

Common Use Cases

- **Real-time analytics** - dynamic data analysis and reporting based on data entered into a system less than a minute before effective time of use
- **Map/reduce** - distributed data processing of large data sets across a computational grid
- **Near-zero downtime** - allows for database schema changes without homebrew master/slave configurations or proprietary RDBMS dependencies

GigaSpaces XAP NoSQL Drawbacks

- **Complexity** - the server, transactional, and grid model are more complex than for other NoSQL systems
- **Application server model** - the API and components are geared toward building applications and transactional logic
- **Steeper learning curve**
- **Higher TCO** - brings a requirement of a specialized, well-trained system administration team with higher requirements than other NoSQL systems

GOOGLE APP ENGINE DATASTORE

The Datastore is the main scalability feature of Google App Engine applications. It's not a relational database or a façade for one. Datastore is a public API for accessing Google's Bigtable high-performance distributed database system. Think of it as a sparse array distributed across multiple servers that also allows an infinite number of columns and rows. Applications may even define new columns "on the fly". The Datastore scales by adding new servers to a cluster; Google provides this functionality without user participation.

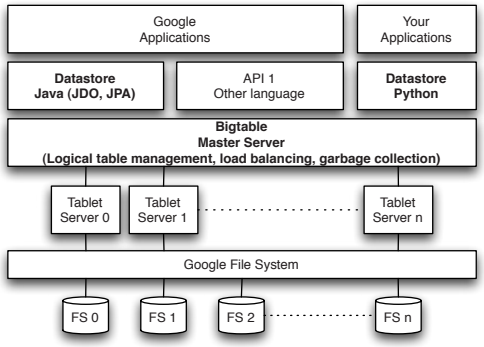


Figure 6 - Datastore Architecture

Datastore operations are defined around entities. Entities can have one-to-many or many-to-many relationships. The Datastore assigns unique IDs unless the application specifies a unique key. Datastore also disallows some property names that it uses for housekeeping. The complete Datastore documentation is available from:
http://code.google.com/appengine/docs/python/datastore/

Hot Tip Did you notice the parallels between Datastore and mongoDB so far? Many NoSQL database implementations have solved similar problems in similar ways.

Transactions and Entity Groups

Datastore supports transactions. These transactions are only effective on entities that belong to the same entity group. Entities in a given group are guaranteed to be stored on the same server.

Datastore Programming

The programming model is based on inheritance of basic entities, db.Model and db.Expando. Persistent data is mapped onto an entity specialization of either of these classes. The API provides persistence and querying instance methods for every entity managed by the Datastore.

Programming Example

The Datastore API is simpler than other NoSQL APIs and is highly optimized to work in the App Engine environment. In this example we insert data into the data store, then run a query:

```
from google.appengine.ext import db

class Person(db.Model):
  name = db.StringProperty(required=True)
  age  = db.IntegerProperty(require=True)

person = Person(name = 'Tom', age = 42)

person.put()

person = Person(name = 'Sue', age = 69)

person.put()
```

```
# find a specific person

query = Person.all() # every entity!

query.filter('age > ', 20)
query.order('name')

peopleList = query.fetch() # up to 1000

for person in peopleList:
  print person.name
```

This example performs these operations:

- Defines a Person kind and associates it with the Datastore
- Persists new items using the put() method
- Defines and executes a query; notice that the query conditions are expressed as strings

Gql - the Google Query Language

Datastore also allows queries in a custom, SQL-like language. The query in the previous example could be expressed as:

```
SELECT * FROM Person WHERE age > 20
    ORDER BY name ASC
```

Gql is useful for writing more expressive queries than those written in the Python or Java APIs.

Hot Tip Careful! Python vs. Gql queries have different performance and quota characteristics that may impact your cost or functionality! Refer to the Datastore documentation for a discussion of how they differ.

Common Use Cases

- **Massive scalability -** Datastore offers ultimate scalability by leveraging Google's own infrastructure for persistent storage
- **Google App Engine applications -** there is no alternative mechanism for data storage on this platform
- **Data rich RESTful Web services -** use the Datastore and App Engine infrastructure to offload traditional data centers when stateless, data-intensive web services must be implemented

Datastore Drawbacks

- **Vendor lock-in -** persistence and queries are tightly coupled with the Datastore and the Datastore API is far from being an industry standard
- **Availability -** Datastore has been known to fail and the EULA doesn't allow more than 4-nines SLAs
- **Quotas -** Datastore utilization costs per data access and for processor time
- **Query limits -** Result sets are limited to return a maximum of 1,000 entities, forcing queries be needlessly complex

STAYING CURRENT

Do you want to know about specific projects and use cases where NoSQL and data scalability are the hot topics? Join the scalability newsletter:

http://eugeneciurana.com/scalablesystems

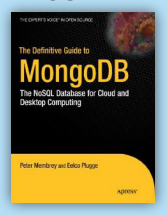
ABOUT THE AUTHOR

Eugene Ciurana is an open-source evangelist who specializes in the design and implementation of mission-critical, high-availability large scale systems. Over the last two years, Eugene designed and built hybrid cloud scalable systems for leading financial, software, insurance, and healthcare companies in the US, Japan, and Europe. As chief liaison between Walmart.com Global and the ISD Technology Council, he led the official adoption of Linux and other open-source technologies at Walmart Stores Information Systems Division in 2006.

Publications

- Developing with Google App Engine
- DZone Refcard #43: Scalability and High Availability
- DZone Refcard #38: SOA Patterns
- The Tesla Testament: A Thriller

RECOMMENDED BOOK



MongoDB, a cross-platform NoSQL database, is the fastest-growing new database in the world. MongoDB provides a rich document orientated structure with dynamic queries that you'll recognize from RDMBS offerings such as MySQL. In other words, this is a book about a NoSQL database that does not require the SQL crowd to re-learn how the database world works!

BUY NOW
books.dzone.com/books/mongodb

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
140 Preston Executive Dr.
Suite 100
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-75-2
ISBN-10: 1-934238-75-9
50795
9 781934 238752
$7.95

Version 1.0