NoSQL Data Management: Concepts and Systems

November 8, 2018

BUAN 6320 Database Foundations

History

- SQL Databases were dominant for decades
  - Persistent storage
  - Standards based
  - Concurrency Control
  - Application Integration
  - ACID
  - Designed to run on a single big machine

- Cloud computing changes that dramatically
  - Cluster of machines
  - Large amount of unreliable machines
  - Distributed System
  - Schema-free unstructured Big Data

Methods to Run a Database

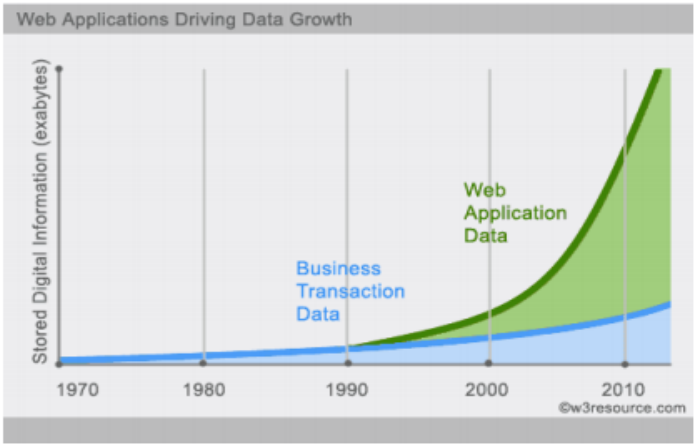
- Virtual Machine Image
  - Users purchase virtual machine instances to run a database on these
  - Upload and setup own image with database, or use ready-made images with optimized database installations
  - E.g. Oracle Database 11g Enterprise Edition image for Amazon EC2 and for Microsoft Azure.
- Database as a service (DBaaS)
  - Using a database without physically launching a virtual machine instance
  - No configuration or management needed by application owners
  - E.g. Amazon Web Services provide SimpleDB, Amazon Relational Database Service (RDS), DynamoDB
- Managed database hosting
  - Not offered as a service, but hosted and managed by the cloud database vendor
  - E.g. Rackspace offers managed hosting for MySQL
- TOSCA
  - Description of Cloud Services as Topology combined with the database stack
  - Vendor-neutral automatic provisioning and management with OpenTOSCA
  - Policies to define security requirements of the Cloud Service
  - Portable and interoperable definition of data security and compliance aspects

Which Data Model?

- Relational Databases
  - Standard SQL database available for Cloud Environments as Virtual Machine Image or as a service depending on the vendor
  - Not cloud-ready: Difficult to scale
- NoSQL databases
  - Database which is designed for the cloud
  - Built to serve heavy read/write loads
  - Good ability to scale up and down
  - Applications built based on SQL data model require a complete rewrite
  - E.g. Apache Cassandra, CouchDB and MongoDB

How to scale the data management?

- Vertical scaling – Scale up



- Horizontal scaling – Scale out

Who Uses NoSQL?

Google – Big Table, Google Apps, Google Search

Facebook – Social network

Twitter

Amazon – DynamoDB and SimpleDB

CERN

GitHub

Definition and Goals of NoSQL databases

- No formal NoSQL definition available!
- Store very large scale data called “Big data”
- Typically scale horizontally
- Simple query mechanisms
- Often designed and set up for
- a concrete application
- Typical NoSQL characteristics:
  - Non-relational
  - Schema-free
  - Open Source
  - Simple API
  - Distributed
  - Eventual consistency

Non-relational are Schema-free

- NoSQL databases generally do not follow the relational model
- Do not provide tables with flat fixed-column records
- Work with self-contained (hierarchical) aggregates or BLOBs
- No need for object-relational mapping and data normalization
- No complex and costly features like query languages, query planners, referential integrity, joins, ACID
- Most NoSQL databases are schema-free or have relaxed schemas
- No need for definition of any sort of schema of the data
- Allows heterogeneous structures of data in the same domain

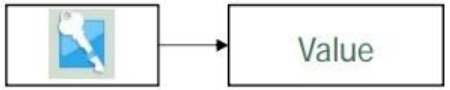
Simple API and Distributed

- Often simple interfaces for storage and querying data provided
- APIs often allow low-level data manipulation and selection methods
- Often no standard based query language is used
- Text-based protocols often using HTTP REST with JSON
- Web-enabled databases running as internet-facing services

- Several NoSQL databases can be executed in a distributed fashion
- Providing auto-scaling and fail-over capabilities
- Often ACID is sacrificed for scalability and throughput
- Often no synchronous replication between distributed nodes is possible, e.g. asynchronous Multi-Master Replication, peer-to-peer, HDFS Replication
- Only providing eventual consistency

Core Categories of NoSQL Systems

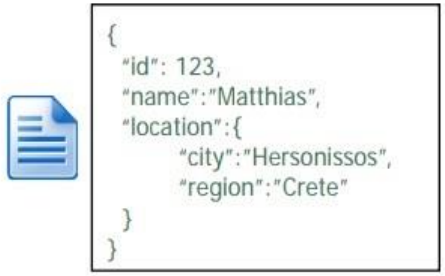
- Key-Value Stores
  - Manage associative arrays
  - Big hash table



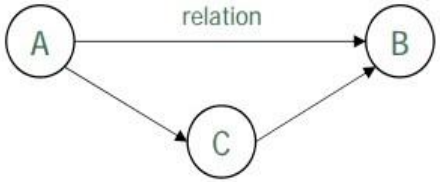
- Wide Column Stores
  - Each storage block contains only data from one column
  - Read and write is done using columns
  - (rather than rows – like in SQL)

Row ID	Columns...		
1	Name	Website	
	Preston	www.example.com	
2	Name	Website	
	Julia	www.example.com	
3	Name	Email	Website
	Alice	example@example.com	www.example.com

- Document Stores
  - Store documents consisting of tagged values
  - Data is a collection of key value pairs
  - Provides structure and encoding of the managed data
  - Encoded using XML, JSON, BSON
  - Schema-free



- Graph DB
  - Network database using graphs with node and edges for storage
  - Nodes represent entities, edges represent their relationships



SQL and NoSQL Systems

Relational: **MySQL**, PostgreSQL, SQLite, Firebird, MariaDB, **Oracle DB**, **SQL server**, IBM DB2, IBM Informix, Teradata

Key value-stores: **Memcachedb**, Redis, Riak, **Amazon DynamoDB**, Voldemort, FoundationDB, leveldb, BangDB, KAI, hamsterdb, Tarantool, Maxtable, HyperDex, Genomu

Column family: **Big table**, Hbase, hyper table, **Cassandra**, Apache Accumulo

Document: **Mongo DB**, Couch DB, Rethink DB, Raven DB, terrastore, Jas DB, Raptor DB, djon DB, EJDB, denso DB, Couchbase

Graph databases: AllegroGraph, **Neo4j**, OrientDB, InfiniteGraph, graphbase, sparkledb, flockdb, BrightstarDB

Object: ZODB, DB4O, Eloquera, Versant, Objectivity DB, VelocityDB

RDF Stores: Apache Jena, Sesame

Multimodel Databases: arangodb, Datomic, Orient DB, FatDB, AlchemyDB

XML Databases: BaseX, Sedna, eXist

Hierarchical: InterSystems Caché, GT.M

SQL to NoSQL Terminology

SQL Terms/Concepts	MongoDB Terms/Concepts
database	database
table	collection
row	document or BSON document
column	field
index	index
table joins	$lookup, embedded documents
primary key Specify any unique column or column combination as primary key.	primary key In MongoDB, the primary key is automatically set to the _idfield.
aggregation (e.g. group by)	aggregation pipeline See the SQL to Aggregation Mapping Chart.
transactions	transactions **TIP** For many scenarios, the denormalized data model (embedded documents and arrays) will continue to be optimal for your data and use cases instead of multi-document transactions. That is, for many scenarios, modeling your data appropriately will minimize the need for multi-document transactions.

NoSQL Overview

- Introduction to NoSQL

- Basic Concepts for NoSQL
  - CAP-Theorem
  - Eventual Consistency
  - Consistent Hashing
  - MVCC-Protocol
  - Query Mechanisms for NoSQL

- Overview of NoSQL Systems

SQL to NoSQL

The following table presents some database executables and the corresponding MongoDB executables. This table is *not* meant to be exhaustive.

MongoDB	MySQL	Oracle	MS SQL	DB2
Database Server	mongod	mysqld	MSSQL$SQLEXPRESS	IDS
Database Client	mongo	mysql	SSMS	DB-Access

GENERAL STRUCTURE OF NOSQL DATA

- {
- _id: ObjectId("509a8fb2f3f4948bd2f983a0"),
- user_id: "abc123",
- age: 55,
- status: 'A'
- }

CREATE TABLE

The following table presents the various SQL statements related to table-level actions and the corresponding MongoDB statements.

SQL Schema Statements	MongoDB Schema Statements
CREATE TABLE people (id MEDIUMINT **NOT NULL** AUTO_INCREMENT, user_id Varchar(30), age Number, status char(1), **PRIMARY KEY** (id))	Implicitly created on first `insertOne()` or `insertMany()`operation. The primary key `_id` is automatically added if `_id` field is not specified. db.people.insertOne({ user_id: "abc123", age: 55, status: "A" }) However, you can also explicitly create a collection: db.createCollection("people")
DROP TABLE people	db.people.drop()

ALTER TABLE ADD COLUMN

SQL Schema Statements	MongoDB Schema Statements
ALTER TABLE people **ADD** join_date DATETIME	Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level. However, at the document level, `updateMany()` operations can add fields to existing documents using the `$set` operator. db.people.updateMany({ }, { $set: { join_date: **new** Date() } })

ALTER TABLE DROP COLUMN

SQL Schema Statements	MongoDB Schema Statements
ALTER TABLE people **DROP COLUMN** join_date	Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level. However, at the document level, `updateMany()` operations can remove fields from documents using the `$unset` operator. db.people.updateMany({ }, { $unset: { "join_date": "" } })

CREATE INDEX

SQL Schema Statements	MongoDB Schema Statements
CREATE INDEX idx_user_id_asc **ON** people(user_id)	db.people.createIndex({ user_id: 1 })
CREATE INDEX idx_user_id_asc_age_desc **ON** people(user_id, age **DESC**)	db.people.createIndex({ user_id: 1, age: -1 })
DROP TABLE people	db.people.drop()

INSERT SQL AND NoSQL

The following table presents the various SQL statements related to inserting records into tables and the corresponding MongoDB statements.

SQL Schema Statements	MongoDB Schema Statements
INSERT INTO people(user_id, age, status) **VALUES** ("bcd001", 45, "A")	db.people.insertOne({ user_id: "bcd001", age: 45, status: "A" })

SELECTS

The following table presents the various SQL statements related to reading records from tables and the corresponding MongoDB statements.

NOTE: The find() method always includes the _id field in the returned documents unless specifically excluded through projection. Some of the SQL queries below may include an _id field to reflect this, even if the field is not included in the corresponding find() query.

SQL SELECT Statements	MongoDB find() Statements
SELECT * FROM people	db.people.find()
SELECT id, user_id, status FROM people	db.people.find({ }, { user_id: 1, status: 1 })

SELECTS, Cont.

SQL Schema Statements	MongoDB Schema Statements
SELECT user_id, status **FROM** people	db.people.find({ }, { user_id: 1, status: 1, _id: 0 })
SELECT * **FROM** people **WHERE** status = "A"	db.people.find({ status: "A" })
SELECT user_id, status **FROM** people **WHERE** status = "A"	db.people.find({ status: "A" }, { user_id: 1, status: 1, _id: 0 })

SELECTS, Cont.

SQL Schema Statements	MongoDB Schema Statements
SELECT * **FROM** people **WHERE** status != "A"	db.people.find({ status: { $ne: "A" } })
SELECT * **FROM** people **WHERE** status = "A" **AND** age = 50	db.people.find({ status: "A", age: 50 })
SELECT * **FROM** people **WHERE** status = "A" **OR** age = 50	db.people.find({ $or: [{ status: "A" } , { age: 50 }] })
SELECT * **FROM** people **WHERE** age > 25	db.people.find({ age: { $gt: 25 } })

SELECTS, Cont.

SQL Schema Statements	MongoDB Schema Statements
SELECT * **FROM** people **WHERE** age < 25	db.people.find({ age: { $lt: 25 } })
SELECT * **FROM** people **WHERE** age > 25 **AND** age <= 50	db.people.find({ age: { $gt: 25, $lte: 50 } })
SELECT * **FROM** people **WHERE** user_id **like** "%bc%"	db.people.find({ user_id: /bc/ }) -or- db.people.find({ user_id: { $regex: /bc/ } })

SELECTS, Cont.

SQL Schema Statements	MongoDB Schema Statements
SELECT * FROM people WHERE user_id like "bc%"	db.people.find({ user_id: /^bc/ }) -or- db.people.find({ user_id: { $regex: /^bc/ } })
SELECT * FROM people WHERE status = "A" ORDER BY user_id ASC	db.people.find({ status: "A" }).sort({ user_id: 1 })
SELECT * FROM people WHERE status = "A" ORDER BY user_id DESC	db.people.find({ status: "A" }).sort({ user_id: -1 })

SELECTS, Cont.

SQL Schema Statements	MongoDB Schema Statements
SELECT COUNT(*) **FROM** people	db.people.count() *or* db.people.find().count()
SELECT COUNT(user_id) **FROM** people	db.people.count({ user_id: { $exists: **true** } }) *or* db.people.find({ user_id: { $exists: **true** } }).count()

SELECTS, Cont.

SQL Schema Statements	MongoDB Schema Statements
SELECT COUNT(*) **FROM** people **WHERE** age > 30	db.people.count({ age: { $gt: 30 } }) *or* db.people.find({ age: { $gt: 30 } }).count()
SELECT DISTINCT(status) **FROM** people	db.people.aggregate([{ $group : { _id : "$status" } }]) or, for distinct value sets that do not exceed the BSON size limit db.people.distinct("status")

SELECTS, Cont.

SQL Schema Statements	MongoDB Schema Statements
SELECT * **FROM** people **LIMIT** 1	db.people.findOne() *or* db.people.find().limit(1)
SELECT * **FROM** people **LIMIT** 5 SKIP 10	db.people.find().limit(5).skip(10)
EXPLAIN SELECT * **FROM** people **WHERE** status = "A"	db.people.find({ status: "A" }).explain()

Update Records

The following table presents the various SQL statements related to updating existing records in tables and the corresponding MongoDB statements.

SQL Update Statements	MongoDB updateMany() Statements
UPDATE people **SET** status = "C" **WHERE** age > 25	db.people.updateMany({ age: { $gt: 25 } }, { $set: { status: "C" } })
UPDATE people **SET** age = age + 3 **WHERE** status = "A"	db.people.updateMany({ status: "A" } , { $inc: { age: 3 } })

Delete Records

The following table presents the various SQL statements related to updating existing records in tables and the corresponding MongoDB statements.

SQL Delete Statements	MongoDB deleteMany() Statements
DELETE FROM people **WHERE** status = "D"	db.people.deleteMany({ status: "D" })
DELETE FROM people	db.people.deleteMany({})