# NoSQL

# Today

- NoSQL: ”new” database systems
  - not typically RDBMS
  - relax on some requirements, gain efficiency and scalability
- New systems choose to use/not use several concepts we learnt so far
  - e.g. “System ---” does not use locks but uses multi-version CC (MVCC) or,

# New Systems

- We will examine a number of SQL and so- called "NoSQL" systems or "data stores"
- Designed to scale simple application loads
  - to do updates as well as reads
  - in contrast to traditional DBMSs and data warehouses
  - to provide good horizontal scalability for simple read/write database operations distributed over many servers
- Originally motivated by Web 2.0 applications
  - these systems are designed to scale to thousands or millions of users

# New Systems vs. RDMS

- When you study a new system, compare it with RDBMS-s on its
  - data model
  - consistency mechanisms
  - storage mechanisms
  - durability guarantees
  - availability
  - query support
- These systems typically sacrifice some of these dimensions
  - e.g. database-wide transaction consistency, in order to achieve others, e.g. higher availability and scalability

# NoSQL

- Many of the new systems are referred to as “NoSQL” data stores
- NoSQL stands for “Not Only SQL” or “Not Relational”
  - not entirely agreed upon
- Next: six key features of NoSQL systems

# NoSQL: Six Key Features

1. the ability to horizontally scale “simple operations” throughput over many servers
2. the ability to replicate and to distribute (partition) data over many servers
3. a simple call level interface or protocol (in contrast to SQL binding)
4. a weaker concurrency model than the ACID transactions of most relational (SQL) database systems
5. efficient use of distributed indexes and RAM for data storage
6. the ability to dynamically add new attributes to data records

# BASE (not ACID ☺)

- Recall ACID for RDBMS desired properties of transactions:
  - Atomicity, Consistency, Isolation, and Durability

- NOSQL systems typically do not provide ACID

- Basically Available
- Soft state
- Eventually consistent

# ACID vs. BASE

- The idea is that by giving up ACID constraints, one can achieve much higher performance and scalability

- The systems differ in how much they give up
  - e.g. most of the systems call themselves "eventually consistent", meaning that updates are eventually propagated to all nodes
  - but many of them provide mechanisms for some degree of consistency, such as multi-version concurrency control (MVCC)

# “CAP” Theorem

- Often Eric Brewer’s CAP theorem cited for NoSQL

- A system can have only two out of three of the following properties:
- Consistency
  - do all clients see the same data?
- Availability
  - is the system always on?
- Partition-tolerance
  - even if communication is unreliable, does the system function?

- The NoSQL systems generally give up consistency
  - However, the trade-offs are complex

# Two foci for NoSQL systems

1. “Simple” operations

2. Horizontal Scalability

# 1. “Simple” Operations

- Reading or writing a small number of related records in each operation
  - e.g. key lookups
  - reads and writes of one record or a small number of records
- This is in contrast to complex queries, joins, or read-mostly access
- Inspired by web, where millions of users may both read and write data in simple operations
  - e.g. search and update multi-server databases of electronic mail, personal profiles, web postings, wikis, customer records, online dating records, classified ads, and many other kinds of data

# 2. Horizontal Scalability

- Shared-Nothing Horizontal Scaling
- The ability to distribute both the data and the load of these simple operations over many servers
  - with no RAM or disk shared among the servers
- Not “vertical” scaling
  - where a database system utilizes many cores and/or CPUs that share RAM and disks
- Some of the systems we describe provide both vertical and horizontal scalability

# What is different in NOSQL systems

- When you study a new NOSQL system, notice how it differs from RDBMS in terms of

1. Concurrency Control
2. Data Storage Medium
3. Replication
4. Transactions

# Choices in NOSQL systems: 1. Concurrency Control

a) Locks
   - some systems provide one-user-at-a-time read or update locks
   - MongoDB provides locking at a field level

b) MVCC

c) None
   - do not provide atomicity
   - multiple users can edit in parallel
   - no guarantee which version you will read

d) ACID
   - pre-analyze transactions to avoid conflicts
   - no deadlocks and no waits on locks

# Choices in NOSQL systems: 2. Data Storage Medium

a) Storage in RAM
   - snapshots or replication to disk
   - poor performance when overflows RAM

b) Disk storage
   - caching in RAM

# Choices in NOSQL systems: 3. Replication

- whether mirror copies are always in sync

a) Synchronous

b) Asynchronous
   - faster, but updates may be lost in a crash

c) Both
   - local copies synchronously, remote copies asynchronously

# Choices in NOSQL systems: 4. Transaction Mechanisms

a) support
b) do not support
c) in between
   – support local transactions only within a single object or “shard”
   – shard = a horizontal partition of data in a database

# Comparison from Cattell's paper (2011)

| System | Conc Contol | Data Storage | Repli-cation | Tx |
|---|---|---|---|---|
| Redis | Locks | RAM | Async | N |
| Scalaris | Locks | RAM | Sync | L |
| Tokyo | Locks | RAM or disk | Async | L |
| Voldemort | MVCC | RAM or BDB | Async | N |
| Riak | MVCC | Plug-in | Async | N |
| Membrain | Locks | Flash + Disk | Sync | L |
| Membase | Locks | Disk | Sync | L |
| Dynamo | MVCC | Plug-in | Async | N |
| SimpleDB | None | S3 | Async | N |
| MongoDB | Locks | Disk | Async | N |
| Couch DB | MVCC | Disk | Async | N |
| Terrastore | Locks | RAM+ | Sync | L |
| HBase | Locks | Hadoop | Async | L |
| HyperTable | Locks | Files | Sync | L |
| Cassandra | MVCC | Disk | Async | L |
| BigTable | Locks+stamps | GFS | Sync+Async | L |
| PNUTs | MVCC | Disk | Async | L |
| MySQL Cluster | ACID | Disk | Sync | Y |
| VoltDB | ACID, no lock | RAM | Sync | Y |
| Clustrix | ACID, no lock | Disk | Sync | Y |
| ScaleDB | ACID | Disk | Sync | Y |
| ScaleBase | ACID | Disk | Async | Y |
| NimbusDB | ACID, no lock | Disk | Sync | Y |

# Data Model Terminology for NoSQL

- Unlike SQL/RDBMS, the terminology for NoSQL is often inconsistent
- All systems provide a way to store scalar values
  - e.g. numbers and strings
- Some of them also provide a way to store more complex nested or reference values

# Data Model Terminology for NoSQL

- The systems all store sets of attribute-value pairs
  - but use four different data structures

1. Tuple
2. Document
3. Extensible Record
4. Object

# 1. Tuple

- Same as before
- A “tuple” is a row in a relational table
  - attribute names are pre-defined in a schema
  - the values must be scalar
  - the values are referenced by attribute name
  - in contrast to an array or list, where they are referenced by ordinal position

# 2. Document

- Allows values to be nested documents or lists as well as scalar values
  - think about XML or JSON

- The attribute names are dynamically defined for each document at runtime

- A document differs from a tuple in that the attributes are not defined in a global schema
  - and a wider range of values are permitted

# 3. Extensible Record

- A hybrid between a tuple and a document
- families of attributes are defined in a schema
- but new attributes can be added (within an attribute family) on a per-record basis
- Attributes may be list-valued

# 4. Object

- Analogous to an object in programming languages
  - but without the procedural methods
- Values may be references or nested objects

# Example NOSQL systems

- Key-value Stores:
  - Project Voldemort, Riak, Redis, Scalaris, Tokyo Cabinet, Memcached/Membrain/Membase
- Document Stores:
  - Amazon SimpleDB, CouchDB, MongoDB, Terrastore
- Extensible Record Stores:
  - Hbase, HyperTable, Cassandra, Yahoo's PNUTS
- Relational Databases:
  - MySQL Cluster, VoltDB, Clustrix, ScaleDB, ScaleBase, NimbusDB, Google Megastore (a layer on BigTable)

# SQL vs. NOSQL

# Why choose RDBMS over NoSQL : 1/3

1. If new relational systems can do everything that a NoSQL system can, with analogous performance and scalability, and with the convenience of transactions and SQL, NoSQL is not needed

2. Relational DBMSs have taken and retained majority market share over other competitors in the past 30 years
   - (network, object, and XML DBMSs)

# Why choose RDBMS over NoSQL : 2/3

3. Successful relational DBMSs have been built to handle other specific application loads in the past:
   - read-only or read-mostly data warehousing
   - OLTP on multi-core multi-disk CPUs
   - in-memory databases
   - distributed databases, and
   - now horizontally scaled databases

# Why choose RDBMS over NoSQL : 3/3

4. While no “one size fits all” in the SQL products themselves, there is a common interface with SQL, transactions, and relational schema that give advantages in training, continuity, and data interchange

# Why choose NoSQL over RDBMS : 1/3

1. We haven't yet seen good benchmarks showing that RDBMSs can achieve scaling comparable with NoSQL systems like Google's BigTable

2. If you only require a lookup of objects based on a single key
   - then a key-value store is adequate and probably easier to understand than a relational DBMS
   - Likewise for a document store on a simple application: you only pay the learning curve for the level of complexity you require

# Why choose NoSQL over RDBMS : 2/3

3. Some applications require a flexible schema
   - allowing each object in a collection to have different attributes
   - While some RDBMSs allow efficient “packing” of tuples with missing attributes, and some allow adding new attributes at runtime, this is uncommon

# Why choose NoSQL over RDBMS : 3/3

4. A relational DBMS makes "expensive" (multi- node multi-table) operations "too easy"
   – NoSQL systems make them impossible or obviously expensive for programmers

5. While RDBMSs have maintained majority market share over the years, other products have established smaller but non-trivial markets in areas where there is a need for particular capabilities
   – e.g. indexed objects with products like BerkeleyDB, or graph-following operations with object-oriented DBMSs