NoSQL

- "NoSQL" was originally the name of a specific RDBMS project that did not use a SQL interface
- Was co-opted years later to refer to a growing category of non-relational distributed databases
  - Was changed even later to stand for "Not only SQL" instead of "No SQL"
    - When more mature members of the community realized that their "new" approach was not the be-all and end-all of everything
NoSQL

- NoSQL has come to refer to any non-relational database
  - But there are many types of non-relational databases
    - Key-Value
    - Document oriented
    - Column based
    - Graph
NoSQL

• Common characteristic is the lack of a relationship between records
  – No "joins"

• Complexity of searches may also be limited
  – E.g., a key-value store only allows searching by key
Distributed

• Another common characteristic is the ability to *scale out*
  – Distribute across multiple machines
Distributed

• It is (relatively) easy to scale out the application part of a web application
  – Can run multiple application servers with load balancing, etc.
Distributed

• It is difficult to scale out a traditional RDBMS
  – Problem is ACID compliance
    • Atomicity
    • Consistency
    • Isolation
    • Durability
  – Particular problem with consistency
    • If a database is distributed across multiple server then all servers would need to be updated atomically to preserve consistency
BASE

• But not all situations require full ACID compliance
  – Does it really matter if some users see a tweet a few seconds before some others?

• BASE
  – Basically Available
  – Soft state
  – Eventual consistency
CAP Theorem

• The CAP theorem states that in a distributed system you can have at most two, but not all three, of
  – Consistency
    • Data is the same across all nodes
  – Availability
    • Cluster is always available
  – Partition tolerance
    • Continues to function even if there are communication failures between nodes or nodes go down
CAP Theorem

• Possibilities
  – CA (Consistency and Availability)
    • Data is consistent across all nodes as long as all nodes are online
    • Communication failure between nodes could cause inconsistency
  – CP (Consistency and Partition tolerance)
    • Data is consistent across all nodes but entire cluster becomes unavailable if there is a failure
  – AP (Availability and Partition tolerance)
    • Cluster remains available even if there is a communication failure between nodes, but the data across nodes may not be consistent
CAP Theorem

- RDBMSs fall into the CA category
- "NoSQL" databases (or distributed databases in general) tend to fall into either the CP or AP categories
  - Give up either availability or consistency for partition tolerance
  - CP
    - BigTable, Hbase, MongoDB, Redis
  - AP
    - Dynamo, Cassandra, SimpleDB, CouchDB
Queries

• If a database does not support SQL then how is data inserted or retrieved?
  – Generally a program must be written
    • Client libraries for programming languages
    • Database-system specific scripting language
Example

• MongoDB
  – Document-oriented database
  – Key-value where the values are JSON objects
    • Actually stored in BSON (Binary JSON) format
• From the mongo shell:
  – user1 = { name : "Amy", password : "foo" }
  – user2 = { name : "Bob", password : "bar" }
  – db.users.insert(user1)
  – db.users.insert(user2)
  – db.users.find()

    • Will return something like

{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "Amy", password : "foo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "name" : "Bob", password : "bar" }
When you query a collection, MongoDB returns a cursor

- Can iterate over the results using the cursor

```javascript
var results = db.users.find()
while (results.hasNext())
    printjson(results.next())
```
MapReduce

- MapReduce
  - Parallel programming model for processing large distributed data sets
  - Inspired by the map and reduce (a.k.a fold) functions from functional programming
MapReduce

• Functional programming
  – Map
    • Applies a given function to each element of a list
      – (map square [1, 2, 3])
        » Will apply square to each element of [1, 2, 3] resulting in [1, 4, 9]
  – Fold
    • Combines the elements of a list using a function and a base value
      – (fold add 0 [1, 2, 3])
        » Will iterate over the list adding each value to the accumulated value (initialized to the base value) resulting in 6
MapReduce

• Functional programming
  – Fold right vs fold left
    • \((\text{foldl } f \ x \ [a, b, c])\)
      – \((f \ (f \ (f \ x \ a) \ b) \ c)\)
    • \((\text{foldr } f \ [a, b, c] \ x)\)
      – \((f \ a \ (f \ b \ (f \ c \ x)))\)
MapReduce

• The distributed MapReduce algorithm applies this pattern across a cluster
  – Map
    • Master node
      – Takes the input
      – Divides into smaller sub-problems
      – Distributed sub-problems to worker nodes
    • Worker node
      – Processes the sub-problem
      – Returns result to master node
MapReduce

– Reduce
  • Master node
    – Collects the results from the worker nodes
    – Combines the results to form the final answer
  • Could actually have multiple "reduce" nodes

– Parallelization
  • Work nodes can operate independently in parallel

– Pipelining
  • "Map" nodes often return results as they are found
    – Present an iterator interface to the "reduce" nodes
MapReduce

• Count the appearance of each word in a set of documents

```javascript
function map(String name, String document):
   // name: document name
   // document: document contents
   for each word w in document:
      emit (w, 1)

function reduce(String word, Iterator partialCounts):
   // word: a word
   // partialCounts: a list of aggregated partial counts
   sum = 0
   for each pc in partialCounts:
      sum += ParseInt(pc)
   emit (word, sum)
```
MongoDB example

```javascript
db.orders.mapReduce(
    map
    reduce
    query
    output

    function() { emit( this.cust_id, this.amount ); },
    function(key, values) { return Array.sum( values ) },
    { query: { status: "A" },
      out: "order_totals"
 })
```
Transactions

• "NoSQL" databases typically do not support transactions
  – Only operation on a single row/document are atomic
  – Need to use something like two phase commit for transactions
Two Phase Commit

• Phase 1: prepare
  – Manager tells resources to prepare
  – Resources report back when prepared

• Phase 2: commit or rollback
  – Manager tells resources to commit or rollback
  – Resources report back when committed or rolled back
Two Phase Commit

• MongoDB
  – Use two phase commit to update multiple documents "atomically"
    1. Create transaction document in "initial" state with references to all affected rows
    2. Set transaction document to "pending" state
    3. For each affected row
       – Apply update and add transaction document reference to row only if transaction document not already in row
    4. Set transaction document state to "committed"
    5. Remove transaction document reference from all affected rows
    6. Set transaction document state to "done"
Two Phase Commit

1. Create transaction document in "initial" state with references to all affected rows

   ```
   db.transactions.save({source: "A", destination: "B", value: 100, state: "initial"})
   ```

2. Set transaction document to "pending" state

   ```
   t = db.transactions.findOne({state: "initial"})
   db.transactions.update({_id: t._id}, {$set: {state: "pending"}})
   ```
Two Phase Commit

3. Apply update and add transaction document reference to rows

```javascript
db.accounts.update({name: t.source,
    pendingTransactions: {$ne: t._id}},
    {$inc: {balance: -t.value}, $push: {pendingTransactions: t._id}})
```

```javascript
db.accounts.update({name: t.destination,
    pendingTransactions: {$ne: t._id}},
    {$inc: {balance: t.value}, $push: {pendingTransactions: t._id}})
```

4. Set transaction document state to "committed"

```javascript
db.transactions.update({_id: t._id}, {$set: {state: "committed"}})
```
Two Phase Commit

5. Remove transaction document reference from all affected rows

```javascript
db.accounts.update({name: t.source},
    {$pull: {pendingTransactions: t._id}})
```
```
db.accounts.update({name: t.destination},
    {$pull: {pendingTransactions: t._id}})
```

6. Set transaction document state to "done"

```javascript
db.transactions.update({_id: t._id}, {$set: {state: "done"}})
```
Two Phase Commit

• All failures that occur after the first step but before the third step
  – To recover, applications should get a list of transactions in the pending state and resume from the second step

• All failures that occur after the third but before the fifth
  – To recover, application should get a list of transactions in the committed state and resume from the fourth step
Conclusion

• RDBMSs and "NoSQL" database have different strengths and weaknesses
• Pick the right tool for the right job