An important reason for the rise of NoSQL databases was the need for web-based applications to deal with massive amounts of:

- data
- traffic / queries

Solution: use clusters of small commodity machines

- use both fragmentation (aka sharding) and replication

Problem: RDBMs do not scale well to large clusters.

Most NoSQL databases are designed for use on clusters.
Recall: Replication

- Replication involves putting copies of the same collection of records at different sites.

<table>
<thead>
<tr>
<th>account type</th>
<th>interest rate</th>
<th>monthly fee</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard</td>
<td>0%</td>
<td>$10</td>
</tr>
<tr>
<td>bigsaver</td>
<td>2%</td>
<td>$50</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Each copy is referred to as a replica.

Reasons for Replication

- To provide *high availability*
  - allows a database to remain available in the event of a failure at one site
- Can also be used to increase performance.
  - accesses to the database can be spread across the replicas
  - putting replicas in different geographic areas can reduce the distance that data needs to travel
Recall: Types of Replication

- **Synchronous replication**: transactions are guaranteed to see the most up-to-date value of an item.
  - read-any, write-all
  - voting
  - can be too slow in some situations

- **Asynchronous replication**: transactions may not see the most up-to-date value.
  - primary-site
  - peer-to-peer

Replication in MongoDB

- MongoDB uses primary-site replication.
  - one replica is designated the *primary or master* replica
  - all writes go to it
  - the other replicas (the *secondaries*) can only be read
  - changes to the primary are propagated asynchronously to the secondaries

Terminology clarifications:
- In prior notes, we mentioned *client* as an alternate term for secondary, but we'll restrict ourselves to secondary here.
- We'll use client to refer instead to an application that is accessing the database.
- A *replica set* is a group of MongoDB server processes that host the same set of documents.
Replication in MongoDB (cont.)

- By default, reads in MongoDB also go to the primary.
  - guarantees that clients see the most recent version
  - eliminates the performance gains that replication can give
- Other options for reads are also available (see later slides).

Replication in MongoDB (cont.)

- The default picture looks like this:

```
Client Application
    \-----\-----
     |     |     \\
     |     |     \\
 Driver |     |     \\
     |     |     \\
    \-----\-----
             \----
              \----
Primary \-----\-----
      \-----\-----
        \-----\-----
          \-----\-----
Secondary \-----\-----
    \-----\-----
             \----
              \----
Secondary \-----\-----
```

http://docs.mongodb.org/manual/core/replication-introduction

(note: the driver is the language-specific library that clients use when communicating with MongoDB)
More Detail About Reads

- As mentioned earlier, reads go to the primary by default.
- For performance reasons, clients can specify a different read preference.

<table>
<thead>
<tr>
<th>read preference</th>
<th>how it works</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary</td>
<td>all reads go to the primary (the default)</td>
</tr>
<tr>
<td>primaryPreferred</td>
<td>reads usually go to the primary, but can go to a secondary if the primary is unavailable</td>
</tr>
<tr>
<td>secondary</td>
<td>all reads go to a secondary</td>
</tr>
<tr>
<td>secondaryPreferred</td>
<td>opposite of primaryPreferred</td>
</tr>
<tr>
<td>nearest</td>
<td>reads go to the nearest member of the replica set</td>
</tr>
</tbody>
</table>

Reads and Consistency

- Earlier in the semester, we used the term consistency in the context of transactions (the C in ACID).
- In the context of a distributed database, consistency has a somewhat different meaning.
  - used to specify whether reads reflect prior writes

  - **Strict consistency**:
    - reads always reflect prior writes
    - get the most up-to-date value

  - **Eventual consistency**:
    - reads may not immediately reflect prior writes
      - may get stale data
    - given enough time, reads will eventually reflect prior writes
Reads and Consistency (cont.)

• By default, MongoDB provides strict consistency.
  • reading from the primary gives you the latest value

• Reads with non-primary read preferences provide eventual consistency (unless you take special steps).
  • a secondary may not yet have the latest value, but it will eventually get it
  • in the meantime, a given read may get stale data

More Detail About Writes

• All writes go to the primary.

• After applying a write, the primary also logs it.

• Copies of the log are periodically sent to the secondaries, which apply the writes to their replicas.

• To provide increased consistency for clients reading from secondaries, specify a special write concern value.
  • allows you to specify the number of replica-set members that must acknowledge the write before the write operation returns to the client
High Availability

- Because writes (and, by default, reads) go to the primary, MongoDB needs to handle cases in which the primary becomes inaccessible.

- In such cases, the system performs automatic failover.
  - the secondaries hold an election to select a new primary
  - can give secondaries priorities that affect their likelihood of becoming the new primary

- In order to become the new primary, a secondary must be able to reach a majority of replica set's members.
  - why does this make sense?
  - thus, you need to be careful when deciding where the replicas are located
    - put a majority of the replica set in one data center

High Availability (cont.)

- After failover, if the failed primary recovers, it can rejoin the replica set as a secondary.
  - may need to rollback its state to an earlier set of values in order to be consistent with the state of the new primary

- If the failed primary cannot recover, it should be replaced as soon as possible to ensure continued availability.
Sharding in MongoDB

- Sharding == horizontal fragmentation.
  - divides a collection of documents among multiple servers
  - each subset of the documents is a referred to as a *shard*
  - if the database is also replicated, each shard corresponds to a replica set.

![Sharding Diagram](http://docs.mongodb.org/manual/core/sharding-introduction/)

Sharding in MongoDB (cont.)

- Sharding allows MongoDB to handle:
  - large amounts of data
  - large volumes of requests
    - by providing increased parallelism
  - *Query routers* send client requests to the correct shard(s).
    - based on the specified *shard key*:
      - the field or combination of fields that is used to determine which shard a given document belongs to
      - how could the choice of shard key affect performance?
Sharding in MongoDB

- Given the shard key, MongoDB determines which shard keys should go on which shard.

- It uses either:
  - range-based sharding: divide the range of possible shard-key values into subranges
    - each subrange goes to one shard
    - example: shard 1 handles zip codes from 01010-02515, shard 2 handles zip codes from 02516-04545, ...
  - hash-based sharding: hash the shard key to determine which shard the document belongs to

- Implications of the approach taken?

- MongoDB adjusts the shards over time.
  - to ensure an even distribution of docs across the shards

Recall: Data Model for the Movie Database

- Use three collections: movies, people, oscars

- Use references as follows:
  - in movie documents, include ids of the actors and directors
  - in oscar documents, include ids of the person and movie

- Whenever we refer to a person or movie, we also embed the associated entity's name.

- In addition, add two boolean fields to person documents:
  - hasActed, hasDirected
  - only include when true
Sample Movie Document

{ _id: "0499549",
  name: "Avatar",
  year: 2009,
  rating: "PG-13",
  runtime: 162,
  genre: "AVYS",
  earnings_rank: 1,
  actors: [ { id: "0000244",
             name: "Sigourney Weaver" },
            { id: "0002332",
             name: "Stephen Lang" },
            { id: "0735442",
             name: "Michelle Rodriguez" },
            { id: "0757855",
             name: "Zoe Saldana" },
            { id: "0941777",
             name: "Sam Worthington" } ],
  directors: [ { id: "0000116",
                name: "James Cameron" } ] }

Sample Person and Oscar Documents

{ _id: "0000059",
  name: "Laurence Olivier",
  dob: "1907-5-22",
  pob: "Dorking, Surrey, England, UK",
  hasActed: true,
  hasDirected: true
}

{ _id: ObjectId("528bf38ce6d3df97b49a0569"),
  year: 2013,
  type: "BEST-ACTOR",
  person: { id: "0000358",
            name: "Daniel Day-Lewis" },
  movie: { id: "0443272",
           name: "Lincoln" } }
Extra Practice Writing Queries

1) Find the names of all people in the database who acted in *Avatar*.
   - **SQL:**
     ```sql
     SELECT P.name
     FROM Person P, Actor A, Movie M
     WHERE P.id = A.actor_id
     AND M.id = A.movie_id
     AND M.name = 'Avatar';
     ```
   - **MongoDB:**

Extra Practice Writing Queries (cont.)

2) How many people in the database who were born in California have won as Oscar?
   - **SQL:**
     ```sql
     SELECT COUNT(DISTINCT P.id)
     FROM Person P, Oscar O
     WHERE P.id = O.person_id
     AND P.pob LIKE '%,California%';
     ```
   - Can't easily answer this question using our MongoDB version of the database. Why not?
3) How many actors in the database were born in California?
   - **SQL:**
     ```sql
     SELECT COUNT(DISTINCT P.id)
     FROM Person P, Actor A
     WHERE P.id = A.actor_id
     AND P.pob LIKE '%, California%';
     ```
   - **MongoDB:**

4) Which people in the database have directed a movie in which Tom Hanks has acted?
   - **SQL:**
     ```sql
     SELECT DISTINCT DirP.name
     FROM Person ActP, Actor A,
          Director D, Person DirP
     WHERE ActP.id = A.actor_id
     AND A.movie_id = D.movie_id
     AND D.director_id = DirP.id
     AND ActP.name = 'Tom Hanks';
     ```
   - **MongoDB:**
Extra Practice Writing Queries (cont.)

5) Find all years before 1970 that have 6 or more movies in the database.
   
   • SQL:

   ```sql
   SELECT year, COUNT(*)
   FROM Movie
   WHERE year < 1970
   GROUP BY year
   HAVING COUNT(*) >= 6;
   ```

   • MongoDB: