NoSQL Databases

Computer Science E-66
Harvard University
David G. Sullivan, Ph.D.

Recall: The Conventional Approach

• Use a DBMS that employs the relational model and SQL
• Typically follow a client-server model
  • the database server manages the data
  • applications act as clients
• Support \textit{transactions} and the associated guarantees
Relational vs. Object-Oriented

- As we've seen, OO databases provided the first serious threat to the conventional approach.

- Relational DBMSs survived the OO "threat".
  - incorporated some key OO features ➔ object-relational model
  - companies were reluctant to change legacy systems

- RDMSs also benefited from the need for applications to share data.

Two Approaches to Data Sharing

1. Use a *centralized database* to store all of the data.
   - applications use SQL to access the data

2. Use separate *application databases*.
   - each database is only directly accessed by one application
   - exchange data between applications through *web services*
     - use an API based on HTTP
     - transfer the data in text form using XML or similar model
The Rise of NoSQL

- The centralized approach to data sharing / app integration helped RDBMSs retain their dominance until the early 2000s.

- With the shift to application databases and web services, the importance of the relational model decreased.
  - SQL was no longer needed to access shared data
  - applications could change their own database schema as long as they continued to support the same API

- Thus, developers were now free to try alternative approaches.

The Rise of NoSQL (cont.)

- In addition, it became necessary for web-based applications to deal with massive amounts of:
  - data
  - traffic / queries

- Scalability is crucial.
  - load can increase rapidly and unpredictably

- Large servers are expensive and can only grow so large.

- Solution: use clusters of small commodity machines
  - use both fragmentation/sharding and replication
  - cheaper
  - greater overall reliability
  - can take advantage of cloud-based storage
The Rise of NoSQL (cont.)

- Problem: RDBMs do not scale well to large clusters.

- Google and Amazon each developed their own alternative approaches to data management on clusters.
  - Google: BigTable
  - Amazon: DynamoDB

- The papers that Google and Amazon published about their efforts got others interested in developing similar DBMSs.

  ➔ noSQL

What Does NoSQL Mean?

- Not well defined.

- Typical characteristics of NoSQL DBMSs:
  - don't use SQL / the relational model
  - open-source
  - designed for use on clusters
    - support for sharding/fragmentation and replication
    - schema-less or flexible schema

- One good overview:

Flavors of NoSQL

• Various taxonomies have been proposed

• Three of the main classes are:
  • key-value stores
  • document databases
  • column-family (aka big-table) stores

• Some people also include graph databases.
  • very different than the others
  • example: they are not designed for clusters

Key-Value Stores

• We’ve already worked with one of these: Berkeley DB

• There are many others: Riak, Redis, MemcacheDB, Amazon’s DynamoDB, Voldemort

• Simple data model: key/value pairs
  • the DBMS does not attempt to interpret the value

• Queries are limited to query by key.
  • get/put/update/delete a key/value pair
  • iterate over key/value pairs
Document Databases

• Examples include: MongoDB, CouchDB, Terrastore

• Also store key/value pairs

• Unlike key-value stores, the value is *not* opaque.
  • it is a *document* containing semistructured data
  • it can be examined and used by the DBMS

• Queries:
  • can be based on the key (as in key/value stores)
  • more often, are based on the contents of the document

• Here again, there is support for sharding and replication.
  • the sharding can be based on values within the document

Column-Family Databases

• Google's BigTable and systems based on it
  • HBase, Cassandra, Hypertable, Amazon SimpleDB, etc.

• To understand the motivation behind their design, consider one type of problem BigTable was designed to solve:
  • You want to store info about web pages.
  • For each URL, you want to store:
    • its contents
    • its language
    • for each other page that links to it, the *anchor text* associated with the link (i.e., the text that you click on)
Storing Web-Page Data in a Traditional Table

<table>
<thead>
<tr>
<th>page URL</th>
<th>language</th>
<th>contents</th>
<th>anchor text from <a href="http://www.cnn.com">www.cnn.com</a></th>
<th>anchor from <a href="http://www.bu.edu">www.bu.edu</a></th>
<th>one col per page</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.cnn.com">www.cnn.com</a></td>
<td>English</td>
<td>&lt;html&gt;…</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.bu.edu">www.bu.edu</a></td>
<td>English</td>
<td>&lt;html&gt;…</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.nytimes.com">www.nytimes.com</a></td>
<td>English</td>
<td>&lt;html&gt;…</td>
<td>&quot;news story&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="http://www.lemonde.fr">www.lemonde.fr</a></td>
<td>French</td>
<td>&lt;html&gt;…</td>
<td>&quot;French elections&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- One row per web page
- Single columns for its language and contents
- One column for the anchor text from each possible page, since in theory any page could link to any other page!
- Leads to a huge sparse table – most cells are empty/unused.

Storing Web-Page Data in BigTable

- Rather than defining all possible columns, define a set of column families that each row should have.
  - example: create a column family called anchor that replaces all of the separate anchor columns on the last slide
  - can also have column families that are like typical columns
- In a given row, only store columns with an actual value, representing them as (column key, value) pairs
  - column key = column family: qualifier
  - ex: ("anchor: www.bu.edu", "news story")
    
    column family --- qualifier
                           \--------- value
    ------------------- column key
Data Model for Column-Family Databases

• In addition to column keys:
  • row keys are used to index the rows
  • can also associate a timestamp with a given column value

• You thus have a multi-dimensional map:
  • (row key, column key, timestamp) → value
  • example:
    ("www.nytimes.com", "anchor:www.bu.edu", t1) → "news story"

• Different rows can have different schema.
  • i.e., different sets of column keys
  • (column key, value) pairs can be added or removed from a given row over time

• The set of column families in a given table rarely change.

Advantages of Column Families

• Gives an additional unit of data, beyond just a single row.

• Column families are used for access controls.
  • can restrict an application to only certain column families

• Column families can be divided up into locality groups that are stored together.
  • based on which column families are typically accessed together
  • advantage?
Picturing a Row In a Column-Family Database

source: Sadalage and Fowler

Aggregate Orientation

- Key-value, document, and column-family stores all lend themselves to an aggregate-oriented approach.
  - group together data that "belongs" together
    - i.e., that will tend to be accessed together

<table>
<thead>
<tr>
<th>type of database</th>
<th>unit of aggregation</th>
</tr>
</thead>
<tbody>
<tr>
<td>key-value store</td>
<td>the value part of the key/value pair</td>
</tr>
<tr>
<td>document database</td>
<td>a document</td>
</tr>
<tr>
<td>column-family store</td>
<td>a row (plus column-family sub-aggregates)</td>
</tr>
</tbody>
</table>

- Relational databases can't fully support aggregation.
  - focus on normalization and avoiding duplicated data
  - give each type of entity its own table, rather than grouping together entities/attributes that are accessed together
Aggregate Orientation (cont.)

• Example: data about customers
  • RDBMS: store a customer’s address in only one table
    • use foreign keys in other tables that refer to the address
  • aggregate-oriented system: store the full customer address in several places:
    • customer aggregates
    • order aggregates
    • etc.

• Benefits of an aggregate-based approach in a NoSQL store:
  • provides a unit for sharding across the cluster
  • allows us to get related data without needing to access many different nodes

Schemalessness

• NoSQL systems are completely or mostly schemaless.

• Key-value stores: put whatever you like in the value

• Document databases: no restrictions on the schema used by the semistructured data inside each document.
  • although some do allow a schema, as with XML

• Column-family databases:
  • specify the column families in a given table
  • no restrictions on the columns within a given column family
  • different rows can have different columns
Schemalessness (cont.)

- Advantages:
  - allows the types of data that are stored to evolve over time
  - makes it easier to handle nonuniform data
    - e.g., sparse tables
- Despite the fact that a schema is not required, programs that use the data need at least an *implicit* schema.
- Disadvantages of an implicit schema:
  - the DBMS can't enforce it
  - the DBMS can't use it to try to make accesses more efficient
  - different programs that access the same database can have conflicting notions of the schema

Example Document Database: MongoDB

- Mongo (from humongous)
- Key features include:
  - replication for high availability
  - auto-sharding for scalability
  - documents are expressed using JSON/BSON
  - queries can be based on the contents of the documents
- Related documents are grouped together into *collections*. 
JSON

• JSON is an alternative data model for semistructured data.
  • JavaScript Object Notation

• Built on two key structures:
  • an object, which is a sequence of fields (name:value pairs)
    ```json
    { id: "1000",
      name: "Sanders Theatre",
      capacity: 1000 }
    ```
  • an array of values
    ```json
    [ "123-456-7890", "222-222-2222", "333-333-3333"
    ```

• A value can be:
  • an atomic value: string, number, true, false, null
  • an object
  • an array

Example: JSON Object for a Person

```json
{   firstName: "John",
    lastName: "Smith",
    age: 25,
    address: { 
        streetAddress: "21 2nd Street",
        city: "New York",
        state: "NY",
        postalCode: "10021"
    },
    phoneNumbers: [
        {   type: "home",
            number: "212-555-1234"
        },
        {   type: "mobile",
            number: "646-555-4567"
        }
    ]
}
```
BSON

- MongoDB actually uses BSON.
  - a binary representation of JSON
  - BSON = marshalled JSON!

- BSON includes some additional types that are not part of JSON.
  - in particular, a type called ObjectID for unique id values.

The _id Field

- Every MongoDB document must have an _id field.
  - its value must be unique within the collection
  - acts as the primary key of the collection
  - it is the key in the key/value pair

- If you create a document without an _id field:
  - MongoDB adds the field for you
  - assigns it a unique BSON ObjectID
**MongoDB Terminology**

<table>
<thead>
<tr>
<th>relational term</th>
<th>MongoDB equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>database</td>
</tr>
<tr>
<td>table</td>
<td>collection</td>
</tr>
<tr>
<td>row</td>
<td>document</td>
</tr>
<tr>
<td>attributes</td>
<td>fields (name:value pairs)</td>
</tr>
<tr>
<td>primary key</td>
<td>the _id field, which is the key associated with the document</td>
</tr>
</tbody>
</table>

- Documents in a given collection typically have a similar purpose.
- However, no schema is enforced.
  - different documents in the same collection can have different fields

**Data Modeling in MongoDB**

- Need to determine how to map entities and relationships → collections of documents
- Could in theory give each type of entity:
  - its own (flexibly formatted) type of document
  - those documents would be stored in the same collection
- However, as mentioned in the earlier notes, it can make sense to group different types of entities together.
- Determining the aggregate boundaries == deciding how to represent relationships
Capturing Relationships in MongoDB

- Two options:
  1. store references to other documents using their \_id values

![Diagram of user document and contact document relationships]

(source: docs.mongodb.org/manual/core/data-model-design)

- similar to what constructs in the other models we've seen?

Capturing Relationships in MongoDB (cont.)

- Two options (cont.):
  2. embed documents within other documents

![Diagram of user document with embedded contact and access documents]

(source: docs.mongodb.org/manual/core/data-model-design)

- similar to what constructs in the other models we've seen?
Factors Relevant to Data Modeling

• A given MongoDB query can only access a single collection.
  • joins of documents are not supported
  • need to issue multiple requests
    → group together data that would otherwise need to be joined

• Atomicity is only provided for operations on a single document (and its embedded subdocuments).
  → group together data that needs to be updated as part of single logical operation (e.g., a balance transfer!)
  → group together data items A and B if A's current value affects whether/how you update B

Factors Relevant to Data Modeling (cont.)

• If an update makes a document bigger than the space allocated for it on disk, it may need to be relocated.
  • slows down the update, and can cause disk fragmentation
  • MongoDB adds padding to documents to reduce the need for relocation
    → use references if embedded documents could lead to significant growth in the size of the document over time
Factors Relevant to Data Modeling

- Pluses and minuses of embedding (a partial list):
  + need to make fewer requests for a given logical operation
  + less network/disk I/O
  + enables atomic updates
    - duplication of data
    - possibility for inconsistencies between different versions of duplicated data
    - can lead documents to become very large, and to document relocation

- Pluses and minuses of using references:
  • take the opposite of the pluses and minuses of the above!
  + allow you to capture trees/hierarchies/graphs

Data Model for the Movie Database

- Recall our movie database from PS 1.
  Person(id, name, dob, pob)
  Movie(id, name, year, rating, runtime, genre, earnings_rank)
  Oscar(movie_id, person_id, type, year)
  Actor(actor_id, movie_id)
  Director(director_id, movie_id)

- Three types of entities: movies, people, oscars

- Need to decide how we should capture the relationships
  • between movies and actors
  • between movies and directors
  • between Oscars and the associated people and movies
Data Model for the Movie Database (cont.)

• Assumptions about the relationships:
  • there are only one or two directors per movie
  • there are approx. five actors associated with each movie
  • the number of people associated with a given movie is fixed
  • each Oscar has exactly one associated movie
    and at most one associated person

• Assumptions about the queries:
  • Queries that involve both movies and people usually involve
    only the names of the people, not their other info.
    common: Who directed Avatar?
    common: Which movies did Tom Hanks act in?
    less common: Which movies have actors from Boston?
  • Queries that involve both Oscars and other entities usually
    involve only the name(s) of the person/movie.

Data Model for the Movie Database (cont.)

• Given our assumptions, we can take a hybrid approach
  that includes both references and embedding.

• 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.
  • allows us to satisfy common queries like Who acted in…?

• For less common queries that involve info. from multiple
  entities, use the references.
Data Model for the Movie Database (cont.)

- In addition, add two boolean fields to person documents:
  - hasActed, hasDirected
  - only include when true
  - allows us to find all actors/directors that meet criteria involving their pob/dob

- Note that most per-entity state appears only once, in the main document for that entity.

- The only duplication is of people/movie names and ids.

Sample Movie Document

```json
{  _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

```json
{  _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" }
}
```

Queries in MongoDB

- Each query can only access a single collection of documents.
- Use a method called `db.collection.find()`

```javascript
db.collection.find(<selection>, <projection>)
```

- `collection` is the name of the collection
- `<selection>` is an optional document that specifies one or more selection criteria
  - omitting it gets all documents in the collection
- `<projection>` is an optional document that specifies which fields should be returned
  - omitting it gets all fields in the document

- Example: find the names of all R-rated movies:
  ```javascript
db.movies.find({ rating: 'R' }, { name: 1 })
  ```
Comparison with SQL

- Example: find the names and runtimes of all R-rated movies that were released in the 2000.

- SQL:
  ```sql
  SELECT name, runtime
  FROM Movie
  WHERE rating = 'R' and year = 2000;
  ```

- MongoDB:
  ```javascript
  db.movies.find({ rating: "R", year: 2000 },
  { name: 1, runtime: 1 })
  ```

Query Selection Criteria

\[ \text{db.collection.find(}<\text{selection}>,<\text{projection}>) \]

- To find documents that match a set of field values, use a selection document consisting of those name/value pairs (see previous example).

- Operators for other types of comparisons:
  
<table>
<thead>
<tr>
<th>MongoDB</th>
<th>SQL equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>$gt, $gte</td>
<td>&gt;, &gt;=</td>
</tr>
<tr>
<td>$lt, $lte</td>
<td>&lt;, &lt;=</td>
</tr>
<tr>
<td>$ne</td>
<td>!=</td>
</tr>
</tbody>
</table>

- Example: find the names of movies with an earnings rank <= 200
  ```javascript
  db.movies.find({ earnings_rank: { $lte: 200 } })
  ```

- Note that the operator is the field name of a subdocument.
Query Selection Criteria (cont.)

• Logical operators: $and, $or, $not, $nor
  • take an array of selection subdocuments
  • example: find all movies rated R or PG-13:
    ```javascript
    db.movies.find({ $or: [ { rating: "R" }, { rating: "PG-13" } ] })
    ```
  • example: find all movies except those rated R or PG-13:
    ```javascript
    db.movies.find({ $nor: [ { rating: "R" }, { rating: "PG-13" } ] })
    ```

• To test for set-membership or lack thereof: $in, $nin
  • example: find all movies rated R or PG-13:
    ```javascript
    db.movies.find({ rating: { $in: ["R", "PG-13"] } })
    ```
  • example: find all movies except those rated R or PG-13:
    ```javascript
    db.movies.find({ rating: { $nin: ["R", "PG-13"] } })
    ```
  • note: $in/$nin is generally more efficient than $or/$nor

• To test for the presence/absence of a field: $exists
  • example: find all movies with an earnings rank:
    ```javascript
    db.movies.find({ earnings_rank: { $exists: true } })
    ```
  • example: find all movies without an earnings rank:
    ```javascript
    db.movies.find({ earnings_rank: { $exists: false } })
    ```
Logical AND

- You get an implicit logical AND by simply specifying a list of selection subdocuments.
  - recall our previous example:
    
    ```
    db.movies.find({ rating: "R", year: 2000 })
    ```
  
  - example: find all R-rated movies shorter than 90 minutes:
    
    ```
    db.movies.find({ rating: "R",
                     runtime: { $lt: 90 }
                   })
    ```

Logical AND (cont.)

- `$and` is needed if the subconditions involve the same field
  - can't have duplicate field names in a given document

- Example: find all Oscars given in the 1990s.
  - the following would **not** work:
    
    ```
    db.oscars.find({ year: { $gte: 1990 },
                     year: { $lte: 1999 }
                   })
    ```

  - one option that would work:
    
    ```
    db.oscars.find({ $and: [ { year: { $gte: 1990 } },
                              { year: { $lte: 1999 } } ]
                   })
    ```

  - another option: use an implicit AND on the operator subdocs:
    
    ```
    db.oscars.find({ year: { $gte: 1990, $lte: 1999 } })
    ```
Pattern Matching

- Use a regular expression surrounded with //</br>
  - with wildcards like the ones we used in XML schema (*, ?, +)</br>
  - example: find all people born in Boston</br>
    ```javascript
    db.people.find({ pob: /Boston,/ })
    ```</br>
  - Note: you essentially get a * wildcard by default on either end of the expression.</br>
    ```javascript
    /Boston,/ is the same as /*Boston,*/
    ```</br>
  - use: ^ to match the beginning of the value</br>
    $ to match the end of the value</br>
    ```javascript
    /^Boston,/ would match "South Boston, Mass"
    ```</br>
  - /^Boston,/ would not, because the ^ indicates "Boston" must be at the start of the value</br>
- Use the i flag for case-insensitive matches: /pg-13/i

Queries on Arrays/Subdocuments

- If a field has an array type</br>
  ```javascript
  db.collection.find( { arrayField: val } )
  ```</br>
  finds all documents in which val is at least one of the elements in the array associated with arrayField</br>
- Example: suppose that we stored a movie's genres as an array:</br>
  ```javascript
  ```</br>
  to find all animated movies – ones with a genre of "N":</br>
  ```javascript
  db.movies.find( { genre: "N" } )
  ```</br>
- Given that we actually store the genres as a single string (e.g., "NCF"), how would we find animated movies?
Queries on Arrays/Subdocuments (cont.)

- Use dot notation to access fields within a subdocument, or within an array of subdocuments:
  - example: find all Oscars won by the movie *Gladiator*:

```javascript
> db.oscars.find( { "movie.name": "Gladiator" } )
```

```
{ 
    _id: <ObjectID1>, 
    year: 2001, 
    type: "BEST-PICTURE", 
    movie: { id: "0172495", 
              name: "Gladiator" } 
} 
{ 
    _id: <ObjectID2>, 
    year: 2001, 
    type: "BEST-ACTOR", 
    movie: { id: "0172495", 
              name: "Gladiator" }, 
    person: { id: "0000128", 
              name: "Russell Crowe" }
}
```

- **Note**: When using dot notation, the field name must be surrounded by quotes.

Queries on Arrays/Subdocuments (cont.)

- example: find all movies in which Tom Hanks has acted:

```javascript
> db.movies.find( { "actors.name": "Tom Hanks" } )
```

```
{ 
    _id: "0107818", 
    name: "Philadelphia", 
    year: 1993, 
    rating: "PG-13", 
    runtime: 125, 
    genre: "D"
    actors: [ 
              { id: "0000158", 
                name: "Tom Hanks" },
              { id: "0000243", 
                name: "Denzel Washington" },
              ... 
            ],
    directors: [ 
                  { id: "0001129", 
                    name: "Jonathan Demme" } 
                ]
} 
{ 
    _id: "0109830", 
    name: "Forrest Gump", 
    year: 1994, 
    rating: "PG-13", 
    runtime: 142, 
    genre: "CD"
    actors: [ 
              { id: "0000158", 
                name: "Tom Hanks" },
              ... 
            ]
} 
```
Projections

\[ \text{db.collection.find(<selection>, <projection>)} \]

- The projection document is a list of \textit{fieldname:value} pairs:
  - a value of 1 indicates the field should be included
  - a value of 0 indicates the field should be excluded

- Recall our previous example:
  \[
  \text{db.movies.find(\{\ rating: "R", year: 2000 \}, \{ name: 1, runtime: 1 \})}
  \]

- Example: find all info. about R-rated movies except their genres:
  \[
  \text{db.movies.find(\{\ rating: "R" \}, \{ genre: 0 \})}
  \]

Projections (cont.)

- The _id field is returned unless you explicitly exclude it.
  \[
  > \text{db.movies.find(\{\ rating: "R", year: 2011 \}, \{ name: 1 \})}
  \]
  \[
  \{\ _id\ : \ "1411697", \ name\ : \ "The\ Hangover\ Part\ II" \}
  \{\ _id\ : \ "1478338", \ name\ : \ "Bridesmaids" \}
  \{\ _id\ : \ "1532503", \ name\ : \ "Beginners" \}
  \]

  \[
  > \text{db.movies.find(\{\ rating: "R", year: 2011 \}, \{ name: 1, _id: 0 \})}
  \]
  \[
  \{\ name\ : \ "The\ Hangover\ Part\ II" \}
  \{\ name\ : \ "Bridesmaids" \}
  \{\ name\ : \ "Beginners" \}
  \]

- A given projection should either have:
  - all values of 1: specifying the fields to include
  - all values of 0: specifying the fields to exclude
  - one exception: specify fields to include, and exclude _id
Iterating Over the Results of a Query

- `db.collection.find()` returns a cursor that can be used to iterate over the results of a query.

- In the MongoDB shell, if you don't assign the cursor to a variable, it will automatically be used to print up to 20 results.
  - if more than 20, use the command `it` to continue the iteration.

- Another way to view all of the result documents:
  - assign the cursor to a variable:
    ```javascript
    var cursor = db.movies.find({ year: 2000 })
    ```
  - use the following method call to print each result document in JSON:
    ```javascript
    cursor.forEach(printjson)
    ```

Aggregation

- Recall the aggregate operators in SQL: `AVG()`, `SUM()`, etc.

- More generally, aggregation involves computing a result from a collection of data.

- MongoDB supports several approaches to aggregation:
  - single-purpose aggregation methods
  - an aggregation pipeline
  - map-reduce
Single-Purpose Aggregation Methods

- `db.collection.count(<selection>)`
  - returns the number of documents in the collection that satisfy the specified selection document
  - ex: how many R-rated movies are shorter than 90 minutes?
    ```
    db.movies.count({ rating: "R", runtime: { $lt: 90 } })
    ```

- `db.collection.distinct(<field>, <selection>)`
  - returns an array with the distinct values of the specified field in documents that satisfy the specified selection document
  - if omit the selection, get all distinct values of that field
  - ex: which actors have been in one or more of the top 10 grossing movies?
    ```
    db.movies.distinct("actors.name", { earnings_rank: { $lte: 10 } })
    ```

Aggregation Pipeline

- A more general-purpose and flexible approach to aggregation is to use a `pipeline` of aggregation operations.

- Each stage of the pipeline:
  - takes a set of documents as input
  - applies a `pipeline operator` to those documents, which transforms / filters / aggregates them in some way
  - produces a new set of documents as output

```
  db.collection.aggregate({
    <pipeline-op1>: <pipeline-expression1> },
    <pipeline-op2>: <pipeline-expression2> },
    ...
    <pipeline-opN>: <pipeline-expressionN> })
```
**Aggregation Pipeline Example**

```javascript
db.orders.aggregate(
  { $match: { status: "A" } },
  { $group: { _id: "$cust_id", total: { $sum: "$amount"} } }
)
```

source: docs.mongodb.org/manual/core/aggregation-pipeline

---

**Pipeline Operators**

- **$project** — include, exclude, rename, or create fields

  - Example of a single-stage pipeline using $project:
    ```javascript
db.people.aggregate(
  { $project: { 
    name: 1,
    whereBorn: "$pob",
    yearBorn: { $substr: ["$dob", 0, 4] } 
  } 
})
```

  - for each document in the people collection, extracts:
    - `name` (1 = include, as in earlier projection documents)
    - `pob`, which is renamed `whereBorn`
    - a new field called `yearBorn`, which is derived from the existing `pob` values (yyyy-m-d → yyyy)

  - **note**: use $ before a field name to obtain its value
Pipeline Operators (cont.)

- **$group** – like GROUP BY in SQL
  
  $group: { _id: <field or fields to group by>,
  <computed-field-1>,
  ..., <computed-field-N> }
  
  - example: compute the number of movies with each rating
    
    ```
    db.movies.aggregate(
      { $group: { _id: "$rating",
                 numMovies: { $sum: 1 } }
    })
    ```
  
  - `{ $sum: 1 }` is equivalent to COUNT(*) in SQL
    
    - for each document in a given subgroup, adds 1 to that subgroup’s value of the computed field
    
  - can also sum values of a specific field (see earlier slide)
  
  - **$sum** is one example of an **accumulator**
  
  - others include: $min, $max, $avg, $addToSet

Pipeline Operators (cont.)

- **$match** – selects documents according to some criteria
  
  $match: <selection>
  
  where <selection> has identical syntax to the selection documents used by `db.collection.find()`

- **$unwind** – like the unnest operator in object-relational SQL
  
  - see the next slide for an example

- others include: $limit, $sort, $geoNear

- See the MongoDB manual for more detail:
  
  docs.mongodb.org/manual/reference/operator/aggregation
Example of a Three-Stage Pipeline

db.movies.aggregate(
    { $match: { year: 2013 }},
    { $project: { _id: 0,
        movie: "$name",
        actor: "$actors.name" } },
    { $unwind: "$actor" }
)

• What does each stage do?

Map-Reduce

• MongoDB includes support for map-reduce.
  • also defines a pipeline, but it's more flexible
    • example: allows for user-defined pipeline functions
  • in MongoDB, this can be less efficient than the pipeline operators for many types of aggregation
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

- A replica set is a group of MongoDB server processes that host the same set of documents.
Replication and Reads

- 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

- 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

Sharding in MongoDB

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

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