Recall: NoSQL Characteristics

- Typical characteristics include:
  - 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
Recall: Flavors of NoSQL

- Four main types:
  - key-value stores
  - document databases
  - column-family (aka big-table) stores
  - graph databases

- In these notes, we will focus on:
  - document databases
  - MongoDB in particular

Recall: Document Databases

- Like key-value stores, document databases 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

- Flexible schema:
  - no restrictions on what a document looks like
  - schema can evolve over time

- Queries:
  - can be based on the key (as in key-value stores)
  - more often, are based on the contents of the document
Example Document Database: MongoDB

- Mongo (from humongous)

- Key features include:
  - JSON-style documents
    - actually uses BSON (JSON's binary format)
  - replication for high availability
  - auto-sharding for scalability
  - document-based queries
  - can create an index on any attribute
    - for faster reads

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 (field-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
Recall: 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 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**

- BSON is a binary representation of JSON
  - it is marshalled JSON!

- It includes some additional types that are not part of JSON.

- In particular, it adds a type called ObjectID for unique id values.
  - consists of 12 bytes:
    - 4 bytes: seconds since the Unix epoch (1/1/1970, 12 am UCT)
    - 3 bytes: machine identifier
    - 2 bytes: process id
    - 3 bytes: a counter, starting with a random value

- BSON balances two goals:
  - efficient use of space
  - *traversability* – i.e., efficient searching / unmarshalling

---

**Marshalling JSON into BSON**

- Precede each document with its length. Terminate it with a null character ("\0" = one byte of all 0s).

- Precede each name/value pair with a one-byte code for the value’s type.

- Precede variable-length values (e.g., strings) with their lengths.
  - but not field names, which are just null-terminated. why?

```json
{"rating": "PG-13"}
```

- How does this differ from the approach we took to marshalling? Why does it differ?
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
  - example from the MongoDB shell:

```
> db.test.save({ rating: "PG-13" })
> db.test.find()
{ "_id" : ObjectId("528bf38ce6d3df97b49a0569"),
 "rating" : "PG-13" }
```

- Note: quoting field names is optional (see rating above)

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.
  - create an aggregate containing data that tends to be accessed 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 showing user document, contact document, and access 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

```json
{
    _id: <ObjectId1>,
    username: "123xyz",
    contact: {
        phone: "123-456-7890",
        email: "xyz@example.com"
    },
    access: {
        level: 5,
        group: "dev"
    }
}
```

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 University Database

• Recall our university database:
  Student(id, name)
  Department(name, office)
  Room(id, name, capacity)
  Course(name, start_time, end_time, room)
  MajorsIn(student, dept)
  Enrolled(student, course, credit_status)

• Need to decide how we should capture the relationships
  • between courses and rooms
  • between students and courses
  • between students and departments

• Where might it make sense to use embedding?
• Where might it make sense to use references?
Data Model for the Movie Database

- Recall our movie database from PS 2.
  - 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

{ _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" } }
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

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

- 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 \( \leq 200 \)
  \[ \text{db.movies.find({ earnings_rank: { $lte: 200 } })} \]

- 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:
    \[ \text{db.movies.find({ $or: [ { rating: "R" }, { rating: "PG-13" } ] })} \]

  - example: find all movies except those rated R or PG-13:
    \[ \text{db.movies.find({ $nor: [ { rating: "R" }, { rating: "PG-13" } ] })} \]
Query Selection Criteria (cont.)

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

- To test for the presence/absence of a field: \texttt{exists}
  - example: find all movies with an earnings rank:
    \begin{verbatim}
    db.movies.find({ earnings_rank: { $exists: true } })
    \end{verbatim}
  - example: find all movies \textit{without} an earnings rank:
    \begin{verbatim}
    db.movies.find({ earnings_rank: { $exists: false } })
    \end{verbatim}

Logical AND

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

- $\textit{and}$ 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 \textit{not} work:
    ```javascript
    db.oscars.find({
      year: { $gte: 1990 },
      year: { $lte: 1999 }
    })
    ```
  - one option that would work:
    ```javascript
    db.oscars.find({
      $and: [{
        year: { $gte: 1990 }
      },
      { year: { $lte: 1999 } }
    }]
    ```
  - another option: use an implicit AND on the operator subdocs:
    ```javascript
    db.oscars.find({
      year: { $gte: 1990, $lte: 1999 }
    })
    ```

Pattern Matching

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

• If a field has an array type
  
  \[
  \text{db.collection.find( } \{ \text{arrayField: val } \} \text{ )}
  \]
  
  finds all documents in which `val` is at least one of the elements in the array associated with `arrayField`

• Example: suppose that we stored a movie's genres as an array:
  
  \[
  \{ \_id: "0317219", \text{name: "Cars"}, \text{year: 2006,}
  \text{rating: "G", runtime: 124, earnings_rank: 80,}
  \text{genre: ["N", "C", "F"], ...} \}
  \]

  • to find all animated movies – ones with a genre of "N":
    
    \[
    \text{db.movies.find( } \{ \text{genre: "N"} \} \text{ )}
    \]

  • 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`:
    
    \[
    \text{> db.oscars.find( } \{ \text{movie.name}: "Gladiator" \} \text{ )}
    \]
    
    \[
    \{ \_id: <ObjectID1>, \text{year: 2001,}
    \text{type: "BEST-PICTURE",}
    \text{movie: \{ id: "0172495",}
    \text{ name: "Gladiator" \}}
    \}
    \{ \_id: <ObjectID2>, \text{year: 2001,}
    \text{type: "BEST-ACTOR",}
    \text{movie: \{ id: "0172495",}
    \text{ name: "Gladiator" },
    \text{person: \{ id: "0000128",}
    \text{ name: "Russell Crowe" \}}
    \}
    \]

• **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:
  ```
  > 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

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

- The projection document is a list of `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:
  ```
  db.movies.find({ rating: "R", year: 2000 }, { name: 1, runtime: 1 })
  ```

- Example: find all info. about R-rated movies except their genres:
  ```
  db.movies.find({ rating: "R" }, { genre: 0 })
  ```
Projections (cont.)

• The _id field is returned unless you explicitly exclude it.

```javascript
> db.movies.find({ rating: "R", year: 2011 },
                 { name: 1 })
{ "_id" : "1411697", "name" : "The Hangover Part II" }
{ "_id" : "1478338", "name" : "Bridesmaids" }
{ "_id" : "1532503", "name" : "Beginners" }
```

```javascript
> 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 query, 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

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

Aggregation Pipeline Example

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:
    
    ```
    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
  
  \[
  \text{$match: <selection>}
  \]
  
  where \(<selection>\) has identical syntax to the selection documents used by \(\text{db.collection.read()}\)

- **$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.mongo-db.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

- Map-reduce is a widely known framework for performing computations on a large collection of data spread across multiple machines.
  - spreads the computation across the machines, letting them work in parallel
  - also tries to minimize the amount of data that needs to be transferred across the network.
- Became well-known through Google's MapReduce system.
- An open-source version is part of the Hadoop project.
- Some databases – including MongoDB – have their own implementations.

Map-Reduce (cont.)

- Example problem: a company that sells widgets
  - data = a collection of customer-order information that is sharded across multiple machines
  - sample order document:
    ```json
    {
      cust_id: "A123",
      amount: 500,
      status: "A"
    }
    ```
  - desired aggregation – compute, for each customer, the total amount in that customer's active orders
  - Inefficient approach: ship all of the order info to one node and compute the totals there.
Map-Reduce (cont.)

- Map-reduce approach:
  - *map:* each node processes the orders that are stored locally
    - map each order to a (cust_id, amount) pair
  - *reduce:* take multiple (cust_id, amount) pairs with the same cust_id and reduce them to a single (cust_id, total)
    - The reduce step could first be done on each shard, and then the per-shard subtotals could be combined.
  - More generally, map-reduce uses two functions:
    - a *map* function that maps each document into some number of (key, value) pairs
    - a *reduce* function that aggregates (key, value) pairs with the same key
    - Note that it also effectively defines a type of pipeline.

Map-Reduce in MongoDB

- Use the `db.collection.mapReduce()` function
  - takes the map and reduce functions as arguments

```
Collection

db.orders.mapReduce(
    reduce: function() {
        emit(this.cust_id, this.amount);
    },
    function(key, values) {
        return Array.sum(values);
    },
    query: { status: "Y" },
    out: "order_totals"
)
```

source: docs.mongodb.org/manual/core/map-reduce
Comparing the Aggregation Approaches

- from the MongoDB manual:
  
  *For most aggregation operations, the Aggregation Pipeline provides better performance and a more coherent interface. However, map-reduce operations provide some flexibility that is not presently available in the aggregation pipeline.*

- Like map-reduce, the aggregation pipeline can process sharded collections.

- Only map-reduce supports outputting the results to a collection, and combining the current results with earlier ones.
  - allows for incremental aggregation

- The aggregation pipeline is less flexible.
  - limited to its pipeline operators
  - vs. map-reduce's user-defined map and reduce functions
  - good enough for our purposes