CS 4604: Introduction to Database Management Systems

NoSQL databases

Virginia Tech CS 4604 Sprint 2021
Instructor: Yinlin Chen
Today’s Topics

• NoSQL
  – Key-Value database
  – Document database
Two Classes of Relational Database App

• OLTP (Online Transaction Processing)
  – Queries are simple lookups: 0 or 1 join
    • E.g., find customer by ID and their orders
  – Many updates. E.g., insert order, update payment
  – Consistency is critical: we need transactions

• OLAP (Online Analytical Processing)
  – aka “Decision Support”
  – Queries have many joins, and group-by’s
    • E.g., sum revenues by store, product, clerk, date
  – No updates
NoSQL Motivation

• Originally motivated by Web 2.0 applications
  – E.g., Facebook, Amazon, Instagram, etc.
  – Startups need to scale up from 10 to $10^7$ clients quickly
• Needed: very large scale OLTP workloads
• Give up on consistency, give up OLAP
• NoSQL: reduce functionality
  – Simpler data model
  – Very restricted updates
Replicating the Database

• Scale up through **partitioning** – “sharding”
  – Partition the database across many machines in a cluster
  – Database now fits in main memory
  – Queries spread across these machines
  – Can increase throughput
  – Easy for writes but reads become expensive!

• Scale up through **replication**
  – Create multiple copies of each database partition
  – Spread queries across these replicas
  – Can increase throughput and lower latency
  – Can also improve fault-tolerance
  – Easy for reads but writes become expensive!

• **Consistency** is much harder to enforce
Relational Model $\rightarrow$ NoSQL

- Relational DB: difficult to replicate/partition
  - Partition: we maybe forced to join across servers
  - Replication: local copy has inconsistent versions
  - Consistency is hard in both cases
- NoSQL: simplified data model
  - Given up on functionality
  - Application must now handle joins and consistency
Relational Model vs NoSQL

• Relational DB (ACID)
  – Atomicity
  – Consistency
  – Isolation
  – Durability

• NoSQL (BASE)
  – Basic Availability
    • Application must handle partial failures itself
  – Soft State
    • DB state can change even without inputs
  – Eventually Consistency
    • DB will “eventually” become consistent
What’s NoSQL?

- The misleading term “NoSQL” is short for “Not Only SQL”.
- Non-relational, schema-free, non-(quite)-ACID – More on ACID transactions later in class
- Horizontally scalable, distributed, easy replication support
- Simple API
## NoSQLs

<table>
<thead>
<tr>
<th>Key value</th>
<th>Document</th>
<th>In-memory</th>
<th>Graph</th>
<th>Search</th>
<th>Time series</th>
<th>Ledger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-latency, key lookups with high throughput and fast ingestion of data</td>
<td>Indexing and storing documents with support for query on any attribute</td>
<td>Microseconds latency, key-based queries, and specialized data structures</td>
<td>Creating and navigating data relations easily and quickly</td>
<td>Indexing and searching semistructured logs and data</td>
<td>Collect, store, and process data sequenced by time</td>
<td>Complete, immutable, and verifiable history of all changes to application data</td>
</tr>
</tbody>
</table>

| Real-time bidding, shopping cart, social | Content management, personalization, mobile | Leaderboards, real-time analytics, caching | Fraud detection, social networking, recommendation engine | Product catalog, help, and FAQs, full text | IoT applications, event tracking | Systems of record, supply chain, healthcare, registrations, financial |
Key-value (K-V) Stores

- **Data model**: (key, value) pairs
  - Key = string/integer, unique for the entire data
  - Value = can be anything (very complex object)

- **Operations**
  - get(key), put(key, value)
  - Operations on value not supported

- **Distribution / Partitioning** – w/ hash function
  - No replication: key k is stored at server h(k)
  - Multi way-way replication: e.g., key k stored at h1(k), h2(k), h3(k)

- Amazon DynamoDB, Voldemort, Memcached, …
Key-Value Stores Internals

• Partitioning:
  – Use a hash function $h$
  – Store every (key, value) pair on server $h(\text{key})$

• Replication:
  – Store each key on (say) three servers
  – On update, propagate change to the other servers; eventual consistency
  – Issue: when an app reads one replica, it may be stale

• Usually: combine partitioning + replication
Amazon DynamoDB Demo
Document Database

- Designed to store and query data as JSON-like documents
- Flexible schema and indexing, powerful ad hoc queries, and analytics over collections of documents
- Enable developers to store and query data in a database by using the same document-model format they use in their application code
- The document model works well with use cases like
  - Catalogs, user profiles, and content management systems where each document is unique and evolves over time
Document Database

- MongoDB
- Amazon DocumentDB (with MongoDB compatibility)
- SimpleDB
- CouchDB
- ...

...
Motivation

• In Key-Value stores, the Value is often a very complex object
  – Key = ‘2010/7/1’, Value = [all flights that date]
• Better: value to be structured data
  – JSON or XML or Protobuf
  – Called a “document” but it’s just data
**MongoDB Data Model**

<table>
<thead>
<tr>
<th><strong>MongoDB</strong></th>
<th><strong>DBMS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>Database</td>
</tr>
<tr>
<td>Collection</td>
<td>Relation</td>
</tr>
<tr>
<td>Document</td>
<td>Row/Record</td>
</tr>
<tr>
<td>Field</td>
<td>Column</td>
</tr>
</tbody>
</table>

Document = {..., field: value, ...}

Where value can be:
- Atomic
- A document
- An array of atomic values
- An array of documents

\{qty:1, status:"D", size:{h:14,w:21}, tags: ["a", "b"]\},

- JSON data model
- Internally stored as BSON = Binary JSON
- Client libraries can directly operate on this natively
MongoDB Data Model

• Can use JSON schema validation
  – Some integrity checks, field typing and ensuring the presence of certain fields

• Special field in each document: _id
  – Primary key
  – Will also be indexed by default
  – If it is not present during ingest, it will be added
  – Will be first attribute of each doc.
  – This field requires special treatment during projections
plot: "After being held captive in an Afghan cave, an industrialist creates a...

genres: Array
runtime: 126
metacritic: 79
rated: "PG-13"

cast: Array
  0: "Robert Downey Jr."
  1: "Terrence Howard"
  2: "Jeff Bridges"
  3: "Gwyneth Paltrow"

poster: "https://m.media-amazon.com/images/M/MV5BMTczNTI2ODUwOF5BMl5BanBnXkFtZT...
title: "Iron Man"

fullplot: "Tony Stark. Genius, billionaire, playboy, philanthropist. Son of legen...

languages: Array
released: 2008-05-02T00:00:00.000+00:00

directors: Array

writers: Array

awards: Object
  wins: 20
  nominations: 51
  text: "Nominated for 2 Oscars. Another 18 wins & 51 nominations."

lastupdated: "2015-08-23 00:04:50"

year: 2008

imdb: Object
  rating: 7.9
  votes: 615059
  id: 371746

countries: Array

type: "movie"
JSON - Overview

- JavaScript Object Notation = lightweight text-based open standard designed for human-readable data interchange. Interfaces in C, C++, Java, Python, Perl, etc.
- The filename extension is .json
- Semi structured data
  - Does not have the same level of organization and predictability of structured data
  - The data does not reside in fixed fields or records, but does contain elements that can separate the data into various hierarchies
Nested JSON Object Example

```json
{
    "name":"John",
    "age":30,
    "cars": {
        "car1":"Ford",
        "car2":"BMW",
        "car3":"Fiat"
    }
}
```
JSON vs Relational

- Relational data model
  - Rigid flat structure (tables)
  - Schema must be **fixed** in advanced
  - Binary representation: good for performance, bad for exchange
  - Query language based on **Relational Algebra**
- Semi-structured data model / JSON
  - Flexible, nested structure (trees)
  - Does not require predefined schema ("self-describing")
  - Text representation: good for exchange, bad for performance
  - Most common use: Language API; query languages emerging
JSON Types

• Primitive: number, string, Boolean, null
• Object: collection of name-value pairs:
  – {“name1”: value1, “name2”: value2, ...}
  – “name” is also called a “key”
• Array: ordered list of values:
  – [obj1, obj2, obj3, ...]
JSON Syntax

https://www.json.org/
Avoid Using Duplicate Keys

• The standard allows them, but many implementations don’t

```
{"id":"07",
 "title": "Databases",
 "author": "Garcia",
 "author": "Ullman",
 "author": "Widom"
}
```

```
{"id":"07",
 "title": "Databases",
 "author": ["Garcia",
             "Ullman",
             "Widom"]
}
```
JSON Semantics: Tree presentation

```json
{
    "person": [
        {
            "name": "Mary",
            "address": {
                "street": "Maple",
                "no": 345,
                "city": "SF"
            }
        },
        {
            "name": "John",
            "address": {
                "street": "Thailand",
                "phone": 2345678
            }
        }
    ]
}
```
Intro to Semi-structured Data

- JSON is self-describing
- Schema elements become part of the data
  - Relational schema: person(name, phone)
  - In JSON “person”, “name”, “phone” are part of the data, and are repeated many times
- JSON is more flexible
  - Schema can change per tuple
Storing JSON in RDBMS

• Using JSON as a data type provided by RDBMSs
  – Declare a column that contains either json or jsonb (binary)
  – CREATE TABLE people (person json) [or jsonb for binary]

• Some databases support
  – E.g. MySQL:
    • SELECT * FROM students
      WHERE JSON_EXTRACT(student, '$.age') = 12;

• Translate JSON documents into relations
Mapping Relational Data to JSON

<table>
<thead>
<tr>
<th>Person</th>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>3634</td>
<td></td>
</tr>
<tr>
<td>Sue</td>
<td>6343</td>
<td></td>
</tr>
<tr>
<td>Dirk</td>
<td>6363</td>
<td></td>
</tr>
</tbody>
</table>

```json
{
  "person": [
    {
      "name": "John",
      "phone": 3634
    },
    {
      "name": "Sue",
      "phone": 6343
    },
    {
      "name": "Dirk",
      "phone": 6363
    }
  ]
}
```
Mapping Relational Data to JSON

May inline multiple relations based on foreign keys

<table>
<thead>
<tr>
<th>Person</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>phone</td>
</tr>
<tr>
<td>John</td>
<td>3634</td>
</tr>
<tr>
<td>Sue</td>
<td>6343</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Orders</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>personName</td>
<td>date</td>
</tr>
<tr>
<td>John</td>
<td>2002</td>
</tr>
<tr>
<td>John</td>
<td>2004</td>
</tr>
<tr>
<td>Sue</td>
<td>2002</td>
</tr>
</tbody>
</table>

```json
{
"Person": [
{"name": "John",
 "phone": 3646,
 "Orders": [
 {"date": 2002, "product": "Gizmo"},
 {"date": 2004, "product": "Gadget"}
 ]
},
{"name": "Sue",
 "phone": 6343,
 "Orders": [
 {"date": 2002, "product": "Gadget"}
 ]
}
}
```
Mapping Relational Data to JSON

Many-many relationships are more difficult to represent

<table>
<thead>
<tr>
<th>Person</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>phone</td>
<td></td>
</tr>
<tr>
<td>John</td>
<td>3634</td>
<td></td>
</tr>
<tr>
<td>Sue</td>
<td>6343</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>prodName</td>
<td>price</td>
<td></td>
</tr>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td></td>
</tr>
<tr>
<td>Phone</td>
<td>29.99</td>
<td></td>
</tr>
<tr>
<td>Gadget</td>
<td>9.99</td>
<td></td>
</tr>
</tbody>
</table>

Orders

<table>
<thead>
<tr>
<th>personName</th>
<th>date</th>
<th>product</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>2002</td>
<td>Gizmo</td>
</tr>
<tr>
<td>John</td>
<td>2004</td>
<td>Gadget</td>
</tr>
<tr>
<td>Sue</td>
<td>2002</td>
<td>Gadget</td>
</tr>
</tbody>
</table>

Options for the JSON file:
- 3 flat relations: Person, Orders, Product
- Person→Orders→Products products are duplicated
- Product→Orders→Person persons are duplicated
Mapping Semi-structured Data to Relations

- Missing attributes:

```
{"person": [
{"name":"John", "phone":1234},
{"name":"Joe"}]
}
```

- Could represent in a table with nulls

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>1234</td>
</tr>
<tr>
<td>Joe</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Mapping Semi-structured Data to Relations

- Repeated attributes

```
{"person":
  [{"name":"John", "phone":1234},
   {"name":"Mary", "phone":[1234,5678]}]
}
```

- Impossible in one table:

<table>
<thead>
<tr>
<th></th>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mary</td>
<td>2345</td>
</tr>
<tr>
<td>2</td>
<td>Mary</td>
<td>3456</td>
</tr>
</tbody>
</table>

Two phones!
Mapping Semi-structured Data to Relations

- Attributes with different types in different objects
  ```
  {
  "person":
  [{
   "name": "Sue",
   "phone": 3456,
  },
  {
   "name": {
   "first": "John",
   "last": "Smith"},
   "phone": 2345
  }
  }
  ```

- Nested collections
- Heterogeneous collections

- These are difficult to represent in the relational model
Why Semi-Structured Data?

• Semi-structured data works well as *data exchange formats*
  – i.e., exchanging data between different apps
  – Examples: XML, JSON, Protobuf (protocol buffers)

• Systems use them as a data model for DBs:
  – SQL Server supports for XML-valued relations
  – CouchBase, MongoDB, Snowflake: JSON
  – Dremel (BigQuery): Protobuf
Query Languages for Semi-Structured Data

• XML: XPath, XQuery (see textbook Ch 27)
  – Supported inside many RDBMS (SQL Server, DB2, Oracle)
  – Several standalone XPath/XQuery engines

• Protobuf: SQL-ish language (Dremel) used internally by google, and externally in BigQuery

• JSON:
  – CouchBase: N1QL
  – AsterixDB: SQL++ (based on SQL)
  – MongoDB: has a pattern-based language
  – JSONIQ: http://www.jsoniq.org/
MongoDB Query Language (MQL)

• Input = collections, output = collections
• Three main types of queries in the query language
  – Retrieval: Restricted SELECT-WHERE-ORDER BY-LIMIT type queries
  – Aggregation: A bit of a misnomer; a general pipeline of operators
    • Can capture Retrieval as a special case
    • But worth understanding Retrieval queries first...
  – Updates
• All queries are invoked as
  – db.collection.operation1(...).operation2(...)
    • collection: name of collection
  – Unlike SQL which lists many tables in a FROM clause, MQL is centered around manipulating a single collection
Some MQL Principles : Dot (.) Notation

- "." is used to drill deeper into nested docs/arrays
- Recall that a value could be atomic, a nested document, an array of atomics, or an array of
  - nested documents
- Examples:
  - "instock.qty" → qty field within the instock field
    - Applies only when instock is a nested doc or an array of nested docs
      - If instock is a nested doc, then qty could be a nested field
      - If instock is an array of nested docs, then qty could be a nested field within documents
        in the array
  - "instock.1" → second element within the instock array
    - Element could be an atomic value or a nested document
  - "instock.1.qty" → qty field within the second document within the instock array
- Note: such dot expressions need to be in quotes
Some MQL Principles: Dollar ($) Notation

- $ indicates that the string is a special keyword
  - E.g., $gt, $lte, $add, $elemMatch, ...
- Used as the "field" part of a "field: value" expression
- So if it is a binary operator, it is usually done as:
  - \{ LOperand : \{ $keyword : ROperand \}\}
  - e.g., \{qty : \{$gt : 30\}\}
- Alternative: arrays
  - \{ $keyword : [argument list] \}
  - e.g., \{ $add : [1, 2] \}
- Exception: $fieldName, used to refer to a previously defined field on the value side
  - Purpose: disambiguation
  - Only relevant for aggregation pipelines
Retrieval Queries Template

- `db.collection.find(<predicate>, optional <projection>)`
  returns documents that match `<predicate>`
  keep fields as specified in `<projection>`
  both `<predicate>` and `<projection>` expressed as documents in fact, most things are documents!

- `db.inventory.find( {} )`
  returns all documents

- `db.collection.find(<predicate>, optional <projection>)`
  - `find( {title: "Iron Man"} )`
    • all documents with title is “Iron Man”
  - `find({qty:{gte:50}})`
    • all documents with qty >= 50
  - `find({beds : 6, qty:{gte:50}})`
    • all documents that satisfy both
  - `find({$or:[{beds : 6},{qty:{lt:30}}]})`
    • all documents that satisfy either
Retrieval Queries: Nested Documents

- `db.collection.find(<predicate>, optional <projection>)`
  - `find({size:{h:14,w:21,uom:"cm"}})`
    - exact match of nested document, including ordering of fields
  - `find ( { "size.uom" : "cm", "size.h" : {$gt : 14 } } )`
    - querying a nested field
    - Note: when using . notation for sub-fields, expression must be in quotes
    - Also note: binary operator handled via a nested document
Retrieval Queries: Arrays

- Slightly different example dataset for Arrays and Arrays of Document Examples:
  ```javascript
  db.collection.find(<predicate>, optional <projection>)
  ```
- `find({tags: ['red', 'blank']})`
  - Exact match of array
- `find({tags: 'red'})`
  - If one of the elements matches "red"
- `find({tags: 'red', tags: 'plain'})`
  - If one matches "red", one matches "plain"
- `find({dim: {$gt: 15, $lt: 20}})`
  - If one element is >15 and another is <20
- `find({dim: {$elemMatch: { $gt: 15, $lt: 20 } }})`
  - If a single element is >15 and <20
- `find({"dim.1":{$gt:25}})`
  - If second item > 25
  - Notice again that we use quotes to when using . notation
Retrieval Queries: Arrays of Documents

- `db.collection.find(<predicate>, optional <projection>)`
- `find( { instock: { loc: "A", qty: 5 } } )`
  - Exact match of document [like nested doc/atomic array case]
- `find( { "instock.qty": { $gte : 20 } } )`
  - One nested doc has $\geq 20$
- `find( { "instock.0.qty": { $gte : 20 } } )`
  - First nested doc has $\geq 20$
- `find( { "instock": { $elemMatch: { qty: { $gt: 10, $lte: 20 } } } } )`
  - One doc has $20 \geq qty >10$
- `find( { "instock.qty": { $gt: 10, $lte: 20 } } )`
  - One doc has $20 \geq qty$, another has $qty>10$
Retrieval Queries Template: Projection

- `db.collection.find(<predicate>, optional <projection>)`
- Use 1s to indicate fields that you want
  - Exception: `_id` is always present unless explicitly excluded
- OR Use 0s to indicate fields you don’t want
- Mixing 0s and 1s is not allowed for non `_id` fields

```
MongoDB> db.movies.find({cast: "Rebecca Ferguson", cast: "Tom Cruise"}, {title: 1})
{ "_id" : ObjectID("573a1397f29313caabce8ce7"), "title" : "Risky Business" }
{ "_id" : ObjectID("573a1398f29313caabce9932"), "title" : "Legend" }
{ "_id" : ObjectID("573a1398f29313caabcea315"), "title" : "Top Gun" }
```
Retrieval Queries : Addendum

- Two additional operations that are useful for retrieval:
  - Limit (k) like LIMIT in SQL
    - e.g., `db.inventory.find( {} ).limit(1)`
  - Sort ({} ) like ORDER BY in SQL
    - List of fields, -1 indicates decreasing 1 indicates ascending
    - e.g., `db.inventory.find( {}, { _id : 0, instock : 0 } ).sort( { "dim.0" : -1, item : 1 } )`
Retrieval Queries: Summary

find() = SELECT <projection>
      FROM Collection
      WHERE <predicate>

limit() = LIMIT

sort() = ORDER BY

```
db.inventory.find(
    { tags : red },
    {_id : 0, instock : 0}
).sort ( { "dim.0": -1, item: 1 } ).limit (2)
```

FROM
WHERE
SELECT
ORDER BY
LIMIT
Aggregation Pipelines

- Composed of a linear *pipeline* of *stages*
- Each stage corresponds to one of:
  - match // first arg of find()
  - project // second arg of find() but more expressiveness
  - sort/limit // same
  - group
  - unwind
  - lookup
  - … lots more!!
- Each stage manipulates the existing collection in some way

Syntax:
```
db.collection.aggregate([{
  $stage1Op: { }
},
  $stage2Op: { }
, ...
  $stageNOp: { }
])
```
```javascript
db.orders.aggregate( [ 
  { $match: { status: "A" } },
  { $group: { _id: "$cust_id", total: { $sum: "$amount" } } }
])
```
Grouping Syntax

- $group : {
  _id: <expression>, // Group By Expression
  <field1>: { <aggfunc1> : <expression1> },
  ...
}

Returns one document per unique group, indexed by _id

Agg.func. can be standard ops like $sum, $avg, $max

- Also MQL specific ones:
  - $first: return the first expression value per group
    • makes sense only if docs are in a specific order [usually done after sort]
  - $push: create an array of expression values per group
    • didn’t make sense in a relational context because values are atomic
  - $addToSet: like $push, but eliminates duplicates
Grouping Example

```
db.sales.insertMany([  
  { "_id" : 1, "item" : "abc", "price" : NumberDecimal("10"), "quantity" : NumberInt("2"), "date" : ISODate("2014-03-01T08:00:00Z") },  
  { "_id" : 2, "item" : "jkl", "price" : NumberDecimal("20"), "quantity" : NumberInt("1"), "date" : ISODate("2014-03-01T09:00:00Z") },  
  { "_id" : 3, "item" : "xyz", "price" : NumberDecimal("5"), "quantity" : NumberInt("10"), "date" : ISODate("2014-03-15T09:00:00Z") },  
  { "_id" : 4, "item" : "xyz", "price" : NumberDecimal("5"), "quantity" : NumberInt("20"), "date" : ISODate("2014-04-04T11:21:39.736Z") },  
  { "_id" : 6, "item" : "def", "price" : NumberDecimal("7.5"), "quantity" : NumberInt("5"), "date" : ISODate("2015-06-04T05:08:13Z") },  
  { "_id" : 7, "item" : "def", "price" : NumberDecimal("7.5"), "quantity" : NumberInt("10"), "date" : ISODate("2015-09-10T08:43:00Z") },  
  { "_id" : 8, "item" : "abc", "price" : NumberDecimal("10"), "quantity" : NumberInt("5"), "date" : ISODate("2016-02-06T20:20:13Z") } ]
])
```

The operation returns the following result:

```
{ "_id" : null, "count" : 8 }
```

This aggregation operation is equivalent to the following SQL statement:

```
SELECT COUNT(*) AS count FROM sales
```
Multiple Agg. Example

Find, for every state, the biggest city and its population

```
aggregate([
{ $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
{ $sort: { pop: -1 } },
{ $group: { _id : "$_id.state", bigCity: { $first: "$_id.city" }, bigPop: { $first: "$pop" } } },
{ $sort : {bigPop : -1} }
])
```

Approach:
- Group by pair of city and state, and compute population per city
- Order by population descending
- Group by state, and find first city and population per group (i.e., the highest population city)
- Order by population descending

```
{ "_id" : "IL", "bigCity" : "CHICAGO", "bigPop" : 2452177 }
{ "_id" : "NY", "bigCity" : "BROOKLYN", "bigPop" : 2300504 }
{ "_id" : "CA", "bigCity" : "LOS ANGELES", "bigPop" : 2102295 }
{ "_id" : "TX", "bigCity" : "HOUSTON", "bigPop" : 2095918 }
{ "_id" : "PA", "bigCity" : "PHILADELPHIA", "bigPop" : 1610956 }
{ "_id" : "MI", "bigCity" : "DETROIT", "bigPop" : 963243 }
```
If we only want to keep the state and city ...

```javascript
aggregate([
    {$group: {_id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } }},
    {$sort: { pop: -1 }},
    {$group: {_id: "$_id.state", bigCity: { $first: "$_id.city" }, bigPop: { $first: "$pop" } }},
    {$sort: {bigPop: -1}}
    {$project: {bigPop: 0}}
])
```

```
{ "_id": "IL", "bigCity": "CHICAGO" }
{ "_id": "NY", "bigCity": "BROOKLYN" }
{ "_id": "CA", "bigCity": "LOS ANGELES" }
{ "_id": "TX", "bigCity": "HOUSTON" }
{ "_id": "PA", "bigCity": "PHILADELPHIA" }
...
```
If we wanted to nest the name of the city and population into a nested document:

```javascript
aggregate([{
  $group: {_id: {state: "$state", city: "$city"}, pop: {$sum: "$pop"}}},
  $sort: {pop: -1}},
  $group: {_id: "$_id.state", bigCity: {$first: "$_id.city"}, bigPop: {$first: "$pop"}}},
  $sort: {bigPop: -1}},
  $project: {_id: 0, state: "$_id", bigCityDeets: {name: "$bigCity", pop: "$bigPop"}}}
])
```

Can construct new nested documents in output, unlike vanilla projection.

```json
{ "state": "IL", "bigCityDeets": { "name": "CHICAGO", "pop": 2452177 } }
{ "state": "NY", "bigCityDeets": { "name": "BROOKLYN", "pop": 2300504 } }
{ "state": "CA", "bigCityDeets": { "name": "LOS ANGELES", "pop": 2102295 } }
{ "state": "TX", "bigCityDeets": { "name": "HOUSTON", "pop": 2095918 } }
{ "state": "PA", "bigCityDeets": { "name": "PHILADELPHIA", "pop": 1610956 } }
...
```
Advanced Projection vs. Vanilla Projection

• In addition to excluding/including fields like in projection during retrieval (find), projection in the aggregation pipeline allows you to:
  – Rename fields
  – Redefine new fields using complex expressions on old fields
  – Reorganize fields into nestings or unnestings
  – Reorganize fields into arrays or break down arrays
Unwinding Arrays

• Deconstructs an array field from the input documents to output a document for each element

• Each output document is the input document with the value of the array field replaced by the element.

```javascript
db.inventory.insertOne({ "_id" : 1, "item" : "ABC1", sizes: [ "S", "M", "L"] })
```

The following aggregation uses the `$unwind` stage to output a document for each element in the `sizes` array:

```javascript
db.inventory.aggregate( [ { $unwind : "$sizes" } ] )
```

The operation returns the following results:

```javascript
[{ "_id" : 1, "item" : "ABC1", "sizes" : "S" }]
[{ "_id" : 1, "item" : "ABC1", "sizes" : "M" }]
[{ "_id" : 1, "item" : "ABC1", "sizes" : "L" }]
```
Looking Up Other Collections

• Conceptually, for each document
  – find documents in other collection that join (equijoin)
    • local field must match foreign field
  – place each of them in an array
• Thus, a left outer equi-join, with the join results stored in an array
• Recall: One of the key tenants of MongoDB schema design is to design to avoid the need for joins
Some Rules of Thumb when Writing Queries

- `$project` is helpful if you want to construct or deconstruct nestings (in addition to removing fields or creating new ones)
- `$group` is helpful to construct arrays (using `$push` or `$addToSet`)
- `$unwind` is helpful for unwinding arrays
- `$lookup` is your only hope for joins. Be prepared for a mess. Lots of `$project` needed
Creating Documents

db.products.insertMany([{
  "name": "GUCCI Handbag",
  "sku": "3451290",
  "description": "Fashion Hand bags for all ages",
  "inventory": 75
},
{
  "name": "Round hat",
  "sku": "8976045",
  "inventory": 200
},
{
  "name": "Polo shirt",
  "sku": "6497023",
  "description": "Cool shirts for hot summer",
  "inventory": 25
},
{
  "name": "Swim shorts",
  "sku": "8245352",
  "description": "Designer swim shorts for athletes",
  "inventory": 200
},
{
  "name": "Running shoes",
  "sku": "3243662",
  "description": "Shoes for work out and trekking",
  "inventory": 20
}])

{ "acknowledged": true, "insertedIds": [
  ObjectId("5f2ef849ec2a4c45809ae553"),
  ObjectId("5f2ef849ec2a4c45809ae554"),
  ObjectId("5f2ef849ec2a4c45809ae555"),
  ObjectId("5f2ef849ec2a4c45809ae556"),
  ObjectId("5f2ef849ec2a4c45809ae557")
] }
Updating Documents

```javascript
const update = db.products.update(
  { "sku": "1590234" },
  {
    $set: {
      reviews: [
        { rating: 4, review: "perfect glasses" },
        { rating: 4.5, review: "my priced posession" },
        { rating: 5, review: "Just love it" }
      ]
    }
  }
)

const result = WriteResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
```

The output indicates the number of documents that were matched, upserted and modified.
Deleting Documents

```
db.products.remove({"sku":"8976045"})
```

The output indicates the number of documents removed:

```
WriteResult({ "nRemoved" : 1 })
```
MongoDB: Summary

• MongoDB has now evolved into a mature "DBMS" with some different design decisions, and relearning many of the canonical DBMS lessons

• MongoDB has a flexible data model and a powerful (if confusing) query language.

• Many of the internal design decisions as well as the query & data model can be understood when compared with DBMSs
  – DBMSs provide a "gold standard" to compare against.
  – In the "wild" you’ll encounter many more NoSQL systems, and you’ll need to do the same thing that we did here!
MongoDB Demo
Reading and Next Class

• NoSQL
• Next: Transactions Part 1: Intro. to ACID: Ch 17