The React Datepicker is a popular component used to allow users to select dates or date ranges in React applications. It is often included in forms or UI where date inputs are required. One of the widely used libraries for this is react-datepicker, but other libraries like Material-UI Pickers, React-Day-Picker, and custom implementations also exist.

Here's a detailed discussion about react-datepicker, which is widely used for its flexibility and ease of integration.

Installation
To use react-datepicker, you need to install it via npm or yarn:

```
npm install react-datepicker
npm install date-fns  # Peer dependency for date manipulation
```

Basic Usage
Here is a minimal example of how to use it:

```
import React, { useState } from 'react';
import DatePicker from 'react-datepicker';
import 'react-datepicker/dist/react-datepicker.css'; // Required for styling

function App() {
  const [startDate, setStartDate] = useState(new Date());

  return (
    <DatePicker
      selected={startDate}
      onChange={(date) => setStartDate(date)}
      dateFormat="yyyy/MM/dd"
    />
  );
}

export default App;
```

Key Features

Single Date Selection

- ✓ Allows the selection of a single date.
- ✓ Use the selected and onChange props to manage the date state.

Range Selection

- ✓ Allows users to select a date range.
- ✓ Use startDate, endDate, and the selectsRange prop:

```
<DatePicker
  selected={startDate}
```

```
  onChange={(dates) => {
    const [start, end] = dates;
    setStartDate(start);
    setEndDate(end);
  }}
  startDate={startDate}
  endDate={endDate}
  selectsRange
/>
```

Custom Date Formats

Customize the date display format using the dateFormat prop.
Example: dateFormat="dd/MM/yyyy" or dateFormat="MMMM d, yyyy"

Time Selection

Add time selection using the showTimeSelect and timeFormat props.

```
<DatePicker
  selected={startDate}
  onChange={(date) => setStartDate(date)}
  showTimeSelect
  timeFormat="HH:mm"
  dateFormat="MMMM d, yyyy h:mm aa"
/>
```

Custom Styling

Style the component by overriding CSS classes. You can modify the appearance using CSS provided by the library or entirely customize it.

Data Between Parent and Child Components

Introduction

React is a popular JavaScript library for building user interfaces. One of the key features of React is its ability to manage state, which represents the data that drives the UI. In React, components are the building blocks of the UI, and they can have their own state. However, there are cases where you need to share state between components, and this is where lifting state up comes in. In this article, we will explore the concept of lifting state up in React, why it is important, and how to implement it effectively.

What is Lifting State Up?

Lifting state up is a pattern in React where we move the state from a lower-level component to a higher-level component, so that it can be shared between multiple child components.

In other words, we are moving the state up the component tree, so that it becomes the responsibility of a higher-level component.

By doing this, we are centralising the state management in a single component, which makes it easier to manage and avoids duplicating state across components.

Why is Lifting State Up Important?

Lifting state up is important in React for several reasons:

1. Avoiding Prop Drilling: Prop drilling is the process of passing data from a higher-level component to a lower-level component through props. This can lead to a lot of props being passed down the component tree, which can make the code harder to read and maintain. By lifting state up, we can avoid prop drilling and make the code more maintainable.

2. Centralising State Management: By lifting state up, we are centralising the state management in a single component, which makes it easier to manage and reduces the chances of bugs and inconsistencies in the state.

3. Reusability: By lifting state up, we are creating reusable components that can be used in different parts of the application. This reduces code duplication and makes the code more modular.

How to Lift State Up in React:

There are several ways to lift state up in React, and the approach you choose depends on the specific use case. In this section, we will explore three common approaches:

1. Using Callbacks:

- One way to lift state up in React is by using callbacks.
- In this approach, we pass a function down the component tree as a prop, and the child component calls the function when it needs to update the state.
- The function updates the state in the parent component, and the updated state is passed down to the child components as props.

Here's an example of lifting state up using callbacks:

```
//Parent.js
import React, { useState } from 'react';
import Child from './Child'
function Parent() {
  const [count, setCount] = useState(0);
  const handleIncrement = () => {
    setCount(count + 1);
  }
const handleDecrement = () => {
    setCount(count - 1);
  }
  return (
    <div>
      <h1>Count: {count}</h1>
      <Child onIncrement={handleIncrement}
                onDecrement={ handleDecrement}
 />
    </div>
  );
}
export default Parent;
```

```
//Child.js
import React from 'react';
function Child(props) {
  const { onIncrement, onDecrement } = props;
  return (
    <div>
      <button onClick={onIncrement}>Increment</button>
                
      <button onClick={onDecrement}>Decrement</button>
    </div>
  );
}
export default Child;

//App.js
import  React from ‘react’
import Parent from ‘./Parent’
function App()
{
      return(
        <div>
            <Parent />
         </div>
}
export  default App;
```

In this example, the `Parent` component has a state variable `count` that represents the count value. The `Parent` component also has a function `handleIncrement` that increments the count value.
The `Parent` component passes the `handleIncrement` function down to the `Child` component as a prop called `onIncrement`.
The `Child` component receives the `onIncrement` prop and uses it as an event handler for the `button` element. When the button is clicked, the `onIncrement` function is called, which updates the state in the `` `Parent `` component using the `setCount` function. This updates the state in the `Parent` component, and the updated count value is passed down to the `Child` component as a prop.

1. Using Context:

Another way to lift state up in React is by using context. Context provides a way to share data between components without having to pass it down through props explicitly. With context, you create a context object that holds the state and provide it to the components that need access to it. The child components can access the context and update the state by using the `useContext` hook.

Here's an example of lifting state up using context:

```
import React, { createContext, useContext, useState } from 'react';

const CountContext = createContext();

function Parent() {
  const [count, setCount] = useState(0);
  const handleIncrement = () => {
    setCount(count + 1);
  }
  return (
    <CountContext.Provider value={{ count, handleIncrement }}>
      <div>
        <h1>Count: {count}</h1>
        <Child />
      </div>
    </CountContext.Provider>
  );
}

function Child() {
  const { count, handleIncrement } = useContext(CountContext);
  return (
    <div>
      <button onClick={handleIncrement}>Increment</button>
      <p>Count: {count}</p>
    </div>
  );
}
```

In this example, the `Parent` component creates a context object called `CountContext` using the `createContext` function.
The `Parent` component has a state variable `count` that represents the count value. The `Parent` component also has a function `handleIncrement` that increments the count value.
The `Parent` component wraps the child components in the `CountContext.Provider` component and provides the `count` and `handleIncrement` values as the context value.

The `Child` component uses the `useContext` hook to access the context and get the `count` and `handleIncrement` values. The `Child` component uses the `handleIncrement` function as an event handler for the `button` element, which updates the state in the `Parent` component. The updated count value is passed down to the `Child` component as a prop.

1. Using Redux:

Redux is a popular state management library for React that provides a way to store and manage the state of the entire application in a single store. With Redux, you can lift the state up by moving it to the Redux store and accessing it in the child components using the `useSelector` hook.

Here's an example of lifting state up using Redux:

```
import React from 'react';
import { createStore } from 'redux';
import { Provider, useSelector } from 'react-redux';

const INCREMENT = 'INCREMENT';
```

```
function increment() {
  return { type: INCREMENT };
}

function counter(state = { count: 0 }, action) {
  switch (action.type) {
    case INCREMENT:
      return { count: state.count + 1 };
    default:
      return state;
  }
}

const store = createStore(counter);
function Parent() {
  return (
    <Provider store={store}>
      <div>
        <h1>Count: <Counter /></h1>
        <Child />
      </div>
    </Provider>
  );
}

function Counter() {
  const count = useSelector(state => state.count);
  return (
    <span>{count}</span>
  );
}

function Child() {
  const dispatch = useDispatch();
  const handleIncrement = () => {
    dispatch(increment());
  }
  return (
    <div>
      <button onClick={handleIncrement}>Increment</button>
    </div>
  );
}
```

In this example, the `Parent` component creates a Redux store using the `createStore` function from the `redux` library. The `counter` function is a reducer function that manages the state of the store.
The `increment` function is an action creator function that creates an action object with a type of `INCREMENT`.

The `Provider` component from the `react-redux` library wraps the child components and provides the Redux store as a prop.

The `Counter` component uses the `useSelector` hook to access the state from the store and get the `count` value. The `count` value is rendered as a span element.

The `Child` component uses the `useDispatch` hook to get the `dispatch` function, which is used to dispatch the `increment` action when the button is clicked. The `increment` action updates the state in the store, which updates the count value in the `Counter` component.

Conclusion:

Lifting state up in React is a common pattern used to manage the state of multiple components. By lifting the state up, you can pass data between components and keep the state in a single source of truth. You can lift the state up by using props, context, or a state management library like Redux.

When deciding which approach to use, consider the complexity of your application and the level of state management required. For simple applications, lifting state up using props may be sufficient. For more complex applications, using context or a state management library like Redux may be necessary.

Regardless of the approach you choose, lifting state up in React can help you build more scalable and maintainable applications.

CORS (Cross-Origin Resource Sharing) is a security feature implemented in web browsers to control how resources hosted on one domain can be accessed by a different domain. It's a mechanism that allows or restricts web applications running at one origin to interact with resources on another origin.

Key Concepts of CORS

Origin:

Defined as a combination of the protocol (e.g., http or https), domain, and port (if specified).

Example:

http://example.com is a different origin from https://example.com because of the protocol difference.

Similarly, http://example.com:3000 is a different origin from http://example.com.

Same-Origin Policy (SOP):

A fundamental security feature of browsers.

It restricts how a script loaded from one origin can interact with resources from another origin.

Without CORS, SOP would prevent most cross-origin HTTP requests (e.g., fetching data from an API hosted on another domain).

CORS as a Relaxation of SOP:

CORS provides a controlled way to allow cross-origin requests while maintaining security.

It's essentially a contract between the client (browser) and the server, negotiated using HTTP headers.

CORS Workflow

Preflight Requests:

Before making certain types of cross-origin requests (e.g., PUT, DELETE, or requests with custom headers), the browser sends a preflight request using the HTTP OPTIONS method.

The preflight request checks with the server to see if the actual request is allowed.

If allowed, the actual request follows; otherwise, the browser blocks the request.

HTTP Headers Involved:

Request Headers:

Origin: Indicates the origin of the request.

Access-Control-Request-Method: Specifies the HTTP method being used in the actual request (for preflight).

Access-Control-Request-Headers: Lists the headers the actual request intends to use.

Response Headers:

Access-Control-Allow-Origin: Specifies which origins are allowed access. (* allows all origins.)

Access-Control-Allow-Methods: Lists allowed HTTP methods (e.g., GET, POST).

Access-Control-Allow-Headers: Lists allowed headers in the request.

Access-Control-Allow-Credentials: Indicates if credentials (e.g., cookies, HTTP authentication) are allowed.

Access-Control-Expose-Headers: Specifies which headers can be accessed by the client.

Types of Requests

Simple Requests:

Requests that meet specific criteria:

HTTP methods: GET, POST, HEAD.

No custom headers (only standard headers like Content-Type).

If the request qualifies as simple, the browser skips the preflight request.

Non-Simple Requests:

Reuests that don’t meet the criteria for simple requests.

Require a preflight request to verify permissions.

Common CORS Errors

Access to XMLHttpRequest has been blocked by CORS policy:

The server has not included the necessary Access-Control-Allow-Origin header in its response.

CORS preflight channel did not succeed:

The preflight request failed, likely because the server didn't respond with the required headers.

Introduction to NoSQL Databases

- **NoSQL**:
  - Non Relational SQL
  - Not Only SQL
  - Not SQL
- Is an approach to database design that provides flexible schemas for the storage and retrieval of data beyond the traditional table structures found in relational databases.
- NoSQL database are classified into four types
  1. Key value stores
  2. Wide column stores
  3. Document stores
  4. Graph databases

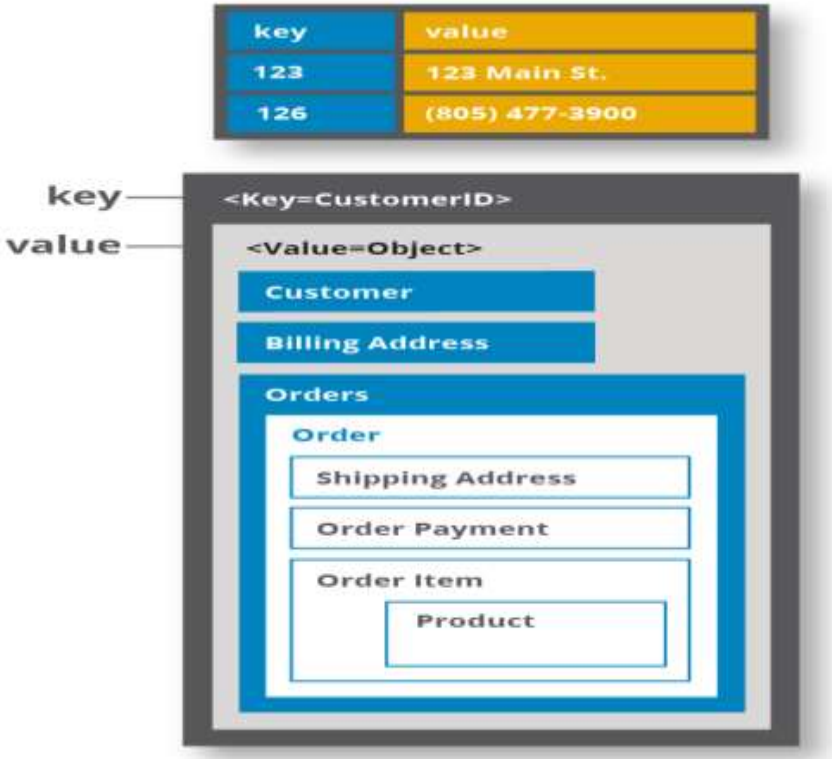
why we use NoSQL databases

- NoSQL was Born from Frustration with SQL
- Storing large volumes of data without structure
- Using cloud computing and storage-Scalability
- Indexing

Key-Value stores

- In Key-Value stores data is stored in the form of key and value pairs.

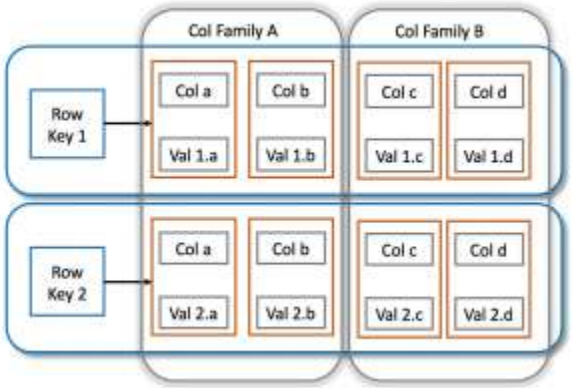
  Ex: Redis, Memcached, HazelcastRedis etc



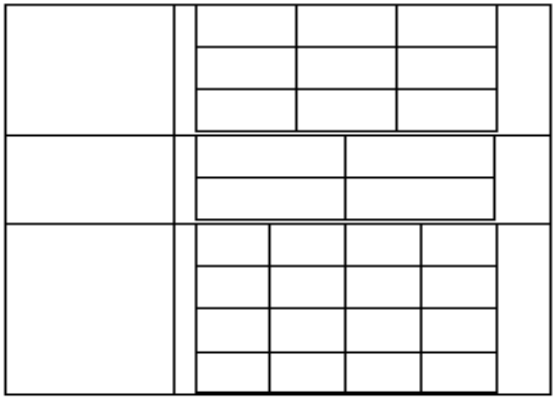
Key-Value database

Wide Column Stores

- In wide column stores data is stored in row-by-row fashion such that the columns in a row are stored together instead of storing seperatly.
- Ex: Cassandra, HBase, Google Bigtable etc



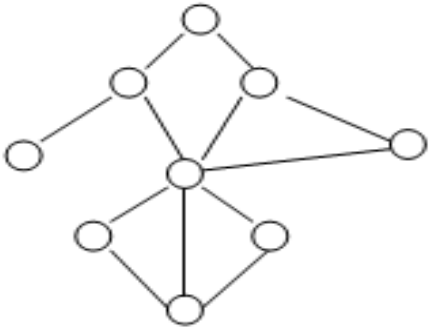
wide column store

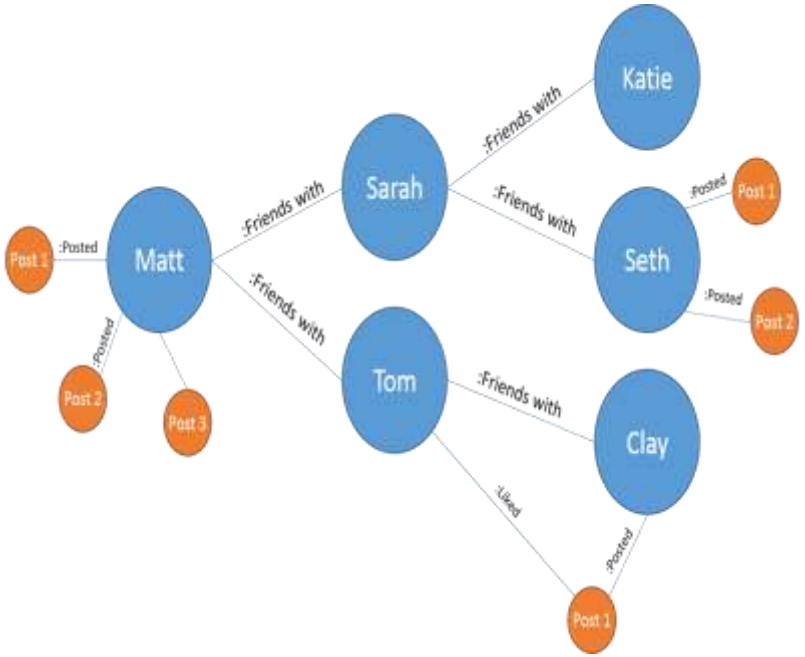


Graph Databases

- In graph databases data is stores in form of vertices and edges. Vertices contains actual object and edges represents relationships among those objcets.
- Ex: Neo4j, Giraph, JanusGraph etc

Graph database

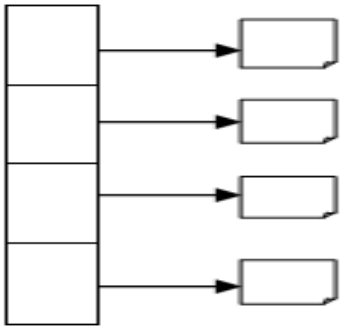


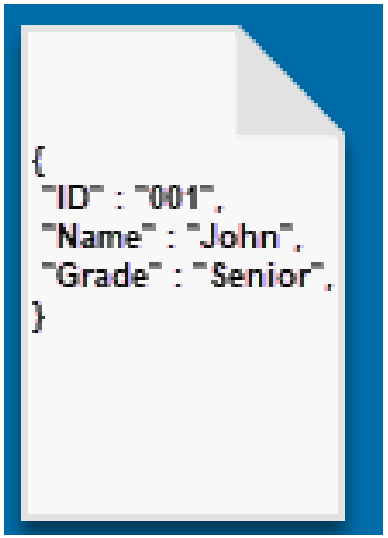


Document Store

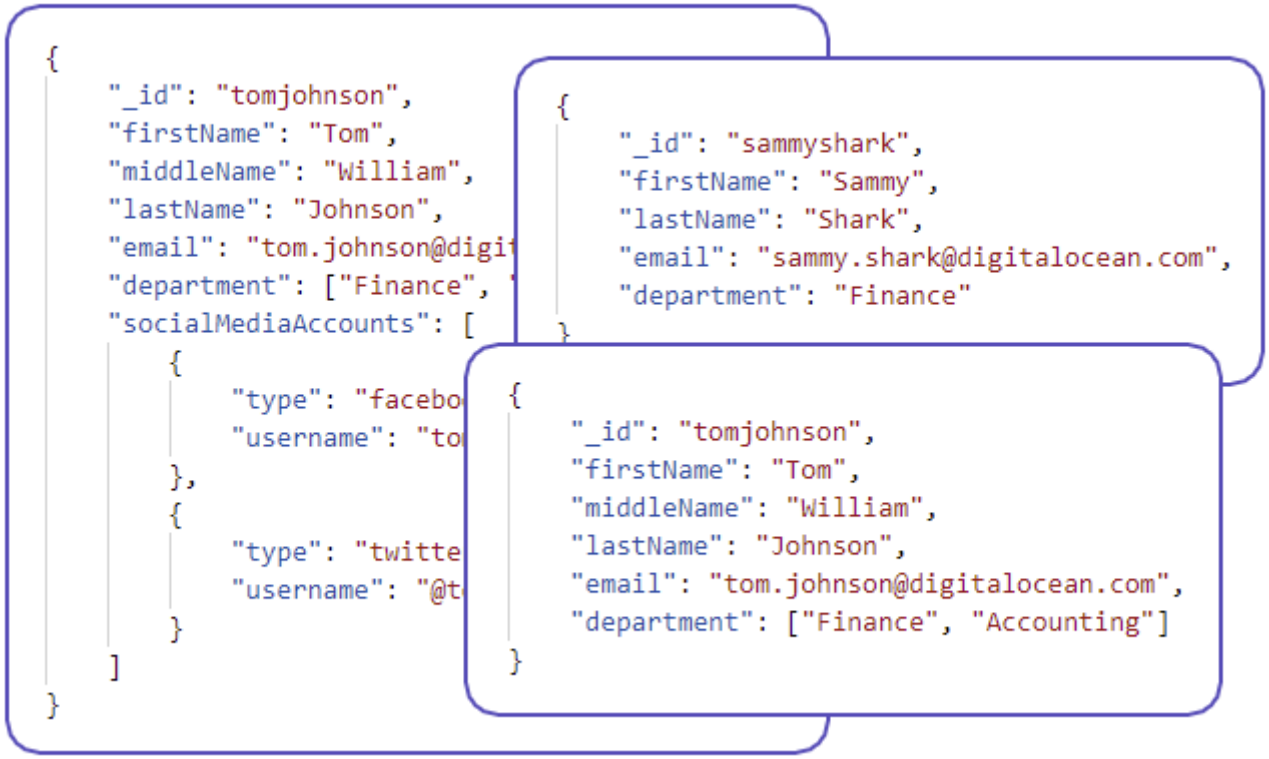
- In Document oriented databases data is stored in the form of collections which contains documents.
- Ex: MongoDB, Couchbase, CouchDB etc

Document store





- Instead of storing data in fixed rows and columns, document databases use flexible documents.
- A document is a record in a document database. A document typically stores information about one object and any of its related metadata.
- Documents store data in field-value pairs. Documents can be stored in formats like JSON, BSON, and XML.


{
"_id": "tomjohnson",
"firstName": "Tom",
"middleName": "William",
"lastName": "Johnson",
"email": "tom.johnson@digi
"department": ["Finance",
"socialMediaAccounts": [
{
"type": "facebo
"username": "to
},
{
"type": "twitte
"username": "@t
}
]
}
{
"_id": "sammyshark",
"firstName": "Sammy",
"lastName": "Shark",
"email": "sammy.shark@digitalocean.com",
"department": "Finance"
}
{
"_id": "tomjohnson",
"firstName": "Tom",
"middleName": "William",
"lastName": "Johnson",
"email": "tom.johnson@digitalocean.com",
"department": ["Finance", "Accounting"]
}

key features

- **Document model:** Data is stored in documents (unlike other databases that store data in structures like tables or graphs). Documents map to objects in most popular programming languages, which allows developers to rapidly develop their applications.
- **Flexible schema**: Document databases have a flexible schema, meaning that not all documents in a collection need to have the same fields.
- **Distributed and resilient:** Document databases are distributed, which allows for horizontal scaling
- **Querying through an API or query language:** Document databases have an API or query language that allows developers to execute the CRUD operations on the database. Developers have the ability to query for documents based on unique identifiers or field values.

Introduction to MongoDB

- MongoDB is an open source, document oriented database that stores data in form of documents.
- This means that it doesn't use tables and rows to store its data, but instead *collections* of JSON-like *documents*
- MongoDB was developed by **Eliot Horowitz** and **Dwight Merriman** in the year **2007**, when they experienced some scalability issues with the relational database.
- MongoDB also provides the feature of Auto-Scaling

Cont..

- A Document is nothing but a data structure with key-value pairs like in JSON. It is very easy to map any custom Object of any programming language with a MongoDB Document.

```
{   name : “Anil",
    rollno : 1,
subjects : ["C Language", "C++", "Core Java"]}
```

Translating between relational and document data models

RDBMS	MongoDB
Database	Database
Table	Collection
Row	Document (BSON)
Index	Index
JOIN	Embedded Document or Reference

MongoDB vs MYSQL

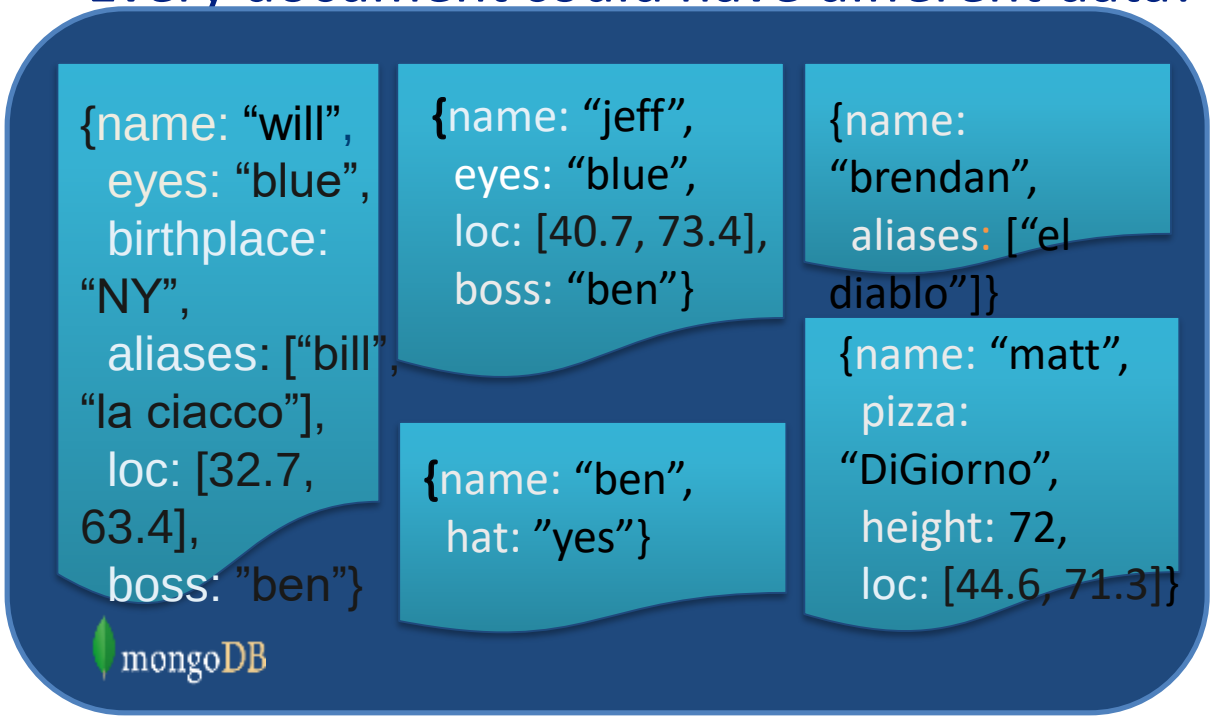
Feature	MySQL	MongoDB
Data Structure	It stores each individual record as a table cell with rows and columns	It stores unrelated data in JSON like documents
Schema	MySQL requires a schema definition for the tables in the database	MongoDB doesn't require any prior schema
Languages	Supports Structured Query Language (SQL)	Supports JSON Query Language to work with data
Foreign Key	Supports the usage of Foreign keys	Doesn't support the usage of Foreign keys
Replication	Supports master-slave replication and master-master replication	Supports sharding and replication
Scalability	SQL Database can be scaled vertically	MongoDB database can be scaled both vertically and horizontally
Join Operation	Supports Join operation	Doesn't support Join operation
Performance	Optimized for high performance joins across multiple tables	Optimized for write performance
Risks	Prone to SQL injection attack	Since there's no schema, lesser risks involved

Mongodb MySQL feature comparison

Features	MySQL	MongoDB
Rich Data Model	No	Yes
Dynamic Schema	No	Yes
Typed Data	Yes	Yes
Data Locality	No	Yes
Field Updates	Yes	Yes
Easy for Programmers	No	Yes
Complex Transactions	Yes	No
Auditing	Yes	Yes
Auto-Sharding	No	Yes

Schema Free

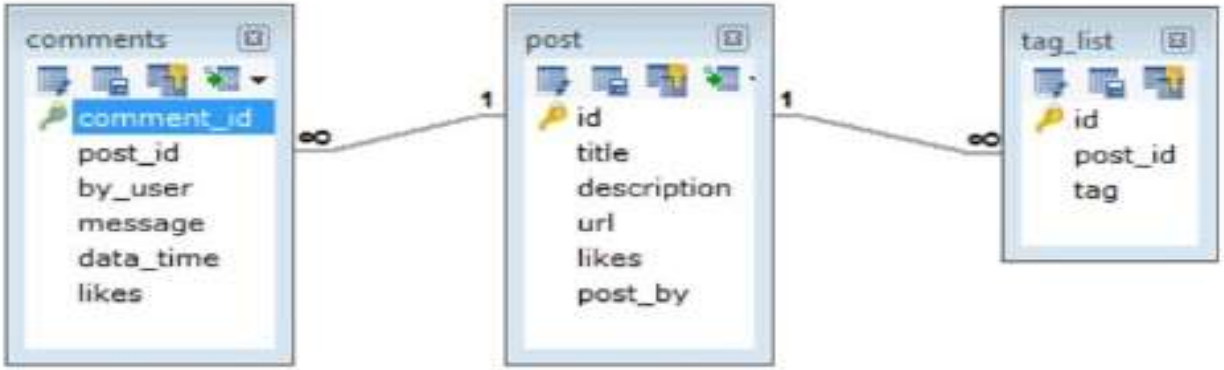
- MongoDB does not need any pre-defined data schema
- Every document could have different data!




Collections
{
"_id" : ObjectId("527b3cc65ceafed9b2254a94"),
"f_name" : "Zenny",
"sex" : "Female",
"class" :
"age" :
"grd_poi
}
Document1
{
"_id" : ObjectId("527b3cc65ceafed9b2254a95"),
"f_name" : "Paul",
"sex" : "Male",
"class" : "VII",
"age" : 13,
Document2
{
"_id" : ObjectId("527b3cc65ceafed9b2254a97"),
"f_name" : "Lassy",
"sex" : "Female",
"class" : "VIII",
"age" : 13,
"grd_point" : 28.2514
}
Document3

Designing Schema in MongoDB

- Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.
  - Every post has the unique title, description and url.
  - Every post can have one or more tags.
  - Every post has the name of its publisher and total number of likes.
  - Every post has comments given by users along with their name, message, data-time and likes.
  - On each post, there can be zero or more comments.
  - In RDBMS schema, design for above requirements will have minimum three tables.
- In RDBMS schema, design for above requirements will have minimum three tables.



- While in MongoDB schema, design will have one collection post and the following structure

```
{
   _id: POST_ID
   title: TITLE_OF_POST,
   description: POST_DESCRIPTION,
   by: POST_BY,
   url: URL_OF_POST,
   tags: [TAG1, TAG2, TAG3],
   likes: TOTAL_LIKES,
   comments: [
      {
         user:'COMMENT_BY',
         message: TEXT,
         dateCreated: DATE_TIME,
         like: LIKES
      },
      {
         user:'COMMENT_BY',
         message: TEXT,
         dateCreated: DATE_TIME,
         like: LIKES
      }
   ]
}
```

Why MongoDB

- It is flexible
- It supports dynamic schema
- It support fast bulk writes
- It supports Map-Reduce
- It supports complex quering
- The most attracting feature is it allows multi type documents in a single collection.
- Provides expressive object data model
- Supports in-memory processing
- Auto sharding
- Built-in text search
- Automatic failover of nodes
- Powerful indexing – Secondary indexes
- Multi-Document Transaction

Working with MongoDB

- Databases
  - Create or switch Database
    - Syntax: use DATABASE_NAME
    - Example: use spmvv
  - Check current Database
    - db
  - Check list of Databases
    - show dbs
  - Drop Database
    - db.dropDatabase()

Datatypes supported by MongoDB

- String
- Integer
- Boolean
- Double
- Min/ Max keys
- Arrays
- Timestamp
- Object
- Null
- Date
- Object ID
- Binary data
- Code
- Regular expression

Cont..

- Collections
  - Creating collection
    - Syntax: db.createCollection(name, options)
    - Example: db.createCollection(“student”)
  - Dropping Collection
    - Syntax: db.COLLECTION_NAME.drop()
    - Example: db.student.drop()

Cont..

- Documents
  - Inserting
    - use **insert()** or **save()**
    - Syntax: db.COLLECTION_NAME.insert(document)
    - Eample: db.student.insert({id:1,name:"smith“})
      db.student.insert([{ id:2,name:"two"},
      {id:3,name:"three"},
      {id:4,name:"four"}])
  - Find() and pretty():
    - db.student.find()
    - db.student.find().pretty()
  - insertOne()
    - Syntax: db.COLLECTION_NAME.insertOne(document)
    - Example: db.student.insertOne({ id:5,name:"five"})

Cont..

- insertMany()
  - Syntax: db.COLLECTION_NAME.insertMany(documents)
  - Example: db.student.insertMany([{id:6,name:"six"},{id:7,name:"seven"}, {id:8,name:"eight"}])
- findOne()
  - Syntax: db.COLLECTION_NAME.findOne()
  - Example: db.student.findOne()
  - Example: db.student.findOne({id : 3})
- find().limit()
  - Syntax: db.COLLECTION-NAME.find().limit(no.of documents)
  - Example: db.student.find().limit(2)
- Count()
  - Gives the no.of documents in a specified collection
  - db.student.count()

Cont..

- Remove()
  - Syntax: db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)
  - Example: db.student.remove({"id":2})
- Update()
  - Syntax: db.COLLECTION_NAME.update(SELECTION_CRITERIA, UPDATED_DATA)
  - Example: db.student.update({"id":1}, {$set:{"name":"rajesh"}})
- updateMany()

Cont..

- Quering
  - Relational Operators
    - $eq Matches values that are equal to a specified value.
    - $gt Matches values that are greater than a specified value.
    - $gte Matches values that are greater than or equal to a specified value.
    - $in Matches any of the values specified in an array.
    - $lt Matches values that are less than a specified value.
    - $lte Matches values that are less than or equal to a specified value.
    - $ne Matches all values that are not equal to a specified value.
    - $nin Matches none of the values specified in an array.

Cont..

- **Logical**
  - $and
    - { $and: [{ <expression1> }, { <expression2> } , ... , { <expressionN> }] }
  - $not
    - { field: { $not: { <operator-expression> } } }
  - $nor(returns all that fail to match both clauses)
    - { $nor: [{ <expression1> }, { <expression2> }, ... { <expressionN> }] }
  - $or
    - { $or: [{ <expression1> }, { <expression2> }, ... , { <expressionN> }] }

Cont..

- **Element**
- $exists Matches documents that have the specified field.
- { field: { $exists: <boolean> } }
- $type Selects documents if a field is of the specified type.
- { field: { $type: <BSON type> } }

Example-1

1.create a database called employee

2.create a collection called emp

3.insert atleast 5 to 10 records individually with following fields fname,lname,city,age,salary

```
db.emp.insert({fname:"Raj",lname:"roy",city:"city1",age:34,salary:34000})
db.emp.insert({fname:"Sasi",lname:"raj",city:"cit2",age:45,salary:10000})
db.emp.insert({fname:"Paul",lname:"john",city:"cit3",age:25,salary:10000})
db.emp.insert({fname:"Ravan",lname:"setu",city:"city4",age:20,salary:15000})
db.emp.insert({fname:"Chitra",lname:"ss",city:"city5",age:34,salary:1000})
```

4.display the top 5 records

```
db.emp.find().limit(5)
```

Cont..

5. update salary to 10000 where salary equals to 5000

```
db.emp.find({salary:5000})
db.emp.update({salary:5000},{$set:{salary:10000}})
db.emp.update({salary:5000},{$set:{salary:10000}},{multi:true})
```

6. update field fname with “Raj” to “Mr.Raj”

```
db.emp.update({fname:"raj"},{$set:{fname:"Mr.Raj"}})
```

7. Delete documents with salary equals to 5000

```
db.emp.remove({salary:5000})
```

Cont..

8.Display emp records with fname,lname and Salary

```
db.emp.find({},{fname:1,lname:1,salary:1,_id:0})
```

9.Display emp records excluding fname

```
db.emp.find({},{fname:0})
```

Cont..

10.insert a emp record giving your own id to the employee

```
db.emp.insert({_id:123,fname:"over",lname:"ride",city:"gnt",age:33,salary:26000})
```

Excerise on Relational Operators

- Create a collection called stock, the collection should have _id(override),qty, item(this field has subdocument with fields name,code)

```
db.stock.insertMany([
    { _id: 1, item: { name: "ab", code: "123" }, qty: 15},
    { _id: 2, item: { name: "cd", code: "123" }, qty: 20},
    { _id: 3, item: { name: "ij", code: "456" }, qty: 25},
    { _id: 4, item: { name: "xy", code: "456" }, qty: 30},
    { _id: 5, item: { name: "mn", code: "000" }, qty: 20}])
```

1.Find all documents with qty=20

```
db.stock.find({qty:{$eq:20}})
```

2.find all documents with name=”ab”

```
db.stock.find({"item.name":{$eq:"ab"}})
```

3.Display documents whose qty >15

```
db.stock.find({qty:{$gt:15}})
```

4.Display documents whose qty < 15

```
db.stock.find({qty:{$lt:15}})
```

5. Display documents whose qty >=15

```
db.stock.find({qty:{$gte:15}})
```

6.Display documents whose qty<=30

```
db.stock.find({qty:{$lte:30}})
```

7.Display docs whose qty not equal to 30

```
db.stock.find({qty:{$ne:30}})
```

Excerise on Logical Operators

1.Display whose items have name=”ab” and qty=15

```
db.stock.find( {$and:[{"item.name":"ab"},{qty:15}]}  )
```

2. Display whose items have code=123 and qty!=15

```
db.stock.find( {$and:[{"item.code":"123"},{qty:{$ne:15}}]}  )
```

3.insert the following record :_id:6
Item:{name:”something”} qty:50

```
db.stock.insert({_id:6, item:{name:"something"},qty:50})
```

4.display whose qty!=15 and item.name exists

```
db.stock.find(
   {$and:[{qty:{$ne:15}},{"item.name":{$exists:true} }]}  )
```

5.display whose qty!=15 and item.code exists

```
db.stock.find( {$and:[{qty:{$ne:15}},{"item.code":{$exists:true}
   }]}  )
```

Cont..

6.display items whose qty not greater than 15

```
db.stock.find( {qty:{  $not:{$gt:15}}} )
```

7.display whose qty is 15 or 20

```
db.stock.find( {$or:[{qty:15},{qty:20}] } )
```

8.*** insert a document with item.name=123(integer type)

```
db.stock.insert({ _id:7, item:{name:123},qty:50})
```

9.find all docs with item.name type as “number”

```
db.stock.find({"item.name":{$type:"number"}})
```

10. find all docs whose item.name type is “string”

```
db.stock.find({"item.name":{$type:"string"}})
```

Arrays

- **Array Query Operators**
- $all Matches arrays that contain all elements specified in the query.
- { <field>: { $all: [<value1> , <value2> ...] } }
- $elemMatch Selects documents if element in the array field matches all the specified $elemMatch conditions
- $size Selects documents if the array field is a specified size.
- $push
- $pop

Arrays

1. Create a collection by name “arr” with an id field and array field “a” with values 1,2,3,4

   db.arr.insert({_id:1,a:[1,2,3,4]})

2. update the array by adding 5 and 6 (use update)

   db.arr.update({_id:1},{a:[1,2,3,4,5,6]})

3. push element 7 in the array

   db.arr.update({_id:1}, {$push:{a:7}})

4. push elements 8,9,10

   db.arr.update({_id:1}, {$push:{a:[8,9,10]}})

5. insert two records using insertMany _id:3 and a:[52,3] , _id:4,a:[22,34,56,1,2]

   db.arr.insertMany([{_id:3,a:[52,3]},
   {_id:4,a:[22,34,56,1,2]}])

Cont..

6. Display docs which have 1,2 in the array “a”

```
db.arr.find({a:{$all:[1,2]}})
```

7. Write the “and” version of the above query

```
db.arr.find({$and:[{a:1},{a:2}]})
```

8. find all docs having array elements match(atleast one) >20 and <90

```
db.arr.find( {a:{$elemMatch:{$gte:20,$lt:90}}})
```

9. find all whose array size is 2

```
db.arr.find({a:{$size:2}})
```

10. find array whose sizes are 2 or 3

```
db.arr.find({$or:[{a:{$size:2}}, {a:{$size:3}} ]})
```

Cont..

11. find docs whose array size is nor 2 and 3

```
db.arr.find({$nor:[{a:{$size:2}}, {a:{$size:3}} ]})
```

12.display array size having > 3

```
db.arr.find({"a.2":{$exists:true}})
```

13.add 200,300,400 to _id:1

```
db.arr.update({_id:1},{$push:{a:{$each:[300,400,500] } }} )
```

14. remove first element from array in _id:1

```
db.arr.update( { _id: 1 }, { $pop: { a: -1 } } )
```

14.remove last element from array where _id:1

```
db.arr.update( { _id: 1 }, { $pop: { a: 1 } } )          //pop accepts 1 or -1
```

15.sort records based on _id's

```
db.arr.find().sort({_id:1})          //ascending
db.arr.find().sort({_id:-1})
```

16. Sort the elements in array

```
db.arr.update({},{$push: {a: {$each:[], $sort:-1}}}, {multi:true})
```

Indexes

- Creating Index:
  - Syntax: db.COLLECTION_NAME.createIndex({KEY:1})
  - Example: db.emp.createIndex({“_id":1})
- Get Indexes
  - Syntax: db.COLLECTION_NAME.getIndexes()
  - Example: db.emp.getIndexes()
- Drop Index:
  - Syntax:db.COLLECTION_NAME.dropIndex({KEY:1})
  - Example: db.emp.dropIndexes({“_id":1})

MapReduce

- Analytics:

To perform analytics we need to handle large data sets. We can directly import those data sets into MongoDB using the command

syntax: mongoimport --db dbname --collection collectionname --type csv --headerline --file filename

example: mongoimport --db mydb --collection movie --type csv --headerline --file movies_data.csv

We can display the documents using

```
db.movie.find().pretty()
```

We can find no.of documents in collection using

```
db.movie.count()
```

suppose if we want to display movies released in 1999 then

```
db.movie.find({year:1999}).pretty()
db.movie.find({year:1999}).count()
```

To access the data quickly we can add indexes to the fields in document

```
db.movie.createIndex({title:"text"})
```

we can get the indexes which are add to the collection using

```
db.movie.getIndexes()
```

Cont..

find all movies that have a synopsis that contains the word "Boys"

***to use text index use $text operator, there must be only one text index*

```
db.movie.find({$text:{$search:"Boys"}}).pretty()
```

find all movies that have a synopsis that contains the word "Boys" or "Mysteries"

```
db.movie.find({$text:{$search:"Boys Mysteries"}}).pretty()
```

find all movies that have a synopsis that contains the word " Inspector " and "Mysteries"

```
db.movie.find({$text:{$search:"\"Inspector\"  \"Mysteries\""}} ).pretty()
```

find all movies that have a synopsis that contains the word "Boys" and not the word "Mysteries"

```
db.movie.find({$text:{$search:"Boys -Mysteries"}}).pretty()
```

Cont..

MapReduce:

Two basic elements

Mapper: Mapper maps key with its value<k1,v1>→list(<k2,v2>)

Reducer: Reducer reduces the values associated with the key. <k2,list(v2)>→<k3,v3>

Example

Hadoop is good <#001, Hadoop is good>

Hadoop is bad <#002, Hadoop is bad>

Hadoop is very good <#003, Hadoop is very good>

Split: hadoop, is, good, hadoop, is, bad, hadoop, is, very, good

Map:
<hadoop, 1>
<is,1>
<good,1>
<hadoop, 1>
<is,1>
<bad, 1>
<hadoop, 1>
<is,1>
<very,1>
<good,1>

Reduce:
<hadoop, 1,1,1>
<is, 1,1,1>
<good, 1,1>
<bad, 1>
<very, 1>

Output:
<hadoop, 3>
<is, 3>
<good,2>
<bad, 1>
<very, 1>

Cont..

- write a MapReduce algorithm to find no.of movies released in each year
- write a MapReduce algorithm to find top rating in each year
- write a MapReduce algorithm to find average rating in each year
- write a MapReduce algorithm to find no.of citations for each patent
- write a MapReduce algorithm to find references of each patent

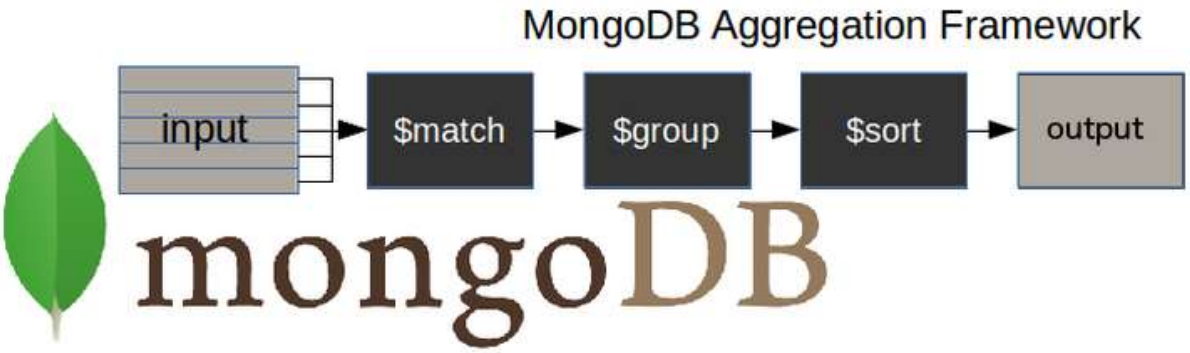
MongoDB-aggregation

- In MongoDB, aggregation operation collects values from various documents and groups them together and then performs different types of operations on that grouped data like sum, average, minimum, maximum, etc to return a computed result.
- **The aggregate() Method**

  For the aggregation in MongoDB, we can use **aggregate()** method.

Aggregation pipeline

- **Aggregation** is a way of processing a large number of documents in a collection by means of passing them through different stages. The stages make up what is known as a pipeline.



- $match stage – filters those documents we need to work with, those that fit our needs
- $group stage – does the aggregation job
- $sort stage – sorts the resulting documents the way we require (ascending or descending)

MongoDB aggregate pipeline syntax

- This is an example of how to build an aggregation query:
- **db.*collectionName*.aggregate(*pipeline*, *options*)**,
- where ***collectionName*** – is the name of a collection,
- ***pipeline*** – is an array that contains the aggregation stages,
- ***options*** – optional parameters for the aggregation
- example of the aggregation pipeline syntax:
- pipeline = [{ $match : { … } }, { $group : { … } }, { $sort : { … } }]

- db.movie.aggregate({$match:{year:1999, rating:3.5}})
- db.movie.aggregate([{$group:{‘_id’:”$year”, total: {$sum:1}}},{$sort:{_id:-1}}]).pretty()
- db.movie.aggregate([{$match:{rating:3.5}},{$group:{‘_id’:$year, total: {$sum:1}}},{$sort:{_id:-1}}]).pretty()

- **Accumulators:** These are basically used in the group stage
- **sum:** It sums numeric values for the documents in each group
- **count:** It counts total numbers of documents
- **avg:** It calculates the average of all given values from all documents
- **min:** It gets the minimum value from all the documents
- **max:** It gets the maximum value from all the documents
- **first:** It gets the first document from the grouping
- **last:** It gets the last document from the grouping
- db.movie.aggregate({$group:{‘_id’:$year, total: {$sum:1}, max_rating:{$max:’$rating}}}).pretty()

Unwinding, out, limit

- **Unwinding** works on array , array will be deconstructed and the output will be the documents for each element in the array.
- db.student.aggregate([{$unwind:‘$subject’}])
- **OUT:**
- This is an unusual type of stage because it allows you to carry the results of your aggregation over into a new collection
- The $out stage must be the last stage in the pipeline.
- db.movie.aggregate([{$match:{year:1999, rating:3.5}},{$out:’result’}])
- **Limit:**
- db.movie.aggregate([{$match:{year:1999, rating:3.5}},{$limit:1}])