

Practical No: 01

Aim: To design EER diagrams using StarUML

Description:

An **Entity-Relationship (ER) Diagram** is a visual representation of the structure of a database. It shows the major entities in a system, their attributes, and the relationships between those entities.

- **Entities** represent real-world objects or concepts, such as *Student*, *Customer*, *Product*, etc.
- **Attributes** are the properties or details of each entity, like *name*, *ID*, *address*, etc.
- **Relationships** illustrate how entities are connected, for example, a *Student enrolls in a Course*.

ER diagrams help in designing a clear and logical database structure before actual implementation. They are widely used in database design to ensure data consistency, normalization, and integrity.

- 1) For a given Scenario draw EER diagram and convert entities and relationship into tables

Scenario: Hospital Management System

The system keeps records of patients, hospitals, doctors, and their medical records. Patients are admitted to hospitals. Each hospital employs multiple doctors. Each patient can have multiple medical records.

Step1:Entities and Attributes Identified

1. Patient

- Pat_id (*Primary Key*)
- PName
- PDiagnosis
- PAddress

2. Hospital

- Hos_id (*Primary Key*)
- HName
- HAddress
- HCity

3. Doctor

- Doc_id (*Primary Key*)
- DName
- Qualification
- Salary

4. Medical Record

- Precord_id (*Primary Key*)
- Date_of_examination
- Problem

Step2: Relationships Identified

1. Admitted_in (Patient - Hospital)

- Many-to-One (M:1)
- One hospital can have many patients.
- Foreign Key: Hos_id in Patient table.

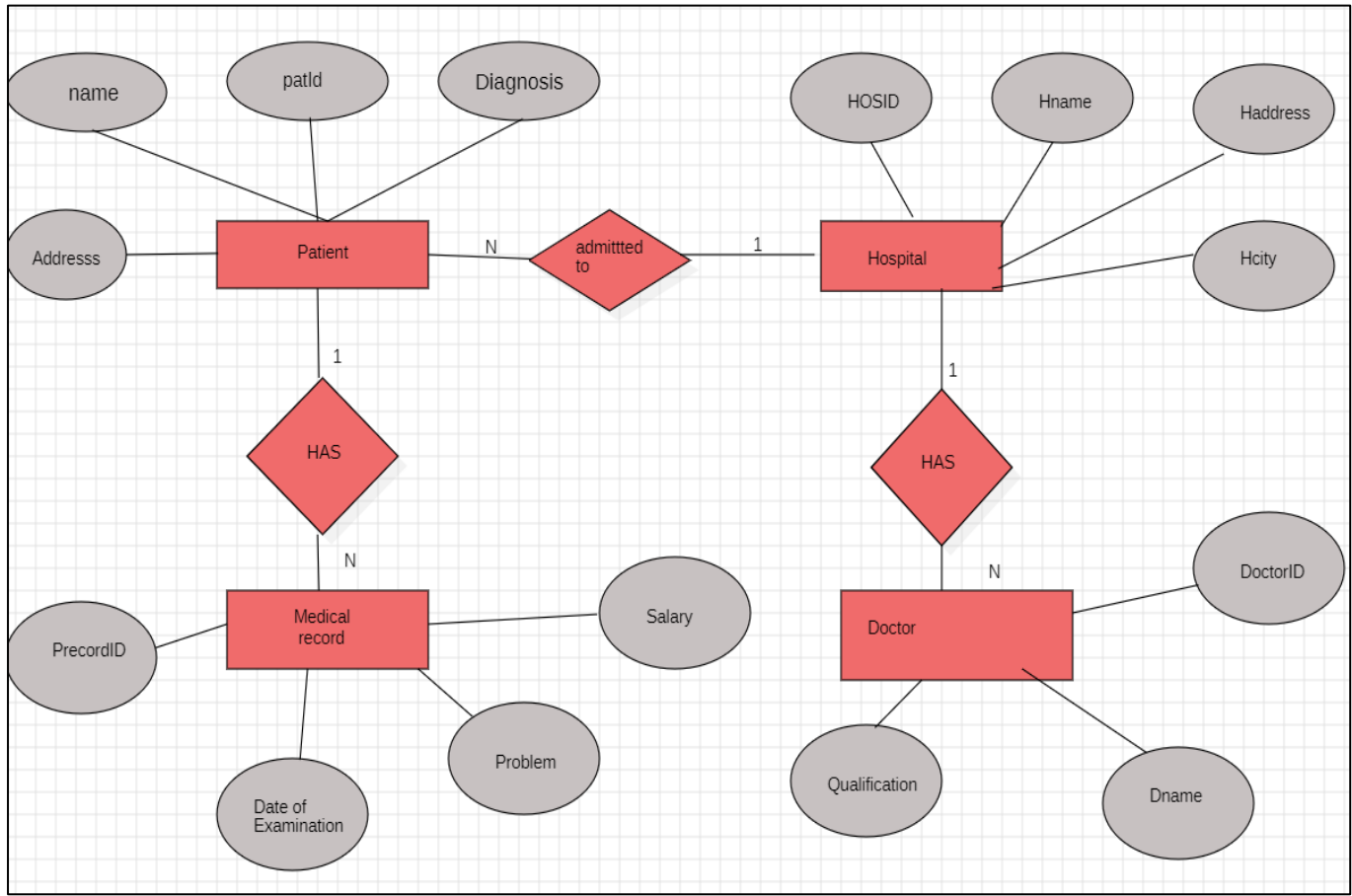
2. Has (Hospital - Doctor)

- One-to-Many (1:N)
- One hospital has many doctors.
- Foreign Key: Hos_id in Doctor table.

3. Has (Patient - Medical Record)

- One-to-Many (1:N)
- One patient has multiple medical records.
- Foreign Key: Pat_id in MedicalRecord table.

Step 3: Draw EER Diagram in StarUML



Description: This ER diagram represents a **Hospital Management System** involving entities like Patient, Hospital, Doctor, and Medical Record.

- **Patient** has attributes like patId, name, Addresss, and Diagnosis.
- Each **Patient** is *admitted to* one **Hospital**, but a **Hospital** can admit many patients (1:N).
- **Hospital** has attributes like HOSID, Hname, Haddress, and Hcity.
- A **Hospital** *has* many **Doctors** (1:N), and each **Doctor** is associated with one Hospital.
- **Doctor** has attributes like DoctorID, Dname, Qualification, and Salary.
- A **Patient** *has* many **Medical Records** (1:N), but each record is tied to one patient.
- **Medical Record** has attributes such as PrecordID, Date of Examination, Problem, and Salary.

Each relationship (like *admitted to* or *has*) shows cardinality and links relevant entities clearly.

Step 4: Convert Entities and Relationships into Tables using SQL Write

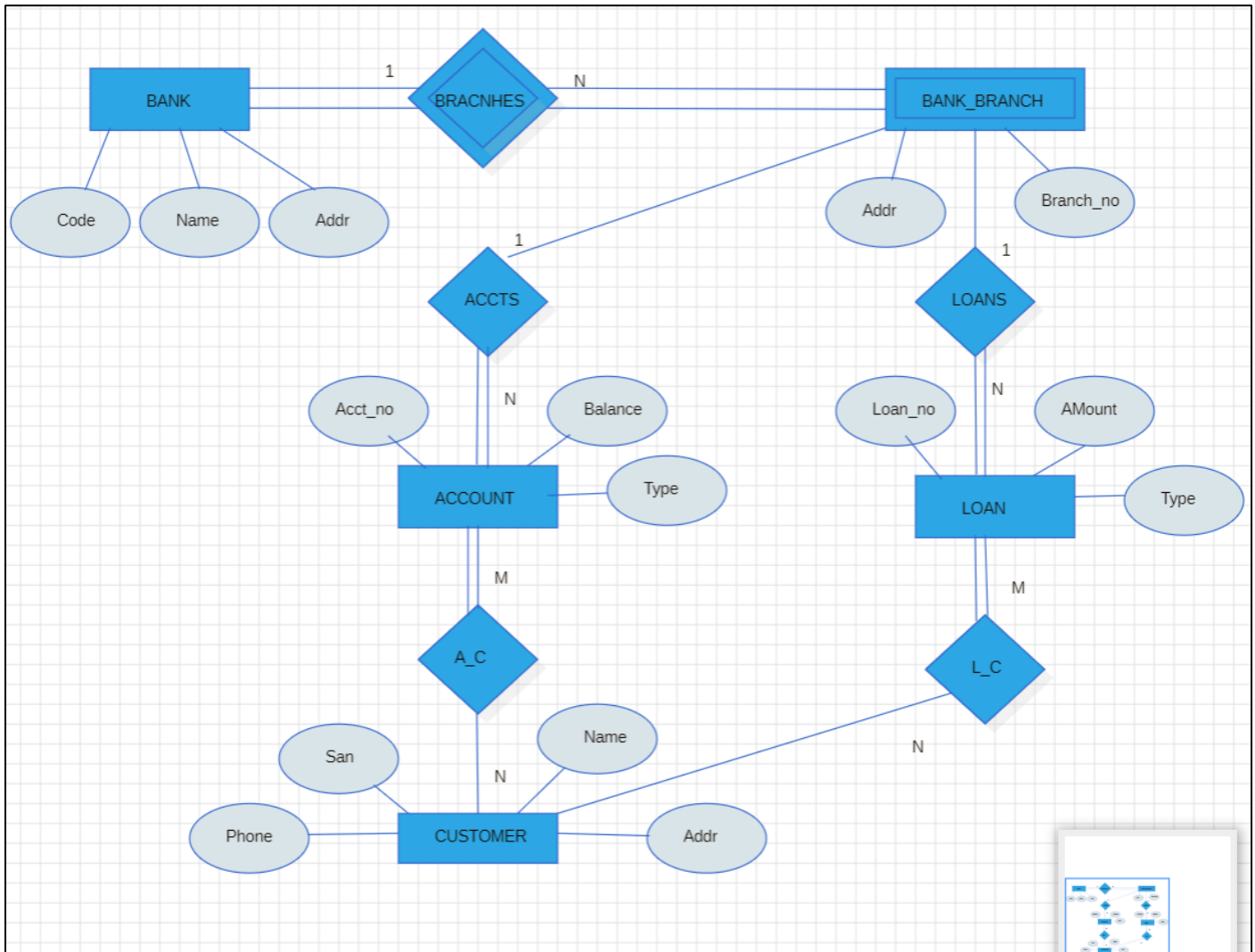
SQL CREATE TABLE queries to represent entities and relationships.

```
CREATE TABLE Patient (  
    Pat_id INT PRIMARY KEY,  
    PName VARCHAR(50),  
    PDiagnosis VARCHAR(100),  
    PAddress VARCHAR(100),  
    Hos_id INT,  
    FOREIGN KEY (Hos_id) REFERENCES Hospital(Hos_id)  
);  
  
CREATE TABLE Hospital (  
    Hos_id INT PRIMARY KEY,  
    HName VARCHAR(50),  
    HAddress VARCHAR(100),  
    HCity VARCHAR(50)  
);  
  
CREATE TABLE Doctor (  
    Doc_id INT PRIMARY KEY,  
    DName VARCHAR(50),  
    Qualification VARCHAR(50),  
    Salary DECIMAL(10,2),  
    Hos_id INT,  
    FOREIGN KEY (Hos_id) REFERENCES Hospital(Hos_id)  
);  
  
CREATE TABLE MedicalRecord (  
    Precord_id INT PRIMARY KEY,  
    Date_of_examination DATE,  
    Problem VARCHAR(100),  
    Pat_id INT,  
    FOREIGN KEY (Pat_id) REFERENCES Patient(Pat_id) );
```

2) For a given scenario, draw an EER diagram and show multiple entities and relationships

Scenario: Bank Management System

A bank has multiple branches. Each branch manages several accounts and issues multiple loans. Customers can hold multiple accounts and take multiple loans. Each account or loan can be shared by more than one customer.



Description: This ER diagram represents a **Banking System** with entities like Bank, Branch, Customer, Account, and Loan.

- A **Bank** (Code, Name, Addr) has multiple **Bank Branches**, each with a Branch_no and Addr.
- Each **Branch** can have multiple **Accounts** and **Loans** (1:N with both ACCTS and LOANS).
- **Account** (Acct_no, Balance, Type) and **Loan** (Loan_no, Amount, Type) are managed per branch.
- A **Customer** (San, Name, Phone, Addr) can hold multiple accounts (A_C M:N) and take multiple loans (L_C M:N).
- The many-to-many relationships between **Customer–Account** and **Customer–Loan** are managed through associative entities A_C and L_C.

Practical No.2

Aim: To perform basic Create, Read, Update, and Delete operations using MongoDB.

MongoDB is a NoSQL document-oriented database that stores data in JSON-like BSON documents. It allows fast development, horizontal scaling, and flexible schema design. The four fundamental operations used in any database system are CRUD : Create, Read, Update, and Delete.

1. Create / Switch to Database:

The use command is used to create a new database or switch to an existing one. In MongoDB, if the database does not exist, it will be created only after inserting some data.

Query:

```
use Practical2
```

Output:

```
test> use Practical2
switched to db Practical2
Practical2> |
```

2. Create Collection:

Collections in MongoDB are similar to tables in relational databases. This command creates a collection named students. However, MongoDB allows you to insert directly into a collection even if it doesn't exist; it will automatically create it on insertion.

Query:

```
db.createCollection("students")
```

Output:

```
Practical2> db.createCollection("students")
{ ok: 1 }
```

3. Insert Data (Create):

The insertMany() function is used to insert multiple documents at once into the students collection. Each document is a JSON-like object (BSON). This represents the Create operation in CRUD.

Query:

```
db.students.insertMany([
{name: "Asha", age: 21, course: "BDA"},
{name: "John", age: 22, course: "CS"},
{name: "Meena", age: 20, course: "IT"},
{name: "Rahul", age: 21, course: "Bio Tech"},
{name: "Shreenesh", age: 20, course: "BMS"},
{name: "Vishnu", age: 22, course: "IT"},
{name: "Akshay", age: 22, course: "Finance"},
{name: "Tejas", age: 19, course: "CS"}
])
```

Output:

```
Practical2> db.students.insertMany([
... {name: "Asha", age: 21, course: "BDA"},
... {name: "John", age: 22, course: "CS"},
... {name: "Meena", age: 20, course: "IT"},
... {name: "Rahul", age: 21, course: "Bio Tech"},
... {name: "Shreenesh", age: 20, course: "BMS"},
... {name: "Vishnu", age: 22, course: "IT"},
... {name: "Akshay", age: 22, course: "Finance"},
... {name: "Tejas", age: 19, course: "CS"}
... ])
...
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('688d976f00a5fecae4eec4a9'),
    '1': ObjectId('688d976f00a5fecae4eec4aa'),
    '2': ObjectId('688d976f00a5fecae4eec4ab'),
    '3': ObjectId('688d976f00a5fecae4eec4ac'),
    '4': ObjectId('688d976f00a5fecae4eec4ad'),
    '5': ObjectId('688d976f00a5fecae4eec4ae'),
    '6': ObjectId('688d976f00a5fecae4eec4af'),
    '7': ObjectId('688d976f00a5fecae4eec4b0')
  }
}
```

4. Read Data:

a) Fetch all documents:

find() retrieves all documents from the collection.

Query:

```
db.students.find()
```

Output:

```
Practical2> db.students.find()
[
  {
    _id: ObjectId('688d976f00a5fecae4eec4a9'),
    name: 'Asha',
    age: 21,
    course: 'BDA'
  },
  {
    _id: ObjectId('688d976f00a5fecae4eec4aa'),
    name: 'John',
    age: 22,
    course: 'CS'
  },
  {
    _id: ObjectId('688d976f00a5fecae4eec4ab'),
    name: 'Meena',
    age: 20,
    course: 'IT'
  },
  {
    _id: ObjectId('688d976f00a5fecae4eec4ac'),
    name: 'Rahul',
    age: 21,
    course: 'Bio Tech'
  },
  {
    _id: ObjectId('688d976f00a5fecae4eec4ad'),
    name: 'Shreenesh',
    age: 20,
    course: 'BMS'
  },
]
```

b) Filter documents:

Using a filter, you can narrow results (e.g., only students from BDA course)

Query:

```
db.students.find({course: "BDA"})
```

Output:

```
Practical2> db.students.find({course: "BDA"})
[
  {
    _id: ObjectId('688d976f00a5fecae4eec4a9'),
    name: 'Asha',
    age: 21,
    course: 'BDA'
  }
]
```

c) Project specific fields:

The projection part lets you control which fields to display. **{name: 1, _id: 0}** means only name will be shown, and _id will be excluded.

Query:

```
db.students.find({}, {name: 1, _id: 0})
```

Output:

```
Practical2> db.students.find({}, {name: 1, _id: 0})
[
  { name: 'Asha' },
  { name: 'John' },
  { name: 'Meena' },
  { name: 'Rahul' },
  { name: 'Shreenesh' },
  { name: 'Vishnu' },
  { name: 'Akshay' },
  { name: 'Tejas' }
]
```

4. Update Data:

updateOne() finds the first matching document and updates the specified fields. \$set is an update operator that modifies the value of a field. Asha's age will be updated from 21 to 22.

Query:

```
db.students.updateOne(  
  {name: "Asha"},  
  {$set: {age: 22}}  
)
```

Output:

```
Practical2> db.students.updateOne(  
... {name: "Asha"},  
... {$set: {age: 22}}  
... )  
...  
{  
  acknowledged: true,  
  insertedId: null,  
  matchedCount: 1,  
  modifiedCount: 1,  
  upsertedCount: 0  
}
```

5. Delete Data:

deleteOne() removes the first document that matches the filter condition. This command deletes the student document where name is "John".

Query:

```
db.students.deleteOne({name: "John"})
```

Output:

```
Practical2> db.students.deleteOne({name: "John"})  
{ acknowledged: true, deletedCount: 1 }
```

Practical No:03

Aim: To implement indexing and sorting (ordering) in MongoDB

1. Create Index:

- Creating an index in MongoDB helps speed up search queries by organizing data for faster access. It works like an index in a book, letting MongoDB find results without scanning every document. Indexes can be created on one or more fields in ascending or descending order.

```
db.students.createIndex({name: 1})
```

```
Practical2> db.students.createIndex({name: 1})  
name_1
```

2. View Existing Indexes:

- It is used to view all indexes on a MongoDB collection. It shows details like index fields, type (e.g., single-field, compound), and whether it's unique or default.

```
db.students.getIndexes()
```

```
Practical2> db.students.getIndexes()  
[  
  { v: 2, key: { _id: 1 }, name: '_id_' },  
  { v: 2, key: { name: 1 }, name: 'name_1' }  
]
```

3. Drop an Index:

- dropIndex() is used to delete a specific index from a MongoDB collection. This is useful if an index is no longer needed or is affecting write performance.

```
db.students.dropIndex({name: 1})
```

```
Practical2> db.students.dropIndex({name: 1})  
{ nIndexesWas: 2, ok: 1 }  
Practical2> |
```

4. Ordering Operations (Sorting):

- sort() in MongoDB is used to order query results based on one or more fields. You can sort in ascending (1) or descending (-1) order.

```
db.students.find().sort({age: 1})
```

```
Practical2> db.students.find().sort({age: 1})
[
  {
    _id: ObjectId('688d976f00a5fecae4eec4b0'),
    name: 'Tejas',
    age: 19,
    course: 'CS'
  },
  {
    _id: ObjectId('688d976f00a5fecae4eec4ab'),
    name: 'Meena',
    age: 20,
    course: 'IT'
  },
  {
    _id: ObjectId('688d976f00a5fecae4eec4ad'),
    name: 'Shreenesh',
    age: 20,
    course: 'BMS'
  },
  {
    _id: ObjectId('688d976f00a5fecae4eec4ac'),
    name: 'Rahul',
    age: 21,
    course: 'Bio Tech'
  },
]
```

```
db.students.find().sort({name: -1})
```

```
Practical2> db.students.find().sort({name: -1})
[
  {
    _id: ObjectId('688d976f00a5fecae4eec4ae'),
    name: 'Vishnu',
    age: 22,
    course: 'IT'
  },
  {
    _id: ObjectId('688d976f00a5fecae4eec4b0'),
    name: 'Tejas',
    age: 19,
    course: 'CS'
  },
  {
    _id: ObjectId('688d976f00a5fecae4eec4ad'),
    name: 'Shreenesh',
    age: 20,
    course: 'BMS'
  },
  {
    _id: ObjectId('688d976f00a5fecae4eec4ac'),
    name: 'Rahul',
    age: 21,
    course: 'Bio Tech'
  },
  {
    _id: ObjectId('688d976f00a5fecae4eec4ab'),
    name: 'Meena',
    age: 20,
    course: 'IT'
  },
]
```

5. Limit and Skip Results:

- limit() in MongoDB restricts the number of documents returned by a query.
- skip() is used to bypass a specified number of documents, useful for pagination.

```
db.students.find().sort({age: 1}).limit(2)
```

```
Practical2> db.students.find().sort({age: 1}).limit(2)
[
  {
    _id: ObjectId('688d976f00a5fecae4eec4b0'),
    name: 'Tejas',
    age: 19,
    course: 'CS'
  },
  {
    _id: ObjectId('688d976f00a5fecae4eec4ad'),
    name: 'Shreenesh',
    age: 20,
    course: 'BMS'
  }
]
```

```
db.students.find().sort({age: 1}).skip(1)
```

```
Practical2> db.students.find().sort({age: 1}).skip(1)
[
  {
    _id: ObjectId('688d976f00a5fecae4eec4ab'),
    name: 'Meena',
    age: 20,
    course: 'IT'
  },
  {
    _id: ObjectId('688d976f00a5fecae4eec4ad'),
    name: 'Shreenesh',
    age: 20,
    course: 'BMS'
  },
  {
    _id: ObjectId('688d976f00a5fecae4eec4ae'),
    name: 'Anshu',
    age: 21,
    course: 'IT'
  }
]
```

Practical No: 04

Aim: To perform Create, Read, Update, and Delete operations on a CouchDB database through the Futon web interface

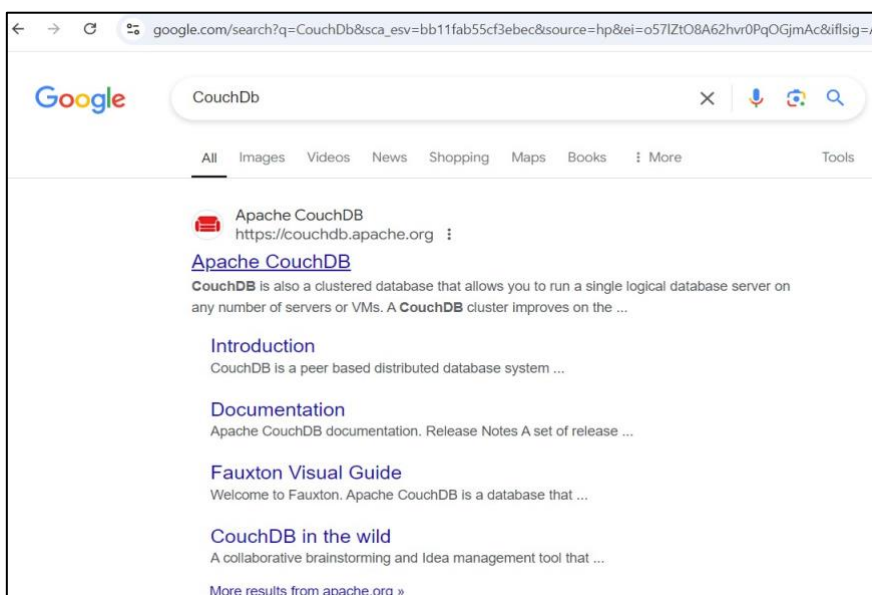
Description:

CouchDB is an open-source NoSQL database that stores data as flexible JSON documents. It uses HTTP and RESTful APIs to interact with the database, making it easy to work with over the web. CouchDB supports master-master replication, fault tolerance, and conflict resolution, making it great for distributed systems. It also includes a web-based UI called Fauxton for managing databases and documents.

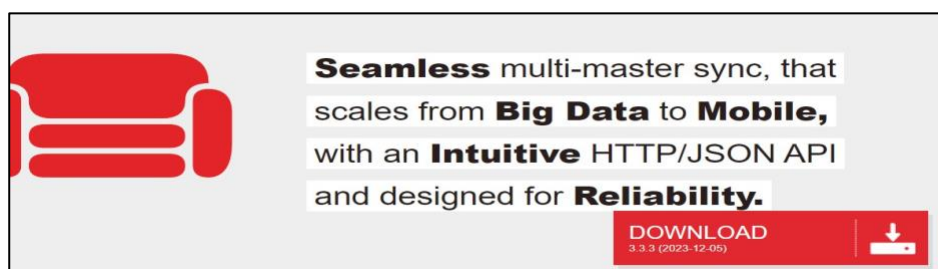
Step 1 : Install CouchDB

Search CouchDB on Google

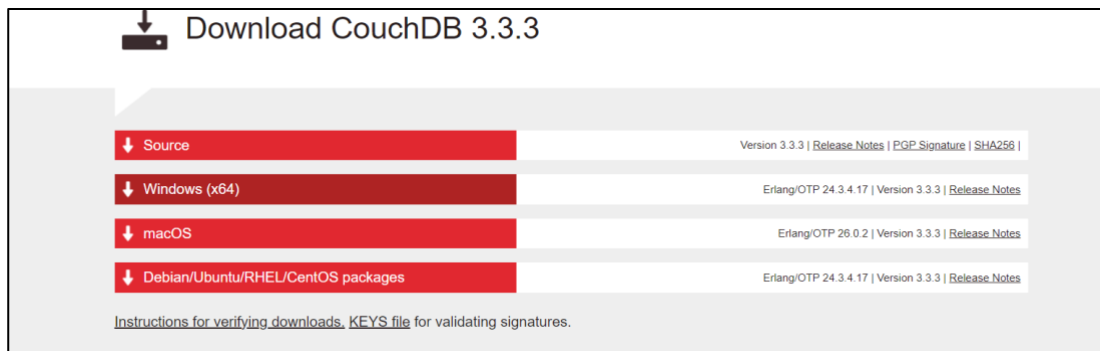
Click on the first link.



Click on the download button and download CouchDb



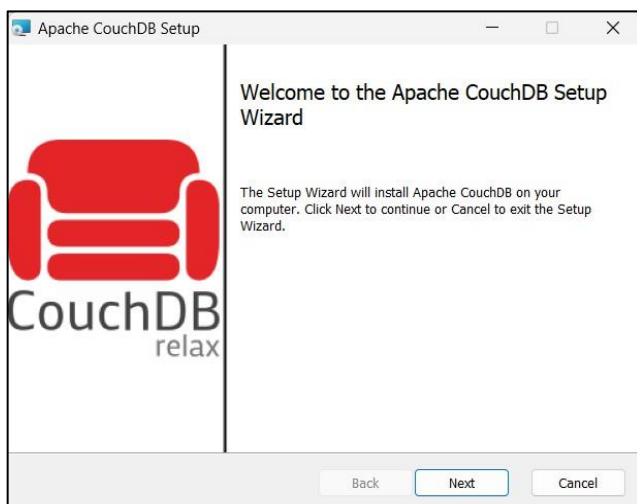
Select the appropriate version and click on the appropriate icon of your operating system.



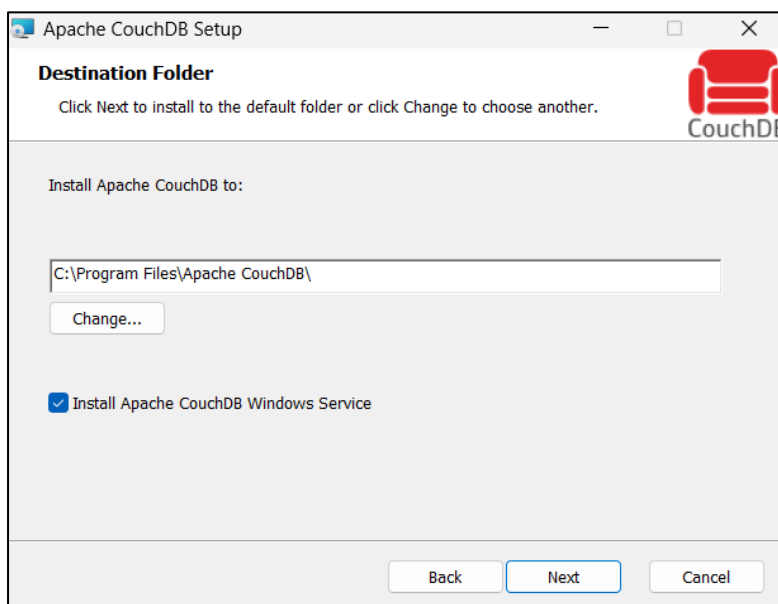
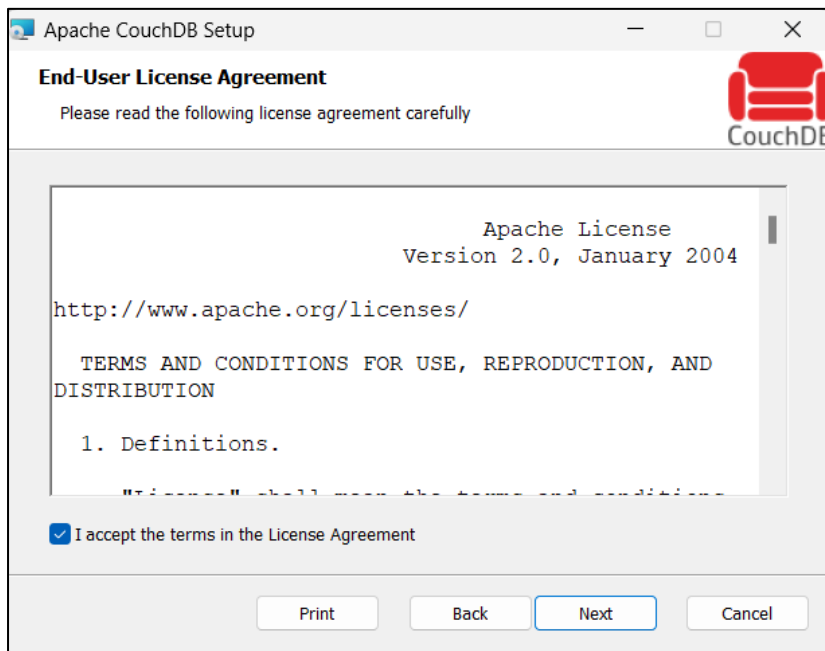
Click on Download CouchDb



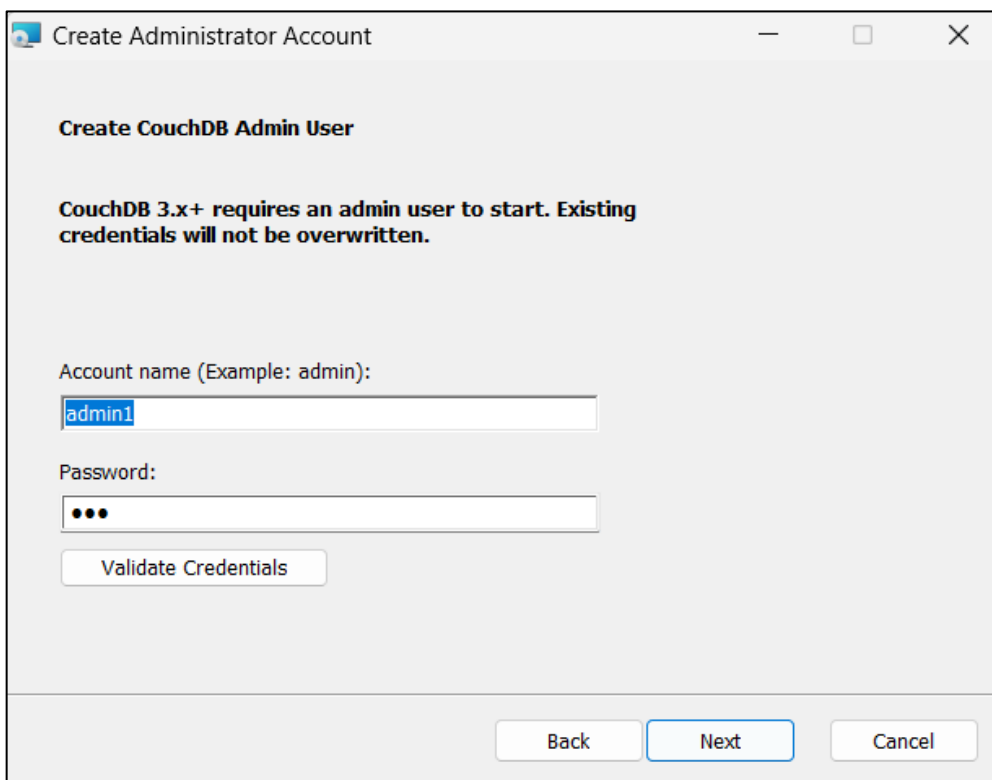
Open the file explorer of your Computer/Laptop. Go to Downloads and click on the windows install packager of couchdb



Click on next

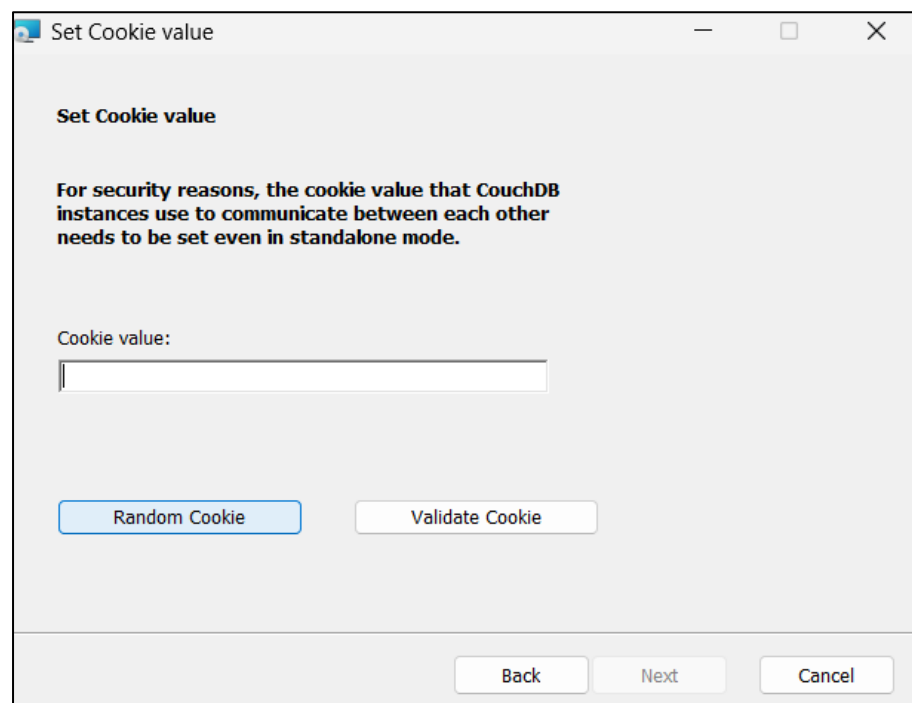


Create CouchDB Admin Account by giving account name and password



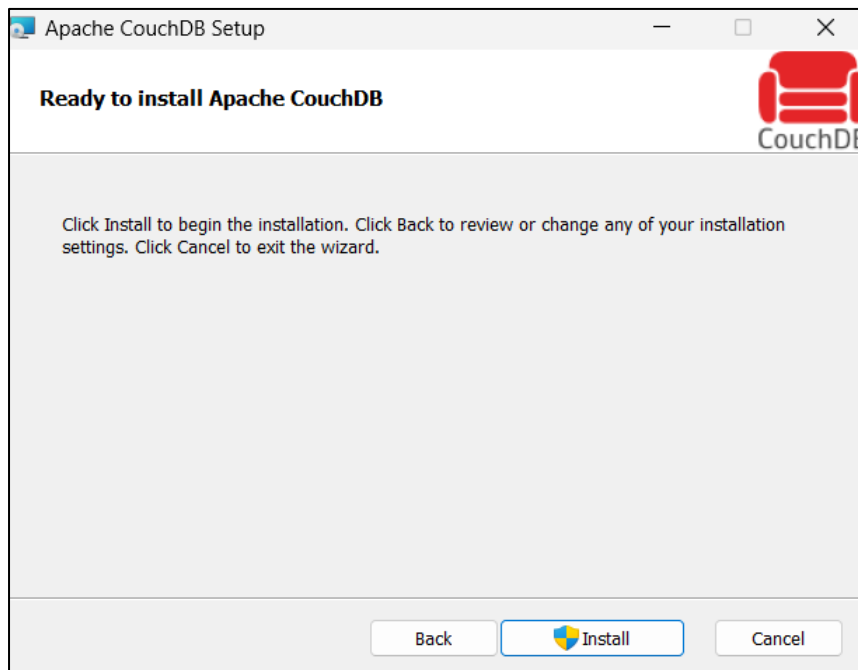
The screenshot shows a dialog box titled "Create Administrator Account". Inside, the heading is "Create CouchDB Admin User". Below this, a message states: "CouchDB 3.x+ requires an admin user to start. Existing credentials will not be overwritten." There are two input fields: "Account name (Example: admin):" with the text "admin1" entered, and "Password:" with three dots indicating a masked password. A "Validate Credentials" button is positioned below the password field. At the bottom of the dialog, there are three buttons: "Back", "Next" (which is highlighted with a blue border), and "Cancel".

Click on Random Cookie. It will randomly generate a cookie value

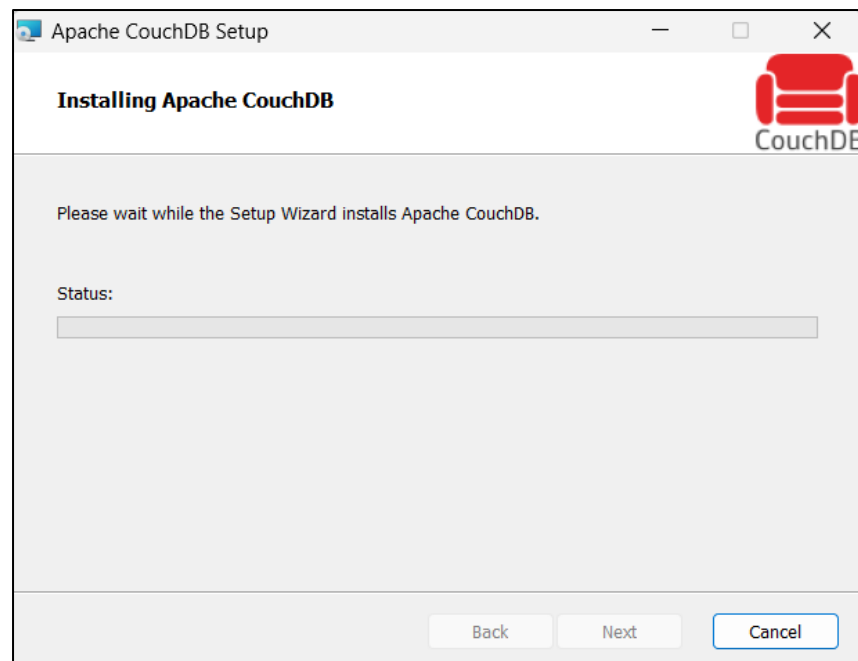


The screenshot shows a dialog box titled "Set Cookie value". Inside, the heading is "Set Cookie value". Below this, a message states: "For security reasons, the cookie value that CouchDB instances use to communicate between each other needs to be set even in standalone mode." There is a single input field labeled "Cookie value:". Below the input field, there are two buttons: "Random Cookie" (highlighted with a blue border) and "Validate Cookie". At the bottom of the dialog, there are three buttons: "Back", "Next", and "Cancel".

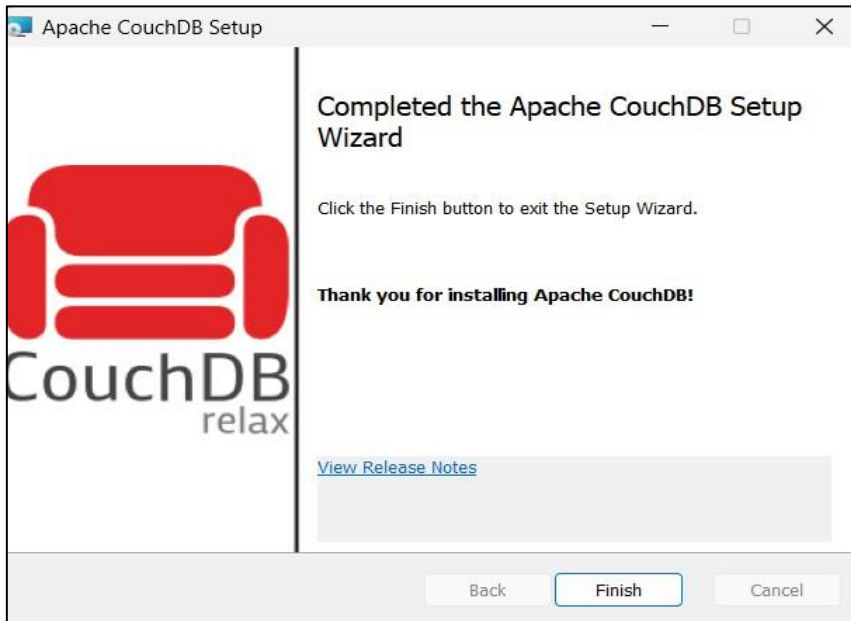
Click on Install



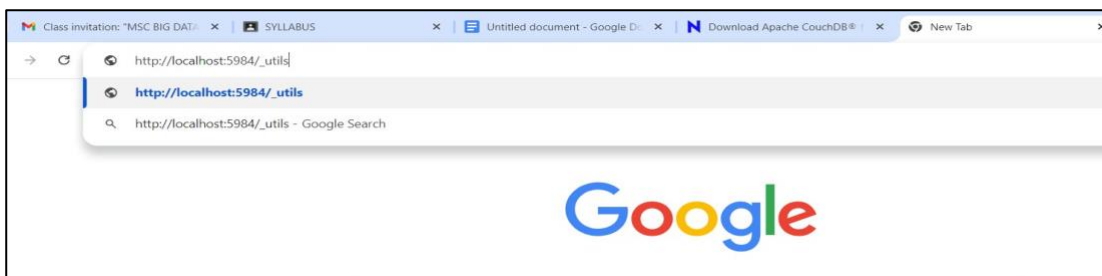
Let the system setup the wizard installs of Apache CouchDB



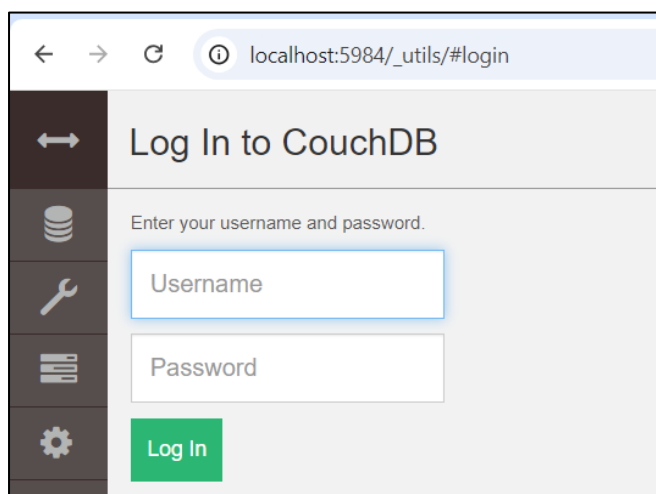
Click on Finish



Open Google. Type the url “http://localhost:5984/_utils” on the address bar.

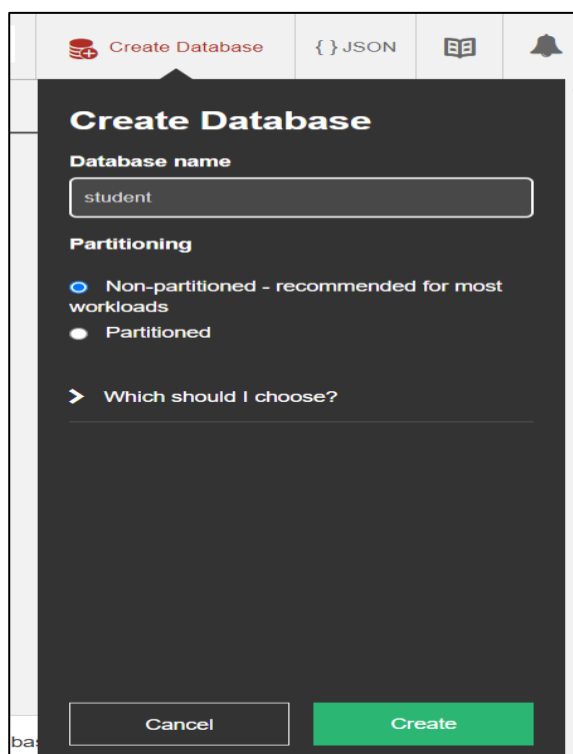


Login with the username and password which we had set during the installation.







Step 2: Click on “Create Database”

Give the database name and then click on create.

**Step 3: Create 5 Documents (CRUD: Create)**

Manually create 5 student documents using the Futon interface.

1. Click on the student database.
2. Click “+ New Document”
3. Click “Edit” and enter the following JSON fields:

 Options	 JSON		
<div>Create Document</div>			

Create 5 Documents (CRUD : Create)

```
{  
  "RollNo": 101,  
  "Name": "Alice",  
  "Dept": "CSE",  
  "Year": 2  
}
```


```
{  
  "RollNo": 102,  
  "Name": "Bob",  
  "Dept": "IT",  
  "Year": 1  
}
```

```
{  
  "RollNo": 103,  
  "Name": "Carol",  
  "Dept": "ECE",  
  "Year": 3  
}
```


```
{  
  "RollNo": 104,  
  "Name": "David",  
  "Dept": "MECH",  
  "Year": 4  
}
```

```
{  
  "RollNo": 105,  
  "Name": "Eva",  
  "Dept": "Civil",  
}
```


```
"Year": 2  
}
```

 Create Document Cancel

```
1 {  
2   "_id": "35150554d8a47b7311cd2894aa000dc8"  
3   "Roll no": 101,  
4   "Name": "ALice",  
5   "Dept": "CSE",  
6   "Year": 2  
7 }
```

 Create Document Cancel

```
1 {  
2   "_id": "35150554d8a47b7311cd2894aa00550b",  
3   "RollNo": 104,  
4   "Name": "David",  
5   "Dept": "MECH",  
6   "Year": 4  
7 }
```

 Create Document Cancel

```
1 {  
2   "_id": "35150554d8a47b7311cd2894aa00550b",  
3   "RollNo": 105,  
4   "Name": "Eva",  
5   "Dept": "Civil",  
6   "Year": 2  
7 }
```

☐

Table Metadata {} JSON

Create Document

	Dept	Name	RollNo	Year	_id
<input type="checkbox"/>	IT	Bob	102	1	35150554d8a47b7311cd289...
<input type="checkbox"/>	CSE	Alice	101	2	35150554d8a47b7311cd289...
<input type="checkbox"/>	ECE	Carol	103	3	35150554d8a47b7311cd289...
<input type="checkbox"/>	Civil	Eva	105	2	35150554d8a47b7311cd289...
<input type="checkbox"/>	MECH	David	104	4	35150554d8a47b7311cd289...

Step 4: Read Documents (CRUD: Read)

View and verify the inserted documents.

- Open the student database.
- Click on “All Documents”.
- Click on any document to view the contents.

Each document will display the fields entered in JSON format.

Save Changes Cancel

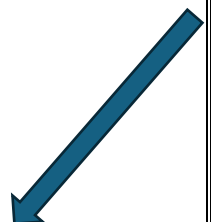
```
1 {  
2   "_id": "35150554d8a47b7311cd2894aa00199c",  
3   "_rev": "1-59a45cd0f47f07360aa3fc5bdcb6341e",  
4   "RollNo": 101,  
5   "Name": "Alice",  
6   "Dept": "CSE",  
7   "Year": 2  
8 }
```

Step 5: Update a Document (CRUD: Update)

Update RollNo 103 (Carol) to change the year from 3 to 4.

1. Click on the document for Carol (RollNo 103).
2. Click Edit.
3. Change the Year field from 3 to 4.
4. Click Save Document.

Document is updated and a new _rev value will be generated automacally.



Document ID

Document saved successfully.

Create Document

	Dept	Name	RollNo	Year	_id
<input type="checkbox"/>	IT	Bob	102	1	35150554d8a47b7311cd289...
<input type="checkbox"/>	CSE	Alice	101	2	35150554d8a47b7311cd289...
<input type="checkbox"/>	ECE	Carol	103	4	35150554d8a47b7311cd289...
<input type="checkbox"/>	Civil	Eva	105	2	35150554d8a47b7311cd289...
<input type="checkbox"/>	MECH	David	104	4	35150554d8a47b7311cd289...

Step 6: Delete a Specific Document (CRUD: Delete)

Delete the document of RollNo 104 (David).

1. Open the student database.
2. Click on the document where RollNo = 104.
3. Click the Delete (trash icon) on top-right.
4. Confirm the prompt: "Are you sure you want to delete this doc?"
5. Click Delete Document.

David's document is now removed. Only 4 documents should remain.

Document ID

Your document has been successfully deleted.

Create Document

	Dept	Name	RollNo	Year	_id
<input type="checkbox"/>	IT	Bob	102	1	35150554d8a47b7311cd289...
<input type="checkbox"/>	CSE	Alice	101	2	35150554d8a47b7311cd289...
<input type="checkbox"/>	ECE	Carol	103	4	35150554d8a47b7311cd289...
<input type="checkbox"/>	Civil	Eva	105	2	35150554d8a47b7311cd289...

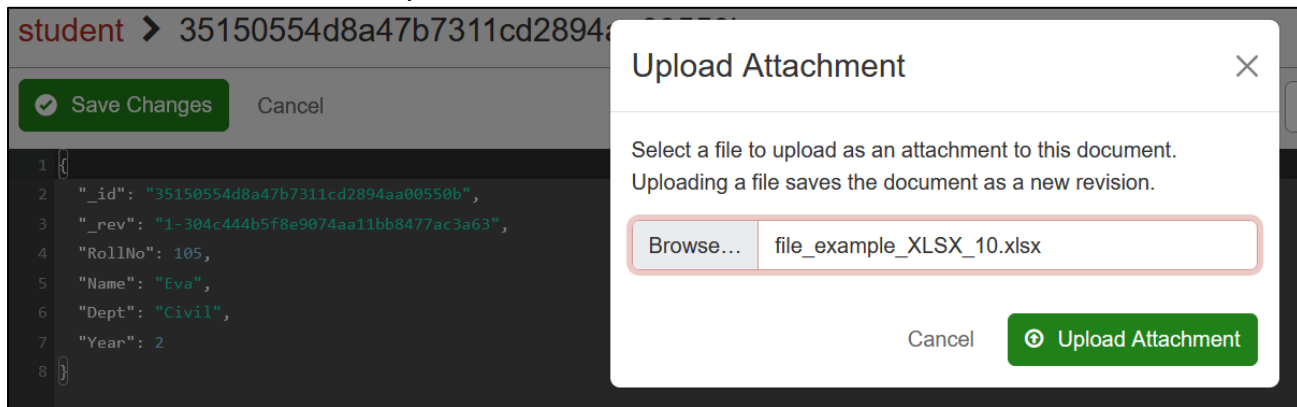
Step 7: Upload an Excel File to a Document (Attachments)

Attach a sample Excel file (e.g., student_marks.xlsx) to Roll No 105 (Eva).

1. Open the document for Roll No 105.
2. Click "Upload Attachment".
3. Browse and select the Excel file (e.g., student_marks.xlsx).
4. Click Upload.

5. The file will appear under the `_attachments` field.

You can now download or preview the attached Excel file



```
1 {
2   "_id": "35150554d8a47b7311cd2894aa00550b",
3   "_rev": "2-cbb8278359f82ba752d62575425fb924",
4   "RollNo": 105,
5   "Name": "Eva",
6   "Dept": "Civil",
7   "Year": 2,
8   "_attachments": {
9     "file_example_XLSX_10.xlsx": {
10       "content_type": "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet",
11       "revpos": 2,
12       "digest": "md5-gjs/eptHmS4Dn/RZsSG8Eg==",
13       "length": 5425,
14       "stub": true
15     }
16   }
17 }
```

PRACTICAL NO: 5

Aim: To demonstrate creation, insertion, update, deletion, and retrieval operations on a Cassandra database using CQL

Description:

- Apache Cassandra is a highly scalable, distributed NoSQL database designed to handle large amounts of data across many servers.
- It provides high availability with no single point of failure, making it ideal for mission-critical applications.
- Cassandra uses a peer-to-peer architecture and supports replication across multiple data centers.
- Its write-optimized design ensures fast performance for insert-heavy workloads.
- Data is stored in a column-family format, which allows flexible schema design. Popular use cases include real-time analytics, IoT data, and time-series applications.

Login and Database Setup

- ❖ Go to: <https://auth.cloud.datastax.com/>
- ❖ Sign in with your email ID.
- ❖ Create a new database:
 - **Database Name:** bigdata_db
 - **Keyspace Name:** bigdata
 - **Region**

Create database ESC X

Serverless (vector) ⓘ
An all-in-one database solution, optimized for Vector and Generative AI workloads

Serverless (non-vector)
A more traditional database solution without any of our new vector capabilities

Managed clusters ⓘ
A database with dedicated compute and storage, and advanced functionality—[upgrade for access](#)

Database name *
bigdata_db
Give it a memorable name – this can't be changed later.

Provider *
Google Cloud Amazon Web Services Microsoft Azure

Region *
us-east-2 X v

Cancel Create database

Open

CQL Console

Once the database is active, click "**CQL Console**" from the dashboard to execute your queries.

1. Use Keyspace

```
USE bigdata;
```

A **keyspace** in Cassandra is the top-level namespace that defines how data is stored on the cluster. It holds one or more tables and defines replication settings like the strategy and replication factor. Think of it as similar to a database in traditional RDBMS systems.

```
Connected to cndb at cassandra.ingress:9042.  
[cqlsh 6.8.0 | Cassandra 4.0.0.6816 | CQL spec 3.4.5 | Native protocol v4 | TLS]  
Use HELP for help.  
token@cqlsh> USE bigdata;  
token@cqlsh:bigdata> |
```

2. Create Table

```
CREATE TABLE users (  
  userid UUID PRIMARY KEY,  
  name TEXT,  
  age INT,  
  email TEXT,  
  city TEXT  
);
```

```
token@cqlsh:bigdata> CREATE TABLE users (  
  ... userid UUID PRIMARY KEY,  
  ... name TEXT,  
  ... age INT,  
  ... email TEXT,  
  ... city TEXT  
  ... );  
token@cqlsh:bigdata>
```

3.Insert Data (CRUD: Create)

Insert 5 user records into the table.

```
INSERT INTO users (userid, name, age, email, city)  
VALUES (2734f2e4-726e-445e-97d3-6f25a7f06afa, 'Alice', 25, 'alice@gmail.com',  
'Chennai');
```

```
INSERT INTO users (userid, name, age, email, city)  
VALUES (8e3503f2-1722-4d12-af25-a0859136ea85, 'Bob', 30, 'bob@gmail.com',  
'Mumbai');
```

```
INSERT INTO users (userid, name, age, email, city)  
VALUES (cee9efa1-2b6f-4555-a880-43b20783ae36, 'Carol', 28, 'carol@gmail.com',  
'Bangalore');
```

```
INSERT INTO users (userid, name, age, email, city)  
VALUES (943270f2-51e8-49c5-b0c6-e0dfcb9fca40, 'David', 35, 'david@gmail.com',  
'Delhi');
```

```
INSERT INTO users (userid, name, age, email, city)  
VALUES (915a4c1c-2d47-4e98-b394-29cd48e51b8f, 'Eva', 27, 'eva@gmail.com',  
'Hyderabad');
```

```
token@cqlsh:bigdata> INSERT INTO users (userid, name, age, email, city)
... VALUES (2734f2e4-726e-445e-97d3-6f25a7f06afa, 'Alice', 25, 'alice@gmail.com', 'Chennai');
token@cqlsh:bigdata> INSERT INTO users (userid, name, age, email, city)
... VALUES (8e3503f2-1722-4d12-af25-a0859136ea85, 'Bob', 30, 'bob@gmail.com', 'Mumbai');
token@cqlsh:bigdata> INSERT INTO users (userid, name, age, email, city)
... VALUES (cee9efa1-2b6f-4555-a880-43b20783ae36, 'Carol', 28, 'carol@gmail.com', 'Bangalore');
token@cqlsh:bigdata> INSERT INTO users (userid, name, age, email, city)
... VALUES (943270f2-51e8-49c5-b0c6-e0dfcb9fca40, 'David', 35, 'david@gmail.com', 'Delhi');
token@cqlsh:bigdata> INSERT INTO users (userid, name, age, email, city)
... VALUES (915a4c1c-2d47-4e98-b394-29cd48e51b8f, 'Eva', 27, 'eva@gmail.com', 'Hyderabad');
token@cqlsh:bigdata>
```

You may copy the UUID from one of the inserted rows to use in UPDATE/DELETE commands.

4. Display Records (CRUD: Read)

```
SELECT * FROM users;
```

```
token@cqlsh:bigdata> SELECT * FROM users;
```

userid	age	city	email	name
943270f2-51e8-49c5-b0c6-e0dfcb9fca40	35	Delhi	david@gmail.com	David
915a4c1c-2d47-4e98-b394-29cd48e51b8f	27	Hyderabad	eva@gmail.com	Eva
cee9efa1-2b6f-4555-a880-43b20783ae36	28	Bangalore	carol@gmail.com	Carol
2734f2e4-726e-445e-97d3-6f25a7f06afa	25	Chennai	alice@gmail.com	Alice
8e3503f2-1722-4d12-af25-a0859136ea85	30	Mumbai	bob@gmail.com	Bob

(5 rows)

```
token@cqlsh:bigdata>
```

5. Update Record (CRUD: Update)

```
UPDATE users
```

```
SET age = 31, city = 'Pune'
```

```
WHERE userid = 915a4c1c-2d47-4e98-b394-29cd48e51b8f;
```

```
token@cqlsh:bigdata> UPDATE users
... SET age = 31, city = 'Pune'
... WHERE userid = 915a4c1c-2d47-4e98-b394-29cd48e51b8f;
token@cqlsh:bigdata> SELECT * FROM users;
```

userid	age	city	email	name
943270f2-51e8-49c5-b0c6-e0dfcb9fca40	35	Delhi	david@gmail.com	David
915a4c1c-2d47-4e98-b394-29cd48e51b8f	31	Pune	eva@gmail.com	Eva
cee9efa1-2b6f-4555-a880-43b20783ae36	28	Bangalore	carol@gmail.com	Carol
2734f2e4-726e-445e-97d3-6f25a7f06afa	25	Chennai	alice@gmail.com	Alice
8e3503f2-1722-4d12-af25-a0859136ea85	30	Mumbai	bob@gmail.com	Bob

(5 rows)

6. Delete Record (CRUD: Delete)

Delete the record of user named **Carol** (replace UUID):

```
DELETE FROM users  
WHERE userid = cee9efa1-2b6f-4555-a880-43b20783ae36;
```

Confirm deletion:

```
SELECT * FROM users;
```

```
token@cqlsh:bigdata> DELETE FROM users  
... WHERE userid = cee9efa1-2b6f-4555-a880-43b20783ae36;  
token@cqlsh:bigdata> SELECT * FROM users;
```

userid	age	city	email	name
943270f2-51e8-49c5-b0c6-e0dfcb9fca40	35	Delhi	david@gmail.com	David
915a4c1c-2d47-4e98-b394-29cd48e51b8f	31	Pune	eva@gmail.com	Eva
2734f2e4-726e-445e-97d3-6f25a7f06afa	25	Chennai	alice@gmail.com	Alice
8e3503f2-1722-4d12-af25-a0859136ea85	30	Mumbai	bob@gmail.com	Bob

(4 rows)

7. Drop Table

```
DROP TABLE users;
```

```
token@cqlsh:bigdata> DROP TABLE users;  
token@cqlsh:bigdata> SELECT * FROM users;  
InvalidRequest: Error from server: code=2200 [Invalid query] message="table users does not exist"  
token@cqlsh:bigdata>
```

Practical No. 6

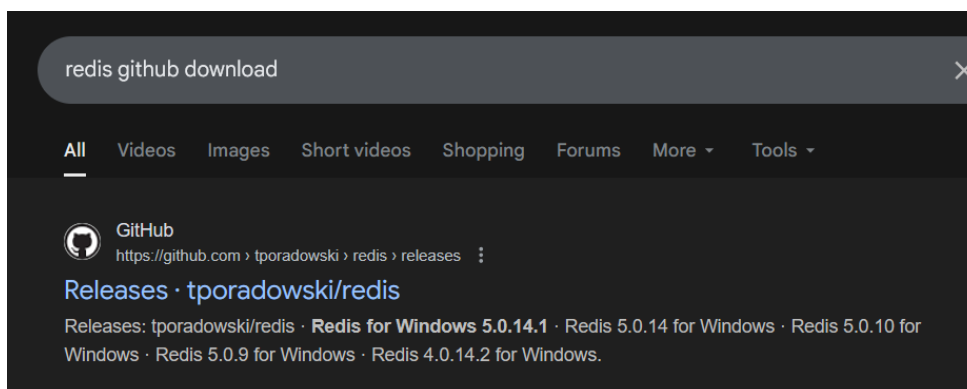
Aim: To perform various operations in Redis using CLI.

Description:

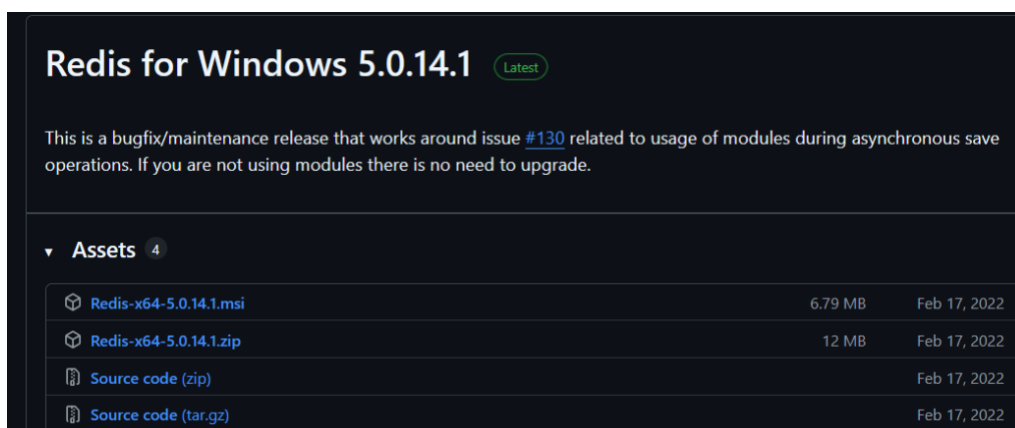
Redis is an open-source, in-memory data store that works as a database, cache, and message broker. It delivers extremely low latency since operations run directly in RAM. It supports rich data structures like strings, hashes, lists, sets, sorted sets, and streams. Persistence options let you save snapshots or logs to disk for durability. Common use cases include caching, real-time leaderboards, chat systems, and analytics. Its simplicity and speed make it a core tool for high-performance applications.

- Installation and Setup :

Step 1 : Search for “Redis Github Download” on any Browser. And Open the first link.



Step 2 : Download the .msi file



Step 3 : Run the msi file and finish the setup by just clicking next for every window within the redis setup wizard.



Step 4: Go to program files in the drive you installed Redis and open the Redis folder.

Step 5: Run the file “redis-cli” to open the cli of redis.



PING

Tests the connection with Redis server.

PING

```
127.0.0.1:6379> PING
PONG
127.0.0.1:6379> |
```

CRUD with SET and GET (Strings)

1. Create (Insert Records)

Store simple values like name, age, and course for 3 students.

```
SET student:101:name "Shrinesh"
SET student:101:course "BMS"

SET student:102:name "Shweta"
SET student:102:course "AI"

SET student:103:name "Kiran"
SET student:103:course "DBMS"
```



```
127.0.0.1:6379> SET student:101:name "Shrinesh"
OK
127.0.0.1:6379> SET student:101:course "BMS"
OK
127.0.0.1:6379> SET student:102:name "Shweta"
OK
127.0.0.1:6379> SET student:102:course "AI"
OK
127.0.0.1:6379> SET student:103:name "Kiran"
OK
127.0.0.1:6379> SET student:103:course "DBMS"
OK
```

2. Read (Retrieve Records)

```
GET student:101:name
GET student:102:course
```

```
127.0.0.1:6379> GET student:101:name
"Shrinesh"
127.0.0.1:6379> GET student:102:course
"AI"
```

3. Update (Modify Records)

```
SET student:103:course "Deep Learning"
```

```
127.0.0.1:6379> SET student:103:course "Deep Learning"
OK
127.0.0.1:6379> GET student:103:course
"Deep Learning"
```

4. Delete (Remove Data)

```
DEL student:102:course
DEL student:103:name
```

```
127.0.0.1:6379> DEL student:102:course
(integer) 1
127.0.0.1:6379> DEL student:103:name
(integer) 1
```

```
127.0.0.1:6379> GET student:103:name
(nil)
```

Limitation:

With SET/GET, each field needs a separate key (student:201:name, student:201:course).

For large records, this becomes hard to manage. We use **Hashes (HMSET, HGETALL)**.

1. Create (Insert 5 New Student Records)

HMSET student:201 id 201 name "Amit" course "Data Science"
HMSET student:202 id 202 name "Sneha" course "Artificial Intelligence"
HMSET student:203 id 203 name "Vikram" course "Machine Learning"
HMSET student:204 id 204 name "Pooja" course "Operating Systems"
HMSET student:205 id 205 name "Rohan" course "Computer Networks"

```
127.0.0.1:6379> HMSET student:201 id 201 name "Amit" course "Data Science"
OK
127.0.0.1:6379> HMSET student:202 id 202 name "Sneha" course "Artificial Intelligence"
OK
127.0.0.1:6379> HMSET student:203 id 203 name "Vikram" course "Machine Learning"
OK
127.0.0.1:6379> HMSET student:204 id 204 name "Pooja" course "Operating Systems"
OK
127.0.0.1:6379> HMSET student:205 id 205 name "Rohan" course "Computer Networks"
OK
```

2. Read (Retrieve Records)

Get all fields of student 201

HGETALL student:201

```
127.0.0.1:6379> HGETALL student:201
1) "id"
2) "201"
3) "name"
4) "Amit"
5) "course"
6) "Data Science"
```

Get only the name of student 202

HGET student:202 name

```
127.0.0.1:6379> HGET student:202 name
"Sneha"
```

Get multiple fields (id and course) of student 203

HMGET student:203 id course

```
127.0.0.1:6379> HMGET student:203 id course
1) "203"
2) "Machine Learning"
```

Show all keys (all students)

KEYS student:*

```
127.0.0.1:6379> KEYS student:*
1) "student:205"
2) "student:102:name"
3) "student:204"
4) "student:203"
5) "student:202"
6) "student:101:name"
7) "student:103:course"
8) "student:101:course"
9) "student:201"
```

3. Update (Modify Records)

Update course of student 201 to "Cloud Computing"

HSET student:201 course "Cloud Computing"

```
127.0.0.1:6379> HSET student:201 course "Cloud Computing"
(integer) 0
127.0.0.1:6379> HGET student:201 course
"Cloud Computing"
```

Change name of student 205 to "Rohan Sharma"

HSET student:205 name "Rohan Sharma"

```
127.0.0.1:6379> HSET student:205 name "Rohan Sharma"
(integer) 0
127.0.0.1:6379> HGET student:205 name
"Rohan Sharma"
```

4. Delete (Remove Data)

Remove the course field of student 204

HDEL student:204 course

```
127.0.0.1:6379> HDEL student:204 course
(integer) 1
127.0.0.1:6379> HGET student:204 course
(nil)
```

Delete full record of student 203

DEL student:203

```
127.0.0.1:6379> DEL student:203
(integer) 1
```

Verify deletion

EXISTS student:203

```
127.0.0.1:6379> EXISTS student:203  
(integer) 0
```

Common Utility Commands in Redis

1. KEYS

Lists all keys matching a pattern.

```
KEYS student:*
```

```
127.0.0.1:6379> KEYS student:*  
1) "student:205"  
2) "student:102:name"  
3) "student:204"  
4) "student:202"  
5) "student:101:name"  
6) "student:103:course"  
7) "student:101:course"  
8) "student:201"
```

2. EXISTS

Checks whether a key exists in Redis.

```
EXISTS student:201
```

```
127.0.0.1:6379> EXISTS student:201  
(integer) 1
```

3. TYPE

Shows what type of data is stored in a key (string, hash, list, set, etc.).

```
TYPE student:201
```

```
127.0.0.1:6379> TYPE student:201  
hash
```

4. EXPIRE

Set a time-to-live (in seconds) for a key, after which it will be deleted automatically.

```
EXPIRE student:202 60
```

```
127.0.0.1:6379> EXPIRE student:202 60  
(integer) 1
```

→ student:202 will be deleted after 60 seconds.

5. TTL (Time To Live)

Shows the remaining expiry time of a key in seconds.

```
TTL student:202
```

```
127.0.0.1:6379> TTL student:202  
(integer) 21
```

Practical No. 07

Aim: To implement MongoDB aggregation using the aggregate() method with operators such as SUM, AVG, MIN, MAX, push, addToSet, first, and last.

Description:

MongoDB uses a powerful aggregation framework to process data like a pipeline. Instead of row-by-row queries, it transforms documents through stages. Stages include \$match, \$group, \$project, \$sort, \$lookup, etc. This allows filtering, reshaping, and combining data efficiently. It works like SQL's GROUP BY + joins, but with JSON-style flexibility. Optimized for handling large datasets with complex transformations.

1) Create DB And Collection

```
use Stores
```

```
test> use Stores  
switched to db Stores  
Stores> |
```

```
db.createCollection("Sales")
```

```
Stores> db.createCollection("Sales")  
{ ok: 1 }
```

2) Inserting values

```
db.Sales.insert({"id":12,"item":"Soap","price":10,"Quantity":2,"date":ISODate("2014-01-01T08:00:00Z")})
```

```
db.Sales.insert({"id":13,"item":"Soap","price":20,"Quantity":2,"date":ISODate("2014-02-01T09:00:00Z")})
```

```

db.Sales.insert({"id":17,"item":"Shampoo","price":50,"Quantity":1,"date":ISODate("2015-01-01T09:00:00Z")})

db.Sales.insert({"id":18,"item":"Biscuit","price":90,"Quantity":4,"date":ISODate("2016-01-01T09:00:00Z")})

db.Sales.insert({"id":8,"item":"Toffee","price":80,"Quantity":7,"date":ISODate("2016-03-01T09:00:00Z")})

db.Sales.insert({"id":9,"item":"Facewash","price":100,"Quantity":1,"date":ISODate("2016-09-01T09:00:00Z")})

db.Sales.insert({"id":10,"item":"Facewash","price":100,"Quantity":1,"date":ISODate("2017-09-01T09:00:00Z")})

db.Sales.insert({"id":23,"item":"Toffee","price":50,"Quantity":5,"date":ISODate("2016-04-01T09:00:00Z")})

db.Sales.insert({"id":98,"item":"Biscuit","price":290,"Quantity":14,"date":ISODate("2016-05-01T09:00:00Z")})

db.Sales.insert({"id":27,"item":"Shampoo","price":100,"Quantity":2,"date":ISODate("2015-01-01T09:00:00Z")})

```

```

Stores> db.Sales.insert({"id":12,"item":"Soap","price":10,"Quantity":2,"date":ISODate("2014-01-01T08:00:00Z")})
DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany, or bulkWrite.
{
  acknowledged: true,
  insertedIds: { '_id': ObjectId('68d275389921ce69a0eec4a9') }
}
Stores> db.Sales.insert({"id":13,"item":"Soap","price":20,"Quantity":2,"date":ISODate("2014-02-01T09:00:00Z")})
... db.Sales.insert({"id":17,"item":"Shampoo","price":50,"Quantity":1,"date":ISODate("2015-01-01T09:00:00Z")})
... db.Sales.insert({"id":18,"item":"Biscuit","price":90,"Quantity":4,"date":ISODate("2016-01-01T09:00:00Z")})
... db.Sales.insert({"id":8,"item":"Toffee","price":80,"Quantity":7,"date":ISODate("2016-03-01T09:00:00Z")})
... db.Sales.insert({"id":9,"item":"Facewash","price":100,"Quantity":1,"date":ISODate("2016-09-01T09:00:00Z")})
... db.Sales.insert({"id":10,"item":"Facewash","price":100,"Quantity":1,"date":ISODate("2017-09-01T09:00:00Z")})
... db.Sales.insert({"id":23,"item":"Toffee","price":50,"Quantity":5,"date":ISODate("2016-04-01T09:00:00Z")})
... db.Sales.insert({"id":98,"item":"Biscuit","price":290,"Quantity":14,"date":ISODate("2016-05-01T09:00:00Z")})
... db.Sales.insert({"id":27,"item":"Shampoo","price":100,"Quantity":2,"date":ISODate("2015-01-01T09:00:00Z")})
{
  acknowledged: true,
  insertedIds: { '_id': ObjectId('68d275529921ce69a0eec4b2') }
}

```

3) find():

To select data from a collection in MongoDB, we can use the find() method.

```
db.Sales.find()
```

```
Stores> db.Sales.find()
[
  {
    _id: ObjectId('68d275389921ce69a0eec4a9'),
    id: 12,
    item: 'Soap',
    price: 10,
    Quantity: 2,
    date: ISODate('2014-01-01T08:00:00.000Z')
  },
  {
    _id: ObjectId('68d275529921ce69a0eec4aa'),
    id: 13,
    item: 'Soap',
    price: 20,
    Quantity: 2,
    date: ISODate('2014-02-01T09:00:00.000Z')
  },
]
```

The aggregate() Method

For the aggregation in MongoDB, you should use aggregate() method.

Syntax:

Basic syntax of aggregate() method is as follows –

```
>db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)
```

4)SUM:

\$sum: MongoDB \$sum returns the sum of numeric values

Syntax: { \$sum: <expression> }

Code:

```
db.Sales.aggregate([{$group:{_id:"$item",sumPrice:{$sum:"$price"}}}])
```

```
Stores> db.Sales.aggregate([{$group:{_id:"$item",sumPrice:{$sum:"$price"}}}])
[
  { _id: 'Biscuit', sumPrice: 380 },
  { _id: 'Toffee', sumPrice: 130 },
  { _id: 'Soap', sumPrice: 30 },
  { _id: 'Shampoo', sumPrice: 150 },
  { _id: 'Facewash', sumPrice: 200 }
]
```

5)Average:

\$avg: The MongoDB \$avg returns the average value of numeric values.

Syntax: { \$avg: <expression> }

Code:

```
db.Sales.aggregate([{$group:{_id:"$item",AVG:{$avg:"$price"}}}])
```

```
Stores> db.Sales.aggregate([{$group:{_id:"$item",AVG:{$avg:"$price"}}}])
[
  { _id: 'Biscuit', AVG: 190 },
  { _id: 'Toffee', AVG: 65 },
  { _id: 'Facewash', AVG: 100 },
  { _id: 'Soap', AVG: 15 },
  { _id: 'Shampoo', AVG: 75 }
]
```

6)Minimum:

\$min: The MongoDB \$min returns the minimum value.

Syntax: { \$min: <expression> }

Code:

```
db.Sales.aggregate([{$group:{_id:"$item",Min:{$min:"$price"}}}])
```

```
Stores> db.Sales.aggregate([{$group:{_id:"$item",Min:{$min:"$price"}}}])
[
  { _id: 'Biscuit', Min: 90 },
  { _id: 'Toffee', Min: 50 },
  { _id: 'Facewash', Min: 100 },
  { _id: 'Soap', Min: 10 },
  { _id: 'Shampoo', Min: 50 }
]
```

7)Maximum:

\$max: The MongoDB \$max returns the maximum value.

Syntax: { \$max: <expression> }

Code:

```
db.Sales.aggregate([{$group:{_id:"$item",Max:{$max:"$price"}}}])
```

```
Stores> db.Sales.aggregate([{$group:{_id:"$item",Max:{$max:"$price"}}}])
[
  { _id: 'Biscuit', Max: 290 },
  { _id: 'Toffee', Max: 80 },
  { _id: 'Facewash', Max: 100 },
  { _id: 'Shampoo', Max: 100 },
  { _id: 'Soap', Max: 20 }
]
```

8)addToSet

\$addToSet: The \$addToSet operator adds a value to an array unless the value is already present, in which case \$addToSet does nothing to that array.

Syntax: { \$addToSet: { <field1>: <value1>, ... } }

Code:

```
db.Sales.aggregate([{$group:{_id:"$item",Price:{$addToSet:"$price"},Quantity:{$addToSet:"$Quantity"}}}])
```



```
Stores> db.Sales.aggregate([{$group:{_id:"$item",Price:{$addToSet:"$price"},
Quantity:{$addToSet:"$Quantity"}}}])
[
  { _id: 'Biscuit', Price: [ 90, 290 ], Quantity: [ 4, 14 ] },
  { _id: 'Toffee', Price: [ 80, 50 ], Quantity: [ 5, 7 ] },
  { _id: 'Soap', Price: [ 20, 10 ], Quantity: [ 2 ] },
  { _id: 'Shampoo', Price: [ 100, 50 ], Quantity: [ 1, 2 ] },
  { _id: 'Facewash', Price: [ 100 ], Quantity: [ 1 ] }
]
```

```
db.Sales.aggregate([{$group:{_id:{Day:{$dayOfYear:"$date"},Year:{$year:"$date"}},Item_sold:{$addToSet:"$item"}}}])
```

```
Stores> db.Sales.aggregate([{$group:{_id:{Day:{$dayOfYear:"$date"},Year:{$year:"$date"}},Item_sold:{$addToSet:"$item"}}}])
[
  { _id: { Day: 1, Year: 2014 }, Item_sold: [ 'Soap' ] },
  { _id: { Day: 1, Year: 2015 }, Item_sold: [ 'Shampoo' ] },
  { _id: { Day: 32, Year: 2014 }, Item_sold: [ 'Soap' ] },
  { _id: { Day: 92, Year: 2016 }, Item_sold: [ 'Toffee' ] },
  { _id: { Day: 122, Year: 2016 }, Item_sold: [ 'Biscuit' ] },
  { _id: { Day: 245, Year: 2016 }, Item_sold: [ 'Facewash' ] },
  { _id: { Day: 244, Year: 2017 }, Item_sold: [ 'Facewash' ] },
  { _id: { Day: 1, Year: 2016 }, Item_sold: [ 'Biscuit' ] },
  { _id: { Day: 61, Year: 2016 }, Item_sold: [ 'Toffee' ] }
]
```

9)push

\$push: The \$push operator appends a specified value to an array

Syntax: { \$push: { <field1>: <value1>, ... } }

Code:

```
db.Sales.aggregate([{$group:{_id:"$item",Pushq:{$push:"$Quantity"}}}])
```

```
Stores> db.Sales.aggregate([{$group:{_id:"$item",Push:{$push:"$Quantity"}}}])
[
  { _id: 'Biscuit', Push: [ 4, 14 ] },
  { _id: 'Toffee', Push: [ 7, 5 ] },
  { _id: 'Facewash', Push: [ 1, 1 ] },
  { _id: 'Soap', Push: [ 2, 2 ] },
  { _id: 'Shampoo', Push: [ 1, 2 ] }
]
```

10)first

\$first: Returns the result of an expression for the first document in a group of documents. Only meaningful when documents are in a defined order

Syntax: { \$first: <expression> }

Code:

```
db.Sales.aggregate([{$group:{_id:"$item","Date":{$first:"$date"}}}])
```

```
Stores> db.Sales.aggregate([{$group:{_id:"$item","Date":{$first:"$date"}}}]
)
[
  { _id: 'Biscuit', Date: ISODate('2016-01-01T09:00:00.000Z') },
  { _id: 'Toffee', Date: ISODate('2016-03-01T09:00:00.000Z') },
  { _id: 'Facewash', Date: ISODate('2016-09-01T09:00:00.000Z') },
  { _id: 'Soap', Date: ISODate('2014-01-01T08:00:00.000Z') },
  { _id: 'Shampoo', Date: ISODate('2015-01-01T09:00:00.000Z') }
]
```

11)last

\$last: Returns the result of an expression for the last document in a group of documents. Only meaningful when documents are in a defined order.

Syntax: { \$last: <expression> }

Code:

```
db.Sales.aggregate([{$group:{_id:"$item","Date":{$last:"$date"}}}])
```

```
Stores> db.Sales.aggregate([{$group:{_id:"$item","Date":{$last:"$date"}}}])
[
  { _id: 'Biscuit', Date: ISODate('2016-05-01T09:00:00.000Z') },
  { _id: 'Toffee', Date: ISODate('2016-04-01T09:00:00.000Z') },
  { _id: 'Facewash', Date: ISODate('2017-09-01T09:00:00.000Z') },
  { _id: 'Soap', Date: ISODate('2014-02-01T09:00:00.000Z') },
  { _id: 'Shampoo', Date: ISODate('2015-01-01T09:00:00.000Z') }
]
```

Practical 8

Aim: Implementation of MapReduce using PySpark in Google Colab.

Description: MapReduce in PySpark is done with RDD transformations. `map()` applies a function to each element, creating key-value pairs if needed. `reduceByKey()` or `groupByKey()` then combines values with the same key. This mimics the map (parallel data prep) and reduce (aggregation) stages. Unlike Hadoop's MapReduce, PySpark runs in memory, so it's faster. You can chain multiple transformations and then call an action like `collect()` or `saveAsTextFile()`.

Install and import PySpark.

```
!pip install pyspark
```

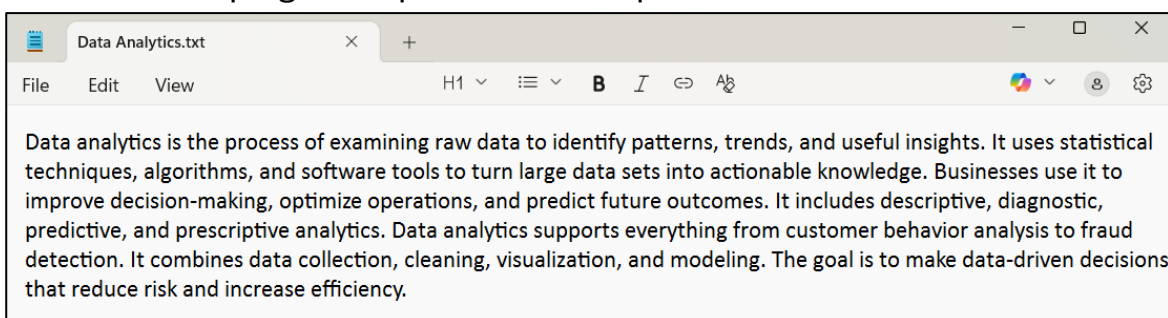
```
!pip install pyspark

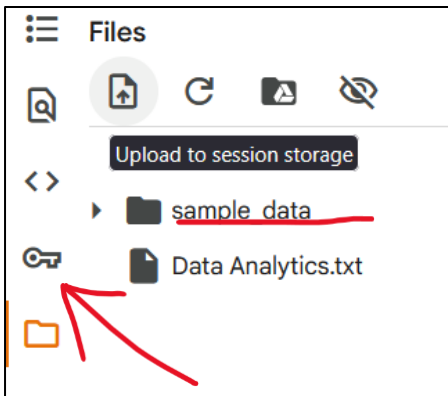
Requirement already satisfied: pyspark in /usr/local/lib/python3.12/dist-packages (3.5.1)
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.12/dist-packages (from pyspark) (0.10.9.7)
```

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[*]").appName("MapReduceExample").getOrCreate()
sc = spark.sparkContext
```

Upload your text file into Colab.

File Icon in top right -> upload icon -> upload the desired file.





```
Words = sc.textFile("Data Analytics.txt")
```

Split lines into words.

```
WordsCount = Words.flatMap(lambda line: line.split(" "))  
print("Total Words:", WordsCount.count())
```

```
Total Words: 84
```

Map each word into key-value pairs (word, 1).

```
WordPairs = WordsCount.map(lambda word: (word, 1))
```

```
WordPairs.take(10) # Display first 10 pairs
```

```
WordPairs = WordsCount.map(lambda word: (word, 1))  
WordPairs.take(10) # Display first 10 pairs
```

```
[('Data', 1),  
 ('analytics', 1),  
 ('is', 1),  
 ('the', 1),  
 ('process', 1),  
 ('of', 1),  
 ('examining', 1),  
 ('raw', 1),  
 ('data', 1),  
 ('to', 1)]
```

Reduce by key to count occurrences of each word.

```
DistinctWordsCount = WordPairs.reduceByKey(lambda a, b: a + b)  
print("Distinct Words:", DistinctWordsCount.count())
```

```
DistinctWordsCount = WordPairs.reduceByKey(lambda a, b: a + b)
print("Distinct Words:", DistinctWordsCount.count())
```

```
⇒ Distinct Words: 68
```

Collect the results.

```
DistinctWordsCount.collect()
```

```
DistinctWordsCount.collect()
```

```
⇒ [('analytics', 2),
    ('of', 1),
    ('examining', 1),
    ('raw', 1),
    ('to', 5),
    ('identify', 1),
    ('trends', 1),
    ('and', 6),
    ('useful', 1),
    ('It', 3),
    ('uses', 1),
    ('statistical', 1),
    ('turn', 1),
    ('large', 1),
    ('actionable', 1),
    ('use', 1),
    ('it', 1),
    ('optimize', 1),
    ('operations', 1),
    ('future', 1),
    ('outcomes.', 1),
    ('predictive', 1),
    ('prescriptive', 1),
    ('analytics.', 1),
    ('supports', 1),
    ('everything', 1),
    ('from', 1),
    ('behavior', 1),
```

Sort the words by frequency (top 5).

```
SortedWordsCount = DistinctWordsCount.map(lambda a: (a[1], a[0])).sortByKey(False)
```

```
SortedWordsCount.top(5)
```

```
SortedWordsCount = DistinctWordsCount.map(lambda a: (a[1], a[0])).sortByKey(False)
SortedWordsCount.top(5)
```

```
⇒ [(6, 'and'), (5, 'to'), (3, 'data'), (3, 'It'), (2, 'is')]
```