

Data Management, Warehousing, And Analytics (Summer 2023)

Assignment 2

Submitted by: Arihant Dugar (B00917961)

GitLab repo:

https://git.cs.dal.ca/dugar/csci5408_s23_b00917961_arihant_dugar/tree/main/Assignment2

Problem 1:

Summary:

Today's interconnected world makes distributed systems more relevant than ever[1]. This paper emphasizes the increasing feasibility of integrating database and networking technologies to create efficient distributed database systems. Distributed database design can be divided into three phases: initial design, redesign, and materialization. The research paper provides an overview of fragmentation techniques and correctness rules for fragmented data.

This research paper's related work section highlights numerous studies and techniques in the topic of distributed database systems. In order to maximise query evaluation locally while minimising communication costs, Seung-Jin Lim suggested fragmentation and allocation algorithms in a distributed deductive database system. Van Nghia Luong presented an algorithm for horizontal and vertical fragmentation based on knowledge-oriented clustering techniques, which reduced execution time and data fragmentation. Akash Kumar Patel examined various fragmentation methods and emphasised the importance of accurate and efficient approaches. Bhuyar demonstrated a fragmentation technique that may be applied at various stages of distributed systems to efficiently solve early fragmentation difficulties. Nicoleta-Magdalena Iacob investigated data allocation and replication costs to improve distributed database design. Adrian Runceanu created an objective function-based heuristic technique for vertical fragmentation. In heterogeneous network environments, HABABEH proposed a simple approach for fragmentation, allocation, and replication of data. To support this methodology, Navathe presented algorithms for generating vertical and horizontal fragmentation schemes.

The research paper focuses on fragmentation strategies to explore the major challenges in creating a distributed database system. Fragmentation is the process of splitting classes or relations into smaller segments while maintaining the integrity of the original database. The fragmentation types covered include horizontal (HF), vertical (VF), and mixed (MF). HF divides a relation into distinct tuples that are stored at different nodes, which improves reliability, performance, and storage efficiency. VF divides a relation into sets of columns, each of which has the main key property. This improves processing functions across several sites. MF is a combination of HF and VF that allows for more complicated fragmentation

methods. The paper discusses the benefits and drawbacks of each type of fragmentation, emphasising the necessity of local query optimisation and addressing unique application needs in distributed database design.

The division of objects into fragments stored across a network is a critical feature of distributed database design. This research investigates three types of fragmentation strategies: horizontal, vertical, and mixed. Horizontal fragmentation splits a table into distinct tuples, allowing each site to keep relevant data for speedier queries. Vertical fragmentation is the process of dividing a table into groupings of columns that share at least one column for rebuilding. Mixed fragmentation employs both horizontal and vertical methodologies to produce subsets of rows and characteristics. Completeness (ensuring that all data items are present in fragments), reconstruction (ability to reassemble the entire table), and disjointness (no duplication of data items across fragments) are all defined as correctness rules for fragmentation. There is a comparison of fragmentation tactics presented, highlighting their conditions and capabilities.

The concept of data fragmentation, including its forms, is introduced, and fragmentation solutions for each fragment of a relation are compared. Each fragmentation type's merits and cons are discussed. The study finishes by emphasising this work's potential utility for new researchers interested in working with distributed databases.

Scope of Improvement:

Some technical improvements that could be considered for the paper:

- **Enhance Efficiency:** Ways to improve the performance of fragmentation and allocation processes. This can involve studying algorithms or methods that reduce communication costs, balance the workload, or enhance the overall execution of queries in distributed systems.
- **Ensuring System Reliability:** Strategies or methods to handle failures or network disruptions in a distributed database system. This can involve discussing mechanisms for data replication, maintaining consistency, and recovering from potential issues. By addressing these aspects, the system can become more robust and reliable.
- **Ensuring Data Security:** Paying attention to the security aspects of the distributed database system, particularly regarding data fragmentation and allocation. Methods like data encryption, access control, and privacy protection to safeguard the confidentiality and integrity of the distributed data[4].
- **Dynamic Node Management:** Examining the fragmentation techniques adapt to changes in the distributed system, such as the addition or removal of nodes. Mechanisms for redistributing fragments, rebalancing workloads, and ensuring efficient utilization of resources during dynamic node management[2].

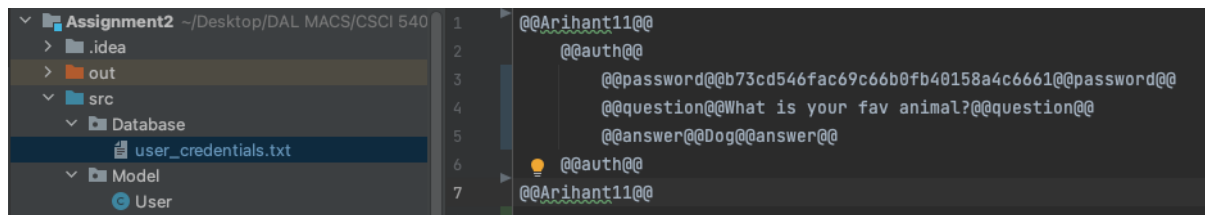
Problem 2:

Creating a new user :

```
Select the operation:
1. Login
2. Register
3. Exit
2
Enter user name
Arihant11
Enter password
Arihant@123
Enter security question
What is your fav animal?
Enter security answer
Dog
User registered successfully.
```

Fig 2.1: User registration for database access

Once the user is created, the data is stored in the Database folder in the file **user_credentials.txt** in the below format:



The screenshot shows a file explorer on the left with the following structure:

- Assignment2 ~./Desktop/DAL MACS/CSCI 540
 - .idea
 - out
 - src
 - Database
 - user_credentials.txt
 - Model
 - User

The main editor window displays the content of **user_credentials.txt** with line numbers 1 through 8:

```
1 @@Arihant11@@
2 @@auth@@
3 @@password@@b73cd546fac69c66b0fb40158a4c6661@@password@@
4 @@question@@What is your fav animal?@@question@@
5 @@answer@@Dog@@answer@@
6 @@auth@@
7 @@Arihant11@@
8
```

Fig 2.2: User data stored in user_credentials.txt

The password is hashed and stored in the file for security reasons.

Attempt to login:

```

Select the operation:
1. Login
2. Register
3. Exit
1
Enter user name|
Arihant11
Enter password
Arihant@123
Please answer the below security question -
What is your fav animal?
Dog
Successfully logged in
Select the operation:
1. Create a Database
2. Execute Query
3. Generate ER Diagram
4. Transaction
5. Logout & Exit

```

Fig 2.3: User Authentication steps and output

Once the user is logged in, he can create a database, execute a query, create an ER diagram for the relational mapping and also perform transaction operations.

Creating a database :

```

Select the operation:
1. Create a Database
2. Execute Query
3. Generate ER Diagram
4. Transaction
5. Logout & Exit
1
Enter the database name to |create
RestaurantManager
Successfully created database

```

Fig 2.4: Successful database creation for a user

Once the database is created, there is a new directory inside **Database** directory named **username.databaseName**, in our case it is **Arihant11.RestaurantBuilder** . And there is an entry for the user database inside the **user_credentials.txt** for maintaining the user data. See the below for reference :



Fig 2.5: Storage design for user database

Every user is **limited to one database** as per the requirement. So when creating another database, we get the following error:

```
Select the operation:
1. Create a Database
2. Execute Query
3. Generate ER Diagram
4. Transaction
5. Logout & Exit
1
Enter the database name to create
Test
Each user is limited to one database only
```

Fig 2.6: Output depicting that each user can create only one database

Design of Storage:

The storage architecture is designed to utilize files as the underlying data storage mechanism, with the flexibility to customize the delimiter used for data separation. The delimiter, which is set as "@@", can be easily modified and is globally declared within classes, allowing for easy configuration and adaptability to specific requirements.

And the storage structure is mentioned above when creating a database and will dive deep when exploring the table creation and storing table data.

Queries implementation:

CREATE

Created two tables with the fields and key relationships. The below screenshot shows the query execution.

```
Select the operation:
1. Create a Database
2. Execute Query
3. Generate ER Diagram
4. Transaction
5. Logout & Exit
>
Enter the query to execute
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(100)
);

Select the operation:
1. Create a Database
2. Execute Query
3. Generate ER Diagram
4. Transaction
5. Logout & Exit
>
Enter the query to execute
CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
);
```

Fig 2.7: Execution of CREATE statements

Once the queries are executed, we see 3 generated files – Customers.txt, Orders.txt and table_desc.txt

The Customers.txt and Orders.txt contains table headers or column names separated by delimiter specified. Whereas, the table_desc.txt contains all key information's of the table retrieved from the query. Below is the screenshot for reference :

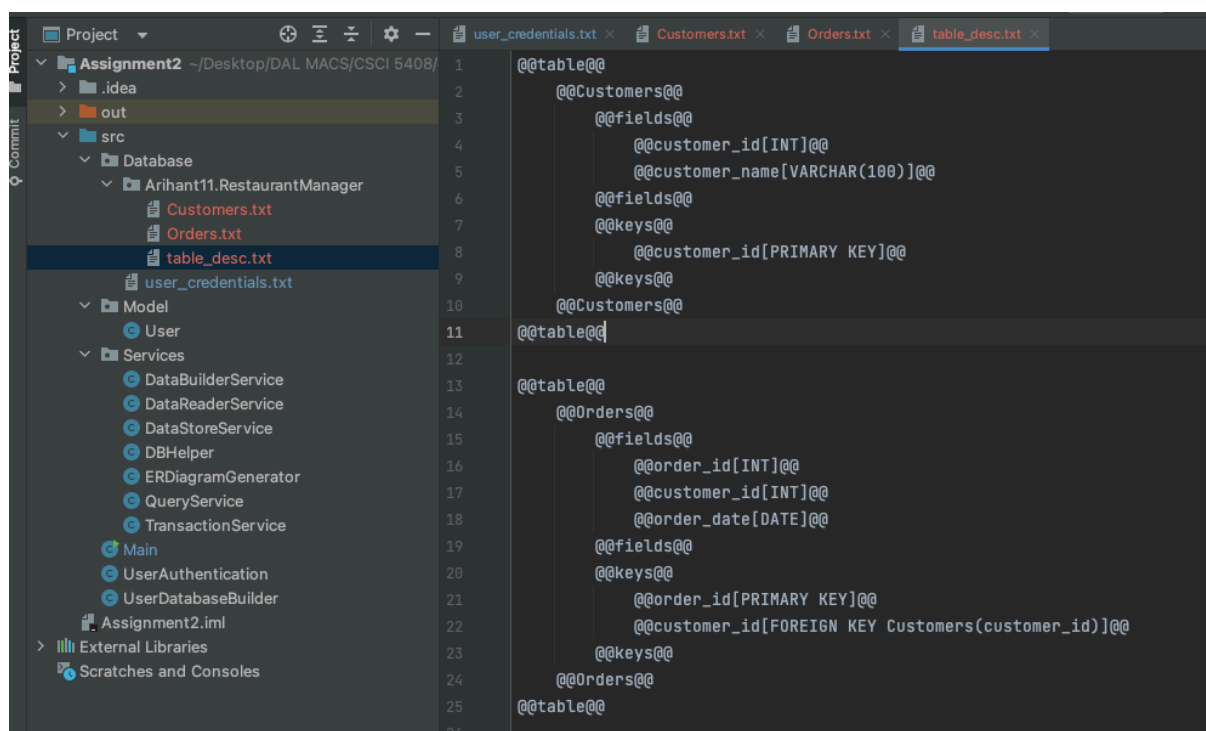
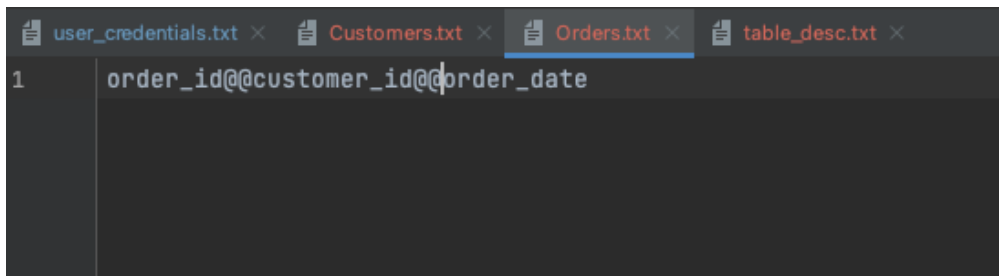


Fig 2.8: File structure and table information storage design



order_id@@customer_id@@order_date

Fig 2.9: Table headers when the table is created

INSERT

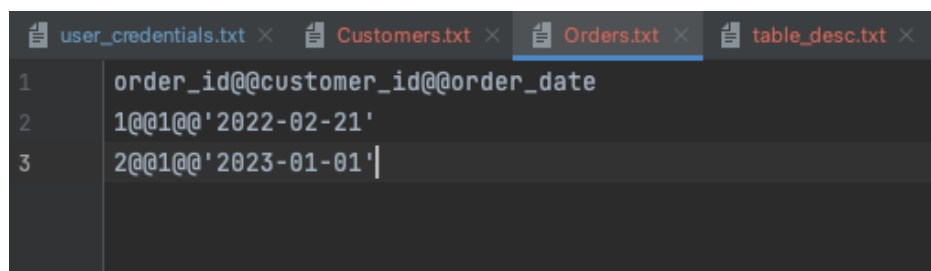
Inserting some records for Orders in the table:

```
Select the operation:
1. Create a Database
2. Execute Query
3. Generate ER Diagram
4. Transaction
5. Logout & Exit
2
Enter the query to execute
INSERT INTO Orders (order_id, customer_id, order_date) VALUES (1, 1, '2022-02-21')

Select the operation:
1. Create a Database
2. Execute Query
3. Generate ER Diagram
4. Transaction
5. Logout & Exit
2
Enter the query to execute
INSERT INTO Orders (order_id, customer_id, order_date) VALUES (2, 1, '2023-01-01')
```

Fig 2.10: Execution of INSERT statements

One the query is executed, we can see the data added to the Orders.txt



order_id@@customer_id@@order_date
1@@1@@'2022-02-21'
2@@1@@'2023-01-01'

Fig 2.11: Table data after successful insertion

SELECT

The select query can be used to fetch data from tables and show the result displayed in tabular format in the console. See the below image for reference:

```
Select the operation:
1. Create a Database
2. Execute Query
3. Generate ER Diagram
4. Transaction
5. Logout & Exit
2
Enter the query to execute
SELECT * FROM Orders
```

order_id	customer_id	order_date
1	1	'2022-02-21'
2	1	'2023-01-01'

Fig 2.12: Execution of SELECT statements to get all data

We can also select specific columns while performing the SELECT operation :

```
Select the operation:
1. Create a Database
2. Execute Query
3. Generate ER Diagram
4. Transaction
5. Logout & Exit
2
Enter the query to execute
SELECT order_id, order_date FROM Orders
```

order_id	order_date
1	'2022-02-21'
2	'2023-01-01'

Fig 2.13: Execution of SELECT statements when filtered only for selected fields

UPDATE

The update can be done using a specified condition provided for **WHERE** clause.


```
Select the operation:
1. Create a Database
2. Execute Query
3. Generate ER Diagram
4. Transaction
5. Logout & Exit
2
Enter the query to execute
UPDATE Orders
SET order_date = '2023-01-02'
WHERE customer_id = 1

Update successful.
```

Fig 2.14: Execution of UPDATE statements

Once the update is successful, you can see the data in the file changes. Below is the screenshot for reference:

The screenshot shows a database application interface. At the top, a list of operations is displayed: 1. Create a Database, 2. Execute Query, 3. Generate ER Diagram, 4. Transaction, and 5. Logout & Exit. Option 2, 'Execute Query', is selected. Below this, a prompt 'Enter the query to execute' is followed by the SQL statement: `SELECT * FROM Orders;`. The application then displays the results of the query in a table format.

order_id	customer_id	order_date
1	1	'2023-01-02'
2	1	'2023-01-02'

Fig 2.15: Table data after successful UPDATE

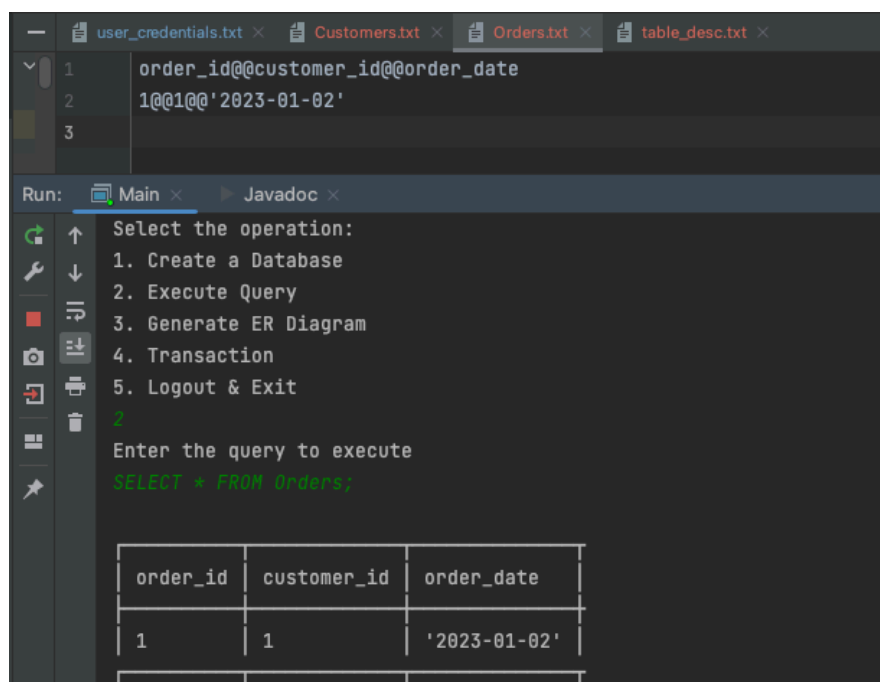
DELETE

The table data can be deleted based on the condition provided for the **WHERE** clause:

```
Select the operation:
1. Create a Database
2. Execute Query
3. Generate ER Diagram
4. Transaction
5. Logout & Exit
2
Enter the query to execute
DELETE FROM Orders WHERE order_id = 2
Delete successful.
```

Fig 2.16: Execution of DELETE statements

Once the delete is successful, the data is no longer present in the file and it can be verified as below:



The screenshot shows a Java application window with several tabs: 'user_credentials.txt', 'Customers.txt', 'Orders.txt', and 'table_desc.txt'. The 'Orders.txt' tab is active, displaying a table with three columns: 'order_id', 'customer_id', and 'order_date'. The table contains one row of data: '1', '1', and '2023-01-02'. Below the table, there is a 'Run' button and a 'Main' tab. The 'Main' tab is active, displaying a menu with five options: '1. Create a Database', '2. Execute Query', '3. Generate ER Diagram', '4. Transaction', and '5. Logout & Exit'. The '2. Execute Query' option is selected, and the text 'Enter the query to execute' is displayed. Below this, the query 'SELECT * FROM Orders;' is entered. The output of the query is displayed as a table with three columns: 'order_id', 'customer_id', and 'order_date'. The table contains one row of data: '1', '1', and '2023-01-02'.

order_id	customer_id	order_date
1	1	'2023-01-02'

Fig 2.17: Table data after successful DELETE

Transactions :

The transaction handling logic is implemented in such a way that, if the transaction is not committed then the operations are not performed and the data is not modified. On commit only the data would be saved to the database, and on rollback it will be not saved.

Below is the example of a transaction operation :

```
Select the operation:
1. Create a Database
2. Execute Query
3. Generate ER Diagram
4. Transaction
5. Logout & Exit
4
BEGIN TRANSACTION;
SELECT * FROM Orders;
COMMIT;
END TRANSACTION;
```

order_id	customer_id	order_date
1	1	'2023-01-02'

Fig 2.18: Execution of Transactional statements with commit

When the transaction is not committed then the query is not executed:

```
Select the operation:
1. Create a Database
2. Execute Query
3. Generate ER Diagram
4. Transaction
5. Logout & Exit
4
BEGIN TRANSACTION;
UPDATE Orders
SET order_date = '2023-11-11'
WHERE customer_id = 1;
ROLLBACK;
END TRANSACTION;
```

Fig 2.19: Execution of Transactional statements with Rollback

In the above case the order_date is not modified since it is rolled back.

ER Diagram:

The diagram generated represents the relational mapping for the tables, this has limited functionality but depicts the relationship between tables. Below is the result of the generated ERD based on the key constraints.



Fig 2.20: ERD for the Orders and Customers relational mapping using the keys specified during table creation.

The above diagram depicts relationship between the two tables and the columns in the tables in a tabular format.

SOLID Principles use case:

Single Responsibility Principle :

The code is structured and functionalities are designed in such a way that each perform a single responsibility[3].

Example:

DataBuilderService is used to perform operations related to data building before storage.

DataStoreService is used to perform operations related to data storage.

DataReaderService is used to perform data read operations from files.

ERDiagramGenerator is used to generate ER diagram and it only performs a single responsibility.

Open-Closed Principle :

The **QueryService** and **TransactionService** are designed in such a way that they are open for extension for closed for modification.

Going forward more queries can be added and it is easily extensible.

Dependency Inversion principle:

The system architecture follows the principle of dependency inversion, where high-level modules are decoupled from low-level modules, and dependencies are based on abstractions rather than concrete details. This design approach promotes modularity, flexibility, and maintainability by allowing modules to rely on interfaces or abstract classes rather than specific implementations.

References:

- [1] A. H. Al-Sanhani, A. M. Hamdan, A. Al-Thaher, and A. Al-Dahoud, "A comparative analysis of data fragmentation in distributed database," May 2017. [Online]. Available: <https://doi.org/10.1109/icitech.2017.8079934> (Accessed Jul. 12, 2023).
- [2] Azzam Sleit, Wesam Almobaideen, Samih Alareqi, and Abdulaziz Al-Nahari, "A Dynamic Object Fragmentation and Replication Algorithm In Distributed Database Systems," ResearchGate, Aug. 2007.
https://www.researchgate.net/publication/26459633_A_Dynamic_Object_Fragmentation_and_Replication_Algorithm_In_Distributed_Database_Systems (Accessed Jul. 12, 2023).
- [3] S. Millington, "A Solid Guide to SOLID Principles | Baeldung," Baeldung, Feb. 05, 2019. <https://www.baeldung.com/solid-principles> (Accessed Jul. 12, 2023).
- [4] S. Subashini and V. Kavitha, "Kavitha, V.: A Survey on Security Issues in Service Delivery Models of Cloud Computing. Journal of Network...", ResearchGate, Jan. 31, 2011.
https://www.researchgate.net/publication/222675501_Kavitha_V_A_Survey_on_Security_Issues_in_Service_Delivery_Models_of_Cloud_Computing_Journal_of_Network_and_Computer_Application_341_1-11 (Accessed Jul. 12, 2023).