

# **7086 CEM**

## **Data Management Systems**

### **Course Work**

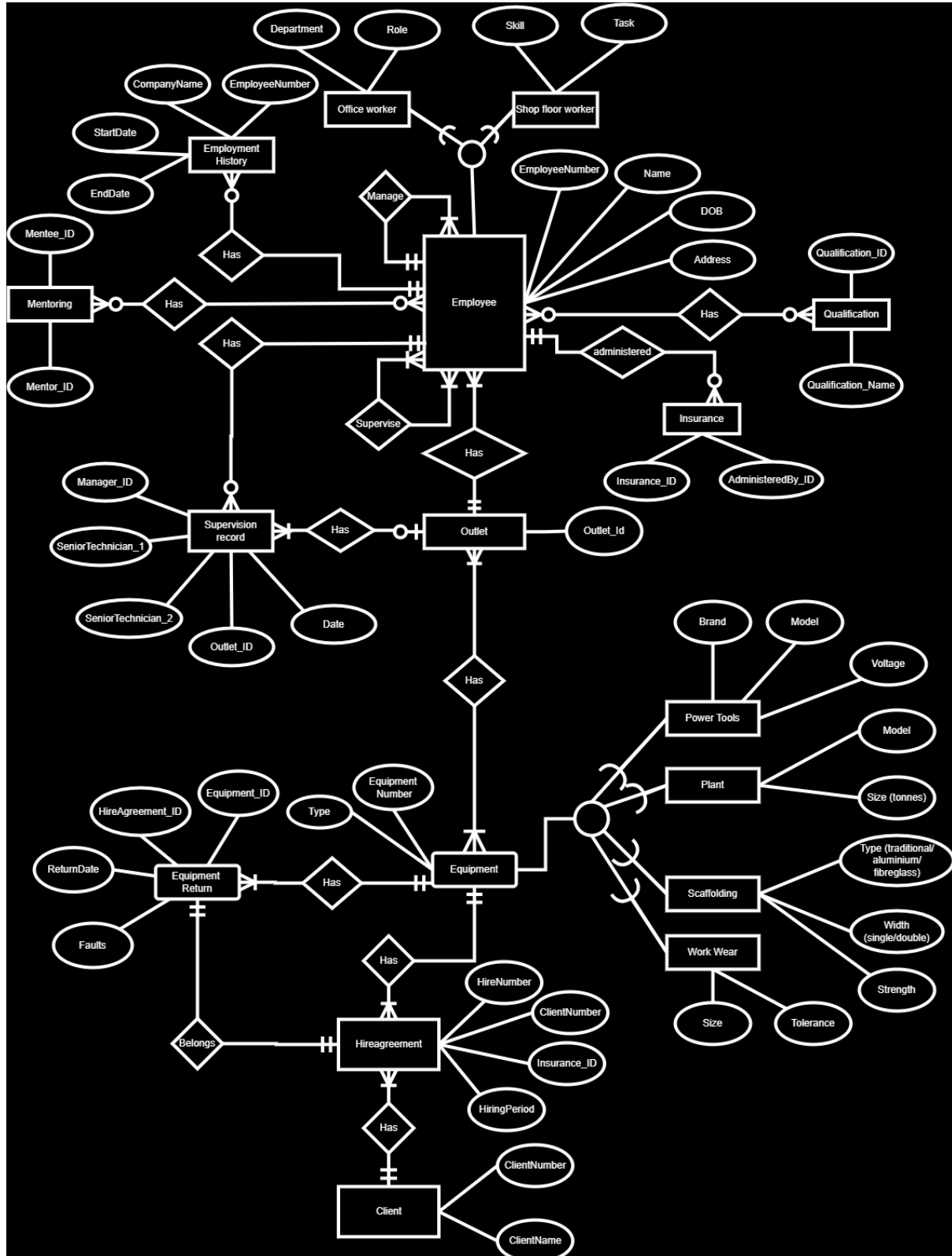
Sujan Tumbaraguddi.....14194733  
RAMANA kulanthaivelu.....14231913

## Table of Contents

1. Part A: Conceptual modeling.....	03
2. Part B: SQL programming.....	06
3. Part C: Sequential and parallel processing.....	12
4. Part D: Research report.....	16
<i>References</i> .....	18

## Part A: Conceptual modeling

1.



## 2.

Based on the ER diagram, here are the relational tables with justifications:

### 1. Client (Client\_ID (PK), Name)

Justification: Client is an entity that has a name (Name) and a unique identifier (Client\_ID).

### 2. Employee (Employee\_ID (PK), Name, DOB, Address, Skill (optional), Task (optional), Role (optional), Department (optional))

Justification: An entity called "Employee" has a unique identity (Employee\_ID) and personal information such as name, address, and date of birth. In order to support the attributes of both ShopFloorWorker and OfficeWorker, Skill, Task, Role, and Department are included. If an employee does not have those specified roles, these can be null.

### 3. Equipment (Equipment\_ID (PK), Type)

Justification: An entity called "Equipment" has a type categorization (Type) and a unique identifier (Equipment\_ID).

### 4. PowerTool (Brand, Model, Voltage)

Not a separate table (use separate columns in Equipment).

Justification: "PowerTool" adds particular information (Brand, Model, Voltage) as additional columns to the Equipment database and inherits attributes from Equipment (Equipment\_ID, Type).

### 5. Plant (Model, Size)

Not a separate table (use separate columns in Equipment).

Justification: Like 'PowerTool,' 'Plant' adds specialized information (Model, Size) as additional columns to the Equipment table and takes over attributes from Equipment.

### 6. Scaffolding (Type, Width, Strength)

Not a separate table (use separate columns in Equipment).

Justification: Using the same reasoning, 'Scaffolding' adds details (Type, Width, Strength) as extra columns to the Equipment table and inherits attributes from Equipment.

### 7. WorkWear (Tolerance, Size)

Not a separate table (use separate columns in Equipment).

Justification: "WorkWear" adds information (Tolerance, Size) as additional columns to the Equipment table and takes over its attributes from Equipment.

### 8. Outlet (Outlet\_ID (PK))

Justification: Outlet is an entity with a unique identifier (Outlet\_ID).

### 9. HireAgreement (HireAgreement\_ID (PK), Client\_ID (FK), HireStartDate, HireEndDate, Insurance\_ID (FK))

Justification: 'HireAgreement' records agreements between the company and a 'Client' (foreign key: Client\_ID). It contains start and finish dates, a unique identifier (HireAgreement\_ID), and a foreign key (Insurance\_ID) that references the insurance that was used.

10. SupervisionRecord (RecordID (PK), Date, Outlet\_ID (FK), Manager\_ID (FK - Employee as Manager), SeniorTechnician1\_ID (FK - Employee as SeniorTechnician1), SeniorTechnician2\_ID (FK - Employee can be null))

Justification: For an Outlet (foreign key: Outlet\_ID), the supervision data are tracked by the 'SupervisionRecord'. It references the Manager (foreign key - Manager\_ID from Employee table) and SeniorTechnician1 (foreign key - SeniorTechnician1\_ID from Employee table), and it has a unique record identifier (RecordID) and date. Optional is SeniorTechnician2 (foreign key may be null).

11. Qualification (Qualification\_ID (PK), Qualification\_Name)

Justification: "Qualification" is an entity that has a name that denotes the type of qualification and a unique identifier (Qualification\_ID).

12. EmployeeQualification (Employee\_ID (FK), Qualification\_ID (FK))

Justification: The many-to-many link between the employee and the qualification is shown in this table. The connection is established by the presence of foreign keys for both Employee\_ID and Qualification\_ID.

13. EmploymentHistory (Employee\_ID (FK), CompanyName, StartDate, EndDate)

Justification: EmploymentHistory tracks past work experiences for each Employee (foreign key - Employee\_ID). It includes the company name, start date, and end date.

Note: The ER diagram depicts a many-to-many relationship between Employee and Qualification. To represent this relationship in relational tables, a separate table "EmployeeQualification" is necessary.

14. Mentoring (Mentor\_ID (FK), Mentee\_ID (FK))

Justification: This table, which is similar to "EmployeeQualification," records mentoring ties. The Employee table has foreign keys for Mentor\_ID and Mentee\_ID, which together constitute the many-to-many relationship.

15. EquipmentReturn (EquipmentReturn\_ID (PK), Equipment\_ID (FK), HireAgreement\_ID (FK), ReturnDate, Faults)

Justification: "EquipmentReturn" monitors the equipment's return. It references the particular equipment (foreign key: Equipment\_ID), has a unique identity (EquipmentReturn\_ID), and is associated with the HireAgreement.

## Part B: SQL programming

### 1.

```
CREATE TABLE Employee (
    empId varchar2(5) PRIMARY KEY,
    name varchar2(50),
    address varchar2(250),
    DOB date,
    job varchar2(50),
    salaryCode varchar2(20),
    deptId varchar2(10),
    manager varchar2(5),
    schemeId varchar2(10),
    FOREIGN KEY (deptId) REFERENCES Department(deptId),
    FOREIGN KEY (schemeId) REFERENCES PensionScheme(schemeId),
    FOREIGN KEY (salaryCode) REFERENCES SalaryGrade(salaryCode)
);
```

```
CREATE TABLE Department (
    deptId varchar2(10) PRIMARY KEY,
    name varchar2(50)
);
```

```
CREATE TABLE SalaryGrade (
    salaryCode varchar2(20) PRIMARY KEY,
    startSalary number(10),
    finishSalary number(10)
);
```

```
CREATE TABLE PensionScheme (
    schemeId varchar2(10) PRIMARY KEY,
    name varchar2(50),
    rate number(10,2)
);
```

```
INSERT INTO Employee VALUES ('E101', 'Keita, J.', '1 high street',
    TO_DATE('06/03/76','DD/MM/YY'), 'Clerk', 'S1', 'D10', 'E110', 'S116');
INSERT INTO Employee VALUES ('E301', 'Wang,F. ', '22 railway road',
    TO_DATE('11/04/80','DD/MM/YY'), 'Sales Person', 'S2', 'D30', 'E310', 'S124');
INSERT INTO Employee VALUES ('E310', 'Flavel,K.', '14 crescent road',
    TO_DATE('25/11/69','DD/MM/YY'), 'Manager', 'S5', 'D30', '', 'S121');
```

```

INSERT INTO Employee VALUES ('E501', 'Payne, J.', '7 heap street',
TO_DATE('09/02/72','DD/MM/YY'), 'Analyst', 'S5', 'D50', 'E310', 'S121');
INSERT INTO Employee VALUES ('E102', 'Patel,R.', '16 glade close',
TO_DATE('13/07/74','DD/MM/YY'), 'Clerk', 'S1', 'D10', 'E110', 'S116');
INSERT INTO Employee VALUES ('E110', 'Smith,B.', '199 London road',
TO_DATE('22/05/70','DD/MM/YY'), 'Manager', 'S5', 'D10', ' ', 'S121');

```

```

INSERT INTO Department VALUES ('D10', 'Administration');
INSERT INTO Department VALUES ('D20', 'Finance');
INSERT INTO Department VALUES ('D30', 'Sales');
INSERT INTO Department VALUES ('D40', 'Maintenance');
INSERT INTO Department VALUES ('D50', 'IT Support');

```

```

INSERT INTO SalaryGrade VALUES ('S1', 17000, 19000);
INSERT INTO SalaryGrade VALUES ('S2', 19001, 24000);
INSERT INTO SalaryGrade VALUES ('S3', 24001, 26000);
INSERT INTO SalaryGrade VALUES ('S4', 26001, 30000);
INSERT INTO SalaryGrade VALUES ('S5', 30001, 39000);

```

```

INSERT INTO PensionScheme VALUES ('S110', 'AXA', 0.5);
INSERT INTO PensionScheme VALUES ('S121', 'Premier', 0.6);
INSERT INTO PensionScheme VALUES ('S124', 'Stakeholder', 0.4);
INSERT INTO PensionScheme VALUES ('S116', 'Standard', 0.4);

```

## 2. a).

Retrieve the employee names (ordered by name in ascending order and department ID in descending order), starting salaries, and department IDs.

```

SELECT E.name, S.startSalary, E.deptId
FROM Employee E
JOIN SalaryGrade S ON E.salaryCode = S.salaryCode
ORDER BY E.name ASC, E.deptId DESC;

```

Result:

NAME	STARTSALARY	DEPTID
Flavel,K.	30001	D30
Keita, J.	17000	D10
Patel,R.	17000	D10
Payne, J.	30001	D50

Smith,B.	30001	D10
Wang,F.	19001	D30

The SQL code uses the ORDER BY clause and JOIN syntax in accordance with Oracle standard SQL syntax. A database has responded to the query, and the outcome is shown.

**SQL Worksheet**

```

1 SELECT E.name, S.startSalary, E.deptId
2 FROM Employee E
3 JOIN SalaryGrade S ON E.salaryCode = S.salaryCode
4 ORDER BY E.name ASC, E.deptId DESC;

```

NAME	STARTSALARY	DEPTID
Flavel,K.	30001	D30
Keita, J.	17000	D10
Patel,R.	17000	D10
Payne, J.	30001	D50
Smith,B.	30001	D10
Wang,F.	19001	D30

## 2. b).

Determine how many workers are enrolled in each pension plan.

```

SELECT P.name AS Name_of_Scheme , COUNT(E.schemeId) AS Employee_count
FROM PensionScheme P
JOIN Employee E ON P.schemeId = E.schemeId
GROUP BY P.name;

```

NAME_OF_SCHEME	EMPLOYEE_COUNT
Premier	3
Standard	2
Stakeholder	1



The JOIN and GROUP BY clauses are used in the SQL code in accordance with Oracle standard SQL syntax. After running the query over a database, the outcome is shown.

SQL Worksheet	
<pre> 1 ✓ SELECT P.name AS Name_of_Scheme , COUNT(E.schemeId) AS Employee_count 2   FROM PensionScheme P 3   JOIN Employee E ON P.schemeId = E.schemeId 4   GROUP BY P.name; </pre>	
NAME_OF_SCHEME	EMPLOYEE_COUNT
Premier	3
Standard	2
Stakeholder	1

## 2. c).

Determine how many non-managerial staff members make more than or equivalent to \$35,000.

```

SELECT COUNT(E.name) AS Count_of_Employee
FROM Employee E
JOIN SalaryGrade S USING (salaryCode)
WHERE E.job <> 'Manager' AND S.finishSalary >= 35000;

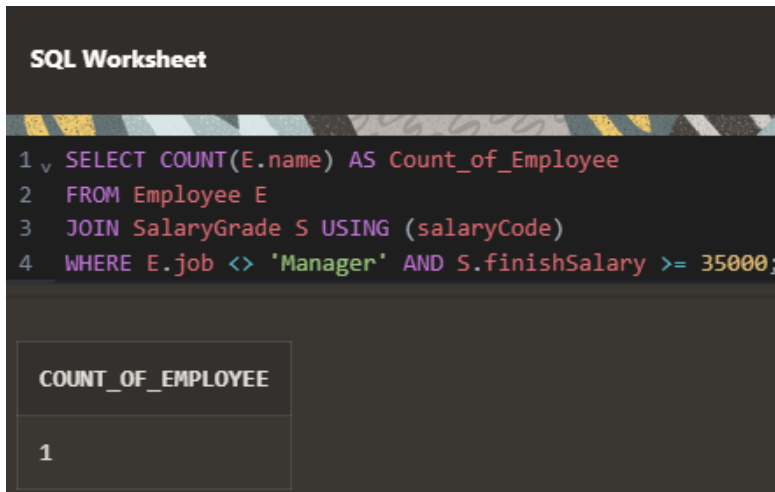
```

Result:

COUNT\_OF\_EMPLOYEE

1

Using JOIN and WHERE clauses, the SQL code complies with Oracle's standard SQL syntax. After running the query over a database, the outcome is shown.



The screenshot shows an SQL Worksheet with a dark background. At the top, it says "SQL Worksheet". Below that is a decorative horizontal bar with a pattern of yellow, blue, and grey. The SQL query is as follows:

```
1 v SELECT COUNT(E.name) AS Count_of_Employee
2 FROM Employee E
3 JOIN SalaryGrade S USING (salaryCode)
4 WHERE E.job <> 'Manager' AND S.finishSalary >= 35000;
```

Below the query, the result is displayed in a table with two columns: "COUNT\_OF\_EMPLOYEE" and a single row with the value "1".

## 2. d).

Get the names, IDs, and corresponding management names of all the employees.

```
SELECT e.empId AS employee_id,
       e.name AS employee_name,
       m.name AS manager_name
FROM Employee e
LEFT JOIN Employee m ON e.manager = m.empId;
```

EMPLOYEE_ID	EMPLOYEE_NAME	MANAGER_NAME
E301	Wang,F.	Flavel,K.
E501	Payne, J.	Flavel,K.
E101	Keita, J.	Smith,B.
E102	Patel,R.	Smith,B.
E310	Flavel,K.	-
E110	Smith,B.	-

The LEFT JOIN function in the SQL code connects employees and managers. It is defined in the Oracle standard SQL syntax. After running the query over a database, the outcome is shown.

**SQL Worksheet**

```
1 ✓ SELECT e.empId AS employee_id,  
2       e.name AS employee_name,  
3       m.name AS manager_name  
4 FROM Employee e  
5 LEFT JOIN Employee m ON e.manager = m.empId;
```

EMPLOYEE_ID	EMPLOYEE_NAME	MANAGER_NAME
E301	Wang, F.	Flavel, K.
E501	Payne, J.	Flavel, K.
E101	Keita, J.	Smith, B.
E102	Patel, R.	Smith, B.
E310	Flavel, K.	-
E110	Smith, B.	-

## Part C: Sequential and parallel processing

### 1. a)

```
CREATE TABLE FlightData (
    Year NUMBER(4),
    Month NUMBER(2),
    DayOfMonth NUMBER(2),
    DayOfWeek NUMBER(1),
    DepartureTime NUMBER(4),
    ActualDepartureTime NUMBER(4),
    ArrivalTime NUMBER(4),
    AirlineCarrier VARCHAR2(3) UNIQUE,
    FlightNumber VARCHAR2(10),
    DepartureDelay NUMBER(3),
    ArrivalDelay NUMBER(3),
    WeatherDelay NUMBER(3)
);
```

Justification:

- A table column and each attribute in the data structure match.
- Assuming that carrier codes consist of three characters, AirlineCarrier is defined as VARCHAR(3).
- The description provided does not include a primary key or any other limitations, but these might be added in response to particular needs.

### 1. b)

```
SELECT AirlineCarrier, COUNT(*) AS DelayedFlights
FROM FlightData
WHERE DepartureDelay > 0 OR ArrivalDelay > 0 OR WeatherDelay > 0
GROUP BY AirlineCarrier;
```

Justification:

- The carrier code (AirlineCarrier) is chosen in this SQL statement, and for each carrier, the number of flights (COUNT(\*)) when the delay in departure, arrival, or weather is greater than 0 is counted.
- The data are grouped by AirlineCarrier under the GROUP BY clause, which makes it possible to determine the number of delayed flights for each carrier independently.

## 2.

We'll create a solution that maps flight records to (key, value) pairs, shuffles them according to the carrier code, then reduces to determine the total number of delayed flights for each carrier in order to handle the flight delay analysis using MapReduce. This is the methodical way that we can achieve it:

### **Map Function:**

**Input:** Every entry in the file of flight data.

**Output:** A list of (key, value) pairs.

**Key:** Airlines Carrier Code (Unique Identifier)

**Value:** 1 (represents one flight record)

**Justification:** We parse every flight record here. We emit a key-value pair with the carrier code as the key and 1 as the value if there is a departure delay (departure delay > 0). This indicates that the carrier has experienced one delayed flight.

### **Code(Python):**

```
def map_function(record):
    carrier_code = record[7] # 8th attribute
    arrival_delay = record[10] # 11th attribute

    if arrival_delay > 0:
        yield (carrier_code, 1)
```

### **Shuffle Function:**

All key-value pairs that share the same carrier code are grouped.

This guarantees that all records of delayed flights belonging to a certain carrier are consolidated for subsequent handling.

### **Reduce Function:**

**Input:** A list of 1s (values) and a list of key-value pairs with the same carrier code (key).

**Output:** A single key-value pair.

**Key:** Airlines Carrier Code

**Value:** The total number of delayed flights for that carrier is represented by the sum of all 1s.

**Justification:** The list of 1s connected to every carrier code is iterated through by the reduce function. It just adds up all of these 1s to get the total number of flights that the carrier has had delayed.

### **Code(Python):**

```
def reduce_function(carrier_code, delays):
```

```
total_delays = sum(delays)
yield (carrier_code, total_delays)
```

### **Benefits of MapReduce:**

**MapReduce Paradigm:** MapReduce is a good fit for this situation since it can handle massive amounts of data in a distributed fashion.

**Key-Value Pair Structure:** To make the process of counting delayed flights for each airline easier, use pairs of (carrier\_code, 1).

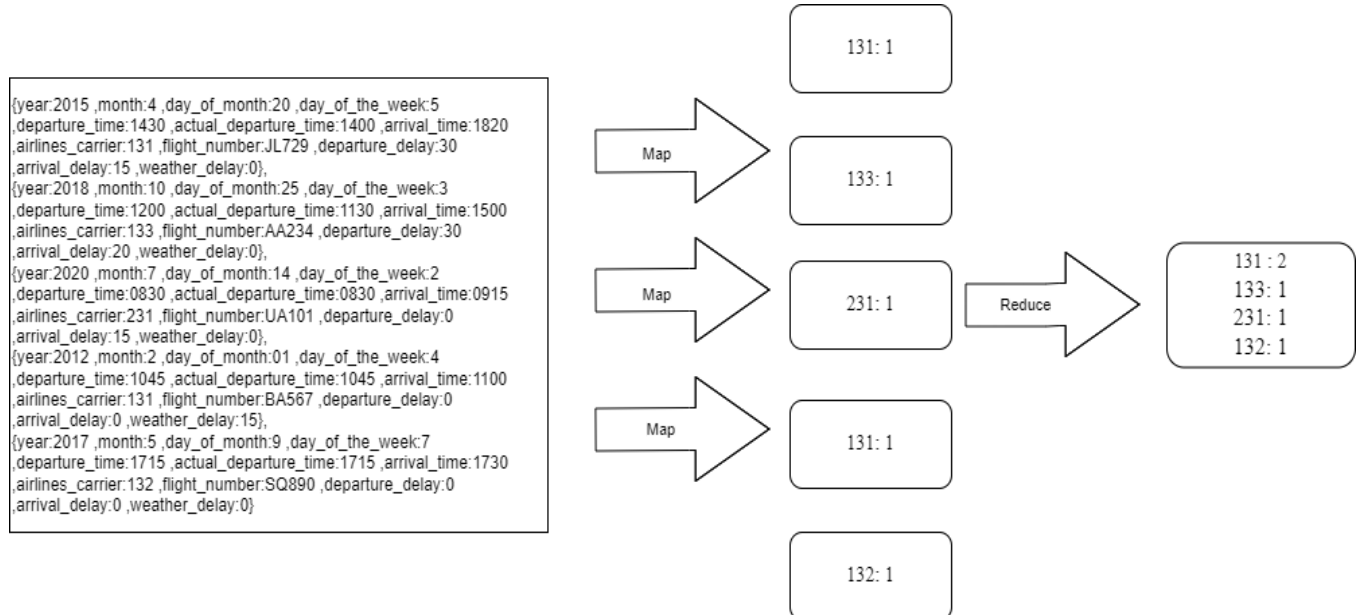
**Decentralized Processing:** As the data is too big to handle centrally and is stored in a regular file, we'll employ a decentralized MapReduce method. A piece of the file will be processed by each node in the distributed system, carrying out the map and reducing operations locally. The final output will then be obtained by combining the results from every node.

**Scalability:** Large datasets can be handled by the MapReduce framework by splitting up the work among several processors.

**Parallelization:** The processing can be accelerated by carrying out the map and reduction tasks simultaneously.

**Fault Tolerance:** The MapReduce framework can recover and re-execute the failed job on another computer in the event that a machine crashes during processing.

### **Diagram:**



### **Additional Flight Records:**

To further demonstrate the MapReduce approach, you can generate more flight records with different delay settings. Here are few instances:

- (2018, 10, 25, 3, 1200, 1130, 1500, 133, AA234, 30, 20, 0) - Delayed Departure (Departure Delay > 0)
- (2020, 7, 14, 2, 0830, 0830, 0915, 231, UA101, 0, 15, 0) - Delayed Arrival (Arrival Delay > 0)
- (2012, 2, 1, 4, 1045, 1045, 1100, 131, BA567, 0, 0, 15) - Weather Delay (Weather Delay > 0)
- (2017, 5, 9, 7, 1715, 1715, 1730, 131, SQ890, 0, 0, 0) - On-time flight (No Delay)

Utilizing the specified MapReduce functions to process these records, you can determine how many aircraft were delayed overall for every airline in the flight data file.

## Part D: Research report

In 2011, Craigslist encountered challenges with relational MySQL databases that couldn't manage their large and complex data volumes, and they demanded improved performance, scalability, and flexibility. As a result, they switched to NoSQL MongoDB servers. To understand this decision, we'll compare the characteristics of relational and NoSQL databases in relation to Craigslist's specific needs.

Relational databases like MySQL boast structured data models, ACID transactions, and advanced SQL capabilities. However, they face limitations when dealing with rapidly incoming and unstructured or semi-structured data.

These limitations primarily stem from,

**Schema Rigidity:** Relational databases require a predefined schema. Changing this schema can be complex and time-consuming. For instance, Craigslist's dynamic data structure required frequent schema updates, leading to downtime and operational challenges due to the high cost and complexity of schema modifications.

**Scalability:** Traditional relational databases struggle to scale horizontally across multiple systems. Instead, they rely on vertical scaling, which can be expensive and challenging to implement in practice. MySQL typically uses vertical scaling to handle increased data and user requests. However, for websites like Craigslist, horizontal scalability was crucial to handle massive daily data growth and user traffic.

**Performance:** As Craigslist's data expanded, MySQL's performance in processing complex queries and handling simultaneous requests may have been compromised. Relational databases often face challenges in efficiently indexing and querying large datasets, leading to potential performance bottlenecks and slower response times.

However, NoSQL databases such as MongoDB provide an alternative feature set that better fits Craigslist's needs:

**Schema flexibility:** NoSQL databases enable effortless and uninterrupted changes to their data structures due to their lack of rigid schemas or the flexibility of their existing schemas. This was essential for Craigslist, as they were able to swiftly adapt to evolving data needs without experiencing any downtime.

**Horizontal scalability:** Due to their horizontal scalability architecture, NoSQL databases enable the seamless distribution of data across multiple nodes or clusters. This architecture ensures high availability and performance, as it allows for elastic scaling to meet changing workload requirements.



**High performance:** NoSQL databases, like MongoDB, prioritize quick reads and writes. This makes them ideal for workloads with a high number of transactions and massive data sets. Craigslist leverages MongoDB's sharding and replication features to ensure reliability and fast responses, even during heavy traffic.

**Native support for unstructured data:** NoSQL databases like MongoDB excel at handling data that lacks a predefined structure, such as the data found in online marketplaces like Craigslist. MongoDB's flexible document model allows for storing different types of data in a single collection, simplifying data management and retrieval.

In Conclusion, Craigslist upgraded from MySQL to MongoDB due to the limitations of relational databases in handling Craigslist's growing data. MongoDB's NoSQL architecture offered solutions to these drawbacks:

**Performance Enhancement:** MongoDB's architecture improved data processing speed.

**Horizontal Scalability:** Craigslist could easily expand the database by adding more servers without disrupting operations.

**Schema Flexibility:** MongoDB's flexible data structure allowed Craigslist to modify their data formats as needed without rebuilding the database. With these advancements, MongoDB enabled Craigslist to seamlessly manage their expanding data volume and data structure changes, ensuring the smooth operation of their classified listings platform.

## ***References:***

*MapReduce 101: What It Is & How to Get Started.* (n.d.). Talend - a Leader in Data Integration & Data Integrity.  
<https://www.talend.com/uk/resources/what-is-mapreduce/>

*MongoDB Case Study: Craigslist.* (n.d.). MongoDB.  
<https://www.mongodb.com/post/15781260117/mongodb-case-study-craigslist>

*MySQL and NoSQL at Craigslist.* (n.d.). Software Development Tutorials and Videos | Software Development Videos and Tutorials: Java, .NET, Ruby, Python, JavaScript, Agile, Software Testing. Retrieved March 26, 2024, from  
<https://www.softdevtube.com/2012/09/12/mysql-and-nosql-at-craigslist/>

*The Craigslist Dilemma: A case study for big data and NoSQL solutions.* (n.d.). TheServerSide.com.  
<https://www.theserverside.com/feature/The-Craigslist-Dilemma-A-case-study-for-big-data-and-NoSQL-solutions#:~:text=So%20what%20was%20the%20NoSQL>

*Why these organizations migrated to MongoDB?* (n.d.). Wwww.linkedin.com. Retrieved March 26, 2024, from  
[https://www.linkedin.com/pulse/why-organizations-migrated-mongodb-jignesh-solanki/?trk=related\\_article\\_Why%20these%20organizations%20migrated%20to%20MongoDB%3F\\_article-card\\_title](https://www.linkedin.com/pulse/why-organizations-migrated-mongodb-jignesh-solanki/?trk=related_article_Why%20these%20organizations%20migrated%20to%20MongoDB%3F_article-card_title)