Coventry University

Name: Yuvaraj Bojjinettam

Student ID: 13395670

Email ID: bojjinetty@coventry.ac.uk

Name: Saitejesh Reddy Devireddygari

Student ID: 14089646

Email ID: saitejeshd@coventry.ac.uk

Module Name: Data Management Systems

Module Code: 7086CEM

Assignment Title: Database Management Systems

Contents

Part A

1 Question:

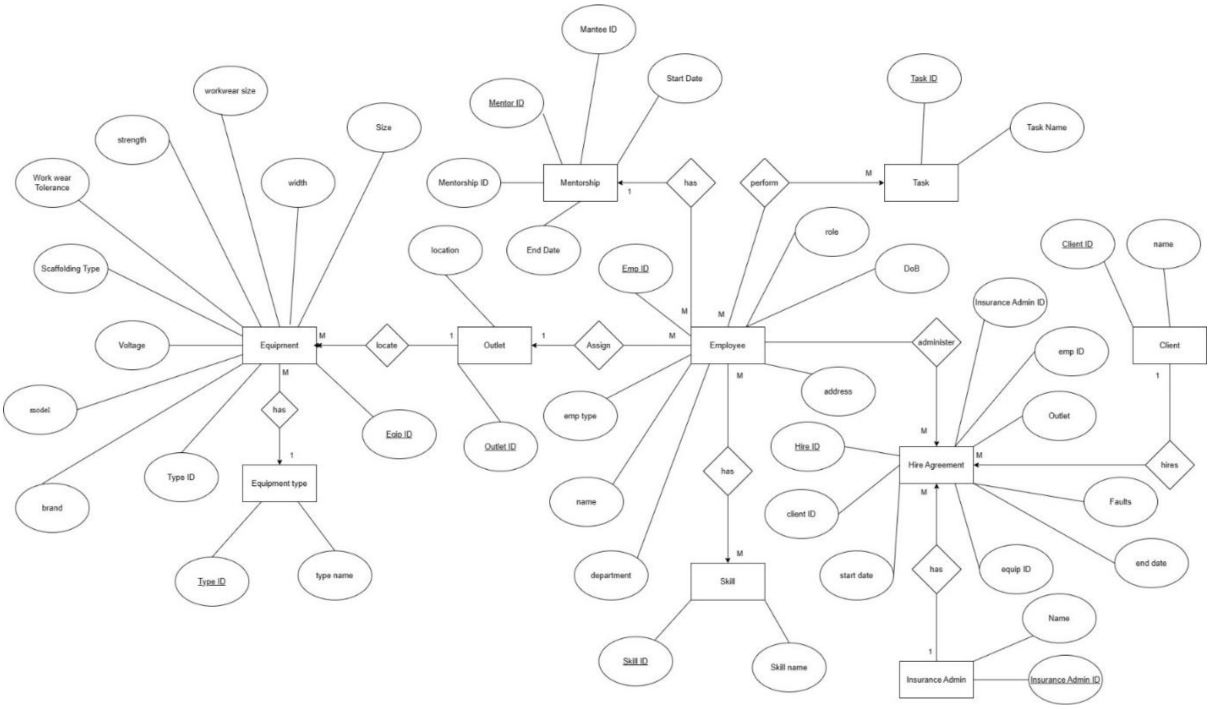


Figure 1: ER Diagram

2 Question :

1. Entity-Relationship (ER) Diagram

Entities:

- **EquipmentType**:

  Attributes: TypeID (Primary Key), TypeName

- **Equipment**:

  Attributes: EquipID (Primary Key), TypeID (Foreign Key), Brand, Model, Voltage (nullable), Size (nullable), ScaffoldingType (nullable), ScaffoldingWidth (nullable), ScaffoldingStrength (nullable), WorkWearTolerance (nullable), WorkWearSize (nullable)

- **Outlet**:

  Attributes: OutletID (Primary Key), Location

- **Employee**:

  Attributes: EmpID (Primary Key), Name, DOB, Address, EmpType, Role, Department

- **Skill**:

  Attributes: SkillID (Primary Key), SkillName

- **Task**:

  Attributes: TaskID (Primary Key), TaskName

- **HireAgreement**:

  Attributes: HireID (Primary Key), ClientID (Foreign Key), EquipID (Foreign Key), OutletID (Foreign Key), EmpID (Foreign Key), InsuranceAdminID (Foreign Key), StartDate, EndDate, Faults

- **Client**:

  Attributes: ClientID (Primary Key), Name

- **InsuranceAdmin**:

  Attributes: InsuranceAdminID (Primary Key), Name

- **Mentorship**:

  Attributes: MentorshipID (Primary Key), MentorID (Foreign Key), MenteeID (Foreign Key), StartDate, EndDate

Justification:

1. **EquipmentType:** This entity represents the different types of equipment available, such as scaffolding, work wear, or tools. The TypeID is the primary key, and TypeName stores the name of the equipment type.

2. **Equipment:** This entity stores information about individual pieces of equipment. The EquipID is the primary key, TypeID is a foreign key referencing the EquipmentType entity, and other attributes like Brand, Model, Voltage, Size, etc., capture specific details of the equipment.

3. **Outlet:** This entity represents the locations or outlets where equipment is stored or assigned. The OutletID is the primary key, and Location stores the address or details of the outlet.

4. **Employee:** This entity stores information about employees working in the organization. The EmpID is the primary key, and attributes like Name, DOB, Address, EmpType, Role, and Department capture employee details.

5. **Skill:** This entity represents the various skills that employees possess. The SkillID is the primary key, and SkillName stores the name of the skill.

6. **Task:** This entity represents the different tasks or jobs that employees perform. The TaskID is the primary key, and TaskName stores the name of the task.

7. **HireAgreement:** This entity stores information about hire agreements made with clients for renting out equipment. The HireID is the primary key, and foreign keys ClientID, EquipID, OutletID, EmpID, and InsuranceAdminID link to the respective entities. Other attributes like StartDate, EndDate, and Faults capture details of the hire agreement.

8. **Client:** This entity represents the clients who hire equipment from the organization. The ClientID is the primary key, and Name stores the client's name.

9. **InsuranceAdmin:** This entity represents the insurance administrators responsible for managing insurance-related aspects of hire agreements. The InsuranceAdminID is the primary key, and Name stores the administrator's name.

10. **Mentorship:** This entity stores information about mentorship relationships between employees. The MentorshipID is the primary key, and foreign keys MentorID and MenteeID link to the Employee entity (recursive relationship). StartDate and EndDate capture the duration of the mentorship.

Keys:

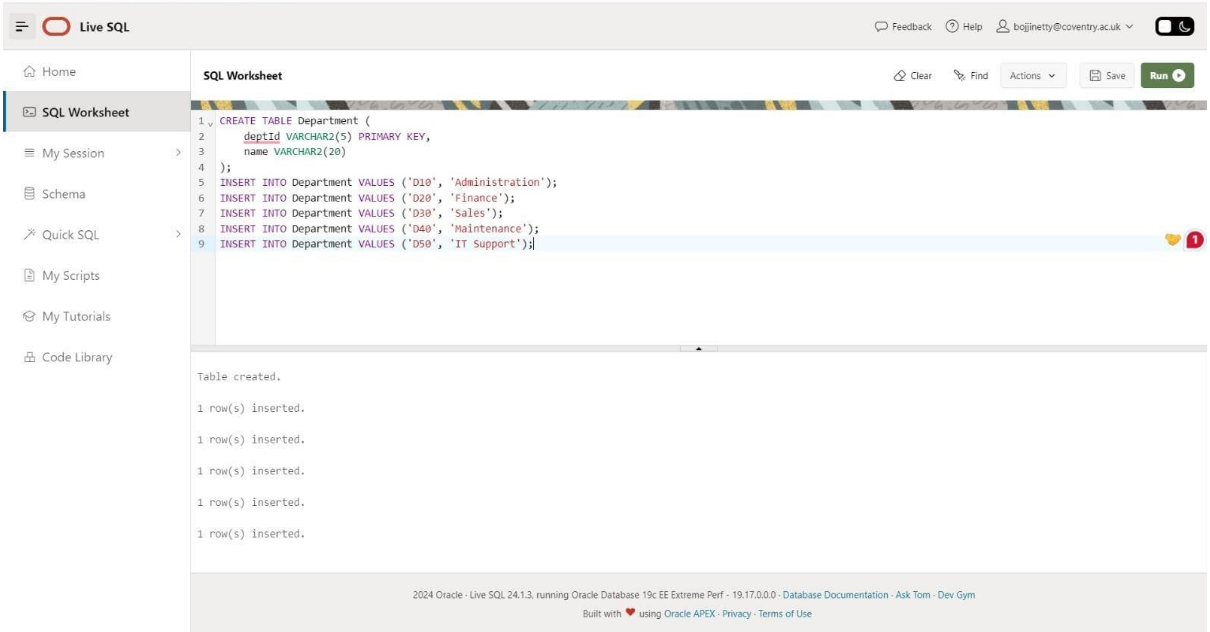
- **Primary keys**
  TypeID, EquipID, OutletID, EmpID, SkillID, TaskID, HireID, ClientID, InsuranceAdminID, and MentorshipID are used to uniquely identify records in their respective entities.

- **Foreign keys**
  TypeID in Equipment, ClientID, EquipID, OutletID, EmpID, and InsuranceAdminID in HireAgreement, MentorID and MenteeID in Mentorship establish relationships between entities.

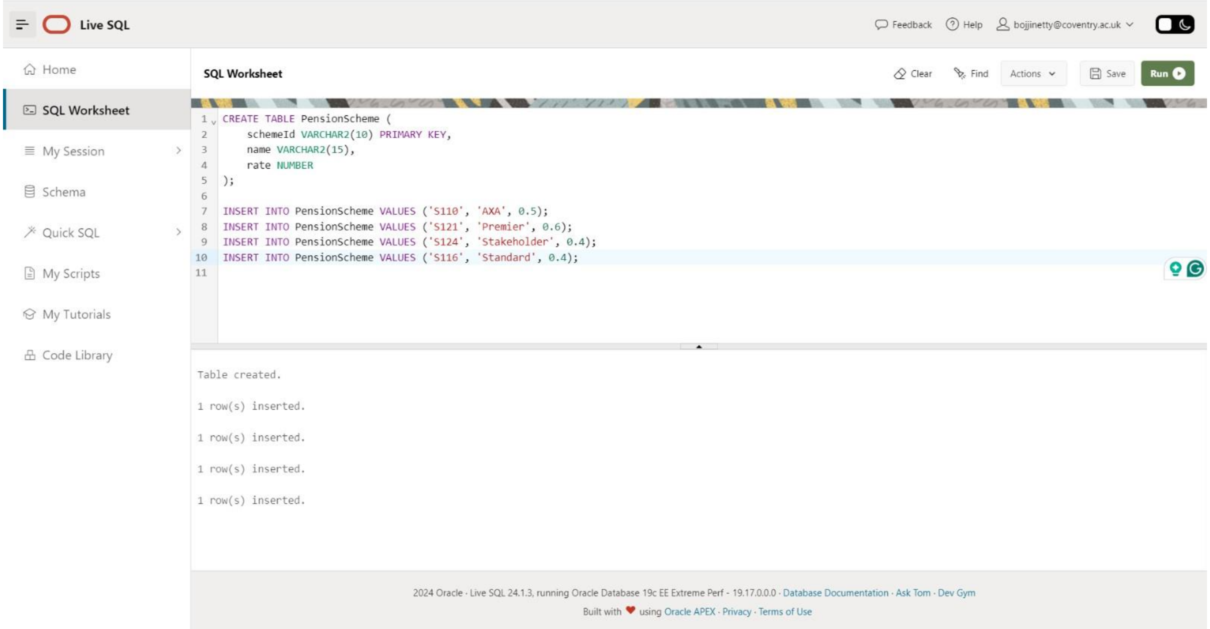
PART B

1 Question :

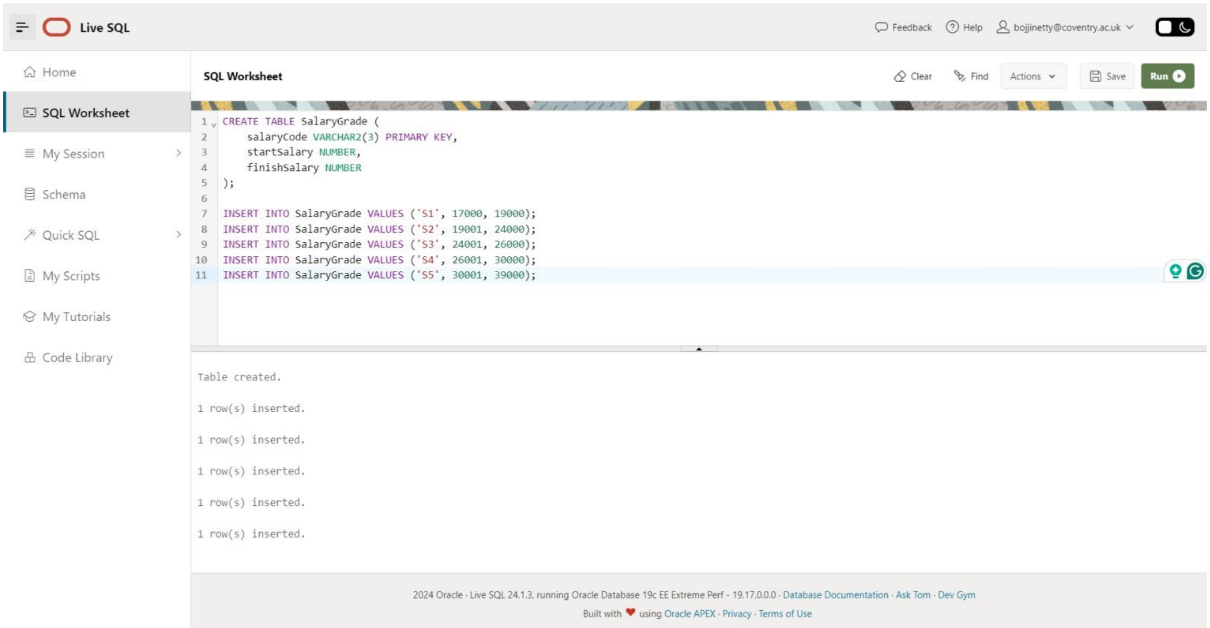
```sql
CREATE TABLE Department (
    deptId VARCHAR2(5) PRIMARY KEY,
    name VARCHAR2(20)
);
INSERT INTO Department VALUES ('D10', 'Administration');
INSERT INTO Department VALUES ('D20', 'Finance');
INSERT INTO Department VALUES ('D30', 'Sales');
INSERT INTO Department VALUES ('D40', 'Maintenance');
INSERT INTO Department VALUES ('D50', 'IT Support');
```




```sql
CREATE TABLE SalaryGrade (
    salaryCode VARCHAR2(3) PRIMARY KEY,
    startSalary NUMBER,
    finishSalary NUMBER
);

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);
```

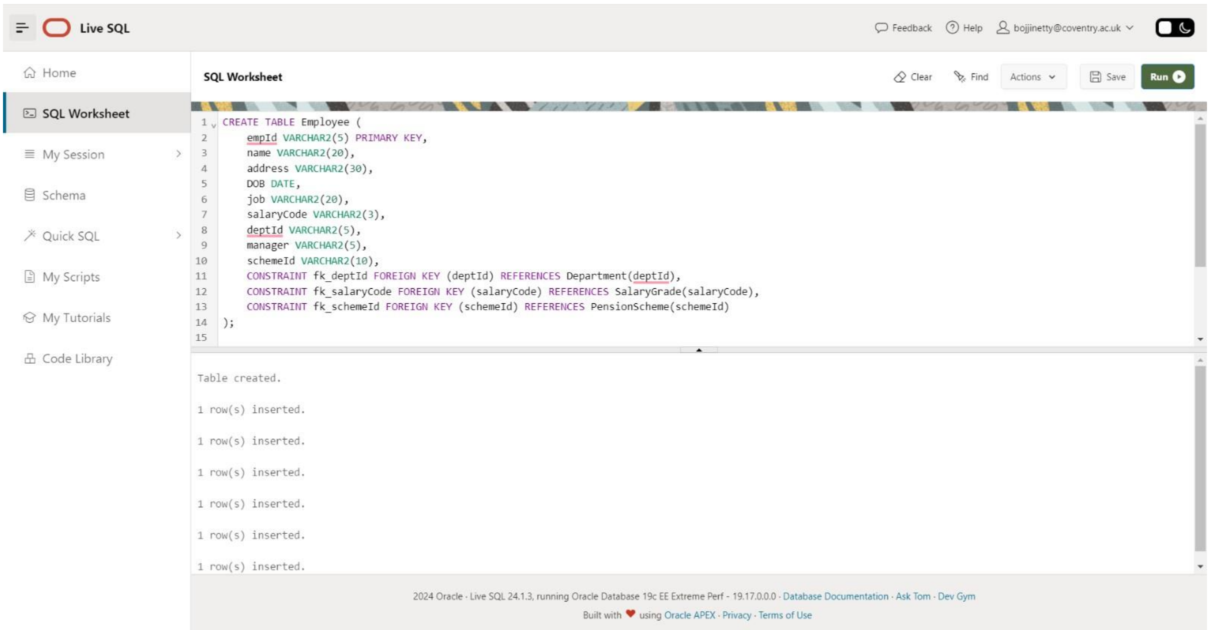



```
CREATE TABLE PensionScheme (
    schemeId VARCHAR2(10) PRIMARY KEY,
    name VARCHAR2(15),
    rate NUMBER
);

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);
```




```
CREATE TABLE Employee (
    empId VARCHAR2(5) PRIMARY KEY,
    name VARCHAR2(20),
    address VARCHAR2(30),
    DOB DATE,
```

```
    job VARCHAR2(20),
    salaryCode VARCHAR2(3),
    deptId VARCHAR2(5),
    manager VARCHAR2(5),
    schemeId VARCHAR2(10),
    CONSTRAINT fk_deptId FOREIGN KEY (deptId) REFERENCES
Department(deptId),
    CONSTRAINT fk_salaryCode FOREIGN KEY (salaryCode) REFERENCES
SalaryGrade(salaryCode),
    CONSTRAINT fk_schemeId FOREIGN KEY (schemeId) REFERENCES
PensionScheme(schemeId)
);

INSERT INTO Employee VALUES ('E101', 'Keita, J.', '1 high street',
TO_DATE('1976-03-06', 'YYYY-MM-DD'), 'Clerk', 'S1', 'D10', 'E110', 'S116');
INSERT INTO Employee VALUES ('E301', 'Wang, F.', '22 railway road',
TO_DATE('1980-04-11', 'YYYY-MM-DD'), 'Sales_Person', 'S2', 'D30', 'E310',
'S124');
INSERT INTO Employee VALUES ('E310', 'Flavel, K.', '14 crescent road',
TO_DATE('1969-11-25', 'YYYY-MM-DD'), 'Manager', 'S5', 'D30', NULL, 'S121');
INSERT INTO Employee VALUES ('E501', 'Payne, J.', '7 heap street',
TO_DATE('1972-02-09', 'YYYY-MM-DD'), 'Analyst', 'S5', 'D50', 'E310',
'S121');
INSERT INTO Employee VALUES ('E102', 'Patel, R.', '16 glade close',
TO_DATE('1974-07-13', 'YYYY-MM-DD'), 'Clerk', 'S1', 'D10', 'E110', 'S116');
INSERT INTO Employee VALUES ('E110', 'Smith, B.', '199 London road',
TO_DATE('1970-05-22', 'YYYY-MM-DD'), 'Manager', 'S5', 'D10', NULL, 'S121');
```

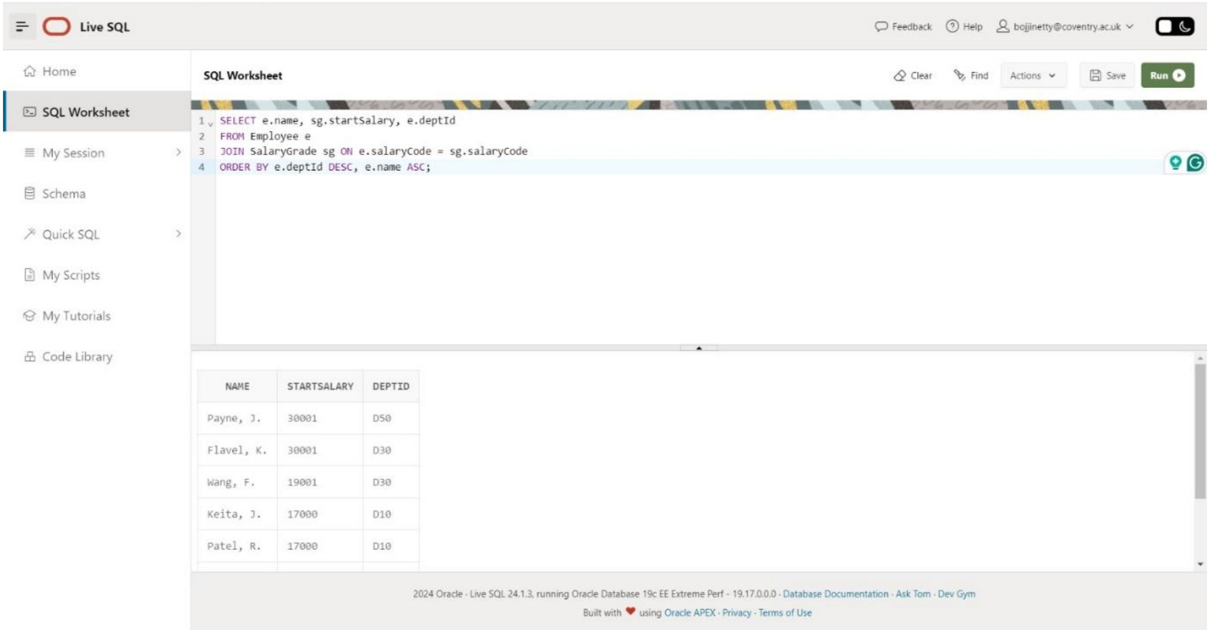


2 Question

--2a

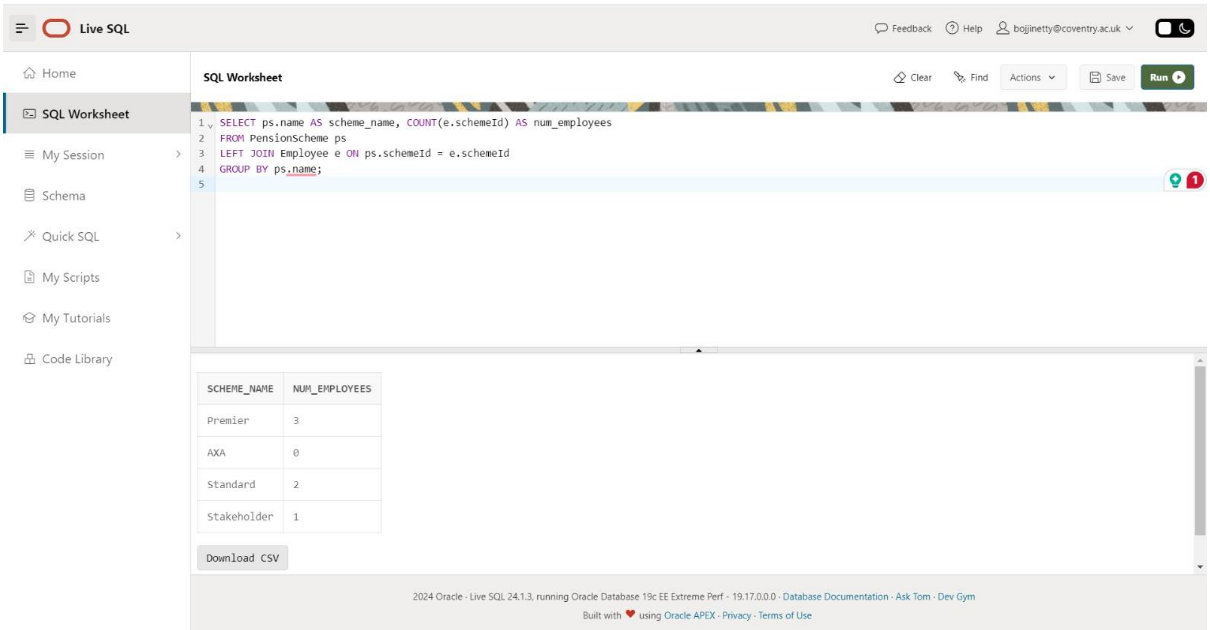
```
SELECT e.name, sg.startSalary, e.deptId
```

```sql
FROM Employee e
JOIN SalaryGrade sg ON e.salaryCode = sg.salaryCode
ORDER BY e.deptId DESC, e.name ASC;
```



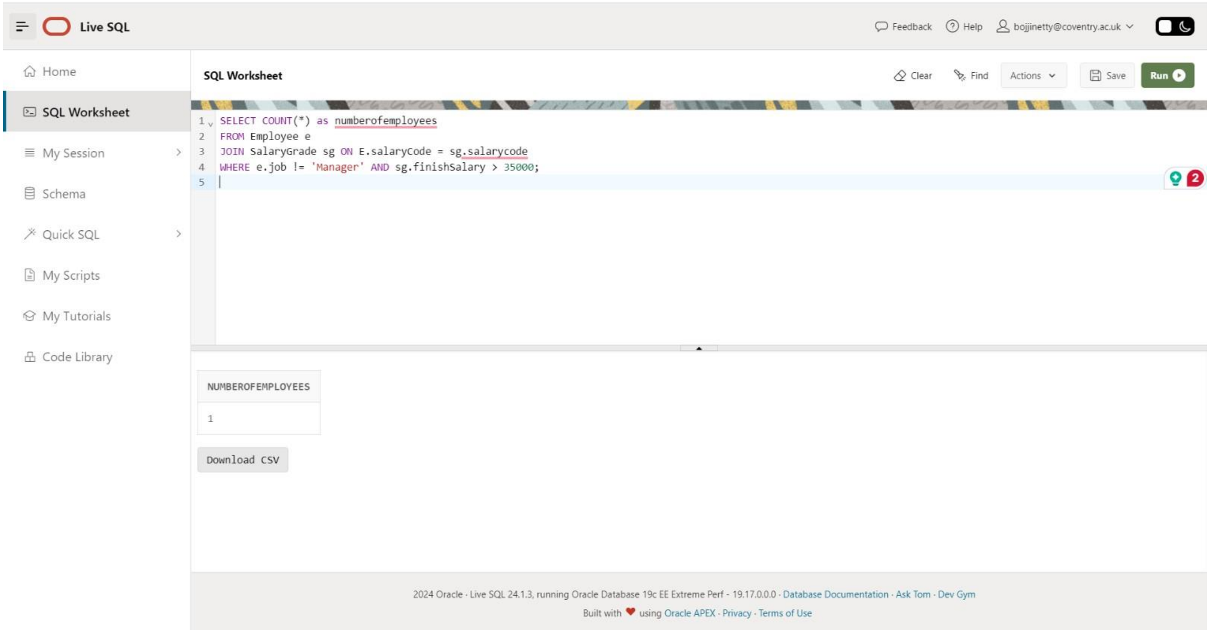
--2b

```sql
SELECT ps.name AS scheme_name, COUNT(e.schemeId) AS num_employees
FROM PensionScheme ps
LEFT JOIN Employee e ON ps.schemeId = e.schemeId
GROUP BY ps.name;
```



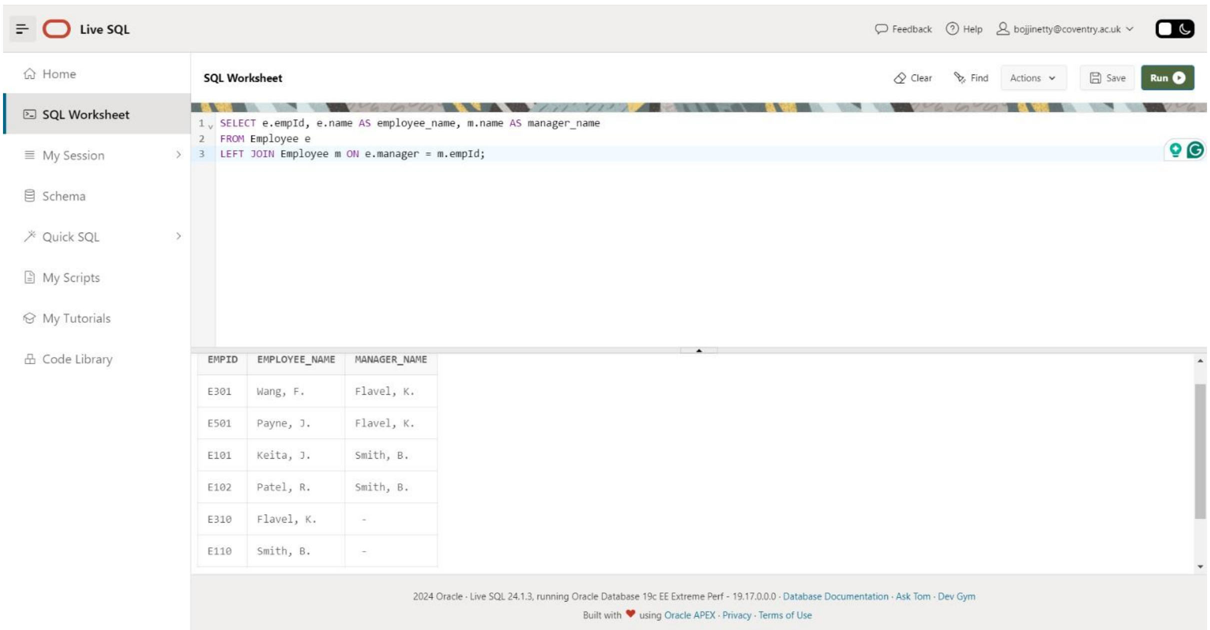
--2c

```sql
SELECT COUNT(*) as numberofemployees
FROM Employee e
JOIN SalaryGrade sg ON E.salaryCode = sg.salarycode
WHERE e.job != 'Manager' AND sg.finishSalary > 35000;
```



--2d

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



PART C

1 Question

1a)

```
CREATE TABLE flight_data (
    year INT CHECK (year BETWEEN 2000 AND 2020),
    month INT CHECK (month BETWEEN 1 AND 12),
    day_of_month INT CHECK (day_of_month BETWEEN 1 AND 31),
    day_of_week INT CHECK (day_of_week BETWEEN 1 AND 7),
    departure_time_recorded INT CHECK (departure_time_recorded BETWEEN 0
AND 2359),
    scheduled_departure_time INT CHECK (scheduled_departure_time BETWEEN 0
AND 2359),
    arrival_time_recorded INT CHECK (arrival_time_recorded BETWEEN 0 AND
2359),
    airline_carrier VARCHAR(3) NOT NULL,
    flight_number VARCHAR(10) NOT NULL,
    departure_delay INT,
    arrival_delay INT,
    weather_delay INT,
    PRIMARY KEY (year, month, day_of_month, airline_carrier, flight_number)
);
```

Justification:

- The table name is flight_data.
- Each attribute is defined as a column with the appropriate data type and constraints.
- The year, month, day_of_month, airline_carrier, and flight_number columns together form the primary key to ensure uniqueness of each record.
- Check constraints are applied to ensure that the values fall within the specified ranges.
- The NOT NULL constraint is applied to airline_carrier and flight_number columns as they are required.

1b)

```
SELECT airline_carrier, COUNT(*) AS delayed_flights
FROM flight_data
WHERE departure_delay > 0 OR arrival_delay > 0
GROUP BY airline_carrier;
```

Justification:

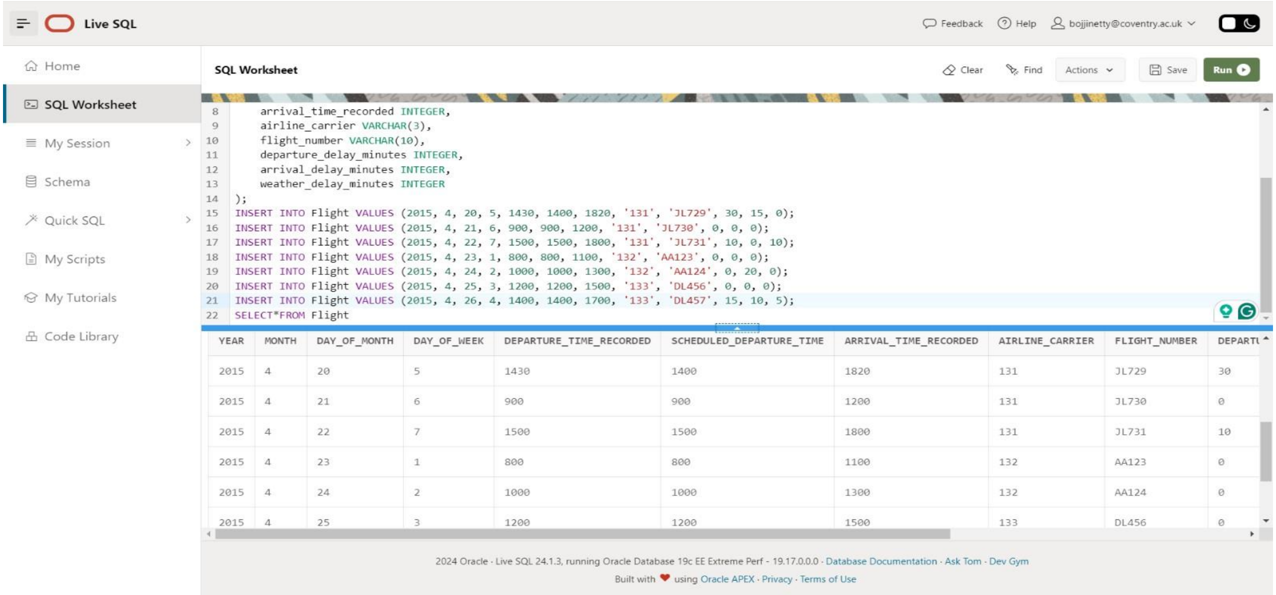
- The SELECT statement retrieves the airline_carrier column and counts the number of records for each carrier using COUNT(*).
- The AS keyword renames the COUNT(*) expression to delayed_flights for better readability.

- The FROM clause specifies the flight_data table as the source.
- The WHERE clause filters the records where either departure_delay or arrival_delay is greater than 0, indicating a delay.
- The GROUP BY clause groups the records by airline_carrier to aggregate the count for each carrier.

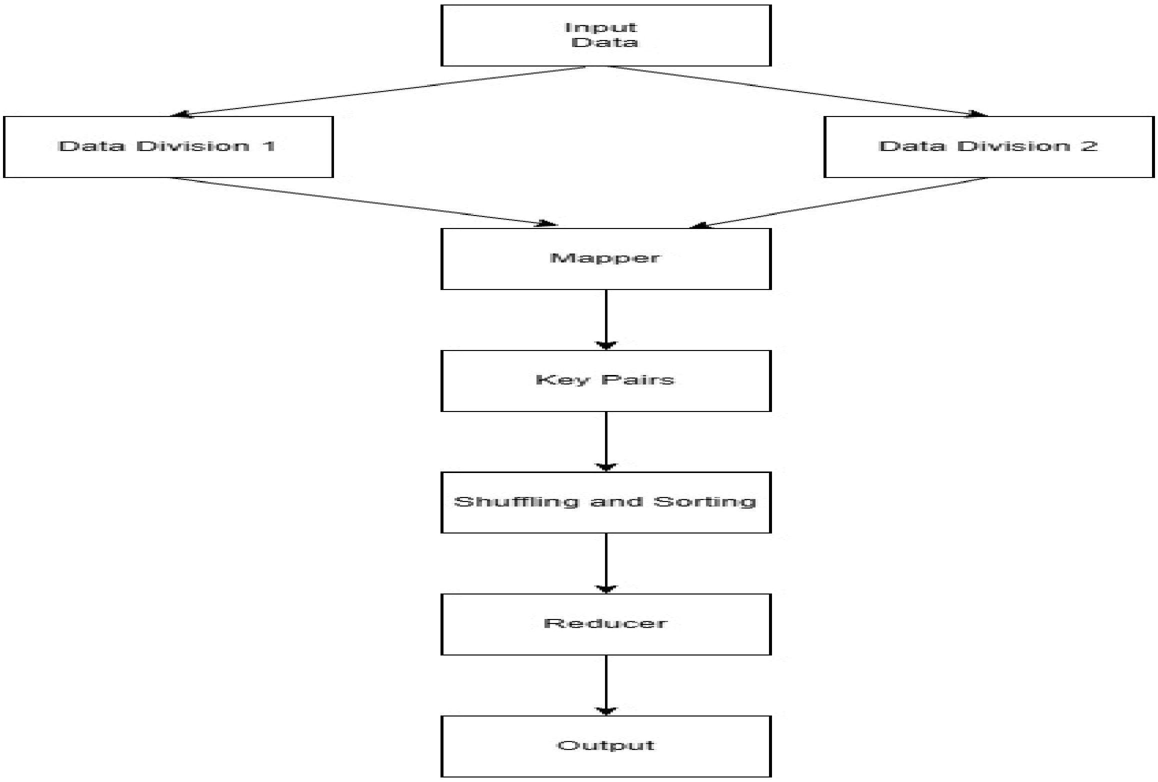
2 Question

Sample data for Flight delay

```
CREATE TABLE Flight (
    year INTEGER CHECK (year BETWEEN 2000 AND 2020),
    month INTEGER CHECK (month BETWEEN 1 AND 12),
    day_of_month INTEGER CHECK (day_of_month BETWEEN 1 AND 31),
    day_of_week INTEGER CHECK (day_of_week BETWEEN 1 AND 7),
    departure_time_recorded INTEGER,
    scheduled_departure_time INTEGER,
    arrival_time_recorded INTEGER,
    airline_carrier VARCHAR(3),
    flight_number VARCHAR(10),
    departure_delay_minutes INTEGER,
    arrival_delay_minutes INTEGER,
    weather_delay_minutes INTEGER
);
INSERT INTO Flight VALUES (2015, 4, 20, 5, 1430, 1400, 1820, '131',
'JL729', 30, 15, 0);
INSERT INTO Flight VALUES (2015, 4, 21, 6, 900, 900, 1200, '131', 'JL730',
0, 0, 0);
INSERT INTO Flight VALUES (2015, 4, 22, 7, 1500, 1500, 1800, '131',
'JL731', 10, 0, 10);
INSERT INTO Flight VALUES (2015, 4, 23, 1, 800, 800, 1100, '132', 'AA123',
0, 0, 0);
INSERT INTO Flight VALUES (2015, 4, 24, 2, 1000, 1000, 1300, '132',
'AA124', 0, 20, 0);
INSERT INTO Flight VALUES (2015, 4, 25, 3, 1200, 1200, 1500, '133',
'DL456', 0, 0, 0);
INSERT INTO Flight VALUES (2015, 4, 26, 4, 1400, 1400, 1700, '133',
'DL457', 15, 10, 5);
SELECT*FROM Flight
```



General Process:



Map Phase:

Objective: Transform the input data into (key, value) pairs, where the key is the airline carrier code, and the value is a tuple containing the departure and arrival delay minutes.

Process: The mapper function processes each record and emits key-value pairs.

Sample Mapper Output:

- For the record (2015, 4, 20, 5, 1430, 1400, 1820, '131', 'JL729', 30, 15, 0), the mapper will emit ('131', (30, 15)).

- For the record (2015, 4, 21, 6, 900, 900, 1200, '131', 'JL730', 0, 0, 0), the mapper will emit ('131', (0, 0)).
- For the record (2015, 4, 22, 7, 1500, 1500, 1800, '131', 'JL731', 10, 0, 10), the mapper will emit ('131', (10, 0)).

Justification:

The mapper emits the airline carrier code as the key, which will allow the shuffle phase to group records by carrier. The value contains the relevant delay information needed for the reduce phase.

Shuffle Phase:

Objective: Group the key-value pairs emitted by the mappers based on the keys (airline carrier codes).

Process: All key-value pairs with the same key are grouped together.

Sample Shuffle Output:

- For the key '131', the shuffle phase will group together the values (30, 15), (0, 0), (10, 0).
- For the key '132', the shuffle phase will group together the values (0, 0), (0, 20).
- For the key '133', the shuffle phase will group together the values (0, 0), (15, 10).

Justification: Grouping the key-value pairs by the airline carrier code allows the reduce phase to process all records for each carrier together.

Sort Phase (optional):

Objective: Sort the grouped key-value pairs based on the keys (airline carrier codes).

Process: Based on the keys, the grouped key-value pairs are sorted..

Sample Sort Output: (same as Shuffle Output, but sorted by the keys)

- For the key '131', the values (30, 15), (0, 0), (10, 0) are sorted based on the key.
- For the key '132', the values (0, 0), (0, 20) are sorted based on the key.
- For the key '133', the values (0, 0), (15, 10) are sorted based on the key.

Justification: Sorting the key-value pairs by the keys can aid in the reduce phase, especially if the reducer needs to process the values in a specific order.

Reduce Phase:

Objective: Count the number of records with non-zero delay values (either departure or arrival delay) for each airline carrier code.

Process: The reducer function processes the key-value pairs for each key (airline carrier code). It counts the number of records where either the departure delay or arrival delay is non-zero, and emits the airline carrier and the corresponding count of delayed flights.

Sample Reducer Output:

- For the key '131', the reducer will process the values (30, 15), (0, 0), (10, 0) and emit ('131', 2), since two records have non-zero delay values.
- For the key '132', the reducer will process the values (0, 0), (0, 20) and emit ('132', 1), since one record has a non-zero arrival delay.
- For the key '133', the reducer will process the values (0, 0), (15, 10) and emit ('133', 1), since one record has non-zero departure and arrival delays.

Justification: The reducer processes all records for each carrier together and counts the number of delayed flights, which is the desired output.

Decentralized Solution with Justification:

The MapReduce solution provided is a decentralized solution suitable for processing large datasets that cannot be handled efficiently in a centralized manner. Here's the justification:

1. **Data Parallelism**: MapReduce allows the input data to be split into multiple blocks, which can be processed in parallel by different mappers. This parallelism enables efficient processing of large datasets.
2. **Fault Tolerance**: If a mapper or reducer fails during the computation, MapReduce can automatically re-execute the failed tasks on other nodes, providing fault tolerance and reliability.
3. **Scalability**: Adding additional nodes to the cluster allows MapReduce to scale horizontally. More processing capacity can be added to handle the increasing workload as the dataset becomes bigger.

4. **Simplified Programming Model**: The MapReduce programming model abstracts away the complexities of parallel and distributed computing, allowing developers to focus on the map and reduce functions rather than low-level details like data partitioning, task scheduling, and fault handling.
5. **Flexible Data Formats**: MapReduce can process various data formats, including structured, semi-structured, and unstructured data, making it suitable for a wide range of data processing tasks.
6. **Data Locality**: MapReduce tries to schedule tasks close to the data they need to process, minimizing network traffic and improving performance.
7. **Batch Processing**: While not suitable for real-time processing, MapReduce is efficient for batch processing of large datasets, making it a good fit for the given problem of analyzing historical flight data.

PART D

Research Report

Introduction

Craigslist is a popular classified listing site. With over 1.5 million new ads submitted every day, and over a billion records, Craigslist faced huge problems with managing their massive volume of data. Furthermore, records had to be moved to an archive after a 60-day time limit per legal regulations. These difficulties forced Craigslist to evaluate its database setup, which resulted in the switch in 2011 from relational MySQL servers to NoSQL MongoDB servers.

Characteristics of Relational Databases and NoSQL Databases:

Relational databases, such as MySQL, are based on the relational model and utilize structured query language (SQL) for data manipulation and retrieval. They are characterized by predefined schemas, ACID transactions, and strong consistency. In contrast, NoSQL databases, like MongoDB, adopt a non-relational, distributed approach, offering flexible schemas, eventual consistency, horizontal scalability, and support for unstructured or semi-structured data.

Reasons for Migration:

1. **Scalability**: MySQL's rigid schema and vertical scaling limitations posed challenges in accommodating the exponential growth of data on Craigslist. MongoDB's flexible document model and horizontal scalability allowed for seamless expansion to handle increasing data volumes without sacrificing performance.

2. **Schema Flexibility**: The evolving nature of Craigslist's data structure necessitated frequent changes to the database schema, resulting in costly and disruptive downtime. MongoDB's schema-less design eliminated the need for predefined schemas, enabling agile development and accommodating schema changes without downtime.

3. **Operational Complexity**: Managing the transition of data between live and archival storage, as mandated by legislation, was cumbersome and time-consuming with MySQL. MongoDB's native support for sharding and data partitioning simplified data management tasks, including archival processes, reducing operational overhead.

4. **Development Agility**: Introducing new features or modifying existing ones in MySQL's rigid schema environment was challenging and time-intensive. MongoDB's flexible data model facilitated rapid application development and iteration, empowering developers to innovate and adapt to evolving user requirements more efficiently.

5. **Performance and Scalability**: MySQL's performance deteriorated as data volumes grew, necessitating costly hardware upgrades. MongoDB's distributed architecture and horizontal scalability ensured consistent performance, even with massive datasets, while accommodating future growth seamlessly.

Conclusion

The necessity to get over traditional relational databases' limitations in handling huge quantities of dynamic, unstructured data forced Craigslist to switch from MySQL to MongoDB. Craigslist needed a system that could be agile, scalable, and performant; MongoDB's flexible schema, horizontal scalability, operational simplicity, and developer-friendly capabilities met these needs. Craigslist positioned itself to effectively manage its data-intensive operations, comply with legal obligations, and create innovation in platform development by using NoSQL technology.

References

1. Chodorow, K. (2013). MongoDB: The Definitive Guide. O'Reilly Media.
2. Han, J., Haihong, E., & Le, G. (2011). Survey on NoSQL database. In P. M. Atzeni et al. (Eds.), Proceedings of the 2011 6th International Conference on Pervasive Computing and Applications (pp. 363-366). IEEE.
3. Kreps, J. (2014). Questioning the Lambda Architecture. Retrieved from https://www.oreilly.com/radar/questioning-the-lambda-architecture/.
4. MongoDB. (n.d.). MongoDB vs. SQL: Day 1-2. Retrieved from https://www.mongodb.com/blog/post/mongodb-vs-sql-day-1-2.
5. Stonebraker, M. (2010). SQL Databases v. NoSQL Databases. Communications of the ACM, 53(4), 10-11.
6. Wiederhold, G. (2012). NoSQL Evaluation. In H.-C. Raghavan et al. (Eds.), Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (pp. 1001-1004). ACM.