UNIVERSITÉ Concordia UNIVERSITY

GINA CODY SCHOOL OF ENGINEERING AND COMPUTER SCIENCE

COMP 6521

Advanced Database Technology and Applications

Mini Project 1

Submitted To: N. Shiri

By

Mayur Jodhani (40230634)

Prachi Patel (40261038)

Snehee Patel (40231497)

Dharmil Vaghasiya (40230633)

<u>INTRODUCTION:</u>

In the modern data world, it becomes extremely important to understand where the data comes from. An essential capability in the data management landscape of today is the capacity to explain query results. In order to accommodate this feature, research investigates the expansion of the relational data model to annotated relations. An additional attribute(s) I is included in annotated relations, indicated as R'(A1,..., An, I), to record additional information for each tuple in the relation. These annotations shed light on the methods used to retrieve each tuple from the database in a query result.

With the use of input tuple annotations, this project aims to produce explanations for query results. In the literature, these justifications are frequently referred to as "provenance". The study by Green et al. (PODS 2007), which provides insights into the syntax and semantics, lays the groundwork for this research.

In our database D, we have specialised annotated relations R(A1, A2, ..., An, I) and S(B1, B2, ..., Bm, J). Within these relations, attributes A1 to An and B1 to Bm are the standard features, while I and J serve as unique tuple IDs, essential for explaining the origins of query results.

Our objective revolves around formulating a SQL query, denoted as Q, that effectively merges the R and S relations. This query is designed to cherry-pick specific attributes from these relations, harnessing the power of tuple IDs (I and J) to meticulously trace the lineage of tuples present in the query output. The resultant query output, aptly named T(A1, ..., An, B1, ..., Bm, K), boasts an additional attribute denoted as K. This attribute is pivotal as it encapsulates a comprehensive set of annotations linked to each tuple within T. These annotations, derived from the contributing tuples in both R and S, provide intricate insights into the complex journey of each tuple present in the final output.

<u>PROVENANCE RULES:</u>

For the given database R:

R:

A	**B**	**C**	**Provenance**
a	b	c	p
d	b	e	q
f	g	e	r

Operation	**Provenance**
Join	p.p
Union	p + p
Projections	p + p
Select	0/1 based on the selection
AND	p ∧ p
OR	p ∨ p

IMPLEMENTATION:

Our project has been meticulously executed using Java. Postgres, a potent open-source relational database management system, was integrated to support our complex data structures and activities, ensuring the safekeeping and retrieval of our annotated relations.

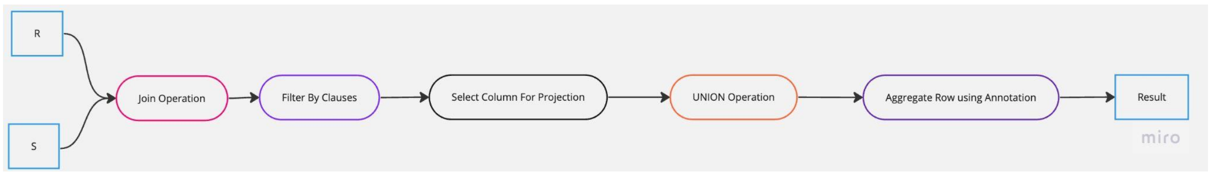
BASIC FLOW:

The program consists of the

1. Main class,
2. Query class,
3. Union class,
4. Table class,
5. Row class and
6. Clause class

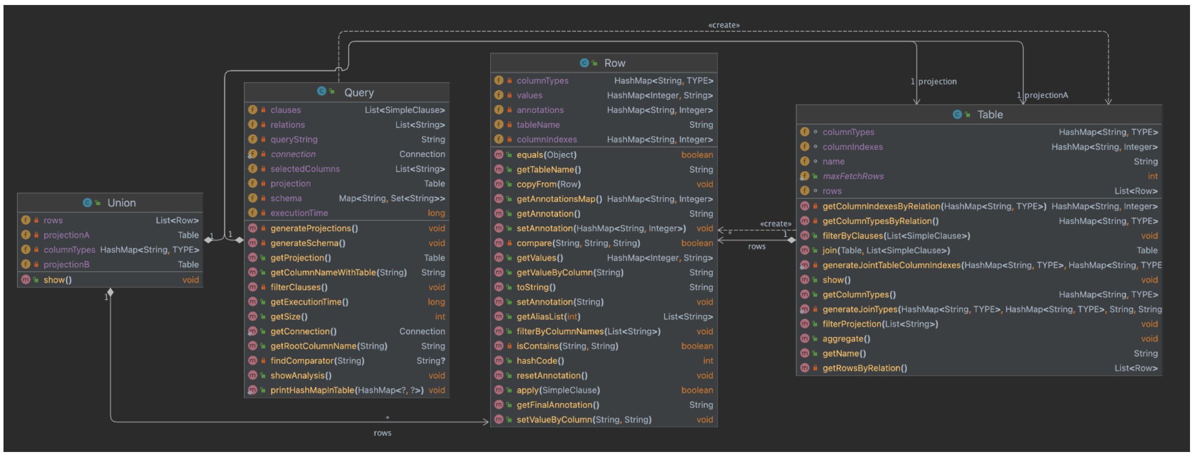
The Query class, which handles SQL queries, and the Table class, which represents database tables, are the two main parts. The application starts by analysing the incoming SQL query, extracting the required data, and creating the proper projections and schema. Using JDBC, it creates a connection to the PostgreSQL database.

The program creates schema information, filters clauses, and joins tables when necessary in the query class. Following query processing, it creates a projection, filters data using the WHERE clauses, and aggregates the outcomes. The Table class is in charge of managing certain database tables, together with their columns and rows.



The application divides the input query into two pieces, processes each separately, and then uses the Union class to aggregate the outcomes for union operations and then generates the annotations based on the query provided.

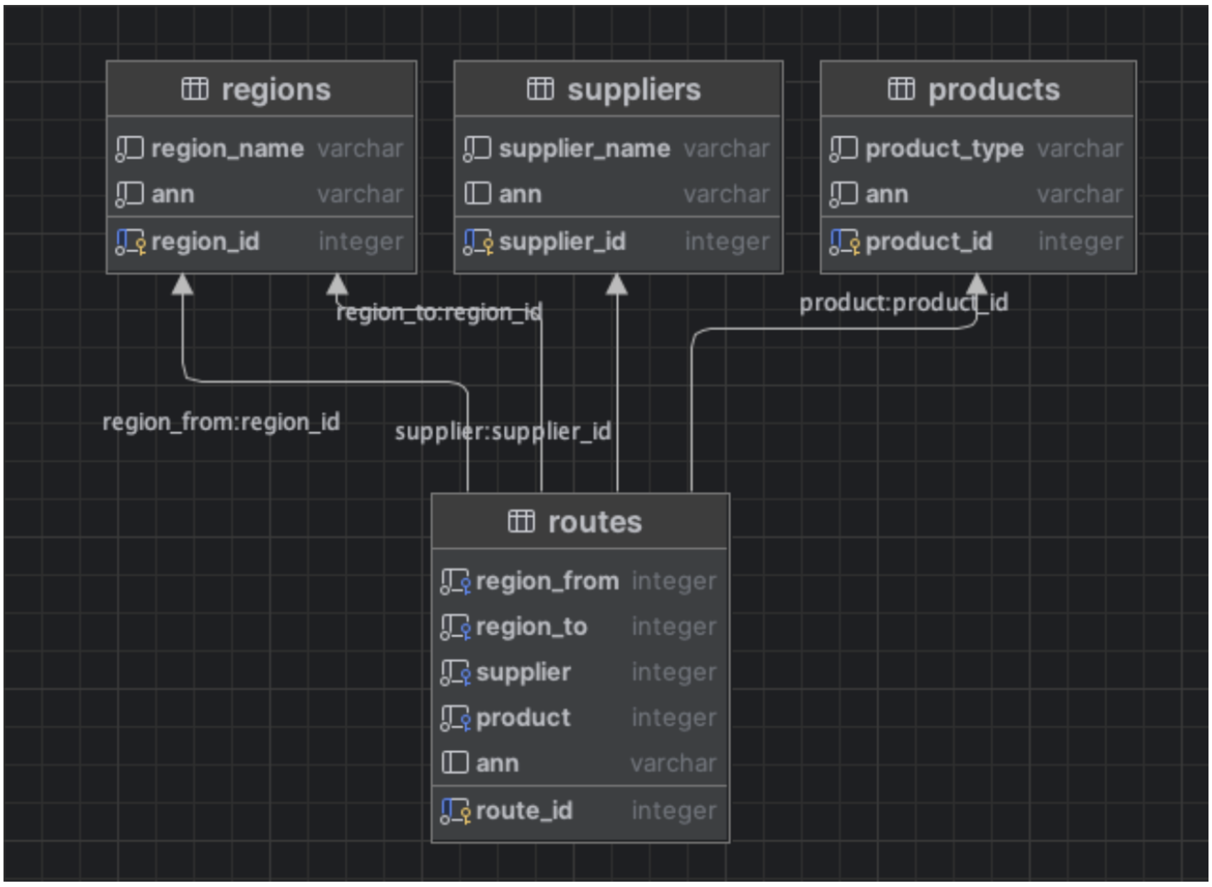
The UML diagram for our solution is:



«create»
Query
clauses List<SimpleClause>
relations List<String>
queryString String
connection Connection
selectedColumns List<String>
projection Table
schema Map<String, Set<String>>
executionTime long
generateProjections() void
generateSchema() void
getProjection() Table
getColumnNameWithTable(String) String
filterClauses() void
getExecutionTime() long
getSize() int
getConnection() Connection
getRootColumnName(String) String
findComparator(String) String?
showAnalysis() void
printHashMapInTable(HashMap<?, ?>) void
Union
rows List<Row>
projectionA Table
columnTypes HashMap<String, TYPE>
projectionB Table
show() void
1
1
1
*
rows
Row
columnTypes HashMap<String, TYPE>
values HashMap<Integer, String>
annotations HashMap<String, Integer>
tableName String
columnIndexes HashMap<String, Integer>
equals(Object) boolean
getTableName() String
copyFrom(Row) void
getAnnotationsMap() HashMap<String, Integer>
getAnnotation() String
setAnnotation(HashMap<String, Integer>) void
compare(String, String, String) boolean
getValues() HashMap<Integer, String>
getValueByColumn(String) String
toString() String
setAnnotation(String) void
getAliasList(int) List<String>
filterByColumnNames(List<String>) void
isContains(String, String) boolean
hashCode() int
resetAnnotation() void
apply(SimpleClause) boolean
getFinalAnnotation() String
setValueByColumn(String, String) void
«create»
*
1
rows
1 projection
1 projectionA
Table
columnTypes HashMap<String, TYPE>
columnIndexes HashMap<String, Integer>
name String
maxFetchRows int
rows List<Row>
getColumnIndexesByRelation(HashMap<String, TYPE>) HashMap<String, Integer>
getColumnTypesByRelation() HashMap<String, TYPE>
filterByClauses(List<SimpleClause>) void
join(Table, List<SimpleClause>) Table
generateJointTableColumnIndexes(HashMap<String, TYPE>, HashMap<String, TYPE
show() void
getColumnTypes() HashMap<String, TYPE>
generateJoinTypes(HashMap<String, TYPE>, HashMap<String, TYPE>, String, String
filterProjection(List<String>) void
aggregate() void
getName() String
getRowsByRelation() List<Row>

<u>DATA:</u>

For the following queries we have used the sample data provided to us in sample.sql file:
We have the following schema for the data:



QUERY 1:

```
select supplier_name from suppliers, routes where supplier = suppliers.supplier_id
AND region_to = 5
```

EXPLANATION:

The query asks to join the tables suppliers and routes based on the supplier_id which is further filtered by region_ to = 5. It returns the supplier_name as an output column.

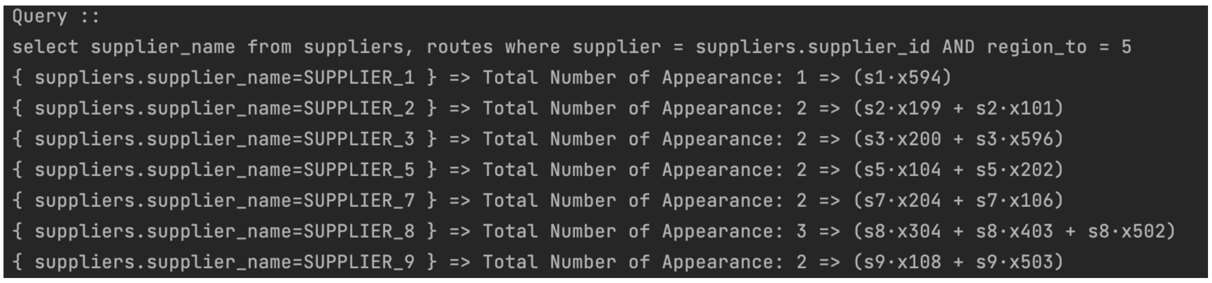
PROGRAM EXECUTION:

In our solution, the application joins the tables for "suppliers" and "routes" and merges them using a shared "supplier_id." The annotation for this particular operation is added to an ArrayList during this join, resulting in an annotation pattern similar to sN.xN, where 'sN' stands for the supplier details and 'xN' for the related route information. The integrated data's provenance is made obvious by the concatenated annotation. The software then filters the combined tuples after the join, choosing only those where the 'region_to' attribute equals 5. Tuples having an annotation value of 1 are specially picked, and a selection criterion based on the annotation is also used. The program not only consolidates relevant data from the 'suppliers' and 'routes' columns through a filtering method but also provides a comprehensive record of the data's production process. The programme ensures that the produced tuples match particular requirements by emphasising both the region criterion and the annotation value, improving the accuracy of the outcomes. This thorough method of selecting and processing data demonstrates the application's capacity to manage difficult relational operations while keeping a clear and transparent record of the data's origin and changes.

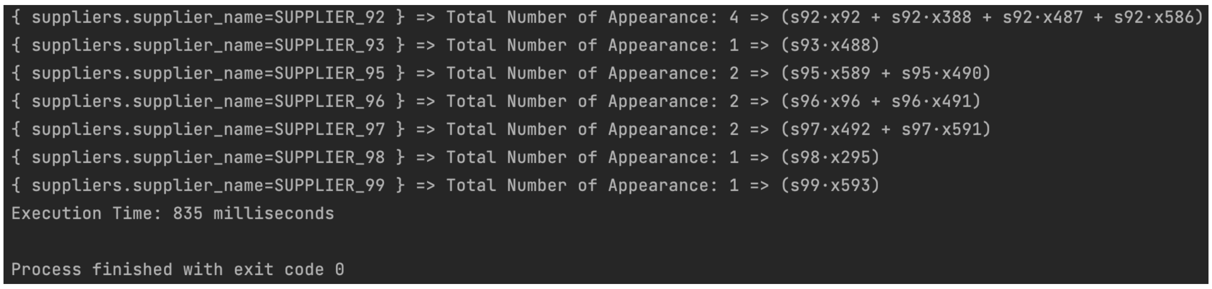
EXECUTION TIME:

The execution time for the entire query is 835 ms.

OUTPUT:



```
Query ::
select supplier_name from suppliers, routes where supplier = suppliers.supplier_id AND region_to = 5
{ suppliers.supplier_name=SUPPLIER_1 } => Total Number of Appearance: 1 => (s1·x594)
{ suppliers.supplier_name=SUPPLIER_2 } => Total Number of Appearance: 2 => (s2·x199 + s2·x101)
{ suppliers.supplier_name=SUPPLIER_3 } => Total Number of Appearance: 2 => (s3·x200 + s3·x596)
{ suppliers.supplier_name=SUPPLIER_5 } => Total Number of Appearance: 2 => (s5·x104 + s5·x202)
{ suppliers.supplier_name=SUPPLIER_7 } => Total Number of Appearance: 2 => (s7·x204 + s7·x106)
{ suppliers.supplier_name=SUPPLIER_8 } => Total Number of Appearance: 3 => (s8·x304 + s8·x403 + s8·x502)
{ suppliers.supplier_name=SUPPLIER_9 } => Total Number of Appearance: 2 => (s9·x108 + s9·x503)
```



```
{ suppliers.supplier_name=SUPPLIER_92 } => Total Number of Appearance: 4 => (s92·x92 + s92·x388 + s92·x487 + s92·x586)
{ suppliers.supplier_name=SUPPLIER_93 } => Total Number of Appearance: 1 => (s93·x488)
{ suppliers.supplier_name=SUPPLIER_95 } => Total Number of Appearance: 2 => (s95·x589 + s95·x490)
{ suppliers.supplier_name=SUPPLIER_96 } => Total Number of Appearance: 2 => (s96·x96 + s96·x491)
{ suppliers.supplier_name=SUPPLIER_97 } => Total Number of Appearance: 2 => (s97·x492 + s97·x591)
{ suppliers.supplier_name=SUPPLIER_98 } => Total Number of Appearance: 1 => (s98·x295)
{ suppliers.supplier_name=SUPPLIER_99 } => Total Number of Appearance: 1 => (s99·x593)
Execution Time: 835 milliseconds

Process finished with exit code 0
```

QUERY 2:

```
SELECT product_type, region_name FROM products, regions where
origin_region = regions.region_id;
```

EXPLANATION:

The query returns product_type and region_name columns after joining the tables based on the origin_region from the products table and region_id from the regions table.

PROGRAM EXECUTION:

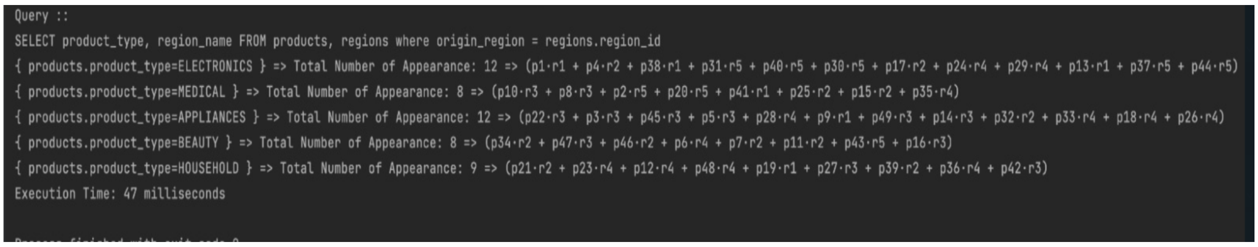
In the program, for this query first of all the join function is performed based on the shared attributes "origin_region" and "region_id," the application first executes an implicit join operation between the "products" and "regions" tables. The programme annotates the merged tuples with provenance data, designated as pN.xN , throughout this join process, where 'pN' stands for product information and 'xN' for information about the region.

The program filters the merged tuples after the join using the constraint origin_region = regions.region_id. Only tuples whose origin_region attribute matches the region_id are preserved as a result of this filtering step. The application then examines the selected tuples, paying particular attention to the attributes "product_type" and "region_name." The tuples with the same product_types will be merged leading to merging of the annotations using '+'. The output is then displayed, showing the values for "product_type" and "region_name" that adhere to the join and filtering criteria.

EXECUTION TIME:

The execution time for the query was 47ms.

OUTPUT:



```
Query ::
SELECT product_type, region_name FROM products, regions where origin_region = regions.region_id
{ products.product_type=ELECTRONICS } => Total Number of Appearance: 12 => (p1·r1 + p4·r2 + p38·r1 + p31·r5 + p40·r5 + p30·r5 + p17·r2 + p24·r4 + p29·r4 + p13·r1 + p37·r5 + p44·r5)
{ products.product_type=MEDICAL } => Total Number of Appearance: 8 => (p10·r3 + p8·r3 + p2·r5 + p20·r5 + p41·r1 + p25·r2 + p15·r2 + p35·r4)
{ products.product_type=APPLIANCES } => Total Number of Appearance: 12 => (p22·r3 + p3·r3 + p45·r3 + p5·r3 + p28·r4 + p9·r1 + p49·r3 + p14·r3 + p32·r2 + p33·r4 + p18·r4 + p26·r4)
{ products.product_type=BEAUTY } => Total Number of Appearance: 8 => (p34·r2 + p47·r3 + p46·r2 + p6·r4 + p7·r2 + p11·r2 + p43·r5 + p16·r3)
{ products.product_type=HOUSEHOLD } => Total Number of Appearance: 9 => (p21·r2 + p23·r4 + p12·r4 + p48·r4 + p19·r1 + p27·r3 + p39·r2 + p36·r4 + p42·r3)
Execution Time: 47 milliseconds
```

QUERY 3:

```
SELECT product_type FROM products, routes where product = products.product_id AND
region_from in (1, 2);
```

EXPLANATION:

The query returns the product_type column after joining the tables products and routes given the conditions that the product column of the routes matches the product_id in the product table also the region_from is either 1 or 2.

PROGRAM EXECUTION:

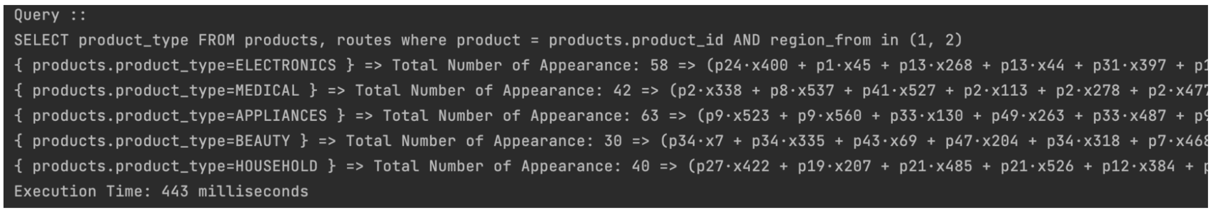
The join operation is performed first on the tables “product” and “routes” based on the common attribute “product_id”. The program performs the annotations for the join operation and converts them into the form of product indicating that the tuple formed is obtained by the joining of the two tables. It is in the form of pN.xN where, pN is the annotation for “products” table and xN is the annotation for the “routes” table.

The program then filters the tuples that have region_from either 1 or 2. Only the tuples for which the region_from matches either 1 or 2 are taken as a result of the query. The tuples with the same product type are merged by using the “+” operator. The output is then displayed showing the values for “product_types” , total appearance in the output and their annotation.

EXECUTION TIME:

The query is executed in 443 ms.

OUTPUT:

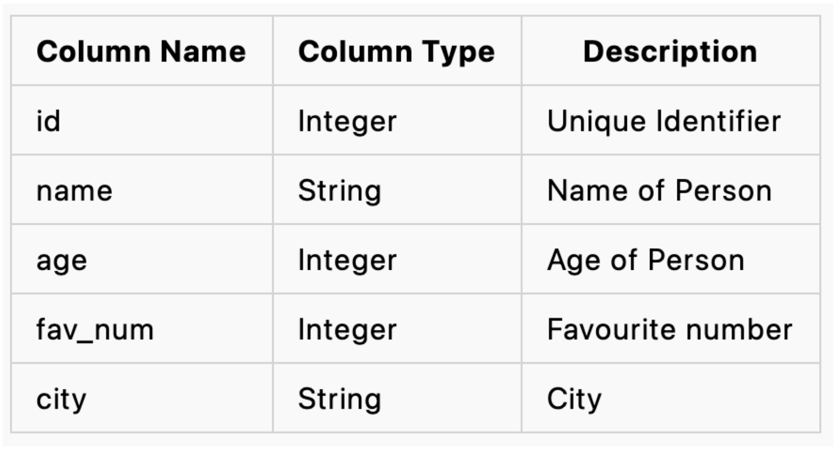


```
Query ::
SELECT product_type FROM products, routes where product = products.product_id AND region_from in (1, 2)
{ products.product_type=ELECTRONICS } => Total Number of Appearance: 58 => (p24·x400 + p1·x45 + p13·x268 + p13·x44 + p31·x397 + p1
{ products.product_type=MEDICAL } => Total Number of Appearance: 42 => (p2·x338 + p8·x537 + p41·x527 + p2·x113 + p2·x278 + p2·x47
{ products.product_type=APPLIANCES } => Total Number of Appearance: 63 => (p9·x523 + p9·x560 + p33·x130 + p49·x263 + p33·x487 + p9
{ products.product_type=BEAUTY } => Total Number of Appearance: 30 => (p34·x7 + p34·x335 + p43·x69 + p47·x204 + p34·x318 + p7·x46
{ products.product_type=HOUSEHOLD } => Total Number of Appearance: 40 => (p27·x422 + p19·x207 + p21·x485 + p21·x526 + p12·x384 + p
Execution Time: 443 milliseconds
```

SCALABILITY OF THE ALGORITHM:

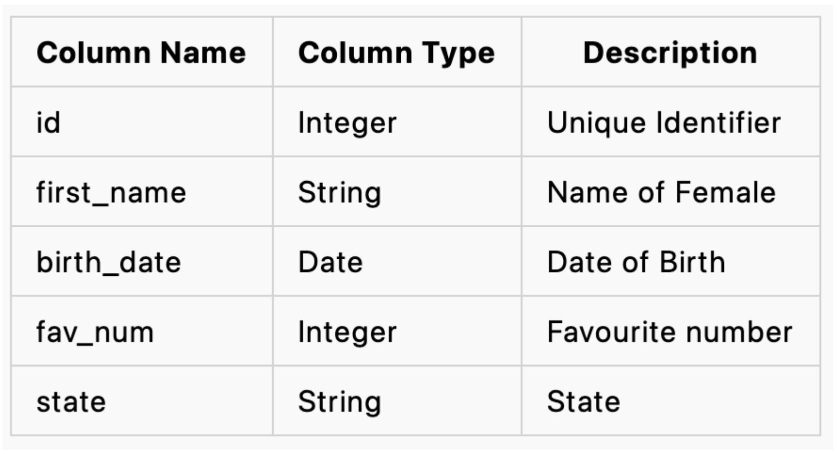
Here we have taken two different relations, to perform scalability of the algorithm.The schema for the relations are as following:

Table R



Column Name	Column Type	Description
id	Integer	Unique Identifier
name	String	Name of Person
age	Integer	Age of Person
fav_num	Integer	Favourite number
city	String	City

Table S



Column Name	Column Type	Description
id	Integer	Unique Identifier
first_name	String	Name of Female
birth_date	Date	Date of Birth
fav_num	Integer	Favourite number
state	String	State

Queries:

We have tested the algorithm on different instances of the query as follow:

Query 1:

```
SELECT s.birthdate FROM r, s WHERE r.fav_num > 5 AND s.fav_num < 100000;
```

EXPLANATION:

This query selects the birthdate column from the table s by performing a join operation between tables r and s. It retrieves rows where the fav_num in table r is greater than 5 and the fav_num in table s is less than 100,000. This query is likely interested in retrieving birthdates of individuals who meet specific criteria related to their favourite numbers. Understanding its performance on different dataset sizes is essential for optimising the retrieval of birthdates based on these conditions.

Query 2:

```
SELECT r.name, s.birthdate,  FROM r, s WHERE s.fav_num > 6522;
```

EXPLANATION:

This query retrieves the name column from table r and the birthdate column from table s. It performs a join between r and s to select rows where the fav_num in table s is greater than 6522. This query likely seeks to identify individuals with specific favourite numbers and retrieve their names and birthdates. Analysing its execution time across different dataset sizes can help determine how efficiently this information is retrieved.

Query 3:

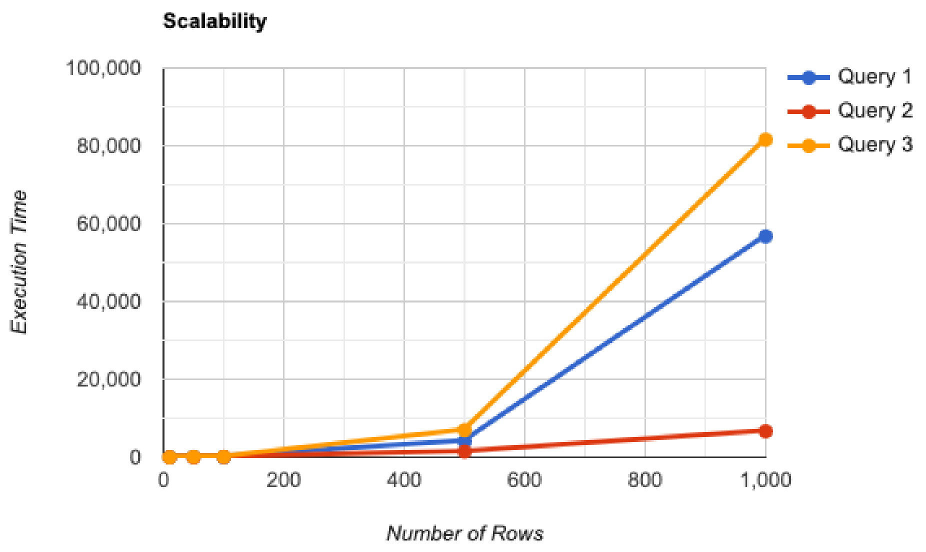
```
SELECT r.name, s.first_name FROM r, s WHERE s.fav_num IN ( 646495, 397375, 956267,
680928, 341340, 638124, 595823, 372669 ) AND s.state in ( 'RhodeIsland', 'Florida',
'Kentucky', 'WestVirginia', 'Ohio', 'Iowa', 'WestVirginia' );
```

EXPLANATION:

This query selects the name column from table r and the first_name column from table s. It performs a join between r and s to retrieve rows where the fav_num in table s matches a list of specified values, and the state in table s matches another list of specified values. This query appears to be looking for individuals with specific favourite numbers and in specific states. The query execution time can help assess the efficiency of finding and returning this combination of information.

Execution Time:

No of rows	Query 1 (ms)	Query 2 (ms)	Query 3 (ms)
10	25	33	28
50	74	32	114
100	48	69	113
500	4058	1423	6953
1000	56720	6699	81591



Limitation:

Our device couldn't handle a query with 10,000 rows in a table due to resource limitation. When we tried processing 1,000 rows, it took about 80 seconds. But when we added 10,000 rows and executed the query, our device ran out of memory and gave an OutOfMemory error.