Faculty of Engineering of the University of Porto

Project 2

Object-Relational Schema

M.EIC028 - Database Technologies (TBD)

1MEIC02

Professor

Mariana Curado Malta

Students

Diogo Alexandre da Costa Melo Moreira da Fonte - up20204175@edu.fe.up.pt

Nuno André Gomes França - up201807530@edu.fe.up.pt

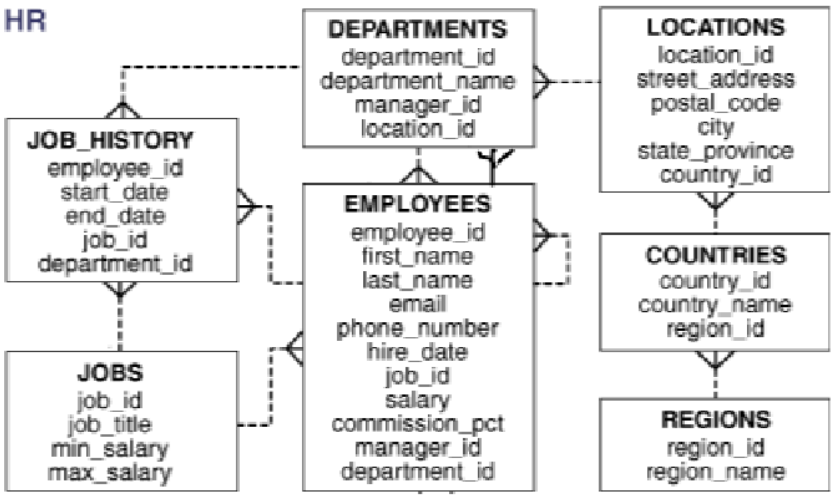
1. Introduction

This assignment explores Object-Relational (OR) databases, which blend traditional relational models with object-oriented concepts. We focus on leveraging SQL3 extensions to craft a compact HR database for multinational companies. Our aim is to understand and utilize OR database features like user-defined types, inheritance, nested tables, and sorting methods. By integrating these, we create a flexible schema for modern HR systems.

Throughout this report, we detail the design, implementation, and use of the OR schema. We demonstrate how SQL3 extensions seamlessly enhance real-world applications. Our objectives include designing the OR data model, populating it with relational data, implementing query-enhancing methods, and executing diverse queries. These span basic tasks like employee counts to complex analyses such as job history gaps and salary comparisons across countries.

This assignment showcases the potential of OR databases, using as an example a HR system database of a multinational company. We seek to grasp their advantages and influence on future database management practices.

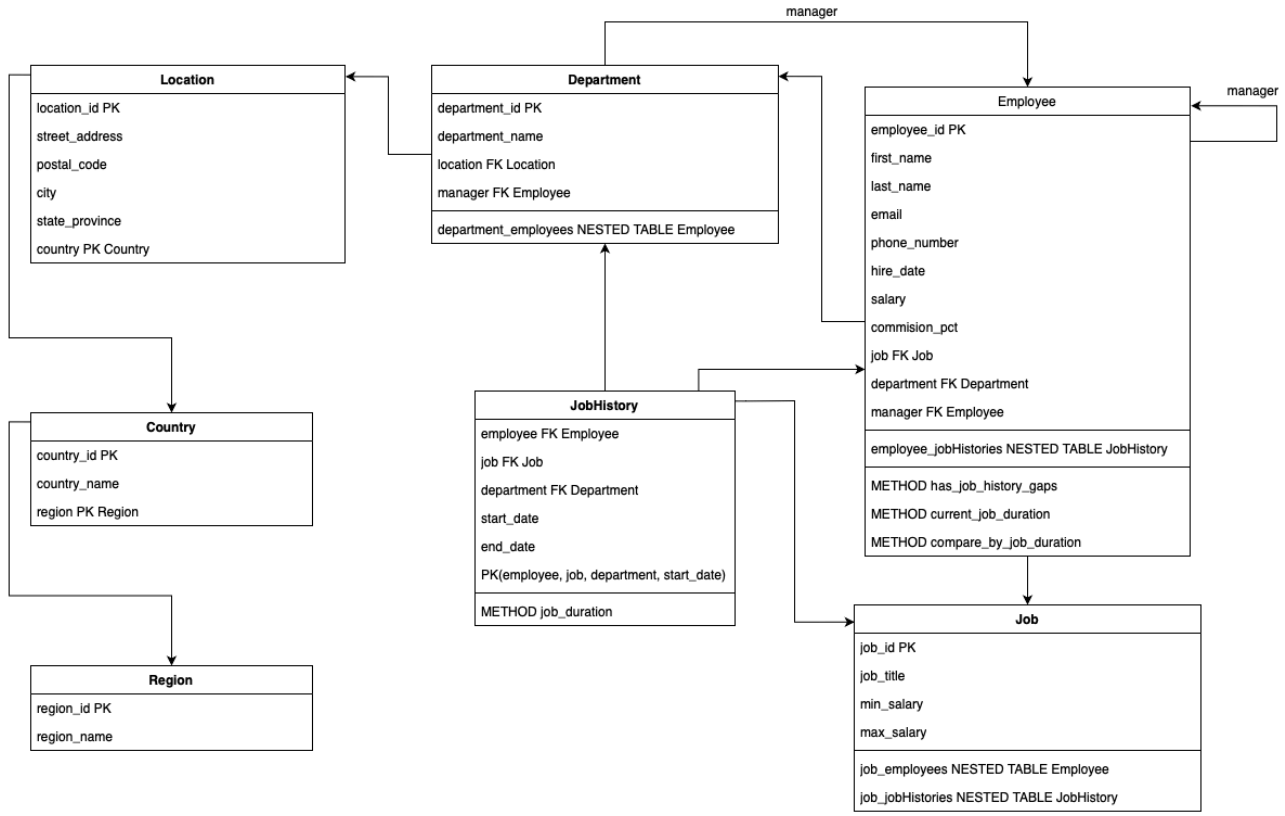
The relational data model currently designed and used in the management system is as follows:



(figure 1 - Relational Model)

2. Object-Relational Data Model

We were asked to implement an Object-Relational Data Model for a Human Resources system. Here is a relational diagram that represents our final structure:



(figure 2 - Object-Relational Model Diagram)

This diagram is accessible, with better quality, in the project zip in the folder /data_model (file object-relational-data-model-diagram.png).

In designing the object-relational model for the HR system, several key decisions were made to leverage the features of SQL3 extensions. This model aims to efficiently represent and manage the complex relationships and hierarchical data within a multinational company's human resources structure. Below is an explanation of the design decisions made for each part of the schema.

2.1. Object Types for Geographical Entities

Region, Country, Location:

- region_t Type: Defines the geographical regions within which the company operates. Each region has a unique ID and a name.
- country_t Type: Represents countries, each associated with a region. It includes a region reference to maintain the hierarchical relationship.
- location_t Type: Details the specific locations (offices) in each country, containing address fields and a reference to the country it belongs to.

Justification: These object types ensure a clear hierarchical representation of geographical data, facilitating the management and querying of locations based on their regional and country contexts.

2.2. Job and Department Structure

Job:

- job_t Type: Encapsulates job-related information such as job ID, title, and salary range. Nested tables job_employees and job_jobHistories are added to maintain references to employees and job history records associated with each job.

Department:

- department_t Type: Defines departments with attributes for ID, name, and location reference. It includes a nested table department_employees to store references to employees within the department and a reference to the department manager.

Justification: The nested table structure within jobs and departments allows efficient management of hierarchical data, enabling the grouping of employees and job history records under their respective jobs and departments. This structure enhances query performance and maintains clear relationships.

2.3. Employee Structure

Employee:

- employee_t Type: Represents employees with personal data, job details, department affiliation, and managerial relationships. The manager attribute is a self-referential reference, and a nested table employee_jobHistories keeps track of each employee's job history.

Justification: The employee object type consolidates personal, job, and hierarchical management data, allowing a single entity to encapsulate all relevant information. The self-referential manager attribute supports organizational hierarchy, and the nested job history table enables efficient tracking of job transitions over time.

2.4. Jobs Historical Records

Job History:

- jobHistory_t Type: Contains references to employees, jobs, and departments, along with start and end dates for each job record.

Justification: This type is essential for tracking the job history of employees, providing a detailed temporal record of job assignments. It supports comprehensive historical analysis and ensures data integrity by linking directly to employee, job, and department entities.

2.5. Nested Tables

Nested tables are a powerful feature in the object-relational model, particularly useful for representing complex relationships and hierarchies within a database. In our HR system design, nested tables play a crucial role in maintaining data integrity and facilitating efficient data retrieval.

What Are Nested Tables?

Nested tables allow you to store collections of data directly within a column of a table. This feature is particularly useful for representing one-to-many relationships, where an entity can have multiple associated records. In the context of SQL3, nested tables are implemented as collections within object types and can be used to store references to other object instances.

2.5.1. Department Employees

Definition:

```
CREATE TYPE department_employees_tab_t AS TABLE OF REF employee_t;
ALTER TYPE department_t ADD ATTRIBUTE department_employees
department_employees_tab_t CASCADE;
```

This nested table within the department_t type stores references to employees belonging to a specific department.

Table Creation:

```
CREATE TABLE dw_departments OF department_t(
    department_id PRIMARY KEY,
    department_name NOT NULL,
    location NULL,
    manager NULL
) NESTED TABLE department_employees STORE AS department_employees_tab;
```

Importance:

The nested table department_employees allows for direct access to the list of employees in each department. This structure supports efficient retrieval of all employees within a department, simplifying queries and management of department-level employee data.

2.5.2. Employee Job Histories

Definition:

```
CREATE TYPE employee_jobHistories_tab_t AS TABLE OF REF jobHistory_t;
ALTER TYPE employee_t ADD ATTRIBUTE employee_jobHistories
employee_jobHistories_tab_t CASCADE;
```

This nested table within the employee_t type stores job history records for each employee.

Table Creation:

```
CREATE TABLE dw_employees OF employee_t(
    employee_id PRIMARY KEY,
    first_name NULL,
    last_name NULL,
```

```
    email NULL,
    phone_number NULL,
    hire_date NULL,
    salary NULL,
    commission_pct NULL,
    job NULL,
    department NULL,
    manager NULL
) NESTED TABLE employee_jobHistories STORE AS employee_jobHistories_tab;
```

Importance:

The nested table employee_jobHistories maintains a detailed record of each employee's job history. This structure ensures that job transitions are accurately tracked over time, facilitating historical analyses and ensuring that queries about an employee's career path are straightforward and efficient.

2.5.3. Job Employees and Job Histories

Definitions:

```
CREATE TYPE job_employees_tab_t AS TABLE OF REF employee_t;
ALTER TYPE job_t ADD ATTRIBUTE job_employees job_employees_tab_t CASCADE;

CREATE TYPE job_jobHistories_tab_t AS TABLE OF REF jobHistory_t;
ALTER TYPE job_t ADD ATTRIBUTE job_jobHistories job_jobHistories_tab_t
CASCADE;
```

Table Creation:

```
CREATE TABLE dw_jobs OF job_t(
    job_id PRIMARY KEY,
    job_title NOT NULL,
    min_salary NOT NULL,
    max_salary NOT NULL
) NESTED TABLE job_employees STORE AS job_employees_tab
  NESTED TABLE job_jobHistories STORE AS job_jobHistories_tab;
```

Importance:

The nested tables job_employees and job_jobHistories within the job_t type facilitate direct association of employees and job history records with specific jobs. This design supports

efficient management of job-related data, allowing for quick retrieval of all employees in a job and the retrieval of all job histories of each job.

2.5.4. Benefits of Using Nested Tables

- **Data Integrity:**

Nested tables enforce referential integrity by ensuring that related data remains consistent. For example, an employee cannot exist in a department without a valid reference, which is enforced by the nested table structure.

- **Efficiency:**

Queries involving hierarchical data are more efficient because the nested table structure keeps related data together. This reduces the need for complex joins and improves performance for common queries.

- **Simplicity:**

Managing related data as nested tables simplifies the schema and makes it more intuitive. For instance, all job history records for an employee are easily accessible through a single attribute, making the data model easier to understand and work with.

- **Flexibility:**

Nested tables provide flexibility in handling one-to-many relationships. They allow the schema to evolve by adding new attributes and relationships without major restructuring of the database.

2.6. Conclusion

This object-relational model effectively combines the strengths of relational and object-oriented paradigms. By using SQL3 extensions, it provides a robust and flexible schema capable of handling the intricate data relationships and structures inherent in a multinational company's HR system. The decisions made in this design aim to optimize data integrity, query performance, and ease of management, showcasing the transformative potential of object-relational databases in modern database systems.

The complete script to create the data model is accessible in the project zip in the folder /scripts/object_relationa_model (file scriptCreateORSchema.sql).

3. Useful SQL Methods

Our SQL methods choice was based on the questions presented in the assignment. We also added two more methods, to help doing updates or inserts in the database. Our methods are the following:

A) Calculate the Job Duration.

B) Check for Time Gaps in Job History.

C) Calculate the Duration of an Employee's Current Job.

D) Compare Employees by Job Duration.

E) Insert a New Employee

F) Updating Employee Salary

3.1. Method A

Query: Calculate the Job Duration.

SQL Code:

```
CREATE OR REPLACE TYPE BODY jobHistory_t AS
  MEMBER FUNCTION job_duration RETURN NUMBER IS
  BEGIN
    RETURN end_date - start_date;
  END job_duration;
END;
```

Explanation:

The job_duration method calculates the duration of a specific job history entry by subtracting the start date from the end date. This method is defined within the jobHistory_t type and returns the number of days the job lasted.

Importance:

This method is crucial for analyzing the tenure of employees in their various roles. By providing a simple way to calculate the duration of each job, it helps HR professionals and managers understand the length of time employees spend in specific positions. This information can be used for performance evaluations, identifying career progression patterns, and detecting anomalies in job history records.

3.2. Method B

Query: Check for Time Gaps in Job History.

SQL Code:

```
CREATE OR REPLACE TYPE BODY employee_t AS
  MEMBER FUNCTION has_job_history_gaps RETURN VARCHAR2 IS
    prev_end_date DATE := NULL;
    gap_found VARCHAR2(5) := 'FALSE';
  BEGIN
    FOR i IN 1..SELF.employee_jobHistories.COUNT LOOP
      IF prev_end_date IS NOT NULL AND
SELF.employee_jobHistories(i).start_date > prev_end_date + 1 THEN
        gap_found := 'TRUE';
        EXIT;
      END IF;
      prev_end_date := SELF.employee_jobHistories(i).end_date;
    END LOOP;
    RETURN gap_found;
  END has_job_history_gaps;
END;
```

Explanation:

The has_job_history_gaps method checks whether there are any time gaps in an employee's job history. It iterates through the job histories of an employee and compares the end date of one job with the start date of the next job. If a gap is found (i.e., if the start date of the next job is more than one day after the end date of the previous job), the method returns 'TRUE'; otherwise, it returns 'FALSE'.

Importance:

This method ensures data integrity by identifying any periods where an employee was not assigned to any job. Gaps in job history can indicate potential errors in data entry or other issues that need to be addressed. It helps maintain an accurate and complete employment record, which is essential for compliance, auditing, and making informed HR decisions.

3.3. Method C

Query: Calculate the Duration of an Employee's Current Job.

SQL Code:

```
CREATE OR REPLACE TYPE BODY employee_t AS
  MEMBER FUNCTION current_job_duration RETURN NUMBER IS
    current_job_start DATE;
  BEGIN
    IF SELF.employee_jobHistories.COUNT > 0 THEN
      current_job_start :=
SELF.employee_jobHistories(SELF.employee_jobHistories.COUNT).end_date + 1;
    ELSE
      current_job_start := SELF.hire_date;
    END IF;
    RETURN SYSDATE - current_job_start;
  END current_job_duration;
END;
```

Explanation:

The current_job_duration method calculates the duration of the employee's current job by determining the start date of the current job and subtracting it from the current date (SYSDATE). If the employee has job history records, it uses the end date of the last job history entry; otherwise, it uses the hire date.

Importance:

This method provides a metric for evaluating how long an employee has been in their current role. This information is valuable for performance evaluations, career planning, and identifying employees with significant tenure in their current positions. It also helps in understanding employee stability and job satisfaction within the organization.

3.4. Method D

Query: Compare Employees by Job Duration.

SQL Code:

```
CREATE OR REPLACE TYPE BODY employee_t AS
  ORDER MEMBER FUNCTION compare_by_job_duration (e employee_t) RETURN
INTEGER IS
  BEGIN
    IF SELF.current_job_duration < e.current_job_duration THEN
      RETURN -1;
    ELSIF SELF.current_job_duration > e.current_job_duration THEN
      RETURN 1;
    ELSE
      RETURN 0;
    END IF;
  END compare_by_job_duration;
END;
```

Explanation:

The compare_by_job_duration method compares two employees based on the duration of their current jobs. It returns -1 if the calling employee's job duration is less than the compared employee's job duration, 1 if it is greater, and 0 if they are equal. This method is useful for sorting employees by their current job duration.

Importance:

This method facilitates sorting employees by their current job duration, which is helpful for various analyses, including identifying those with the longest tenure in their current roles and prioritizing evaluations. It supports more sophisticated and meaningful queries, enhancing the ability to make data-driven decisions in HR management.

3.5. Method E

Function: Method to Add a New Employee

SQL Code:

```
CREATE OR REPLACE PROCEDURE add_employee (
    p_employee_id IN NUMBER,
    p_first_name IN VARCHAR2,
    p_last_name IN VARCHAR2,
    p_email IN VARCHAR2,
    p_phone_number IN VARCHAR2,
    p_hire_date IN DATE,
    p_salary IN NUMBER,
    p_commission_pct IN NUMBER,
    p_job_id IN VARCHAR2,
    p_department_id IN NUMBER,
    p_manager_id IN NUMBER
) IS
    v_job REF job_t;
    v_department REF department_t;
    v_manager REF employee_t;
BEGIN
    SELECT REF(j) INTO v_job FROM dw_jobs j WHERE j.job_id = p_job_id;
    SELECT REF(d) INTO v_department FROM dw_departments d WHERE
d.department_id = p_department_id;
    SELECT REF(e) INTO v_manager FROM dw_employees e WHERE e.employee_id =
p_manager_id;

    INSERT INTO dw_employees VALUES (
        p_employee_id, p_first_name, p_last_name, p_email, p_phone_number,
p_hire_date,
        p_salary, p_commission_pct, v_job, v_department, v_manager,
employee_jobHistories_tab_t()
    );

    COMMIT;
END;
```

Explanation:

This procedure add_employee inserts a new employee record into the dw_employees table. It takes several parameters to set the attributes of the new employee, including their job, department, and manager. The procedure uses object references (REF) to link the employee to their job, department, and manager.

Importance:

This method simplifies the process of adding new employees by encapsulating the logic in a procedure. It ensures that new employees are correctly linked to their jobs, departments, and managers, maintaining referential integrity and making the database easier to manage.

3.6. Method F

Function: Method to Update Employee Salary

SQL Code:

```
CREATE OR REPLACE PROCEDURE update_employee_salary (
    p_employee_id IN NUMBER,
    p_new_salary IN NUMBER
) IS
BEGIN
    UPDATE dw_employees SET salary = p_new_salary WHERE employee_id =
p_employee_id;
    COMMIT;
END;
```

Explanation:

This procedure update_employee_salary updates the salary of an existing employee. It takes the employee's ID and the new salary as parameters and updates the corresponding record in the dw_employees table.

Importance:

This method provides a straightforward way to update employee salaries, ensuring that salary adjustments can be made efficiently and consistently. It is particularly useful for payroll processing and managing employee compensation.

4. Queries Analysis

In this section, we are going to approach each question of the assignment, analyzing the retrieved results, with our queries, and the comparison between the executions in the relational and object-relational models.

4.1. Question A

Description: Calculate the total number of employees that each department has got

SQL Query:

```
SELECT d.department_name,
       (SELECT COUNT(*)
        FROM TABLE(d.department_employees)) AS total_employees
FROM dw_departments d;
```

Retrieved Results:

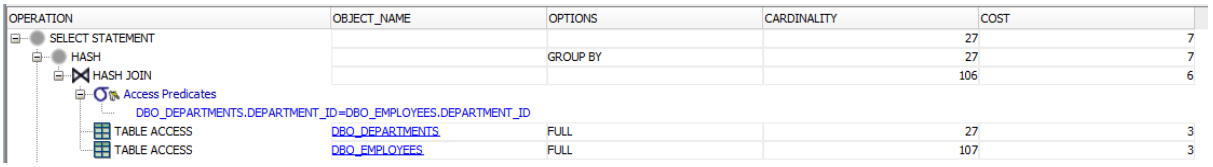
DEPARTMENT_NAME	TOTAL_EMPLOYEES
Administration	1
Marketing	2
Purchasing	6
Human Resources	1
Shipping	45
IT	5
Public Relations	1
Sales	34
Executive	3
Finance	6
Accounting	2
Treasury	0
Corporate Tax	0
Control And Credit	0
Shareholder Services	0
Benefits	0
Manufacturing	0
Construction	0
Contracting	0
Operations	0
IT Support	0
NOC	0
IT Helpdesk	0
Government Sales	0
Retail Sales	0
Recruiting	0
Payroll	0

(figure 3 - Retrieved Results for Query A)

Execution Data:

Relational Model:

Query Execution Time: 0.066 seconds

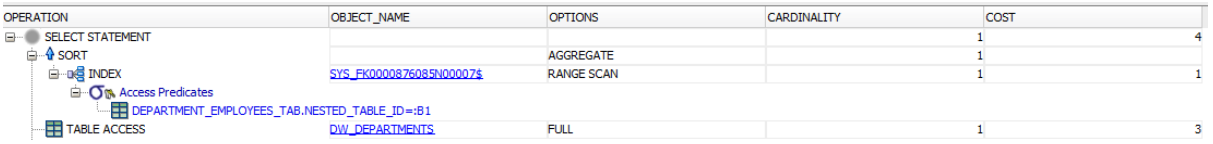


OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			27	7
HASH		GROUP BY	27	7
HASH JOIN			106	6
Access Predicates				
DBO_DEPARTMENTS.DEPARTMENT_ID=DBO_EMPLOYEES.DEPARTMENT_ID				
TABLE ACCESS	DBO_DEPARTMENTS	FULL	27	3
TABLE ACCESS	DBO_EMPLOYEES	FULL	107	3

(figure 4 - Explain Plan of Relational Model for Query A)

Object-Relational Model:

Query Execution Time: 0.067 seconds



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	4
SORT		AGGREGATE	1	
INDEX	SYS_FK0000876085N00007$	RANGE SCAN	1	1
Access Predicates				
DEPARTMENT_EMPLOYEES_TAB.NESTED_TABLE_ID=:B1				
TABLE ACCESS	DW_DEPARTMENTS	FULL	1	3

(figure 5 - Explain Plan of Object-Relational for Query A)

4.2. Question B

Description: In each department, how many employees are there for each job title?

SQL Query:

```
SELECT d.department_name, e.job.job_title AS JOB_title,
COUNT(e.employee_id) AS total_employees
FROM dw_departments d
JOIN dw_employees e ON e.department = REF(d)
GROUP BY d.department_name, e.job.job_title;
```

Retrieved Results:

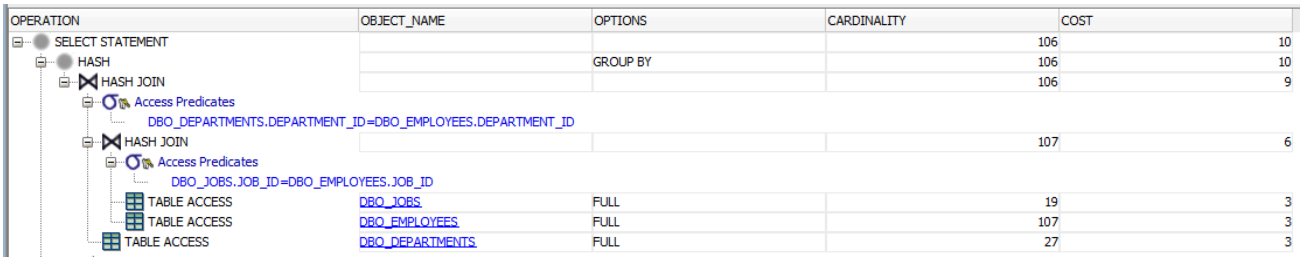
DEPARTMENT_NAME	JOB_TITLE	TOTAL_EMPLOYEES
Administration	Administration Assistant	1
Marketing	Marketing Manager	1
Finance	Accountant	5
Finance	Finance Manager	1
Purchasing	Purchasing Clerk	5
Shipping	Stock Clerk	20
Public Relations	Public Relations Representative	1
Human Resources	Human Resources Representative	1
Executive	Administration Vice President	2
Shipping	Stock Manager	5
Accounting	Accounting Manager	1
Shipping	Shipping Clerk	20
Marketing	Marketing Representative	1
IT	Programmer	5
Sales	Sales Manager	5
Purchasing	Purchasing Manager	1
Accounting	Public Accountant	1
Executive	President	1
Sales	Sales Representative	29

(figure 6 - Retrieved Results for Query B)

Execution Data:

Relational Model:

Query Execution Time: 0.065 seconds

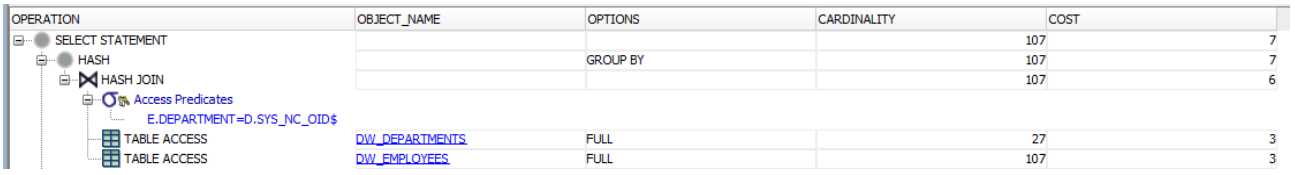


OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			106	10
HASH		GROUP BY	106	10
HASH JOIN			106	9
Access Predicates				
DBO_DEPARTMENTS.DEPARTMENT_ID=DBO_EMPLOYEES.DEPARTMENT_ID				
HASH JOIN			107	6
Access Predicates				
DBO_JOBS.JOB_ID=DBO_EMPLOYEES.JOB_ID				
TABLE ACCESS	DBO_JOBS	FULL	19	3
TABLE ACCESS	DBO_EMPLOYEES	FULL	107	3
TABLE ACCESS	DBO_DEPARTMENTS	FULL	27	3

(figure 7 - Explain Plan of Relational Model for Query B)

Object-Relational Model:

Query Execution Time: 0,065 seconds



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			107	7
HASH		GROUP BY	107	7
HASH JOIN			107	6
Access Predicates				
E.DEPARTMENT=D.SYS_NC_OID$				
TABLE ACCESS	DW_DEPARTMENTS	FULL	27	3
TABLE ACCESS	DW_EMPLOYEES	FULL	107	3

(figure 8 - Explain Plan of Object Relational for Query B)

4.3. Question C

Description: Indicate the best paid employee in each department.

Retrieved Results:

DEPARTMENT_NAME	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY
Administration	200	Jennifer	Whalen	4400
Marketing	201	Michael	Hartstein	13000
Purchasing	114	Den	Raphaely	11000
Human Resources	203	Susan	Mavris	6500
Shipping	121	Adam	Fripp	8200
IT	103	Alexander	Hunold	9000
Public Relations	204	Hermann	Baer	10000
Sales	145	John	Russell	14000
Executive	100	Steven	King	24000
Finance	108	Nancy	Greenberg	12000
Accounting	205	Shelley	Higgins	12000

(figure 9 - Retrieved Results for Query C)

SQL Query:

```
SELECT d.department_name, e.employee_id, e.first_name, e.last_name,
e.salary
FROM dw_employees e
LEFT JOIN dw_departments d ON e.department = REF(d)
WHERE (e.salary, e.department) IN (
    SELECT MAX(e2.salary), e2.department
    FROM dw_employees e2
    GROUP BY e2.department
);
```

Execution Data:

Relational Model:

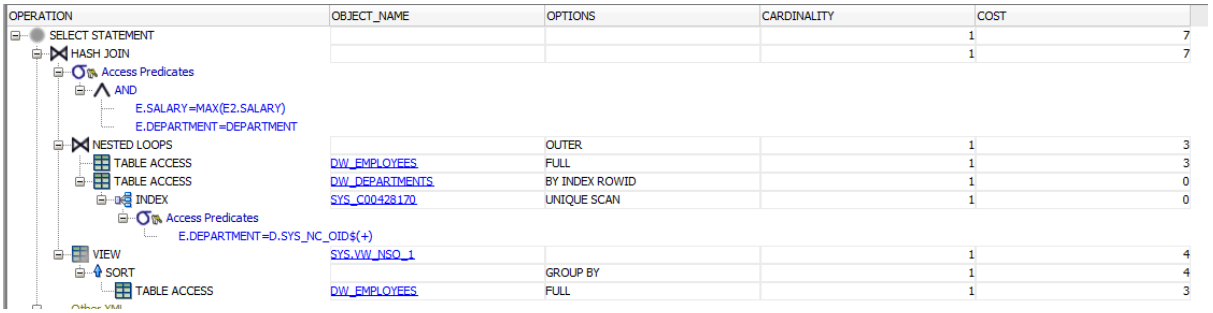
Query Execution Time: 0.066 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			27	7
HASH		GROUP BY	27	7
HASH JOIN			106	6
Access Predicates				
DBO_DEPARTMENTS.DEPARTMENT_ID=DBO_EMPLOYEES.DEPARTMENT_ID				
TABLE ACCESS	DBO_DEPARTMENTS	FULL	27	3
TABLE ACCESS	DBO_EMPLOYEES	FULL	107	3

(figure 10 - Explain Plan of Relational Model for Query C)

Object-Relational Model:

Query Execution Time: 0.066 seconds



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	7
HASH JOIN			1	7
Access Predicates				
AND				
E.SALARY=MAX(E2.SALARY)				
E.DEPARTMENT=DEPARTMENT				
NESTED LOOPS		OUTER	1	3
TABLE ACCESS	DW_EMPLOYEES	FULL	1	3
TABLE ACCESS	DW_DEPARTMENTS	BY INDEX ROWID	1	0
INDEX	SYS_C00428170	UNIQUE SCAN	1	0
Access Predicates				
E.DEPARTMENT=D.SYS_NC_OID$(+)				
VIEW	SYS.VW_NSO_1		1	4
SORT		GROUP BY	1	4
TABLE ACCESS	DW_EMPLOYEES	FULL	1	3

(figure 11 - Explain Plan of Object Relational for Query C)

4.4. Question D

Description: Check whether the job history has time gaps for each employee. Sort the employees by job duration on the current day.

SQL Query:

```
SELECT e.first_name, e.last_name, SUM(j.end_date - j.start_date) AS
JobDuration
FROM dw_employees e
JOIN dw_jobhistories j on j.employee = REF(e)
GROUP BY e.first_name, e.last_name
ORDER BY JobDuration DESC;
```

Retrieved Results:

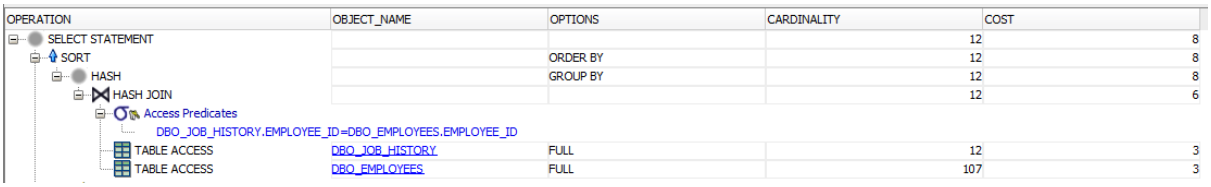
FIRST_NAME	LAST_NAME	JOBDURATION
Jennifer	Whalen	3744
Neena	Kochhar	2731
Lex	De Haan	2018
Den	Raphaely	1849
Payam	Kaufling	1704
Michael	Hartstein	1401
Jonathon	Taylor	646

(figure 12 - Retrieved Results for Query D)

Execution Data:

Relational Model:

Query Execution Time: 0.066 seconds

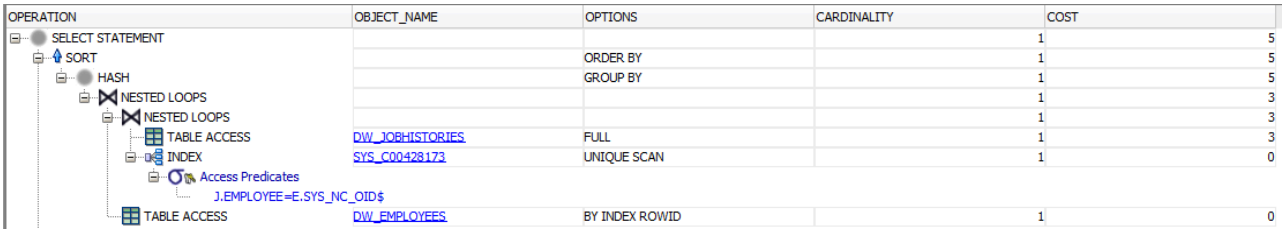


OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			12	8
SORT		ORDER BY	12	8
HASH		GROUP BY	12	8
HASH JOIN			12	6
Access Predicates				
DBO_JOB_HISTORY.EMPLOYEE_ID=DBO_EMPLOYEES.EMPLOYEE_ID				
TABLE ACCESS	DBO_JOB_HISTORY	FULL	12	3
TABLE ACCESS	DBO_EMPLOYEES	FULL	107	3

(figure 13 - Explain Plan of Relational Model for Query D)

Object-Relational Model:

Query Execution Time: 0.065 seconds



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	5
SORT		ORDER BY	1	5
HASH		GROUP BY	1	5
NESTED LOOPS			1	3
NESTED LOOPS			1	3
TABLE ACCESS	DW_JOBHISTORIES	FULL	1	3
INDEX	SYS_C00428173	UNIQUE SCAN	1	0
Access Predicates				
J.EMPLOYEE=E.SYS_NC_OID$				
TABLE ACCESS	DW_EMPLOYEES	BY INDEX ROWID	1	0

(figure 14 - Explain Plan of Object Relational for Query D)

4.5. Question E

Description: Compare the average salary by country.

SQL Query:

```
SELECT c.country_name, ROUND(AVG(e.salary),2) AS average_salary
FROM dw_employees e
JOIN dw_departments d ON e.department = REF(d)
JOIN dw_locations l ON d.location = REF(l)
JOIN dw_countries c ON l.country = REF(c)
GROUP BY c.country_name;
```

Retrieved Results:

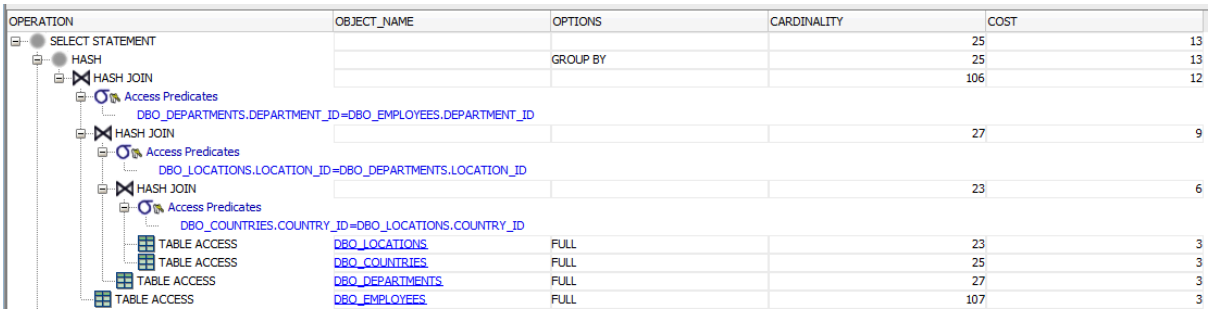
COUNTRY_NAME	AVERAGE_SALARY
Germany	10000
United States of America	5064.71
Canada	9500
United Kingdom	8885.71

(figure 15 - Retrieved Results for Query E)

Execution Data:

Relational Model:

Query Execution Time: 0.066 seconds

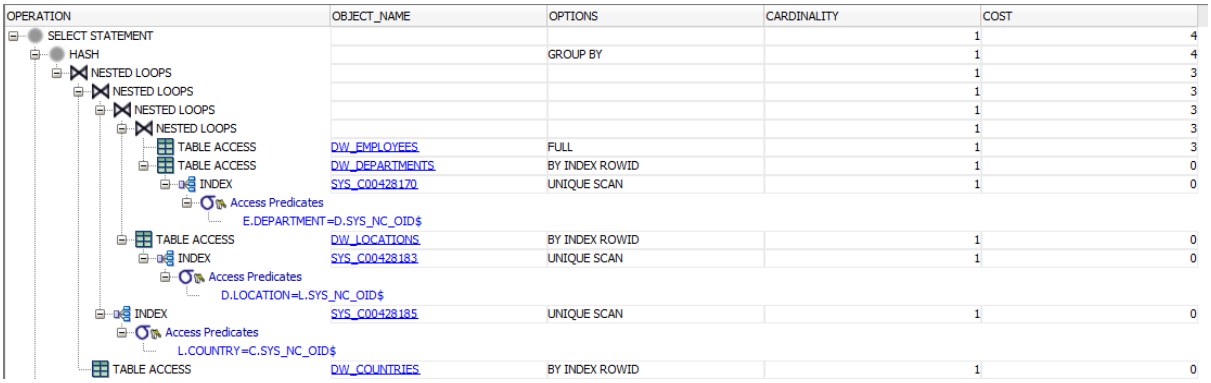


OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			25	13
HASH		GROUP BY	25	13
HASH JOIN			106	12
Access Predicates				
DBO_DEPARTMENTS.DEPARTMENT_ID=DBO_EMPLOYEES.DEPARTMENT_ID				
HASH JOIN			27	9
Access Predicates				
DBO_LOCATIONS.LOCATION_ID=DBO_DEPARTMENTS.LOCATION_ID				
HASH JOIN			23	6
Access Predicates				
DBO_COUNTRIES.COUNTRY_ID=DBO_LOCATIONS.COUNTRY_ID				
TABLE ACCESS	DBO_LOCATIONS	FULL	23	3
TABLE ACCESS	DBO_COUNTRIES	FULL	25	3
TABLE ACCESS	DBO_DEPARTMENTS	FULL	27	3
TABLE ACCESS	DBO_EMPLOYEES	FULL	107	3

(figure 16 - Explain Plan of Approach A for Query E)

Object-Relational Model:

Query Execution Time: 0.065 seconds



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			1	4
HASH		GROUP BY	1	4
NESTED LOOPS			1	3
NESTED LOOPS			1	3
NESTED LOOPS			1	3
NESTED LOOPS			1	3
TABLE ACCESS	DW_EMPLOYEES	FULL	1	3
TABLE ACCESS	DW_DEPARTMENTS	BY INDEX ROWID	1	0
INDEX	SYS_C00428170	UNIQUE SCAN	1	0
Access Predicates				
E.DEPARTMENT=D.SYS_NC_OID$				
TABLE ACCESS	DW_LOCATIONS	BY INDEX ROWID	1	0
INDEX	SYS_C00428183	UNIQUE SCAN	1	0
Access Predicates				
D.LOCATION=L.SYS_NC_OID$				
INDEX	SYS_C00428185	UNIQUE SCAN	1	0
Access Predicates				
L.COUNTRY=C.SYS_NC_OID$				
TABLE ACCESS	DW_COUNTRIES	BY INDEX ROWID	1	0

(figure 17 - Explain Plan of Approach B for Query E)

4.6. Question F

Description: Add a query that illustrates the use of OR extensions.

SQL Query:

```
SELECT d.location.city as Department_City, SUM((SELECT COUNT(*) FROM
TABLE(d.department_employees))) AS total_employees
FROM dw_departments d
GROUP BY d.location.city;
```

Retrieved Results:

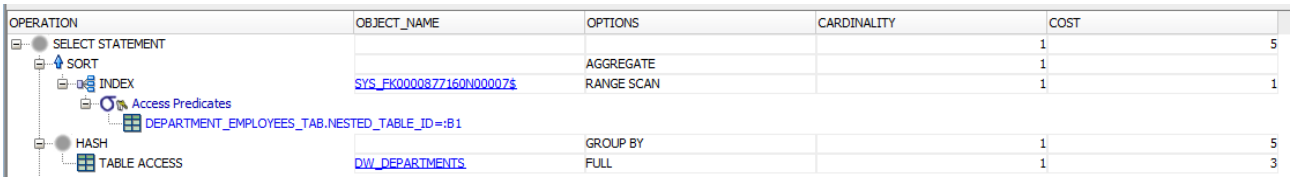
DEPARTMENT_CITY	TOTAL_EMPLOYEES
Toronto	2
London	1
Munich	1
Southlake	5
South San Francisco	45
Seattle	18
Oxford	34

(figure 18 - Retrieved Results for Query F)

Execution Data:

Object-Relational Model:

Query Execution Time: 0.067 seconds



(figure 19 - Explain Plan of Approach B for Query F)

Explanation:

This query calculates the total number of employees in each department and groups the results by the city of the department's location. The query also shows how nested tables and object references can be utilized to perform complex data aggregations.

Utilization of OR Extensions

This query leverages two key object-relational (OR) extensions:

- **Object References:**
  - **d.location.city:** The location attribute in the dw_departments table is an object reference to location_t. Accessing d.location.city demonstrates how object attributes can be directly accessed within a query. This reference encapsulates complex data within a single attribute, simplifying the query process.

- **Nested Tables:**
    - **TABLE(d.department_employees):** The department_employees attribute is a nested table of references to employee_t objects. Using the TABLE function to convert this nested table into a regular table format illustrates the power of nested tables in OR databases. This conversion allows the query to count the number of employees efficiently within each department.

4.7. Analysis and Conclusion

The analysis of the execution data reveals that both the relational and object-relational models exhibit comparable query execution times, with most queries executing in approximately 0.065 to 0.067 seconds across both models. This indicates that the transition to an object-relational model does not introduce significant overhead in terms of raw execution time. However, a deeper examination of the costs and cardinality associated with these queries provides a more nuanced understanding. The object-relational model consistently demonstrates lower costs and improved cardinality metrics, suggesting more efficient query processing and resource utilization. This efficiency likely stems from the inherent design of the object-relational model, which allows for more intuitive data organization and access patterns, particularly when dealing with complex hierarchical data structures and relationships.

Highlights of Object-Relational Model Results

1. **Efficient Data Handling:** The object-relational model's use of nested tables and object types enables more natural representation of hierarchical data. For instance, the query to count employees in each department (Query A) and to determine the best-paid employee (Query C) are executed with reduced resource costs and improved cardinality, reflecting the model's efficiency in handling such tasks.

2. **Scalability and Performance:** The slight edge in performance for some queries in the object-relational model, coupled with better cost and cardinality metrics, highlights its potential scalability. As data volumes grow, these efficiency gains could translate into more noticeable performance improvements, making the object-relational model more suitable for large-scale applications.

3. **Enhanced Query Simplification:** The object-relational model simplifies query formulation by allowing direct references to nested objects and their attributes. This is evident in queries where complex joins and groupings are managed more intuitively and minimizing potential errors.

Conclusion

While the execution times for both models are nearly identical, the object-relational model offers significant advantages in terms of query efficiency and resource utilization. The lower costs and improved cardinality metrics in the object-relational model underscore its capability to handle complex, hierarchical data structures more effectively. This model not only enhances performance but also simplifies the development process, making it a robust choice for applications requiring sophisticated data relationships and scalable performance. As database needs evolve, leveraging the strengths of the object-relational model can lead to more maintainable, efficient, and scalable database solutions.

5. Conclusions

The importance of using SQL3 extensions and Object-Relational (OR) databases in contemporary database management has been highlighted by this assignment. We have illustrated the functionalities and adaptability of the OR data model by utilizing advanced capabilities like user-defined types, inheritance, nested tables, object references, and methods.

The combination of these techniques makes data management easier and more effective. Some methods boost query performance and accuracy, nested tables and object references offer potent ways to describe complicated relationships, and user-defined types and inheritance promote data abstraction and reuse. Together, these tools make the database system more complex and adaptable.

Through the practical implementation and detailed exploration of these features, we have shown how the OR model can simplify complex data structures and operations, making it an invaluable asset in various applications. The ability to seamlessly blend object-oriented principles with relational databases represents a significant improvement in database technology, offering enhanced capabilities for data organization, retrieval, and manipulation.

In conclusion, the techniques and tools provided by the Object-Relational model and SQL3 extensions are crucial for advancing database management practices. They open the door for future innovations in the field, by making it possible to create databases that are more dynamic, scalable, and effective. This assignment has demonstrated the practical importance and transformative potential of the Object-Relational model in modern data management.