University of British Columbia, Vancouver
Department of Computer Science

CPSC 304 Project Cover Page

Milestone #: 4
Date: April 1, 2025
Group Number: 50

Name	Student Number	CS Alias (Userid)	Preferred E-mail Address
Maya Dong	37406162	c0w0b	maya.mengya@gmail.com
Alan Wang	28413145	d9o7h	yinghsu@student.ubc.ca

By typing our names and student numbers in the above table, we certify that the work in the attached assignment was performed solely by those whose names and student IDs are included above. (In the case of Project Milestone 0, the main purpose of this page is for you to let us know your e-mail address, and then let us assign you to a TA for your project supervisor.)

In addition, we indicate that we are fully aware of the rules and consequences of plagiarism, as set forth by the Department of Computer Science and the University of British Columbia

Repo link (also included in Canvas submission):
https://github.students.cs.ubc.ca/CPSC304-2024W-T2/project_c0w0b_d9o7h_w4r0g

Project Description:

The project is a database management system that organizes and catalogs Pokémon data. The domain of the project focuses on the Pokémon universe, specifically the various species, their moves, abilities, evolutions, and regional differences. It is designed for players and enthusiasts who want to explore detailed information about Pokémon, their interactions, and their evolution paths across different game versions and regions.

SQL:

A SQL script for creating tables and inserting example data is located in the sql folder, titled: **pokemon.sql**

https://github.students.cs.ubc.ca/CPSC304-2024W-T2/project_c0w0b_d9o7h_w4r0g/blob/main/sql/pokemon.sql

Exported tables after running pokemon.sql
pokemon.pdf

https://github.students.cs.ubc.ca/CPSC304-2024W-T2/project_c0w0b_d9o7h_w4r0g/blob/main/sql/pokemon.pdf

Our final schema is consistent with the initial submission, with the intended many-to-many relationships included. However, we were unable to implement the necessary assertion statements to enforce the many-to-many integrity constraints at this stage due to this version of SQL*PLUS not supporting schema-level constraints (for example ASSERTIONS). We instead use procedures to ensure that each pokemon added will have at least one corresponding move, ability and type.

Schema

Primary keys: underlined
Foreign keys: **bold**

Entity/Relations	**Definition**
Pokemon	Pokemon(PokemonID: INTEGER, PokemonDescription: VARCHAR, PokemonName: VARCHAR) Candidate Key: PokemonName
Learns	Learns(**PokemonID: INTEGER, MoveID: INTEGER**) Constraints: MoveID is NOT NULL
Move_Associates	Move_Associates1(MoveID: INTEGER, MoveName: VARCHAR, Power: INTEGER, Accuracy: INTEGER, PowerPoints: INTEGER, **MoveEffect: VARCHAR**) Move_Associates2(MoveEffect: VARCHAR, **TypeName: VARCHAR**) Candidate Key: MoveName
Type	Type(TypeName: VARCHAR, TypeDescription: VARCHAR)
Effect	Effect(**TypeName1: VARCHAR**, **TypeName2: VARCHAR**, Percentage: INTEGER)
Belongs	Belongs(**PokemonID: INTEGER, TypeName: VARCHAR)** Constraints: TypeName is NOT NULL
Possesses	Possesses(**PokemonID: INTEGER, AbilityID: INTEGER)**

	Constraints: AbilityID is NOT NULL
Ability	Ability(AbilityID: INTEGER, AbilityEffect: VARCHAR)
Item_Owns	Item_Owns(ItemName: VARCHAR, ItemEffect: VARCHAR, ItemType: VARCHAR, **ItemEffect: VARCHAR, PokemonID: INTEGER**) Item_Owns2(ItemEffect: VARCHAR, ItemType: VARCHAR)
Sells	Sells(**ItemName: VARCHAR**, **LocationName: VARCHAR, RegionName: VARCHAR**) Constraints: ItemName is NOT NULL
Pokemart	Pokemart(**LocationName: VARCHAR, RegionName: VARCHAR)**
Gym	Gym(**LocationName: VARCHAR**, **RegionName: VARCHAR,** Badge: VARCHAR)
Trainer_Defends	Trainer_Defends(TrainerName: VARCHAR, Winnings: INTEGER, **LocationName: VARCHAR, RegionName: VARCHAR**)
Location	Location(LocationName: VARCHAR, **RegionName: VARCHAR**, Function: VARCHAR) Constraints: RegionName is NOT NULL
Region	Region(RegionName: VARCHAR, RegionDescription: VARCHAR)
AppearsIn	AppearsIn(**RegionName: VARCHAR, PokemonID: INTEGER**)
Owns (for Trainer-Pokemon Relation)	Owns(**TrainerName: VARCHAR**, **PokemonID: INTEGER**)
EvolvesInto	EvolvesInto(**PreEvolutionID: INTEGER**, **PostEvolutionID: INTEGER**, Condition: VARCHAR)

SQL Queries

1. INSERT

appService.js(Line 185)

Because of our integrity constraint, a pokemon needs to have an associated type, ability, and move, thus when a pokemon is inserted into its own entity, relations Belongs, Possesses, and Learns that corresponds to pokemon's relation to type, ability, and move are also inserted.

```
// INSERTING POKEMON & associated BELONGS, LEARNS and POSSESSES into Database
async function insertPokemon(id, description, name, type, abilityID, moveID) {
    return await withOracleDB(async (connection) => {
        // Check if type exists
        const checkType = await connection.execute(
            `SELECT COUNT(*) FROM Type t WHERE t.typeName = :type`,
            { type }
        );
        if (checkType.rows[0][0] < 1) {
            console.log('TypeName does not exist.');
            return false;
        }

        // Check if ability exists
        const checkAbility = await connection.execute(
            `SELECT COUNT(*) FROM Ability a WHERE a.AbilityID = :abilityID`,
            { abilityID }
        );
        if (checkAbility.rows[0][0] < 1) {
            console.log('Ability does not exist.');
            return false;
        }

        // Check if move exists
        const checkMove = await connection.execute(
            `SELECT COUNT(*) FROM Move_Associates1 m WHERE m.MoveID = :moveID`,
            { moveID }
        );
        if (checkMove.rows[0][0] < 1) {
            console.log('Move does not exist.');
            return false;
        }

        // Execute insert queries separately
        await connection.execute(
            `INSERT INTO Pokemon (PokemonID, PokemonName, PokemonDescription)
             VALUES (:id, :name, :description)`,
            { id, name, description }
        );

        await connection.execute(
            `INSERT INTO Belongs (PokemonID, TypeName)
             VALUES (:id, :type)`,
            { id, type }
        );

        await connection.execute(
            `INSERT INTO Possesses (PokemonID, AbilityID)
             VALUES (:id, :abilityID)`,
            { id, abilityID }
        );

        await connection.execute(
            `INSERT INTO Learns (PokemonID, MoveID)
             VALUES (:id, :moveID)`,
            { id, moveID }
        );

        await connection.commit();
        return true;
    }).catch((err) => {
        console.error(err);
        return false;
    });
}
```

2. The UPDATE query is not required for this group as per modified milestone requirements.

3. DELETE

appService.js(Line 251)

```
//DELETE POKEMON FROM DATABASE
async function deletePokemon(id) {
    return await withOracleDB(async (connection) => {
        await connection.execute(
          `DELETE FROM Pokemon p WHERE p.pokemonid = :id`,
            { id });

        await connection.commit();
        return true;
    }).catch((err) => {
        console.error(err);
        return false;
    });
}
```

Note: id is a bound variable here, which prevents attackers from injecting malicious SQL code as database treating the bound variables as data and not part of the SQL command

4. SELECTION

appService.js(Line 139)

```
// SELECT & JOIN Types + Effect from DB
async function fetchTypesEffectParamsFromDb(parameters) {
    var query = 'SELECT t.Typename, t.typeDescription, e.percentage, e.typename2 ' +
        'FROM Type t, Effect e WHERE t.TypeName = e.TypeName1';

    let type = parameters[0];
    let op1 = parameters[1];
    let num1 = parameters[2];
    let logic = parameters[3];
    let op2 = parameters[4];
    let num2 = parameters[5];

    let bindValues = {};

    if (type === 'All' && op1 === 'None') {
        return fetchTypesEffectFromDb();
    }
    if (type !== 'All') {
        query = query + ' and t.Typename = :type';
        bindValues.type = type;
    }
    if (op1 !== 'None' && num1 >= 0 && num1 != '') {
        query = query + ` and (e.percentage ${op1} :num1`;
        bindValues.num1 = num1;
        if (logic === 'None') {
            query = query + ')'
        } else {
            if (op2 !== 'None' && num2 >= 0 && num2 != '') {
                query = query + ` ${logic} e.percentage ${op2} :num2)`;
                bindValues.num2 = num2;
            } else {
                query = query + ')'
            }
        }
    }

    console.log("Final Query: ", query);
    console.log("Bind Values: ", bindValues);

    try {
        return await fetchQuery(query,bindValues);
    } catch (error) {
        console.error("Error message element not found:", error.message);
    }
}
```

Note: operators (>,<,=) & logical statements (AND, OR) cannot be bound in SQL*PLUS

5. PROJECTION

appService.js(Line 277)

```
// PROJECTION & JOIN (join statement in SQL by creating VIEW, joining MoveAssociate_1 &
// MoveAssociate_2 to find Move's associated Type)
async function fetchMoveAttributesFromDb(attributes) {
    const typeFilter = attributes.pop();

    if (!Array.isArray(attributes) || attributes.length === 0) {
        throw new Error('Attributes must be a non-empty array');
    }

    var query = '';

    if (typeFilter === 'All') {
        query = `SELECT ${attributes} FROM Movetype`;
        return await queryFromOracle(query);
    } else {
        query = `SELECT ${attributes} FROM Movetype mt WHERE mt.typename = :typeFilter`;
        return await queryFromOracle(query, { typeFilter });
    }
}
```

6. JOIN

appService.js(Line 132)

```
// SELECT & JOIN Types + Effect from DB
async function fetchTypesEffectFromDb() {
    const query = 'SELECT t.Typename, t.typeDescription, e.percentage, e.typename2 ' +
        'FROM Type t, Effect e WHERE t.TypeName = e.TypeName1';
    return await fetchQuery(query);
}
```

Note: we use join on multiple occasions filtering by additional attributes, for example, in PROJECTION & SELECTION in 4 and 5.

Specifically: JOIN in the function in SELECTION (4) joins on two different relations, moves and type, and filters by type by adding t.TypeName = :type to the WHERE clause.

7. Aggregation with GROUP BY

appService.js(Line 320)

```
async function fetchItemCountByType() {
    return await withOracleDB(async (connection) => {
        const result = await connection.execute(
            `SELECT io.ItemType, COUNT(i.ItemName) AS ItemCount
            FROM Item_Owns i
            JOIN Item_Owns2 io ON i.ItemEffect = io.ItemEffect
            GROUP BY io.ItemType
            ORDER BY ItemCount DESC`,
        );
        return result.rows;
    }).catch(() => {
        return [];
    });
}
```

This query counts the number of items for each ItemType by joining the Item_Owns and Item_Owns2 tables on the ItemEffect field. The result is grouped by ItemType and ordered by the item count in descending order.

8. Aggregation with HAVING

appService.js(Line 353)

```
// Filter Poke Marts by item type and minimum quantity
async function fetchPokeMartByTypeAndMin(itemType, minQuantity) {
    const minQuantityNum = parseInt(minQuantity, 10); // Convert minQuantity from a string to an integer (base 10)
    return await withOracleDB(async (connection) => {
        const result = await connection.execute(
            `SELECT
                p.LocationName,
                p.RegionName,
                io2.ItemType,
                COUNT(s.ItemName) AS ItemQuantity
            FROM
                Item_Owns io
            JOIN
                Item_Owns2 io2 ON io.ItemEffect = io2.ItemEffect
            JOIN
                Sells s ON io.ItemName = s.ItemName
            JOIN
                Pokemart p ON s.LocationName = p.LocationName AND s.RegionName = p.RegionName
            WHERE
                io2.ItemType = :itemType
            GROUP BY
                p.LocationName, p.RegionName, io2.ItemType
            HAVING
                COUNT(s.ItemName) >= :minQuantity
            ORDER BY
                p.LocationName`,
            { itemType, minQuantity: minQuantityNum }
        );
        console.log(result)
        return result.rows;
    }).catch(() => {
        return [];
    });
}
```

This query filters Poke Marts based on the item type and ensures that each selected Poke Mart has a minimum quantity of a specific item type. It groups the results by location and item type, and uses the HAVING clause to filter for marts meeting the minimum quantity requirement.

9. **Nested aggregation with GROUP BY**

appService.js(Line 388)

```
// NESTED AGGREGATION with GROUP BY
async function fetchAverageWinningAggregate(operator) {
    const query = 'SELECT AVG(td.winnings) FROM trainer_defends td WHERE td.winnings' + operator +
        'ALL (SELECT AVG(td2.winnings) FROM trainer_defends td2 GROUP BY td2.locationname, td2.regionname)';
    return await fetchQuery(query);
}
```

This query calculates the average winnings for trainers and compares it to the average winnings for each group of trainers based on their location and region. It uses a nested aggregation with a GROUP BY clause to find the average winnings for each location-region pair and then compares it to a global average using the selected operator.

10. The DIVISION query is not required for this group as per modified milestone requirements.