Lebanese American University

School of arts and sciences

CSC375: Database Management Systems

Section:14

Instructor: DR. Khaleel Mershad

Project Phase4

Zeina Merchad - 202209100
Mohammad Al Asal - 202202154
Raghad Hmede - 202206230

Date of submission:12/12/2023

Table of Contents

Table of Figures

Introduction

The social security system is composed of the government-administered programs that provide financial support to individuals in various stages of their lives, including retirement, disability, survivorship, and other circumstances. The Social Security Administration (SSA) is responsible for managing these programs and ensuring that they function effectively and efficiently. The Social Security Administration Information System (SSAIS) is a critical component of the social security system, enabling the SSA to manage records pertaining to workers, employers, social security accounts, appeal processes, online services, and healthcare programs to provide efficient and responsive services to individuals at various life stages.

The reason for SSAIS is simple. Social security systems are here to help people, and they need to work well. The motivation for the SSAIS lies in the operational imperative of social security systems: to provide efficient and responsive services to individuals at various life stages, including retirement, disability, and survivorship. The SSAIS equips administrators, employees, and beneficiaries with the tools and information necessary to navigate the intricate landscape of social security, from calculating benefits and managing appeals to ensuring the successful implementation of Medicare programs and recording the history of interactions. It is within the construct of this dynamic database that critical data is stored, accessed, and transformed into actionable insights and services that directly impact the lives of countless individuals and families. It's the engine that drives the daily operations and decision-making within the Social Security Administration.

Database Requirements

Overview

The Social Security Administration Information System (SSAIS) is a comprehensive database designed to efficiently manage a wide range of information related to workers, employers, Social Security accounts, appeal processes, online services, Medicare programs. SSAIS serves as the central repository for crucial data necessary for the administration of Social Security benefits and services.

Workers

The database captures detailed information about workers enrolled in the Social Security system. Each worker's record includes a unique Worker ID, their full name, age, gender, marital status, dependencies related information, address. Workers are categorized into three main types: Survivors, Retired, and Disabled. Survivors' records include specifications about their condition, while Retired workers have details about their earning history and age at retirement. Disabled workers have specifications of their disabilities. Workers' medical history is encompassed by medical operations performed with details saved such as name, description, date, cost and status.

Employee

Employee information is also stored in SSAIS, with each employee assigned a unique Employer ID. Key data about employees includes details regarding their full name, age, gender, marital status, address and salary. This data is crucial for managing the contributions and interactions between employees and the Social Security system.

Corporation

The database holds detailed information about the corporations that employ workers and therefore enrolls them in social security. The corporation has an ID associated with it along with its name, and detailed address (Country, Rue, City, floor number (if possible))

Social Security Account

The Social Security Account section of the database focuses on individual accounts. Each account is identified by a unique Account ID and date of creation. These accounts are central to tracking and managing individual Social Security benefits. Workers may as well order viewing statements. Statements contain crucial information, including the ID of the generation, name, and further descriptions.

Appeal Process

The Appeal Process segment of the database records appeals submitted by workers and monitors their progress. Each process is assigned a unique Process ID, the type of appeal, and the date of application. Appeal processes are first considered by an employee who may later monitor the appeal if chosen to proceed to be reviewed by a judge in court. The judge's name and type of court specialized in shall be demonstrated.

Medicare Program

The database encompasses the Medicare Program, which offers various insurance plans, such as life, health, long-term disability, and auto insurance. Within health plans, there are four categories: Bronze, Silver, Gold, and Platinum, each with its coverage percentage. Platinum covers 90%, Gold covers 80%, Silver covers 70%, and Bronze covers 60% of average medical costs. Health Plan includes drug prescription coverage with the respective percentage coverage. Therefore, the database shall comprehensively address this relationship between drugs prescribed and health program and workers' drugs intake. In relation to the drug entity, its type, production date, expiry date, manufacturing company shall be stored within the database. Each drug is uniquely identified by an ID. The SSAIS database serves as the backbone of the Social Security Administration, providing the infrastructure needed to manage and deliver essential services, calculate benefits accurately, track appeals, and administer Medicare programs. It is a critical tool in ensuring the efficient operation of the Social Security system and the welfare of its beneficiaries.

ER- model

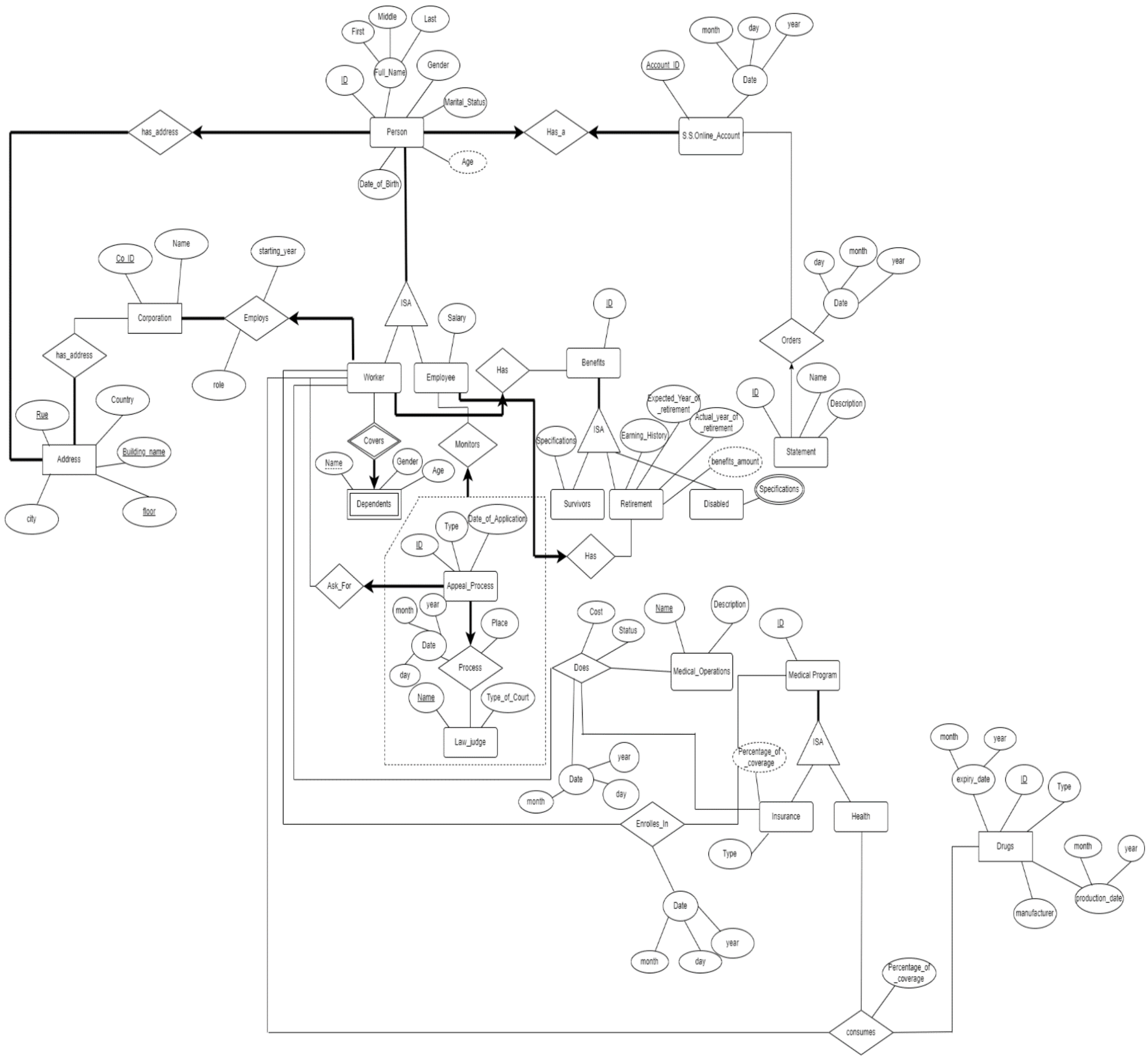

Figure 1: ER-model

ER- model description

Person-Dependents-Benefits

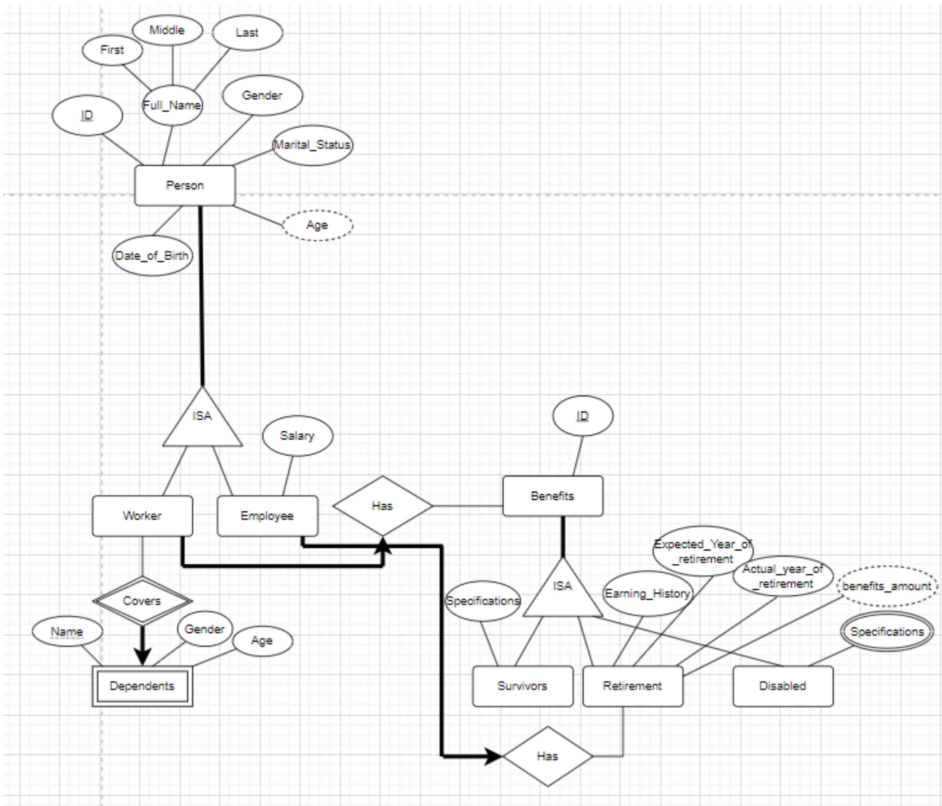

Figure 2: ER-model part 1

Each person can be either a worker or an employee in the social security administration. The worker covers many dependents, however a dependent can be covered by at least and at most one worker. Worker has benefits that can be divided into 3 types only: Survivors, Retirement and Disabled benefits. However, an employee can only benefit from the Retirement benefits where the benefits amount is calculated through the earning history and the actual year of retirement.

Person-S.S.Online_Account-Statement

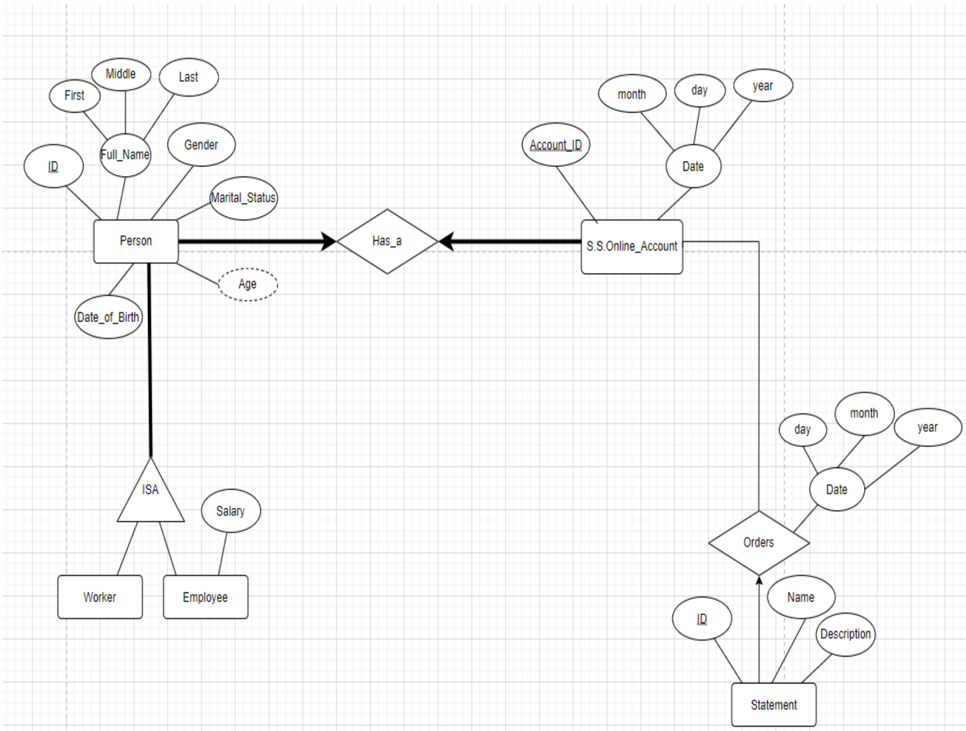

Figure 3: ER-model part 2

Each person should have 1 social security account that should also be related to only 1 person. Through the social security account a person can order many statements of different types: Health insurance treatment (providing medications and tests), end of service compensation transaction…

Person-Corporation-Address

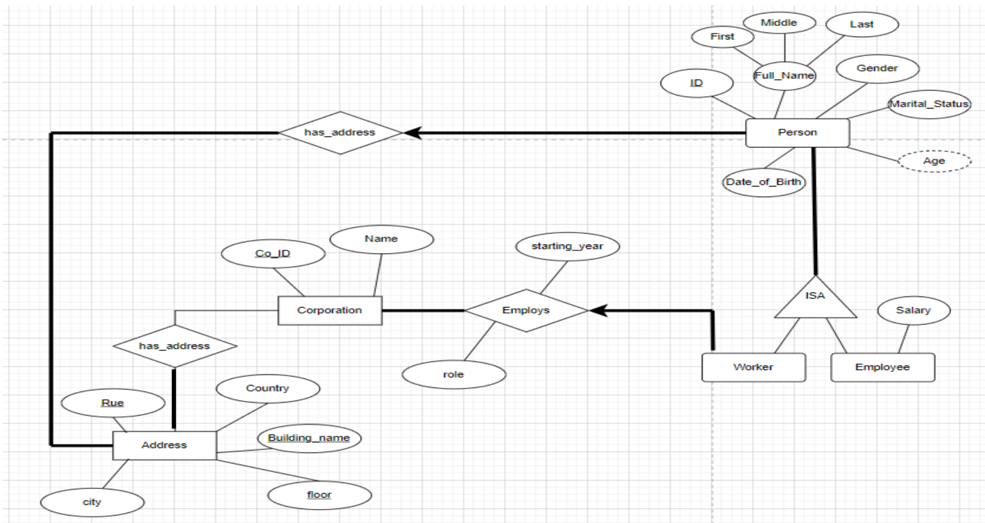

Figure 4: ER-model part 3

A person should have exactly one address, however, an address can be associated with 1 or more persons. The worker is enrolled in the social security system by a corporation (on the name of the corporation) that also have many addresses (many branches)

Employee-Worker-Appeal_Process-Law_judge

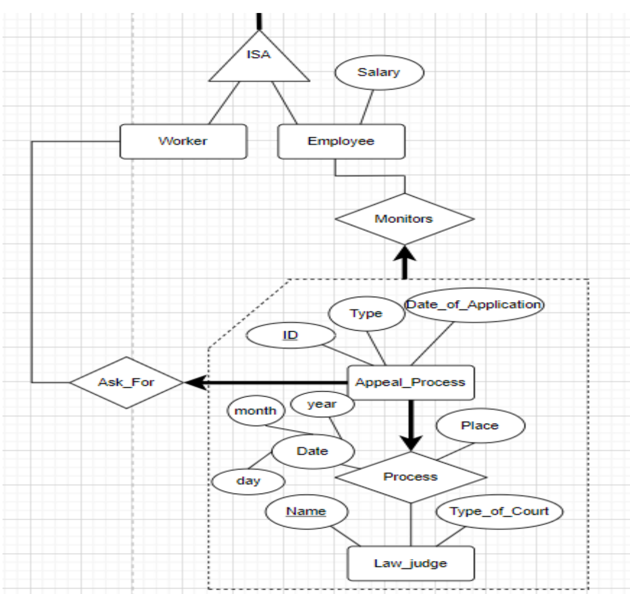

Figure 5: ER-Model part 4

The worker can ask for 0 or more appeal processes; however, an appeal process should be associated with exactly one worker. The appeal process can be processed by exactly one law judge that can process many other appeal processes. The full appeal process is handled by one employee.

Worker-Medical_operations-Medical Program-Drugs

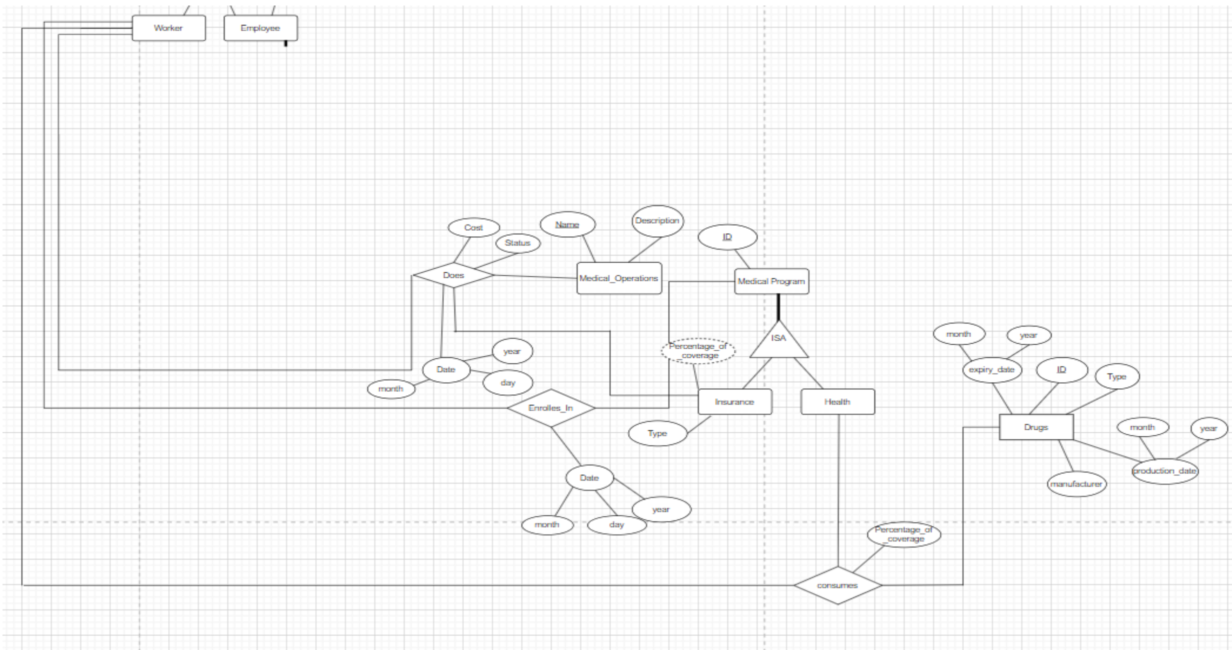

Figure 6: ER- Model part 5

A worker may do 0 or more medical operations, that can also be done by an insurance program. A worker can be enrolled in 0 or more medical programs and the database should store the date of enrollment. The worker along with the health medical program consume drugs and the database should store the percentage of coverage for these drugs.

Relational schema

Person (Person_ID, First, Middle, Last, Gender, Marital_Status, Date_of_Birth, #Rue (NOT NULL), #Building_Name (NOT NULL), #floor (NOT NULL), #Account_ID (NOT NULL))

Worker (#Person_ID, #Corporation_ID (NOT NULL), role, starting_year, #Benefits_ID (NOT NULL), #Account_ID (NOT NULL), #Rue (NOT NULL), #Building_Name (NOT NULL), #floor (NOT NULL))

Employee (#Person_ID, Salary, #Benefits_ID (NOT NULL), #Account_ID (NOT NULL), #Rue (NOT NULL), #Building_Name (NOT NULL), #floor (NOT NULL))

Corporation (Corporation_ID, Corporation_Name)

Address (Rue, Building_Name, floor, city, country)

Has_Address (#Rue, #Building_Name, #floor, #Corporation_ID)

S.S. Online_Account (Account_ID, month, day, year)

Statement (Statement_ID, Statement_Name, Description, #Account_ID, day, month, year)

Benefits (Benefits_ID)

Survivors (#Benefits_ID, Specifications)

Retirement (#Benefits_ID, Earning_History, Expected_Year_of_Retirement, Actual_Year_of_Retirement)

Disabled (#Benefits_ID, Disability_Type)

Dependants (Dependants_Name, #Person_ID, Gender, age)

Appeal_Process (Process_ID, type, Date_of_Application, #Person_ID (NOT NULL), #judge_name(NOT NULL))

Law_Judge (Judge_Name, Type_of_Court)

Process (#Process_ID, #Judge_Name, day, month, year, place)

Monitors (#Process_ID, #Person_ID)

Medical_Operations (Operation_Name, Description)

Medical_Program (Program_ID)

Insurance (#Program_ID, type)

Health (#Program_ID)

Does (#Person_ID, #Operation_Name, #Program_ID, cost, status, day, month, year)

Enrolls_In (#Person_ID, #Program_ID, day, month, year)

Drugs (Drug_ID, Emonth, Eyear, type, Pmonth, Pyear, manufacturer)

Consumes (#Person_ID, #Program_ID, #Drug_ID, Percentage_of_Coverage)

Database implementation

DDL queries

First and before anything we created our database named project_db by using the DDL statement "CREATE DATABASE project_db", therefore defining and initializing the database with a specific name.

After initializing the database we started creating the tables: the fundamental objects of the database to organize and store data by using the DDL statement "CREATE TABLE *name_of_the_table"*

1- Address table:

```sql
CREATE DATABASE project__db;
use project__db;
CREATE TABLE address
 (
    rue           VARCHAR(30),
    building_name VARCHAR(30),
    floor_nb      TINYINT,
    city          VARCHAR(50),
    country       VARCHAR(50),
    PRIMARY KEY(rue, building_name, floor_nb)
 );
```

Figure 7: Address table

Inside the table, the attributes (tables) with their corresponding domain are specified: rue is the column and VARCHAR(30) is the domain of rue, meaning that rue can hold strings up to 30 characters in length. TINYINT: the domain of floor_nb is a datatype that requires a small storage space, it represents signed integers.
In the table address, and as the relational model shows has 3 primary keys: rue, building_name and floor_nb. Specified as primary keys in the database using the integrity constraint
"PRIMARY KEY (*name_of_the_primary_key)*".

2- Corporation table:

```
CREATE TABLE corporation
(
   corporation_id   INT UNSIGNED,
   corporation_name VARCHAR(50) NOT NULL,
   PRIMARY KEY(corporation_id)
);
```

Figure 8: Corporation table

The table corporation contains 2 columns: corporation_id of domain INT UNSIGNED and corporation name of type VARCHAR(50). Since the name of the corporation cannot be null, an integrity constraint is added to it: NOT NULL. The primary key that uniquely identifies the rows in this table is corporation_id specified using the integrity constraint: “PRIMARY KEY(corporation_id)”.

3- Benefits table:

```
CREATE TABLE benefits
(
   benefit_id INT UNSIGNED DEFAULT 0,
   PRIMARY KEY(benefit_id)
);
```

Figure 9: Benefits table

The only attribute in the benefits table is benefit_id, with domain: INT UNSIGHNED. If no benefit_id is specified a default value of 0 will be assigned to the benefit_id, since it cannot be null, this is done by using an integrity constraint: “DEFAULT 0”. The primary key of this table is benefit_id specified by using the integrity constraint: “PRIMARY KEY(benefit_id)”.

3- survivors table:

```
CREATE TABLE survivors
(
    survivor_id     INT UNSIGNED,
    specifications VARCHAR(100) not null,
    PRIMARY KEY(survivor_id),
    FOREIGN KEY (survivor_id) REFERENCES benefits(benefit_id) ON UPDATE CASCADE
    ON DELETE CASCADE
);
```

Figure 10: survivors table

This table has 2 columns: survivor_id of domain INT UNSIGNED, that is also the primary key specified by using the integrity constraint: “PRIMARY KEY(survivor_id)”, in addition to another column: specifications of domain: VARCHAR(100) with an integrity constraint “NOT NULL” to ensure that the column specifications cannot have a null value.
As shown in the relational model, benefit_id is a foreign key referencing the primary key column in the benefits table (we changed its name to survivor_id so we can differentiate correctly between them), this is specified by using the integrity constraint: “FOREIGN KEY (survivor_id) REFERENCES benefits (benefit_id)”, to ensure that no modification or change will occur to the two table linked together by the foreign key, a keyword “ON UPDATE CASCADE ON DELETE CASCADE” meaning that the same action that will be performed on benefit_id of the table benefit will be reflected to the column survivor_id.

4- Retirement table:

```
CREATE TABLE retirement
(
    retirement_id                INT UNSIGNED,
    earning_history             INT UNSIGNED,
    expected_year_of_retirement SMALLINT not null,
    actual_year_of_retirement   SMALLINT not null,
    PRIMARY KEY (retirement_id),
    FOREIGN KEY (retirement_id) REFERENCES benefits(benefit_id) ON UPDATE CASCADE
    ON DELETE CASCADE
);
```

Figure 11: retirement table

This table has 4 columns: retirement_id of domain: INT UNSIGNED, earning_history also of domain INT UNSIGNED, expected_year_of_retirement of domain SMALLINT with the integrity constraint not null to ensure that no null values are given to this attribute, and actual_year_of_retirement also of domain SMALLINT and a NOT NULL integrity constraint.
The primary key of this table is retirement_id specified by “PRIMARY KEY(retirement_id)”

retirement_id is a foreign key referencing benefit_if of the table benefits declared by using the integrity constraint: “FOREIGN KEY (retirement_id) REFERENCES benefits(benefit_id) ON UPDATE CASCADE ON DELETE CASCADE” to ensure that whenever a modification is made to the column benefit_id the same modification will be made to the retirement_id column.

5- Disabled table:

```
CREATE TABLE disabled
(
   disabled_id      INT UNSIGNED,
   disability_type VARCHAR(100) not null,
   PRIMARY KEY (disabled_id, disability_type),
   FOREIGN KEY (disabled_id) REFERENCES benefits(benefit_id) ON UPDATE CASCADE
   ON DELETE CASCADE
);
```

Figure 12: disabled table

The table disabled has 2 colums: disabled_id of domain INT UNSIGNED and disability_type of domain VARCHAR(100) with a NOT NULL constraint to ensure that no null values are entered to this column, both disabled_id and disability_type are the primary keys of this table specified by the integrity constraint: “PRIMARY KEY (disabled_id, disability_type). disabled_id is a foreign key referencing the column benefit_id of the table benefits declared by using the integrity constraint: “FOREIGN KEY (disabled_id) REFERENCES benefits(benefit_id) ON UPDATE CASCADE ON DELETE CASCADE” to ensure that whenever a modification is made to the column benefit_id the same modification will be made to the disabled_id column.

6- law_judge table:

```
CREATE TABLE law_judge
(
   judge_name     VARCHAR(40),
   type_of_court VARCHAR(30) not null,
   PRIMARY KEY(judge_name)
);
```

Figure 13: law_judge table

The law_judge table has 2 columns: judge_name of domain VARCHAR(40), that is also the primary key of this table declared by using the integrity constraint: “PRIMARY KEY (judge_name)” and the second column: type_of_court of domain VARCHAR(30) with a NOT NULL integrity constraint to ensure that a valid name of court is entered and not a null value.

7- medical_operations table:

```
CREATE TABLE medical_operations
(
   operation_name        VARCHAR(100),
   operation_description VARCHAR(100) not null,
   PRIMARY KEY (operation_name)
);
```

Figure 14: medical_operations table

The medical_operations table has 2 columns: operation_name of domain: VARCHAR(100) that is also the primary key of this table specified by the integrity constraint: "PRIMARY KEY (operation_name)", and the operation_description column of domain VARCHAR(100) with a NOT NULL constraint to ensure that no null values are assigned to this column.

8- medical_program table:

```
CREATE TABLE medical_program
(
   program_id INT UNSIGNED,
   PRIMARY KEY (program_id)
);
```

Figure 15: medical_program table

This table has only one column: program_id of domain: INT UNSIGNED, it is the primary key of the table (PRIMARY KEY (program_id)).

9- insurance table:

```
CREATE TABLE insurance
(
   insurance_id      INT UNSIGNED,
   insurance_type VARCHAR(30) not null,
   PRIMARY KEY(insurance_id),
   FOREIGN KEY (insurance_id) REFERENCES medical_program(program_id) ON UPDATE
   CASCADE ON DELETE CASCADE
);
```

Figure 16: insurance table

The insurance table has 2 columns: insurance_id of domain: INT UNSIGNED that is also the primary key of the table (PRIMARY KEY (insurance_id)), insurance_type of domain VARCHAR(30) with NOT NULL integrity constraint.
Insurance_id is a foreign key that references program_id in the medical_program table ("FOREIGN KEY (insurance_id) REFERENCES benefits(program_id) ON UPDATE CASCADE ON DELETE CASCADE" to ensure that whenever a modification is made to the column program_id the same modification will be made to the insurance_id column).

10- health table:

```
CREATE TABLE health
(
   health_id INT UNSIGNED,
   PRIMARY KEY(health_id),
   FOREIGN KEY (health_id) REFERENCES medical_program(program_id) ON UPDATE
   CASCADE ON DELETE CASCADE
);
```

Figure 17: health table

This table has 1 column: health_id of domain: INT UNSIGNED, it is the primary key of the table (PRIMARY KEY(health_id)) and also a foreign key referencing the column program_id from the table medical_program ("FOREIGN KEY (health_id) REFERENCES benefits(program_id) ON UPDATE CASCADE ON DELETE CASCADE" to ensure that whenever a modification is made to the column program_id the same modification will be made to the health_id column).

11- online_account table:

```
CREATE TABLE online_account
(
   account_id    INT UNSIGNED,
   creation_date DATE NOT NULL,
   PRIMARY KEY(account_id)
);
```

Figure 18: online account table

This table contains 2 columns: account_id of domain: INT UNSIGNED, that is also the primary key of the table (PRIMARY KEY (account_id)), and creation_date of domain: DATE with a NOT NULL integrity constraint.

12- person table:

```
CREATE TABLE person
(
   person_id      INT UNSIGNED,
   first_name     VARCHAR(20) not null,
   middle_name    VARCHAR(20),
   last_name      VARCHAR(20) not null,
   gender         VARCHAR(10) not null,
   marital_status VARCHAR(20) DEFAULT "SINGLE",
   date_of_birth  DATE,
   rue            VARCHAR(30) NOT NULL,
   building_name  VARCHAR(30) NOT NULL,
   floor_nb       TINYINT NOT NULL,
   account_id     INT UNSIGNED,
   PRIMARY KEY(person_id),
   FOREIGN KEY (rue, building_name, floor_nb) REFERENCES address(rue,
   building_name, floor_nb) ON UPDATE CASCADE ON DELETE RESTRICT,
   FOREIGN KEY (account_id) REFERENCES online_account(account_id) ON UPDATE
   CASCADE ON DELETE CASCADE
);
```

Figure 19: person table

The table person has 11 columns: person_id, first_name, middle_name, last_name, gender, marital_status, date_of_birth, rue, building_name, floor_nb, and account_id. The primary key of this table is person_id specified by the integrity constraint: “PRIMARY KEY (person_id)”. The 3 columns: rue, building_name and floor_nb are foreign keys referencing the 3 columns: rue, building_name and floor_nb of the table address “FOREIGN KEY (rue, building_name, floor_nb) REFERENCES address(rue,

building_name, floor_nb) ON UPDATE CASCADE ON DELETE RESTRICT", meaning that the deletion is restricted to maintain referential integrity.
account_id is also a foreign key thate references account_id from the table online_account("FOREIGN KEY (account_id) REFERENCES online_account(account_id) ON UPDATE CASCADE ON DELETE CASCADE" to ensure that whenever a modification is made to the column account_id the same modification will be made to the account_id column).

13- worker table:

```
CREATE TABLE worker
(
    worker_id       INT UNSIGNED,
    corporation_id  INT UNSIGNED NOT NULL,
    worker_role     VARCHAR(20),
    starting_year   SMALLINT,
    benefit_id      INT UNSIGNED NOT NULL,
    rue             VARCHAR(30) NOT NULL,
    building_name   VARCHAR(30) NOT NULL,
    floor_nb        TINYINT NOT NULL,
    PRIMARY KEY(worker_id),
    FOREIGN KEY (corporation_id) REFERENCES corporation(corporation_id) ON
    UPDATE CASCADE ON DELETE CASCADE,
    FOREIGN KEY (benefit_id) REFERENCES benefits(benefit_id) ON UPDATE CASCADE
    ON DELETE SET DEFAULT,
    FOREIGN KEY (rue, building_name, floor_nb) REFERENCES address(rue,
    building_name, floor_nb) ON UPDATE CASCADE ON DELETE RESTRICT,
    FOREIGN KEY (worker_id) REFERENCES person(person_id) ON UPDATE CASCADE ON
    DELETE CASCADE
);
```

Figure 20: worker table

The worker table has 7 columns: worker_id, corporation_id, worker_role, starting_year, benefit_id, rue, building_name, floor_nb. The primary key in this table is worker_id (PRIMARY KEY (worker_id)).
corporation_id is a foreign key that references corporation_id from the table corporation("FOREIGN KEY (corporation_id) REFERENCES corporation(corporation_id) ON UPDATE CASCADE ON DELETE CASCADE" to ensure that whenever a modification is made to the column corporation_id the same modification will be made to the corporation_id column).
benefit_id is also a foreign key that references benefit_id from the table benefits "FOREIGN KEY (benefit_id) REFERENCES benefits(benefits_id) ON UPDATE CASCADE ON DELETE SET DEFAULT" meaning that any modifications in the benefit_id column in the benefits table, the benefit_id column in the worker table will be set to its default value.
The 3 columns: rue, building_name and floor_nb are foreign keys referencing the 3 columns: rue, building_name and floor_nb of the table address "FOREIGN KEY (rue, building_name, floor_nb)

REFERENCES address(rue, building_name, floor_nb) ON UPDATE CASCADE ON DELETE RESTRICT", meaning that the deletion is restricted to maintain referential integrity.
worker_id is also a foreign key that references person_id from the table person "FOREIGN KEY (worker_id) REFERENCES person(person_id) ON UPDATE CASCADE ON DELETE CASCADE" to ensure that whenever a modification is made to the column corporation_id the same modification will be made to the corporation_id column.

14- employee table

```
CREATE TABLE employee
(
   employee_id     INT UNSIGNED,
   salary        INT UNSIGNED,
   retirement_id    INT UNSIGNED NOT NULL,
   rue           VARCHAR(30) NOT NULL,
   building_name VARCHAR(30) NOT NULL,
   floor_nb      TINYINT NOT NULL,
   PRIMARY KEY(employee_id),
   FOREIGN KEY (employee_id) REFERENCES person(person_id) ON UPDATE CASCADE ON
   DELETE CASCADE,
   FOREIGN KEY (rue, building_name, floor_nb) REFERENCES address(rue,
   building_name, floor_nb) ON UPDATE CASCADE ON DELETE RESTRICT,
   FOREIGN KEY (retirement_id) REFERENCES retirement(retirement_id) ON UPDATE CASCADE
   ON DELETE SET DEFAULT
);
```

Figure 21: employee table

The table employee has 4 columns: employee_id, salary, retirement_id, rue, building_name, floor_nb. The primary key of this table is employee_id (PRIMARY KEY (employee_id)), it is also a foreign key referencing the column person_id from the table person (FOREIGN KEY (employee_id) REFERENCES person(person_id) ON UPDATE CASCADE ON DELETE CASCADE).
The 3 columns: rue, building_name and floor_nb are foreign keys referencing the 3 columns: rue, building_name and floor_nb of the table address "FOREIGN KEY (rue, building_name, floor_nb) REFERENCES address(rue, building_name, floor_nb) ON UPDATE CASCADE ON DELETE RESTRICT", meaning that the deletion is restricted to maintain referential integrity.
retirement_id is also a foreign key that references retirement_id from the table retirement"FOREIGN KEY (retirement_id) REFERENCES retirement(retirement_id) ON UPDATE CASCADE ON DELETE SET DEFAULT" meaning that any modifications in the retirement_id column in the retirement table, the retirement_id column in the employee table will be set to its default value.

15- has_address table:

```
CREATE TABLE has_address
  (
     rue            VARCHAR(30),
     building_name  VARCHAR(30),
     floor_nb       TINYINT,
     corporation_id INT UNSIGNED,
     PRIMARY KEY( rue, building_name, floor_nb, corporation_id),
     FOREIGN KEY (rue, building_name, floor_nb) REFERENCES address(rue,
     building_name, floor_nb) ON UPDATE CASCADE ON DELETE RESTRICT,
     FOREIGN KEY(corporation_id) REFERENCES corporation(corporation_id) ON
     UPDATE CASCADE ON DELETE CASCADE
  );
```

Figure 22: has_address table

The has_address table has 4 columns: rue, building_name, floor_nb, and corporation_id. All the columns are primary keys (PRIMARY KEY(rue, building_name, floor_nb, corporation_id)).
The 3 columns: rue, building_name and floor_nb are foreign keys referencing the 3 columns: rue, building_name and floor_nb of the table address "FOREIGN KEY (rue, building_name, floor_nb) REFERENCES address(rue, building_name, floor_nb) ON UPDATE CASCADE ON DELETE RESTRICT". Corporation_id is also a foreign key referencing the column corporation_id from the table corporation (FOREIGN KEY (corporation_id) REFERENCES corporation(corporation_id) ON UPDATE CASCADE ON DELETE CASCADE).

16- drugs table:

```
CREATE TABLE drugs
(
   drug_id      INT UNSIGNED,
   emonth       SMALLINT,
   eyear        SMALLINT,
   type_of_drug VARCHAR(100),
   pmonth       SMALLINT,
   pyear        SMALLINT,
   manufacturer VARCHAR(50),
   PRIMARY KEY(drug_id)
);
```

Figure 23: drugs table

The table drugs has 7 columns: drug_id, emonth, eyear, type_of_drug, pmonth, pyear, and manufacturer. Its primary key is drug_id "PRIMARY KEY (drug_id)".

17- consumes table:

```
CREATE TABLE consumes
(
   worker_id               INT UNSIGNED,
   health_id              INT UNSIGNED,
   drug_id                 INT UNSIGNED,
   percentage_of_coverage SMALLINT,
   PRIMARY KEY(drug_id, worker_id, health_id),
   FOREIGN KEY (health_id) REFERENCES health(health_id) ON UPDATE
   CASCADE ON DELETE CASCADE,
   FOREIGN KEY (worker_id) REFERENCES worker(worker_id) ON UPDATE CASCADE ON
   DELETE CASCADE,
   FOREIGN KEY (drug_id) REFERENCES drugs(drug_id) ON UPDATE CASCADE ON DELETE
   RESTRICT
);
```

Figure 24: consumes table

The table has 4 columns: worker_id, health_id, drug_id (primary keys of the table) and percentage_of_coverage. Health_id is a foreign key that references health_id in the table health, and any deletion or addition to a health_id in the health table will be deleted/added to the consumes table (ON UPDATE CASCADE ON DELETE CASCADE).
Also worker_id is a foreign key that references worker_id from worker table and any modification in the worker_id in the worker table will also be modified in the consumes table.
drug_id is a foreign key that references drug_id in the table drugs, where the deletion is restricted to maintain referential integrity (the deletion of a row in the drugs table will be restricted if there are corresponding rows in the current table that reference it.

18- enrolls in table:

```
CREATE TABLE enrolls_in
(
   person_id          INT UNSIGNED,
   program_id         INT UNSIGNED,
   date_of_enrollment DATE,
   PRIMARY KEY( person_id, program_id ),
   FOREIGN KEY (program_id) REFERENCES medical_program(program_id) ON UPDATE
   CASCADE ON DELETE RESTRICT,
   FOREIGN KEY (person_id) REFERENCES person(person_id) ON UPDATE CASCADE ON
   DELETE CASCADE
);
```

Figure 25: enrolls in table

In this table there are 3 columns: person_id, program_id, and date_of_enrollment. person_id and program_id are the primary keys.
program_id is a foreign key that references program_id in the medical_program table where the deletion is restricted: a progam_id from the table medical_program cannot be deleted if there are corresponding rows that are referencing it from the table enrolls_in.
person_id is also a foreign key that references person_id from the table person, and each modification in the person_id of the person table affects the person_id in the enrolls_in table.

19- appeal_process table:

```
CREATE TABLE appeal_process
(
   process_id          INT UNSIGNED,
   process_type        VARCHAR(100),
   date_of_application DATE,
   worker_id           INT UNSIGNED NOT NULL,
   judge_name          VARCHAR(40) NOT NULL,
   place               VARCHAR(20),
   PRIMARY KEY(process_id),
   FOREIGN KEY (worker_id) REFERENCES worker(worker_id) ON UPDATE CASCADE ON
   DELETE RESTRICT,
   FOREIGN KEY (judge_name) REFERENCES law_judge(judge_name) ON UPDATE CASCADE
   ON DELETE RESTRICT
);
```

Figure 26: appeal_process table

The table has 6 columns: process_id, process_type, date_of application, worker_id, judge_name and place. Where process_id id the primary key “PRIMARY KEY (process_id)”. There are 2 foreign keys in the table: 1) worker_id referencing worker_id from the worker table, and 2) judge_name referencing judge_name from the law_judge table. Both restrict the deletion of a row in the referenced table “ON DELETE RESTRICT”.

20- dependants table:

```
CREATE TABLE dependants
(
    dependants_name VARCHAR(30),
    worker_id       INT UNSIGNED,
    gender          VARCHAR(10),
    date_of_birth   DATE,
    PRIMARY KEY(dependants_name, worker_id),
    FOREIGN KEY (worker_id) REFERENCES worker(worker_id) ON UPDATE CASCADE ON
    DELETE CASCADE
);
```

Figure 27: dependants table

The dependants table has 4 columns: dependants_name, worker_id (that are the primary keys of the table), in addition to gender, and date_of_birth. Worker_id is a foreign key that references worker_id from the worker table, where each modification (update or delete) in the worker table will affect the worker_id in the dependants table.

21- statement table:

```
CREATE TABLE statement
(
    statement_id   INT,
    statement_name VARCHAR(20),
    statement_description     VARCHAR(100),
    account_id     INT UNSIGNED,
    statement_date DATE,
    PRIMARY KEY(statement_id),
    FOREIGN KEY(account_id) REFERENCES online_account(account_id) ON UPDATE
    CASCADE ON DELETE CASCADE
);
```

Figure 28: statement table

The statement table contains 5 columns: statement_id(primary key), statement_name, statement_description, account_id and statement_date.
account_id is a foreign key that references account_id from the online_account table, where each modification in the account_id of the table online_account will affect the account_id of the statement table.

22- does table:

```
CREATE TABLE does
(
   worker_id          INT UNSIGNED,
   operation_name     VARCHAR(30),
   insurance_id         INT UNSIGNED,
   cost               FLOAT,
   status             VARCHAR(30) not null,
   date_of_operation DATE,
   PRIMARY KEY( worker_id, operation_name, insurance_id ),
   FOREIGN KEY (insurance_id) REFERENCES insurance(insurance_id) ON UPDATE
   CASCADE ON DELETE RESTRICT,
   FOREIGN KEY (worker_id) REFERENCES worker(worker_id) ON UPDATE CASCADE ON
   DELETE CASCADE,
   FOREIGN KEY (operation_name) REFERENCES medical_operations(operation_name)
   ON UPDATE CASCADE ON DELETE RESTRICT
);
```

Figure 29: does table

The does table contains 6 columns: worker_id, operation_name, insurance_id (primary keys of the table), cost, status and date_of_operation.
insurance_id is a foreign key that references insurance_id from the insurance table, also operation_name is a foreign key that references operation_name from the table medical_operation: both have a restriction on the deletion of rows in the referenced table.
worker_id is also a foreign key that references worker_id from the worker table, where each modification in the worker table will affect the does table.

DML queries

In order to populate the tables, first we used the database: “use project_db”, and a basic DML query is used: INSERT INTO *name_of_table(atrr1, attr2,..)* VALUES (*value_of attr1, value_of_attr2,…)*

(value_of_attr1, value_of_attr2, …);

```sql
use project__db;

INSERT INTO law_judge (judge_name, type_of_court) VALUES
('Magistrate Smith', 'Magistrate Court'),
('Magistrate Johnson', 'Magistrate Court'),
('Judge Brown', 'Probate Court'),
('Judge Davis', 'Probate Court'),
('Judge John', 'Supreme Court'),
('Judge Jack', 'District Court');

Insert Into online_Account(account_id,creation_date)
Values(101,'2022-01-01'),
(102,'2022-05-01'),
(103,'2021-06-01'),
(104,'2019-06-01'),
(105,'2020-05-01'),
(106,'2022-01-01');
```

Figure 30: tables population

Basic SQL Queries

<u>Query 1:</u> for every process type, get the number of workers that requested it

SELECT process_type, Count(*)

FROM worker

JOIN appeal_process using (worker_id)

GROUP BY process_type;

Explanation: in this query we need the get the count of the workers that requested a certain process type, so for each process type (group by) get the count of workers. Therefore, we join the worker table and the appeal_process table based on the woker_id, so the workers that requested a certain process will appear in this table. Now we group by the process_type and get the count for each type.

process_type	Count(*)
Appeal for Disability Benefits	1
Workplace Injury Appeal	1
Social Security Appeal	1

Figure 31: query 1 output

<u>Query 2:</u> Retrieve the statements for a specific account within a given date range

SELECT statement_name

FROM statement

WHERE statement_date BETWEEN 2023-01-31 AND 2023-04-15

AND account_id = 102;

Explanation: we are asked to get the statement (can get either statement_name or statement_id), the statement has to be within a specific date range (here we chose it to be between 2023-01-31 and 2023-04-15), and finaly since we need the statement for a specific account_id we specify it in the conditions (where statement), here we chose it to be 102.

statement_name
Quarterly Report

Figure 32: query 2 output

<u>Query 3:</u> Find the number of appeals processed by each law judge

SELECT judge_name, Count(*)

FROM appeal_process

GROUP BY judge_name;

Explanation: From the table appeal_process we need to get the the number of appeals processed by each law judge, hence we need to group by law judge, and then select the count along with the judge_name.

judge_name	Count(*)
Judge Davis	1
Magistrate Johnson	1
Magistrate Smith	1

Figure 33: query 3 output

<u>Query 4:</u> Find the average salary of employees who have a retirement plan

SELECT Avg(salary)

FROM employee e, retirement r

WHERE e.retirement_id = r.retirement_id;

Explanation: in this query we are asked to get the average salary of employees who have a retirement plan, hence we need to select the Avg(salary), from the table employee self-join retirement (used self-join for diversity). With a where condition that the retirement_id of the employee is equal to the retirement_id in the retirement table (to ensure that the employee has a retirement plan).

Avg(salary)
65000.0000

Figure 34: query 4 output

<u>Query 5:</u> Retrieve the names of people who underwent a medical operation of cost >5000

```
SELECT  CONCAT(first_name, middle_name, last_name)
FROM   person
WHERE  person_id IN (SELECT person_id
               FROM   does
               WHERE  cost > 5000);
```

Explanation: the objective is to get the names of people who underwent a medical operation of cost > 5000, but since person has the name divided into first_name, last_name and middle_name, we concatenate them in the select statement using: CONCAT(first_name, middle_name, last_name), we select them from the table person where the person's id exists in the does table and the cost is >5000. (we can also join person and does but for more diversity in the queries IN).

CONCAT(first_name, middle_name, last_name)
JohnADoe
JaneBSmith
AliceCJohnson
MichaelDWilliams
SophiaEMiller
ChristopherFBrown

Figure 35: query 5 output

Query 6: List the corporations that have number of employees greater than or equal to 1

```
SELECT C.corporation_name
FROM   corporation C
WHERE  EXISTs (SELECT corporation_id
          FROM   corporation
          JOIN worker using(corporation_id)
          WHERE C.corporation_id = corporation_id);
```

Explanation: in this query we aim to list the corporations (name or id, we chose name), that have the number of employees greater than or equal to 1. So if a worker has the same corporation_id as the corporation we are checking than we select it.

corporation_name
ABC Corporation
XYZ Corporation
HealthCare Solutions

Figure 36: query 6 output

Query 7: Find the average percentage of drug coverage for employees in each corporation

```
SELECT corporation_id,
         Avg(percentage_of_coverage)
FROM   worker
JOIN consumes using(worker_id)
GROUP  BY corporation_id;
```

Explanation: in this query we need to get the average percentage of drug coverage for employees in each corporation, for this we have to select the corporation_id and the Avg(percentage_of_coverage) from the table worker join consumes using the worker_id, so we group by the corporation_id, and for each corporation, we get the avg(percentage_of_coverage).

corporation_id	Avg(percentage_of_coverage)
1	80.0000
3	45.0000

Figure 37: query 7 output

<u>Query 8</u>: Find the total cost of medical operations for each corporation, considering all workers

```
SELECT corporation_id,
       Sum(cost)
FROM   worker
JOIN does using(worker_id)
GROUP  BY corporation_id;
```

Explanation: to get the total cost of medical operations for each corporation, we need to get the sum of cost along with the corporation_id. Since fo each corporation we need the total cost then we group by the corporation_id, and we select the sum of cost from the table worker joins does(to get the cost for each worker).

corporation_id	Sum(cost)
1	5000
2	8000
3	12000

Figure 38: query 8 output

<u>Query 9</u>: List all persons who have enrolled in insurance and the corresponding insurance type

```
SELECT CONCAT(first_name, middle_name, last_name), insurance_type
FROM   enrolls_in
        NATURAL JOIN insurance
        NATURAL JOIN person;
```

Explanation: in this query we need to list the people who have enrolled in insurance with the type of the insurance. For this objective we naturally join the tables enrolls_in and insurance first where they have only one column in commune: program_id, and also naturally joined with the person table (based on the person_id), and the name of the person is selected by concatenating the first, middle and last name, and also select the insurance type.

CONCAT(first_name, middle_name, last_name)	insurance_type
JohnADoe	Vision Insurance
JohnADoe	Dental Insurance
JohnADoe	Health Insurance
AliceCJohnson	Vision Insurance
AliceCJohnson	Dental Insurance
AliceCJohnson	Health Insurance
JaneBSmith	Vision Insurance
JaneBSmith	Dental Insurance
JaneBSmith	Health Insurance
ChristopherFBrown	Vision Insurance
ChristopherFBrown	Dental Insurance
ChristopherFBrown	Health Insurance

Figure 39: query 9 output

Query 10: Retrieve the names of workers who have retirement benefits and are also enrolled in a specific medical program

```
SELECT p.first_name, p.middle_name, p.last_name

FROM   person p

WHERE p.person_id IN

          (SELECT person_id

          FROM   worker

          JOIN retirement on (benefit_id=retirement_id)

          UNION

          SELECT person_id

          FROM   enrolls_in

          JOIN medical_program using (program_id));
```

Explanation: in order to get the names of workers who have retirement benefits and are enrolled in a specific medical program, we select the name of the worker from the person table where the id of this worker is present in the worker join retirement table, and also in the enrolls_in table join medical_program so the inner query will select the person_id that has retirement benefits and is enrolled in a medical program.

first_name	middle_name	last_name
John	A	Doe
Jane	B	Smith
Alice	C	Johnson
Michael	D	Williams
Sophia	E	Miller
Christopher	F	Brown

Figure 40: query 10 output

<u>Query 11</u>: List all comapanies whose workers never consumed a drug

SELECT corporation_id

FROM corporation

EXCEPT

SELECT w.corporation_id

FROM worker w

WHERE w.worker_id IN (SELECT worker_id FROM consumes);

Explanation: in this query we select all the corporation_id s except the ones that are selected from the worker table where the worker_id is in the consumes table.

corporation_id
2
4
5

Figure 41: query 11 output

<u>Query 12</u>: List all workers who share the same address of their coroporation

SELECT *

FROM worker

NATURAL JOIN has_address;

Explanation: to get all workers who share the same address of their corporation, we naturally join the worker table with the has_address table, so they will be joined technically based on the rue, building_name, floor and corporation_id.

	worker_id	corporation_id	worker_role	starting_year	benefit_id	rue	building_name	floor_nb
▶	1	1	Manager	2010	1	123 Main St	Corporate Tower	10
	2	2	regular	2015	2	456 Oak St	Business Center	5
	3	3	Executive Level	2011	3	789 Market St	Tech Plaza	15
	6	3	Executive Level	2010	3	101 Innovation Ave	Inno Tower	8
	8	1	Manager	2010	1	123 Main St	Corporate Tower	10
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 42: query 12 output

<u>Query 13</u>: for each company, list the law judges interacting with it

```
SELECT DISTINCT corporation_name, judge_name

FROM worker

JOIN appeal_process using (worker_id)

JOIN corporation  using (corporation_id);
```

Explanation: to get the judge_name interacting with each company, we select the distinct corporation_name and judge_name (to ensure that if the judge_name dealt with the corporation multiple times, it will be only selected once), from the table worker join appeal_process, since if a worker in a corporation asked for an appeal_process it will be processed by a judge, and the worker_id along with the judge_name will be in the appeal_process_table, and then we join this table with corporation to get the corporation_name.

corporation_name	judge_name
ABC Corporation	Magistrate Smith
XYZ Corporation	Magistrate Johnson
HealthCare Solutions	Judge Davis

Figure 43: query 13 output

<u>Query 14</u>: List the workers of all companies consisting of one worker only

```
SELECT *

FROM worker w

JOIN (

   SELECT corporation_id

   FROM worker

   GROUP BY corporation_id

   HAVING COUNT(*) = 1

) AS T ON w.corporation_id = T.corporation_id;
```

Explanation: in this query we are asked to get the list of workers of all companies that consists of only 1 worker: we select the workers from worker join the table that contains the corporation_id's that have a count = 1, based on the corporation_id.

worker_id	corporation_id	worker_role	starting_year	benefit_id	rue	building_name	floor_nb	corporation_id
1	1	Manager	2010	1	123 Main St	Corporate Tower	10	1
2	2	regular	2015	2	456 Oak St	Business Center	5	2

Figure 44: query 14 output

<u>Query 15</u>: Find the number of workers in each corporation who are not enrolled in any medical program

SELECT w.corporation_id, Count(w.worker_id)

FROM worker w

WHERE w.worker_id NOT IN (SELECT person_id FROM enrolls_in NATURAL JOIN medical_program)

GROUP BY w.corporation_id;

Explanation: since we need the number of workers in each corporation, we group by the corporation_id, and we select the corporation_id, and the count of worker_id, from the table worker, where the worker_id is not in the table that gets the workers that are enrolled in a medical program.

corporation_id	Count(w.worker_id)
1	2
10	1

Figure 45: query 15 output

<u>Query 16</u>: Find the average age of dependents for workers in each corporation

SELECT corporation_id, AVG(TIMESTAMPDIFF(YEAR, date_of_birth, CURDATE()))

FROM worker

JOIN dependants using (worker_id)

GROUP BY corporation_id;

Explanation: in this query we are asked to find the average age of dependents in each corporation so we group by the corporation_id, and select from the table worker join dependants, the corporation_id along with the average age. Here we used a built in function: TIMESTAMPDIFF that gets the difference between the current date and the date of birth for each worker.

	corporation_id	AVG(TIMESTAMPDIFF(YEAR, date_of_birth, CURDATE()))
▶	1	13.0000
	2	11.0000
	3	6.5000

Figure 46: query 16 output

Query 17: Get all the workers who's name starts with an M that do not have dependents

SELECT worker_id

FROM worker

JOIN person on(worker_id=person_id)

WHERE first_name LIKE 'M%'

EXCEPT

SELECT worker_id

FROM dependants;

Explanation: to get the workers that have their names starts with an M, we check the condition in the where statement: first_name LIKE 'M%'. and for the next part of the query where the workers don't have dependents, we choose the workers that are not in the dependants table.

	worker_id
▶	6
	10
	11

Figure 47: query 17 output

Query 18: Find the number of processes processed by each law judge in a magistrate court

SELECT judge_name, Count(worker_id)

FROM appeal_process

JOIN law_judge using (judge_name)

WHERE type_of_court = "magistrate court"

GROUP BY judge_name;

Explanation: to get the number of processes processed by each law_jude in an magistrate court, from the table appeal_process join law_judge we select the judge name and the count of worker_id (the judge will process an appeal process only if a worker requested it), where the type of court = “magistrate court” and since were counting for each law judge then we group by the judge name.

judge_name	Count(worker_id)
Magistrate Johnson	1
Magistrate Smith	1

Figure 48: query 18 output

Query 19: Find the number of persons monitored by each law judge in an Magistrate court

```
SELECT judge_name, Count(DISTINCT worker_id)
FROM   appeal_process
       JOIN law_judge using (judge_name)
WHERE  type_of_court = "magistrate court"
GROUP  BY judge_name;
```

Explanation: to get the number of persons monitored by each law_jude in a Magistrate court, from the table appeal_process join law_judge we select the judge name and the count of distinct worker_id (since we need to count for each person once), where the type of court = “magistrate court” and since were counting for each law judge then we group by the judge name.

judge_name	Count(DISTINCT worker_id)
Magistrate Johnson	1
Magistrate Smith	1

Figure 49: query 19 output

Note that query 18 and query 19 have the same output, since as explained in the query 18, the law judge won’t process an appeal process unless a worker requested it.

Query 20: Find the number of workers in each corporation who are enrolled in an insurance program

```
SELECT W.corporation_id,
       Count(DISTINCT W.worker_id)
FROM   worker AS W
       JOIN corporation  C ON ( W.corporation_id = C.corporation_id )
       JOIN enrolls_in  E ON ( W.worker_id = E.person_id )
       JOIN insurance  I ON ( E.program_id = I.insurance_id )
GROUP  BY W.corporation_id;
```

Explanation: to get the number of workers in each corporation that are enrolled in an insurance program from the table worker join corporation (since we are finding the number in each coerporation) join enrolls_in and insurance, we select the corporation_id along with the count of distinct worker_id (since we want to count the worker once even if enrolled in many programs) and we group by corporation_id.

corporation_id	Count(DISTINCT W.worker_id)
1	1
2	1
3	2

Figure 50: query 20 output

Query 21: List all employees who have an online account and the total number of statements they have

```
SELECT *
FROM   person
            NATURAL JOIN employee
    NATURAL JOIN (SELECT account_id, Count(statement_id) AS nb_of_statements
                  FROM   online_account
                  JOIN statement using(account_id)
                  GROUP  BY account_id) as T;
```

Explanation: the inner query gets the account_id and the number of statements for each online account, the outer query naturally joins the person table with the employee and the table of the inner query, therefore we will obtain the employees that have an online account along with the total number of statements .

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

account_id	rue	building_name	floor_nb	person_id	first_name	middle_name	last_name	gender	marital_status	date_of_birth	employee_id	salary	retirement_id	nb_of_stater
101	123 Main St	Corporate Tower	10	1	John	A	Doe	Male	SINGLE	1990-05-15	4	70000	3	1
102	456 Oak St	Business Center	5	2	Jane	B	Smith	Female	MARRIED	1988-08-25	5	60000	6	1

Figure 51: query 21 output

Query 22: list all workers who have at least one coworker

```
SELECT DISTINCT w.worker_id
FROM worker w
JOIN worker co ON (co.worker_id <> w.worker_id AND co.corporation_id = w.corporation_id);
```

Explanation: to get the workers who have at least one coworker, we select the workers that work in the same corporation (have the same corporation id), so we join the table worker with itself and for each 2 workers if they don't have the same worker_id and have the same corporation_id we select them.

	worker_id
▶	3
	6

Figure 52: query 22 output

Advanced SQL queries

Query 1: Find workers who have not applied for any medical programs.

```
Select *
from worker W
where NOT Exists (Select *
                  from Enrolls_in E
                  where W.worker_id = E.person_id);
```

Explanation: the subquery uses the EXISTS clause in order to check the existence of worker_id's in the Enrolls_in where the worker's ID from the outer query (W.worker_id) matches the person ID in the Enrolls_in table (E.person_id). The NOT EXISTS condition is used to filter out workers who have applied for medical programs.

	worker_id	corporation_id	worker_role	starting_year	benefit_id	rue	building_name	floor_nb
▶	8	1	Manager	2010	1	123 Main St	Corporate Tower	10
	10	1	Manager	2010	1	123 Main St	Corporate Tower	10
	11	10	Manager	2010	1	123 Main St	Corporate Tower	10
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 53: query 23 output

Query 2: Retrieve the average earning history of employees enrolled in retirement for each corporation, along with the total number of workers in each corporation

```
Select corporation_id,count(*) as count, (Select  avg(earning_history)
                                          from  Worker W2 Join retirement R on
                                   (W2.benefit_id=R.retirement_id)
                                          where W2.corporation_id =C.corporation_id) as
                                   average_earning_history)
From Worker W Join Corporation C using (corporation_id)
GROUP BY corporation_id;
```

Explanation: This SQL query is designed to provide information about each corporation, including the count of workers associated with the corporation (count), and the average earning history for workers in that corporation. A subquery is used to compute the average earning history from the table worker join retirement for each corporation.

corporation_id	count	average_earning_history
1	1	NULL
2	1	NULL
3	2	101.0000

Figure 54: query 24 output

<u>Query 3</u>: Retrieve a list of workers with title "Regular" along with the number of dependents each has.

```
Select *, (Select count(*)
        from dependants D1 where W.worker_id =D1.worker_id) as dependants_count
from Worker W
where W.worker_id IN( Select worker_id
                      from worker
                      where worker_role LIKE "regular");
```

Explanation: This SQL query is designed to retrieve information about workers, including a count of their dependants, for workers whose roles are labeled as "regular." There are 2 subqueries in this query, one in the select statement to get the number of dependants for the worker selected, and one in the where statement to ensure that that the worker has a role “regular”.

worker_id	corporation_id	worker_role	starting_year	benefit_id	rue	building_name	floor_nb	dependants_count
2	2	regular	2015	2	456 Oak St	Business Center	5	1

Figure 55: query 25 output

<u>Query 4</u>: Retrieve all Single workers and their corresponding medical "Completed " operations descriptions (if any)

```
Select * from Person P1
where P1.marital_status Like "single" and Exists(
Select worker_id, (Select operation_description from medical_operations where operation_name=D.operation_name)
from Does D
where status Like "Completed" and P1.person_id=D.worker_id);
```

Explanation: Outer Query (Select * From Person P1 Where P1.marital_status Like "single"):Selects all columns from the "Person" table for individuals whose marital status is "single", the EXISTS statement checks for the existence of records in the Does table. The query filters based on the completion status ("Completed") and matches the person ID from the outer query (P1.person_id) with the worker ID in the Does table (D.worker_id). The subquery (Select operation_description From medical_operations Where operation_name = D.operation_name): retrieves the operation description from the "medical_operations" table based on the operation name in the Does table.

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

	person_id	first_name	middle_name	last_name	gender	marital_status	date_of_birth	rue	building_name	floor_nb	account_id
▶	1	John	A	Doe	Male	SINGLE	1990-05-15	123 Main St	Corporate Tower	10	101
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 56: query 26 output

<u>Function 1:</u>

```
DELIMITER //

CREATE FUNCTION nb_of_employees_per_company(C_id INTEGER)

RETURNS INTEGER

BEGIN

   DECLARE c_count INTEGER;

   SELECT COUNT(*) INTO c_count

   FROM worker w

   WHERE w.corporation_id = C_id;

   RETURN c_count;

END //

DELIMITER ;
```

Description: this function gets the number of employees per company and takes as input the company id, it returns an integer which is the c_count (the number of corporations in the worker table therefore the number of employees per company).

<u>Query 5</u>: Retrieve the corporation with nb of employees >=1 ordered by average starting year of its employees

```
SELECT *

FROM corporation C2

JOIN worker USING (corporation_id)

WHERE nb_of_employees_per_company(C2.corporation_id) >=1

ORDER BY (SELECT AVG(starting_year) FROM worker w2 WHERE w2.corporation_id =
C2.corporation_id);
```

Explanation: the select statement retrieves all columns from the "corporation" and "worker" tables using a JOIN operation. the where statement filters the results to include only corporations where the number of employees, calculated using the nb_of_employees_per_company function, is greater than or equal to 1. And then we order the result set based on the average starting year of workers in each corporation. The subquery calculates the average starting year for workers within each corporation.

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

corporation_id	corporation_name	worker_id	worker_role	starting_year	benefit_id	rue	building_name	floor_nb
1	ABC Corporation	1	Manager	2010	1	123 Main St	Corporate Tower	10
1	ABC Corporation	8	Manager	2010	1	123 Main St	Corporate Tower	10
1	ABC Corporation	10	Manager	2010	1	123 Main St	Corporate Tower	10
10	BSS	11	Manager	2010	1	123 Main St	Corporate Tower	10
3	HealthCare Solutions	3	Executive Level	2011	3	789 Market St	Tech Plaza	15
3	HealthCare Solutions	6	Executive Level	2010	3	101 Innovation Ave	Inno Tower	8

Result 1

Figure 57: query 27 output

<u>Query 6:</u> Retrieve the worker with the highest number of dependents

SELECT COUNT(*) AS nb_of_dependants, worker_id

FROM worker

NATURAL JOIN dependants

GROUP BY worker_id

ORDER BY nb_of_dependants DESC

LIMIT 1;

Explanation: the query counts the number of dependants for each worker by performing a natural join between worker and dependants tables. Selects the count of dependants and the worker ID. Groups by the worker ID, so the count of dependants is calculated for each worker individually. And orders the result set in descending order based on the count of dependants (nb_of_dependants).
Limit Clause (LIMIT 1): Restricts the result set to only one record, which corresponds to the worker with the highest number of dependants due to the descending order.

nb_of_dependants	worker_id
2	3

Figure 58: query 28 output

View 1: Create a view that combines information from the worker tables along with their corporation details.

create View Employees_Combine_corporation(first_name,last_name,worker_id,worker_role,corporation_name)

as Select first_name,last_name,worker_id,worker_role,corporation_name

from Person P join worker W on P.person_id=W.worker_id Join corporation C on W.corporation_id= C.corporation_id;

Explanation: The JOIN clauses connect the tables person, worker, and corporation based on their respective IDs (person_id, worker_id, and corporation_id). The selected columns are then combined into the view named Employees_Combine_corporation.

Query 9: select * from Employees_Combine_corporation;

This query is to test the view Employees_Combine_corporation

first_name	last_name	worker_id	worker_role	corporation_name
John	Doe	1	Manager	ABC Corporation
Jane	Smith	2	regular	XYZ Corporation
Alice	Johnson	3	Executive Level	HealthCare Solutions
Christopher	Brown	6	Executive Level	HealthCare Solutions

Figure 59: query 29 output

Query 10: Update the roles of workers based on their starting year in a specific company with id = 3

UPDATE Worker W

SET worker_role = CASE

WHEN YEAR(current_date) - starting_year >= 0 AND YEAR(current_date()) - starting_year <= 2 THEN 'Entry Level'

WHEN YEAR(current_date) - starting_year >= 3 AND YEAR(current_date()) - starting_year <= 5 THEN 'Mid Level'

WHEN YEAR(current_date) - starting_year >= 6 AND YEAR(current_date()) - starting_year <= 8 THEN 'Senior Level'

WHEN YEAR(current_date) - starting_year >= 9 THEN 'Executive Level'

ELSE 'Not Applicable'

END

WHERE W.corporation_id = 3;

Explanation: this query updates the roles of workers to 3 roles based on their starting year in a company with id = 3, this is checked in the where statement (where W.corporation_id = 3).

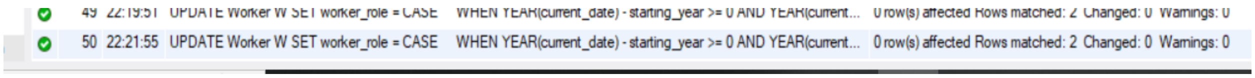

Figure 60: query 30 output

<u>Query 11</u>: Retrieve the person with the highest total drug coverage percentage

```
SELECT C.worker_id

FROM consumes AS C

GROUP BY C.worker_id

HAVING SUM(C.percentage_of_coverage) >= ALL (

  SELECT SUM(C2.percentage_of_coverage)

  FROM consumes AS C2

  GROUP BY C2.worker_id

);
```

Explanation: To get the highest total sum coverage, from the table consumes alised as C we group by the worker_id, and we select the worker that has the sum of its percentage of coverage greater than all other percentages.

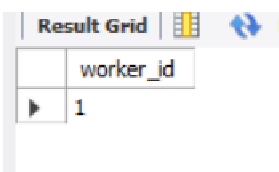

Figure 61: query 31 output

<u>Query 12:</u> Retrieve the number of dependent each worker has

```
SELECT COUNT(dependants_name) AS nb_of_dependants, worker_id

FROM worker W

right JOIN dependants D USING (worker_id)

GROUP BY worker_id;
```

Explanation: this query selects the count of the dependants_name and the worker_id from the table worker right joined with dependants table, this ensures that all rows from the dependants table are included in the result set, even if there are no matching rows in the worker table. And we group by the worker_id.

nb_of_dependants	worker_id
1	1
1	2
2	3

Figure 62: query 32 output

<u>Function 2:</u>

```
DELIMITER //

CREATE FUNCTION percentage_of_coverage_of_insurance(insurance_type VARCHAR(30))

RETURNS INTEGER

deterministic

BEGIN

  DECLARE coverage_percentage INTEGER;

  CASE insurance_type

    WHEN 'Platinum' THEN SET coverage_percentage = 90;

    WHEN 'Gold' THEN SET coverage_percentage = 70;

    WHEN 'Silver' THEN SET coverage_percentage = 60;

    ELSE SET coverage_percentage = 50;

  END CASE;

  RETURN coverage_percentage;

END //

DELIMITER ;
```

Explanation: this stored function named percentage_of_coverage_of_insurance takes an insurance_type parameter as input and returns an INTEGER. Declares a local variable named coverage_percentage of type INTEGER to store the calculated coverage percentage. And uses a CASE statement to determine the coverage percentage based on the input insurance_type. Sets the value of coverage_percentage based on the conditions specified in the CASE statement. And finally returns the calculated coverage percentage.

A table is created so that the stored function is able to store the values it needs dynamically: CorporateFunds

CorporateFunds Table:

```
CREATE TABLE CorporateFunds (
   corporate_id INT PRIMARY KEY,
   amount DECIMAL(10, 2) DEFAULT 0.00 CHECK (amount > 0),
 );
```

Description: the table has corporate_id as a primary key, amount of type decimal with default value 0.00, and the check statement ensures that the amount is always positive.

Trigger 1: to automatically add the new coroporations later

```
DELIMITER //
CREATE TRIGGER AfterInsertCorporation
AFTER INSERT
ON corporation FOR EACH ROW
BEGIN
      INSERT INTO CorporateFunds (corporate_id) VALUES (NEW.corporation_id);
END//
DELIMITER ;
```

Explanation: AFTER INSERT specifies that the trigger should be activated after the insert on the corporation table. And FOR EACH ROW indicates that the trigger is row-level and will be executed once for each row affected by the INSERT operation. It automatically insert the new values of corporation id's in the CorporateFunds table.

Trigger 2: DELIMITER //

```
CREATE TRIGGER AfterDeleteDoes
AFTER DELETE
ON does FOR EACH ROW
BEGIN
   DECLARE cost_coverage FLOAT;
   DECLARE perc_coverage FLOAT;
   DECLARE funds FLOAT;
   DECLARE corp INT UNSIGNED;
```

```
   SET perc_coverage = percentage_of_coverage_of_insurance((SELECT i.insurance_type FROM insurance i WHERE i.insurance_id = OLD.insurance_id));

   SET cost_coverage = OLD.cost * perc_coverage;

   SELECT w.corporation_id INTO corp FROM worker w WHERE w.worker_id = OLD.worker_id;

   SELECT amount INTO funds

   FROM CorporateFunds cf

   WHERE cf.corporate_id = corp;

   CALL UPDATE_FUNDS(corp, -cost_coverage);

END //

DELIMITER ;
```

Explanation: AFTER DELETE specifies that the trigger should be activated after deletion on the does table. FOR EACH ROW indicates that the trigger is row-level and will be executed once for each row affected by the DELETE operation. It declares local variables to store calculated values and results from SELECT statements. Set values for the perc_coverage and cost_coverage variables based on calculations involving the values in the deleted row. It retrieves values from the database and store them in local variables. Call a stored procedure named UPDATE_FUNDS to update the funds in the "CorporateFunds" table using the calculated cost_coverage.

<u>Trigger 3</u>:

```
 DELIMITER //

CREATE TRIGGER InsteadOfInsertDoes

BEFORE INSERT

ON does FOR EACH ROW

BEGIN

        DECLARE cost_coverage INT;

        DECLARE perc_coverage FLOAT;

        DECLARE funds FLOAT;

        DECLARE corp INT UNSIGNED;

        SET perc_coverage = percentage_of_coverage_of_insurance((SELECT i.insurance_type FROM insurance i WHERE i.insurance_id = NEW.insurance_id));

        SET cost_coverage = NEW.cost * perc_coverage;


        SELECT w.coroporation_id INTO corp FROM worker w WHERE w.worker_id = NEW.worker_id;
```

```
        SELECT amount INTO funds
        FROM CorporateFunds cf
        WHERE cf.corporation_id = corp;
        IF cost_coverage + funds > @fund_limit THEN
    SIGNAL SQLSTATE '23000'
    SET MESSAGE_TEXT = 'Integrity constraint violation: fund limit exceeded.';
  END IF;
        CALL UPDATE_FUNDS(corp, cost_coverage);
END //
DELIMITER ;
```

Explanation: BEFORE INSERT specifies that the trigger should be activated before insert on the does table. FOR EACH ROW indicates that the trigger is row-level and will be executed once for each row affected by the INSERT operation. Declare local variables to store calculated values and results from SELECT statements. Set values for the perc_coverage and cost_coverage variables based on calculations involving the values in the new row. It retrieves values from the database and store them in local variables. It checks if the sum of the calculated cost_coverage and the existing funds exceeds a predefined fund limit (@fund_limit). If the limit is exceeded, a SQLSTATE signal is raised with an integrity constraint violation message. Calls a stored procedure named UPDATE_FUNDS to update the funds in the "CorporateFunds" table using the calculated cost_coverage.

Procedure 1: -- UPDATE_FUNDS stored procedure

```
DELIMITER //
CREATE PROCEDURE UPDATE_FUNDS(IN corp_id INT UNSIGNED, IN fund_change FLOAT)
BEGIN
  UPDATE CorporateFunds
  SET amount = amount + fund_change
  WHERE corporate_id = corp_id;
END //

DELIMITER ;
```

Explanation: The procedure UPDATE_FUNDS updates the CorporateFunds table by adding the provided fund_change to the existing amount in the row where the corporate_id matches the provided corp_id. It specifies two parameters: corp_id of type INT UNSIGNED and fund_change of type FLOAT.

<u>Assertion1</u>:

CREATE ASSERTION ISA_Person

CHECK((SELECT COUNT(worker_id) FROM worker WHERE worker_id NOT IN (SELECT employee_id FROM employee)) + (SELECT COUNT(employee_id) FROM employee WHERE employee_id NOT IN (SELECT worker_id FROM worker)) = (SELECT COUNT(person_id) FROM person));

Explanation: this assertion checks if workers and employees cover person and also checks if it is disjoint.

Figure 63: Assertion1 output

Note: Consider the following sets: Person P, Employee E, and Worker W

if |W-E| + |E-W| = |C| and EUW is a subset of P, then we can prove using elementary set algebra that W intersection E is empty and E union W is P.

Thus, the disjointness and covering properties are maintained in the ISA relationship of P, E, and W.

Note that this assertion forces the user to use transactions to insert to table P. Since any insertion to P must insure a corresponding element in E or W, due to the assertion.

And vice-versa, due to the foreign key constraint. Transactions are used so that whole block of insertions are checked, instead of each insertion on its own. The following is an example:

BEGIN TRANSACTION;

-- Multiple insert statements here

COMMIT;

(The other 2 assertions are similar to this concept)

<u>Assertion2:</u>

CREATE ASSERTION ISA_medical_program

CHECK((SELECT COUNT(insurance_id) FROM insurance WHERE insurance_id NOT IN (SELECT health_id FROM health)) + (SELECT COUNT(health_id) FROM health

WHERE health_id NOT IN (SELECT insurance_id FROM insurance) = (SELECT COUNT(program_id) FROM medical_program);

Explanation: this assertion checks if insureance and health cover medical_program and also checks if it is disjoint.

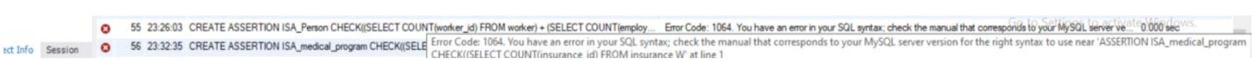

Figure 64: Assertion 2 output

<u>Assertion 3:</u>

CREATE ASSERTION ISA_Benefits

CHECK((SELECT COUNT(survivor_id) FROM survivor WHERE survivor_id NOT IN (SELECT retirement_id FROM retirement) and Not In (SELECT disabled_id FROM disabled))+ (SELECT COUNT(retirement_id) FROM retirement

WHERE retirement_id NOT IN (SELECT survivor_id FROM survivor) and Not in (SELECT disabled_id FROM disabled))+(SELECT COUNT(disabled_id) FROM disabled

WHERE disabled_id NOT IN (SELECT survivor_id FROM survivor) and Not in (SELECT retirement_id FROM retirement)) = (SELECT COUNT(benefit_id) FROM benefits));

Explanation: this assertion checks if survivor, disabled and retirement cover Benefits and also checks if it is disjoint.

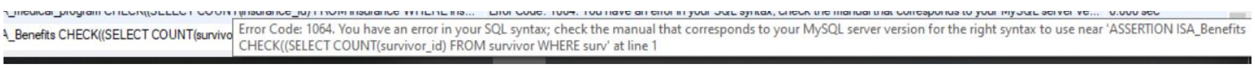

Figure 65: Assertion 3 output

Conclusion

The Social Security Administration Information System (SSAIS) is a vital component of the Social Security System, managing a diverse range of information critical for administering benefits and services. To ensure the efficiency and responsiveness of the SSAIS, continuous improvement strategies must be implemented, focusing on key areas such as indexing, query optimization, backup and recovery, and regular performance testing.

Making Searches Faster:

Indexing stands as a fundamental strategy for improving database performance, specifically enhancing search and retrieval operations. By analyzing query patterns and identifying frequently used columns, the SSAIS database can benefit from strategically implemented indexes. These indexes, applied to columns involved in WHERE clauses and JOIN conditions, significantly expedite data retrieval. Regular index maintenance tasks, such as rebuilding or reorganizing indexes, ensure that the indexes remain effective over time.

Making Conversations with the Database Smoother:

Query optimization is paramount for ensuring that database interactions are streamlined and efficient. By employing tools to profile slow-performing queries, the SSAIS can identify areas for improvement. Queries should be optimized through restructuring, the addition of appropriate indexes, and the avoidance of inefficient practices like using 'SELECT *'. The latter can be meditated by urging developers to be precise about the data columns they need. Optimized queries not only enhance the speed of data retrieval but also contribute to overall system responsiveness.

Keeping Data Safe and Recoverable:

The establishment of a robust backup and recovery strategy is crucial for safeguarding the integrity of the SSAIS database. Regular backups, performed on a scheduled basis, serve as a safety net in the event of system failures or data corruption. Furthermore, testing recovery procedures ensures the system can swiftly and accurately restore data when needed. Integrating backup practices with data archiving helps maintain a comprehensive data protection and preservation strategy.

Keeping a Record for Data Analysis:

In addition to the aforementioned improvements, it is imperative to log the transaction history of all insertions and queries within the SSAIS database. Logging transactions serves as a foundation for in-depth data analysis, offering insights into user behaviors, system usage patterns, and potential areas for further optimization. This historical data allows administrators to track changes over time, diagnose issues, and make informed decisions about future enhancements.

Views (the most crucial improvement):

The integration of views into the SSAIS database architecture introduces a layer of abstraction that simplifies data access and analysis for end-users and analysts. By creating views, users can access relevant information without directly interacting with the underlying database structure. This not only enhances the user experience but also facilitates security by restricting access to sensitive information. Views also play a crucial role in supporting more intelligent reporting and analysis, allowing for the creation of customized perspectives on the data while maintaining the integrity of the original dataset.

In simpler words, we're making sure our information system is fast, safe, and easy to use. We're also keeping a record of everything that happens so we can learn and improve, making it a helpful tool for everyone using the Social Security System.