Projeto de Base de Dados 2020/2021 - Entrega III

EATBUÉ

T1G1

André Pereira up201905650
João Marinho up201905952
Matilde Oliveira up201906954

4 de Abril de 2021

Índice

Definição do Tema

Um serviço de entregas de comida pretende guardar informações sobre os seus clientes, funcionários e serviços prestados.

Do **Client** interessa saber o/os **Card** associados (número do cartão e data de validade). Sabendo que cada Card pode estar associado a Clients diferentes e um Client pode ter mais do que um Card. Note-se que um Card pode deixar de estar associado a um Client se este assim o decidir, mas continua associado a todas as Requests feitas com esse mesmo.

Um **Employee** escolhe um **County** (concelho) de atuação de onde as entregas lhe ficarão alocadas. Cada Employee tem no mínimo um **Vehicle** associado do qual interessa saber a matrícula e **VehicleType** (por exemplo: mota, carro,...). Sabendo que um Vehicle pode ser partilhado por mais do que um Employee. A avaliação do Employee (evaluation) é obtida pela média da evaluation dada às Delivery executadas por ele.

O serviço de entregas tem parcerias com **Restaurant**s, dos quais é preciso saber o nome, morada e ementa.

Cada Restaurant tem **Period**s de funcionamento que estão definidos por uma hora de abertura e uma hora de fecho para um dia da semana. Note-se que os dias da semana vêm dado em números (como visto no modelo conceptual seguinte).

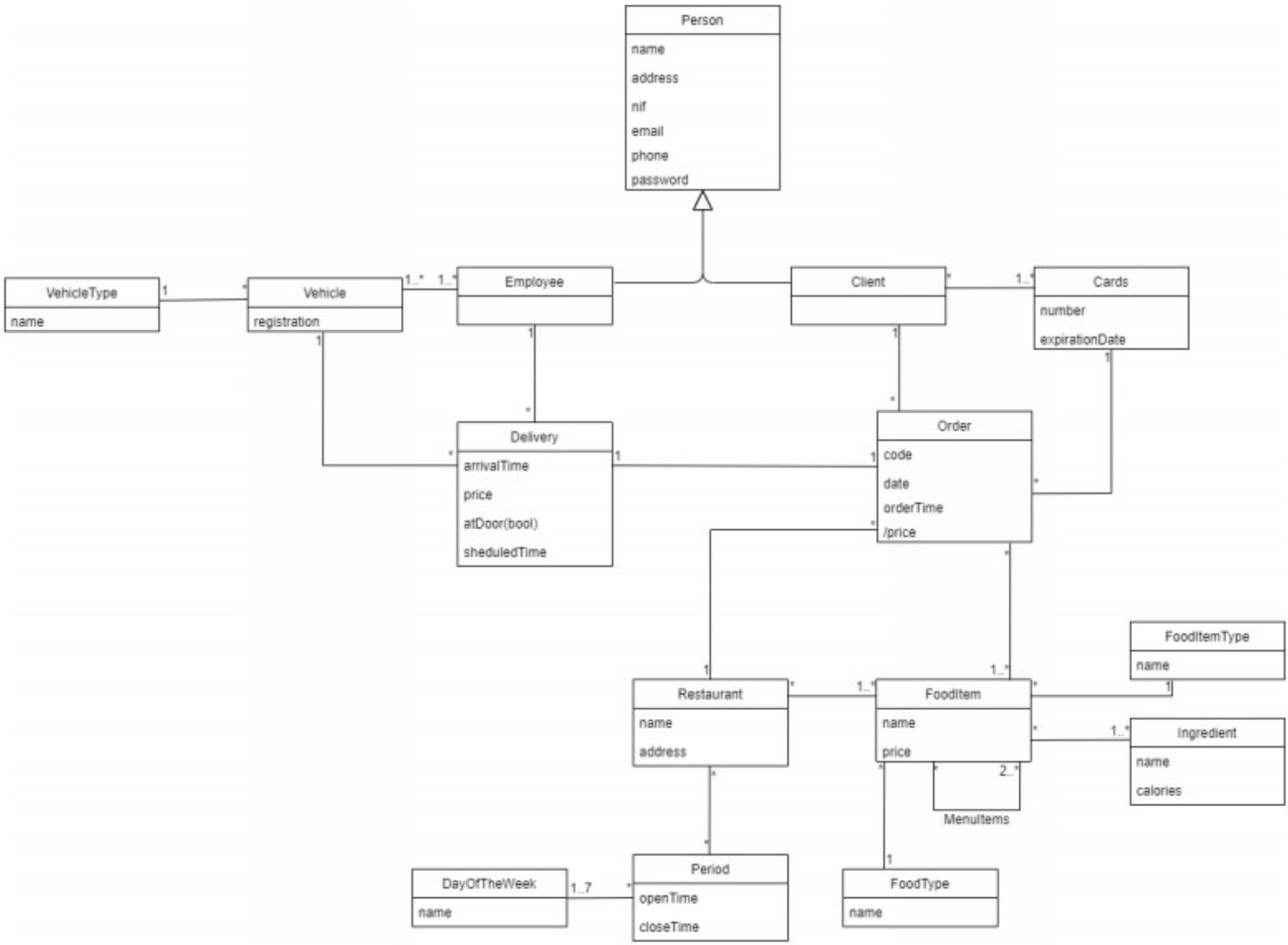
Da ementa é preciso saber quais os **FoodItem** disponíveis. Estes têm um nome, um **FoodItemType** (exemplos: bebida, prato, sobremesa, menu) , um **FoodType** (exemplos: italiano, português, americano), **Ingredient**s e preço. No caso de um FoodItem ser menu de FoodItemType este pode ser associado a 2 ou mais FoodItems que constituem o menu.

O **Request** de um Client para um Restaurant tem um identificador/código, quais os ítens a entregar e a quantidade de cada ítem, data, hora do pedido e preço total (calculado mediante o preço dos FoodItem pedidos). De forma a concluir o Request, é necessário especificar qual o Card usado.

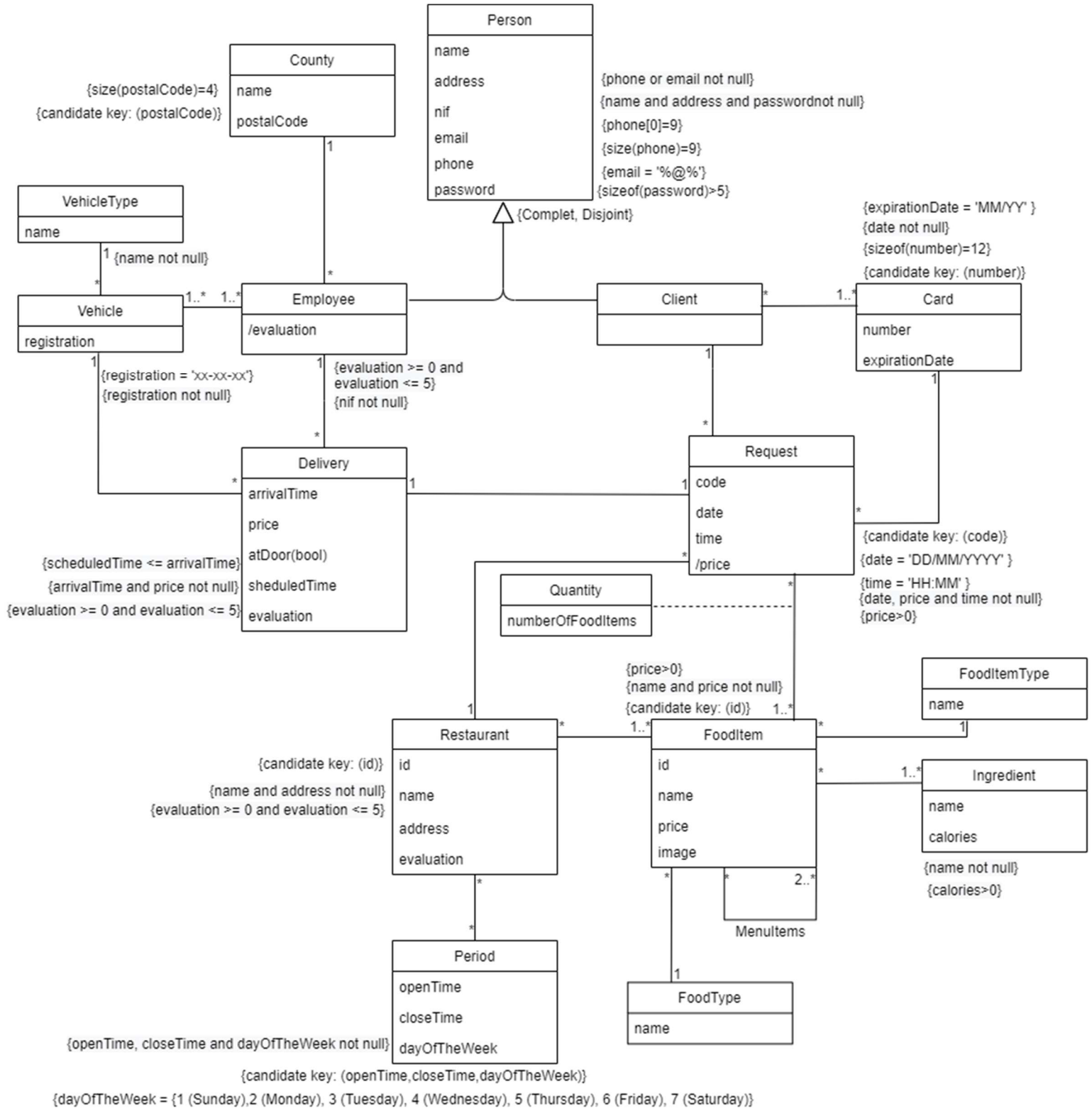
Uma **Delivery** tem associado um **Request**, o Employee responsável, o Vehicle utilizado, o custo de entrega, o modo de entrega, que apenas pode ser à mão ou à porta, a hora esperada para entrega e a hora em que efetivamente se deu a entrega.

No fim de cada Delivery o client pode avaliar a encomenda ficando assim associado um**a** evaluation.

Definição do Modelo Conceptual



Definição do Modelo Conceptual (Revisto)



Definição do Esquema Relacional

Client (id, name, address, nif, email, phone, password)

Card (number, expirationDate)

Request (code, date, time, price, clientID -> Client, cardNumber -> Card, restaurantID -> Restaurant)

Restaurant (id, name, address, evaluation)

Period (openTime, closeTime, dayOfTheWeek)

FoodItem (id, name, price, image, foodItemTypeID -> FoodItemType, foodTypeID -> FoodType)

FoodItemType (id, name)

Ingredient (id, name, calories)

FoodType (id, name)

Employee (id, name, address, nif, email, phone, password, evaluation, postalCode -> County)

Vehicle (id, registration, nameOfVehicleTypeID -> VehicleType)

VehicleType (id, name)

County (name, postalCode)

Delivery (arrivelTime, price, atDoor, sheduledTime, code -> Request, evaluation, vehicleID -> Vehicle, employeeID -> Employee)

Owns (clientID -> Client, cardNumber -> Card)

FoodItemIngredient (foodItemID -> FoodItem, ingredientID ->Ingredient)

Menu (foodItemID1 -> FoodItem, foodItemID2 -> FoodItem)

RestaurantPeriod (restaurantID -> Restaurant, openTime -> Period, closeTime -> Period, dayOfTheWeek -> Period)

RequestFoodItems (code -> Request, foodItemID -> FoodItem, numberOfFoodItems)

RestaurantFoodItem (restaurantID -> Restaurant, foodItemID -> FoodItem)

VehicleEmployee (vehicleID -> Vehicle, employeeID -> Employee)

Conversão do Modelo Conceptual para Modelo Relacional

Iremos, de seguida, mostrar os principais modos de conversão usados para o modelo relacional apresentado acima.

A **generalização** de Person em Client e Employee seguiu a estratégia **Object-Oriented**. Dado que esta era completa e disjunta este método revelou-se prático pois apenas tivemos que converter as duas classes derivadas nas duas relações correspondentes. Mais à frente explicaremos a implicação que esta teve na implementação de restrições.

Nas **associações muitos-para-muitos** criaram-se as últimas seis **relações** do modelo relacional que contêm uma **foreign key para cada lado da associação**. Sendo que no caso de RequestFoodItem, numberOfFoodItems, atributo da **classe de associação** Quantity, passou a **pertencer** a esta relação.

Na associação MenuItems, que é uma **self-associação**, o modelo descrito anteriormente foi igualmente o modelo escolhido. Foi criada a relação Menu que junta dois foodItems pelo menos.

No caso das **associações muitos-para-um** a estratégia passou por adicionar uma **chave estrangeira da classe para o lado de muitos** da relação. A relação request é um exemplo disso, contendo chaves estrangeiras para Client, Card e Restaurant, não sendo a única.

Apenas há um caso de uma **associação um-para-um**: entre Delivery e Request. Optamos por adicionar uma **chave estrangeira para Request em Delivery**. O atributo que partilham: **code** revela-se importante já que é a **primary key** para as duas relações, enfatizando que para cada Delivery temos uma Request.

Análise de Dependências Funcionais e Formas Normais

Nesta secção iremos analisar as dependências funcionais para cada relação do modelo relacional apresentado. Assim, para cada relação foram pensadas quais as dependências funcionais não triviais que existem e as possíveis chaves candidatas.
Para descobrir as violações à **Forma Normal de Boyce-Codd (BCNF)** analisaram-se todas as dependências funcionais não triviais. Para cada dependência funcional se o conjunto de atributos do lado esquerdo desta forem chaves então esta encontra-se na BCNF. As violações a esta forma normal são, consequentemente as dependências funcionais cujo lado esquerdo não possua uma das chaves indicadas.
A importância desta análise é devido às conclusões da estrutura e mapeamento que dela podem ser retiradas quanto a problemas de redundância no nosso modelo relacional.
Na sequência do estudo da BCNF e como complemento, encontramos também as violações à **Terceira Forma Normal (3NF)**. Assim uma relação está na 3NF se para cada dependência funcional não trivial os atributos do lado esquerdo da dependência são chaves candidatas ou os atributos do lado direito da mesma são atributos primos, isto é, são membros de uma chave.
Na maioria das relações que analisamos não existem quaisquer violações às duas Formais Normais mencionadas pelo que concluimos que conseguimos chegar a um baixo nível de redundância e uma melhor qualidade na organização dos dados de acordo com as regras estabelecidas por estes modelos. Permitindo assim com maior facilidade a deteção de anomalias e divisão dos dados por relações coerentes e fáceis de manusear, para além de deixar de necessitar de um grande e não desejado número de nulls nas tabelas.

Client (id, name, address, nif, email, phone, password)

Dependências Funcionais:

Id -> name, address, nif, email, phone, password
nif -> name, address, id, email, phone, password
phone -> name, address, nif, email, id, password
email -> name, address, nif, id, phone, password

Candidate Keys: {id}, {nif}, {phone}, {email}

Violação à BCNF: não tem
Violação à 3NF: não tem

Card (number, expirationDate)

Dependências Funcionais:

number -> expiration Date

Candidate Key: {number}

Violação à BCNF: não tem
Violação à 3NF: não tem

Request (code, date, time, price, clientID -> Client, cardNumber -> Card, restaurantID -> Restaurant)
asdadad
Dependências Funcionais:

code -> date, time, price, clientID, cardNumber, restaurantID
cardNumber, date, time, clientID -> price, code, restaurantID

Candidate Keys: {id}, {cardNumber, date, time, clientID}

Violação à BCNF: não tem
Violação à 3NF: não tem

Restaurant (id, name, address, evaluation)

Dependências Funcionais:

id -> name, address, evaluation
address -> id, name, address, evaluation

Candidate Keys: {id}, {address}

Violação à BCNF: não tem
Violação à 3NF: não tem

Period (openTime, closeTime, dayOfTheWeek)

Dependências Funcionais:

Não existem dependências funcionais não triviais.

Candidate Key: {openTime, closeTime, dayOfTheWeek}

Violação à BCNF: não tem
Violação à 3NF: não tem

FoodItem (id, name, price, image, foodItemTypeID -> FoodItemType, foodTypeID -> FoodType)

Dependências Funcionais:

id -> name, price, image, foodItemTypeID, foodTypeID
name, price, image -> foodItemTypeID, foodTypeID, id

Candidate Keys: {id}, {name, price, image}

Violação à BCNF: não têm
Violação à 3NF: não têm

FoodItemType (id, name)

Dependências Funcionais:

name -> id
id -> name

Candidate Keys: {id}, {name}

Violação à BCNF: não tem
Violação à 3NF: não tem

Ingredient (id, name, calories)

Dependências Funcionais:

name -> calories, id
id -> name, calories

Candidate Keys: {id}, {name}

Violação à BCNF: não tem
Violação à 3NF: não tem

FoodType (id, name)

Dependências Funcionais:

name -> id
id -> name

Candidate Keys: {id}, {name}

Violação à BCNF: não tem
Violação à 3NF: não tem

Employee (id, name, address, nif, email, phone, password, evaluation, postalCode -> County)

Dependências Funcionais:

id -> name, address, nif, email, phone, password, evaluation, postalCode
nif ->name, address, id, email, phone, password, evaluation, postalCode
phone -> name, address, nif, email, id, password, evaluation, postalCode
email -> name, address, nif, id, phone, password, evaluation, postalCode

Candidate Keys: {id}, {nif}, {phone}, {email}

Violação à BCNF: não tem
Violação à 3NF: não tem

Vehicle (id, registration, nameOfVehicleTypeID -> VehicleType)

Dependências Funcionais:

registration->nameOfVehicleTypeID, id
id -> registration, nameOfVehicleTypeID

Candidate Keys: {id}, {registration}

Violação à BCNF: não tem
Violação à 3NF: não tem

VehicleType (id, name)

Dependências Funcionais:

id->name
name -> id

Candidate Keys: {id}, {name}

Violação à BCNF: não tem
Violação à 3NF: não tem

County (name, postalCode)

Dependências Funcionais:

postalCode -> name

Candidate Keys: {postalCode}

Violação à BCNF: não tem
Violação à 3NF: não tem

Delivery (arrivelTime, price, atDoor, sheduledTime, code -> Request, evaluation, vehicleID -> Vehicle, employeeID -> Employee)

Dependências Funcionais:

code -> arrivelTime, price, atDoor, sheduledTime, evaluation, employeeID, registration

Candidate Keys: {code}

Violação à BCNF: não tem
Violação à 3NF: não tem

Nota: as relações criadas a partir de associações do modelo conceptual apenas contém atributos que são chaves estrangeiras e todos constituem a chave primária da nova relação. Assim, nenhuma das relações seguintes tem dependências funcionais não triviais. E por isso não há violações à Boyce-Code e Terceira Formas Normais. No entanto, na relação RequestFoodItem como temos uma classe de associação o atributo dessa foi incluído na nesta.

Owns (clientID -> Client, cardNumber -> Card)

Key: {clientID, cardNumber}

FoodItemIngredient (foodItemID -> FoodItem, ingredientID ->Ingredient)

Key: {foodItemID, ingredientID}

Menu (foodItemID1 -> FoodItem, foodItemID2 -> FoodItem)

Key: {foodItemID1, foodItemID2}

RestaurantPeriod (restaurantID -> Restaurant, openTime, closeTime, dayOfTheWeek -> Period)

Key: {restaurantID, openTime, closeTime, dayOfTheWeek}

RequestFoodItems (code -> Request, foodItemID -> FoodItem, numberOfFoodItems)

Dependências Funcionais:

code, foodItemID -> numberOfFoodItems

Key: {code, foodItemID}

Violação à BCNF: não tem
Violação à 3NF: não tem

RestaurantFoodItem (restaurantID -> Restaurant, foodItemID -> FoodItem)

Key: {restaurantID, foodItemID}

VehicleEmployee (vehicleID -> Vehicle, employeeID -> Employee)

Key: {vehicleID, employeeID}

Lista e Forma de Implementação de Restrições

De seguida vamos listar as restrições impostas para a nossa base de dados, analisando cada tabela uma a uma.

Nota: todas as chaves primárias (id e entre outros) das tabelas são únicas logo a restrição do unique não é necessária.

TABLE: Client
RESTRICTIONS:

- **nif**: é UNIQUE já que é um das candidate keys e tem apenas 9 numeros.

  ```
  CONSTRAINT nifLength CHECK (LENGTH (nif) == 9)
  ```

- **phone**: é UNIQUE já que é um das candidate keys e tem apenas 9 numeros sendo que o primeiro é 9 e por isso o modo BETWEEN revelou-se importante para a implementação.

  ```
  CONSTRAINT phoneLength CHECK (phone BETWEEN 900000000 AND 999999999)
  ```

- **email**: é UNIQUE já que é um das candidate keys. Um email tem que ter um '@'.

  ```
  CONSTRAINT emailPattern CHECK (email LIKE '%@%')
  ```

- **address**: é obrigatório ter address definida.

  ```
  CONSTRAINT addressNotNull CHECK (address NOT NULL)
  ```

- **name**: é obrigatório ter name definido.

  ```
  CONSTRAINT nameNotNull CHECK (name NOT NULL)
  ```

- **password**: é obrigatório ter password definida e esta tem que ter um tamanho mínimo de 6 caracteres.

  ```
  CONSTRAINT passRestrictions CHECK (password NOT NULL AND LENGTH
  (password)>5)
  ```

- **meio de contacto**: é obrigatório ter pelo menos um meio de contacto que tanto pode ser phone como email.

  ```
  CONSTRAINT contactNotAvailable CHECK (email NOT NULL OR phone NOT
  NULL)
  ```

Nota: a generalização completa e disjunta de Person para Client e Employee não permite que haja uma Person simultaneamente Client e Employee. Assim sendo, e tendo por base a conversão para o modelo relacional escolhida (Object-Oriented), a implementação desta restrição (ser impossível a criação de um Client com um nif, phone ou email iguais ao de um Employee existente e vice-versa) não pode ser feita sem o acesso e conhecimento de triggers. Consequentemente, implementaremos esta funcionalidade apenas na terceira parte do projeto.

TABLE: **Card**
RESTRICTIONS:

- **number**: é constituido por 12 algarismos.

```
CONSTRAINT cardNumberLength CHECK (LENGTH (number) == 12)
```

- **expirationDate**: é obrigatório ter expirationDate definida. A data vem no formato "MM/AA" como indica a segunda constraint definida.

```
CONSTRAINT dateNotNull CHECK (expirationDate NOT NULL),
CONSTRAINT datePattern CHECK (expirationDate LIKE '__/__')
```

TABLE: **Restaurant**
RESTRICTIONS:

- **name**: é obrigatório ter name definido.

```
CONSTRAINT nameNotNull CHECK (name NOT NULL)
```

- **address**: é obrigatório ter address definido, sendo este UNIQUE pois é impossível a existência de dois ou mais restaurantes na mesma address.

```
CONSTRAINT addressNotNull CHECK (address NOT NULL)
```

- **evaluation**: assume um valor default de 0.0 se não for introduzido e só pode assumir valores entre 0.0 e 5.0.

```
CONSTRAINT evaluationValues CHECK (evaluation BETWEEN 0.0 AND 5.0)
```

TABLE: **Period**
RESTRICTIONS:

- **dayOfTheWeek**: pode assumir valores entre 1 e 7 sendo estes dias da semana, 1 equivale a Domingo e 7 a Sábado, sendo um dos constituintes da primary key não pode ser NULL.

```
CONSTRAINT dayWeekRange CHECK (dayOfTheWeek BETWEEN 1 AND 7),
CONSTRAINT dayOfTheWeekNotNull CHECK (dayOfTheWeek NOT NULL)
```

- **time**: é obrigatório o openTime ser menor que o closeTime (têm que abrir antes de fechar) e como ambos são membros da primary key não podem tommar valores nulos.

```
CONSTRAINT validTimeWindow CHECK (openTime < closeTime),
CONSTRAINT openTimeNotNull CHECK (openTime NOT NULL),
CONSTRAINT closeTimeNotNull CHECK (closeTime NOT NULL)
```

TABLE: **FoodItemType**
RESTRICTIONS:

- **name**: o atributo name não pode assumir o valor nulo. No caso de se criar uma instância sem name especificado ele assume o valor default 'Not Specified'. Dado que é UNIQUE todos os tipos que não tiverem nomes terão que usar este túpulo da tabela.

  ```
  name TEXT UNIQUE DEFAULT 'Not Specified'
  ```

TABLE: **Ingredient**
RESTRICTIONS:

- **name**: é obrigatório ter name definido e este é UNIQUE dado que não há dois ingredientes diferentes com o mesmo nome.

  ```
  CONSTRAINT nameNotNull CHECK (name NOT NULL)
  ```

- **calories**: tem que ser positivo.

  ```
  CONSTRAINT calorieRange CHECK (calories > 0.0)
  ```

TABLE: **FoodType**
RESTRICTIONS:

- **name**: o atributo name não pode assumir o valor nulo. No caso de se criar uma instância sem name especificado ele assume o valor default 'Not Specified'. Dado que é UNIQUE todos os tipos que não tiverem name terão que usar este túpulo da tabela.

  ```
  name TEXT UNIQUE DEFAULT 'Not Specified'
  ```

TABLE: **VehicleType**
RESTRICTIONS:

- **name**: é obrigatório ter name definido. Este é UNIQUE pois não há dois types iguais com o mesmo nome.

  ```
  CONSTRAINT nameNotNull CHECK (name NOT NULL)
  ```

TABLE: **County**
RESTRICTIONS:

- **postalCode**: são apenas considerados os primeiros 4 números do código postal já que são estes que definem o concelho. Sendo este a primary key de County na implementação feita para sqlite foi necessário referir que a tabela não tem rowid (WITHOUT ROWID) de forma a que o atributo não fosse incrementado indevidamente quando inserido na tabela um novo túpulo sem valor de postalCode definido.

  ```
  CONSTRAINT postalCodeLenght CHECK (LENGTH (postalCode) == 4)
  ```

TABLE: **FoodItem**
RESTRICTIONS:

- **name**: é obrigatório ter name definido.

  ```
  CONSTRAINT nameNotNull CHECK (name NOT NULL)
  ```

- **image**: tem que vir no formato ".png" ou ".jpeg" e vem em default "blank.png".

  ```
  CONSTRAINT imagePattern CHECK (image LIKE '%.png' OR image LIKE
  '%.jpeg')
  ```

- **price**: é obrigatório ter price definido e este tem que ser positivo.

  ```
  CONSTRAINT priceNotNull CHECK (price NOT NULL),
  CONSTRAINT priceRange CHECK (price > 0.0)
  ```

- **foodItemTypeID**: é obrigatório ter foodItemType definido já que é uma informação relevante para a organização de foodItems. Para além disso, caso um type seja apagado, os foodItem derivados deixam de existir (Cascade) e quando o atributo é updated os valores têm de ser alterados em conformidade (Cascade).

  ```
  CONSTRAINT foodItemTypeFK FOREIGN KEY (foodItemTypeID) REFERENCES
  FoodItemType(id) ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT foodItemTypeNotNull CHECK (foodItemTypeID NOT NULL)
  ```

- **foodTypeID**: é obrigatório ter foodType definido já que é uma informação relevante para a organização de foodItems. Para além disso, caso um type seja apagado, os foodItem derivados deixam de existir (Cascade) e quando o atributo é updated os valores têm de ser alterado em conformidade (Cascade).

  ```
  CONSTRAINT foodTypeFK FOREIGN KEY (foodTypeID) REFERENCES
  FoodType(id) ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT foodTypeNotNull CHECK (foodTypeID NOT NULL)
  ```

TABLE: **Vehicle**
RESTRICTIONS:

- **registration**: é obrigatório ter registration definido, no formato "XX-XX-XX", e este é único dado que não há dois vehicles com resgistrations diferentes.

  ```
  CONSTRAINT registrationNotNull CHECK (registration NOT NULL),
  CONSTRAINT registrationPattern CHECK (registration LIKE '__-__-__')
  ```

- **vehicleType**: é obrigatório ter vehicleType definido já que é uma informação relevante para a organização de vehicles. Para além disso, caso deixe de existir esse type na tabela correspondente tem que deixar de existir os vehicles correspondentes (Cascade). No caso de update, o type será updated da mesma forma nos vehicles que o contêm (Cascade).

  ```
  CONSTRAINT vehicleTypeFK FOREIGN KEY (vehicleTypeID) REFERENCES
  VehicleType(id) ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT vehicleTypeNotNull CHECK (vehicleTypeID NOT NULL)
  ```

TABLE: **Request**
RESTRICTIONS:

- **data**: é obrigatório ter data definida. A data vem no formato “DD/MM/AAAA” como indica a segunda constraint definida.

```
CONSTRAINT dateNotNull CHECK (data NOT NULL),
CONSTRAINT datePattern CHECK (data LIKE '__/__/____')
```

- **time**: é obrigatório ter time definida. O time vem no formato “HH:MM” como indica a segunda constraint definida.

```
CONSTRAINT timeNotNull CHECK (time NOT NULL),
CONSTRAINT datePattern CHECK (data LIKE '__:__')
```

- **price**: é obrigatório ter price definido e este tem que ser positivo.

```
CONSTRAINT priceNotNull CHECK (price NOT NULL),
CONSTRAINT priceRange CHECK (price > 0.0)
```

- **clientID**: é obrigatório ter clientID definido, sendo este uma foreign key ficou estabelecido que como não podemos perder a informação de todos os requests feitos se o client tentar sair da base de dados, é impedido de apagar (Restrict) e se der update deste atributo será atualizado da mesma forma (Cascade).

```
CONSTRAINT clientFK FOREIGN KEY (clientID) REFERENCES Client(id) ON
DELETE RESTRICT ON UPDATE CASCADE,
CONSTRAINT clientNotNull CHECK (clientID NOT NULL)
```

- **restaurantID**: é obrigatório ter restaurantID definido, sendo este uma foreign key ficou decidido que como não podemos perder a informação de todos os requests feitos se o restaurant tentar sair da base de dados, tendo feito requests, é impedido (Restrict) e se der update será também atualizado (Cascade).

```
CONSTRAINT restaurantFK FOREIGN KEY (restaurantID) REFERENCES
Restaurant(id) ON DELETE RESTRICT ON UPDATE CASCADE,
CONSTRAINT restaurantNotNull CHECK (restaurantID NOT NULL)
```

- **cardNumber**: cardNumber não pode assumir o valor nulo, sendo que consiste numa foreign key, ficou definido que como não podemos perder a informação de todos os requests feitos com este cartão se o cartão tentar sair da base de dados é impedido (Restrict) e se der update o mesmo método se aplica (Restrict) pois estas eram as informações na altura do pedido. A length do cartão mantêm-se como restrição.

```
CONSTRAINT cardFK FOREIGN KEY (cardNumber) REFERENCES Card(number)
ON DELETE RESTRICT ON UPDATE CASCADE,
CONSTRAINT cardNotNull CHECK (cardNumber NOT NULL),
CONSTRAINT cardNumberLength CHECK (LENGTH (cardNumber) == 12)
```

TABLE: **Employee**
RESTRICTIONS:

- **nif**: é UNIQUE já que é um das candidate keys e possui 9 algarismos.

  ```
  CONSTRAINT nifLength CHECK (LENGTH (nif) == 9)
  CONSTRAINT nifNotNull CHECK (nif NOT NULL)
  ```

- **phone**: é UNIQUE já que é um das candidate keys e tem apenas 9 números sendo que o primeiro é 9 e por isso o modo BETWEEN revelou-se importante para a implementação.

  ```
  CONSTRAINT phoneLength CHECK (phone BETWEEN 900000000 AND 999999999)
  ```

- **email**: é UNIQUE já que é um das candidate keys.

  ```
  CONSTRAINT emailPattern CHECK (email LIKE '%@%')
  ```

- **address**: é obrigatório ter address definido.

  ```
  CONSTRAINT addressNotNull CHECK (address NOT NULL)
  ```

- **name**: é obrigatório ter name definido.

  ```
  CONSTRAINT nameNotNull CHECK (name NOT NULL)
  ```

- **password**: é obrigatório ter password definida e esta tem que ter um tamanho mínimo de 6 caracteres.

  ```
  CONSTRAINT passRestrictions CHECK (password NOT NULL AND LENGTH
  (password)>5)
  ```

- **meio de contacto**: é obrigatório ter pelo menos um meio de contacto que tanto pode ser phone como email.

  ```
  CONSTRAINT contactNotAvailable CHECK (email NOT NULL OR phone NOT
  NULL)
  ```

- **evaluation**: assume um valor default de 0.0 se não for introduzido e só pode assumir valores entre 0.0 e 5.0.

  ```
  CONSTRAINT evaluationValues CHECK (evaluation BETWEEN 0.0 AND 5.0)
  ```

- **postalCode**: sendo uma foreign key se não for defenido o postalCode para a área de trabalho vai assumir o null, por sua vez se esta área deixar de estar disponível será colocado o valor null (Set Null). No caso de update, a zona será updated da mesma forma nos employees que a contêm (Cascade).

  ```
  CONSTRAINT countyFK FOREIGN KEY (postalCode) REFERENCES
  County(postalCode) ON DELETE SET NULL ON UPDATE CASCADE
  ```

TABLE: **Delivery**
RESTRICTIONS:

- **code**: é obrigatório ter code definido sendo este igual ao de Request e primary key das duas tabelas. Como não podemos perder a informação de todos os deliveries efetuados se tentarem apagar o request seremos impedidos (Restrict) e a tentativa de mudar informação de Requests já realizadas não é permitida (Restrict).

  ```
  CONSTRAINT requestFK FOREIGN KEY (code) REFERENCES Request(code) ON
  DELETE RESTRICT ON UPDATE RESTRICT
  ```

- **time**: é obrigatório o scheduledTime ser menor e de preferência igual ao arrivalTime sendo que o arrivalTime tem que ser obrigatoriamente definido.

  ```
  CONSTRAINT validTimeWindow CHECK (scheduledTime <= arrivalTime),
  CONSTRAINT arrivalTimeNotNull CHECK (arrivalTime NOT NULL)
  ```

- **price**: é obrigatório ter price definido e este tem que ser positivo.

  ```
  CONSTRAINT priceNotNull CHECK (price NOT NULL),
  CONSTRAINT priceRange CHECK (price > 0.0)
  ```

- **atDoor**: assume-se sempre se não for defenido em contrário que a delivery se dá à porta.

  ```
  atDoor BOOLEAN DEFAULT 1
  ```

- **evaluation**: assume um valor default de 3.0 se não for introduzido e só pode assumir valores entre 0.0 e 5.0.

  ```
  CONSTRAINT evaluationValues CHECK (evaluation BETWEEN 0.0 AND 5.0)
  ```

- **employeeID**: é obrigatório ter employeeID definido assim como o respetivo employee na tabela respetiva, sendo este uma foreign key ficou definido que como não podemos perder a informação de todos os deliveries feitos se o employee tentar sair da base de dados tendo feito deliveries é impedido (Restrict) aquando do update das suas informações estas vão se refletir também em Deliveris (Cascade).

  ```
  CONSTRAINT employeeFK FOREIGN KEY (employeeID) REFERENCES
  Employee(id) ON DELETE RESTRICT ON UPDATE CASCADE,
  CONSTRAINT employeeNotNull CHECK (employeeID NOT NULL)
  ```

- **registrationID**: é obrigatório ter registrationID definido assim como o respetivo vehicle na tabela respetiva, sendo este uma foreign key ficou definido que como não podemos perder a informação de todos os deliveries feitos se o vehicle tentar sair da base de dados tendo feito deliveries é impedido (Restrict) a tentativa de update de algum dos seus atributos não será possível, visto que foi assim que realizou esta delivery (Restrict).

  ```
  CONSTRAINT vehicleFK FOREIGN KEY (registrationID) REFERENCES
  Vehicle(id) ON DELETE RESTRICT ON UPDATE RESTRICT,
  CONSTRAINT registrationNotNull CHECK (registrationID NOT NULL)
  ```

TABLE: **Owns**
RESTRICTIONS:

- **clientID**: caso deixe de existir este client na tabela correspondente tem que deixar de existir a relação de owns com o card (Cascade) e quando for para dar update todas as informações têm que ser alteradas (Cascade).

  ```
  CONSTRAINT clientFK FOREIGN KEY (clientID) REFERENCES Client(id) ON DELETE CASCADE ON UPDATE CASCADE
  ```

- **cardNumber**: caso deixe de existir este card na tabela correspondente tem que deixar de existir a relação de owns com o client (Cascade) e quando for para dar update todas as informações têm que ser mudadas (Cascade).

  ```
  CONSTRAINT cardFK FOREIGN KEY (cardNumber) REFERENCES Card(number) ON DELETE CASCADE ON UPDATE CASCADE
  ```

TABLE: **FoodItemIngredient**
RESTRICTIONS:

- **foodItemID**: caso deixe de existir este foodItem, na tabela correspondente, tem que deixar de existir a relação de foodItem com o ingredient (Cascade) e quando for para dar update todas as informações têm que ser mudadas (Cascade).

  ```
  CONSTRAINT foodItemFK FOREIGN KEY (foodItemID) REFERENCES FoodItem(id) ON DELETE CASCADE ON UPDATE CASCADE
  ```

- **ingredientID**: caso deixe de existir este ingredient não podemos deixar que o foodItem relacionado o perca (Restrict) e quando for alterado todas as instâncias com o valor do ingredient o serão (Cascade).

  ```
  CONSTRAINT ingredientFK FOREIGN KEY (ingredientID) REFERENCES Ingredient(id) ON DELETE RESTRICT ON UPDATE CASCADE
  ```

TABLE: **Menu**
RESTRICTIONS:

- **foodItemID1**: caso deixe de existir este foodItem na tabela correspondente tem que deixar de existir a relação de foodItem com o outro foodItem pois este deixa de ser um Menu possível (Cascade) e quando for para dar update todas as informações têm que ser mudadas (Cascade).

  ```
  CONSTRAINT foodItemFK1 FOREIGN KEY (foodItemID1) REFERENCES FoodItem(id) ON DELETE CASCADE ON UPDATE CASCADE
  ```

- **foodItemID2**: caso deixe de existir este foodItem na tabela correspondente tem que deixar de existir a relação de foodItem com o outro foodItem pois este deixa de ser um Menu possível (Cascade) e quando for para dar update todas as informações têm que ser mudadas (Cascade).

  ```
  CONSTRAINT foodItemFK2 FOREIGN KEY (foodItemID2) REFERENCES FoodItem(id) ON DELETE CASCADE ON UPDATE CASCADE
  ```

TABLE: **RestaurantPeriod**
RESTRICTIONS:

- **restaurantID**: caso este restaurant seja apagado, deixa de estar ligado a este period (Cascade) e todas as informações são alteradas quando exigido (Cascade).

  ```
  CONSTRAINT restaurantFK FOREIGN KEY (restaurantID) REFERENCES
  Restaurant(id) ON DELETE CASCADE ON UPDATE CASCADE
  ```

- **period**: no caso deste period deixe de estar disponível os restaurants não fecham apenas por esta razão (Restrict) quando for exigido update esta permite (Cascade).

  ```
  CONSTRAINT periodFK FOREIGN KEY (openTime,closeTime,dayOfTheWeek)
  REFERENCES Period(openTime,closeTime,dayOfTheWeek) ON DELETE
  RESTRICT ON UPDATE CASCADE
  ```

TABLE: **RequestFoodItem**
RESTRICTIONS:

- **code**: não podemos permitir que uma request seja mudado se já foi efetuado daí esta relação não poder deixar de existir se se tentar apagar o request respetivo.

  ```
  CONSTRAINT requestFK FOREIGN KEY (code) REFERENCES Request(code) ON
  DELETE RESTRICT ON UPDATE RESTRICT
  ```

- **foodItemID**: caso deixe de existir este foodItem na tabela correspondente não podemos deixar que este request deixe de ter este foodItem já que já foi efetuado (Restrict) e quando for para dar update todas as informações têm que ser mudadas (Cascade).

  ```
  CONSTRAINT foodItemFK FOREIGN KEY (foodItemID) REFERENCES
  FoodItem(id) ON DELETE RESTRICT ON UPDATE CASCADE
  ```

- **numberOfFoodItems**: tem por default o valor 1 mas podemos ter 2 foodItems iguais na mesma request, nunca podendo ter valores menor que 0.

  ```
  CONSTRAINT foodItemQuantity CHECK (numberOfFoodItems > 0)
  ```

TABLE: **RestaurantFoodItem**
RESTRICTIONS:

- **restaurantID**: caso deixe de existir este restaurant esta relação deixa de existir (Cascade) assim como, qualquer informação que seja necessário alterar/modificar sobre o restaurant, pode ser premitida (Cascade).

  ```
  CONSTRAINT requestFK FOREIGN KEY (code) REFERENCES Request(code) ON
  DELETE CASCADE ON UPDATE CASCADE
  ```

- **foodItemID**: caso deixe de existir este foodItem na tabela correspondente não podemos deixar que este restaurant deixe de ter este foodItem se esta não for a sua vontade (Restrict) e quando for para dar update todas as informações têm que ser mudadas (Cascade).

  ```
  CONSTRAINT foodItemFK FOREIGN KEY (foodItemID) REFERENCES
  FoodItem(id) ON DELETE RESTRICT ON UPDATE CASCADE
  ```

TABLE: **VehicleEmployee**
RESTRICTIONS:

- **vehicleID**: caso deixe de existir este vehicle na tabela correspondente tem que deixar de existir a relação de vehicle com o employee (Cascade) e quando for para dar update todas as informações têm que ser mudadas (Cascade).

  ```
  CONSTRAINT vehicleFK FOREIGN KEY (vehicleID) REFERENCES Vehicle(id)
  ON DELETE CASCADE ON UPDATE CASCADE
  ```

- **employeeID**: caso deixe de existir este employee na tabela correspondente tem que deixar de existir a relação de employee com o vehicle embora este possa permanecer na sua tabela (Cascade) e quando for para dar update todas as informações têm que ser mudadas (Cascade).

  ```
  CONSTRAINT employeeFK FOREIGN KEY (employeeID) REFERENCES
  Employee(id) ON DELETE CASCADE ON UPDATE CASCADE
  ```

Interrogações à Base de Dados

NOTA: de modo a usar as funções existentes para trabalhar com datas e horas fomos obrigados a mudar a sua formatação das datas para YYYY-MM-DD e horas para HH:MM.

1. Indique todos os itens de comida e respetivo preço por restaurante, assim como o seu nome e localização.

2. Em que restaurantes posso comprar bebida depois das 23:30 ou antes das 6:00, sendo que apenas precisam de estar abertos uns minutos nesse intervalo. Sendo que os nossos Period são referentes a um dia da semana em específico apenas se teve de comparar os tempos de fecho com as 23:30 e o horário de abertura com as 6:00. E por fim verificar se este vendia bebidas.

3. Quais são as Delivery feitas pelos Employee que têm 3 ou mais tipos de veículos associados.

4. Top três de compradores da aplicação (em termos de percentagem de dinheiro gasto nos pedidos). Isto é, quais foram os três clientes que gastaram mais dinheiro na aplicação, compara-se assim a soma de todos os preços dos pedidos feitos por eles com a soma de todo os preços de todos os pedidos alguma vez feitos na aplicação.

5. Indique por avaliação das entregas (por exemplo de 3-3.9), o número total de avaliações feitas e qual o número total dessas entregas cujo atraso foi menor que meia hora. Isto permite avaliar se o atraso pode ou não ter efeito na avaliação de uma Delivery.

6. Indique quais os funcionários que só entregaram pedidos entre as 8:00 e o 12:00 e não em qualquer outro intervalo de tempo do dia.

7. Indique para cada hora (exemplo 9h-9:59h) o Food Type mais pedido e respetiva quantidade pedida nesse horário.

8. Indique qual o restaurante mais aprazível para cada cliente, isto é, que vende o tipo de FoodItem os clients mais compram e de um preço semelhante. Primeiramente são calculadas as médias de preços de FoodItems comprados por cada cliente e também as médias de preços dos FoodItems vendidos por cada Restaurant, sendo também calculado qual ou quais os FoodTypes mais comprados por cada cliente. Sendo que depois é escolhido o Restaurant que têm uma média de preços de FoodItems mais próxima da usual do client e que simultaneamente vende pelo menos um FoodItem do tipo dos que o cliente mais compra.

9. Indique qual o trabalhador que passou mais tempo sem realizar entregas, isto é, cujo intervalo de tempo entre entregas consecutivas foi maior.

10. Indique quais os clientes que fizeram encomendas em todos os restaurantes que vendem FoodItems de Comida Indiana.

NOTA: Inicialmente, o nosso objetivo era ter interrogações as mais úteis e pertinentes para um possível utilizador da base de dados. Contudo, obedecer a isto nas 10 interrogações provou-se difícil, tendo em conta que também era do nosso interesse diversificar a forma de implementação de cada uma e aumentar a sua complexidade, sendo que, a maioria das interrogações pertinentes para a nossa base de dados eram de fácil implementação (pouca complexidade). Assim, decidimos abdicar um pouco da pertinência das nossas interrogações de modo a compensar os restantes fatores.

De seguida apresentamos uma lista de operadores que usamos e em quantas interrogações aparecem:

OPERADORES:

- AGREGATION ||
- JOIN (INNER) |||||||||
- LEFT JOIN |
- ORDER BY |||
- DESC |
- GROUP BY |||||
- HAVING ||
- EXISTS / NOT |
- NOT IN / IN |||
- MAX ||
- MIN ||
- SUM |
- AVG |
- COUNT ||
- ABS |
- DISTINCT |||
- EXCEPT |
- INSERCT |
- BETWEEN |
- IF NULL |
- LIMIT |
- SUBQUERIES ||
- VIEWS |
- JULIANDAY ||
- STRFTIME ||
- USING |
- OR |

Gatilhos

1. Update Request Price

Um preço de um pedido é a soma dos preços de todos os FoodItem do Request. A associação dos FoodItem ao Request é feita na relação RequestFoodItem, assim, é ao inserir nesta tabela elementos que explicitamente acrescentamos FoodItems aos pedidos. Em cada momento em que se for acrescentar FoodItems a um Request, insere-se uma nova instância de um RequestFoodItem. Este trigger obriga que na inserção desta linha da tabela, seja simultaneamente alterado o valor do price do Request somando ao que já estava o valor do novo FoodItem pretendido, multiplicando pela quantidade desse FoodItem pedida.

2. Valid Scheduled Time

Para agendar um Request é necessário que o Restaurant esteja aberto na hora pedida, deste modo antes da inserção de uma Delivery tem que ser verificada se o scheduledTime cumpre os requisitos de ser numa hora pertencente a um Period associado ao Restaurante, isto é, ser um dos horários em que ele esteja aberto.

3. Employee Evaluation

Aquando da Inserção de uma nova Delivery, sempre associada a um Employee é necessário recalcular a avaliação que vem sido atribuida a este. Assim, é preciso somar todas as avaliações, incluindo a nova e dividir pelo número de Delivery feitas pelo Employee em causa. Faz-se, então update da informação relativa a um employee dando-se set de um novo valor para a sua evaluation.

Conclusão

A terceira parte do projeto foi certamente a que mais prática nos deu para o futuro e para um melhor entender desta unidade curricular. Sendo assim, e como relatório final que apresentamos aqui, esperamos ter conseguido atingir os objetivos do trabalho e da unidade curricular.