Documentation du projet Base de Données

Équipe 1 Groupe 2 MMIS : Hélène HASSAN, Brice PERES, Virgile HENRY, Paul COLINOT, Dimitri HUBANS, Ryan DELAYAT, Zoé DEPREZ

Remarque générale importante :

Compte tenu de la situation dans laquelle nous avons rendu le projet (coupure des serveurs et du réseau à l'école), nous n'avons pas eu le temps et les ressources nécessaires pour finir la partie programmation et la tester.
Toutefois, nous avons tâché d'expliquer au maximum toutes les fonctionnalités et transactions que nous avions l'intention d'implémenter dans la partie IV.
Merci pour votre indulgence et bonne correction !

I. Analyse du sujet

Dans cette partie, on tâchera de suivre la même procédure qu'en TD. On prend chaque paragraphe (ou "partie importante") du sujet pour lequel on précisera : les dépendances fonctionnelles, les contraintes de valeurs, les contraintes de multiplicité et les contraintes textuelles.

Première partie : Restaurant, Plat, CategorieCuisine.

Dépendances Fonctionnelles	Contraintes de valeur	Contraintes de multiplicité
emailRestaurant → nomRestaurant, numRestaurant, nbPlacesAssises, typeCommandeRestaurant, textePresentation, horaireOuverture, jourOuverture, horaireFermeture, sommeNotes, cardinalNotes	$0 \leqslant sommeNotes / cardinalNotes \leqslant 5$ $nbPlacesAssises > 0$ $prix > 0$	Un restaurant peut avoir une note ou non. emailRestaurant -|→ noteRestaurant

numRestaurant → emailRestaurant emailRestaurant, horaireOuverture, jourOuverture → horaireFermeture emailRestaurant, idPlat → nomPlat, description, prix	(horaireOuverture, horaireFermeture) $\in [[12, 14]]^2 \cup [[19, 23]]^2$ jour $\in$ {Lundi, Mardi, ..., Dimanche} typeCommandeRestaurant $\subset$ {surPlace, livraison, aEmporter} typeCommandeRestaurant $\neq \emptyset$	Un plat peut contenir plusieurs allergènes ou non. idPlat -|->> nomAllergène Les restaurants peuvent se partager des horaires communs. horaireOuverture, horaireFermeture, jourOuverture ->> emailRestaurant Un restaurant peut avoir plusieurs horaires. emailRestaurant ->> horaireOuverture, horaireFermeture, jourOuverture

Contraintes textuelles :
La note d'un restaurant correspond à la moyenne de ses évaluations.
Un client peut parcourir la liste des restaurants en les triant selon leur note moyenne.

***Remarque** : nous avons décidé d'ajouter dès le début deux attributs SommeNotes et CardinalNotes (qui servent en quelque sorte de 'Cache') afin de ne pas avoir à les recalculer à chaque fois lorsqu'on souhaite trier les restaurants par catégories de cuisines recommandées. Cela peut poser des soucis de coût puisqu'on devrait en théorie mettre à jour ces deux attributs lors d'un ajout d'une évaluation pour garder une cohérence dans les données d'un restaurant mais en principe on ne le ferait que toutes les T périodes de temps pour que cela reste moins coûteux.*

Deuxième partie : Catégories de cuisine

Contraintes textuelles :
Une catégorie fille a une unique catégorie mère mais une catégorie mère peut avoir plusieurs catégories filles.

Un client peut parcourir la liste des restaurants en se basant sur l'organisation des catégories ou sur un ensemble de catégories recommandées en fonction de son historique de commandes.

Les recommandations des catégories se basent sur l'historique des commandes des utilisateurs ainsi que les notes qui leur sont reliées.

- En priorité, les recommandations concernent les catégories pour lesquelles l'utilisateur a donné la meilleure note moyenne.
- Deux catégories se trouvant à égalité sont départagées en regardant celles qui contiennent le plus de notes déposées quel que soit l'utilisateur.

Troisième partie : Comptes utilisateurs

Dépendances Fonctionnelles	Contraintes de multiplicité
emailClient → nomClient, prenomClient, adresseClient, motDePasse emailClient → idClient	Droit à l'oubli (un client peut vouloir supprimer ses données personnelles). idClient -|-> emailClient

Quatrième partie : Commandes

Dépendances Fonctionnelles	Contraintes de valeur	Contraintes de multiplicité
idCommande → dateCommande, heureCommande, idClient, emailRestaurant, contenu, prixFinal, statut	prixFinal > 0 statut ∈ {attenteConfirmation, validée,	Une livraison peut contenir optionnellement un texte pour le livreur.

idLivraison → adresseLivraison, heureLivraison idSurPlace → nbPersonnesSurPlace, heureArrivee idPlat, dateCommande, heureCommande → quantité (*association ternaire avec propriété propre 'Quantité')*	disponible, annuleeClient, annuleeRestaurant} nbPersonnesSurPlace > 0 {idLivraison} ∩ {idSurPlace} = ∅ {idLivraison} ∪ {idSurPlace} ⊆ {idCommande} Ext(idLivraison) ⊆ Ext(idCommande) Ext(idSurPlace) ⊆ Ext(idCommande) 0 ⩽ noteEvaluation ⩽ 5 quantité > 0 et quantité est un entier	idLivraison -|→ texteLivreur Un texte de livraison peut être associé à une ou plusieurs commandes. texteLivreur ->> idLivraison Une commande peut avoir une évaluation. idCommande -| → avis,noteEvaluation Une commande contient au moins un plat (avec sa quantité associée). idCommande ->> idPlat, quantité Un plat dans un restaurant peut n'être associé à aucune commande. idPlat -|->> idCommande

Contraintes textuelles :
Le nombre de places indiquées dans une commande sur place ne doit pas excéder le nombre de places disponibles dans le restaurant (c'est-à-dire : la somme des places des commandes sur place ne doit pas excéder le nombre de places assises du restaurant).

On suppose que l'heure de livraison est dans les horaires du restaurant :

heureLivraison ∈ [heureOuvertureRestaurant, heureFermetureRestaurant]

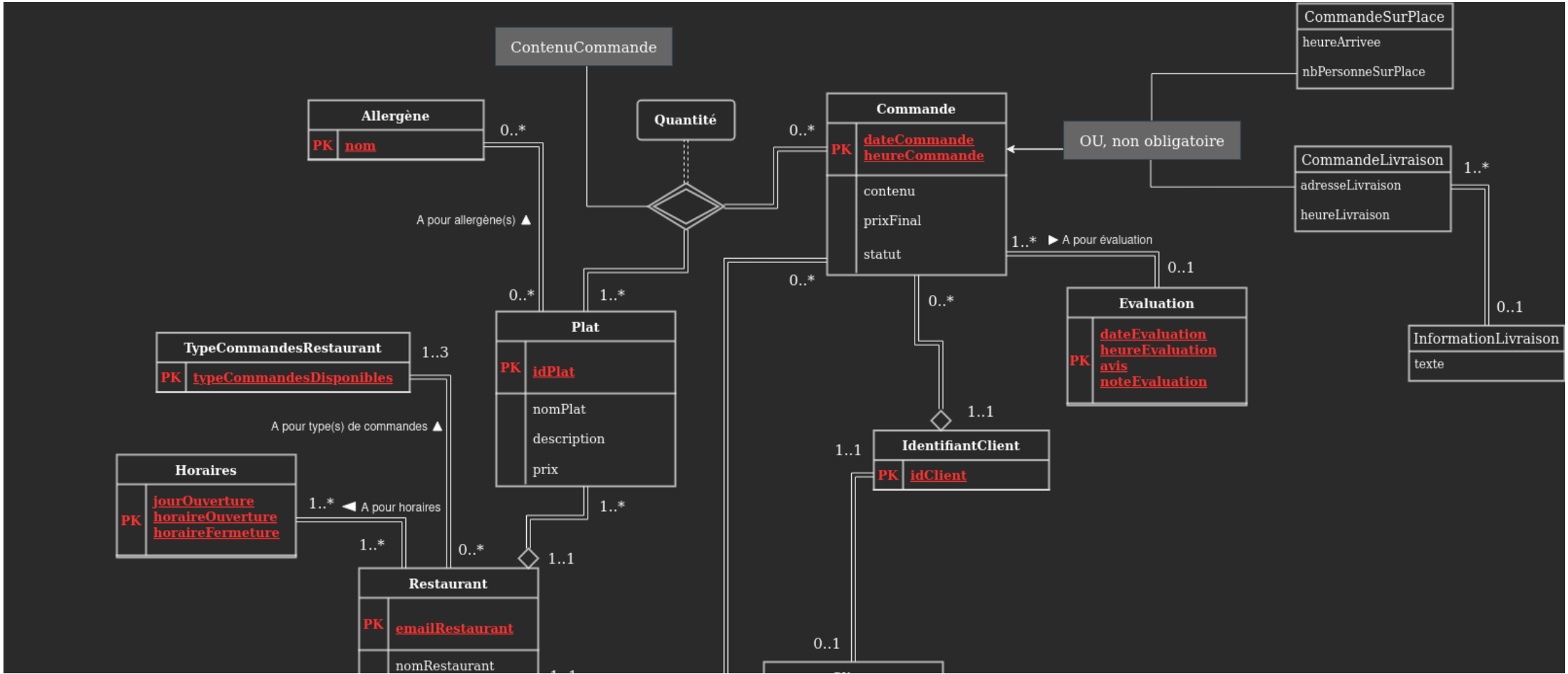
Le restaurant doit être ouvert à la date de la commande dateCommande.

Une évaluation d'une commande ne peut être faite qu'après la naissance de la commande.

dateEvaluation ≥ dateCommande ET heureEvaluation ≥ heureCommande

II. Schéma Entités/Associations déduit de l'analyse

Remarque : nous avons donné des noms aux associations qui amèneront la création d'une nouvelle relation dans la partie du passage au relationnel. Nous n'avons pas donné de nom aux associations contenant une cardinalité 1..1 ou les entités faibles pour plus de lisibilité (et aussi car aucune relation nouvelle ne résultera de ces associations-là).



Justification des choix difficiles

Nous avons réalisé plusieurs choix de conception pour notre schéma d'Entités/Associations. Nous allons tâcher de représenter les plus importants.

Plats :
Chaque plat a un identifiant unique pour chaque restaurant (par exemple, un plat qui s'appelle 'Choucroute garnie' aura deux identifiants différents pour deux restaurants différents) donc un plat a été modélisé par une entité faible d'un restaurant.

Allergènes :
Un plat est associé à aucun ou plusieurs allergènes (d'où la relation 0..*) et un allergène peut être associé à aucun ou plusieurs plats différents.

Catégories de cuisine :
Cette entité n'était pas facile à faire car les catégories de cuisine sont décrites de manière particulière sous la forme d'un arbre avec des sous-catégories de cuisine. Chaque catégorie 'Fille' possède une unique catégorie 'Mère' , par contre une catégorie 'Mère' peut avoir plusieurs catégories 'Fille'. Ceci nous a amenés à faire une unique entité 'Catégorie' avec un attribut nom (seul attribut caractérisant une catégorie) possédant une association vers elle-même. Nous avons mis 0..1 au lieu de 1..1 pour le côté Catégories Mères car la catégorie 'racine' de l'arbre n'a pas de catégorie mère (seulement les catégories 'filles' de la catégorie racine n'ont qu'une seule catégorie mère).

Contenu commande :
Nous avons utilisé une structure ternaire avec une propriété propre 'Quantité' liée aux entités Plats et Commande : cette dernière permet, pour chaque contenu de commande, d'associer différents plats avec une quantité spécifique pour chacun d'eux (ce qui correspond en soi à une commande classique sur n'importe quelle application type UberEats, etc.). Au départ, nous n'avions pas le souvenir du concept de propriété propre donc nous avions créé une entité Quantité avec un attribut entier (la quantité) en lien (via une association ternaire) avec les entités Plats et Commandes ce qui n'était pas très pratique.

Client/idClient :

Nous avons choisi de séparer les identifiants des clients dans une entité à part idClient. Ceci permet de prendre en compte la RGPD car chaque client peut alors supprimer à tout moment ses informations personnelles stockées dans la table client. Cette suppression est ainsi faite sans pour autant perdre les informations relatives aux commandes que ce client a passé, notamment ses évaluations qui comptent dans la note attribuée à chaque restaurant auquel ce dernier a commandé.

Horaires :
Pour représenter les horaires nous avons choisi de définir une entité Horaires contenant une clé primaire jourOuverture, horaireOuverture et horaireFermeture. L'entité Horaires est liée à l'entité Restaurant par la cardinalité 1..* dans la mesure où un triplet (jourOuverture, horaireOuverture, horaireFermeture) (par exemple : (Jeudi, 12h, 14h)) peut correspondre à au moins un restaurant). Nous n'avons pas mis 0..* dans la mesure où une horaire n'a pas de raison d'exister sans restaurant. En mettant une cardinalité 1..* et en choisissant la clé primaire au dessus, nous pouvons faire en sorte de partager une horaire commune entre plusieurs restaurants, ce qui optimise la mémoire dans notre base de données.

Commandes :

Nous avons choisi de définir chaque commande comme une entité faible de Restaurants et de idClient car une commande est unique pour un client et un restaurant donné (en tant que client, on ne commande qu'une fois à une date donnée dans un restaurant). Cette conception fonctionne car une entité faible peut être associée à plusieurs entités fortes.

Pour les évaluations :
Nous avions à l'origine envisagé de représenter une évaluation comme une entité simple de Commandes. Une évaluation aurait été ainsi unique pour chaque commande. Cependant, dans le cas très particulier où deux clients émettent exactement la même évaluation (au même moment, avec le même avis et la même note) on aurait stocké 2 fois cette même évaluation. Envisager une évaluation avec une cardinalité 1..* vers Commandes permet donc de stocker moins de données en mémoire car deux commandes différentes peuvent se partager une évaluation identique. Même si le cas donné au dessus (deux utilisateurs qui laissent la même évaluation au même moment) est assez peu probable pour notre application, nous avons décidé de laisser une liaison simple plutôt qu'une liaison entre entité faible (Evaluation, avec une cardinalité 0..1) - entité forte (Commandes avec une cardinalité 1..1) car économiser le plus possible de mémoire est une démarche que l'on se doit de respecter le plus possible quand on le peut lorsqu'on modélise une base de données.

La clé primaire de l'entité Evaluation correspond à dateEvaluation, heureEvaluation, avis et noteEvaluation car ce sont ces attributs qui caractérisent une évaluation de manière unique.

Pour les types de Commandes :
Il existe 3 types de commandes : sur place, à emporter et par livraison. Seulement 2 de ces types de commandes ont des attributs spécifiques donc toutes les commandes à emporter correspondent à l'entité 'Commandes' par défaut et les commandes sur place et en livraison définissent chacune une sous-entité. D'où l'utilisation d'un "OU, non obligatoire" entre commandeSurPlace et commandeLivraison.

III. Passage au relationnel

(remarque importante : la clé primaire de chaque relation est mise en rouge)

Entités de base (non modifiées par les associations dans lesquelles elles sont impliquées) :

On crée une relation par entité de base spécifiée dans le schéma UML des entités et associations (sauf entités faibles et sous-entités). Les relations données ci-dessouscorrespondent à des entités qui n'ont pas été modifiées par le passage au relationnel.

Schéma	**Restaurant(**emailRestaurant, nomRestaurant, numRestaurant, textePresentation, nbPlacesAssises, sommeNotes, cardinalNotes,**)**
Clef(s)	clefs = {**{emailRestaurant}**, {numRestaurant}}
Contraintes	emailRestaurant de longueur <= 64 nomRestaurant de longueur <= 32 textePresentation de longueur <= 1024 nbPlacesAssise > 0 cardinalNotes*5 >= sommeNotes >= 0

Forme normale (et justification)	**3FNBCK** en effet, on a emailRestaurant → nomRestaurant, numRestauant, nbPlacesAssises, sommeNote, cardinalNote numRestaurant → emailRestaurant, nomRestaurant, nbPlacesAssises, sommeNote, cardinalNote Ainsi, tout attribut non clé est pleinement et directement dépendant de toutes les clefs
Depndance fonctionnelles	emailRestaurant ->nomRestaurant, numRestaurant, nbPlacesAssises, sommeNote, cardinalNote numRestaurant -> emailRestaurant

Schéma	**CatégorieCuisine**(nomCatégorie)
Clef(s)	**{nomCategorie}**
Contraintes	nomCatégorie de longueur <= 32
Forme normale (et justification)	**3FN** Il n'y a qu'un attribut qui est une clé, donc tout attribut non clé est pleinement et directement dépendant de toutes les clefs (vu qu'il n'y en a pas)
Dependance fonctionnelles	/

Schéma	**TypeCommandesRestaurant**(typeCommandesDisponibles)
Clef(s)	**{typeCommandesDisponibles}**
Contraintes	typeCommandeDisponibles in {aEmporter, surPlace, livraison}

Forme normale (et justification)	**3FNBCK** Il n'y a qu'un attribut qui est une clé, donc tout attribut non clé est pleinement et directement dépendant de toutes les clefs (vu qu'il n'y en a pas)
Dépendances fonctionnelles	/

Schéma	**Allergène**(nomAllergène)
Clef(s)	**{nomAllergène}**
Contraintes	nomAllergène de longueur <= 64
Forme normale (et justification)	**3FNBCK** Il n'y a qu'un attribut qui est une clé, donc tout attribut non clé est pleinement et directement dépendant de toutes les clefs (vu qu'il n'y en a pas)
Dépendances fonctionnelles	/

Schéma	**Horaires**(heureOuverture, jourOuverture, heureFermeture)
Clef(s)	**{heureOuverture, jourOuverture, heureFermeture}**
Contraintes	jourOuverture in {Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche} 0 <= heureOuverture, heureFermeture < 24

Forme normale (et justification)	**3FNBCK** Les 3 attributs font partie de la clé et il n'y a pas de dépendance fonctionnelle
Dépendances fonctionnelles	/

Schéma	**IdentifiantClient**(idClient)
Clef(s)	**{idClient}**
Contraintes	/
Forme normale (et justification)	**3FNBCK** Il n'y a qu'un attribut qui est une clé, donc tout attribut non clé est pleinement et directement dépendant de toutes les clefs (vu qu'il n'y en a pas)
Dépendances fonctionnelles	/

Schéma	**Evaluations**(dateEvaluation, heureEvaluation, avis, noteEval)
Clef(s)	**{dateEvaluation, heureEvaluation, avis, noteEval}**
Clé(s) étrangère(s)	emailRestaurant référence Restaurant.emailRestaurant idClient référence IdentifiantClient.idClient dateCommande référence Commande.dateCommande heureCommande référence Commande.heureCommande

Contraintes	dateEvaluation est une date 0 <= heureEvaluation < 24 0 <= noteEval <= 5 Longueur avis <= 1024
Forme normale (et justification)	**3FNBCK** Tous les attributs font parti de la clé et il n'y a pas de dépendance fonctionnelle
Dépendance fonctionnelles	/

Entités faibles :

Schéma	**Plat**(emailRestaurant, idPlat, nomPlat, descriptionPlat, prixPlat)
Clef(s)	{**{emailRestaurant, idPlat}**, {emailRestaurant, nomPlat}}
Clé(s) étrangère(s)	emailRestaurant reference Restaurant.emailRestaurant
Contraintes	prixPlat >0 Longueur nomPlat <= 64 Longueur descriptionPlat <= 128
Forme normale (et justification)	**3FNBCK** Les seuls dépendance fonctionnelles sont issues des clés

Dépendance fonctionnelles	emailRestaurant, nomPlat -> idPlat, descriptionPlat, prixPlat emailRestaurant, idPlat -> nomPlat, descriptionPlat, prixPlat

Schéma	**Commandes**(emailRestaurant, idClient, dateCommande, heureCommande, prixFinal, statut)
Clef(s)	**{emailRestaurant, idClient, dateCommande, heureCommande}**
Clé(s) étrangère(s)	emailRestaurant référence Restaurant.emailRestaurant idClient référence IdentifiantsClient.idClient
Contraintes	statut in ('AttenteConfirmation, Validee, Disponible, EnLivraison, AnnuleeParClient, AnnuleeParRestaurant') heureCommande est dans horaires du Restaurant
Forme normale (et justification)	**3FNBCK** Les seuls dépendance fonctionnelles sont issus de la clé
Dépendance fonctionnelles	emailRestaurant, idClient, dateCommande, heureCommande -> prixFinal, statut

---

Sous-Entités :

La justification du sous-type de commandes utilisé est donnée lors de la soutenance.

Schéma	**CommandesLivraison**(emailRestaurant, idClient, dateCommande, heureCommande, adresseLivraison, heureLivraison)
Clef(s)	**{emailRestaurant, idClient, dateCommande, heureCommande}**
Clé(s) étrangère(s)	emailRestaurant référence Restaurant.emailRestaurant idClient référence IdentifiantClient.idClient heureCommande est dans horaires du Restaurant
Contraintes	dateCommande est une date 0 <= heureLivraison < 24 heureCommande est dans horaires du Restaurant
Forme normale (et justification)	**3FNBCK** Les seuls dépendance fonctionnelles sont issus de la clé
	emailRestaurant, idClient, dateCommande, heureCommande -> adresseLivraison, heureLivraison

Schéma	**CommandesSurPlace**(emailRestaurant, idClient, dateCommande, heureCommande, nbPersonnesSurPlace, heureArrivee)
Clef(s)	**{emailRestaurant, idClient, dateCommande, heureCommande}**
Clé(s) étrangère(s)	emailRestaurant référence Restaurant.emailRestaurant idClient référence IdentifiantClient.idClient heureCommande est dans horaires du Restaurant
Contraintes	dateCommande est une date heureCommande, heureArrivee est dans horaires du Restaurant
Forme	**3FNBCK**

normale (et justification)	Les seuls dépendance fonctionnelles sont issues de la clé
Dépendance fonctionnelle	emailRestaurant, idClient, dateCommande, heureCommande -> nbPersonnesSurPlace, heureArrivée

---

Entités (non faibles) avec une association binaire possédant une cardinalité 1..1 :

Schéma	**Clients**(idClient, emailClient, mdp, nomClient, prenomClient, adresseClient)
Clef(s)	{{idClient}, **{emailClient}**}
Clé(s) étrangère(s)	idClient référence IdentifiantClient.idClient
Contraintes	nomClient, prenomClient de longueur <= 32 emailRestaurant de longueur <= 64 mdp de longueur <= 64 adresseClient de longueur <= 128
Forme normale (et justification)	**3FNBCK** Car les seuls dépendances fonctionnelles sont issus des clés
Dépendances fonctionnelles	idClient -> emailClient, mdp, nomClient, prenomClient, adresseClient emailClient ->idClient, mdp, nomClient, prenomClient, adresseClient

Schéma	**InformationsLivraison**(emailRestaurant, idClient, dateCommande, heureCommande, texteLivraison)

Clef(s)	**{emailRestaurant, idClient, dateCommande, heureCommande, texteLivraison}**
Clé(s) étrangère(s)	emailRestaurant reference Restaurant.emailRestaurant idClient reference IdentifiantsClient.idClient
Contraintes	dateCommande est une date 0 <= heureCommande< 24 Longueur texteLivraison < 64
Forme normale (et justification)	**3FNBCK** Il n'y as pas d'autre DF que celle issue de la clé
Dépendance fonctionnelles	emailRestaurant, idClient, dateCommande, heureCommande -> texteLivraison

---

Nouvelles relations issues d'associations binaires entre entités uniquement avec des 0..1 ou x..* :

Schéma	**EstCatégorieFilleDe**(nomFille, nomMère)
Clef(s)	**{nomFille, nomMère}**
Clé(s) étrangère(s)	nomFille référence CatégorieCuisine.nomCatégorie nomMère référence CatégorieCuisine.nomCatégorie
Contraintes	/
Forme	**3FNBCK**

normale (et justification)	Les seuls dépendance fonctionnelles sont issues de la clé
Dépendance fonctionnelles	/

Schéma	**APourAllergène**(idPlat, nomAllergène, emailRestaurant)
Clef(s)	**{idPlat, nomAllergène, emailRestaurant}**
Clé(s) étrangère(s)	idPlat référence Plat.idPlat emailRestaurant reference Restaurants.emailRestaurant nomAllergène référence Allergène.nomAllergène
Contraintes	/
Forme normale (et justification)	**3FNBCK** Les seuls dépendance fonctionnelles sont issues de la clé
Dépendance fonctionnelles	/

Schéma	**APourTypeCommande**(emailRestaurant, typeCommandeDisponible)
Clef(s)	**{emailRestaurant, typeCommandeDisponible}**
Clé(s) étrangère(s)	emailRestaurant référence Restaurant.emailRestaurant typeCommandeDisponible référence CommandesRestaurant.typeCommandesDisponibles

Contraintes	/
Forme normale (et justification)	**3FNBCK** Les seuls dépendance fonctionnelles sont issues de la clé
Dépendance fonctionnelles	/

Schéma	**ContenuCommande**(emailRestaurant, idPlat,idClient, dateCommande, heureCommande, quantite)
Clef(s)	**{emailRestaurant, idClient, dateCommande, heureCommande, idPlat}**
Clé(s) étrangère(s)	emailRestaurant référence Restaurant.emailRestaurant idClient référence IdentifiantClient.idClient dateCommande référence Commande.dateCommande heureCommande référence Commande.heureCommande idPlat référence Plat.idPlat
Contraintes	quantité > 0 et quantité est un entier
Forme normale (et justification)	**3FNBCK** Les seuls dépendance fonctionnelles sont issues de la clé
Dépendance fonctionnelles	emailRestaurant, idPlat, idClient, dateCommande, heureCommande -> quantite

Schéma	**APourCategorie**(emailRestaurant, nomCategorie)
Clef(s)	**{emailRestaurant, nomCategorie}**
Clé(s) étrangère(s)	emailRestant référence Restaurant.emailRestaurant nomCatégorie référence CategorieCuisine.nomCatégorie
Contraintes	/
Forme normale (et justification)	**3FNBCK** Les seuls dépendance fonctionnelles sont issues de la clé
Dépendance fonctionnelles	/

Schéma	**APourHoraire**(emailRestaurant, horaireOuverture, jourOuverture, heureFermeture)
Clef(s)	**{emailRestaurant, jourOuverture, heureOuverture, heureFermeture}**
Clé(s) étrangère(s)	emailRestaurant référence Restaurant.emailRestaurant horaireOuverture référence Horaires.horaireOuverture jourOuverture référence Horaires.jourOuverture heureFermeture référence Horaires.heureFermeture
Contraintes	/
Forme normale (et justification)	**3FNBCK** Les seuls dépendance fonctionnelles sont issues de la clé

Dépendances fonctionnelles	emailRestaurant, horaireOuverture, jourOuverture -> heureFermeture emailRestaurant, horaireOuverture, heureFemeture -> heureOuverture

Schéma	**APourEvaluation**(emailRestaurant, idClient, dateCommande, heureCommande, dateEvaluation, heureEvaluation, avis, note)
Clef(s)	**{emailRestaurant, idClient, dateCommande, heureCommande, dateEvaluation, heureEvaluation, avis, note}**
Clé(s) étrangère(s)	emailRestaurant référence Restaurant.emailRestaurant idClient référence identifiantClient.idClient dateEvaluation, heureEvaluation, avis, note références Evaluations
Contraintes	/
Forme normale	**3FNBCK** Les seuls dépendance fonctionnelles sont issues de la clé
Dépendances fonctionnelles	emailRestaurant, horaireOuverture, jourOuverture -> heureFermeture emailRestaurant, horaireOuverture, heureFemeture -> heureOuverture

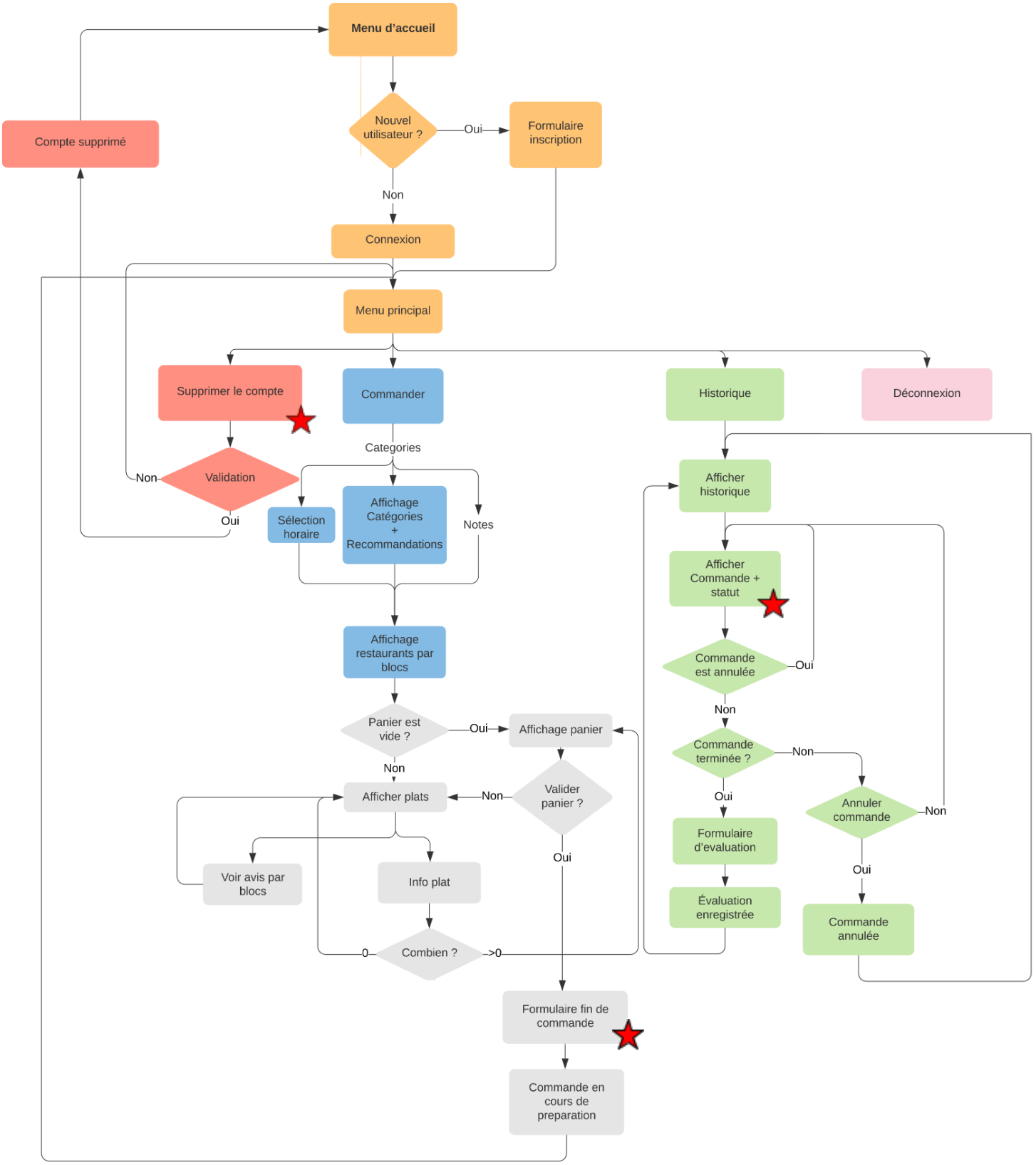
Contraintes non implémentables en relationnel

- somme des nbPersonnesSurPlace de emailRestaurant < nbPlacesDisponibles de emailRestaurant
- prixFinal est la somme des prixPlat * quantitePlat de APourContenu
- heureCommande est dans les horaires d'ouverture du restaurant pour Commande et ses filles

- Un client peut parcourir la liste des restaurants en se basant sur l'organisation des catégories ou sur un ensemble de catégories recommandées en fonction de son historique de commandes.
- un restaurant contient au moins 1 plat : ceci signifie que lors de la création de la base de données, il ne faut pas commit une insertion de restaurant sans avoir inséré de plats dans la même transaction (le client ne doit pas voir qu'il y a un restaurant sans plat dans la base de donnée donc cette transaction s'effectue avec une isolation de type *serializable*).

- Les recommandations des catégories se basent sur l'historique des commandes des utilisateurs ainsi que les notes qui leur sont reliées.
    - En priorité, les recommandations concernent les catégories pour lesquelles l'utilisateur a donné la meilleure note moyenne.
    - Deux catégories se trouvant à égalité sont départagées en regardant celles qui contiennent le plus de notes déposées quel que soit l'utilisateur.

IV. Analyse des fonctionnalités et implémentation des requêtes SQL2



Nous avons réalisé un automate pour le fonctionnement de notre application dont le schéma est donné ci-dessus.

Description de l'automate

Les grandes sections de l'application ont une couleur spécifique. (ex : orange → menus du début jusqu'au menu principal, vert → menus liés à l'historique des commandes...). Chaque carré correspond généralement à un menu spécifique, et les losanges correspondent à des questions en fin de menu où l'utilisateur doit faire un choix.
Réaliser un automate pour organiser notre application était une assez bonne idée dans la mesure où cela nous a permis de définir un cahier des charges des différentes fonctionnalités de notre application. Ce cahier des charges correspond en quelque sorte à ce qu'un client ayant élaboré le sujet aurait attendu de notre application.
D'un point de vue 'développeur', ce type de schéma permet de déterminer quelles requêtes SQL2 sont nécessaires et comment les mettre en relation avec l'interface pour réaliser les fonctionnalités voulues par le client sur son application.

Explications rapides sur l'interface

Comme spécifié en début de compte rendu, il se peut que des parties n'aient pas été implémentées.
Nous avons fait des choix au niveau de l'implémentation de l'interface qui facilitent certains aspects liés aux bases de données (comme par exemple, les transactions, qui seront détaillées en fin de partie).

Dans le package *app* se trouvent toutes les classes nécessaires à l'affichage des menus sur la console :

- Les blocs de même couleur correspondent à un menu à part entière (c'est-à-dire une sous-classe de *Menu* qui est elle-même une classe abstraite : par exemple, *DisplayRestaurants* est une sous-classe de Menu est a pour but d'afficher les restaurants selon un certain ordre de tri, par exemple par recommandations).

- Une classe InfoSession permet de stocker les informations de la session de l'utilisateur. Par exemple, son email (donné lors de sa connexion) ainsi que les informations de commande si ce dernier veut passer une commande.

- La classe *Navigator* correspond à une classe abstraite statique permettant de gérer les menus de l'utilisateur. Cette classe est intéressante car elle met en place une structure de donnée de type FIFO pour les menus qui sont stockés les uns à la suite des autres. Cette structure est très pratique car quand un utilisateur veut retourner en arrière et annuler la transaction en cours, il suffit juste de dépiler la structure d'un menu (sans faire de commit pour la transaction en cours). De même, la classe implémente une méthode *rollback* qui permet d'exécuter une nouvelle fois le menu

courant dans lequel l'utilisateur se trouve. Toutes les classes présentes dans le package *data* sont liées aux requêtes SQL.

Les fonctionnalités que nous avons retenues pour notre application sont donc listées ci-après (pour les plus importantes).
Remarque : Les requêtes SQL ne reposent que sur la lecture de la base de données. Pour créer des requêtes valides ici, il suffit donc de s'assurer que la base de données est déjà dans un état cohérent avant de récupérer les informations.

1. Parcours des restaurants disponibles

a. Tri par catégories de cuisine

Comme indiqué dans le sujet, les catégories de cuisine sont stockées sous forme d'arbre avec une catégorie racine ("root") qui ne correspond à une catégorie de cuisine réelle (il s'agit juste d'une catégorie abstraite pour pouvoir définir une racine).

Toutes les requêtes sur les catégories de cuisine se font à l'aide de la relation EstCategorieFilleDe.
L'accès à une catégorie de cuisine à l'aide de son nom se fait via une requête SQL présente dans la méthode *get* de la classe Categorie :
(syntaxe avec les queries écrites en java)

```
ResultSet result = Table.sendQuery(String.format("SELECT * FROM %s WHERE nomFille = '%s'", tableName, name));
```

Il est également possible, à partir d'une catégorie de cuisine, d'obtenir toutes ses catégories filles. Cela se fait dans la méthode *getDirectChildren* :
(syntaxe avec les queries écrites en java)

```
ResultSet result = Table.sendQuery(String.format("SELECT nomFille FROM %s WHERE nomMere = '%s'", tableName, nom));
```

L'interêt est que l'on ne recupère uniquement les noms des catégories que l'on affiche à l'écran. On ne recupère ainsi pas l'ensemble de la table a chaque fois que l'on souhaite afficher une categorie.

Il est également nécessaire d'ordonner les catégories selon les notes de l'utilisateur :

```
SELECT nomCategorie
  FROM ( SELECT apc.nomCategorie, SUM(ape.note)/COUNT(*)
          FROM Restaurant r
          JOIN APourCategorie apc
           ON apc.emailRestaurant=r.emailRestaurant
          JOIN APourEvaluation ape
           ON ape.emailRestaurant=r.emailRestaurant
           WHERE apc.idClient='<idClient>'
          GROUP BY apc.nomCategorie
       ) client
  JOIN ( SELECT r.nomCategorie, SUM(r.sommeNotes/r.cardinalNotes)/COUNT(*)
          FROM Restaurant r
          JOIN APourCategorie apc
           ON apc.emailRestaurant=r.emailRestaurant
          GROUP BY nomCategorie
       ) average
  ON client.nomCategorie=average.nomCategorie
 ORDER BY client.note, average.note
```

b. Tri par horaires d'ouverture

Nous n'avons pas pu implémenter ce tri dans notre code mais la requête SQL2 correspondante est :

```
SELECT r.nomRestaurant, r.sommeNotes/r.cardinalNotes
FROM Restaurant r
JOIN APourHoraire aph ON aph.emailRestaurant = r.emailRestaurant
WHERE aph.jourOuverture = <jourSelectionne> AND aph.horaireOuverture <= <horaireSelectionne> AND
aph.horaireFermeture >= <horaireSelectionne>
```

2. Création/Suppression d'un compte client

a. Création de compte utilisateur (client)

Les requêtes liées à cette fonctionnalité sont implémentées dans la méthode *create* de la classe Clients.
Il faut bien veiller à ajouter l'identifiant du client (qui correspond à l'identifiant maximum de la table IdentifiantClient au moment de la requête incrémenté de 1 à chaque ajout de nouveau client) dans la relation IdentifiantClient en même temps d'ajouter les informations personnelles du client dans la relation Clients.

Insertion de l'identifiant du client dans la relation IdentifiantClient :
(syntaxe avec les queries écrites en java)

```
ResultSet result = Table.sendQuery(String.format("SELECT MAX(idClient) FROM %s", tableClients));
result.next();
int newId = result.getInt(1) + 1;
Table.sendQuery(String.format("INSERT INTO %s VALUES (%d)", tableIdClients, newId));
```

Insertion des données personnelles du client dans la relation Clients :
(syntaxe avec les queries écrites en java)

```
Table.sendQuery(String.format("INSERT INTO %s VALUES (%d, '%s', '%s', '%s', '%s')", tableClients, newId, mdp, nom,
prenom, adresse));
```

b. Suppression d'un compte client

Comme nous avions tout mis en oeuvre (dans la partie analyse et passage au schéma Entités/Associations avec la création de 2 entités IdentifiantClient et Clients) pour rendre facile la suppression des données personnelles d'un client, la prise en compte de la norme RGPD fut simple à faire. Cette suppression se fait dans la classe Clients dans la méthode *remove*.
Cette dernière consiste en une unique requête prenant en argument l'email du client (ce dernier permettant d'identifier un client de manière unique) que l'on note ici *email*:

```
DELETE FROM Clients WHERE emailClient='email'
```

(syntaxe avec les queries écrites en java)

```
Table.sendQuery(String.format("DELETE FROM %s WHERE emailClient='%s'", tableClients, email));
```

3. Commande

Ici, la base de données va être modifiée : il faut être sûr de toujours rester dans un état cohérent.

Créer une commande à emporter correspond à la requête SQL2 suivante :

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRANSACTION

INSERT INTO Commandes VALUES (<emailRestaurant>, <idClient>, <dateActuelle>, <heureActuelle>, <prixFinal>, ‘AttenteConfirmation’)

COMMIT
```

Créer une commande à livrer correspond à la requête SQL2 suivante :

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRANSACTION

INSERT INTO Commandes VALUES (<emailRestaurant>, <idClient>, <dateActuelle>, <heureActuelle>, <prixFinal>, ‘AttenteConfirmation’)

INSERT INTO CommandesLivraison VALUES (<emailRestaurant>, <idClient>, <dateActuelle>, <heureActuelle>, <adresseLivraison>, <heureLivraison>)
COMMIT
```

Créer une commande sur place correspond à la requête SQL2 suivante :

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRANSACTION
```

```
INSERT INTO Commandes VALUES (<emailRestaurant>, <idClient>, <dateActuelle>, <heureActuelle>,
<prixFinal>, ‘AttenteConfirmation’)

INSERT INTO CommandesSurPlace VALUES (<emailRestaurant>, <idClient>, <dateActuelle>,
<heureActuelle>, <nbDePersonnesSurPlace>, <heureArrivee>)

UPDATE Restaurant
SET nbPlacesAssises = nbPlacesAssises - <nbDePersonneReservation>
WHERE emailRestaurant=<emailRestaurant>

COMMIT
```

Annuler une commande correspond à la requête SQL2 suivante :

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRANSACTION

INSERT INTO Commandes VALUES (<emailRestaurant>, <idClient>, <dateActuelle>, <heureActuelle>,
<prixFinal>, ‘AttenteConfirmation’)

UPDATE Commandes
SET statut = ‘AnnuleeParClient’
WHERE emailRestaurant=<emailRestaurant>

// Si il s’agit d’une commande sur place alors on ajoute :
UPDATE Restaurant
SET nbPlacesAssises = nbPlacesAssises + <nbDePersonneReservation>
WHERE emailRestaurant=<emailRestaurant>

COMMIT
```

Gestion des transactions (IMPORTANT)

Le choix du niveau d'isolation pour nos transactions ne fut pas évident à faire.

Par exemple, plaçons-nous le cas particulier où un utilisateur (notons-le U1) est en train de passer une commande sur place dans un restaurant X et qu'un autre client (U2) commence également à passer une commande sur place dans ce même restaurant à un temps très proche (mais supérieur). Si U2 valide sa commande (et donc sa transaction) avant U1 et qu'il réserve le nombre restant de places disponibles dans le restaurant, on veut que U1 puisse voir qu'il n'y a plus de places disponibles dans le restaurant au moment de sa commande (s'il devait voir qu'il n'y a plus de places après avoir validé sa commande il serait bien embêté car il croyait que le nombre de places était suffisant jusqu'à la fin et se préparait à manger un délicieux plat en famille). Le choix d'isolation ici est donc ***serializable*** : il faut que les transactions se suivent pour que chaque utilisateur les valeurs mises à jour sans risque de collisions.

Une deuxième mise en situation serait la suivante : un utilisateur U1 crée un compte client sur l'application et un autre utilisateur U2 veut lui aussi créer au même moment un compte client. Admettons qu'avant les transactions des deux utilisateurs, l'identifiant maximum de la relation IdentifiantClient soit de 4 (chiffre aléatoire, sans importance). Si U1 valide sa création de compte avant U2 l'identifiant maximal passera de 5 à 6 (ajout d'un nouvel identifiant). A ce moment-là il faut absolument que U2 soit au courant PENDANT sa création de compte que l'identifiant maximal a changé et est passé à 6 ! Il est donc nécessaire ici aussi d'effectuer les transactions EN SÉRIE pour pouvoir mettre à jour les tables sans collisions. L'isolation est donc aussi de type ***serializable*** dans ce cas-là.

Plus généralement, toutes les transactions contenant au moins une requête SQL modifiant la base de données (c'est-à-dire toutes celles spécifiées avec une étoile rouge sur notre automate) ont été faites avec un niveau d'isolation *serializable*. A chaque menu doté d'une étoile rouge on ajout donc les deux lignes suivantes : `BEGIN TRANSACTION` et `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE`. On liste alors comme transactions :

- La création d'un compte client (lecture du dernier identifiant client attribué et incrémentation pour donner le nouvel identifiant + insertion d'un nouveau tuple dans les deux relations Clients et IdentifiantClient
- La suppression d'un compte client (suppression d'un tuple uniquement dans la relation Clients, Cf droit à l'oubli)

- L'ajout d'une évaluation à une commande (insertion d'un nouveau tuple dans les relation Evaluations et APourEvaluation + modification de la note d'un restaurant). Cela n'est pas spécifié sur l'automate mais le client peut également annuler cette transaction.
- La modification du statut d'une commande (modification de l'attribut statut d'une commande, ce qui peut impacter le nombre de places disponibles dans un restaurant par exemple, donc cela doit être compté dans une transaction).
- L'ajout d'une commande (insertion d'une nouvelle commande, modification du nombre de places disponibles dans un restaurant s'il s'agit d'une commande sur place).

Il s'agit par ailleurs des seules transactions que l'on fait car les autres requêtes SQL dans les menus ne sont que des lectures qui ne modifient pas la base de données. L'utilisation d'une isolation de type *serializable* est donc coûteuse mais cette dernière reste nécessaire pour éviter les collisions.

Ainsi, pour chacune des transactions listées ci-dessus, lorsque l'utilisateur valide la transaction, ce dernier passe dans une branche dans l'arbre de l'automate qui contiendra des requêtes SQL modifiant la base de données, suivies d'un ***commit***. Si le client décide d'annuler la transaction en cours, le code appellera la méthode *rollback* de la classe ***Navigator*** et qui permettra de recommencer le menu en cours (et donc la transaction). Ainsi, l'utilisation d'un arbre de décisions permet de ne pas avoir à utiliser les notions de *savepoint* et *rollback* en SQL.

Remarque : toutes les transactions ont été faites par défaut avec un autocommit OFF car chaque commit doit se faire à chaque fin de menu lorsque le client valide sa demande et non pas en continu.

V. Bilan du projet

1. Organisation du projet :

Contrairement à certains groupes où les membres étaient amenés à soit faire que de la programmation, soit faire que de l'analyse et où la plupart du temps une seule personne faisait une tâche, nous avons fait en sorte que tout le monde dans le groupe comprenne le projet dans sa totalité. C'est-à-dire : on voulait que tout le monde ait une bonne idée de toutes les étapes qui amènent à la construction d'une base de données : de sa conception (modélisation du problème du client) à son

développement (création de la table, interface pour communiquer avec le serveur, etc). En effet, pouvoir comprendre à la fois les parties théorique et pratique reste primordial si on est amenés à travailler là-dessus plus tard dans notre carrière donc ce projet était une bonne occasion de couvrir les deux d'un seul coup, pour tous les membres du groupe. En revanche, fonctionner comme ça a été fastidieux car prendre des décisions lorsqu'on est 7 est plus compliqué : nous avons mis 4-5 séances entières pour passer de l'analyse (théorique) du sujet au passage relationnel. Le reste des séances était consacré au passage à la programmation, malgré les quelques soucis à la fin pour pouvoir tester notre code (problème informatique à l'Ensimag).

Au niveau de la partie théorique (analyse) :
Nous avons scindé notre groupe en deux parties. Les deux parties effectuent **exactement** le même travail chacune de leur côté indépendamment de l'autre (sans communication) jusqu'à que chacune des deux parties ait fini sa propre analyse. A la fin, nous avons pu mettre nos solutions en commun et nous avons débattu sur les points difficiles (exposés dans la partie 3) pour arriver à un résultat final qui satisfasse les deux parties. De cette manière, chaque partie pouvait apporter un regard critique sur le travail de l'autre partie sans pour autant n'avoir jamais travaillé dessus car certains points du sujet nécessitent de bien réfléchir. Avoir un groupe de personnes n'ayant jamais travaillé sur le sujet n'aurait pas été utile dans ce cas-là (ou du moins cela n'aurait pas aidé à avancer sur les points difficiles car tout le monde n'avait pas la même compréhension du cours).

Au niveau de la partie pratique (programmation) :
Pour la partie programmation, nous nous sommes cette fois-ci répartis les tâches car de toute façon chaque partie du code nécessitait d'avoir un minimum de compréhension pour les autres parties (coder une interface nécessite de savoir comment sont faites les requêtes SQL2 pour pouvoir afficher les bonnes informations en passant les bons arguments, etc.).
Nous nous sommes répartis les tâches par binôme (la septième personne du groupe avait pour rôle de commencer le compte rendu et relire la partie analyse pour détecter d'éventuelles erreurs/oublis). Un binôme a donc travaillé sur les scripts bash permettant de créer et de rafraîchir la base de données, un binôme s'est occupé de la partie avec les requêtes SQL2 permettant de tirer les bonnes informations de la base de données, et un dernier binôme a travaillé sur l'interface utilisateur, permettant d'afficher les bonnes informations sous forme de menus.

2. Difficultés rencontrées :

L'une des principales difficultés rencontrées dans ce sujet est sans aucun conteste l'interprétation du sujet et le choix d'implémentation des fonctionnalités. En plus de l'intégrité des données, nous avons dû faire attention à l'empreinte mémoire de notre base de données, au coût d'accès à travers les différentes requêtes SQL2. Ce type d'interrogations n'était que très peu pris en compte lorsqu'on avait fait une analyse en TD, alors que construire un schéma Entité/Associations garantissant une empreinte mémoire minimale (ou du moins optimisée) est primordial, surtout lorsqu'on se retrouve devant des bases de données immenses.
De ce fait, l'élaboration du schéma Entités/Associations fut fastidieuse car nous avons dû faire des changements pour optimiser notre base de données (par exemple, mettre en commun les entités de même valeur plutôt que de les dupliquer, etc.). Ce schéma a été modifié jusqu'à la dernière semaine, notamment sur la cardinalité des relations.
Cette élaboration fut d'autant plus difficile que nous n'avions sensiblement pas tous bien assimilés certains concepts du cours (entités faibles, notamment, que nous avions mal utilisées au départ). Toutefois, il n'est jamais trop tard pour comprendre car cet exercice nous a permis de mieux appréhender les notions d'entités faibles, sous-entités, cardinalité, etc.
Enfin, le dernier point difficile fut la partie programmation car nous n'avions aucune idée de comment réaliser une interface, de la structure de notre code. Nous avons au départ essayé d'élaborer une vision globale du programme avant de se séparer en binômes afin que chaque partie de chaque binôme soit cohérente avec celle des autres.

Malheureusement, nous n'avons pas de prototype fonctionnel car nous n'avons pas pu tester notre code sur machine la dernière semaine de projet en raison des problèmes informatiques de l'école.

Merci pour votre correction !