# [Initiez-vous à l'algèbre relationnelle avec le langage SQL](https://openclassrooms.com/fr/courses/4449026-initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql)

Table des matières

[Initiez-vous à l'algèbre relationnelle avec le langage SQL 1](#_Toc57644097)

[**Tirez parti de ce cours** 7](#_Toc57644098)

[**Comment ce cours est-il organisé ?** 7](#_Toc57644099)

[**Qu'allons-nous faire ?** 7](#_Toc57644100)

[**Débutons par le modèle relationnel** 8](#_Toc57644101)

[**Découvrez le concept de relation** 10](#_Toc57644102)

[**Un peu d'histoire, pour les motivés !** 10](#_Toc57644103)

[**L'objet phare de ce cours : la relation** 10](#_Toc57644104)

[**Un peu de vocabulaire** 11](#_Toc57644105)

[**En résumé** 11](#_Toc57644106)

[**Comprenez l'importance des clés** 12](#_Toc57644107)

[**Réfléchissons...** 12](#_Toc57644108)

[**Qu'est-ce qu'une clé ?** 12](#_Toc57644109)

[**Un peu d’entraînement** 13](#_Toc57644110)

[**En résumé** 15](#_Toc57644111)

[**Choisissez votre clé primaire parmi les candidates** 16](#_Toc57644112)

[**Comment choisir sa clé primaire ?** 16](#_Toc57644113)

[**Qu'est-ce qu'une clé artificielle ?** 16](#_Toc57644114)

[**Clés primaires et valeurs nulles** 17](#_Toc57644115)

[**Représenter la clé primaire** 17](#_Toc57644116)

[**Aller plus loin : pourquoi aime-t-on les clés artificielles dans les BDD ?** 18](#_Toc57644117)

[**En résumé** 18](#_Toc57644118)

[**Créez du lien entre vos relations grâce aux clés étrangères** 20](#_Toc57644119)

[**Réfléchissons...** 20](#_Toc57644120)

[**Les clés étrangères** 20](#_Toc57644121)

[**En résumé** 22](#_Toc57644122)

[**Évitez la redondance** 23](#_Toc57644123)

[**Réfléchissons ...** 23](#_Toc57644124)

[**Comment éviter la redondance ?** 24](#_Toc57644125)

[**Aller plus loin : les dépendances fonctionnelles** 25](#_Toc57644126)

[**Aller plus loin : la normalisation** 26](#_Toc57644127)

[**En résumé** 26](#_Toc57644128)

[**Utilisez les tables d'association** 27](#_Toc57644129)

[**Réfléchissons...** 27](#_Toc57644130)

[**La table d'association pour une cardinalité plusieurs-à-plusieurs** 28](#_Toc57644131)

[**Aller plus loin : Quelle clé primaire pour une table d'association ?** 29](#_Toc57644132)

[**Aller plus loin : modéliser les différents types de cardinalités** 29](#_Toc57644133)

[**En résumé** 30](#_Toc57644134)

[Le modèle relationnel 31](#_Toc57644135)

[Compétences évaluées 31](#_Toc57644136)

[Description 31](#_Toc57644137)

[ Question 1 31](#_Toc57644138)

[ Question 2 31](#_Toc57644139)

[ Question 3 32](#_Toc57644140)

[ Question 4 33](#_Toc57644141)

[ Question 5 33](#_Toc57644142)

[ Question 6 33](#_Toc57644143)

[ Question 7 34](#_Toc57644144)

[**Découvrez la projection et la restriction** 35](#_Toc57644145)

[**La projection** 35](#_Toc57644146)

[**La restriction** 36](#_Toc57644147)

[**Notations** 36](#_Toc57644148)

[**En résumé** 37](#_Toc57644149)

[**Découvrez les opérateurs ensemblistes** 38](#_Toc57644150)

[**L'union** 38](#_Toc57644151)

[**La différence** 38](#_Toc57644152)

[**L'intersection** 39](#_Toc57644153)

[**Notations** 40](#_Toc57644154)

[**En résumé** 40](#_Toc57644155)

[**Effectuez un produit cartésien** 42](#_Toc57644156)

[**Réfléchissons...** 42](#_Toc57644157)

[**Le produit cartésien** 42](#_Toc57644158)

[**Aller plus loin : la division** 43](#_Toc57644159)

[**Notations** 44](#_Toc57644160)

[**En résumé** 45](#_Toc57644161)

[**Liez des relations grâce aux jointures** 46](#_Toc57644162)

[**Réfléchissons...** 46](#_Toc57644163)

[**La jointure interne** 47](#_Toc57644164)

[**Les jointures externes** 48](#_Toc57644165)

[**Notations** 50](#_Toc57644166)

[**Aller plus loin : voir la jointure autrement** 50](#_Toc57644167)

[**Aller plus loin : la jointure naturelle** 52](#_Toc57644168)

[**En résumé** 52](#_Toc57644169)

[N'oubliez pas l'agrégation ! 53](#_Toc57644170)

[A quoi sert l'agrégation ? 53](#_Toc57644171)

[Les deux éléments qui constituent l'agrégation 53](#_Toc57644172)

[Aller (beaucoup) plus loin : le Map Reduce 56](#_Toc57644173)

[En résumé 57](#_Toc57644174)

[Ne perdez pas de vue vos clés ... 58](#_Toc57644175)

[Pourquoi les clés sont-elles si importantes ? 58](#_Toc57644176)

[Les clés dans les jointures 59](#_Toc57644177)

[Aller plus loin : tester si un groupe d'attributs est une clé candidate 60](#_Toc57644178)

[En résumé 61](#_Toc57644179)

[L'algèbre relationnelle 62](#_Toc57644180)

[Compétences évaluées 62](#_Toc57644181)

[Description 62](#_Toc57644182)

[ Question 1 62](#_Toc57644183)

[ Question 2 63](#_Toc57644184)

[ Question 3 63](#_Toc57644185)

[ Question 4 64](#_Toc57644186)

[ Question 5 65](#_Toc57644187)

[ Question 6 65](#_Toc57644188)

[ Question 7 65](#_Toc57644189)

[Comprenez les bases de données SQL 67](#_Toc57644190)

[Télécharger la base de données 67](#_Toc57644191)

[Petite introduction au SQL 67](#_Toc57644192)

[Un petit mot sur les SGBDR 67](#_Toc57644193)

[Prenez vos précautions ! 68](#_Toc57644194)

[En résumé 68](#_Toc57644195)

[Explorez les Panama Papers 69](#_Toc57644196)

[Le contexte 69](#_Toc57644197)

[La base de données 69](#_Toc57644198)

[Qu'y a-t-il dans notre base de données ? 70](#_Toc57644199)

[Notre investigation 70](#_Toc57644200)

[Un peu de vocabulaire 71](#_Toc57644201)

[Attention 71](#_Toc57644202)

[En résumé 72](#_Toc57644203)

[Posez vos clés sur la table ! 73](#_Toc57644204)

[Créer une table avec des clés 73](#_Toc57644205)

[Insérer des données 74](#_Toc57644206)

[En résumé 75](#_Toc57644207)

[Exécutez une requête avec SELECT, FROM et WHERE 76](#_Toc57644208)

[La projection 76](#_Toc57644209)

[La restriction 76](#_Toc57644210)

[Le produit cartésien 78](#_Toc57644211)

[Projections : les fonctions scalaires, et le mot clé AS 78](#_Toc57644212)

[Écrivez une union, une différence et une intersection 80](#_Toc57644213)

[Faisons le point sur notre investigation 80](#_Toc57644214)

[L'union, l'intersection et la différence 80](#_Toc57644215)

[Oui mais... 81](#_Toc57644216)

[En résumé 82](#_Toc57644217)

[Joignez dans tous les sens avec JOIN ! 84](#_Toc57644218)

[Le point sur notre investigation 84](#_Toc57644219)

[La jointure interne 84](#_Toc57644220)

[Joindre avec une table d'association 86](#_Toc57644221)

[Aller plus loin : les jointures externes 87](#_Toc57644222)

[Aller plus loin : la jointure naturelle 88](#_Toc57644223)

[En résumé 88](#_Toc57644224)

[Agrégez vos données grâce au GROUP BY 89](#_Toc57644225)

[L'agrégation 89](#_Toc57644226)

[Réfléchissons... 89](#_Toc57644227)

[Continuons notre investigation 91](#_Toc57644228)

[Aller plus loin : fonctions d'agrégation et fonctions scalaires 93](#_Toc57644229)

[En résumé 94](#_Toc57644230)

[Triez vos données avec ORDER BY 95](#_Toc57644231)

[La clause ORDER BY 95](#_Toc57644232)

[En résumé 95](#_Toc57644233)

[Améliorez vos agrégations grâce à HAVING 97](#_Toc57644234)

[Réfléchissons... 97](#_Toc57644235)

[La clause HAVING 98](#_Toc57644236)

[En résumé 99](#_Toc57644237)

[Recherchez dans une chaîne de caractères grâce à LIKE 100](#_Toc57644238)

[L'opérateur LIKE 100](#_Toc57644239)

[Majuscules et minuscules 101](#_Toc57644240)

[Les doublons 101](#_Toc57644241)

[Aller plus loin : gérer les doublons dans les Panama Papers 103](#_Toc57644242)

[En résumé 103](#_Toc57644243)

[Imbriquez des requêtes avec IN, ALL, ANY et EXISTS 104](#_Toc57644244)

[Réfléchissons... 104](#_Toc57644245)

[Le mot clé IN 105](#_Toc57644246)

[Les mots clés ANY et ALL 105](#_Toc57644247)

[Le mot clé EXISTS 106](#_Toc57644248)

[Aller plus loin : utiliser IN sur plusieurs colonnes 107](#_Toc57644249)

[Aller plus loin : les tables temporaires 107](#_Toc57644250)

[En résumé 108](#_Toc57644251)

[Appréhendez le fenêtrage avec OVER et PARTITION BY 109](#_Toc57644252)

[Réfléchissons... 109](#_Toc57644253)

[Le fenêtrage 110](#_Toc57644254)

[OVER (PARTITION BY...) 110](#_Toc57644255)

[OVER (PARTITION BY... ORDER BY...) 111](#_Toc57644256)

[La somme cumulée et les numéros de rang 113](#_Toc57644257)

[En résumé 114](#_Toc57644258)

L’information est partout. C’est l’or noir de nouvelles disciplines comme la data science ou l’analyse de données. Ces données peuvent souvent être représentées sous forme de *relation*. Une relation s’apparente à un tableau avec des lignes et des colonnes, tout simplement !

Dans ce cours, **vous apprendrez à manipuler des relations** à l’aide des opérateurs de l’algèbre relationnelle. Ensuite, **vous appliquerez ces concepts théoriques à un langage très utilisé : le SQL**, permettant d’interagir avec des bases de données… relationnelles !

Si vous avez déjà utilisé le langage SQL, alors ce cours vous semblera très simple ! Si c'est le cas, vous avez probablement manipulé des *tables* dans une base de données ; ces tables sont simplement la traduction du concept de relation. En plus, ce cours vous permettra de formaliser des concepts que vous utilisez naturellement en SQL, afin de les réutiliser dans d’autres langages (Python ou R). Vous découvrirez quelques subtilités du langage SQL pour construire des requêtes de plus en plus complexes !

Le SQL est un langage essentiel des Data Analysts, alors lancez-vous !

**Objectifs pédagogiques :**

* Manipuler de l’information à l’aide des différentes opérations de l’algèbre relationnelle
* Appliquer les opérations de l’algèbre relationnelle aux requêtes SQL
* Composer des requêtes SQL simples
* Composer des requêtes SQL avancées

**Prérequis :** Être familier avec ces quelques concepts basiques des langages de programmation :

* les fonctions
* les types de variables (entier : integer, décimal : float, chaîne de caractères, booléen, date)
* les opérateurs logiques (ET, OU, NON)
* les booléens TRUE et FALSE

Si vous ne les connaissez pas, cela ne devrait pas être trop problématique. N'hésitez pas à rechercher la définition de ces termes si besoin. ;)

*Ce cours fait partie d'un parcours de formation Data Analyst réalisé en collaboration avec l'ENSAE-ENSAI*

**Tirez parti de ce cours**

**Comment ce cours est-il organisé ?**

Le cours est organisé en **4 parties** :

* 2 parties concernent le **modèle relationnel**,
* 2 autres parties traduisent les concepts du modèle relationnel en **SQL**.

Ce cours est constitué de courts chapitres. En fonction de votre motivation, certaines sections seront optionnelles. Leur titre commence par "Aller plus loin…"

Il s'agit d'approfondissements, d'ouvertures sur d'autres sujets, ou d'éléments de réflexion.

Les parties "Aller plus loin" ne sont ni traitées dans les vidéos, ni évaluées dans les évaluations à la fin de chaque partie. Bref, elles sont optionnelles ! ;)

**La pratique**

Dans les parties qui traitent du SQL, vous aurez la possibilité de tester les requêtes montrées dans le cours.

Bonne nouvelle, vous n'avez rien besoin d’installer sur votre ordinateur ! Vous pourrez tester directement en ligne les requêtes SQL. C'est simple et rapide !

Pour tester les requêtes SQL en ligne, j'ai mis à la fin de chaque chapitre SQL une console interactive.

C'est en pratiquant que l'on apprend. Vous pouvez copier les requêtes SQL du cours dans la console interactive, vous les comprendrez bien mieux. Mais le must, c'est de "bidouiller" un peu les requêtes, de les modifier pour voir ce qu'elles donnent !

Si la console interactive ne vous convient pas, vous pouvez installer [**SQLiteStudio**](https://sqlitestudio.pl/index.rvt) sur votre ordinateur. C'est d'ailleurs ce logiciel que j'utiliserai dans les vidéos, pour une question d'affichage. Vous aurez la possibilité de télécharger la base de données plus tard dans ce cours.

**Qu'allons-nous faire ?**

L'objectif de ce cours est de découvrir comment manipuler des données.

Pour cela, nous verrons ensemble le **modèle relationnel**, qui est très souvent utilisé pour représenter et manipuler des données structurées.

Ce modèle est à l'origine des **bases de données relationnelles**, qui sont massivement utilisées, et dont découle le célèbre **langage SQL.**

De nos jours, les données ont tendance à être de moins en moins structurées (ce sont les problématiques liées au fameux [**Big Data**](https://openclassrooms.com/courses/comprendre-le-big-data-a-travers-les-films-de-cinema)), et un nombre florissant de bases de données **non relationnelles** voient le jour pour répondre à ce changement.

Cependant, la majorité des bases de données utilisées aujourd'hui restent relationnelles*.* De plus, les concepts du modèle relationnel sont *fondamentaux* pour comprendre les BDD (qu'elles soient relationnelles ou non) et plus largement le monde de la Data Science.

Le modèle relationnel étant sous-jacent au langage SQL, il est important de comprendre ces concepts théoriques, car ceux-ci sont également utilisés dans d'autres langages de programmation, comme le langage R ou Python (via la librairie *Pandas*, [**que vous pouvez découvrir ici !**](https://openclassrooms.com/courses/manipulez-des-donnees-avec-python-1/p2c1-creez-votre-premier-graphique-nico))

**Ce que nous n'aborderons pas**

Nous nous concentrerons sur l'aspect *manipulation* de la donnée, plus que sur sa *représentation*.

Ce cours ne traite pas des bases de données. Ce que nous apprendrons ici, c'est comment questionner et analyser les données mises à notre disposition. Si vous voulez vous former sur les bases de données, je vous conseille le cours [Administrez vos bases de données avec Mysql](https://openclassrooms.com/courses/administrez-vos-bases-de-donnees-avec-mysql), qui vous apprendra à utiliser et administrer un type de base de données : Mysql.

Le langage SQL a de multiples utilisations : remplir une base de données, actualiser, supprimer les données, ou interroger la base. Nous ne nous concentrerons ici que sur la partie *interrogation*. Encore une fois, le cours [Administrez vos bases de données avec Mysql](https://openclassrooms.com/courses/administrez-vos-bases-de-donnees-avec-mysql) va plus loin avec SQL.

Nous ne verrons pas non plus en profondeur le**modèle relationnel**, nous verrons juste ce qu'il faut pour analyser des données. Si vous devez un jour concevoir une base de données relationnelle, ce sera par ici : [Faites une base de données avec UML](https://openclassrooms.com/courses/faites-une-base-de-donnees-avec-uml).

**Débutons par le modèle relationnel**

Qu'est-ce que le modèle relationnel ?

C'est tout simplement une manière de *représenter* et de *manipuler* des données. Plus précisément, il s'agit d'un ensemble de concepts théoriques (pas très compliqués, je vous rassure !).

Ce type de représentation de la donnée est indépendant de la manière dont la donnée est physiquement stockée dans la mémoire de l'ordinateur. Ainsi, quel que soit le type de technologie (processeur, système d'exploitation, etc.) que vous utilisez, vous serez toujours capable de dialoguer avec d'autres personnes utilisant des technologies différentes de la vôtre.

Oui mais moi, je n'aime pas trop ce qui est théorique...

Je vous comprends. C'est vrai que ce cours est composé d'une première partie théorique, mais ensuite, on va pratiquer !

En fait, j'aurais pu vous expliquer le SQL en même temps que le modèle relationnel, car ce dernier est le fondement de ce langage. En général, on apprend directement les requêtes SQL sans passer par la partie théorique.

Mais les concepts théoriques du modèle relationnel, et plus particulièrement *l'algèbre relationnelle*, sont applicables également à d'autres langages souvent utilisés en Data Science (langage R, ou Python et sa librairie Pandas). J'ai donc décidé ici de séparer la partie théorique de la partie pratique en SQL. Vous m'en direz des nouvelles . ;)

**Découvrez le concept de relation**

Tout au long de ce cours, nous jonglerons avec le vocabulaire : il n'est pas toujours simple et surtout, plusieurs termes peuvent désigner une même chose. Revenez donc régulièrement au bas de ce chapitre lorsque vous aurez un doute. ;)

**Un peu d'histoire, pour les motivés !**

C'est un beau jour ensoleillé de 1970 que Monsieur Codd, du laboratoire de recherche d'IBM de San José, crée le [modèle relationnel](https://fr.wikipedia.org/wiki/Mod%C3%A8le_relationnel).

C'était bien vu, car celui-ci a réussi à s'imposer et se retrouve aujourd'hui dans la majorité des bases de données ! :soleil: En effet, ce modèle a été implémenté dans les systèmes de gestion des bases de données relationnelles (SGBDR) comme le IBM System R, Oracle, DB2 ou MySQL. Il a également donné naissance au langage de manipulation des données SQL.

**L'objet phare de ce cours : la relation**

L'objet (théorique) que nous manipulerons tout au long de ce cours est appelé **relation**.

J'espère que vous aimez les pommes, car tout au long de ce cours, nous illustrerons les concepts avec ce délicieux fruit. Une relation, ce sera un peu comme un panier de pommes.

Une relation, c'est un tableau dans lequel on met des données. Un tableau dans lequel une ligne représente un objet, et où chaque ligne représente des objets de même nature.

Si nous représentons des pommes, nous pouvons les caractériser par leur masse, leur diamètre et leur couleur. Nous pouvons aussi leur attribuer un identifiant, c'est-à-dire un nombre (ou un code) *unique* qui permet de les différencier.

|  |  |  |  |
| --- | --- | --- | --- |
| **identifiant** | **masse** | **diamètre** | **couleur** |
| 1 | 151 g | 8.3 cm | rouge |
| 2 | 169 g | 9.1 cm | jaune |
| 3 | 134 g | 8.0 cm | jaune |
|  |  |  |  |

 Dans une relation, l'ordre des lignes n'a pas d'importance et pas de signification.

**Un peu de vocabulaire**

Il est très fréquent de mélanger les termes définis par le *modèle relationnel* et les termes propres au jargon des *bases de données relationnelles*.

Par exemple, une **relation** sera souvent appelée **table** !

De même, le terme **tuple**désigne ce que l'on appelle une **ligne** d'une table. Dans l'exemple précédent, la pomme numéro 2 est représentée par le **tuple** suivant : (2, 169, 9.1, "jaune").

Pour désigner un tuple ou une ligne, vous trouverez aussi parfois les termes **n-uplet**, **enregistrement** ou **vecteur**. Vous avez l'embarras du choix !

Également, le terme **attribut** est l'équivalent, dans le vocabulaire des BDD, d'une **colonne**. Dans l'exemple précédent, *masse* est un attribut.

Le **schéma** est l'ensemble des attributs d'une relation.

Enfin, le **domaine** désigne l'ensemble des valeurs que peuvent prendre les attributs. Dans le jargon des bases de données, le domaine est assimilé au **type de colonne**.

Petite précision sur les domaines. Dans notre exemple, l'attribut *masse* peut prendre toute valeur réelle et positive (c'est à dire tout nombre à virgule supérieur à 0). Également, la colonne

*identifiant* est de type entier (elle peut prendre toute valeur de nombre entier). Enfin, l'attribut *couleur* a comme domaine l'ensemble {vert, rouge, jaune}.

**En résumé**

* Dans ce cours, on va manipuler des tableaux, appelés **relations** ou **tables**.
* Dans ces tableaux, une ligne (= un tuple) représente un objet, et chaque ligne représente des objets de même nature.
* On jonglera souvent entre le vocabulaire de l'**algèbre relationnelle** et le vocabulaire propre aux **bases de données** : connaissez bien les correspondances entre l'un et l'autre !

**Comprenez l'importance des clés**

Je ne vous ai pas tout dit à propos du modèle relationnel. J'ai oublié une chose très importante :

Une relation ne peut contenir deux tuples identiques : chaque ligne est donc unique !

**Réfléchissons...**

Cette contrainte ne devrait normalement pas nous poser de problème, même si nous souhaitons représenter deux objets identiques. Si nous reprenons l'exemple de nos pommes, il est tout à fait possible que deux pommes distinctes aient exactement les mêmes caractéristiques :

|  |  |  |  |
| --- | --- | --- | --- |
| **identifiant** | **masse** | **diamètre** | **couleur** |
| 1 | 151 g | 8.3 cm | rouge |
| 2 | 169 g | 9.1 cm | jaune |
| 3 | 134 g | 8.0 cm | jaune |
| 4 | 134 g | 8.0 cm | jaune |

Ici, les pommes 3 et 4 ont les mêmes caractéristiques. Cependant, les lignes correspondantes ne sont pas identiques, car la valeur de leurs attributs *identifiant* est différente : 3 et 4.

Cela nous amène à l'*identification* des tuples d'une relation. C'est là qu'interviennent les clés !

**Qu'est-ce qu'une clé ?**

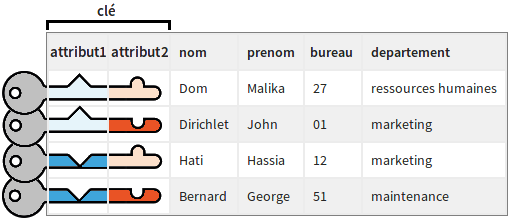
Dans la vie quotidienne, une clé sert à ouvrir une porte pour **accéder** à votre maison. Une caractéristique essentielle d'une clé est qu'elle doit être **unique**, car si votre clé est identique à celle de la maison de votre voisin, il se pose un problème évident !

Dans le modèle relationnel, une clé a donc pour vocation d'**accéder** à un tuple, et donc d'identifier ce dernier.

 Ainsi, pour une relation donnée, une clé est un *groupe d'attribut minimum déterminant un tuple unique.*

Autrement dit, si nous pouvons être sûrs que n'importe quelle ligne peut être retrouvée en connaissant la valeur de quelques-uns de ses attributs, c'est que ces quelques attributs forment une clé !

Cette illustration (qui n'a rien de formel, et qui est une pure vue de l'esprit) vous aidera peut-être à visualiser ce concept :



Dans notre exemple des pommes, l'attribut *identifiant* est une clé, car si je vous dis…

Donne-moi la pomme dont l'identifiant est 2 !

… vous savez exactement quelle ligne me donner, car l'identifiant de chaque pomme est unique !

Cependant, si je vous demande…

Donne-moi la pomme dont la masse est de 169 grammes !

… dans ce cas, vous pourriez me donner la pomme numéro 2. Mais si j'ajoute à ma relation une autre pomme qui a aussi une masse de 169 grammes, vous ne saurez plus laquelle des deux pommes choisir. L'attribut *masse* n'est donc pas une clé.

**Un peu d’entraînement**

Entraînons-nous à repérer des clés avec une relation que nous appellerons **voiture**, qui représente des voitures possédées par des personnes *ayant la nationalité française* (ce détail aura son importance pour la suite) :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **immatriculation** | **pays\_immatriculation** | **couleur** | **nombre\_places** | **propriétaire** | **prop\_num\_secu** | **prop\_num\_voiture** |
| 2983-AA | France | bleue | 5 | Luc | 1134212 | 1 |
| 1923-BD | France | verte | 7 | Lucia | 2021726 | 1 |
| PLT-28-190 | Portugal | bleue | 7 | Gaël | 1012612 | 1 |
| 2383-ZN | France | noire | 9 | Leïla | 2125312 | 2 |
| 1209-NQ | France | noire | 5 | Leïla | 2125312 | 1 |
| 2634-OS | France | bleue | 5 | Gaël | 1162732 | 1 |

Voici la signification des attributs :

* **immatriculation** : numéro d'immatriculation de la voiture
* **pays\_immatriculation** : pays dans lequel le véhicule est immatriculé
* **couleur** : couleur de la voiture
* **nombre\_places** : nombre de places de la voiture
* **propriétaire** : prénom du propriétaire de la voiture
* **prop\_num\_secu** : numéro de sécurité sociale du propriétaire (chaque personne de nationalité française possède un numéro de sécurité sociale unique)
* **prop\_num\_voiture** : si un propriétaire possède plusieurs voitures, il peut les numéroter pour les repérer. S'il n'en possède qu'une, prop\_num\_voiture vaut 1. Par exemple, Leïla numérote ici ses voitures de cette manière : "1209-NQ est ma première voiture, et 2383-ZN est ma voiture numéro 2".

Le groupe d'attributs [immatriculation] est-il une clé pour notre relation ?

Non ! Même si en France, le numéro d'immatriculation d'une voiture est unique, notre relation est visiblement prévue pour contenir des voitures de pays différents. Or, il n'est pas exclu qu'un autre pays utilise (ou utilisera un jour) le même système d'immatriculation que la France. Dans ce cas, deux voitures dans le monde pourraient avoir le même numéro !

Le groupe d'attributs [immatriculation, pays\_immatriculation] est-il une clé pour notre relation ?

Oui ! A condition de supposer qu'un numéro d'immatriculation ne peut pas être réattribué à un autre véhicule une fois qu'une voiture est détruite.

Et le groupe [immatriculation, pays\_immatriculation, couleur] ?

Oui et Non... Comme nous venons de le voir, le groupe [immatriculation , pays\_immatriculation] est déjà une clé. Ceci implique que deux lignes ne peuvent pas avoir la même valeur pour leurs attributs *immatriculation* et *pays\_immatriculation.*

De manière évidente, deux lignes ne peuvent donc pas avoir les mêmes valeurs pour les 3 attributs ci-dessus.

Cependant, dans la définition de *clé* donnée plus haut, nous avons dit que le groupe d'attributs devait être minimal. Ici, notre groupe de 3 attributs n'est pas minimal. En effet, si on enlève l'attribut *couleur*, on se retrouve avec [immatriculation, pays\_immatriculation] qui est une clé (on vient de le voir au cas précédent, j'espère que vous suivez ;) ). *Couleur* est donc inutile ici.

Et le groupe [prop\_num\_secu] ?

Non. Bon, certes, si nous avions une relation qui représentait des personnes de nationalité française, alors le numéro de sécurité sociale serait une clé. Mais ici, ce n'est pas le cas : notre relation représente des voitures, pas des personnes. Deux voitures peuvent avoir le même propriétaire, et donc avoir la même valeur pour l'attribut *prop\_num\_secu.*

Et le groupe [prop\_num\_secu, prop\_num\_voiture] ?

Oui, c'est une clé. En effet, si deux voitures ont le même propriétaire (identifié par *prop\_num\_secu*), elles ne peuvent pas avoir à la fois le même propriétaire ET le même *prop\_num\_voiture*.

Pour ceux qui veulent se "triturer" le cerveau : si nous enlevions la supposition selon laquelle les voitures sont possédées par des personnes de nationalité française, et qu'il existait un pays ayant la même numérotation de sécurité sociale qu'en France, alors ce dernier groupe d'attribut ne serait plus une clé !

**En résumé**

* Une clé, c'est un groupe d'attributs (= un groupe de colonnes).
* Pour qu'un groupe d'attribut soit une clé, il faut être sûr que deux tuples (= deux lignes) n'auront jamais des valeurs identiques pour ces attributs.
* Ce groupe d'attributs doit être minimal, c'est-à-dire que si on retire l'un des attributs de ce groupe, la phrase précédente n'est plus vérifiée.

**Choisissez votre clé primaire parmi les candidates**

Nous venons de voir que plusieurs ensembles d'attributs peuvent être des clés. Mais dans une base de données, nous souhaitons nous mettre d'accord une bonne fois pour toutes sur l'une d'entre elles.

Nous organisons donc un "grand concours" : l'élection de la **clé primaire**! La clé primaire est tout simplement une clé que nous choisissons comme celle qui nous permettra d’identifier un tuple dans une relation.

A ce grand concours, toutes les clés possibles sont appelées **clés candidates**. Ce sont en fait toutes les clés qui remplissent les critères définis dans le chapitre précédent.

**Comment choisir sa clé primaire ?**

Dans le contexte des **bases de données relationnelles**, la notion de clé primaire est importante. En effet, certains SGBDR (systèmes de gestion de base de données relationnelles) ne sont vraiment pas contents :colere:lorsque vous créez une table sans lui spécifier de clé primaire, et ils ne manqueront pas de vous le faire savoir !

Le choix d'une clé primaire peut très bien être arbitraire. Cependant, on peut prendre en compte certains critères. En effet, une clé primaire est généralement choisie de façon à ce qu'elle soit *simple*, c'est-à-dire à ce qu'elle contienne le moins d'attributs possibles. De plus, on préfère généralement les attributs "basiques" (par exemple des entiers ou des chaînes de caractères courtes). Le contexte aide généralement à trouver une clé primaire qui semble la plus "logique" et intelligible pour un être humain.

Oui mais parfois, aucune des clés candidates ne contient un nombre d'attributs raisonnable et ne semble "simple" !

Exact. Dans ce cas, il est possible de créer une **clé artificielle**.

**Qu'est-ce qu'une clé artificielle ?**

Une clé artificielle est un attribut que l'on ajoute à la relation. Cet attribut n'a pas de réelle signification dans le domaine que l'on modélise, mais sa seule fonction est d'identifier de manière unique les tuples de la relation.

Vous vous souvenez de notre relation **pommes** ?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **identifiant** | **masse** | **diamètre** | **couleur** | **nom\_variété** |
| 1 | 151 g | 8.3 cm | rouge | Ariane |
| 2 | 169 g | 9.1 cm | jaune | Gala |
| 3 | 134 g | 8.0 cm | jaune | Gala |
| 4 | 134 g | 8.0 cm | jaune | Ariane |

Dans le domaine modélisé (celui des pommes), la masse, le diamètre et la couleur caractérisent bien une indication sur la pomme (ici, sur sa morphologie). Cependant, la colonne *identifiant* ne caractérise absolument pas notre pomme, en effet, on aurait très bien pu permuter les identifiants, et attribuer l'identifiant 3 (au lieu de 1) à la pomme rouge sans que cela ne modifie l'exactitude de notre représentation.

Vous l'aurez compris, l'attribut identifiant est donc une **clé artificielle**, car elle n'a été créée que pour différencier et identifier les pommes !

En général, dans le contexte des bases de données, on aime bien utiliser des clés artificielles. Dans certaines applications, elles sont même utilisées de manière systématique. Pour savoir pourquoi, descendez jusqu'au paragraphe *Aller plus loin : pourquoi aime-t-on les clés artificielles dans les BDD ?*

**Clés primaires et valeurs nulles**

Il arrive parfois que la valeur de certains attributs pour un tuple donné soit inconnue, ou bien qu'elle n'existe pas. Il est donc possible de laisser une "case blanche" dans la table.

Pour représenter une case vide, on note sa valeur comme étant NULL.

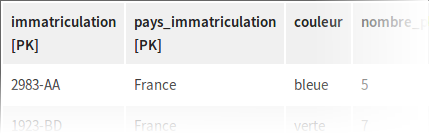
Quasiment chaque langage de programmation a son propre équivalent pour spécifier qu'une variable n'a pas de valeur : None en Python, NULL (pour les pointeurs) en C et C++, null en Java, NULL en SQL, etc.

Cependant, en ce qui concerne les attributs d'une clé, il est nécessaire que ceux-ci ne soient jamais NULL. En effet, comme c'est la clé qui permet d'accéder à une ligne donnée, il faut absolument que tous ses attributs soient renseignés !

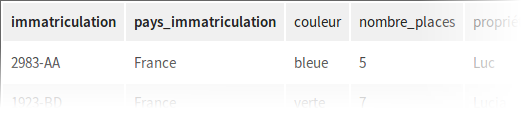
Lorsque vous choisissez une clé primaire, ayez bien en tête qu'à l'ajout d'une nouvelle ligne, les valeurs des attributs de cette clé devront être connues. S'il y a un risque que cela ne soit pas le cas, alors c'est que c'est une mauvaise clé primaire ! On n'ouvre pas une maison avec une clé incomplète !

**Représenter la clé primaire**

En termes de notation, il est courant d'indiquer les attributs de la clé primaire en indiquant leur nom suivi de *[PK]* (mis pour*primary key*, clé primaire en anglais) :



Une autre notation possible est de les indiquer en gras :



Dans ce cours, je représente les noms de colonne (sur la 1ère ligne) toujours en gras, pour les différencier des tuples de la relation. Ayez donc bien conscience que dans ce cours, des noms d'attributs en gras ne représentent PAS une clé primaire.

**Aller plus loin : pourquoi aime-t-on les clés artificielles dans les BDD ?**

N'oubliez pas que tout au long de ce cours, les sections commençant par *Aller plus loin* sont facultatives, et ne sont pas présentes dans les évaluations.

En général, dans le contexte des bases de données, on aime bien utiliser des clés artificielles. Dans certaines applications, elles sont même utilisées de manière systématique. Pourquoi ?

* Au chapitre précédent, nous nous demandions si les deux attributs *[ prop\_num\_secu, prop\_num\_voiture ]* constituaient une clé candidate. La réponse était oui, à condition de supposer que l'on ne prenait en compte, dans notre application, que des propriétaires de nationalité française. Cependant, si notre application évolue, et que nous souhaitons généraliser notre application à des personnes d'autres nationalités, alors notre clé n'est plus bonne. Une clé artificielle est donc une bonne solution à l'*évolutivité* de la base de données.
* Pour des questions de *performance* (en termes de temps de calcul), les clés non artificielles ne sont souvent pas optimisées lorsqu'il est demandé au SGBDR de retrouver une ligne dans la table.

**En résumé**

* Quand plusieurs clés sont possibles sur une table, on les appelle des **clés candidates**.
* Parmi les clés candidates, on en choisit une qui servira de référence : c'est la **clé primaire**. En anglais : *primary key (PK)*.
* Si aucune clé candidate n'est simple et intelligible, on crée une **clé artificielle**, souvent appelée **identifiant**.

**Créez du lien entre vos relations grâce aux clés étrangères**

Nous avons pour le moment une relation qui contient des **pommes**. C'est bien, mais sachez que dans une base de données, il est très rare de ne trouver qu'une seule table ! Les applications informatiques qui utilisent des bases de données sont en général complexes et nécessitent de nombreuses tables.

**Réfléchissons...**

En général, les tables d'une base de données ne sont pas indépendantes entre elles, elles sont liées.

Je décide par exemple d'avoir une relation **pomme**, et également une seconde relation **variété** destinée à répertorier les différentes variétés de pommes connues. Ces deux relations sont *liées* entre elles, car une pomme **est** d'une certaine variété. Il y a donc un lien entre la pomme et sa variété.

Dans le contexte d'un magasin de fruits, on peut dire que le prix d'une pomme dépend de sa variété. Le prix au kilogramme est donc une caractéristique propre à la variété. On aura donc dans la relation **variété** un attribut *prix\_au\_kilo*:

|  |  |  |  |
| --- | --- | --- | --- |
| Variete | | | |
| **libellé** | **prix\_au\_kilo** | **maturation** | **goût** |
| Ariane | 3.19 | tardive | sucré/acidulé |
| Gala | 3.49 | précoce | sucré |
| Reinette | 3.19 | mi-saison | sucré |
| Boskoop | 2.99 | mi-saison | acidulé |
| [...] | [...] | [...] | [...] |

Si je prends une pomme, et que je veux connaître son prix au kilo, il faudra d'abord que je connaisse sa variété. Ensuite, à partir de sa variété, il faudra que je retrouve dans la table **variété** la ligne correspondante pour connaître le prix.

Retrouver une ligne dans une table ?!? Ce n'est pas justement le rôle des **clés** ?

Tout à fait ! Cette question nous amène au paragraphe suivant :

**Les clés étrangères**

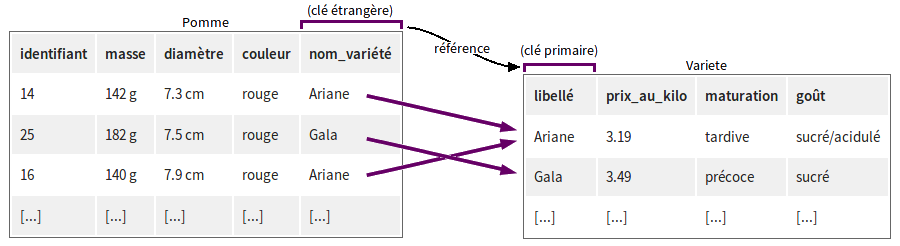
Dans notre relation **variété**, il y a une clé candidate : l'attribut *libellé*. Ainsi, si pour chaque ligne de la table **pomme** nous renseignons le nom de la variété dans une colonne, alors il sera possible, grâce à cette clé, de retrouver le prix au kilo. Voici à quoi pourrait ressembler notre nouvelle table **pomme** :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Pomme | | | | |
| **identifiant** | **masse** | **diamètre** | **couleur** | **nom\_variété** |
| 14 | 142 g | 7.3 cm | rouge | Ariane |
| 25 | 182 g | 7.5 cm | rouge | Gala |
| 16 | 140 g | 7.9 cm | rouge | Ariane |
| [...] | [...] | [...] | [...] | [...] |

Dans la table **pomme**, la colonne *nom\_variété* n'est pas une clé candidate. Cependant, cette colonne fait *quand même* référence à la clé primaire d'une autre table (la table **variété**). On dit donc que :

La colonne *nom\_variété* est une **clé étrangère** référençant la colonne *libellé* de la table **variété**.

Vous savez maintenant lier des tables grâce à des clés étrangères !



"nom\_varieté" est une clé étrangère qui référence la colonne "libellé" de la table variété.

La clé primaire d'une table *A* peut être composée de plusieurs colonnes. Si une table *B* veut référencer *A*, alors la clé étrangère sur *B* sera composée d'autant de colonnes que pour la clé primaire de *A*.

Pour indiquer qu'un attribut (ou groupe d'attributs) est une clé étrangère, on peut le spécifier en faisant suivre le nom de cet attribut (ou de chacun de ces attributs) par *[FK]*, qui signifie *foreign key* (clé étrangère en anglais).

**En résumé**

* Une clé étrangère sert à lier des relations (= des tables) entre elles.
* On dit qu'une clé étrangère d'une table A **référence** la clé primaire d'une table B.
* Une clé étrangère se dit en anglais *foreign key* (FK)

**Évitez la redondance**

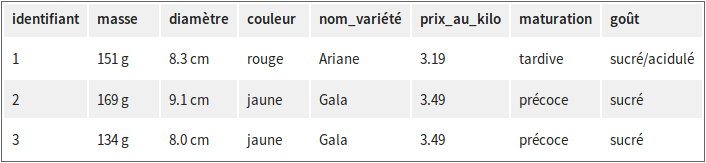
Ce chapitre est un peu technique. La vidéo est ici bien plus explicite que le texte, et vous aidera à mieux comprendre.

**Réfléchissons ...**

C'est bien beau d'avoir plusieurs tables avec des clés étrangères entre elles, mais dans ma base de données, je préfère que toutes mes informations soient regroupées dans une seule grande table !

C'est une très mauvaise idée ! :pirate:

En effet, en suivant ce raisonnement, nous aurions ce type de relation, dans laquelle les informations sur les pommes et sur les variétés seraient regroupées :



Cela pose plusieurs problèmes :

* Si le prix au kilo d'une variété change, il faudra le modifier sur *toutes* les lignes correspondant à des pommes de la variété en question. En informatique, on n'aime pas vraiment qu'une information soit stockée à plusieurs endroits (i.e. la **redondance de données**), car en cas de modification, il est probable d'oublier d'actualiser l'information *partout* où c'est nécessaire. Si l'information n'est pas modifiée partout, alors la cohérence est perdue !
* Si nous souhaitons ajouter les informations d'une nouvelle variété à notre base de données, mais que nous n'avons pas encore de pomme de cette variété, nous ne pouvons pas le faire. En effet, chaque ligne de la relation ci-dessus représente une pomme. Si nous n'avons pas plus de pommes qu'avant, il nous est interdit d'ajouter une ligne à cette table, même si nous avons découvert les caractéristiques d'une nouvelle variété !

**Comment éviter la redondance ?**

Très bien, mais si j'avais été mis face à cette relation (qui regroupe à la fois des pommes et des variétés), comment aurais-je pu savoir qu'il fallait la diviser en deux relations ?

**La règle**

Il y a une règle pour cela ! ;) Elle est compliquée, mais je donne un exemple juste en dessous :

Dans une relation, si un attribut *A* dépend uniquement d'un groupe d'attributs *G*(et que ce groupe d'attributs n'est pas une clé candidate), alors il est possible de créer une nouvelle relation qui contiendra les attributs A et G.

Il faut cependant s'assurer que *G* soit minimal (c'est-à-dire que l'on ne puisse pas enlever d'attribut au groupe*G* sans casser la dépendance entre *A* et *G*).

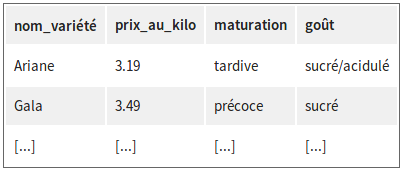
G sera d'ailleurs une clé candidate pour la nouvelle relation créée.

Si un autre attribut *B* dépend également uniquement de G, alors il faut aussi le déplacer dans la nouvelle relation créée !

**Avons-nous appliqué cette règle avec nos relations pomme et variété ?**

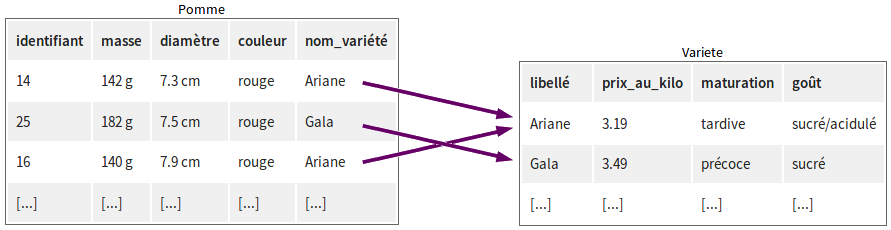
Dans l'exemple en haut de cette page, l'attribut *prix\_au\_kilo* ne dépend que de la variété, mais *nom\_variété* n'est pas une clé candidate.

Ainsi on peut créer une nouvelle table dans laquelle *nom\_variété* est la clé primaire. Elle contiendra l'attribut *prix\_au\_kilo*. De même, les attributs *maturation* et *goût* ne dépendent eux aussi que de *nom\_variété*! On peut donc les ajouter à notre nouvelle relation.



Notre nouvelle relation

Nous voici donc avec une relation toute nouvelle et toute belle ! Que remarquez-vous ? Que cette nouvelle relation est exactement la même que la table **variété** que nous avions définie dans les chapitres précédents (au nom de colonne près) :



**Faut-il à tout prix éviter la redondance ?**

Dans les chapitres suivants, lorsque nous manipulerons les données, nous serons amenés à former de nouvelles tables, dont certaines ressembleront à la table du haut de cette page, qui est redondante. Ce n'est pas un problème lorsqu'on *manipule* les données pour les questionner. La redondance, il faut l'éviter dans le *stockage* des données.

**Aller plus loin : les dépendances fonctionnelles**

Oui, mais comment savoir si "un attribut dépend d'un groupe d'attributs" ? Cela ne semble pas toujours évident !

C'est vrai. Voici donc un petit "truc" :

Soit une relation **pomme.** Imaginez-vous une pomme hypothétique, donc vous ne connaissez pas les caractéristiques. Ou plutôt, vous ne connaissez d'elle que *certaines* caractéristiques, c'est-à-dire certains attributs seulement. Appelons ces attributs *G*. Maintenant, demandons-nous si un attribut *A* dépend (ou non) de *G*. La question à se poser est la suivante :

En ne connaissant que G, et en ayant à disposition la relation **pomme**, puis-je trouver de manière certaine A ?

**Exemple :** Je vous dis que j'ai une pomme cachée derrière mon dos. Je vous donne également un papier sur lequel est imprimée la totalité de la relation **pomme**. Si la seule chose que je veuille bien vous dire est...

* que cette pomme est de variété *Gala* (**G=[nom\_variete]**). Pouvez-vous me dire si son goût sera sucré ou acidulé (**A=gout**)?
  + La réponse est **oui.** Remarquez que vous n'avez même pas besoin de savoir précisément de quelle pomme il s'agit, car de toute manière, toutes les pommes *Gala* sont toujours sucrées !
* que cette pomme est de variété *Gala*. Pouvez-vous me donner sa masse ?
  + La réponse est **non**.
* que cette pomme a pour identifiant 2 et qu'elle est de couleur jaune (**G=[identifiant,couleur]**). Pouvez-vous me donner sa masse ?
  + La réponse est **oui**.

Si la réponse est non, alors *A* ne dépend pas (ou pas uniquement) de *G*. Si la réponse est oui, alors *A* dépend de *G*.

Cependant, dans le 3e exemple, G n'est pas minimal, car si on enlève *couleur* à G, on ne casse pas la dépendance entre **A=masse** et **G=[identifiant]**.

**Aller plus loin : la normalisation**

Séparer des relations comme nous venons de le faire évite la **redondance** de données. En effet, cela permet qu'une information ne soit présente qu'à un seul endroit.

Dans le domaine des bases de données, le fait d'enlever les redondances s'appelle la **normalisation**.

Les concepts de la normalisation sont très importants à connaître si vous concevez vous-même une base de données.

Pour en savoir plus, je vous renvoie vers le chapitre [Optimisez votre modèle relationnel avec les formes normales](https://openclassrooms.com/courses/faites-une-base-de-donnees-avec-uml/optimisez-votre-modele-relationnel-avec-les-formes-normales) du cours *Faites une base de données avec UML*, qui explique très bien la normalisation. Mais pensez à revenir ici ensuite, sinon je serai jaloux ! ;)

**En résumé**

* Dans les bases de données, on n'aime pas la redondance d'information.
* Si une table contient de la redondance, mieux vaut la séparer en plusieurs tables avant de la stocker dans la base de données.
* Il y a une règle pour savoir comment séparer une table redondante.

**Utilisez les tables d'association**

**Réfléchissons...**

Nous avons créé un lien entre **pomme** et **variété** à l'aide d'une clé étrangère.

Cela fonctionne plutôt bien, car une pomme n'a qu'une seule variété. Mais que se passerait-il si, dans un monde imaginaire, une pomme pouvait avoir plusieurs variétés ?

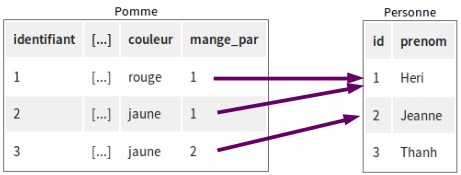
C'est un peu bizarre ton truc là ! :'(

C'est vrai. Oublions la variété, et prenons un meilleur exemple.

Disons qu'une pomme peut être mangée par quelqu'un. Du coup, j'introduis ici une table **personne**, qui contient des gens susceptibles de manger des pommes.

Une personne peut manger plusieurs pommes, mais plusieurs personnes peuvent se partager une même pomme (s'ils n'ont pas très faim, ou si la pomme est très grosse !).

Dans la table **pomme**, mettons une colonne *mangé\_par* qui est une clé étrangère vers la table **personne**.



Clé étrangère de Pomme vers Personne

Mais si la pomme est mangée par 2 personnes, que fait-on ?

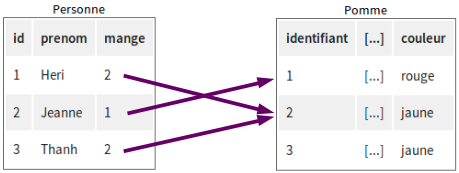
On pourrait mettre 2 colonnes dans **pomme** : *mangé\_par\_1*, et *mangé\_par\_2*. Elles seraient toutes deux clés étrangères vers **personne**.

Mais si la pomme est mangée par plus de deux personnes, combien de colonnes doit-on mettre ? Doit-on en prévoir 1000, au cas où ?

Malheureusement, ce raisonnement est plutôt mauvais. En effet, une table avec plus de 1000 colonnes, c'est volumineux, et pas très pratique à manipuler. De plus, nous ne pouvons pas être certains qu'un jour, nous n'aurons pas besoin d'ajouter une 1001ième clé étrangère.

Bon. Et si on essayait autre chose. Plutôt que de mettre une clé étrangère de **pomme** vers **personne**, pourquoi ne pas en mettre une de **personne** vers **pomme** ?

Pourquoi pas, cela donnerait cela :



Clé étrangère de Personne vers Pomme

On voit qu'une pomme peut être mangée par plusieurs personnes.

Mais... :'( Inverser le sens de la clé étrangère n'a fait qu'inverser le problème ! Certes, une pomme peut être mangée par plusieurs personnes, mais avec notre nouveau modèle, une personne ne peut plus manger qu'une seule pomme !

Or dans la réalité, une seule personne a le droit de manger plusieurs pommes, non ?

Se poser la question *"Combien de pommes peut manger une personne, et par combien de personnes peut être mangée une pomme"*, c'est se poser la question de la *cardinalité* du lien entre **pomme** et **personne**.

Plusieurs cardinalités sont possibles :

* **de 1 à plusieurs** (ex : une pomme pour plusieurs personnes)
* **de plusieurs à 1**
* **de plusieurs à plusieurs** (ex : plusieurs pommes pour 1 personne, plusieurs personnes sur 1 pomme)
* **de 1 à 1** (ex : une pomme par personne, et basta !)

La solution à notre problème est de créer une troisième table, que l'on appelle parfois *table d'association*, ou *table de composition*, ou bien d'un tout autre nom si vous en avez envie (pourquoi pas "mange" dans notre cas) !

**La table d'association pour une cardinalité plusieurs-à-plusieurs**

La table d'association est utilisée dans le cadre d'une cardinalité plusieurs-à-plusieurs entre deux objets. Elle est composée d'au moins 2 clés étrangères, référençant chacune l'un des 2 objets.

Voici un exemple dans le cas d'une pomme pouvant être mangée par plusieurs personnes, où chacune de ces personnes peut également manger plusieurs pommes différentes :

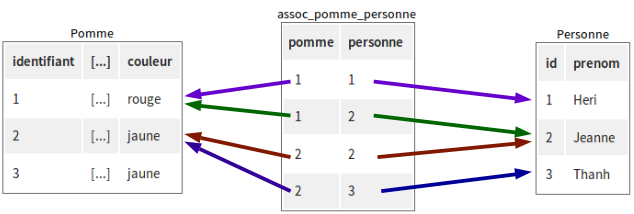


Table d'association

On remarque ici que Jeanne mange à la fois la pomme 1 et la pomme 2, mais que la pomme 2 est également mangée par Thanh !

**Aller plus loin : Quelle clé primaire pour une table d'association ?**

La clé primaire d'une telle table est composée *au moins* des 2 clés étrangères. Mais il est parfois nécessaire d'ajouter des colonnes supplémentaires à la clé.

Par exemple, si nous avions ajouté la notion de date dans notre table d'association, on pourrait dire "Heri a mangé la pomme 2 le 21 avril". Mais il aurait également pu manger la même pomme à une autre date (s'il ne l'a pas mangée en entier le 21 avril, il peut la finir le 22 avril). Ainsi, la clé primaire aurait été [pomme, date, personne].

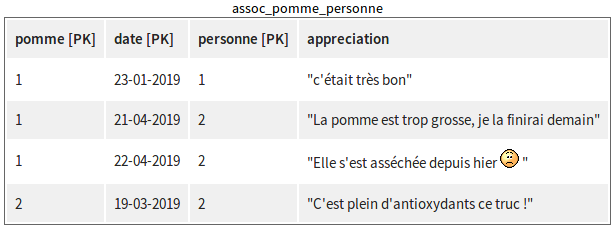


Table d'association avec une PK de 3 colonnes

**Aller plus loin : modéliser les différents types de cardinalités**

Si vous souhaitez mieux comprendre, ou mieux formaliser le concept de cardinalité et son implémentation, je vous invite à jeter un coup d’œil au chapitre [Mettez en œuvre les différents types de relations à l’aide des clés étrangères](https://openclassrooms.com/courses/faites-une-base-de-donnees-avec-uml/mettez-en-oeuvre-les-differents-types-de-relations-a-laide-des-cles-etrangeres) du cours *Faites une base de données avec UML*.

Dans ce cours, quand Loïc parle de « type de relation », le mot *relation* n’a pas du tout le même sens que celui auquel nous sommes habitués ici ! *Relation* n’y est pas synonyme de *table*, mais plutôt de « lien entre objets ».

**En résumé**

* Se poser la question "Combien de pommes peut manger une personne, et par combien de personnes peut être mangée une pomme", c'est se poser la question de la **cardinalité**.
* Pour une cardinalité plusieurs-à-plusieurs, on doit introduire une **table d'association**.
* Une table d'association contient au moins deux clés étrangères vers les 2 objets qu'elle lie.

# Le modèle relationnel

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Comprendre la notion de clé
* Comprendre la notion de relation

### Description

Testez vos compétences sur les concepts de base du modèle relationnel.

### Question 1

**Quel est l'objet de base du modèle relationnel ?**

* + 

La clé primaire

* + 

La clé étrangère

* + 

Les pommes

* + 

La relation

* + 

L'union

*Besoin d'un petit rappel? Rendez-vous au chapitre****Découvrez le concept de relation et le vocabulaire du cours !***

### Question 2

**Trouvez les 2 phrases qui sont correctes :**

*Attention, plusieurs réponses sont possibles.*

* + 

Une relation est implémentée dans une base de données par une table.

* + 

Un tuple contient des lignes.

* + 

Les termes tuple, ligne, vecteur et enregistrement sont tous synonymes.

* + 

Les termes vecteur, enregistrement, n-uplet, clé sont tous synonymes.

*C'est une question de vocabulaire, pensez à vous reporter régulièrement au chapitre****Découvrez le concept de relation et le vocabulaire du cours****!*

### Question 3

**Vous gérez la base de données de votre entreprise. Dans celle-ci se trouve la relation "employés" qui répertorie tous les employés de l'entreprise. Voici les premières lignes de cette table :**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **nom** | **prenom** | **telephone\_professionnel** | **numero\_bureau** | **identifiant** | **departement** |
| Dom | Malika | 01 29 38 \*\* 01 | 27 | 2893 | ressources humaines |
| Dirichlet | John | 01 29 38 \*\* 02 | 01 | 2983 | marketing |
| Hati | Hassia | 01 29 38 \*\* 03 | 12 | 1829 | marketing |
| Bernard | George | null | 51 | 2993 | maintenance |

**Quelle phrase est fausse ?**

* + 

Il est possible que la colonne departement soit une clé étrangère vers une autre table.

* + 

L'attribut telephone\_professionnel est une clé artificielle.

* + 

Pour que [ numero\_bureau ] soit une clé candidate, il faut absolument qu'aucun employé ne partage son bureau avec un autre.

* + 

Si les attributs [nom, prenom] sont choisis comme clé primaire, alors il sera impossible d'ajouter à la base de données deux personnes ayant les même noms et prénoms.

*Pour plus de précisions, rendez-vous aux chapitres****Comprenez l'importance des clés****,****Choisissez votre clé primaire parmi les candidates****et****Créez du lien entre vos relations grâce aux clés étrangères****.*

### Question 4

**Trouvez la phrase correcte :**

* + 

Dire qu'une colonne est de type entier est équivalent à dire que le domaine de l'attribut en question est l'ensemble des nombres entiers.

* + 

Dire qu'une colonne est de type entier est équivalent à dire que cette colonne n'est pas incomplète.

*C'est une question de vocabulaire, pensez à vous reporter régulièrement au chapitre****Découvrez le concept de relation et le vocabulaire du cours.***

### Question 5

**Lorsque l'on stocke une table dans une base de données, si cette table est redondante :**

* + 

C'est bien.

* + 

Ce n'est pas bien, il faut éviter la redondance (sauf dans certains cas non traités dans ce cours).

*Il faut éviter la redondance lorsque l'on stocke une information dans une base de données. Pour savoir pourquoi, allez faire un tour au chapitre****Évitez la redondance****.*

*Il arrive cependant que la redondance soit autorisée dans certains cas non traités dans ce cours, par exemple pour des raisons de performance (accès plus rapide à l'information) ou dans le cas de****Data warehouses****où il faut*[*dénormaliser*](https://fr.wikipedia.org/wiki/Forme_normale_(bases_de_donn%C3%A9es_relationnelles))*les données.*

### Question 6

**Lorsqu'une table contient de la redondance, comment peut-on supprimer cette redondance ?**

* + 

En la séparant en plusieurs tables, après avoir analysé les dépendances entre les colonnes.

* + 

En supprimant les doublons dans la table redondante.

*Il faut la séparer en plusieurs tables, après avoir analysé les dépendances entre les colonnes.*

*Pour plus de précisions, faites un petit tour au chapitre****Évitez la redondance****!*

### Question 7

**Si un employé peut avoir plusieurs chefs, et que plusieurs chefs peuvent diriger plusieurs employés, comment modéliser cette situation ?**

* + 

Il faut créer une table d'association entre une table ***employé*** et une table ***chef***.

* + 

Il faut créer une clé étrangère sur une table ***employé***, référençant une table ***chef***.

* + 

Il faut créer une clé étrangère sur une table ***chef***, référençant une table **employé**.

*Pour modéliser un lien de type plusieurs-à-plusieurs, il faut utiliser une table d'association. Voir le chapitre****Utilisez les tables d'association****.*

**Découvrez la projection et la restriction**

Dans les précédents chapitres, nous avons vu que le modèle relationnel permettait de représenter et de manipuler des données. Nous y avons abordé la partie représentation, avec les concepts de relation et de clé étrangère.

Maintenant, passons donc à la partie manipulation ! Dans le modèle relationnel, tout ce qui concerne la manipulation des relations est appelé**l'algèbre relationnelle.** Peut-être que le mot algèbre vous procure des frissons, et fait remonter en vous de vieux souvenirs de cours de mathématiques. Laissez-moi vous rassurer, l'algèbre relationnelle, c'est bien plus fun !

Pour répondre à l'évaluation de fin de partie, il vous faudra en plus de la vidéo, consulter la section *Notations* ci-dessous.

Sans plus tarder, passons aux deux premières opérations de l'algèbre relationnelle : la **projection** et la **restriction**.

Ces deux opérations portent sur une seule relation. Reprenons ici notre relation **pomme** :

|  |  |  |  |
| --- | --- | --- | --- |
| **identifiant** | **masse** | **diamètre** | **couleur** |
| 1 | 151 g | 8.3 cm | rouge |
| 25 | 182 g | 7.5 cm | rouge |
| 16 | 140 g | 5.9 cm | rouge |

**La projection**

La projection, c'est simplement le fait de sélectionner les attributs d'une relation que l'on souhaite, en éliminant les autres.

Sur notre exemple des pommes, la projection de la relation **pomme** sur ses attributs *diamètre* et *couleur* donne comme résultat une nouvelle relation ne comptant que deux attributs : *diamètre* et *couleur*, et contenant les mêmes tuples que la relation **pomme**.

Voici le résultat de cette projection :

|  |  |
| --- | --- |
| **diamètre** | **couleur** |
| 8.3 cm | rouge |
| 7.5 cm | rouge |
| 5.9 cm | rouge |

Bon, tout cela est exprimé en termes un peu pompeux, mais au final, on a juste supprimé les colonnes *identifiant*et *masse* !

**La restriction**

Avec la projection, on se contentait de sélectionner les colonnes qui nous intéressaient. La restriction, c'est l'équivalent de la projection, mais pour les lignes.

Cependant, dans une base de données, nous connaissons les noms des colonnes, car ils ne varient pas. En ce qui concerne les lignes, on ne les connaît pas à l'avance, et elles varient au cours du temps. C'est bien là la vocation d'une base de données : stocker des données dynamiques, que l'on ajoute, que l'on supprime, ou que l'on modifie au fil de l'utilisation de l'application !

Comme les lignes à garder (ou enlever) sont dynamiques, il nous faut donc établir une condition qui nous permette de *restreindre* les lignes.

Voici un exemple : la restriction de la relation **pomme**, étant donnée la condition C : diamètre > 6 cm donne une relation contenant les tuples vérifiant C.

Voici le résultat de cette restriction, une relation avec 2 lignes :

|  |  |  |  |
| --- | --- | --- | --- |
| **identifiant** | **masse** | **diamètre** | **couleur** |
| 1 | 151 g | 8.3 cm | rouge |
| 25 | 182 g | 7.5 cm | rouge |

Une restriction peut également porter sur plusieurs colonnes, par exemple diamètre > 2 \* masse.

Cet exemple n'a aucun sens, je vous l'accorde, mais sachez que c'est possible.

En résumé, une restriction, c'est juste un "filtrage" de lignes selon une condition !

**Notations**

Pour décrire les opérations que nous étudions, il existe plusieurs notations possibles. Mettons-nous d'accord sur l'une d'entre elles. (En fait je ne vous laisse pas le choix, désolé. :-° )

Pour les deux opérations que nous venons d'effectuer, nous pouvons utiliser cette notation :

pomme\_proj = Projection (pomme, diamètre, couleur)

pomme\_restr = Restriction(pomme, diamètre > 6 cm)

Ces notations ne sont pas officielles, et ne serviront qu'à communiquer entre nous lors du quiz de fin de partie. Pas besoin de les retenir très longtemps ! ;)

Voilà ! C'est plutôt simple pour le moment, n'est-ce-pas ? On continue dans le prochain chapitre !

**En résumé**

* La **projection**, c'est sélectionner certaines colonnes d'une table en supprimant les autres.
* La **restriction**, c'est filtrer certaines lignes d'une table selon une certaine condition.

**Découvrez les opérateurs ensemblistes**

Pour répondre à l'évaluation de fin de partie, il vous faudra en plus de la vidéo, consulter la section *Notations* ci dessous.

Les opérateurs ensemblistes sont des opérations qui portent sur 2 relations de [même schéma](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/decouvrez-le-concept-de-relation#r-4514893).

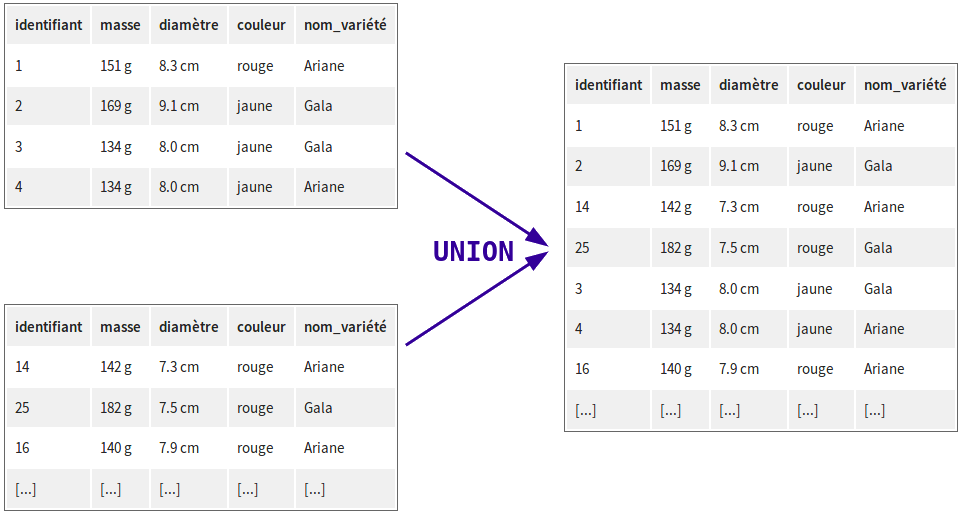
La plus simple d'entre elles porte un nom très poétique : l'**union**.

**L'union**

Si nous avons deux relations qui représentent des pommes, nous pouvons imaginer que chacune de ces deux relations est un panier de pommes. L'**union** des deux paniers, c'est simplement verser leurs contenus respectifs dans un 3ème panier !

Plus formellement :

L'union de deux relations **R1** et **R2** de même schéma produit une troisième relation, également de même schéma, qui contient l'ensemble des tuples de **R1** et de **R2**.



L'union

**La différence**

La **différence**, c'est l'opération inverse de l'union.

 Petite digression algébrique. Écrivons cette équation, qui fait intervenir un opérateur : l'addition.

**3 + 2 = 5**

L'opération inverse de l'addition, c'est l'opérateur qui peut être placé dans cette équation :

**5***[opérateur inverse de l'addition]***2 = 3**

Vous l'aurez deviné, il s'agit de la **soustraction**, car on peut écrire :

**5 - 2 = 3**

Pour l'algèbre relationnelle, c'est le même principe. L'union peut s'écrire :

**R1 union R2 = R3**

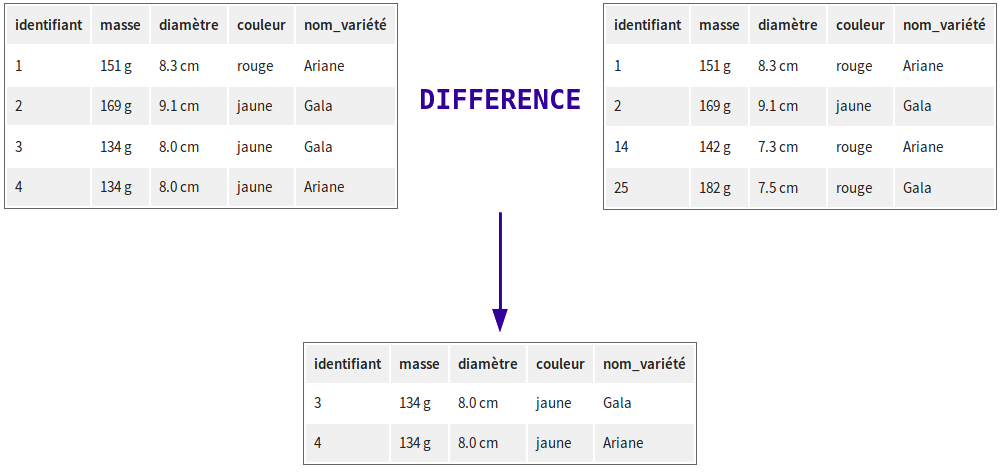
Et la différence, c'est l'opération qui satisfait cette équation :

**R3 différence R2 = R1**

La différence équivaut à enlever des pommes du panier **R3** en les plaçant dans **R2**. Les pommes restantes de **R3** sont ensuite déplacées dans **R1**. On aura ainsi R3 différence R2 = R1.

La différence entre une relation **R3** et **R2** donne une relation **R1** qui contient tous les tuples de **R3** qui n'appartiennent pas à **R2**.

De même que 5 - 2 ne donne pas le même résultat que 2 - 5, la différence entre **R3** et **R2** ne donne pas le même résultat que la différence entre **R2** et **R3**. On dit que cette opération n'est pas *commutative*.



La différence

 Il est tout à fait possible d'effectuer la différence entre R1 et R2 même si R2 contient des tuples non présents dans R1.

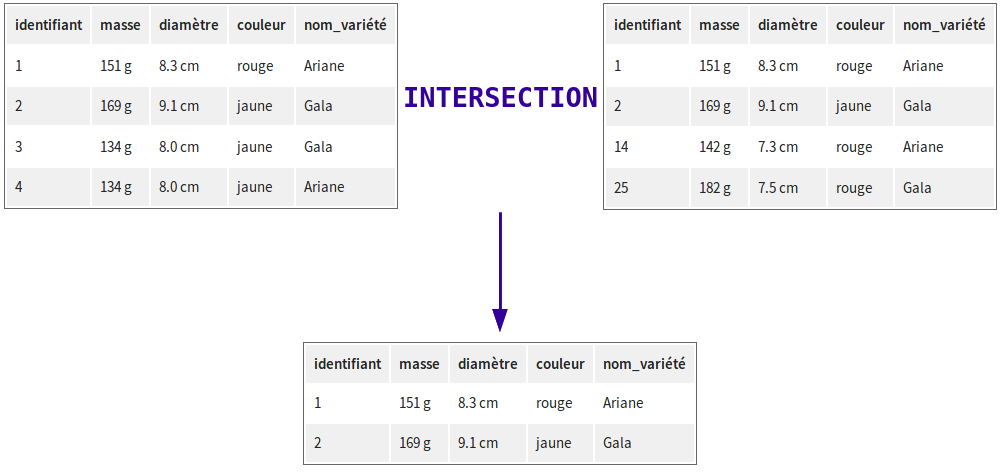
**L'intersection**

Enfin, voici le troisième opérateur : l'**intersection**.

Dans notre exemple avec les paniers de pommes, il nous est difficile d'illustrer l'intersection. Pourquoi ? Parce que dans la vie réelle, une pomme ne peut pas être contenue dans deux paniers à la fois. Or en algèbre relationnelle, deux relations différentes peuvent très bien contenir le même tuple.

Voici sa définition formelle :

L'intersection entre deux relations **R1** et **R2** donne une troisième relation contenant les tuples qui sont présents à la fois dans **R1** et dans **R2**.



L'intersection

L'**intersection** est équivalente à deux **différences** successives. En effet, R1 intersection R2est équivalent à R1 différence (R1 différence R2). Je vous laisse prendre un papier et un crayon pour découvrir pourquoi !

**Notations**

Pour les trois opérations de ce chapitre, je vous propose les notations suivantes :

R1 union R2

R1 différence R2

R1 intersection R2

Ces notations ne sont pas officielles, et ne serviront qu'à communiquer entre nous lors du quiz de fin de partie. Pas besoin de les retenir très longtemps ! ;)

**En résumé**

* L'union de deux tables R1 et R2 contient l'ensemble des lignes de R1 et de R2.
* La différence entre R3 et R2 contient toutes les lignes de R3 qui ne sont pas présentes dans R2.
* L'intersection de R1 et R2 contient les lignes qui sont présentes à la fois dans R1 et dans R2.

**Effectuez un produit cartésien**

N'oubliez pas de regarder les illustrations des tables ci-dessous, elles ne sont pas données dans la vidéo.

Pour répondre à l'évaluation de fin de partie, il vous faudra en plus de la vidéo, consulter la section *Notations* ci-dessous.

**Réfléchissons...**

Vous organisez une séance de dégustation de pommes auprès de personnes qui devront noter le goût de différentes variétés. Miam !

Si je vous dis qu'il y a 3 variétés à faire goûter à 4 personnes, et que je vous demande combien de pommes vous devez prévoir, quelle opération effectuerez-vous ?

3 \* 4 = 12 Il vous faudra au total 12 pommes.

Nous avons ici utilisé l'opération \* : la *multiplication*(aussi appelée "produit").

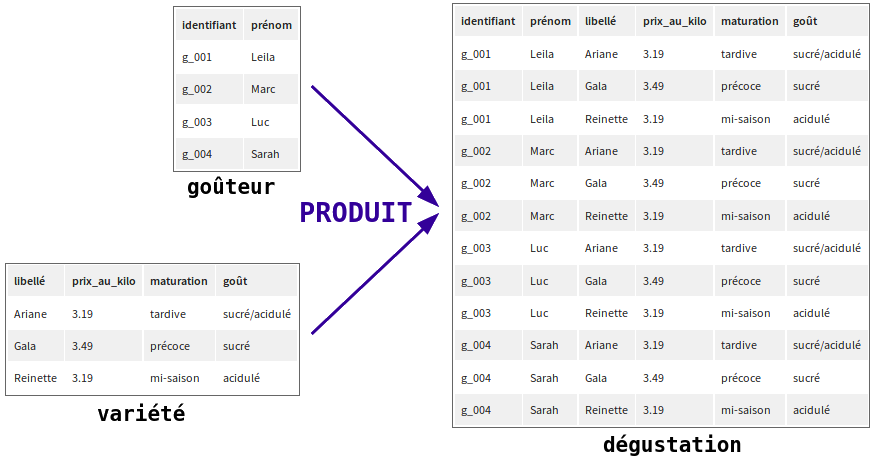
En algèbre relationnelle, le produit (cartésien) a la même signification ! En calculant 3 \* 4, vous avez imaginé **chaque goûteur** essayer **chaque variété**. Vous avez donc créé toutes les combinaisons possibles entre les variétés et les goûteurs.

**Le produit cartésien**

En algèbre relationnelle, le **produit cartésien** entre deux relations **R1** et **R2** est justement composé de toutes les combinaisons possibles entre les tuples de **R1** et les tuples de **R2**.

Ainsi, si nous avons une table représentant les 3 variétés de pommes et une table représentant nos 4 goûteurs, le produit cartésien produira une table :

* qui contiendra les colonnes de R1 ainsi que celles de R2,
* et qui aura comme lignes l'ensemble des pommes que vous donnerez à goûter.



Le produit cartésien

Vous voyez donc que le nombre de lignes d'un produit cartésien entre une relation de**3 tuples** par une relation de 4**tuples** donne justement une relation de**12 tuples** !

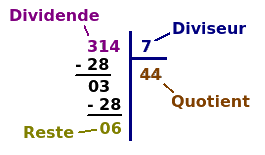
**Aller plus loin : la division**

La division est rarement utilisée lorsque l'on manipule des données. Cette section est donc facultative !

Vous ne serez pas étonné si je vous dis que la **division** est l'opération inverse du **produit**.

En effet, si j'écris a \* b = c, cela est équivalent à écrire c / b = a (à condition que **b** ne soit pas nul).

Cependant, si j'ajoute la contrainte selon laquelle nous ne manipulons que des nombres entiers, la division de A par B A / B ne donne pas toujours un nombre entier. Par exemple, 314 / 7 donne 44.8571. Si nous ne manipulons que des entiers, la division devient une division *euclidienne*. Vous savez, cette opération où vous manipulez*un dividende, un diviseur, un quotient*et*un reste.*



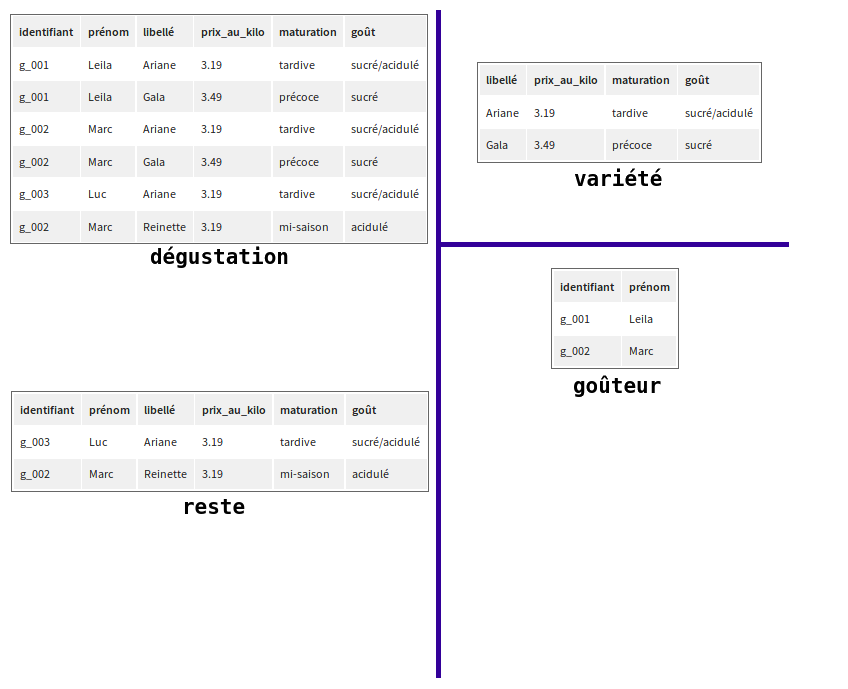
Une belle division euclidienne

La contrainte de ne manipuler que des nombres entiers introduit un reste à notre division, et ainsi314 = 44 \* 7 + 6.

La transposition de ce concept en algèbre relationnelle demande un peu de gymnastique intellectuelle. Voici la définition formelle de la division :

La division d'une relation **R1** par une relation **R2 (**sachant que **R1** et **R2** ont au moins un attribut commun) donne une troisième relation **R3** comportant tous les attributs de **R1** qui n'appartiennent pas à **R2,** et qui contient l'ensemble des tuples qui, assemblés à ceux de **R2** , donnent toujours un tuple de **R1**.

Voilà ce que cela peut donner avec nos goûteurs et nos pommes, en supposant que certains goûteurs ne goûtent pas toutes les variétés de pommes :



 On peut écrire ici que  Dégustation = Variété \* Goûteur + Reste  .

**Notations**

Pour le produit entre une relation R1 et une relation R2, je vous propose la notation suivante :

R1 Produit R2

 Ces notations ne sont pas officielles, et ne serviront qu'à communiquer entre nous lors du quiz de fin de partie. Pas besoin de les retenir très longtemps ! ;)

**En résumé**

* Le **produit cartésien** entre deux relations R1 et R2 est composé de toutes les combinaisons possibles entre les lignes de R1 et les lignes de R2.
* Le nombre de lignes est égal à la multiplication du nombre de lignes de R1 par le nombre de lignes de R2.

**Liez des relations grâce aux jointures**

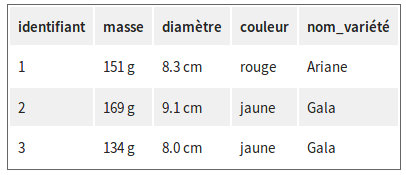
Nous voici arrivés à un concept fondamental de l'algèbre relationnelle, que vous utiliserez très régulièrement : la **jointure**.

Pour répondre à l'évaluation de fin de partie, il vous faudra en plus de la vidéo, consulter la section *Notations* ci-dessous.

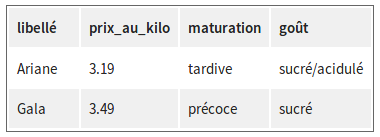
**Réfléchissons...**

Remettons-nous en tête le concept des *clés étrangères*, qui servent à lier des relations. Nous savons qu'elles sont liées, oui, mais nous ne savons pas encore comment *exploiter* ces liaisons. C'est donc ce que nous allons voir ! :D

Reprenons l'exemple de nos deux relations **variété** et **pomme**.



Pomme



Variété

Dans le chapitre [Créez du lien entre vos relations grâce aux clés étrangères](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/creez-du-lien-entre-vos-relations-grace-aux-cles-etrangeres), nous voulions connaître le prix au kilo d'une pomme donnée, et nous avions écrit la phrase suivante :

Si je prends une pomme, et que je veux connaître son prix au kilo, il faudra d'abord que je connaisse sa variété. Ensuite, il faudra que je retrouve dans la table **variété** la ligne correspondante, pour connaître le prix.

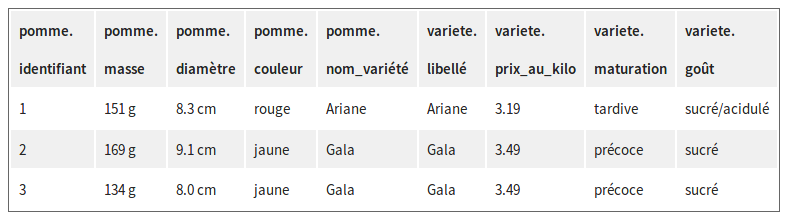
La jointure aura pour but de "coller" les tables **pomme** et **variété**. Elle créera une grande table qui contiendra à la fois les informations de ma pomme et également les informations sur sa variété.

**La jointure interne**

Plus formellement, on dit que l'on effectue une *jointure interne* de la relation **pomme** et de la relation **variété** selon la condition pomme.nom\_variete = variete.libelle.

Et oui, il ne faut pas oublier de spécifier la condition de la jointure. Ici, cette condition dit que c'est la colonne *nom\_variété* de **pomme** qui référence la colonne *libellé* de **variété**.

Voilà le résultat :



Le résultat de la jointure

Nous avons atteint notre objectif. En connaissant l'identifiant de ma pomme, j'ai accès directement à son prix au kilo : il est sur la même ligne !

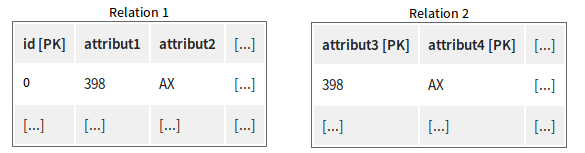
En général, quand on emploie le terme *jointure*, on parle de *jointure interne*.

**Faire une jointure sur plusieurs colonnes**

Vous vous souvenez qu'une clé primaire peut être [composée de plusieurs colonnes](https://openclassrooms.com/courses/4449026/comprenez-limportance-des-cles). Sachez que nous pouvons effectuer une jointure sur plusieurs colonnes (allez, disons 2 colonnes).

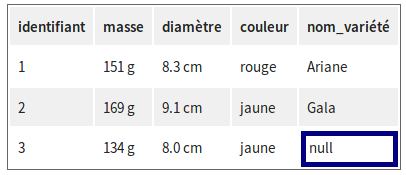
Si c'est le cas, alors une clé étrangère qui référence cette clé primaire sera forcément composée de 2 colonnes. La condition de jointure sera de cette forme :

relation1.attribut1 = relation2.attribut3 AND relation1.attribut2 = relation2.attribut4



**Les jointures externes**

Que se passe-t-il si nous ne connaissons pas la variété de la pomme 3 ? Dans ce cas, la clé étrangère aura une valeur nulle, elle n'aura donc pas de correspondance dans la table **variété**.

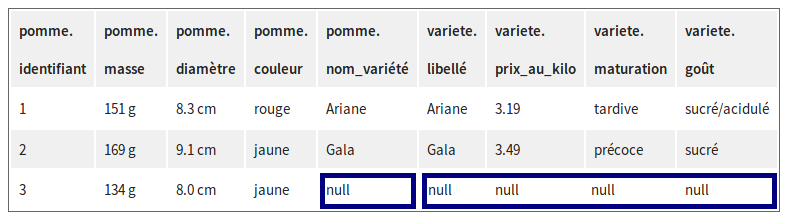


Pomme

La condition de jointure que nous avons vu précédemment ne sera pas satisfaite :pomme.nom\_variete = variete.libelle  car pomme.nom\_variete est nul.

Conséquence ? Comme la condition de jointure n'est pas satisfaite, la pomme 3 ne sera plus présente dans la table finale !

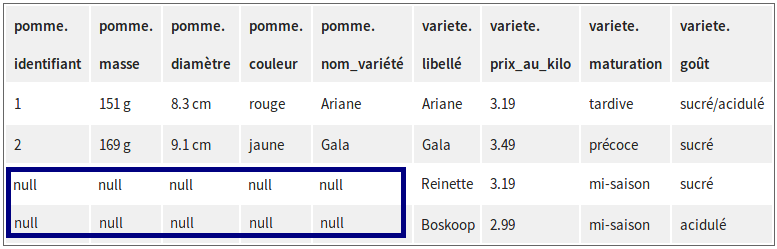
Si nous souhaitons quand même garder les pommes dont on ne connaît pas la variété, il faut faire une *jointure externe*. Si on considère que la table **pomme** est à gauche, et que la table **variété** est à droite, on effectue une *jointure externe à gauche*, car on garde toutes les lignes de la table de gauche :



Résultat d'une jointure externe à gauche

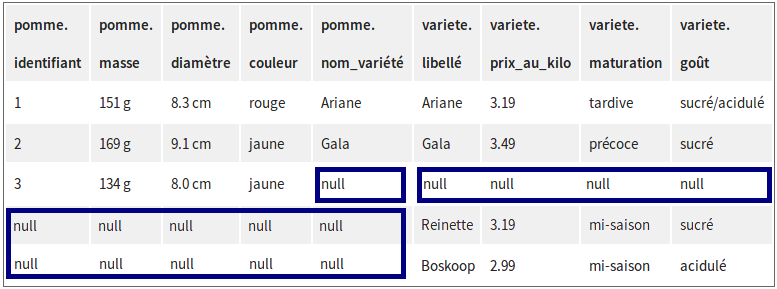
Ici, on a pris l'exemple d'une pomme dont on ne connaît pas la variété (null). Ce problème peut aussi se poser si on connaît la variété de la pomme, mais que cette variété n'est pas répertoriée dans la table **variété,** comme par exemple la variété "Boskoop". C'est d'ailleurs l'exemple illustré dans la vidéo.

De même, s'il n'y avait aucune pomme de la variété Gala, mais que nous souhaitions quand même garder les informations sur cette variété après la jointure, il aurait fallu faire une jointure externe à droite. On aurait donc gardé toutes les lignes de la table de droite (**variété**) même si certaines variétés n'ont pas de correspondance dans la table **pomme**:



Résultat d'une jointure externe à droite

 Et si nous voulons garder à la fois toutes les informations de **pomme** et de **variété**, j'ai en stock un troisième type de jointure externe : la *jointure totale* ! Voici à quoi elle ressemblerait :



Résultat d'une jointure externe totale

**Notations**

Voici la notation que je vous propose pour la jointure interne :

 Jointure (pomme, variété, pomme.nom\_variete = variete.libelle )

Pour les jointures externes droite, gauche et totale :

JointureGauche (pomme, variété, pomme.nom\_variete = variete.libelle )

JointureDroite (pomme, variété, pomme.nom\_variete = variete.libelle )

JointureEntière (pomme, variété, pomme.nom\_variete = variete.libelle )

Ces notations ne sont pas officielles, et ne serviront qu'à communiquer entre nous lors du quiz de fin de partie. Pas besoin de les retenir très longtemps ! ;)

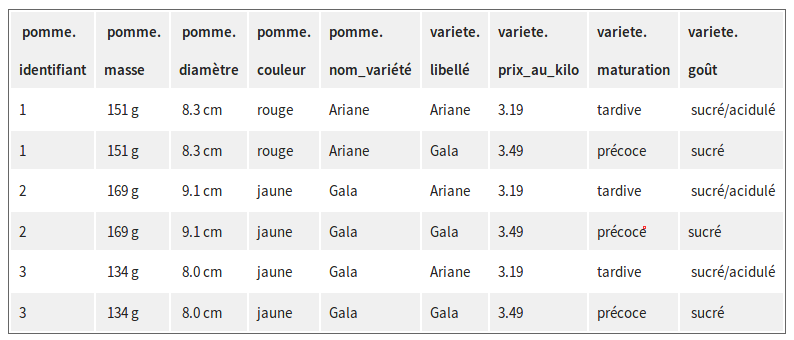
**Aller plus loin : voir la jointure autrement**

Grâce à ce paragraphe facultatif, vous allez encore mieux comprendre pourquoi il faut une condition dans une jointure !

Je vais vous dévoiler un secret : la jointure est en fait l'enchaînement d'un produit cartésien et d'une restriction !

Effectivement, si nous voulons joindre 2 tables, il suffit d'effectuer le produit cartésien de celles-ci, puis de restreindre le résultat selon la condition de jointure.

Voici le produit cartésien de **pomme** par **variété** :



Produit cartésien des relations pomme et variété

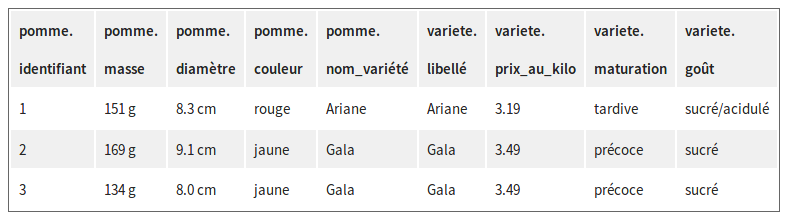
Pour le moment, cela n'a pas vraiment de sens, car on a associé chacune des pommes avec chacune des variétés. C'est totalement inutile car des pommes sont associées à des variétés auxquelles elles n’appartiennent pas. Il y a donc des lignes inutiles car incohérentes.

Cependant, si ensuite nous appliquons une restriction selon la condition pomme.nom\_variete = variete.libelle, nous éliminons les lignes incohérentes :



Lignes ne satisfaisant pas la condition

:magicien: Nous obtenons ici le même résultat que précédemment. C'est magique !



Résultat de la jointure

**Aller plus loin : la jointure naturelle**

Maintenant, un petit "truc" assez pratique : la jointure naturelle.

C'est une jointure classique. Mais si elle intervient sur deux relations dont les colonnes à lier ont exactement le même nom d'une table à l'autre, alors la jointure peut être implicite.

Ainsi, si je renommais la colonne *libellé* de la table **variété** en *nom\_variété*, alors les deux tables auraient ce nom en commun. Comme nous souhaitons effectuer une jointure sur l'ensemble des colonnes ayant le même nom sur les deux relations (ici il n'y en a qu'une seule), nous aurions pu utiliser la **jointure naturelle**.

Avec la jointure naturelle, plus besoin d'indiquer une condition, car elle est implicite, vu que les colonnes ont le même nom !

Donc

Jointure (pomme, variété, pomme.nom\_variété = variété.nom\_variété) est équivalent à

JointureNaturelle (pomme, variété).

**En résumé**

* Une jointure "colle" deux tables selon une certaine **condition**.
* Une **jointure interne** ne garde pas forcément toutes les informations des deux tables jointes.
* Pour garder toutes les informations de l'une des tables (ou des 2 tables), il faut utiliser une **jointure externe**.

## N'oubliez pas l'agrégation !

Oh non, il ne faut surtout pas oublier **l’agrégation** !  Pourquoi ? Parce que l'agrégation n'est pas une opération faisant partie de l'algèbre relationnelle.

Néanmoins, elle est extrêmement importante ! ;)

Ce chapitre est un peu technique, et la vidéo pourra vous aider (mieux que le texte) à y voir plus clair.

### A quoi sert l'agrégation ?

On utilise l'agrégation lorsque l'on veut calculer un résultat qui porte sur plusieurs lignes d'une table. On dit que l'on agrège ces lignes, c'est-à-dire que l'on forme des agrégats pour effectuer sur eux une opération.

Par exemple, si je pose la question suivante...

Quelle est la masse moyenne de chaque couleur de pomme ?

... il s'agit de calculer une valeur pour chaque agrégat :

* **une** valeur (la masse moyenne) qui porte sur **plusieurs** pommes (**toutes** les pommes vertes)
* **la** valeur de la masse moyenne portant sur **les** pommes jaunes
* **la** valeur de la masse moyenne portant sur **les** pommes rouges
* etc.

Vous aurez compris qu'il y aura autant de valeurs de masse moyenne calculées que de couleurs de pommes.

Vous sentez déjà l'importance de l'agrégation, n'est-ce-pas ?

Pour faire une belle agrégation, il faut deux étapes, et donc deux ingrédients :

1. un groupe d'**attributs de partitionnement,**
2. une (ou des)**fonction(s) d'agrégation.**

### Les deux éléments qui constituent l'agrégation

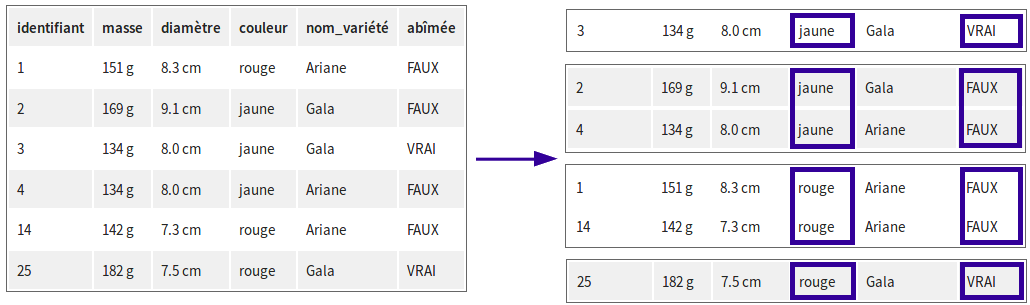
#### Le partitionnement

Si je choisis de partitionner mes pommes par couleur, cela signifie que je veux former des groupes de pommes (on appelle ces groupes des **agrégats**), toutes les pommes d'un même groupe devant avoir la même couleur.

Plus formellement :

Le partitionnement a pour but de créer des groupes de lignes de telle manière à ce que deux lignes se trouvant dans un même groupe aient les mêmes valeurs pour les attributs de partitionnement.

Je peux également partitionner sur plusieurs attributs. Par exemple, disons que j'ajoute à ma table pomme un attribut abîmée, attribut booléen qui vaut VRAI si la pomme est abîmée. Je peux partitionner mes pommes par les attributs couleur et abîmée. Ainsi, au sein d'un même groupe de pommes, la couleur de celles-ci et la valeur de leur attribut abîmée seront identiques :



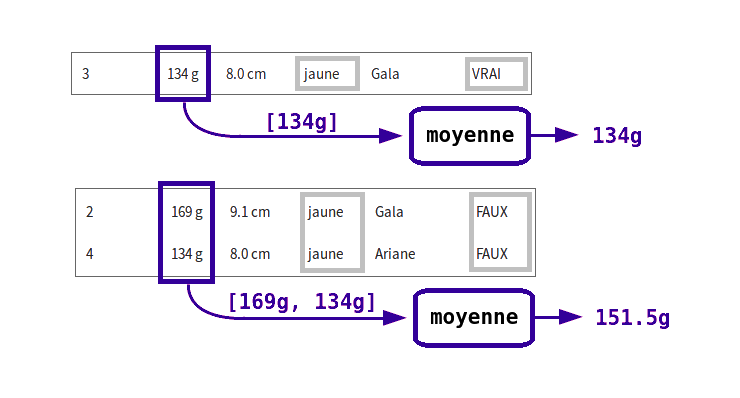
Partitionnement selon les attributs [couleur, abîmée]

#### La fonction d'agrégation

Une fois les agrégats formés, il faut bien en faire quelque chose !

C'est là qu'intervient la fonction d'agrégation. :zorro: Son rôle est de prendre en entrée un groupe de plusieurs lignes, d'effectuer un calcul sur celles-ci, puis de retourner une unique valeur pour chacun des groupes.

Par exemple, on peut appliquer la fonction qui calculera la masse moyenne de chaque groupe de pommes : d'abord le groupe des pommes jaunes abîmées, puis le groupe des pommes jaunes non abîmées, etc.



La fonction d'agrégation MOYENNE qui s'applique à chacun des agrégats

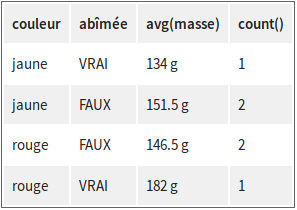
En général, une fonction d'agrégation prend en entrée une **liste** et renvoie une unique valeur. Par exemple, pour calculer le prix moyen au kilo des pommes, la fonction moyenne prendra en entrée la liste des masses des pommes de l'agrégat, et retournera la moyenne de cette liste. C'est ce qui se passe sur l'image ci-dessus.

La liste en entrée correspond à 1 attribut (ici, l'attribut masse). Mais il est également possible de trouver des fonctions d'agrégation qui prennent plusieurs attributs en entrée (donc plusieurs listes) ou parfois même l'ensemble des attributs d'une table.

#### Le résultat

Voici donc le résultat d'une agrégation avec partitionnement selon les attributs couleur et abîmée, en appliquant deux fonctions d'agrégation :

1. la fonction moyenne (en anglais average, notée avg) sur la masse des pommes,
2. la fonction compter (en anglais count), qui compte le nombre de pommes de l'agrégat.



Le résultat de l'agrégation

Grâce à ce résultat, on peut dire que la masse moyenne des 2 pommes jaunes non abîmées est de 151.5 grammes. C'est la classe, non ? :soleil:

Comme la fonction d'agrégation retourne une valeur pour chacun des agrégats, la table qui résultera de l'agrégation aura autant de lignes que d'agrégats.

La nature des objets représentés par une ligne avant l'agrégation (ici : une pomme) ne sera pas la même qu'une ligne après agrégation (ici : un ensemble de pommes).

### Aller (beaucoup) plus loin : le Map Reduce

Dans cette section, on va aller beaucoup plus loin que nécessaire ! On anticipe, pour le jour où vous serez confronté au problème du Big Data, dans lequel les données à traiter sont très volumineuses.

Lorsque les données sont trop volumineuses, on demande souvent à plusieurs ordinateurs (des serveurs) de les traiter en parallèle, pour que cela aille plus vite ! Seulement, il faut bien un serveur qui se charge de coordonner les autres, et de répartir les données de manière intelligente vers les serveurs parallélisés. L'une des manières de faire, c'est d'utiliser la technique du [Map Reduce](https://fr.wikipedia.org/wiki/MapReduce).

En Map Reduce, on découpe l'information en petits morceaux. Chacune de ces parties contient une clé et une valeur. On définit ensuite une fonction appelée reduce. Au moment de l'exécution du calcul, tous les morceaux d'informations qui ont la même clé seront dirigés vers un même serveur parallélisé. Ce serveur applique ensuite la fonction reduce à toutes les valeurs des morceaux qui lui sont parvenues avec la même clé.

Sachez que le Map Reduce fonctionne exactement de la même manière qu'une agrégation, selon cette correspondance :

|  |  |
| --- | --- |
| **Map Reduce** | **Agrégation** |
| Morceau d'information | 1 ligne d'une table |
| Clé | Attributs de partitionnement |
| Valeur | Attributs envoyés à la fonction d'agrégation |
| Fonction Reduce | Fonction d'agrégation |

Jetez à nouveau un coup d’œil sur le [graphique ci-dessus](https://openclassrooms.com/fr/courses/4449026-initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/4568776-noubliez-pas-lagregation#/id/r-4569357) qui explique le partitionnement selon les attributs. La partie droite de ce graphique montre une table fractionnée en plusieurs parties. En Map Reduce, chacune de ces parties est envoyée sur un serveur différent. Ensuite, chaque serveur appliquera la fonction reduce, qui est, par exemple, la fonction moyenne.

C'est aussi simple que cela !

Envie d'aller plus loin dans le Big Data ? Passez voir le parcours [**Data Architect**](https://openclassrooms.com/paths/data-architect). Pour approfondir le Map Reduce, je vous conseille le cours [**Réalisez des calculs distribués sur des données massives**](https://openclassrooms.com/courses/realisez-des-calculs-distribues-sur-des-donnees-massives).

### En résumé

* Les deux composantes de l'agrégation sont :
  + les **attributs de partitionnement**,
  + la/les **fonction(s) d'agrégation**.
* Une fonction d'agrégation basique prend plusieurs valeurs en entrée, et renvoie une unique valeur.
* Le résultat d'une agrégation donne une table avec moins de lignes (autant de lignes que d'agrégats)...
* ... et ces lignes ne représentent pas les mêmes objets.

## Ne perdez pas de vue vos clés ...

Ce chapitre est un chapitre de bonnes pratiques. Il n'y a rien de technique ici, mais juste quelqu'un qui vous fait la morale :ange: et vous met en garde contre quelques pièges dans lesquels ne pas tomber !

Plus précisément, je vais vous expliquer l'importance de bien connaître les clés candidates de vos tables.

Je ne ferai ici pas la distinction entre les termes [**clé primaire et clé candidate**](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/choisissez-votre-cle-primaire-parmi-les-candidates). En gros, je vais vous dire que pour chaque nouvelle table que vous générez (par des jointures, agrégations, etc.), il faudra bien réfléchir aux clés candidates de celle-ci. Mais en réalité, par besoin de réfléchir à toutes les clés candidates de votre table, une seule suffit !

### Pourquoi les clés sont-elles si importantes ?

Connaître au moins une clé candidate par table, c'est très important.

Pourquoi ? Parce qu’elles déterminent la nature des objets que vous représentez dans une table.

Imaginez-vous que quelqu'un vous envoie un fichier CSV contenant ce tableau :

|  |  |  |  |
| --- | --- | --- | --- |
| **id\_pomme** | **propriétaire** | **couleur** | **masse** |
| 1 | Luc | verte | 130 g |
| 2 | Luc | verte | 160 g |
| 3 | Leila | rouge | 134 g |
| [...] | [...] | [...] | [...] |

Vous vous dites "Cette table représente des pommes : 1 ligne = 1 pomme. La clé primaire est donc [id\_pomme]".

Vous décidez donc de calculer la masse totale de toutes ces pommes. Vous trouvez 9.734 kg.

Sauf que... en explorant un peu plus ce tableau vers le bas, vous découvrez ceci :

|  |  |  |  |
| --- | --- | --- | --- |
| **id\_pomme** | **propriétaire** | **couleur** | **masse** |
| [...] | [...] | [...] | [...] |
| 39 | Yari | rouge | 134 g |
| 39 | Doriane | rouge | 134 g |
| 40 | Thanh | jaune | 120 g |
| [...] | [...] | [...] | [...] |

Vous découvrez que la pomme 39 a deux propriétaires : Yari et Doriane.

Vous n'auriez pas supposé une minute qu'une pomme puisse appartenir à 2 personnes en même temps ! Et ça change beaucoup de choses :

* La clé primaire n'est plus [id\_pomme], mais [id\_pomme, propriétaire] car id\_pomme contient des doublons (ici,39 est en double dans id\_pomme).
* La table en votre possession ne représente plus des pommes, mais plutôt des associations pomme-propriétaire (voir le chapitre sur les [tables d'associations](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/utilisez-les-tables-dassociation)). Votre supposition1 ligne = 1 pomme n'est plus vraie !

Ainsi, cette table contient de la redondance, car l'information "la pomme 39 pèse 134 g" est présente sur 2 lignes. Mais vous ne pouvez pas vraiment blâmer l'expéditeur du fichier, car il n'a pas lu [le chapitre sur la redondance](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/evitez-la-redondance) !

Pire encore, le calcul de la masse totale est totalement faussé, car vous avez comptabilisé la masse de la pomme 39 en double (et peut-être pas que la pomme 39 !). A cause de votre erreur de calcul, vous perdez la confiance de la personne qui vous a demandé de calculer la masse totale... Bref, c'est le désastre ! :'(

Conclusion : il fallait vérifier avant tout calcul si [id\_pomme] était une clé candidate. Ici, ce n'était pas le cas. Comment faire ? Descendez jusqu'à la section Aller plus loin. ;)

### Les clés dans les jointures

Bien connaître ses clés, c'est également très important lors des jointures.

En général, on effectue une jointure :

* d'une clé étrangère d'une table A...
* .... vers la clé primaire (ou vers une clé candidate) d'une table B.

Dans ce cas, le nombre de lignes de la table que vous obtiendrez aura :

* autant, ou moins de lignes que A si vous faites une **jointure interne**,
* autant de lignes que A si vous faites une **jointure externe à gauche** (avec A à gauche).

A la fin de votre jointure, pensez bien à compter le nombre de lignes.

Si vous avez plus de lignes que prévu, c’est peut-être que ce que vous pensiez être une clé (primaire ou candidate) sur B n’en est pas une ! Vous risquez de propager cette erreur tout au long de la chaîne de traitement de données !

 J'ai déjà fait cette erreur. Malheureusement, ce n'était pas sur des pommes... Il m'a fallu 1 jour entier pour remonter la chaîne et trouver la source d'un résultat incohérent en bout de chaîne.

En règle générale, dans votre condition de jointure

A.cle\_étrangere = B.cle\_candidate

assurez-vous qu’au moins l’un des 2 termes de part et d’autre du = soit une clé candidate ou primaire.

Si ce n’est pas le cas, ce n'est pas interdit, mais vérifiez bien que vous ne faites pas une erreur de raisonnement. Soyez-sûr que c'est ce dont vous avez vraiment besoin. Surtout, vérifiez bien la table finale en termes de cohérence et surtout en terme de nombre de lignes.

### Aller plus loin : tester si un groupe d'attributs est une clé candidate

#### Comment vérifier si un groupe d'attributs G est une clé candidate ?

Pour le savoir, faites une projection sur G (en gros, supprimez les colonnes qui ne sont pas G), puis regardez s'il y a des doublons.

Dans la table donnée plus haut, la projection sur [id\_pomme] donne la table suivante :

|  |
| --- |
| **id\_pomme** |
| [...] |
| 39 |
| 39 |
| 40 |
| [...] |

 Elle contient au moins un doublon(39) .

S'il y a des doublons, alors votre G n'est pas une clé.

S'il n'y a pas de doublons, alors il y a deux possibilités :

1. Si vous êtes sûrs qu'aucune nouvelle ligne ne sera ajoutée, alors vous pouvez considérer que G est une clé.
2. Si d'autres lignes seront peut-être ajoutées par la suite, il faut prendre contact avec celui (celle) qui a conçu la table, ou regarder dans la documentation qu'il (elle) a écrit, pour savoir si G est une clé, ou pour savoir ce que représente une ligne.

Oui, mais comment vérifier si une table contient des doublons ?

#### Vérifier si une table contient des doublons

Si votre table contient beaucoup de lignes, il est difficile de détecter à l’œil nu les doublons ! Voici donc une recette à implémenter dans votre langage de programmation préféré :

1. Comptez le nombre de lignes de votre table.
2. Supprimez les doublons.
3. Re-comptez le nombre de lignes de votre table.

Si le nombre de lignes de l'étape 1 est différent de l'étape 3, alors votre table contenait des doublons !

Beaucoup de langages d'analyse de données permettent d’éliminer facilement les doublons d'une table :

* En SQL, il faut employer le mot utiliser [**SELECT DISTINCT**](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/lancez-votre-premiere-requete-avec-select-from-et-where).
* En R, il faut utiliser la fonction unique().
* En Python, il faut employer la méthode drop\_duplicates() de la librairie [**Pandas**](https://openclassrooms.com/courses/4449026/parts/4568781/pandas.pydata.org).

### En résumé

* A chaque nouvelle table (que vous générez vous-même, ou que vous découvrez), connaissez au moins une **clé candidate** ! Vous saurez ainsi ce que représente une ligne.
* Il y a une méthode pour vérifier si un groupe d'attributs est une **clé** ou non.
* A la suite d'une **jointure**, vérifiez bien le nombre de lignes obtenues.
* Lorsque vous faites une **jointure** sur autre chose qu'une clé étrangère, faites très attention à ce que vous faites !

# L'algèbre relationnelle

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

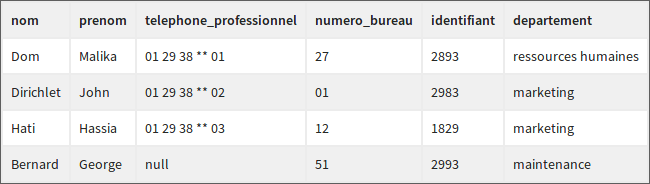
* Transformer une question posée en un enchaînement d’opérations d’algèbre relationnelle
* Effectuer une jointure, une projection, une restriction, une agrégation
* Utiliser les opérateurs ensemblistes de l'algèbre relationnelle

### Description

Testez vos compétences sur le modèle relationnel, plus particulièrement les opérateurs de l'algèbre relationnelle.

### Question 1

**Vous gérez la base de données de votre entreprise. Dans celle-ci se trouve la relation "employés" qui répertorie tous les employés de l'entreprise. Voici les premières lignes de cette table :**

****

**Vous recevez un appel du service des ressources humaines. Au bout du fil, on vous dit : "Notre logiciel ne fonctionne plus, et j'ai besoin rapidement de la liste des noms et prénoms des employés travaillant au département marketing". Quelles opérations allez-vous faire sur la relation employé pour lui fournir cette liste ?**

* + 

Projection (employe, departement = "marketing") puis Restriction (employe, nom, prenom)

* + 

Restriction (employe, nom, prenom) puis Projection (employe, departement = "marketing")

* + 

Restriction (employe, departement = "marketing") puis Projection (employe, nom, prenom)

* + 

Projection (employe, nom, prenom) puis Restriction (employe, departement = "marketing")

*Il faut tout d'abord restreindre les lignes selon la condition departement = "marketing" puis projeter en ne gardant que les colonnes nom et prenom.*

*Attention, la projection ne doit pas intervenir avant la restriction, car après la projection, l'attribut departement n'est plus disponible, il n'est donc plus possible d'appliquer la condition departement = "marketing".*

*Attention à l'ordre des opérations !*

### Question 2

**De nouveaux employés sont entrés en poste ce matin même. Comme le logiciel des ressources humaines ne fonctionne toujours pas, le service des ressources humaines vous envoie un fichier provenant d'un logiciel tableur. Ce fichier contient les nouveaux employés. Vous importez ce fichier dans une table temporaire "nouvel\_employe\_temp" qui a le même schéma que la table "employés". Comment vérifiez-vous si le fichier envoyé ne contient pas d'employés déjà présents dans la base ?**

**Au brouillon, effectuez chacun des 4 ensembles d'opérations proposés ci-dessous.**

**On cherche à répondre à cette question : "Comment vérifiez-vous si le fichier envoyé ne contient pas d'employés déjà présents dans la base ?"**

**Trois des quatre réponses vous permettent de répondre à la question ci dessus : lesquelles ?**

*Attention, plusieurs réponses sont possibles.*

* + 

En vérifiant que employe union nouvel\_employe\_temp contient au moins une ligne.

* + 

En vérifiant que employe intersection nouvel\_employe\_temp ne contient aucune ligne.

* + 

En vérifiant que employe différence (employe différence nouvel\_employe\_temp) ne contient aucune ligne.

* + 

En vérifiant que nouvel\_employe\_temp différence (nouvel\_employe\_temp différence employe) ne contient aucune ligne.

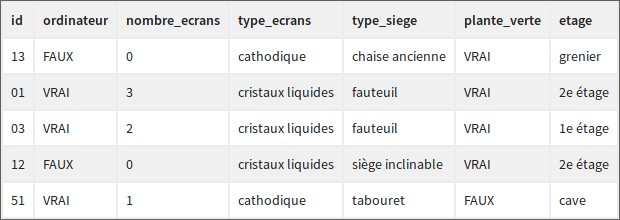
*Il faut ici réaliser une intersection entre employe et nouvel\_employe\_temp.*

*Attention, l'intersection est également l'enchaînement de deux différences. En effet,A intersection B = A différence (A différence B) = B différence (B différence A).*

*Pour vous rafraîchir la mémoire, revoyez le chapitre "Découvrez les opérateurs ensemblistes".*

### Question 3

**Vous avez également accès à des informations sur les bureaux des employés grâce à la table "bureau" dont voici les premières lignes :**

****

**Il est supposé ici que deux écrans placés sur le même bureau sont de même type.**

**Le directeur arrive et vous demande "cela fait un petit bout de temps que nous n'avons plus de nouvelles du stagiaire George Bernard, peux-tu me dire à quel étage il travaille ?"**

**Au brouillon, effectuez chacun des ensembles d'opérations proposés ci-dessous. L'une de ces réponses vous permet de répondre à la question de votre directeur : laquelle ?**

* + 

Restriction(employe, nom = "Bernard" and prenom = "George") Intersection bureau

* + 

Restriction(Jointure(employe, bureau, employe.numero\_bureau = bureau.id), nom = "Bernard" and prenom = "George")

* + 

bureau Intersection Restriction(employe, nom = "Bernard" and prenom = "George" and employe.numero\_bureau = bureau.id)

*On a besoin de données dans les deux tables à la fois. Il faut donc effectuer une jointure (en utilisant la clé étrangère)employe.numero\_bureau = bureau.id. On effectue ensuite une restriction pour trouver la nom de la personne recherchée.*

### Question 4

**Soit une fonction sum qui prend en entrée une liste de nombres et qui retourne la somme de ces nombres.**

**Vous retrouvez dans l'historique des requêtes de la base de données l'opération suivante :**

**Agregation ( bureau, [etage, type\_ecran] , sum(nombre\_ecrans) )**

**Quelle est la question la plus probable que s'est posé l'auteur de cette requête ?**

* + 

Combien y a-t-il d'écrans dans chaque étage ?

* + 

Combien y a-t-il d'étages dans le bâtiment, quels sont les types d'écrans présents, et combien y a-t-il d'écrans au total dans le bâtiment ?

* + 

Quel est le nombre moyen d'écrans dans chaque étage et pour chaque type d'écran ?

* + 

Pour chaque étage, combien d'écrans de chaque type sont présents ?

*Comme on agrège selon les attributs de partitionnement [etage, type\_ecran], cela signifie que l'on veut avoir un résultat pour chaque étage et chaque type d'écran. Pour chaque étage et type d'écran, on fait la somme du nombre d'écrans, ce qui nous donnera le nombre d'écrans total de chaque type pour chaque étage.*

### Question 5

**Quelles sont les deux éléments constituant une agrégation ?**

* + 

 Le partitionnement et la fonction d'agrégation

* + 

Le produit cartésien et la restriction

*Peut-être souhaitez-vous refaire un tour au chapitre "N'oubliez pas l'agrégation !" ?*

### Question 6

**Le produit cartésien entre une relation contenant 38 tuples et une relation contenant 29 tuples ...**

* + 

produit une 3e relation contenant 1102 tuples.

* + 

produit une 3e relation contenant 3829 tuples.

*Rendez-vous au chapitre sur le produit cartésien ;)*

### Question 7

**Vous faites face à une jointure entre une relation A et une relation B, et dont la condition est  A.attr1 = B.attr2  .**

**Mais  attr2  n'est ni une clé primaire, ni une clé candidate pour la relation B.**

**Choisissez le bon comportement :**

* + 

Il n'y a aucun souci, je continue à travailler sans trop réfléchir

* + 

Je fais très attention : je vérifie si cette jointure correspond bien au besoin. Je vérifie le résultat de la jointure en termes de nombre de lignes, et de cohérence de la table obtenue.

*Rendez-vous au chapitre "Ne perdez pas de vue vos clés !"*

## Comprenez les bases de données SQL

Passons maintenant au SQL, un langage qui a été créé pour dialoguer avec des bases de données relationnelles.

### Télécharger la base de données

La base de données que nous utiliserons jusqu'à la fin du cours est en ligne, vous pourrez l'interroger grâce à une console interactive située au bas des prochains chapitres.

Cependant, si vous souhaitez la télécharger, elle se situe [ici](https://s3-eu-west-1.amazonaws.com/static.oc-static.com/prod/courses/files/parcours-data-analyst/Cours_initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/database_sqlite3.zip), au format sqlite3. Vous pouvez l'ouvrir grâce à tout logiciel qui prend en compte ce format, par exemple [SQLiteStudio](https://sqlitestudio.pl/). Pour un petit guide rapide de son utilisation, regardez la vidéo. ;)

### Petite introduction au SQL

Ce langage offre trois modalités :

1. le **LDD** (langage de définition de données) destiné à **créer ou supprimer des objets** dans la base de données (tables, contraintes, etc.)
2. le **LCD** (langage de contrôle de données) qui **gère les utilisateurs d'une base de données ainsi que leurs droits** sur les objets (droit de consultation, modification etc.)
3. le **LMD** (langage de manipulation de données) destiné à **manipuler les données contenues dans les tables**, c'est-à-dire à manipuler les lignes de celles-ci. Les 4 opérations possibles sont :
   1. La **création** de lignes
   2. La **lecture** de lignes
   3. L'**actualisation** de lignes
   4. La **suppression** de lignes

Dans cette partie, nous ne verrons qu'une partie du langage. Nous ne traiterons que le **LMD**, et plus particulièrement la partie lecture des données.

Nous supposons donc que la base de données est déjà construite et déjà remplie. Ce qui nous intéresse ici, c'est d**'interroger ces données** afin de répondre aux questions que nous nous posons.

### Un petit mot sur les SGBDR

Un **Système de Gestion de Base de Données**, c'est un logiciel.

Ce logiciel permet de manipuler des bases de données, au sein desquelles sont stockées des informations.

Lorsque ces bases de données suivent les règles du **modèle relationnel**, alors on les qualifie de bases de données **relationnelles**, et le SGBD qui les manipule devient alors un système de gestion de bases de données relationnelles, ou SGBD**R**. Logique non ?

Et le SQL dans tout cela ?

Si votre SGBD est un SGBD**R**, alors il est fort probable que pour dialoguer avec lui, votre langue de communication privilégiée soit le SQL !

Les SGBDR [les plus connus](https://openclassrooms.com/courses/comprendre-le-web/les-bases-de-donnees-7) sont MySQL (et son petit frère MariaDB), PostgreSQL, Microsoft Access, Oracle database et SQLite.

Si vous n'en avez jamais utilisé, je vous conseille [**SQLite**](https://sqlite.org/). Ce n'est pas le plus performant, mais c'est le plus simple à utiliser, et il suffit amplement à s'initier au langage SQL.

### Prenez vos précautions !

Attention, le langage SQL a évolué au cours du temps. Plusieurs normes ont vu le jour (SQL-1, SQL-2, etc.). De plus, les SGBDR ne sont pas tous en accord total sur la syntaxe du SQL : chacun d'entre eux a quelques variantes.

Le SQL que nous traiterons dans ce cours est généralement compris de tous les SGBDR, mais il est possible que des variations apparaissent. Si l'une des lignes de code de ce cours ne fonctionne pas dans votre SGBDR, je vous invite à effectuer [une petite recherche sur internet](https://openclassrooms.com/courses/comprendre-le-web/les-secrets-bien-gardes-pour-devenir-developpeur-1) !

### En résumé

* Nous ne verrons dans ce cours que le LMD.
* Le SQL permet de dialoguer avec des BDD relationnelles.
* Un SGBDR, c'est un logiciel qui permet de manipuler des bases de données relationnelles.

## Explorez les Panama Papers

Dans les chapitres suivants, nous allons mener une enquête grâce à la base de données des Panama Papers !

### Le contexte

En avril 2016, le journal allemand [Süddeutsche Zeitung](http://www.sueddeutsche.de/) ainsi que le Consortium International des Journalistes d'Investigation ([ICIJ](https://www.icij.org/)) publient des documents confidentiels provenant d'un cabinet d'avocats panaméen.

Cette publication fait grand bruit à travers le monde, car les documents sur lesquels ont enquêté les journalistes du consortium international révèlent des informations sur plus de 214 000 [sociétés offshores](https://fr.wikipedia.org/wiki/Soci%C3%A9t%C3%A9_extraterritoriale) ainsi que le nom des actionnaires de celles-ci. C'est l'affaire des [*Panama Papers*](https://fr.wikipedia.org/wiki/Panama_Papers).

Si l'affaire a fait tant de bruit, c'est parce qu'elle dévoile un système complexe, massif et secret permettant à des entreprises ou à des particuliers de cacher de grosses sommes d'argent sur des comptes bancaires. Ces comptes sont généralement situés dans des pays où la législation est avantageuse, que ce soit en termes de secret bancaire, de taxation, ou de contrôle de la provenance de l'argent. Ce phénomène est appelé [l'évasion fiscale](https://fr.wikipedia.org/wiki/%C3%89vasion_fiscale).

Si ces pratiques sont souvent légales, l'opinion publique les voit en général d'un mauvais œil. En effet, des sommes d'argents générées au sein d'un Etat donné (les bénéfices d'une entreprise par exemple) sont taxées par ce même Etat. Le fruit de cette taxation est redistribué (entre autres) aux services publics dont bénéficie la population du pays en question.

Cependant, l'évasion fiscale consiste à transférer les bénéfices du pays d'origine vers des pays à législation avantageuse (appelés les [**paradis fiscaux**](https://fr.wikipedia.org/wiki/Paradis_fiscal)). Ainsi, les sommes d'argent générées dans un pays donné échappent en partie à l'impôt, et ne bénéficient donc plus aux populations locales. Dans certains pays, le montant estimé de l'évasion fiscale est égal ou supérieur au budget annuel de l'Etat, lorsqu'en parallèle leurs hôpitaux peinent à assurer les soins nécessaires ([**source : ICIJ**](https://www.youtube.com/watch?v=F6XnH_OnpO0)).

Dans les Panama Papers se trouvaient par exemple les noms de plusieurs responsables politiques à travers le monde. Certains d'entre eux ont dû [démissionner](https://fr.wikipedia.org/wiki/Panama_Papers#Liste_compl.C3.A8te_des_personnes_morales_et_physiques_concern.C3.A9es) suite à la pression de l’opinion publique. Mais les Panama Papers ont aussi mis en lumière des moyens de financement de [réseaux criminels ou terroristes](https://www.youtube.com/watch?v=F6XnH_OnpO0).

En France, l'affaire a été rélévée par 2 groupes de journalistes : Premières Lignes Production (qui a réalisé [ce documentaire](https://www.youtube.com/watch?v=L3ZIO-mBxfE) ), et [Le Monde](https://www.lemonde.fr/panama-papers/).

### La base de données

Les Panama Papers sont composés de près de 11,5 millions de documents (emails, courriers, contrats, etc.), pour un volume d'environ 2 Go. De ces documents écrits, l'ICIJ a tenté d'extraire les informations essentielles grâce à des algorithmes. Le résultat de cette extraction a été placé dans une base de données [rendue publique](https://offshoreleaks.icij.org/).

Cette base n'est pas exacte, elle contient par exemple beaucoup de doublons ou de champs erronés.

### Qu'y a-t-il dans notre base de données ?

Les termes en gras sont définis un peu plus bas, dans la section Un peu de vocabulaire.

Grossièrement, la BDD des Panama Papers contient des **sociétés offshores**. Celles-ci sont créées pour des **bénéficiaires** par des **fournisseurs de services offshores**. Des **intermédiaires** se chargent généralement de faire le lien entre les bénéficiaires et les fournisseurs de services offshores.

Il y a 4 tables principales dans la BDD :

* La table **entity**. C'est elle qui contient les sociétés offshores.
* La table **intermediary**, qui contient les intermédiaires.
* La table **address**, qui contient les adresses de certaines sociétés et intermédiaires.
* La table **officer**, contenant entre autres les bénéficiaires des sociétés.

Ces tables contiennent les données publiées par l'ICIJ, auxquelles ont été ajoutées quelques données fictives spécialement pour ce cours, notamment la société Big Data Crunchers Limited. Elle a été créée de toutes pièces pour servir de fil rouge pour ce cours.

Une société peut être domiciliée dans un pays, mais être enregistrée dans un autre. Dans ce cas, cette société répondra à la juridiction dans laquelle elle est enregistrée, même si son adresse officielle n'est pas dans cette juridiction.

Souvent, les termes juridiction et pays sont confondus. En général, les lois sont les mêmes à l'intérieur d'un même pays. Mais parfois, un pays possède plusieurs juridictions : c'est souvent le cas des états fédéraux, dans lesquels chaque état possède des lois différentes. Par exemple, l'état du [Delaware](https://fr.wikipedia.org/wiki/Delaware) aux USA est souvent considéré comme un paradis fiscal, car les lois y sont plus avantageuses pour les sociétés que dans les autres états des USA.

### Notre investigation

Je vous propose de vous mettre dans la peau d'un enquêteur qui enquête sur le financement d'un réseau criminel.

Vous avez au cours de votre enquête intercepté une facture émise par une mystérieuse société qui s'appelle Big Data Crunchers Ltd. Sur cette facture, l'adresse de cette société n'est pas indiquée. Vous ne savez pas qui se cache derrière cette société, mais vous pensez que cette dernière peut être une **société écran**. Une société écran ne se crée pas si facilement que cela. En général, il faut demander de l'aide à des services spécialisés. On les appellera ici des **intermédiaires**.

Vous allez donc enquêter sur cette mystérieuse société, mais aussi sur les intermédiaires qui ont aidé à la créer, car vous pensez qu'il sera peut-être possible d'accuser ces derniers de complicité.

### Un peu de vocabulaire

Qu'est-ce qu'une **société** ?

En économie, une société est la forme juridique la plus répandue des entreprises ; c'est un terme souvent utilisé pour désigner une entreprise. (Source : [Wikipedia](https://fr.wikipedia.org/wiki/Soci%C3%A9t%C3%A9))

Qu'est-ce qu'une **société offshore** ?

Il s'agit d'une société « extraterritoriale ». En pratique, il s'agit d'une société créée dans un pays dans lequel le bénéficiaire économique final n'est pas résident et qui est dirigée hors du pays dans lequel elle est immatriculée. Elles sont souvent utilisées dans des pays où la fiscalité est avantageuse. La société offshore est une forme de **société écran**, qui présente toutes les caractéristiques d'une société réelle (elle est immatriculée par exemple), mais dont l'apparence ne correspond pas à la réalité. (Source : [Le Figaro](http://www.lefigaro.fr/economie/le-scan-eco/decryptage/2016/04/04/29002-20160404ARTFIG00111-panama-papers-qu-est-ce-qu-une-societe-offshore.php))

Qu'est-ce qu'une **société écran** ?

Une société écran est une société fictive, créée pour dissimuler les transactions financières d'une ou de plusieurs autres sociétés. (Source : [Wikipedia](https://fr.wikipedia.org/wiki/Soci%C3%A9t%C3%A9_%C3%A9cran))

Qu'est-ce qu'un **intermédiaire** ?

Un intermédiaire est dans la plupart des cas une personne ou un cabinet d'avocats agissant pour des clients recherchant un fournisseur de services offshores ou demandant la création d'une société offshore. (Source : [ICIJ](https://offshoreleaks.icij.org/pages/about))

 Qu'est-ce qu'un **fournisseur de services offshore** ?

(en anglais : offshore service provider ou agent) C'est une société qui fournit des services dans une juridiction offshore, sur demande d'un client. Ces services peuvent être la création, l'enregistrement ou la gestion de sociétés offshores. (Source : [ICIJ](https://offshoreleaks.icij.org/pages/about))

Qu'est-ce qu'un **bénéficiaire** ?

(en anglais : beneficial owner ou beneficiary) C'est la personne réellement propriétaire de la société. Dans le monde offshore, l'identité du bénéficiaire est souvent gardée secret. (Source : [ICIJ](https://offshoreleaks.icij.org/pages/about))

Les autres termes que vous retrouverez plus tard dans les données sont définis en bas de [**cette page**](https://offshoreleaks.icij.org/pages/about).

### Attention

L'ICIJ tient à préciser à toute personne souhaitant utiliser la base de données les points suivants :

* L'utilisation de sociétés offshores et de trusts n'est pas toujours illégale. Les personnes, sociétés ou autres entités citées dans la base de données n'ont donc pas forcément enfreint la loi ou agi de manière illégitime.
* Beaucoup de personnes ou entités ont des noms similaires. Avant de conclure que deux noms correspondent à la même personne ou entité, il est conseillé de vérifier leurs adresses respectives ou toute autre information pertinente.
* En cas d'erreur dans la base de données, [prendre contact avec l'ICIJ](mailto:data@icij.org).

### En résumé

* La BDD des Panama Papers contient des sociétés offshores. Celles-ci sont créées pour des bénéficiaires par des fournisseurs de services offshores. Parfois, il s'agit de société écran créées pour dissimuler les transactions financières.
* Il y a 4 tables principales dans la BDD : **entity**, **intermediary**, **address** et **officer**.
* Vous allez enquêter sur la mystérieuse société Big Data Crunchers Ltd., ainsi que sur les intermédiaires qui ont aidé à créer cette société.

## Posez vos clés sur la table !

Dans les parties précédentes, vous avez déjà vu les tables et les clés. Nous sommes ici au début d'une nouvelle partie, dans laquelle nous nous concentrons sur l'interrogation d'une base de données en langage SQL.

Mais avant de pouvoir les interroger, il est préférable de savoir créer une **table**, et comment implémenter les **clés**.

### Créer une table avec des clés

Créons ensemble la table **entity**, qui accueillera les sociétés offshores contenues dans les Panama Papers.

CREATE TABLE entity (

id *INTEGER*,

name *TEXT* NOT NULL,

jurisdiction *TEXT*,

jurisdiction\_description *TEXT*,

company\_type *TEXT*,

id\_address *INTEGER*,

incorporation\_date *DATE*,

inactivation\_date *DATE*,

status *TEXT*,

service\_provider *TEXT*,

country\_codes *TEXT*,

countries *TEXT*,

source *TEXT*,

PRIMARY KEY(id),

FOREIGN KEY(id\_address) REFERENCES address(id)

)

C'est assez simple à comprendre : nous utilisons CREATE TABLE, et spécifions ensuite le nom de la table à créer. Nous ouvrons les parenthèses, et spécifions chacune des colonnes en les séparant par des virgules. Pour chaque colonne, nous spécifions son nom et son type.

Remarquez que grâce aux mots clés NOT NULL, nous imposons que la valeur de name soit toujours renseignée lors de l'insertion d'une nouvelle ligne : on ne pourra donc jamais ajouter une société offshore sans spécifier son nom !

Il y a plusieurs types de colonnes possibles. Les plus communs sont les suivants : INTEGER, FLOAT, NUMERIC, VARCHAR, TEXT, TIMESTAMP, DATE, BOOLEAN. Je vous invite à jeter un coup d’œil [**ici**](https://www.w3schools.com/sql/sql_datatypes_general.asp) pour comprendre à quoi ils correspondent !

Nous avons également donné ici les clés étrangères et primaires de la table **entity**grâce aux clauses suivantes :

* PRIMARY KEY
* FOREIGN KEY ... REFERENCES ...

Pour la clé étrangère, ce qui nous avons écrit ici signifie que la colonne id\_address de la table **entity** fait référence à la colonne id de la table **address**.

Nous supposons ici que la table **address** est déjà créée.

### Insérer des données

Maintenant que nous avons créé la structure de la table, insérons une ligne.

INSERT INTO entity (id, name, jurisdiction, jurisdiction\_description, incorporation\_date) VALUES (0, 'Une société', 'IMG', 'Le Pays Imaginaire', '2020-01-01');

Nous spécifions ici la table à remplir grâce à INSERT INTO, puis nous indiquons entre parenthèses les colonnes que nous voulons compléter (id, name, etc.) . Enfin, nous donnons les valeurs à insérer après le mot clé VALUES : ces valeurs doivent être dans le même ordre que les colonnes.

Ici, certaines colonnes seront laissées vides, car nous ne leur avons donné aucune valeur.

Il est aussi possible de remplir une table à partir d'un fichier CSV.  En effet, un fichier CSV se présente exactement de la même manière qu'une table d'une base de données ; si vous ne connaissez pas ce format, je vous invite à jeter un coup d'oeil [ici](https://fr.wikipedia.org/wiki/Comma-separated_values). ;)

Cependant, chaque SGBDR a sa propre manière d'importer des données à partir d'un tel fichier :'(. Certains utilisent une syntaxe SQL, pour d'autres, c'est un peu plus compliqué. Il faudra donc [vous renseigner](https://openclassrooms.com/courses/comprendre-le-web/les-secrets-bien-gardes-pour-devenir-developpeur-1) en fonction du SGBDR que vous utilisez...

### En résumé

* On crée une table avec CREATE TABLE.
* On insère des données grâce à INSERT INTO (...) VALUES (...).

## Exécutez une requête avec SELECT, FROM et WHERE

Voyons voir à quoi ressemble une requête SQL de type SELECT :

SELECT \* FROM entity ;

Vous vous souvenez que le SQL permet de communiquer avec le SGBDR. On communique avec lui grâce à des **requêtes**. Les requêtes qui commencent par SELECT sont en fait des questions que l'on pose. Quand le SGBDR nous répond, on dit qu'il **renvoie** (ou **retourne**) une table.

Lorsque nous **écrivons** puis **exécutons** cette requête, le SGBDR renvoie tout simplement la table **entity**, avec toutes ses lignes et toutes ses colonnes.

En **algèbre relationnelle**, il n'y a pas de **doublons** dans les relations. Mais dans les **bases de données**, il peut y avoir des doublons dans une table (c'est-à-dire de lignes identiques). Si vous souhaitez éliminer les doublons d'un résultat d'une requête SQL, il faut utiliserSELECT DISTINCT au lieu de SELECT. Nous en avions déjà parlé [**précédemment**](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/ne-perdez-pas-de-vue-vos-cles#r-4629016).

SELECT DISTINCT \* FROM entity ;

A partir de cette structure de base, nous allons pouvoir effectuer toutes les opérations de l'algèbre relationnelle que nous avons vues précédemment.

C'est parti ! Commençons avec la projection. :)

### La projection

La projection, c'est l'affaire du mot clé SELECT. C'est lui qui sera chargé de sélectionner les colonnes que vous souhaitez afficher.

Pour le moment, nous avons mis le caractère \* derrière le SELECT. Cette étoile signifie que nous souhaitons obtenir toutes les colonnes disponibles.

Nous pouvons cependant remplacer ce caractère par les colonnes que nous souhaitons. Voici un exemple :

SELECT id, name, status FROM entity ;

### La restriction

Vous vous souvenez que la restriction consiste à ne sélectionner que certaines lignes de la table selon une condition.

Pour faire cela, nous spécifions cette condition grâce au mot clé WHERE.

Commençons notre investigation ! Nous allons rechercher notre mystérieuse société dont le nom est Big Data Crunchers Ltd.

SELECT \* FROM entity WHERE name = 'Big Data Crunchers Ltd.' ;

Cette requête renvoie une table d'une ligne, qui correspond à la société que nous recherchons.

Dans cette requête, nous avons utilisé le signe **=**, qui teste si la variable name et la chaîne'Big Data Crunchers Ltd.' ont la même valeur.

D'autres opérateurs sont possibles en SQL :

* Les opérateurs de comparaison :

|  |  |
| --- | --- |
| **Opérateur** | **Teste si ...** |
| A = B | A égal à B |
| A <> B | A différent de B |
| A > BetA < B | A supérieur à B / A inférieur à B |
| A >= BetA <= B | A supérieur ou égal à B / A inférieur ou égal à B |
| A BETWEEN B AND C | A est compris entre B et C |
| A LIKE 'chaîne de caractères' | (nous verrons cet opérateur dans un prochain chapitre) |
| A IN (B1, B2, B3, etc.) | A est présent dans la liste (B1, B2, etc.) |
| A IS NULL | A n'a pas de valeur |

* Les opérateurs logiques :
  + OR
  + AND
  + NOT

Ils signifient respectivement OU, ET, NON.

Grâce à ces opérateurs, on peut complexifier un peu notre condition :

SELECT \* FROM entity

WHERE (id < 10000004 AND (NOT id < 10000000)) OR (name = 'Big Data Crunchers Ltd.');

Cette requête affichera toutes les sociétés dont l'identifiant est compris entre 10000000 (inclus) et 10000004 (non inclus) ainsi que toutes les sociétés dont le nom est Big Data Crunchers Ltd.. Relisez à tête reposée, c'est de la gymnastique de neurones ! ^^

### Le produit cartésien

Pour effectuer le produit cartésien entre deux tables, il faut simplement spécifier ces deux tables derrière la clause FROM. Rien de plus simple !

SELECT \* FROM entity, address ;

Vous pouvez même spécifier plus de 2 tables ! Le résultat sera ainsi l'ensemble des combinaisons possibles entre toutes les lignes de chacune des tables.

Cette requête peut être coûteuse en ressources (en temps de calcul et en mémoire vive). En effet, le nombre de lignes renvoyé sera égal au nombre de lignes de la table **entity** multiplié par le nombre de lignes de **address**. Prenez garde, ce nombre peut vite exploser !

Peut-on omettre FROM ?

Bonne question !

Une requête sans le mot clé FROM est une requête qui ne se fait sur aucune table. Elle ne renvoie donc qu'une seule ligne. Par exemple, la requête suivante :

SELECT 45, 20, 'bonjour' ;

renvoie une ligne composée de 3 attributs (donc 3 colonnes). La valeur de ceux-ci sera 45, 20 et bonjour.

### Projections : les fonctions scalaires, et le mot clé AS

 Il est possible d'appliquer des **fonctions** sur les colonnes. Par exemple, nous pouvons utiliser la fonction multiplication, en multipliant l'identifiant de toutes les entités par 2 (et pourquoi pas ?).

SELECT id \* 2, name, status FROM entity ;

Il est même possible de combiner des fonctions. Par exemple, calculons la valeur absolue de l'opposé de l'identifiant multiplié par 2 (soyons fous !) :  o_O

SELECT ABS( (- id) \*2 ) AS calcul\_bizarre, name, status FROM entity ;

Ici, j'ai utilisé le mot clé AS. Il permet de renommer une colonne. Sans ce mot clé, la colonne résultat du calcul mathématique aurait comme nom "ABS((-id)\*2)", ce qui n'est pas très compréhensible ! je l'ai donc renommée en "calcul\_bizarre".

Il existe beaucoup de types de fonctions ! Par exemple, des fonctions sur des chaînes de caractères. Si on souhaite obtenir le nom d'une société et ajouter à cette chaîne son statut entre parenthèses, on peut utiliser l'opérateur de concaténation || :

SELECT name || '(' || status || ')' AS name\_and\_status FROM entity ;

-- Version MySQL :

SELECT concat(name,'(',status,')') AS name\_and\_status FROM entity ;

Il y a également des fonctions utilisant des dates, ou des fonctions binaires (qui renvoient TRUE ou FALSE). Voici un exemple :

SELECT CURRENT\_DATE() > incorporation\_date FROM entity ;

Ici, CURRENT\_DATE() renvoie la date actuelle. Celle-ci est ensuite comparée à l'attribut incorporation\_date grâce à l'opérateur >, qui renvoie TRUE(ou  1  selon votre SGBD) si incorporation\_date est antérieure à la date actuelle, FALSE(ou  0  ) sinon.

## Écrivez une union, une différence et une intersection

Nous allons voir comment écrire une **union**, une **différence** et une **intersection**. Plus précisément, nous testerons ces opérateurs binaires sur les deux tables **entity** et **intermediary**.

### Faisons le point sur notre investigation

**Entity** contient une liste de sociétés. **Intermediary** contient des intermédiaires. Dans certains cas, ces intermédiaires sont aussi des sociétés. Il y a peut-être donc des sociétés qui se trouvent à la fois dans les deux tables.

Testons si c'est le cas de Big Data Crunchers Ltd. ! Si c'est le cas, cela signifiera que cette société a servi d'intermédiaire.

Pour vous rafraîchir la mémoire sur la notion d'intermédiaire, c'est [**par ici**](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/explorez-les-panama-papers#r-4552280), ou bien au début de la vidéo ci-dessus.

### L'union, l'intersection et la différence

Pour utiliser un opérateur binaire sur deux tables, il faut que ces deux tables aient le même [schéma](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/decouvrez-le-concept-de-relation#r-4514893), ce qui n'est pas le cas de **entity** et **intermediary**.

Pour résoudre ce problème, nous allons les ramener au même schéma, grâce à une projection. Ainsi, au lieu d'utiliser les tables **entity** et **intermediary** telles quelles, nous prendrons plutôt les tables résultant de ces deux projections, qui elles sont bien de même schéma :

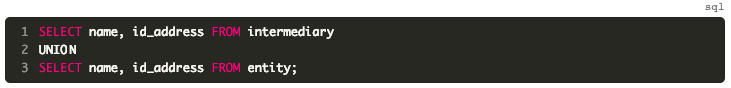
SELECT name, id\_address FROM intermediary ;

SELECT name, id\_address FROM entity ;

Deux sociétés peuvent avoir un même nom. Dans la base de données des Panama Papers, il ne faut pas considérer que deux noms identiques correspondent forcément à la même société. Cependant, si ces deux sociétés ont la même adresse, il est probable qu'il s'agisse d'une même et unique société, même si ce n'est pas une certitude. Nous ferons quand même cette supposition dans ce chapitre.

#### Les mots clés UNION, EXCEPT et INTERSECT

Pour avoir la liste de toutes les sociétés de entity ainsi que la liste de tous les intermédiaires, il faut utiliser le mot clé UNION :



La liste des sociétés de la table **entity** qui ne sont pas des intermédiaires est donnée par :

SELECT name, id\_address FROM entity

EXCEPT

SELECT name, id\_address FROM intermediary ;

Si nous souhaitons connaître les sociétés à la fois présentes dans la table **entity** et dans la table **intermediary** :

SELECT name, id\_address FROM entity

INTERSECT

SELECT name, id\_address FROM intermediary ;

### Oui mais...

Malheureusement, il y a un Mais. :euh: Il faut que je vous avoue que la **différence** et l'**intersection** sont rarement effectuées de cette manière.

D'ailleurs, certains SGBDR n'acceptent même pas les mots clés EXCEPT et INTERSECT ! :waw:

C'est notamment le cas de MySQL, utilisé dans la console interactive du bas de chapitre.

#### Réfléchissons

Par exemple, on veut savoir si une ligne d'une première table est présente dans une seconde table. On doit donc comparer la valeur de chacun de ses attributs un à un. S'il y a beaucoup d'attributs, ceci n'est pas optimal. En fait il y a plus simple, mais à une condition : que les deux tables soient extraites d'une table originelle contenant au moins une clé candidate.

Prenons une table **personne** contenant des personnes de nationalité française, décrites par 20 colonnes. La colonne numéro de sécurité sociale est l'une des clés candidates, car elle identifie chacun des individus. De cette table originelle **personne**, nous créons une table **adulte**, qui contient les personnes de plus de 18 ans, et une table **marseillais** contenant les personnes habitant dans la ville de Marseille.

Pour savoir si une personne est présente à la fois dans **adulte** et dans **marseillais**, pas besoin de comparer la totalité des 20 colonnes : il suffit juste de regarder si son numéro de sécurité sociale est présent dans chacune des deux tables !

#### L'autre méthode

Voici comment faire :

SELECT \*

FROM adulte

WHERE numero\_securite\_sociale IN (

SELECT numero\_securite\_sociale FROM marseillais

) ;

Traduite en français, cette requête serait :

* Sélectionner toutes les lignes de la table **adulte** pour lesquelles le numéro de sécurité sociale se trouve dans :
  + Sélectionner tous les numéros de sécurité sociale de la table **marseillais**

Ceci est une requête imbriquée, car on récupère le résultat d'une requête pour ensuite la traiter au sein d'une requête englobante.

Le code précédent effectue une intersection. Pour effectuer une différence, il suffit de remplacer le mot clé IN par NOT IN.

### En résumé

* L'union s'effectue grâce à UNION.
* L'intersection et la différence s'effectuent respectivement grâce à INTERSECT et EXCEPT.
* Quand on peut utiliser des clés primaires, il y a plus efficace que INTERSECT et EXCEPT !

## Joignez dans tous les sens avec JOIN !

### Le point sur notre investigation

Pour l'instant, nous avons juste retrouvé notre mystérieuse société Big Data Crunchers Ltd. dans la table **entity**. Prochaine étape : retrouver son adresse !

Nous irons même plus loin : nous chercherons également les [intermédiaires](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/explorez-les-panama-papers#r-4552280) qui ont aidé à créer Big Data Crunchers Ltd. Ces intermédiaires peuvent être aussi bien des personnes que des sociétés (banques, cabinets d'avocats, etc.)

### La jointure interne

Notre objectif : "coller" les tables **entity** et **intermediary** pour obtenir une table dans laquelle les lignes contiennent les informations sur les sociétés, avec pour chacune d'entre elles leur adresse. En SQL, il y a deux manières principales d'écrire des jointures :

#### La première méthode : avec FROM et WHERE

Vous vous souvenez du chapitre [Liez des relations grâce aux jointures](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/liez-des-relations-grace-aux-jointures#r-4569305) ? J'y expliquais dans le chapitre Voir la jointure autrement qu'une jointure était en fait l'enchaînement d'un produit cartésien et d'une restriction.

Je commence donc par effectuer le produit cartésien des deux tables...

SELECT \* FROM entity, address ;

... auquel j'ajoute la condition de restriction :

SELECT \* FROM entity, address WHERE entity.id\_address = address.id\_address ;

Voilà, notre jointure est faite !

#### La seconde méthode : avec JOIN et ON

La seconde méthode introduit les nouveaux mots clés JOIN et ON :

SELECT \*

FROM entity

JOIN address ON entity.id\_address = address.id\_address ;

On peut inverser **entity** et **address**, le résultat sera identique :

SELECT \* FROM address JOIN entity ON entity.id\_address = address.id\_address ;

#### Joindre sur plusieurs colonnes

Si votre clé étrangère contient 2 attributs (ou plus), alors il faut  utiliser AND. Prenons l'exemple d'une table t1 ayant une clé étrangère de 2 attributs : fk1 et fk2. Cette clé étrangère référence la table t2, dont la clé primaire est elle aussi composée de 2 attributs : pk1 et pk2 :

-- Première méthode :

SELECT \* FROM t1, t2 WHERE (t1.fk1 = t2.pk1 AND t1.fk2 = t2.pk2);

-- Seconde méthode :

SELECT \* FROM t1 JOIN t2 ON (t1.fk1 = t2.pk1 AND t1.fk2 = t2.pk2);

#### Bien joué !

Nous avons atteint notre objectif : coller nos deux tables. Dans le résultat, il nous faut maintenant retrouver Big Data Crunchers Ltd. Pour cela, une petite restriction s'impose. On l'ajoute dans le WHERE. Comme il y a déjà une condition, on utilise AND :

SELECT \*

FROM entity, address

WHERE entity.id\_address = address.id\_address

AND entity.name = 'Big Data Crunchers Ltd.';

### Joindre avec une table d'association

Retrouvons maintenant les intermédiaires qui ont participé à la création de la société Big Data Crunchers Ltd !

[Précédemment](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/utilisez-les-tables-dassociation), nous avions vu qu'une table d'association servait à modéliser un lien de cardinalité plusieurs à plusieurs entre deux objets. C'est le cas du lien qui unit **entity** et **intermediary**, car plusieurs intermédiaires peuvent créer une société, et un unique intermédiaire peut créer plusieurs sociétés. Ainsi, il y a une table qui porte le nom de **assoc\_inter\_entity**, contenant entre autres :

* une colonne entity, clé étrangère qui référence la table **entity**,
* une colonne intermediary, clé étrangère qui référence la table **intermediary**.

Il nous faut donc faire une jointure sur 3 tables, comme ceci :

SELECT

i.id as intermediary\_id,

i.name as intermediary\_name,

e.id as entity\_id,

e.name as entity\_name,

e.status as entity\_status

FROM

intermediary i,

assoc\_inter\_entity a,

entity e

WHERE

a.entity = e.id

AND a.inter = i.id

AND e.name = 'Big Data Crunchers Ltd.' ;

Nous spécifions les 3 tables dans le FROM. Dans le WHERE, nous donnons les deux conditions de jointure, ainsi que le nom de notre société.

Cette requête est complexe ! :waw: Pour la simplifier, nous avons fait suivre le nom des tables para, e et i dans leFROM. Ce sont des **alias**: ils permettent simplement de donner des surnoms aux tables, pour remplacer le nom de celles-ci par leur alias. C'est plus lisible, non ?

### Aller plus loin : les jointures externes

Vous vous souvenez des [jointures externes](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/liez-des-relations-grace-aux-jointures#r-4569277) ? Elles permettent de garder des lignes qui auraient été ignorées lors d'une jointure interne. Si nous souhaitons obtenir une table avec toutes les sociétés de **entity** avec leur adresse, sans oublier les sociétés dont l'adresse est inconnue, alors il nous faut une **jointure externe à gauche**.

SELECT \*

FROM entity

LEFT OUTER JOIN address ON entity.id\_address = address.id\_address;

Si au contraire, nous voulons garder toutes les adresses (qu'elles correspondent à une société ou non), il faut alors faire une jointure **externe à droite** :

SELECT \*

FROM entity

RIGHT OUTER JOIN address ON entity.id\_address = address.id\_address;

Bien entendu, une jointure externe à **droite** est équivalente à une jointure externe à **gauche** pour laquelle on aurait interverti **entity** et **address**. Certains SGBDR n'acceptent d'ailleurs que l'une de ces deux jointures : c'est alors à vous d'intervertir les tables ! :p

Vous me voyez venir...? Oui ! Il ne me reste plus qu'à vous donner la syntaxe de la jointure **entière** ! Celle-ci nous permet de garder à la fois toutes les adresses et à la fois toutes les sociétés :

SELECT \*

FROM entity

FULL OUTER JOIN address ON entity.id\_address = address.id\_address;

MySQL ne supporte pas la jointure externe entière, mais vous pouvez [**contourner cela**](https://stackoverflow.com/questions/4796872/how-to-do-a-full-outer-join-in-mysql).

### Aller plus loin : la jointure naturelle

Dans le chapitre [Liez des relations grâce aux jointures](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/liez-des-relations-grace-aux-jointures), j’expliquais le concept de la jointure naturelle. Voici comment l'implémenter en SQL :

SELECT \* FROM entity NATURAL JOIN address ;

### En résumé

* Il y a deux méthodes pour effectuer une jointure :
  + avec le FROM et le WHERE
  + avec JOIN etON
* Lorsque nous avons affaire à une table d'association, alors il faut joindre 3 tables entre elles !
* Pour simplifier des requêtes, nous pouvons utiliser des **alias**.

Si vous souhaitez un mémo pour tous les types de jointure, je vous conseille cette [**merveilleuse page Wikipedia**](https://fr.wikipedia.org/wiki/Jointure_(informatique)) !

## Agrégez vos données grâce au GROUP BY

Voyons maintenant l'agrégation !

Vous vous [souvenez](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/noubliez-pas-lagregation) qu'une agrégation est composée de deux éléments :

1. un groupe d'**attributs de partitionnement**
2. une ou plusieurs **fonction(s) d'agrégation**

Nous allons voir ici comment implémenter ces deux éléments en SQL.

### L'agrégation

Bon, ce chapitre est un peu long, afin de bien appréhender les concepts.

Mais l'essentiel se résume en une ligne de code :

SELECT status, count(\*) FROM entity GROUP BY status;

Nous avons placé l'attribut de partitionnement (ici status) derrière le mot clé GROUP BY, et la fonction d'agrégation COUNT()(en français "compter") dans le  SELECT  . Optionnellement, on peut aussi ajouter dans le SELECT les attributs de partitionnement.

Voilà, vous savez tout (ou presque) !

### Réfléchissons...

Testons tout d’abord les fonctions d’agrégation, sans nous soucier des attributs de partitionnement.

#### Les fonctions d'agrégation

Petit rappel : une fonction d’agrégation prend en entrée plusieurs lignes, et retourne une unique ligne.

Par exemple, si nous écrivons :

SELECT max(incorporation\_date) AS maxi FROM entity ;

alors la fonction max prend l'attribut incorporation\_date de plusieurs lignes (toutes les lignes de la table **entity**), et renvoie un attribut (que nous appelons ici maxi) sur une seule ligne. Exécutez cette requête pour bien comprendre. ;)

##### **Entrées et sorties…**

La majorité des fonctions d’agrégation basiques en SQL (celles qui sont déjà prédéfinies par le langage) prennent en entrée un seul attribut et renvoient également un seul attribut. C’est le cas de la fonction max que nous venons de voir.

Il y a cependant une fonction qui échappe à cette règle : la fonction count. La fonction count renvoie un nombre entier : le nombre de lignes qu’elle a reçu en entrée. Pour compter le nombre de lignes, pas besoin de connaître la valeur d’un quelconque argument de ces lignes ! Alors on lui donne simplement le caractère \* :

SELECT count(\*) FROM entity ;

##### **Une petite énigme...**

Si entity contient 1000 lignes, combien de ligne renverrons chacune de ces 2 requêtes ?

SELECT incorporation\_date FROM entity ;

SELECT min(id) FROM entity ;

SELECT incorporation\_date, min(id) FROM entity ;

La première requête renverra 1000 lignes, et la seconde n'en renverra qu'une seule. Cependant, avec la 3e requête, votre SGBDR devrait vous retourner une erreur, car celui-ci ne saura pas s’il faut retourner 1 ou 1000 lignes.

Par contre, la requête suivante fonctionnera, et renverra 1 ligne :

SELECT max(incorporation\_date), min(id) FROM entity ;

En effet, les 2 colonnes que nous demandons résultent toutes deux d’une fonction d’agrégation.

Peut-on en conclure que dans le SELECT, on ne peut faire appel à une fonction d’agrégation uniquement si toutes les autres colonnes résultent elles aussi d’une fonction d’agrégation ?

Oui ! Sauf si votre requête contient un GROUP BY. Dans ce cas, cette phrase de conclusion devra être complétée, c’est ce que nous allons voir dans la partie suivante.

#### Les attributs de partitionnement

Dans une agrégation, les attributs de partitionnement sont à spécifier dans le GROUP BY :

SELECT count(\*) FROM entity GROUP BY status ;

Ici, on compte le nombre de sociétés pour chacun des statuts présents dans **entity**.

Oui, mais cette requête ne renvoie qu'une colonne avec des nombres ! On ne sait pas à quoi correspond chaque nombre.

Effectivement ! Pour résoudre ce problème, on peut afficher la valeur de status de cette manière :

SELECT status, count(\*) FROM entity GROUP BY status ;

... ce qui nous amène à compléter la règle énoncée un peu plus haut, qui est une règle d'or :

#### La règle d'or de l'agrégation

Dans le SELECT, on ne peut faire appel à une fonction d’agrégation uniquement si toutes les autres colonnes :

* résultent elles aussi d’une fonction d’agrégation,
  + OU
* sont présentes dans la clause GROUP BY.

### Continuons notre investigation

Prêts à reprendre la grosse requête du chapitre précédent ? Courage, vous en êtes capables !

SELECT

i.id as intermediary\_id,

i.name as intermediary\_name,

e.id as entity\_id,

e.name as entity\_name,

e.status as entity\_status

FROM

intermediary i,

assoc\_inter\_entity a,

entity e

WHERE

a.entity = e.id

AND a.inter = i.id

AND e.name = 'Big Data Crunchers Ltd.' ;

Vous vous souvenez qu'elle lie des intermédiaires avec des entités. Elle nous a permis de trouver 2 intermédiaires qui ont les identifiants 5000 et 5001.

Nous souhaitons ici connaître le nombre d'entités (le nombre de sociétés) qu'ont créé ces 2 intermédiaires dans chacune des juridictions (juridiction = pays, à peu de choses près ;) ). Il va donc falloir utiliser le GROUP BY !

D'après cette phrase, nous pouvons déduire que les attributs de partitionnement seront :

* la juridiction
* l'intermédiaire (prenons par exemple son identifiant et son nom)

Commençons par modifier la requête précédente. Tout d'abord, cette dernière est limitée à l'entité Big Data Crunchers Ltd.. Ici, nous souhaitons étudier toutes les entités créées par les 2 intermédiaires. Remplaçons donc la ligne 14 par AND (i.id = 5000 OR i.id = 5001)  .

Ajoutons nos 3 attributs de partitionnement dans le GROUP BY, puis supprimons du SELECT toutes les colonnes qui ne sont pas des attributs de partitionnement (c'est-à-dire les lignes 4,5 et 6).

Il ne reste plus que notre fonction d'agrégation : COUNT(), car nous souhaitons compter le nombre d'entités. Voici le résultat :

SELECT

i.id as intermediary\_id,

i.name as intermediary\_name,

e.jurisdiction,

count(\*)

FROM

intermediary i,

assoc\_inter\_entity a,

entity e

WHERE

a.entity = e.id

AND a.inter = i.id

AND (i.id = 5000 OR i.id = 5001)

GROUP BY

i.id, i.name, e.jurisdiction;

### Aller plus loin : fonctions d'agrégation et fonctions scalaires

Nous avons vu dans le chapitre sur [**la clause SELECT**](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/lancez-votre-premiere-requete-avec-select-from-et-where#r-4552922) qu'il est possible utiliser des fonctions dans le SELECT. S’agit-il de fonctions d'agrégation ?

SELECT abs(id) AS valeur\_absolue FROM entity ;

Non. :( Dans cette requête (qui renvoie la valeur absolue de l’identifiant de la société), abs renvoie un résultat sur autant de lignes qu’elle en a reçu en entrée. S’il y a 1000 sociétés dans la table **entity**, alors il y aura 1000 valeurs absolues à calculer. Au contraire, une fonction d’agrégation ne renvoie qu’une seule ligne.

La fonction abs est donc appelée fonction scalaire.

### En résumé

* Dans une agrégation, les fonctions d'agrégation se placent dans le SELECT ...
* ... et les attributs de partitionnement dans le GROUP BY.
* Dans le SELECT, on ne peut faire appel à une fonction d’agrégation uniquement si toutes les autres colonnes résultent elles aussi d’une fonction d’agrégation, OU sont présentes dans la clause GROUP BY.

## Triez vos données avec ORDER BY

En algèbre relationnelle, l'ordre des tuples dans une table n'a pas d'importance ni de signification. Nous touchons ici du doigt l'une des différences entre l'**algèbre relationnelle** et le **langage SQL**, car ce dernier offre la possibilité de trier les lignes d'une table.

### La clause ORDER BY

Voici comment la clause ORDER BY se présente :

SELECT \* FROM entity ORDER BY lifetime ;

Cette requête renvoie l'ensemble des lignes de la table **entity**, et classe les sociétés selon leur attribut lifetime.

Par défaut, le tri se fait dans l'ordre ascendant, c'est-à-dire du plus petit au plus grand. La requête précédente est donc équivalente à celle-ci :

SELECT \* FROM entity ORDER BY incorporation\_date ASC ;

Et pour trier dans l'ordre descendant ?

Il suffit de remplacer ASC par DESC :

SELECT \* FROM entity ORDER BY incorporation\_date DESC ;

#### Trier sur plusieurs colonnes

Parfois, on peut vouloir trier selon plusieurs colonnes. C'est bien entendu possible ! Il suffit de les séparer par une virgule.

### En résumé

* On trie des lignes grâce à ORDER BY.
* L'ordre de tri peut être défini grâce à ASC et DESC.

## Améliorez vos agrégations grâce à HAVING

### Réfléchissons...

Reprenons notre dernière requête, écrite au chapitre précédent :

SELECT

i.id AS intermediary\_id,

i.name AS intermediary\_name,

e.jurisdiction,

e.jurisdiction\_description,

count(\*) as cnt

FROM

intermediary i,

assoc\_inter\_entity a,

entity e

WHERE

a.entity = e.id AND

a.inter = i.id AND

(i.id = 5000 OR i.id = 5001)

GROUP BY

i.id, i.name, e.jurisdiction, e.jurisdiction\_description

ORDER BY

cnt DESC

LIMIT 5;

Imaginons que je ne souhaite avoir que les lignes pour lesquelles le nombre de sociétés est supérieur à 100, car je considère que ce sont les plus importantes.

La première idée qui nous viendrait à l'esprit serait d'effectuer une **restriction**, et donc d'ajouter une condition dans le WHERE.

Malheureusement, ce n'est pas la bonne solution. En effet, les conditions du WHERE sont appliquées sur les lignes avant que l'agrégation ne soit effectuée.

Quand on lance une **agrégation** sur une table, les lignes de la table obtenue ne représentent plus les mêmes objets. La nouvelle table n'est plus de même nature. Il n'est donc pas possible de mélanger dans le WHERE des conditions portant sur la table avant agrégation et des conditions portant sur la table obtenue après agrégation.

Ainsi, pour effectuer une restriction après l'agrégation, on utilise la clause HAVING.

### La clause HAVING

Si nous souhaitons connaître les juridictions dans lesquelles les intermédiaires ont créé plus de 100 sociétés, voici ce qu'il faut écrire :

SELECT

i.id AS intermediary\_id,

i.name AS intermediary\_name,

e.jurisdiction,

e.jurisdiction\_description,

count(\*) as cnt

FROM

intermediary i,

assoc\_inter\_entity a,

entity e

WHERE

a.entity = e.id AND

a.inter = i.id AND

(i.id = 5000 OR i.id = 5001)

GROUP BY

i.id, i.name, e.jurisdiction, e.jurisdiction\_description

HAVING

count(\*) > 100 ;

Notez que dans le HAVING, il est tout naturel d'avoir une fonction d'agrégation, ce qui n'est pas possible dans le WHERE.

Il est tout à fait possible d'écrire HAVING count(\*) > 100 même s'il n'y a pas dans le SELECT une colonne count(\*).

Ce n'est pas bien compliqué, mais c'est un point qu'il faut retenir pour ne pas faire de bêtises !

### En résumé

* La condition dans le WHERE est vérifiée avant l'agrégation.
* Pour restreindre les lignes après l'agrégation, il faut utiliser HAVING.

## Recherchez dans une chaîne de caractères grâce à LIKE

Dans une base de données, il arrive souvent que deux chaînes de caractères différentes puissent décrire la même information.

Par exemple, les chaînes de caractères 'Jean-Michel' et 'Jean Michel' font référence au même prénom, mais elles diffèrent à cause du tiret.

### L'opérateur LIKE

Pour cela, il existe un opérateur très pratique en SQL : c'est LIKE.

Celui-ci permet d'effectuer une recherche dans des chaînes de caractères. Il est utilisé lorsque l'on ne connaît qu'une partie de la chaîne de caractères, ou que l'on connaît sa forme générale.

Ce que j'appelle forme générale d'une chaîne de caractères est souvent appelé pattern dans le monde de la programmation informatique.

On utilise LIKE avec les caractères \_ et %. Le premier est employé pour remplacer un caractère inconnu, le second pour remplacer 0, 1 ou plusieurs caractères inconnus.

Par exemple, pour connaître toutes les sociétés qui commencent par le caractère A, nous pouvons écrire ceci :

SELECT \* FROM entity WHERE name LIKE 'A%' ;

#### Un peu d'entraînement

Voici dans ce tableau différentes expressions utilisant LIKE (colonne de gauche), ainsi que la valeur renvoyée par l'expression en question (colonne de droite).

|  |  |
| --- | --- |
| **Expression** | **Résultat** |
| 'OpenClassrooms' LIKE '%Class%' | TRUE |
| 'OpenClassrooms' LIKE '%Class%ms' | TRUE |
| 'OpenClassrooms' LIKE '%Class%ms%' | TRUE |
| 'OpenClassrooms' LIKE 'Open\_lassrooms' | TRUE |
| 'OpenClassrooms' LIKE 'Open\_\_lassrooms' | FALSE |
| 'OpenClassrooms' LIKE '\_OpenClassrooms' | FALSE |
| 'OpenClassrooms' LIKE 'Op\_\_Cla%' | TRUE |
| 'OpenClassrooms' LIKE '%OpenClas%srooms%' | TRUE |

Si vous souhaitez tester ces expressions, écrivez dans votre SGBDR :

SELECT 'OpenClassrooms' LIKE '%Class%' ;

La réponse sera une table d'une ligne et d'une colonne. En fonction de votre SGBDR, la valeur contenue dans cette table peut varier : TRUE est par exemple équivalent à 1 en SQLite ou t en Postgresql ; et FALSE est équivalent à 0 en SQLite et f en Postgresql.

### Majuscules et minuscules

Que renvoie l'expression 'OpenClassrooms' LIKE 'openclassrooms' ?

Et bien... cela dépend ! Les SGBDR n'ont pas tous le même comportement, certains vous renverront TRUE, d'autres FALSE. :'(

Pour ne jamais vous tromper, je vous conseille donc de vous y prendre comme ceci :

SELECT \* FROM entity WHERE lower(name) LIKE 'a%' ;

De cette manière, quelle que soit sa valeur, name sera converti en minuscules grâce à la fonction lower. Une fois la conversion faite, écrivez votre pattern sans aucune majuscule (ici le pattern est 'a%').

Cette requête vous renverra toutes les sociétés dont le nom commence par un A, qu'il soit en majuscule ou minuscule.

### Les doublons

Dans les Panama Papers, il arrive qu'une société ait plusieurs orthographes différentes.

Par exemple, nous avons trouvé un intermédiaire dont le nom est Pacher Banking S.A.

Recherchons donc s'il existe des doublons :

SELECT \* FROM intermediary WHERE lower(name) LIKE '%pacher%banking%' ;

Oui ! Il y a 2 lignes qui ont un nom similaire. Correspondent-elles à la même société ? Comme elles ont la même adresse, nous considérerons ici que oui (même si ce n'est pas une certitude absolue).

Nous pouvons donc adapter la requête du chapitre précédent, en intégrant l'opérateur LIKE :

SELECT

i.id AS intermediary\_id,

i.name AS intermediary\_name,

e.jurisdiction,

e.jurisdiction\_description,

count(\*) as cnt

FROM

intermediary i,

assoc\_inter\_entity a,

entity e

WHERE

a.entity = e.id AND

a.inter = i.id AND

( lower(i.name) LIKE '%pacher%banking%'

OR

lower(i.name) LIKE '%plouf%financial%services%')

GROUP BY

i.id, i.name, e.jurisdiction, e.jurisdiction\_description

HAVING

count(\*) > 100 ;

### Aller plus loin : gérer les doublons dans les Panama Papers

Conscients des nombreux doublons dans leur base de données, mais assez prudents pour ne pas fusionner à tort deux sociétés de même nom, les concepteurs de la base des Panama Papers ont ajouté des indications supplémentaires pour nous aiguiller.

Ils ont en effet lié certaines lignes des tables **entity**, **intermediary** et **officers** avec des liens de type "same name as" ou "same address as" (respectivement "même nom que", "même adresse que").

Vous pouvez trouver ces liens dans les tables **assoc\_intermediaries**,  **assoc\_entities** et **assoc\_officers**.

### En résumé

* Pour rechercher des chaînes de caractères incomplètes, on utilise  LIKE.
* On spécifie un pattern, où  \_  remplace un caractère, et où  %  remplace plusieurs (ou aucun) caractères.

## Imbriquez des requêtes avec IN, ALL, ANY et EXISTS

Saviez-vous qu'il était possible d'imbriquer des requêtes ? Normalement oui, vous en avez déjà vu passer [subrepticement](http://www.larousse.fr/dictionnaires/francais/subrepticement/75084?q=subrepticement#74231) dans ce cours !

Les mots clés ALL et ANY sont disponibles dans la plupart des SGBDR, mais pas dans la plupart des versions de SQLite. Si vous utilisez ce dernier, testez vos requêtes dans la console interactive ;)

### Réfléchissons...

Prenons un exemple. Dans le [chapitre sur le mot clé HAVING](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/ameliorez-vos-agregations-grace-a-having), j'expliquais que celui-ci servait à effectuer une restriction sur une table après agrégation. Mais si HAVING vous semble compliqué, il est possible d'utiliser une requête imbriquée :

-- REQUETE n°1

SELECT \*

FROM (SELECT id\_address, count(\*) AS cnt FROM entity GROUP BY id\_address) a

WHERE a.cnt > 500 ;

-- REQUETE n°2, équivalente à la requête n°1 :

SELECT id\_address, count(\*) AS cnt

FROM entity

GROUP BY id\_address

HAVING count(\*) > 500 ;

Que retournent ces requêtes ? Elles retournent les adresses auxquelles sont domiciliées plus de 500 sociétés.

Certaines sociétés n'ont pas ou très peu d'employés. Il n'est donc pas impossible de trouver un grand nombre de sociétés enregistrées dans un même bâtiment, même si ce dernier n'est pas un gratte-ciel. Certaines de ces sociétés sont qualifiées de coquilles vides ou de sociétés boîte aux lettres.

### Le mot clé IN

Ces deux requêtes ne renvoient que l'identifiant de l'adresse, qui est un nombre entier. Comment obtenir l'adresse de manière intelligible ?

Nous pourrions faire une jointure avec la table **address**, mais nous pouvons également utiliser le mot clé IN, qui signifie en français dans, en posant une question du type :

Donne-moi les lignes de la table **address** pour lesquelles l'identifiant se trouve **DANS** la colonne id\_address de la table obtenue par la requête n°2.

Nous sommes encore face à une requête imbriquée, qui utilise le résultat de la requête n°2 (en haut de cette page) :

SELECT \*

FROM address a

WHERE a.id\_address IN (

SELECT id\_address

FROM entity

GROUP BY id\_address

HAVING count(\*) > 500

);

### Les mots clés ANY et ALL

Y a-t-il des intermédiaires qui sont plus actifs que les deux intermédiaires que nous avons trouvés précédemment : Pacher Banking S.A. et Plouf Financial Services Corp. ?

D'abord, nous avons créé une table **nb\_entities**, indiquant pour chaque intermédiaire le nombre de sociétés dont il a participé à la création :

|  |  |  |
| --- | --- | --- |
| nb\_entities | | |
| **id\_intermediary** | **intermediary\_name** | **cnt\_entities** |
| 5000 | Pacher Banking S.A. | 2184 |
| 5001 | Plouf Financial Services Corp. | 720 |
| [...] | [...] | [...] |

Pour répondre à cette question, il faut utiliser un opérateur suivi du mot clé ALL ou ANY :

SELECT \* FROM nb\_entities WHERE cnt\_entities > ALL(SELECT cnt\_entities FROM nb\_entities WHERE intermediary\_id IN (5000,5001,5002));

Voici la requête de création de **nb\_entities**. Elle est très similaire à [**celle du chapitre précédent**](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/effectuer-une-recherche-dans-une-chaine-de-caractere#r-4561736).

CREATE TABLE nb\_entities AS

SELECT i.id AS intermediary\_id,

i.name AS intermediary\_name,

e.jurisdiction,

e.jurisdiction\_description,

count(\*) AS cnt\_entities

FROM intermediary i, assoc\_inter\_entity a, entity e

WHERE a.entity = e.id AND a.inter = i.id

GROUP BY i.id, i.name, e.jurisdiction, e.jurisdiction\_description;

### Le mot clé EXISTS

Reprenons la requête que nous avons vu plus haut, dans le paragraphe sur IN, et mettons-nous dans la peau de votre SGBDR, qui devra exécuter cette requête :

SELECT \*

FROM address a

WHERE a.id\_address IN (

SELECT id\_address

FROM entity

GROUP BY id\_address

HAVING count(\*) > 500

);

Pour chaque ligne de la table address, vous êtes censé vérifier si la condition du WHERE est vraie ou fausse, n'est-ce pas ? La condition sera testée autant de fois qu'il y a de lignes.

On est ok là-dessus. Mais ici, dans le WHERE, il y a une sous-requête imbriquée (de la ligne 4 à la ligne 7). Allez-vous recalculer cette sous-requête à chaque fois que vous allez vérifier la condition du WHERE, pour chaque ligne de address ??? Ce serait bête, car le résultat de cette sous-requête est toujours le même : autant garder son résultat en mémoire plutôt que de le recalculer à chaque fois !

La plupart des SGBD le font : ils gardent le résultat dans une mémoire temporaire, appelée un **cache**.

Sauf que... si cette sous-requête dépend de la ligne pour laquelle on est en train de tester le WHERE, il n'est plus possible de mettre le résultat en cache. :( S'il faut recalculer la sous-requête à chaque fois, cela prend du temps !

Si l'on est dans ce cas de figure, il est préférable (du point de vue performance) d'utiliser EXISTS, qui teste si une sous-requête contient au moins une ligne. S'il en trouve au moins une, il arrête l'exécution de la sous-requête, car il est satisfait !

### Aller plus loin : utiliser IN sur plusieurs colonnes

Dans ce chapitre, on a vérifié si une valeur est présente parmi une liste de valeurs.

Il est également possible de vérifier si un couple de deux valeurs (valeur1, valeur2) est présent dans une liste de couples (c1, c2), de cette manière :

SELECT \* FROM t1 WHERE t1.valeur1, t1.valeur2 IN (SELECT c1, c2 FROM t2) ;

### Aller plus loin : les tables temporaires

Théoriquement, il n'y a pas vraiment de limite à l'imbrication de requêtes. Il est possible d'imbriquer des requêtes imbriquées elles-même dans une requête… etc ! :soleil:

Votre SGBDR peut cependant limiter le nombre d'imbrications.

Afin de ne pas avoir des requêtes à rallonge, sachez qu'il est parfois possible (selon votre SGBDR) de créer des tables temporaires. Celles-ci permettent de créer de nouvelles tables pour y stocker le résultat d'une requête, puis de l'utiliser dans une autre requête.

Par exemple, la requête n°1 en haut de cette page est équivalente à ce code :

CREATE TEMP TABLE une\_table\_temporaire AS

SELECT id\_address, count(\*) AS cnt

FROM entity

GROUP BY id\_address ;

SELECT \* FROM une\_table\_temporaire a WHERE a.cnt > 500 ;

Ces tables temporaires seront ensuite effacées à la fermeture de votre session avec le SGBDR.

### En résumé

* Pour tester si une valeur se trouve **dans** le résultat d'une requête, on utilise IN.
* Soit B une liste de valeurs renvoyées par une requête. Pour tester si une **condition** (par exemple A < B) est vraie pour...
  + ... toutes les valeurs de B, on utilise ALL,
  + ... au moins une valeur de B, on utilise ANY.
* EXISTS vérifie si la sous-requête contient au moins une ligne.

## Appréhendez le fenêtrage avec OVER et PARTITION BY

Les fonctions de fenêtrage sont souvent utilisées pour calculer (entre autres) :

* des sommes cumulées,
* des numéros de rang, lorsque l'on classe des lignes.

Vous allez voir, il y a beaucoup de similitudes entre les agrégations et le fenêtrage. N'hésitez-donc pas à vous [rafraîchir la mémoire](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/noubliez-pas-lagregation).

MySQL et SQLite ne gèrent pas les fonctions de fenêtrage. PostgreSQL est ici plus adapté.

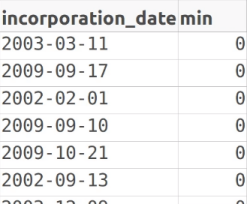
### Réfléchissons...

[**Nous avions vu**](https://openclassrooms.com/courses/initiez-vous-a-lalgebre-relationnelle-avec-le-langage-sql/agregez-vos-donnees-grace-au-group-by#r-4616745) que la requête suivante renvoie à une erreur, car incorporation\_date renvoie autant de lignes qu'en a entity, mais min(id) ne renvoie qu'une seule valeur :

SELECT incorporation\_date, min(id) FROM entity ;

Mettons-y un petit coup de baguette magique... :magicien:

SELECT incorporation\_date, min(id) OVER() FROM entity ;



Tadaaam ! Il n'y a plus d'erreur ! Le fait d'ajouter OVER() a pour effet que la fonction d'agrégation min() renvoie autant de lignes que n'en a **entity**, ce qui n'est pas le comportement classique des fonctions d'agrégation !

Le comportement de la fonction min a ici été modifié par OVER, et tout est une question de nombre de lignes. C'est ce que nous allons voir plus bas.

### Le fenêtrage

Oui oui, il y a bien des similitudes entre les agrégations et le fenêtrage. Tous deux se réalisent en 2 étapes, et seule la seconde étape diffère :

|  | **Agrégation** | **Fenêtrage** |
| --- | --- | --- |
| **Etape 1** | Partitionnement selon les **attributs de partitionnement** | Partitionnement selon les **attributs de partitionnement** |
| **Etape 2** | Application d'une **fonction d'agrégation** | Application...   * d'une **fonction d'agrégation** avec un comportement modifié * OU d'une **fonction de rang**. |
| **Résultat** | Autant de lignes que d'agrégats | Autant de lignes que la table d'origine |

Pour appliquer un fenêtrage, il y a 2 cas différents, qui ont chacun des comportements... différents ! Voici les 2 syntaxes correspondantes :

* OVER (PARTITION BY ...)
* OVER (PARTITION BY ... ORDER BY ...)

Nous allons étudier le fonctionnement de ces deux syntaxes ci-dessous. Mais bon... C'est le dernier chapitre du cours... et vous êtes peut-être fatigués et pressés de finir... :ange: . Alors si vous voulez juste savoir comment faire une somme cumulée et calculer un rang, je vous autorise à passer les 2 sections qui suivent.

### OVER (PARTITION BY...)

Analysons ces deux requêtes, qui utilisent la table **nb\_entities** du chapitre précédent :

-- REQUETE n°1

SELECT sum(cnt\_entities) FROM nb\_entities GROUP BY id\_intermediary ;

-- REQUETE n°2

SELECT sum(cnt\_entities) OVER (PARTITION BY id\_intermediary) FROM nb\_entities ;

La requête 2 fonctionne presque comme l'agrégation de la requête 1 ; mais la seule chose qui diffère, c'est ce que retourne sum.

Dans la requête 2, si sum reçoit en entrée la liste de valeurs [10, 1, 5], elle ne renverra pas 16 mais plutôt [16, 16, 16].

Dans les deux cas, on calcule la somme des sociétés créées par chaque intermédiaire. Mais voici comment le résultat est présenté avec la fonction de fenêtrage :

SELECT id\_intermediary,

jurisdiction,

cnt\_entities,

sum(cnt\_entities) OVER (PARTITION BY id\_intermediary) AS entities\_by\_intermediary

FROM nb\_entities ;

|  |  |  |  |
| --- | --- | --- | --- |
| **id\_intermediary** | **jurisdiction** | **cnt\_entities** | **entities\_by\_intermediary** |
| 4000 | SAM | 10 | 16 |
| 4000 | PMA | 1 | 16 |
| 4000 | CYP | 5 | 16 |
| 4001 | SAM | 3 | 5 |
| 4001 | NEV | 2 | 5 |
| [...] | [...] | [...] | [...] |

### OVER (PARTITION BY... ORDER BY...)

#### Avec une fonction d'agrégation

Lorsque l'on ajoute ORDER BY dans le OVER, alors la fonction d'agrégation se comporte d'une manière encore différente !

En effet, elle considère que les valeurs sont ordonnées. Comme précédemment, la fonction d'agrégation va renvoyer autant de lignes qu'elle n'en reçoit en entrée. Mais voici ce qu'elle va faire :

1. Elle effectue son calcul sur la 1ère valeur, et renvoie son résultat ;
2. Elle effectue son calcul sur les 1ère et 2ème valeurs, et renvoie son résultat ;
3. Elle effectue son calcul sur les 1ère, 2ème et 3ème valeurs, et renvoie son résultat ;
4. Ainsi de suite.

Ainsi, si la fonction sum reçoit [10, 5, 1], elle renverra [10, 15, 16], car

* 10 = 10
* 10 + 5 = 15
* 10 + 5 + 1 = 16

Ici, sum se comporte comme une somme cumulée !

#### Avec une fonction de rang

Si on utilise une fonction de rang (par exemple rank), alors celle-ci va simplement donner le rang des lignes. Ces lignes ont préalablement été triées grâce à ORDER BY.

Petit exemple : nous sommes dans une course où nous avons enregistré le temps mis par chaque coureur pour finir le parcours. Pour savoir qui sera sur le podium, il faut classer ces sportifs selon leur temps :

SELECT

prenom,

temps,

rank() OVER (ORDER BY temps)

FROM course ;

|  |  |  |
| --- | --- | --- |
| **prenom** | **temps** | **rank()** |
| Naïma | 3 min 15 s | 1 |
| Sarah | 3 min 19 s | 3 |
| Sonia | 3 min 16 s | 2 |
| Luc | 10 min 39 s | 4 |

Nous n'avons pas utilisé de PARTITION BY ici, car nous n'avons pas eu besoin de former d'agrégat. C'était juste pour vous montrer ce que fait rank.

### La somme cumulée et les numéros de rang

Nous souhaitons ici connaître les juridictions "préférées" de nos intermédiaires. Celles-ci se comptent en nombre de sociétés créées.

Attribuons donc un rang à chaque juridiction pour chacun des intermédiaires. Appelons notre classement rank. Pour leur attribuer un rang, je les classe par cnt\_entities décroissant. Comme je veux un classement pour chacun des intermédiaires, j'écris PARTITION BY id\_intermediary.

Nous sommes également capables d'effectuer une somme cumulée ! Faisons une somme cumulée de cnt\_entities pour chaque intermédiaire, en triant les lignes par cnt\_entities décroissant. Appelons notre somme cumulée cum\_sum.

SELECT id\_intermediary,

jurisdiction,

cnt\_entities,

rank()

OVER(PARTITION BY id\_intermediary ORDER BY cnt\_entities DESC) AS rank,

sum(cnt\_entities)

OVER(PARTITION BY id\_intermediary ORDER BY cnt\_entities DESC) AS cum\_sum

FROM nb\_entities ;

Voici le résultat :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id\_intermediary** | **jurisdiction** | **cnt\_entitites** | **rank** | **cum\_sum** |
| 4000 | SAM | 10 | 1 | 10 |
| 4000 | CYP | 5 | 2 | 15 |
| 4000 | PMA | 1 | 3 | 16 |
| 4001 | SAM | 3 | 1 | 3 |
| 4001 | NEV | 2 | 2 | 5 |
| [...] | [...] | [...] | [...] | [...] |

### En résumé

* Le fenêtrage est très similaire à une agrégation, sauf qu'il ne modifie pas le nombre de lignes.
* Le mot clé OVER peut être utilisé avec des **fonctions d'agrégation** ou des **fonctions de rang**.
* C'est grâce au mot clé OVER que l'on peut réaliser des **sommes cumulées** ou des **calculs de rang**.