****

**Fonctions** **d’agrégations**

Table des matières

[**Cours**](#_ylaz12tvtk5y) **3**

[Fonctions d’agrégation](#_kiddla7gzxyy) 3

[COUNT / COUNTIF](#_vaov9rayuhwt) 4

[MAX / MIN](#_k3mqzdpu5obg) 5

[SUM / SUMIF](#_qylemj63tccv) 5

[AVG](#_ivldwsjk19jf) 6

[GROUP BY](#_1glrl19ddlou) 6

[HAVING](#_8k3hfnxu138p) 9

[Travail sur les dates](#_jk6j8pihox1z) 9

[Sous-requêtes](#_mcza9pm7hnlv) 10

[**Résumé**](#_1hxc9p19k10t) **12**

[**Pour aller plus loin**](#_f07akaisw0cc) **13**

# Cours

## Fonctions d’agrégation

Dans le cours précédent, nous avons vu comment interroger le data warehouse ou la base de données opérationnelle via le SGBD BigQuery. Nous avons appris à sélectionner, filtrer et trier une table donnée. Néanmoins, pour conduire des analyses, il est difficile de tirer des conclusions lorsqu’on est face à un nombre de lignes important. On a besoin de résumer l’information. C’est quelque chose qu’on fait naturellement au quotidien, par exemple, vous demanderez à votre ami combien il dépense en moyenne quand il fait ses courses au supermarché et non pas de vous donner la liste de l’ensemble de ses achats. Vous seriez d’ailleurs incapable de calculer une moyenne à la volée. Dans le cours d’aujourd’hui, nous verrons qu’il est facile en SQL de faire appel à des fonctions d’agrégation, comme calculer une moyenne, et qu’on peut les appliquer à toute la table ou bien à des sous groupes de cette table.

Pour illustrer les concepts, nous utiliserons l’unique table **purchases** représentée ci-dessous:

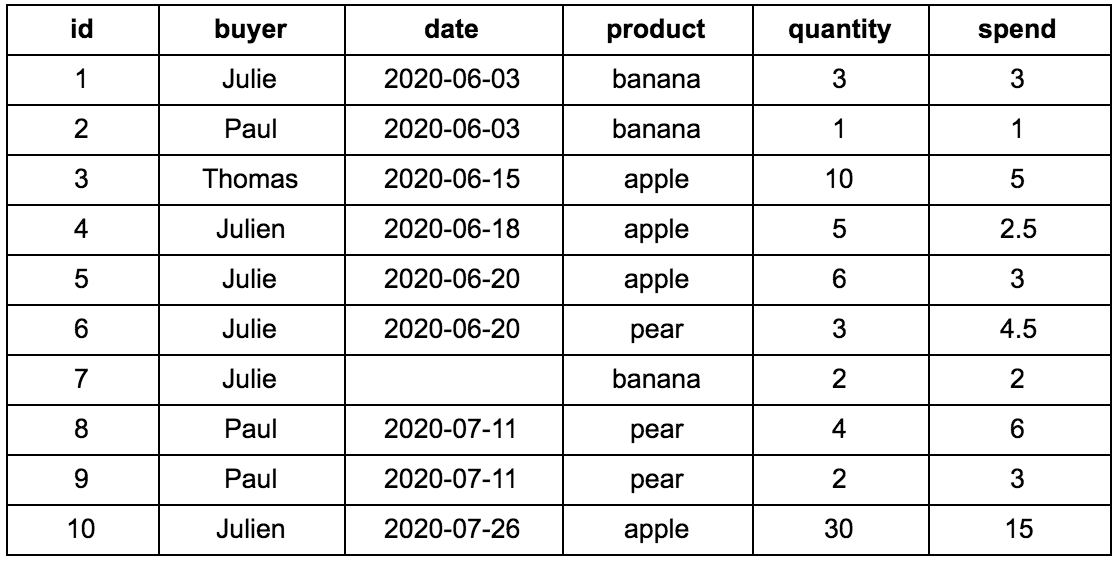


Table *purchases*

Cette table regroupe toutes les ventes effectuées par un primeur (vendeur de fruits et légumes), on retrouve en colonnes l’id permettant d’identifier de manière unique chaque vente, le prénom de l’acheteur, la date de l’achat, le produit acheté, la quantité achetée et le prix dépensé pour ce produit.   
Remarque: Cette table serait mal construite dans l’absolu, car il y a un fort risque d’homonymes. Il faudrait plutôt utiliser un id de l’acheteur et pas simplement son prénom.

### COUNT / COUNTIF

La première fonction d’agrégation que nous verrons est la fonction COUNT. Elle permet comme son nom l’indique en anglais de compter. Elle compte le nombre de lignes **non NULL** de la colonne en question. Prenons par exemple la requête suivante:

| SELECT COUNT(buyer) FROM purchases |
| --- |

Cette requête utilise la fonction d’agrégation COUNT dans la clause SELECT pour la colonne *buyer*. Elle compte simplement le nombre de lignes ayant une valeur non nulle dans la colonne *buyer*. La réponse sera donc 10, car aucune ligne a une valeur manquante. En revanche si on exécute maintenant cette requête:

| SELECT COUNT(date) FROM purchases |
| --- |

La réponse ne sera que de 9, car il y a bien une ligne qui a une valeur manquante, donc NULL, dans la colonne data. Pour compter le nombre de lignes de notre table, on utilisera COUNT mais sur l’ensemble des colonnes. En effet, toutes les colonnes d’une table ne peuvent pas être NULL, sinon ça voudrait dire qu’elle ne contient aucune information et donc elle n’aurait pas été enregistrée. La requête s’écrira:

| SELECT COUNT(\*) FROM purchases |
| --- |

Il existe une variante de cette fonction d’agrégation qui ajoute une condition à ce que vous lui demandez de compter, c’est la fonction COUNTIF. Imaginons que nous voulions compter combien de fois un client a-t-il acheté plus de 3 en quantité de n’importe quel produit. Cela revient à compter le nombre de lignes de la table pour celles qui ont plus de 3 dans la colonne *quantity*. Rien de plus facile à faire:

| SELECT COUNTIF(quantity > 3) FROM purchases |
| --- |

Cette requête renverra la réponse 5.

Compter le nombre de lignes est à la base de beaucoup de calculs en analyse. Mais on a besoin de faire d’autres opérations.

### MAX / MIN

La clause MAX (resp. MIN) permet de récupérer la valeur maximale (resp. minimale) d’une colonne donnée. Si la colonne contient des valeurs manquantes (des valeurs NULL), elles sont ignorées. Par exemple quelle a-été la dépense la plus élevée dans la boutique pour un produit ?

| SELECT MAX(spend) FROM purchases |
| --- |

De même, pour trouver la quantité minimum achetée pour un produit, on utilisera MIN:

| SELECT MIN(quantity) FROM purchases |
| --- |

### SUM / SUMIF

La clause SUM permet quant à elle de sommer les valeurs d’une colonne. De même que pour MAX et MIN, les valeurs NULL sont ignorées et n’affectent donc pas le calcul de cette somme.

J’attire votre attention sur un point important: vous ne pouvez utiliser SUM que sur des colonnes qui ont un type sommable comme INTEGER, FLOAT ... Si vous essayez d’appliquer cette fonction sur des données de type TEXT, une erreur sera levée car SQL comme nous, on ne sait pas ce que ça veut dire de sommer des lettres entre elles. Si nous voulons donc calculer le revenu fait par la boutique, il nous suffit de sommer la colonne spend comme cela:

| SELECT SUM(spend) FROM purchases |
| --- |

De la même manière que pour la fonction COUNT, la fonction SUM a son son homologue SUMIF.

### AVG

La clause AVG permet de calculer la moyenne d’une colonne. Elle ignore les valeurs NULL et nous permet de gagner du temps car, en temps normal, pour calculer la moyenne, nous aurions dû combiner deux fonctions d’agrégation pour retrouver le résultat de celle-ci. En effet, si on souhaite calculer quelle est la dépense moyenne d’un acheteur pour un produit, on aurait pu écrire la requête suivante:

| SELECT SUM(spend) / COUNT(spend) AS avg\_product\_spend FROM purchases |
| --- |

Elle va sommer toutes les dépenses puis la diviser (le signe /) par le nombre de lignes, c’est à dire le nombre d’achats de produits. Ce qui nous donnera bien le résultat escompté. Notez que nous avons donné un alias pour renommer la colonne de résultat en un nom un peu plus intelligible.

Cette même requête pourrait être écrite plus simplement en utilisant la fonction d’agrégation AVG comme suit:

| SELECT AVG(spend) AS avg\_product\_spend FROM purchases |
| --- |

Il existe bien évidemment beaucoup d’autres fonctions d’agrégations mais celles-ci permettent de faire la plupart des demandes d’analyse.

Les fonctions d’agrégation peuvent aussi être utilisées sur des sous groupes de la table, ce qui peut s’avérer être utile quand vous voulez calculer une statistique par rapport à une dimension. Vous verrez une nouvelle fois qu’Excel est l’héritage du SQL, car c’est exactement ce que fait un tableau croisé dynamique.

## GROUP BY

La clause GROUP BY permet d’indiquer à la base de données que vous voulez qu’elle découpe la table en sous groupes (ou sous table) avant d’appliquer les fonctions d’agrégations choisies. Prenons l’exemple de la somme des dépenses par acheteur dans la boutique avec cette requête:

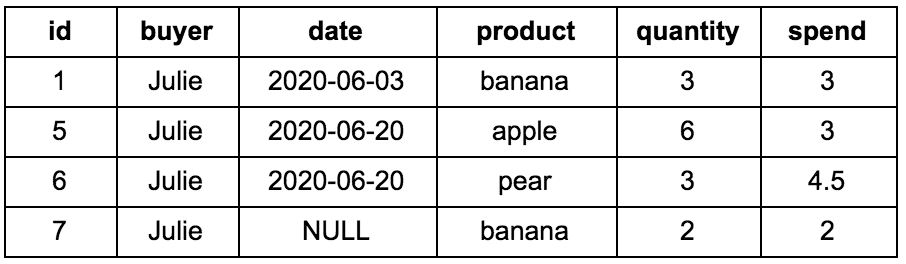
| SELECT buyer, SUM(spend) AS total\_spend FROM purchases GROUP BY buyer |
| --- |

Avant d’explorer ce qu’il y a dans la clause SELECT, intéressons-nous à l’écriture de la clause GROUP BY. On voit ici la ligne:

| GROUP BY buyer |
| --- |

Elle indique à la base de donnée que quelque soit ce qu’on demandera après, on souhaite qu’elle le fasse **par buyer**. Concrètement, ce que la base va faire ici est de découper en interne notre table autant de fois que d’acheteurs uniques (remarque si nous avions eu des NULL dans la colonne buyer, il l’aurait considéré comme un acheteur également):

* D’abord il compte le nombre de valeurs uniques, ici il y a 4 acheteurs uniques: Julie, Paul, Thomas, Julien.
* Pour chacun de ces acheteurs, il va filtrer la table sur les lignes concernées (c’est-à-dire les achats de la personne en question). Par exemple, pour Julie la sous-table sera:

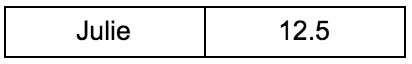


Sur chacune de ces sous tables, il faut maintenant lui appliquer ce qu’on veut sélectionner, c’est à dire ce qu’il y a dans la clause SELECT. Pour rappel:

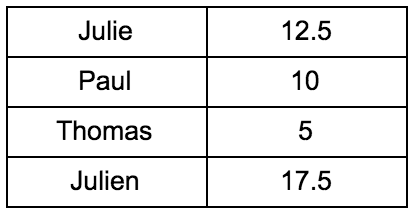
| SELECT buyer, SUM(spend) AS total\_spend |
| --- |

On voit qu’il y a deux valeurs qu’on souhaite retourner. La première est l’acheteur, pas de doute possible, il est unique par sous table donc ici Julie. La seconde n’est rien d’autre que ce que vous connaissez déjà, la somme de la colonne spend qui est de 3 + 3 + 4.5 + 2 = 12.5.

Et donc, l’information pour Julie est résumée par le résultat:



On applique le même procédé aux valeurs uniques de buyer restants, à savoir Paul, Thomas, Julien et on obtiendra en résultat final:



On a bien le résultat voulu. Vous comprenez alors pourquoi si nous avions pas mis de fonction d’agrégation autour de la colonne spend, une erreur serait apparue ! En effet, vous ne sauriez pas non plus quoi faire avec chaque colonne spend de chacune de vos sous tables !

Plusieurs remarques importantes concernant cette clause:

* Est-ce que buyer était indispensable dans le SELECT de la requête ? Pas du tout, en revanche si vous ne la mettez pas, la requête renverra les sommes dépensées uniquement ! Vous serez donc dans l’incapacité de faire le lien entre ces sommes et les acheteurs. **C’est pourquoi une règle très importante est de mettre dans le SELECT toutes les colonnes que vous utilisez dans la clause GROUP BY et d’utiliser des fonctions d’agrégation sur le reste des colonnes !**
* Vous pouvez grouper par une ou plusieurs colonnes. Si par exemple vous groupez par buyer et product, il va considérer toutes les valeurs uniques de ces deux colonnes prises ensemble et pour chacune d’elles créer une sous table et enfin appliquer la ou les fonctions d’agrégation de la clause SELECT.
* L’ordre des clauses est fondamental, le GROUP BY vient s’insérer après le WHERE (s’il y en a un) sinon après le FROM.

Lorsque la requête contient à la fois une clause WHERE et une clause GROUP BY, la clause WHERE est d’abord exécutée en premier. On va venir filtrer les lignes de la table selon les conditions. Puis on applique le GROUP BY et on résume l’information avec l’aide des fonctions d’agrégation.

Cas particulier du GROUP BY sans fonction d’agrégation:

| SELECT buyer FROM purchases GROUP BY buyer |
| --- |

Vous trouverez peut-être cette écriture étrange, mais à vrai dire si vous reprenez les étapes du fonctionnement du GROUP BY, vous vous apercevrez que cette requête retourne donc les valeurs uniques des acheteurs. Cette requête supprime les doublons et est équivalente à la clause DISTINCT que vous aviez vu pendant les exercices précédents.

En combinant la clause GROUP BY et les fonctions d’agrégation, vous avez à votre disposition des outils puissants pour l’analyse de données. Les combiner astucieusement vous permettra de tirer des conclusions sur vos données d’entreprises.

Intéressons-nous maintenant au filtrage des résultats du GROUP BY avec la clause HAVING

### HAVING

La clause HAVING filtre les résultats en sortie du GROUP BY, c’est pourquoi on parle d’un filtrage **à posteriori** contrairement au WHERE qui fait un filtrage **à priori**. Reprenons la requête précédente pour la complexifier un peu, on cherche à avoir tous les acheteurs qui ont dépensé au moins 10 euros dans la boutique. Il nous suffit de reprendre la requête et d’ajouter cette nouvelle condition:

| SELECT buyer, SUM(spend) AS total\_spend FROM purchases GROUP BY buyer HAVING total\_spend >= 10 |
| --- |

Notez bien l’ordre qui est essentiel, le HAVING vient toujours après le GROUP BY.

## Travail sur les dates

Savoir manipuler les dates en SQL est essentiel car cela permet d’observer des tendances dans le temps. Devoir gérer soit même le calcul des numéros de semaine de l’année, les jours de la semaine, le tout en prenant en compte les années bissextiles n’est pas une chose simple à faire. Heureusement pour nous, ce travail a déjà été fait et est disponible via des fonctions. Ces fonctions ne sont pas des fonctions d’agrégation comme vues précédemment, mais elles permettent de transformer/extraire des informations d’un champs date.

Dans BigQuery, les dates sont représentées par défaut avec le pattern “AAAA-MM-JJ” c’est à dire d’abord l’année, puis le mois et enfin le jour séparés par des tirets. Si une colonne est bien reconnue comme une date, alors on peut lui appliquer une série d’actions qu’on retrouve [ici](https://cloud.google.com/bigquery/docs/reference/standard-sql/date_functions#extract) par l’intermédiaire de la fonction EXTRACT. Prenons l’exemple suivant:

| SELECT EXTRACT(month FROM date) AS month, SUM(spend) AS total\_spend FROM purchases GROUP BY 1 |
| --- |

Ici, on vient extraire le mois de la colonne *date* et on renomme cette colonne en sortie en *month*. Puis on calcule le total des ventes associé à chacun de ces mois grâce à la clause GROUP BY. On vient ici de calculer le chiffre d’affaire par mois de la boutique.

Une autre fonction est également très utile pour extraire l’année, le mois et le jour sous la forme “AAAA-MM-JJ”. C’est la fonction DATE. Elle est particulièrement adaptée si la colonne est de type TIMESTAMP et qu’on souhaite enlever le détail des minutes, secondes …

## Sous-requêtes

Il est souvent utile en SQL de combiner des requêtes, vous en avez vu un exemple dans les exercices du cours précédent lors de l’identification des utilisateurs français n’utilisant par leur langue natale sur leur téléphone. Il existe un mécanisme rendant les requêtes plus lisibles pour créer ce qu’on appelle des tables temporaires. Les tables temporaires n’existent qu’en mémoire et ne sont en rien des nouvelles tables créées dans la base de données.

Pour utiliser ce mécanisme, il faut faire appel à la clause WITH AS. Prenons un exemple, si nous voulions savoir combien il y a d’acheteurs ayant acheté au total plus de 10 en quantité ? Nous pourrions écrire la requête comme cela:

| WITH buyers\_quantities\_over\_10 AS (  SELECT buyer, SUM(quantity) AS total\_quantity  FROM purchases  GROUP BY buyer  HAVING total\_quantity > 10 ) SELECT COUNT(buyer) FROM buyers\_quantities\_over\_10 |
| --- |

La première partie de la requête utilisant le WITH AS permet d’obtenir la liste des acheteurs ayant acheté plus de 10 en quantité. Remarquez bien l’utilisation de la fonction d’agrégation SUM dans le SELECT.

La seconde partie de la requête interroge les résultats de la première en rappelant dans le FROM le nom de la table temporaire que nous avions choisi lors de l’écriture de la clause WITH AS, à savoir *buyers\_quantities\_over\_10*. Puis elle compte simplement le nombre d’acheteurs.

Vous voyez à quel point vous pouvez maintenant envisager des requêtes beaucoup plus complexes ! Vous allez pouvoir vous entraîner avec les exercices et commencer à vous faire vos premières analyses !

# Résumé

* Les fonctions d’agrégation permettent de résumer un ensemble de lignes par une statistique descriptive
* On retrouve parmi ces fonctions:
  + COUNT: permet de compter le nombre de lignes non NULL
  + COUNTIF: permet de compter le nombre de lignes non NULL avec une condition
  + SUM: permet de sommer une colonne
  + SUMIF: permet de sommer une colonne avec une condition
  + MAX / MIN: permettent de calculer respectivement le maximum et le minimum
  + AVG: permet de calculer la moyenne
* La clause GROUP BY permet d’appliquer ces fonctions d’agrégations en découpant la table de départ par dimension en sous tables pour mener des analyses
* La clause HAVING permet de filtrer les résultats de sortie d’un GROUP BY selon une ou plusieurs conditions. On parlera d’un filtrage à posteriori contrairement au WHERE qui est un filtrage à priori
* L’ordre des clauses est fondamental:
  + SELECT
  + FROM
  + WHERE
  + GROUP BY
  + HAVING
  + ORDER BY

Si celui-ci n’est pas respecté, une erreur sera levée

* La manipulation des dates se fait par la fonction EXTRACT et ne s’applique qu’aux colonnes de type DATETIME:

EXTRACT( unité FROM nom\_de\_la\_colonne\_date), unité peut valoir YEAR, MONTH, WEEK …

* La fonction DATE permet quant à elle d’extraire directement l’année, le mois et le jour sous la forme “AAAA-MM-JJ”. Elle s’écrit simplement:

DATE(nom\_de\_la\_colonne\_date)

* La combinaison de plusieurs requêtes peut s’avérer indispensable pour certaines analyses. C’est pourquoi on utilise la clause WITH AS comme cela:

WITH nom\_table\_temporaire AS (la requête à écrire ici)

SELECT …

FROM nom\_table\_temporaire

# Pour aller plus loin

Cours en ligne sur le SQL, chapitres concernant l’agrégation

<https://www.w3schools.com/sql/default.asp>

Les éléments de date qui peuvent être utilisés dans la fonction PARSE\_DATE:

https://cloud.google.com/bigquery/docs/reference/standard-sql/date\_functions?hl=fr#supported\_format\_elements\_for\_date