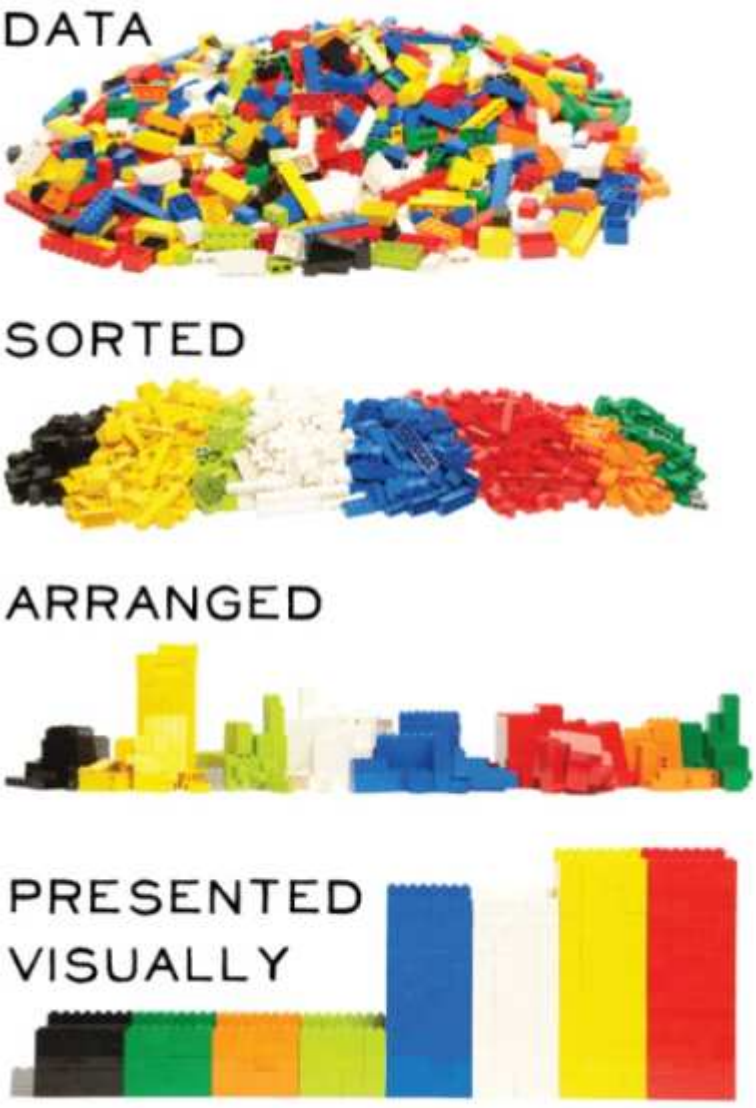
Préparer ses données avec R et le Tidyverse

Groupe des référents R

2018-05-15

Chapitre 1 Introduction



Ce module va vous permettre de découvrir un ensemble de méthodes sous R afin de préparer ses données. Préparer ses données sous R, cela veut dire :

- Savoir les importer dans un environnement R

- Mettre ses données dans de bons formats (date, catégorielle) et gérer les données manquantes
- Rajouter des variables en fonction de variables existantes
- Regrouper des modalités de variables
- Joindre des tables entre elles pour obtenir des informations de plusieurs sources
- Aggréger des données
- Bien définir notre table de travail en fonction des indicateurs à analyser et à leurs dimensions d'analyse …

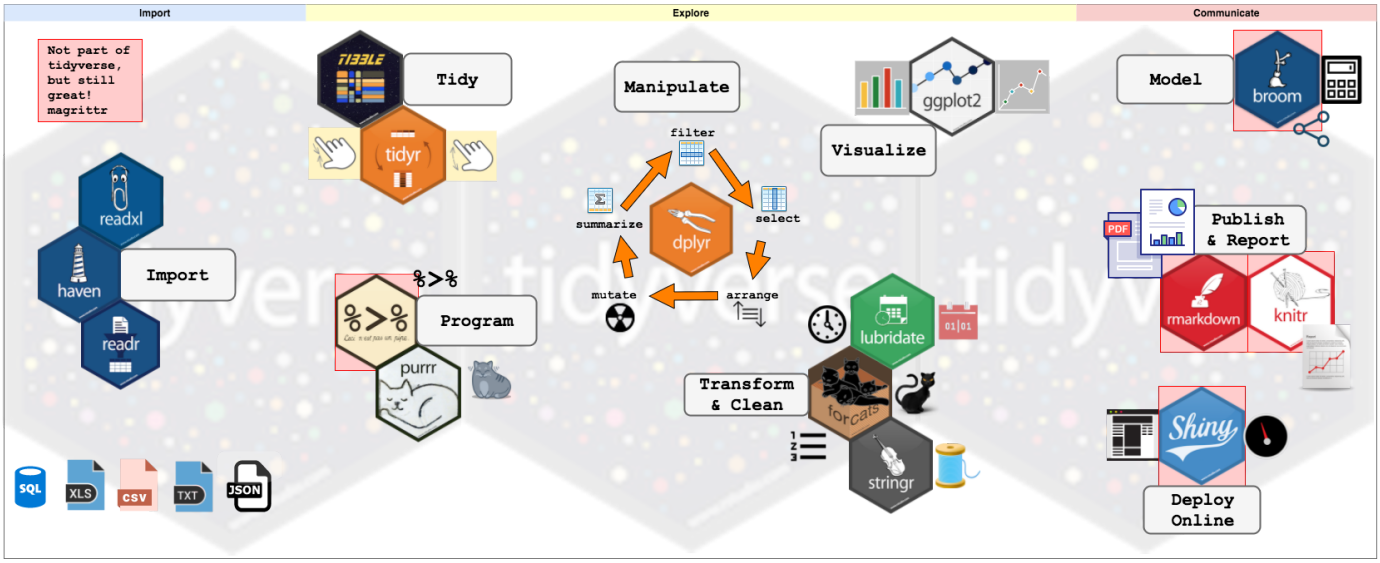
Bref, tout le travail technique préalable entre la collecte de la donnée et la valorisation proprement dite de ces sources. On estime qu'un scientifique de la donnée passe en général ***la moitié de son temps*** à cela.

Sous R, comme souvent, il y a plusieurs façons d'aborder cette question. Nous choisirons, lors de ce module de formation, d'explorer principalement les packages du framework *tidyverse*, qui ont l'avantage d'aborder ces différentes questions d'une façon intégrée et cohérente, d'une part entre elles, mais également avec d'autres.

Chapitre 2 Le Tidyverse

Le tidyverse est un ensemble de packages proposant une syntaxe cohérente pour remplir l'essentiel des traitements propres à la science de la données, de la lecture des données à la valorisation en passant par la modélisation. Le manifeste du tidyverse comprend 4 principes clefs pour les packages du tidyverse :

- Utiliser les structures de données existantes : ne pas créer des objets ad hoc
- Utiliser l'opérateur pipe
- S'intègrer dans l'approche de programmation fonctionnelle de R
- Designé pour les être humains : favoriser la facilité d'usage à la performance machine



2.1 Présentation des packages

2.1.1 Des packages pour lire des données

2.1.1.1 tidyverse

- readr pour les fichiers plats
- readxl pour les fichiers tableur Excel
- haven pour les données stockées sous des formats propriétaires (SAS, SPSS, …)

2.1.1.2 Hors tidyverse

- odbc / Rposgresql pour accéder à des données stockées sous forme de base de données
- sf pour lire des données spatiales
- rsdmx pour lire des données sdmx

2.1.2 Des packages pour manipuler des données

2.1.2.1 tidyverse

- dplyr fonctions correspondant à des “verbes” pour manipuler ses données
- tidyr fonctions pour modifier l’agencement de nos tables entre les lignes et les colonnes

2.1.3 Des packages pour nettoyer des données

2.1.3.1 tidyverse

- forcats permet de manipuler les variables de type catégorielle (ou factor en R)
- stringr permet de manipuler des chaînes de caractères
- lubridate permet de manipuler des dates

2.1.3.2 Hors tidyverse

- RcppRoll qui regroupe des opérations fenêtrées ou glissantes

2.2 Activer les packages

```
library (dplyr)
library (tidyr)
library (forcats)
library (lubridate)
library (stringr)
library (RcppRoll)
library (DT)
library (readxl)
library (dbplyr)
library (RPostgreSQL)
library (rsdmx)
library (sf)
```

2.3 Les spécificités du tidyverse

Quelques spécificités des fonctions de ce package :

- Ces packages sont orientés manipulation de *dataframes* et non de *vecteurs*
- En conséquence, on utilise jamais l'indexation des colonnes de tables (le "$") pour appeler une variable
- Chaque fonction ne fait qu'une chose et une seule (c'est une opération élémentaire)
- L'ensemble des fonctions obéissent à la même logique, ce qui permet de simplifier l'apprentissage
- l'ensemble de ces opérations élémentaires peuvent s'enchaîner à la manière d'un ETL avec le pipe

2.4 D'autres approches possibles

Les fonctions que nous allons voir obéissent à une logique intégrée et simple, qui permet des manipulations complexes, à partir du moment ou l'on est capable d'identifier et de sérier chaque *opération élémentaire* à réaliser. D'autres packages permettent également de réaliser ce type de manipulations. La différence est qu'ils sont souvent dédiés à une tâche spécifique, ce qui rend la cohérence moins évidente lorsque l'on doit réaliser plusieurs opérations. Un autre package propose toutefois une vision intégrée de la sorte : data.table. Plusieurs différences sont à noter :

- *data.table* est plus rapide sur d'importants volumes de données, le code est très succinct.
- *dplyr* est plus simple à apprendre, le code est plus lisible, il peut s'appliquer à des formats de données multiples, il s'intègre dans un framework global qui va de la lecture des données (readr, readxl,haven…) à leur valorisation (ggplot2).

Chapitre 3 Lire des données

3.1 readxl : lire des données Excel

La fonction *read_excel()* permet d'importer les données d'un fichier Excel. On peut spécifier :

- la feuille, les colonnes, les lignes ou la zone à importer
- les lignes à supprimer avant importation
- si on souhaite importer la première ligne comme des noms de variables ou non
- le format des variables importées
- la valeur qui sera interprétée comme étant la valeur manquante

```
sitadel <- read_excel ("data/ROES_201702.xls", sheet = "AUT_REG",
                       col_types = c ("text","text","numeric","numeric","numeric","numeric"
datatable (sitadel)
```

Show 10 entries Search:

	date	REG	log_AUT	ip_AUT	ig_AUT	colres_AUT
1	200001	01	440	194	12	234
2	200002	01	564	228	18	318
3	200003	01	348	220	19	109
4	200004	01	315	220	42	53
5	200005	01	390	250	66	74
6	200006	01	749	269	214	266
7	200007	01	420	185	76	159
8	200008	01	578	243	19	316
9	200009	01	496	299	42	155
10	200010	01	569	238	32	299

Showing 1 to 10 of 5,356 entries

Previous 1 2 3 4 5 … 536 Next

3.2 read_delim : lire des fichiers plats

La fonction *read_delim()* permet d'importer les données d'un fichier csv. Elle fonctionne de la même façon que *read_excel()*. On peut spécifier :

- le délimiteur de colonne
- les lignes à supprimer avant importation
- si on souhaite importer la première ligne comme des noms de variables ou non
- le *locale* du fichier
- la valeur qui sera interprétée comme étant la valeur manquante

read_csv(), *read_csv2()* et *read_tsv()* sont des implémentations prérenseignées de read_delim pour lire des fichiers plats avec séparateurs **,** **;** et **tabulaire**.

3.3 Télécharger des données disponibles sur le web

Parfois, les données que nous exploitons sont disponibles sur le web. Il est possible, directement depuis R, de télécharger ces données et, si nécessaire, de les décompresser (dans le répertoire de travail). Exemple sur les données SEQUOIA de l'ACOSS :

```
url <- "http://www.acoss.fr/files/Donnees_statistiques/SEQUOIA_TRIM_REGION.zip"
download.file (url, destfile = "SEQUOIA_TRIM_REGION.zip", method = "auto")
unzip (zipfile = "SEQUOIA_TRIM_REGION.zip")
SEQUOIA <- read_excel ("SEQUOIA_TRIM_REGION_BRUT.xls", sheet = "PAYS_DE_LA_LOIRE")
datatable (SEQUOIA)
```

3.4 Lire des fichiers avec une dimension spatiale

Le package sf (pour simple feature) permet d'importer dans R un fichier ayant une dimension spatiale. Après importation, le fichier est un dataframe avec une variable d'un type nouveau : la géométrie. Deux exemples ici pour lire des données au format shape et geojson.

```
Carte_EPCI_France <- st_read (dsn = "refgeo2017", layer = "Contour_epci_2017_region")
plot (Carte_EPCI_France)

communes2017 <- st_read (dsn = "refgeo2017/communes2017.geojson")
plot (communes2017)
```

Le package sf contient l'ensemble des fonctions permettant des manipulations sur fichiers géomatiques. On ne traitera pas ici de toutes ces fonctions en détail, mais la documentation se trouve ici.

A noter que sf étant complètement compatible avec les packages du tidyverse, la géométrie se conçoit comme une une donnée comme une autre, sur laquelle par exemple on peut réaliser des aggrégations.

3.5 Lire des données sous PostgreSQL

Deux approches possibles pour lire les données du patrimoine de la Dreal :

- *Importer* toutes ces données dans l'environnement R
- se *connecter* à ces données et utiliser un interpréteur permettant de traduire du code R comme une requête SQL.

3.5.1 Lire des données sous PostgreSQL : première approche

```
#Définition du driver
drv <- dbDriver ("PostgreSQL")

#Définition de la base de données
con <- dbConnect (drv, dbname = "dbname", host = "ip", port = numero_du_port,
                  user = "user_name", password = "pwd")

#Spécification de l'encodage, obligatoire avec Windows
postgresqlpqExec (con, "SET client_encoding = 'windows-1252'")

#Téléchargement de la table analyse du schéma pesticide
parametre <- dbGetQuery (con, "SELECT * FROM pesticides.parametre")

#Téléchargement de données avec dimension spatiale via la fonction st_read_db du package
station = st_read_db (con, query = "SELECT * FROM pesticides.station")
```

On voit que pour importer notre table analyse, on a simplement lancé une requête SQL. On peut bien sûr avec la même fonction lancer n'importe quelle requête sur la base et reccueillir le résultat.

3.5.2 Lire des données sous PostgreSQL : seconde approche

```
#définition du driver
drv <- dbDriver ("PostgreSQL")

#définition de la base de données
con <- dbConnect (drv, dbname = "dbname", host = "ip", port = numero_du_port, user = "us

#spécification de l'encodage, obligatoire avec windows
postgresqlpqExec (con, "SET client_encoding = 'windows-1252'")

#téléchargement de la table analyse du schéma pesticide
analyse_db <- tbl (con, in_schema ("pesticides", "analyse"))
```

Ici la table *analyse* n'est pas chargée dans l'environnement R, R s'est juste *connecté* à notre base de données.

On peut réaliser des opérations sur la table analyse avec du code R très simplement, par exemple ici pour filtrer sur les analyses relatives au Glyphosate :

```
analyse_db <-  filter (analyse_db, code_parametre == 1506)
```

Attention, ce code ne touche pas la base de donnée, il n'est pas exécuté. Pour l'exécuter, il faut par exemple afficher la table.

```
analyse_db
```

Même une fois le code exécuté, cette base n'est pas encore un dataframe. Pour importer la table, on utile la fonction **collect()**

```
analyse_db <- collect (analyse_db)
```

Cette approche est à conseiller sur d'importantes bases de données, et sans dimension spatiale, car dbplyr ne sait pas encore lire ce type de variable (ce qui ne saurait tarder).

3.6 Lire des données du webservice Insee

L'insee met à disposition un webservice d'accès à des données de référence sous un format appelé sdmx. Un package r, rsdmx permet de se connecter directement à ces données. Deux approches sont possibles. La première permet d'accéder à une série particulère.

```
url <- "https://bdm.insee.fr/series/sdmx/data/SERIES_BDM/001564471"
datainsee <- as.data.frame (readSDMX (url))
```

Cette approche peut être utilisée pour télécharger plusieurs séries en même temps. Ici par exemple nous téléchargeons l'ensemble des données sur les créations et défaillances d'entreprises pour les secteurs de la construction et de l'immobilier sur les Pays de la Loire.

```
url <- "https://bdm.insee.fr/series/sdmx/data/SERIES_BDM/001564471+001564503+001564799+(
datainsee <- as.data.frame (readSDMX (url))
```

L’autre approche permet de télécharger un ensemble de données d’une thématique appelé dataflow. Ici, par exemple, on va télécharger l’ensemble des données relatives à la construction neuve :

```
url <- "https://bdm.insee.fr/series/sdmx/data/CONSTRUCTION-LOGEMENTS"
datainsee <- as.data.frame (readSDMX (url))
```

Chapitre 4 Manipuler des données

4.1 Les principes des packages dplyr

Le but de dplyr est d'identifier et de rassembler dans un seul package les outils de manipulation de données les plus importantes pour l'analyse des données. Ce package rassemble donc des fonctions correspondant à un ensemble d'opérations élémentaires (ou *verbes*) qui permettent de :

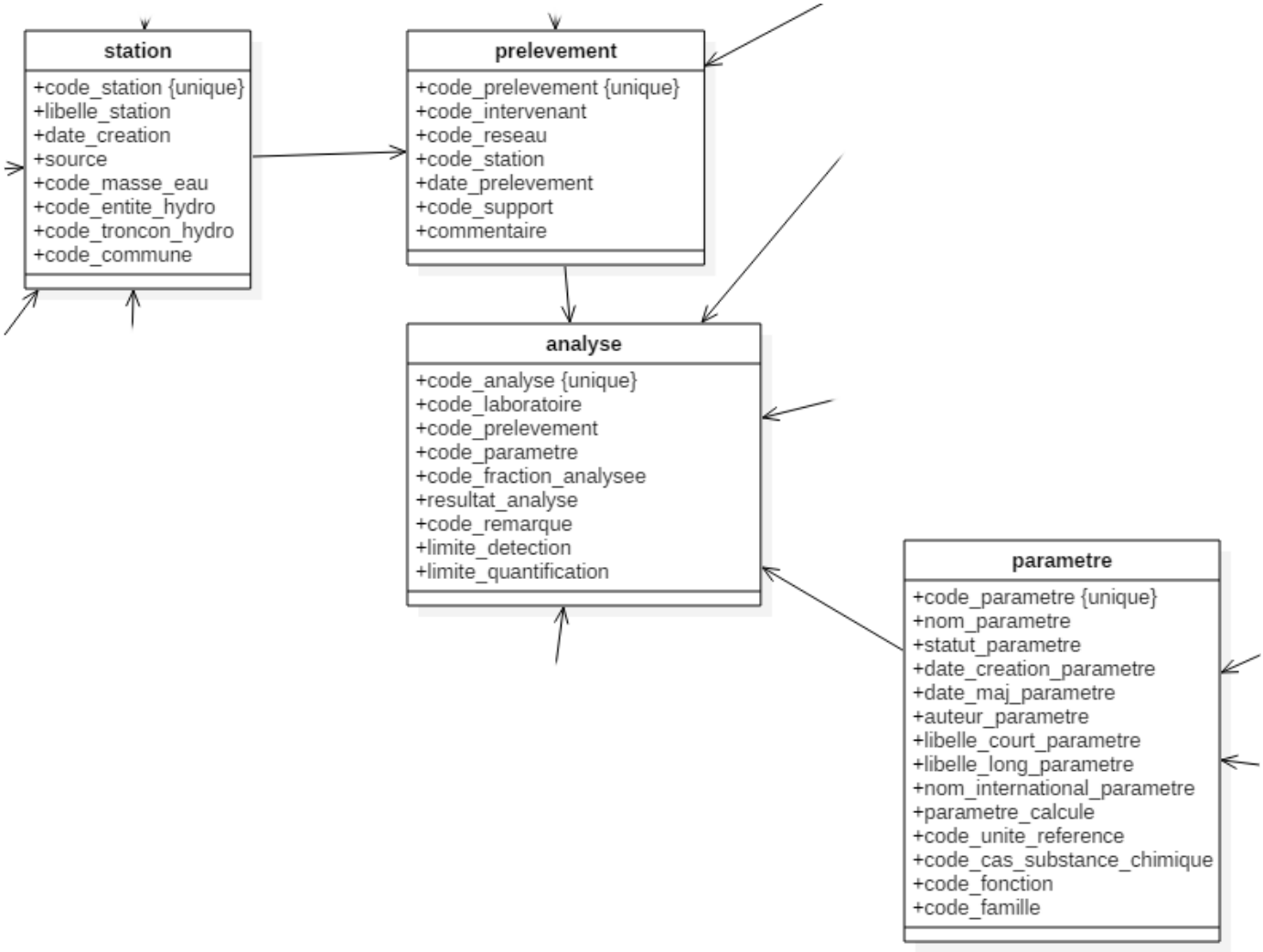
- Sélectionner un ensemble de variables : ***select()***
- Sélectionner un ensemble de lignes : ***filter()***
- Ajouter/modifier/renommer des variables : ***mutate()*** ou ***rename()***
- Produire des statistiques aggrégées sur les dimensions d'une table : ***summarise()***
- Trier une table : ***arrange()***
- Manipuler plusieurs tables : ***left_join()***, ***right_join()***, ***full_join()***, ***inner_join()***…

D'appliquer cela sur des données, quel que soit leur format : data frames, data.table, base de données sql, big data…

D'appliquer cela en articulation avec **group_by()** qui change la façon d'interpréter chaque fonction : d'une interprétation *globale* sur l'ensemble d'une table, on passe alors à une approche *groupe par groupe* : chaque groupe étant défini par un ensemble des modalités des variables défini dans l'instruction **group_by()**.

4.2 Présentation des données

On va travailler sur ce module principalement à partir des données sitadel en date réelle estimée et à partir des données de qualité des eaux de surface.



4.3 Chargement des données

```
load (file = "data/FormationPreparationDesDonnées.RData")
```

4.4 Les verbes clefs de dplyr pour manipuler une table

4.4.1 Sélectionner des variables : select()

Nous allons ici sélectionner un ensemble de variables de la table des prélèvements

```
prelevementb <- select (prelevement, date_prelevement, code_prelevement,
                        code_reseau, code_station)
datatable (prelevementb)
```

Show 10 entries Search:

	date_prelevement	code_prelevement	code_reseau	code_station
1	1992-07-08	1	ARS	044000110
2	1992-12-15	2	ARS	044000110
3	1993-04-21	3	ARS	044000110
4	1993-11-25	4	ARS	044000110
5	1994-04-26	5	ARS	044000110
6	1995-04-11	6	ARS	044000110
7	1996-09-23	8	ARS	044000110
8	1997-04-22	11	ARS	044000110
9	1997-05-25	12	ARS	044000110
10	1997-09-23	13	ARS	044000110

Showing 1 to 10 of 22,224 entries

Previous 1 2 3 4 5 … 2223 Next

```
prelevementb <- select (prelevement, -commentaire)
names (prelevementb)
```

```
## [1] "code_prelevement" "code_intervenant" "code_reseau"
## [4] "code_station"     "date_prelevement" "code_support"
```

select() possède ce qu'on appelle des helpers qui permettent de gagner du temps dans l'écriture de notre select. A partir du moment où les conventions de nommage sont correctement effectuées, cela permet de gagner également en reproductibilité d'une année sur l'autre.

Exemple d'usage : récupérer toutes les variables qui commencent par "code" :

```
prelevementb <- select (prelevement, starts_with ("code_"))
```

Exemple d'usage : récupérer toutes les variables qui commencent par "code" et date_prelevement :

```
prelevementb <- select (prelevement, starts_with ("code_"), one_of ("date_prelevement"))
```

4.4.2 Trier une table : arrange()

```
prelevementb <- arrange (prelevementb, date_prelevement)
```

4.4.3 Renommer une variable : rename()

```
prelevementb <- rename (prelevementb, date_p = date_prelevement)
```

On peut aussi directement renommer une variable dans l'opération **select()**

```
prelevementb <- select (prelevement, date_p = date_prelevement, code_prelevement,
                        code_reseau, code_station)
```

4.4.4 Filter une table : filter()

On va ici récupérer les analyses produites par l'ARS

```
ars <- filter (prelevement, code_reseau == "ARS")
```

L'exemple ci-dessus n'exerce un filtre que sur une condition unique.

Pour des conditions cumulatives (toutes les conditions doivent être remplies), le “&” ou la “,”

```
ars <- filter (prelevement, code_reseau == "ARS", code_intervenant == "44")
```

Pour des conditions non cumulatives (au moins une des conditions doit être remplie), le “|”

```
ars <- filter (prelevement, code_reseau == "ARS" | code_reseau == "FREDON")
```

Si une condition non cumulative s’applique sur une même variable, privilégier un test de sélection dans une liste avec le %in%

```
ars <- filter (prelevement, code_reseau %in% c ("ARS", "FREDON"))
```

Pour sélectionner des observations qui ne répondent pas à la condition, le !

Toutes les observations ayant été réalisées par un autre réseau que l’ARS

```
non_ars <- filter (prelevement, !(code_reseau == "ARS"))
```

4.4.5 Modifier/rajouter une variable : mutate()

mutate() est le verbe qui permet la transformation d’une variable existante ou la création d’une nouvelle variable dans le jeu de données.

Création de nouvelles variables

```
prelevementb <- mutate (prelevementb,
                        code_prelevement_caract = as.character (code_prelevement),
                        code_reseau_fact = as.factor (code_reseau))
```

Modification de variables existantes

```
prelevementb <- mutate (prelevementb,
                        code_prelevement = as.character (code_prelevement),
                        code_reseau = as.factor (code_reseau))
```

mutate() possède une variante, **transmute()**, qui fonctionne de la même façon que **mutate()**, mais ne garde que les variables modifiées ou crées par le verbe.

4.4.6 Extraire un vecteur : pull()

pull() permet d'extraire une variable d'un dataframe comme un vecteur

```
stations_de_la_table_prelevement <- pull (prelevement, code_station)
stations_de_la_table_prelevement <- unique (stations_de_la_table_prelevement)
```

4.5 La boîte à outils pour créer et modifier des variables avec R

4.5.1 Manipuler des variables numériques

Vous pouvez utiliser beaucoup de fonction pour créer des variables avec **mutate()**.

- Les opérations arithmétiques : +,-,**,/,^*
- Arithmétique modulaire : *%/%* (division entière) et *%%* (le reste), où x == y * (x %/% y) + (x %% y)
- Logarithmes : *log()*, *log2()*, *log10()*
- Navigations entre les lignes : *lead()* et *lag()* qui permettent d'avoir accès à la valeur suivante et précédente d'une variable.

```
x <- sample (1:10)
lagx <- lag (x)
leadx <- lead (x)
lag2x <- lag (x, n = 2)
lead2x <- lead (x, n = 2)
cbind (x = x, lagx = lagx, lag2x = lag2x, leadx = leadx, lead2x = lead2x)
```

```
##        x lagx lag2x leadx lead2x
##  [1,]  3   NA    NA     1      9
##  [2,]  1    3    NA     9      7
##  [3,]  9    1     3     7      4
##  [4,]  7    9     1     4     10
##  [5,]  4    7     9    10      2
##  [6,] 10    4     7     2      5
##  [7,]  2   10     4     5      6
##  [8,]  5    2    10     6      8
##  [9,]  6    5     2     8     NA
## [10,]  8    6     5    NA     NA
```

- opérations cumulatives ou glissantes :
    - R fournit des fonctions pour obtenir **opérations cumulatives** les somme, produit, minimum et maximum cumulés, dplyr fournit l'équivalent pour les moyennes : *cumsum()*, *cumprod()*, *cummin()*, *cummax()*, *cummean()*
    - Pour appliquer des **opérations glissantes**, on peut soit créer l'opération avec l'instruction *lag()*, soit exploiter le package **RcppRoll** qui permet d'exploiter des fonctions prédéfinies.

Exemple de somme glissante sur un pas de 2 observations.

```
x <- sample (1:10)
cumsumx <- cumsum (x)
rollsumx <- roll_sum (x, n=2)
rollsumx
```

```
## [1]  6 11  9  7  6  9 17 18 17
```

La fonction `roll_sumr()` fait en sorte d'obtenir un vecteur de même dimension que l'entrée x

```
rollsumrx <- roll_sumr (x, n=2)
rollsumrx
```

```
##  [1] NA  6 11  9  7  6  9 17 18 17
```

```
length(rollsumrx) == length(x)
```

```
## [1] TRUE
```

Nous pouvons obtenir une matrice des différenntes valeurs calculées :

```
cbind (x = x, cumsumx = cumsum (x), rollsumx = rollsumx, rollsumrx = roll_sumr (x, n=2
```

```
##        x cumsumx rollsumx rollsumrx
##  [1,]  1       1        6        NA
##  [2,]  5       6       11         6
##  [3,]  6      12        9        11
##  [4,]  3      15        7         9
##  [5,]  4      19        6         7
##  [6,]  2      21        9         6
##  [7,]  7      28       17         9
##  [8,] 10      38       18        17
##  [9,]  8      46       17        18
## [10,]  9      55        6        17
```

- Comparaisons logiques : <, <=, >, >=, *!=*
- Rangs : **min_rank()** devrait être la plus utile, il existe aussi notamment **row_number()**, **dense_rank()**, **percent_rank()**, **cume_dist()**, **ntile()**.
- **coalesce (x, y)** : permet de remplacer les valeurs manquantes de x par celle de y

- ***variable = ifelse (condition (x), valeursi, valeursinon)*** permet d'affecter *valeursi* ou *valeursinon* à *variable* en fonction du fait que *x* répond à *condition*. Exemple : création d'une variable résultat pour savoir si le résultat de nos analyses est bon ou non.

```
analyseb <- mutate (analyse,
                    resultat_ok = ifelse (code_remarque %in% c (1,2,7,10),
                                          yes = TRUE, no = FALSE))
```

- ***case_when()*** permet d'étendre la logique de ifelse à des cas plus complexes. Les conditions mises dans un ***case_when()*** ne sont pas exclusives. De ce fait, il faut pouvoir déterminer l'ordre d'évaluation des conditions qui y sont posées. Cet ordre s'effectue de bas en haut, c'est à dire que la dernière condition évaluée (celle qui primera sur toutes les autres) sera la première à écrire. Exemple: On va ici recalculer des seuils fictifs sur les analyses.

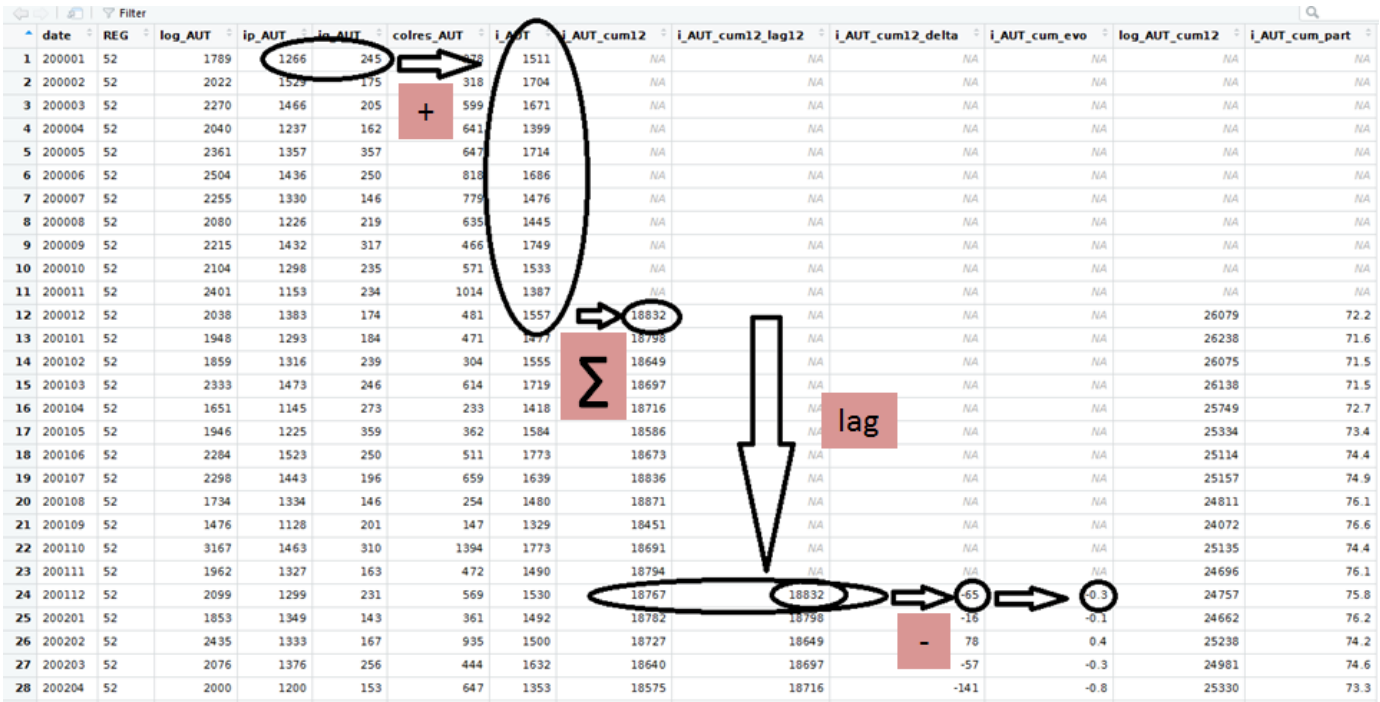
```
analyseb <- mutate (analyse, classe_resultat_analyse = case_when (
                        resultat_analyse == 0  ~ "1",
                        resultat_analyse <= 0.001  ~ "2",
                        resultat_analyse <= 0.01  ~ "3",
                        resultat_analyse <= 0.1  ~ "4",
                        resultat_analyse > 0.1  ~ "5",
                        TRUE ~ ""
                    ))
```

4.5.2 Exercice : Les données mensuelles sitadel

A partir du fichier sitadel de février 2017 (ROES_201702.xls), sur la région Pays de la Loire (code région 52), livrer un fichier contenant pour chaque mois, pour les logements individuels (i_AUT = ip_AUT + ig_AUT) :

- le cumul des autorisations sur 12 mois glissants(i_AUT_cum12)
- le taux d'évolution du cumul sur 12 mois (i_AUT_cum_evo, en %)
- la part de ce cumul dans celui de l'ensemble des logements autorisés (log_AUT), en pourcentage

```
sitadel <- read_excel ("data/ROES_201702.xls", sheet = "AUT_REG",
                       col_types = c ("text","text","numeric","numeric","numeric","numeric"
```



	date	REG	log_AUT	ip_AUT	ip_AUT_	colres_AUT	i_AUT	i_AUT_cum12	i_AUT_cum12_lag12	i_AUT_cum12_delta	i_AUT_cum_evo	log_AUT_cum12	i_AUT_cum_part
1	200001	52	1789	1266	245	[illegible]	1511	NA	NA	NA	NA	NA	NA
2	200002	52	2022	1529	175	318	1704	NA	NA	NA	NA	NA	NA
3	200003	52	2270	1466	205	599	1671	NA	NA	NA	NA	NA	NA
4	200004	52	2040	1237	162	641	1399	NA	NA	NA	NA	NA	NA
5	200005	52	2361	1357	357	647	1714	NA	NA	NA	NA	NA	NA
6	200006	52	2504	1436	250	818	1686	NA	NA	NA	NA	NA	NA
7	200007	52	2255	1330	146	779	1476	NA	NA	NA	NA	NA	NA
8	200008	52	2080	1226	219	635	1445	NA	NA	NA	NA	NA	NA
9	200009	52	2215	1432	317	466	1749	NA	NA	NA	NA	NA	NA
10	200010	52	2104	1298	235	571	1533	NA	NA	NA	NA	NA	NA
11	200011	52	2401	1153	234	1014	1387	NA	NA	NA	NA	NA	NA
12	200012	52	2038	1383	174	481	1557	18832	NA	NA	NA	26079	72.2
13	200101	52	1948	1293	184	471	1477	18798	NA	NA	NA	26238	71.6
14	200102	52	1859	1316	239	304	1555	18649	NA	NA	NA	26075	71.5
15	200103	52	2333	1473	246	614	1719	18697	NA	NA	NA	26138	71.5
16	200104	52	1651	1145	273	233	1418	18716	NA	NA	NA	25749	72.7
17	200105	52	1946	1225	359	362	1584	18586	NA	NA	NA	25334	73.4
18	200106	52	2284	1523	250	511	1773	18673	NA	NA	NA	25114	74.4
19	200107	52	2298	1443	196	659	1639	18836	NA	NA	NA	25157	74.9
20	200108	52	1734	1334	146	254	1480	18871	NA	NA	NA	24811	76.1
21	200109	52	1476	1128	201	147	1329	18451	NA	NA	NA	24072	76.6
22	200110	52	3167	1463	310	1394	1773	18691	NA	NA	NA	25135	74.4
23	200111	52	1962	1327	163	472	1490	18794	NA	NA	NA	24696	76.1
24	200112	52	2099	1299	231	569	1530	18767	18832	-65	-0.3	24757	75.8
25	200201	52	1853	1349	143	361	1492	18782	18798	-16	-0.1	24662	76.2
26	200202	52	2435	1333	167	935	1500	18727	18649	78	0.4	25238	74.2
27	200203	52	2076	1376	256	444	1632	18640	18697	-57	-0.3	24981	74.6
28	200204	52	2000	1200	153	647	1353	18575	18716	-141	-0.8	25330	73.3

4.5.3 Manipuler des dates

Parmi l'ensemble des manipulations de variables, celle des dates et des heures est toujours une affaire complexe. Le framework tidyverse propose le package ***lubridate*** qui permet de gérer ces informations de façon cohérente.

- gestion des dates :

```
dmy ("jeudi 21 novembre 2017")
dmy ("21112017")
ymd ("20171121")
```

- gestion des dates/heures :

```
dmy_hms ("mardi 21 novembre 2017 9:30:00")
```

- combien de jours avant Noël ?

```
dmy ("25 décembre 2018") - dmy ("16 avril 2018")
```

- le jour de la semaine d'une date :

```
wday (dmy ("19012038"), label = TRUE)
```

La fonction make_date et make_datetime vous permettent de transformer un ensemble de variables en un format date ou date heure. Utile par exemple lorsque l'on a une variable pour l'année, le mois et le jour.

Exercice : convertir les données de la table exercice pertinentes au format date.

4.5.4 Manipuler des chaînes de caractères

Le package *stringr* compile l'ensemble des fonctions de manipulation de chaînes de caractère utiles sur ce type de données.

On peut diviser les manipulations de chaîne de caractère en 4 catégories :

- manipulations des caractères eux-mêmes
- gestion des espaces
- opérations liées à la langue
- manipulations de "pattern", notamment des expressions régulières.

4.5.4.1 Manipulations sur les caractères

Obtenir la longueur d'une chaîne

```
str_length ("abc")
```

```
## [1] 3
```

Extraire une chaîne de caractère

str_sub prend 3 arguments : une chaîne de caractère, une position de début, une position de fin. Les positions peuvent être positives, et dans ce cas, on compte à partir de la gauche, ou négatives, et dans ce cas on compte à partir de la droite.

```
x <- c ("abcdefg", "hijklmnop")
str_sub (string = x, start = 3, end = 4)
```

```
## [1] "cd" "jk"
```

```
str_sub (string = x, start = 3, end = -2)
```

```
## [1] "cdef"   "jklmno"
```

str_sub peut être utilisé pour remplacer un caractère

```
str_sub (x, start = 3, end = 4) <- "CC"
x
```

```
## [1] "abCCefg"   "hiCClmnop"
```

4.5.4.2 Gestion des espaces

la fonction **str_pad()** permet de compléter une chaîne de caractère pour qu'elle atteigne une taille fixe. Le cas typique d'usage est la gestion des codes communes Insee.

```
code_insee <- 1001
str_pad (code_insee, 5, pad = "0")
```

```
## [1] "01001"
```

On peut choisir de compléter à gauche, à droite, et on peut choisir le "pad". Par défaut, celui-ci est l'espace.

La fonction inverse de **str_pad()** est **str_trim()** qui permet de supprimer les espaces aux extrémités de notre chaîne de caractères.

```
proust <- "   Les paradoxes d'aujourd'hui sont les préjugés de demain.  "
str_trim (proust)
```

```
## [1] "Les paradoxes d'aujourd'hui sont les préjugés de demain."
```

```
str_trim (proust, side = "left")
```

```
## [1] "Les paradoxes d'aujourd'hui sont les préjugés de demain.  "
```

Les **expressions régulières** permettent la détection de “patterns” sur des chaîne de caractères.

```
txt <- c ("voiture","train", "voilier", "bus", "avion", "tram", "trotinette")
str_detect (string = txt, pattern = "^tr") # les éléments qui commencent pas les lettre
```

```
## [1] FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE
```

```
txt [str_detect (string = txt, pattern = "^tr")]
```

```
## [1] "train"      "tram"       "trotinette"
```

```
str_detect (string = txt, pattern = "e$") # les éléments qui terminent par la lettre e
```

```
## [1]  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE
```

```
txt [str_detect (string = txt, pattern = "e$")]
```

```
## [1] "voiture"    "trotinette"
```

4.5.4.3 Opérations liées à la langue

Ces différentes fonctions ne donneront pas le même résultat en fonction de la langue par défaut utilisée. La gestion des majuscules/minuscules :

```
proust <- "Les paradoxes d'aujourd'hui sont LES préjugés de Demain."
str_to_upper (proust)
```

```
## [1] "LES PARADOXES D'AUJOURD'HUI SONT LES PRÉJUGÉS DE DEMAIN."
```

```
str_to_lower (proust)
```

```
## [1] "les paradoxes d'aujourd'hui sont les préjugés de demain."
```

```
str_to_title (proust)
```

```
## [1] "Les Paradoxes D'aujourd'hui Sont Les Préjugés De Demain."
```

La gestion de l'ordre :

```
x <- c ("y", "i", "k")
str_order (x)
```

```
## [1] 2 3 1
```

```
str_sort (x)
```

```
## [1] "i" "k" "y"
```

Suppression des accents (base::iconv) :

```
proust2 <- "Les paradoxes d'aujourd'hui sont les préjugés de demain ; et ça c'est embêta
iconv (proust2, to = "ASCII//TRANSLIT")
```

```
## [1] "Les paradoxes d'aujourd'hui sont les prejuges de demain ; et ca c'est embetant"
```

4.5.5 Manipuler des variables factorielles (=qualitatives)

Les fonctions du module forcats permettent de modifier les modalités d'une variable factorielle, notamment :

- Changer les modalités des facteurs et/ou leur ordre
- Regrouper des modalités

On va ici utiliser cette fonction pour modifier le tri des stations en fonction de leur fréquence d'apparition dans la table "prelevement""

forcats permet beaucoup d'autres possibilités de tri :

- manuellement des facteurs (***fct_relevel()***);
- en fonction de la valeur d'une autre variable (***fct_reorder()***);
- en fonction de l'ordre d'apparition des modalités (***fct_inorder()***).

Consulter la doc du module pour voir toutes les possibilités très riches de ce module.

En quoi ces fonctions sont utiles ?

Elles permettent notamment :

- lorsqu'on fait des graphiques, d'afficher les occurences les plus importantes d'abord ;
- de lier l'ordre d'une variable en fonction d'une autre (par exemple les code Insee des communes en fonction des régions).

Exemple : ordonner les modalités d'un facteur pour améliorer l'aspect d'un graphique

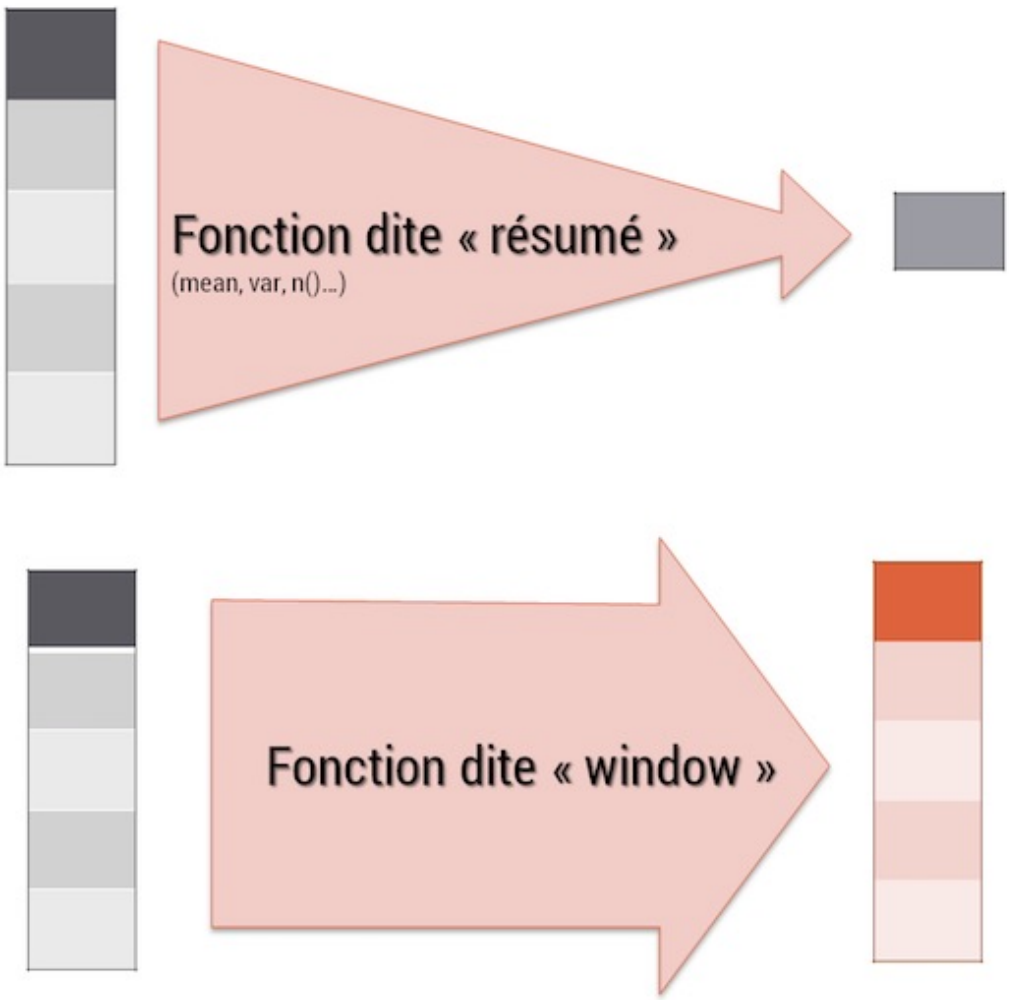
```
library (ggplot2)
library (forcats)
num <- c (1, 8, 4, 3, 6, 7, 5, 2, 11, 3)
cat <- c (letters [1:10])
data <- data.frame (num, cat)

ggplot (data, aes (x = cat, num)) +
  geom_bar (stat = "identity") +
    xlab (label = "Facteur") + ylab (label ="Valeur")
```

```
ggplot (data, aes (x = fct_reorder (cat, -num), num)) +
  geom_bar (stat = "identity") +
  xlab (label = "Facteur ordonné") + ylab (label ="Valeur")
```

4.6 Aggréger des données : summarise()

La fonction ***summarise()*** permet d'aggréger des données, en appliquant une fonction sur les variables pour construire une statistique sur les observations de la table. **summarise()** est une fonction dite de "résumé". À l'inverse de **mutate()**, quand une fonction summarise est appelée, elle retourne une seule information. La moyenne, la variance, l'effectif…sont des

informations qui condensent la variable étudiée en une seule information.



La syntaxe de summarise est classique. Le resultat est un dataframe

```
summarise (exercice,
           mesure_moyenne = mean (resultat_analyse, na.rm = T))
```

On peut calculer plusieurs statistiques sur une aggrégation

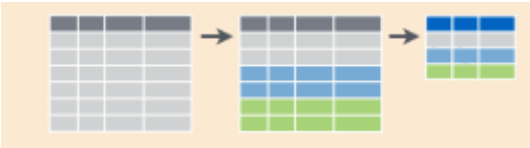
```
summarise (exercice,
           mesure_moyenne = mean (resultat_analyse, na.rm = T),
           mesure_total = sum (resultat_analyse, na.rm = T)
           )
```

4.6.1 Quelques fonctions d'aggrégations utiles

- compter : *n()*
- sommer : *sum()*

- compter des valeurs non manquantes *sum(!is.na())*
- moyenne : *mean()*, moyenne pondérée : *weighted.mean()*
- écart-type : *sd()*
- médiane : *median()*, quantile : *quantile(.,quantile)*
- minimum : *min()*, maximum : *max()*
- position : *first()*, *nth(., position)*, *last()*

4.7 Aggréger des données par dimension : group_by()

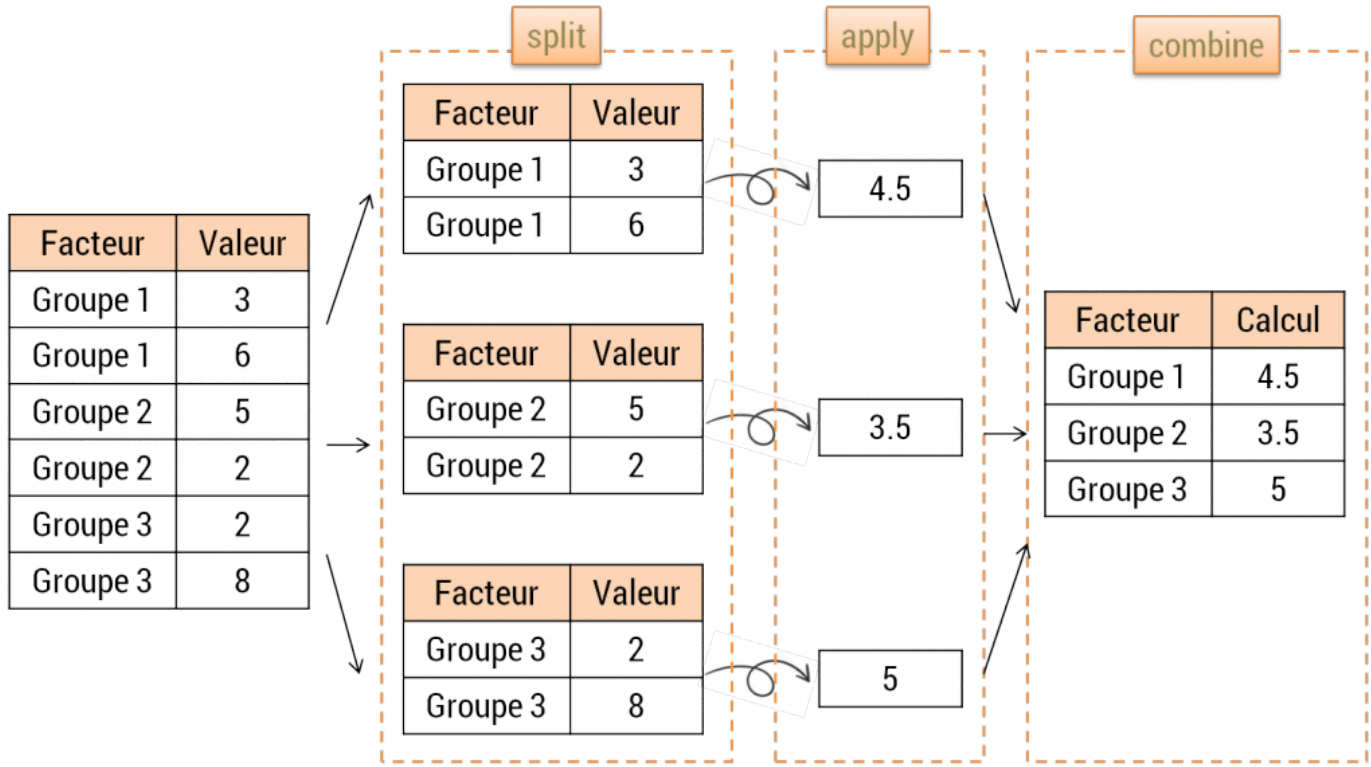


Summarise est utile, mais la plupart du temps, nous avons besoin non pas d'aggréger des données d'une table entière, mais de construire des aggrégations sur des sous-ensembles : par années, départements… La fonction ***group_by()*** va permettre *d'éclater* notre table en fonction de dimensions de celle-ci.

Ainsi, si on veut construire des statistiques agrégées non sur l'ensemble de la table, mais pour chacune des modalités d'une ou de plusieurs variables de la table. Il faut deux étapes :

- Utiliser prélablement la fonction ***group_by()*** pour définir les variables sur lesquelles on souhaite aggréger les données.
- Utiliser ***summarise()*** ou ***summarise_XX()*** sur la table en sortie de l'étape précédente

Découper un jeu de données pour réaliser des opérations sur chacun des sous-ensembles afin de les restituer ensuite de façon organisée est appelée stratégie du split – apply – combine schématiquement, c'est cette opération qui est réalisée par dplyr dès qu'un ***group_by()*** est introduit sur une table.



Exemple pour calculer les statistiques précédentes par mois :

```
exercice <- mutate (exercice,
                    annee = year (date_prelevement))

paran <- group_by (exercice, annee)

summarise (paran,
           mesure_moyenne = mean (resultat_analyse, na.rm = T),
           mesure_total = sum (resultat_analyse, na.rm = T)
          )
```

```
## # A tibble: 26 x 3
##    annee mesure_moyenne mesure_total
##    <dbl>          <dbl>        <dbl>
##  1 1991.         0.0981         4.32
##  2 1992.         0.137          8.33
##  3 1993.         0.123          6.14

##  4 1994.         0.0684         4.72
##  5 1995.         0.0803         6.99
##  6 1996.         0.0915         6.86
##  7 1997.         0.0529         5.14
##  8 1998.         0.131         46.5
##  9 1999.         0.0547        89.7
## 10 2000.         0.118        191.
## # ... with 16 more rows
```

Pour reprendre des traitements “table entière”, il faut mettre fin au ***group_by()*** par un ***ungroup()***

4.8 Le pipe

Le pipe est la fonction qui va vous permettre d’écrire votre code de façon plus lisible pour vous et les utilisateurs. Comment ? En se rapprochant de l’usage usuel en grammaire.

`verbe(sujet,complement)` devient `sujet %>% verbe(complement)`

Quand on enchaîne plusieurs verbes, l’avantage devient encore plus évident :

`verbe2(verbe1(sujet,complement1),complement2)` devient `sujet %>% verbe1(complement1) %>% verbe2(complement2)`

En reprenant l'exemple précédent, sans passer par les étapes intermédiaires, le code aurait cette tête :

```
summarise (
  group_by (
    mutate (
      exercice,
      annee = year (date_prelevement)
            ),
      annee
            ),
          mesure_moyenne = mean (resultat_analyse, na.rm = T),
          mesure_total = sum (resultat_analyse, na.rm = T)
          )
```

```
## # A tibble: 26 x 3
##    annee mesure_moyenne mesure_total
##    <dbl>          <dbl>        <dbl>
##  1 1991.         0.0981         4.32
##  2 1992.         0.137          8.33
##  3 1993.         0.123          6.14
##  4 1994.         0.0684         4.72
##  5 1995.         0.0803         6.99
##  6 1996.         0.0915         6.86
##  7 1997.         0.0529         5.14
##  8 1998.         0.131         46.5
##  9 1999.         0.0547        89.7
## 10 2000.         0.118        191.
## # ... with 16 more rows
```

Avec l'utilisation du pipe (raccourci clavier CTrl + Maj + M), il devient :

```
exercice %>%
  mutate (annee = year (date_prelevement)) %>%
  group_by (annee) %>%
    summarise (mesure_moyenne = mean (resultat_analyse, na.rm = T),
               mesure_total = sum (resultat_analyse, na.rm = T))
```

```
## # A tibble: 26 x 3
##    annee mesure_moyenne mesure_total
##    <dbl>          <dbl>        <dbl>
##  1 1991.         0.0981         4.32
##  2 1992.         0.137          8.33
##  3 1993.         0.123          6.14
##  4 1994.         0.0684         4.72
##  5 1995.         0.0803         6.99
##  6 1996.         0.0915         6.86
##  7 1997.         0.0529         5.14
##  8 1998.         0.131         46.5
##  9 1999.         0.0547        89.7
## 10 2000.         0.118        191.
## # ... with 16 more rows
```

4.9 La magie des opérations groupées

L'opération **group_by()** que nous venons de voir est très utile pour les aggrégations, mais elle peut aussi servir pour créer des variables ou filtrer une table, puisque **group_by()** permet de traiter notre table en entrée comme *autant de tables séparées* par les modalités des variables de regroupement.

4.10 Exercice

Sur les données "sitadel", effectuer les opérations suivantes en utilisant l'opérateur %>% :

- les mêmes calculs que ceux réalisés sur la région 52, mais sur chacune des régions
- les aggrégations par année civile pour chacune des régions, puis leur taux d'évolution d'une année sur l'autre (exemple : (val2015-val2014)/val2014)

4.11 Exercice

Sur les données "FormationPreparationDesDonnées.RData", table "exercice" :

- calculer le taux de quantification pour chaque molécule (code_parametre), chacune des année : nombre de fois où elle a été retrouvée (code_remarque=1) sur le nombre de fois où elle a été cherchée (code_remarque = 1,2,7 ou 10)
    - créer la variable “annee”
    - créer la variable de comptage des présences pour chaque analyse (1=présent, 0=absent)
    - créer la variable de comptage des recherches pour chaque analyse (1=recherchée, 0=non recherchée)
    - pour chaque combinaison année x code_parametre, calculer le taux de quantification
- trouver pour chaque station, sur l’année 2016, le prélèvement pour lequel la concentration cumulée, toutes substances confondues, est la plus élevée (~ le prélèvement le plus pollué)
    - filtrer les concentrations quantifiées (code_remarque=1) et l’année 2016
    - sommer les concentrations (resultat_analyse) par combinaison code_station x code_prelevement
    - ne conserver que le prélèvement avec le concentration maximale

4.12 Les armes non conventionnelles de la préparation des donnéees

Nous venons de voir les verbes de manipulation d’une table les plus fréquents de dplyr. Ces verbes sont pour la plupart déclinés dans des versions encore plus puissantes, que l’on pourrait appeler conditionnelles. Dans l’univers dplyr, ces verbes sont appelés des *scoped variants*

- **xx_at()**, ou xx est l’un des verbes précédents, permet d’appliquer une opération sur un ensemble de variables définies
- **xx_if()**, ou xx est l’un des verbes précédents, permet d’appliquer une opération sur toutes les variable de la table en entrée remplissant une condition particulière
- **xx_all()**, ou xx est l’un des verbes précédents, permet d’appliquer une opération sur toutes les variables de la table en entrée

La syntaxe diffère un peu sur ces versions. On peut la globaliser ainsi : fonction(***selectiondevariables***,***funs(opérationaréalisersurcesvariables)***) La sélection de variable diffère ensuite des fonctions :

- **xx_at()**, on donne une liste de variables
- **xx_if()**, on donne une condition que doivent remplir ces variables
- **xx_all()**, on prend toutes les variables

Exemple sur l'exercice sur les données sitadel.

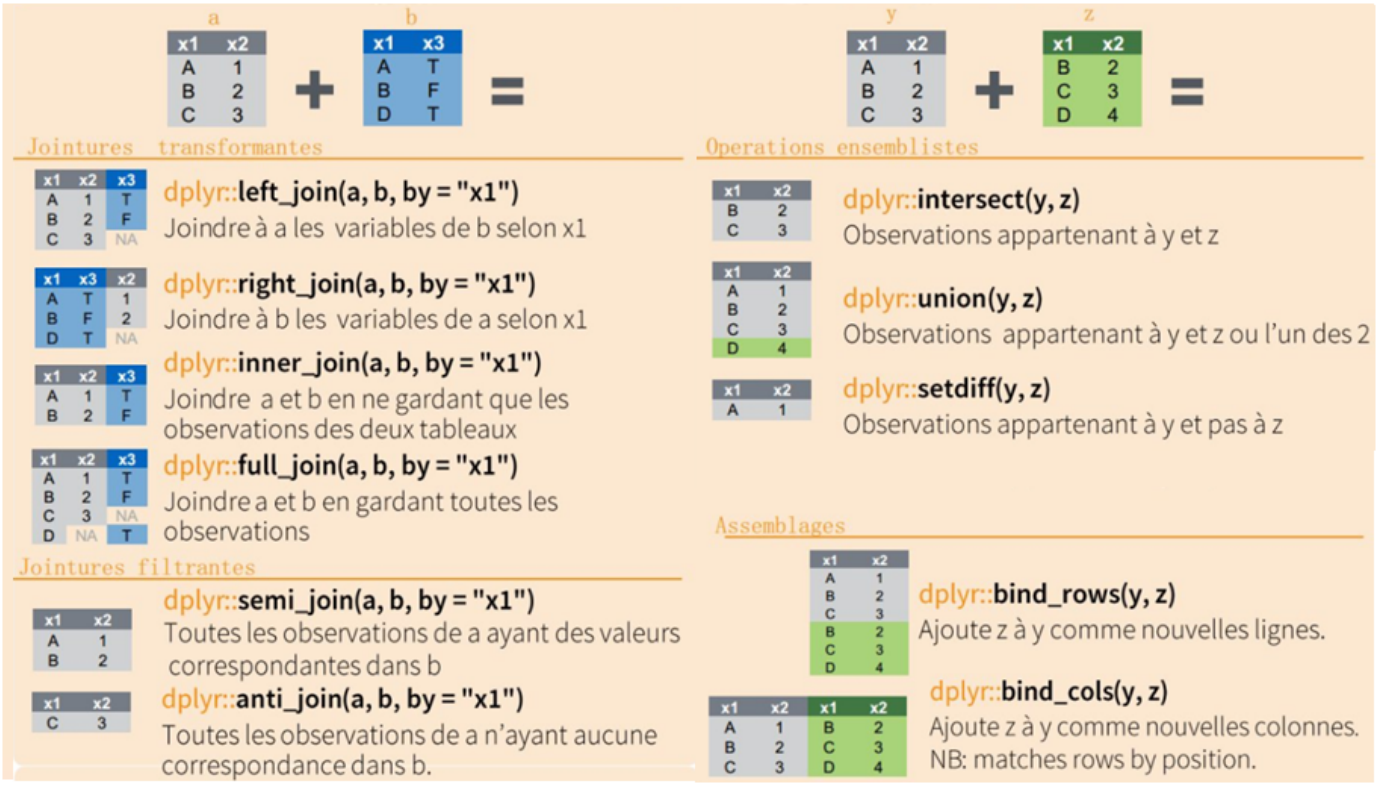
```
sitadel <- read_excel ("data/ROES_201702.xls", "AUT_REG") %>%
  group_by (REG) %>%
  mutate_if (is.numeric, funs (cumul12 = roll_sumr (., n = 12))) %>%
  mutate_at (vars (ends_with ("cumul12")), funs (evo = 100 * . / lag (., 12) - 100)) %>%
  mutate_at (vars (ends_with ("cumul12")), funs (part = 100 *./ log_AUT_cumul12))
```

Les verbes ayant ces variantes sont les suivants : select(), arrange(), rename(), filter(), mutate(), transmute(), group_by(), summarise().

Chapitre 5 Manipuler plusieurs tables

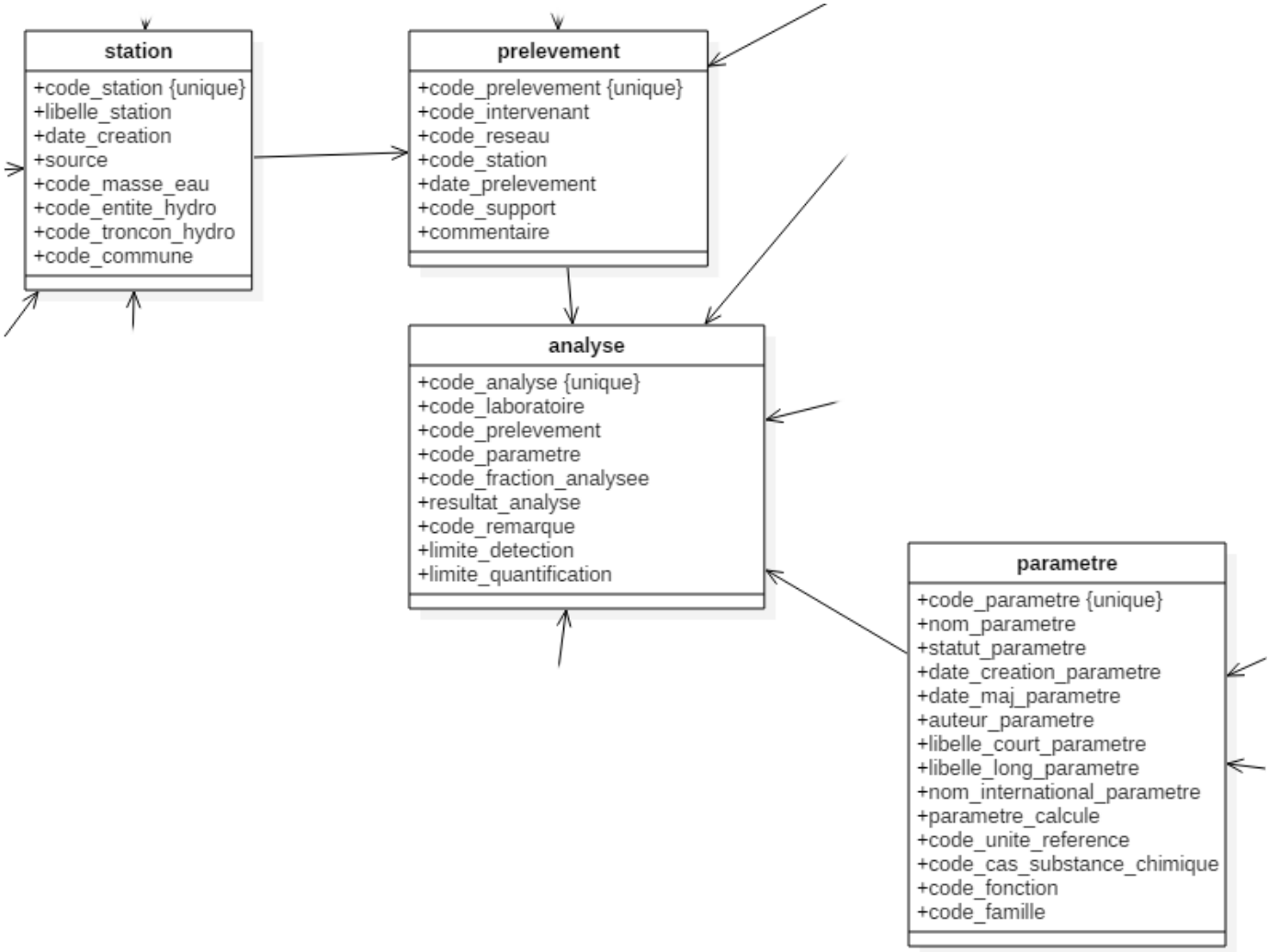
Le package *dplyr* possède également plusieurs fonctions permettant de travailler sur deux tables. On va pouvoir regrouper ces fonctions en plusieurs catégories de manipulations :

- pour fusionner des informations de deux tables entre elles : jointures transformantes
- pour sélectionner des observations d'une table en fonction de celles présentes dans une autre table : jointures filtrantes
- pour traiter deux tables ayant les mêmes colonnes et sélectionner sur celles-ci des observations de l'une et l'autre : opérations ensemblistes
- Des manipulations visant à additionner deux tables ensembles : assemblages



5.1 Exercices

- reconstituer le dataframe "exercice" à partir des données contenues dans les tables "analyse", "prelevement" et "station" (jointures)



- calculer le nombre d’analyses réalisées sur des molécules (code_parametre) présentes dans le référentiel
- produire une liste des **code_parametre** associés à des analyses mais absents du référentiel
- produire une table des analyses “orphelines”, c’est-à-dire qui ne correspondent pas à un prélèvement

Chapitre 6 Structurer ses tables

6.1 Pourquoi se pencher sur la structuration des tables ?

Pour bien manipuler des données, leur structuration est fondamentale.

- Qu'est ce qu'une ligne de notre table ?
- Qu'est ce qu'une colonne de notre table ?

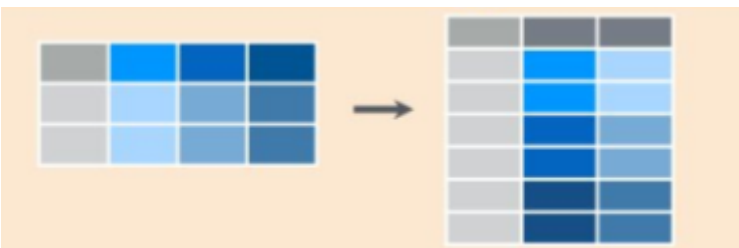
Sur une table non aggrégée (un répertoire, une table d'enquête…), la structuration naturelle est une ligne par observation (un individu, une entreprise…), une colonne par variable (âge, taille…) sur cette observation.

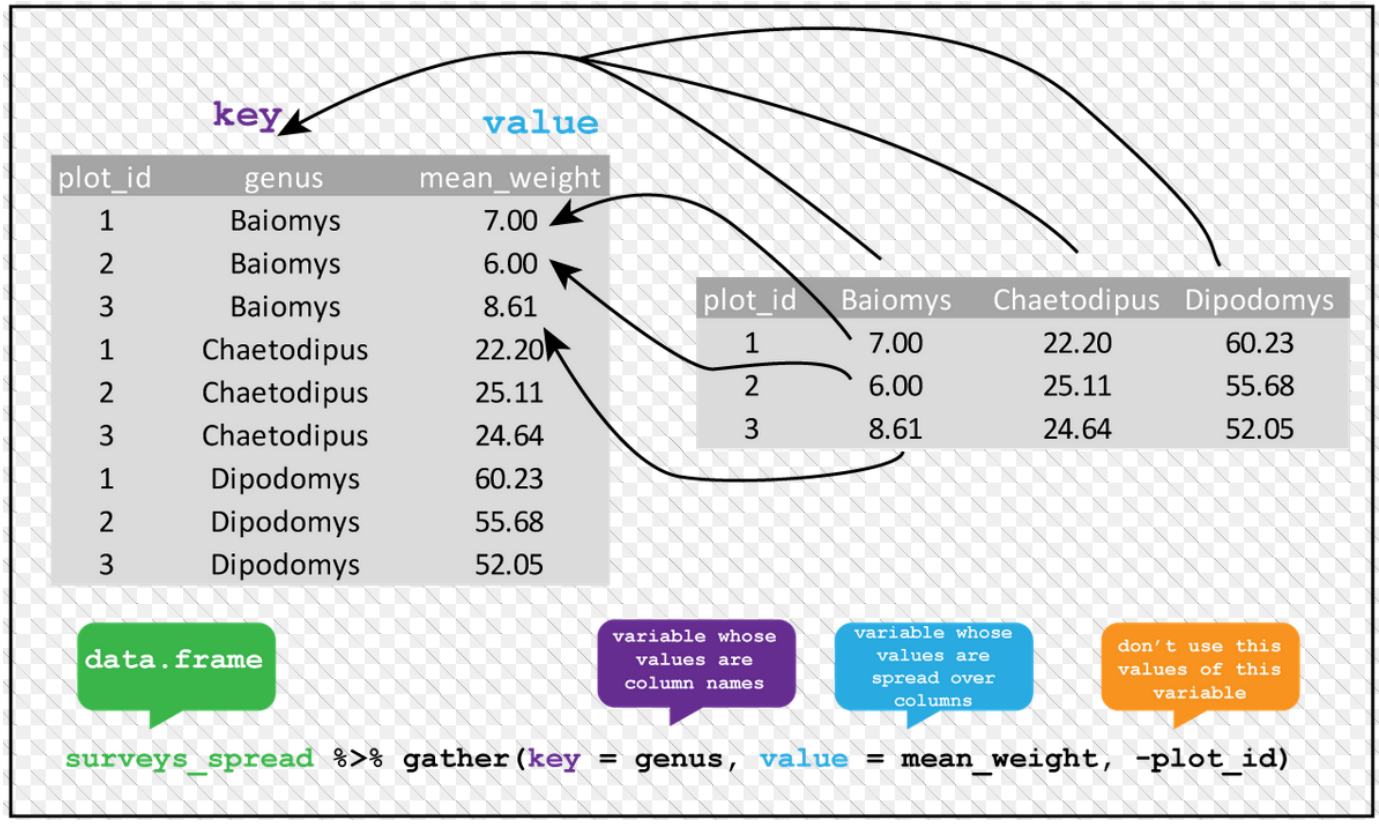
Mais dès qu'on aggrège une telle table pour construire des tables structurées par dimensions d'analyse et indicateurs, se pose toujours la question de savoir ce qu'on va considérer comme des dimensions et comme des indicateurs.

La bonne réponse, c'est que ça dépend de ce que l'on veut en faire. L'important est de pouvoir facilement passer de l'un à l'autre suivant ce que l'on doit faire. C'est l'intérêt du module *tidyr*.

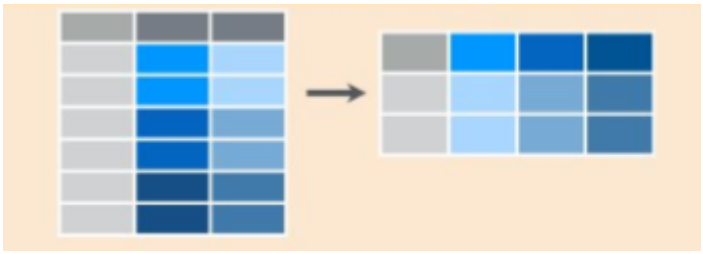
6.2 Les deux fonctions clefs de tidyr

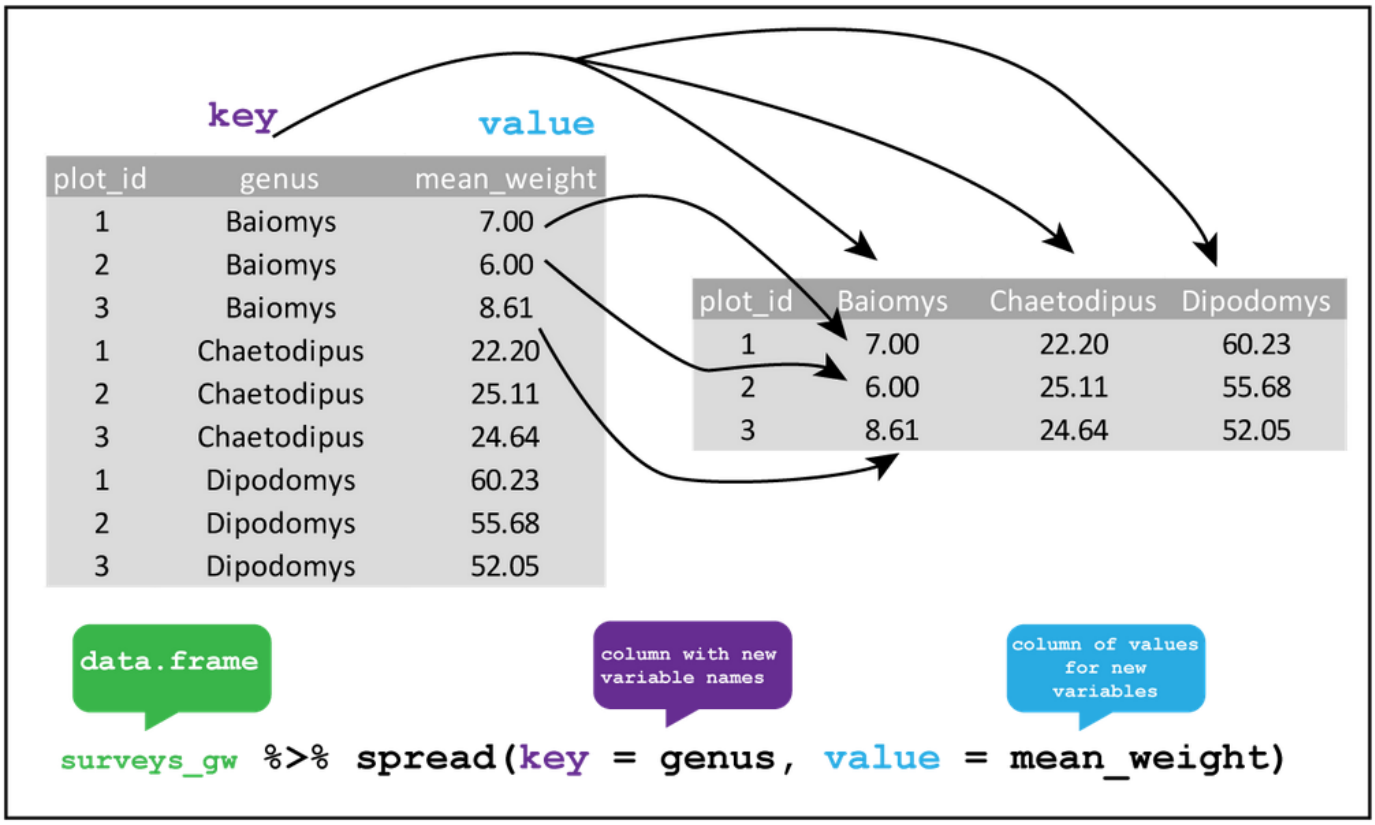
- **gather()** permet d'empiler plusieurs colonnes (correspondant à des variables quantitatives). Elles sont repérées par création d'une variable qualitative, à partir de leurs noms. Le résultat est une table au format *long*





- **spread()** fait l'inverse. Cette fonction crée autant de colonnes qu'il y a de modalités d'une variable qualitative, en remplissant chacune par le contenu d'une variable numérique. Le résultat est une table au format *large*





Un exemple : obtenir un fichier avec une ligne par région, et une colonne par année qui donne l'évolution en % de la construction neuve par rapport à l'année précédente

```
sitadel_long <- read_excel ("data/ROES_201702.xls", "AUT_REG") %>%
  mutate (ANNEE = str_sub (date, 1, 4)) %>%
  group_by (REG, ANNEE) %>%
    summarise_if (is.numeric, funs (sum (., na.rm = T))) %>%
    mutate_if (is.numeric, funs (EVO = 100 * . / lag (.) - 100)) %>%
    select (REG, ANNEE, log_AUT_EVO) %>%
  ungroup ()

sitadel_large <- sitadel_long %>%
  spread (key = ANNEE, value = log_AUT_EVO, sep = "_")

sitadel_long2 <- sitadel_large %>%
  gather (key = annee, value = log_aut_evo, -REG)
```

Chapitre 7 Exercice : Les données majic

Calculer à partir des tables fournies dans le fichier *majic.RData* issues des fichiers fonciers et du recensement de la population un indicateur d'étalement urbain entre 2009 et 2014 à la commune et à l'epci sur la région Pays de la Loire. La méthode utilisée sera celle du CEREMA. On peut consulter le rapport ici.

Le référentiel des communes a changé sur la période, dans un seul sens : il y a eu des fusions.

La table *com2017* permet de rattacher toute commune ayant existé sur la région à sa commune de rattachement dans la carte communale 2017.

Les surface artificialisé se calculent comme cela à partir de la typologie d'occupation du sol de majic : SA=dcnt07+dcnt09+ dcnt10+dcnt11+dcnt12+dcnt13.

Deux indices à calculer :

- un indice d'étalement urbain simple= Evolution de la surface artificialisée / Evolution de la population
- un indice d'étalement urbain avancé en classes

Classe 1	Régression des surfaces artificialisées avec gain de population (%TA<0 et %P>=0)
Classe 2a	Croissance de la population supérieure ou égale à la croissance des surfaces artificialisées cadastrées (%TA>=0 et %P>=0 et 0<=R<=1)
Classe 2b	Perte de population inférieure ou égale à la régression des surfaces artificialisées (%P<0 et %TA<0 et R>1)
Classe 2c	Recul des surfaces artificialisées inférieur au recul de la population (0<=R<=1 et %P<0)
Classe 3	Croissance des surfaces artificialisées relativement faible mais supérieure à la population (%P>=0 et 0<=%TA<=1,7 et R>1)
Classe 4	Croissance forte des surfaces artificialisées mais moins rapide que 2 fois celle de la population (%P>=0 et %TA>1,7 et 1<R<=2)
Classe 5	Croissance forte des surfaces artificialisées et deux fois plus rapide que celle de la population (%P>=0 et %TA>1,7 et R>2)
Classe 6	Croissance des surfaces artificialisées avec perte de la population (R<0 et %P<0)

R = (évolution de la surface artificialisée) / (évolution de la population)

%TA = évolution de la surface artificialisée

%P = évolution de la population

1,7% correspond à la croissance de la surface artificialisée observée entre 2009 et 2011 en France métropolitaine

Typologie de territoires en 6 classes

Chapitre 8 Aller plus loin

Quelques références :

- R for data science : http://r4ds.had.co.nz/
- Dplyr, Introduction : https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html
- Dplyr, manipulation de deux tables : https://cran.r-project.org/web/packages/dplyr/vignettes/two-table.html
- Tidyr : https://cran.r-project.org/web/packages/tidyr/tidyr.pdf
- Aide mémoire de Rstudio sur dplyr et tidyr : https://www.rstudio.com/wp-content/uploads/2016/01/data-wrangling-french.pdf
- Si vous préférez vous mettre à data.table https://s3.amazonaws.com/assets.datacamp.com/img/blog/data+table+cheat+sheet.pdf

Chapitre 9 Correction des exercices

9.1 Exercice 4.5.2

Exercice : Les données mensuelles sitadel

A partir du fichier sitadel de février 2017 (ROES_201702.xls), sur la région Pays de la Loire (code région 52), livrer un fichier contenant pour chaque mois, pour les logements individuels (i_AUT = ip_AUT + ig_AUT) :

- le cumul des autorisations sur 12 mois glissants(i_AUT_cum12)
- le taux d'évolution du cumul sur 12 mois (i_AUT_cum_evo, en %)
- la part de ce cumul dans celui de l'ensemble des logements autorisés (log_AUT), en pourcentage

```
rm (list = ls ())

sitadel <- read_excel("data/ROES_201702.xls", sheet = "AUT_REG",
                      col_types = c("text", "text", "numeric", "numeric", "numeric", "nume

sitadel52 <- filter (sitadel, REG == "52")

sitadel52 <- mutate (sitadel52,

                     i_AUT = ip_AUT + ig_AUT, # somme des logements individuels autoris
                     i_AUT_cum12 = roll_sumr (i_AUT, 12), # cumul sur 12 mois
                     i_AUT_cum12_lag12 = lag (i_AUT_cum12, 12), # décalage de 12 mois
                     i_AUT_cum12_delta = i_AUT_cum12 - i_AUT_cum12_lag12,
                     i_AUT_cum_evo = round (100 * i_AUT_cum12_delta /  i_AUT_cum12_lag1

                     log_AUT_cum12 = roll_sumr (log_AUT, 12), # somme des logements aut
                     i_AUT_cum_part = round (100 * i_AUT_cum12 / log_AUT_cum12, 1) # pc
                     )
```

9.2 Exercice 4.5.3

Convertir les données de la table exercice pertinentes au format date.

```
rm (list = ls ())
load (file = "data/FormationPreparationDesDonnées.RData")
exercice <- mutate (exercice,
                    date_prelevement = ymd (date_prelevement),
                    date_creation = ymd (date_creation),
                    date_formatee = format (date_prelevement, "%d/%m/%Y")) # plus joli,
```

9.3 Exercices 4.10

9.3.1 Sitadel

Sur les données “sitadel”, effectuer les opérations suivantes en utilisant l’opérateur %>% :

- les mêmes calculs que ceux réalisés sur la région 52, mais sur chacune des régions
- les aggrégations par année civile pour chacune des régions, puis leur taux d’évolution d’une année sur l’autre (exemple : (val2015-val2014)/val2014)

```
rm (list = ls())

sitadel <- read_excel ("data/ROES_201702.xls", sheet = "AUT_REG",
                       col_types = c ("text","text","numeric","numeric","numeric","numeri
  group_by (REG) %>%
  mutate (i_AUT = ip_AUT + ig_AUT,
          i_AUT_cum12 = roll_sumr (i_AUT, 12),
          i_AUT_cum12_lag12 = lag (i_AUT_cum12, 12),
          i_AUT_cum12_delta = i_AUT_cum12 - i_AUT_cum12_lag12,
          i_AUT_cum_evo = round (100 * i_AUT_cum12_delta /  i_AUT_cum12_lag12, 1),

          log_AUT_cum12 = roll_sumr (log_AUT, 12),
          i_AUT_cum_part = round (100 * i_AUT_cum12 / log_AUT_cum12, 1)
          )

sitadel <- read_excel ("data/ROES_201702.xls", sheet = "AUT_REG",
                       col_types = c ("text","text","numeric","numeric","numeric","numer
  mutate (annee = str_sub (date, 1, 4),
          i_AUT = ip_AUT + ig_AUT) %>%
  group_by (REG, annee) %>%
  summarise(
    log_AUT_cum = sum (log_AUT),
    i_AUT_cum = sum (i_AUT)) %>%
  ungroup () %>%
  group_by (REG) %>%
  mutate (i_AUT_cum_lag = lag (i_AUT_cum, 1), # décalage de 1 année
          i_AUT_cum_delta = i_AUT_cum - i_AUT_cum_lag,
          i_AUT_cum_evo = round (100 * i_AUT_cum_delta /  i_AUT_cum_lag, 1),# taux d'évo

          i_AUT_cum_part = round (100 * i_AUT_cum / log_AUT_cum, 1) # part de l'individ
          )
```

9.3.2 Pesticides

Sur les données “FormationPreparationDesDonnées.RData”, table “exercice” :

- calculer le taux de quantification pour chaque molécule (code_parametre), chacune des année : nombre de fois où elle a été retrouvée (code_remarque=1) sur le nombre de fois où elle a été cherchée (code_remarque = 1,2,7 ou 10)
    - créer la variable “annee”
    - créer la variable de comptage des présences pour chaque analyse (1=présent, 0=absent)
    - créer la variable de comptage des recherches pour chaque analyse (1=recherchée, 0=non recherchée)
    - pour chaque combinaison année x code_parametre, calculer le taux de quantification
- trouver pour chaque station, sur l’année 2016, le prélèvement pour lequel la concentration cumulée, toutes substances confondues, est la plus élevée (~ le prélèvement le plus pollué)
    - filtrer les concentrations quantifiées (code_remarque=1) et l’année 2016
    - sommer les concentrations (resultat_analyse) par combinaison code_station x code_prelevement
    - ne conserver que le prélèvement avec le concentration maximale

```
rm (list = ls ())
load (file = "data/FormationPreparationDesDonnées.RData")
taux_de_quantification <- exercice %>%
  mutate (year = year (date_prelevement),
          num = 1 * (code_remarque == 1),
          denom = 1 * (code_remarque %in% c (1,2,7,10))) %>%
  group_by (year, code_parametre) %>%
  summarise (taux_de_quantification = 100 * sum (num) / sum (denom))

datatable (taux_de_quantification)
```

Show 10 ▾ entries Search:

	year	code_parametre	taux_de_quantification
1	1991	1107	100
2	1991	1129	0
3	1991	1130	0
4	1991	1136	0
5	1991	1176	0
6	1991	1199	0
7	1991	1203	100
8	1991	1208	50
9	1991	1209	100
10	1991	1212	0

Showing 1 to 10 of 6,760 entries

Previous 1 2 3 4 5 … 676 Next

```
pire_echantillon_par_station_en_2016 <- exercice %>%
  filter (code_remarque == 1, year (date_prelevement) == 2016) %>%
  group_by (libelle_station, code_prelevement) %>%
      summarise (concentration_cumulee = sum (resultat_analyse)) %>%
  group_by (libelle_station) %>%
      filter (concentration_cumulee == max (concentration_cumulee)) %>%
  ungroup ()

datatable (pire_echantillon_par_station_en_2016)
```

Show 10 ▾ entries Search:

	libelle_station	code_prelevement	concentration_cumulee
1	ANGLE GUIGNARD-RETENUE	43003	0.04
2	ANXURE SAINT-GERMAIN-D'ANXURE	42230	0.381
3	APREMONT-RETENUE	42892	0.074
4	ARAIZE CHATELAIS	41450	0.044
5	ARON MOULAY	41357	0.1
6	AUBANCE LOUERRE	41567	0.099
7	AUBANCE MURS-ERIGNE	41540	0.448
8	AUBANCE SAINT-SATURNIN-SUR-LOIRE	41573	0.579
9	AUTHION LES PONTS-DE-CE	42532	0.27
10	AUTISE SAINT-HILAIRE-DES-LOGES	41998	0.048

Showing 1 to 10 of 191 entries

Previous 1 2 3 4 5 … 20 Next

9.4 Exercice 5.1

- reconstituer le dataframe “exercice” à partir des données contenues dans les tables “analyse”, “prelevement” et “station” (jointures)
- calculer le nombre d’analyses réalisées sur des molécules (code_parametre) présentes dans le référentiel
- produire une liste des **code_parametre** associés à des analyses mais absents du référentiel
- produire une table des analyses “orphelines”, c’est-à-dire qui ne correspondent pas à un prélèvement

```
rm (list = ls ())
load (file = "data/FormationPreparationDesDonnées.RData")

recalcul_exercice <- analyse %>%
  inner_join (prelevement) %>%
  inner_join (station) %>%
  mutate (date_creation = as.character (date_creation),
          annee = year (date_prelevement))

nb_analyses_presentes_dans_referentiel <- analyse %>%
  inner_join (parametre) %>%
  summarise (n = count (.)) %>%
  pull (n)

nb_analyses_presentes_dans_referentiel2 <- analyse %>%
  inner_join (parametre) %>%
  nrow ()

codes_modecules_absents_du_referentiel <- analyse %>%
  anti_join (parametre) %>%
  group_by (code_parametre) %>%
    tally ()

analyses_avec_code_prelevement_non_retrouve_dans_table_prelevement <- analyse %>%
  anti_join (prelevement)

analyse_avec_code_prelevement_non_retrouve_dans_table_prelevement2 <- analyse %>%
  filter(!(code_prelevement %in% unique(prelevement$code_prelevement)))
```

9.5 Exercice 7

Calculer à partir des tables fournies dans le fichier *majic.RData* issues des fichiers fonciers un indicateur d'étalement urbain entre 2009 et 2014 à la commune et à l'epci sur la région Pays de la Loire.

```
rm (list = ls ())
library(ggplot2)
load("data/majic.RData")
#pour chaque millésime de majic, on remet les données sur la nouvelle carte des territoi

majic_2009 <- bind_rows (majic_2009_com44, majic_2009_com49, majic_2009_com53, majic_20(
  left_join (com2017, by = c ("idcom" = "depcom")) %>%
  select (-idcom, -idcomtxt) %>%
  group_by (epci_2017, depcom2017) %>%
  summarise_all (funs (sum)) %>%
  ungroup %>%
  mutate (artif_2009=dcnt07+dcnt09+dcnt10+dcnt11+dcnt12+dcnt13) %>%
  select(-starts_with("dcnt"))


majic_2014 <- bind_rows (majic_2014_com44, majic_2014_com49, majic_2014_com53, majic_201
  left_join (com2017, by = c ("idcom" = "depcom")) %>%
  select (-idcom, -idcomtxt) %>%
  group_by (epci_2017, depcom2017) %>%
  summarise_all (funs (sum)) %>%
  ungroup %>%
  mutate (artif_2014=dcnt07+dcnt09+dcnt10+dcnt11+dcnt12+dcnt13) %>%
  select(-starts_with("dcnt"))


#on passe également les données de population sur la nouvelle carte des territoires

p_2009 <- population_2009 %>%
  left_join (com2017, by = c ("idcom" = "depcom")) %>%
  select (-idcom) %>%
  group_by (epci_2017, depcom2017) %>%
  summarise(population_2009=sum(Population)) %>%
  ungroup
p_2014 <-population_2014 %>%
  left_join (com2017, by = c ("idcom" = "depcom")) %>%
  select (-idcom) %>%
  group_by (epci_2017, depcom2017) %>%
  summarise(population_2014=sum(Population)) %>%
```

```
  ungroup

#indicateur à la commune
# on joint les 4 tables précédentes par commune et on calcul les indicateurs
etalement_urbain_commune <- majic_2009 %>%
  left_join(majic_2014) %>%
  left_join (p_2009) %>%
  left_join (p_2014) %>%
  mutate (evoarti = 100 * artif_2014 / artif_2009 - 100,
          evopop = 100 * population_2014 / population_2009 - 100,
          indicateur_etalement_simple=evoarti/evopop,
          indicateur_etalement_avance = case_when (
            evoarti < 0 & evopop >= 0 ~ "1",
            evoarti >= 0 & evopop >= 0 & (evoarti / evopop <= 1 | evopop==0) ~ "2a",
            evoarti < 0 & evopop < 0 & evoarti / evopop > 1 ~ "2b",
            evopop < 0 & evoarti / evopop >= 0 & evoarti / evopop <= 1 ~ "2c",
            evopop > 0 & evoarti > 0 & evoarti <= 4.9 & evoarti / evopop > 1 ~ "3",
            evopop > 0 & evoarti> 4.9 & evoarti / evopop > 1 & evoarti / evopop <= 2 ~ '
            evopop > 0 & evoarti> 4.9 & evoarti / evopop > 2 ~ "5",
            evopop < 0 & evoarti / evopop < 0 ~ "6"
          )
  )

#indicateur à l'epci
# on joint les 4 tables précédentes par commune, on aggrege les compteurs par epci et or

etalement_urbain_epci <- majic_2009 %>%
  left_join(majic_2014) %>%
  left_join (p_2009) %>%
  left_join (p_2014) %>%
  select(-depcom2017) %>%
  group_by(epci_2017) %>%
  summarise_all(funs(sum(.))) %>%
  mutate (evoarti = 100 * artif_2014 / artif_2009 - 100,
          evopop = 100 * population_2014 / population_2009 - 100,
          indicateur_etalement_simple=evoarti/evopop,
          indicateur_etalement_avance = case_when (
            evoarti < 0 & evopop >= 0 ~ "1",
```

```
            evoarti >= 0 & evopop >= 0 & (evoarti / evopop <= 1 | evopop==0) ~ "2a",
            evoarti < 0 & evopop < 0 & evoarti / evopop > 1 ~ "2b",
            evopop < 0 & evoarti / evopop >= 0 & evoarti / evopop <= 1 ~ "2c",
            evopop > 0 & evoarti > 0 & evoarti <= 4.9 & evoarti / evopop > 1 ~ "3",
            evopop > 0 & evoarti> 4.9 & evoarti / evopop > 1 & evoarti / evopop <= 2 ~ '
            evopop > 0 & evoarti> 4.9 & evoarti / evopop > 2 ~ "5",
            evopop < 0 & evoarti / evopop < 0 ~ "6"
          )
  )
# Deux graphiques de visualisation de notre indicateur
ggplot(data=etalement_urbain_epci) +
  geom_point(aes(x=evoarti,y=evopop,color=indicateur_etalement_avance))+
  theme_minimal() +
  labs(title="Indicateur d'étalement urbain sur les epci de la région Pays de la Loire",

ggplot(data=etalement_urbain_commune) +
  geom_point(aes(x=evoarti,y=evopop,color=indicateur_etalement_avance),size=.5,alpha=.5)
  theme_minimal()+
  labs(title="Indicateur d'étalement urbain sur les communes de la région Pays de la Loi
```