Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN INGÉNIEUR DE GESTION À FINALITÉ SPÉCIALISÉE EN DATA SCIENCE

Application de la Business Intelligence à la base de données Microsoft Northwind

NICOLAY, Julien

Award date:
2023

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 03. May. 2024

Application de la Business Intelligence à la base de données Microsoft Northwind

Julien NICOLAY

Directeur: Prof. S. FAULKNER

Mémoire présenté
en vue de l'obtention du titre de
Master 120 en ingénieur de gestion, à finalité spécialisée
en data science

ANNEE ACADEMIQUE 2022-2023

Université de Namur, ASBL
Faculté des Sciences économiques, sociales et de gestion – Département des Sciences de gestion

Rempart de la Vierge 8, B-5000 Namur, Belgique, Tel. +32 [0]81 72 49 58/48 41

Avant-propos

Ce mémoire marque la conclusion de mon Master en Ingénieur de gestion à finalité spécialisée en Data Science, symbolisant ainsi l'aboutissement d'années d'efforts et de détermination. Avant de plonger dans le vif du sujet, il est essentiel pour moi de consacrer quelques mots de reconnaissance.

Je tiens tout d'abord à exprimer ma sincère gratitude envers mon promoteur, le Professeur Stéphane Faulkner, dont l'accompagnement inestimable et les précieux conseils ont grandement contribué à la réalisation de ce travail mais également à l'enrichissement de mon bagage académique dans son ensemble.

Je saisis également cette opportunité pour chaleureusement remercier mes soeurs, dont les encouragements constants ont été une source de motivation inépuisable. Un remerciement tout particulier revient à ma maman pour son soutien inconditionnel et ses innombrables et minutieuses relectures tout au long de mon cursus.

Enfin, une dernière pensée va vers les amis qui m'ont accompagné, dans les études comme en dehors. Leur présence a apporté une touche de légèreté aux moments de rigueur et leur soutien a été une bouée d'espoir dans les moments les plus difficiles.

Résumé/Summary

Résumé

Dans un monde où les entreprises se retrouvent inondées de données, la capacité à extraire des informations significatives à partir de ces données devient un atout stratégique essentiel. La Business Intelligence (BI) fournit des outils et des méthodologies pour transformer ces données brutes en connaissances exploitables. L'objectif de ce mémoire est d'offrir une compréhension approfondie des principaux concepts de la BI et de démontrer leur application pratique à travers des exemples concrets basés sur la base de données Northwind. Ce mémoire se veut avant tout être un outil pédagogique visant à présenter de façon avancée différents concepts clés de la Business Intelligence. L'audience ciblée comprend des étudiants de niveau universitaire possédant déjà des acquis en termes de bases de données relationnelles et de leur traitement en SQL mais n'étant pas initiés au domaine de la BI.

Summary

In a world where companies find themselves flooded with data, the ability to extract meaningful information from this data is becoming an essential strategic asset. Business Intelligence (BI) provides the tools and methodologies to transform this raw data into actionable knowledge. The aim of this thesis is to provide an in-depth understanding of the main concepts of BI, and to demonstrate their practical application through concrete examples based on the Northwind database. Above all, this thesis is intended as a pedagogical tool to present various key Business Intelligence concepts in an advanced way. The target audience includes university-level students already familiar with relational databases and their handling in SQL, but who are not initiated into the field of BI.

Table des matières

Table des figures

1 Introduction

1.1 Contexte et objectifs

Dans un monde où les entreprises se retrouvent inondées de données, la capacité à extraire des informations significatives à partir de ces données devient un atout stratégique essentiel. La Business Intelligence (BI) fournit des outils et des méthodologies pour transformer ces données brutes en connaissances exploitables. Comme le mentionnent Grossmann et Rinderle-Ma, 2015, *"presque toutes les moyennes et grandes entreprises et organisations utilisent déjà des logiciels de BI ou prévoient de le faire dans les prochaines années."*. L'objectif de ce mémoire est d'offrir une compréhension approfondie des principaux concepts de la BI et de démontrer leur application pratique à travers des exemples concrets basés sur la base de données Northwind, tout en s'appuyant sur l'approche de Vaisman et Zimányi, 2014. Ce mémoire se veut avant tout être un outil pédagogique visant à présenter de façon avancée différents concepts de la Business Intelligence. L'audience ciblée comprend des étudiants de niveau universitaire possédant déjà des acquis en termes de bases de données relationnelles et de leur traitement en SQL mais n'étant pas initiés au domaine de la BI.

1.2 Structure

Ce mémoire est structuré de manière à guider le lecteur à travers un processus de Business Intelligence, en fournissant des explications théoriques et des illustrations pratiques. Dans la Section 2, nous présenterons la base de données opérationnelle Northwind, qui servira de cas d'étude tout au long du mémoire. La Section 3 introduira différents concepts clés de la Business Intelligence, notamment les systèmes OLAP (OnLine Analytical Processing), les data warehouses, le cube multidimensionnel et ses composantes telles que les faits, mesures, dimensions et hiérarchies.

La Section 4 explorera le processus d'ETL (Extract, Transform & Load) qui est central dans la transformation des données brutes en informations exploitables. Plusieurs opérations OLAP que nous pouvons appliquer à un cube multidimensionnel seront conceptuellement expliquées et illustrées de façon détaillée dans la Section 5.

La Section 6 mettra en application les opérations OLAP au moyen des langages SQL et MDX (Multi-Dimensional eXpressions). Des exemples concrets illustreront chaque opération et mettront en évidence les différences d'approche entre ces deux langages.

Pour conclure, la Section 7 reprendra les points clés abordés tout au long du mémoire ainsi que les contributions apportées par celui-ci. Nous n'hésiterons pas à mentionner des pistes potentielles pour de futurs travaux.

Ensemble, ces sections fourniront une vue globale des concepts fondamentaux de la Business Intelligence, tout en mettant en lumière leur application pratique à travers des exemples basés sur la base de données Northwind.

2 Présentation de la base de données Northwind

Dans cette section, nous présenterons le jeu de données Northwind. Il s'agit notamment de l'ensemble de données servant d'exemple principal dans Vaisman et Zimányi, 2014. Nous nous reposerons également sur celui-ci pour étayer les exemples présents tout au long de ce travail.

2.1 Conception

Northwind a été introduit pour la première fois par Microsoft dans les années 1990. L'objectif initial était de servir d'exemple de base de données relationnelle afin de démontrer les fonctionnalités du système de gestion de bases de données Microsoft Access. Par la suite, la base de données fictive a également été intégrée à Microsoft SQL Server. Northwind est devenu un ensemble de données populaire et est largement utilisé dans le domaine de la recherche en sciences des données et des bases de données.

2.2 Contenu

Tout d'abord, il est important de noter que pour les besoins de ce travail, les noms de tables et de colonnes présentes dans le jeu de données, initialement en anglais, ont été traduits en français. Le script permettant de créer les tables reprises dans la base de données se trouve dans l'annexe A [1].

Northwind est une entreprise fictive dont l'activité commerciale est l'importation et l'exportation de denrées alimentaires. Elle sert de grossiste à un réseau de points de vente au détail dans le monde entier. Ainsi la base de données est constituée de tables permettant d'analyser les opérations de vente de l'entreprise. Les tables les plus importantes sont les suivantes :

- **Produits** : La table *Produits* reprend les informations essentielles des produits commercialisés par l'entreprise Northwind telles que le nom du produit, sa catégorie, son fournisseur, son prix unitaire ou encore les unités en stock et en commande.
- **Employés** : La table *Employes* reprend les informations essentielles des employés de l'entreprise, notamment leurs noms et prénoms, leurs dates de naissance et d'embauche, l'adresse de leur domicile ou encore leur supérieur hiérarchique.
- **Clients** : La table *Clients* reprend les informations essentielles des clients, à savoir le nom de l'entreprise, le nom et le poste de la personne de contact, la localisation de l'entreprise et les numéros de téléphone et de fax de l'entreprise.

1. Ce script est adapté de celui présent sur un dépôt GitHub de Microsoft, disponible à l'adresse URL suivante : https://github.com/microsoft/sql-server-samples/tree/master/samples/databases/northwind-pubs

- **Fournisseurs** : La table *Fournisseurs* reprend les informations essentielles des fournisseurs. Celles-ci sont similaires aux informations concernant les clients de Northwind.
- **Transporteurs** : La table *Transporteurs* reprend les informations essentielles des transporteurs, à savoir le nom et le numéro de téléphone de l'entreprise.
- **Commandes** : La table *Commandes* reprend les informations essentielles des commandes. Ces informations concernent notamment le client, l'employé responsable de la commande, les dates de commande, requise et d'envoi ainsi que toutes les informations relatives à la livraison.
- **Commande Détails** : La table *Commande Details* concerne les détails spécifiques à chaque commande. Chaque ligne est identifiée par un produit spécifique pour une commande spécifique et reprend le prix unitaire du produit, la quantité commandée et le montant d'une éventuelle ristourne.

La Figure 2.1 représente le schéma relationnel de la base de données Northwind.

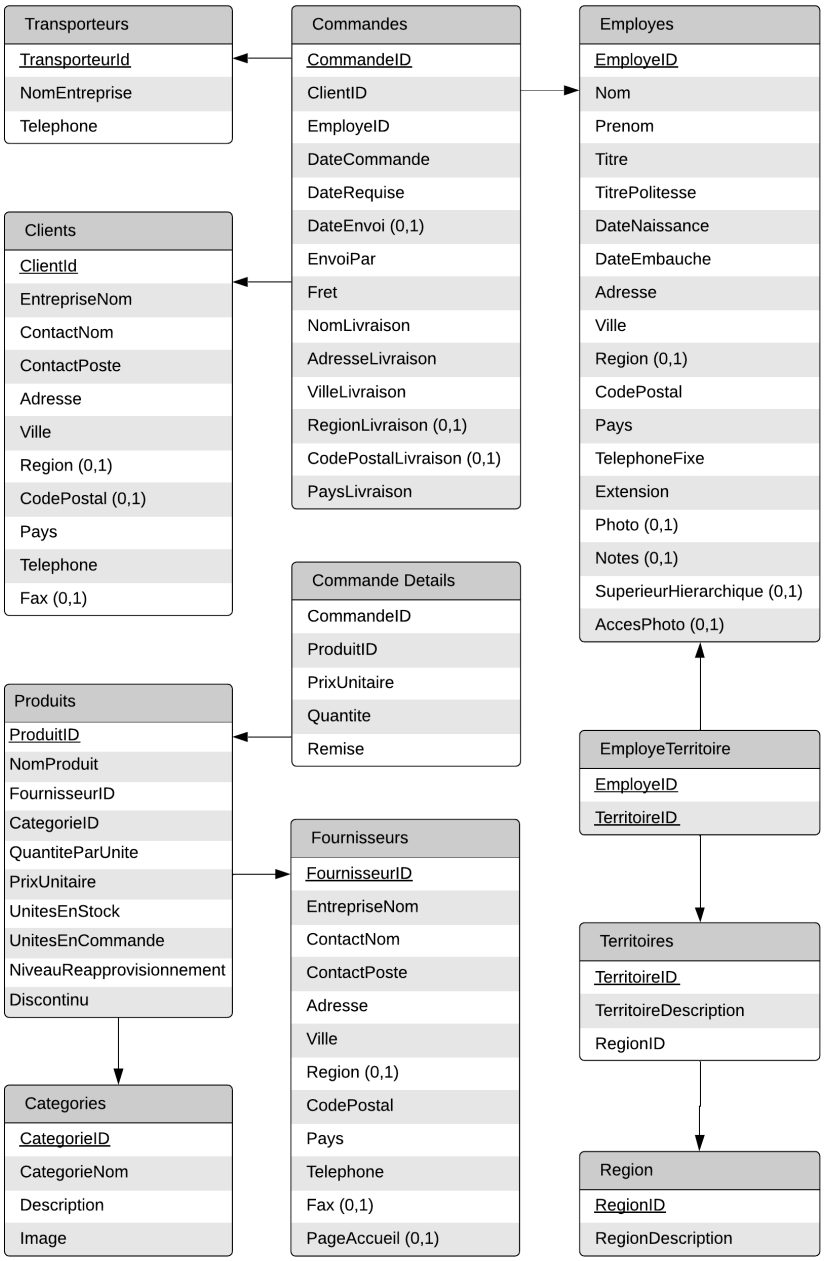


FIGURE 2.1 – Schéma relationnel de la base de données Northwind

3 Introduction à la Business Intelligence

Selon Vaisman et Zimányi, 2014, la Business Intelligence en entreprise *"comprend un ensemble de méthodologies, de processus, d'architectures et de technologies qui transforment les données brutes en informations significatives et utiles pour la prise de décision"*. Nous aborderons dans cette section plusieurs concepts théoriques nécessaires à une compréhension optimale. Ceux-ci seront illustrés sur base du cas d'étude Northwind.

3.1 Systèmes OLTP et OLAP (Sinha, 2021)

La base de données présentée dans la Section 2 correspond à une approche opérationnelle des bases de données. De telles bases de données s'inscrivent dans le paradigme des systèmes **OLTP** (**OnLine Transaction Processing**). Celui-ci est essentiellement basé sur l'exécution de transactions quotidiennes. Dans le cadre de la base de données Northwind, de telles transactions reviendraient, par exemple, à encoder une nouvelle commande et créer un bon de commande si le point de réapprovisionnement des produits commandés est atteint.

Depuis maintenant plusieurs décennies, l'intérêt des entreprises s'est tourné vers la capacité à analyser de gros volumes de données historisées afin de les assister dans le processus de prise de décision. Objectif pour lequel les systèmes du paradigme OLTP ne sont pas conçus. C'est pour cela qu'a été introduit le paradigme **OLAP** (**OnLine Analytical Processus**). Là où les systèmes OLTP étaient centrés sur les transactions, les systèmes OLAP sont focalisés sur des besoins d'analyse et doivent par conséquent être capables de supporter une forte charge de requêtes analytiques. Dans le cadre de la base de données Northwind, de telles requêtes reviendraient, par exemple, à demander le montant total des ventes par produit et par client ou encore les produits les plus commandés pour un mois spécifique.

Ceci nous amène donc aux concepts de data warehouse et de cube multidimensionnel qui seront abordés dans les sous-sections suivantes.

La Table 3.1 reprend plusieurs critères différenciant les systèmes opérationnels (OLTP) des data warehouses (OLAP).

Critère	Systèmes opérationnels	Data warehouses
Utilisateurs	Opérateurs, employés de bureau	Managers, cadres
Utilisation	Prévisible, répétitif	Ad hoc, non structurée (selon le besoin)
Contenu des données	Données actuelles et détaillées	Données historiques et résumées
Organisation des données	En fonction des besoins opérationnels	En fonction des besoins d'analyse
Structures de données	Optimisées pour les petites transactions	Optimisées pour les requêtes complexes
Fréquence d'accès	Élevée	De moyenne à faible
Type de transactions	Read, Insert, Update, Delete	Read
Nombre d'enregistrements par accès	Peu	Nombreux
Temps de réponse	Court	Peut être long
Accès concurrents	Élevé	Faible
Utilisation de verrouillage	Nécessaire	Non nécessaire
Fréquence de mise à jour	Élevée	Aucune
Redondance des données	Faible (tables normalisées)	Élevée (tables non normalisées)
Modélisation des données	UML, modèle ER	Modèle multidimensionnel

TABLE 3.1 – Comparaison entre systèmes opérationnels (OLTP) et data warehouses (OLAP) (VAISMAN et ZIMÁNYI, 2014)

3.2 Data Warehouse

INMON et HACKATHORN, 1994 définissent un data warehouse comme étant *"une collection de données orientées sujet, intégrées, variables dans le temps et non volatiles, destinée à soutenir le processus de prise de décision des gestionnaires"*. Détaillons ces différentes caractéristiques :

- Contrairement aux bases de données opérationnelles qui mettent l'accent sur les tâches spécifiques que le système doit effectuer, les données d'un data warehouse sont **orientées sujet** et se concentrent donc sur les besoins analytiques des différents départements d'une entreprise. Dans le cadre du data warehouse Northwind, les besoins analytiques sont concentrés sur les commandes effectuées par les clients.
- Le fait que les données d'un data warehouse soient **intégrées** signifie qu'elles sont généralement issues de sources diverses et variées telles que des bases de données opérationnelles, des fichiers plats ou encore des bases de données NoSQL. Ceci implique la nécessité de prendre en compte de potentielles différences en termes de définition de données et de contenu. Par exemple, deux sources pourraient contenir

des champs présentant un même nom mais n'ayant pas la même signification. En ce qui concerne, le data warehouse Northwind, nous utiliserons comme sources la base de données opérationnelle présentée dans la Section 2, un fichier .txt ainsi qu'un fichier .xml. Ce point est abordé plus en détails dans la Section 4.1.

- L'**historisation** des données implique la possibilité de retenir plusieurs valeurs pour une même information à différents moments dans le temps. Ceci s'explique par la présence d'une dimension temporelle avec laquelle les données sont associées. À l'opposé, les systèmes opérationnels, centrés sur les opérations quotidiennes, peuvent ne pas avoir de support temporel explicite.
- La **non-volatilité** indique que la durabilité des données du data warehouse est garantie pour une durée plus longue que ce que permettent généralement les bases de données opérationnelles. Cela est assuré par le fait de ne pas pouvoir effectuer de suppression ou de modification sur les données du data warehouse. Alors que les données des systèmes opérationnels sont souvent stockées pour une durée limitée de quelques mois, un data warehouse permet de recueillir des données couvrant plusieurs années, en général 5 à 10 ans voire même plus. Ainsi, dans notre cas d'étude Northwind, les données des commandes s'étalent sur 2 années.

3.3 Modélisation multidimensionnelle

Le modèle multidimensionnel, aussi appelé cube multidimensionnel ou hypercube, permet de visualiser les données dans un espace à n dimensions. Ce concept est central pour les systèmes OLAP tels que les data warehouses. Un cube se définit par la présence de faits et de dimensions. En organisant les données en faits et en dimensions, la modélisation dimensionnelle fournit une structure qui facilite l'interrogation, l'analyse et l'établissement de rapports dans un data warehouse. La Figure 3.1 représente une portion des données de Northwind et nous servira d'illustration dans les sous-sections suivantes.

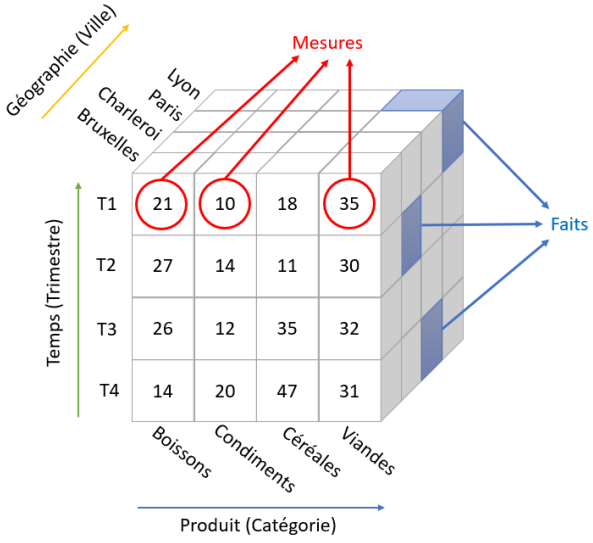


FIGURE 3.1 – Cube OLAP à 3 dimensions

3.3.1 Faits et mesures

Les faits sont représentés par les cellules du cube multidimensionnel (Figure 3.1). Chaque fait symbolise un événement singulier que l'entreprise souhaite analyser pour en tirer des enseignements. Ceux-ci sont caractérisés par différentes mesures correspondant généralement à des valeurs numériques permettant d'effectuer des analyses quantitatives. Dans le cas de Northwind, les faits concernent donc les ventes réalisées tandis que les différentes mesures que nous pouvons retrouver sont notamment la quantité vendue, le prix unitaire du produit vendu ou encore le montant de la vente. Pour des questions de lisibilité, le cube de la Figure 3.1 ne présente que la mesure Quantité.

En fonction du niveau de précision selon lequel les données sont analysées, les mesures présentes dans les faits peuvent être amenées à être agrégées. De ce fait, celles-ci sont classifiées de la manière suivante (Vaisman et Zimányi, 2014) :

- **Mesures additives** : Il s'agit de mesures pour lesquelles une agrégation par somme peut faire sens quelle que soit la dimension utilisée. Dans le cadre de Northwind, la mesure *Quantité* est une mesure additive. En effet, elle peut être agrégée selon n'importe quelle dimension sans pour autant perdre de sens.
- **Mesures semi-additives** : Il s'agit de mesures pour lesquelles une agrégation par somme peut faire sens pour une partie des dimensions et non pour l'ensemble des dimensions. L'illustration la plus parlante de mesure semi-additive est l'inventaire d'un magasin. En effet, il ferait, par exemple, peu de sens d'additionner les inventaires sur deux années différentes.
- **Mesures non additives** : Il s'agit de mesures qui ne peuvent être agrégées par somme, quelle que soit la dimension utilisée. Dans le cas de Northwind, la mesure *PrixUnitaire* constitue un bon exemple de mesure non additive.

3.3.2 Dimensions

Les dimensions sont les attributs descriptifs qui fournissent le contexte et décrivent les caractéristiques des faits. En d'autres termes, les dimensions sont les axes par lesquels les faits peuvent être analysés et découpés. Dans le cube de la Figure 3.1, les dimensions représentent les arêtes du cube. *"Les dimensions fournissent le contexte (qui, quoi, où, quand, pourquoi et comment) entourant un fait"* (Kimball et Ross, 1996). En ce qui concerne Northwind, des dimensions appropriées seraient les produits, clients, employés, fournisseurs ou encore les transporteurs. De manière plus générale, on retrouve régulièrement une dimension temporelle ainsi qu'une dimension géographique.

Une dimension peut être scindée en différents niveaux qui correspondent au niveau de détail auquel les données ont été agrégées pour cette même dimension. La Figure 3.1 illustre notamment les dimensions *Géographie*, *Temps* et *Produit*, respectivement aux niveaux *Ville*, *Saison* et *Catégorie*. L'exemple le plus parlant est sans doute la dimension *Temps* qui peut, par exemple, avoir pour niveaux *Jour, Mois* ou encore *Année*. Aussi, les

instances d'une dimension sont appelées des membres. Ainsi, *Hiver, Printemps, Automne* et *Été* constituent des membres de la dimension *Temps* au niveau *Saison*.

Enfin, des attributs permettent d'apporter une meilleure contextualisation aux dimensions. Par exemple, la dimension *Produit* dans le cadre de Northwind a notamment comme attributs *NomProduit* ou *QuantiteParUnite*.

3.3.3 Hiérarchies (Malinowski et Zimányi, 2006)

Comme expliqué dans la sous-section précédente, les dimensions peuvent être organisées par niveaux, nous utilisons alors le terme de hiérarchies. Celles-ci sont des éléments cruciaux pour les systèmes OLAP car elles permettent une analyse à différents niveaux de détail. Selon la terminologie, les niveaux supérieurs sont appelés niveaux parent tandis que les niveaux inférieurs sont appelés niveaux enfant. Plusieurs types de hiérarchies peuvent être modélisés selon le modèle multidimensionnel. Nous allons maintenant tâcher d'en introduire les principaux. Notons toutefois que tous les types de hiérarchie présentés ne se retrouvent pas nécessairement dans le cube Northwind.

- Une **hiérarchie équilibrée** ne présente qu'un seul chemin permettant de parcourir ses différents niveaux. Dans le cas de Northwind, c'est le cas de la hiérarchie *Produit –> Categorie* (Figure 3.3). Ainsi, tous les membres d'un niveau parent possèdent au moins un membre dans un niveau enfant tandis que chaque membre d'un niveau enfant appartient exactement à un niveau parent.

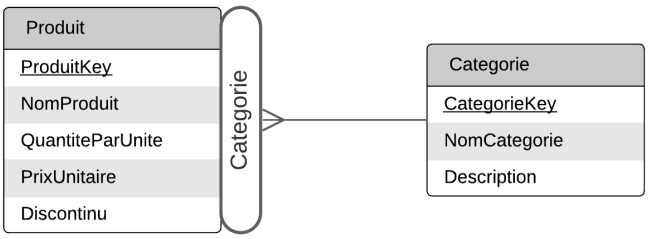


Figure 3.2 – Représentation schématique d'une hiérarchie équilibrée

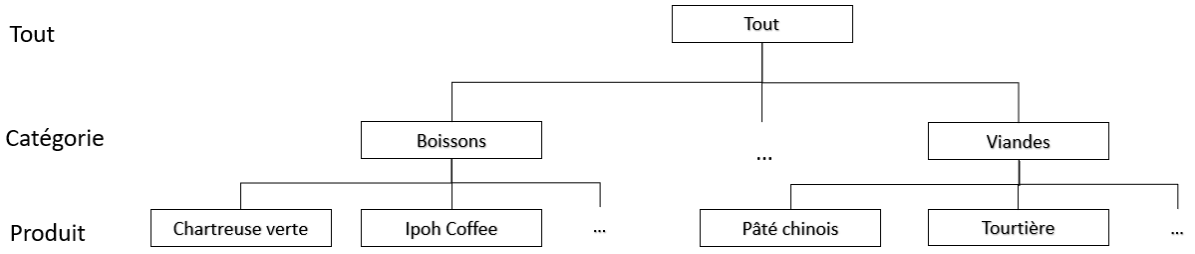


Figure 3.3 – Exemple d'instances de la hiérarchie équilibrée *Produit –> Categorie*

- Une **hiérarchie non équilibrée** ne présente également qu'un seul chemin pour parcourir ses différents niveaux. Cependant, au moins un de ses niveaux est optionnel. Cela signifie qu'un membre d'un niveau parent pourrait ne pas être associé à un membre d'un niveau enfant. Ce type de hiérarchie comprend le cas particulier que sont les **hiérarchies récursives**. Dans une hiérarchie récursive, un même niveau occupe à la fois les rôles de niveau parent et de niveau enfant. C'est notamment le cas de la dimension *Employe* dans le data warehouse Northwind (Figure 3.5).

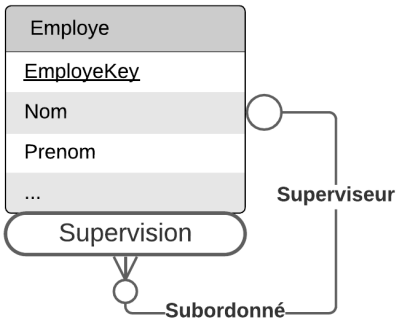


FIGURE 3.4 – Représentation schématique d'une hiérarchie récursive

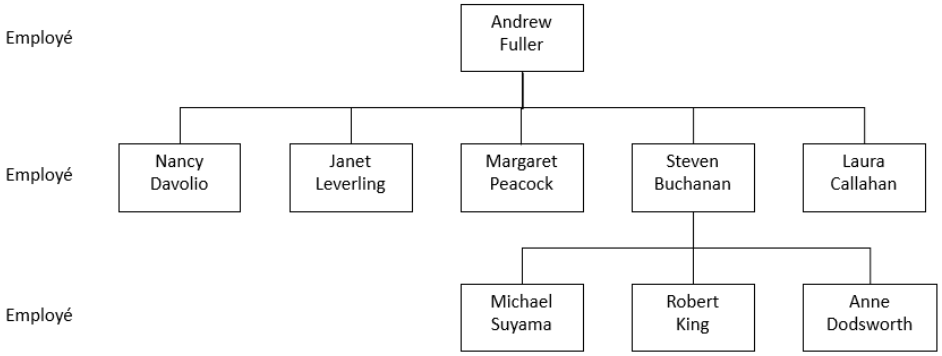


FIGURE 3.5 – Exemple d'instances de la hiérarchie récursive *Employe*

- Lorsque les membres d'un même niveau sont de natures différentes, nous sommes face à une **hiérarchie généralisée**. Dans ce type de hiérarchie, nous retrouvons plusieurs chemins exclusifs partageant au minimum le niveau final. Cela peut notamment arriver dans le cas où les clients d'une entreprise sont aussi bien des entreprises que des personnes physiques. Cet exemple est illustré au niveau des instances par la Figure 3.7. Il faut donc considérer deux chemins d'agrégation : *Client –> Secteur –> Branche* pour les entreprises et *Client –> Profession –> Branche* pour les personnes physiques.

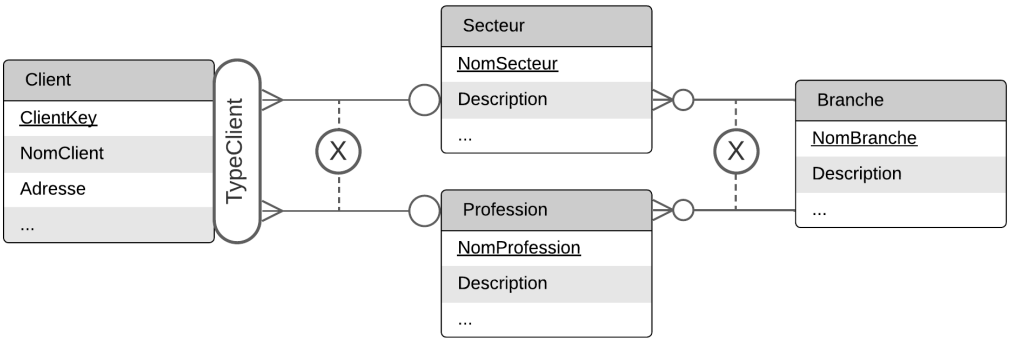


FIGURE 3.6 – Représentation schématique d'une hiérarchie généralisée

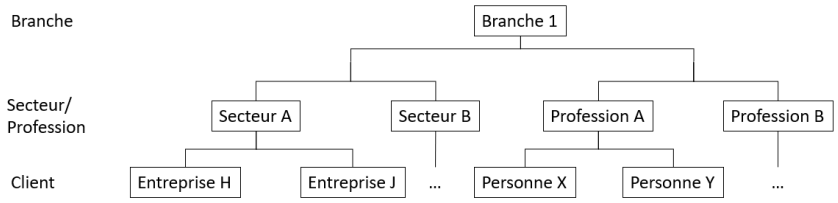


FIGURE 3.7 – Exemple d'instances d'une hiérarchie généralisée

- Un cas particulier des hiérarchies généralisées concerne les **hiérarchies non couvrantes**, aussi appelées hiérarchies en dent de scie. Dans le contexte de Northwind, cela correspond à la hiérarchie *Ville* –> *État* –> *Région* –> *Pays*. En effet, alors que des pays tels que la Belgique sont scindés en régions, d'autres pays comme l'Allemagne ne le sont pas. Aussi, il arrive que certains pays tels que Singapour ne possèdent ni régions ni provinces ni états. De ce fait, une hiérarchie non couvrante est obtenue en passant un ou plusieurs niveaux intermédiaires d'une hiérarchie généralisée. La Figure 3.9 nous donne un exemple d'instances pour une telle hiérarchie.

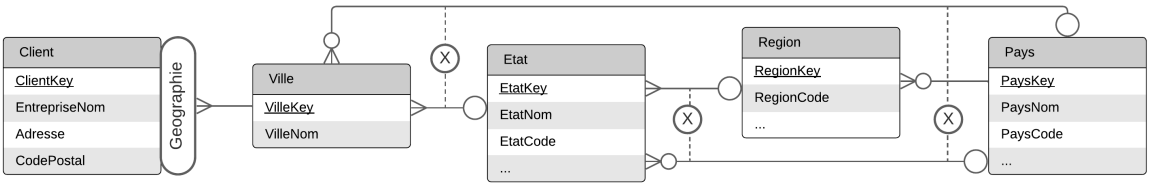


FIGURE 3.8 – Représentation schématique d'une hiérarchie non couvrante

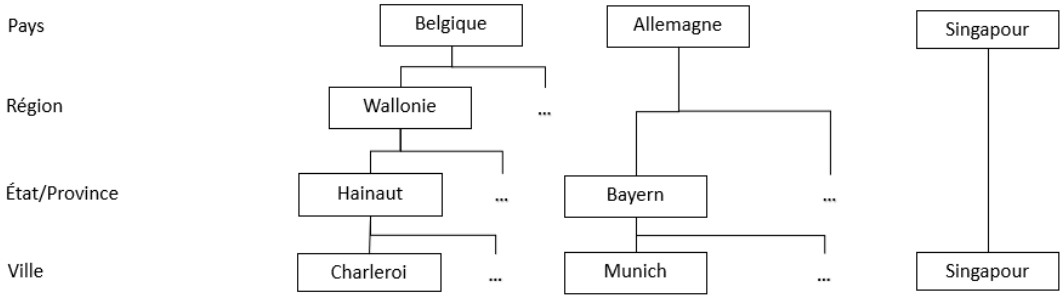


FIGURE 3.9 – Exemple d'instances de la hiérarchie non couvrante *Ville –> État –> Région –> Pays*

- **N**ous rencontrons des **hiérarchies alternatives** dans le cas de plusieurs hiérarchies non exclusives partageant au minimum le niveau final. Un exemple commun de hiérarchies alternatives arrive lorsque les données d'une entreprise peuvent être analysées selon un calendrier classique ou selon un calendrier fiscal (Figure 3.11).

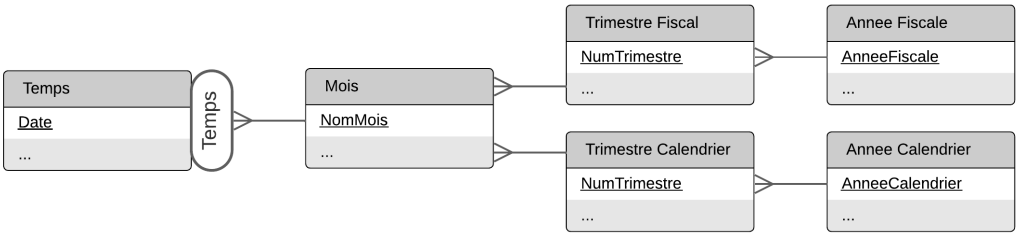


FIGURE 3.10 – Représentation schématique de hiérarchies alternatives

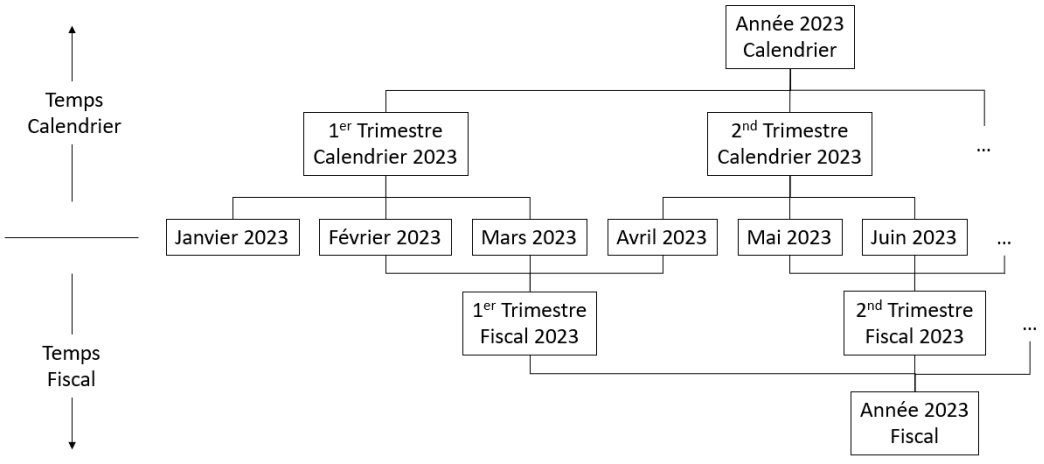


FIGURE 3.11 – Exemple d'instances de hiérarchies alternatives

- Nous sommes confrontés à des **hiérarchies parallèles** lorsqu'une dimension peut être associée à plusieurs hiérarchies correspondant à des critères d'analyse différents. Celles-ci peuvent être dépendantes ou indépendantes si elles comportent ou non un niveau commun. Bien que ce type de hiérarchies présente des similarités avec les hiérarchies alternatives, il est nécessaire de les différencier. En effet, alors qu'il serait inutile de combiner des niveaux issus de différentes hiérarchies alternatives, cela est totalement réalisable dans le cas de hiérarchies parallèles.

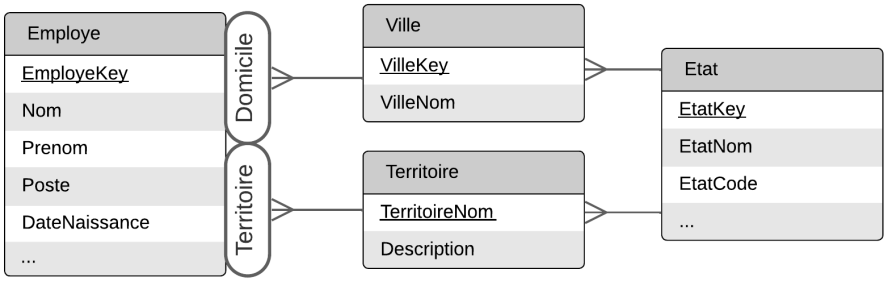


FIGURE 3.12 – Représentation schématique de hiérarchies parallèles

- Les hiérarchies précédemment illustrées se basent sur l'hypothèse d'une relation One-to-Many entre les niveaux parent et les niveaux enfant, à savoir qu'un membre d'un niveau parent peut être associé à plusieurs membres de niveau enfant tandis qu'un membre de niveau enfant ne peut être associé qu'à un seul membre de niveau parent. Cependant, nous pouvons également rencontrer le cas de relations Many-to-Many. Ainsi, lorsqu'une hiérarchie comporte au moins une relation Many-to-Many, nous la considérons comme une **hiérarchie non stricte**. Dans le contexte de Northwind, nous pouvons observer une telle relation entre les employés et les territoires qui leur sont assignés. La Figure 3.14 présente un exemple d'instances constituant cette hiérarchie non-stricte.

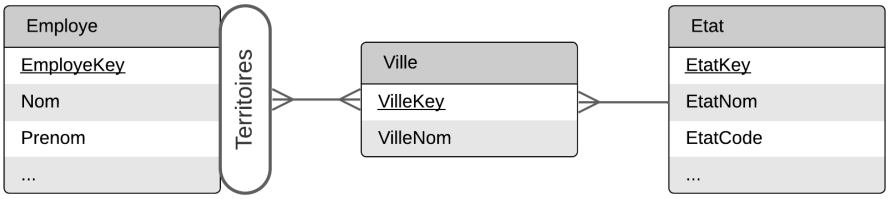


FIGURE 3.13 – Représentation schématique d'une hiérarchie non stricte

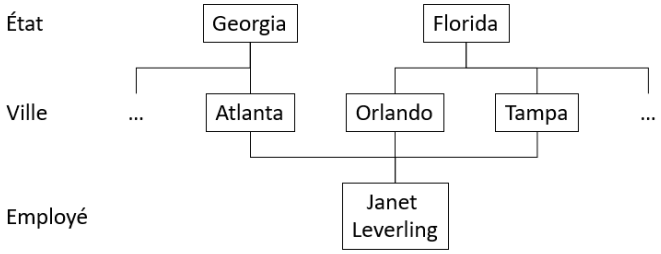


FIGURE 3.14 – Exemple d'instances de hiérarchie non stricte basée sur Northwind

3.4 Stockage de données en BI

En Business Intelligence, les données multidimensionnelles peuvent être stockées selon différentes approches. Cette section se penche sur les caractéristiques des trois approches principales : ROLAP, MOLAP et HOLAP.

Les systèmes **ROLAP** (Relational OLAP) stockent les données multidimensionnelles dans des tables relationnelles. Les agrégats sont pré-calculés dans les tables relationnelles afin d'augmenter les performances lors des requêtes analytiques. Les opérations OLAP sont effectuées sur ces tables, ce qui peut donner lieu à des instructions SQL complexes. Toute la gestion des données repose sur le SGBD relationnel sous-jacent. Les systèmes ROLAP sont bien standardisés et offrent une grande capacité de stockage. Les performances peuvent toutefois être amoindries lorsque nous interrogeons de grandes quantités de données car les requêtes SQL impliquent des agrégations à la volée et des jointures complexes.

Les systèmes **MOLAP** (Multidimensional OLAP), quant à eux, stockent les cubes de données dans des tableaux multidimensionnels, en combinaison avec des techniques de hachage et d'indexation. Les opérations OLAP peuvent ainsi être mises en œuvre efficacement car elles sont très naturelles et simples à réaliser. Le MDX, un langage de requête, a été spécifiquement conçu pour la réalisation d'opérations OLAP sur des structures MOLAP. Celui-ci sera introduit dans la Section 6. La gestion des données est assurée par le moteur multidimensionnel, qui offre généralement une capacité de stockage inférieure à celle des systèmes ROLAP. Cependant, les systèmes MOLAP présentent de meilleures performances lors de l'interrogation et l'agrégation de données multidimensionnelles. Habituellement, nous les utilisons pour interroger des data marts [2] dont le nombre de dimensions est relativement faible, moins de dix en règle générale.

Enfin, les systèmes **HOLAP** (Hybrid OLAP) visent à tirer profit des approches ROLAP et OLAP. De ce fait, ils stockent de grands volumes de données détaillées dans des bases de données relationnelles tout en conservant des données agrégées selon l'approche MOLAP. Ainsi, les systèmes HOLAP permettent de trouver un équilibre entre les capacités de traitement de MOLAP et les capacités de stockage de ROLAP.

2. Un data mart est un data warehouse simplifié, centré sur une activité spécifique de l'entreprise (ventes, finance, marketing,...)

3.5 Implémentations ROLAP

Dans ce travail nous nous concentrerons sur l'approche ROLAP pour le stockage des données. Cette approche comprend quatre méthodes offrant une représentation relationnelle d'un modèle multidimensionnel. Cette section sera consacrée à la présentation de ces quatre implémentations.

3.5.1 Schéma en étoile

Le type de représentation relationnelle le plus simple et le plus couramment utilisé est le **schéma en étoile**. Ce type de schéma est caractérisé par l'usage de dénormalisation. Ce qui signifie que les tables de dimensions se présentent généralement sous une forme entièrement dénormalisée, avec des relations hiérarchiques minimales ou inexistantes entre les attributs. Ainsi, un modèle en étoile classique sera composé d'une table de faits unique reliée à une seule table par dimension. La Figure 3.15 présente un potentiel schéma en étoile basé sur les données de Northwind.

La structure dénormalisée du schéma en étoile permet des requêtes et des agrégations rapides. Comme les tables de dimensions ne sont pas normalisées, moins de jointures sont nécessaires pour récupérer les données, ce qui améliore les performances des requêtes.

En revanche, ce type de modélisation présente un haut taux de redondance. En effet, les dimensions étant stockées dans des tables dénormalisées, les attributs des niveaux supérieurs des dimensions seront potentiellement encodés à de nombreux endroits différents. Aussi, cela affecte grandement la visibilité des hiérarchies présentes dans les différentes dimensions.

3.5.2 Schéma en flocon

Dans un **schéma en flocon**, les tables de dimensions sont normalisées, ce qui signifie que certains attributs sont normalisés dans des tables supplémentaires. Il en résulte une structure qui ressemble à un flocon de neige, avec une table de faits unique au centre et plusieurs tables pour une même dimension (typiquement une table par niveau). Nous pouvons, par exemple, constater dans la Figure 3.16 que la dimension géographique compte quatre tables alors qu'elle n'en présente qu'une seule dans la Figure 3.15.

Les schémas en flocon peuvent être plus efficaces en termes de stockage que les schémas en étoile, en particulier lorsqu'il s'agit de grandes tables de dimensions. En normalisant les données, les attributs redondants sont réduits, ce qui se traduit par des besoins de stockage potentiellement moindres. Aussi, cette modélisation rend explicite les hiérarchies présentes dans les différentes dimensions.

La Figure 3.16 représente la modélisation des données de Northwind selon un schéma en flocon. En effet, nous pouvons constater que les dimensions sont normalisées. Il s'agit du type de modélisation que nous utiliserons pour la suite de ce travail.

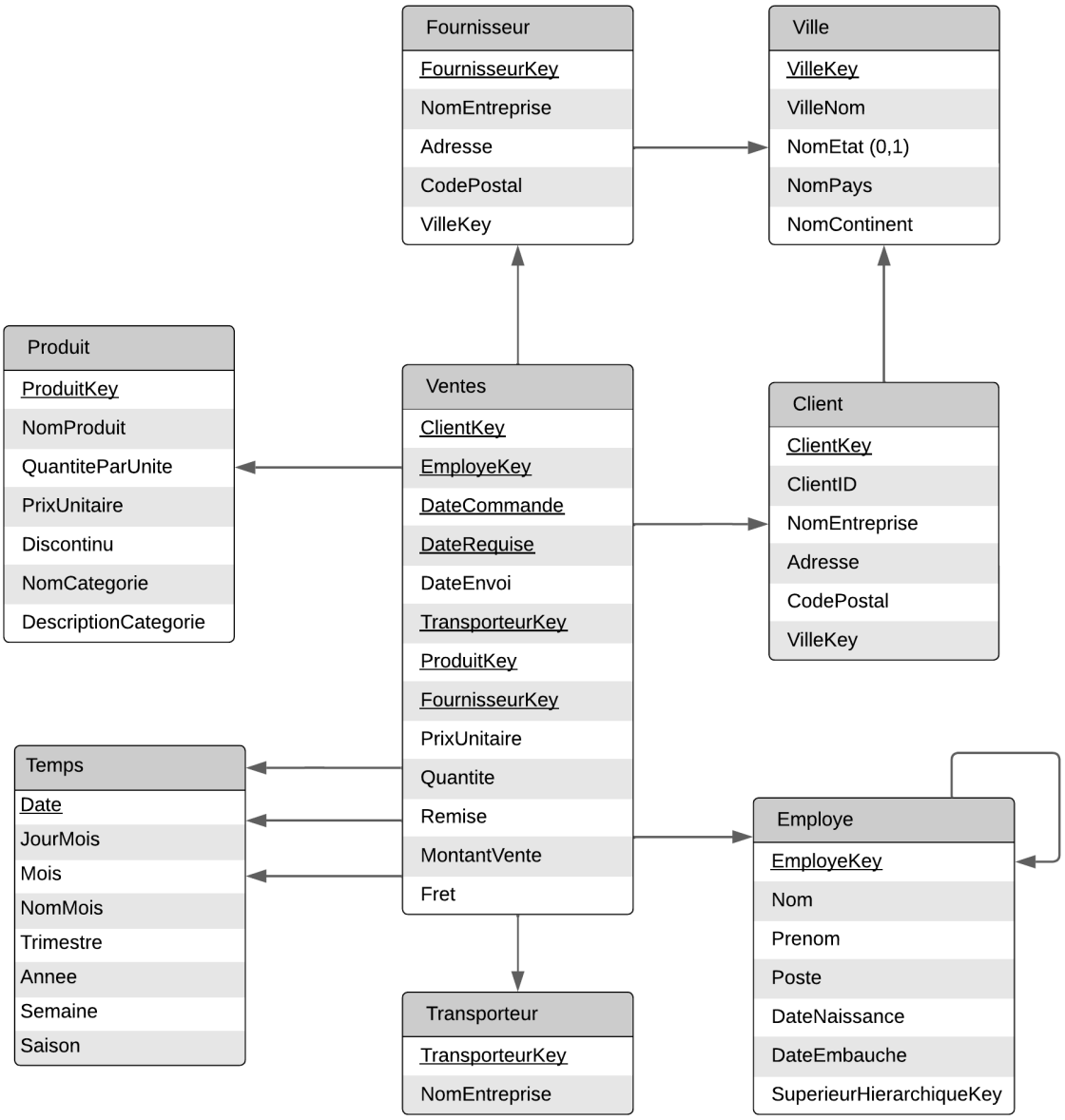


FIGURE 3.15 – Schéma en étoile basé sur les données de Northwind

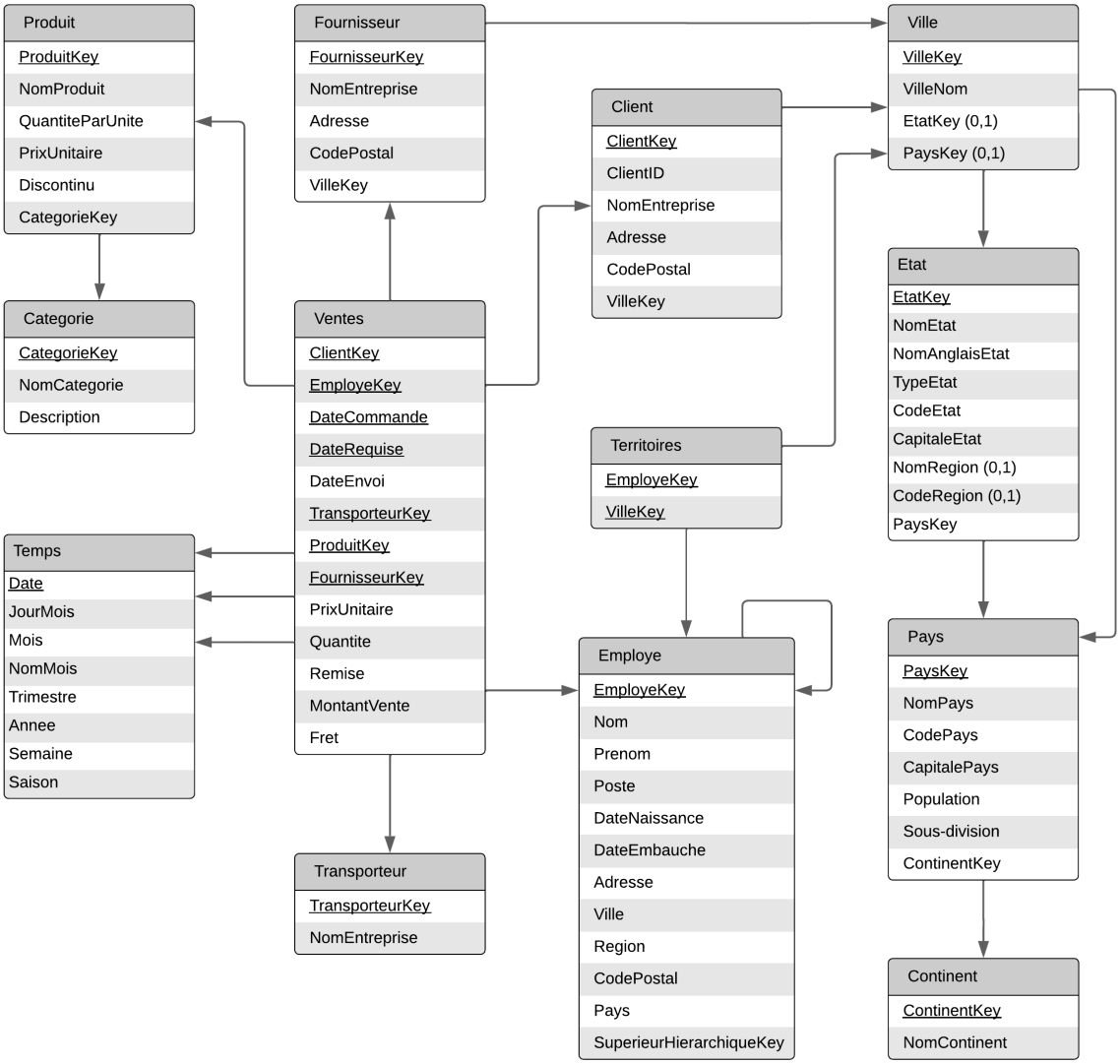


FIGURE 3.16 – Schéma en flocon basé sur les données de Northwind

3.5.3 Autres types de modélisation dimensionnelle

Par extension aux schémas en étoile et en flocon, un **schéma en flocon d'étoile**[3] comporte une table de faits unique ainsi que certaines dimensions normalisées et certaines dimensions dénormalisées. Dans le cas de Northwind, nous obtiendrions un schéma en flocon d'étoile en remplaçant, par exemple, les tables *Produit* et *Categorie* de la Figure 3.16 par la table *Produit* de la Figure 3.15 tout en ne changeant pas le reste des tables de la Figure 3.16.

Enfin, nous pouvons rencontrer un **schéma en constellation** lorsque les données comportent plusieurs tables de faits pouvant être analysées selon les mêmes dimensions. À noter que toutes les dimensions ne doivent pas nécessairement être partagées entre les tables de faits. La Figure 3.17 illustre un exemple de schéma en constellation reprenant les données de ventes et d'achats d'une chaine de magasins.

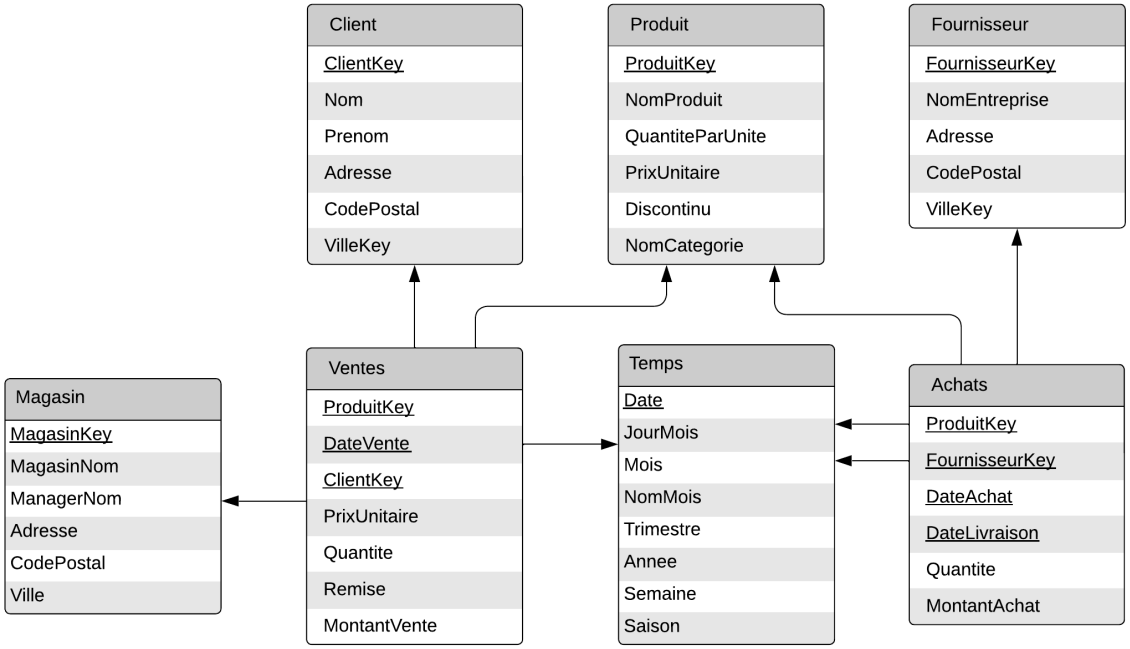


FIGURE 3.17 – Schéma en constellation

3. Le terme anglophone étant *starflake schema*, la traduction *schéma en flocon d'étoile* semble la plus appropriée

4 Réalisation du processus d'ETL

Une phase cruciale dans tout projet de Business Intelligence est le processus d'ETL (**Extract, Transform & Load**). Celui-ci permet de consolider les données nécessaires au travers des trois actions correspondant à son acronyme. Premièrement, le processus d'ETL est chargé d'extraire les données de sources variées. Il peut aussi bien s'agir de sources internes que de sources externes, de bases de données relationnelles que de fichiers plats,... Ensuite, les données extraites sont transformées afin d'être exploitables. Elles sont donc nettoyées, notamment en supprimant les données inutilisables ainsi que les doublons, et standardisées pour être conformes à l'utilisation qui en est prévue. Enfin, les données exploitables sont chargées dans le système final, généralement un data warehouse. Les données du data warehouse sont ensuite utilisées à des fins d'analyse au moyen de requêtes OLAP (Section 5).

Afin de pouvoir réaliser des opérations OLAP sur les données de Northwind, nous mettrons donc en place un processus d'ETL permettant de passer d'une base de données opérationnelle (Figure 2.1) à un data warehouse fonctionnel (Figure 3.16). Pour cela, nous utiliserons l'extension SSIS (SQL Server Integration Services) présente sur le logiciel Visual Studio.

4.1 Sources de données

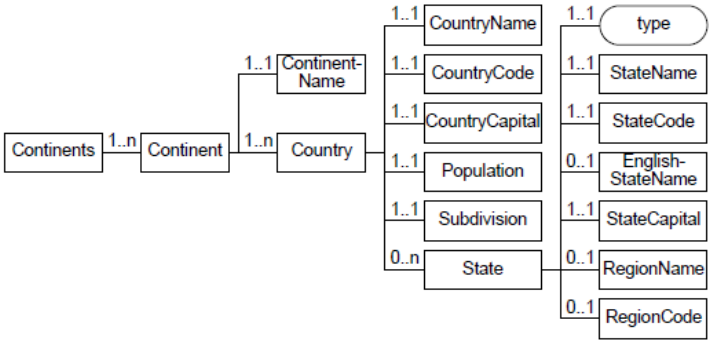
Le processus d'ETL que nous réaliserons comporte plusieurs sources de données. Tout d'abord, nous utiliserons la base de données opérationnelle présentée dans la Section 2, celle-ci étant hébergée sur une instance de serveur local. Ensuite, deux fichiers externes [4] seront utilisés afin de développer une dimension géographique à laquelle sont associées les tables de dimensions *Client* et *Fournisseur*.

Premièrement, les données permettant de développer la hiérarchie *Ville* –> *Etat* –> *Pays* –> *Continent* sont issues d'un fichier XML nommé Territories.xml. XML (eXtensible Markup Language) est un format de fichiers structurant les données au moyen de balises. L'intérêt de ce format réside dans le fait qu'il peut aisément être compris par une machine et par une personne (Iqbal, 2019). Les balises présentes dans un fichier XML sont appelées des éléments. Ceux-ci peuvent disposer d'attributs descriptifs inscrits au sein même de la balise (Amazon Web Services, s. d.). La Figure 4.1a présente le début du fichier Territories.xml. Afin de pouvoir être correctement parcouru par SSIS, un fichier contenant le schéma XML des données (Figure 4.1b) y est associé. Notons que les rectangles correspondent aux éléments XML tandis que les formes arrondies correspondent aux attributs XML. Ainsi, l'attribut type peut, par exemple, avoir comme valeur *state* pour l'Autriche ou *province* pour la Belgique, le premier étant illustré dans la Figure 4.1a.

4. Ces fichiers sont disponibles à l'adresse suivante sous le libellé "External data for extending the Geography dimension" : https://cs.ulb.ac.be/public/teaching/infoh419/tp

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Continents>
    <Continent>
        <ContinentName>Europe</ContinentName>
        <Country>
            <CountryName>Austria</CountryName>
            <CountryCode>AT</CountryCode>
            <CountryCapital>Vienna</CountryCapital>
            <Population>8316487</Population>
            <Subdivision>Austria is divided into nine Bundeslnder,
                or simply Lnder (states; sing. Land).</Subdivision>
            <State type="state">
                <StateName>Burgenland</StateName>
                <StateCode>BU</StateCode>
                <StateCapital>Eisenstadt</StateCapital>
            </State>
            <State type="state">
                <StateName>Krnten</StateName>
                <StateCode>KA</StateCode>
                <EnglishStateName>Carinthia</EnglishStateName>
                <StateCapital>Klagenfurt</StateCapital>
            </State>
            ...
```

(a) Aperçu du fichier Territories.xml



(b) Schéma XML du fichier Territories.xml

FIGURE 4.1 – Territories.xml

Deuxièmement, afin de pouvoir associer correctement une ville aux états/provinces et pays correspondants, nous utiliserons un fichier nommé cities.txt. Celui-ci est constitué de trois colonnes (Ville, Etat et Pays) séparées par une tabulation. Notons que si un pays ne possède ni état ni province, par exemple Singapour, une valeur NULL est placée dans la colonne. La Table 4.1 illustre le début du fichier cities.txt. Afin de correctement utiliser le contenu de ce fichier, nous créerons une table temporaire nommée *VilleTemp*.

Ville	Etat	Pays
Atlanta	Georgia	USA
Austin	Texas	USA
Beachwood	Ohio	USA
Bedford	Indiana	USA
Bellevue	Kentucky	USA
Bentonville	Arkansas	USA
...	...	...

TABLE 4.1 – Aperçu du fichier *cities.txt*

4.2 Flux de contrôle

Lors de la mise en place d'un processus d'ETL via SSIS, nous pouvons différencier deux types de flux : le flux de contrôle et les flux de données. Le flux de contrôle présente une vue de haut niveau de l'ETL, c'est-à-dire l'ordre d'exécution des différents éléments composant le processus et permettant de charger correctement les données dans un data warehouse. Trois types d'éléments peuvent être retrouvés dans un flux de contrôle :

1. Les **tâches** qui fournissent des fonctionnalités au package. Il s'agit notamment du type correspondant aux flux de données.
2. Les **conteneurs** permettant d'organiser le flux de contrôle en regroupant de façon logique les tâches. Il est également possible d'imbriquer plusieurs conteneurs.
3. Les **contraintes de précédence** connectent les tâches et conteneurs ensemble et définissent ainsi l'ordre dans lequel les éléments du flux de contrôle sont exécutés.

4.2.1 Flux de contrôle utilisé pour Northwind

La Figure 4.2 représente le flux de contrôle mis en place dans le cas d'étude Northwind. Celui-ci commence par une tâche d'exécution de script SQL [5] **[Création Dimension Temporelle]** permettant de créer et de remplir la dimension temporelle du data warehouse. Le script utilisé se trouve dans l'Annexe B. Nous noterons qu'une méthode plus commune consiste à récupérer et regrouper toutes les dates présentes dans les données opérationnelles. Ensuite, une autre tâche d'exécution de script SQL **[Création Tables]** crée le reste des tables présentes dans le data warehouse, y compris la table *VilleTemp*, sans les remplir. Enfin, un conteneur de séquence est exécuté. Celui-ci contient toutes les tâches de flux de données utilisées pour remplir les tables. Ce deuxième script peut être trouvé dans l'Annexe C.

Toutes les tâches présentes dans le conteneur de séquence **[Remplissage Tables]** n'étant pas affectées par une contrainte de précédence s'exécutent une fois que la seconde tâche d'exécution de script SQL est terminée. Pour une compréhension plus simple, nous pouvons estimer que les contraintes de précédence sont liées à la présence de clés étrangères dans les tables du data warehouse. Par exemple, les tables de dimension *Fournisseur* et *Client* ainsi que la table de liaison *Territoires* ne peuvent être peuplées qu'une fois la table *Ville* remplie car elles la référencent toutes. Aussi, la table de faits *Ventes* ne peut être remplie qu'une fois toutes les tables de dimensions complétées. Notons toutefois que bien que le lien entre contrainte de précédence et clé étrangère s'applique dans le cas présent, il ne s'agit pas d'une généralité. En effet, cette remarque peut être faite car nous stockerons le data warehouse dans une base de données relationnelle (selon l'approche ROLAP), ce qui n'est pas une règle générale.

5. Le script SQL a été librement adapté de celui présenté dans la vidéo suivante : https://www.youtube.com/watch?v=5OieIJeNXZA

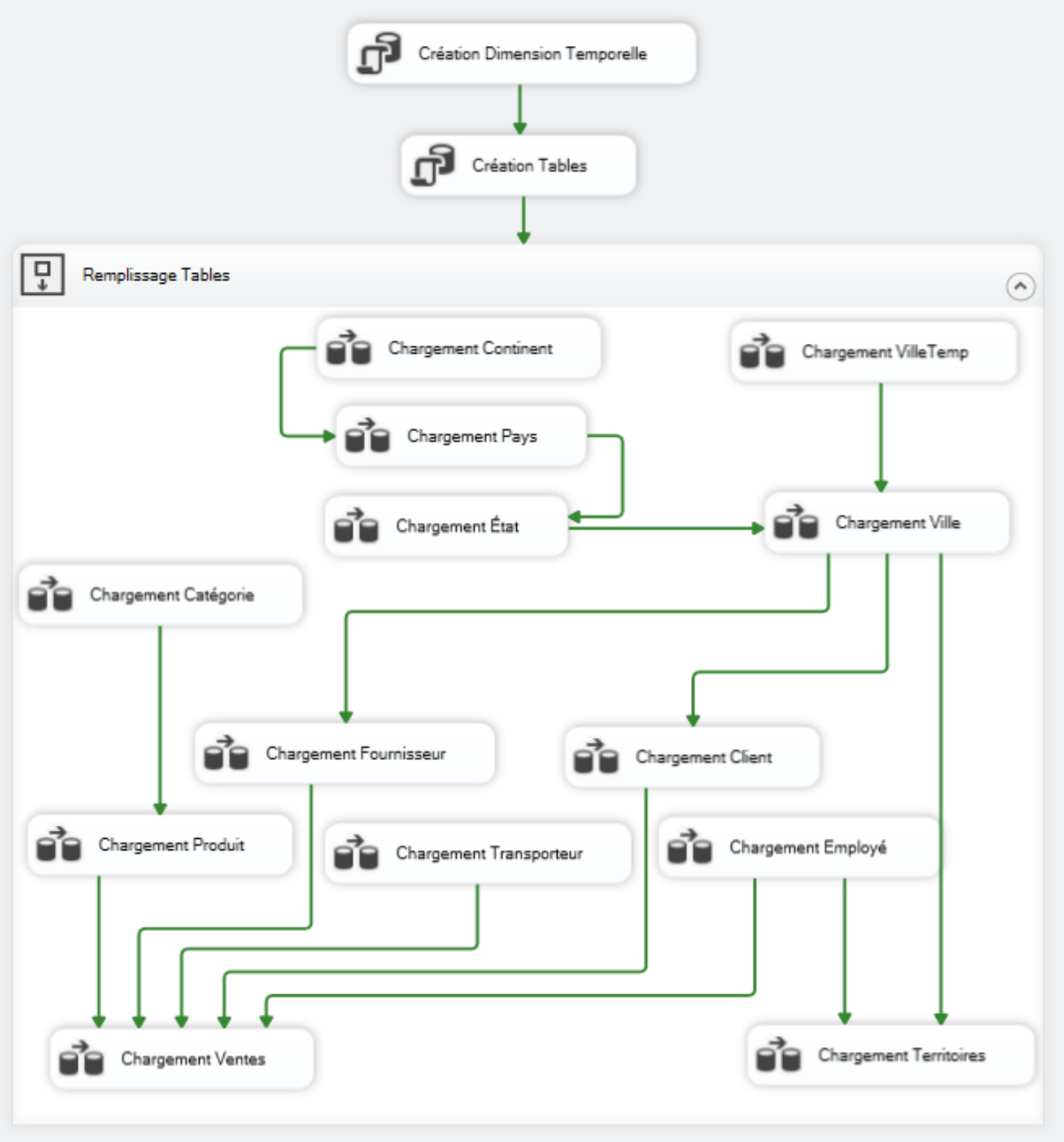


FIGURE 4.2 – Flux de contrôle de l'ETL dans SSIS

4.3 Flux de données

Les flux de données mis en place pour le cas d'étude Northwind sont de complexités variées. Ainsi, certains flux de données sont directs. C'est notamment le cas pour ceux permettant de remplir les tables de dimensions *Categorie*, *Produit*, *Transporteur* et *Employe*. Ceux-ci sont simplement composés d'une tâche Source OLE DB **[NorthwindDB Categories]** extrayant toutes les données de la table correspondante dans la base de données opérationnelle et d'une seconde tâche Destination OLE DB **Categorie Load (DW)** recevant ces données et les stockant dans le data warehouse après avoir effectué un mappage entre les colonnes de source et celles de destination. La Figure 4.3 illustre le flux de données **[Chargement Catégorie]** tandis que la Figure 4.4 présente le mappage réalisé pour celui-ci.

FIGURE 4.3 – Flux de données pour la table de dimension *Categorie*

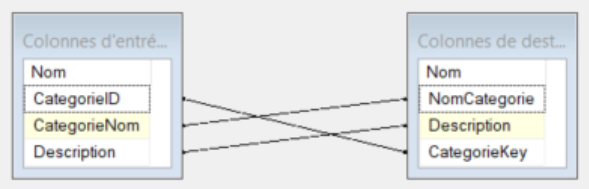


FIGURE 4.4 – Mappage pour le flux de données *Chargement Categorie*

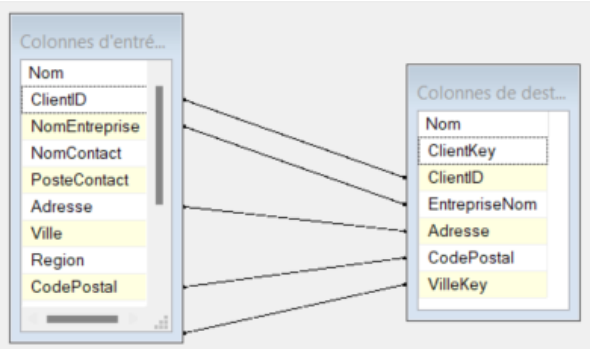


FIGURE 4.5 – Mappage pour le flux de données *Chargement Client*

Notons que pour certaines tables, nous ne garderons pas l'identifiant présent dans la base de données opérationnelle et devrons donc en générer un nouveau. Cela impacte notamment la phase de mappage. De ce fait, le mappage d'une table conservant l'identifiant initial correspondra à la Figure 4.4 alors que celui d'une table pour laquelle il est nécessaire de générer un nouvel identifiant correspondra à la Figure 4.5. Nous noterons donc que la colonne *ClientKey* présente dans le data warehouse n'est mappée à aucune colonne de la base de données. En effet, celle-ci ayant été définie comme une colonne identité dans le script de création des tables, un identifiant incrémental sera automatiquement généré lors du remplissage de la table.

Les flux de données relatifs à la hiérarchie *Etat* –> *Pays* –> *Continent* ainsi qu'à la table temporaire *VilleTemp* sont eux aussi relativement directs. L'unique différence avec les flux de données précédents réside dans l'ajout d'une tâche **[Conversion de données]**

permettant de s'assurer que les données possèdent le bon format pour être intégrées au data warehouse. Les Figures 4.6 et 4.7 illustrent les flux de données associés aux tables *Continent* et *VilleTemp*. Le fichier Territories.xml comportant différents niveaux (Continent, Pays et Etat), nous noterons que pour le flux de données de la Figure 4.6, il est nécessaire de préciser le niveau utilisée dans la sortie de la tâche **[Source XML Territories]**.

FIGURE 4.6 – Chargement de la table de dimension *Continent*

FIGURE 4.7 – Flux de données pour la table temporaire *VilleTemp*

Pour le flux de données de la table *Ville* (Figure 4.8), il est nécessaire d'associer à chaque ville issue de la table temporaire *VilleTemp* un identifiant correspondant à l'état ou au pays dans lequel elle se situe. Cela dépendant du fait que le pays en question soit constitué ou non d'états. Pour cela, une première tâche de **[Fractionnement conditionnel]** scinde les données extraites de la table *VilleTemp* selon que la valeur de la colonne Etat est nulle ou non. Dans le premier cas de figure, une tâche de recherche **[Lookup PaysKey avec PaysNom]** permet d'associer à la ville l'identifiant du pays correspondant. Dans le second cas, trois tâches de recherche successives sont utilisées pour associer l'identifiant correct de l'état dans lequel la ville se situe. Ces trois tâches permettent de prendre en compte toutes les possibilités d'encodage, à savoir dans la langue du pays, en anglais ou par acronyme. Elles fonctionnent de la façon suivante :

- La première recherche **[Lookup EtatKey avec EtatNom, PaysNom]** traite les données où les valeurs de Etat et de Pays correspondent aux valeurs de EtatNom et de PaysNom. C'est notamment le cas pour la province du *Hainaut* située en *Belgique*.
- La seconde recherche **[Lookup EtatKey avec EtatNomAnglais, PaysNom]** traite les données où les valeurs de Etat et de Pays correspondent respectivement aux valeurs de EtatNomAnglais et de PaysNom. Il s'agit notamment de l'état *Saxony*, dont le nom allemand est *Sachsen*, situé en Allemagne. La valeur relative au pays est *Germany* car seuls les noms de tables et de colonnes ont été traduits, comme mentionné dans la Section 2.
- Enfin, la troisième recherche **[Lookup EtatKey avec EtatNom, PaysCode]** considère les données où les valeurs de Etat et de Pays correspondent respectivement aux valeurs de EtatNom et de PaysCode. Un exemple concret est l'état de *Washington* situé aux *USA*.

Enfin, une tâche d'union **[Unir tout]** agrège les sorties des quatre tâches de recherche avant qu'une tâche Destination OLE DB **[NorthwindDW Ville]** les stocke conformément dans le data warehouse. Dans les cas où aucun état ou pays n'a pu être associé à la ville, les données sont stockées dans un fichier texte.

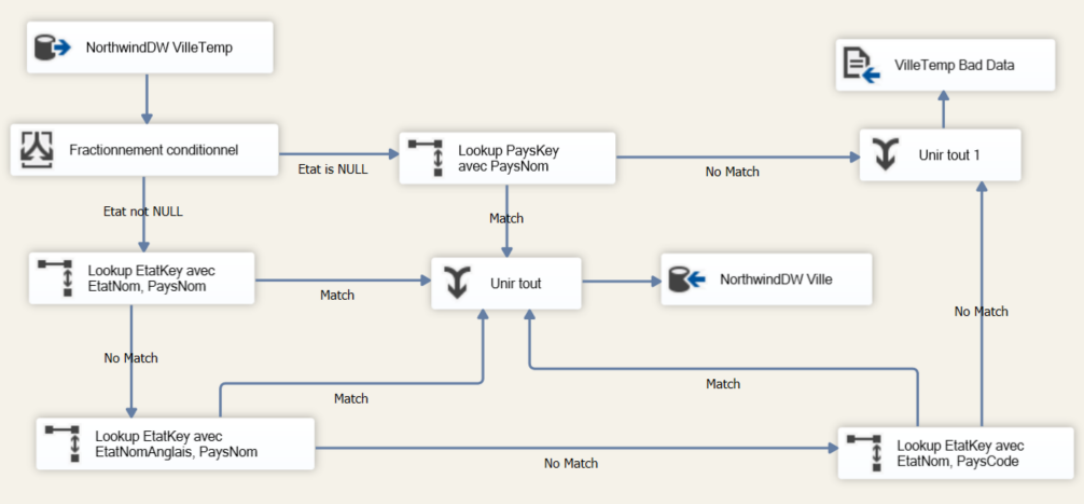


FIGURE 4.8 – Flux de données pour la table de dimension *Ville*

Les tables de dimensions *Fournisseur* et *Client* présentent des flux de données presque identiques. Nous ne détaillerons donc que le flux de données **[Chargement Fournisseur]** (Figure 4.9). Tout d'abord, il est nécessaire de commencer par une tâche de **[Fractionnement conditionnel]** car une valeur de région nulle peut être renseignée pour certains fournisseurs et cet attribut correspond aux états et provinces. Ainsi, dans le cas où la valeur de Region est nulle, nous effectuons une tâche de recherche **[Lookup Etat avec Ville, Pays]** afin de retrouver l'état sur base de la ville et du pays grâce à la table *VilleTemp*. Comme cette valeur peut également être nulle, une seconde tâche de **[Fractionnement conditionnel]** vient scinder les données. Si la valeur de Etat est nulle, une nouvelle tâche de recherche **[Lookup VilleKey avec VilleNom, PaysNom]** tente de retrouver l'identifiant correct de la ville sur base des noms de la ville et de son pays. La requête SQL utilisée pour récupérer les données utiles à la recherche est la suivante :

```
SELECT VilleKey, VilleNom, PaysNom
FROM Ville V
JOIN Pays P ON V.PaysKey = P.PaysKey
```

D'autre part, pour les clients n'ayant pas de valeur nulle pour Region, la colonne est renommée Etat. Le déroulement du reste du flux de données est semblable à celui présenté pour la table de dimension *Ville*. De ce fait, plusieurs tâches de recherche se succèdent afin d'associer l'identifiant correct pour la ville de chaque fournisseur. Enfin, tous les résultats positifs de ces tâches sont rassemblés avant d'être stockés dans le data warehouse par une tâche finale de Destination OLE DB **[NorthwindDW Fournisseur]**. Notons également, que les résultats négatifs des tâches de recherche sont quant à eux regroupés pour être stockés dans un fichier texte.

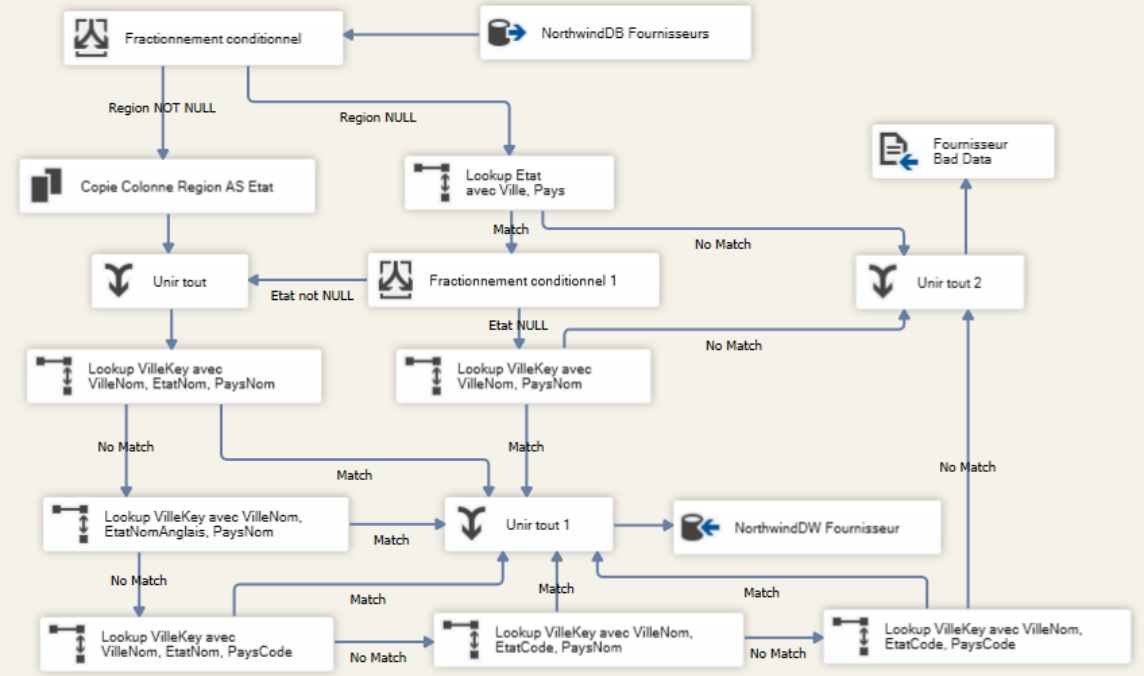


FIGURE 4.9 – Flux de données pour la table de dimension *Fournisseur*

La Figure 4.10 représente le flux de données relatif à la table de liaison *Territoires*. Celui-ci démarre par une tâche de requête SQL **[NorthwindDB Requête SQL]** qui joint les tables *EmployeTerritoires* et *Territoires* de la base de données opérationnelle. La requête est la suivante :

```
SELECT E.*, DescriptionTerritoire
FROM EmployeTerritoires E
JOIN Territoires T ON E.TerritoireID = T.TerritoireID
```

Après cela, une tâche de colonne dérivée **[Trim DescriptionTerritoire]** est utilisée pour enlever les espaces en début et fin de chaines de caractères pour la colonne DescriptionTerritoire. Ensuite, une tâche de recherche **[Lookup VilleKey avec DescriptionTerritoire]** associe les identifiants de ville corrects en faisant correspondre les valeurs de DescriptionTerritoire avec les valeurs de VilleNom de la table *Ville*. Enfin, les doublons sont supprimés avant que les données ne soient stockées dans le data warehouse.

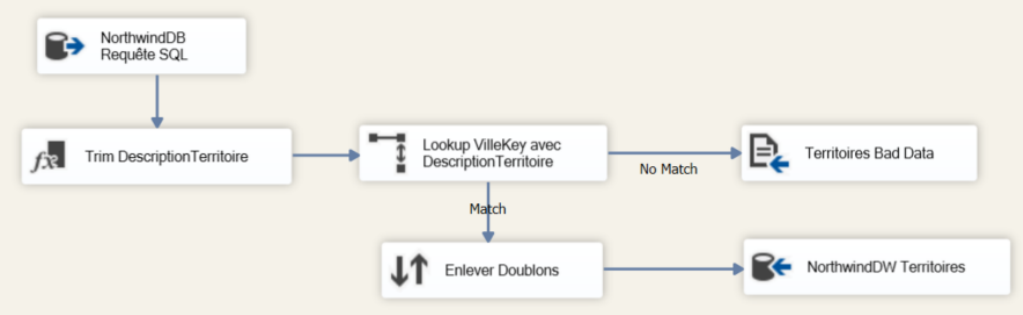


FIGURE 4.10 – Flux de données pour la table de liaison *Territoires*

Enfin, la table de faits *Ventes* est chargée dans le data warehouse selon le flux de données illustré par la Figure 4.11. Celui-ci commence par obtenir les données pertinentes de la base de données opérationnelle via la requête SQL suivante :

```
SELECT ClientID, EmployeID, DateCommande, DateRequise, DateEnvoi,
    EnvoiPar, ProduitID, PrixUnitaire, Quantite, Remise,
    Fret/COUNT(*) OVER (PARTITION BY D.CommandeID) AS Fret,
    CONVERT(MONEY, PrixUnitaire*(1-Remise)*Quantite) AS MontantVente
FROM Commandes C, [Details Commande] D
WHERE C.CommandeID = D.CommandeID
```

Ensuite, deux tâches de recherche **[Lookup ClientKey avec ClientID]** et **[Lookup FournisseurKey avec ProduitID]** sont utilisées pour récupérer l'identifiant de la ville et du fournisseur depuis les tables de dimensions respectives. Après cela, une tâche de **[Fractionnement conditionnel]** vérifie qu'aucune instance ne possède de valeur nulle pour une des colonnes constituant la clé primaire de la table de faits. Enfin, les données valides sont stockées dans le data warehouse par une tâche de Destination OLE DB **[NorthwindDW - Ventes]**.De la même façon que pour les flux de données précédents, les données sans correspondance après les tâches de recherche sont stockées dans un fichier texte.

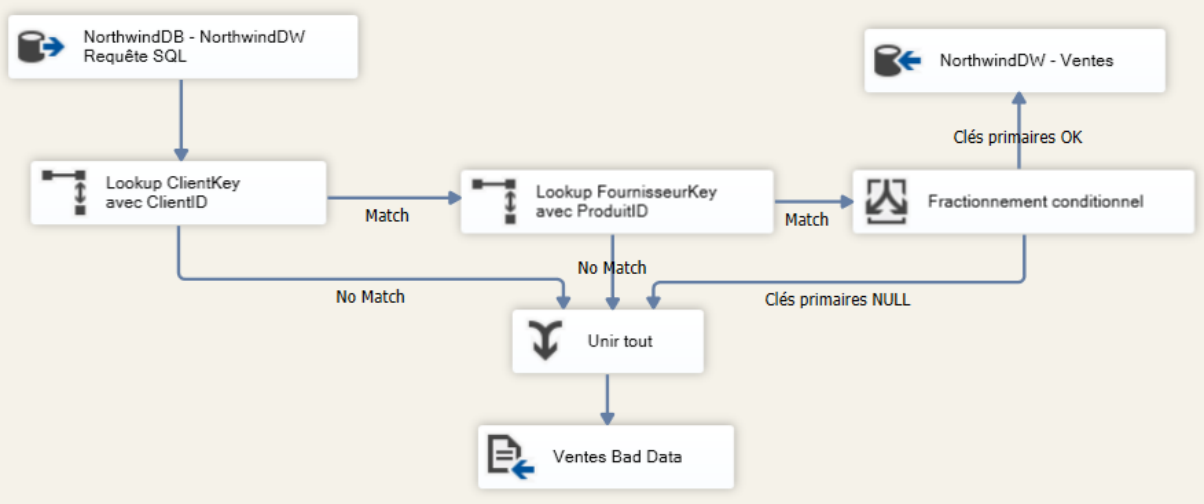


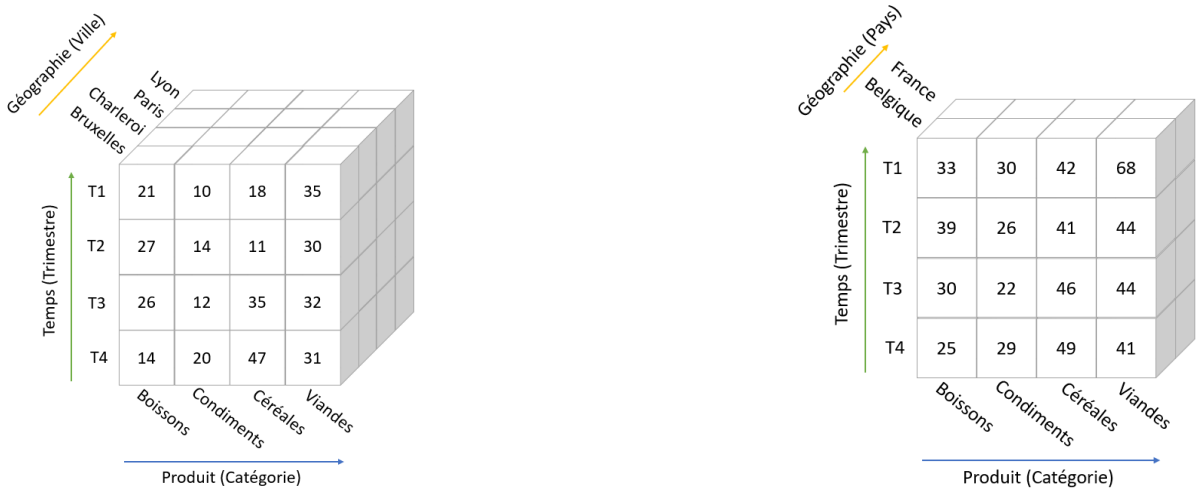
FIGURE 4.11 – Flux de données pour la table de faits *Ventes*

5 Opérations OLAP (De Aguiar Ciferri et al., 2013)

Comme nous l'avons mentionné dans la Section 5, une des principales caractéristiques de la modélisation multidimensionnelle est la possibilité de visualiser les données sous plusieurs angles et à différents degrés de détail. En manipulant les dimensions et leurs hiérarchies, les opérateurs OLAP permettent de matérialiser ces vues et niveaux de détail. Dans cette section, nous présenterons les principales opérations réalisables dans le cadre des systèmes OLAP. Celles-ci seront systématiquement illustrées sous forme de cube multidimensionnel. Notons que ces exemples sont adaptés de ceux présents dans Vaisman et Zimányi, 2014

5.1 Roll-Up

L'opération de **roll-up** est utilisée afin d'agréger les mesures du cube selon un niveau de dimension hiérarchique supérieur. Cela permet donc de visualiser les données à un niveau d'abstraction plus élevé.



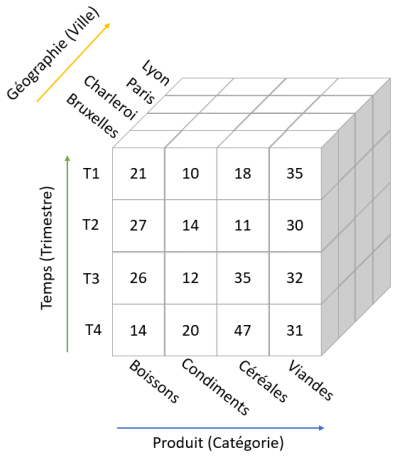
(a) Cube originel

(b) Roll-up vers le niveau *Pays*

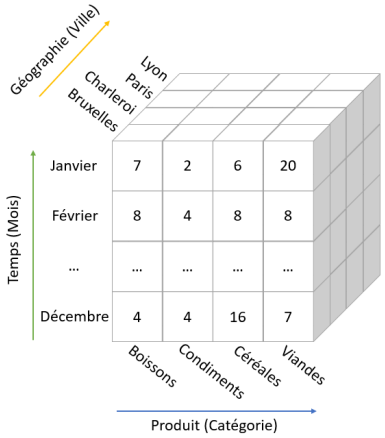
Figure 5.1 – Illustration de *roll-up* sur un cube OLAP

5.2 Drill-Down

L'opération de **drill-down** est l'inverse du roll-up. Elle permet aux utilisateurs de visualiser les données à un niveau de précision plus élevé. Il s'agit ainsi de passer d'un niveau plus général à un niveau plus spécifique dans une hiérarchie.



(a) Cube originel

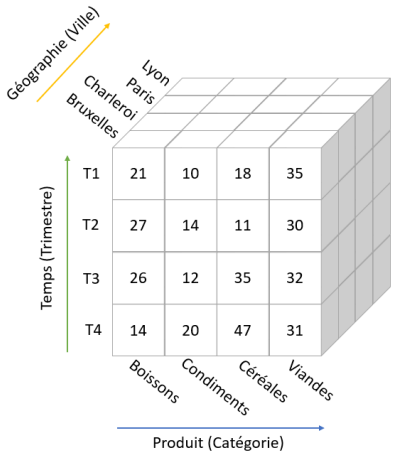


(b) Drill-down vers le niveau *Mois*

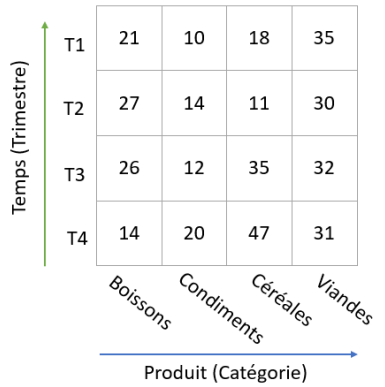
FIGURE 5.2 – Illustration de *drill-down* sur un cube OLAP

5.3 Slice

L'opération de **slice** consiste à sélectionner une valeur spécifique pour une dimension, ce qui permet de filtrer le cube pour se concentrer sur un sous-ensemble de données. Si nous considérons le cube initial comme un ayant n dimension, le nouveau cube résultant de l'opération présentera alors *n-1* dimensions.



(a) Cube originel

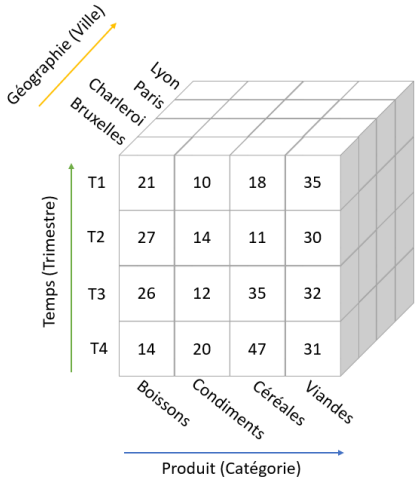


(b) Slice sur *Ville='Bruxelles'*

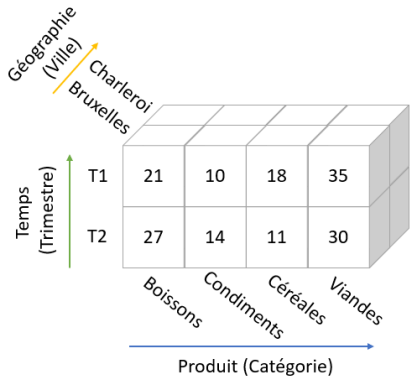
FIGURE 5.3 – Illustration de *slice* sur un cube OLAP

5.4 Dice

L'opération de **dice** consiste à sélectionner une combinaison spécifique de valeurs pour plusieurs dimensions afin de créer un sous-cube. Elle permet de ne voir qu'un sous-ensemble particulier de données en spécifiant des valeurs pour plusieurs dimensions.



(a) Cube originel

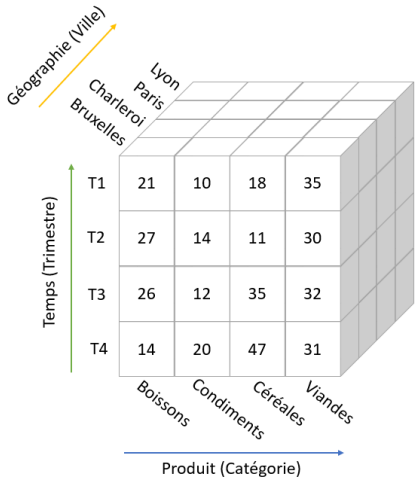


(b) Dice sur *Ville='Bruxelles' or 'Charleroi'* et *Trimestre = 'T1' or 'T2'*

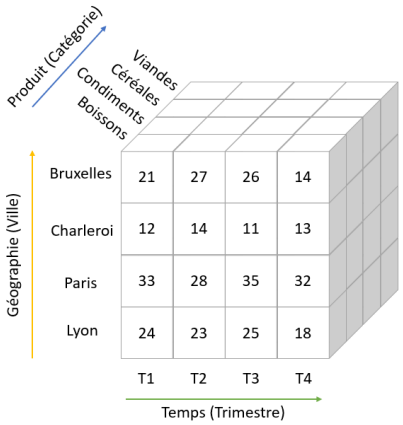
FIGURE 5.4 – Illustration de *dice* sur un cube OLAP

5.5 Pivot

L'opération de **pivot** consiste à faire pivoter le cube pour visualiser les données d'un point de vue différent. Il permet de modifier l'orientation des dimensions et des mesures afin d'analyser les données sous différents angles.



(a) Cube originel

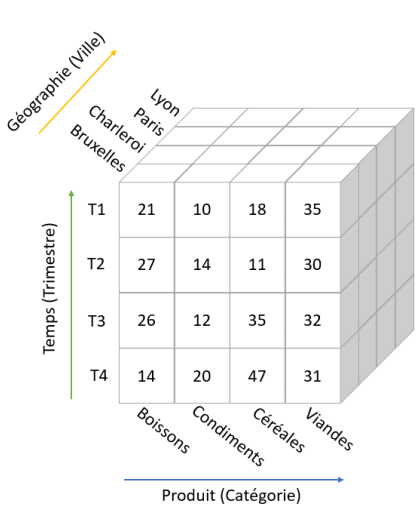


(b) Cube après pivot

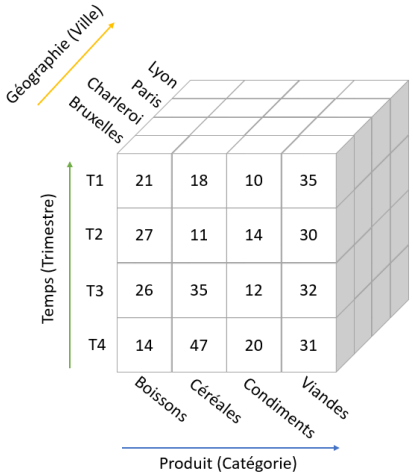
FIGURE 5.5 – Illustration de *pivot* sur un cube OLAP

5.6 Sort

L'opération de **sort** permet d'obtenir un cube dont les membres d'une dimension ont été triés selon un critère établi.



(a) Cube originel



(b) Cube après tri des produits par nom

FIGURE 5.6 – Illustration de *sort* sur un cube OLAP

6 Application des opérations OLAP en SQL et MDX

Comme précédemment expliqué, les systèmes OLAP ont été conçus pour répondre aux besoins analytiques des entreprises (Section 3). Dans la Section 5, nous avons introduit les principales opérations utilisées pour manipuler les données d'un cube multidimensionnel. La présente section abordera différentes requêtes permettant de réaliser ces opérations. Nous nous baserons sur les données présentes dans le cube de la Section 5. Celles-ci concernent donc les villes de *Bruxelles*, *Charleroi*, *Paris* et *Lyon* ainsi que les catégories de produits *Viandes*, *Céréales*, *Condiments* et *Boissons*. Cependant il nous faut noter que peu de cellules du cube contiennent réellement des mesures. Pour pallier cela et fournir une meilleure illustration des opérations, des mesures fictives ont été introduites. Les données réelles du cube peuvent être récupérées par la requête suivante :

```
SELECT Trimestre, NomCategorie, VilleNom, SUM(Quantite) AS QuantiteTotale
FROM Ventes
JOIN DimDate ON Ventes.DateCommande = DimDate.Date
JOIN Produit ON Ventes.ProduitKey = Produit.ProduitKey
JOIN Categorie ON Produit.CategorieKey = Categorie.CategorieKey
JOIN Client ON Ventes.ClientKey = Client.ClientKey
JOIN Ville ON Client.VilleKey = Ville.VilleKey
WHERE VilleNom IN ('Bruxelles', 'Charleroi', 'Paris', 'Lyon')
AND NomCategorie IN ('Meat/Poultry', 'Grains/Cereals', 'Condiments', 'Beverages')
GROUP BY VilleNom, Trimestre, NomCategorie;
```

Notons que pour les requêtes correspondant aux différentes opérations OLAP, nous ne mentionnerons plus les conditions s'appliquant aux colonnes *VilleNom* et *NomCategorie* car elles permettent simplement de récupérer les données du sous-cube présenté dans la Section 5. Nous constatons également le grand nombre de jointures nécessaires. Cela est dû à l'utilisation d'un schéma en flocon pour le stockage des données (Figure 3.16). Le résultat renvoyé par la requête est le suivant :

Trimestre	NomCategorie	VilleNom	QuantiteTotale
1	Condiments	Bruxelles	6
1	Grains/Cereals	Bruxelles	16
4	Beverages	Bruxelles	30
1	Beverages	Charleroi	20
1	Condiments	Charleroi	40
1	Grains/Cereals	Charleroi	95
2	Grains/Cereals	Charleroi	30
3	Beverages	Charleroi	12
4	Grains/Cereals	Charleroi	2
1	Beverages	Lyon	20
3	Grains/Cereals	Lyon	21
1	Beverages	Paris	19

TABLE 6.1 – Données effectives du cube de la Figure 3.1

6.1 Multi-Dimensional eXpressions (Whitehorn et al., 2005)

Dans cette Section, nous serons amenés à présenter l'application des opérations OLAP au moyen des langages de requêtes SQL et MDX. Avant cela, il est donc nécessaire de fournir une brève présentation de ce second langage. Nous n'irons toutefois pas dans les détails car cela ne constitue pas un point central de ce travail.

MDX (Multi-Dimensional eXpressions) est un langage de requête conçu spécifiquement pour interroger des bases de données stockées de façon multidimensionnelle selon l'approche MOLAP. De ce fait, alors que le langage SQL agit sur des tables et attributs, le langage MDX fonctionne directement sur les éléments d'un cube de données. C'est-à-dire que les requêtes sont appliquées sur des dimensions, des hiérarchies, les membres de ces hiérarchies ou encore les mesures composant le cube.

Un concept primordial en MDX est celui de tuple. Un tuple permet d'identifier précisément une cellule du cube de données et se construit de la manière suivante :

```
(Dimension1.Niveau1.Membre1, Dimension2.Niveau2.Membre2,...)
```

Dans le cadre de Northwind, si nous souhaitons identifier la cellule reprenant les ventes de condiments à Bruxelles lors du premier trimestre, nous pouvons donc utiliser le tuple suivant :

```
(Produit.Categorie.Condiments, DimDate.Trimestre.T1, Client.Ville.Bruxelles)
```

Précisons que l'ordre dans lequel les dimensions sont mentionnées n'a pas d'importance. Notons également qu'en MDX, les mesures agissent de la même façon que les dimensions. Ainsi, la dimension *Mesures* comprendra autant de membres qu'il n'y a de mesures présentes dans le cube multidimensionnel. La syntaxe utilisée en MDX est sensiblement similaire à celle que nous utilisons lors de requêtes SQL, bien que la sémantique soit différente pour les deux langages. La requête suivante illustre la structure générale d'une requête MDX :

```
SELECT  [Measures].[Mesure], ON COLUMNS,
        [Dimension1].[Niveau1].MEMBERS ON ROWS
        [Dimension2].[Niveau2].MEMBERS ON PAGES
FROM [Cube]
WHERE ([Dimension1].[Niveau1].[Membre1], [Dimension2].[Niveau2].[Membre2])
```

Décomposons à présent cette structure. Tout d'abord, la commande SELECT permet de définir les axes composant la requête ainsi que les membres présents sur chacun de ces axes. Nous pouvons retrouver jusqu'à 128 axes dans une requête MDX. Les cinq premiers axes disposent de noms prédéfinis, à savoir *COLUMNS*, *ROWS*, *PAGES*, *CHAPTERS* et *SECTIONS*. Ceux-ci peuvent également être mentionnés par *AXIS(n)*, où n correspond au numéro de l'axe en question. Ensuite, la commande FROM spécifie le ou les cube(s) à interroger pour la requête. Enfin, la commande WHERE est optionnelle et permet d'établir une condition sur les données renvoyées.

6.2 Roll-Up

6.2.1 Requête SQL

Lemahieu et al., 2018 nous expliquent que SQL :1999 [6] a introduit plusieurs extensions à la commande GROUP BY afin de faciliter l'exécution d'opérations OLAP et l'agrégation de données. Ces extensions comprennent notamment l'opérateur ROLLUP. Celui-ci permet d'effectuer des opérations de regroupement avancées en calculant les sous-totaux et totaux sur une combinaison de colonnes spécifiée. Ainsi, lorsque nous utilisons cet opérateur, le résultat renvoyé contient, en plus des agrégations obtenues avec un GROUP BY standard, des lignes représentant des agrégations partielles et globales des données. La requête SQL permettant d'effectuer le roll-up illustré par la Figure 5.1b est la suivante :

```
SELECT Trimestre, NomCategorie, PaysNom,
        SUM(Quantite) AS QuantiteTotale
FROM Ventes
JOIN DimDate ON Ventes.DateCommande = DimDate.Date
JOIN Produit ON Ventes.ProduitKey = Produit.ProduitKey
JOIN Categorie ON Produit.CategorieKey = Categorie.CategorieKey
JOIN Client ON Ventes.ClientKey = Client.ClientKey
JOIN Ville ON Client.VilleKey = Ville.VilleKey
JOIN Etat ON Ville.EtatKey = Etat.EtatKey
JOIN Pays ON Etat.PaysKey = Pays.PaysKey
GROUP BY ROLLUP (PaysNom, Trimestre, NomCategorie)
```

Le résultat généré par la requête SQL est présenté dans la Table 6.2. Nous pouvons constater la présence de lignes contenant la valeur NULL. Celles-ci constituent les sous-totaux et totaux générés par l'opérateur ROLLUP. De ce fait, la quatrième ligne correspond à la quantité totale de produits vendus en Belgique lors des premiers trimestres de chaque année tandis que la douzième ligne correspond à la quantité totale vendue en Belgique. Enfin la dernière ligne reprend la quantité totale vendue. Bien que ces totaux et sous-totaux ne soient pas explicitement affichés sur le cube de la Figure 5.1b, ceux-ci constituent un outil précieux lors de l'analyse des données de ventes.

6. SQL :1999, communément appelé SQL 3, constitue la quatrième révision apportée au langage SQL. Un historique de l'évolution du langage est disponible à l'adresse suivante : https://learnsql.com/blog/history-of-sql-standards/

Trimestre	NomCategorie	PaysNom	QuantiteTotale
1	Beverages	Belgium	20
1	Condiments	Belgium	46
1	Grains/Cereals	Belgium	111
1	NULL	Belgium	177
2	Grains/Cereals	Belgium	30
2	NULL	Belgium	30
3	Beverages	Belgium	12
3	NULL	Belgium	12
4	Beverages	Belgium	30
4	Grains/Cereals	Belgium	2
4	NULL	Belgium	32
NULL	NULL	Belgium	251
1	Beverages	France	39
1	NULL	France	39
3	Grains/Cereals	France	21
3	NULL	France	21
NULL	NULL	France	60
NULL	NULL	NULL	311

TABLE 6.2 – Résultat de la requête de roll-up en SQL

En réalité, l'opérateur ROLLUP est une variante de l'opérateur GROUPING SETS. Celui-ci nécessite de spécifier explicitement les agrégations à réaliser et permet donc de couvrir de plus nombreuses possibilités. Ainsi, si nous utilisions l'opérateur GROUPING SETS pour la requête précédemment utilisée avec l'opérateur ROLLUP, cela nous donnerait la requête suivante :

```
SELECT Trimestre, NomCategorie, PaysNom, SUM(Quantite) AS QuantiteTotale
FROM Ventes
JOIN DimDate ON Ventes.DateCommande = DimDate.Date
JOIN Produit ON Ventes.ProduitKey = Produit.ProduitKey
JOIN Categorie ON Produit.CategorieKey = Categorie.CategorieKey
JOIN Client ON Ventes.ClientKey = Client.ClientKey
JOIN Ville ON Client.VilleKey = Ville.VilleKey
JOIN Etat ON Ville.EtatKey = Etat.EtatKey
JOIN Pays ON Etat.PaysKey = Pays.PaysKey
GROUP BY GROUPING SETS
        ((PaysNom, Trimestre, NomCategorie), (PaysNom, Trimestre), (PaysNom), ())
```

L'opérateur GROUPING SETS comporte également l'opérateur CUBE. Celui-ci est similaire à l'opérateur ROLLUP. La différence principale réside dans le fait qu'il calcule l'ensemble des totaux de la liste d'attributs fournie. De ce fait, alors que l'ordre des attributs spécifiés est important pour l'opérateur ROLLUP, ce n'est pas le cas pour l'opérateur CUBE. La requête suivante illustre l'utilisation de celui-ci et le résultat renvoyé est présenté dans la Table 6.3 :

```
SELECT Trimestre, NomCategorie, PaysNom, SUM(Quantite) AS QuantiteTotale
FROM Ventes
JOIN DimDate ON Ventes.DateCommande = DimDate.Date
JOIN Produit ON Ventes.ProduitKey = Produit.ProduitKey
JOIN Categorie ON Produit.CategorieKey = Categorie.CategorieKey
JOIN Client ON Ventes.ClientKey = Client.ClientKey
JOIN Ville ON Client.VilleKey = Ville.VilleKey
JOIN Etat ON Ville.EtatKey = Etat.EtatKey
JOIN Pays ON Etat.PaysKey = Pays.PaysKey
GROUP BY CUBE (PaysNom, Trimestre, NomCategorie)
```

6.2.2 Requête MDX

En MDX, les fonctions DRILLUP et DRILLDOWN sont des fonctions de navigation permettant de manipuler la granularité des données présentées du cube multidimensionnel. De ce fait, elles sont comparables aux opérations de roll-up et de drill-down. Ces deux fonctions présentent plusieurs variantes, nous n'en détaillerons toutefois qu'une seule.

Ainsi, la fonction DRILLUPLEVEL renvoie les membres d'un ensemble appartenant à un niveau supérieur à celui spécifié dans cet ensemble. La syntaxe théorique de cette fonction est la suivante :

```
DrillupLevel(Set_Expression [ , Level_Expression ] )
```

D'une part, l'argument *Set_Expression* constitue l'expression MDX permettant de définir l'ensemble considéré. D'autre part, *Level_Expression* est un argument optionnel qui permet de spécifier le niveau vers lequel nous souhaitons effectuer le de roll-up.

Trimestre	NomCategorie	PaysNom	QuantiteTotale
1	Beverages	Belgium	20
1	Beverages	France	39
1	Beverages	NULL	59
3	Beverages	Belgium	12
3	Beverages	NULL	12
4	Beverages	Belgium	30
4	Beverages	NULL	30
NULL	Beverages	NULL	101
1	Condiments	Belgium	46
1	Condiments	NULL	46
NULL	Condiments	NULL	46
1	Grains/Cereals	Belgium	111
1	Grains/Cereals	NULL	111
2	Grains/Cereals	Belgium	30
2	Grains/Cereals	NULL	30
3	Grains/Cereals	France	21
3	Grains/Cereals	NULL	21
4	Grains/Cereals	Belgium	2
4	Grains/Cereals	NULL	2
NULL	Grains/Cereals	NULL	164
NULL	NULL	NULL	311
NULL	Beverages	Belgium	62
NULL	Condiments	Belgium	46
NULL	Grains/Cereals	Belgium	143
NULL	NULL	Belgium	251
NULL	Beverages	France	39
NULL	Grains/Cereals	France	21
NULL	NULL	France	60
1	NULL	Belgium	177
1	NULL	France	39
1	NULL	NULL	216
2	NULL	Belgium	30
2	NULL	NULL	30
3	NULL	Belgium	12
3	NULL	France	21
3	NULL	NULL	33
4	NULL	Belgium	32
4	NULL	NULL	32

TABLE 6.3 – Résultat de la requête utilisant l'opérateur CUBE en SQL

6.3 Slice

6.3.1 Requête SQL

En ce qui concerne l'opération de slice, l'utilisation de la clause WHERE en SQL permet de fixer une valeur spécifique pour une dimension. Notons également que cette clause peut être utilisée pour fixer les valeurs de plusieurs dimensions, correspondant de ce fait à l'opération de dice. Nous ne développerons dès lors pas cette opération. Pour notre exemple, la requête suivante permet donc de réaliser un slice en fixant la valeur de VilleNom à Bruxelles :

```
SELECT Trimestre, NomCategorie, VilleNom,
        SUM(Quantite) AS QuantiteTotale
FROM Ventes
JOIN DimDate ON Ventes.DateCommande = DimDate.Date
JOIN Produit ON Ventes.ProduitKey = Produit.ProduitKey
JOIN Categorie ON Produit.CategorieKey = Categorie.CategorieKey
JOIN Client ON Ventes.ClientKey = Client.ClientKey
JOIN Ville ON Client.VilleKey = Ville.VilleKey
WHERE VilleNom = 'Bruxelles'
GROUP BY VilleNom, Trimestre, NomCategorie;
```

Le résultat de cette requête est repris dans la Table 6.4. Nous pouvons donc constater que peu de commandes ont été réalisées dans la ville de Bruxelles. En effet, seules trois catégories de produits et deux trimestres sont repris dans la table. Ceci atteste du fait que le cube soit très peu dense.

Trimestre	NomCategorie	VilleNom	QuantiteTotale
4	Beverages	Bruxelles	30
1	Condiments	Bruxelles	6
1	Grains/Cereals	Bruxelles	16

TABLE 6.4 – Résultat de la requête de slice en SQL

6.3.2 Requête MDX

En MDX, la clause WHERE permet également de définir une condition à appliquer sur les données d'une requête. Ainsi la requête suivante permet d'effectuer une opération de slice similaire à celle de la requête SQL précédemment explicitée :

```
SELECT Measures.MEMBERS ON COLUMNS,
        DateCommande.Trimestre.MEMBERS ON ROWS,
        Produit.Categorie.MEMBERS ON PAGES,
        Client.Ville.MEMBERS ON CHAPTERS
FROM Ventes
WHERE (Client.Ville.Bruxelles)
```

6.4 Pivot

6.4.1 Requête SQL

La démarche la plus pertinente pour réaliser l'opération de pivot en SQL consiste à transposer certaines lignes. Ainsi, par exemple, nous pouvons présenter les ventes réalisées par catégorie de produit au lieu de disposer d'une colonne précisant la catégorie s'appliquant à une vente.

Une première approche pour cela revient à utiliser la clause CASE. Celle-ci permet de créer de nouvelles colonnes basées sur des agrégations conditionnelles. La requête suivante présente une application de cette clause, avec pour objectif de transposer les quantités vendues pour certaines catégories de produits. Le résultat renvoyé par la requête est illustré dans la Table 6.5.

```
SELECT Trimestre, VilleNom,
SUM(CASE WHEN Categorie.NomCategorie = 'Meat/Poultry' THEN Quantite ELSE 0 END)
        AS Ventes_Viandes,
SUM(CASE WHEN Categorie.NomCategorie = 'Grains/Cereals' THEN Quantite ELSE 0 END)
        AS Ventes_Cereales,
SUM(CASE WHEN Categorie.NomCategorie = 'Condiments' THEN Quantite ELSE 0 END)
        AS Ventes_Condiments,
SUM(CASE WHEN Categorie.NomCategorie = 'Beverages' THEN Quantite ELSE 0 END)
        AS Ventes_Boissons
FROM Ventes
JOIN DimDate ON Ventes.DateCommande = DimDate.Date
JOIN Produit ON Ventes.ProduitKey = Produit.ProduitKey
JOIN Categorie ON Produit.CategorieKey = Categorie.CategorieKey
JOIN Client ON Ventes.ClientKey = Client.ClientKey
JOIN Ville ON Client.VilleKey = Ville.VilleKey
GROUP BY Trimestre, VilleNom
```

Trimestre	VilleNom	Ventes_Viandes	Ventes_Cereales	Ventes_Condiments	Ventes_Boissons
1	Bruxelles	0	16	6	0
4	Bruxelles	0	0	0	30
1	Charleroi	0	95	40	20
2	Charleroi	0	30	0	0
3	Charleroi	0	0	0	12
4	Charleroi	0	2	0	0
1	Lyon	0	0	0	20
3	Lyon	0	21	0	0
1	Paris	0	0	0	19

TABLE 6.5 – Résultat de la requête utilisant la clause CASE en SQL

Une seconde approche permettant de réaliser une opération de pivot fait usage de l'opérateur PIVOT. Cette approche nécessite de faire appel à une sous-requête afin de sélectionner les données pertinentes des différentes tables tandis que l'opérateur est utilisé pour réaliser le pivot en tant que tel. Pour cela, il nous faut spécifier les valeurs de la colonne que nous souhaitons transposer ainsi que la fonction d'agrégation à appliquer sur ces données. Là où l'utilisation de la clause CASE requiert de définir chaque colonne à transposer, l'opérateur PIVOT permet de générer automatiquement les colonnes transposées. Aussi, la requête correspondante est relativement plus compréhensible. Il nous faut toutefois noter que l'opérateur PIVOT n'est pas disponible pour toutes les versions de SQL (notamment pour PostgreSQL) alors que la clause CASE peut être utilisée dans toutes les versions de SQL sans dépendre d'extensions spécifiques. La requête suivante présente une application de l'opérateur PIVOT, avec pour objectif de transposer les quantités vendues pour certaines catégories de produits. Le résultat de cette requête est disponible dans la Table 6.6.

```
SELECT Trimestre, VilleNom,
[Meat/Poultry] AS Ventes_Viandes,
[Grains/Cereals] AS Ventes_Cereales,
[Condiments] AS Ventes_Condiments,
[Beverages] AS Ventes_Boissons
FROM (
    SELECT Trimestre, VilleNom, NomCategorie, Quantite
    FROM Ventes
    JOIN DimDate ON Ventes.DateCommande = DimDate.Date
    JOIN Produit ON Ventes.ProduitKey = Produit.ProduitKey
    JOIN Categorie ON Produit.CategorieKey = Categorie.CategorieKey
    JOIN Client ON Ventes.ClientKey = Client.ClientKey
    JOIN Ville ON Client.VilleKey = Ville.VilleKey
) AS SourceTable
PIVOT (
    SUM(Quantite)
    FOR NomCategorie IN
            ([Meat/Poultry], [Grains/Cereals], [Condiments], [Beverages])
) AS PivotTable;
```

Trimestre	VilleNom	Ventes_Viandes	Ventes_Cereales	Ventes_Condiments	Ventes_Boissons
1	Bruxelles	NULL	16	6	NULL
4	Bruxelles	NULL	NULL	NULL	30
1	Charleroi	NULL	95	40	20
2	Charleroi	NULL	30	NULL	NULL
3	Charleroi	NULL	NULL	NULL	12
4	Charleroi	NULL	2	NULL	NULL
1	Lyon	NULL	NULL	NULL	20
3	Lyon	NULL	21	NULL	NULL
1	Paris	NULL	NULL	NULL	19

TABLE 6.6 – Résultat de la requête utilisant l'opérateur PIVOT en SQL

Les approches précédemment démontrées offrent un résultat similaire. La différence majeure réside dans le fait que la clause CASE renvoie la valeur 0 dans le cas où il n'y a pas de mesure pour une combinaison spécifique de dimensions tandis que l'opérateur PIVOT renvoie la valeur NULL dans ce même cas. Le choix entre les approches réside donc dans la compréhension de l'utilisateur. Ces deux approches partagent toutefois un même inconvénient majeur. En effet, elles ne permettent pas de réaliser un pivot de façon dynamique. Ceci peut être résolu par l'utilisation de procédures stockées. La requête suivante, adaptée de celle proposée par Das, 2020, présente la mise en place d'une potentielle procédure stockée basée sur l'opérateur PIVOT. Celle-ci permet à l'utilisateur de définir la colonne ainsi que les valeurs de cette colonne qui seront utilisées pour réaliser le pivot. S'en suit un exemple d'appel à la procédure renvoyant un résultat similaire à la Table 6.6.

```
CREATE PROCEDURE [dbo].[DynamicPivot]
  @ColumnToPivot  NVARCHAR(255),
  @ListToPivot    NVARCHAR(255)
AS
BEGIN
  DECLARE @SqlStatement NVARCHAR(MAX)
  SET @SqlStatement = '
    SELECT *
        FROM (
            SELECT
                Trimestre,
                VilleNom,
                Categorie.NomCategorie,
                Quantite
            FROM Ventes
            JOIN DimDate ON Ventes.DateCommande = DimDate.Date
            JOIN Produit ON Ventes.ProduitKey = Produit.ProduitKey
            JOIN Categorie ON Produit.CategorieKey = Categorie.CategorieKey
            JOIN Client ON Ventes.ClientKey = Client.ClientKey
            JOIN Ville ON Client.VilleKey = Ville.VilleKey
        ) AS SourceTable
        PIVOT (
            SUM(Quantite)
            FOR ['+@ColumnToPivot+'] IN ('+ @ListToPivot + ')
        ) AS PivotTable';
  EXEC(@SqlStatement)
END


EXEC [dbo].[DynamicPivot]
    @ColumnToPivot = 'NomCategorie',
    @ListToPivot = '[Meat/Poultry], [Grains/Cereals], [Condiments], [Beverages]'
```

6.4.2 Syntaxe MDX

Nous ne retrouvons pas de fonction spécifique permettant de réaliser une opération de pivot en MDX. Cela n'est toutefois pas nécessaire. En effet, comme le langage agit directement sur un cube multidimensionnel, cela est aisément réalisable en intervertissant l'ordre dans lequel nous spécifions les axes de la requête.

Ainsi, la requête suivante nous permet d'obtenir un résultat semblable au cube de la Figure 3.1 :

```
SELECT Produit.Categorie.MEMBERS ON COLUMNS,
        DateCommande.Trimestre.MEMBERS ON ROWS,
        Client.Ville.MEMBERS ON PAGES
FROM Ventes
```

Supposons maintenant que le résultat attendu correspond en fait au cube de la Figure 5.5b. Cela est réalisable au moyen de la requête suivante :

```
SELECT DateCommande.Trimestre.MEMBERS ON COLUMNS,
        Client.Ville.MEMBERS ON ROWS,
        Produit.Categorie.MEMBERS ON PAGES
FROM Ventes
```

6.5 Sort

6.5.1 Requête SQL

L'opération de sort consistant simplement à trier les membres d'une dimension, cela peut être réalisé en SQL au moyen de la commande ORDER BY. La requête suivante permet d'illustrer cette application en renvoyant un résultat (Table 6.7) dont les lignes ont été triées par ordre alphabétique des noms de catégorie.

```
SELECT Trimestre, VilleNom, NomCategorie, SUM(Quantite) AS QuantiteTotale
FROM Ventes
JOIN DimDate ON Ventes.DateCommande = DimDate.Date
JOIN Produit ON Ventes.ProduitKey = Produit.ProduitKey
JOIN Categorie ON Produit.CategorieKey = Categorie.CategorieKey
JOIN Client ON Ventes.ClientKey = Client.ClientKey
JOIN Ville ON Client.VilleKey = Ville.VilleKey
GROUP BY VilleNom, Trimestre, NomCategorie
ORDER BY NomCategorie
```

Trimestre	VilleNom	NomCategorie	QuantiteTotale
4	Bruxelles	Beverages	30
1	Charleroi	Beverages	20
3	Charleroi	Beverages	12
1	Lyon	Beverages	20
1	Paris	Beverages	19
1	Bruxelles	Condiments	6
1	Charleroi	Condiments	40
1	Bruxelles	Grains/Cereals	16
1	Charleroi	Grains/Cereals	95
2	Charleroi	Grains/Cereals	30
4	Charleroi	Grains/Cereals	2
3	Lyon	Grains/Cereals	21

TABLE 6.7 – Résultat de la requête de sort en SQL

6.5.2 Requête MDX

Contrairement à SQL, le résultat d'une requête MDX est généralement ordonné selon les niveaux composant les différentes hiérarchies. Ainsi, dans le cas d'une requête concernant les ventes par pays, ces derniers seront affichés selon le continent auquel ils appartiennent. Si nous souhaitons modifier cela pour les classer par ordre alphabétique, par exemple, nous pouvons faire appel à la fonction ORDER. La requête suivante présente une utilisation de cette fonction :

```
SELECT Measures.Quantite ON COLUMNS,
        DateCommande.Trimestre.MEMBERS ON ROWS,
        ORDER(Client.Ville.MEMBERS, Measures.Quantite, BDESC) ON PAGES
FROM Ventes
```

Dans le résultat renvoyé par cette requête, les villes seront classées par ordre décroissant de quantité de vente. Notons que le B présent dans l'argument *BDESC* indique que l'ordre hiérarchique peut être outrepassé.

7 Conclusion

Ce mémoire a abordé en profondeur plusieurs concepts de la Business Intelligence (BI) en se servant de la base de données Northwind comme d'un fil conducteur et en se référant principalement au livre "Data Warehouse Systems" de Vaisman et Zimányi, 2014. En résumé, les principales contributions de ce mémoire peuvent être mises en évidence.

Tout d'abord, nous avons procédé à la traduction en français et à la présentation détaillée de la base de données Northwind qui a servi de terrain d'application pour illustrer divers concepts de la BI. Cette étape a permis de fournir le contexte nécessaire pour toutes les illustrations ultérieures.

Ensuite, nous avons expliqué divers concepts clés de la Business Intelligence de manière approfondie, en fournissant des illustrations systématiques basées sur la base de données Northwind. Les notions de systèmes OLTP et OLAP, de modélisation multidimensionnelle, de stockage de données en BI et d'opérations OLAP ont été exposées de manière claire et concrète, en utilisant Northwind comme exemple.

Le processus d'ETL a été abordé avec une mise en pratique sur la base de données Northwind. Nous avons démontré comment concevoir un data warehouse selon l'approche ROLAP, conformément aux enseignements de Vaisman et Zimányi, 2014. Des propositions de requêtes SQL ont été présentées pour illustrer l'application des opérations OLAP en SQL, et une introduction au langage MDX a été donnée pour compléter cette compréhension.

En envisageant des pistes pour des travaux futurs, plusieurs directions se profilent. Une extension naturelle serait l'application des concepts de la BI et des opérations OLAP dans le contexte des bases de données NoSQL, étant donné leur pertinence croissante. De plus, l'application du processus ETL pour concevoir un data warehouse selon l'approche MOLAP ouvrirait des perspectives pour comparer cette approche avec celle présentée dans ce mémoire sur base de critères tels que la rapidité d'exécution et la facilité de compréhension des requêtes. Enfin, l'exploration approfondie des opérations OLAP avec le langage MDX pourrait offrir une compréhension plus complète des capacités d'analyse multidimensionnelle.

En conclusion, ce mémoire a éclairé de façon détaillée les concepts clés de la Business Intelligence en les illustrant systématiquement avec des exemples concrets tirés de la base de données Northwind. Les fondements posés par Vaisman et Zimányi, 2014 ont constitué un guide précieux tout au long de cette exploration. La Business Intelligence continue de jouer un rôle essentiel dans la prise de décision, et ce mémoire espère avoir contribué à une meilleure compréhension et application de ces concepts dans un environnement en constante évolution.

Références

Grossmann, W., & Rinderle-Ma, S. (2015, juin 12). *Fundamentals of business intelligence*. Springer.

Vaisman, A., & Zimányi, E. (2014). *Data Warehouse Systems : Design and Implementation*. Springer. https://doi.org/10.1007/978-3-642-54655-6

Sinha, T. (2021, mars 16). *OLAP vs. OLTP: What's the difference? - IBM blog*. https://www.ibm.com/blog/olap-vs-oltp/

Inmon, W. H., & Hackathorn, R. D. (1994). *Using the Data Warehouse*. Wiley-QED Publishing.

Kimball, R., & Ross, M. (1996). *The Data Warehouse Toolkit : The Complete Guide to Dimensional Modeling*.

Malinowski, E., & Zimányi, E. (2006). Hierarchies in a multidimensional model: from conceptual modeling to logical representation. *Data and Knowledge Engineering*, *59*(2), 348-377. https://doi.org/10.1016/j.datak.2005.08.003

Iqbal, K. (2019, septembre 10). *XML file format*. https://docs.fileformat.com/web/xml/

Amazon Web Services. (s. d.). *Qu'est-ce que le XML ? – le langage de balisage extensible (XML) expliqué – AWS*. https://aws.amazon.com/fr/what-is/xml/

De Aguiar Ciferri, C. D., Ciferri, R. R., Gómez, L. I., Schneider, M., Vaisman, A., & Zimányi, E. (2013). Cube algebra. *International Journal of Data Warehousing and Mining*, *9*(2), 39-65. https://doi.org/10.4018/jdwm.2013040103

Whitehorn, M., Zare, R., & Pasumansky, M. (2005, décembre). *Fast Track to MDX* (2e éd.). Springer London. https://doi.org/10.1007/1-84628-182-2

Lemahieu, W., vanden Broucke, S., & Baesens, B. (2018). *Principles of Database Management : The Practical Guide to Storing, Managing and Analyzing Big and Small Data*. Cambridge University Press. https://doi.org/10.1017/9781316888773

Das, A. (2020, avril 2). *Dynamic Pivot Tables in SQL Server*. https://www.sqlshack.com/dynamic-pivot-tables-in-sql-server/

Annexes

A Script SQL - Création tables DB Northwind

```
/*
** Copyright Microsoft, Inc. 1994 - 2000
** All Rights Reserved.
*/

-- This script does not create a database.
-- Run this script in the database you want the objects to be created.
-- Default schema is dbo.

SET NOCOUNT ON
GO

set quoted_identifier on
GO

/* Set DATEFORMAT so that the date strings are interpreted correctly
   regardless of
   the default DATEFORMAT on the server.
*/
SET DATEFORMAT mdy
GO

GO
if exists (select * from sysobjects where id = object_id('dbo.Commande
   Details') and sysstat & 0xf = 3)
  drop table "dbo"."Commande Details"
GO
if exists (select * from sysobjects where id = object_id('dbo.Commandes
   ') and sysstat & 0xf = 3)
  drop table "dbo"."Commandes"
GO
if exists (select * from sysobjects where id = object_id('dbo.Produits'
   ) and sysstat & 0xf = 3)
  drop table "dbo"."Produits"
GO
if exists (select * from sysobjects where id = object_id('dbo.
   Categories') and sysstat & 0xf = 3)
  drop table "dbo"."Categories"
GO
if exists (select * from sysobjects where id = object_id('dbo.Clients')
    and sysstat & 0xf = 3)
  drop table "dbo"."Clients"
GO
if exists (select * from sysobjects where id = object_id('dbo.
   Transporteurs') and sysstat & 0xf = 3)
  drop table "dbo"."Transporteurs"
GO
if exists (select * from sysobjects where id = object_id('dbo.
```

```
   Fournisseurs') and sysstat & 0xf = 3)
  drop table "dbo"."Fournisseurs"
GO
if exists (select * from sysobjects where id = object_id('dbo.
   EmployeTerritoires') and sysstat & 0xf = 3)
  drop table "dbo"."EmployeTerritoires"
GO
if exists (select * from sysobjects where id = object_id('dbo.
   Territoires') and sysstat & 0xf = 3)
  drop table "dbo".Territoires
GO
if exists (select * from sysobjects where id = object_id('dbo.Region')
   and sysstat & 0xf = 3)
  drop table "dbo".Region
GO
if exists (select * from sysobjects where id = object_id('dbo.Employes'
   ) and sysstat & 0xf = 3)
  drop table "dbo"."Employes"
GO

CREATE TABLE "Employes" (
  "EmployeID" "int" IDENTITY (1, 1) NOT NULL ,
  "Nom" nvarchar (20) NOT NULL ,
  "Prenom" nvarchar (10) NOT NULL ,
  "Titre" nvarchar (30) NULL ,
  "TitrePolitesse" nvarchar (25) NULL ,
  "DateNaissance" "datetime" NULL ,
  "DateEmbauche" "datetime" NULL ,
  "Adresse" nvarchar (60) NULL ,
  "Ville" nvarchar (15) NULL ,
  "Region" nvarchar (15) NULL ,
  "CodePostal" nvarchar (10) NULL ,
  "Pays" nvarchar (15) NULL ,
  "TelephoneFixe" nvarchar (24) NULL ,
  "Extension" nvarchar (4) NULL ,
  "Photo" "image" NULL ,
  "Notes" "ntext" NULL ,
  "SuperieurHierarchique" "int" NULL ,
  "AccesPhoto" nvarchar (255) NULL ,
  CONSTRAINT "PK_Employes" PRIMARY KEY  CLUSTERED
  (
    "EmployeID"
  ),
  CONSTRAINT "FK_Employes_Employes" FOREIGN KEY
  (
    "SuperieurHierarchique"
  ) REFERENCES "dbo"."Employes" (
    "EmployeID"
  ),
  CONSTRAINT "CK_DateNaissance" CHECK (DateNaissance < getdate())
)
GO
 CREATE  INDEX "Nom" ON "dbo"."Employes"("Nom")
GO
 CREATE  INDEX "CodePostal" ON "dbo"."Employes"("CodePostal")
```

```
92  GO
93
94  CREATE TABLE "Categories" (
95    "CategorieID" "int" IDENTITY (1, 1) NOT NULL ,
96    "CategorieNom" nvarchar (20) NOT NULL ,
97    "Description" "ntext" NULL ,
98    "Image" "image" NULL ,
99    CONSTRAINT "PK_Categories" PRIMARY KEY  CLUSTERED
100   (
101     "CategorieID"
102   )
103 )
104 GO
105  CREATE  INDEX "CategorieNom" ON "dbo"."Categories"("CategorieNom")
106 GO
107
108 CREATE TABLE "Clients" (
109   "ClientID" nchar (5) NOT NULL ,
110   "EntrepriseNom" nvarchar (40) NOT NULL ,
111   "ContactNom" nvarchar (30) NULL ,
112   "ContactPoste" nvarchar (35) NULL ,
113   "Adresse" nvarchar (60) NULL ,
114   "Ville" nvarchar (15) NULL ,
115   "Region" nvarchar (15) NULL ,
116   "CodePostal" nvarchar (10) NULL ,
117   "Pays" nvarchar (15) NULL ,
118   "Telephone" nvarchar (24) NULL ,
119   "Fax" nvarchar (24) NULL ,
120   CONSTRAINT "PK_Clients" PRIMARY KEY  CLUSTERED
121   (
122     "ClientID"
123   )
124 )
125 GO
126  CREATE  INDEX "Ville" ON "dbo"."Clients"("Ville")
127 GO
128  CREATE  INDEX "EntrepriseNom" ON "dbo"."Clients"("EntrepriseNom")
129 GO
130  CREATE  INDEX "CodePostal" ON "dbo"."Clients"("CodePostal")
131 GO
132  CREATE  INDEX "Region" ON "dbo"."Clients"("Region")
133 GO
134
135 CREATE TABLE "Transporteurs" (
136   "TransporteurID" "int" IDENTITY (1, 1) NOT NULL ,
137   "EntrepriseNom" nvarchar (40) NOT NULL ,
138   "Telephone" nvarchar (24) NULL ,
139   CONSTRAINT "PK_Transporteurs" PRIMARY KEY  CLUSTERED
140   (
141     "TransporteurID"
142   )
143 )
144 GO
145 CREATE TABLE "Fournisseurs" (
146   "FournisseurID" "int" IDENTITY (1, 1) NOT NULL ,
```

```
  "EntrepriseNom" nvarchar (40) NOT NULL ,
  "ContactNom" nvarchar (30) NULL ,
  "ContactPoste" nvarchar (40) NULL ,
  "Adresse" nvarchar (60) NULL ,
  "Ville" nvarchar (15) NULL ,
  "Region" nvarchar (15) NULL ,
  "CodePostal" nvarchar (10) NULL ,
  "Pays" nvarchar (15) NULL ,
  "Telephone" nvarchar (24) NULL ,
  "Fax" nvarchar (24) NULL ,
  "PageAccueil" "ntext" NULL ,
  CONSTRAINT "PK_Fournisseurs" PRIMARY KEY  CLUSTERED
  (
    "FournisseurID"
  )
)
GO
 CREATE  INDEX "EntrepriseNom" ON "dbo"."Fournisseurs"("EntrepriseNom")
GO
 CREATE  INDEX "CodePostal" ON "dbo"."Fournisseurs"("CodePostal")
GO

CREATE TABLE "Commandes" (
  "CommandeID" "int" IDENTITY (1, 1) NOT NULL ,
  "ClientID" nchar (5) NULL ,
  "EmployeID" "int" NULL ,
  "DateCommande" "datetime" NULL ,
  "DateRequise" "datetime" NULL ,
  "DateEnvoi" "datetime" NULL ,
  "EnvoiPar" "int" NULL ,
  "Fret" "money" NULL CONSTRAINT "DF_Commandes_Fret" DEFAULT (0),
  "NomLivraison" nvarchar (40) NULL ,
  "AdresseLivraison" nvarchar (60) NULL ,
  "VilleLivraison" nvarchar (15) NULL ,
  "RegionLivraison" nvarchar (15) NULL ,
  "CodePostalLivraison" nvarchar (10) NULL ,
  "PaysLivraison" nvarchar (15) NULL ,
  CONSTRAINT "PK_Commandes" PRIMARY KEY  CLUSTERED
  (
    "CommandeID"
  ),
  CONSTRAINT "FK_Commandes_Clients" FOREIGN KEY
  (
    "ClientID"
  ) REFERENCES "dbo"."Clients" (
    "ClientID"
  ),
  CONSTRAINT "FK_Commandes_Employes" FOREIGN KEY
  (
    "EmployeID"
  ) REFERENCES "dbo"."Employes" (
    "EmployeID"
  ),
  CONSTRAINT "FK_Commandes_Transporteurs" FOREIGN KEY
  (
```

```
    "EnvoiPar"
  ) REFERENCES "dbo"."Transporteurs" (
    "TransporteurID"
  )
)
GO
 CREATE  INDEX "ClientID" ON "dbo"."Commandes"("ClientID")
GO
 CREATE  INDEX "ClientsCommandes" ON "dbo"."Commandes"("ClientID")
GO
 CREATE  INDEX "EmployeID" ON "dbo"."Commandes"("EmployeID")
GO
 CREATE  INDEX "EmployesCommandes" ON "dbo"."Commandes"("EmployeID")
GO
 CREATE  INDEX "DateCommande" ON "dbo"."Commandes"("DateCommande")
GO
 CREATE  INDEX "DateEnvoi" ON "dbo"."Commandes"("DateEnvoi")
GO
 CREATE  INDEX "TransporteursCommandes" ON "dbo"."Commandes"("EnvoiPar"
   )
GO
 CREATE  INDEX "CodePostalLivraison" ON "dbo"."Commandes"("
   CodePostalLivraison")
GO

CREATE TABLE "Produits" (
  "ProduitID" "int" IDENTITY (1, 1) NOT NULL ,
  "NomProduit" nvarchar (40) NOT NULL ,
  "FournisseurID" "int" NULL ,
  "CategorieID" "int" NULL ,
  "QuantiteParUnite" nvarchar (30) NULL ,
  "PrixUnitaire" "money" NULL CONSTRAINT "DF_Produits_PrixUnitaire"
   DEFAULT (0),
  "UnitesEnStock" "smallint" NULL CONSTRAINT "DF_Produits_UnitesEnStock
   " DEFAULT (0),
  "UnitesEnCommande" "smallint" NULL CONSTRAINT "
   DF_Produits_UnitesEnCommande" DEFAULT (0),
  "NiveauReapprovisionnement" "smallint" NULL CONSTRAINT "
   DF_Produits_NiveauReapprovisionnement" DEFAULT (0),
  "Discontinu" "bit" NOT NULL CONSTRAINT "DF_Produits_Discontinu"
   DEFAULT (0),
  CONSTRAINT "PK_Produits" PRIMARY KEY  CLUSTERED
  (
    "ProduitID"
  ),
  CONSTRAINT "FK_Produits_Categories" FOREIGN KEY
  (
    "CategorieID"
  ) REFERENCES "dbo"."Categories" (
    "CategorieID"
  ),
  CONSTRAINT "FK_Produits_Fournisseurs" FOREIGN KEY
  (
    "FournisseurID"
  ) REFERENCES "dbo"."Fournisseurs" (
```

```
250     "FournisseurID"
251   ),
252   CONSTRAINT "CK_Produits_PrixUnitaire" CHECK (PrixUnitaire >= 0),
253   CONSTRAINT "CK_NiveauReapprovisionnement" CHECK (
      NiveauReapprovisionnement >= 0),
254   CONSTRAINT "CK_UnitesEnStock" CHECK (UnitesEnStock >= 0),
255   CONSTRAINT "CK_UnitesEnCommande" CHECK (UnitesEnCommande >= 0)
256 )
257 GO
258  CREATE  INDEX "CategoriesProduits" ON "dbo"."Produits"("CategorieID")
259 GO
260  CREATE  INDEX "CategorieID" ON "dbo"."Produits"("CategorieID")
261 GO
262  CREATE  INDEX "NomProduit" ON "dbo"."Produits"("NomProduit")
263 GO
264  CREATE  INDEX "FournisseurID" ON "dbo"."Produits"("FournisseurID")
265 GO
266  CREATE  INDEX "FournisseursProduits" ON "dbo"."Produits"("
       FournisseurID")
267 GO
268
269 CREATE TABLE "Commande Details" (
270   "CommandeID" "int" NOT NULL ,
271   "ProduitID" "int" NOT NULL ,
272   "PrixUnitaire" "money" NOT NULL CONSTRAINT "
      DF_Commande_Details_PrixUnitaire" DEFAULT (0),
273   "Quantite" "smallint" NOT NULL CONSTRAINT "
      DF_Commande_Details_Quantite" DEFAULT (1),
274   "Remise" "real" NOT NULL CONSTRAINT "DF_Commande_Details_Remise"
      DEFAULT (0),
275   CONSTRAINT "PK_Commande_Details" PRIMARY KEY  CLUSTERED
276   (
277     "CommandeID",
278     "ProduitID"
279   ),
280   CONSTRAINT "FK_Commande_Details_Commandes" FOREIGN KEY
281   (
282     "CommandeID"
283   ) REFERENCES "dbo"."Commandes" (
284     "CommandeID"
285   ),
286   CONSTRAINT "FK_Commande_Details_Produits" FOREIGN KEY
287   (
288     "ProduitID"
289   ) REFERENCES "dbo"."Produits" (
290     "ProduitID"
291   ),
292   CONSTRAINT "CK_Remise" CHECK (Remise >= 0 and (Remise <= 1)),
293   CONSTRAINT "CK_Quantite" CHECK (Quantite > 0),
294   CONSTRAINT "CK_PrixUnitaire" CHECK (PrixUnitaire >= 0)
295 )
296 GO
297  CREATE  INDEX "CommandeID" ON "dbo"."Commande Details"("CommandeID")
298 GO
299  CREATE  INDEX "CommandesCommande_Details" ON "dbo"."Commande Details"(
```

```
    "CommandeID")
GO
 CREATE  INDEX "ProduitID" ON "dbo"."Commande Details"("ProduitID")
GO
 CREATE  INDEX "ProduitsCommande_Details" ON "dbo"."Commande Details"("
    ProduitID")
GO

/* Here should be the script to populate the tables */

/* The following adds tables to the Northwind database */

CREATE TABLE [dbo].[Region]
  ( [RegionID] [int] NOT NULL ,
  [RegionDescription] [nchar] (50) NOT NULL
) ON [PRIMARY]
GO

CREATE TABLE [dbo].[Territoires]
  ([TerritoireID] [nvarchar] (20) NOT NULL ,
  [TerritoireDescription] [nchar] (50) NOT NULL ,
        [RegionID] [int] NOT NULL
) ON [PRIMARY]
GO

CREATE TABLE [dbo].[EmployeTerritoires]
  ([EmployeID] [int] NOT NULL,
  [TerritoireID] [nvarchar] (20) NOT NULL
) ON [PRIMARY]

/* Here should be the code to populate the newly added tables */

--  The following adds constraints to the Northwind database


ALTER TABLE Region
  ADD CONSTRAINT [PK_Region] PRIMARY KEY  NONCLUSTERED
  (
    [RegionID]
  )  ON [PRIMARY]
GO

ALTER TABLE Territoires
  ADD CONSTRAINT [PK_Territoires] PRIMARY KEY  NONCLUSTERED
  (
    [TerritoireID]
  )  ON [PRIMARY]
GO

ALTER TABLE Territoires
  ADD CONSTRAINT [FK_Territoires_Region] FOREIGN KEY
  (
    [RegionID]
  ) REFERENCES [dbo].[Region] (
    [RegionID]
```

```
  )
GO

ALTER TABLE EmployeTerritoires
  ADD CONSTRAINT [PK_EmployeTerritoires] PRIMARY KEY  NONCLUSTERED
  (
    [EmployeID],
    [TerritoireID]
  ) ON [PRIMARY]
GO

ALTER TABLE EmployeTerritoires
  ADD CONSTRAINT [FK_EmployeTerritoires_Employes] FOREIGN KEY
  (
    [EmployeID]
  ) REFERENCES [dbo].[Employes] (
    [EmployeID]
  )
GO


ALTER TABLE EmployeTerritoires
  ADD CONSTRAINT [FK_EmployeTerritoires_Territoires] FOREIGN KEY
  (
    [TerritoireID]
  ) REFERENCES [dbo].[Territoires] (
    [TerritoireID]
  )
GO
```

B Script SQL - Création dimension temporelle

```
IF ( EXISTS ( SELECT * FROM INFORMATION_SCHEMA . TABLES WHERE
   TABLE_SCHEMA = 'dbo' AND TABLE_NAME = 'DimDate'))
BEGIN
DROP TABLE DimDate
END
go

GO
CREATE TABLE [dbo].[DimDate](
[Date] [datetime] NOT NULL PRIMARY KEY,
[JourMois] [int],
[Mois] [int],
[MoisNom] [nvarchar](9),
[Trimestre] [int],
[Annee] [int],
[Semaine] [int] ,
[Saison] [nvarchar](9)
)
GO
/* ********************************************** */
-- Specify Start Date and End date here
--Value of Start Date Must be Less than Your End Date

DECLARE @StartDate DATETIME = '01/01/1996' -- Starting value of Date
   Range
DECLARE @EndDate DATETIME = '31/12/1998'  --End Value of Date Range

-- Temporary Variables To Hold the Values During Processing of Each
   Date of Year

DECLARE
@DayOfWeekInMonth INT ,
@DayOfWeekInYear INT ,
@DayOfQuarter INT ,
@WeekOfMonth INT ,
@CurrentYear INT ,
@CurrentMonth INT ,
@CurrentQuarter INT

/* Table Data type to store the day of week count for the month and
   year */
DECLARE @DayOfWeek TABLE (DOW INT , MonthCount INT , QuarterCount INT ,
   YearCount INT)

INSERT INTO @DayOfWeek VALUES (1, 0, 0, 0)
INSERT INTO @DayOfWeek VALUES (2, 0, 0, 0)
INSERT INTO @DayOfWeek VALUES (3, 0, 0, 0)
INSERT INTO @DayOfWeek VALUES (4, 0, 0, 0)
INSERT INTO @DayOfWeek VALUES (5, 0, 0, 0)
INSERT INTO @DayOfWeek VALUES (6, 0, 0, 0)
INSERT INTO @DayOfWeek VALUES (7, 0, 0, 0)

```

```
48 -- Extract and assign various parts of Values from Current Date to
       Variable
49
50 DECLARE @CurrentDate AS DATETIME = @StartDate
51 SET @CurrentMonth = DATEPART (MM , @CurrentDate )
52 SET @CurrentYear = DATEPART (YY , @CurrentDate )
53 SET @CurrentQuarter = DATEPART (QQ , @CurrentDate )
54
55 /* ********************************************** */
56 -- Proceed only if Start Date ( Current date ) is less than End date
       you specified above
57
58 WHILE @CurrentDate < @EndDate
59 BEGIN
60
61 /* Begin day of week logic */
62
63     /* Check for Change in Month of the Current date if Month changed
       then Change variable value */
64 IF @CurrentMonth != DATEPART (MM , @CurrentDate )
65 BEGIN
66   UPDATE @DayOfWeek
67   SET MonthCount = 0
68   SET @CurrentMonth = DATEPART (MM , @CurrentDate )
69 END
70
71     /* Check for Change in Quarter of the Current date if Quarter
       changed then change Variable value */
72
73 IF @CurrentQuarter != DATEPART (QQ , @CurrentDate )
74 BEGIN
75   UPDATE @DayOfWeek
76   SET QuarterCount = 0
77   SET @CurrentQuarter = DATEPART (QQ , @CurrentDate )
78 END
79
80 /* Check for Change in Year of the Current date if Year changed then
       change Variable value */
81
82
83 IF @CurrentYear != DATEPART (YY , @CurrentDate )
84 BEGIN
85   UPDATE @DayOfWeek
86   SET YearCount = 0
87   SET @CurrentYear = DATEPART (YY , @CurrentDate )
88 END
89
90 -- Set values in table data type created above from variables
91
92 UPDATE @DayOfWeek
93 SET
94   MonthCount = MonthCount + 1,
95   QuarterCount = QuarterCount + 1,
96   YearCount = YearCount + 1
97 WHERE DOW = DATEPART (DW , @CurrentDate )
```

```
98
99  SELECT
100   @DayOfWeekInMonth = MonthCount ,
101   @DayOfQuarter = QuarterCount ,
102   @DayOfWeekInYear = YearCount
103 FROM @DayOfWeek
104 WHERE DOW = DATEPART (DW , @CurrentDate )
105
106 /* End day of week logic */
107
108
109 /* Populate Your Dimension Table with values */
110
111 INSERT INTO [dbo].[DimDate]
112
113 SELECT
114   @CurrentDate AS Date ,
115   DATEPART (DD , @CurrentDate ) AS JourMois ,
116   DATEPART (MM , @CurrentDate ) AS Mois ,
117   DATENAME (MM , @CurrentDate ) AS MoisNom ,
118   DATEPART (QQ , @CurrentDate ) AS Trimestre ,
119   DATEPART (YEAR , @CurrentDate ) AS Annee ,
120   DATEPART (WW , @CurrentDate ) AS Semaine ,
121   CASE
122     WHEN DATEPART (DY , @CurrentDate ) BETWEEN 80 and 172 THEN '
      Printemps'
123     WHEN DATEPART (DY , @CurrentDate ) BETWEEN 173 and 264 THEN '  t'
124     WHEN DATEPART (DY , @CurrentDate ) BETWEEN 265 and 355 THEN '
     Automne'
125     ELSE 'Hiver'
126     END AS Saison
127
128   SET @CurrentDate = DATEADD (DD , 1, @CurrentDate )
129 END
130
131 /* ********************************************** */
132 go
133 SELECT * FROM [dbo].[DimDate]
```

C Script SQL - Création tables DW Northwind

```
CREATE TABLE [VilleTemp](
[Ville] [nvarchar](15),
[ tat /Province] [nvarchar](15),
[Pays] [nvarchar](15)
);

CREATE TABLE [Continent](
[ContinentKey] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
[ContinentNom] [nvarchar](15)
);

CREATE TABLE [Pays](
[PaysKey] IDENTITY(1,1) NOT NULL PRIMARY KEY,
[PaysNom] [nvarchar](15),
[PaysCode] [nvarchar](15),
[PaysCapitale] [nvarchar](15),
[Population] [int],
[Sous-division] [ntext],
[ContinentKey] [int] FOREIGN KEY REFERENCES Continent(ContinentKey)
);

CREATE TABLE [Etat](
[EtatKey] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
[EtatNom] [nvarchar](15),
[EtatNomAnglais] [nvarchar](15),
[EtatType] [nvarchar](15),
[EtatCode] [nvarchar](15),
[EtatCapitale] [nvarchar](15),
[RegionNom] [nvarchar](15),
[RegionCode] [nvarchar](15),
[PaysKey] [int] FOREIGN KEY REFERENCES Pays(PaysKey)
);

CREATE TABLE [Ville](
[VilleKey] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
[VilleNom] [nvarchar](15),
[EtatKey] [int] FOREIGN KEY REFERENCES Etat(EtatKey),
[PaysKey] [int] FOREIGN KEY REFERENCES Pays(PaysKey)
);

CREATE TABLE [Client](
[ClientKey] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
[ClientID] [nvarchar](5),
[EntrepriseNom] [nvarchar](40),
[Adresse] [nvarchar](60),
[CodePostal] [nvarchar](10),
[VilleKey] [int] FOREIGN KEY REFERENCES Ville(VilleKey),
);

CREATE TABLE [DimDate](
[DateKey] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
[Date] [date],
```

```
[NumJourSemaine] [int],
[JourSemaine] [varchar](50),
[NumJourMois] [int],
[NumMois] [int],
[Mois] [varchar](50),
[Anne] [int],
[Semestre] [int],
[Trimestre] [int]
);

CREATE TABLE [Employe](
[EmployeKey] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
[Nom] [nvarchar](20),
[Prenom] [nvarchar](10),
[Poste] [nvarchar](30),
[DateNaissance] [date],
[DateEmbauche] [date],
[Adresse] [nvarchar](60),
[Ville] [nvarchar](15),
[Region] [nvarchar](15),
[CodePostal] [nvarchar](10),
[Pays] [nvarchar](15),
[SuperieurHierarchiqueKey] [int] FOREIGN KEY REFERENCES Employe(
    EmployeKey)
);

CREATE TABLE [Categorie](
[CategorieKey] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
[NomCategorie] [nvarchar](15),
[Description] [ntext]
);

CREATE TABLE [Produit](
[ProduitKey] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
[NomProduit] [nvarchar](40),
[QuantiteParUnite] [nvarchar](30),
[PrixUnitaire] [money],
[Discontinu] [bit],
[CategoryKey] [int] FOREIGN KEY REFERENCES Categorie(CategorieKey),
[NomCatgorie] [nvarchar](15)
);

CREATE TABLE [Fournisseur](
[FournisseurKey] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
[EntrepriseNom] [nvarchar](40),
[Adresse] [nvarchar](60),
[CodePostal] [nvarchar](10),
[VilleKey] [int] FOREIGN KEY REFERENCES Ville(VilleKey)
);

CREATE TABLE [Transporteur](
[TransporteurKey] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
[EntrepriseNom] [nvarchar](40)
);

```

```
CREATE TABLE [Territoires](
[EmployeKey] [int] FOREIGN KEY REFERENCES Employe(EmployeKey),
[VilleKey] [int] FOREIGN KEY REFERENCES Ville(VilleKey)
PRIMARY KEY (EmployeKey, VilleKey)
);

CREATE TABLE [Ventes](
[ClientKey] [int] FOREIGN KEY REFERENCES Client(ClientKey),
[EmployeKey] [int] FOREIGN KEY REFERENCES Employe(EmployeKey),
[DateCommandeKey] [int] FOREIGN KEY REFERENCES DimDate(DateKey),
[DateRequiseKey] [int] FOREIGN KEY REFERENCES DimDate(DateKey),
[DateEnvoiKey] [int] FOREIGN KEY REFERENCES DimDate(DateKey),
[TransporteurKey] [int] FOREIGN KEY REFERENCES Transporteur(
   TransporteurKey),
[ProduitKey] [int] FOREIGN KEY REFERENCES Produit(ProduitKey),
[FournisseurKey] [int] FOREIGN KEY REFERENCES Fournisseur(
   FournisseurKey),
[PrixUnitaire] [money],
[Quantite][smallint],
[Discount] [real],
[Fret] [money]
PRIMARY KEY (
  ClientKey,
  EmployeKey,
  DateCommandeKey,
  DateRequiseKey,
  DateEnvoiKey,
  TransporteurKey,
  ProduitKey,
  FournisseurKey)
);
```