Académie de Montpellier
Université Montpellier II
Sciences et Techniques du Languedoc

MÉMOIRE DE STAGE DE MASTER 2

effectué au Laboratoire d'Informatique de Robotique
et de Micro-électronique de Montpellier

Spécialité : **Professionelle et Recherche unifiée en Informatique**

Fouille de flot de données multidimensionnelles

par **Yoann PITARCH**

Sous la direction de :

Anne LAURENT, Marc PLANTEVIT, Pascal PONCELET

Date de soutenance : **1 juillet 2008**

Résumé

Avec le développement des nouvelles technologies, de nombreuses applications (*e.g.* surveillance en temps réel, analyse du trafic, analyse de transactions financières, etc.) doivent faire face à un flot éventuellement infini de données multidimensionnelles. Dans un contexte aussi dynamique, il n'est plus possible pour le décideur d'utiliser toutes les données à un faible niveau de granularité et il devient indispensable de proposer de nouvelles approches d'agrégation prenant en compte ces différentes contraintes. Nous adaptons les technologies OLAP à un contexte temps réel pour proposer une structure qui (1) permette une analyse multidimensionnelle et multi-niveaux efficace et (2) satisfasse une contrainte critique dans les flots de données : l'espace de stockage. Traditionnellement, l'historique des données de granularité faible n'est consulté que sur un passé proche et vouloir les stocker après ce délai devient superflu. Par conséquent, nous proposons de les agréger en fonction de l'évolution du flot au cours du temps passé en étendant le principe de fenêtres temporelles à toutes les dimensions hiérarchisées. Nous introduisons ainsi les *fonctions de précision* pour déterminer à quel moment un niveau de granularité devient superflu. Ces fonctions sont ensuite combinées afin de proposer une structure compacte et rapidement maintenable. Une méthode automatique pour construire cette structure en fonction des habitudes des utilisateurs est décrite. Enfin, nous proposons une technique pour interroger la structure proposée.

Abstract

Real-time surveillance systems, traffic analysis systems, and other dynamic environments often generate tremendous (potentially infinite) volume of multi-dimensional stream data. This volume is too huge to be scanned multiple times. Moreover, this data goes around at rather low level of abstraction. So it is unrealistic to stock such data for two main reasons. The first one is due to technical limits on today's computers. Second, analysts are mostly interested in higher levels of abstraction. To discover such high-level characteristics, one may need to perform on-line multi-level and multi-dimensional analytical processing of stream data. We propose a compact architecture to perform such analysis. Since time and space are critical, our architecture is based on two techniques. First, a *tilted-time model* is used to compress the temporal dimension : the more recent is the data, the finer it is registered. Second, recent data are mostly interrogated on fine precision levels. So, we extend the *tilted-time model* to other multi-level dimensions : precision levels wich are never interrogated are not materialized. Based on this design methodology, a cube can be constructed and maintained incrementally with a low amount of memory and a reasonable computation cost. Moreover, we automate the cube construction thanks to query logs. At last, a way to query our framework is proposed.

Table des matières

Remerciements

Je tiens naturellement à remercier mes encadrants Anne Laurent, Marc Plantevit et Pascal Poncelet pour leur grande disponibilité et leur encadrement de qualité. Grâce à leur complémentarité, ils ont su m'accompagner sans me brider.

Ce stage ne se serait pas si bien déroulé sans une ambiance générale sereine et conviviale. Merci donc à l'ensemble de l'équipe TATOO. J'en profite pour remercier Chedy pour son aide lors de ma phase d'implémentation.

Je remercie également mes deux rapportrices Mountaz Hascoët et Thérèse Libourel pour avoir accepté de lire le présent mémoire. J'espère simplement que la lecture sera agréable.

Je tiens à remercier l'INRIA pour avoir financé ce travail.

Comme le veut la coutume, je remercie mes proches (parents, sœur, copine et amis) pour leur soutien. Vous n'aurez certainement pas le courage et l'envi de lire l'intégralité de ce mémoire mais puissent ces quelques mots vous atteindre malgré tout.

Chapitre 1

Introduction

1.1 Contexte général

Ces dernières années, les supports de stockage ont connu un développement considérable. Parallèlement, l'évolution des technologies de communication a elle aussi été rapide. Ainsi, de plus en plus de données circulent sous la forme de flots (*data streams*) de façon continue, à un rythme rapide et éventuellement de manière infinie. Ces flots de données interviennent dans des domaines aussi variés que le trafic TCP/IP, les transactions financières en ligne, les réseaux de capteurs, les flots textuels (dépêches AFP, bourse), etc. De plus, la majorité des données circulant dans ces flots ont un caractère multidimensionnel. Par exemple, l'entête d'un paquet TCP se décompose en plusieurs champs tels que l'adresse IP de l'expéditeur, le port du destinataire, etc. Par conséquent, même si les capacités de stockage des disques durs augmentent, il est impossible de stocker intégralement ces flots. Dès lors, les techniques classiques d'interrogation et d'extraction de connaissances sur des données statiques deviennent inadaptées à un contexte aussi dynamique puisque la majorité de ces approches requiert plusieurs passages sur les données. Ces spécificités amènent à proposer de nouvelles structures de résumé et des méthodes originales pour interroger et fouiller ces données. Ainsi, de nombreux travaux de recherche ont été récemment proposées [Agg07] pour répondre au principal challenge induit par cette dynamicité : *trouver le meilleur compromis entre l'efficacité (*e.g. *le traitement d'une valeur d'un flot doit être assez rapide pour ne pas le bloquer) et la précision des résultats (*e.g. *puisque il est impossible de tout stocker, des approximations ou des imprécisions sont acceptables) ?*

La plupart des approches proposées pour résumer un flot approximent les données grâce à des outils mathématiques tels que les ondelettes, les régressions linéaires, etc. Ces approches ont le mérite d'avoir des bonnes propriétés en terme d'applicabilité, d'efficacité mémoire et de robustesse mais sont inadaptées face à des données multidimensionnelles. Dans le cadre de ce stage, nous proposons donc une méthode permettant de traiter un flot de données multidimensionnelles en les agrégeant sous la forme de cube de données. Les cubes de données sont couramment utilisés dans les entrepôts de données (*data*

warehouses) et permettent une analyse efficace d'un grand volume de données. Cette analyse se révèle utile dans un processus d'aide à la décision. En effet, cette structure logique autorise le déploiement d'outils OLAP qui proposent des opérateurs fonctionnels pour manipuler un cube. Par exemple, une dimension peut souvent être considérée selon plusieurs niveaux de précision. Les opérateurs OLAP de généralisation (*roll-up*) et de spécialisation (*drill-down*) supportent cette vision multi-niveaux des données. Ainsi, l'analyse des ventes d'une chaîne de magasins pourra être indifféremment effectuée au niveau du magasin, du département ou du pays. L'adaptation de telles techniques dans un contexte dynamique faciliterait grandement l'analyse de flots de données et contribuerait ainsi à entretenir la réactivité des décideurs en leur offrant une meilleure appréhension de leurs données.

Outre, les différentes contraintes énoncées, les flots de données multidimensionnelles doivent faire face au problème suivant : généralement, les données d'un flot sont observables à un faible niveau de granularité (*i.e.* un niveau trop précis pour être exploitable par un décideur) et il est bien entendu impossible pour un décideur d'observer et d'analyser ces données sans que celles-ci ne soient agrégées.

Pour s'en convaincre, considérons le cas d'une étude de la consommation électrique d'une population. Il est irréaliste, dans un cadre d'aide à la décision (*e.g.* pour connaître les tendances générales de la population) de travailler sur les données brutes comme la consommation par minute et par compteur. Dans ce contexte, un niveau de précision intéressant pourrait être la consommation électrique par rue et par heure des habitants de la population étudiée. Ainsi, une question se pose : *comment agréger les données à un niveau de granularité suffisant pour pouvoir aider le décideur sachant que ces données sont disponibles sous la forme de flot et qu'on ne peut y accéder qu'une seule fois ?*

L'objectif principal de ce travail est donc de proposer une technique de construction et de mise à jour d'un cube de données alimenté par les données d'un flot. Dans les sections suivantes, nous présentons un aperçu des bases de données multidimensionnelles et des flots de données. Une description plus formelle de ces domaines est proposée dans le chapitre 3. Nous présentons ensuite les principales contributions de ce mémoire et terminons ce chapitre en énonçant le plan du mémoire.

1.2 Les bases de données multidimensionnelles

Les bases de données multidimensionnelles se sont naturellement imposées dans le monde industriel car elles pallient les limites des bases de données relationnelles. En effet, dans un contexte d'analyse décisionnelle, l'utilisateur a besoin d'informations agrégées, résumées et observables sur plusieurs niveaux de précision. L'intêret des décideurs ne se situe alors plus dans l'analyse détaillée des individus mais plutôt dans la recherche de tendances générales concernant un ensemble d'individus. Dès lors, un modèle de traitement OLTP (*On-Line Transaction Processing*) pour réaliser de telles opérations est inadéquat car de nombreuses et coûteuses jointures seraient nécessaires et dégraderaient

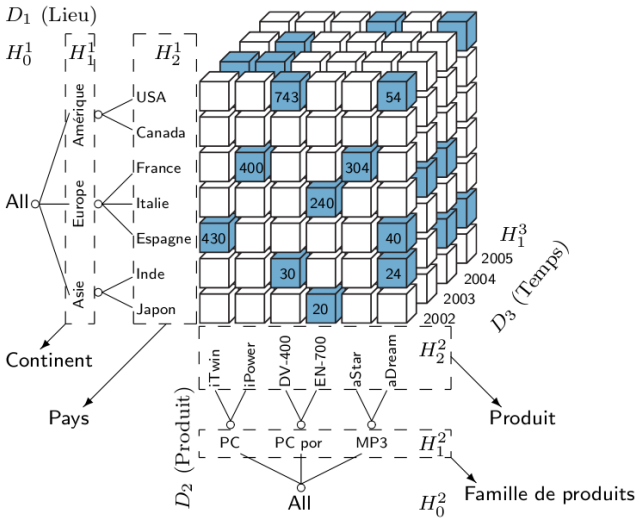


FIG. 1.1: Représentation graphique d'un cube de données [Mes06]

les performances du système. Pour répondre à ce besoin et permettre des réponses efficaces à des requêtes complexes posées sur un grand volume de données, les entrepôts de données sont apparus. Une définition de ces entrepôts est donnée dans [Inm96] : *A datawarehouse is a subject-oriented, integrated, time-variant and non-volatile collection of data in support of management decision making progress.* Un entrepôt est donc un ensemble de données organisées autour de la même thématique dans un objectif de prise de décision. Alors, pour analyser les données d'un entrepôt et faciliter le processus, un ensemble d'applications a été proposé pour traiter des données multidimensionnelles : les outils OLAP (*On-Line Analytical Processing*).

Les bases de données multidimensionnelles permettent donc d'analyser, de synthétiser les informations et représentent à ce titre une alternative intéréssante aux systèmes relationnels classiques. Les avantages des bases de données multidimensionnelles sont les suivants :

- Gestion optimisée des agrégats,
- Navigation aisée le long des hiérarchies des dimensions,
- Visualisation intuitive des données

Généralement, les données sont stockées dans des (hyper)cubes. Un cube est composé de cuboïdes où un cuboïde est un ensemble de données multidimensionnelles dont les valeurs d'une dimension sont définies sur le même niveau de précision. Le cuboïde dont toutes les dimensions sont définies sur leur plus fin niveau de précision est appelé *cuboïde de base*. Un cuboïde est composé de cellules où une cellule est une donnée définie par une position de chaque dimension. Pour illustrer ces concepts, considérons la figure 1.1 extraite de [Mes06]. Celle-ci présente un cube de données défini sur 3 dimensions, dont

2 sont hiérarchisées. $\langle Produits, Pays, Temps\rangle$ et $\langle Famille\,de\,produits, Continent\rangle$ sont deux cuboïdes. Le premier est le *cuboïde de base* et stocke les ventes de produits par pays et par année alors que le second matérialise les ventes par famille de produits et par continent toutes années confondues. $\langle(Espagne, iTwin, 2002) : 430\rangle$ est une cellule du cuboïde de base signifiant que 430 ordinateurs de bureau iTwin ont été vendus en Espagne au cours de l'année 2002.

Les bases de données multidimensionnelles satisfont donc les besoins des décideurs en fournissant des outils autorisant des requêtes complexes sur des grands ensembles de données.

1.3 Les flots de données

L'évolution matérielle des ordinateurs (espace de stockage et vitesse de(s) processeur(s)) a rendu possible la récupération de données de manière continue (stockage des transactions financières, des appels téléphoniques, des logs réseau, etc.). L'analyse de ces données peut s'avérer très utile dans de nombreux domaines (détection de fraudes, décision marketing, etc.).
Intuitivement, un flot est un ensemble de données potentiellement infini, continu et dont le débit peut être rapide. Cependant, l'analyse de données aussi volumineuses et rapides soulève de nouveaux challenges qu'il faut considérer afin de proposer des méthodes efficaces. Nous considérons ici les contraintes induites par cette dynamicité et détaillons les principales problématiques de recherche engendrées.

Vitesse

Une des principales caractéristiques des flots est la vitesse à laquelle les données y circulent. Cette spécificité oblige les algorithmes portant sur les flots à respecter deux contraintes. Premièrement, le temps de traitement d'une valeur (ou d'un groupe de valeurs) du flot doit être plus rapide que l'intervalle séparant l'arrivée de deux valeurs (ou groupes de valeurs). Cette condition est nécessaire pour ne pas bloquer le flot. De plus, puisque le débit d'un flot peut être élevé, chaque élément du flot ne peut pas être observé plusieurs fois. Ainsi, un traitement *à la volée* (*i.e.* une seule lecture) des valeurs du flot est indispensable.

Mémoire

Un flot étant potentiellement infini, le stocker intégralement impliquerait de disposer d'un espace mémoire illimité. Bien entendu, cela est impossible. Les approches sur les flots doivent donc nécessairement prendre en compte cet aspect et proposer des méthodes compactes pour résumer un flot. Différentes techniques sont utilisées dans la littérature telles que l'échantillonnage, l'agrégation, la construction de synopsis...

Le difficile compromis entre justesse et efficacité

Face à ce volume infini de données et puisqu'il est indispensable de borner l'espace occupé en mémoire par l'historique, il faut trouver le meilleur compromis entre la précision des données stockées et les besoins (en mémoire et en temps de traitement). Ce compromis dépend des applications. Ainsi, une requête sur l'historique d'un flot devra répondre au mieux aux attentes de l'utilisateur tout en lui spécifiant le taux d'erreur/imprécision de la réponse.

1.4 Contributions

Les principales contributions présentées dans ce mémoire sont :

1. Généralement, l'historique des données de faible granularité n'est consulté que sur un passé récent. A partir de ce constat, nous proposons une technique inspirée des *tilted-time windows* pour ne matérialiser, sur chaque niveau de précision, que les périodes pendant lesquelles il est consulté. Nous réduisons alors significativement la quantité d'éléments stockés pour chaque dimension. Nous étendons ensuite cette idée à toutes les dimensions et proposons une structure compacte autorisant une analyse multidimensionnelle et multi-niveaux efficace. Les différentes expérimentations menées sur des jeux de données synthétiques et réels témoignent de l'interêt de notre approche. En effet, même dans des conditions extrêmes, les résultats obtenus montrent que notre structure est adaptée aux flots de données.
2. Nous exploitons les habitudes des utilisateurs (à travers des logs de requêtes) pour automatiser la construction de la structure et borner l'imprécision générée.
3. Enfin, une méthode pour interroger la structure est proposée. Les algorithmes développés déterminent (1) la satisfaisabilité d'une requête (*i.e.* si la structure permet une réponse précise) grâce aux métadonnées et (2) proposent plusieurs requêtes alternatives le cas échéant.

1.5 Organisation du mémoire

La suite de ce mémoire est organisée de la manière suivante. Le chapitre 2 effectue un tour d'horizon des approches existantes dans les domaines orthogonaux à ce travail. Nous abordons les techniques de construction de cubes de données dans un environnement statique et étudions les propositions existantes sur les flots de données. Nous dégageons alors la problématique à laquelle nous sommes confrontés. Le chapitre 3 définit formellement les concepts nécessaires à notre proposition. Notre proposition est décrite dans les chapitres 4, 5 et 6. En effet, le chapitre 4 présente la structure proposée. Ensuite, le chapitre 5 propose une automatisation du processus. Enfin, le chapitre 6 présente une solution pour interroger la structure proposée. Nous décrivons nos expérimentations et discutons les résultats obtenus par notre approche dans le chapitre 7. Enfin, nous concluons et évoquons quelques perspectives de recherche à ce travail dans le dernier chapitre de ce mémoire.

Chapitre 2

État de l'art

Notre travail se situe à l'intersection de deux domaines actifs de recherche : les bases de données multidimensionnelles et les flots de données. Nous consacrons donc les deux premières sections de ce chapitre à dresser un panorama des approches existantes dans ces deux domaines. Nous présentons ainsi les définitions et concepts associés en nous appuyant sur une étude de l'existant. Ensuite, puisqu'à notre connaissance une seule approche [HCD+05] aborde notre problématique, nous consacrons la section 2.3 à cette approche, appelée StreamCube. Enfin, dans la dernière section, nous proposons une discussion où nous synthétisons les approches présentées et dressons leurs limites. Cette synthèse permettra alors de définir clairement la problématique de ce travail. Les objectifs généraux de notre approche sont présentés dans la dernière section de ce chapitre.

2.1 Les bases de données multidimensionnelles

La construction efficace des cubes de données est un problème de recherche majeur dans le processus OLAP [GBLP96]. De nombreuses approches ont donc été proposées pour répondre au principal défi d'une telle construction : trouver le meilleur compromis entre la taille du cube matérialisé et le temps de réponse à une requête. En effet, matérialiser tous les cuboïdes du treillis permettrait des réponses immédiates aux requêtes mais est trop coûteux en espace de stockage. A l'inverse, matérialiser uniquement le *cuboïde de base* est avantageux d'un point de vue matérialisation mais engendrerait des temps de réponses innaceptables pour les utilisateurs. Ces deux aspects doivent pourtant être conciliés.

Nous dressons donc ici un panorama des différentes approches existantes en les classant en 3 catégories : les approches basées sur des approximations, celles dites *MOLAP* (*Multidimensional On-Line Analytical Processing*) et les approches *ROLAP* (*Relational On-Line Analytical Processing*).

Les approches basées sur des approximations partent du principe qu'il n'est pas in-

dispensable pour un système de fournir des résultats très précis pour aider à la décision. Cette hypothèse permet donc de réduire l'espace de stockage nécessaire puisque seule une description approximative du cube est stockée. Ces approches utilisent différentes techniques telles que la transformation en ondelettes [VW99, VWI98], les polynômes multivariés [BS97] (le cube est découpé en zones qui sont approximées via un modèle statistique utilisant une régression linéaire), l'échantillonnage [AGP00], les histogrammes [PG99] ou d'autres [GKTD00]. Dans le cadre de ce travail, il est demandé de ne pas s'intéresser à ces techniques [1]. Nous ne détaillons donc pas plus ces approches.

Les approches MOLAP stockent le cube sous la forme de tableaux multidimensionnels. Ainsi, pour chaque mesure du cube, il n'est pas nécessaire de stocker les dimensions associées puisqu'elles sont determinées directement par la position de la mesure dans le tableau. L'inconvénient majeur de ces approches est qu'en pratique les cubes sont peu denses. Par conséquent, beaucoup de cellules du tableau seront vides ce qui dégrade les performances de stockage. Le caractère éparse des cubes a motivé la proposition de Multi-way [ZDN97]. Le cuboïde de base est stocké dans un tableau qui est partitionné pour réduire l'utilisation de la mémoire vive (toutes ses parties ne sont pas nécessaires en même temps). Ce découpage couplé à un plan d'exécution Top-down (le cube est construit en partant du cuboïde de base jusqu'au cuboïde le plus abstrait) permettent la réutilisation des valeurs agrégées pour calculer plusieurs cuboïdes descendants en une seule passe. Par exemple, avec 4 dimensions (A, B, C et D), le calcul de ACD permet également le calcul de AD et A. Les expérimentations menées montrent que cette approche obtient de bons résultats quand le produit des cardinalités des dimensions n'est pas élevé. Dans le cas contraire, les parties de tableau ne tiennent pas en mémoire vive. Cette approche a inspiré MM-Cubing [SHX04] qui propose d'utiliser une technique similaire pour les sous-ensembles denses du jeu de données. En effet, les expérimentations menées sur de nombreuses propositions montrent que le paramètre ayant le plus d'impact sur les performances est la densité du jeu de données. MM-Cubing s'appuie sur cette observation et applique différentes techniques sur le jeu de donnée en fonction de la densité de la région étudiée.

Nous nous intéressons maintenant aux approches ROLAP (*Relational OLAP*) qui stockent et modélisent les bases de données multidimensionnelles à l'aide d'un modèle relationnel. Cette technique a l'avantage de ne pas nécessiter d'investissement financier supplémentaires puisque ce modèle tire parti des ressources logicielles et matérielles existantes. Nous présentons quelques approches existantes.

– BUC (*Bottom Up Computation of Sparse and Iceberg CUBE*) [BR99] construit le cube en débutant par le cuboïde leplus général. Le principe de la construction est le suivant : BUC débute par la première dimension et effectue un partitionnement selon les valeurs de cette dimension. Pour chaque partition, BUC calcule

[1] Ce travail se déroule dans le cadre d'un projet d'ANR MIDAS. Une des tâches allouée à l'équipe TATOO du LIRMM est la construction de cubes de données alimentés par un flot

récursivement les dimensions restantes. Cette approche *Bottom-Up* (le cube est construit du cuboïde de base vers le cuboïde le plus général) permet un élagage similaire à la *Apriori* [AS94] pour le calcul d'*iceberg cubes*. Un *iceberg cube* est un cube où les cellules sont matérialisées si et seulement si elles respectent la *condition de l'iceberg*. Un exemple classique de cette condition est qu'une cellule est matérialisée si elle apparaît au moins x fois dans la base où x est un seuil fixé a priori. Cette condition est proche du support minimum (*minSupp*) dans les règles d'association (*e.g.* une règle est considérée comme fréquente si elle apparaît au moins *minSupp* fois dans le jeu de données). La propriété d'anti-monotonie [AS94] (si une cellule c ne satisfait pas la condition de l'iceberg, alors aucune des cellules plus spécifique que c ne la satisfaira) autorsise un élagage efficace. Cette approche obtient de bons résultats sur des jeux de données peu denses mais les performances se dégradent considérablement quand le jeu de données est plus complexe (comportant des zones denses et d'autres éparses). Nous notons également que cette approche ne prend pas en compte les dimensions hiérarchisées.

– CURE [MI06] est une proposition s'inspirant de BUC puisque le cube est également construit selon un plan d'exécution *Bottom-Up*. Contrairement à BUC, (1) CURE ne propose pas la construction d'*iceberg cube* et (2) grâce à un plan d'exécution judicieux, cette approche prend en compte les dimensions hiérarchisées. En effet, dans la plupart des applications réelles, les dimensions sont observables à plusieurs niveaux de précision. De plus, les décideurs ont besoin de cette vision multi-niveaux pour dégager des tendances, détecter des exceptions, etc. La prise en compte des hiérarchies est donc un atout important. Notons que, tout comme dans [ZDN97], un partitionnement du cube permet de ne stocker en mémoire vive que les parties du cube nécessaires à un moment donné.

– H-Cubing [HPDW01] utilise une structure de hachage arborescente (*H-tree*) pour faciliter la construction et le stockage du cube. Dans cette approche, les dimensions hiérarchisées ne sont pas prises en compte. Cette structure autorise les méthodes *Top-Down* et *Bottom-Up* pour calculer le cube et réduit par nature la redondance des données. Toutefois, comme [BR99], il n'est pas possible d'utiliser les résultats intermédiaires pour le calcul simultané des cuboïdes.

– Pour accélerer la construction du cube, StarCubing [XHLW03] intègre dans une seule proposition les forces des approches *Top-down* (calcul simultané de plusieurs cuboïdes) et *Bottom-up* (élagage à la *Apriori*). Cette approche s'appuie sur une extension de la structure de hachage arborescente proposée dans [HPDW01] appelée Star-Tree. Cette structure permet une compression sans perte du cube grâce à la suppression des nœuds ne satisfaisant pas la *condition de l'iceberg*.

– [PMC05] constate que toutes les approches proposées pour construire le cube s'appuient sur une seule table regroupant tous les n-uplets. En pratique, cette si-

tuation est rare puisque un entrepôt de données est souvent découpé en plusieurs tables. Par conséquent, les approches existantes nécessitent le calcul d'une jointure conséquente qui dégrade les performances. [PMC05] propose alors un algorithme efficace, CTC (*Cross Table Cubing*), pour construire des *iceberg cubes* à partir de plusieurs tables.

2.2 Les flots de données

Pour les raisons évoquées dans le chapitre précèdent, la gestion et l'extraction de connaissances sur des flots sont des thématiques de recherche actives depuis une dizaine d'années. Deux catégories d'approches sont proposées pour les flots de données : les Data Stream Management Systems (DSMS) qui permettent l'interrogation continue des flots et les approches de fouille de données sur les flots dont l'objectif est d'extraire des connaissances sur ces données. Nous présentons donc séparemment ces deux catégories.

2.2.1 Data Stream Management Systems

Les DSMS ont été conçus pour permettre à l'utilisateur de surveiller et d'interroger un ou plusieurs flots de données via des requêtes comme s'il s'agissait d'une base de données traditionnelle. Cependant, dans une base de donnée traditionnelle, les requêtes sont ponctuelles et concernent un état figé de la base alors que dans un DSMS, les requêtes peuvent être continues. La mise en place de mécanismes efficaces de mise à jour est donc obligatoire pour ne pas fausser les réponses. Nous présentons ici les principaux DSMS existants en les regroupant par domaine d'application.

STREAM [MWA+03] est un DSMS généraliste basé sur un modèle relationnel mis au point par l'université de Stanford. Il met l'accent sur la gestion de la mémoire et approxime le résultat des requêtes en fonction des resources matérielles dont il dispose. Le langage d'interrogation utilisé est CQL [ABW06].

GIGASCOPE [CJSS03] a été spécialement conçu pour des applications réseau par AT&T. L'architecture de ce DSMS est distribuée. De plus, certains opérateurs de requêtes sont déplacés dans les routeurs pour accroître les performances. Le système fournit un langage d'interrogation déclaratif appelé GSQL.

Hancock [CFP+04] est également un système développé par AT&T et conçu pour des applications réseaux. Le système fournit un langage procédural basé sur le langage de programmation C. Il a été conçu pour gérer les logs de 100 millions d'appels et détecter des fraudes en temps réel.

Les réseaux de capteurs étant en pleine expansion, certains DSMS ont été conçus pour gérer les flots émis par des capteurs. Aurora [ACc+03] est un DSMS permettant à l'utilisateur d'interroger le flot via une interface graphique. Le plan d'exécution de la requête est donc défini directement par l'utilisateur. Aurora l'optimise ensuite avant de

l'exécuter. TelegraphCQ [CCD+03] est également un DSMS adapté aux réseaux de capteurs optimisant l'exécution de requêtes grâce à une évaluation partagée et un traitement adaptatif des requêtes.

2.2.2 Algorithmes de fouille de données sur les flots

Clustering

Dans un environnement statique, le clustering d'éléments consiste à déterminer, sans connaissances préalables, des groupes d'éléments possédants des caractéristiques communes. Cependant, les approches de clustering classiques sont inadaptées car elles exigent plusieurs passages sur la base de données. Nous présentons ici deux approches adaptées au contexte des flots.

BIRCH [ZRL06] (Balanced Iterative Reducing anc Clustering using Hierarchies) est un algorithme adapté à des données volumineuses. L'approche est basée sur une classification effectuée sur un résumé compact des données au lieu des données originales. BIRCH peut ainsi traiter un grand volume de données en utilisant une mémoire limitée. De plus, il s'agit d'une approche incrémentale qui nécessite une seule passe sur les données.

Jusqu'à CLUSTREAM [AHWY03], les approches existantes ne prenaient pas en compte l'évolution des données. Par conséquent, les clusters construits devenaient mauvais quand la distribution des données évoluait de façon importante au cours du temps. CLUSTREAM résoud ce problème et autorise l'exploration des clusters sur différentes parties du flot. Cet algorithme fonctionne en deux étapes : la première dite *en ligne* stocke périodiquement des statistiques de résumés détaillés (micro-classes). Tout nouvel élément du flot est intégré à la micro-classe la plus proche. Si aucune n'est suffisamment proche de cet élément, une nouvelle micro-classe est créee et deux autres sont fusionnées garantissant une utilisation de la mémoire fixe. La seconde étape, dite *hors ligne*, utilise les informations statistiques à des fins d'analyse.

Classification

Dans un environnement statique, la classification permet d'affecter une classe prédéterminée à des éléments. Contrairement au clustering, une phase d'apprentissage est nécessaire pour déterminer les caractéristiques de chaque classe. Par exemple, la classification de textes en *article scientifique* ou *recette de cuisine* demande l'apprentissage des caractéristiques de chaque catégories grâce à un ensemble de textes dont les catégories sont connues. Cet ensemble de textes déjà étiqueté est appelé *jeu d'apprentissage*. Les textes à classer sont ensuite comparés aux caractéristiques des catégories afin de prédire la classe.

La plupart des propositions de classification sur les flots de données ne répondent pas simultanément aux trois principales contraintes induites par les flots (vitesse, mémoire limitée et évolution de la distribution des données). Nous choisissons de présenter uniquement On-Demand Classification [AHWY04] qui réalise un tel compromis.

On-Demand Classification [AHWY04] reprend l'idée de micro-classes introduites dans CLUSTREAM. Cette approche se divise en deux parties. La première stocke de façon continue des statistiques résumées. Ces résumés sont sous la forme de micro-classes labélisées (chaque micro-classe est étiquetée). La seconde exploite ces statistiques pour effectuer la classification. Le terme *On-Demand* est du au fait que les deux étapes sont réalisables en ligne.

Découverte de motifs fréquents

La problématique de l'extraction de motifs a plus d'une dizaine d'année (*i.e.* les premiers travaux proposés datent des années 90 et concernent l'extraction d'itemsets) et pendant de nombreuses années les principales approches se sont intéressées à des bases de données statiques. Le développement des applications autour des flots de données a nécessité de revoir considérablement les approches précédentes dans la mesure où ces dernières nécessitaient d'avoir accès à l'intégralité de la base de données. L'extraction de motifs dans des flots permet d'offrir de nouveaux types de résumés qui sont particulièrement adaptés à des applications de supervision où l'on recherche par exemple quels sont les éléments qui apparaissent le plus dans le flot. Même s'il existe de très nombreux motifs que l'on peut extraire, l'objectif de cette partie est de se focaliser sur deux principaux : les itemsets (utilisées notamment dans le cas des règles d'association) et les motifs séquentiels.

Li et al. [LLS04] utilisent une extension d'une représentation basée sur un ensemble d'arbres (*Item-Suffix Frequent Itemset Forest*) dans leur algorithme DSM-FI . Cet algorithme propose aussi une approximation de l'ensemble des itemsets fréquents. Ainsi, dans leur approche, les auteurs proposent de relaxer le seuil de fréquence minimal permettant ainsi de mieux garantir la précision de l'extraction. Tous les sous-itemsets extraits sont stockés dans une forêt d'arbres suffixés et de tables d'en-têtes. Les auteurs soulignent que cette représentation par forêt est plus compacte que la représentation par arbre préfixé, malheureusement ce gain en espace mémoire se paye par un plus grand temps de calcul.

Les auteurs de [CL03], quant à eux, proposent de ne travailler que sur la partie récente du flot et introduisent ainsi une approche basée sur le taux d'oubli afin de limiter et de diminuer au fur et à mesure du temps le poids des anciennes transactions. Ainsi l'algorithme permet de se concentrer uniquement sur les parties récentes du flot, mais ce principe d'oubli introduit un trop grand taux d'erreur sur la fréquence et peut s'avérer difficile à gérer dans le cas d'applications critiques.

La recherche de séquences fréquentes (*sequential patterns*) sur des bases de données statiques a donné lieu à un grand nombre de propositions. Cette recherche vise à faire émerger des comportements fréquents en recherchant des corrélations entre attributs et en prenant en compte le temps. Par exemple, $\langle(raquette, tennis)(balles)\rangle$ est une séquence fréquente si plus de $x\%$ (où x est un seuil fixé a priori) des clients achètent une raquette et des tennis puis reviennent au magasin acheter des balles. *raquette* est un *item* et $(raquette, tennis)$ un itemset. Peu d'algorithmes ont été proposés pour l'extraction

de séquences fréquentes sur les flots de données. A notre connaissance trois approches existent dans la littérature. Nous les présentons brièvement :

Dans [MM06], Marascu et al. proposent l'algorithme SMDS (Sequence Mining in Data Streams). L'algorithme SMDS se base sur une approche par segmentation (clustering) de séquences.Des ensembles de séquences similaires sont alors construites au fur et à mesure de l'avancée du flot.

[RPT06] proposent un algorithme d'extraction de motifs séquentiels maximaux sur les flots de données appelé SPEED. L'idée principale de l'approche est de conserver les séquences potentiellement fréquentes (séquences sous-fréquentes). Grâce à une représentation des itemsets au moyen d'un entier et à l'association d'un nombre premier pour chaque item, il est possible d'utiliser des propriétés d'arithmétiques pour les tests d'inclusions de séquences.

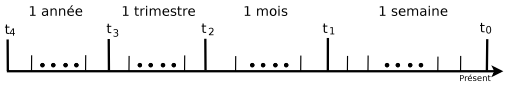
La troisième approche d'extraction de motifs séquentiels est quelque peu différente des deux précédentes puisqu'elle prend en compte l'extraction sur un ensemble de flots de données. Les auteurs Chen et al. [CWZ05] considèrent ainsi que les motifs extraits de cet ensemble de flots de données sont des motifs multidimensionnels. Les auteurs choisissent un modèle de fenêtre glissante pour observer le flot. Leur algorithme, nommé MILE, utilise l'algorithme PrefixSpan [PHMa+01] à chaque étape de glissement afin d'extraire ces motifs.

Dans la section 2.1, nous avons présenté quelques approches pour construire un cube de données. La section 2.2 est consacré aux deux catégories d'approches sur les flots de données : les DSMS et les technique de fouilles de données sur les flots. Nous consacrons la prochaine section à présenter la seule approche supportant la multidimensionnalité des données dans un flot.

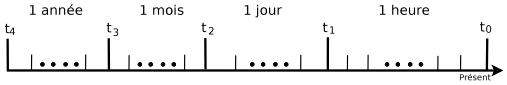
2.3 Gestion de la multidimensionnalité des données dans les flots

A notre connaissance, seule l'approche StreamCube [HCD+05] propose une solution pour construire et mettre à jour un cube alimenté par un flot et permettre ainsi une analyse multidimensionnelle et multi-niveaux. Nous la présentons donc en détail dans cette section.

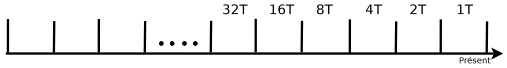
Puisqu'il n'est pas possible de stocker avec un même niveau de précision tout l'historique d'un flot, le modèle de *tilted-time windows* (ou *fenêtres temporelles*) [GHP+02] est utilisé pour modéliser et compresser la dimension temporelle. Ce modèle s'inspire fortement du mécanisme d'oubli de la mémoire humaine et permet de stocker avec une précision maximale les données les plus récentes. En effet, plus les données vieillissent, plus le niveau de précision diminue. La vitesse à laquelle la précision diminue dépend du domaine d'application. La figure 2.1 expose trois exemples de *tilted-time window*. En effet, [GHP+02] propose deux modèles différents. Le premier (2.1(a) et 2.1(b)) est celui des fenêtres temporelles naturelles. Les niveaux de granularité sont fixés en fonction de l'application et suivent un découpage naturel du temps. Par exemple, si l'on considère



(a): *Tilted-time* naturelle 1



(b): *Tilted-time* naturelle 2



(c): *Tilted-time* logarithmique

FIG. 2.1: Exemples de *tilted-time*

les ventes d'un magasin, la figure 2.1(a) stockera pendant une semaine les ventes par la journée. Ensuite, pendant un mois, les ventes seront stockées à la semaine et ainsi de suite. Ce modèle permet une réduction non négligeable du nombre de valeurs stockées. Si l'on matérialisait toutes les ventes à la journée pendant un an, il faudrait stocker 365 valeurs. Le modèle des tilted-time illustré dans la figure 2.1(a) stockera seulement 18 valeurs (7+4+3+4). Le second modèle de tilted-time (2.1(c)) effectue un découpage logarithmique. Dans le cas d'un log en base 2 (comme sur la figure), pour représenter 365 valeurs il faudrait seulement $log_2(365)$ valeurs.

Formellement, nous notons $T = \langle t_0, t_1, ..., t_k \rangle$ une *tilted-time window* de taille k avec t_0 pour désigner le présent. Chaque intervalle $[t_i, t_{i+1}]$ (avec $0 \leq i < k$) représente un niveau de précision et est appelé *fenêtre*. Un tel modèle est donc tout à fait adapté à un contexte d'aide à la décision dans la mesure où le décideur est généralement intéressé par l'analyse des faits récents avec une grande précision mais veut malgré tout conserver une trace des données historiques pour, par exemple, évaluer les tendances.

Pourtant, même si un tel modèle permet une compression importante de la dimension temporelle, il reste impossible de matérialiser les données sur toutes les dimensions et à tous les niveaux de précision, *i.e.* matérialiser l'intégralité du cube. Pour répondre à cette problématique, [HCD+05] propose une matérialisation partielle du cube en fixant deux cuboïdes particuliers appelés *critical layers* : le *minimal observation layer* (*m-layer*) et le *observation layer* (*o-layer*). Le choix du *m-layer* se justifie par l'intuition que les données du flot arrivent à un niveau de précision beaucoup trop fin pour être exploitées en l'état par les utilisateurs et il est donc inutile de matérialiser ces données brutes. Il correspond donc au niveau de précision maximal matérialisé par l'utilisateur pour observer le flot. Bien entendu, le choix de la granularité entraîne l'impossibilité de consulter l'historique du flot à un niveau plus fin que celui défini par le *m-layer*. Le *o-layer*, par contre, est défini comme le niveau courant d'analyse pour l'utilisateur et représente le cuboïde le plus souvent consulté permettant au décideur de détecter les tendances globales et les exceptions. De manière à pouvoir répondre aux requêtes OLAP avec suffisamment de précision et des temps de réponse acceptables, outre la détermination des deux *critical*

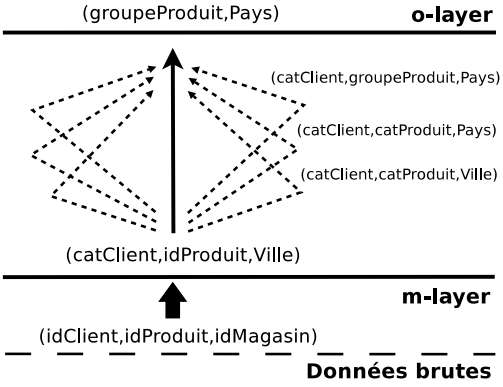


FIG. 2.2: Les différents niveaux de précision de [HCD+05]

layers, de nombreuses possibilités existent pour matérialiser ou pas les cuboïdes situés entre ces deux niveaux. [HCD+05] utilise pour cela le *popular path*. Il s'agit en fait d'un ensemble de cuboïdes formant un chemin dans le treillis reliant le *o-layer* au *m-layer*. Cet ensemble est déterminé à partir de requêtes habituellement formulées afin de minimiser les temps de réponses en réduisant les calculs nécessaires.

De manière à illustrer ces différents niveaux, considérons des données circulant sur le flot définies au niveau de précision *(idClient,idProduit,idMagasin)* (C.f. Figure 2.2).

Ce niveau étant jugé trop précis par l'utilisateur, il fixe alors *(catClient,idProduit,Ville)* comme étant le niveau de précision maximale pour consulter l'historique du flot (niveau *m-layer*). Ensuite, comme la plupart des interrogations concernent le cuboïde *(*,groupeProduit,Pays)*, celui-ci est choisi comme *o-layer*. Enfin les cuboïdes mentionnés entre les deux *critical layers* font partie du *popular path*.

2.4 Motivations et objectifs

2.4.1 Discussion des travaux existants

Nous constatons que de nombreuses approches existent pour construire des cubes de données satisfaisant le compromis espace de stockage/temps de réponse aux requêtes. Les techniques par approximation utilisent des outils mathématiques pour produire une description fidèle et compacte des données réelles mais sortent du cadre de ce travail. Ensuite, les approches *Bottom-Up* et *Top-down* permettent d'optimiser le temps de construction et l'espace occupé par le cube. L'approche *Top-down* facilite le calcul de plusieurs cuboïdes pendant une seule passe alors que *Bottom-up* autorise un élagage a la *Apriori* lors de la construction d'*iceberg cubes*. Nous analysons maintenant la possible adaptation de ces travaux dans un contexte dynamique.

Premièrement, les approches citées ci-dessus pour construire un cube de données évaluent leurs performances sur des jeux de données tenant en mémoire vive. Ce premier

point est une limite majeure puisqu'il rentre en forte opposition avec le caractère infini d'un flot.

Ensuite, dans un flot, les données sont observables à un faible niveau de granularité. La prise en compte des hiérarchies est donc une nécessité. Cependant, parmi les approches proposées, seul CURE [MI06] propose de répondre à ce problème.

De plus, la circulation des données du flot à un faible niveau d'abstraction motive l'utilisation d'une approche *Bottom-up* pour construire le cube. En effet, une stratégie *Top-down* impliquerait, pour chaque valeur du flot, (1) une généralisation pour se positionner sur le cuboïde le plus général et (2) une succession de spécialisation pour atteindre le cuboïde de base. Le débit élevé d'un flot rend cette approche irréalisable.

Notons qu'une stratégie *Bottom-up* n'autorise pas un élagage à la Apriori dans les flots de données. Pour s'en convaincre, imaginons qu'à un instant t une cellule c de mesure m ne satisfasse pas la condition de l'*iceberg*. Par définition, celle-ci n'est donc pas matérialisée. Plus tard, c peut à nouveau apparaître dans le flot avec une mesure m'. Comme m et m' sont nécessaires pour vérifier si la *condition de l'iceberg* est respectée et que m a été précédemment écarté, le calcul correct de la nouvelle mesure est impossible. Le calcul d'*iceberg cube* ne peut donc pas être envisagé en l'état.

Une approche *Bottum-up* stockant tous les cuboïdes comme dans [MI06] n'est pas envisageable car le cube construit serait trop volumineux.

Enfin, une approche MOLAP est inapplicable dans notre situation. En effet, la cardinalité des dimensions n'est pas connue a priori dans un flot. Par conséquent, construire une table multidimensionnelle comme dans [ZDN97] peut s'avérer délicat et entraîner une explosion mémoire. La densité du jeu de données n'est pas non plus connue à l'avance. Une approche telle que [SHX04] qui découpe le jeu de données selon la densité et leur applique des techniques différentes est donc irréalisable.

Malgré de nombreuses propositions pour construire efficacement un cube de données, la transposition de ces approches à un cadre dynamique n'est pas triviale. Plusieurs contraintes doivent être respectées. Premièrement, la matérialisation totale du cube n'est pas envisageable. La construction d'un iceberg cube n'étant pas possible, il faut trouver une alternative réduisant le coût de matérialisation sans utiliser des techniques mathématiques d'approximation. Ensuite, la construction du cube doit suivre une approche *Bottom-up*. La prise en compte des hiérarchies est également une nécessité pour permettre une analyse multi-niveau efficace. Enfin, les techniques de construction de cube de données doivent absolument être testées sur des jeux de données ne tenant pas en mémoire vive pour que l'on puisse déterminer si un passage à l'échelle est possible.

Nous nous intéressons maintenant à l'analyse des techniques existantes sur les flots et nous attardons plus particulièrement sur l'analyse des DSMS et de StreamCube [HCD+05]. Nous ne discutons pas des techniques de fouille sur les flots puisqu'elles sortent du contexte de ce travail.

Les DSMS présentent des caractéristiques intéressantes dans notre contexte. Première-

ment, ces systèmes ont du prendre en compte le caractère infini d'un flot et proposer des méthodes pour borner l'espace de stockage nécessaire. Malheureusement, ces techniques de compression utilisent pour la plupart des techniques mathématique d'approximation plutôt que l'agrégation. De plus, ces sytèmes ont élaboré un langage d'interrogation adapté aux flots. Malgré ces points communs, ces systèmes ne sont pas adaptés à notre contexte pour deux raisons. Premièrement, il n'existe pas de DSMS pour les flots multidimensionnels. Par conséquent, une analyse multi-niveaux (prise en compte des hiérarchies) et multidimensionnelle est impossible. La prise en compte de cette multidimensionnalité est indispensable afin de fournir à l'utilisateur un outil efficace pour analyser un flot. Deuxièmement, l'agrégation doit être préférée à l'approximation pour minimiser la taille du cube.

Avant de dresser un tableau synthétisant les caractéristiques de ces approches nécessaires dans notre contexte, nous critiquons StreamCube [HCD+05]. Cette proposition étant la seule appliquable à notre contexte, elle s'inscrit naturellement comme la référence de notre travail. Même si StreamCube ouvre donc de nouvelles pistes de recherche, cette proposition nous apparaît insuffisante pour deux raisons :

- La propagation systématique des données du *m-layer* jusqu'au *o-layer* en suivant le *popular path* augmente le temps d'insertion d'une valeur dans la structure. Nous pensons que cette étape n'est pas obligatoire dans la mesure où l'opération de généralisation dans les cubes de données permet d'obtenir ces informations à partir du *m-layer* uniquement. En effet, nous considérons qu'il est acceptable pour un décideur de patienter quelques instants (le temps que la généralisation s'effectue) lorsqu'il interroge la structure. A l'inverse, il n'est pas envisageable que le temps de traitement d'une donnée (ou d'un ensemble de données) soit plus lent que le débit du flot. Cette situation entraînerait une perte de données et doit être absolument évitée.
- Malgré une matérialisation partielle du cube, le coût de stockage peut encore être réduit significativement sans perte majeure d'information. En effet, [HCD+05] stocke, pour chaque cuboïde matérialisé, l'intégralité de la *tilted-time window*. Ce choix n'est pas justifié car les décideurs ne consultent pas, pour chaque niveau, l'intégralité de l'historique.

Le tableau 2.1 synthétise et conclut la discussion des travaux existants. Nous évaluons les principales approches discutées selon six critères classés en trois catégories.

La première catégorie concerne la multidimensionnalité des données. Le premier critère est validé si les approches prennent en compte des données multidimensionnelles. Le second évalue si les approches gèrent explicitement les dimensions hiérarchisées. Naturellement, les propositions de la section 2.1 considèrent la multidimensionnalité des données. Par contre, mis à part StreamCube, cet aspect n'est pas géré dans les propositions sur les flots. Concernant la gestion des hiérarchies, ce chapitre exhibe un manque de propositions dans la littérature. En effet, seuls [MI06, HCD+05] les considèrent.

La deuxième catégorie de critère s'intéresse aux particularités des flots de données. Nous évaluons si les approches évoquées gèrent des données dynamiques et si un passage à

Approches	Critères *cubes*		Critères *flots*		Compression	
	Multidimensionnalité	Hiérarchies	Données dynamiques	Passage à l'échelle	Agrégation	Approximation
BUC [BR99]	√				√	
CURE [MI06]	√	√			√	
H-Cubing [HPDW01]	√				√	
StarCubing [XHLW03]	√				√	
DSMS			√	√		√
StreamCube [HCD+05]	√	√	√		√	

TAB. 2.1: Synthèse des approches existantes

l'échelle a été démontré. Les propositions de la section 2.1 ne valident pas ces deux critères car (1) elles ont été concues pour un environnement statique et (2) les expérimentations ont été conduites sur un jeu de données tenant en mémoire vive. Naturellement, le caractère dynamique des données est considéré dans les propositions des sections 2.2 et 2.3. Ensuite, puisque les DSMS mentionnés sont utilisés dans des applications réelles, nous considérons que ces systèmes passent à l'échelle. A l'inverse, pour les raisons déjà évoquées, ce critère n'est pas validé par StreamCube. Les expérimentations décrites au chapitre 7 confirment cette dernière remarque.

Enfin, nous distinguons dans la dernière catégorie la méthode employée pour compresser les données : agrégation ou utilisation d'outils mathématiques pour approximer les données.

2.4.2 Objectifs

Dans le cadre de ce travail, nous proposons une architecture pour résumer un flot multidimensionnel s'inspirant des technologies OLAP. Ce chapitre a mis en évidence plusieurs objectifs à atteindre pour obtenir un résultat satisfaisant. Nous terminons ce chapitre en les définissant clairement.

1. **Prise en compte des hiérarchies** : ce critère est essentiel à une analyse décisionnelle efficace.
2. **Caractère compact de la structure** : dans notre contexte, cet objectif est doublement justifié. En effet, les cubes de données sont par nature bien plus volumineux que le jeu de données. De plus, dans un contexte de flot, ce jeu de données est potentiellement infini. Il est donc indispensable de proposer une approche dont

la taille est fixe ou peu sensible à la taille du jeu de données. Une contrainte liée au contexte de ce stage est que l'approximation des données est à proscrire. En effet, nous nous souhaitons être cohérents avec une approche OLAP et considérons donc des valeurs agrégées.

3. **Traitement rapide des données** : un flot pouvant avoir un débit rapide, la mise à jour de la structure proposée (*i.e.* l'insertion d'une valeur du flot dans la structure) doit être la plus rapide et la plus simple possible. Pour des raisons déjà mentionnées, cette contrainte est essentielle sous peine de laisser passer des valeurs du flot.

Chapitre 3

Problématique

Nous consacrons ce chapitre à décrire la problématique de ce travail. Pour ce faire, nous définissons dans la section 3.1 les concepts préliminaires nécessaires pour bien appréhender le sujet. Dans la section 3.2, nous formalisons la problématique et nous appuyons sur un exemple pour l'illustrer.

3.1 Définitions préliminaires

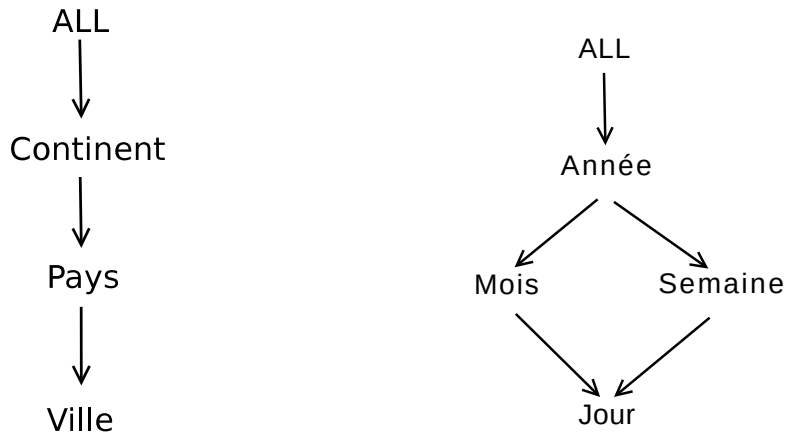
3.1.1 Cubes de données

Un cube de données est une application de n dimensions d'analyse $(D_1, ..., D_n)$ sur un ensemble de mesures $\mathcal{M}$ $(M_1, ..., M_k)$.

$$dom(D_1) \times .. \times dom(D_n) \rightarrow dom(M_1) \times ..dom(M_k)$$

Chaque dimension D_i est définie sur un ensemble (fini ou non) de valeurs noté $dom(D_i)$ et peut être observée à différents niveaux de granularité tels que *Ville*, *Département*, *Région* pour une dimension géographique. Ces niveaux constituent une hiérarchie de précision propre à chaque dimension D_i. Pour chacune d'entre elles, il existe une valeur joker (notée ALL_i ou $*$) qui peut être interprétée comme *toutes les valeurs de* D_i. Nous notons D_i^p le niveau p dans la hiérarchie de D_i (avec $0 \leq p \leq max_i$ et $D_i^0 = ALL_i$ le niveau le plus élevé) et avons $D_i^p > D_i^{p'}$ si D_i^p représente un niveau plus fin que $D_i^{p'}$. Nous notons $x \in dom(D_i^p)$ le fait qu'un élément est défini sur un niveau de hiérarchie p. A l'inverse, $Level(x)$ est utilisé pour spécifier dans quel niveau de hiérarchie x se trouve. Une hiérarchie pour laquelle un niveau se spécialise en un niveau unique est appelée hiérarchie *simple* autrement elle est dite *complexe* (*i.e.* sa représentation forme un DAG (*Directed Acyclic Graph*)).

Exemple 1 *La figure 3.1 présente deux exemples de hiérarchies. La première 3.1(a) illustre une hiérarchie géographique simple où chaque niveau se spécialise en un unique niveau. La figure 3.1(b) est une hiérarchie complexe.*



(a): Hiérarchie simple

(b): Hiérarchie complexe

FIG. 3.1: Exemples de hiérarchies

Un cuboïde est un sous-ensemble de l'application définissant le cube. Formellement, un cuboïde M est défini par :

$$D_1^{l_1} \times D_2^{l_2} \times .. \times D_n^{l_2} \rightarrow M$$

avec $0 \leq l_i \leq max_i$ pour $1 \leq i \leq n$. L'ensemble des cuboïdes constitue un treillis et forme le cube de données.

Exemple 2 *Soient les dimensions Geo(Continent, Pays) et Produit(CatProduit, IdProduit). La figure 3.2 représente le treillis des cuboïdes correspondant.*

Soit $x \in dom(D_i)$ (avec D_i une dimension hiérarchisée). Nous notons $x^{\uparrow}$ l'ensemble des généralisations possibles de x selon la hiérarchie spécifiée. Notons que tout élément est inclus dans sa généralisation (*i.e.* nous avons $ALL_i^{\uparrow} = \{ALL_i\}$).

Exemple 3 *Si l'on reprend la dimension géographique de l'exemple précédent, nous avons :* $France^{\uparrow} = \{France, Europe, ALL\}$.

Pour une dimension D_i et un niveau de granularité p donné, nous proposons la fonction $p^{\sharp}$ qui, pour $x \in Dom(D_i)$, renvoie la généralisation de x au niveau p. Cette fonction est formellement définie ainsi :

$$p^{\sharp}(x) = \begin{cases} \emptyset & \text{si } D_i^p > Level(x) \\ x^{\uparrow} \cap dom(D_i^p) & \text{sinon} \end{cases}$$

Exemple 4 *Sur la même dimension géographique, nous avons :*
- $Continent^{\sharp}(France) = Europe$
- $Pays^{\sharp}(Asie) = \emptyset$.

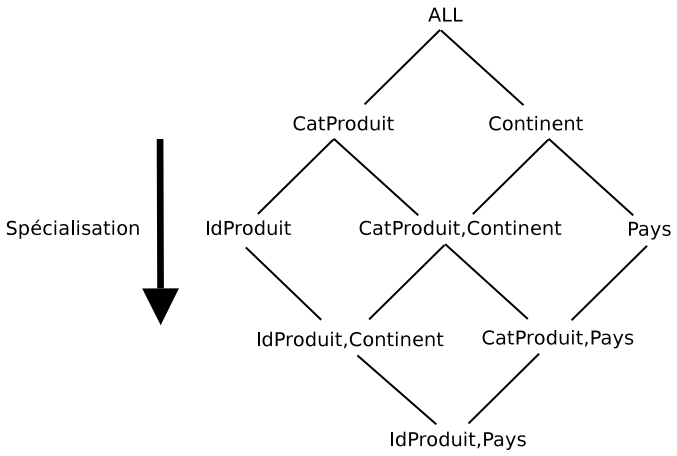


FIG. 3.2: Exemple de treillis

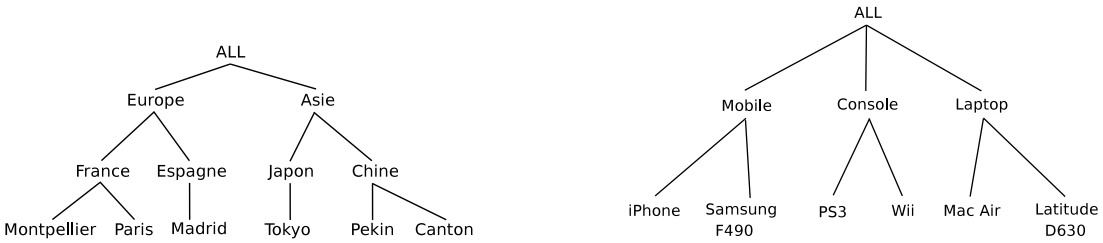
3.1.2 Flots de données

Un flot de données $S = B_0, B_1, ..., B_n$ est une séquence infinie de batchs (ensemble de tuples arrivant dans un même laps de temps) où chaque batch est estampillé avec une date t (B_t). B_n désigne le batch le plus récent.
Un batch $B_i = \{\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3, ..., \mathcal{T}_k\}$ est défini comme un ensemble de n-uplets multidimensionnels arrivant dans le flot pendant les $i^{ièmes}$ unités de temps. Dans un flot, la taille de chaque batch (*i.e.* le nombre de *n-uplets* les composant) peut être variable en fonction du mode de découpage choisi. Soit le découpage est effectué à intervalle de temps fixe auquel cas les batchs auront une taille différente si le débit du flot est variable, soit le flot est découpé selon un nombre de *n-uplets* fixe même si le débit du flot varie. Nous supposons que tous les batchs sont définis sur le même ensemble de dimensions $\mathcal{D}$.

3.2 Problématique

Soit $\mathcal{F}$ une séquence infinie de batchs $B_0, B_1, ..., B_m$ avec B_m le batch le pus récent. Un batch $B_i = \{\mathcal{T}_1, ..., \mathcal{T}_{l_i}\}$ est un ensemble de n-uplets où chaque n-uplet est défini sur un ensemble $\mathcal{D} = \{D_1, ..., D_n\}$ de dimensions d'analyse et sur un ensemble $\mathcal{M} = \{M_1, ..., M_k\}$ de mesures. Une dimension $D_i \in \mathcal{D}$ est organisée en une hiérarchie $\mathcal{H}_i$ et nous notons max_i le niveau le plus précis de cette hiérarchie. Un n-uplet est alors de la forme $\langle (d_1, ..., d_n) : m_1, ..., m_k \rangle$ où $d_i \in Dom(D_i^{max_i})$. Puisque $\mathcal{F}$ est infini, il est impossible de matérialiser l'ensemble des batchs pour conserver l'historique détaillé du flot. La problématique étudiée consiste à trouver une méthode de résumé de flot de données qui (1) minimise l'espace occupé par la matérialisation et (2) maximise la qualité des réponses aux requêtes multi-niveaux et historiques.

Afin d'illustrer les définitions précédentes et notre problématique, considérons le flot



(a): Localisation (b): Produit

FIG. 3.3: Hiérarchies des dimensions de l'exemple

des ventes journalières d'une chaîne de magasins spécialisés en appareils multimédias. Le tableau (a) de la figure 3.5 présente un extrait d'un tel flot. Chaque n-uplet circulant dans ce flot est observable au niveau de précision (*Produit*, *Magasin*). La figure 3.3 présente les différents niveaux de précision possibles pour observer ces ventes et aider les décideurs à les analyser. Conserver l'ensemble des ventes sur tous les niveaux de précision serait idéal mais il faudrait alors stocker tout l'historique du flot sur chacun des 9 cuboïdes du treillis. Le tableau (b) de la figure 3.5 illustre, à partir d'un log de requêtes d'utilisateurs, les fréquences d'interrogation de chaque niveau de précision en fonction de l'historique du flot (t_0 correspond à la date courante). Nous observons que les précisions élevées sont majoritairement consultées sur une période récente du flot. Par exemple, le niveau *IdProduit* est consulté par 100% des utilisateurs uniquement sur la dernière période écoulée, *i.e.* dans l'intervalle de temps $[t_0; t_1]$. La problématique étudiée consiste donc à tenir compte de ces informations pour ne matérialiser, pour chaque niveau de précision, que les périodes pendant lesquelles il est consulté. Par exemple, pour garantir une réduction du coût de matérialisation et obtenir des réponses les plus proches possibles aux requêtes des utilisateurs, il ne faut matérialiser que $[t_0; t_1]$ sur le niveau *Ville*.

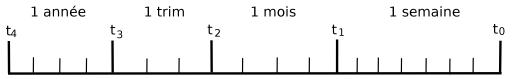


FIG. 3.4: *Tilted-time window* utilisée dans l'exemple

Date	**Produit**	**Magasin**	**Vente**
1	Wii	Montpellier	14
1	iPhone	Pekin	22
2	Mac Air	Tokyo	3
3	PS3	Paris	20
3	F490	Madrid	8
4	iPhone	Tokyo	200
5	D630	Montpellier	8
6	Wii	Pekin	40
6	Mac Air	Madrid	32
7	D630	Paris	10
8	Wii	Canton	5
9	Mac Air	Montpellier	35
10	PS3	Tokyo	100

(a): Extrait du flot multidimensionnel

Niveau	**Intervalle**	**Fréquence**
DIMENSION GEO		
Ville	$[t_0, t_1]$	98%
	$[t_1, t_2]$	2%
Pays	$[t_0, t_1]$	25%
	$[t_1, t_2]$	74%
	$[t_2, t_3]$	1%
Continent	$[t_3, t_4]$	100%
DIMENSION PRODUIT		
IdProduit	$[t_0, t_1]$	100%
CatProduit	$[t_1, t_2]$	70%
	$[t_2, t_3]$	20%
	$[t_3, t_4]$	10%

(b): Fréquence d'interrogation des niveaux de précision

FIG. 3.5: Extrait du flot et logs de requêtes

Chapitre 4

WindowCube, une structure à l'écoute de ses utilisateurs

Maintenant que nous avons présenté le contexte de ce travail, discuté des propositions existantes et défini les concepts utiles pour notre approche, nous définissons la structure proposée : WindowCube. Dans la section 4.1, les fonctions de précision sont introduites. Ensuite, la section 4.2 montre comment combiner ces fonctions. La section 4.3 exhibent les algorithmes développés dans notre approche. Enfin, la dernière section résume les principaux points de cette structure.

4.1 Fonctions de précision

4.1.1 Idée générale

Dans les flots de données, la quantité de mémoire occupée par la structure est une contrainte forte. [HCD+05] l'intègre en compressant l'historique du flot sur la dimension temporelle grâce aux *tilted-time windows* présentées dans la section 2.3. Cette approche nous apparaissant insuffisante, nous proposons alors d'étendre cette compression à toutes les dimensions hiérarchisées.

Pour se convaincre qu'il est inutile de stocker tout l'historique pour chaque niveau de précision, observons le tableau 3.5(b). Nous constatons par exemple que le niveau *IdProduit* n'est consulté que sur le premier intervalle de la *tilted-time window*. Il est donc naturel de se demander pourquoi le reste de l'historique devrait être conservé à ce niveau de précision.

Nous nous appuyons alors sur l'idée que les fenêtres de la *tilted-time window* matérialisés pour chaque niveau de la hiérarchie d'une dimension doivent tenir compte des habitudes des utilisateurs. Par exemple, un cube stockant les ventes par *IdProduit* pour le mois courant et généralisant pour le reste de l'historique sur le niveau supérieur (*CatProduit*) minimiserait l'espace de stockage tout en garantissant des réponses précises aux requêtes courantes. Effectivement, le nombre de catégories est plus petit que le nombre de

produits. La matérialisation proposée réduirait donc le nombre total de fenêtres stockées. Cet exemple illustre à la fois les avantages en terme d'espace d'un tel mécanisme mais aussi l'importance de définir correctement le moment où un niveau de précision ne sera plus stocké. Ces choix doivent nécessairement être faits par les utilisateurs du système afin de maximiser les réponses aux requêtes habituellement formulées. Nous réfléchissons aux méthodes pour bien définir ces fonctions dans la suite de ce chapitre et consacrons le chapitre suivant à exposer une solution. Une formalisation de cette intuition est maintenant proposée.

4.1.2 Formalisation

Nous souhaitons définir une valeur fin qui, pour chaque niveau p d'une dimension D_i (D_i^p), indique le délai à partir duquel D_i^p n'est plus (ou presque plus) interrogé. Nous notons fin_i^p cette valeur pour le niveau D_i^p avec $fin^p \in t_1, .., t_k$ et $fin_i^1 = t_k$. En d'autres termes, fin_i^p (fin^p quand il n'y a pas d'ambiguïté sur les dimensions) est une extrémité d'une des fenêtres de la *tilted-time window* et nous fixons, pour chaque dimension D_i, fin_i^1 (*i.e.* le niveau le moins précis excepté ALL_i) égal à t_k (*i.e.* la limite de la tilted-time). Pour chaque dimension D_i, ALL_i n'est pas défini. Nous proposons dans un premier temps, de ne plus matérialiser pour chaque niveau fin_i^p, la partie de la tilted-time correspondante à $[fin_p; t_k]$ afin d'être fidèle à l'intuition décrite plus haut. Cependant, la matérialisation de $[t_0; fin_p]$ pour chaque niveau conduirait à une forme de redondance. En effet, imaginons que $fin^{idProduit} = t_1$ et $fin^{catProduit} = t_4$. L'opérateur de généralisation OLAP permet d'obtenir par le calcul les données au niveau $catProduit$ sur l'intervalle $[t_0; t_1]$ sans avoir à les stocker.

Nous introduisons alors $debut^p = fin^{p+1}$ avec $1 \leq p < max_i$ et $debut^{max_i} = t_0$ ($debut^p$ quand il n'y a pas d'ambiguïté sur la dimension) pour limiter cette redondance.

Ces deux bornes permettent de définir, pour chaque niveau de précision de chaque dimension, la fonction *Precision* suivante :

$$Precision(D_i^p) = [debut^p; fin^p]$$

$Precision(D_i^p)$ représente alors l'intervalle minimal en terme de stockage permettant de répondre aux requêtes des utilisateurs. Nous remarquons que $\{Precision(D_i^p), 0 \leq p \leq max_i\}$ est une partition de T pour une dimension donnée D_i.

Exemple 5 *Les fonctions de précision des dimensions utilisées dans l'exemple sont présentées dans la figure 4.1 . Par exemple,* $Precision(Pays) = [t_1; t_3]$ *signifie que (1) si l'on interroge ce niveau sur le dernier mois ou dernier trimestre, la réponse est instantanée car le résultat est déjà calculé et stocké ; (2) il ne sera pas possible de connaître le niveau des ventes par ville sur cette période. En effet, connaissant les ventes en France du dernier mois, il n'est pas possible d'en déduire les ventes sur* Montpellier *et* Paris. *Connaissant la distribution des ventes (e.g. que l'on vend deux fois plus de consoles à Paris etc..), il serait possible d'approcher les résultats mais sans garantir leur exactitude. (3) A l'inverse, il est tout à fait possible de connaître la quantité exacte des ventes*

sur cette période au niveau des continents. Ce résultat s'obtient sans difficulté grâce à l'opérateur de généralisation défini dans les cubes de données.

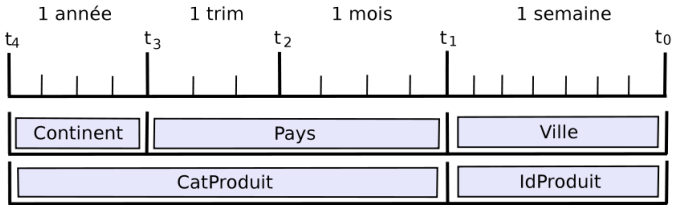


FIG. 4.1: Représentation graphique des précisions

Nous définissons la fonction inverse de $Precision$ qui, pour un niveau $[t_j; t_{j+1}]$ de la *tilted-time window* et une dimension D_i renvoie le niveau associé dans la hiérarchie et la notons :

$$Precision_i^{-1}([t_j; t_{j+1}]) = X$$

avec $Precision(X) = [t_k; t_l]$ et $k \leq j < l$.

Si l'on reprend les données de l'exemple précédent, nous avons $Precision_{Geo}^{-1}([t_0; t_1]) = Ville$.

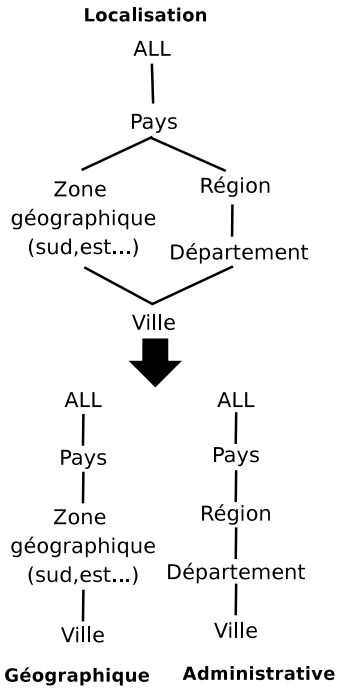


FIG. 4.2: Découpage d'une hiérarchie complexe

Remarque 1 *Jusqu'à présent, nous n'avons utilisé que des hiérarchies simples pour illustrer notre propos. Si des hiérarchies complexes sont définies, une approche naïve serait de les découper et créer ainsi plusieurs hiérarchies simples. La figure 4.2 illustre un tel découpage mais cette solution introduirait de la redondance. En effet, les niveaux* Ville *et* Pays *sont communs aux deux hiérarchies. Les éléments associés seraient donc stockées deux fois. En ce sens, ce découpage n'est pas optimal et la recherche d'une meilleure alternative est une perspective de recherche à court terme.*

4.1.3 Mise à jour

Le modèle proposé étend les *tilted-time windows*. Sa mise à jour en reprend le principe de base. La seule différence est que, pour un niveau D_i^p, lorsque l'intervalle de la *tilted-time window* défini par $Precision(D_i^p)$ est plein (*i.e.* il faut passer à un niveau de granularité plus faible), une double agrégation s'opère. Classiquement, les mesures de la première fenêtre sont agrégées en une valeur v selon la fonction définie. Ensuite, toutes les valeurs v issues de cellules c de même niveau et dont les $(p-1)^{\uparrow}$ sont identiques (*i.e.* les cellules dont la généralisation sur le prochain niveau fournit la même valeur) sont agrégées. Le résultat de cette agrégation est alors stocké dans la première case de l'intervalle de la *tilted-time* de $(p-1)^{\uparrow}(c)$.

Exemple 6 *Supposons que nous ne nous intéressions qu'à la dimension geographique de l'exemple. La séquence à insérer dans la* tilted-time window *de la cellule* Montpellier *est* $\langle 14, 0, 0, 0, 8, 0, 0, 0, 35, 0\rangle$*. Cette séquence peut se traduire ainsi : il y a eu 14 ventes dans le premier batch (jour) à Montpellier, aucune le second, ... La fonction d'agrégation utilisée ici est la* somme*. Selon le mécanisme classique des* tilted-time windows *et en supposant toutes les fenêtres initialement vides, les données sont insérées dans la première fenêtre de la* tilted-time window *jusqu'à ce qu'elle soit pleine. Ainsi, après le batch 7, nous nous trouvons dans la situation illustrée dans la partie supérieure de la figure 4.3. Le traitement du batch suivant implique le passage d'une fenêtre à une autre. Dans un premier temps, la première fenêtre de chaque cellule est agrégée en 22 pour Montpellier et 30 pour Paris. La seconde étape consiste à agréger entre-elles les valeurs dont la généralisation sur le niveau* Pays *conduit au même résultat. Comme* $Ville^{\uparrow}(Montpellier) = Ville^{\uparrow}(Paris) = France$*, 22 est ajouté à 30. Enfin, le résultat est placé dans la première case de la* tilted-time window *associée à* $France$ *et les valeurs du batch 8 (0 et 0) sont insérées dans les* tilted-time windows *correspondantes. Le résultat produit est illustré dans la partie inférieure de la figure 4.3.*

Cet exemple illustre la nécessité de bien déterminer les fonctions de précision pour chaque dimension. Nous consacrons la prochaine sous-section à la description des méthodes adoptées pour leur définition.

4.1.4 Caractérisation des fonctions

Une bonne définition des fonctions de précision est cruciale pour le modèle afin de couvrir le plus de requêtes possibles. Ces fonctions doivent donc obligatoirement être

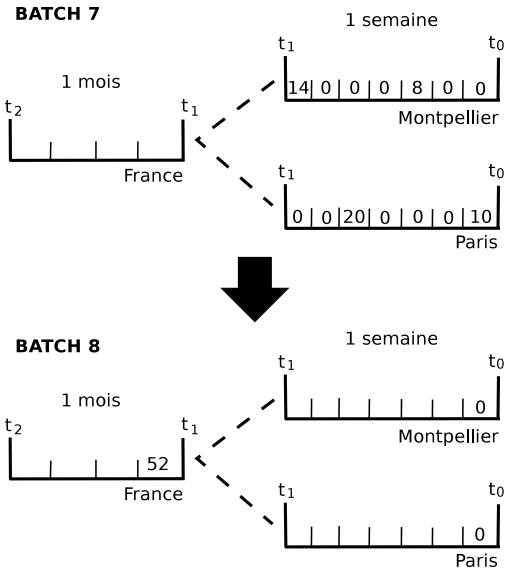


FIG. 4.3: Illustration d'un changement de fenêtre

déterminées par rapport aux préférences de l'utilisateur selon le domaine d'application et en tenant compte de ses requêtes habituelles. Pour ce faire, deux stratégies peuvent être adoptées :

1. Une stratégie *naïve* consisterait à laisser l'utilisateur déterminer manuellement ses préférences (e.g. à l'aide un formulaire générant un fichier XML). Cette solution s'avère cependant peu réaliste puisqu'il est rare qu'il n'y ait qu'un seul utilisateur et que par conséquent les préférences de l'utilisateur ayant renseigné les fonction de précision peuvent être inadaptées pour les autres utilisateurs.
2. Une solution *avancée* consisterait à définir automatiquement les fonctions de précision en utilisant des logs de requêtes et les statistiques sur les niveaux de précision interrogés en fonction du temps. Cette solution est développée au chapitre 5 de ce mémoire.

4.1.5 Considérations sur l'espace de stockage

Nous quantifions ici le gain en terme d'espace de stokage engendré par notre proposition par rapport à [HCD+05]. Pour ce faire, nous utilisons la dimension *Geographique* et la *tilted-time window* présentées dans l'exemple et étudions le nombre d'éléments matérialisés selon que l'on utilise une des deux fonctions de précision définies ainsi :

1. Precision(Ville)=$[t_0; t_4]$, tous les intervalles sont stockés sur le niveau le plus fin. Nous notons cette configuration $prec_1$
2. Precision(Continent)=$[t_0; t_4]$, tous les intervalles sont stockés sur le niveau le moins précis. Nous notons cette configuration, $prec_2$.

Ces deux configurations permettent de borner le nombre d'éléments matérialisés. Pour simplifier notre propos, l'hypothèse est faite que chaque intervalle de *tilted-time window* matérialisé vaut 1 unité.

Puisque $\mid Dom_{Ville} \mid = 6$, le coût de $prec_1$ est 24 (6×4). Pour $prec_2$, $\mid Dom_{Continent} \mid = 2$. Par conséquent, son coût de matérialisation est 8.

Calculons maintenant le coût de l'approche StreamCube sur cet exemple. Nous considérons ici que $Ville$ appartient au *m-layer*, $Continent$ au *o-layer* et $Pays$ au *popular path*. Avec ces hypothèses et puisque [HCD+05] propose de matérialiser l'intégralité de la *tilted-time window* sur les niveaux énoncés, le coût de la matérialisation serait de 48. Nous remarquons ainsi que dans la pire des situations ($prec_1$), notre approche divise par 2 le nombre d'éléments matérialisés comparé à StreamCube. L'objectif ici n'est pas de montrer que nous avons tout intêret à matérialiser l'ensemble de l'historique sur le niveau le plus fin (ce choix doit être guidé par l'utilisateur) mais plutôt d'affirmer que, quelles que soient les fonctions de précision utilisées, notre approche réduit significativement l'espace nécessaire pour stocker l'historique du flot.

4.2 Extension à toutes les dimensions

Dans la section précédente, nous avons défini les *fonctions de précision* qui réduisent significativement le nombre d'éléments stockés pour chaque dimension tout en satisfaisant les besoins des utilisateurs. Nous nous intéressons maintenant à la combinaison de ces fonctions. Nous proposons une méthode utilisant l'opérateur d'intersection ensembliste.

Soit $C = D_1^{p_1} D_2^{p_2} .. D_k^{p_k}$ un cuboïde (avec $1 \leq k \leq n$ et $0 \leq p_i \leq P_{max_i}$ pour $1 \leq i \leq k$). La fonction $Materialize(C)$ est définie ainsi :

$$Materialize(C) = \begin{cases} \emptyset, & \text{si } k \neq n \\ \bigcap_{i=1}^{k} Precision(D_i^{p_i}), & \text{sinon} \end{cases}$$

Un cuboïde C sera matérialisé si et seulement si $Materialize(C) \neq \emptyset$. Chaque mesure de ce cuboïde sera alors une partie d'une *tilted-time window* définie par $Materialize(C)$.

Propriété 1 *Il n'existe pas de cuboïde* $C = D_1^{p_1} D_2^{p_2} .. D_k^{p_k}$ *tel que* $Materialize(C) \nsubseteq Precision(D_i^{p_i})$ *pour tout* $D_i^{p_i} \in \{D_1^{p_1}, D_2^{p_2}, .., D_k^{p_k}\}$.

Preuve Cette propriété découle trivialement de la définition de l'intersection.

□

Exemple 7 *La figure 4.4 illustre l'intersection des fonctions de précision des dimensions de l'exemple. Cette opération entraîne la matérialisation de trois cuboïdes :* (Ville,

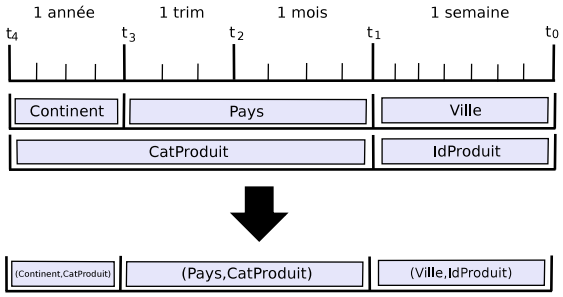


FIG. 4.4: Résultat de l'intersection des fonctions de précision

IdProduit) *où sera stockée la fenêtre* $[t_0; t_1]$*, le cuboïde* (Pays,CatProduit) *contenant l'intervalle* $[t_1; t_3]$ *et* (Continent, CatProduit) *pour* $[t_3; t_4]$*. Nous constatons que la propriété 1 est vérifiée.*

Propriété 2 *Au plus,* $min(\mathcal{H} - 1, \mid T \mid)$ *cuboïdes sont matérialisés par notre approche avec* $\mathcal{H}$ *la hauteur du treillis et* $\mathcal{T}$ *le nombre de fenêtres de la* tilted-time window*.*

Preuve $\mathcal{H}$ est égal à la somme du nombre de niveaux (excepté ALL) plus 1. Puisque par définition aucun intervalle n'est défini sur le niveau ALL, le nombre de cuboïdes matérialisés est borné par $\mathcal{H} - 1$.

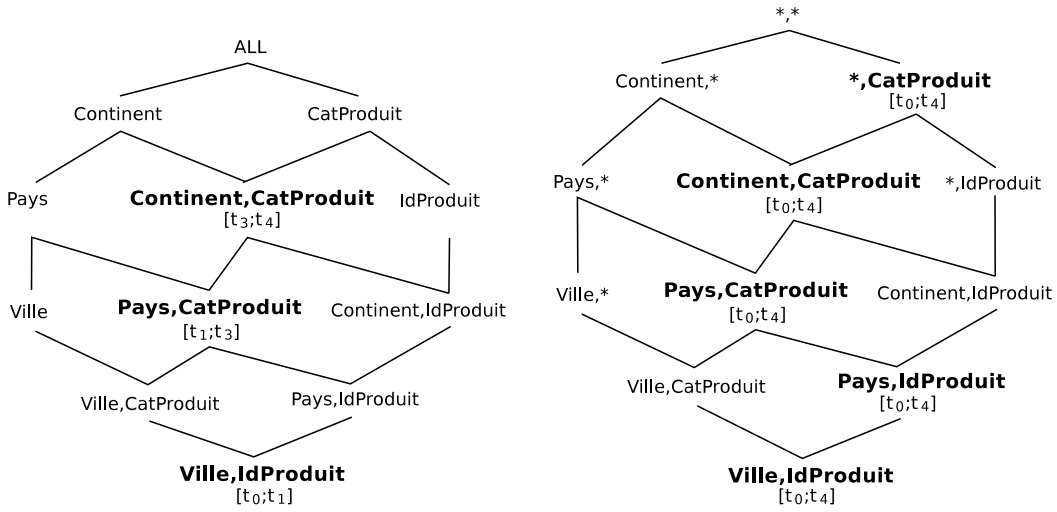
Toutefois, cette borne est inférieure à $\mathcal{H}-1$ dans certaines situations. En effet, chaque intervalle de la *tilted-time window* n'est matérialisé que sur un cuboïde du treillis. Par conséquent, la pire des situations serait de matérialiser un intervalle par cuboïde. Une seconde borne est donc $\mathcal{T}$.

□

Exemple 8 *La figure 4.5 compare le nombre de cuboïdes matérialisés entre WindowCube (figure 4.5(a)) et StreamCube (4.5(b)). Nous supposons que le* popular path *utilisé est celui indiqué en gras. Deux remarques peuvent être faites. Premièrement, le nombre de cuboïdes matérialisés par notre approche est plus faible que pour StreamCube. Ensuite, l'intervalle de temps conservé sur chaque cuboïde matérialisé est indiqué sous le nœud du treillis. Nous constatons que notre approche conserve bien moins d'éléments que StreamCube. Cette remarque est étayé dans le chapitre 7.*

4.3 Algorithmes

Nous présentons ici les algorithmes développés pour cette proposition. Dans un premier temps, nous décrivons le fonctionnement général de la méthode et nous intéressons ensuite à une étape importante : l'insertion d'une cellule dans la structure.



(a): WindowCube (b): StreamCube

FIG. 4.5: Comparaison de la matérialisation

L'algorithme général contient deux phases distinctes : l'initialisation et le traitement. La première consiste à (1) récupérer ou calculer les fonctions de précisions pour chaque dimension et (2) les combiner. Une fois la structure initialisée, il ne reste plus qu'à traiter les n-uplets arrivant dans le flot sous forme de batch. Il s'agit de l'étape d'insertion et de mise à jour de la structure. Dans un contexte aussi dynamique que celui de notre proposition, il est indispensable que cette étape soit la plus rapide possible.

L'algorithme 1 décrit le fonctionnement général de la méthode et détaille les fonctions appelées pour la phase d'initialisation :

- *precision()* permet de calculer les fonctions de précision présentées dans ce chapitre,
- *intersection()* calcule les cuboïdes à matérialiser et les intervalles de *tilted-time* qu'ils stockeront. Comme expliqué dans la partie 4.2, la fonction effectue l'intersection des fonctions de précision et stocke les résultats dans une structure globale.

Maintenant que nous avons décrit le fonctionnement général de la méthode nous présentons l'algorithme d'insertion d'une mesure dans la structure.

Le mécanisme d'insertion d'une mesure dans le cube est très similaire à celui décrit dans la section 4.1.3 pour une dimension. L'algorithme 2 formalise les principales étapes. Si la cellule n'existe pas, un constructeur est appelé (ligne 3). Celui-ci a pour rôle de créer la structure associée et d'insérer la mesure. Dans le cas contraire, il faut vérifier si la cellule n'a pas été mise à jour au cours du même batch (le même n-uplet apparaît plusieurs fois dans le batch). Dans ce cas, il faut simplement ajouter *mes* au début de l'intervalle correspondant (ligne 6). Sinon, une insertion *classique* s'opère. Si l'intervalle n'est pas plein (ligne 8), cela signifie que l'on peut insérer la mesure sans avoir à changer de fenêtre. L'opération consiste alors à insérer *mes* au début de l'intervalle (entraînant un décalage des autres mesures de l'intervalle). Enfin, s'il faut changer de fenêtre, deux

Algorithme 1 : Général

Données : $nbBatchs$ le nombre de Batchs à générer, $sizeOfBatch$ le nombre de tuples dans un batch

```
1 début
    /* INITIALISATION                                          */
2   precision() ;
3   intersection() ;
4   pour i ∈ {0, 1, ..., nbBatchs − 1} faire
5     pour chaque cell ∈ BATCHi faire
6       insèreValeur(cell,mes,1,i);
7     update(i);
8 fin
```

cas apparaissent :

- Si l'intervalle suivant est stocké dans la même cellule (ligne 12), alors il y a agrégation de l'intervalle courant selon la fonction choisie (ligne 13), insertion de la mesure au début de l'intervalle et appel récursif de insèreValeur avec les paramètres de la ligne 16
- Sinon, on suit la même démarche à la différence que l'appel récursif ne se fait pas sur *cell* mais sur la généralisation de *cell* stockant l'intervalle supérieur (ligne 20)

Remarque 2 *Si l'insertion d'une valeur ne nécessite pas un décalage, cinq opérations élémentaires sont nécessaires. Dans le cas contraire, la pire des situations serait de propager les valeurs autant de fois qu'il y a de fenêtres dans la* tilted-time window*. La complexité temporelle de l'algorithme 2 est donc $O(\mid T \mid)$ où T est le nombre d'intervalles de la* tilted-time window*.*

4.4 Conclusion

Dans ce chapitre, nous proposons WindowCube, une structure de données compacte permettant une interrogation multi-niveaux de l'historique du flot. En effet, les fonctions de précision permettent (1) de ne pas conserver les fenêtres de la *tilted-time window* qui sont rarement consultés et (2) de minimiser la redondance en ne matérialisant pas ce qui peut être obtenu par généralisation. La combinaison de ces fonctions grâce à une intersection ensembliste forme une structure de données compacte aux propriétés intéressantes. Enfin, l'algorithme proposé pour insérer une valeur est de complexité linéaire.

Algorithme 2 : insèreValeur

Données : *cell* l'identifiant de la cellule où insérer la mesure, *mes* la mesure à insérer, *window* l'intervalle de la *tilted-time window* où il faut insérer *mes*, *batch* le numéro du batch

```
1  début
2      si (∄cell) alors
3          newCellule(cell,mes,batch) ;
4      sinon
5          si (LastMAJ(cell) = batch) alors
6              cell.window[0]+=mes ;
7          sinon
8              si (!full(cell.window)) alors
9                  unshift(cell.intervalle,mes);
10                 LastMAJ(cell) = numBatch ;
11             sinon
12                 si here(cell,window + 1) alors
13                     agr=agregation(cell.window);
14                     cell.window[0]=mes ;
15                     LastMAJ(cell) = batch ;
16                     insereValeur(cell,agr,window + 1,batch) ;
17                 else
18                     agr=agregation(cell.window);
19                     cell.window[0]=mes ;
20                     cellUp=up(cell,window + 1);
21                     LastMAJ(cell) = batch ;
22                     insereValeur(cellUp,agr,window + 1,batch) ;
23 fin
```

Chapitre 5

De l'art de bien définir les fonctions de précision

Dans le chapitre précédent, nous proposons de résumer un flot multidimensionnel grâce à une structure exploitant les habitudes des utilisateurs. Pour ce faire, nous introduisons les fonctions de précision et les combinons. Pour chaque niveau d'une dimension, la fonction de précision renvoie l'intervalle de temps permettant (1) de répondre à la plupart des requêtes habituellement formulées par les utilisateurs et (2) de supprimer toute redondance dans la structure (*i.e.* il n'est pas utile de matérialiser des éléments pouvant être obtenus par généralisation). Une bonne définition de ces fonctions de précision est essentielle pour notre approche. Nous consacrons ce chapitre à la description d'une méthode pour automatiser les définitions des fonctions de précision. Dans la première section, nous présentons l'exemple utilisé dans ce chapitre pour illustrer notre méthode. La section 5.2 présente cette approche.

5.1 Exemple

Nous illustrons la proposition de ce chapitre avec l'exemple suivant.

Soit une dimension A avec 3 niveaux de précision A_3, A_2, A_1 et $A_0 = ALL_A$ et une *tilted-time window* $T = (t_1, .., t_4)$ où t_1 désigne le présent. Les logs de requêtes sont regroupés dans le tableau 5.1. Pour mieux comprendre sa sémantique, observons la deuxième ligne : le niveau de précision 1 est interrogé 3 fois sur l'intervalle 2 (*i.e.* l'intervalle $[t_2; t_3]$).

5.2 Solution envisagée

5.2.1 Intuition

L'idée de la proposition est de trouver un compromis entre l'imprécision engendrée par la fonction de précision et ce que l'on va appeler *effort de généralisation*. En effet, dans l'exemple, si une seule requête concerne le niveau 3 à l'intervalle t_4 alors que

Niveau	Intervalle	Nombre d'apparitions dans le log
1	1	1
1	2	3
1	3	2
1	4	3
2	1	0
2	2	2
2	3	4
2	4	1
3	1	3
3	2	2
3	3	2
3	4	1

TAB. 5.1: Log de requêtes

beaucoup interrogent l'intervalle t_4 mais pour le niveau de précision 1, nous sommes confrontés à un choix. D'un côté, la solution consistant à matérialiser l'intervalle t_4 sur le niveau 3 a le mérite de ne pas générer d'imprécision mais entraînera de nombreuses généralisations. A l'inverse, si nous choisissons de matérialiser l'intervalle t_4 sur le niveau 1, il arrivera dans de très rares cas que l'on ne puisse répondre précisemment à une requête mais la réponse sera immédiate la plupart du temps. Nous proposons de résoudre automatiquement ce choix en calculant pour chaque intervalle de la *tilted-time window* le niveau offrant le meilleur compromis.

5.2.2 Fonction de coût

La méthode proposée s'appuie sur une représentation enrichie des logs du tableau tableau 5.1. Cette représentation est présentée dans le tableau 5.2. Pour mieux comprendre sa sémantique, analysons la ligne correspondant au niveau 2 sur l'intervalle 2. La colonne *% satisfiable* indique le pourcentage de requêtes auxquelles on pourrait répondre si on matérialisait l'intervalle 2 au niveau 2. Ici, nous pourrions répondre à $\frac{5}{7}$ requêtes. Nous obtenons facilement la colonne *% perte* $(1-\frac{5}{7})$. La dernière colonne, *Effort de généralisation*, comptabilise le pourcentage de requêtes auxquelles on peut répondre en généralisant les données plus précises. Dans l'exemple, si l'on matérialise l'intervalle 2 au niveau 2, alors, l'effort de généralisation serait $\frac{3}{5}$ puisque 3 requêtes parmi les 5 satisfiables concernent un niveau plus général.

Plus formellement, nous avons donc pour un intervalle t et un niveau n donnés :

- $S_t(n) = \dfrac{s_t(n)}{Q_t}$ avec $s_t(n)$ le nombre de requêtes auxquelles on pourrait répondre (directement ou avec généralisation) si on matérialisait t au niveau n et Q_t le nombre total de requêtes concernant l'intervalle t (Q est le nombre total de requêtes dans le log),

Niveau	**% satisfiable** ($S_t(n)$)	**% perte** ($P_t(n)$)	**Effort de généralisation** ($E_t(n)$)	**Coût** ($cout_t(N)$)
Intervalle 1				
3	$\frac{4}{4}$	$\frac{0}{4}$	$\frac{1}{4}$	$\frac{1}{8}$
2	$\frac{1}{4}$	$\frac{3}{4}$	$\frac{1}{1}$	$\frac{7}{8}$
1	$\frac{1}{4}$	$\frac{3}{4}$	$\frac{0}{1}$	$\frac{3}{8}$
Intervalle 2				
3	$\frac{7}{7}$	$\frac{0}{7}$	$\frac{5}{7}$	$\frac{5}{14}$
2	$\frac{5}{7}$	$\frac{2}{7}$	$\frac{3}{5}$	$\frac{31}{70}$
1	$\frac{3}{7}$	$\frac{4}{7}$	$\frac{0}{3}$	$\frac{4}{14}$
Intervalle 3				
3	$\frac{8}{8}$	$\frac{0}{8}$	$\frac{6}{8}$	$\frac{3}{8}$
2	$\frac{6}{8}$	$\frac{2}{8}$	$\frac{2}{6}$	$\frac{7}{24}$
1	$\frac{2}{8}$	$\frac{6}{8}$	$\frac{0}{2}$	$\frac{3}{8}$
Intervalle 4				
3	$\frac{5}{5}$	$\frac{0}{5}$	$\frac{4}{5}$	$\frac{2}{5}$
2	$\frac{4}{5}$	$\frac{1}{5}$	$\frac{3}{4}$	$\frac{19}{40}$
1	$\frac{3}{5}$	$\frac{2}{5}$	$\frac{0}{3}$	$\frac{1}{5}$

Tab. 5.2: Log enrichi

- $P_t(n) = \dfrac{p_t(n)}{Q_t}$ avec $p_t(n)$ le nombre de requêtes auxquelles on ne pourrait pas répondre si on matérialisait t au niveau n,
- $E_t(n) = \dfrac{g_t(n)}{s_t(n)}$ avec $g_t(n)$ le nombre de requêtes auxquelles on ne pourrait répondre que grâce à une généralisation si on matérialisait t au niveau n.

Intuitivement, pour chaque intervalle t, nous cherchons le niveau qui minimisera à la fois le pourcentage de perte (l'imprécision générée) et l'effort de généralisation. La fonction de coût (dernière colonne du tableau 5.2) est donc définie comme la somme pondérée du taux de perte et de l'effort de généralisation. Formellement nous avons donc :

$$cout_t(N) = \alpha P_t(N) + (1 - \alpha) S_t(N)$$

Le paramètre α permet de favoriser l'une ou l'autre composante de la fonction de

coût. Dans un premier temps, α sera fixé à 0,5 pour que les deux composantes soient d'égale importance. Une étude de la valeur optimale (si elle existe) à donner à ce paramètre peut constituer une perspective de recherche à court terme.

Nous appellons *configuration* une suite de niveaux de hiérarchie $\langle l_1, .., l_{|T|} \rangle$ avec $l_1 =\mid P \mid$ (où $\mid P \mid$ est à la fois le nombre de niveaux dans la hiérarchie et le niveau le plus fin), $\mid T \mid$ la taille de la *tilted-time window* (l'intervalle 1 est le plus récent), $l_i \in \{1, .., \mid P \mid\}$ et $l_{i+1} \leq l_i$. Nous nommons cette contrainte *contrainte d'ordre*.

Le coût total d'une configuration est la somme des coûts de chaque composante. Nous cherchons à déterminer la configuration de coût minimum. Par exemple, si l'on considère la configuration $\langle 3, 3, 2, 2 \rangle$, son coût est donc $\frac{1}{8} + \frac{5}{14} + \frac{7}{24} + \frac{19}{40}$ ($\approx 1,248$).

5.2.3 Transposition du problème sous forme de graphe

La recherche de la configuration de coût minimum peut facilement se ramener à un problème largement étudié en théorie des graphes : la recherche d'un plus court chemin dans un graphe valué. Les nœuds sont étiquetés ainsi : $T2\,L3\,5/14$ signifie simplement que le coût de la matérialisation de l'intervalle 2 au niveau 3 est 5/14. Un arc relie un nœud k à un nœud l si et seulement si $T_l = T_k + 1$ et $L_l \leq L_k$. La figure 5.1 représente l'exemple sous forme d'un graphe. Notons que la figure 5.1 ne représente pas certains nœuds comme $T2\,L1$. Cela est du au fait que nous fixons un paramètre *seuil* d'imprécision maximale. Le bénéfice de ce paramètre est double. Premièrement, il permet de ne pas prendre en compte les solutions générant trop d'imprécision. Deuxièmement, le nombre de nœuds du graphe est réduit. Puisque les algorithmes de recherche de plus court chemin s'appliquent sur des graphes dont les arcs sont valués (et non les nœuds), nous transformons la figure 5.1 en figure 5.2.

5.2.4 Choix de l'algorithme de recherche du plus court chemin

Les algorithmes les plus connus pour la recherche de plus court chemin à source unique sont Dijkstra et Bellman-Ford [CLR]. Le choix entre ces deux méthodes a été naturellement guidé par notre cadre d'application. En effet, le graphe obtenu (figure 5.2) est un graphe sans cycle. S'il en contenait un, cela impliquerait qu'il existe au moins un arc (k, l) tel que $T_k > T_l$. Or ceci est impossible par construction. Cette acyclicité entraîne le choix de l'algorithme de Bellman-Ford pour résoudre le problème du plus court chemin entre le nœud *début* et les autres nœuds du graphe.

Pour optimiser cette recherche, nous allons mettre en place un ordre topologique sur les nœuds de ce graphe.

Définition 1 *Un ordre topologique σ est un ordre tel que pour tout n_1, n_2 si $\sigma(n_1) > \sigma(n_2)$ alors il n'existe pas d'arc de n_1 vers n_2.*

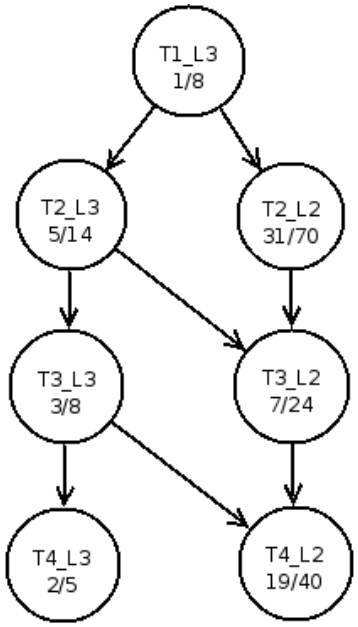


FIG. 5.1: Graphe des configurations

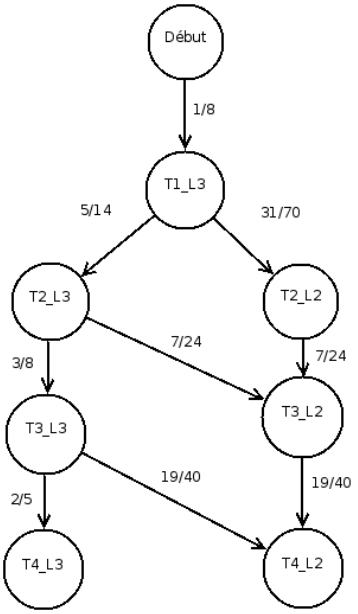


FIG. 5.2: Graphe des configurations modifié

Habituellement, l'algorithme de parcours en profondeur d'un graphe est utilisé pour déterminer cet ordre. Programmer itérativement, cet algorithme est de complexité O(| X |+| A |) où | X | est le nombre de nœuds du graphe et | A | le nombre d'arcs. Dans notre cas, ce calcul est inutile puisqu'il est possible d'affecter un ordre en fonction des étiquettes des nœuds. En effet, le nombre de nœuds du graphe est borné par $borne = | T | \times | L |$. Nous appliquons alors l'algorithme 3 pour déterminer l'ordre topologique.

Propriété 3 *L'algorithme 3 renvoie un ordre topologique σ.*

Preuve Raisonnons par l'absurde. Si σ n'est pas un ordre topologique alors il existe au moins un arc (n_1, n_2) tel que $\sigma(n_1) > \sigma(n_2)$. Si cet arc existe, nous avons par construction $T_{n_2} = T_{n_1} + 1$. Par conséquent, l'algorithme traiterait en premier n_2 et nous aurions $\sigma(n_2) > \sigma(n_1)$.

□

Remarque 3 *Avec un choix de structure judicieux (par exemple une table de hashage de la forme $t \rightarrow$ liste des nœuds d'étiquette T_t), la complexité de cet algorithme sera O(| X |).*

Remarque 4 *Il est peu probable que borne soit le nombre de nœuds réels du graphe. Cela signifirait qu'aucun niveau pour aucun intervalle n'a un taux d'erreur supérieur à* seuil. *La variable debut est alors l'indice du nœud entrée dans le tableau ordreTopo.*

Algorithme 3 : ordreTopologique

Données : G=(X,A) le graphe modifié et borne le nombre de nœuds max
Résultat : ordreTopo[]

```
1 début
2     indice=borne ;
3     pour i =| X | ; i > 0 ; i − − faire
4         pour chaque nœud tel que nœud in X et T_nœud=i faire
5             ordreTopo[indice]=nœud;
6             indice − −;
7     ordreTopo[indice]=entree;
8     début=indice retourner ordreTopo et début;
9 fin
```

Remarque 5 *Le calcul de l'ordre topologique peut être inclu au moment où nous générerons le graphe.*

Maintenant que nous avons déterminé un ordre topologique sur les nœuds de ce graphe, nous appliquons une version améliorée de l'algorithme de Bellman-Ford (algorithme 4) qui permet de mémoriser les chemins les plus court de la source vers les autres nœuds du graphe. De plus, puisque la configuration recherchée est obligatoirement de taille $\mid T \mid$, nous ajoutons une étape supplémentaire à l'algorithme classique qui récupèrera le plus court chemin de taille $\mid T \mid$ (*i.e.* équivaut aux chemins se terminant par un nœud étiqueté $T_{|T|}$) parmi ceux retournés par l'algorithme.

Propriété 4 *L'imprécision générée par la configuration retournée par l'une des deux méthodes est bornée par seuil.*

Preuve Soit $\langle m_1, ..., m_{|T|} \rangle$ la configuration retournée. Par construction, nous avons pour chaque m_i

$$P_i(m_i) \leq seuil\,,$$

donc,

$$p_i(m_i) \leq seuil \times Q_i\,,$$

En sommant les membres de cette inéquation pour tous les intervalles, nous obtenons

$$\sum_{i=1}^{|T|} p_i(m_i) \leq \sum_{i=1}^{|T|} seuil \times Q_i\,,$$

Comme $\sum_{i=1}^{|T|} = Q$, nous obtenons donc

$$\frac{\sum_{i=1}^{|T|} p_i(m_i)}{Q} \leq seuil$$

Algorithme 4 : Bellman-Ford

Données : *entrée* la source unique pour calculer les PCC vers tous les autres sommets
ordreTopo avec *ordreTopo[début]=entrée*

Résultat : *PCC[]* où *PCC[n]* indiquera la valeur du plus court chemin de *entrée* à *n*
Pred[] où *Pred[n]* indiquera le nœud précédeant *n* dans le plus court chemin de *entrée* à **n**

```
1 début
    /* Initialisation                                          */
2   Pred[entrée]=null ;
3   PCC[entrée]=0 ; pour chaque nœud ∈ {X − entree} faire
4     Pred[entrée]=null ;
5     PCC[entrée]=+∞ ;
    /* Corps de l'algorithme                                   */
6   pour i=debut+1 ; i≤ borne ; i++ faire
7     PCC[i]=min(PCC[j] + c_ji) (avec j un prédécesseur de i et c_ji le coût de
      l'arc (ji) );
8     Pred[i] = j (avec j le sommet satisfaisant la ligne au dessus) ;
9 fin
```

□

La figure 5.3 présente le résultat de l'algorithme 4. Après comparaison des coûts des deux chemins de taille 4, le sommet T4_L2 est élu. En remontant grâce à Pred[], nous trouvons donc que $\langle 3, 3, 2, 2\rangle$ est la configuration de poids minimum.

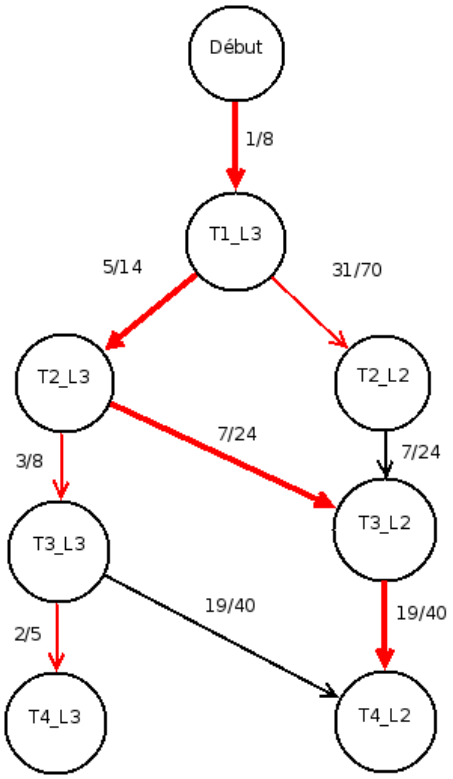


FIG. 5.3: Application de Bellman-Ford sur l'exemple

Chapitre 6

Solution pour interroger WindowCube

Dans le chapitre 4, nous proposons une méthode pour construire un cube de données issues d'un flot multidimensionnel. Afin de minimiser l'espace occupé par la structure, nous introduisons les fonctions de précision et les combinons. Ces fonctions permettent (1) de ne pas conserver les informations très peu interrogées et (2) de ne pas introduire de la redondance dans la strucutre en ne matérialisant pas les élèments calculables par généralisation. Cette compression introduit de l'imprécision. En effet, un niveau de précision p d'une dimension D_i n'est plus consultable aprés fin_i^p. Dans le chapitre 5, nous bornons cette imprécision. Dans ce chapitre, nous proposons une méthode pour interroger la structure qui s'inspire du protocole proposé dans [PJD99].

Dans [PJD99] les données manipulées concernent des diagnostics établis qui peuvent être imprécis par nature. Par exemple, un diagnostic peut être précis (diabétique dépendant à l'insuline) ou plus vague (diabétique) en fonction des symptomes et examens effectués. Dans notre cas, les fonctions de précision sont à l'origine de l'imprécision générée. Néanmoins, nous sommes confrontés aux mêmes problématiques :

1. Comment déterminer, sans parcourir le cube, si une réponse précise peut être donnée à une requête ?
2. Si ce n'est pas le cas (*i.e.* si l'intervalle de temps concerné par la requête n'est plus matérialisé au niveau de précision désiré), quelle stratégie adopter pour malgré tout fournir une réponse pertinente à l'utilisateur ?

Dans la section 6.1, nous proposons une technique pour déterminer la satisfaisabilité d'une requête sans parcourir le treillis des cuboïdes. Le cas échéant, la section 6.2 décrit une technique pour gérer l'imprécsion grâce à la génération de *requêtes alternatives*. Enfin, une conclusion résume les principaux points de ce chapitre.

6.1 Déterminer la satisfaisabilité d'une requête

Une requête est satisfiable si le cube contient les informations nécessaires à une réponse précise. Pour déterminer la satisfaisabilité d'une requête, une méthode naïve serait de chercher dans le cube pour déterminer si la réponse précise à la requête est matérialisée. Parcourir le cube n'est pas une solution optimale et augmenterait les temps de réponses aux requêtes. Nous proposons une solution pour déterminer la satisfaisabilité d'une requête sans parcourir la structure. En effet, grâce aux métadonnées que sont les fonctions $Precision$ et $Precision^{-1}$, nous définissons le protocole suivant :

1. Le critère déterminant pour prédire la satisfaisabilité d'une requête est l'intervalle de l'historique interrogé. En effet, pour un niveau de précision D_i^p, une requête ne sera pas satisfiable si elle concerne un intervalle de l'historique antérieur à fin_i^p. Il convient donc de déterminer dans un premier temps l'intervalle $[t_k; t_l]$ $(0 \leq k < l \leq \mid T \mid)$ concerné par la requête.

2. Afin d'appliquer l'algorithme présenté dans la prochaine étape, la requête est réecrite en utilisant le formalisme suivant :

$$Q' = \langle D_1^{p_1} = val_1, ..., D_n^{p_n} = val_n : [t_k; t_{k+1}], ..., [t_{l-1}; t_l] \rangle$$

3. Cette reformulation effectuée, l'algorithme 5 détermine la satisfaisabilité d'une requête. Son principe est le suivant : pour chaque niveau D_i^p concerné par la requête, si $Precision_i^{-1}$ de l'intervalle $[t_{l-1}; t_l]$ (l'intervalle interrogé le plus éloigné du présent) est moins fin que D_i^p, alors la requête n'est pas satisfiable. En d'autres termes, une requête est satisfiable si et seulement si tous les intervalles interrogés sont matérialisés sur les niveaux concernés ou sur des niveaux plus fins

Exemple 9 *Soit l'exemple décrit dans la section 3.2, les fonctions de précision associées (chapitre 4) et la requête* $\mathcal{Q}_1$ = "Quelles sont les ventes en France et par produit effectuées entre $[t_1; t_2]$*". Nous appliquons la méthodologie proposée.*

- ***Etape 1 et 2*** *:* $\mathcal{Q}_1$ *se réecrit ainsi :* $\langle Pays = France, IdProduit = all : [t_1; t_2] \rangle$
- ***Etape 3*** *: nous appliquons maintenant l'algorithme 5 en débutant par* Pays*. Comme* $Precision^{-1}_{Localisation}([t_1; t_2]) = Pays$*, les données du cubes sont suffisamment précises pour cette parite de la requête. Pour la dimension* Produit*, la requête s'intéresse aux ventes par produit entre* t_1 *et* t_2*. Nous avons* $Precision(IdProduit)$ $= [t_0; t_1]$*. Par conséquent, après* t_1*, les données sont agrégées au niveau CatProduit. Les informations matérialisées dans le cube ne sont donc pas suffisantes pour répondre avec précision à* $\mathcal{Q}_1$*. L'algorithme retourne alors* false*.*

La complexité de cet algorithme est $O(\mid D \mid)$ où $\mid D \mid$ est le nombre de dimensions et aucune interrogation du cube n'est nécessaire. Malgré tout, le fait de pouvoir dire si une requête est satisfiable ou pas n'est pas suffisant. En effet, dans le cas où une requête ne

Algorithme 5 : SatisfRequete

Données : Une requête traduite
$Q' = \langle D_1^{p_1} = val_1, ..., D_n^{p_n} = val_n : [t_k; t_{k+1}], ..., [t_{l-1}; t_l] \rangle$, les fonctions $Precision^{-1}$ pour chaque dimension
Résultat : Satisf $\in \{true, false\}$
début
  Satisf $\leftarrow$ true ;
  `/* Pour chaque dimension` D_i `*/`
  pour chaque $D_i^{p_i} \in \{D_1^{p_1}, ..., D_n^{p_n}\}$ **faire**
    `/* Si la précision demandée est trop fine */`
    si $D_i^{p_i} > Precision_i^{-1}([t_{l-1}; t_l])$ **alors**
      Satisf $\leftarrow$ false ;
      retourner *Satisf*;
  `/* Les données du cubes sont assez précises pour répondre */`
  Satisf $\leftarrow$ true ;
  retourner *Satisf*;
fin

l'est pas, il faut définir une méthode pour *répondre au mieux* à la requête initiale tout en indiquant à l'utilisateur que la réponse sera imprécise. Dans la section suivante, nous développons une méthode pour gérer cette imprécision.

6.2 Gérer l'imprécision

Nous nous inspirons de [PJD99] et proposons aux utilisateurs des requêtes *alternatives* quand une requête n'est pas satisfiable. Nous présentons maintenant le protocole mis en place :

1. Si la requête n'est pas satisfiable, une *requête alternative* est proposée à l'utilisateur. Deux approches différentes peuvent être considérées pour proposer une requête alternative :
   (a) Privilégier le temps à la granularité : si une réponse précise à une requête $\mathcal{Q}$ n'est pas possible, cela signifie qu'au moins un niveau de précision concerné n'est pas conservé suffisamment longtemps. Ainsi, une requête alternative $\mathcal{Q}'$ qui favoriserait le temps au niveau de précision serait une requête telle que (1) l'intervalle de temps de $\mathcal{Q}$ et $\mathcal{Q}'$ serait identique et (2) les niveaux de précision de $\mathcal{Q}'$ serait en adéquation avec l'intervalle de temps concerné par la requête. Par exemple, une telle requête alternative appliquée à $\mathcal{Q}_1$ serait $\mathcal{Q}'_1$ = "*Quelles sont les ventes par catégories de produits effectuées en France entre* t_1 *et* t_2".
   (b) Privilégier les niveaux de granularité au temps : à l'inverse, une solution possible est d'adapter l'intervalle de temps de la requête initiale pour que les ni-

Algorithme 6 : RequêteAlternative

Données : Une requête non satisfiable
$Q = \langle D_1^{p_1} = val_1, ..., D_n^{p_n} = val_n : [t_k; t_{k+1}], ..., [t_{l-1}; t_l] \rangle$, un graphe de précision G_{prec}

Résultat : $Q_{alt} = \langle D_1^{p'_1} = val'_1, ..., D_n^{p'_n} = val'_n : [t_k; t_{k+1}], ..., [t_{l-1}; t_l] \rangle$

début

`/* Pour chaque dimension` D_i `*/`

pour chaque $D_i^{p_i} \in \{D_1^{p_1}, ..., D_n^{p_n}\}$ **faire**

`/* Si la précision demandée est trop fine */`

si $D_i^{p_i} > Precision_i^{-1}([t_{l-1}; t_l])$ **alors**

$D_i^{p'_i} \leftarrow Precision_i^{-1}([t_{l-1}; t_l])$;

$val'_i \leftarrow p_i'^{\sharp}(val_i)$;

sinon

$D_i^{p'_i} \leftarrow D_i^{p_i}$;

$val'_i \leftarrow val_i$;

`/* On compose la requête et on renvoie */`

$Q_{alt} \leftarrow \langle D_1^{p'_1} = val'_1, ..., D_n^{p'_n} = val'_n : [t_k; t_{k+1}], ..., [t_{l-1}; t_l] \rangle$;

retourner Q_{alt};

fin

veaux de granularité concernés soient interrogeables avec précision. Appliquée à $\mathcal{Q}_1$, nous obtiendrons $\mathcal{Q}''_1$ = "*Quelles sont les ventes par produit effectuées en France entre* t_0 *et* t_1".

La première solution est préférée. En effet, selon nous, l'intervalle de temps concerné par une requête est le paramètre le plus important pour un utilisateur. Si la requête concerne les ventes des 6 derniers mois selon un certain niveau p (observable uniquement sur la semaine en cours), il est plus pertinent de fournir les ventes des 6 derniers mois à un niveau moins fin plutôt que les ventes de la semaine au niveau demandé. L'algorithme 6 retourne la requête alternative. Notons que, dans ce mémoire, les algorithmes 5 et 6 sont disjoints pour faciliter la compréhension. Etant très similaires, leur fusion est envisageable.

Si l'utilisateur accepte la requête alternative, alors celle-ci est exécutée.

2. Imaginons maintenant que l'utilisateur refuse cette requête alternative. Là non plus, il n'est pas souhaitable que le système mette un terme à l'échange avec l'utilisateur en indiquant qu'il ne peut rien faire pour lui. Dans une optique coopérative et pour satisfaire au mieux l'utilisateur, nous proposons de retourner l'ensemble des réponses *proches* de sa requête initiale. Nous appelons cette réponse *verbose answer*.

Algorithme 7 : ReponseImprécise

Data : Une requête non satisfiable
$Q = \langle D_1^{p_1} = val_1, ..., D_n^{p_n} = val_n : [t_k; t_{k+1}], ..., [t_{l-1}; t_l] \rangle$, les fonctions $Precision^{-1}$ associées
Result : Ens_Q un ensemble de requêtes répondant au mieux à Q
begin
  `/* Pour chaque intervalle` $[t_j; t_{j+1}]$ `concerné par` Q `*/`
  foreach $[t_j; t_{j+1}] \in \{[t_k; t_{k+1}], ..., [t_{l-1}; t_l]\}$ **do**
    $Q_j \leftarrow$
    $\langle \bigcup_{1 \leq i \leq n} \Big(Precision_i^{-1}([t_j; t_{j+1}]) = \big(Precision_i^{-1}([t_j; t_{j+1}])\big)_i^\sharp (val_i) \Big) : [t_j; t_{j+1}] \rangle$;
    $Ens_Q \leftarrow Ens_Q \cup Q_j$;
  return Ens_Q;
end

L'algorithme 7 présente la méthode pour parvenir à un tel résultat. Son principe est le suivant : pour chaque intervalle concerné par la requête initiale, il faut d'abord trouver le cuboïde stockant cet intervalle puis exécuter une requête intermédiaire concernant cet intervalle et les niveaux de granularité du cuboïde.

Exemple 10 *Considèrons* $\mathcal{Q}_2$ = "Quelles sont les ventes par produits à Montpellier entre t_0 et t_4 ?". *La requête alternative serait* $\mathcal{Q}'_2$ = "Quelles sont les ventes de produits en France entre t_0 et t_3 ?". *En cas de refus, voici ce que le système retournerait à l'utilisateur :*

- *Les ventes par produits à Montpellier entre* t_0 *et* t_1 *car :*
  - *(a)* $Precision_{Produit}^{-1}([t_0; t_1]) = IdProduit$
  - *(b)* $Precision_{Loc}^{-1}([t_0; t_1]) = Ville$
  - *(c)* $(Precision_{Loc}^{-1}([t_0; t_1]))^\sharp(Montpellier) = Montpellier$
- *Les ventes par catégories de produits en France entre* t_1 *et* t_3
- *Les ventes par catégories de produits en Europe entre* t_3 *et* t_4

6.3 Conclusion

Dans ce chapitre, nous proposons une méthode pour gérer l'imprécision engendrée par notre architecture. Dans un premier temps, un algorithme est utilisé pour déterminer si une requête est satisfiable (e.g. si une réponse précise est possible). Le cas échéant, le une requête alternative est proposée à l'utilisateur. Enfin, s'il désire obtenir le maximum d'informations en rapport avec sa requête initiale, une réponse détaillée (*verbose answer*) lui est transmise.

Chapitre 7

Expérimentations

Dans ce chapitre, nous évaluons notre approche en nous comparant à StreamCube [HCD+05] et en réalisant une étude des performances sur des jeux de données synthétiques et réels. Pour ce faire, l'influence de plusieurs paramètres (nombre de dimensions, degré et profondeur des hiérarchies et fonctions de précision) est examinée.

Deux conditions sont indispensables pour valider notre approche :

1. borner la mémoire utilisée. Cette condition est respectée lors de nos expérimentations car la mémoire nécessaire est toujours inférieure à 10M.
2. borner le temps de calcul entre chaque batch. Cette condition est aussi respectée car pour chaque étape, l'implémentation de notre approche ne dépasse pas la taille du batch.

Les jeux de test synthétiques proviennent d'un générateur de batchs de données multidimensionnelles aléatoires satisfaisant certaines contraintes (nombre de dimensions, degré des hierarchies, etc.). Nous pouvons ainsi évaluer précisement notre approche. La convention pour nommer les jeux de données est la suivante : D4L3C5B100T20 signifie qu'il y a 4 dimensions avec 3 niveaux de précision chacune, que les arbres des hiérarchies sont de degré 5 et que les tests ont été réalisés sur 100 batchs de 20K n-uplets. Toutes les expérimentations ont été exécutées sur un Pentium Dual Core 2GHz avec 2Go de mémoire vive sous un Linux 2.6.20. Les méthodes (WindowCube et StreamCube) ont été implémentées en Perl version 5 et utilisent une base de données MySQL. Nous utilisons une fenêtre temporelle logarithmique de taille 8 ($log_2(250) = 7,96...$) au cours de ces expérimentations.

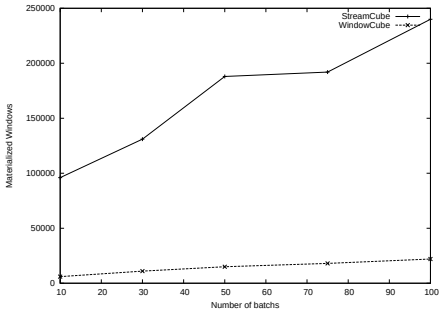
Dans la section 7.1, nous comparons notre approche à StreamCube. En effet, la principale critique faite à cette proposition est sa matérialisation excessive. Nous examinons alors le nombre de fenêtres matérialisées par les deux approches sur des jeux de données synthétiques. Ensuite, la section 7.2 présente et discute les résultats obtenus sur différents jeux de données synthétiques. La mémoire vive occupée et le temps d'insertion d'un batch dans WindowCube démontrent que notre structure est bien adaptée

au contexte des flots. Ces bonnes propriétés établies, nous appliquons notre méthode sur des données issues de sondes réseaux. Les résultats obtenus montrent qu'une telle structure serait avantageuse pour un administrateur réseau.

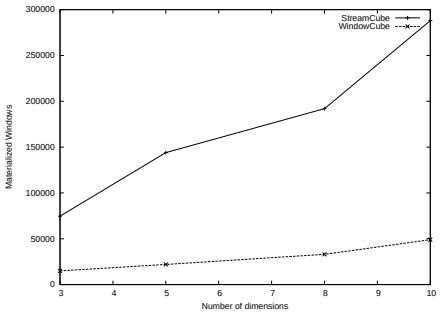
7.1 Comparaison avec StreamCube

Dans la section 2.4.1, nous discutons de l'approche StreamCube. La principale critique faite est sa matérialisation excessive. En effet, conserver tout l'historique du flot sur les cuboïdes du *popular path* entraîne de la redondance et n'est pas pertinent dans la mesure où des parties de l'historique ne sont jamais consultées à certains niveaux de précision.

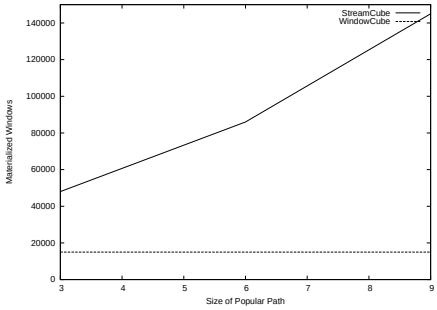
Nous comparons le nombre de fenêtres matérialisées sur des jeux de données synthétiques en reprenant la démarche de la section 4.1.5 (page 31).



(a): Nb de batchs (D5C5L3T1)



(b): Nb de dimensions (C5L3B50T1)



(c): Longueur du popular path (C5D5B50T1)

FIG. 7.1: Comparaison du nombre de fenêtres matérialisées par rapport à StreamCube

La figure 7.1(a) présente le nombre de fenêtres matérialisées en fonction du nombre de batchs. Dans cette série de tests, la longueur du *popular path* et le nombre de cuboïdes matérialisés par WindowCube sont égaux à 6. Le jeu de test utilisé est D5C5L3T1. Nous remarquons alors que le treillis est de hauteur 16. Les résultats obtenus sont intéressants

puisqu'en moyenne, notre approche matérialise 10 fois moins de fenêtres que StreamCube. En outre, l'impact du nombre de batchs sur les fenêtres matérialisées est faible. A l'inverse, le nombre de batchs impacte grandement sur la matérialisation de StreamCube. Ainsi, puisque un flot est une séquence potentiellement infinie de batchs, notre approche semble particulièrement adaptée par rapport à StreamCube.

La figure 7.1(b) présente le nombre de fenêtres matérialisées en fonction du nombre de dimensions d'un n-uplet. De même que dans la précédente série de tests, la longueur du *popular path* et le nombre de cuboïdes matérialisés sont équivalents. Ainsi, pour 10 dimensions, notre approche matérialise 11 cuboïdes. Nous notons que le treillis est de hauteur 31. Le jeu de test utilisé est C5L3B50T1. Les résultats obtenus sont très satisfaisants. En effet, notre approche matérialise entre 4 et 6 fois moins de fenêtres que StreamCube. Là aussi, les courbes présentées suivent un comportement similaire au précédent test. Ainsi, le nombre de dimensions du jeu de données a peu d'influence sur les performances de notre approche. A l'inverse, ce paramètre dégrade considérablement les performances de StreamCube en terme de matérialisation.

Dans les précédents tests, nous fixions la longueur du *popular path* pour qu'elle soit équivalente au nombre de batchs matérialisés par notre approche. Cette considération avait pour objectif de ne pas biaiser les résultats. Cependant, dans le cas où le treillis est de taille 30, un *popular path* de longueur 10 est peut être insuffisant. En effet, il est très possible que celui-ci soit plus long. Ainsi, nous testons le nombre de fenêtres matérialisées en fonction de la longueur du *popular path*. Puisque, pour notre approche, le nombre de cuboïdes matérialisés est dépendant des fonctions de précision et qu'elles restent identiques, le nombre de fenêtres stockées par notre approche reste inchangé. Le jeu de test utilisé est D5C5L3B50T1. Nous notons que notre approche matérialise 6 cuboïdes. La figure 7.1(c) présente les résultats obtenus. Même quand la longueur du popular path est de longueur 3, StreamCube matérialise deux fois plus de fenêtres que notre proposition. Ce rapport passe à 7 quand la longueur du popular path atteint 9.

Les chapitres précédents supposaient que notre approche était moins coûteuse que StreamCube. Dans cette section, nous le démontrons empiriquement. En effet, les résultats obtenus attestent du réel bénéfice de notre proposition. Dans la prochaine section, nous évaluons notre structure sur des jeux de données synthétiques difficiles.

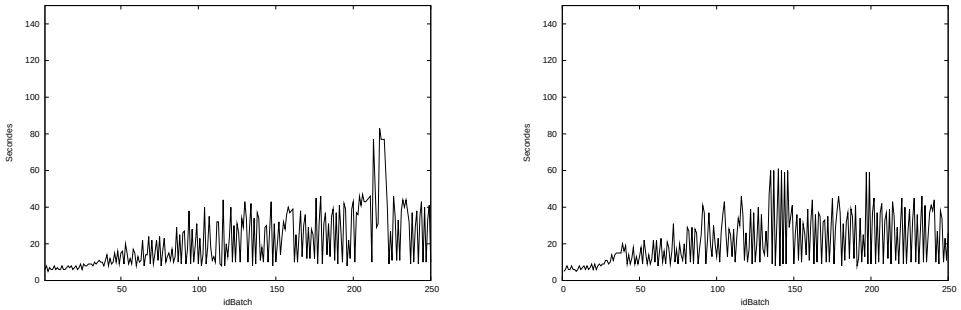
7.2 Résultats sur des jeux de données synthétiques

Dans cette section, nous évaluons notre approche sur des jeux de données synthétiques en faisant varier de nombreuses caractéristiques du flot. Aini, nous déterminons quels sont les paramètres impactant le plus sur notre approche et analysons le comportement de notre structure face à des conditions extrêmes. Dans la section précédente, nous avons vu que le nombre de fenêtres stockées est faible comparé à StreamCube. Nous nous intéressons maintenant à deux autres paramètres critiques dans les flots de

données : la mémoire vive occupée et le temps d'insertion d'un batch dans WindowCube. Les conditions d'évaluation sont identiques à la section précédente.

Intervalles	***Expensive***	***Slim***	***Balanced***
$[t_0, t_1]$	$A_3B_3C_3D_3E_3$	$A_3B_3C_3D_3E_3$	$A_3B_3C_3D_3E_3$
$[t_1, t_2]$			
$[t_2, t_3]$			$A_3B_2C_2D_3E_3$
$[t_3, t_4]$			$A_2B_2C_2D_3E_2$
$[t_4, t_5]$		$A_1B_1C_1D_1E_1$	$A_1B_2C_2D_3E_1$
$[t_5, t_6]$			$A_1B_1C_1D_2E_1$
$[t_6, t_7]$			$A_1B_1C_1D_1E_1$
$[t_7, t_8]$	$A_1B_1C_1D_1E_1$		

TAB. 7.1: Cuboïdes matérialisés



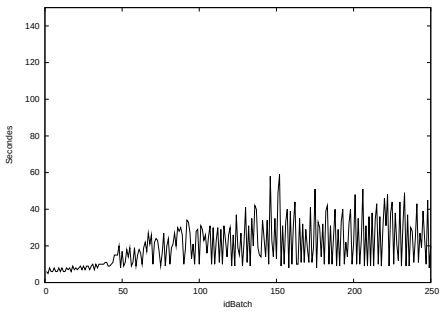
(a): Temps vs *Expensive* (D5L3C5B250T20)

(b): Temps vs *Slim* (D5L3C5B250T20)

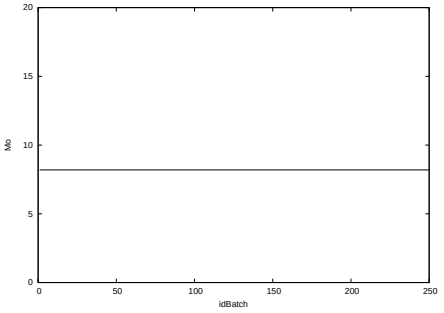
FIG. 7.2: Influence des fonctions de précision (1/2)

Le rôle essentiel des fonctions de précision dans le coût de matérialisation a déjà été évoqué dans la section 4.1.5. Dans cette première serie d'expérimentations nous étudions leur impact sur les performances de notre proposition. Les configurations testées sont présentées dans le tableau 7.1. La première (*expensive*) conserve l'essentiel de l'historique sur le cuboïde le plus précis. A l'inverse, *Slim* décrit une configuration conservant la majorité de l'historique au plus haut niveau d'abstraction. Enfin, la configuration *Balanced* est une configuration un peu plus réaliste car (1) les fonctions ne sont pas identiques et (2) la plupart des intervalles ne sont pas matérialisés sur le même cuboïde. Les autres caractéristiques du jeu de données sont D5L3C5B250T20.
Les figures 7.2 et 7.3 présentent les résultats obtenus. Les temps de traitement sont exposés dans les figures 7.2(b), 7.2(a) et 7.3(a). Les courbes obtenues suivent un comportement similaire. Premièrement, le temps de traitement augmente légèrement pendant les premiers batchs à cause du faible nombre de cellules présentes dans la structure. Nous observons ensuite une alternance entre des temps très rapides et et d'autres plus longs. Les pics correspondent aux batchs où un changement de fenêtre s'effectue. Enfin,

le temps maximal est borné à environ 60 secondes. Nous notons également que l'impact des fonctions de précision est minime sur l'aspect temporel de notre évaluation. Concernant la mémoire vive consommée, celle-ci est présentée dans la figure 7.3(b). Nous ne présentons ici qu'une seule courbe pour les trois tests car celles-ci sont identiques et constantes.



(a): Temps vs *Balanced* (D5L3C5B250T20)

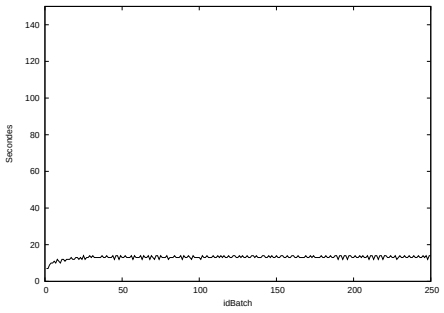


(b): Mémoire vive consommée (D5L3C5B250T20)

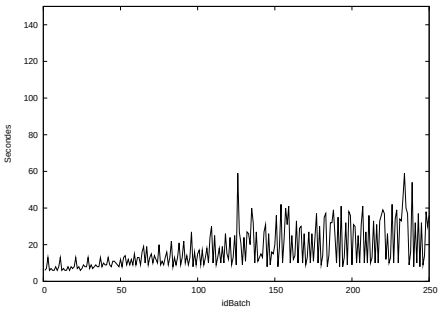
FIG. 7.3: Influence des fonctions de précision (2/2)

Dans un contexte de flot multidimensionnel, il est naturel de tester les limites de notre approche sur un nombre variable de dimensions. Le jeu de tests utilisé est L3C5B250T20. Les fonctions de précisions utilisées sont équilibrées (proches de la configuration *Balanced*). Les figures 7.4 et 7.5 présentent les résultats obtenus sur l'aspect spatial. Ces courbes présentent les mêmes caractéristiques que lors de la première série de tests. De plus, même si les performances se dégradent avec l'augmentation du nombre de dimensions, le temps de traitement reste borné. Nous ne présentons pas ici de courbe sur la mémoire vive consommée car les résultats sont identiques à la première série.

L'influence du nombre de dimensions est plus important que celle des fonctions de précision mais reste malgré tout limitée. De plus, même avec 15 dimensions, le temps de traitement ne dépasse pas une minute.

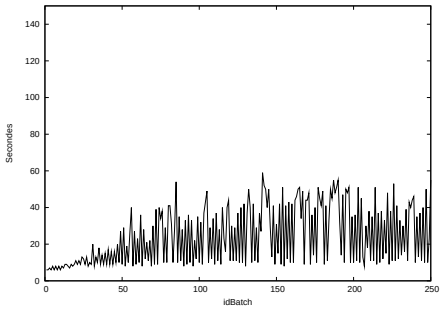


(a): Temps vs 2 dimensions (L3C5B250T20)

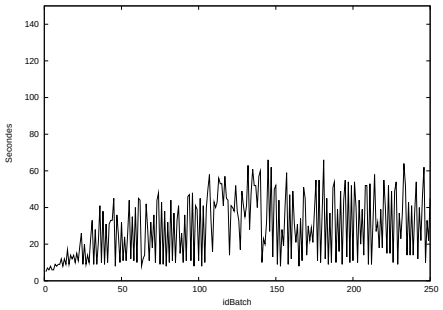


(b): Temps vs 5 dimensions (L3C5B250T20)

FIG. 7.4: Influence du nombre de dimensions (1/2)



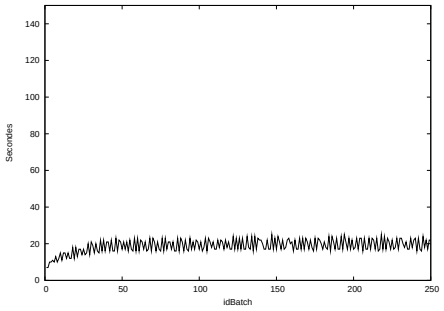
(a): Temps vs 10 dimensions (L3C5B250T20)



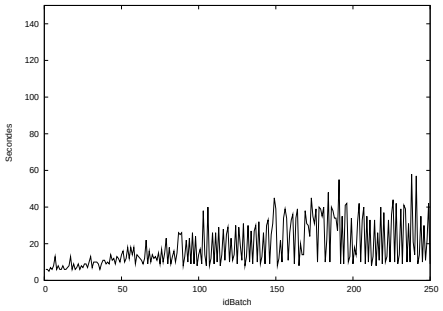
(b): Temps vs 15 dimensions (L3C5B250T20)

FIG. 7.5: Influence du nombre de dimensions (2/2)

Le degré des hiérarchies détermine le nombre de n-uplets pouvant apparaître dans le flot. Puisque celui-ci influe directement sur le nombre de cellules de notre structure, nous faisons varier ce paramètre pour mesurer son impact sur la proposition. Le jeu de données utilisé est D5L3B250T20. Les figures 7.6 et 7.7 montrent des résultats très semblables à la précédente série de tests. La consommation de mémoire étant identique aux autres séries de tests, nous ne les présentons pas. Une fois de plus, les résultats obtenus témoignent des bonnes propriétés du maintien de notre structure.



(a): Temps vs degré 2 (D5L3B250T20)

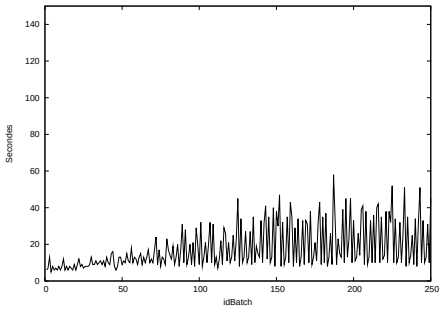


(b): Temps vs degré 5 (D5L3B250T20)

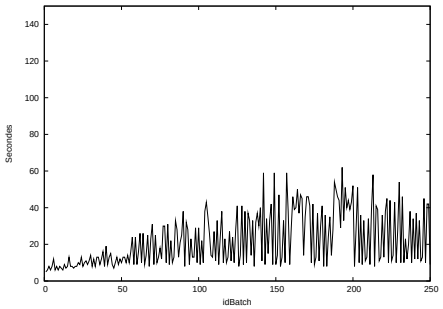
FIG. 7.6: Influence du degré des hiérarchies (1/2)

Nous testons maintenant l'influence de la profondeur des hiérarchies qui déterminent le nombre de cuboïdes matérialisés et influent sur le nombre de généralisation effectuées pour passer d'une fenêtre à une autre. Nous cherchons à déterminer ici quel est l'impact de ces généralisations. Le jeu de test utilisé est D5C5B250T20. Les résultas sont présentés dans les figures 7.8 et 7.9. Le temps de traitement suit le même comportement que pour les autres expérimentations. La consommation de mémoire est encore stable à 10Mo.

Pour conclure ces séries de tests sur des données synthétiques nous dressons quelques conclusions : malgré de grandes variations dans nos jeux de données, (1) la consommation de mémoire vive reste stable et très faible et (2) le temps de traitement maximal par batch se stabilise après une cinquantaine de batchs pour atteindre une valeur très

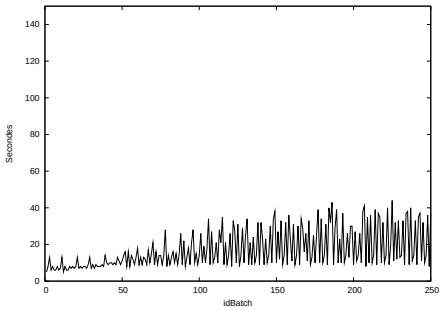


(a): Temps vs degré 10 (D5L3B250T20)

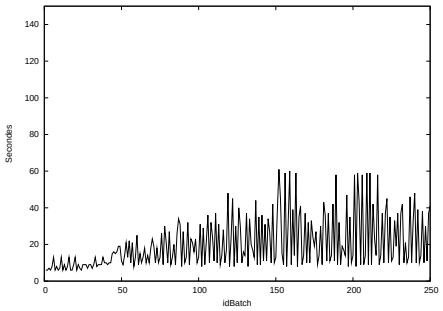


(b): Temps vs degré 20 (D5L3B250T20)

FIG. 7.7: Influence du degré des hiérarchies (2/2)

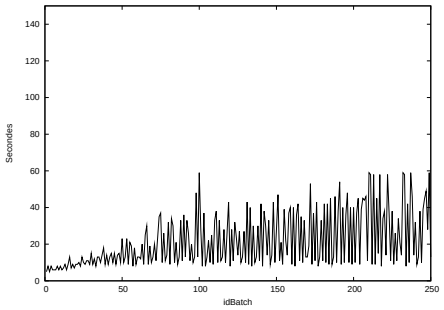


(a): Temps vs profondeur 3 (D5C5B250T20)

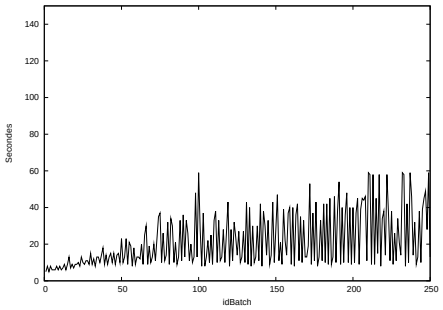


(b): Temps vs profondeur 4 (D5C5B250T20)

FIG. 7.8: Influence de la profondeur des hiérarchies (1/2)



(a): Temps vs profondeur 5 (D5C5B250T20)



(b): Temps vs profondeur 7 (D5C5B250T20)

FIG. 7.9: Influence de la profondeur des hiérarchies (2/2)

acceptable. Ces bons résultats sur les deux aspects critiques dans les flots témoignent des caractéristiques intéressantes de notre proposition.

7.3 Application sur des données réelles

Les deux sections précédentes permettent de conclure que (1) notre approche est moins coûteuse que StreamCube et que (2), même dans des conditions extrêmes notre approche obtient de bons résultats. Nous testons maitnenant notre structure sur des données réelles issues de sondes réseaux. Ce jeu de données a un intérêt certain puisque l'analyse de réseau est un domaine sensible. En effet, que cela soit pour optimiser le trafic, déterminer les habitudes des utilisateurs ou détecter des attaques, il est nécessaire de disposer d'outils d'analyse offrant une bonne réactivité à l'administrateur. Le jeu de données contient 204 batchs de 20 K n-uplets. Un n-uplet est décrit sur 13 dimensions. Ces dimensions sont extraites des entêtes des trames TCP et des datagramme IP. Elles ont été choisies par un expert pour leur pertinence (port source, port destinataire, TTL, etc.). La figure 7.3 (resp. 7.3) illustre la structure d'une trame TCP (resp. d'un datagramme IP). La moitié des dimensions sont hiérarchisées (profondeur 2 ou 3). Par exemple, les ports de destination supérieurs à 32771 sont généralisés en *multimédia* car ces ports sont utilisés par des applications de streaming vidéo et audio.

La figure 7.12 illustre les résultats obtenus sur ce jeu de données. Nous constatons que (1) le temps d'insertion d'un batch dans la structure est borné à 15 secondes et (2) que la mémoire vive consommée reste constante à 8 Mo. Ces bonnes performances permettent d'envisager la mise en place d'un tel système pour la surveillance d'un réseau. Ainsi, un administrateur pourra, par exemple, consulter les plages d'adresses IP atteintes la semaine précédente. Dans le cas d'une attaque par deni de service sur un serveur, l'administrateur doit réagir rapidement.Il n'est donc pas utile de stocker l'historique des adresses destinataires sur six mois. Ainsi, en ne conservant l'historique des adresses IP atteintes que sur l'heure en cours, le gain de matérialisation sera conséquent et il sera

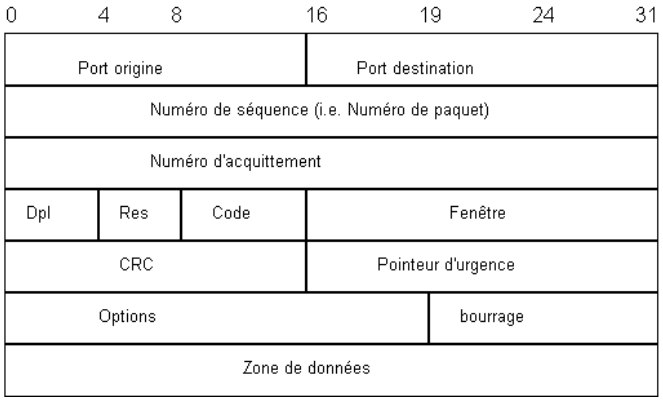


FIG. 7.10: Structure d'une trame TCP

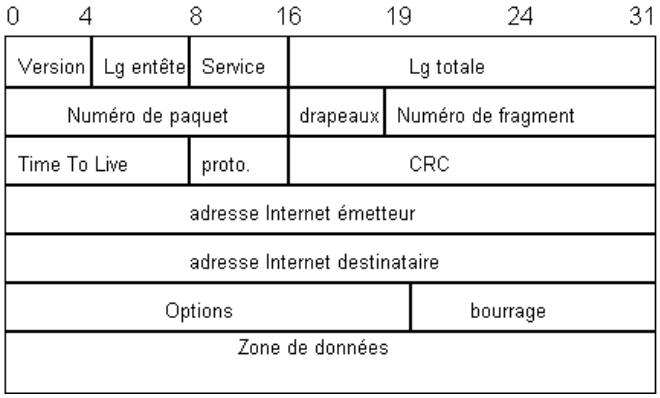
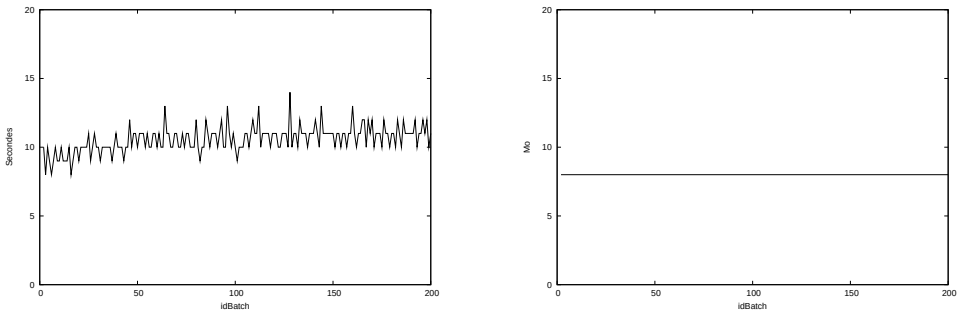


FIG. 7.11: Structure d'un datagramme IP

malgré tout possible d'exhiber des accès anormalement élevés à un serveur (*i.e.* détecter une éventuelle attaque par deni de service). Notre structure permettrait de minimiser l'espace occupé par l'historique tout en garantissant la réactivite de l'administrateur réseau.

7.4 Conclusion

Dans ce chapitre, nous montrons que notre approche possède des propriétés intéressantes. Dans un premier temps, nous vérifions que le nombre d'éléments matérialisés est significativement moindre par rapport à StreamCube. Ensuite, les résultats obtenus sur des jeux de données difficiles témoignent du bon comportement de notre structure du point de vue consommation de la mémoire vive et temps d'insertion d'un batch. Enfin, nous confrontons notre approche à un jeu de données sensible : des logs réseaux. Les réultats obtenus sont très satisfaisants et permettent d'envisager la mise en place d'une telle structure dans un système de supervision de réseau.



(a): Temps (b): Mémoire

FIG. 7.12: Résultats obtenus sur des données réelles

Chapitre 8

Conclusion

Dans ce mémoire, nous nous intéressons à l'adaption des techniques OLAP au contexte des flots. Pour ce faire, nous présentons une structure compacte autorisant une analyse multidimensionnelle et multi-niveaux. Puisque tout l'historique du flot n'est pas interrogé pour tous les niveaux de précision, nous étendons le principe des *tilted-time windows* à toutes les dimensions hiérarchisées. Les *fonctions de précision* définissent alors le moment où un niveau de granularité n'est plus consulté. La combinaison de ces fonctions conduit à une structure extrêmement compacte mais malgré tout capable de répondre à la plupart des requêtes habituelles des utilisateurs.

Afin de borner l'imprécision générée par les fonctions de précision, nous proposons une méthode pour automatiser leur définition. Pour cela, nous exploitons les habitudes des utilisateurs et ramenons le problème de la définition d'une configuration optimale de fonctions à un problème de recherche de plus court chemin dans un graphe. Ces fonctions ont un impact essentiel sur la qualité de notre structure. L'automatisation de leur définition est donc un véritable atout car elle évite à l'utilisateur d'agir sur un paramètre essentiel. La construction de la structure est ainsi entièrement automatisée.

Ensuite, nous définissons une technique pour interroger la structure et gérer l'imprécision. Dans un premier temps, un algorithme évalue la satisfaisabilité d'une requête sans parcourir le cube. Si une réponse précise est impossible, des *requêtes alternatives* sont proposées à l'utilisateur pour le satisfaire au mieux.

Les expérimentations menées sur des jeux de données synthétiques et réels pour évaluer la construction et la mise à jour de notre structure obtiennent des résultats très satisfaisants sur les deux aspects critiques : le temps d'insertion d'une cellule et l'espace occupé par la structure. Nous montrons empiriquement que la mémoire vive reste fixe et faible et que le temps d'insertion d'un batch dans le cube est borné. Ces résulats nous permettent d'affirmer que notre structure possède de bonnes propriétés. Notre approche et les résultats obtenus permettent d'envisager de nombreuses perspectives.

L'interrogation de la structure n'est pas une tâche suffisante et la mise en place de techniques de fouilles plus sophistiquées (recherche de motifs fréquents, classification, détection d'anomalies...) serait un atout important. Les bonnes propriétés de notre approche nous permettent d'envisager la mise en place de telles techniques. Par exemple, l'utilisation de mesures plus complexes ou d'autres fonctions d'agrégation est une solution à envisager.

Ensuite, la génération d'un langage d'interrogation tels que ceux existant dans les DSMS est une étape nécessaire dans la mise en place d'un système de gestion de flot de données multidimensionnelles. Ainsi, une perspective délicate mais prometeuse serait l'adaptation des outils de navigation OLAP dans un contexte dynamique. L'étude de [YP06] peut constituer une base pour élaborer un tel langage.

Une autre perspective à considérer est la prise en compte de plusieurs flots de données. De nombreuses nouvelles contraintes à appréhender apparaissent alors. De plus, l'étude de corrélations entre ces différents flots est un enjeu important.

Enfin, notre travail est destiné à faciliter l'analyse et la fouille de flots par les décideurs. Il est important de fournir un outil pour visualiser le cube et les résultats des techniques de fouilles proposées. Les contraintes engendrées par cette perspective sont nombreuses car (1) la visualisation de cubes de données dans un environnement statique est une thématique active de recherche et (2) l'aspect dynamique des données soulève des nouvelles contraintes. Pour ce faire, nous étudierons les approches existantes dans les entrepôts de données. La visualisation multi-échelle de données complexes est certainement une thématique de recherche intéressante dans notre contexte. En effet, ce domaine est proche de notre contexte puisque l'interface de visualisation à mettre en place doit permettre une navigation multi-niveaux.

Bibliographie

[ABW06] A. Arasu, S. Babu, and J. Widom. The cql continuous query language : semantic foundations and query execution. *The VLDB Journal*, 15(2) :121–142, 2006.

[ACc+03] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora : a new model and architecture for data stream management. *The VLDB Journal*, 12(2) :120–139, 2003.

[Agg07] C. C. Aggarwal. *Data Streams : Models and Algorithms*. Advances in Database Systems. 2007.

[AGP00] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *SIGMOD '00 : Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 487–498, New York, NY, USA, 2000. ACM.

[AHWY03] C.C. Aggarwal, J. Han, J. Wang, and P.S. Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th VLDB Conference*, Berlin, Germany, 2003.

[AHWY04] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. On demand classification of data streams. In *KDD*, pages 503–508, 2004.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB '94 : Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[BR99] K. Beyer and Raghu Ramakrishnan. Bottom-up computation of sparse and iceberg cube. *SIGMOD Rec.*, 28(2) :359–370, 1999.

[BS97] D. Barbará and M. Sullivan. Quasi-cubes : exploiting approximations in multidimensional databases. *SIGMOD Rec.*, 26(3) :12–17, 1997.

[CCD+03] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq : Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[CFP+04] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock : A language for analyzing transactional data streams. *ACM Trans. Program. Lang. Syst.*, 26(2) :301–338, 2004.

[CJSS03] C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope : A stream database for network applications. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *SIGMOD Conference*, pages 647–651. ACM, 2003.

[CL03] J. H. Chang and W. S. Lee. Finding recent frequent itemsets adaptively over online data streams. In *Proceedings of 9th International ACM SIGKDD international conference on Knowledge Discovery and Data Mining*, pages 487–492, New York, NY, USA, 2003.

[CLR] T. Cormen, C. Leiserson, and R. Rivest. *Introduction à l'algorithmique, publisher = Dunod, ISBN= 2 10 003128 7, year=1990.*

[CWZ05] G. Chen, X. Wu, and X. Zhu. Sequential pattern mining in multiple streams. In *ICDM '05 : Proceedings of the Fifth IEEE International Conference on Data Mining*, pages 585–588, Washington, DC, USA, 2005. IEEE Computer Society.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube : A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE '96 : Proceedings of the Twelfth International Conference on Data Engineering*, pages 152–159, Washington, DC, USA, 1996. IEEE Computer Society.

[GHP+02] C. Giannella, J. Han, J. Pei, X. Yan, and P. Yu. Mining frequent patterns in data streams at multiple time granularities, 2002.

[GKTD00] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. *SIGMOD Rec.*, 29(2) :463–474, 2000.

[HCD+05] J. Han, Y. Chen, G. Dong, J. Pei, B. W. Wah, J. Wang, and Y. D. Cai. Stream cube : An architecture for multi-dimensional analysis of data streams. *Distrib. Parallel Databases*, 18(2), 2005.

[HPDW01] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. *SIGMOD Rec.*, 30(2) :1–12, 2001.

[Inm96] W. H. Inmon. *Building the data warehouse (2nd ed.)*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[LLS04] H.-F. Li, S.Y. Lee, and M.-K. Shan. An efficient algorithm for mining frequent itemsets over the entire history of data streams. In *Proceedings of 1st International Workshop on Knowledge Discovery in Data Streams*, Pisa, Italy, 2004.

[Mes06] R. B. Messaoud. *Couplage de l'analyse en ligne et de la fouille de données pour l'exploration, l'agrégation et l'explication des données complexes.* PhD

thesis, Ecole Doctorale Sciences Cognitives de l'Université Lumière Lyon 2, 2006.

[MI06] K. Morfonios and Y. Ioannidis. Cure for cubes : cubing using a rolap engine. In *VLDB '06 : Proceedings of the 32nd international conference on Very large data bases*, pages 379–390. VLDB Endowment, 2006.

[MM06] A. Marascu and F. Masseglia. Mining sequential patterns from data streams : a centroid approach. *J. Intell. Inf. Syst.*, 27(3) :291–307, 2006.

[MWA+03] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Singh Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[PG99] V. Poosala and V. Ganti. Fast approximate answers to aggregate queries on a data cube. In *SSDBM '99 : Proceedings of the 11th International Conference on Scientific on Scientific and Statistical Database Management*, pages 24–33, Washington, DC, USA, 1999. IEEE Computer Society.

[PHMa+01] J. Pei, J. Han, B. Mortazavi-asl, H. Pinto, Q. Chen, and U. Dayal. Prefixspan : Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of 17th International Conference on Data Engineering (ICDE 01)*, pages 215–224, Heidelberg, Germany, 2001.

[PJD99] T. B. Pedersen, C. S. Jensen, and C. E. Dyreson. Supporting imprecision in multidimensional databases using granularities. In *SSDBM '99 : Proceedings of the 11th International Conference on Scientific on Scientific and Statistical Database Management*, page 90, Washington, DC, USA, 1999. IEEE Computer Society.

[PMC05] J. Pei, M.Cho, and D. Wai-Lok Cheung. Cross table cubing : Mining iceberg cubes from data warehouses. In *SDM*, 2005.

[RPT06] C. Raissi, P. Poncelet, and M. Teisseire. Speed : Mining maximal sequential patterns over data streams. In *Proceedings of the 3rd IEEE International Conference on Intelligent Systems (IEEE IS 2006)*, London, UK, 2006.

[SHX04] Z. Shao, J. Han, and D. Xin. Mm-cubing : Computing iceberg cubes by factorizing the lattice space. In *SSDBM*, pages 213–222, 2004.

[VW99] J. Scott Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *SIGMOD '99 : Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 193–204, New York, NY, USA, 1999. ACM.

[VWI98] J. Scott Vitter, M. Wang, and B. Iyer. Data cube approximation and histograms via wavelets. In *CIKM '98 : Proceedings of the seventh international conference on Information and knowledge management*, pages 96–104, New York, NY, USA, 1998. ACM.

[XHLW03] D. Xin, J. Han, X. Li, and B. W. Wah. Star-cubing : computing iceberg cubes by top-down and bottom-up integration. In *vldb'2003 : Proceedings of*

the 29th international conference on Very large data bases, pages 476–487. VLDB Endowment, 2003.

[YP06] X. Yin and T. B. Pedersen. What can hierarchies do for data streams ? In *BIRTE*, pages 4–19, 2006.

[ZDN97] Y. Zhao, Prasad M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. *SIGMOD Rec.*, 26(2) :159–170, 1997.

[ZRL06] T. Zhang, R. Ramakrishnan, and M. Livny. Birch : An efficient data clustering method for very large databases. In *SIGMOD*, Montreal, Canada, June 2006.